TDSQL PostgreSQL 版

产品文档



目录

TDSQL PostgreSQL 版 14
- 产品简介 14
  - 产品概述 14
  - 技术特点 15
  - 产品架构 17
  - 产品优势 19
- 购买指南 21
  - 计费概述 21
  - 购买指引 22
  - 查看账单 23
- 快速入门 24
  - 快速入门（V5.06） 24
    - 客户端psql的使用 24
    - psql常用命令使用 28
  - 快速入门（V5.21） 39
    - SQL基础操作 39
      - PG模式 39
      - Oracle模式 50
- 操作指南 61
  - 实例管理 61
  - 隔离方式说明 70
  - 数据库使用（V5.06） 71
    - 数据库使用说明 71
    - 客户端psql的使用 82
    - psql常用命令使用 86
  - 数据库使用（V5.21） 98
    - 连接TDSQL PG 98
      - 使用psql连接TDSQL PG 98
      - 使用数据库连接池 104
    - SQL基础操作 111
      - PG模式 111
      - Oracle模式 122
- 开发者指南 133
  - 开发指南（V5.06） 133
    - 概述 133
    - 连接到数据库 134
      - 使用TSudio 134
        - TStudio登录 134
        - 通过TStudio连接数据库实例 135
        - TStudio常用操作 137
          - 表管理 137
          - 运行脚本 143
          - 数据导入导出 144
      - 使用shell 145
        - 使用shell连接数据库实例 145
        - 常用操作命令 146
      - 使用ems postgresql manager 147
    - 关于DML与事务 148
      - DML语法 148
        - 关于SELECT 148
        - SELECT使用场景 150
          - with子查询及递归的使用 150
            - with 子查询 150
            - with递归 151
          - 访问函数 153
          - 数据排序 154
          - where条件使用 158
          - 分页查询 161
          - 合并多个查询结果 162
          - 返回两个结果的交集 163
          - 返回两个结果的差集 164
          - any用法 165
          - all用法 167
          - some用法 169
          - 聚集查询 170
          - 多表关联 172
          - 聚合函数并发计算 173
          - not in中包含了null，结果全为真 174
          - 只查某个dn的数据 175
          - 特殊应用 176
          - 查询记录所在dn 177
          - grouping sets/rollup/cube用法 178
            - group by 用法 178
            - 使用grouping sets 179
            - 使用rollup 180
            - 使用cube 181
          - PREPARE预备使用 182
            - 创建一个预备 182
            - 释放一个预备 183
        - 关于INSERT 184
        - INSERT使用场景 185
          - 插入单条记录 185
          - 插入多数记录 187
          - 使用子查询插入数据 188
          - 从另外一个表取数据进行批量插入 189
          - 大批量的生成数据 190
          - 返回插入数据，轻松获取插入记录的serial值 191
          - insert..update更新 192
          - insert all 193

- 关于UPDATE ... 194
- UPDATE使用场景 ... 195
  - 单表更新 ... 195
  - 多表关联更新 ... 196
  - 返回更新的数据 ... 197
  - 多列匹配更新 ... 198
  - shard key不准许更新 ... 199
- 关于DELETE ... 200
- DELETE使用场景 ... 201
  - 带条件删除 ... 201
  - 多表关联删除数据 ... 202
  - 返回删除数据 ... 203
  - 删除所有数据 ... 204
- 部分DML操作示例 ... 205
  - select支持别名不用as修饰 ... 205
  - update支持别名 ... 206
- 关于事务 ... 207
- 事务控制 ... 207
- 事务提交 ... 208
- 事务回滚 ... 209
- 事务一致性 ... 210
- 事务使用场景 ... 211
  - 开始一个事务 ... 211
  - 提交事务 ... 212
  - 回滚事务 ... 214
  - 事务读一致性REPEATABLE READ ... 215
  - 行锁在事务中的运用 ... 216
    - 数据表准备 ... 216
    - 直接update获取 ... 217
    - select...for update获取 ... 218
    - 与mysql获取行级锁的区别 ... 219
- 执行计划 ... 220
- 查看执行计划 ... 220
- 解读执行计划 ... 222
- 执行计划节点类型 ... 226
- 分布式执行计划示例 ... 230
- 执行计划分类 ... 235
  - 顺序扫描 ... 235
  - index scan扫描 ... 236
  - index only scan扫描 ... 237
  - 位图扫描bitmap index scan ... 238
  - join分类 ... 239
    - nest loop ... 239
    - merge join ... 240
    - hash join ... 241
  - 数据重分布 ... 242
  - 并行扫描 ... 243
- 创建和管理数据库对象（DDL） ... 244
- 创建GROUP ... 244
- 创建数据表默认的default group ... 244
- 为default group创建shardmap ... 245
- 更多group的使用方法 ... 246
  - 创建扩展group ... 246
  - 删除group ... 247
  - 查询集群中group的数量 ... 248
- 创建和管理模式 ... 249
- 模式管理 ... 249
  - 创建模式 ... 249
  - 修改模式属性 ... 250
  - 删除模式 ... 251
  - 配置用户访问模式权限 ... 252
  - 配置访问模式的顺序 ... 253
- 创建和管理表 ... 255
- 创建表 ... 255
- 数据类型 ... 256
  - 数字类型 ... 256
  - 字符类型 ... 257
  - 二进制数据类型 ... 258
  - 日期类型 ... 259
  - 布尔类型 ... 260
  - 异构数据库类型对照表 ... 261
    - 与ORACLE数据类型对比 ... 261
    - 与MYSQL数据类型对照表 ... 262
    - 与SQLSERVER数据类型对照表 ... 264
    - 与INFORMIX数据类型对照表 ... 266
  - 自定义复合类型 ... 267
    - 增加一个复合类型 ... 267
    - 数据表使用复合类型 ... 268
  - json/jsonb类型 ... 269
    - json应用 ... 269
      - 创建json类型字段表 ... 269
      - 插入数据 ... 270
      - 通过键获得 JSON 对象域 ... 271
      - 以文本形式获取对象值 ... 272
    - jsonb应用 ... 273
      - 创建jsonb类型字段表 ... 273
      - 插入数据 ... 274
      - 更新数据 ... 275
      - jsonb_set()函数更新数据 ... 276
    - jsonb函数应用 ... 277
      - jsonb_each()将json对象转变键和值 ... 277

- jsonb_each_text()将json对象转变文本类型的键和值 278
- row_to_json()将一行记录变成一个json对象 279
- json_object_keys()返回一个对象中所有的键 280
- jsonb索引使用 281
- 创建立jsonb索引 281
- 测试查询的性能 282
- 部分数据类型示例 283
- varchar2 283
- number 284
- blob 285
- clob 286
- 外键创建与删除 287
- 外键创建 287
- 删除外键 289
- 创建和管理索引 290
- 创建和删除索引 290
- 索引类型 291
- 全局索引 291
- 普通索引 296
- 唯一索引 297
- 表达式索引 298
- 条件索引 299
- gist索引 301
- gin索引 302
- 多字段索引 306
- 删除索引 309
- 修改表 310
- 修改表结构场景 310
- 修改表名 310
- 给表或字段添加注释 311
- 给表增加字段 312
- 修改字段类型 313
- 修改字段默认值 314
- 删除字段 315
- 添加主键 316
- 删除主键 317
- 重建主键 318
- 添加外键 319
- 删除外键 320
- 修改表所属模式 321
- 修改表所属用户 322
- 修改字段名 323
- 修改表的填充率 324
- 添加触发器 325
- 删除触发器 329
- truncate普通表 330
- 删除表 331
- 表管理场景 332
- 不用指定shard key 建表方式 332
- 指定shard key 建表方式 333
- 指定group 建表方式 334
- 复制表 335
- 列存表管理 336
- 创建列存表 336
- 指定压缩类型 337
- tdx外表管理 341
- tdx服务 341
- 创建exttable_fdw 342
- 创建外表 343
- 删除外表 345
- 将外表数据导入到物理表中 346
- 外表管理 347
- postgres_fdw 347
- 创建postgres_fdw extension 347
- 创建一个fdw server 348
- 配置user mapping 349
- 创建外表 350
- 访问外表 351
- oracle_fdw 352
- 创建oracle_fdw extension 352
- 创建一个fdw server 353
- 配置user mapping 354
- 创建外表 355
- 访问外表 356
- dblink外表 357
- oracle_dblink 357
- 创建oracle_fdw 357
- 创建dblink 358
- 查询dblink系统表 359
- 访问远程表 360
- 删除dblink 361
- tbase_dblink 362
- 创建postgres_fdw 362
- 创建dblink 363
- 查询dblink系统表 364
- 访问远程表 365
- 删除dblink 366
- 指定模式创建表 367
- 使用将查询结果创建数据表 368
- 删除数据表 369
- copy的使用 370

- 实验表结构及数据 370
- copy to用法详解--复制数据到文件中 371
  - 导出所有列 371
  - 导出部分列 372
  - 导出查询结果 373
  - 指定生成文件格式 374
  - 使用delimiter指定列与列之间的分隔符 375
  - NULL值的处理 376
  - 生成列标题名 377
  - 导出oids系统列 378
  - 使用quote自定义引用字符 379
  - 使用escape自定义逃逸符 380
  - 强制给某个列添加引用字符 381
  - 使用encoding指定导出文件内容编码 382
- copy from用法详解--复制文件内容到数据表中 383
  - 导入所有列 383
  - 导入部分指定列 384
  - 指定导入文件格式 385
  - 使用delimiter指定列与列之间的分隔符 386
  - NULL值处理 387
  - 自定义quote字符 388
  - 自定义escape字符 389
  - csv header忽略首行 390
  - 导入oid列值 391
  - 使用FORCE_NOT_NULL把某列中空值变成长度为0的字符串，而不是NULL值 392
  - encoding指定导入文件的编码 393
  - position指定字段长度导入 394
  - position指定字段长度+函数处理导入 395
- copy支持多字节分隔符 396

- 创建和管理分区表 399
  - 范围分区 399
    - 创建时间范围分区表 399
    - 范围分区类型 400
    - 范围分区表使用 403
  - 列表分区 404
    - 列表分区使用 404
  - 散列分区 406
    - 散列分区使用 406
  - 多级分区 407
    - 多级分区表使用 407
  - 分区拆分与合并 409
    - 分区拆分 409
    - 分区合并 411
  - 添加分区子表 413
  - 访问子分区 414
  - truncate分区 415
  - 自研分区 416
  - 冷热分区表 417
  - truncate分区表 419
  - 部分分区表示例 422
    - partition访问子分区 422
    - 支持创建分区表时将默认分区中属于新分区的数据转移到新的分区表中 423
    - truncate partition子分区 424
    - 分区拆分 425
    - 分区合并 426
    - 支持不同父表下相同的子表名称 427
- 创建和管理视图 428
  - 创建视图 428
    - 创建视图示例 428
  - 修改视图 430
  - 删除视图 431
    - 删除视图示例 431
  - 物化视图 432
    - 创建物化视图 432
    - 访问物化视图 433
    - 增量数据刷新 434
  - 视图触发器创建与删除 435
    - 创建视图触发器 435
    - 删除视图触发器 436
- 创建和管理序列 437
  - 自增列与序列的用法 437
    - 序列的创建和访问 437
    - 序列在DML中使用 439
    - 序列做为字段的默认值使用 440
    - 序列做为字段类型使用 441
    - 删除序列 442
    - 查询gtm中的序列对象列表 443
- 创建和管理同义词 444
  - 部分同义词示例 444
    - 创建同义词 444
    - 同义词访问 445
    - 同义词系统表 446
- 关于PL/SQL 447
  - PL/SQL介绍 447
- PL/SQL控制语句 448
  - 条件选择语句 448
  - 循环语句 451
  - 其他控制语句 458
- PL/SQL集合和记录 461
  - 集合概述 461
  - 关联数组 462

- 关联数组定义和初始化 462
- 关联数组的使用 471
- 可变数组 472
- 嵌套表 474
- 记录类型 477
- PL/SQL静态SQL 478
- 静态SQL概述 478
- 游标 479
- 显示游标 479
- 隐式游标 480
- 使用for循环遍历游标 481
- 游标的使用 482
- 定义一个游标 482
- 提取下一行数据 483
- 提取前一行数据 484
- 提取最后一行 485
- 提取第一行 486
- 提取该查询的第x行 487
- 提取当前位置后的第x行 488
- 向前提取x行数据 489
- 向前提取剩下的所有数据 490
- 向后提取x行数据 491
- 向后提取剩下的所有数据 492
- 事务处理与控制 493
- 自治事务 494
- PL/SQL动态SQL 495
- PL/SQL子程序 497
- 过程 497
- 函数 499
- PL/SQL触发器 501
- PL/SQL错误处理 502
- 参数详细介绍 504
- 参数模式 504
- IN模式 504
- OUT模式 505
- INOUT模式 506
- VARIADIC模式 507
- 参数引用 509
- 无命名参数 509
- 给标识符指定别名 510
- 命名参数 511
- 参数数据类型 512
- 基本类型 512
- 复合类型 513
- 行类型 514
- 域类型 516
- 游标类型 517
- 多态类型 518
- 参数默认值 520
- 返回值详细介绍 521
- 返回值介绍 521
- 返回值类型介绍 522
- 没有返回值 522
- 返回简单类型 523
- 返回一个复合类型 525
- 返回行类型 526
- 返回TABLE类型 527
- 返回RECORD类型 528
- 返回一个游标 529
- 返回记录集 531
- 返回多态类型 536
- 变量使用 538
- 变量使用介绍 538
- 变量使用实例 539
- 变量声明语法 539
- 定义一个普通变量 540
- 定义CONSTANT变量 541
- 定义NOT NULL变量 542
- 定义COLLATE变量 543
- 变量赋值 545
- PL/SQL函数实战 546
- 批量设置表owner的函数 546
- 批量设置表的加密规则函数 548
- oracle to_date函数的实现 549
- 关于包(PACKAGE) 550
- 语法说明 550
- 包结构 550
- 创建包 551
- 删除包 552
- 包的使用 553
- 函数在包中的用法 553
- 存储过程在包中的用法 554
- 函数与存储过程一起使用 555
- 变量使用方法 556
- 其他常见使用 558
- 关于触发器 559
- 创建触发器 559
- 表触发器创建与删除 560
- 创建触发器 560
- 删除触发器 561
- 删除触发器 562

- 触发器函数 563
  - INSERT事件触发器函数 563
  - UPDATE事件触发器函数 564
  - DELETE事件触发器函数 565
  - 删除触发器 566
  - 触发器使用限制 567
- ORACLE兼容性特性与SQL参考 568
- 启用Oracle兼容性 568
- 系统package 569
  - ALL_ARGUMENTS 569
  - DBMS_JOB 572
  - DBMS_ASSERT 573
  - DBMS_LOB 574
  - DBMS_OUTPUT 576
  - DBMS_PIPE 593
  - DBMS_RANDOM 598
    - value()函数示例 598
  - DBMS_SQL 599
  - DBMS_STATS 605
  - DBMS_UTILITY 607
  - UTL_FILE 609
  - UTL_RAW 612
- 系统视图 615
  - all_arguments 615
  - all_users 617
  - all_tab_columns 618
  - all_col_comments 620
  - all_constraints 621
  - all_synonyms 622
  - all_indexes 623
  - all_ind_columns 626
  - all_cons_columns 627
  - dba_ind_columns 628
  - dba_constraints 629
  - dba_indexes 630
  - dba_synonyms 633
  - dba_tab_columns 634
  - dba_users 636
  - user_indexes 637
  - user_synonyms 640
  - user_cons_columns 641
  - user_tab_columns 642
  - user_ind_columns 644
  - user_constraints 645
  - user_col_comments 646
  - user_users 647
- 函数 648
  - 单行函数 648
    - 数值函数 648
    - 返回字符值的字符函数 650
      - 部分返回字符值的字符函数示例 650
        - regexp_substr 650
        - nlssort 651
        - REGEXP_INSTR 652
        - regexp_replace 653
        - nchr 654
        - nls_upper 655
    - 返回数值的字符函数 656
      - 部分返回数值的字符函数示例 656
        - ASCIISTR 656
        - LENGTHB 657
        - length 658
        - instr 659
        - regexp_count 660
    - 日期时间函数 661
      - 部分日期时间函数示例 661
        - NUMTODSINTERVAL 661
        - DBTIMEZONE 662
        - MONTHS_BETWEEN 663
        - LAST_DAY 664
        - ADD_MONTHS 665
    - 比较函数 666
    - 转换函数 667
      - 部分转换函数示例 667
        - to_number 667
        - to_clob 668
        - ROWIDTOCHAR 669
        - CHARTOROWID 670
    - XML函数 671
      - 部分XML函数示例 671
        - extractvalue 671
        - extract 672
    - 编码和解码函数 673
      - 部分编码和解码函数示例 673
        - vsize 673
    - NULL相关函数 674
      - 部分NULL相关函数示例 674
        - lnnvl 674
        - nvl2 675
    - 环境和标识符函数 676
      - 说明以及示例 676

- 聚合函数 679
- 分析函数 680
- 二进制操作函数 682
  - empty_clob 682
- 统计函数 683
  - listagg 683
- 其他函数的使用 684
  - sys_guid() 684
  - 数据表准备 685
  - row_number() --返回行号，不分组 687
  - row_number() --返回行号，按amount排序 688
  - row_number() --返回行号，按begincity分组，pubtime排序 689
  - rank()--返回行号,对比值重复时行号重复并间断，即返回1,2,2,4... 690
  - dense_rank() --返回行号,对比值重复时行号重复但不间断，即返回1,2,2,3... 691
  - percent_rank()从当前开始，计算在分组中的比例 (行号-1)*(1/(总记录数-1)) 692
  - cume_dist() --返回行数除以记录数值 693
  - ntile(分组数量)--让所有记录尽可以的均匀分布 694
  - lag(value any [, offset integer [, default any]])--返回偏移量值 695
  - lead(value any [,offset integer [, default any]])--返回偏移量值 697
  - first_value(value any)返回第一值 698
  - last_value(value any)返回最后值 699
  - nth_value(value any, nth integer)：返回窗口框架中的指定值 701
  - 统计各个城市的总运费及平均每单的运费 702
  - 窗口函数别名使用 703
  - 获取每个城市运费前两名订单 704
  - 连续百分率：返回一个对应于排序中指定分数的值 705
- 伪列 706
  - ROWNUM 706
  - ROWID 708
- hint 710
  - 加载插件 710
  - 部分hint用法示例 711
    - full全表扫描 711
    - index全表扫描 712
    - no_index全表扫描 713
    - 别名使用 714
    - 强制走nest loop 715
    - 强制走merge join 716
    - 强制走hash join 717
    - 并行执行 718
- 运算符 719
  - 算术运算符 719
  - 逻辑运算符 721
  - 比较运算符 722
  - 连接运算符 723
  - 集合运算符 725
  - 其他运算符 727
    - （+） 727
    - 空值与NULL 729
    - 字符串拼接 731
- GOTO 732
- 其他常用特性 733
  - table函数 733
  - sqlcode 734
  - select for update wait 739
  - 自治事务 741
  - dual 745
  - sysdate 746
  - systimestamp 747
  - merge into 748
  - connect by 749
  - pivot 754
  - unpivot 756
  - offset x rows fetch next x rows only 757
  - 只读事务获取序列值 758
  - ''等同于null 759
  - 创建view使用with read only 760
  - insert all 761
  - q'转义字符 762
  - 匿名块 763
- 关于插件 764
  - 插件查看，添加和删除 764
    - 查看数据库加载了那些插件 764
    - 添加插件 765
    - 删除插件 766
  - 插件uuid-ossp使用 767
    - uuid-ossp功能介绍及添加方法 767
    - uuid常量函数 768
      - uuid_nil() 768
      - uuid_ns_dns() 769
      - uuid_ns_url() 770
      - uuid_ns_oid() 771
      - uuid_ns_x500() 772
    - uuid生成函数 773
      - uuid_generate_v1() 773
      - uuid_generate_v1mc() 774
      - uuid_generate_v3(namespace uuid, name text) 775
      - uuid_generate_v4() 776
      - uuid_generate_v5(namespace uuid, name text) 777
    - 在数据表中使用uuid默认值 778
    - uuid各函数性能对比 779

- uuid与serial 在TDSQL PG中性能对比 781
- 占用空间对比 783
- 使用uuid做为分布列的方案 785
- 使用建议 786
- 插件pg_stat_statements使用 787
- pg_stat_statements功能介绍及添加方法 787
- 获取执行次数最多的语句 788
- 获取执行总时间最长的语句 789
- 获取每句平均执行时间最长的语句 790
- 获取buffer读最多的语句 791
- 插件pg_trgm使用 792
- pg_trgm功能介绍及添加方法 792
- 测试环境准备 793
- gist索引测试 794
- 创建索引消耗时间 794
- 索引占用空间 795
- 模糊查询测试 796
- 数据导入时间测试 798
- pgbench并发查询测试 799
- pgbench并发写入测试 801
- gin索引测试 802
- 创建索引消耗时间 802
- 索引占用空间 803
- 模糊查询测试 804
- 数据导入时间测试 806
- pgbench并发查询测试 807
- pgbench并发写入测试 808
- 无索引字段测试 809
- pgbench并发查询测试 809
- pgbench并发写入测试 811
- 数据对比总结及例外 812
- 数据对比 812
- 插件postgis使用 813
- postgis添加方法 813
- 关于存储过程 814
- 存储过程语法介绍 814
- 建立存储语法 814
- [OR REPLACE] 更新存储介绍 816
- [模式名.]存储过程名介绍 818
- 存储过程与函数不能同名 819
- 删除存储过程 820
- 删除不带参数的存储过程 820
- 删除带参数的存储过程 821
- 存储过程修改名称 822
- 修改不带参数的存储过程名称 822
- 修改带参数的存储过程名称 823
- 修改存储过程所属schema 824
- 修改不带参数的存储过程schema 824
- 修改带参数的存储过程schema 825
- 修改存储过程所属用户 826
- 修改不带参数的存储过程所属用户 826
- 修改带参数的存储过程所属用户 827
- 存储过程执行 828
- 不带参数 828
- 带参数 829
- 中途return返回 830
- 参数详细介绍 831
- 参数模式 831
- IN模式 831
- INOUT模式 832
- 参数引用 833
- 无命名参数 833
- 给标识符指定别名 834
- 命名参数 835
- 参数数据类型 836
- 基本类型 836
- 复合类型 837
- 行类型 838
- 游标类型 839
- 多态类型 840
- 参数默认值 842
- 变量使用 843
- 变量使用介绍 843
- %TYPE 843
- %ROWTYPE 844
- 记录类型 848
- 变量使用示例 849
- 变量声明语法 849
- 定义一个普通变量 850
- 定义CONSTANT变量 851
- 定义NOT NULL变量 852
- 定义COLLATE变量 853
- 变量赋值 855
- BULK COLLECT 856
- 控制结构 858
- 判断语句 858
- IF...THEN...END IF 858
- IF...THEN...ELSE...END IF 859
- IF...THEN...ELSIF...THEN...ELSE...END IF 860
- CASE语句 861
- 循环语句 862

- LOOP循环 862
- WHILE循环 863
- FOR循环 864
- FOR循环查询结果 866
- FOREACH循环一个数组 867
- 其它控制语句 869
- 动态执行 869
- 执行一个没有结果的命令 870
- 获取执行结果 871
- 获取影响行数 872
- GOTO 873
- 俘获错误 874
- 错误俘获处理 874
- 获取错误相关信息 875
- 自治事务 876
- 在存储过程中commit和rollback 876
- 游标使用限制 878
- 自治事务与exception的限制 879
- 消息及异常处理 880
- RAISE NOTICE 880
- RAISE EXCEPTION 881
- RAISE EXCEPTION 自定义ERRCODE 882
- oracle存储过程兼容性 883
- 不带参数的存储过程不需要括号 883
- 支持使用is语法 884
- 支持不使用$$语法 885
- 支持使用"end存储过程名称"结束 886
- 使用"/"结束函数定义 887
- 定义变量无需要declare 888
- 调用存储过程不需要call 889
- 支持存储过程out返回值 890
- oracle语法改写 891
- forall改写 891
- table函数 892
- 应用程序样例 893
- java 893
- 创建数据表 893
- 插入数据 895
- 扩展协议插入数据 896
- 扩展协议插入返回数据 897
- 查询数据 898
- 扩展协议查询数据 899
- copy from 入库 901
- copy from 数据流入库 902
- copy to出库 903
- 兼容oracle字段大写 904
- 配置多个cn负载 905
- 合并多条insert 906
- jdbc驱动包 907
- C程序 908
- 连接服务 908
- 建立数据表 910
- 插入数据 912
- 查询数据 914
- copy入库 916
- shell程序 918
- python程序 919
- 安装psycopg2模块 919
- 连接服务 920
- 创建数据表 921
- 新增数据 922
- 查询数据 923
- copy from方法 924
- php程序 925
- 连接服务 925
- 创建数据表 926
- 插入数据 927
- 查询记录 928
- copy from 方法 930
- copy to 方法 931
- 入库去重方法 932
- golang程序 936
- 连接服务 936
- 建立数据表 938
- 插入数据 940
- 查询数据 943
- copy from 方法 946
- go相关资源包 948
- 关于运维 949
- 数据库管理 949
- 创建数据库 949
- 修改数据库配置 953
- 删除数据库 954
- 会话及锁管理 955
- 查看当前会话的PID 955
- 查看当前节点有那些会话 956
- 杀掉连接 957
- 查看会话持锁情况 958
- 配置会话锁超时 959
- 问题定位及性能优化 960
- 访问日志管理 960

- 配置日志 960
- 日志格式说明 962
- 对日志进行分析 963
- 创建日志表 963
- 导入日志数据 964
- 统计日志数据 965
- 配置只收集慢的sql语句 967
- 如何查询数据是否倾斜 968
- 如何优化有问题的Sql语句 969
- 查看是否为分布键查询 969
- 查看是否使用上索引 970
- 查看是否为分布key join 971
- 查看join发生的节点 972
- 查看并行的worker数 973
- 检查各个节点的执行计划是否一致 974
- 优化实例 975
- count(distinct xx)优化 975
- 增大work_mem减少io访问 977
- not in改写为anti join 979
- 分布key jon+limit优化 981
- 非分布key join使用hash join性能一般最好 983
- exists的优化 987
- 重新聚簇表 989
- jdbc访问数据库优化 990
- 分区表now()剪枝问题 993
- 使用TDSQL PG自研分区表 995
- 插入数据性能测试对比 995
- 自研分区表 995
- PG社区分区表 996
- 查询数据性能测试对比 997
- 自研分区表 997
- PG社区分区表 999
- 数据库开发规范 1001
- 分布键设计规范 1001
- 分布键约束规则 1001
- 分布键选择规范 1002
- 分布键对其它约束影响 1003
- 命名规范 1004
- COLUMN设计 1005
- Constraints 设计 1006
- Index 设计 1007
- 关于NULL 1008
- 开发相关规范 1011
- 数据库开发基础 1017
- TDSQL-PG的数据类型 1017
- 数字类型 1017
- 字符类型 1018
- 二进制数据类型 1019
- 日期类型 1020
- 布尔类型 1021
- 更多的数据类型介绍 1022
- 消息及异常输出 1023
- RAISE NOTICE 1023
- RAISE EXCEPTION 1024
- RAISE EXCEPTION 自定义ERRCODE 1025
- 控制结构 1026
- 判断语句 1026
- IF...THEN...END IF 1026
- IF...THEN...ELSE...END IF 1027
- IF...THEN...ELSIF...THEN...ELSE...END IF 1028
- CASE语句 1029
- 循环语句 1030
- LOOP循环 1030
- WHILE循环 1032
- FOR循环 1033
- FOR循环查询结果 1035
- FOREACH循环一个数组 1036
- 其它控制语句 1038
- 动态执行 1038
- 执行一个没有结果的命令 1039
- 获取执行结果 1040
- 获取影响行数 1041
- 俘获错误 1042
- 错误俘获处理 1042
- 获取错误相关信息 1043
- 应用程序语法介绍 1044
- 建立函数语法 1044
- [OR REPLACE] 更新函数介绍 1045
- [模式名.]函数名介绍 1047
- 连接到数据库（replace） 1048
- shell交互客户端psql 1048
- 连接到一个数据库 1048
- 建立一个新连接 1050
- 显示和设置该连接当前运行参数 1051
- 退出连接 1057
- psql执行一个sql命令 1058
- psql执行一个sql文件中所有命令 1059
- 调用编辑器编写sql脚本 1060
- 调用外部命令 1061
- 将执行的结果保存到文件 1062
- 改变当前的工作目录 1063

- 插件管理 1064
- 数据库相关操作 1065
- 模式相关操作 1066
- 用户相关操作 1067
- 表相关操作 1068
- 视图相关操作 1071
- 物化视图相关操作 1073
- 序列相关操作 1075
- 索引相关操作 1077
- 函数相关操作 1079
- 自定义数据类型相关操作 1083
- 存储过程语句相关操作 1085
- 列出表、视图和序列和它们相关的访问权限 1086
- 列出库或用户定义的配置 1087
- copy命令的使用 1088
- copy FROM stdin使用方法 1089
- 打印当前查询缓冲区到标准输出 1090
- 自定义显示格式 1091
- 显示psql内部操作 1093
- 重复执行上一条语句 1094
- sql命令帮助查看 1095
- ems postgresql manager 1096
- ems postgresql manage介绍 1096
- 连接到tbase服务 1097
- 查询表结构 1099
- 执行SQL语句 1100
- 连接tbase-v5出错处理 1101
- 开发指南（V5.21） 1103
- 概述 1103
- 连接TDSQL PG 1104
- 使用psql连接TDSQL PG 1104
- 使用数据库连接池 1110
- 使用Java框架连接TDSQL PG 1117
- 使用JDBC连接TDSQL PG 1117
- 使用Hibernate框架连接TDSQL PG 1125
- 使用MyBatis框架连接TDSQL PG 1136
- 使用Spring Boot框架连接TDSQL PG 1147
- 使用C/C++连接TDSQL PG 1157
- 使用ODBC连接TDSQL PG 1157
- 使用OCI连接TDSQL PG 1161
- 使用Pro*C连接TDSQL PG 1179
- 创建数据库对象 1189
- 创建数据库 1189
- 创建模式 1195
- 创建表 1198
- 创建表概述 1198
- 创建分布式表 1200
- 创建复制表 1201
- 创建列存表 1202
- 创建分区表 1205
- 创建索引 1228
- 创建DBLink 1235
- 数据写入 1241
- 插入数据 1241
- 更新数据 1247
- 删除数据 1251
- 替换数据 1254
- 数据读取 1261
- 单表查询 1261
- 关联查询 1265
- 子查询 1269
- CTE 1274
- 集合操作 1276
- 事务使用 1279
- 事务概述 1279
- 事务操作 1282
- SAVEPOINT 1285
- Oracle兼容特性 1287
- Oracle兼容特性概述 1287
- SQL基本元素 1289
- SQL数据类型 1289
- 数据类型比较规则 1293
- 字面量 1294
- 格式模型 1295
- 空值 1296
- 注释 1297
- 数据库对象 1298
- 数据库对象名称和限定符 1300
- SQL 语句中模式对象及其组成部分的语法 1301
- 伪列 1302
- 操作符 1303
- SQL表达式 1306
- 条件 1307
- 常见SQL语法 1308
- PL/SQL 1313
- 系统函数 1315
- 系统视图 1331
- 系统包 1334
- Oracle兼容性配置参数 1336
- API文档 1341
- TBase部署中心（tbase） 1341

• 版本（2019-01-07） ····· 1341
• API 概览 ····· 1341
• 调用方式 ····· 1342
• 接口签名v1 ····· 1342
• 接口签名v3 ····· 1349
• 请求结构 ····· 1358
• 返回结果 ····· 1359
• 公共参数 ····· 1362
• TBase部署中心 ····· 1364
• 获取回档时间 ····· 1364
• 查询售卖地域信息 ····· 1366
• 数据结构 ····· 1368
• 错误码 ····· 1391

产品简介
产品概述

TDSQL PostgreSQL 版产品简介

TDSQL PostgreSQL 版分布式数据库系统，具备完整的HTAP处理能力。内置的分布式处理逻辑对业务屏蔽底层数据分布细节，提供完整的分布式事务处理能力，同时支持SQL2003，业务可以像使用单机数据库一样来设计自己的业务逻辑，对于已有的业务系统，可以轻松的完成分布式改造，减少对原有系统的侵入性。

TDSQL PostgreSQL 版不但具有分布式数据库具备的在线扩容，在线热升级等功能外，还提供完整的高可用架构保证用户数据库安全，且内置完整的MLS（multiple level security）安全体系，防止用户数据发生泄漏，是目前安全能力最强的数据库产品。

通过TDSQL PostgreSQL 版的OSS运维操作系统，可以对TDSQL PostgreSQL 版进行完整的运维监控操作，大大简化用户的运维操作成本。

目前TDSQL PostgreSQL 版已经服务政务，银行，保险，互联网金融，能源，烟草，医疗等众多行业。

技术特点

透明数据分布

TDSQL PostgreSQL 版支持shard表，replication表，业务只需建表时制定分表逻辑，TDSQL PostgreSQL 版自动完成底层的数据分布和分库分表逻辑，底层数据存储细节对业务透明。

完整分布式事务支持

提供完整的分布式事务保证，支持分布式事务ACID，提供分布式一致性读能力。业务可以像使用单机数据库一样来进行事务处理，而不用感知分布式事务处理的底层细节。

完整的SQL兼容性

TDSQL PostgreSQL 版完整兼容SQL2003语法，兼容PostgreSQL语法，同时兼容Oracle语法；在这个基础上，TDSQL PostgreSQL 版内置的高效执行器可以高效的处理各种复杂查询，包括分布式join，聚合，窗口函数，子查询等，业务逻辑无需感知底层数据分布逻辑。用户可以像使用普通数据库一样使用TDSQL PostgreSQL 版数据库。

行列混合HTAP（Hybrid Transactional/Analytical Processing）能力

单集群千万QPS分布式事务处理能力，高效处理OLTP类型的业务；同时支持行存储和列存储，行表和列表可以根据业务需要进行互操作，提升业务处理效率；具备全并行OLAP处理能力，硬件加速进行向量化计算，充分提升OLAP执行效率。

业界最安全的数据库

基于三权分立，构建了业界最高级别的数据安全体系。支持行列矩阵式权限控制，做到对数据的完整权限用户控制;提供特有的透明加密，透明脱敏处理功能，防止文件泄漏引起的数据泄漏；同时提供完整的安全审计能力，具备强大的时候追溯能力。

多租户能力支持

TDSQL PostgreSQL 版提供多种纬度的多租户解决方案，协助用户进行数据共享和资源隔离：物理资源隔离的多租户，用户数据层多租户。

在线扩容缩容能力

TDSQL PostgreSQL 版提供业务无感知的在线扩容缩容能力，后台自动完成扩容缩容任务，业务无须感知扩容缩容逻辑。

PostgreSQL扩展能力

支持PostgreSQL的常用插件，业务可以像使用单机的PostgreSQL一样使用TDSQL PostgreSQL 版。比如fdw，PostGIS，pipelinedb等。同时也提供完整兼容PostgreSQL的数据类型和函数。

多平面读写分离能力

TDSQL PostgreSQL 版提供数据多平面读写分离能力，可以使用集群的备机提供读取服务，在保障分布式一致性的前提下提升系统的资源利用率和系统吞吐量。

冷热数据分离能力

TDSQL PostgreSQL 版针对长尾业务数据量大的问题提出了完整的冷热数据分离处理方案，业务无须感知，TDSQL PostgreSQL 版自动完成业务数据的冷热分离，帮助用户自动完成数据的冷热切分。

内核版本说明

TDSQL PostgreSQL 版

TDSQL PostgreSQL 版主要版本为融合版。这是 TDSQL PG 团队精心打造的 HTAP 国产精品。融合版既支持创建分布式架构的数据库实例，也支持集中式架构的数据库实例。该版本支持 Oracle 和 PostgreSQL 的双模式，采用统一引擎的行列混合存储，能实现同引擎自适应的TP/AP分流。相比其他版本，融合版的性能更为出色。无论是兼容 PostgreSQL 的场景还是兼容 Oracle 的场景，融合版都能提供优秀的服务。

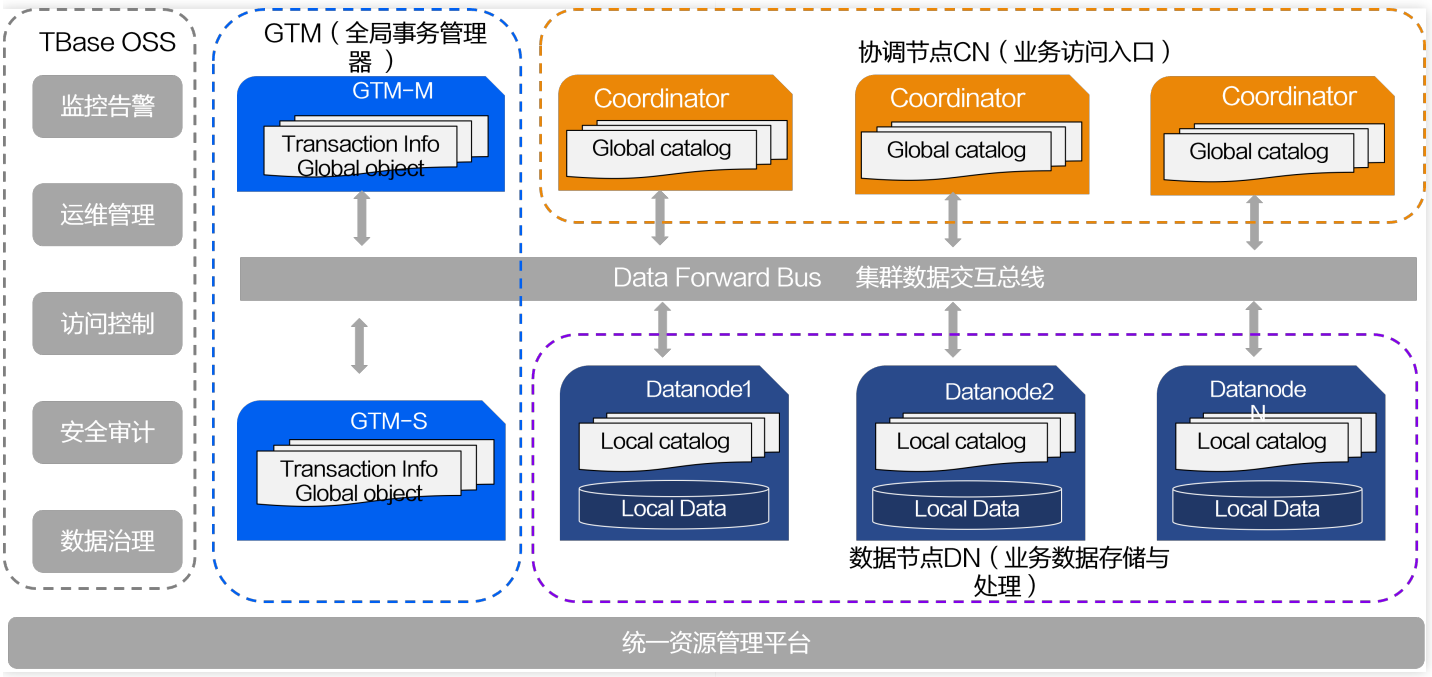
PostgreSQL 版

PostgreSQL 版提供两个版本供您选择，分别是兼容社区的10.5.5版本和 TDSQL PostgreSQL 单机版。其中，TDSQL PostgreSQL 单机版完全兼容社区 PostgreSQL 15.6，确保了无缝的使用体验，让您在使用过程中感到轻松自如。

产品架构

TDSQL PostgreSQL 版架构简介

TDSQL PostgreSQL 版总体架构如图



- Coordinator

  Coordinator（简称 CN）是协调节点，是数据库服务的对外入口，负责数据的分发和查询规划，多个节点位置对等。业务请求发送给 CN 后，无需关心数据计算和存储的细节，由 CN 统一返回执行结果。CN 上只存储系统的元数据，并不存储实际的业务数据，可以配合支持业务接入增长动态增加。

- Datanode

  Datanode（简称 DN）是数据节点，执行协调节点分发的执行请求，实际存储业务数据。各个 DN 可以部署在不同的物理机上，也支持同物理机部署多个 DN节点，DN互为主备节点不能部署在同一台物理机上。DN 节点存储空间彼此之间独立、隔离，是标准的 share nothing 存储拓扑结构。另外TBase-V2与V1最大的不同地方是DN与DN之间可以通信，互相交换数据。

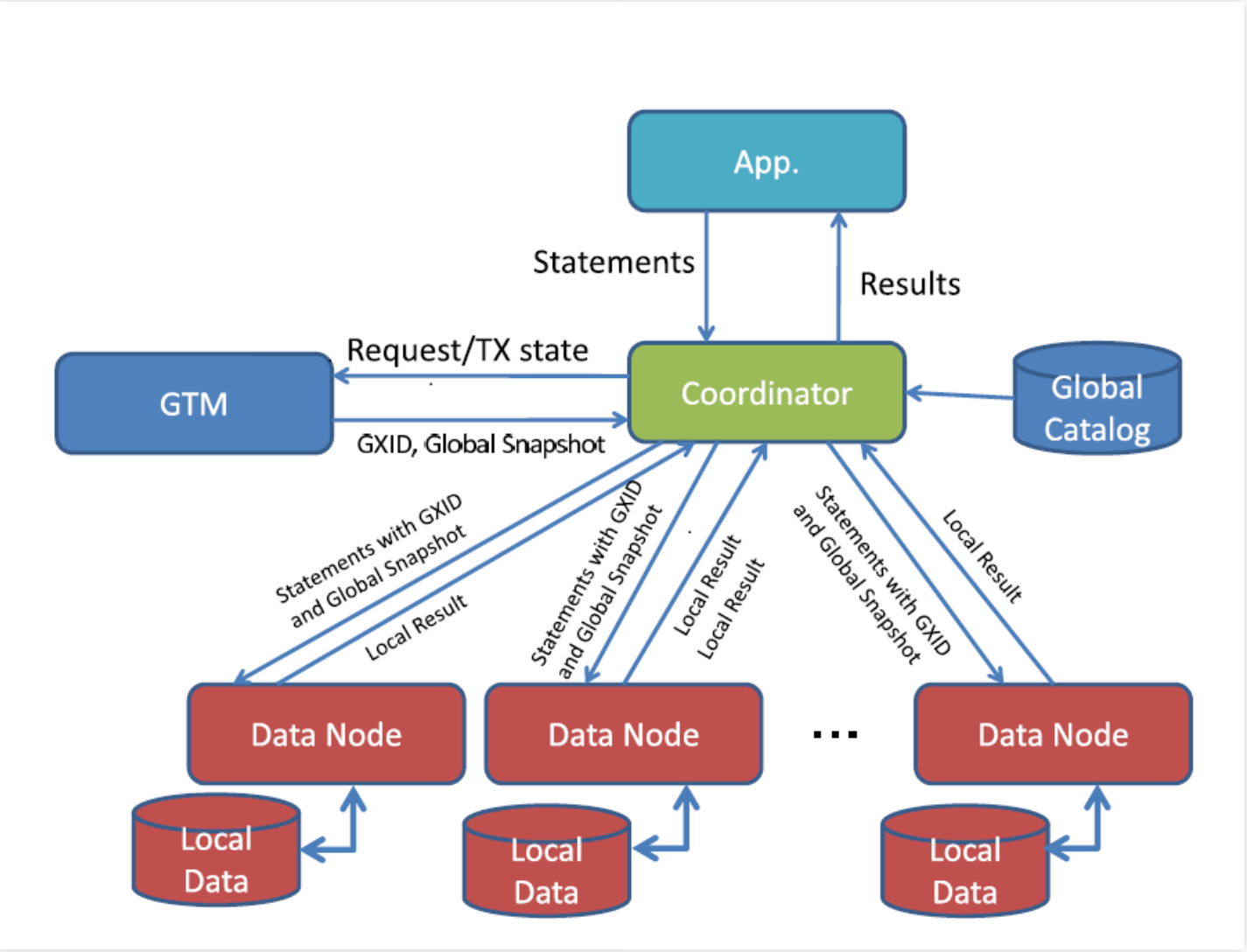
- GlobalTransactionManager

  GlobalTransactionManager（简称 GTM），是全局事务管理器，负责全局事务管理。GTM上不存储业务数据。

  TDSQL for PostgreSQL 每个CN和DN节点可以说是一个PostgreSQL的实例，从实现上来讲，

  TDSQL for PostgreSQL 就是在PG的代码上加了集群的相关功能，从而做成了一个MPP的数据库集群。

SQL执行过程



1. 业务发送请求到Coord节点，Coord节点向GTM请求事务信息；
2. Coord发送SQL语句和事务信息到Datanode；
3. Datanode执行完SQL后返回结果给CN；
4. CN收集DN的结果并汇总会后返回给业务。

产品优势

SQL兼容度高

TDSQL PostgreSQL 版不仅兼容绝大多数的PostgreSQL语法，包括复杂查询、外键、触发器、视图、存储过程等等，满足大部分企业用户的诉求。同时还兼容大部分的Oracle类型、函数。

分布式事务全局一致性

TDSQL PostgreSQL 版引入全局事务管理节点（GTM，Global Transaction Manager）来专门处理分布式事务一致性，通过拥有自主专利的分布式事务一致性技术，包括两阶段提交（Two Phase Commit）以及全局时钟（Global Timestamp）的策略来保证在全分布式环境下的事务一致性。TDSQL PostgreSQL 版单实例的TPCC事务处理能力得到大幅度的提升，而且处理能力会随着实例规模准线性提升。

在线扩容能力

为了迎接业务的快速增长，系统不可避免的需要进行扩容，传统的分布式数据库所采用分库分表模式的实现使得扩容成本高昂，需要对业务进行长时间的中断。TDSQL PostgreSQL 版在新加节点时，只需要把一些shardmap中的shardid映射到新加的节点，并把对应的数据进行搬迁。扩容对业务的中断仅仅是在切换shardmap中映射关系的时刻，时间大大缩短。

HTAP事务及分析双引擎方案

Hybrid Transactional/Analytical Processing，即事务和分析混合处理技术，这个技术要求本来资源诉求矛盾的两种业务类型在同一个数据库中完成处理。传统的数据库因为各方面的限制，偏向于OLTP或OLAP的场景，两者很难兼得。TDSQL PostgreSQL 版 V2经过专门的设计完美的做到了HTAP，同时具备了高效的OLTP能力和海量的OLAP处理能力，降低业务复杂度和业务成本。

多级容灾能力保证

TDSQL PostgreSQL 版在多个维度保证集群的容灾能力。其中采用强同步复制来保证主从数据完全一致，保障主节点故障时数据无丢失；提供基于任意时间点的恢复特性来防止误操作带来的数据丢失。

资源隔离能力-读写多平面

TDSQL PostgreSQL 版作为一款HTAP数据库，天然具备一写多读的能力。一个实例中每个数据节点(DN)有多个备机，同时协调节点(CN)也有多个备机，这样在具备高可靠的同时，可以充分利用备机的计算能力，在备机上执行分析型的业务。即主节点的所组成的主平面提供读写业务(OLTP),而每个备平面提供一种只读业务(OLAP)，从而可以实现业务的读写分离。

卓越的数据安全保障能力

TDSQL PostgreSQL 版提出三权分立的体系，将传统数据库系统DBA的角色分解为三个相互独立的角色，安全管理员，审计管理员，数据管理员；基于此基础提出自己的安全策略，主要细分为三个部分，分别为数据加密，数据脱敏访问，强制访问控制，三者组合提供多个层级的数据安全保障能力。
TDSQL PostgreSQL 版提供两种数据加密方式，一是业务侧加密，业务调用TDSQL PostgreSQL 版内置的加密函数，将加密后的结果写入数据库，正常读取的也是加密后的数据，在应用里面执行解密。二是使用TDSQL PostgreSQL 版的内置加密功能，相对前一种对业务侧来是透明。
TDSQL PostgreSQL 版从多个维度来提供全方位的审计能力，同时TDSQL PostgreSQL 版审计通过旁路检测方式，对数据库运行效率的影响极小。

高效的数据治理能力

TDSQL PostgreSQL 版团队设计了专门的方案来解决数据倾斜问题。为了有效的降低业务的资源消耗成本，TDSQL PostgreSQL 版开发了冷热数据分离的功能，内核原生态支持数据的冷热分离，业务无需感知底层存储介质的不同，对外提供一个统一的数据库视图。

全面的运维能力

OSS系统是TDSQL PostgreSQL 版系统配套使用的平台管理系统，集租户管理、服务器资源管理、项目管理、实例监控运维管理于一体。其中租户管理，服务器资源管理，项目管理是实现多租户管理配套组件。监控运维系统设计的主要目的是监控和维护TDSQL PostgreSQL 版数据库，包括指标实时监控、告警，部分故障自动修复，在线扩容，数据搬迁等功能，与TDSQL PostgreSQL 版数据库组成一套完整的体系，组成一个高效、稳定、可靠的分布式数据库系统。

购买指南
计费概述

计费概述

总费用 = (节点总内存数 × 内存价格 + 节点总磁盘数 × 磁盘价格) × 时长

流量费用目前不计费。

> 说明：
>
> - 节点的总内存数为实例中所有 CN 节点和 DN 节点的内存数量之和，包含所有的备节点。
> - 节点的总磁盘数为实例中所有 CN 节点和 DN 节点的磁盘数量之和，包含所有的备节点。
> - 实例的 GTM 节点当前不收费。

计费示例
【按量计费示例】：在城市1地域下，购买1个按量计费 TDSQL PostgreSQL版，其 CN 节点内存为4GB、硬盘为100GB，CN 节点1主1备，3组；DN 节点内存为12GB，硬盘为200GB，DN 节点1主1备，2组，使用时长为96个小时。

则所需支付的费用计算如下：

实例费用 = ((CN 节点总内存数 + DN 节点总内存数) × 内存价格 + (CN 节点总磁盘数 + DN 节点总磁盘数) × 磁盘价格) × 时长 = ((4GB × 2 × 3 + 12GB × 2 × 2) × XX元/GB/小时 + (100GB × 2 × 3 + 200GB × 2 × 2) × XX元/GB/小时) × 96小时 = XX元

购买指引

购买方式

1. 登录 TDSQL PostgreSQL 版购买页，根据实际需求选择各项配置信息，确认无误后，单击【立即购买】。

- 地域：选择您业务需要部署 TDSQL PostgreSQL版 的地域。

> 说明：
>
> 建议您选择与云服务器同一个地域，不同地域的云产品内网不通，购买后不能更换。

- 网络：TDSQL PostgreSQL版 所属网络。

> 说明：
>
> 建议您选择与云服务器同一个地域下的同一私有网络，否则无法通过内网连接云服务器和数据库，缺省设置为“Default-VPC（默认）”。

- 安全组：安全组创建与管理请参见 配置安全组。

- 购买数量：每个用户在每个可用区可购买按量计费实例的总数量为30个，可购买的包年包月实例总数量为30个。

2. 支付完成后，返回实例列表，会看到实例显示“发货中”（大概需要3min - 5min中，请耐心等待），待实例状态变为“运行中”，即可对实例进行操作。

查看账单

操作步骤

1. 登录控制台，单击右上角【费用】，进入【计费管理】页面。
2. 在左侧导航栏中，选择【账单管理】>【账单明细】>【资源 ID 账单】。

快速入门
快速入门（V5.06）
客户端psql的使用

psql简介

psql是一个基于终端的 TDSQL PostgreSQL 版客户端工具，类似Oracle中的命令行工具sqlplus，但比sqlplus更加强大。它让你能交互式地键入查询，把它们发送给 TDSQL PostgreSQL 版，并且查看查询结果。或者，输入可以来自于一个文件或者命令行参数。此外，psql还提供一些元命令和多种类似shell的特性来为编写脚本和自动化多种任务提供便利。psql还支持命令补全，历史命令回找。

psql使用环境配置

TDSQL PostgreSQL 版-v2是基于多租户设计，安装后所有机器都不会配置 TDSQL PostgreSQL 版的客户端默认使用环境，所以使用psql需要配置其环境变量。切换到数据库用户tbase，如下

```
export PGXZ_HOME=/usr/local/install/tbase_pgxz

export PATH=$PGXZ_HOME/bin:$PATH

export LD_LIBRARY_PATH=$PGXZ_HOME/lib:${LD_LIBRARY_PATH}

PGUSER=tbase

PGHOST=127.0.0.1

PGDATABASE=postgres

PGPORT=11000

export PGHOST PGUSER PGDATABASE PGPORT
```

这样就能使用psql客户端了，因为测试只初始化了一个实例，所以我们可以将它配置到 ~/.bashrc 中。

创建 TDSQL PostgreSQL 版默认的分布式使用环境

TDSQL PostgreSQL 版作为分布式数据库系统，使用前我们必需配置数据表默认分布的数据节点（DN），下面演示如何创建一个default node group。

切换到 TDSQL PostgreSQL 版用户：su - tbase

- 连接数据库

> 注意：此处指连接到cn节点（后面没特别说明，所有数据库操作都是连接到cn节点）

```
psql -p port -d database -U user -h
```

host（psql连接参数就是这些，超级管理用户就是数据库用户tbase，默认数据库为postgres，host如果是在本机连接可省略，默认使用本机ip）

示例：

```
[tbase@node1 ~]$ psql -p 11000 -d postgres

psql (PostgreSQL 10.0 TBase V2)

Type "help" for help.
```

- 查询集群节点配置

```
postgres=# select * from pgxc_node;
node_name | node_type | node_port | node_host | nodeis_primary |
nodeis_preferred | node_id | node_cluster_name
-----------+-----------+-----------+------------+---------------+-----------------+------------
+------------------
agtm_0 | G | 11004 | 172.16.0.12 | t | f | 475343005 | tbase_cluster
cn001 | C | 11000 | 172.16.0.12 | f | f | -264077367 | tbase_cluster
cn002 | C | 11000 | 172.16.0.5 | f | f | -674870440 | tbase_cluster
dn001 | D | 11002 | 172.16.0.12 | f | f | 2142761564 | tbase_cluster
dn002 | D | 11002 | 172.16.0.5 | f | f | -17499968 | tbase_cluster
dn003 | D | 11006 | 172.16.0.12 | f | f | -1956435056 | tbase_cluster
(6 rows)
```

- 查询当前数据节点（DN）,这些DN节点就是上面初始化集群时建立的

```
postgres=# select * from pgxc_node where node_type='D';

node_name | node_type | node_port | node_host | nodeis_primary |

nodeis_preferred | node_id | node_cluster_name

-----------+-----------+-----------+------------+---------------+-----------------+------------
+------------------

dn001 | D | 11002 | 172.16.0.12 | f | f | 2142761564 | tbase_cluster

dn002 | D | 11002 | 172.16.0.5 | f | f | -17499968 | tbase_cluster
```

dn003 | D | 11006 | 172.16.0.12 | f| f | -1956435056 | tbase_cluster

- 创建数据表默认使用的group

  postgres=# create default node group default_group with(dn001, dn002);

- 为default group创建shardmap

配置完成数据表默认使用的DN节点后，我们接下来需要配置记录的分区方案，shardmap就是 TDSQL PostgreSQL 版各个哈希值与DN的对照表，下面演示如何创建一个shardmap给default_group

执行以下指令，即可像单机一样使用 TDSQL PostgreSQL 版集群：

```
postgres=# create sharding group to group default_group;
postgres=# clean sharding;
```

更多group的使用方法

创建扩展group

- 建立group

  postgres=# create node group ext_group with(dn003);

  CREATE NODE GROUP

- 为group创建shard

  postgres=# create extension sharding group to group ext_group;

  CREATE SHARDING GROUP

  postgres=# clean sharding;

  CLEAN SHARDING

  postgres=#

删除group

```
postgres=# drop sharding in group ext_group;

DROP SHARDING GROUP

postgres=# drop node group ext_group ;

DROP NODE GROUP
```

查看集群group

```
postgres=#select * from pgxc_group;
```

psql常用命令使用

连接到一个数据库实例

- 使用参数连接

```
[pgxz@VM_0_3_centos root]$ psql -h 172.16.0.29 -p 15432 -U tbase -d postgres
psql (PostgreSQL 10 (TBase 2.01))
Type "help" for help.

postgres=#
```

- 使用conninfo字符串或者一个URI

```
[pgxz@VM_0_3_centos root]$ psql
postgresql://pgxz@172.16.0.29:15432/postgres
psql (PostgreSQL 10 (TBase 2.01))
Type "help" for help.

postgres=#
```

建立一个新连接

- 查看当前连接会话pid

```
postgres=# select pg_backend_pid();
pg_backend_pid
--
2408
(1 row)
```

- 连接到当前库

```
postgres=# \c
You are now connected to database "postgres" as user "tbase".
postgres=# select pg_backend_pid();
 pg_backend_pid
--
```

```
 2426
(1 row)
```

- 连接到其他库

```
postgres=# \c template1
You are now connected to database "template1" as user "tbase".
template1=#
```

- 连接到其他用户

```
postgres=# \c - postgres
You are now connected to database "postgres" as user "postgres".
```

- 连接到其他服务器上的库

```
postgres=# \c postgres tbase 172.16.0.5 11000

You are now connected to database "postgres" as user "tbase" on host
"172.16.0.5" at port "11000".
```

显示和设置该连接当前运行参数

- 显示当前连接用户

```
postgres=# SELECT CURRENT_USER;
current_user
--------------
pgxz
(1 row)
```

- 显示当前连接的schema

```
postgres=# show search_path ;
search_path
----------------
"$user",public
(1 row)
```

- 显示节点参数值

```
postgres=# show work_mem ;
work_mem
----------
4MB
(1 row)
```

- 设置当前连接的运行参数

```
postgres=# set search_path = "$user",public,pg_catalog;
SET
postgres=# set work_mem = '8MB';
SET
```

- 打开和关闭显示每个sql语句执行的时间

```
postgres=# \timing on
Timing is on.
postgres=# select count(1) from tbase;
count
-------
10000
(1 row)
Time: 5.139 ms
postgres=# \timing off
Timing is off.
postgres=# select count(1) from tbase;
count
-------
10000
(1 row)
```

- 配置行列显示格式

```
postgres=# \x on
Expanded display is on.
postgres=# select * from tbase where id=1;
-[ RECORD 1 ]
id | 1
mc | 1
-[ RECORD 2 ]
id | 1
mc | 2
-[ RECORD 3 ]
```

```
id | 1
mc | 2
postgres=# \x off
Expanded display is off.
postgres=# select * from tbase where id=1;
id | mc
----+----
1 | 1
1 | 2
1 | 2
(3 rows)
```

- 显示和配置客户端编码

```
postgres=# \encoding

UTF8
```

配置客户端编码为SQL_ASCII

```
postgres=# \encoding sql_ascii

postgres=# \encoding

SQL_ASCII
```

退出连接

```
postgres=# \q
```

psql执行一个sql命令

- 显示标题

```
[pgxz@VM_0_3_centos root]$ psql -h 172.16.0.29 -p 15432 -U tbase -d
postgres -c "select count(1) from pg_class"
count
-------
317
(1 row)
```

- 不显示标题

```
[pgxz@VM_0_3_centos root]$ psql -h 172.16.0.29 -p 15432 -U tbase -d
postgres -t -c "select count(1) from pg_class"
317
```

psql执行一个sql文件中所有命令

- 在外部执行

```
[pgxz@VM_0_3_centos ~]$ cat /data/tbase/tbase.sql
set search_path = public;
insert into tbase values(1,2);
select count(1) from tbase;
[pgxz@VM_0_3_centos ~]$ psql -h 172.16.0.29 -p 15432 -U tbase -d postgres
-f /data/tbase/tbase.sql
SET
INSERT 0 1
count
-------
10001
(1 row)
```

- 在内部执行

```
[pgxz@VM_0_3_centos ~]$ psql -h 172.16.0.29 -p 15432 -U tbase -d postgres
psql (PostgreSQL 10 (TBase 2.01))
Type "help" for help.
postgres=# \i /data/tbase/tbase.sql
SET
INSERT 0 1
count
-------
10002
```

(1 row)

数据库相关操作

- 显示当前集群中所有数据库

```
postgres=# \l
List of databases
Name | Owner | Encoding | Collate | Ctype | Access privileges
-----------+-------+----------+------------+------------+-------------------
postgres | tbase | UTF8 | en_US.utf8 | en_US.utf8 |
template0| tbase | UTF8 | en_US.utf8 | en_US.utf8| =c/tbase +
| | | | | tbase=CTc/tbase
template1 | tbase | UTF8 | en_US.utf8 | en_US.utf8 | =c/tbase +
| | | | | tbase=CTc/tbase
(3 rows)
```

- \l+显示当前当前集群中所有数据库（包含库大小及注释）

```
postgres=# \l+
List of databases
Name | Owner | Encoding | Collate | Ctype | Access privileges | Size
| Tablespace | Description
-----------+-------+----------+------------+------------+-------------------+-------+------------
+-------------------------------------------
postgres | tbase | UTF8 | en_US.utf8 | en_US.utf8 | | 31 MB |
pg_default | default administrative connection database

template0 | tbase | UTF8 | en_US.utf8 | en_US.utf8 | =c/tbase +| 27 MB
| pg_default | unmodifiable empty database

| | | | | tbase=CTc/tbase | | |

template1 | tbase | UTF8 | en_US.utf8 | en_US.utf8 | =c/tbase +| 27 MB
| pg_default | default template for new databases

| | | | | tbase=CTc/tbase | | |

(3 rows)
```

- 创建一个新库

```
postgres=# create database mydb;
CREATE DATABASE
```

模式相关操作

- \dn显示当前库所有模式

```
postgres=# \dn
List of schemas
Name | Owner
--------+-------
pgxc | tbase
public | tbase
(2 rows)
```

- \dn+显示当前库所有模式（包含注释）

```
postgres=# \dn+
List of schemas
Name | Owner | Access privileges | Description
--------+-------+-------------------+-----------------------
pgxc | tbase | |
public | tbase | tbase=UC/tbase +| standard public schema
| | =UC/tbase |
(2 rows)
```

- 创建一个新模式

```
postgres=# create schema mysche;

CREATE SCHEMA
```

用户相关操作

- \du显示当前集群中所有数据库用户

```
postgres=# \du
List of roles
```

```
Role name | Attributes | Member of
--------------+------------------------------------------------------------+-----------
audit_admin | No inheritance | {}
mls_admin | No inheritance | {}
tbase | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
tbase01_admin | Superuser, Create role, Create DB | {}
```

- \du+显示当前集群中所有数据库用户（包含注释）

```
postgres=# \du+
List of roles
Role name | Attributes | Member of | Description
--------------+------------------------------------------------------------+-----------+-------------
audit_admin | No inheritance | {} |
mls_admin | No inheritance | {} |
tbase | Superuser, Create role, Create DB, Replication, Bypass RLS | {} |
tbase01_admin | Superuser, Create role, Create DB | {} |
```

- 创建一个新的用户

```
postgres=# create role pgxc with login ;
CREATE ROLE
postgres=# create user pgxz ;（使用create user，那么用户自动拥有login权限）
```

- 配置用户密码

```
postgres=# \password pgxc
Enter new password:
Enter it again:
```

表相关操作

- 建立数据表

postgres=# create table tbase(id int,mc text) distribute by shard(id);
CREATE TABLE

```bash

- \d查看表结构，包括使用的触发器

postgres=# \d tbase

Table "public.tbase"
Column | Type | Modifiers
--------+---------+-----------
id | integer|
mc | text |

- \d+查看表结构（包含注释），表类型，分布节点

postgres=# \d+ tbase
Table "public.tbase"
Column | Type | Modifiers | Storage | Stats target | Description
--------+---------+-----------+----------+--------------+-------------
id | integer | | plain | |
mc | text | | extended | |
Has OIDs: no
Distribute By SHARD(id)
Location Nodes: ALL DATANODES

- \dt查看表列表

postgres=# \dt
List of relations
Schema | Name | Type | Owner
--------+--------------+-------+-------
public | t_time_range | table | tbase
public | tbase | table | tbase
(2 rows)
```

- \dt+查看表列表详细信息，包含表大小和注释

这里连接的节点如果是cn的话，表大小为所有dn节点大小之和，否则为只是该节点的表大小

```
postgres=# \dt+
List of relations
Schema | Name | Type | Owner | Size | Description
--------+--------------+-------+-------+---------+--------------------
public | t_time_range | table | tbase | 0 bytes | 这是一个日期分区表
public | tbase | table | tbase | 576 kB |
(2 rows)
```

- \dt+显示某个模式下的所有表

```
postgres=# \dt+ pgxc.*
List of relations
Schema | Name | Type | Owner | Size | Description
--------+------------+-------+-------+---------+-------------
pgxc | order_main | table | tbase | 0 bytes |
(1 row)
```

- \dt+表名 显示某个表的详细信息

```
postgres=# \dt+ tbase
List of relations
Schema | Name | Type | Owner | Size | Description
--------+-------+-------+-------+--------+-------------
public | tbase | table | tbase | 576 kB |
(1 row)
```

- \dt+通配符列出适配的表

```
postgres=# \dt+ t*
List of relations
Schema | Name | Type | Owner | Size | Description
--------+--------------+-------+-------+---------+--------------------
public | t_time_range | table | tbase | 0 bytes | 这是一个日期分区表
public | tbase | table | tbase | 576 kB |
(2 rows)
```

插件管理

- 查看当前库加载了那些插件

```
postgres=# \dx
List of installed extensions
Name | Version | Schema | Description
------------------+---------+------------+-------------------------------------------------------------
pg_stat_statements | 1.1 | public | track execution statistics of all SQL
statements executed
plpgsql | 1.0 | pg_catalog | PL/pgSQL procedural language
(2 rows)
```

- 给当前库加载插件

```
postgres=# create extension pg_stat_statements;
CREATE EXTENSION
```

- 删除当前库某个插件

```
postgres=# drop extension pg_stat_statements;
DROP EXTENSION
```

sql帮助命令

```
postgres=# \h
postgres=# \h create table;
\h 可以查看一些sql命令的使用方法。
```

快速入门（V5.21）
SQL基础操作
PG模式

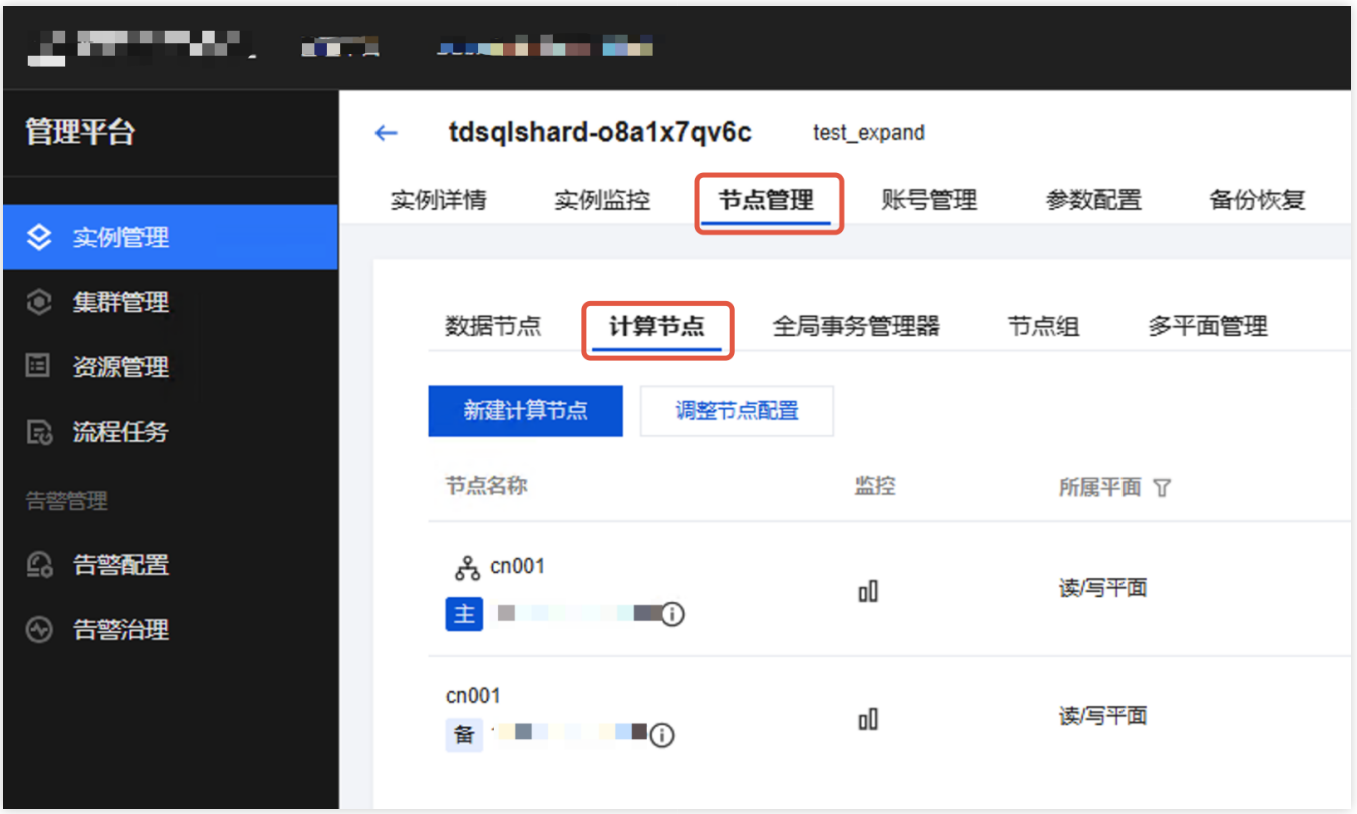
SQL基础操作 PG模式

本节主要介绍 TDSQL PG 在PG模式下的SQL基本操作。

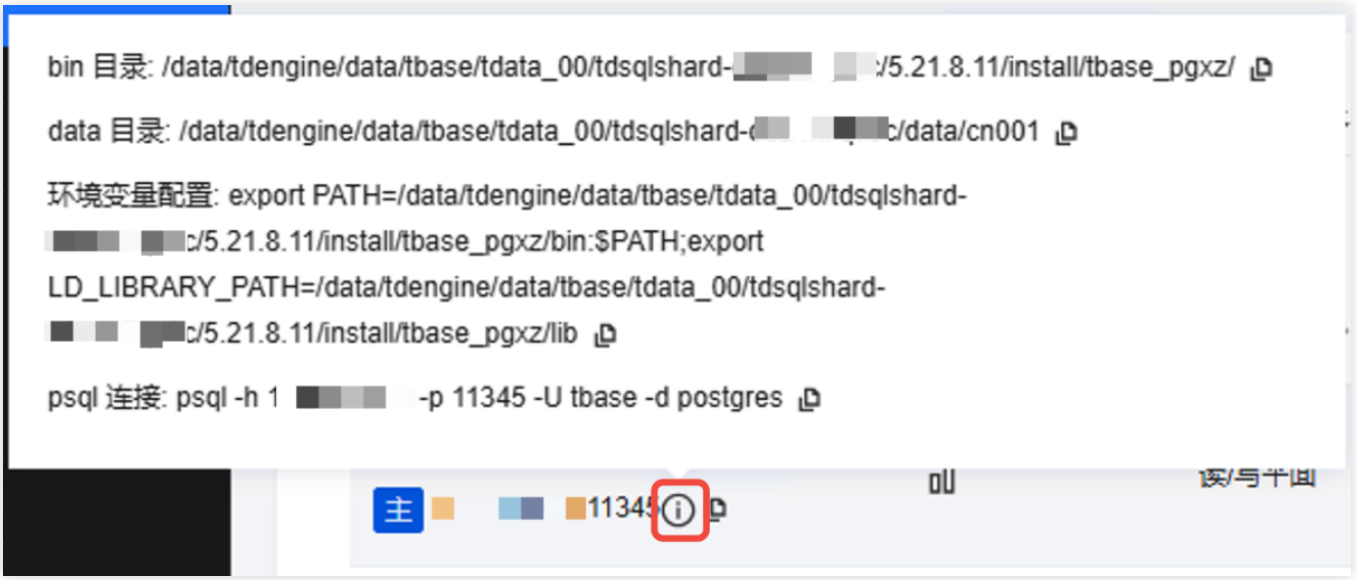
连接数据库

获取数据库连接配置信息，登录数据库系统：

1. 登录TDSQL管理平台在左侧选择 实例管理 并点击需要登录的实例，进入 节点管理 页（分布式下选择 计算节点 页，集中式选择 数据节点 页）。



2. 选择需要登录的节点IP，鼠标悬停在其右侧 图标获得环境配置信息。



3. 登录数据库节点服务器，切换到tbase用户并使用上述的export命令设置环境变量。

```
su - tbase
export PATH=/data1/tdengine/data/tbase/tdata_00/tdsqlshard-o62xkfw1a0/5.21.8.11/install/tbase_pgxz/bin:$PATH;
export LD_LIBRARY_PATH=/data1/tdengine/data/tbase/tdata_00/tdsqlshard-o62xkfw1a0/5.21.8.11/install/tbase_pgxz/lib
```

4. 在本机使用psql命令连接数据库（tbase用户一般只用于本机登录，可以创建普通用户用于业务开发）。

> 注意：
>
> 是连接到cn节点(后面没特别说明，所有数据库操作都是连接到cn节点)。

```
psql -h database_host_ip -p 11345 -U tbase -d postgres
```

创建用户

TDSQL PG可以用CREATE USER创建一个用户，需要使用超级用户或者具有CREATEROLE权限的用户（比如：tbase）。

- 示例：使用 create user 创建用户并指定密码。

```
postgres=# create user mgr password 'mypassword';
CREATE ROLE
```

创建完成后，使用 \du 展示用户信息。

```
postgres=# \du mgr
          List of roles
```

```
 Role name | Attributes | Member of
-----------+------------+-----------
 mgr       |            | {}
```

可以使用ALTER USER重新设置用户密码。

```
postgres=# alter user mgr password 'newpassword';
ALTER ROLE
```

创建数据库

要创建一个数据库，需要使用超级用户或者具有CREATEDB权限的用户。

- 示例：使用 create database 默认参数创建数据库。

```
postgres=# create database tdsql;
CREATE DATABASE
```

创建完成后，可以通过 \l 命令查看当前实例中的所有数据库。

```
postgres=# \l
                                  List of databases
    Name      | Owner | Encoding | Sqlmode  |  Collate   |   Ctype    | Access privileges
--------------+-------+----------+----------+------------+------------+-------------------
 ora          | tbase | UTF8     | oracle   | zh_CN.utf8 | zh_CN.utf8 |
 postgres     | tbase | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 |
 tdsql        | tbase | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 |
 template0    | tbase | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 | =c/tbase        +
              |       |          |          |            |            | tbase=CTc/tbase
 template0_ora | tbase | UTF8     | oracle   | zh_CN.utf8 | zh_CN.utf8 | =c/tbase        +
              |       |          |          |            |            | tbase=CTc/tbase
 template1    | tbase | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 | =c/tbase        +
              |       |          |          |            |            | tbase=CTc/tbase
 template1_ora | tbase | UTF8     | oracle   | zh_CN.utf8 | zh_CN.utf8 | =c/tbase        +
              |       |          |          |            |            | tbase=CTc/tbase
(7 rows)
```

在刚创建的 tdsql 中进行后续操作，使用 \c 命令切换数据库。

```
postgres=# \c tdsql;
You are now connected to database "tdsql" as user "tbase".
tdsql=#
```

创建模式

- 标准语句。

```
tdsql=# create schema tbase_pg;
CREATE SCHEMA
```

- 扩展语法，不存在时才创建。

```
tdsql=# create schema if not exists tbase_pg;
NOTICE:  (42P06) schema "tbase_pg" already exists, skipping
CREATE SCHEMA
```

表操作

创建表

使用命令 create table 创建表

1. 不用指定shard key建表，系统默认使用第一个字段做为表的shard key

```
create table shared_col_table(id serial not null,nickname text);
```

2. 指定shard key建表

```
create table t_appoint_col(id serial not null,nickname text) distribute by shard(nickname);
```

> 注意：
>
> 分布键选择原则
>
> - v5.21.x支持多列分布键
> - 如果有主键，则选择主键做分布键
> - 如果主键是复合字段组合，则选择过滤性好的字段做分布键
> - 没有主键的可以使用UUID来做分布键
> - 让数据尽可能的分布得足够散

3. 表不存在时才创建

```
create table IF NOT EXISTS t(id int,mc text);
```

查看表

使用 \d 命令查看表。

```
tdsql=# \d t_appoint_col;
                        Table "public.t_appoint_col"
  Column  |  Type   | Collation | Nullable |                  Default
----------+---------+-----------+----------+-------------------------------------------
 id       | integer |           | not null | nextval('t_appoint_col_id_seq'::regclass)
 nickname | text    |           |          |
```

修改表

使用命令 alter table 修改表

示例：

1. 修改表名

```
alter table t rename to tbase;
```

2. 增加表字段

```
alter table tbase add column age integer;
```

3. 修改字段类型

```
alter table tbase alter column age type float8;
```

4. 修改表所属模式

```
alter table tbase set schema public;
```

删除表

使用命令 drop table 删除表

示例：当表存在时删除

```
drop table if exists tbase;
```

索引操作

创建索引

使用命令 create index 创建索引

示例：

1. 创建唯一索引

```
create table t_first_col_share(id serial not null,nickname text);
create unique index t_first_col_share_id_uidx on t_first_col_share using btree(id);
```

> 注意：
>
> 非shard key字段不能建立唯一索引

2. 创建多字段索引

```
create table t_mul_idx (f1 int,f2 int,f3 int,f4 int);
create index t_mul_idx_idx on t_mul_idx(f1,f2,f3);
```

查看索引

1. 使用 \d 命令查看索引。

```
tdsql=# \d t_first_col_share;
                        Table "public.t_first_col_share"
 Column  |  Type   | Collation | Nullable |                  Default
---------+---------+-----------+----------+--------------------------------------------
 id      | integer |           | not null | nextval('t_first_col_share_id_seq'::regclass)
 nickname | text   |           |          |
Indexes:
    "t_first_col_share_id_uidx" UNIQUE, btree (id) WITH (checksum='on')
```

2. 使用 \di 命令查看索引

```
tdsql=# \di
                      List of relations
 Schema |          Name           | Type  | Owner |       Table
--------+-------------------------+-------+-------+-------------------
 public | t_first_col_share_id_uidx | index | tbase | t_first_col_share
 public | t_mul_idx_idx           | index | tbase | t_mul_idx
(2 rows)
```

3. 使用 pg_indexes 视图列出索引

```
SELECT
    indexname,
    indexdef
FROM
    pg_indexes
WHERE
```

```
    tablename = 't_first_col_share';
```

示例输出：

```
        indexname        |                                        indexdef
--------------------------
+-----------------------------------------------------------------------------------------------------------------------
 t_first_col_share_id_uidx | CREATE UNIQUE INDEX t_first_col_share_id_uidx ON public.t_first_col_share USING
btree (id) WITH (checksum='on')
(1 row)
```

删除索引

使用命令 DROP INDEX 删除索引

```
drop index t_first_col_share_id_uidx;
```

插入数据

insert用于向一张表中插入数据，期望插入的数据可以是一条，多条或者是一个select查询的结果集。

使用 INSERT INTO... 命令插入数据

- 创建表 tbase 并插入一行数据：

```
tdsql=# create table tbase(id serial not null,nickname text) distribute by shard(id);
CREATE TABLE
tdsql=# insert into tbase(id,nickname) values(1,'hello TDSQL PG');
INSERT 0 1
```

- 向表 tbase 插入多行数据：

```
tdsql=# insert into tbase(id,nickname) values(2,'TDSQL PG'),(3,null),(4,'TDSQL PG good');
COPY 3
tdsql=# select * from tbase;
 id |    nickname
----+----------------
  1 | hello TDSQL PG
  2 | TDSQL PG
  3 |
  4 | TDSQL PG good
(4 rows)
```

删除数据

delete用于删除表数据，可以全部删除或者部分删除。

- 示例，带条件删除：

```
tdsql=# select * from tbase;
 id |    nickname
----+----------------
  1 | hello TDSQL PG
  2 | TDSQL PG
  3 |
  4 | TDSQL PG good
(4 rows)

tdsql=# delete from tbase where id=4;
DELETE 1
```

- 示例，null条件的表达方式

```
tdsql=# delete from tbase where nickname is null;
DELETE 1
tdsql=# select * from tbase;
 id |    nickname
----+----------------
  1 | hello TDSQL PG
  2 | TDSQL PG
(2 rows)
```

更新数据

update用于更新表数据，可以更新一个字段或者多个字段。

```
tdsql=# update tbase set nickname ='Hello TDSQL PG' where id=1;
UPDATE 1
```

查询数据

select用于从一张表或视图中获取数据。
使用 select...from... 命令查询数据

- 创建 t_grouping 表并插入内容

```
create table t_grouping(id int,dep varchar(20),product varchar(20),num int);
insert into t_grouping values(1,'dept no1','phone',90);
insert into t_grouping values(2,'dept no1','computer',80);
insert into t_grouping values(3,'dept no1','phone',70);
insert into t_grouping values(4,'dept no2','computer',60);
insert into t_grouping values(5,'dept no2','phone',50);
insert into t_grouping values(6,'dept no2','computer',60);
insert into t_grouping values(7,'dept no3','phone',70);
insert into t_grouping values(8,'dept no3','computer',80);
insert into t_grouping values(9,'dept no3','phone',90);
```

- 查询 t_grouping 表所有内容

```
tdsql=# select * from t_grouping;
 id |   dep    | product  | num
----+----------+----------+-----
  1 | dept no1 | phone    |  90
  2 | dept no1 | computer |  80
  5 | dept no2 | phone    |  50
  6 | dept no2 | computer |  60
  8 | dept no3 | computer |  80
  9 | dept no3 | phone    |  90
  3 | dept no1 | phone    |  70
  4 | dept no2 | computer |  60
  7 | dept no3 | phone    |  70
(9 rows)
```

- 按dep，product汇总，排序展示

```
tdsql=# select dep,product,sum(num) from t_grouping group by dep,product order by dep,product;
   dep    | product  | sum
----------+----------+-----
 dept no1 | computer |  80
 dept no1 | phone    | 160
 dept no2 | computer | 120
 dept no2 | phone    |  50
 dept no3 | computer |  80
 dept no3 | phone    | 160
(6 rows)
```

- 带条件查询

```
tdsql=# select * from t_grouping where dep='dept no2';
 id |   dep    | product  | num
----+----------+----------+-----
  5 | dept no2 | phone    |  50
```

```
  6 | dept no2 | computer |  60
  4 | dept no2 | computer |  60
(3 rows)
```

提交事务

事务提交用于保存对数据库的更改命令。

命令如下：

COMMIT; 或者 COMMIT TRANSACTION;

例：

```
insert into tbase(id,nickname) values(5,'hello new TDSQL PG');
BEGIN;
delete from tbase where id=5;
COMMIT;
```

> 注意：
>
> 在COMMIT执行之前，tbase表中id为5的记录会被另一会话查询到。

回滚事务

事务回滚用于撤销尚未保存到数据库的命令。

命令语法如下：

ROLLBACK; 或者 ROLLBACK to savepoint_name;

例1：

```
BEGIN;
delete from tbase where id in (1,2);
select * from tbase;
ROLLBACK;
select * from tbase;
```

> 注意：
>
> 在ROLLBACK执行之后，事务对数据的删除不会保存到数据库中。

例2:

```
BEGIN;
delete from tbase where id =1;
select * from tbase;
savepoint A;
delete from tbase where id =2;
```

```
select * from tbase;
ROLLBACK to A;
COMMIT;
```

> 注意：
>
> 在ROLLBACK to A;执行之前同一进程查询不到id为1和2的数据。回滚到保存点之后，id为2数据可查询到。
> ROLLBACK to savepoint_name;不会结束事务，需使用COMMIT;提交事务。

Oracle模式

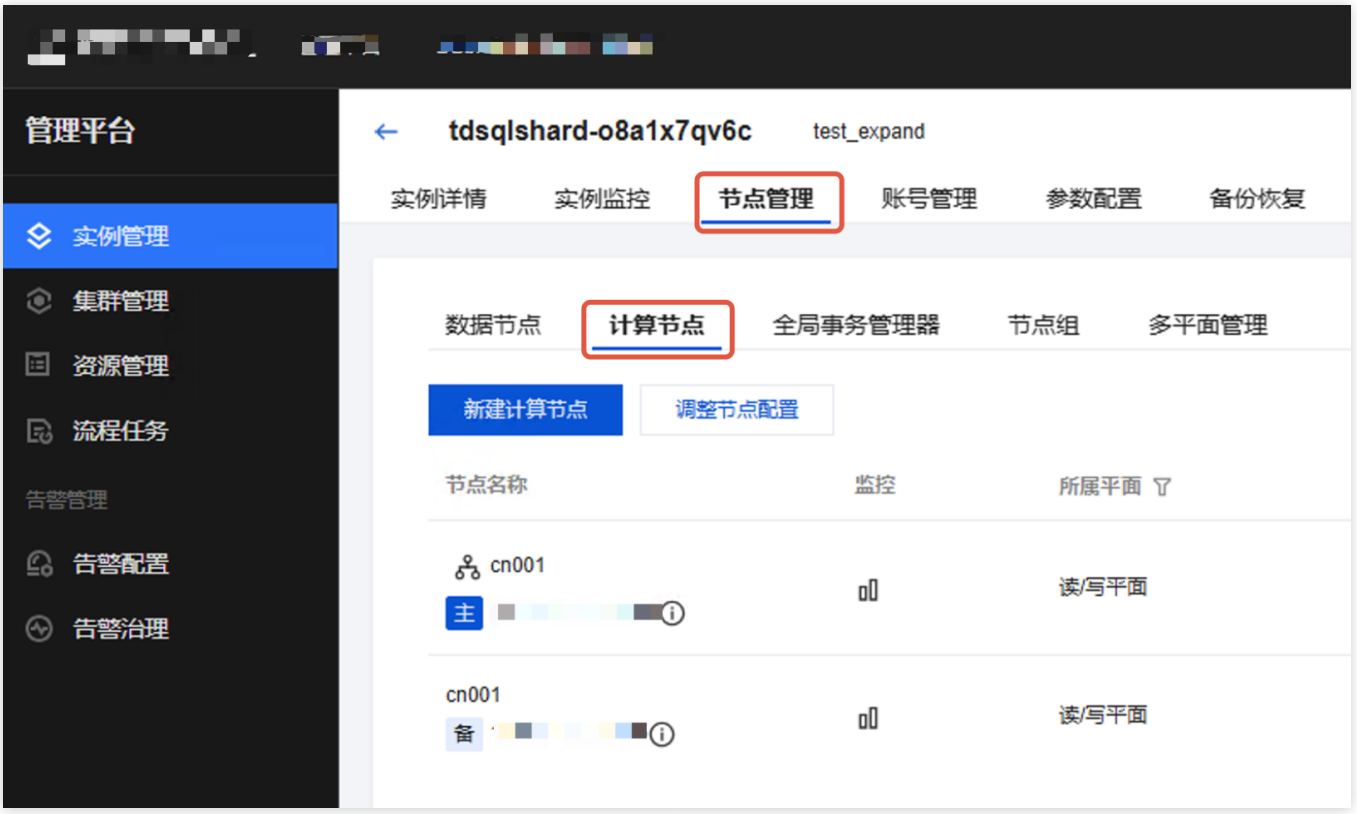
SQL基础操作 Oracle模式

本节主要介绍 TDSQL PG 在Oracle模式下的SQL基本操作。

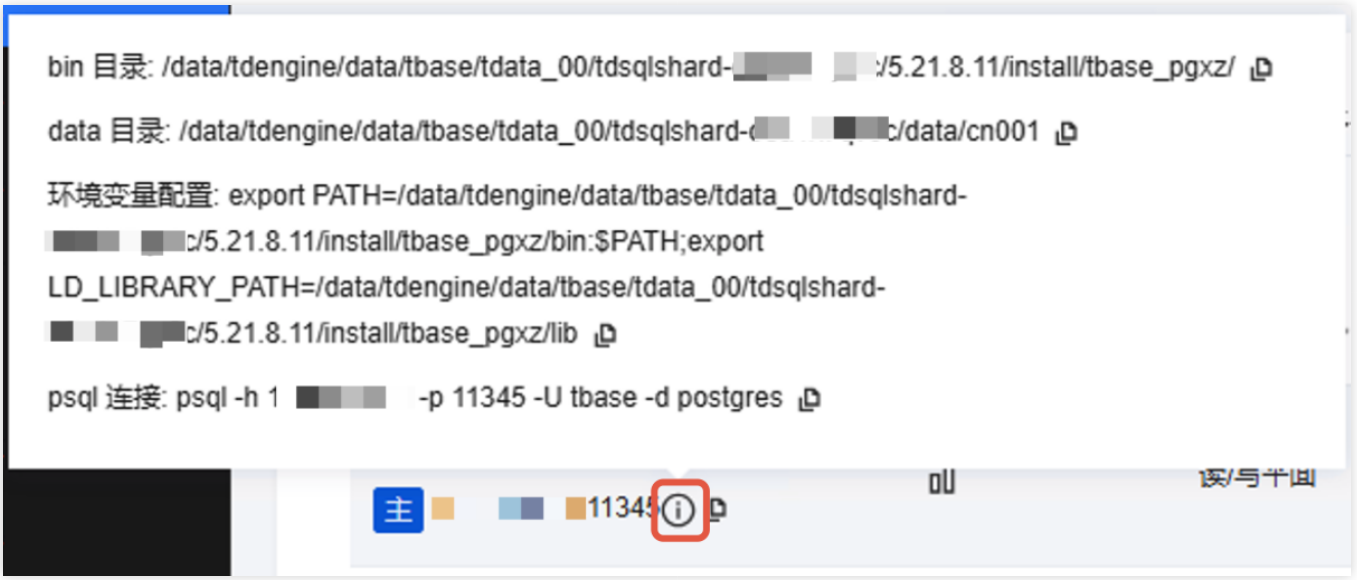
连接数据库

获取数据库连接配置信息，登录数据库系统：

1. 登录TDSQL管理平台在左侧选择 实例管理 并点击需要登录的实例，进入 节点管理 页（分布式下选择 CN 页，集中式选择 DN 页）。



2. 选择需要登录的节点IP，鼠标悬停在其右侧 图标获得环境配置信息。



3. 登录数据库节点服务器，切换到tbase用户并使用上述的export命令设置环境变量。

```
su - tbase
export PATH=/data1/tdengine/data/tbase/tdata_00/tdsqlshard-o62xkfw1a0/5.21.8.11/install/tbase_pgxz/bin:$PATH;
export LD_LIBRARY_PATH=/data1/tdengine/data/tbase/tdata_00/tdsqlshard-o62xkfw1a0/5.21.8.11/install/tbase_pgxz/lib
```

4. 在本机使用psql命令连接数据库（tbase用户一般只用于本机登录，可以创建普通用户用于业务开发）。

> 注意：
>
> 是连接到cn节点(后面没特别说明，所有数据库操作都是连接到cn节点)。

```
psql -h database_host_ip -p 11345 -U tbase -d postgres
```

创建用户

TDSQL PG可以用CREATE USER创建一个用户，需要使用超级用户或者具有CREATEROLE权限的用户（比如：tbase）。

- 示例：使用 create user 创建用户并指定密码。

```
postgres=# create user mgr password 'mypassword';
CREATE ROLE
```

创建完成后，使用 \du 展示用户信息。

```
postgres=# \du mgr
          List of roles
```

```
 Role name | Attributes | Member of
-----------+------------+-----------
 mgr       |            | {}
```

可以使用 ALTER USER 重新设置用户密码。

```
postgres=# alter user mgr password 'newpassword';
ALTER ROLE
```

启用Oracle兼容特性

TDSQL PG内核引擎会根据数据库类型来区分Oracle 兼容模式与PG 模式。
在创建数据库的时候使用 sql mode``db_mode 关键字即可创建一个指定模式的数据库，db_mode可选oracle或postgresql。不指定即默认PostgreSQL模式。
示例：

```
postgres=# create database ora sql mode oracle;
CREATE DATABASE

postgres=# \l ora
                              List of databases
 Name | Owner | Encoding | Sqlmode |  Collate   |   Ctype    | Access privileges
------+-------+----------+---------+------------+------------+-------------------
 ora  | tbase | UTF8     | oracle  | en_US.utf8 | en_US.utf8 |
(1 row)

postgres=# create database pg_db;
CREATE DATABASE
postgres=# \l pg_db
                               List of databases
 Name  | Owner | Encoding | Sqlmode  |  Collate   |   Ctype    | Access privileges
-------+-------+----------+----------+------------+------------+-------------------
 pg_db | tbase | UTF8     | postgres | en_US.utf8 | en_US.utf8 |
(1 row)
```

可以看到 ora 数据库的 Sqlmode 列显示 oracle ，即表示 ora 数据库当前为 Oracle 模式。
而 pg_db 数据库的 Sqlmode 列显示 postgres ，即表示 pg_db 数据库当前为 PostgreSQL 模式。
接下来切换至 ora 数据库或者直接访问 ora 数据库即可使用Oracle模式。

```
\c ora
```

或：

```
psql -h your_server_ip -p your_server_port -U tbase -d ora
```

创建模式

- 标准语句。

```
ora=# create schema tbase_ora;
CREATE SCHEMA
```

- 扩展语法，不存在时才创建。

```
ora=# create schema if not exists tbase_ora;
NOTICE:  schema "TBASE_ORA" already exists, skipping
CREATE SCHEMA
```

表操作

创建表

使用命令 create table 创建表

1. 不用指定shard key建表，系统默认使用第一个字段做为表的shard key

```
create table shared_col_table(id serial not null,nickname text);
```

2. 指定shard key建表

```
create table t_appoint_col(id serial not null,nickname text) distribute by shard(nickname);
```

> 注意：
>
> 分布键选择原则
>
> - v5.21.x支持多列分布键
> - 如果有主键，则选择主键做分布键
> - 如果主键是复合字段组合，则选择过滤性好的字段做分布键
> - 没有主键的可以使用UUID来做分布键
> - 让数据尽可能的分布得足够散

3. 表不存在时才创建

```
create table IF NOT EXISTS t(id int,mc text);
```

查看表

使用 \d 命令查看表

```
ora=# \d t_appoint_col;
                        Table "public.T_APPOINT_COL"
  Column  |  Type   | Collation | Nullable |                 Default
----------+---------+-----------+----------+-----------------------------------------
 ID       | INTEGER |           | not null | NEXTVAL('T_APPOINT_COL_ID_SEQ'::REGCLASS)
 NICKNAME | TEXT    |           |          |
```

使用Oracle兼容特性

使用层次查询获取数据

```
ora=# create table ora_table(a int, b int);
CREATE TABLE
ora=# insert into ora_table values(1, 2);
INSERT 0 1
ora=# insert into ora_table values(1, 2);
INSERT 0 1
ora=# select * from ora_table connect by level < 2;
 A | B
---+---
 1 | 2
 1 | 2
(2 rows)
```

使用ROWNUM对行进行编号

```
ora=# select rownum, a, b from ora_table where rownum = 1;
 ROWNUM | A | B
--------+---+---
      1 | 1 | 2
(1 row)
```

修改表

使用命令 alter table 修改表

示例：

1. 修改表名

```
alter table t rename to tbase;
```

2. 增加表字段

```
alter table tbase add column age integer;
```

3. 修改字段类型

```
alter table tbase alter column age type float8;
```

4. 修改表所属模式

```
alter table tbase set schema public;
```

删除表

使用命令 drop table 删除表
示例：当表存在时删除

```
drop table if exists tbase;
```

索引操作

创建索引

使用命令 create index 创建索引
示例：

1. 创建唯一索引

```
create table t_first_col_share(id serial not null,nickname text);
create unique index t_first_col_share_id_uidx on t_first_col_share using btree(id);
```

> 注意：
>
> 非shard key字段不能建立唯一索引

2. 创建多字段索引

```
create table t_mul_idx (f1 int,f2 int,f3 int,f4 int);
create index t_mul_idx_idx on t_mul_idx(f1,f2,f3);
```

查看索引

1. 使用 \d 命令查看索引。

```
ora=# \d t_first_col_share;
                        Table "public.T_FIRST_COL_SHARE"
  Column  |  Type   | Collation | Nullable |                  Default
----------+---------+-----------+----------+-------------------------------------------
 ID       | INTEGER |           | not null | NEXTVAL('T_FIRST_COL_SHARE_ID_SEQ'::REGCLASS)
 NICKNAME | TEXT    |           |          |
Indexes:
    "T_FIRST_COL_SHARE_ID_UIDX" UNIQUE, btree (ID) WITH (checksum='on')
```

2. 使用 \di 命令查看索引

```
ora=# \di
                      List of relations
 Schema |          Name           | Type  | Owner |       Table
--------+-------------------------+-------+-------+-------------------
 public | T_FIRST_COL_SHARE_ID_UIDX | index | tbase | T_FIRST_COL_SHARE
 public | T_MUL_IDX_IDX             | index | tbase | T_MUL_IDX
(2 rows)
```

3. 使用 user_indexes 视图列出索引

```
ora=# select INDEX_NAME from user_indexes where table_name='T_FIRST_COL_SHARE';
       INDEX_NAME
--------------------------
 T_FIRST_COL_SHARE_ID_UIDX
(1 row)
```

删除索引

使用命令 DROP INDEX 删除索引

```
drop index t_first_col_share_id_uidx;
```

插入数据

insert用于向一张表中插入数据，期望插入的数据可以是一条，多条或者是一个select查询的结果集。

使用 INSERT INTO... 命令插入数据

- 创建表 tbase 并插入一行数据：

```
ora=# create table tbase(id serial not null,nickname text) distribute by shard(id);
CREATE TABLE
ora=# insert into tbase(id,nickname) values(1,'hello TDSQL PG');
```

```
INSERT 0 1
```

- 向表 tbase 插入多行数据：

```
ora=# insert into tbase(id,nickname) values(2,'TDSQL PG'),(3,null),(4,'TDSQL PG good');
COPY 3
ora=# select * from tbase;
 ID |    NICKNAME
----+----------------
  1 | hello TDSQL PG
  2 | TDSQL PG
  3 |
  4 | TDSQL PG good
(4 rows)
```

删除数据

delete用于删除表数据，可以全部删除或者部分删除。

- 示例，带条件删除：

```
ora=# select * from tbase;
 ID |    NICKNAME
----+----------------
  1 | hello TDSQL PG
  3 |
  4 | TDSQL PG good
  2 | TDSQL PG
(4 rows)

ora=# delete from tbase where id=4;
DELETE 1
```

- 示例，null条件的表达方式

```
ora=# delete from tbase where nickname is null;
DELETE 1
ora=# select * from tbase;
 ID |    NICKNAME
----+----------------
  1 | hello TDSQL PG
  2 | TDSQL PG
(2 rows)
```

更新数据

update用于更新表数据，可以更新一个字段或者多个字段。

```
ora=# update tbase set nickname ='Hello TDSQL PG' where id=1;
UPDATE 1
```

查询数据

select用于从一张表或视图中获取数据。
使用 select ... from ... 命令查询数据

- 创建 t_grouping 表并插入内容

```
create table t_grouping(id int,dep varchar(20),product varchar(20),num int);
insert into t_grouping values(1,'dept no1','phone',90);
insert into t_grouping values(2,'dept no1','computer',80);
insert into t_grouping values(3,'dept no1','phone',70);
insert into t_grouping values(4,'dept no2','computer',60);
insert into t_grouping values(5,'dept no2','phone',50);
insert into t_grouping values(6,'dept no2','computer',60);
insert into t_grouping values(7,'dept no3','phone',70);
insert into t_grouping values(8,'dept no3','computer',80);
insert into t_grouping values(9,'dept no3','phone',90);
```

- 查询 t_grouping 表所有内容

```
ora=# select * from t_grouping;
 ID |   DEP    | PRODUCT  | NUM
----+----------+----------+-----
  1 | dept no1 | phone    |  90
  2 | dept no1 | computer |  80
  5 | dept no2 | phone    |  50
  6 | dept no2 | computer |  60
  8 | dept no3 | computer |  80
  9 | dept no3 | phone    |  90
  3 | dept no1 | phone    |  70
  4 | dept no2 | computer |  60
  7 | dept no3 | phone    |  70
(9 rows)
```

- 按dep，product汇总，排序展示

```
ora=# select dep,product,sum(num) from t_grouping group by dep,product order by dep,product;
   DEP    | PRODUCT  | SUM
----------+----------+-----
 dept no1 | computer |  80
 dept no1 | phone    | 160
 dept no2 | computer | 120
 dept no2 | phone    |  50
 dept no3 | computer |  80
 dept no3 | phone    | 160
(6 rows)
```

- 带条件查询

```
ora=# select * from t_grouping where dep='dept no2';
 ID |   DEP    | PRODUCT  | NUM
----+----------+----------+-----
  5 | dept no2 | phone    |  50
  6 | dept no2 | computer |  60
  4 | dept no2 | computer |  60
(3 rows)
```

提交事务

事务提交用于保存对数据库的更改命令。

命令如下：

COMMIT; 或者 COMMIT TRANSACTION;

例：

```
insert into tbase(id,nickname) values(5,'hello new TDSQL PG');
BEGIN;
delete from tbase where id=5;
COMMIT;
```

> 注意：
>
> 在COMMIT执行之前，tbase表中id为5的记录会被另一会话查询到。

回滚事务

事务回滚用于撤销尚未保存到数据库的命令。

命令语法如下：

ROLLBACK; 或者 ROLLBACK to savepoint_name;

例1：

```
BEGIN;
delete from tbase where id in (1,2);
select * from tbase;
ROLLBACK;
select * from tbase;
```

> 注意：
>
> 在ROLLBACK执行之后，事务对数据的删除不会保存到数据库中。

例2:

```
BEGIN;
delete from tbase where id =1;
select * from tbase;
savepoint A;
delete from tbase where id =2;
select * from tbase;
ROLLBACK to A;
COMMIT;
```

> 注意：
>
> 在ROLLBACK to A;执行之前同一进程查询不到id为1和2的数据。回滚到保存点之后，id为2数据可查询到。
> ROLLBACK to savepoint_name;不会结束事务，需使用COMMIT;提交事务。

操作指南
实例管理

实例创建

1. 登录管理控制台选择TDSQL PostgreSQL 版菜单，进入TDSQL PostgreSQL 版的控制台，可查看实例列表。
2. 点击【新建】按钮，创建实例。
3. 新建实例时需指定实例所属地域和可用区，网络类型以及安全组。具体如下图所示：



同时需要指定实例数据库内核版本以及CN、DN的节点配置。



确认实例登录用户名密码、字符集、数据复制方式后点击【立即购买】完成实例创建。



实例销毁

进入TDSQL PostgreSQL 版管理控制台，在实例列表的操作列点击【更多】->【销毁】来销毁一个实例。注意：实例销毁后数据将被清空，请谨慎操作。



修改实例名

进入控制台实例列表页面，点击实例名称旁边的图标可修改实例名称。具体如下图所示。





查看实例详情

在控制台实例列表点击实例ID或者实例操作列点击【管理】可查看实例详情。展示实例基本信息、配置信息以及可用性信息，如下图所示。



重置密码

在TDSQL PostgreSQL 版管理控制台实例列表页面，操作列点击【管理】->【帐号管理】->【重置密码】来重置实例生产时指定的密码。



调整节点配置

可以扩容或者缩容节点配置，在实例详情页面的实例【配置信息】中点击【调整节点配置】，弹框如下：



快速扩容

对于分布式实例，可以进行快速扩容，将一组DN节点快速扩容至2组DN节点。

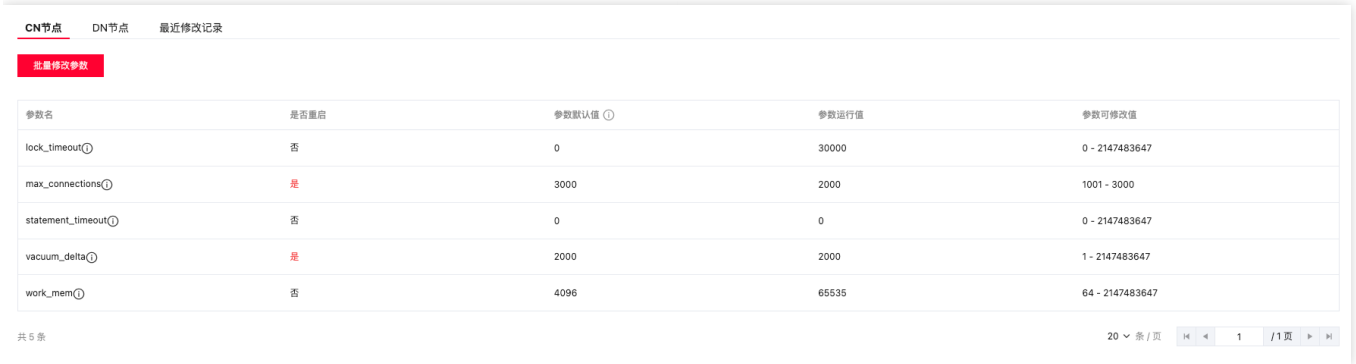


参数配置

批量修改参数

1. 登录 TDSQL PostgreSQL 版控制台，在实例列表，单击实例 ID，进入实例管理页面。
2. 在实例管理页面，选择参数配置页，单击批量修改参数。

> 说明：
>
> CN 节点和 DN 节点可分开单独修改。



3. 根据参数可修改值列的提示，输入目标参数值，单击确认修改保存，单击取消可取消操作。

- lock_timeout：锁等待超时。语句在试图获取表、索引、行或其他数据库对象上的锁时等到超过指定的毫秒数，该语句将被中止。

- max_connections: 数据库最大的并发连接数。
- statement_timeout：控制命令执行时长，单位是 ms。超过设定值，该语句将被中止。0代表不控制。用户可以在 session 级设置参数。
- vacuum_delta：设置执行 VACUUM 和 HOT 清理事务的时间间隔（如果有）。
- work_mem：设置内部排序操作和 Hash 表在开始使用临时磁盘文件之前使用的内存限制，默认4MB。
- lock_timeout：锁等待超时。语句在试图获取表、索引、行或其他数据库对象上的锁时等到超过指定的毫秒数，该语句将被中止。

4. 重启实例，可在实例详情页面的右上角单击重启实例，也可以在实例列表的操作列选择更多 -> 重启。

查看参数修改记录

在实例管理页面，选择参数配置 > 最近修改记录页，可查看近期参数修改记录。



备份

为防止数据丢失或损坏，可以使用自动备份或手动备份的方式来备份数据库。

备份方式

TDSQL PostgreSQL版 支持自动备份和手动备份两种方式来备份数据库。

备份类型

TDSQL PostgreSQL版 支持物理备份，物理数据全拷贝（自动备份与手动备份均支持）。

> 说明
>
> 物理备份期间系统会对每个平面 CN 执行加锁，业务无法执行 DDL 语句。

自动备份

在实例管理页面，选择备份 > 自动备份设置页，单击编辑。

1. 自动备份设置，单击确定。参数说明如下：



参数	说明
数据备份保留时间	系统默认备份文件保留7天
备份开始时间	默认时间为系统自动分配的备份发起时间。 可自定义选择时间区间，建议设置为业务低峰期。备份发起时间只是备份开始启动的时间，并不代表备份结束的时间。例如，选择：02:00 - 04:00点，系统会在02:00 - 04:00时间范围内的某一个时间点发起备份，具体的发起时间点取决于服务端备份策略和备份系统状况。
备份时间间隔	默认设置为每24小时备份一次。

手动备份

手动备份功能允许用户自助发起备份任务。

> 说明
>
> 实例在执行每天的自动备份任务期间，无法发起手动备份。

在实例管理页面，选择备份 > 手动备份。在弹出的对话框，确认无误后，单击确定。


手动备份
您正准备备份此实例，点击确认后备份任务将在一分钟左右开始启动
确定
取消

">

隔离方式说明

TDSQL PostgreSQL 版和PostgreSQL当前支持两种隔离方式：实例级和节点级。分别说明如下：

实例级

实例级隔离指定的是当前实例在每台机器上所能使用的CPU和内存资源的上限。例如，某个实例指定的是实例级隔离方式，指定的隔离资源是10核CPU 20GB内存，实例最终在3台机器上部署，则该实例最终可使用的资源是30核CPU 60GB内存。

节点级

节点级隔离指定的是当前实例的每一个节点所能使用的CPU和内存资源的上限。例如，某实例指定的是节点级隔离方式，指定的隔离资源是10核CPU 20GB内存，实例一共有6个节点，则实例最终可使用的资源是60核120GB内存。

数据库使用（V5.06）
数据库使用说明

psql是一个基于终端的 TDSQL PostgreSQL 版客户端工具，类似Oracle中的命令行工具sqlplus，但比sqlplus强大多了。它让你能交互式地键入查询，把它们发送给 TDSQL PostgreSQL 版，并且查看查询结果。或者，输入可以来自于一个文件或者命令行参数。此外，psql还提供一些元命令和多种类似
shell
的特性来为编写脚本和自动化多种任务提供便利。psql还支持命令补全，历史命令回找。

连接数据库参数介绍

-d dbname

--dbname=dbname

指定要连接到的数据库名，不指定该参数时默认是从PGDATABASE环境变量中取得（如果被设置）。

-h host

--host=host

指定要连接服务器机器的主机名或者ip地址。不指定该参数时默认是从PGHOST环境变量中取得（如果被设置）。

-p port

--port=port

指定要连接服务器监听的 TCP端口，不指定该参数时默认是从PGPORT环境变量中取得（如果被设置），否则使用编译在程序中的默认值。

-U username

--username=username

指定要连接服务的用户名，不指定该参数时默认是从PGUSER环境变量中取得（如果被设置），否则使用当前操作系统用户名做为默认值。

-w

--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那么连接

尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W

--password

强制pg_dump在连接到一个数据库之前提示要求一个口令。

连接到一个数据库

- 使用参数连接

```
[pgxz@VM_0_3_centos root]$ psql -h 172.16.0.29 -p 15432 -U tbase -d
postgres

psql (PostgreSQL 10 (TBase 2.01))

Type "help" for help.

postgres=#
```

- 使用conninfo字符串或者一个URI

```
[pgxz@VM_0_3_centos root]$ psql
postgresql://pgxz@172.16.0.29:15432/postgres

psql (PostgreSQL 10 (TBase 2.01))

Type "help" for help.

postgres=#
```

- 配置证书连接

```
[pgxz@VM_0_3_centos ~]$ psql 'host=172.16.0.29 port=15432 dbname=postgres
user=tbase sslmode=verify-ca sslcert=postgresql.crt sslkey=postgresql.key
sslrooTCloudFinanceZonert=root.crt'

Password:

psql (PostgreSQL 10 (TBase 2.01))

SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
```

```
Type "help" for help.

postgres=> \q
```

- 配置环境变量后快捷连接

```
[pgxz@VM_0_3_centos root]$ PGUSER=tbase

[pgxz@VM_0_3_centos root]$ PGHOST=127.0.0.1

[pgxz@VM_0_3_centos root]$ PGDATABASE=postgres

[pgxz@VM_0_3_centos root]$ PGPORT=15432

[pgxz@VM_0_3_centos root]$ export PGHOST PGUSER PGDATABASE PGPORT

[pgxz@VM_0_3_centos root]$ psql

psql (PostgreSQL 10 (TBase 2.01))

Type "help" for help.

postgres=#
```

变些环境变量可以配置在

```
[pgxz@VM_0_3_centos root]$ cat ~/.bashrc

# .bashrc

PGUSER=tbase

PGHOST=127.0.0.1

PGDATABASE=postgres

PGPORT=15432

export PGHOST PGUSER PGDATABASE PGPORT
```

建立一个新连接

- 连接到另外一个库（也可以是当前库）

```
postgres=# select pg_backend_pid();
```

pg_backend_pid

2408

(1 row)

postgres=# \c

You are now connected to database "postgres" as user "pgxz".

postgres=# select pg_backend_pid();

pg_backend_pid

2426

(1 row)

postgres=# \c template1

You are now connected to database "template1" as user "pgxz".

template1=#

- 连接到外一台服务

postgres=# \c postgres pgxz 172.16.0.47 15432

You are now connected to database "postgres" as user "pgxz" on host "172.16.0.47" at port "15432".

显示和设置该连接当前运行参数

- 显示当前连接的运行参数

postgres=# SELECT CURRENT_USER;

current_user

---

pgxz

(1 row)

postgres=# show search_path ;

search_path

"$user",public

(1 row)

postgres=# show work_mem ;

work_mem

4MB

(1 row)

- 设置当前连接的运行参数

postgres=# set search_path = "$user",public,pg_catalog;

SET

postgres=# set work_mem = '8MB';

SET

- 打开和关闭显示每个sql语句执行的时间

postgres=# \timing on

Timing is on.

postgres=# select count(1) from tbase;

count

10000

(1 row)

Time: 5.139 ms

postgres=# \timing off

Timing is off.

postgres=# select count(1) from tbase;

count

10000

(1 row)

- 打开和关闭显示每个快捷操作符实际运行的sql语句

postgres=# \set ECHO_HIDDEN on

postgres=# \dt

********* QUERY **********

SELECT n.nspname as "Schema",

c.relname as "Name",

CASE c.relkind WHEN 'r' THEN 'table' WHEN 'v' THEN 'view' WHEN 'm' THEN 'materialized view' WHEN 'i' THEN 'index' WHEN 'S' THEN 'sequence' WHEN 's' THEN 'special' WHEN 'f' THEN 'foreign table' END as "Type",

pg_catalog.pg_get_userbyid(c.relowner) as "Owner"

FROM pg_catalog.pg_class c

LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace

```
WHERE c.relkind IN ('r','')
AND n.nspname <> 'pg_catalog'
AND n.nspname <> 'information_schema'
AND n.nspname !~ '\^pg_toast'
AND pg_catalog.pg_table_is_visible(c.oid)
ORDER BY 1,2;
**************************
List of relations
Schema | Name | Type | Owner
--------+--------------+-------+-------
public | t_time_range | table | pgxz
public | tbase | table | pgxz
(2 rows)
postgres=# \set ECHO_HIDDEN off
postgres=# \dt
List of relations
Schema | Name | Type | Owner
--------+--------------+-------+-------
public | t_time_range | table | pgxz
public | tbase | table | pgxz
(2 rows)
```

- 配置输出结果为HTML格式

```
postgres=# \pset format html
```

Output format is html.

postgres=# \d tbase

<table border="1">

<caption>Table \"public.tbase"</caption>

<tr>

<th align="center">Column</th>

<th align="center">Type</th>

<th align="center">Modifiers</th>

</tr>

<tr valign="top">

<td align="left">id</td>

<td align="left">integer</td>

<td align="left">\ </td>

</tr>

<tr valign="top">

<td align="left">mc</td>

<td align="left">text</td>

<td align="left">\ </td>

</tr>

</table>

- 恢复为对齐模式

postgres=# \pset format aligned

Output format is aligned.

```
postgres=# \d tbase
Table "public.tbase"
Column | Type | Modifiers
--------+---------+-----------
id | integer |
mc | text |
```

- 配置行列显示格式

```
postgres=# \x on
Expanded display is on.
postgres=# select * from tbase where id=1;
-[ RECORD 1 ]
id | 1
mc | 1
-[ RECORD 2 ]
id | 1
mc | 2
-[ RECORD 3 ]
id | 1
mc | 2
postgres=# \x off
Expanded display is off.
postgres=# select * from tbase where id=1;
id | mc
```

----+----

1 | 1

1 | 2

1 | 2

(3 rows)

- 显示和配置客户端编码

postgres=# \encoding

UTF8

配置客户端编码为SQL_ASCII

postgres=# \encoding sql_ascii

postgres=# \encoding

SQL_ASCII

退出连接

postgres=# \q

psql执行一个sql命令

- 显示标题

[pgxz@VM_0_3_centos root]$ psql -h 172.16.0.29 -p 15432 -U tbase -d postgres -c "select count(1) from pg_class"

count

317

(1 row)

- 不显示标题

```
[pgxz@VM_0_3_centos root]$ psql -h 172.16.0.29 -p 15432 -U tbase -d
postgres -t -c "select count(1) from pg_class"

317
```

客户端psql的使用

psql简介

psql是一个基于终端的 TDSQL PostgreSQL 版客户端工具，类似Oracle中的命令行工具sqlplus，但比sqlplus更加强大。它让你能交互式地键入查询，把它们发送给 TDSQL PostgreSQL 版，并且查看查询结果。或者，输入可以来自于一个文件或者命令行参数。此外，psql还提供一些元命令和多种类似shell的特性来为编写脚本和自动化多种任务提供便利。psql还支持命令补全，历史命令回找。

psql使用环境配置

Tbase-v2是基于多租户设计，安装后所有机器都不会配置 TDSQL PostgreSQL 版的客户端默认使用环境，所以使用psql需要配置其环境变量。切换到数据库用户 TDSQL PostgreSQL 版，如下

```
export PGXZ_HOME=/usr/local/install/tbase_pgxz

export PATH=$PGXZ_HOME/bin:$PATH

export LD_LIBRARY_PATH=$PGXZ_HOME/lib:${LD_LIBRARY_PATH}

PGUSER=tbase

PGHOST=127.0.0.1

PGDATABASE=postgres

PGPORT=11000

export PGHOST PGUSER PGDATABASE PGPORT
```

这样就能使用psql客户端了，因为测试只初始化了一个实例，所以我们可以将它配置到~/.bashrc 中。

创建 TDSQL PostgreSQL 版默认的分布式使用环境

TDSQL PostgreSQL 版作为分布式数据库系统，使用前我们必需配置数据表默认分布的数据节点（DN），下面演示如何创建一个default node group。

切换到tbase用户：su - tbase

- 连接数据库

> 注意：此处指连接到cn节点（后面没特别说明，所有数据库操作都是连接到cn节点）

```
psql -p port -d database -U user -h
```

host（psql连接参数就是这些，超级管理用户就是数据库用户tbase，默认数据库为postgres，host如果是在本机连接可省略，默认使用本机ip）

示例：

```
[tbase@node1 ~]$ psql -p 11000 -d postgres

psql (PostgreSQL 10.0 TBase V2)

Type "help" for help.
```

- 查询集群节点配置

```
postgres=# select * from pgxc_node;
node_name | node_type | node_port | node_host | nodeis_primary |
nodeis_preferred | node_id | node_cluster_name
-----------+-----------+-----------+-------------+---------------+-----------------+-------------
+------------------
agtm_0 | G | 11004 | 172.16.0.12 | t | f | 475343005 | tbase_cluster
cn001 | C | 11000 | 172.16.0.12 | f | f | -264077367 | tbase_cluster
cn002 | C | 11000 | 172.16.0.5 | f | f | -674870440 | tbase_cluster
dn001 | D | 11002 | 172.16.0.12 | f | f | 2142761564 | tbase_cluster
dn002 | D | 11002 | 172.16.0.5 | f | f | -17499968 | tbase_cluster
dn003 | D | 11006 | 172.16.0.12 | f | f | -1956435056 | tbase_cluster
(6 rows)
```

- 查询当前数据节点（DN）,这些DN节点就是上面初始化集群时建立的

```
postgres=# select * from pgxc_node where node_type='D';

node_name | node_type | node_port | node_host | nodeis_primary |

nodeis_preferred | node_id | node_cluster_name

-----------+-----------+-----------+-------------+---------------+-----------------+-------------
+------------------

dn001 | D | 11002 | 172.16.0.12 | f | f | 2142761564 | tbase_cluster

dn002 | D | 11002 | 172.16.0.5 | f | f | -17499968 | tbase_cluster

dn003 | D | 11006 | 172.16.0.12 | f| f | -1956435056 | tbase_cluster
```

- 创建数据表默认使用的group

  postgres=# create default node group default_group with(dn001, dn002);

- 为default group创建shardmap

配置完成数据表默认使用的DN节点后，我们接下来需要配置记录的分区方案，shardmap就是 TDSQL PostgreSQL 版各个哈希值与DN的对照表，下面演示如何创建一个shardmap给default_group

执行以下指令，即可像单机一样使用 TDSQL PostgreSQL 版集群：

```
postgres=# create sharding group to group default_group;
postgres=# clean sharding;
```

更多group的使用方法

创建扩展group

- 建立group

  postgres=# create node group ext_group with(dn003);

  CREATE NODE GROUP

- 为group创建shard

  postgres=# create extension sharding group to group ext_group;

  CREATE SHARDING GROUP

  postgres=# clean sharding;

  CLEAN SHARDING

  postgres=#

删除group

```
postgres=# drop sharding in group ext_group;
```

DROP SHARDING GROUP

postgres=# drop node group ext_group ;

DROP NODE GROUP

查看集群group

postgres=#select * from pgxc_group;

psql常用命令使用

连接到一个数据库实例

- 使用参数连接

```
[pgxz@VM_0_3_centos root]$ psql -h 172.16.0.29 -p 15432 -U tbase -d postgres
psql (PostgreSQL 10 (TBase 2.01))
Type "help" for help.

postgres=#
```

- 使用conninfo字符串或者一个URI

```
[pgxz@VM_0_3_centos root]$ psql
postgresql://pgxz@172.16.0.29:15432/postgres
psql (PostgreSQL 10 (TBase 2.01))
Type "help" for help.

postgres=#
```

建立一个新连接

- 查看当前连接会话pid

```
postgres=# select pg_backend_pid();
pg_backend_pid
--
2408
(1 row)
```

- 连接到当前库

```
postgres=# \c
You are now connected to database "postgres" as user "tbase".
postgres=# select pg_backend_pid();
 pg_backend_pid
--
```

```
 2426
(1 row)
```

- 连接到其他库

```
postgres=# \c template1
You are now connected to database "template1" as user "tbase".
template1=#
```

- 连接到其他用户

```
postgres=# \c - postgres
You are now connected to database "postgres" as user "postgres".
```

- 连接到其他服务器上的库

```
postgres=# \c postgres tbase 172.16.0.5 11000

You are now connected to database "postgres" as user "tbase" on host
"172.16.0.5" at port "11000".
```

显示和设置该连接当前运行参数

- 显示当前连接用户

```
postgres=# SELECT CURRENT_USER;
current_user
--------------
pgxz
(1 row)
```

- 显示当前连接的schema

```
postgres=# show search_path ;
search_path
----------------
"$user",public
(1 row)
```

- 显示节点参数值

```
postgres=# show work_mem ;
work_mem
----------
4MB
(1 row)
```

- 设置当前连接的运行参数

```
postgres=# set search_path = "$user",public,pg_catalog;
SET
postgres=# set work_mem = '8MB';
SET
```

- 打开和关闭显示每个sql语句执行的时间

```
postgres=# \timing on
Timing is on.
postgres=# select count(1) from tbase;
count
-------
10000
(1 row)
Time: 5.139 ms
postgres=# \timing off
Timing is off.
postgres=# select count(1) from tbase;
count
-------
10000
(1 row)
```

- 配置行列显示格式

```
postgres=# \x on
Expanded display is on.
postgres=# select * from tbase where id=1;
-[ RECORD 1 ]
id | 1
mc | 1
-[ RECORD 2 ]
id | 1
mc | 2
-[ RECORD 3 ]
```

```
id | 1
mc | 2
postgres=# \x off
Expanded display is off.
postgres=# select * from tbase where id=1;
id | mc
----+----
1 | 1
1 | 2
1 | 2
(3 rows)
```

- 显示和配置客户端编码

```
postgres=# \encoding

UTF8
```

配置客户端编码为SQL_ASCII

```
postgres=# \encoding sql_ascii

postgres=# \encoding

SQL_ASCII
```

退出连接

```
postgres=# \q
```

psql执行一个sql命令

- 显示标题

```
[pgxz@VM_0_3_centos root]$ psql -h 172.16.0.29 -p 15432 -U tbase -d
postgres -c "select count(1) from pg_class"
count
-------
317
(1 row)
```

- 不显示标题

```
[pgxz@VM_0_3_centos root]$ psql -h 172.16.0.29 -p 15432 -U tbase -d
postgres -t -c "select count(1) from pg_class"
317
```

psql执行一个sql文件中所有命令

- 在外部执行

```
[pgxz@VM_0_3_centos ~]$ cat /data/tbase/tbase.sql
set search_path = public;
insert into tbase values(1,2);
select count(1) from tbase;
[pgxz@VM_0_3_centos ~]$ psql -h 172.16.0.29 -p 15432 -U tbase -d postgres
-f /data/tbase/tbase.sql
SET
INSERT 0 1
count
-------
10001
(1 row)
```

- 在内部执行

```
[pgxz@VM_0_3_centos ~]$ psql -h 172.16.0.29 -p 15432 -U tbase -d postgres
psql (PostgreSQL 10 (TBase 2.01))
Type "help" for help.
postgres=# \i /data/tbase/tbase.sql
SET
INSERT 0 1
count
-------
10002
```

(1 row)

数据库相关操作

- 显示当前集群中所有数据库

```
postgres=# \l
List of databases
Name | Owner | Encoding | Collate | Ctype | Access privileges
-----------+-------+----------+------------+------------+-------------------
postgres | tbase | UTF8 | en_US.utf8 | en_US.utf8 |
template0| tbase | UTF8 | en_US.utf8 | en_US.utf8| =c/tbase +
| | | | | tbase=CTc/tbase
template1 | tbase | UTF8 | en_US.utf8 | en_US.utf8 | =c/tbase +
| | | | | tbase=CTc/tbase
(3 rows)
```

- \l+显示当前当前集群中所有数据库（包含库大小及注释）

```
postgres=# \l+
List of databases
Name | Owner | Encoding | Collate | Ctype | Access privileges | Size
| Tablespace | Description
-----------+-------+----------+------------+------------+-------------------+-------+------------
+-----------------------------------------
postgres | tbase | UTF8 | en_US.utf8 | en_US.utf8 | | 31 MB |
pg_default | default administrative connection database

template0 | tbase | UTF8 | en_US.utf8 | en_US.utf8 | =c/tbase +| 27 MB
| pg_default | unmodifiable empty database

| | | | | tbase=CTc/tbase | | |

template1 | tbase | UTF8 | en_US.utf8 | en_US.utf8 | =c/tbase +| 27 MB
| pg_default | default template for new databases

| | | | | tbase=CTc/tbase | | |

(3 rows)
```

- 创建一个新库

```
postgres=# create database mydb;
CREATE DATABASE
```

模式相关操作

- \dn显示当前库所有模式

```
postgres=# \dn
List of schemas
Name | Owner
--------+-------
pgxc | tbase
public | tbase
(2 rows)
```

- \dn+显示当前库所有模式（包含注释）

```
postgres=# \dn+
List of schemas
Name | Owner | Access privileges | Description
--------+-------+-------------------+-----------------------
pgxc | tbase | |
public | tbase | tbase=UC/tbase +| standard public schema
| | =UC/tbase |
(2 rows)
```

- 创建一个新模式

```
postgres=# create schema mysche;

CREATE SCHEMA
```

用户相关操作

- \du显示当前集群中所有数据库用户

```
postgres=# \du
List of roles
```

```
Role name | Attributes | Member of
--------------+------------------------------------------------------------+-----------
audit_admin | No inheritance | {}
mls_admin | No inheritance | {}
tbase | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
tbase01_admin | Superuser, Create role, Create DB | {}
```

- \du+显示当前集群中所有数据库用户（包含注释）

```
postgres=# \du+
List of roles
Role name | Attributes | Member of | Description
--------------+------------------------------------------------------------+-----------+-------------
audit_admin | No inheritance | {} |
mls_admin | No inheritance | {} |
tbase | Superuser, Create role, Create DB, Replication, Bypass RLS | {} |
tbase01_admin | Superuser, Create role, Create DB | {} |
```

- 创建一个新的用户

```
postgres=# create role pgxc with login ;
CREATE ROLE
postgres=# create user pgxz ;（使用create user，那么用户自动拥有login权限）
```

- 配置用户密码

```
postgres=# \password pgxc
Enter new password:
Enter it again:
```

表相关操作

- 建立数据表

postgres=# create table tbase(id int,mc text) distribute by shard(id);
CREATE TABLE

```bash

- \d查看表结构，包括使用的触发器

postgres=# \d tbase

Table "public.tbase"
Column | Type | Modifiers
--------+---------+-----------
id | integer|
mc | text |

- \d+查看表结构（包含注释），表类型，分布节点

postgres=# \d+ tbase

Table "public.tbase"
Column | Type | Modifiers | Storage | Stats target | Description
--------+---------+-----------+----------+--------------+-------------
id | integer | | plain | |
mc | text | | extended | |
Has OIDs: no
Distribute By SHARD(id)
Location Nodes: ALL DATANODES

- \dt查看表列表

postgres=# \dt

List of relations
Schema | Name | Type | Owner
--------+--------------+-------+-------
public | t_time_range | table | tbase
public | tbase | table | tbase
(2 rows)
```

- \dt+查看表列表详细信息，包含表大小和注释

这里连接的节点如果是cn的话，表大小为所有dn节点大小之和，否则为只是该节点的表大小

```
postgres=# \dt+
```

```
List of relations
Schema | Name | Type | Owner | Size | Description
--------+--------------+-------+-------+---------+--------------------
public | t_time_range | table | tbase | 0 bytes | 这是一个日期分区表
public | tbase | table | tbase | 576 kB |
(2 rows)
```

- \dt+显示某个模式下的所有表

```
postgres=# \dt+ pgxc.*
```

```
List of relations
Schema | Name | Type | Owner | Size | Description
--------+------------+-------+-------+---------+-------------
pgxc | order_main | table | tbase | 0 bytes |
(1 row)
```

- \dt+表名 显示某个表的详细信息

```
postgres=# \dt+ tbase
```

```
List of relations
Schema | Name | Type | Owner | Size | Description
--------+-------+-------+-------+--------+-------------
public | tbase | table | tbase | 576 kB |
(1 row)
```

- \dt+通配符列出适配的表

```
postgres=# \dt+ t*
```

List of relations
Schema | Name | Type | Owner | Size | Description
--------+--------------+-------+-------+---------+-------------------
public | t_time_range | table | tbase | 0 bytes | 这是一个日期分区表
public | tbase | table | tbase | 576 kB |
(2 rows)

插件管理

- 查看当前库加载了那些插件

postgres=# \dx

List of installed extensions
Name | Version | Schema | Description
-------------------+---------+------------+---
pg_stat_statements | 1.1 | public | track execution statistics of all SQL
statements executed
plpgsql | 1.0 | pg_catalog | PL/pgSQL procedural language
(2 rows)

- 给当前库加载插件

postgres=# create extension pg_stat_statements;

CREATE EXTENSION

- 删除当前库某个插件

postgres=# drop extension pg_stat_statements;

DROP EXTENSION

sql帮助命令

postgres=# \h

postgres=# \h create table;
\h 可以查看一些sql命令的使用方法。
```
```

数据库使用（V5.21）
连接TDSQL PG
使用psql连接TDSQL PG

本节主要介绍如何使用 psql 连接 TDSQL PG。

前置条件

通过 psql 连接 TDSQL PG：

- 通过浏览器访问一体化管控平台，在左侧导航栏选择实例管理，点击需要连接的实例ID， 选择节点管理页面。
- 如果是集中式实例，点击数据节点；如果是分布式实例，点击计算节点。



- 将鼠标移动到节点IP地址后的　　号上，弹出的提示框中显示了该节点配置信息，支持快捷复制。
  - bin 目录：数据库的可执行程序所在的目录
  - data 目录：数据库的数据文件目录
  - 环境变量配置：登录此节点，psql连接数据库前需要配置的环境变量。
  - psql 连接：psql连接此数据库节点的参数信息
- 根据节点IP地址，登录到相应的服务器，执行上一步环境变量配置项中的命令。

连接方式

psql可以通过四种方式连接实例，分别是：使用参数连接、使用conninfo字符串或者URI连接、配置SSL连接、配置环境变量后快捷连接。

使用参数连接

下面列出了参数连接需要配置的内容，建议直接查看一体化管控平台，复制节点的psql 连接信息：

- -h 数据库连接地址，分布式实例通常是计算节点的IP
- -p 数据库服务的端口号
- -U 数据库用户名
- -d 需要访问的数据库名称

  命令格式如下：

```
psql -h <数据库连接地址> -p <端口号> -U <用户名> -d <数据库名称>
```

示例

以 tbase 身份登录 TDSQL 数据库：

```
psql -h 10.211.55.39 -p 11345 -U tbase -d postgres
```

使用conninfo字符串或者URI连接

psql通过conninfo字符串或者URI连接数据库，同样需要在conninfo或URI中指定数据库连接地址等参数。

```
conninfo: psql "host=<数据库连接地址> port=<端口号> dbname=<数据库名> user=用户名"
```

```
URI: postgresql://[用户名[:密码]@][数据库连接地址][:端口号][/数据库名][?param1=value1&...]
```

示例

示例1：通过conninfo连接数据库：

```
psql "host=10.211.55.39 port=11345 dbname=postgres user=tbase"
```

示例2：通过URI连接数据库：

```
psql postgresql://tbase@10.211.55.39:11345/postgres
```

配置SSL连接

TDSQL支持通过配置单/双向认证，开启SSL连接加密后，通过psql连接数据库。

配置单项认证连接

单项认证仅需要服务端持有证书，客户端需要验证服务端证书的有效性，确认后开启通信。

创建服务端证书：

示例采用自签名证书进行演示，实际生产环境中应该使用权威证书机构签署的证书。

> 注意：
>
> 1. 请登录一体化管控平台，在实例的节点管理页面，查看该节点配置的data 目录，确认data 目录的完整路径。
> 2. root.crt、server.crt和server.key需要拷贝到节点data 目录下，并通过chown命令，将文件的所有者修改成与postgresql.conf一致，保证数据库可执行程序有权限读取。
> 3. 配置完成后需要通过pg_ctl reload命令让数据库加载新配置的参数，也可以重启数据库使得参数生效。

1. 通过下述命令生成证书

```
# 1. 创建证书签名请求和密钥文件
openssl req -new -nodes -text -out root.csr -keyout root.key -subj "/CN=root"
# 2. 创建根证书颁发机构
openssl x509 -req -in root.csr -text -days 3650 -extfile /etc/pki/tls/openssl.cnf  -extensions v3_ca -signkey root.key -out root.crt
# 3. 创建由根证书颁发机构签名的服务器证书
openssl req -new -nodes -text -out server.csr -keyout server.key -subj "/CN=postgres"
chmod og-rwx server.key # 确认只有文件拥有者有读写权限，移除其他用户和组用户的权限
openssl x509 -req -in server.csr -text -days 365 -CA root.crt -CAkey root.key -CAcreateserial -out server.crt
```

2. 将上述命令生成的root.crt、server.crt和server.key拷贝到节点数据目录下，并通过chown命令，将文件的所有者修改成与postgresql.conf文件相同。
3. 修改该节点data 目录下的postgresql.conf文件，配置ssl参数。

```
ssl = on
ssl_cert_file = 'server.crt'
ssl_key_file = 'server.key'
```

4. 修改该节点data 目录下的pg_hba.conf，配置访问控制，通过hostssl指定SSL连接。

```
hostssl     all    all    0.0.0.0/0    md5
```

5. 通过pg_ctl reload命令，让数据库加载新配置的参数；如果pg_ctl命令找不到，请参照前置条件进行环境变量配置；请将下述的$data_dir替换为实际的数据目录。

```
pg_ctl reload -D $data_dir
```

6. psql连接节点，通过提示信息 SSL connection 确认该连接为SSL连接。

```
psql "host=10.211.55.39 port=11345 dbname=postgres user=tbase"

SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
Type "help" for help.

postgres=#
```

配置双向认证连接

配置双向认证连接需要在单向认证的基础上，配置客户端证书，服务端和客户端都需要确认彼此证书的有效性，确认后开启通信。
请参考“配置单项认证连接”中的示例，创建服务端证书。

> 注意：
>
> 请登录一体化管控平台，在实例的“节点管理”页面，查看该节点配置的“data 目录”，确认数据目录的完整路径。

1. 修改该节点data 目录下的postgresql.conf文件，配置ssl参数。

```
ssl = on
ssl_cert_file = 'server.crt'
ssl_key_file = 'server.key'
ssl_ca_file = 'root.crt'
```

2. 修改该节点data 目录下的pg_hba.conf，配置访问控制，将认证方法设置为cert，SSL客户端证书认证。

```
hostssl     all    all    0.0.0.0/0    cert
```

3. 通过pg_ctl reload命令，让数据库加载新配置的参数；如果pg_ctl命令找不到，请参照前置条件进行“环境变量配置”；请将下述的$data_dir替换为实际的数据目录。

```
pg_ctl reload -D $data_dir
```

4. 创建客户端证书，示例采用自签名证书进行演示，实际生产环境中应该使用权威证书机构签署的证书.

> 注意：
>
> 请将下述的tbase用户名替换为实际连接此数据库节点的用户名。

```
openssl req -new -nodes -text -out postgresql.csr -keyout postgresql.key  -subj "/CN=tbase"
chmod og-rwx postgresql.key
openssl x509 -req -in postgresql.csr -text -days 365 -CA root.crt -CAkey root.key -CAcreateserial -out p
ostgresql.crt
```

5. 使用psql连接
   示例1：通过conninfo字符串连接
   在客户端机器创建 ~/.postgresql 目录，并将上方创建的root.crt，postgresql.crt，postgresql.key复制到 ~/.postgresql 目录。

```
psql "host=10.211.55.39 port=11345 dbname=postgres user=tbase"
```

```
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
Type "help" for help.

postgres=#
```

示例2：通过conninfo字符串连接，分别通过sslcert、sslkey和sslrootcert参数设置postgresql.crt、postgresql.key和root.crt的文件路径。

```
psql "host=10.211.55.39 port=11345 dbname=postgres user=tbase sslmode=require sslcert=/PATH/TO/postgresql.crt sslkey=/PATH/TO/postgresql.key sslrootcert=/PATH/TO/root.crt"

psql (PostgreSQL 10.0 @ xxx 2024-02-21 10:27:15)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
Type "help" for help.

postgres=#
```

示例3: 分别通过环境变量PGSSLROOTCERT、PGSSLCERT、PGSSLKEY设置root.crt、postgresql.crt和postgresql.key的文件路径。

```
export PGSSLROOTCERT=/PATH/TO/root.crt
export PGSSLCERT=/PATH/TO/postgresql.crt
export PGSSLKEY=/PATH/TO/postgresql.key

psql "host=10.211.55.39 port=11345 dbname=postgres user=tbase sslmode=require"

SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
Type "help" for help.

postgres=#
```

配置环境变量后快捷连接

TDSQL PG 支持通过在环境变量中指定主机名称，端口地址，数据库以及数据库用户后通过命令psql即可快速连接。相关环境变量分别为 PGHOST ， PGPORT ， PGDATABASE ， PGUSER 。

示例

```
export PGUSER=tbase
export PGHOST=10.211.55.39
export PGDATABASE=postgres
export PGPORT=11345
```

psql

psql常用命令

命令	功能	命令	功能
\password	设置密码	\dt[+]	显示表格列表或表格的详细信息
\q \quit	退出psql程序	\dv[+]	显示视图列表或视图的详细信息
\conninfo	输出有关当前数据库连接的信息	\df[+]	显示函数/过程列表或函数/过程的详细信息
\timing	显示SQL语句打开和关闭的时间	\ds[+]	显示序列列表或序列的详细信息
\i	从指定文件中读取输入，并执行	\do[+]	显示操作符列表或操作符的详细信息
\o	指定结果输出文件	\du[+]	显示用户列表或用户的详细信息
\l	列出可用的数据库	\dn[+]	显示模式列表或模式的详细信息
\copy	执行一次前端拷贝	\di[+]	显示索引列表或索引的详细信息
\x	切换扩展表格格式化模式	\dx[+]	显示插件列表或插件的详细信息

更多信息

如果在通过 psql 客户端连接 TDSQL PG的过程中遇到本文档未涉及的问题，您可以查阅PostgreSQL 客户端官方的使用文档。

使用数据库连接池

本节主要介绍如何使用数据连接池连接 TDSQL PG 数据库。

简介

创建和关闭连接是一个比较昂贵的操作，为了避免频繁地创建和销毁 JDBC 连接，通常应用会选择使用数据库连接池来复用连接。Java 中常用的连接池很多 (HikariCP、c3p0、druid等)，应用可以根据业务特点选择使用合适的连接池。本文档以 HikariCP 为例，介绍如何使用数据连接池连接 TDSQL PG 数据库。

连接数据库

准备 TDSQL PG JDBC驱动包

内核Oracle模式下建议使用自研驱动，请联系云平台技术支持工程师获取软件包，可根据开发环境选择合适的驱动包：

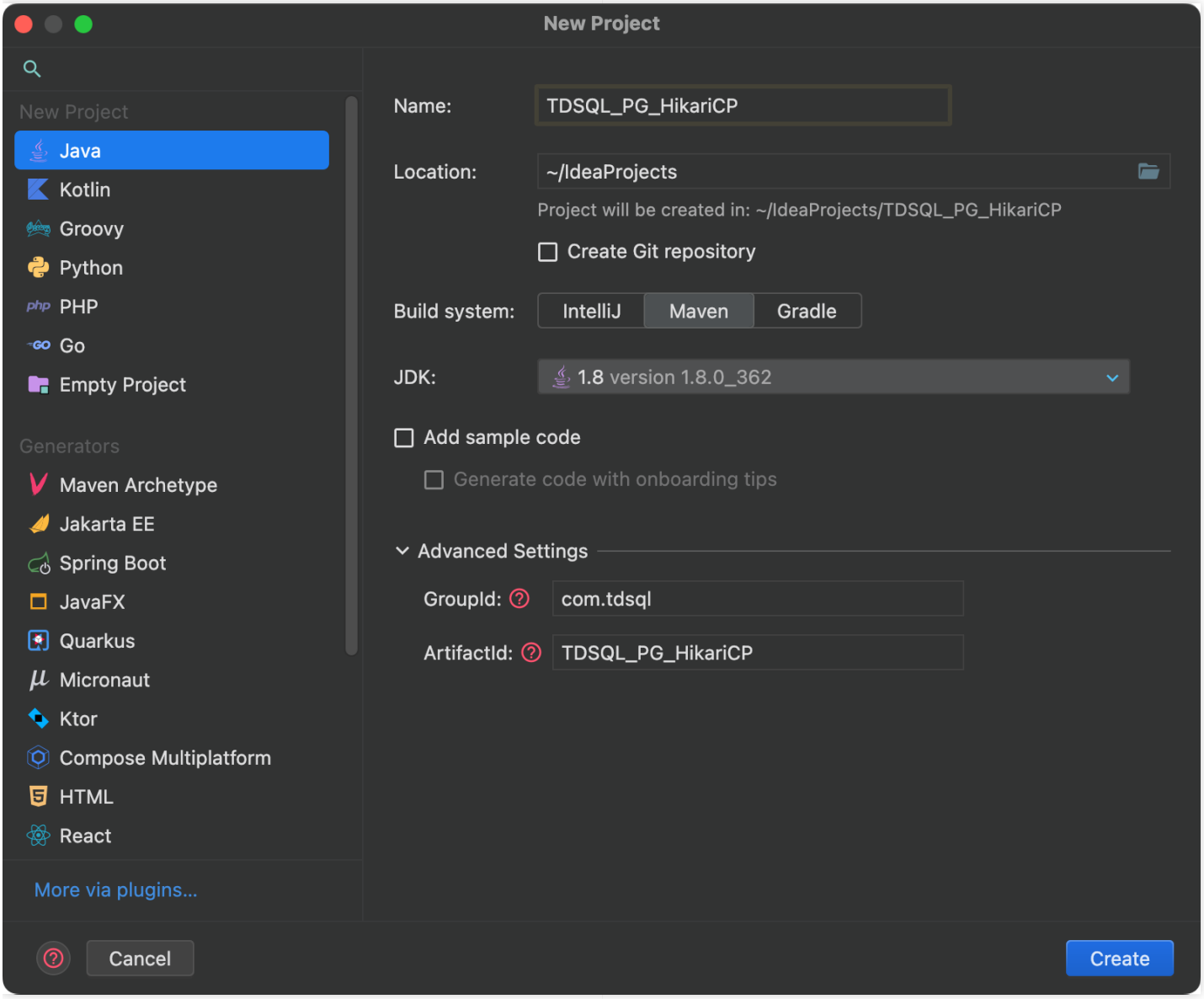
- postgresql-connector-java8-x.x.x.jar 对应 JAVA 8 及以上版本的JDK
- postgresql-connector-java7-x.x.x.jar 对应 JAVA 7版本的JDK
- postgresql-connector-java6-x.x.x.jar 对应 JAVA 6版本的JDK

从管理员获取数据库相关信息：包括数据库IP、端口、实例名、用户、密码等。驱动名和连接URL如下：

- 驱动名 org.postgresql.Driver
- 连接URL（Oracle模式下需打开oracle_compile参数） jdbc:postgresql://host:port/database?oracle_compile=true
- 更多JDBC相关配置请参考“使用JDBC连接TDSQL PG”章节

创建 Maven 项目

创建 Maven 项目，如下：



> 注意
>
> 本文档示例项目使用的 IDE 是 IntelliJ IDEA 2024.3.5 (Ultimate Edition)，也可根据喜好选择合适的 IDE。

配置 pom.xml 文件，导入项目依赖

配置文件 pom.xml 如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.tdsql</groupId>
```

```
    <artifactId>TDSQL_PG_HikariCP</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
      <maven.compiler.source>8</maven.compiler.source>
      <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <dependencies>
      <!-- HikariCP -->
      <dependency>
        <groupId>com.zaxxer</groupId>
        <artifactId>HikariCP</artifactId>
        <version>4.0.3</version>
      </dependency>
      <!-- TDSQL-PG JDBC Driver -->
      <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>1.1.4</version>
        <scope>system</scope>
        <systemPath>${project.basedir}/lib/postgresql-connector-java8-1.1.4.jar</systemPath>
      </dependency>
    </dependencies>
  </project>
```

通过 <dependencies> 定义项目所依赖的组件，主要包括：

1. HikariCP 依赖包
2. 下载 TDSQL-PG JDBC 驱动包，并在 TDSQL_PG_HikariCP 目录下新建 lib 目录，将驱动包拷贝到 lib 目录下。

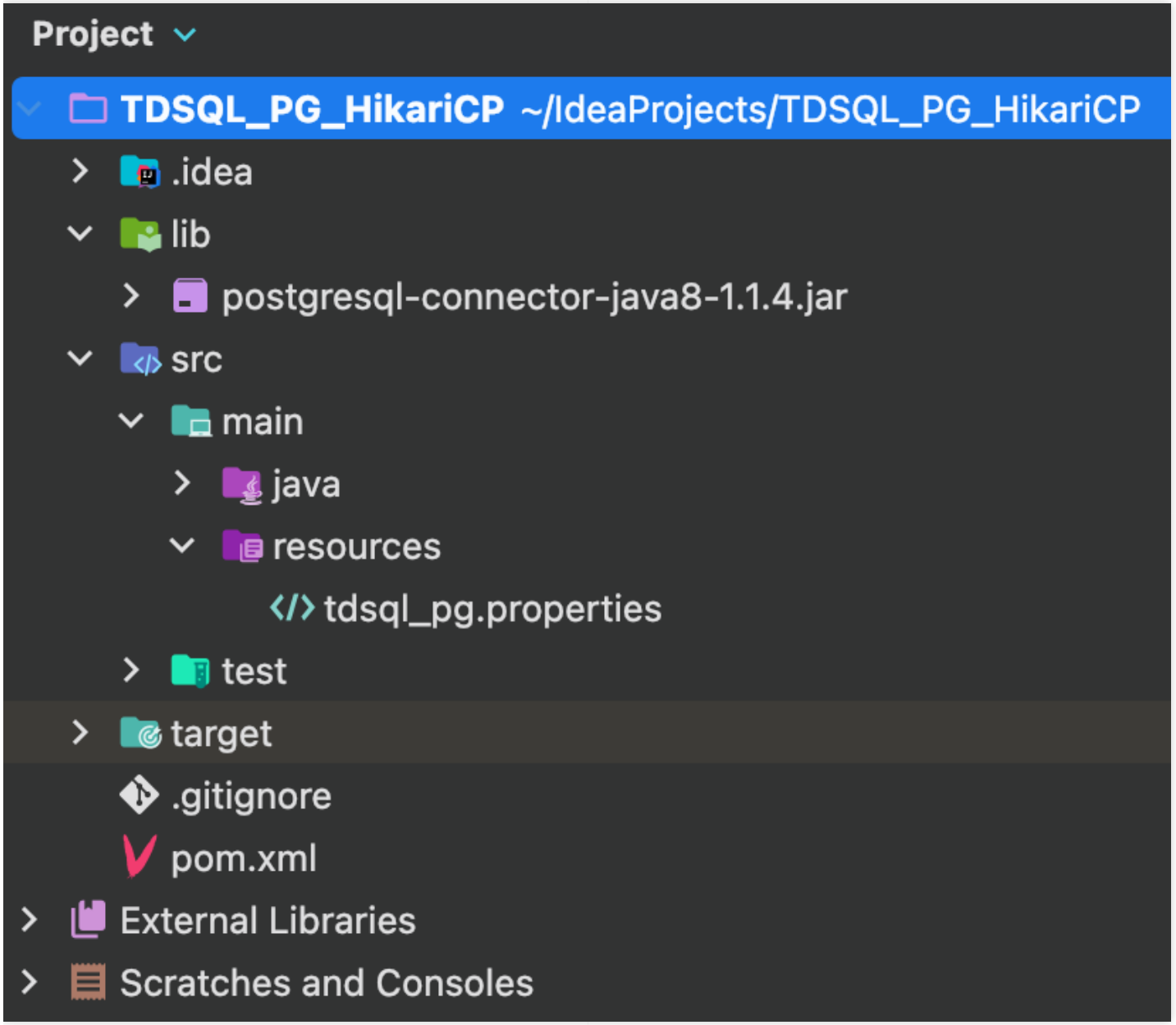
> 注意
>
> 本文所有示例需在启用Oracle兼容内核的情况下执行

配置 HikariCP

在 TDSQL_PG_HikariCP/src/main/resources 目录下新建 tdsql_pg.properties 文件，用于配置 JDBC 连接信息，如下：

```
# 配置JDBC连接信息：host、port、database、user、password 需要替换为数据库管理员提供的数据库连接信息。
jdbcUrl=jdbc:postgresql://host:port/database?oracle_compile=true
username=user
password=password
```

至此，已完成 HikariCP 项目的基本配置，完整的项目目录结构如下图所示：



开发示例

测试

- 开发使用的数据表为 STUDENT 表

```
CREATE TABLE STUDENT(ID INT PRIMARY KEY, NAME VARCHAR(50));
```

- 在 TDSQL_PG_HikariCP/src/main/java/ 下创建 Package: com.tdsql ，
- 在 TDSQL_PG_HikariCP/src/main/java/com/tdsql 下添加测试类 Main ，展示了基本的增删查改功能，如下：

```
package com.tdsql;

import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import javax.sql.DataSource;

public class Main {
    public static void main(String args[]) {
        try {
            // Load HikariCP configuration
            HikariConfig config = new HikariConfig("/tdsql_pg.properties");
            DataSource ds = new HikariDataSource(config);
            // Get connection
            Connection conn = ds.getConnection();
            Statement stmt = conn.createStatement();

            // Insert data
            System.out.println("------Insert data------");
            stmt.execute("insert into STUDENT values (1,'Alex'),(2,'Bob'),(3,'John')");
            System.out.println("Insert 3 records");

            // Query data
            System.out.println("------Query data------");
            ResultSet rs1 = stmt.executeQuery("select * from STUDENT order by id");
            while (rs1.next()) {
                System.out.println(rs1.getInt(1) + " " + rs1.getString(2));
            }

            // Update data
            System.out.println("------Update data------");
            stmt.execute("update STUDENT set name = 'Alice' where id = 1");
            System.out.println("Update the data where id is 1");

            // Query data
            System.out.println("------Query data------");
            ResultSet rs2 = stmt.executeQuery("select * from STUDENT order by id");
            while (rs2.next()) {
                System.out.println(rs2.getInt(1) + " " + rs2.getString(2));
            }

            // Delete data
            System.out.println("------Delete data------");
```

```
            stmt.execute("delete from STUDENT where id = 2");
            System.out.println("Delete the data where id is 2");

            // Query data
            System.out.println("------Query data------");
            ResultSet rs3 = stmt.executeQuery("select * from STUDENT order by id");
            while (rs3.next()) {
                System.out.println(rs3.getInt(1) + " " + rs3.getString(2));
            }

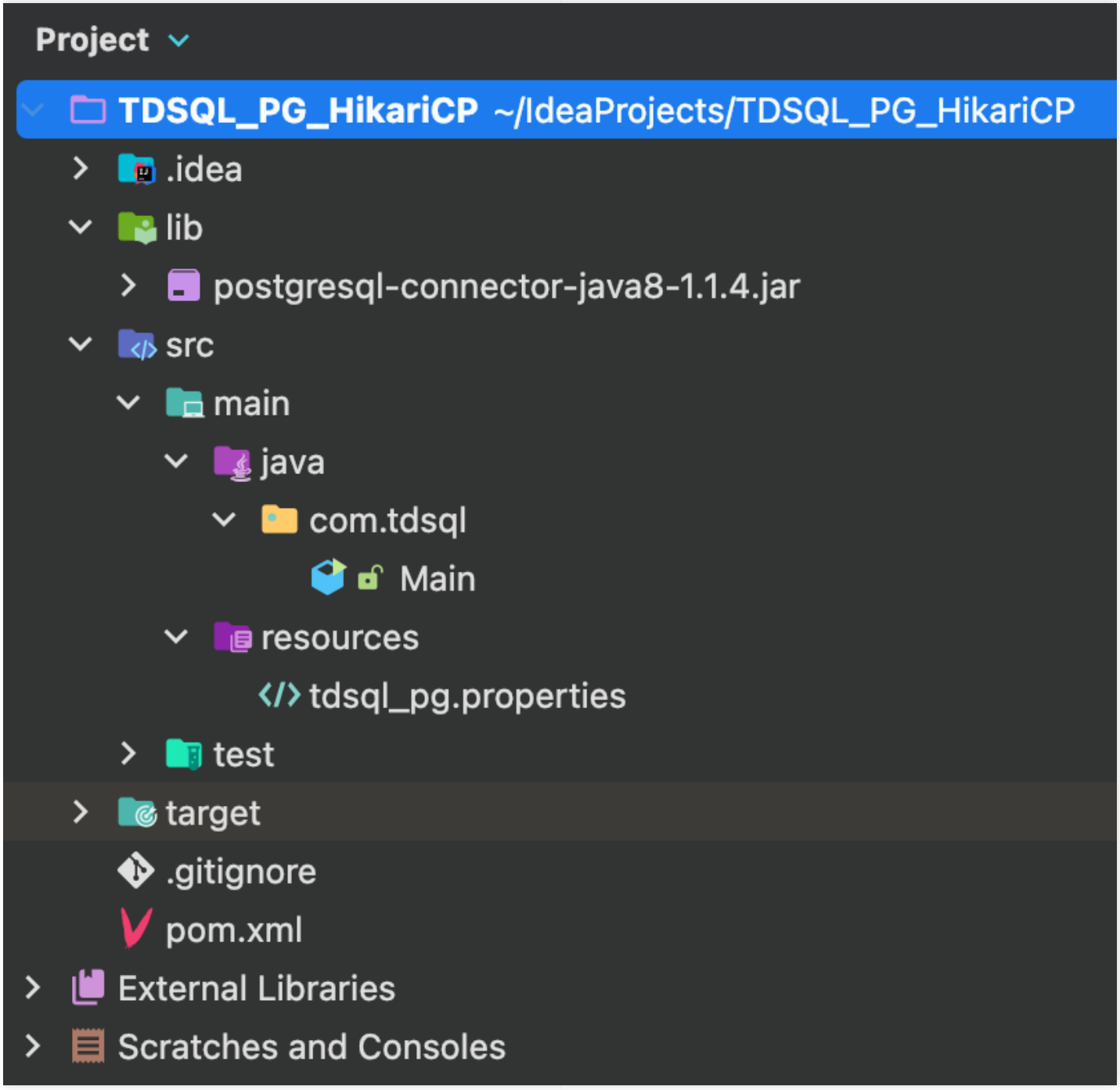
            // Close the object
            rs1.close();
            rs2.close();
            rs3.close();
            stmt.close();
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- 运行结果如下：

```
------Insert data------
Insert 3 records
------Query data------
1 Alex
2 Bob
3 John
------Update data------
Update the data where id is 1
------Query data------
1 Alice
2 Bob
3 John
------Delete data------
Delete the data where id is 2
------Query data------
1 Alice
3 John
```

附：项目结构

至此，完成了 HikariCP 项目的配置和开发示例，项目的整体结构如下图所示：


Project
TDSQL_PG_HikariCP ~/IdeaProjects/TDSQL_PG_HikariCP
.idea
lib
postgresql-connector-java8-1.1.4.jar
src
main
java
com.tdsql
Main
resources
tdsql_pg.properties
test
target
.gitignore
pom.xml
External Libraries
Scratches and Consoles

SQL基础操作
PG模式

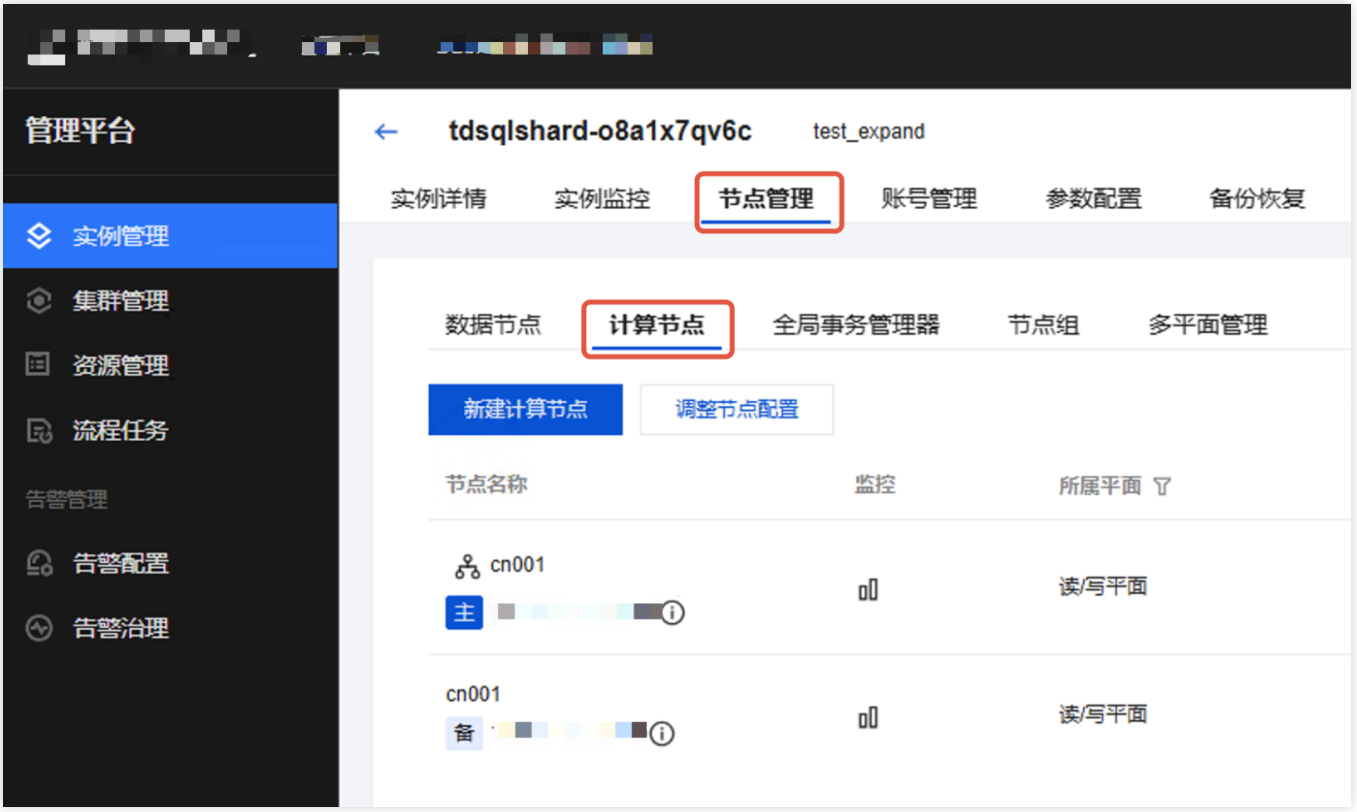
SQL基础操作 PG模式

本节主要介绍 TDSQL PG 在PG模式下的SQL基本操作。

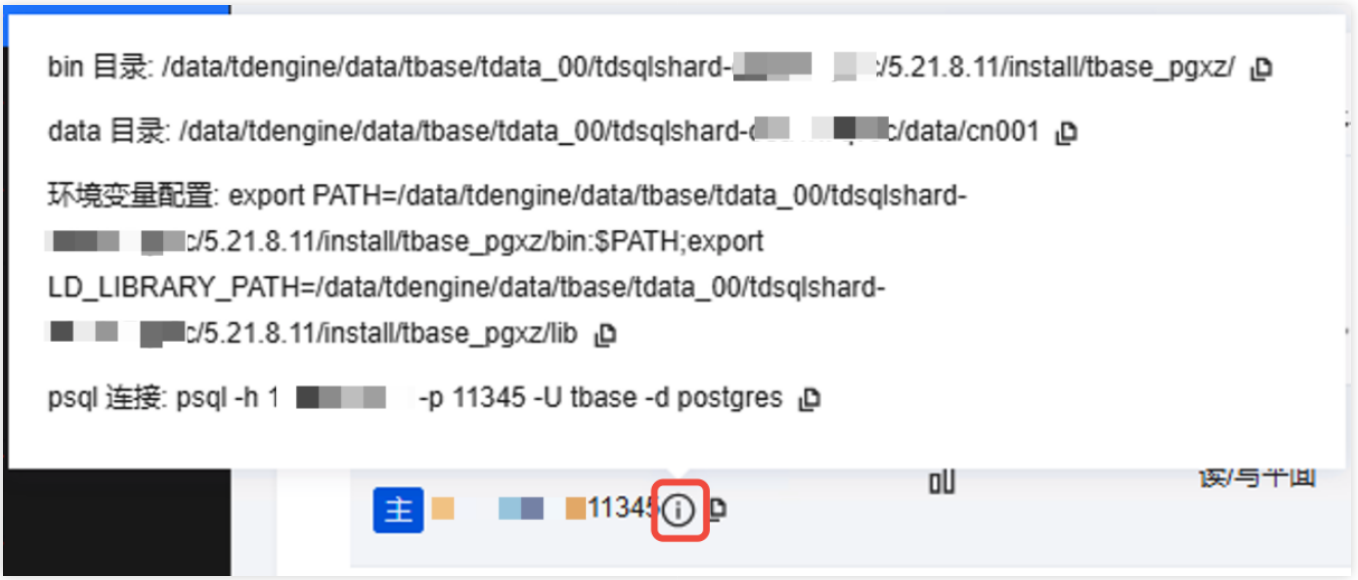
连接数据库

获取数据库连接配置信息，登录数据库系统：

1. 登录TDSQL管理平台在左侧选择 实例管理 并点击需要登录的实例，进入 节点管理 页（分布式下选择 计算节点 页，集中式选择 数据节点 页）。



2. 选择需要登录的节点IP，鼠标悬停在其右侧 图标获得环境配置信息。



3. 登录数据库节点服务器，切换到tbase用户并使用上述的export命令设置环境变量。

```
su - tbase
export PATH=/data1/tdengine/data/tbase/tdata_00/tdsqlshard-o62xkfw1a0/5.21.8.11/install/tbase_pgxz/bin:$PATH;
export LD_LIBRARY_PATH=/data1/tdengine/data/tbase/tdata_00/tdsqlshard-o62xkfw1a0/5.21.8.11/install/tbase_pgxz/lib
```

4. 在本机使用psql命令连接数据库（tbase用户一般只用于本机登录，可以创建普通用户用于业务开发）。

> 注意：
>
> 是连接到cn节点(后面没特别说明，所有数据库操作都是连接到cn节点)。

```
psql -h database_host_ip -p 11345 -U tbase -d postgres
```

创建用户

TDSQL PG可以用CREATE USER创建一个用户，需要使用超级用户或者具有CREATEROLE权限的用户（比如：tbase）。

- 示例：使用 create user 创建用户并指定密码。

```
postgres=# create user mgr password 'mypassword';
CREATE ROLE
```

创建完成后，使用 \du 展示用户信息。

```
postgres=# \du mgr
          List of roles
```

```
 Role name | Attributes | Member of
-----------+------------+-----------
 mgr       |            | {}
```

可以使用ALTER USER重新设置用户密码。

```
postgres=# alter user mgr password 'newpassword';
ALTER ROLE
```

创建数据库

要创建一个数据库，需要使用超级用户或者具有CREATEDB权限的用户。

- 示例：使用 create database 默认参数创建数据库。

```
postgres=# create database tdsql;
CREATE DATABASE
```

创建完成后，可以通过 \l 命令查看当前实例中的所有数据库。

```
postgres=# \l
                                  List of databases
    Name      | Owner | Encoding | Sqlmode  |  Collate   |   Ctype    | Access privileges
--------------+-------+----------+----------+------------+------------+-------------------
 ora          | tbase | UTF8     | oracle   | zh_CN.utf8 | zh_CN.utf8 |
 postgres     | tbase | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 |
 tdsql        | tbase | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 |
 template0    | tbase | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 | =c/tbase        +
              |       |          |          |            |            | tbase=CTc/tbase
 template0_ora | tbase | UTF8     | oracle   | zh_CN.utf8 | zh_CN.utf8 | =c/tbase        +
              |       |          |          |            |            | tbase=CTc/tbase
 template1    | tbase | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 | =c/tbase        +
              |       |          |          |            |            | tbase=CTc/tbase
 template1_ora | tbase | UTF8     | oracle   | zh_CN.utf8 | zh_CN.utf8 | =c/tbase        +
              |       |          |          |            |            | tbase=CTc/tbase
(7 rows)
```

在刚创建的 tdsql 中进行后续操作，使用 \c 命令切换数据库。

```
postgres=# \c tdsql;
You are now connected to database "tdsql" as user "tbase".
tdsql=#
```

创建模式

- 标准语句。

```
tdsql=# create schema tbase_pg;
CREATE SCHEMA
```

- 扩展语法，不存在时才创建。

```
tdsql=# create schema if not exists tbase_pg;
NOTICE:  (42P06) schema "tbase_pg" already exists, skipping
CREATE SCHEMA
```

表操作

创建表

使用命令 create table 创建表

1. 不用指定shard key建表，系统默认使用第一个字段做为表的shard key

```
create table shared_col_table(id serial not null,nickname text);
```

2. 指定shard key建表

```
create table t_appoint_col(id serial not null,nickname text) distribute by shard(nickname);
```

> 注意：
>
> 分布键选择原则
>
> - v5.21.x支持多列分布键
> - 如果有主键，则选择主键做分布键
> - 如果主键是复合字段组合，则选择过滤性好的字段做分布键
> - 没有主键的可以使用UUID来做分布键
> - 让数据尽可能的分布得足够散

3. 表不存在时才创建

```
create table IF NOT EXISTS t(id int,mc text);
```

查看表

使用 \d 命令查看表。

```
tdsql=# \d t_appoint_col;
                        Table "public.t_appoint_col"
  Column  |  Type   | Collation | Nullable |                  Default
----------+---------+-----------+----------+-------------------------------------------
 id       | integer |           | not null | nextval('t_appoint_col_id_seq'::regclass)
 nickname | text    |           |          |
```

修改表

使用命令 alter table 修改表

示例：

1. 修改表名

```
alter table t rename to tbase;
```

2. 增加表字段

```
alter table tbase add column age integer;
```

3. 修改字段类型

```
alter table tbase alter column age type float8;
```

4. 修改表所属模式

```
alter table tbase set schema public;
```

删除表

使用命令 drop table 删除表

示例：当表存在时删除

```
drop table if exists tbase;
```

索引操作

创建索引

使用命令 create index 创建索引

示例：

1. 创建唯一索引

```
create table t_first_col_share(id serial not null,nickname text);
create unique index t_first_col_share_id_uidx on t_first_col_share using btree(id);
```

> 注意：
>
> 非shard key字段不能建立唯一索引

2. 创建多字段索引

```
create table t_mul_idx (f1 int,f2 int,f3 int,f4 int);
create index t_mul_idx_idx on t_mul_idx(f1,f2,f3);
```

查看索引

1. 使用 \d 命令查看索引。

```
tdsql=# \d t_first_col_share;
                        Table "public.t_first_col_share"
 Column  |  Type   | Collation | Nullable |                  Default
---------+---------+-----------+----------+--------------------------------------------
 id      | integer |           | not null | nextval('t_first_col_share_id_seq'::regclass)
 nickname | text   |           |          |
Indexes:
    "t_first_col_share_id_uidx" UNIQUE, btree (id) WITH (checksum='on')
```

2. 使用 \di 命令查看索引

```
tdsql=# \di
                      List of relations
 Schema |          Name           | Type  | Owner |       Table
--------+-------------------------+-------+-------+-------------------
 public | t_first_col_share_id_uidx | index | tbase | t_first_col_share
 public | t_mul_idx_idx           | index | tbase | t_mul_idx
(2 rows)
```

3. 使用 pg_indexes 视图列出索引

```
SELECT
    indexname,
    indexdef
FROM
    pg_indexes
WHERE
```

```
    tablename = 't_first_col_share';
```

示例输出：

```
        indexname        |                                        indexdef
--------------------------
+-----------------------------------------------------------------------------------------------------------------------
 t_first_col_share_id_uidx | CREATE UNIQUE INDEX t_first_col_share_id_uidx ON public.t_first_col_share USING
btree (id) WITH (checksum='on')
(1 row)
```

删除索引

使用命令 DROP INDEX 删除索引

```
drop index t_first_col_share_id_uidx;
```

插入数据

insert用于向一张表中插入数据，期望插入的数据可以是一条，多条或者是一个select查询的结果集。

使用 INSERT INTO... 命令插入数据

- 创建表 tbase 并插入一行数据：

```
tdsql=# create table tbase(id serial not null,nickname text) distribute by shard(id);
CREATE TABLE
tdsql=# insert into tbase(id,nickname) values(1,'hello TDSQL PG');
INSERT 0 1
```

- 向表 tbase 插入多行数据：

```
tdsql=# insert into tbase(id,nickname) values(2,'TDSQL PG'),(3,null),(4,'TDSQL PG good');
COPY 3
tdsql=# select * from tbase;
 id |    nickname
----+----------------
  1 | hello TDSQL PG
  2 | TDSQL PG
  3 |
  4 | TDSQL PG good
(4 rows)
```

删除数据

delete用于删除表数据，可以全部删除或者部分删除。

- 示例，带条件删除：

```
tdsql=# select * from tbase;
 id |    nickname
----+----------------
  1 | hello TDSQL PG
  2 | TDSQL PG
  3 |
  4 | TDSQL PG good
(4 rows)

tdsql=# delete from tbase where id=4;
DELETE 1
```

- 示例，null条件的表达方式

```
tdsql=# delete from tbase where nickname is null;
DELETE 1
tdsql=# select * from tbase;
 id |    nickname
----+----------------
  1 | hello TDSQL PG
  2 | TDSQL PG
(2 rows)
```

更新数据

update用于更新表数据，可以更新一个字段或者多个字段。

```
tdsql=# update tbase set nickname ='Hello TDSQL PG' where id=1;
UPDATE 1
```

查询数据

select用于从一张表或视图中获取数据。
使用 select...from... 命令查询数据

- 创建 t_grouping 表并插入内容

```
create table t_grouping(id int,dep varchar(20),product varchar(20),num int);
insert into t_grouping values(1,'dept no1','phone',90);
insert into t_grouping values(2,'dept no1','computer',80);
insert into t_grouping values(3,'dept no1','phone',70);
insert into t_grouping values(4,'dept no2','computer',60);
insert into t_grouping values(5,'dept no2','phone',50);
insert into t_grouping values(6,'dept no2','computer',60);
insert into t_grouping values(7,'dept no3','phone',70);
insert into t_grouping values(8,'dept no3','computer',80);
insert into t_grouping values(9,'dept no3','phone',90);
```

- 查询 t_grouping 表所有内容

```
tdsql=# select * from t_grouping;
 id |   dep    | product  | num
----+----------+----------+-----
  1 | dept no1 | phone    |  90
  2 | dept no1 | computer |  80
  5 | dept no2 | phone    |  50
  6 | dept no2 | computer |  60
  8 | dept no3 | computer |  80
  9 | dept no3 | phone    |  90
  3 | dept no1 | phone    |  70
  4 | dept no2 | computer |  60
  7 | dept no3 | phone    |  70
(9 rows)
```

- 按dep，product汇总，排序展示

```
tdsql=# select dep,product,sum(num) from t_grouping group by dep,product order by dep,product;
   dep    | product  | sum
----------+----------+-----
 dept no1 | computer |  80
 dept no1 | phone    | 160
 dept no2 | computer | 120
 dept no2 | phone    |  50
 dept no3 | computer |  80
 dept no3 | phone    | 160
(6 rows)
```

- 带条件查询

```
tdsql=# select * from t_grouping where dep='dept no2';
 id |   dep    | product  | num
----+----------+----------+-----
  5 | dept no2 | phone    |  50
```

```
  6 | dept no2 | computer |  60
  4 | dept no2 | computer |  60
(3 rows)
```

提交事务

事务提交用于保存对数据库的更改命令。

命令如下：

COMMIT; 或者 COMMIT TRANSACTION;

例：

```
insert into tbase(id,nickname) values(5,'hello new TDSQL PG');
BEGIN;
delete from tbase where id=5;
COMMIT;
```

> 注意：
>
> 在COMMIT执行之前，tbase表中id为5的记录会被另一会话查询到。

回滚事务

事务回滚用于撤销尚未保存到数据库的命令。

命令语法如下：

ROLLBACK; 或者 ROLLBACK to savepoint_name;

例1：

```
BEGIN;
delete from tbase where id in (1,2);
select * from tbase;
ROLLBACK;
select * from tbase;
```

> 注意：
>
> 在ROLLBACK执行之后，事务对数据的删除不会保存到数据库中。

例2:

```
BEGIN;
delete from tbase where id =1;
select * from tbase;
savepoint A;
delete from tbase where id =2;
```

```
select * from tbase;
ROLLBACK to A;
COMMIT;
```

> 注意：
>
> 在ROLLBACK to A;执行之前同一进程查询不到id为1和2的数据。回滚到保存点之后，id为2数据可查询到。
> ROLLBACK to savepoint_name;不会结束事务，需使用COMMIT;提交事务。

Oracle模式

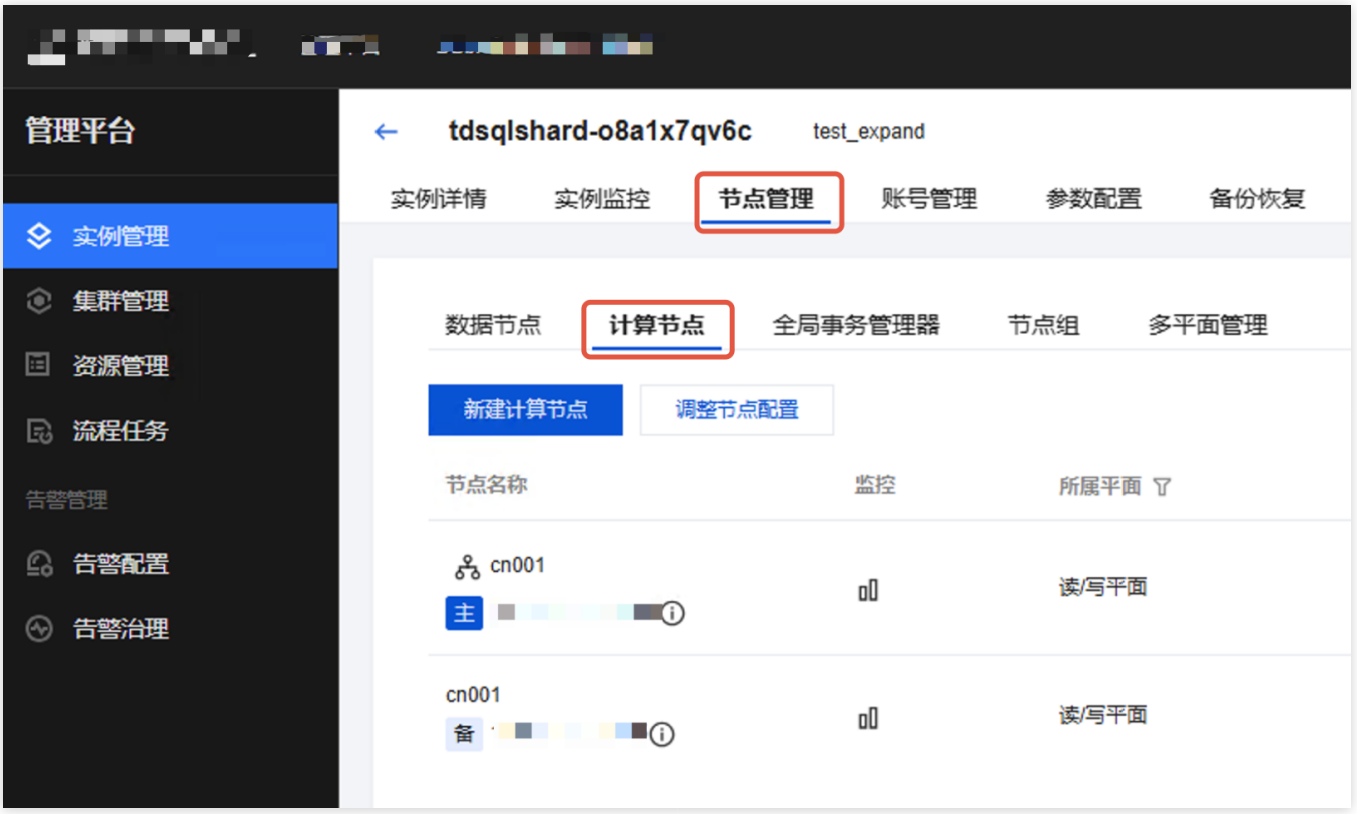
SQL基础操作 Oracle模式

本节主要介绍 TDSQL PG 在Oracle模式下的SQL基本操作。

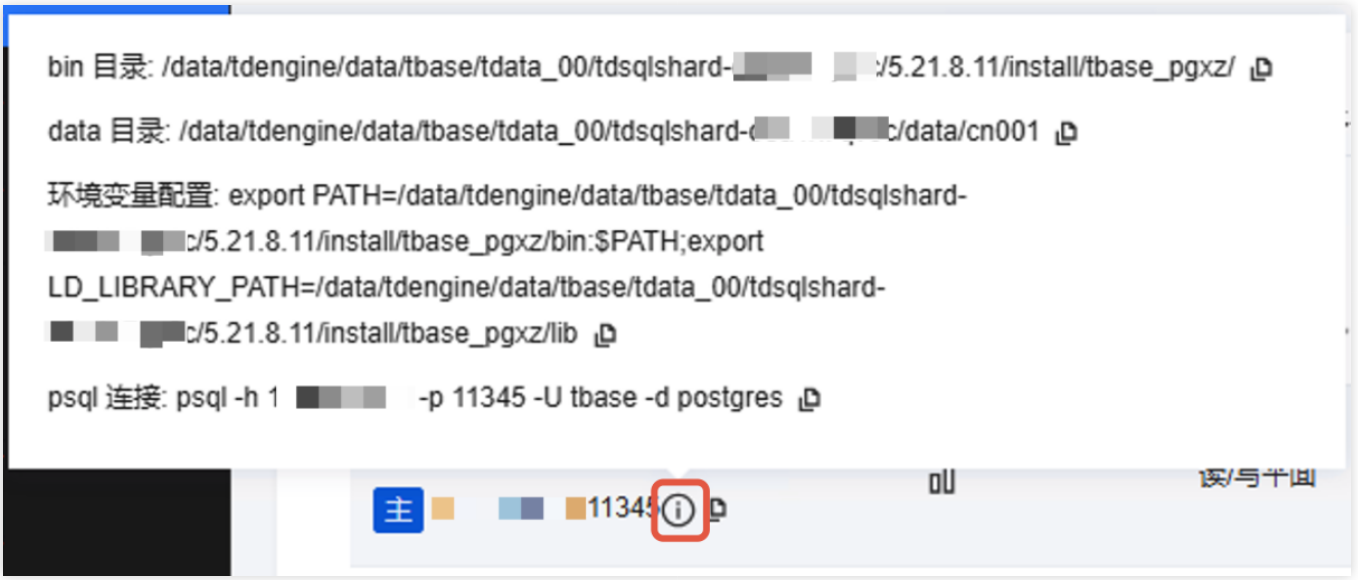
连接数据库

获取数据库连接配置信息，登录数据库系统：

1. 登录TDSQL管理平台在左侧选择 实例管理 并点击需要登录的实例，进入 节点管理 页（分布式下选择 CN 页，集中式选择 DN 页）。



2. 选择需要登录的节点IP，鼠标悬停在其右侧 图标获得环境配置信息。



3. 登录数据库节点服务器，切换到tbase用户并使用上述的export命令设置环境变量。

```
su - tbase
export PATH=/data1/tdengine/data/tbase/tdata_00/tdsqlshard-o62xkfw1a0/5.21.8.11/install/tbase_pgxz/bin:$PATH;
export LD_LIBRARY_PATH=/data1/tdengine/data/tbase/tdata_00/tdsqlshard-o62xkfw1a0/5.21.8.11/install/tbase_pgxz/lib
```

4. 在本机使用psql命令连接数据库（tbase用户一般只用于本机登录，可以创建普通用户用于业务开发）。

> 注意：
>
> 是连接到cn节点(后面没特别说明，所有数据库操作都是连接到cn节点)。

```
psql -h database_host_ip -p 11345 -U tbase -d postgres
```

创建用户

TDSQL PG可以用CREATE USER创建一个用户，需要使用超级用户或者具有CREATEROLE权限的用户（比如：tbase）。

- 示例：使用 create user 创建用户并指定密码。

```
postgres=# create user mgr password 'mypassword';
CREATE ROLE
```

创建完成后，使用 \du 展示用户信息。

```
postgres=# \du mgr
          List of roles
```

```
 Role name | Attributes | Member of
-----------+------------+-----------
 mgr       |            | {}
```

可以使用 ALTER USER 重新设置用户密码。

```
postgres=# alter user mgr password 'newpassword';
ALTER ROLE
```

启用Oracle兼容特性

TDSQL PG内核引擎会根据数据库类型来区分Oracle 兼容模式与PG 模式。
在创建数据库的时候使用 sql mode``db_mode 关键字即可创建一个指定模式的数据库，db_mode可选oracle或postgresql。不指定即默认PostgreSQL模式。
示例：

```
postgres=# create database ora sql mode oracle;
CREATE DATABASE

postgres=# \l ora
                              List of databases
 Name | Owner | Encoding | Sqlmode |  Collate   |   Ctype    | Access privileges
------+-------+----------+---------+------------+------------+-------------------
 ora  | tbase | UTF8     | oracle  | en_US.utf8 | en_US.utf8 |
(1 row)

postgres=# create database pg_db;
CREATE DATABASE
postgres=# \l pg_db
                               List of databases
 Name  | Owner | Encoding | Sqlmode  |  Collate   |   Ctype    | Access privileges
-------+-------+----------+----------+------------+------------+-------------------
 pg_db | tbase | UTF8     | postgres | en_US.utf8 | en_US.utf8 |
(1 row)
```

可以看到 ora 数据库的 Sqlmode 列显示 oracle ，即表示 ora 数据库当前为 Oracle 模式。
而 pg_db 数据库的 Sqlmode 列显示 postgres ，即表示 pg_db 数据库当前为 PostgreSQL 模式。
接下来切换至 ora 数据库或者直接访问 ora 数据库即可使用Oracle模式。

```
\c ora
```

或：

```
psql -h your_server_ip -p your_server_port -U tbase -d ora
```

创建模式

- 标准语句。

```
ora=# create schema tbase_ora;
CREATE SCHEMA
```

- 扩展语法，不存在时才创建。

```
ora=# create schema if not exists tbase_ora;
NOTICE:  schema "TBASE_ORA" already exists, skipping
CREATE SCHEMA
```

表操作

创建表

使用命令 create table 创建表

1. 不用指定shard key建表，系统默认使用第一个字段做为表的shard key

```
create table shared_col_table(id serial not null,nickname text);
```

2. 指定shard key建表

```
create table t_appoint_col(id serial not null,nickname text) distribute by shard(nickname);
```

> 注意：
>
> 分布键选择原则
>
> - v5.21.x支持多列分布键
> - 如果有主键，则选择主键做分布键
> - 如果主键是复合字段组合，则选择过滤性好的字段做分布键
> - 没有主键的可以使用UUID来做分布键
> - 让数据尽可能的分布得足够散

3. 表不存在时才创建

```
create table IF NOT EXISTS t(id int,mc text);
```

查看表

使用 \d 命令查看表

```
ora=# \d t_appoint_col;
                        Table "public.T_APPOINT_COL"
  Column  |  Type   | Collation | Nullable |                 Default
----------+---------+-----------+----------+-----------------------------------------
 ID       | INTEGER |           | not null | NEXTVAL('T_APPOINT_COL_ID_SEQ'::REGCLASS)
 NICKNAME | TEXT    |           |          |
```

使用Oracle兼容特性

使用层次查询获取数据

```
ora=# create table ora_table(a int, b int);
CREATE TABLE
ora=# insert into ora_table values(1, 2);
INSERT 0 1
ora=# insert into ora_table values(1, 2);
INSERT 0 1
ora=# select * from ora_table connect by level < 2;
 A | B
---+---
 1 | 2
 1 | 2
(2 rows)
```

使用ROWNUM对行进行编号

```
ora=# select rownum, a, b from ora_table where rownum = 1;
 ROWNUM | A | B
--------+---+---
      1 | 1 | 2
(1 row)
```

修改表

使用命令 alter table 修改表

示例：

1. 修改表名

```
alter table t rename to tbase;
```

2. 增加表字段

```
alter table tbase add column age integer;
```

3. 修改字段类型

```
alter table tbase alter column age type float8;
```

4. 修改表所属模式

```
alter table tbase set schema public;
```

删除表

使用命令 drop table 删除表
示例：当表存在时删除

```
drop table if exists tbase;
```

索引操作

创建索引

使用命令 create index 创建索引
示例：

1. 创建唯一索引

```
create table t_first_col_share(id serial not null,nickname text);
create unique index t_first_col_share_id_uidx on t_first_col_share using btree(id);
```

> 注意：
>
> 非shard key字段不能建立唯一索引

2. 创建多字段索引

```
create table t_mul_idx (f1 int,f2 int,f3 int,f4 int);
create index t_mul_idx_idx on t_mul_idx(f1,f2,f3);
```

查看索引

1. 使用 \d 命令查看索引。

```
ora=# \d t_first_col_share;
                        Table "public.T_FIRST_COL_SHARE"
  Column  |  Type   | Collation | Nullable |                  Default
----------+---------+-----------+----------+-------------------------------------------
 ID       | INTEGER |           | not null | NEXTVAL('T_FIRST_COL_SHARE_ID_SEQ'::REGCLASS)
 NICKNAME | TEXT    |           |          |
Indexes:
    "T_FIRST_COL_SHARE_ID_UIDX" UNIQUE, btree (ID) WITH (checksum='on')
```

2. 使用 \di 命令查看索引

```
ora=# \di
                      List of relations
 Schema |          Name           | Type  | Owner |       Table
--------+-------------------------+-------+-------+-------------------
 public | T_FIRST_COL_SHARE_ID_UIDX | index | tbase | T_FIRST_COL_SHARE
 public | T_MUL_IDX_IDX             | index | tbase | T_MUL_IDX
(2 rows)
```

3. 使用 user_indexes 视图列出索引

```
ora=# select INDEX_NAME from user_indexes where table_name='T_FIRST_COL_SHARE';
       INDEX_NAME
--------------------------
 T_FIRST_COL_SHARE_ID_UIDX
(1 row)
```

删除索引

使用命令 DROP INDEX 删除索引

```
drop index t_first_col_share_id_uidx;
```

插入数据

insert用于向一张表中插入数据，期望插入的数据可以是一条，多条或者是一个select查询的结果集。

使用 INSERT INTO... 命令插入数据

- 创建表 tbase 并插入一行数据：

```
ora=# create table tbase(id serial not null,nickname text) distribute by shard(id);
CREATE TABLE
ora=# insert into tbase(id,nickname) values(1,'hello TDSQL PG');
```

```
INSERT 0 1
```

- 向表 tbase 插入多行数据：

```
ora=# insert into tbase(id,nickname) values(2,'TDSQL PG'),(3,null),(4,'TDSQL PG good');
COPY 3
ora=# select * from tbase;
 ID |    NICKNAME
----+----------------
  1 | hello TDSQL PG
  2 | TDSQL PG
  3 |
  4 | TDSQL PG good
(4 rows)
```

删除数据

delete用于删除表数据，可以全部删除或者部分删除。

- 示例，带条件删除：

```
ora=# select * from tbase;
 ID |    NICKNAME
----+----------------
  1 | hello TDSQL PG
  3 |
  4 | TDSQL PG good
  2 | TDSQL PG
(4 rows)

ora=# delete from tbase where id=4;
DELETE 1
```

- 示例，null条件的表达方式

```
ora=# delete from tbase where nickname is null;
DELETE 1
ora=# select * from tbase;
 ID |    NICKNAME
----+----------------
  1 | hello TDSQL PG
  2 | TDSQL PG
(2 rows)
```

更新数据

update用于更新表数据，可以更新一个字段或者多个字段。

```
ora=# update tbase set nickname ='Hello TDSQL PG' where id=1;
UPDATE 1
```

查询数据

select用于从一张表或视图中获取数据。
使用 select ... from ... 命令查询数据

- 创建 t_grouping 表并插入内容

```
create table t_grouping(id int,dep varchar(20),product varchar(20),num int);
insert into t_grouping values(1,'dept no1','phone',90);
insert into t_grouping values(2,'dept no1','computer',80);
insert into t_grouping values(3,'dept no1','phone',70);
insert into t_grouping values(4,'dept no2','computer',60);
insert into t_grouping values(5,'dept no2','phone',50);
insert into t_grouping values(6,'dept no2','computer',60);
insert into t_grouping values(7,'dept no3','phone',70);
insert into t_grouping values(8,'dept no3','computer',80);
insert into t_grouping values(9,'dept no3','phone',90);
```

- 查询 t_grouping 表所有内容

```
ora=# select * from t_grouping;
 ID |   DEP    | PRODUCT  | NUM
----+----------+----------+-----
  1 | dept no1 | phone    |  90
  2 | dept no1 | computer |  80
  5 | dept no2 | phone    |  50
  6 | dept no2 | computer |  60
  8 | dept no3 | computer |  80
  9 | dept no3 | phone    |  90
  3 | dept no1 | phone    |  70
  4 | dept no2 | computer |  60
  7 | dept no3 | phone    |  70
(9 rows)
```

- 按dep，product汇总，排序展示

```
ora=# select dep,product,sum(num) from t_grouping group by dep,product order by dep,product;
   DEP    | PRODUCT  | SUM
----------+----------+-----
 dept no1 | computer |  80
 dept no1 | phone    | 160
 dept no2 | computer | 120
 dept no2 | phone    |  50
 dept no3 | computer |  80
 dept no3 | phone    | 160
(6 rows)
```

- 带条件查询

```
ora=# select * from t_grouping where dep='dept no2';
 ID |   DEP    | PRODUCT  | NUM
----+----------+----------+-----
  5 | dept no2 | phone    |  50
  6 | dept no2 | computer |  60
  4 | dept no2 | computer |  60
(3 rows)
```

提交事务

事务提交用于保存对数据库的更改命令。

命令如下：

COMMIT; 或者 COMMIT TRANSACTION;

例：

```
insert into tbase(id,nickname) values(5,'hello new TDSQL PG');
BEGIN;
delete from tbase where id=5;
COMMIT;
```

> 注意：
>
> 在COMMIT执行之前，tbase表中id为5的记录会被另一会话查询到。

回滚事务

事务回滚用于撤销尚未保存到数据库的命令。

命令语法如下：

ROLLBACK; 或者 ROLLBACK to savepoint_name;

例1：

```
BEGIN;
delete from tbase where id in (1,2);
select * from tbase;
ROLLBACK;
select * from tbase;
```

> 注意：
>
> 在ROLLBACK执行之后，事务对数据的删除不会保存到数据库中。

例2:

```
BEGIN;
delete from tbase where id =1;
select * from tbase;
savepoint A;
delete from tbase where id =2;
select * from tbase;
ROLLBACK to A;
COMMIT;
```

> 注意：
>
> 在ROLLBACK to A;执行之前同一进程查询不到id为1和2的数据。回滚到保存点之后，id为2数据可查询到。
> ROLLBACK to savepoint_name;不会结束事务，需使用COMMIT;提交事务。

开发者指南
开发指南（V5.06）
概述

本文档涵盖TDSQL PostgreSQL连接方式，SQL语言开发编写，Oracle兼容特性等内容，目的是指导应用开发。
本文档适用于使用TDSQL PostgreSQL的应用开发人员、数据库开发设计人员、数据库管理员等。
本文档适用于TDSQL PostgreSQL V5.06.4及以上版本。

连接到数据库
使用TSudio
TStudio登录

地址：http://172.16.0.29:5050/
用户名：postgres@postgres.com
密码：postgres
⚠️实际使用时，应将172.16.0.29替换为安装了TStudio服务的机器ip，一般为center master的ip；用户名密码为部署服务时配置。



可以通过主界面-添加新的服务器进行实例连接，对象管理等操作。



可以通过配置tstudio-首选项界面，进行工具配置。

通过TStudio连接数据库实例

通过"添加新的服务器"，可以连接一个TDSQL PostgreSQL实例进行对象操作与管理。



名称（必填）：连接实例的自定义名称
服务器组：连接实例的组别，方便归类管理
注释（选填）：连接实例的描述



主机名称：连接实例的IP
端口：预先配置好的连接端口
维护数据库：登入数据库
用户名：登入数据库用户名
密码：登入数据库密码
SSL模式：SSL安全连接模式，默认即可。
之后即可通过左侧控制台页面，查看实例对象。

TStudio常用操作
表管理

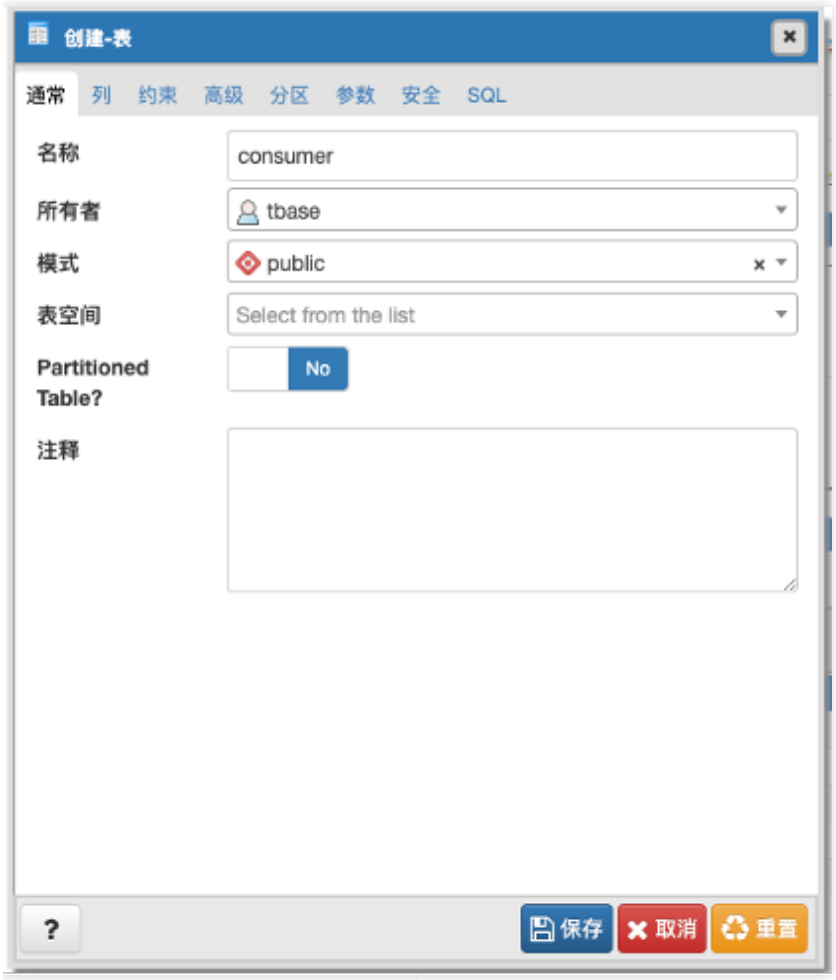
通过控制台创建表

1）Servers实例名称登入数据库名模式名

2）右键“模式名” - 〉创建 - 〉表



3) 在“创建-表”菜单中，根据提示填写表相关信息，保存即可创建表。


创建-表
通常
列
约束
高级
分区
参数
安全
SQL
名称
consumer
所有者
tbase
模式
public
表空间
Select from the list
Partitioned Table?
No
注释
?
保存
取消
重置



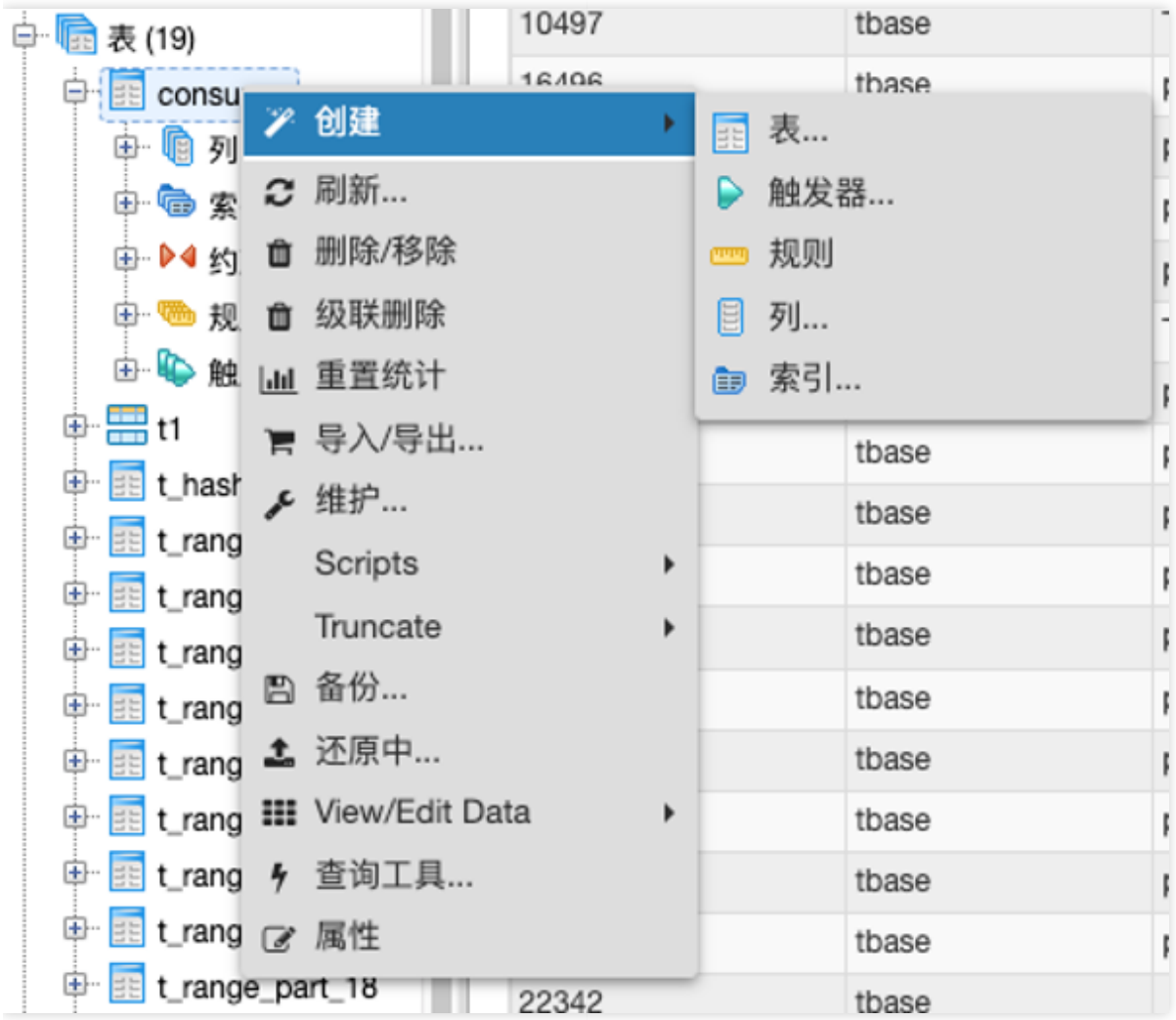
注释：相当于comment on table。

约束页签：可添加主键、外键、检查（check）约束、唯一约束、排除约束。

SQL页签：配置完成之后，可以在该页面查看建表语句。

通过控制台创建表索引

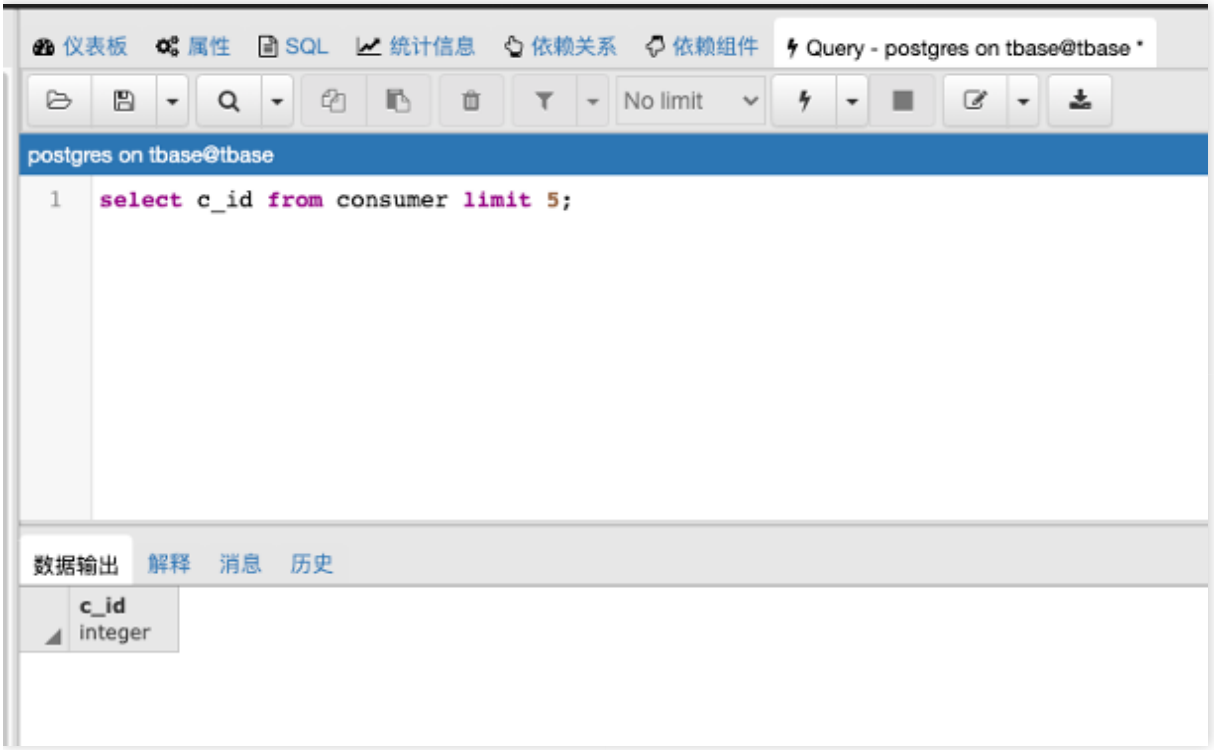
1）右键新创建的表，菜单中选择创建-索引



2）根据页面提示填写创建索引信息，包括索引名称、表空间、注释、访问方法（索引类型）以及索引列等，保存即可创建索引。

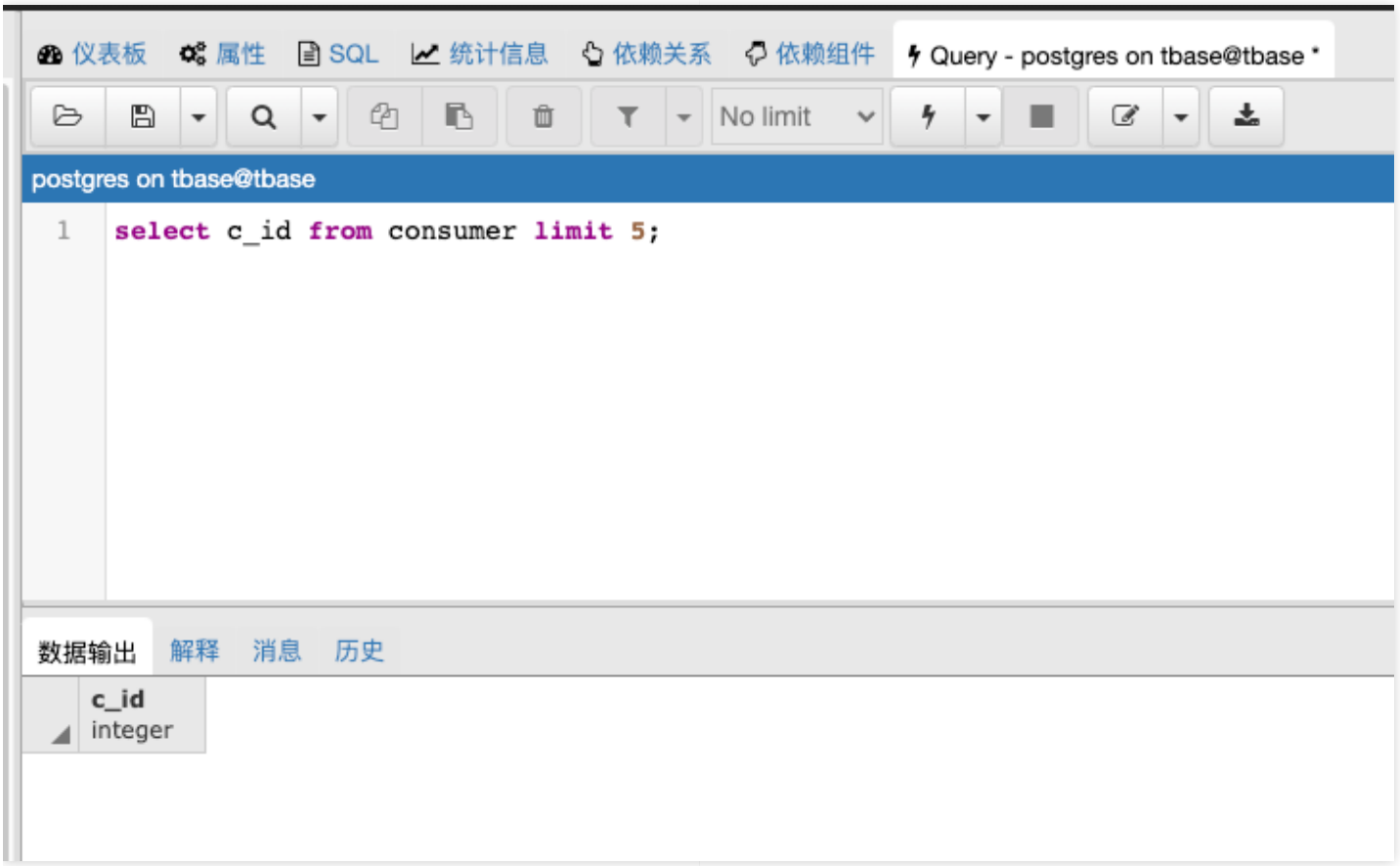
运行脚本

Tstudio也拥有像主流数据库管理工具（例如PL/SQL Developer）一样直接执行SQL命令的功能。
可以通过顶部菜单栏，“工具—查询工具”进入，也可以右键想要执行SQL脚本的数据库对象，选择“查询工具”进入。
点击编辑区的执行按钮或F5，即可执行SQL，下方数据输出展示执行结果。


仪表板
属性
SQL
统计信息
依赖关系
依赖组件
Query - postgres on tbase@tbase *
No limit
postgres on tbase@tbase
1
select c_id from consumer limit 5;
数据输出
解释
消息
历史
c_id
integer

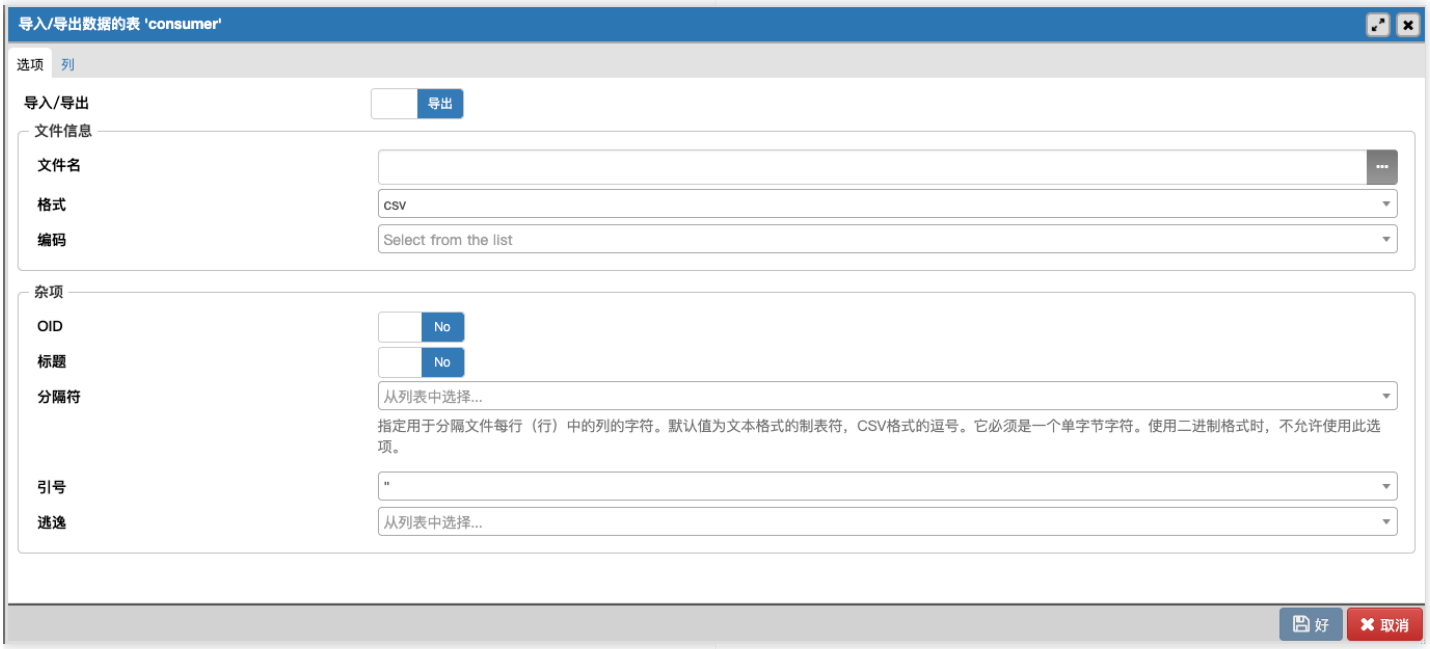
运行脚本

Tstudio也拥有像主流数据库管理工具（例如PL/SQL Developer）一样直接执行SQL命令的功能。
可以通过顶部菜单栏，“工具—查询工具”进入，也可以右键想要执行SQL脚本的数据库对象，选择“查询工具”进入。
点击编辑区的执行按钮或F5，即可执行SQL，下方数据输出展示执行结果。



数据导入导出

Tstudio也拥有像主流数据库管理工具（例如PL/SQL Developer）一样通过控制台导入/导出数据的功能。
选择要操作的表，可以通过顶部菜单栏，“工具—导入/导出”进入，也可以右键想要导入/导出的表，选择“导入/导出”进入。
根据页面提示操作，需要选择导入还是导出功能，文件信息包括文件名、格式，要导入/导出的列。



默认导出数据文件存放于目录/usr/local/install/tstudio/storage/postgres/

使用shell
使用shell连接数据库实例

Shell连接TDSQL PostgreSQL实例首先需要连接集群的Center节点。
例：ssh root@XX.XX.XX.XX。XX为center节点的IP。
可以通过四种方式连接实例。
1）使用参数连接
psql -h 172.16.0.29 -p 15432 -U tbase -d postgres
-h：host
-p：port
-U：数据库user
-d：database
2）使用conninfo字符串或者一个URI
psql postgresql://tbase@172.16.0.29:15432/postgres
3）配置证书连接
psql 'host=172.16.0.29 port=15432 dbname=postgres user=tbase sslmode=verify-ca
sslcert=postgresql.crt sslkey=postgresql.key sslrootcert=root.crt'
sslcert说明
4）配置环境变量后快捷连接
PGUSER=tbase
PGHOST=127.0.0.1
PGDATABASE=postgres
PGPORT=15432
export PGHOST PGUSER PGDATABASE PGPORT
环境变量生效后，命令psql即可快速连接实例

常用操作命令

TDSQL PostgreSQL常用的命令如下：

\password	设置密码
\q	退出
\l	列出所有数据库
\c [database_name]	连接其他数据库，无参数显示当前连接数据库以及用户
\d	列出当前数据库的所有表
\d [table_name]	列出某一张表格的结构
\du	列出所有用户
\e	打开文本编辑器
\conninfo	列出当前数据库和连接的信息
\timing on/off	打开/关闭显示sql语句执行的时间
\i [file_name]	执行sql文件命令
\! [command]	执行外部命令
\o [file_name]	将执行结果保存至文件

⚠️注：
? 可查看所有psql命令列表及用法
\h 可查看所有sql命令解释及用法。例\h select 可以查看select的用法。

使用ems postgresql manager

EMS PostgreSQL Manager，这是一款商业的postgresql的客户端管理工具，可以用来创建对象，执行sql等。
配置页面配置好Host name，Username，Password后，即可连接。其余操作与主流数据库管理工具类似。

关于DML与事务
DML语法
关于SELECT

select用于从一张表或视图中获取数据。
使用select ...from...命令查询数据
示例说明：
示例：
1）分组查询，排序
查询销售明细表，按dep，product汇总，排序展示

```
create table t_grouping(id int,dep varchar(20),product varchar(20),num int);
insert into t_grouping values(1,'业务1部','手机',90);
insert into t_grouping values(2,'业务1部','电脑',80);
insert into t_grouping values(3,'业务1部','手机',70);
insert into t_grouping values(4,'业务2部','电脑',60);
insert into t_grouping values(5,'业务2部','手机',50);
insert into t_grouping values(6,'业务2部','电脑',60);
insert into t_grouping values(7,'业务3部','手机',70);
insert into t_grouping values(8,'业务3部','电脑',80);
insert into t_grouping values(9,'业务3部','手机',90);
select dep,product,sum(num) from t_grouping group by dep,product order by dep,product;
```

2）（左连接）表连接（类型内连接、左连接、右连接、全连接）

```
select * from tbase left join t_appoint_col on tbase.id=t_appoint_col.id;
```

3）子查询，排序，限制条数

```
select * from tbase order by (select id from tbase order by random() limit 1);
```

4）返回结果集的差值

```
select * from t_except1 except select * from t_except2;
```

5） 返回结果集的交集

```
create table t_intersect1(id int,mc text);
insert into t_intersect1 values(1,'tbase'),(2,'tbase');
create table t_intersect2(id int,mc text);
insert into t_intersect2 values(1,'tbase'),(3,'tbase');
```

```
select * from t_intersect1 INTERSECT select * from t_intersect2;
```

6）合并多个查询结果

```
--不过虑重复的记录
select * from tbase union all select * from t_appoint_col;
--过虑重复的记录select * from tbase union select * from t_appoint_col;
```

7）查询指定DN上的数据

```
create table t_direct(id int,mc text);
insert into t_direct values(1,'tbase'),(3,'pgxz');
EXECUTE DIRECT ON (dn001) 'select * from t_direct;';
```

8. 查询记录所在DN

```
select xc_node_id,* from t1;
select t1.xc_node_id,pgxc_node.node_name,t1.* from t1,pgxc_node where t1.xc_node_id=pgxc_node.n
ode_id;
```

SELECT使用场景
with子查询及递归的使用
with 子查询

#实验数据
```
create table t_area(id int,pid int,area varchar);
insert into t_area values(1,null,'某某省');
insert into t_area values(2,1,'城市1');
insert into t_area values(3,1,'城市2');
insert into t_area values(4,2,'A区');
insert into t_area values(5,2,'B区');
insert into t_area values(6,2,'C区');
insert into t_area values(7,3,'A区');
insert into t_area values(8,3,'B区');
```

with 子查询
```
postgres=# with t_gd_city AS (
   select * from t_area where pid=1
)
select  * FROM t_gd_city;
 id | pid |  area
----+-----+--------
  2 |   1 | 城市1
  3 |   1 | 城市2
(2 rows)
```

with递归

```
postgres=# WITH RECURSIVE t_city AS
(
  SELECT id, area FROM  t_area  WHERE id = 1
  UNION ALL
  SELECT
     t_area.id,
     t_city.area || ' > ' || t_area.area
  FROM
     t_city
     INNER JOIN t_area ON t_area.pid = t_city.id
)
SELECT id,area FROM t_city order by area;
 id |          area
----+-------------------------
1 | 某某省
3 | 某某省 > 城市2
8 | 某某省 > 城市2 > A区
7 | 某某省 > 城市2 > B区
2 | 某某省 > 城市1
5 | 某某省 > 城市1 > A区
6 | 某某省 > 城市1 > B区
4 | 某某省 > 城市1 > C区
```

执行non-recursive term，其结果作为recursive term中对t_city的引用，同时将这部分结果放入临时的working table中，重复执行如下步骤，直到working table为空，来看看具体过程：

- 执行

```
SELECT id, area FROM  t_area  WHERE id = 1
```

结果集和working table为

```
1 | 某某省
```

- 执行

```
SELECT
     t_area.id,
     t_city.area || ' > ' || t_area.area
  FROM
     t_city
     INNER JOIN t_area ON t_area.pid = t_city.id
```

结果集和working table为

3 | 某某省 > 城市2
2 | 某某省 > 城市1

- 再次执行recursive query,结果集和working table为：

8 | 某某省 > 城市2 > A区
7 | 某某省 > 城市2 > B区
5 | 某某省 > 城市1 > A区
6 | 某某省 > 城市1 > B区
4 | 某某省 > 城市1 > C区

- 继续执行recursive query，结果集和working table为空
- 结束递归，将执行过程结果合并，返回结果集。

访问函数

```
postgres=# select md5(random()::text);
               md5
----------------------------------
 3eb6c0c8f8355f0b0f0cad7a8f0f7491
```

数据排序

- 按某一列排序。

```
postgres=# INSERT into tbase (id,nickname) VALUES(1,'hello TDSQL PG');
INSERT 0 1
postgres=# INSERT into tbase (id,nickname) VALUES(2,'TDSQL PG好');
INSERT 0 1
postgres=# INSERT into tbase (id,nickname) VALUES(1,'TDSQL PG分布式数据库的时代来了');
INSERT 0 1
postgres=# select * from tbase order by id;
 id |         nickname
----+---------------------------
  1 | hello TDSQL PG
  1 | TDSQL PG分布式数据库的时代来了
  2 | TDSQL PG好
(3 rows)
```

- 按第一列排序。

```
postgres=# select * from tbase order by 1;
 id |         nickname
----+---------------------------
  1 | hello TDSQL PG
  1 | TDSQL PG分布式数据库的时代来了
  2 | TDSQL PG好
(3 rows)
```

- 按id升级排序，再按nickname降序排序。

```
postgres=# select * from tbase order by id,nickname desc;
 id |         nickname
----+---------------------------
  1 | TDSQL PG分布式数据库的时代来了
  1 | hello TDSQL PG
  2 | TDSQL PG好
(3 rows)
```

- 效果与上面的语句一样。

```
postgres=# select * from tbase order by 1,2 desc;
 id |         nickname
```

```
----+----------------------------
  1 | TDSQL PG分布式数据库的时代来了
  1 | hello TDSQL PG
  2 | TDSQL PG好
(3 rows)
```

- 随机排序。

```
postgres=# select * from tbase order by random();
 id |         nickname
----+----------------------------
  1 | TDSQL PG分布式数据库的时代来了
  2 | TDSQL PG好
  1 | hello TDSQL PG
(3 rows)
```

- 计算排序。

```
postgres=# select * from tbase order by md5(nickname);
 id |         nickname
----+----------------------------
  2 | TDSQL PG好
  1 | TDSQL PG分布式数据库的时代来了
  1 | hello TDSQL PG
(3 rows)
```

- 排序子查询。

```
postgres=# select * from tbase order by (select id from tbase order by random() limit 1);
 id |         nickname
----+----------------------------
  1 | hello TDSQL PG
  2 | TDSQL PG好
  1 | TDSQL PG分布式数据库的时代来了
(3 rows)
```

- null值排序结果处理。

```
postgres=# insert into tbase values(4,null);
INSERT 0 1
null值记录排在最前面
postgres=#  select * from tbase order by nickname nulls first;
 id |         nickname
```

```
----+----------------------------
  4 |
  1 | hello TDSQL PG
  1 | TDSQL PG分布式数据库的时代来了
  2 | TDSQL PG好
(4 rows)
null值记录排在最后
postgres=#  select * from tbase order by nickname nulls last;
 id |          nickname
----+----------------------------
  1 | hello TDSQL PG
  1 | TDSQL PG分布式数据库的时代来了
  2 | TDSQL PG好
  4 |
(4 rows)
```

- 按拼音排序。

```
postgres=# select * from (values ('张三'), ('李四'),('陈五')) t(myname) order by myname;
 myname
--------
 张三
 李四
 陈五
(3 rows)
```

如果不加处理，则按汉字的utf8编码进行排序，不符合用户使用习惯。

```
postgres=# select * from (values ('张三'), ('李四'),('陈五')) t(myname) order by convert(myname::bytea,'U
TF-8','GBK');
 myname
--------
 陈五
 李四
 张三
(3 rows)
```

- 使用convert函数实现汉字按拼音进行排序。

```
postgres=# select * from (values ('张三'), ('李四'),('陈五')) t(myname) order by convert_to(myname,'GB
K');
 myname
--------
 陈五
```

```
 李四
 张三
(3 rows)
```

- 使用convert_to函数实现汉字按拼音进行排序。

```
postgres=# select * from (values ('张三'), ('李四'),('陈五')) t(myname) order by myname  collate "zh_CN.u
tf8";
 myname
--------
 陈五
 李四
 张三
(3 rows)
```

通过指定排序规则collact来实现汉字按拼音进行排序。

where条件使用

- 单条件查询。

```
postgres=# select * from tbase where id=1;
 id |          nickname
----+----------------------------
  1 | hello TDSQL PG
  1 | TDSQL PG分布式数据库的时代来了
```

- 多条件and。

```
postgres=# select * from tbase where id=1 and nickname like '%h%';
 id |  nickname
----+-------------
  1 | hello TDSQL PG
(1 row)
```

- 多条件or。

```
postgres=# select * from tbase where id=2 or nickname like '%h%';
 id |  nickname
----+-------------
  1 | hello TDSQL PG
  2 | TDSQL PG好
(2 rows)
```

- ilike不区分大小写匹配。

```
postgres=# create table t_ilike(id int,mc text);
CREATE TABLE
postgres=# insert into t_ilike values(1,'TDSQL PG'),(2,'TDSQL PG');
INSERT 0 2
postgres=# select * from t_ilike where mc ilike '%tb%';
 id |  mc
----+-------
  1 | TDSQL PG
  2 | TDSQL PG
(2 rows)
```

- where条件也能支持子查询。

```
postgres=# select * from tbase where id=(select (random()*2)::integer from tbase order by random() li
mit 1);
 id | nickname
----+----------
(0 rows)

postgres=# select * from tbase where id=(select (random()*2)::integer from tbase order by random() li
mit 1);
 id |         nickname
----+---------------------------
  1 | hello TDSQL PG
  1 | TDSQL PG分布式数据库的时代来了
(2 rows)
```

- null值查询方法。

```
postgres=# select * from tbase where nickname is null;
 id | nickname
----+----------
  4 |
(1 row)

postgres=# select * from tbase where nickname is not null;
 id |         nickname
----+---------------------------
  1 | hello TDSQL PG
  2 | TDSQL PG好
  1 | TDSQL PG分布式数据库的时代来了
(3 rows)
```

- exists，只要有记录返回就为真。

```
postgres=#  create table t_exists1(id int,mc text);
CREATE TABLE

postgres=# insert into t_exists1 values(1,'TDSQL PG'),(2,'TDSQL PG');
INSERT 0 2

postgres=# create table t_exists2(id int,mc text);
CREATE TABLE

postgres=# insert into t_exists2 values(1,'TDSQL PG'),(1,'TDSQL PG');
INSERT 0 2
```

```
postgres=# select * from  t_exists1 where  exists(select 1 from t_exists2 where t_exists1.id=t_exists2.id)
;
 id |  mc
----+-------
  1 | TDSQL PG
(1 row)
```

- exists等价写法。

```
postgres=# select t_exists1.* from t_exists1,(select distinct id from t_exists2) as t where t_exists1.id=t.i
d;;
 id |  mc
----+-------
  1 | TDSQL PG
(1 row)
```

分页查询

- 默认从第一条开始，返回一条记录。

```
postgres=# select * from tbase limit 1;
 id |  nickname
----+-------------
  1 | hello TDSQL PG
(1 row)
```

- 使用offset指定从第几条开始，0表示第一条开如，返回1条记录。

```
postgres=# select * from tbase limit 1 offset 0;
 id |  nickname
----+-------------
  1 | hello TDSQL PG
(1 row)
```

- 从第3条开始，返回二条记录。

```
postgres=# select * from tbase limit 1 offset 2;
 id |          nickname
----+----------------------------
  1 | TDSQL PG分布式数据库的时代来了
(1 row)
```

- 上面的语句没有使用排序，返回结果不可预知，使用order by可以获得一个有序的结果。

```
postgres=# select * from tbase order by id limit 1 offset 2;
 id | nickname
----+-----------
  2 | TDSQL PG好
(1 row)
```

合并多个查询结果

- 不过滤重复的记录。

```
postgres=# select * from tbase union all select * from t_appoint_col;
 id |          nickname
----+----------------------------
  1 | hello TDSQL PG
  2 | TDSQL PG好
  1 | TDSQL PG分布式数据库的时代来了
  1 | hello TDSQL PG
(4 rows)
```

- 过滤重复的记录。

```
postgres=# select * from tbase union select * from t_appoint_col;
 id |          nickname
----+----------------------------
  1 | TDSQL PG分布式数据库的时代来了
  1 | hello TDSQL PG
  2 | TDSQL PG好
(3 rows)
```

返回两个结果的交集

```
postgres=# create table t_intersect1(id int,mc text);
CREATE TABLE
postgres=# insert into t_intersect1 values(1,'TDSQL PG'),(2,'TDSQL PG');
INSERT 0 2
postgres=# create table t_intersect2(id int,mc text);
CREATE TABLE
postgres=# insert into t_intersect2 values(1,'TDSQL PG'),(3,'TDSQL PG');
INSERT 0 2
postgres=# select * from t_intersect1 INTERSECT select * from t_intersect2;
 id |  mc
----+-------
  1 | TDSQL PG
(1 row)
```

返回两个结果的差集

```
postgres=# create table t_except1(id int,mc text);
CREATE TABLE
postgres=# insert into t_except1 values(1,'TDSQL PG'),(2,'TDSQL PG');
INSERT 0 2
postgres=# create table t_except2(id int,mc text);
CREATE TABLE
postgres=# insert into t_except2 values(1,'TDSQL PG'),(3,'TDSQL PG');
INSERT 0 2
postgres=# select * from t_except1 except select * from t_except2;
 id |  mc
----+-------
  2 | TDSQL PG
(1 row)
```

any用法

示例1、

```
postgres=# create table t_any(id int,mc text);
CREATE TABLE
postgres=# insert into t_any values(1,'TDSQL PG'),(2,'TDSQL PG');
INSERT 0 2
postgres=# select * from t_any where id>any (select 1 union select 3);
 id |  mc
----+-------
  2 | TDSQL PG
(1 row)
```

只需要大于其中一个值即为真。

示例2、

```
drop table if exists t_any_dml_20220718_1;

create table t_any_dml_20220718_1(v int, w int);

insert into t_any_dml_20220718_1 values(1, 1);

insert into t_any_dml_20220718_1 values(2, 2);

update t_any_dml_20220718_1 set w = 3 where w > any(0, 1);

delete t_any_dml_20220718_1 where w > any(0, 1);

insert into t_any_dml_20220718_1 values(1, 1);

insert into t_any_dml_20220718_1 select 2, 2 from t_any_dml_20220718_1 where w <> any(0, 1);

with cte_any_update as (select v from t_any_dml_20220718_1 where v > any(0, 1)) update t_any_dml_2
0220718_1 set w = (select v from cte_any_update order by v limit 1) where w > all(0, 1);

select * from t_any_dml_20220718_1 order by v;

with cte_any_delete as (select v from t_any_dml_20220718_1 where v > any(0, 1)) delete t_any_dml_202
20718_1 where w = any(select v from cte_any_delete order by v limit 1);

select * from t_any_dml_20220718_1 order by v;

insert into t_any_dml_20220718_1 values(1, 1);
```

```
with cte_any_insert as (select v, w from t_any_dml_20220718_1 where v > any(0, 1)) insert into t_any_dml_20220718_1 select v, w from cte_any_insert;

select * from t_any_dml_20220718_1 order by v;
```

all用法

示例1、

```
postgres=# create table t_all(id int,mc text);
CREATE TABLE
postgres=# insert into t_all values(2,'TDSQL PG'),(3,'TDSQL PG');
INSERT 0 2

postgres=# select * from t_all where id>all (select 1 union select 2);
 id |  mc
----+-------
  3 | TDSQL PG
(1 row)
```

需要大于所有值才为真。

示例2、

```
drop table if exists t_all_dml_20220718_1;

create table t_all_dml_20220718_1(v int, w int);

insert into t_all_dml_20220718_1 values(1, 1);

insert into t_all_dml_20220718_1 values(2, 2);

update t_all_dml_20220718_1 set w = 3 where w > all(0, 1);

delete t_all_dml_20220718_1 where w > all(0, 1);

select * from t_all_dml_20220718_1 order by v;

insert into t_all_dml_20220718_1 select 2, 2 from t_all_dml_20220718_1 where w >= all(0, 1);

with cte_all_update as (select v from t_all_dml_20220718_1 where v > all(0, 1)) update t_all_dml_202207
18_1 set w = (select v from cte_all_update order by v limit 1) where w > all(0, 1);

select * from t_all_dml_20220718_1 order by v;

with cte_all_delete as (select v from t_all_dml_20220718_1 where v > all(0, 1)) delete t_all_dml_2022071
8_1 where w = all(select v from cte_all_delete order by v limit 1);

select * from t_all_dml_20220718_1 order by v;

with cte_all_insert as (select v, w from t_all_dml_20220718_1 where v >= all(0, 1)) insert into t_all_dml_2
```

0220718_1 select v, w from cte_all_insert;

select * from t_all_dml_20220718_1 order by v;

some用法

```
drop table if exists t_some_dml_20220718_1;

create table t_some_dml_20220718_1(v int, w int);
insert into t_some_dml_20220718_1 values(1, 1);
insert into t_some_dml_20220718_1 values(2, 2);

update t_some_dml_20220718_1 set w = 3 where w > some(0, 1);
delete t_some_dml_20220718_1 where w > some(0, 1);
insert into t_some_dml_20220718_1 values(1, 1);

insert into t_some_dml_20220718_1 select 2, 2 from t_some_dml_20220718_1 where w <> some(0, 1);

with cte_some_update as (select v from t_some_dml_20220718_1 where v > some(0, 1)) update t_some
_dml_20220718_1 set w = (select v from cte_some_update order by v limit 1) where w > all(0, 1);

select * from t_some_dml_20220718_1 order by v;

with cte_some_delete as (select v from t_some_dml_20220718_1 where v > some(0, 1)) delete t_some_
dml_20220718_1 where w = some(select v from cte_some_delete order by v limit 1);

select * from t_some_dml_20220718_1 order by v;

insert into t_some_dml_20220718_1 values(1, 1);

with cte_some_insert as (select v, w from t_some_dml_20220718_1 where v > some(0, 1)) insert into t_s
ome_dml_20220718_1 select v, w from cte_some_insert;

select * from t_some_dml_20220718_1 order by v;
```

聚集查询

- 统计记录数。

```
postgres=# select count(1) from tbase;
 count
-------
     3
(1 row)
```

- 统计不重复值的记录表。

```
postgres=# select count(distinct id) from tbase;
 count
-------
     2
(1 row)
```

- 求和。

```
postgres=# select sum(id) from tbase;
 sum
-----
   4
(1 row)
```

- 求最大值。

```
postgres=# select max(id) from tbase;
 max
-----
   2
(1 row)
```

- 求最小值。

```
postgres=# select min(id) from tbase;
 min
-----
   1
(1 row)
```

- 求平均值。

```
postgres=# select avg(id) from tbase;
        avg
--------------------
 1.3333333333333333
(1 row)
```

多表关联

- 内连接。

```
postgres=# select * from tbase inner join t_appoint_col on tbase.id=t_appoint_col.id;
 id |         nickname          | id |  nickname
----+---------------------------+----+-------------
  1 | hello TDSQL PG            |  1 | hello TDSQL PG
  1 | TDSQL PG分布式数据库的时代来了 |  1 | hello TDSQL PG
(2 rows)
```

- 左外连接。

```
postgres=# select * from tbase left join t_appoint_col on tbase.id=t_appoint_col.id;
 id |         nickname          | id |  nickname
----+---------------------------+----+-------------
  1 | hello TDSQL PG            |  1 | hello TDSQL PG
  2 | TDSQL PG好                |    |
  1 | TDSQL PG分布式数据库的时代来了 |  1 | hello TDSQL PG
(3 rows)
```

- 右外连接。

```
postgres=# select * from tbase right join t_appoint_col on tbase.id=t_appoint_col.id;
 id |         nickname          | id |  nickname
----+---------------------------+----+-------------
  1 | TDSQL PG分布式数据库的时代来了 |  1 | hello TDSQL PG
  1 | hello TDSQL PG            |  1 | hello TDSQL PG
    |                           |  5 | Power TDSQL PG
(3 rows)
```

- 全连接。

```
postgres=# select * from tbase full join t_appoint_col on tbase.id=t_appoint_col.id;
 id |         nickname          | id |  nickname
----+---------------------------+----+-------------
  1 | hello TDSQL PG            |  1 | hello TDSQL PG
  2 | TDSQL PG好                |    |
  1 | TDSQL PG分布式数据库的时代来了 |  1 | hello TDSQL PG
    |                           |  5 | Power TDSQL PG
(4 rows)
```

聚合函数并发计算

- 单核计算。

```
postgres=#\timing
Timing ison.
postgres=#set max_parallel_workers_per_gather to 0;
SET
Time:0.633 ms
postgres=#select count(1) from t_count;
  count
----------
 20000000
(1 row)
 Time:3777.518 ms (00:03.778)
```

- 二核并行。

```
postgres=#set max_parallel_workers_per_gather to 2;
SET
Time:0.478 ms
postgres=#select count(1) from t_count;
  count
----------
 20000000
(1 row)
 Time:2166.481 ms (00:02.166)
```

- 四核并行。

```
postgres=#set max_parallel_workers_per_gather to 4;
SET
Time:0.315 ms
postgres=#select count(1) from t_count;
  count
----------
 20000000
(1 row)
 Time:1162.433 ms (00:01.162)
postgres=#
```

not in中包含了null，结果全为真

```
postgres=# create table t_not_in(id int,mc text);
CREATE TABLE
postgres=# insert into t_not_in values(1,'TDSQL PG'),(2,'pgxz');
INSERT 0 2
postgres=# select * from t_not_in where id not in (3,5);
 id |  mc
----+-------
  1 | TDSQL PG
  2 | pgxz
(2 rows)

postgres=# select * from t_not_in where id not in (3,5,null);
 id | mc
----+----
(0 rows)
```

只查某个dn的数据

```
postgres=# create table t_direct(id int,mc text);
CREATE TABLE
postgres=# insert into t_direct values(1,'TDSQL PG'),(3,'pgxz');
INSERT 0 2
postgres=# EXECUTE DIRECT ON (dn001) 'select * from t_direct';
 id |  mc
----+-------
  1 | TDSQL PG
(1 row)

postgres=# EXECUTE DIRECT ON (dn002) 'select * from t_direct';
 id |  mc
----+------
  3 | pgxz
(1 row)

postgres=# select * from t_direct ;
 id |  mc
----+-------
  1 | TDSQL PG
  3 | pgxz
(2 rows)
```

特殊应用

- 多行变成单行。

```
postgres=# create table t_mulcol_tosimplecol(id int,mc text);
CREATE TABLE

postgres=# insert into t_mulcol_tosimplecol values(1,'TDSQL PG'),(2,'TDSQL PG');
INSERT 0 2

postgres=# select array_to_string(array(select mc from t_mulcol_tosimplecol),',');
 array_to_string
-----------------
 TDSQL PG,TDSQL PG
(1 row)
```

- 一列变成多行。

```
postgres=# create table t_col_to_mulrow(id int,mc text);
CREATE TABLE

postgres=# insert into t_col_to_mulrow values(1,'TDSQL PG,TDSQL PG');
INSERT 0 1

postgres=# select regexp_split_to_table((select mc from t_col_to_mulrow where id=1 limit 1), ',');

 regexp_split_to_table
-----------------------
 TDSQL PG
 TDSQL PG
(2 rows)
```

查询记录所在dn

```
postgres=# select xc_node_id,* from t1;
 xc_node_id | f1 | f2
------------+----+----
 2142761564 |  1 |  3
 2142761564 |  1 |  3
(2 rows)
postgres=# select t1.xc_node_id,pgxc_node.node_name,t1.* from t1,pgxc_node where t1.xc_node_id=p
gxc_node.node_id;
 xc_node_id | node_name | f1 | f2
------------+-----------+----+----
 2142761564 | dn001     |  1 |  3
 2142761564 | dn001     |  1 |  3
(2 rows)
postgres=#
```

grouping sets/rollup/cube用法
group by 用法

#销售明细表

```
create table t_grouping(id int,dep varchar(20),product varchar(20),num int);
insert into t_grouping values(1,'业务1部','手机',90);
insert into t_grouping values(2,'业务1部','电脑',80);
insert into t_grouping values(3,'业务1部','手机',70);
insert into t_grouping values(4,'业务2部','电脑',60);
insert into t_grouping values(5,'业务2部','手机',50);
insert into t_grouping values(6,'业务2部','电脑',60);
insert into t_grouping values(7,'业务3部','手机',70);
insert into t_grouping values(8,'业务3部','电脑',80);
insert into t_grouping values(9,'业务3部','手机',90);
```

```
postgres=# select dep,product,sum(num) from t_grouping group by dep,product order by dep,product
;
   dep   | product | sum
---------+---------+-----
 业务1部 | 电脑    |  80
 业务1部 | 手机    | 160
 业务2部 | 电脑    | 120
 业务2部 | 手机    |  50
 业务3部 | 电脑    |  80
 业务3部 | 手机    | 160
```

按dep,product两级汇总分数

使用grouping sets

grouping sets说明：GROUPING SETS的每个子列表可以指定零个或多个列或表达式，并且与其直接在GROUPBY子句中的解释方式相同。 一个空的分组集合意味着所有的行都被聚合到一个组中
如按name,class单级分别汇总，再计算一个总分

```
postgres=# select dep,product,sum(num) from t_grouping group by grouping sets((dep),(product),()) o
rder by dep,product;
   dep   | product | sum
---------+---------+-----
 业务1部 |         | 240
 业务2部 |         | 170
 业务3部 |         | 240
         | 电脑    | 280
         | 手机    | 370
         |         | 650
```

使用grouping sets代替group by

```
postgres=# select dep,product,sum(num) from t_grouping group by grouping sets((dep,product)) orde
r by dep,product;
   dep   | product | sum
---------+---------+-----
 业务1部 | 电脑    |  80
 业务1部 | 手机    | 160
 业务2部 | 电脑    | 120
 业务2部 | 手机    |  50
 业务3部 | 电脑    |  80
 业务3部 | 手机    | 160
```

使用rollup

使用rollup
* rollup((a),(b))等价于grouping sets((a,b),(a),()) *

```
postgres=# select dep,product,sum(num) from t_grouping group by rollup((dep),(product)) order by d
ep,product;
   dep   | product | sum
---------+---------+-----
 业务1部 | 电脑    |  80
 业务1部 | 手机    | 160
 业务1部 |         | 240
 业务2部 | 电脑    | 120
 业务2部 | 手机    |  50
 业务2部 |         | 170
 业务3部 | 电脑    |  80
 业务3部 | 手机    | 160
 业务3部 |         | 240
         |         | 650
```

该功能等价于grouping sets((dep, product),(dep),())

```
postgres=# select dep,product,sum(num) from t_grouping group by grouping sets((dep, product),( de
p),()) order by dep,product;
   dep   | product | sum
---------+---------+-----
 业务1部 | 电脑    |  80
 业务1部 | 手机    | 160
 业务1部 |         | 240
 业务2部 | 电脑    | 120
 业务2部 | 手机    |  50
 业务2部 |         | 170
 业务3部 | 电脑    |  80
 业务3部 | 手机    | 160
 业务3部 |         | 240
         |         | 650
```

使用cube

#*cube((a),(b))等价于grouping sets((a,b),(a),(b),()) *

```
postgres=#select dep,product,sum(num) from t_grouping group by cube((dep),(product))order by dep,
product;
  dep  | product | sum
---------+---------+-----
 业务1部 | 电脑    |  80
 业务1部 | 手机    | 160
 业务1部 |         | 240
 业务2部 | 电脑    | 120
 业务2部 | 手机    |  50
 业务2部 |         | 170
 业务3部 | 电脑    |  80
 业务3部 | 手机    | 160
 业务3部 |         | 240
         | 电脑    | 280
         | 手机    | 370
         |         | 650
```

#该功能等价于groupingsets((name,class),(name),(class),())

```
postgres=#select dep,product,sum(num) from t_grouping group by grouping sets((dep,product),(dep),
(product),())order by dep,product;
  dep  | product | sum
---------+---------+-----
 业务1部 | 电脑    |  80
 业务1部 | 手机    | 160
 业务1部 |         | 240
 业务2部 | 电脑    | 120
 业务2部 | 手机    |  50
 业务2部 |         | 170
 业务3部 | 电脑    |  80
 业务3部 | 手机    | 160
 业务3部 |         | 240
         | 电脑    | 280
         | 手机    | 370
         |         | 650
```

PREPARE预备使用

创建一个预备

创建一个预备

```
postgres=# create table t1(f1 int,f2 int);
CREATE TABLE
postgres=# insert into t1 values(1,1),(2,2);
COPY 2
postgres=# PREPARE usrrptplan (int) AS SELECT * FROM t1 WHERE f1=$1;
PREPARE
postgres=# EXECUTE usrrptplan(1);
 f1 | f2
----+----
  1 |  1
(1 row)
```

释放一个预备

```
postgres=# DEALLOCATE usrrptplan;
DEALLOCATE
--释放后再执行就会出错
postgres=# EXECUTE usrrptplan(1);
ERROR:  prepared statement "usrrptplan" does not exist
```

关于INSERT

insert用于向一张表中插入数据，期望插入的数据可以是一条，多条或者是一个select查询的结果集。
使用INSERT INTO...命令插入数据
示例说明：

1. 使用default关键字，即值为建表时指定的默认值方式

```
insert into tbase(id,nickname) values(default,'TDSQL PG default');
```

2. 子查询插入

```
insert into tbase(id,nickname) values(1,(select relname from pg_class limit 1));
```

3. 返回插入插入数据

```
insert into tbase(nickname) values('TDSQL PG好') returning *;
```

4. insert update使用

```
insert into tbase values(1,'pgxz') ON CONFLICT (id)  DO UPDATE SET nc = 'TDSQL PG';
```

INSERT使用场景

插入单条记录

- 指定所有字段。

```
postgres=# insert into tbase(id,nickname) values(1,'hello TDSQL PG');
INSERT 0 1
```

- 指定某些字段，不指的有默认值的插入时带上默认值。

```
postgres=# insert into tbase(nickname) values('TDSQL PG好');
INSERT 0 1
```

- 不指定字段则默认为所有字段与建表时一致。

```
postgres=# insert into tbase values(nextval('t_id_seq'::regclass),'TDSQL PG好');
INSERT 0 1
```

- 字段顺序可以任意排列。

```
postgres=# insert into tbase(nickname,id) values('TDSQL PG swap',5);
INSERT 0 1
```

- 使用default关键字，即值为建表时指定的默认值方式。

```
postgres=#  insert into tbase(id,nickname) values(default,'TDSQL PG default');
INSERT 0 1
```

- 上面五次插入记录后产生的数据。

```
postgres=# select * from tbase;
 id |   nickname
----+--------------
  1 | hello TDSQL PG
  2 | TDSQL PG好
  5 | TDSQL PG swap
  3 | TDSQL PG好
  4 | TDSQL PG default
(5 rows)
```

插入多数记录

```
postgres=# insert into tbase(id,nickname) values(1,'hello TDSQL PG'),(2,'TDSQL PG好');
INSERT 0 2
postgres=# select * from tbase;
 id |  nickname
----+-------------
  1 | hello TDSQL PG
  2 | TDSQL PG好
(2 rows)
```

使用子查询插入数据

```
postgres=# insert into tbase(id,nickname) values(1,(select relname from pg_class limit 1));
INSERT 0 1
postgres=# select * from tbase;
 id |   nickname
----+--------------
  1 | pg_statistic
(1 row)
```

从另外一个表取数据进行批量插入

```
postgres=# insert into tbase(nickname) select relname from pg_class limit 3;
INSERT 0 3
postgres=# select * from tbase;
 id |   nickname
----+---------------
  5 | pg_type
  6 | pg_toast_2619
  4 | pg_statistic
(3 rows)
```

大批量的生成数据

```
postgres=# insert into tbase select t,md5(random()::text) from generate_series(1,10000) as t;
INSERT 0 10000
postgres=# select count(1) from tbase;
 count
-------
 10000
(1 row)
```

返回插入数据，轻松获取插入记录的serial值

```
postgres=# insert into tbase(nickname) values('TDSQL PG好') returning *;
 id | nickname
----+-----------
  7 | TDSQL PG好
(1 row)
```

- 指定返回的字段。

```
INSERT 0 1
postgres=# insert into tbase(nickname) values('hello TDSQL PG') returning id;
 id
----
  8
(1 row)
```

insert..update更新

- 使用ON CONFLICT。

```
postgres=# \d+ t
                        Table "public.t"
 Column |  Type   | Modifiers | Storage  | Stats target | Description
--------+---------+-----------+----------+--------------+-------------
 id     | integer |           | plain    |              |
 nc     | text    |           | extended |              |
Indexes:
    "t_id_udx" UNIQUE, btree (id)
Has OIDs: no
Distribute By SHARD(id)
        Location Nodes: ALL DATANODES
postgres=# select * from t;
 id |  nc
----+------
  1 | pgxz
(1 row)
postgres=# insert into t values(1,'pgxz') ON CONFLICT (id)  DO UPDATE SET nc = 'TDSQL PG';
INSERT 0 1
postgres=# select * from t;
 id |  nc
----+-------
  1 | TDSQL PG
(1 row)
```

insert all

```
postgres=# create table t5(f1 int,f2 int);create table t6(f1 int,f2 int);
CREATE TABLE
CREATE TABLE
postgres=# insert all into t5 values(f1,f2) into t6 values(f1,f2) select 1 as f1,1 as f2;
INSERT 0 2
postgres=# select * from t5;
 f1 | f2
----+----
  1 |  1
(1 row)
postgres=# select * from t6;
 f1 | f2
----+----
  1 |  1
(1 row)
```

关于UPDATE

update用于更新表数据，可以更新一个字段或者多个字段。
使用update...命令更新表
示例：
1）多表关联更新

```
update tbase set nickname ='Good TDSQL PG' from t_appoint_col where t_appoint_col.id=tbase.id;
```

2）返回更新的数据

```
update tbase set nickname = nickname where id = (random()*2)::integer returning *;
```

3）多列匹配更新

```
update tbase set ( nickname , age ) = ( 'TDSQL PG好' , (random()*2)::integer );
```

UPDATE使用场景

单表更新

```
postgres=# update tbase set nickname ='Hello TDSQL PG' where id=1;
UPDATE 1
```

- null条件的表达方法。

```
postgres=# update tbase set nickname = 'Good TDSQL PG' where nickname is null;
UPDATE 1
postgres=# select * from tbase;
 id |  nickname
----+-------------
  2 | TDSQL PG好
  1 | Hello TDSQL PG
  3 | Good TDSQL PG
(3 rows)
```

多表关联更新

```
postgres=# update tbase set nickname ='Good TDSQL PG' from t_appoint_col where t_appoint_col.id=
tbase.id;
UPDATE 1
postgres=# select * from tbase;
 id |  nickname
----+------------
  2 | TDSQL PG好
  1 | Good TDSQL PG
(2 rows)
```

返回更新的数据

```
postgres=# update tbase set nickname = nickname where id = (random()*2)::integer returning *;
 id | nickname
----+-----------
  2 | TDSQL PG好
(1 row)
```

上面的语句我们随机更新了一些数据，然后返回更新过的记录，returning机制大在的降低的应用的复杂度。

多列匹配更新

```
postgres=# update tbase set ( nickname , age ) = ( 'TDSQL PG好' , (random()*2)::integer );
UPDATE 2
postgres=# select * from tbase;
 id | nickname  | age
----+-----------+-----
  1 | TDSQL PG好 |   2
  2 | TDSQL PG好 |   0
(2 rows)
```

shard key不准许更新

```
postgres=# update tbase set id=8 where id=1;
ERROR:  Distribute column or partition column can't be updated in current version
```

关于DELETE

delete用于删除表数据，可以全部删除或者部分删除。
示例说明：
1）多表关联删除
使用using关键字，支持字段前表名

```
delete from tbase using t_appoint_col where tbase.id=t_appoint_col.id;
```

2）返回删除数据

```
delete from tbase returning *;
```

3. 支持省略from关键字

```
delete tbase where id=3;
```

DELETE使用场景

带条件删除

```
postgres=# select * from tbase;
 id |  nickname
----+-------------
  2 | TDSQL PG好
  1 | Hello TDSQL PG
  3 |
  4 | TDSQL PG good
(4 rows)
postgres=# delete from tbase where id=4;
DELETE 1
#null条件的表达方式
postgres=# delete from tbase where nickname is null;
DELETE 1
postgres=# select * from tbase;
 id |  nickname
----+-------------
  2 | TDSQL PG好
  1 | Hello TDSQL PG
(2 rows)
```

多表关联删除数据

```
postgres=# select * from tbase;
 id |  nickname
----+-------------
  2 | TDSQL PG好
  1 | Hello TDSQL PG
(2 rows)

postgres=# set prefer_olap to on;
SET
postgres=# delete from tbase using t_appoint_col where tbase.id=t_appoint_col.id;
DELETE 1
postgres=# select * from tbase;
 id | nickname
----+-----------
  2 | TDSQL PG好
(1 row)
```

返回删除数据

```
postgres=# delete from tbase returning *;
 id | nickname
----+-----------
  2 | TDSQL PG好
(1 row)
```

returning机制大在的降低的应用的复杂度。

删除所有数据

```
postgres=# insert into tbase select t,random()::text from generate_series(1,100000) as t;
postgres=# \timing
Timing is on.
postgres=# delete from tbase ;
DELETE 100000
Time: 100.808 ms
#使用truncate方法是全表删除更高效的方法
postgres=# insert into tbase select t,random()::text from generate_series(1,100000) as t;
INSERT 0 100000
Time: 13178.429 ms
postgres=# truncate table tbase;
TRUNCATE TABLE
Time: 24.242 ms
```

部分DML操作示例

select支持别名不用as修饰

```
postgres=# select * from student as st where st.f1=1;
 f1 | f2
----+----
  1 |  2
(1 row)
postgres=# select * from student st where st.f1=1;
 f1 | f2
----+----
  1 |  2
```

update支持别名

```
postgres=# create table student(f1 int,f2 int);
CREATE TABLE
postgres=# insert into student  values(1,1);
INSERT 0 1
postgres=# update student st set st.f2=2 where f1=1;
UPDATE 1
postgres=# select * from student;
 f1 | f2
----+----
  1 |  2
```

关于事务
事务控制

使用下面的命令来控制事务：
BEGIN：开始一个事务。
SAVEPOINT：事务保存点。一个事务可以有多个保存点。
COMMIT：事务提交。
ROLLBACK：事务回滚。可回滚整个当前事务，或与savepoint一起使用，回滚到保存点。
事务控制命令只与 INSERT、UPDATE 和 DELETE 一起使用。他们不能在创建表或删除表时使用，因为这些操作在数据库中是自动提交的。
事务可以使用 BEGINTRANSACTION 命令或简单的 BEGIN 命令来启动。此类事务通常会持续执行下去，直到遇到下一个 COMMIT 或 ROLLBACK 命令。不过在数据库关闭或发生错误时，事务处理也会回滚。以下是启动一个事务的简单语法：
BEGIN; 或者 BEGINTRANSACTION;

事务提交

事务提交用于保存对数据库的更改命令。

命令如下：

COMMIT; 或者COMMIT TRANSACTION;

例：

```
BEGIN;
delete from tbase where id=5;
COMMIT;
```

⚠在COMMIT;执行之前tbase表中id为5的记录会被另一连接进程查询到。

事务回滚

事务回滚用于撤消尚未保存到数据库的命令。
命令语法如下：
ROLLBACK; 或者 ROLLBACK to savepoint_name;
例1：

```
BEGIN;
delete from tbase where id in (3,4);
select * from tbase;
ROLLBACK;
select * from tbase;
```

⚠在ROLLBACK;执行之前同一连接进程查询不到id为3和4的数据。回滚执行之后，数据可查询到。
例2:

```
BEGIN;
delete from tbase where id =3;
select * from tbase;
savepoint A;
delete from tbase where id =4;
select * from tbase;
ROLLBACK to A;
COMMIT;
```

⚠在ROLLBACK to A;执行之前同一进程查询不到id为3和4的数据。回滚到保存点之后，id为3数据可查询到。
ROLLBACK to savepoint_name;不会结束事务，需使用COMMIT;提交事务。

事务一致性

事务一致性指事务自始至终读取的数据都是一致的。
示例：

```
#session 1
create table t_repeatable_read (id int,mc text);
insert into t_repeatable_read values(1,'TDSQL PG');
begin isolation level repeatable read ;
select * from t_repeatable_read ;
#session 2
insert into t_repeatable_read values(1,'TDSQL PG');
select * from t_repeatable_read;
#session 1
select * from t_repeatable_read ;
```

⚠️session1查询到不到session2插入的数据，事务中读一致。

事务使用场景

开始一个事务

```
postgres=# begin;
BEGIN
#或者
postgres=# begin TRANSACTION ;
BEGIN
#也可以定义事务的级别
postgres=# begin transaction isolation level read committed ;
BEGIN
```

提交事务

- 进程#1访问 。

```
postgres=# begin;
BEGIN
postgres=# delete from tbase where id=5;
DELETE 1
postgres=#
postgres=# select * from tbase order by id;
 id|   nickname
----+---------------
  1 |hello TDSQL PG
  2 |TDSQL PG好
  3 |TDSQL PG好
  4 |TDSQL PG default
```

TDSQL PG也是完全支持ACID特性，没提交前开启另一个连接查询，你会看到是5条记录，这是TDSQL PG隔离性和多版本视图的实现，如下所示：

- 进程#2访问。

```
postgres=# select * from tbase order by id;
 id|   nickname
----+---------------
  1 |hello TDSQL PG
  2 |TDSQL PG好
  3 |TDSQL PG好
  4 |TDSQL PG default
  5 |TDSQL PG swap
(5 rows)
```

- 进程#1提交数据。

```
postgres=# commit;
COMMIT
postgres=#
```

- 进程#2再查询数据，这时能看到已经提交的数据了，这个级别叫“已读提交”。

```
postgres=# select * from tbase order by id;
 id|   nickname
```

```
----+---------------
  1 |hello TDSQL PG
  2 |TDSQL PG好
  3 |TDSQL PG好
  4 |TDSQL PG default
(4 rows)
```

回滚事务

```
postgres=# begin;
BEGIN
postgres=# delete from tbase where id in (3,4);
DELETE 2
postgres=# select * from tbase;
 id |  nickname
----+-------------
  1 | hello TDSQL PG
  2 | TDSQL PG好
(2 rows)
postgres=# rollback;
ROLLBACK
#Rollback后数据又回来了
postgres=# select * from tbase;
 id |   nickname
----+---------------
  1 | hello TDSQL PG
  2 | TDSQL PG好
  3 | TDSQL PG好
  4 | TDSQL PG default
(4 rows)
```

事务读一致性REPEATABLE READ

这种事务级别表示事务自始至终读取的数据都是一致的，如下所示：

```
#session1
postgres=# create table t_repeatable_read (id int,mc text);
CREATE TABLE
postgres=# insert into t_repeatable_read values(1,'TDSQL PG');
INSERT 0 1
postgres=# begin isolation level repeatable read ;
BEGIN
postgres=# select * from t_repeatable_read ;
 id |  mc
----+-------
  1 | TDSQL PG
(1 row)

#session2
postgres=# insert into t_repeatable_read values(1,'pgxz');
INSERT 0 1
postgres=# select * from t_repeatable_read;
 id |  mc
----+-------
  1 | TDSQL PG
  1 | pgxz
(2 rows)
#session1
postgres=# select * from t_repeatable_read ;
 id |  mc
----+-------
  1 | TDSQL PG
(1 row)
postgres=#
```

行锁在事务中的运用

数据表准备

```
postgres=# create table t_row_lock(id int,mc text,primary key (id));
CREATE TABLE
postgres=#
postgres=# insert into t_row_lock values(1,'TDSQL PG'),(2,'pgxz');
INSERT 0 2
postgres=# select * from t_row_lock;
 id |  mc
----+-------
  1 | TDSQL PG
  2 | pgxz
(2 rows)
```

直接update获取

```
#session1
postgres=# begin;
BEGIN
postgres=# set lock_timeout to 1;
SET
postgres=# update t_row_lock set mc='postgres' where mc='pgxz';
UPDATE 1
postgres=#
#session2
postgres=# begin;
BEGIN
postgres=# set lock_timeout to 1;
SET
postgres=#  update t_row_lock set mc='postgresql' where mc='TDSQL PG';
UPDATE 1
postgres=#
```

上面session1与session2分别持有mc=pgxz行和mc=TDSQL PG的行锁。

select...for update获取

```
#session1
postgres=#  begin;
BEGIN
postgres=# set lock_timeout to 1;
SET
postgres=# select * from t_row_lock where mc='pgxz' for update;
 id |  mc
----+------
  2 | pgxz
(1 row)
#session2
postgres=# begin;
BEGIN
postgres=# set lock_timeout to 1;
SET
postgres=# select * from t_row_lock where mc='TDSQL PG' for update;
 id |  mc
----+------
  2 | pgxz
(1 row)
```

上面session1与session2分别持有mc=pgxz行和mc=TDSQL PG的行锁。

与mysql获取行级锁的区别

```
mysql> select version();
+-----------+
| version() |
+-----------+
| 5.6.36    |
+-----------+
1 row in set (0.00 sec)
#sessin1
mysql> begin;
Query OK, 0 rows affected (0.00 sec)
mysql> select * from t_row_lock where mc='pgxz' for update;
+----+------+
| id | mc   |
+----+------+
|  2 | pgxz |
+----+------+
1 row in set (0.00 sec)
#session2
mysql> select * from t_row_lock where mc='TDSQL PG' for update;
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
mysql>
#这是因为mysql要使用行级锁需要有索引来配合使用，如下所示,使用id主键来获取行锁
#session1
mysql> begin;
Query OK, 0 rows affected (0.00 sec)
mysql> select * from t_row_lock where id=1 for update;
+----+-------+
| id | mc    |
+----+-------+
|  1 | TDSQL PG |
+----+-------+
1 row in set (0.00 sec)
#session2
mysql> begin;
Query OK, 0 rows affected (0.00 sec)
mysql> select * from t_row_lock where id=2 for update;
+----+------+
| id | mc   |
+----+------+
|  2 | pgxz |
+----+------+
1 row in set (0.00 sec)
```

执行计划
查看执行计划

TDSQL PostgreSQL会为每一个收到的查询产生一个查询计划，可以使用EXPLAIN命令察看生成的查询计划。
EXPLAIN语法如下：
EXPLAIN [(参数 [, ...])] SQL语句
EXPLAIN [ANALYZE] [VERBOSE] SQL语句
这里 参数可以是：
ANALYZE [boolean]
VERBOSE [boolean]
COSTS [boolean]
SETTINGS [boolean]
BUFFERS [boolean]
WAL [boolean]
TIMING [boolean]
SUMMARY [boolean]
FORMAT { TEXT | XML | JSON | YAML }

参数说明：
ANALYZE
执行命令并且显示实际的运行时间和其他统计信息。这个参数默认被设置为FALSE。

VERBOSE
显示关于执行计划的附加信息。具体包括计划树中每个结点的输出列列表、模式限定的表和函数名称、在表达式中使用范围表别名标记变量，并输出显示统计信息的每个触发器的名称。这个参数默认被设置为FALSE。

COSTS
每一个计划结点的估计启动和总代价，以及估计的行数和每行的宽度。这个参数默认被设置为TRUE。

SETTINGS
有关配置参数的信息。具体包括影响查询计划的选项，其值与内置默认值不同。此参数默认为FALSE。

BUFFERS
缓冲区使用的信息。具体包括共享块命中、读取、标记为脏和写入的次数、本地块命中、读取、标记为脏和写入的次数、以及临时块读取和写入的次数。一次命中表示避免了一次读取，因为需要的块已经在缓存中找到了。共享块包含着来自于常规表和索引的数据，本地块包含着来自于临时表和索引的数据，而临时块包含着在排序、哈希、物化计划结点和类似情况中使用的短期工作数据。脏块的数量表示被这个查询改变的之前未被修改块的数量，而写入块的数量表示这个后台在查询处理期间从缓存中替换出去的脏块的数量。为一个较高层结点显示的块数包括它的所有子结点所用到的块数。在文本格式中，只会打印非零值。只有当ANALYZE也被启用时，这个参数才能使用。它的默认被设置为FALSE。

WAL

有关WAL记录生成的信息。 具体包括记录数、整页图像数（fpi）和生成的WAL字节数量。 在文本格式中，仅打印非零值。 此参数只能在同时启用ANALYZE时使用。 它默认为FALSE。

TIMING

在输出中包括实际启动时间以及在每个结点中花掉的时间。反复读取系统时钟的负荷在某些系统上会显著地拖慢查询，因此在只需要实际的行计数而不是实际时间时，把这个参数设置为FALSE可能会有用。即便用这个选项关闭结点层的计时，整个语句的运行时间也总是会被度量。只有当ANALYZE也被启用时，这个参数才能使用。它的默认被设置为TRUE。

SUMMARY

在查询计划之后包含摘要信息（例如，总计的时间信息）。当使用ANALYZE 时默认包含摘要信息。不使用ANALYZE时可以使用此选项仅启用摘要信息。 使用EXPLAIN EXECUTE中的计划时间包括从缓存中获取计划所需的时间 以及重新计划所需的时间（如有必要）。

FORMAT

指定输出格式，可以是 TEXT、XML、JSON 或者 YAML。非文本输出包含和文本输出格式相同的信息。这个参数默认被设置为TEXT。

> 说明：
>
> 为了保证TDSQL PostgreSQL优化器在优化查询时能做出合理的决策，查询中用到所有表的pg_statistic数据应该能保持为最新。通常这个工作会由autovacuum daemon自动完成。但是如果一个表最近发生大量的数据改变，那么需要手动做一次ANALYZE而不是等待 autovacuum 来修改记录。

解读执行计划

使用EXPLAIN命令可以查看TDSQL PostgreSQL优化器为每个查询生成的具体执行计划。EXPLAIN的输出是一个节点树。 其中每一行对应一个数据库执行算子， 显示算子的节点类型和优化器为执行这个节点预估的开销。
通过以下示例来解读执行计划

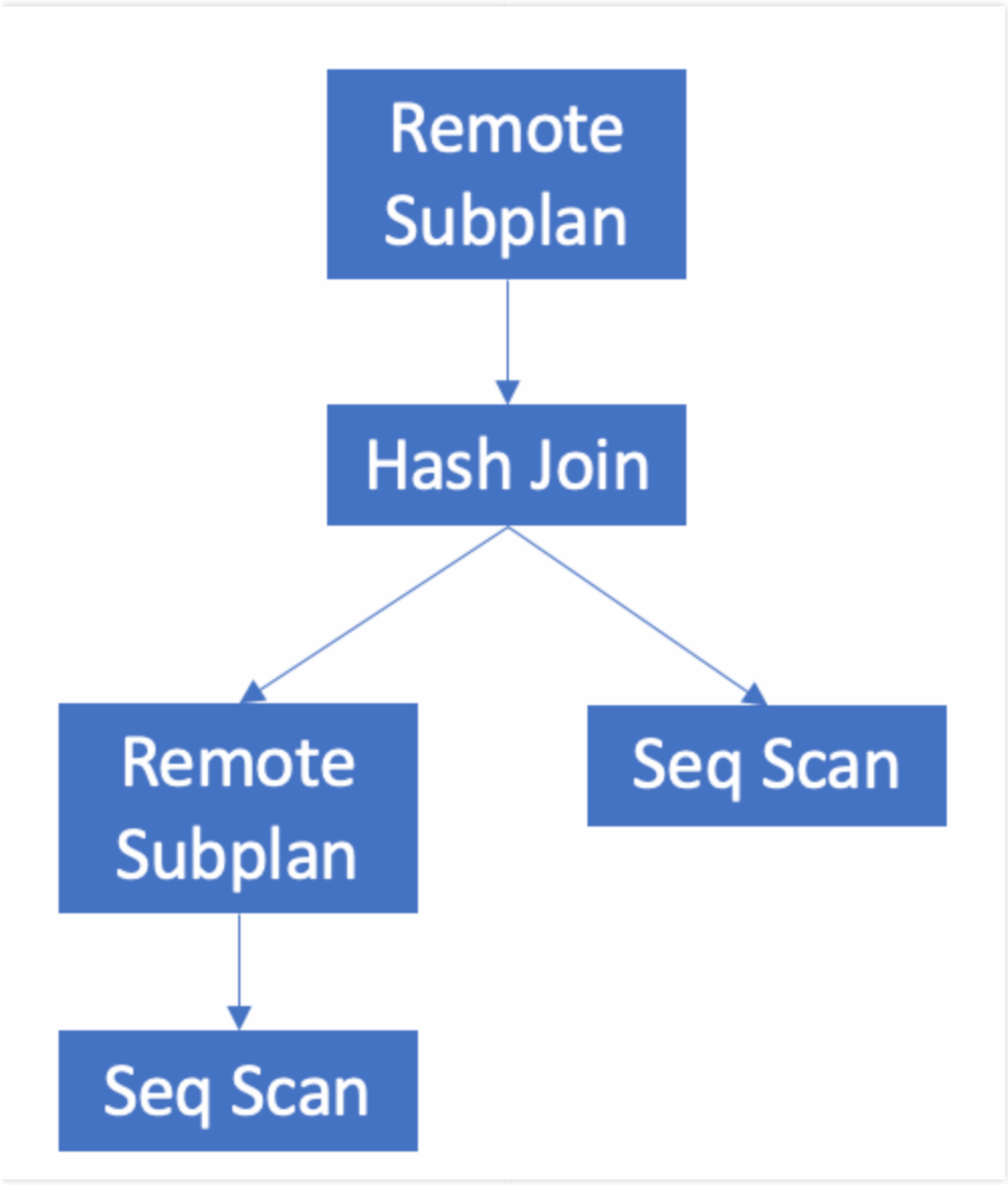
```
---涉及到两个表的DDL
create table t3(f1 int, f2 int) distribute by hash(f1);
create table t4(f1 int, f2 int) distrinute by hash(f1);
```

> 说明：
>
> distribute by hash 表示对t3，t4数据按f1 哈希值进行分布到不同节点dn上。

查看关联SQL的执行计划 explain select * from t3, t4 where t3.f1=t4.f2; 可以得到如下输出结果

```
                          QUERY PLAN
------------------------------------------------------------------------------
Remote Subquery Scan on all (datanode_1,datanode_2) (cost=120.19..222.56 rows=4556  width=16)
  -> Hash Join (cost=120.19..222.56 rows=4556 width=16)
     Hash Cond:(t4.f2 = t3.f1)
     -> Remote Subquery Scan on all (datanode_1,datanode_2) (cost=100.00..120.53  rows=675 width=
8)
        Distribute results by H: f2
        -> Seq Scan on t4 (cost=0.00..11.75 rows=675 width=8)
     -> Hash (cost=11.75..11.75 rows=675 width=8)
        -> Seq Scan on t3 (cost=0.00..11.75 rows=675 width=8)
(8rows)
```

这个执行计划可以简化成以下执行计划树：



执行计划树里的每个节点代笔一个执行算子（在"执行计划节点类型"中详述）。每个节点的EXPLAIN输出（ 如下所示）格式为 算子名称（算子开销预估）。

-> Hash Join(cost=120.19..222.56 rows=4556 width=16)

Hash Cond:(t4.f2 = t3.f1)

括号中的数字从左到右依次是:

- 启动代价: 这是该算子在读取第一条元组前的开销预估。比方说索引扫描算子的启动代价就是读取目标表的索引页， 读取到第一个元组的开销。以上示例中的哈希关联(Hash Join)的启动代价是120.19。
- 算子总代价：这个改算子从执行到结束的总代价。以上示例中的哈希关联(Hash Join)的总代价是222.56。
- 算子输出的总行数。 以上示例中的哈希关联(Hash Join)的预估输出行数是4556。
- 算子输出行的行宽（字节数）。 以上示例中的哈希关联(Hash Join)的预估行宽是16字节。

有些算子在EXPLAIN的输出里除了上述的基本通用信息外， 还有算子的一些特定信息。 比方说上面的哈希关联算子的EXAPLIN输出还打印了关联条件。

在SQL执行计划输出结果中， 通常

- 最底层节点是表扫描节点， 扫描节点返回表中的原数据行。 不同的表有不同的扫描节点类型:顺序扫描，索引扫描和位图索引扫描。最底层的扫描节点也可能是也有非表列源，如VALUES子句并设置FROM返回，它们有自己的扫描类型。
- 如果查询需要关联，聚合，排序或其他操作，会在扫描节点之上增加节点执行这些操作。这些操作通常都有多种方法，因此在这些位置也有可能出现不同的执行节点类型。
- 分布式计划中通常会有个特殊的算子（“Remote Subquery Scan”）。这个算子实现了分布式架构的核心数据shuffle的功能。 这个算子有几种不同的形态， 分别对应分布式架构下不同的数据shuffle 功能:
  （1）数据收集形态 - 作用是CN从DN收集数据。
  （2）数据重分布形态 - 作用是CN根据选定的列按照特定的规则把数据重分布到所有的DN。
  数据重分布的规则有:
  - Redistribute by Hash - 把指定的列按照hash redistribute。 这种情况下， 执行计划的输出里会在“Remote Subquery Scan”的那一行的下面一行打印 - “Distribute results by H： 列名”。 如下所示。 其中"Remote Subquery Scan on all" 后面括号里显示的是该算子在哪些数据节点上执行。
    -> Remote Subquery Scan on all (datanode_1,datanode_2) (cost=100.00..120.53 rows=675 width=8)
      Distribute results by H: f2
      -> Seq Scan on t4 (cost=0.00..11.75 rows=675 width=8)
  - Redistribute by Shard - 把指定的列按照shard redistribute。 这种情况下， 执行计划的输出里会在“Remote Subquery Scan”的那一行的下面一行打印 - “Distribute results by S： 列名”
  - Broadcast - 把指定的表广播到所有的数据节点上。 这种情况下， 执行计划输出里在“Remote Subquery Scan”的那一行下面没有额外的重分布信息。
- 在分布式架构中， 当SQL可以完全下推到各个DN上计算，中间计算结果不需要重新分布到其他DN上的时候，执行计划中会有个分布式快速执行算子 - “Remote Fast Query Execution”， 如下示例。在下面这个例子里，两表关联的列都是分布健。这种情况下，各个DN上各自独立执行SQL，CN只需要把DN返回的结果直接返回即可，无需额外的计算。

postgres@coord1=# explain select * from t3, t4 where t3.f1=t4.f1;

```
                    QUERY PLAN
-------------------------------------------------------------------
Remote Fast Query Execution (cost=0.00..0.00 rows=0 width=0)
 Node/s:datanode_1, datanode_2
 ->Merge Join (cost=187.38..330.81 rows=9112 width=16)
   Merge Cond:(t3.f1=t4.f1)
   -> Sort (cost=93.69..97.07 rows=1350 width=8)
      Sort Key: t3.f1
      -> Seq Scan on t3 (cost=0.00..23.50 rows=1350 width=8)
   -> Sort (cost=93.69..97.07 rows=1350 width=8)
      Sort Key: t4.f1
      -> Seq Scan on t4 (cost=0.00..23.50 rows=1350 width=8)
(10rows)
```

在了解了这些特点之后，我们会对EXPLAIN 命令的输出有一个整体结构的概念。其实EXPLAIN 输出的就是一个用户可视化的查询计划树，可以告诉我们执行了哪些节点（操作），并且每个节点（操作）的代价预估是什么样的。

执行计划节点类型

执行计划节点分为扫描算子、控制算子、物化算子、链接算子等。

扫描算子

扫描节点负责从底层数据来源抽取数据，数据来源可能是来自文件系统，也可能来自网络（分布式查询）。一般而言扫描节点都位于执行树的叶子节点，作为执行树PlanTree的数据输入来源。
关键特征：输入数据、叶子节点、表达式过滤

算子类型	EXPLAIN 显示	含义	出现场景描述
SeqScan	Seq Scan	顺序扫描行存数据	用于扫描行存物理表（没有索引辅助的情况下）
EstoreSeqScan	Estore Seq Scan	扫描列存数据	基础的列存引擎扫描算子， 用于扫描列存物理表（没有索引的情况下）
VtsCacheSeqScan	VtsCache Seq Scan	向量化计算缓存结果集扫描	列存引擎下向量化计算缓存
BitmapIndexScan BitmapHeapScan	Bitmap Index Scan Bitmap Heap Scan	利用Bitmap获取元组	BitmapIndexScan 一次性将满足条件的索引项全部取出，然后交给bitmapHeapScan节点，并在内存中进行排序, 根据取出的索引项访问表数据。
TidScan	Tid Scan	通过Tid获取元组	通过对表的ctid字段进行过滤和查找
SampleScan	Sample Scan	数据抽样	用于数据抽样， SELECT … FROM table_name TABLESAMPLE sampling_method
IndexScan	Index Scan	索引扫描	利用索引进行快速定位符合查询条件的元组
IndexOnlyScan	Index Only Scan	直接从索引返回元组	与IndexScan的差别是无需再次访问基表
SubqueryScan	Subquery Scan	子查询扫描	已一个子查询的结果集作为当前的输入
FunctionScan	Function Scan	函数扫描	将函数的结果集看成一个结果集， 进行后续的计算， 比如 FROM function_name

ValuesScan	Values Scan	扫描Values列表	对 values（ ）子句的元组集合进行扫描
TableFuncScan	Table Function Scan	处理TableFunc相关的扫描	xml table/函数
CteScan	CTE Scan	扫描Common Table Expression	将CTE的输出看成一个集合，进行后续的关系运算， 比如 with子语句定义的CTE子查询
NamedTuplestoreScan	Named Tuplestore Scan	用于某些命名的结果集的扫描	
WorkTableScan	Worktable Scan	扫描中间结果集	扫描查询过程中spillout的结果集
ForeignScan	Foreign Scan	外部表扫描	扫描基于外部数据源的外部表（FDW）
CustomScan	Custom Scan	自定义扫描	

控制算子

控制算子一般不映射代数运算符，通常是为了执行器完成一些特殊的流程引入的算子。

关键特征：用于控制数据流程。

算子类型	EXPLAIN 显示	含义	出现场景描述
Result	Row Result	行存储结果	处理仅需要一次计算的条件表达式或insert中的value子句
VecResult	Vector Result	列存储结果	向量化结果
ModifyTable	取决于具体的操作，可能的显示 • Insert • Update • Delete	Insert/Update/Delete操作的算子	增，删， 改
Append	Append	多个关系集合的追加操作	UNION， UNION-ALL

MergeAppend	Merge Append	多个有序关系集合的追加操作	UNION, 继承表
RecursiveUnion	Recursive Union	执行recursive subquery	with recursive递归子查询
Limit	Limit	控制数据流的返回数据量	处理带limit的语句

物化算子

物化算子一般指算法要求，在做算子逻辑处理的时候，要求把下层的数据进行缓存处理，因为对于下层算子返回的数据量不可提前预知，因此需要在算法上考虑数据无法全部放置到内存的情况。
关键特征：需要扫描所有数据之后才返回。

算子类型	EXPLAIN 显示	含义	使用场景
Materialzie	Materialize	物化	缓存结果集以方便后续重复扫描
Sort	Sort	对下层数据进行排序	Order By, MergeJoin, SortAgg, MergeAppend, 配合Unique去重等
Group	Group	对下层已经排序的数据进行分组	处理group-by分组操作
Agg	取决于优化器选择的聚合策略，有多种可能的显示- • Aggregate • GroupAggregate • HashAggregate • MixedAggregate	对下层数据进行分组或者聚合；可操作有序或无序数据	count/sum/avg/max/min等聚合函数； distinct子句，union去重，group-by子句
Unique	Unique	对下层数据进行去重操作	Distinct. union去重
Hash	Hash	对下层数据进行缓存，存储到一个哈希表里	作为HashJoin算子的输入， 构造HashTable

SetOp	取决于优化器选择的策略， 有多种可能的显示- • SetOp • HashSetOp	对下层数据进行缓存，用于处理集合操作	intersect/intersect-all, except/except-all 等操作
WindowAgg	WindowAgg	窗口函数	包含窗口函数的语句
LockRows	LockRows	处理行级锁	SELECT..FOR SHARE/UPDATE

链接算子

这类算子是为了应对数据库中最常见的关联操作。

算子类型	EXPLAIN 显示	含义	场景描述
Nestloop	Nested Loop	循环嵌套链接操作	Inner, Left-outer, Semi-Join, Anti-Join
MergeJoin	Merge Join	归并链接操作	Inner, left-outer, right-outer, full-outer, semi-join, anti-join
HashJoin	Hash Join	哈希链接操作	Inner, left-outer, right-outer, full-outer, semi-join, anti-join

分布式特有的算子

还有一些分布式架构下特有的算子类型

算子类型	EXPLAIN 显示	含义	场景描述
RemoteSubPlan	Remote Subquery Scan	数据重分布	分布式下需要数据重分布情况
RemoteQuery	Remote Fast Query Execution	SQL下推算子	分布式下SQL可以完全下推到数据节点

分布式执行计划示例

下面是分布式架构下， 不同类型的关联（join)的执行计划的示例。

Local Join

```
--关联的两张表
create table t1 (
    c1 integer,
    c2 integer,
    c3 integer,
    c4 integer,
    c5 integer
) distribute by shard(c1);

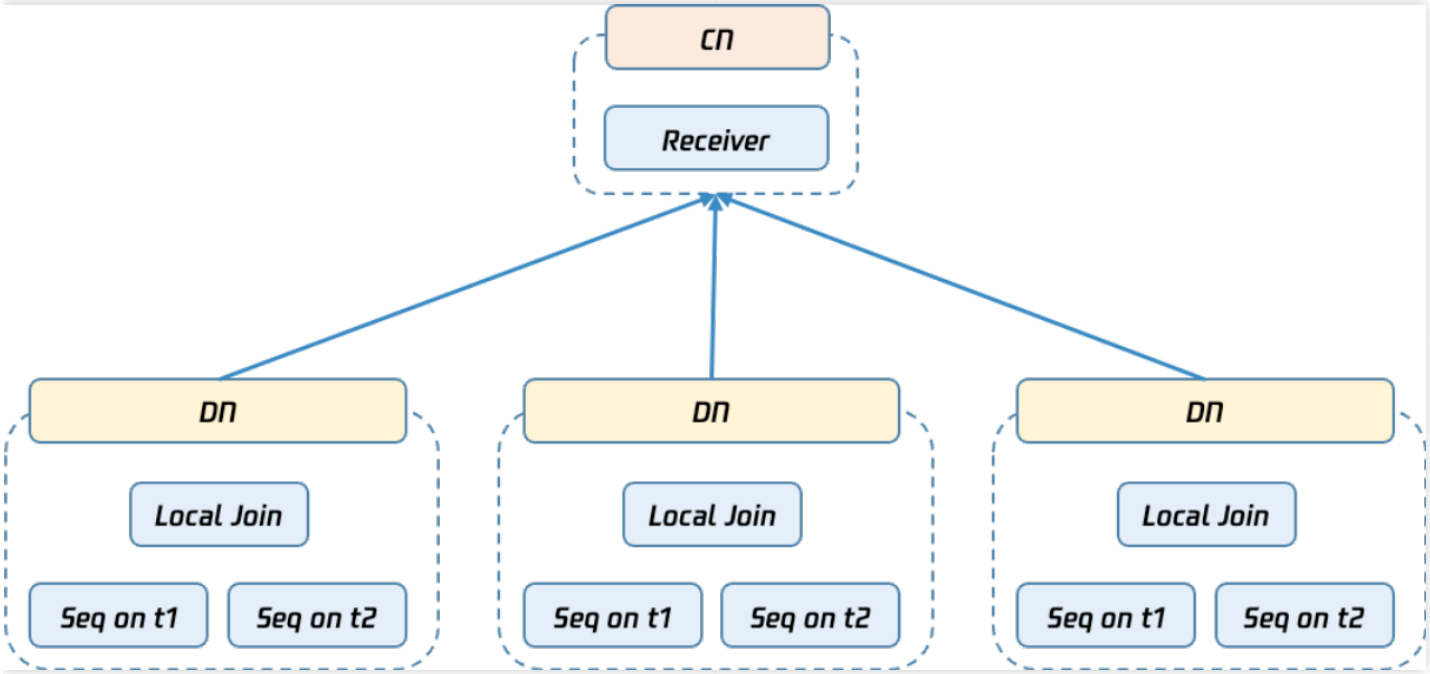
create table t2 (
    c1 integer,
    c2 integer,
    c3 integer,
    c4 integer,
    c5 integer
) distribute by shard(c1);

tbase=# explain select t1.c3, t2.c3 from t1, t2 where t1.c1=t2.c1;
```

结果为

```
                              QUERY PLAN
-----------------------------------------------------------------------------
 Remote Subquery Scan on all (datanode_1, datanode_2, datanode_3)  (cost=17.71..85.28 rows=3192
width=8)
   ->  Hash Join  (cost=17.71..85.28 rows=3192 width=8)
         Hash Cond: (t1.c1 = t2.c1)
         ->  Seq Scan on t1  (cost=0.00..10.65 rows=565 width=8)
         ->  Hash  (cost=10.65..10.65 rows=565 width=8)
               ->  Seq Scan on t2  (cost=0.00..10.65 rows=565 width=8)
```

EXPLAIN对应的执行计划流程



Distributed Join (1)

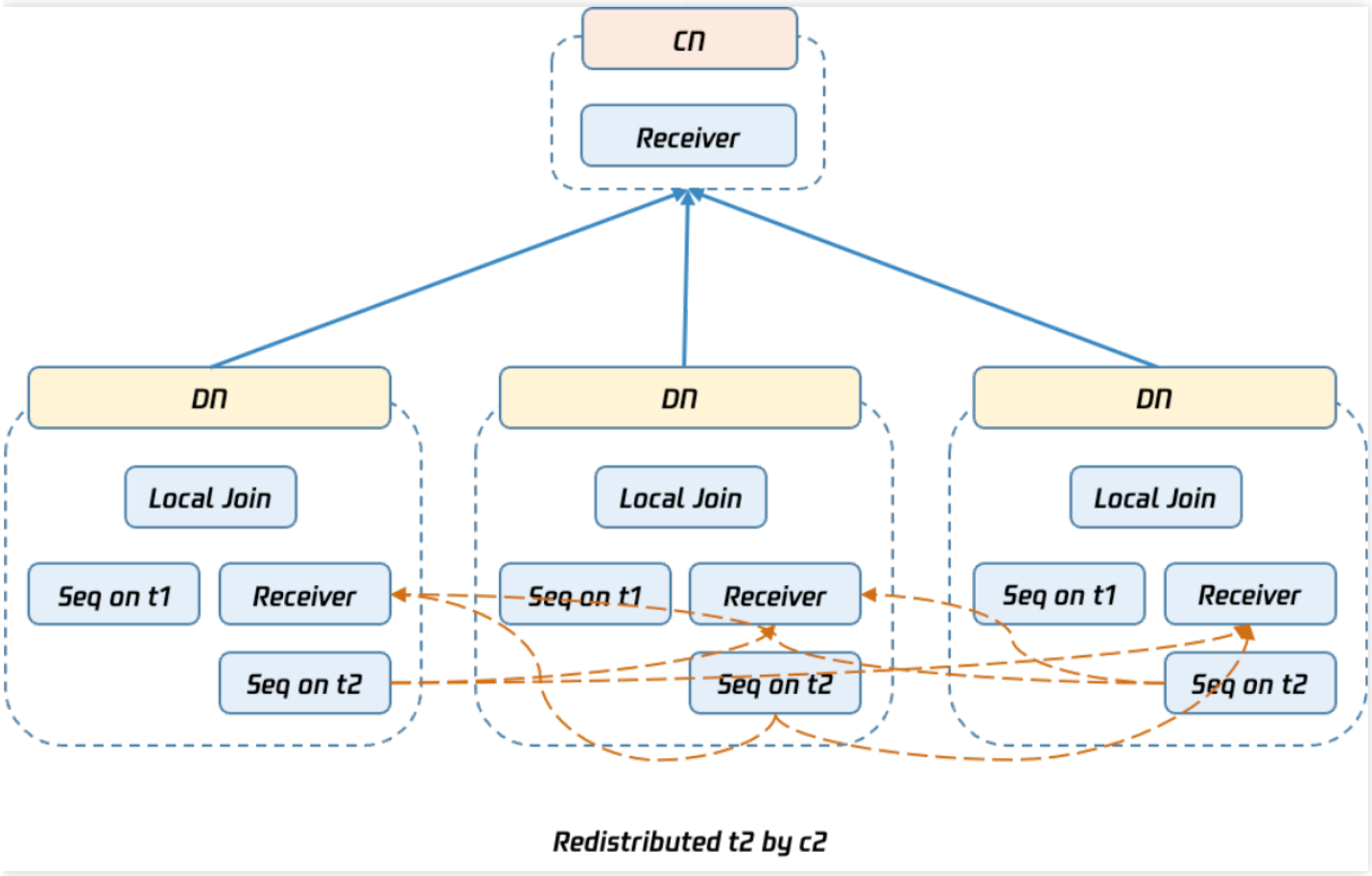
下面这个例子中只有Join的一端（t2表）需要重新分布

```
tbase=# explain select t1.c3, t2.c3 from t1, t2 where t1.c1=t2.c2;
```

结果为

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Remote Subquery Scan on all (datanode_1, datanode_2, datanode_3)  (cost=3285.00..7395.00 rows=1
00000 width=8)
   ->  Hash Join  (cost=3285.00..7395.00 rows=100000 width=8)
         Hash Cond: (t2.c2 = t1.c1)
         ->  Remote Subquery Scan on all (datanode_1, datanode_2, datanode_3)  (cost=100.00..3335.00 r
ows=100000 width=8)
               Distribute results by S: c2
               ->  Seq Scan on t2  (cost=0.00..1935.00 rows=100000 width=8)
         ->  Hash  (cost=1935.00..1935.00 rows=100000 width=8)
               ->  Seq Scan on t1  (cost=0.00..1935.00 rows=100000 width=8)
```

EXPLAIN对应的执行流程： 其中t2表的扫描结果会按照c2列值重分布。



Distributed Join（2）

下面这个例子中Join的两个表（t1和t2）都需要重新分布， 因为关联条件都不是分布列。

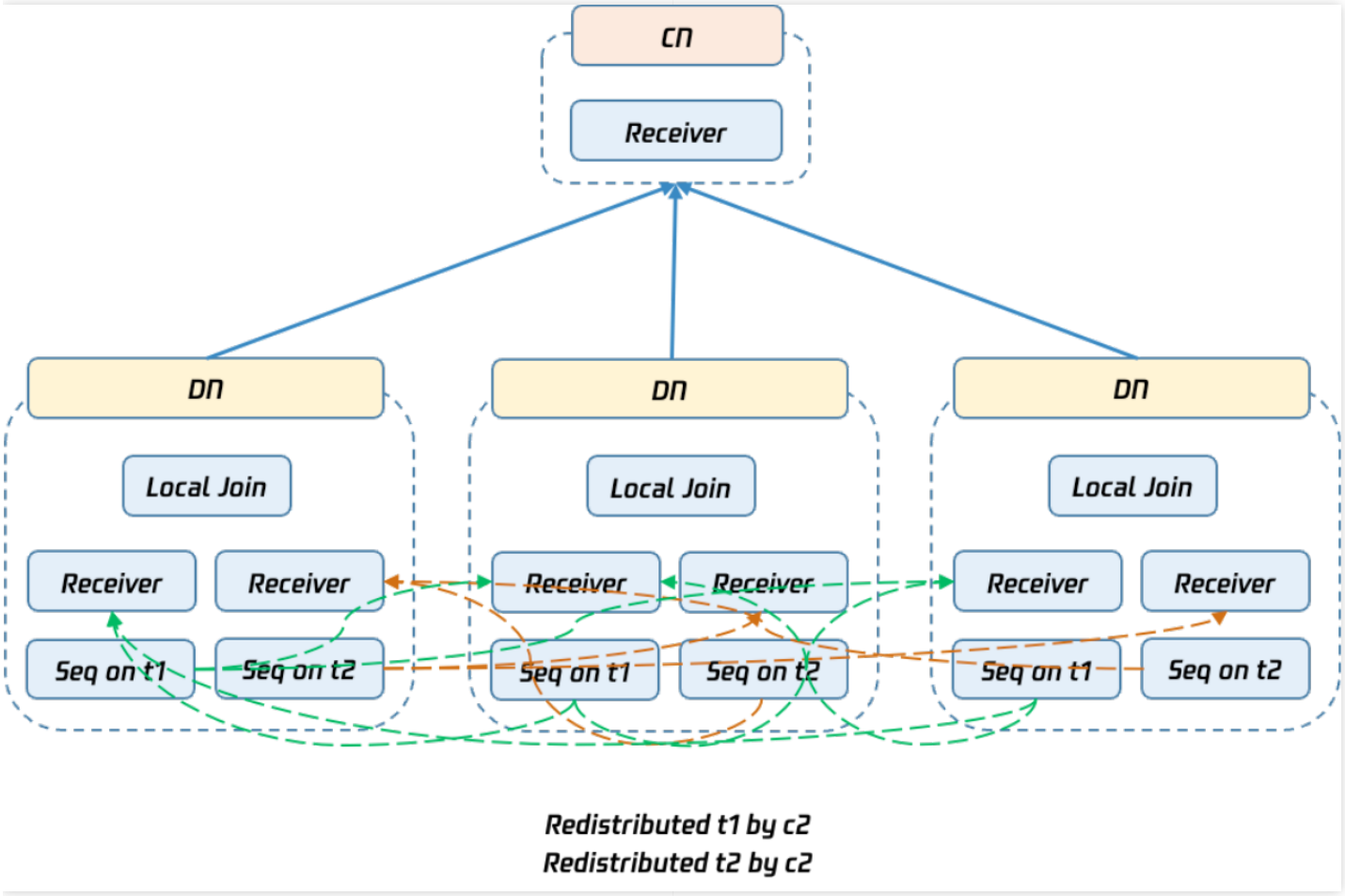
```
tbase =# explain select t1.c3, t2.c3 from t1, t2 where t1.c2=t2.c2;
```

结果为

```
                              QUERY PLAN
-----------------------------------------------------------------------------
 Remote Subquery Scan on all (datanode_1, datanode_2, datanode_3)  (cost=4685.00..8795.00 rows=1
00000 width=8)
   ->  Hash Join  (cost=4685.00..8795.00 rows=100000 width=8)
         Hash Cond: (t1.c2 = t2.c2)
         ->  Remote Subquery Scan on all (datanode_1, datanode_2, datanode_3)  (cost=100.00..3335.00 r
ows=100000 width=8)
               Distribute results by S: c2
               ->  Seq Scan on t1  (cost=0.00..1935.00 rows=100000 width=8)
         ->  Hash  (cost=3335.00..3335.00 rows=100000 width=8)
```

```
        ->  Remote Subquery Scan on all (datanode_1, datanode_2, datanode_3)  (cost=100.00..333
5.00 rows=100000 width=8)
              Distribute results by S: c2
              ->  Seq Scan on t2  (cost=0.00..1935.00 rows=100000 width=8)
```

EXPLAIN对应的执行流程： 其中t1表和t2表的扫描结果会按照各自c2列值重分布。



Broadcast

下面这个例子中，当表t1是大表， t2是小表的时候， 虽然两张表的关联条件都不是分布列， 优化器可能选择broadcast join, 把小表的扫描结果发送到各个数据节点。

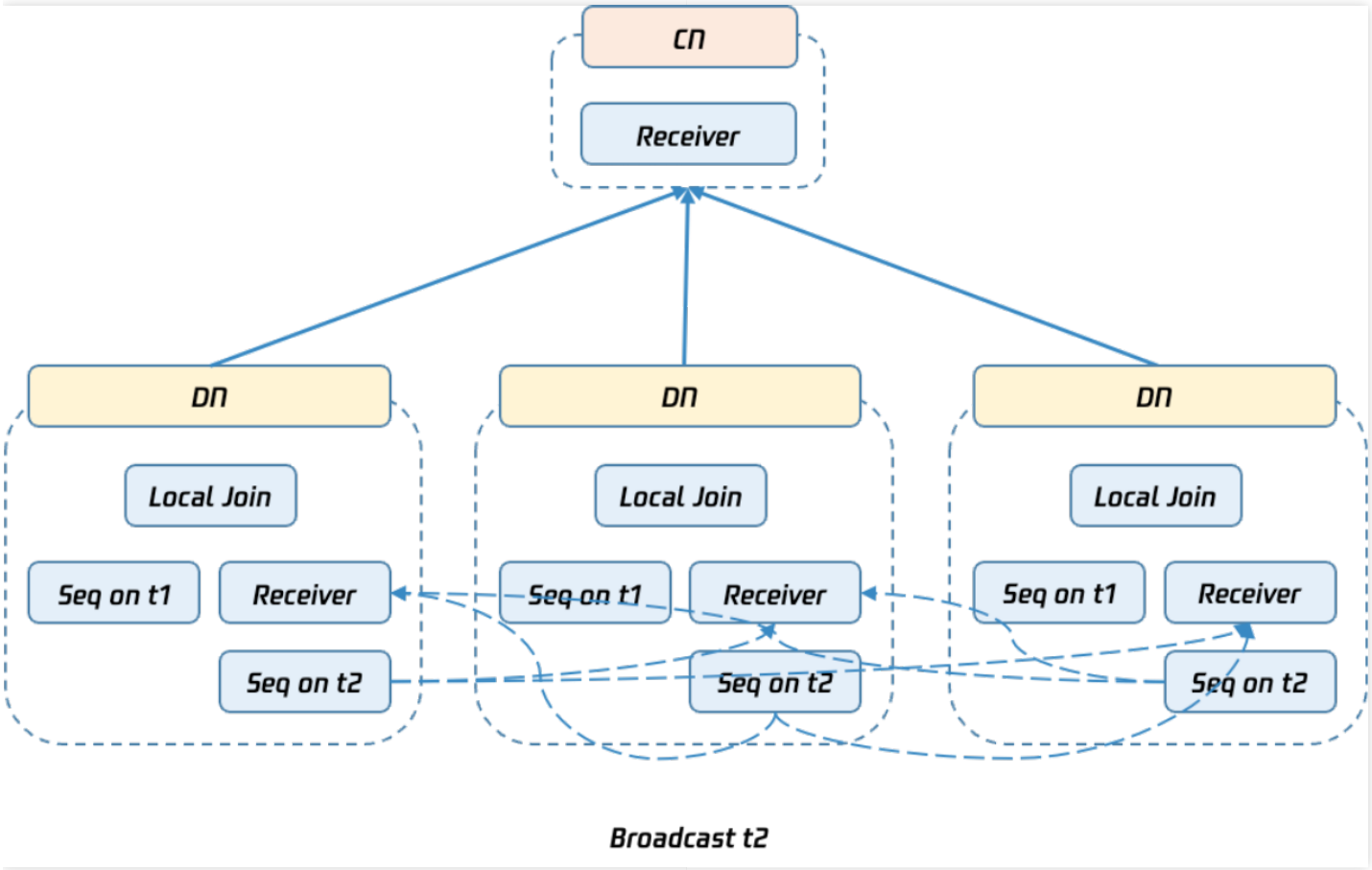
```
tbase =# explain select t1.c3, t2.c3 from t1, t2 where t1.c2=t2.c2;
```

结果为

```
                              QUERY PLAN
------------------------------------------------------------------------------
```

Remote Subquery Scan on all (datanode_1, datanode_2, datanode_3) (cost=6635.00..10070.00 rows=100000 width=8)
-> Hash Join (cost=6635.00..10070.00 rows=100000 width=8)
Hash Cond: (t1.c2 = t2.c2)
-> Seq Scan on t1 (cost=0.00..1935.00 rows=100000 width=8)
-> Hash (cost=4135.00..4135.00 rows=200000 width=8)
-> Remote Subquery Scan on all (datanode_1, datanode_2, datanode_3) (cost=100.00..4135.00 rows=200000 width=8)
-> Seq Scan on t2 (cost=0.00..1935.00 rows=100000 width=8)

EXPLAIN对应的执行流程：



执行计划分类
顺序扫描

```
postgres=# explain select * from t1;
                          QUERY PLAN
---------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Seq Scan on t1  (cost=0.00..10694.64 rows=500564 width=37)
(3 rows)
postgres=#
```

index scan扫描

```
postgres=# explain select * from t1 where f1=1;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001
   ->  Index Scan using tf_f1_idx on t1  (cost=0.42..4.44 rows=1 width=37)
         Index Cond: (f1 = 1)
(4 rows)
```

index only scan扫描

```
postgres=# explain select f1 from t1 where f1=1;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001
   ->  Index Only Scan using tf_f1_idx on t1  (cost=0.42..4.44 rows=1 width=4)
         Index Cond: (f1 = 1)
(4 rows)
```

位图扫描bitmap index scan

```
postgres=# explain select * from t1 where f1 =1  or f1=2;
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Bitmap Heap Scan on t1  (cost=4.86..8.85 rows=2 width=37)
       Recheck Cond: ((f1 = 1) OR (f1 = 2))
       ->  BitmapOr  (cost=4.86..4.86 rows=2 width=0)
           ->  Bitmap Index Scan on tf_f1_idx  (cost=0.00..2.43 rows=1 width=0)
               Index Cond: (f1 = 1)
           ->  Bitmap Index Scan on tf_f1_idx  (cost=0.00..2.43 rows=1 width=0)
               Index Cond: (f1 = 2)
(9 rows)
```

join分类
nest loop

```
postgres=# explain select * from t1,t2 where t1.f1=t2.f1 and t2.f2=1;
                                QUERY PLAN
---------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Nested Loop  (cost=0.70..8.75 rows=1 width=45)
         ->  Index Scan using t2_f2_idx on t2  (cost=0.28..4.30 rows=1 width=8)
               Index Cond: (f2 = 1)
         ->  Index Scan using tf_f1_idx on t1  (cost=0.42..4.44 rows=1 width=37)
               Index Cond: (f1 = t2.f1)
(7 rows)
postgres=#
```

> 注意：
>
> 驱动表要小表。

merge join

```
postgres=# explain select * from t1,t2 where t1.f1=t2.f1;
                                  QUERY PLAN
-------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Merge Join  (cost=1.91..380.65 rows=5039 width=45)
         Merge Cond: (t1.f1 = t2.f1)
         ->  Index Scan using tf_f1_idx on t1  (cost=0.42..15956.88 rows=500564 width=37)
         ->  Index Scan using t2_f1_idx on t2  (cost=0.28..146.87 rows=5039 width=8)
(6 rows)
```

hash join

```
postgres=# explain select * from t3,t4 where t3.f1=t4.f1;
                          QUERY PLAN
------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Hash Join  (cost=76.62..209.52 rows=2561 width=16)
         Hash Cond: (t4.f1 = t3.f1)
         ->  Seq Scan on t4  (cost=0.00..88.39 rows=5039 width=8)
         ->  Hash  (cost=44.61..44.61 rows=2561 width=8)
               ->  Seq Scan on t3  (cost=0.00..44.61 rows=2561 width=8)
(7 rows)
```

数据重分布

```
postgres=# explain select * from t3,t4 where t3.f1=t4.f2 and t4.f1=1;
                                 QUERY PLAN
-----------------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001,dn002)  (cost=200.53..253.91 rows=1 width=16)
   ->  Hash Join  (cost=200.53..253.91 rows=1 width=16)
       Hash Cond: (t3.f1 = t4.f2)
       ->  Seq Scan on t3  (cost=0.00..44.00 rows=2500 width=8)
       ->  Hash  (cost=200.51..200.51 rows=1 width=8)
           ->  Remote Subquery Scan on all (dn001)  (cost=100.00..200.51 rows=1 width=8)
               Distribute results by S: f2
               ->  Seq Scan on t4  (cost=0.00..100.50 rows=1 width=8)
                   Filter: (f1 = 1)
(9 rows)
postgres=# explain select * from t3,t4 where t3.f2=t4.f2 and t4.f1=1;
                                 QUERY PLAN
-----------------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001,dn002)  (cost=300.53..386.41 rows=1 width=16)
   ->  Hash Join  (cost=300.53..386.41 rows=1 width=16)
       Hash Cond: (t3.f2 = t4.f2)
       ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.00..176.50 rows=2500 width=8)
           Distribute results by S: f2
           ->  Seq Scan on t3  (cost=0.00..44.00 rows=2500 width=8)
       ->  Hash  (cost=200.51..200.51 rows=1 width=8)
           ->  Remote Subquery Scan on all (dn001)  (cost=100.00..200.51 rows=1 width=8)
               Distribute results by S: f2
               ->  Seq Scan on t4  (cost=0.00..100.50 rows=1 width=8)
                   Filter: (f1 = 1)
(11 rows)
```

并行扫描

```
postgres=# set max_parallel_workers_per_gather to 2;
postgres=# explain select count(1) from t1;
                                  QUERY PLAN
-------------------------------------------------------------------------------------------------
 Parallel Finalize Aggregate  (cost=9387.28..9387.29 rows=1 width=8)
   ->  Parallel Remote Subquery Scan on all (dn001,dn002)  (cost=9387.17..9387.28 rows=1 width=0)
         ->  Gather  (cost=9287.17..9287.28 rows=1 width=8)
             Workers Planned: 2
             ->  Partial Aggregate  (cost=8287.17..8287.18 rows=1 width=8)
                 ->  Parallel Seq Scan on t1  (cost=0.00..7766.33 rows=208333 width=0)
(6 rows)
```

创建和管理数据库对象（DDL）
创建GROUP
创建数据表默认的default group

TDSQL PG作为做为分布式数据库系统，使用前必需配置实例的默认存储组（也称group），推荐一个实例只创建一个group，该存储组取名为default_group，并且指定该存储组为默认存储组。跨group数据访问会有一些限制，非特殊情况不允许一个实例创建多个group。group中需要添加DN节点，每个group会均匀分配总共4096个shardkey。

下面演示如何创建一个default group：

1. 切换为tbase用户 su tbase。
2. 连接数据库：

> 注意：
>
> 是连接到cn节点(后面没特别说明，所有数据库操作都是连接到cn节点)。

```
psql -h 172.16.0.29 -p 15432 -U tbase -d postgres
```

3. 查询当前什么数据节点（DN）,这些DN节点就是上面初始化集群时建立的。

```
postgres=# select * from pgxc_node where node_type='D';
 node_name | node_type | node_port |  node_host | nodeis_primary | nodeis_preferred |  node_id   | nod
e_cluster_name
-----------+-----------+-----------+------------+----------------+------------------+------------
+------------------
 dn001     | D         |     23001 | 172.16.0.29 | f              | f                |  1485981022 | tbase_cluster
 dn002     | D         |     23002 | 172.16.0.47| f              | f                | -1300059100 | tbase_cluster
(2 rows)
```

4. 建立数据表默认使用的group。

```
postgres=#create default node group default_group with(dn001, dn002);
```

为default group创建shardmap

配置完成数据表默认使用的DN节点后，我们接下来需要配置记录的分区方案，shardmap就是TDSQL PG各个哈希值与DN的对照表，下面演示如何创建一个shardmap给default_group 使用：

```
postgres=# create sharding group to group default_group;
postgres=# clean sharding;
```

至此我们就可以像单机一样使用TDSQL PG集群了。

更多group的使用方法
创建扩展group

- 建立group。

```
postgres=# create node group ext_group with(dn003,dn004);
CREATE NODE GROUP
```

- 为group创建shard。

```
postgres=# create extension sharding group to group ext_group;
CREATE SHARDING GROUP
postgres=# clean sharding;
CLEAN SHARDING
postgres=#
```

删除group

```
postgres=# drop sharding in group ext_group;
DROP SHARDING GROUP
postgres=# drop node group ext_group ;
DROP NODE GROUP
```

查询集群中group的数量

```
postgres=# select * from pgxc_group;
  group_name   | default_group | group_members
---------------+---------------+---------------
 default_group |             1 | 16386 16387
 ext_group     |             0 | 16388 16389
(2 rows)
```

创建和管理模式
模式管理
创建模式

- 标准语句。

```
postgres=# create schema tbase;
CREATE SCHEMA
```

- 扩展语法，不存在时才创建。

```
postgres=# create schema if not exists tbase ;
NOTICE:  schema "tbase" already exists, skipping
CREATE SCHEMA
```

- 指定所属用户。

```
postgres=# create schema tbase_pgxz AUTHORIZATION pgxz;

CREATE SCHEMA

postgres=# \dn tbase_pgxz
 List of schemas
   Name    | Owner
-----------+-------
 tbase_pgxz | pgxz
(1 row)
```

修改模式属性

- 修改模式名。

```
postgres=# alter schema tbase rename to tbase_new;
ALTER SCHEMA
```

- 修改所有者。

```
postgres=# alter schema tbase_pgxz owner to tbase;
ALTER SCHEMA
```

删除模式

```
postgres=# drop schema tbase_new;
DROP SCHEMA
```

当模式中存在对象时，则会删除失败，提示如下：

```
postgres=# create table tbase_pgxz.t(id int);
NOTICE:  Replica identity is needed for shard table, please add to this table through "alter table" comm
and.
CREATE TABLE
postgres=# drop schema tbase_pgxz;
ERROR:  cannot drop schema tbase_pgxz because other objects depend on it
DETAIL:  table tbase_pgxz.t depends on schema tbase_pgxz
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

可以这样强制删除：

```
postgres=# drop schema tbase_pgxz CASCADE;
NOTICE:  drop cascades to table tbase_pgxz.t
DROP SCHEMA
```

配置用户访问模式权限

普通用户对于某个模式下的对象访问除了访问对象要授权外，模式也需要授权。

```
[tbase@VM_0_37_centos root]$ psql -U tbase
psql (PostgreSQL 10.0 TBase V2)
Type "help" for help.

postgres=# create table tbase.t(id int);

CREATE TABLE
```

- 这里授权用户可以访问tbase.t表。

```
postgres=# grant select on tbase.t to pgxz;
GRANT
postgres=# \q
[tbase@VM_0_37_centos root]$ psql -U pgxz
psql (PostgreSQL 10.0 TBase V2)
Type "help" for help.
```

- 在没授权用户可以使用tbase模式前，还是访问不了。

```
postgres=> select * from tbase.t;
ERROR: permission denied for schema tbase
LINE 1: select * from tbase.t;
                      ^
postgres=> \q
[tbase@VM_0_37_centos root]$ psql -U tbase
psql (PostgreSQL 10.0 TBase V2)
Type "help" for help.

postgres=# grant USAGE on SCHEMA tbase to pgxz;
GRANT
postgres=# \q
[tbase@VM_0_37_centos root]$ psql -U pgxz
psql (PostgreSQL 10.0 TBase V2)
Type "help" for help.

postgres=> select * from tbase.t;
 id
----
(0 rows)
```

配置访问模式的顺序

TDSQL PG数据库有一个运行变量叫search_path，其值为模式名列表，用于配置访问数据对象的顺序，如下所示：

1. 当前连接用户。

```
postgres=# select current_user;
 current_user
--------------
 tbase
(1 row)
```

2. 总共三个模式。

```
postgres=# \dn
   List of schemas
     Name     | Owner
--------------+-------
 public       | tbase
 tbase        | tbase
 tbase_schema | tbase
(3 rows)
```

3. 搜索路径只配置为"$user", public，其中"$user"为当前用户名，即上面的current_user值“tbase”。

```
postgres=# show search_path ;
   search_path
-----------------
 "$user", public
(1 row)
```

4. 不指定模式创建数据表，则该表存放于第一个搜索模式下面。

```
postgres=# create table t(id int,mc text);
CREATE TABLE
postgres=# \dt
       List of relations
 Schema | Name | Type  | Owner
--------+------+-------+-------
 tbase  | t    | table | tbase
(1 row)
```

5. 指定表位于某个模式下，不同模式下表名可以相同。

```
postgres=# create table public.t(id int,mc text);
CREATE TABLE
postgres=# \dt public.t
       List of relations
 Schema | Name | Type  | Owner
--------+------+-------+-------
 public | t    | table | tbase
(1 row)

postgres=# create table tbase_schema.t1(id int,mc text);
CREATE TABLE
postgres=#
```

6. 访问不在搜索路径对象时，需要写全路径。

```
postgres=# select * from t1;
ERROR:  relation "t1" does not exist
LINE 1: select * from t1;
                      ^
postgres=# select * from tbase_schema.t1;
 id | mc
----+----
(0 rows)
```

上面出错就是因为模式tbase_schema没有配置在search_path搜索路径中。

创建和管理表
创建表

使用命令create table创建表

1）不用指定shard key建表，系统默认使用第一个字段做为表的shard key

```
create table t_first_col_share(id serial not null,nickname text);
```

2）指定shard key建表

```
create table t_appoint_col(id serial not null,nickname text) distribute by shard(nickname);
```

⚠️分布键选择原则

- 分布键只能选择一个字段
- 如果有主键，则选择主键做分布键
- 如果主键是复合字段组合，则选择字段值选择性多的字段做分布键
- 也可以把复合字段拼接成一个新的字段来做分布键
- 没有主键的可以使用UUID来做分布键
- 让数据尽可能的分布得足够散

3）表不存在时才创建

```
create table IF NOT EXISTS t(id int,mc text);
```

数据类型

数字类型

名字	存储尺寸	描述	范围
smallint	2字节	小范围整数	-32768 to +32767
integer	4字节	整数的典型选择	-2147483648 to +2147483647
bigint	8字节	大范围整数	-9223372036854775808 to +9223372036854775807
decimal	可变	用户指定精度，精确	最高小数点前131072位，以及小数点后16383位
numeric	可变	用户指定精度，精确	最高小数点前131072位，以及小数点后16383位
real	4字节	可变精度，不精确	6位十进制精度
double precision	8字节	可变精度，不精确	15位十进制精度
smallserial	2字节	自动增加的小整数	1到32767
serial	4字节	自动增加的整数	1到2147483647
bigserial	8字节	自动增长的大整数	1到9223372036854775807

字符类型

名字	描述
character varying(*n*), varchar(*n*)	有限制的变长
character(*n*), char(*n*)	定长，空格填充
text	1GB

⚠对于类型CHAR来说，如果不指定n的值，n就会默认为1。 如果要存储的字符串长度小于n，那么类型CHAR的值将会不用空格补齐，与Oracle略有不同。

二进制数据类型

名字	存储尺寸	描述
bytea	1或4字节外加真正的二进制串	变长二进制串

日期类型

名字	存储尺寸	描述	最小值	最大值	解析度
timestamp [(*p*)] [without time zone]	8字节	包括日期和时间（无时区）	4713 BC	294276 AD	1微秒 / 14位
timestamp [(*p*)] [with time zone]	8字节	包括日期和时间（有时区）	4713 BC	294276 AD	1微秒 / 14位
date	4字节	日期（没有一天中的时间）	4713 BC	5874897 AD	1日
time [(*p*)] [without time zone]	8字节	一天中的时间（无时区）	0:00:00	24:00:00	1微秒 / 14位
time [(*p*)] [with time zone]	12字节	仅是一天中的时间，带有时区	00:00:00+1459	24:00:00-1459	1微秒 / 14位
interval [*fields*] [(*p*)]	16字节	时间间隔	-178000000年	178000000年	1微秒 / 14位

布尔类型

名字	存储字节	描述
boolean	1字节	状态为真或假

异构数据库类型对照表
与ORACLE数据类型对比

Oracle	TDSQL PostgreSQL版（Oracle兼容版）
Number	可以对应 TDSQL PostgreSQL版（Oracle兼容版）的 smallint, integer, bigint, numeric(p,s) 等多种数据类型。由于 smallint，Integer，bigint 的算术运算效率比 numeric 高的多，所以要视业务需要转换成对应的 smallint, integer, bigint，无法转换时才转换成 numeric(p,s)
float	float（实际按照 double precision 或 real 存储）
binary_float	binary_float（实际按照 real 存储）
binary_double	binary_double（实际按照 double preciso pion 存储）
char	char
nchar	char
varchar2	varchar2
nvarchar2	nvarchar2
rowid	rowid
urowid	urowid
long	long
clob	clob（将clob转换成text类型，最大可存储1GB）
nclob	nclob
blob	blob（将blob转换成bytea类型，最大可存储1GB）
bfile	bfile
Long raw	Long raw
raw	raw
date	date
timestamp	Timestamp
Interval	interval

与MYSQL数据类型对照表

与Mysql对照表

Mysql	TDSQL PG
int	int
smallint	smallint
bigint	bigint
int AUTO_INCREMENT	serial
smallint AUTO_INCREMENT	smallserial
bigint AUTO_INCREMENT	bigserial
bit	bit
tinyint	boolean
float	real
double	double precision
decimal	numeric
char	char
varchar	varchar
text	text
date	date
time	time
datetime	timestamp
longblob	bytea
Longtext	text
ENUM CREATE TABLE TYPE022(COL1 ENUM('S','M','L','XL','XXL','XXXL') ,COL2 INT PRIMARY KEY);	自定义类型 CREATE TYPE mood AS ENUM ('S','M','L','XL','XXL','XXXL'); CREATE TABLE TYPE022(COL1 mood ,COL2 INT PRIMARY KEY)

Mysql	TDSQL PG
SET类型 CREATE TABLE TYPE023(COL1 SET('A','B','C','D') ,COL2 INT PRIMARY KEY)	CREATE TABLE TYPE023(COL1 VARCHAR check(regexp_split_to_array(col1,',') <@ array['A','B','C','D']) ,COL2 INT PRIMARY KEY);

与SQLSERVER数据类型对照表

与SQLSERVER对照表

SQLSERVER	TDSQL PG
smallint	smallint
int	int
bigint	bigint
tinyint	smallint
real	real
float	double precision
numeric	numeric
bit	bit
char	char
nchar	char
varchar	varchar
nvarchar	varchar
text	text
ntext	text
date	date
time	time
datetime	timestamp
datetime2	timestamp
smalldatetime	timestamp
datetimeoffset	Timestamp
timestamp	money
uniqueidentifier	uuid

SQLSERVER	TDSQL PG
image	bytea
binary	bytea
varbinary	bytea

与INFORMIX数据类型对照表

与informix对照表

informix	TDSQL PG
smallint	smallint
integer	integer
float	double precision
smallfloat	double precision
DECIMAL	numeric
MONEY	numeric
char	char
varchar	varchar
lvarchar	varchar
text	text
date	date
datetime	timestamp
interval	interval
BYTE	bytea

自定义复合类型
增加一个复合类型

```
postgres=# create type my_type as (f1 int,f2 varchar(10));
CREATE TYPE
```

数据表使用复合类型

```
postgres=# create table t_my_type (f1 int,f_my_type my_type);
CREATE TABLE
postgres=# insert into t_my_type values(1,row(1,'TDSQL PG'));
INSERT 0 1
postgres=# select f1,(f_my_type).f2 from t_my_type;
 f1 |  f2
----+-------
  1 | TDSQL PG
(1 row)
postgres=#
```

json/jsonb类型
json应用
创建json类型字段表

```
postgres=# create table t_json(id int,f_json json);
CREATE TABLE
```

插入数据

```
postgres=# insert into t_json values(1,'{"col1":1,"col2":"TDSQL PG"}');
INSERT 0 1
postgres=# insert into t_json values(2,'{"col1":1,"col2":"TDSQL PG","col3":"pgxz"}');
INSERT 0 1
postgres=# select * from t_json;
 id |                f_json
----+---------------------------------------
  1 | {"col1":1,"col2":"TDSQL PG"}
  2 | {"col1":1,"col2":"TDSQL PG","col3":"pgxz"}
(2 rows)
```

通过键获得 JSON 对象域

```
postgres=# select f_json ->'col2' as col2 ,f_json -> 'col3' as col3 from t_json;
  col2   |  col3
---------+--------
 "TDSQL PG" |
 "TDSQL PG" | "pgxz"
(2 rows)
```

以文本形式获取对象值

```
postgres=# select f_json ->>'col2' as col2 ,f_json ->> 'col3' as col3 from t_json;
 col2  | col3
-------+------
 TDSQL PG |
 TDSQL PG | pgxz
(2 rows)
postgres=# select f_json ->>'col2' as col2 ,f_json ->> 'col3' as col3 from t_json where f_json ->> 'col3'
is not null;
 col2  | col3
-------+------
 TDSQL PG | pgxz
(1 row)
```

jsonb应用
创建jsonb类型字段表

```
postgres=# create table t_jsonb(id int,f_jsonb jsonb);
CREATE TABLE
```

插入数据

```
postgres=# insert into t_jsonb values(1,'{"col1":1,"col2":"TDSQL PG"}');
INSERT 0 1
postgres=# insert into t_jsonb values(2,'{"col1":1,"col2":"TDSQL PG","col3":"pgxz"}');
INSERT 0 1
postgres=# select * from t_jsonb;
 id |                 f_jsonb
----+-------------------------------------------
  1 | {"col1": 1, "col2": "TDSQL PG"}
  2 | {"col1": 1, "col2": "TDSQL PG", "col3": "pgxz"}
(2 rows)
```

jsonb插入时会移除重复的键，如下所示：

```
postgres=# insert into t_jsonb values(3,'{"col1":1,"col2":"TDSQL PG","col2":"pgxz"}');
INSERT 0 1
postgres=# select * from t_jsonb;
 id |                 f_jsonb
----+-------------------------------------------
  1 | {"col1": 1, "col2": "TDSQL PG"}
  3 | {"col1": 1, "col2": "pgxz"}
  2 | {"col1": 1, "col2": "TDSQL PG", "col3": "pgxz"}
(3 rows)
```

更新数据

1. 增加元素。

```
postgres=# update t_jsonb set f_jsonb = f_jsonb || '{"col3":"pgxz"}'::jsonb where id=1;
UPDATE 1
```

2. 更新原来的元素。

```
postgres=# update t_jsonb set f_jsonb = f_jsonb || '{"col2":"TDSQL PG"}'::jsonb where id=3;
UPDATE 1
postgres=# select * from t_jsonb;
 id |                  f_jsonb
----+-------------------------------------------
  2 | {"col1": 1, "col2": "TDSQL PG", "col3": "pgxz"}
  1 | {"col1": 1, "col2": "TDSQL PG", "col3": "pgxz"}
  3 | {"col1": 1, "col2": "TDSQL PG"}
(3 rows)
```

3. 删除某个键。

```
postgres=# update t_jsonb set f_jsonb = f_jsonb - 'col3';
UPDATE 3
postgres=# select * from t_jsonb;
 id |          f_jsonb
----+----------------------------
  2 | {"col1": 1, "col2": "TDSQL PG"}
  1 | {"col1": 1, "col2": "TDSQL PG"}
  3 | {"col1": 1, "col2": "TDSQL PG"}
(3 rows)
```

jsonb_set()函数更新数据

jsonb_set(target jsonb, path text[], new_value jsonb, [create_missing boolean])

> 说明：
>
> target指要更新的数据源，path指路径，new_value指更新后的键值，create_missing值为true表示如果键不存在则添加，create_missing值为false表示如果键不存在则不添加。

```
postgres=# update t_jsonb set f_jsonb = jsonb_set( f_jsonb , '{col}' , '"pgxz"' , true ) where id=1;
UPDATE 1
postgres=# update t_jsonb set f_jsonb = jsonb_set( f_jsonb , '{col}' , '"pgxz"' , false ) where id=2;
UPDATE 1
postgres=# update t_jsonb set f_jsonb = jsonb_set( f_jsonb , '{col2}' , '"pgxz"' , false ) where id=3;
UPDATE 1
postgres=# select * from t_jsonb;
 id |                  f_jsonb
----+-------------------------------------------
  1 | {"col": "pgxz", "col1": 1, "col2": "TDSQL PG"}
  2 | {"col1": 1, "col2": "TDSQL PG"}
  3 | {"col1": 1, "col2": "pgxz"}
(3 rows)
```

jsonb函数应用
jsonb_each()将json对象转变键和值

```
postgres=# select  f_jsonb  from t_jsonb where id=1;
                f_jsonb
-------------------------------------------
 {"col": "pgxz", "col1": 1, "col2": "TDSQL PG"}
(1 row)
postgres=#  select * from  jsonb_each((select  f_jsonb  from t_jsonb where id=1));
 key  |  value
------+---------
 col  | "pgxz"
 col1 | 1
 col2 | "TDSQL PG"
(3 rows)
```

jsonb_each_text()将json对象转变文本类型的键和值

```
postgres=#  select * from  jsonb_each_text((select  f_jsonb  from t_jsonb where id=1));
 key  | value
------+-------
 col  | pgxz
 col1 | 1
 col2 | TDSQL PG
(3 rows)
```

row_to_json()将一行记录变成一个json对象

```
postgres=# \d+ tbase
                                Table "public.tbase"
  Column  |  Type   | Collation | Nullable | Default | Storage  | Stats target | Description
----------+---------+-----------+----------+---------+----------+--------------+-------------
 id       | integer |           | not null |         | plain    |              |
 nickname | text    |           |          |         | extended |              |
Indexes:
    "tbase_pkey" PRIMARY KEY, btree (id)
Distribute By: SHARD(id)
Location Nodes: ALL DATANODES

postgres=# select * from tbase;
 id | nickname
----+----------
  1 | TDSQL PG
  2 | pgxz
(2 rows)
postgres=# select row_to_json(tbase) from tbase;
        row_to_json
----------------------------
 {"id":1,"nickname":"TDSQL PG"}
 {"id":2,"nickname":"pgxz"}
(2 rows)
```

json_object_keys()返回一个对象中所有的键

```
postgres=#  select * from json_object_keys((select  f_jsonb  from t_jsonb where id=1)::json);
 json_object_keys
------------------
 col
 col1
 col2
(3 rows)
```

jsonb索引使用

创建立jsonb索引

```
postgres=# create index t_jsonb_f_jsonb_idx on t_jsonb using gin(f_jsonb);
CREATE INDEX
postgres=# \d+ t_jsonb
                                Table "public.t_jsonb"
 Column  |  Type   | Collation | Nullable | Default | Storage  | Stats target | Description
---------+---------+-----------+----------+---------+----------+--------------+-------------
 id      | integer |           |          |         | plain    |              |
 f_jsonb | jsonb   |           |          |         | extended |              |
Indexes:
    "t_jsonb_f_jsonb_idx" gin (f_jsonb)
Distribute By: SHARD(id)
Location Nodes: ALL DATANODES
```

测试查询的性能

```
postgres=# select count(1) from t_jsonb;
  count
----------
 10000000
(1 row)
postgres=# analyze t_jsonb;
ANALYZE
```

- 没有索引开销

```
postgres=# select * from t_jsonb where f_jsonb @> '{"col1":9999}';
  id  |           f_jsonb
------+------------------------------
 9999 | {"col1": 9999, "col2": "9999"}
 9999 | {"col1": 9999, "col2": "9999"}
 9999 | {"col1": 9999, "col2": "9999"}
 9999 | {"col1": 9999, "col2": "9999"}
 9999 | {"col1": 9999, "col2": "9999"}
(5 rows)
Time: 2473.488 ms (00:02.473)
```

- 有索引开销

```
postgres=# select * from t_jsonb where f_jsonb @> '{"col1":9999}';
  id  |           f_jsonb
------+------------------------------
 9999 | {"col1": 9999, "col2": "9999"}
 9999 | {"col1": 9999, "col2": "9999"}
 9999 | {"col1": 9999, "col2": "9999"}
 9999 | {"col1": 9999, "col2": "9999"}
 9999 | {"col1": 9999, "col2": "9999"}
(5 rows)
Time: 217.968 ms
```

部分数据类型示例

varchar2

```
postgres=# create table t_varchar2(f1 varchar2,f2 int);
CREATE TABLE
postgres=# \d+ t_varchar2
                          Table "public.t_varchar2"
 Column |   Type   | Collation | Nullable | Default | Storage  | Stats target | Description
--------+----------+-----------+----------+---------+----------+--------------+-------------
 f1     | varchar2 |           | not null |         | extended |              |
 f2     | integer  |           |          |         | plain    |              |
```

number

```
postgres=# create table t_number(f1 number,f2 number(10),f3 number(10,2));
CREATE TABLE
postgres=#  \d t_number
                Table "public.t_number"
 Column |     Type      | Collation | Nullable | Default
--------+---------------+-----------+----------+---------
 f1     | numeric       |           | not null |
 f2     | numeric(10,0) |           |          |
 f3     | numeric(10,2) |           |          |
```

系统转换成numeric。

blob

```
postgres=# create table t_blob(f1 int,f2 Blob);
CREATE TABLE
postgres=# \d t_blob
              Table "public.t_blob"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 f1     | integer |           | not null |
 f2     | blob    |           |          |
```

TDSQL PG的blob类型支持最大长度为1G。

clob

```
postgres=# create table t_clob(f1 int,f2 clob);
CREATE TABLE
postgres=# \d t_clob
              Table "public.t_clob"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 f1     | integer |           | not null |
 f2     | clob    |           |          |
```

TDSQL PG的clob类型支持最大长度为1G。

外键创建与删除

外键创建

外键（FOREIGN KEY），是用于将两个表链接在一起的键。 FK是一个表中的一个字段（或字段集合），它引用另一个表中的PRIMARY KEY

```
postgres=# create table t_p(f1 int not null,f2 int ,primary key(f1));
CREATE TABLE
postgres=# create table t_f(f1 int not null,f2 int );
CREATE TABLE
postgres=# ALTER TABLE t_f ADD CONSTRAINT t_f_f1_fkey FOREIGN KEY (f1) REFERENCES t_p (f1);
ALTER TABLE
postgres=#
```

外键只是同一个节点内约束有效果，所以外键字段和对应主键字段必需都是表的分布键，否则由于数据分布于不同的节点内会导致更新失败。
分片表支持外键引用表自身。
示例：

```
 --创建表
 create table TIB_SYS_SAP_RFC_IMPORT
(
  fd_id                 VARCHAR2(36) not null,
  fd_order              NUMBER(10),
  fd_parameter_use      NUMBER(1),
  fd_parameter_name     VARCHAR2(200),
  fd_parameter_type     VARCHAR2(10),
  fd_parameter_length   NUMBER(10),
  fd_parameter_type_name VARCHAR2(200),
  fd_parameter_required  NUMBER(1),
  fd_parameter_mark     VARCHAR2(200),
  fd_hierarchy_id       VARCHAR2(1000),
  fd_function_id        VARCHAR2(36),
  fd_parent_id          VARCHAR2(36),
  fd_rfc_param_xml      CLOB
) ;

--创建主键
alter table TIB_SYS_SAP_RFC_IMPORT add primary key (FD_ID);

--创建外键
alter table TIB_SYS_SAP_RFC_IMPORT add constraint FK100552563B582DA5 foreign key (FD_PARENT_I
D) references TIB_SYS_SAP_RFC_IMPORT (FD_ID);
```

删除外键

```
postgres=# \d+ t_f
                                Table "public.t_f"
 Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------+-------------
 f1     | integer |           | not null |         | plain   |              |
 f2     | integer |           |          |         | plain   |              |
Foreign-key constraints:
    "t_f_f1_fkey" FOREIGN KEY (f1) REFERENCES t_p(f1)
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES

postgres=# ALTER TABLE t_f drop constraint t_f_f1_fkey;
ALTER TABLE
postgres=# \d+ t_f
                                Table "public.t_f"
 Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------+-------------
 f1     | integer |           | not null |         | plain   |              |
 f2     | integer |           |          |         | plain   |              |
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES
postgres=#
```

创建和管理索引
创建和删除索引

创建索引

使用命令create index创建索引

示例：

1）创建唯一索引

```
create table t_first_col_share(id serial not null,nickname text);
create unique index t_first_col_share_id_uidx on t_first_col_share using btree(id);
```

⚠非shard key字段不能建立唯一索引

2）创建多字段索引

```
create table t_mul_idx (f1 int,f2 int,f3 int,f4 int);
create index t_mul_idx_idx on t_mul_idx(f1,f2,f3);
```

⚠or查询条件bitmap scan最多支持两个不同字段条件

删除索引

使用命令DROP INDEX修改索引

```
drop index t_first_col_share_id_uidx;
```

索引类型
全局索引

TDSQL PG是一个Share Nothing的MPP分布式数据，表数据通过Shard表的分布key散落到多个DN，DN间数据没有冗余。当业务查询或者DML操作有分布key条件时，CN上可以直接路由到目标DN。
全局索引存储了对应索引数据的存储目标节点信息，在业务查询缺少分布key条件时，也可以精确地定位对应索引key的数据在全局中的位置，不必扫描所有DN节点来获取数据，可以减少对资源的消耗，从而提高吞吐量（故全局索引一般创建在非分布key列）。
TDSQL PG全局索引支持的表类型：shard表，分区表。暂不支持复制表。
数据类型支持：能够用于shard列的类型，smallint, integer, bigint, real, double precision, binary_float, binary_double, smallserial, serial, bigserial, OID, boolean, char, name, varchar, text, varchar2, nvchar, oidvector, abstime, reltime, cash, bpchar, raw, byte, date, time, oracledate, timestamp, timestamptz, interval, numeric, jsonb, rid
（bfile，xml，bytea不支持）。
索引类型默认Btree（暂仅支持该类型）。

语法：
```
create global [unique] index [if not exists] index_name on table_name [using method] (column_name);
```

示例：
1、创建全局索引

```
postgres=#
create table gindex_test(a int,b int,c varchar,d numeric,e float);
postgres=#
create global index on gindex_test using btree (b);
```

---查询创建结果

```
postgres=#
\d+ gindex_test
                              Table "public.gindex_test"
 Column |       Type        | Collation | Nullable | Default | Storage  | Stats target | Description
--------+-------------------+-----------+----------+---------+----------+--------------+-------------
 a      | integer           |           | not null |         | plain    |              |
 b      | integer           |           |          |         | plain    |              |
 c      | character varying |           |          |         | extended |              |
 d      | numeric           |           |          |         | main     |              |
 e      | double precision  |           |          |         | plain    |              |
Indexes:
    "gindex_test_b_idx" GLOBAL, btree (b)
Distribute By: SHARD(a)
```

```
Location Nodes: ALL DATANODES
```

2、创建全局唯一索引

```
postgres=#
create global unique index on gindex_test(c);
```

---查询创建结果

```
postgres=#
\d+ gindex_test
                                Table "public.gindex_test"
 Column |       Type        | Collation | Nullable | Default | Storage  | Stats target | Description
--------+-------------------+-----------+----------+---------+----------+--------------+-------------
 a      | integer           |           | not null |         | plain    |              |
 b      | integer           |           |          |         | plain    |              |
 c      | character varying |           |          |         | extended |              |
 d      | numeric           |           |          |         | main     |              |
 e      | double precision  |           |          |         | plain    |              |
Indexes:
    "gindex_test_c_idx" UNIQUE, GLOBAL, btree (c)
    "gindex_test_b_idx" GLOBAL, btree (b)
Distribute By: SHARD(a)
Location Nodes: ALL DATANODES
```

3、全局索引列的关联查询

---创建表并插入数据

```
postgres=#
create table gi_insert1(a int,f1 int,f2 varchar2(20),f3 numeric,f4 float,f5 char(30),f6 nchar(20),f7 raw(20),f
8 clob,f9 blob,f10 timestamp with local time zone,f11 binary_double,f12 binary_float,f13 real,f14 nclob,
f15 date,f16 timestamp,f17 interval day to second, f18 interval year to month,f19 xmltype,f20 long) ;

postgres=#
insert into gi_insert1 select t,t,t,t,t,to_char(t),to_char(t),hextoraw(t),to_char(t),to_char(t)::blob, timestamp
'2021-06-22 16:02:07.067029'+t*interval '1' day, sin(t),t,t,to_clob(to_char(t)),date '2021-06-22 16:02:0
7'+ t*interval '1' day,timestamp '2021-06-22 16:02:07.067029'+t*interval '1' day, t*interval '1' day,interv
al '3-2' year to month,'<ROW>XMLType2</ROW>' ,to_char(t) from generate_series(1,10)t;

postgres=#
create table gi_insert(a int,f1 int,f2 varchar2(20),f3 numeric,f4 float,f5 char(30),f6 nchar(20),f7 raw(20),f8
clob,f9 blob,f10 timestamp with local time zone,f11 binary_double,f12 binary_float,f13 real,f14 nclob,f1
5 date,f16 timestamp,f17 interval day to second, f18 interval year to month,f19 xmltype,f20 long) ;

postgres=#
insert into gi_insert select t,t,t,t,t,to_char(t),to_char(t),hextoraw(t),to_char(t),to_char(t)::blob, timestamp
```

```
'2021-06-22 16:02:07.067029'+t*interval '1' day, sin(t),t,t,to_clob(to_char(t)),date '2021-06-22 16:02:0
7'+ t*interval '1' day,timestamp '2021-06-22 16:02:07.067029'+t*interval '1' day, t*interval '1' day,interv
al '3-2' year to month,'<ROW>XMLType2</ROW>' ,to_char(t) from generate_series(1,10)t;
```

---创建全局索引

```
create global index on gi_insert1 using btree (f1);
create global index on gi_insert1 using btree (f2);
create  global unique index on gi_insert1 (f3);
create global index on gi_insert1(f4);
create global index if not exists index5 on gi_insert1 using btree (f13);

create global index on gi_insert using btree (f1);
create global index on gi_insert using btree (f2);
create  global unique index on gi_insert (f3);
create global index on gi_insert(f4);
create global index if not exists index51 on gi_insert using btree (f13);
```

---连接查询

```
select a.f1
from gi_insert a
join gi_insert1 b
on a.f1=b.f2
where a.f3=(select floor(avg(f)) from (select max(decode(f4,4,f4*f4,f4,f4+1,f4+10)) f
        from gi_insert
        where f19='<ROW>XMLType2</ROW>'
        group by f4
        having f4>=any (select (case f3 when 1 then 2 when 2 then 3 else 5 end)
                from gi_insert1
                where f1 in (select f1
                        from gi_insert
                        where f13>=any(select t1.f13 from gi_insert t1
                                left join gi_insert1 t2
                                on t1.f13=t2.f13
                                group by t1.f13
                                )
                        group by f1,f13
                        having f1<=(select max(f3)
                                 from (select t1.f3 f3
                                      from gi_insert t1
                                      left join gi_insert1 t2
                                      on t1.f1=t2.f1
                                      group by t1.f3
                                      ) a
                                where f3 >=1
```

```
                                        )
                                    )
                                )
                            ))
    group by a.f1
    having a.f1>=1;

--查询执行结果

    f1
    ----
     8
    (1 row)
```

4、支持分区表创建全局唯一索引

```
    --创建表
    create table t_order_range_20220721_76(id bigserial not null,userid integer,product text, createdate dat
    e not null) partition by range ( createdate ) distribute by shard(id);

    --创建分区
    create table t_order_range_20220721_76_202204 partition of t_order_range_20220721_76(id primary k
    ey,userid,product, createdate) for values from ('2022-04-01') to ('2022-05-01');

    create table t_order_range_20220721_76_202205 partition of t_order_range_20220721_76(id primary k
    ey,userid,product, createdate) for values from ('2022-05-01') to ('2022-06-01');

    create table t_order_range_20220721_76_202206 partition of t_order_range_20220721_76(id primary k
    ey,userid,product, createdate) for values from ('2022-06-01') to ('2022-07-01');

    create table t_order_range_20220721_76_202207 partition of t_order_range_20220721_76(id primary k
    ey,userid,product, createdate) for values from ('2022-07-01') to ('2022-08-01');

    --插入数据
    insert into t_order_range_20220721_76 values(1,7,'123','2022-05-13');
    insert into t_order_range_20220721_76 values(2,8,'234','2022-06-13');
    insert into t_order_range_20220721_76 values(3,9,'345','2022-07-13');
    insert into t_order_range_20220721_76 values(6,10,'456','2022-07-03');
    insert into t_order_range_20220721_76 values(4,11,'567','2022-07-11');
    insert into t_order_range_20220721_76 values(5,12,'678','2022-05-08');

    --create global index

    create global index on t_order_range_20220721_76(userid);
    select a.* from t_order_range_20220721_76 a, t_order_range_20220721_76_userid_idx_idt b where a.use
    rid = b.indexkey and a.ctid = b.location and a.shardid = b.shard and a.tableoid = b.subtableoid order b
```

```
y id;
```

--查询结果

```
 id | userid | product |     createdate
----+--------+---------+---------------------
  1 |      7 | 123     | 2022-05-13 00:00:00
  2 |      8 | 234     | 2022-06-13 00:00:00
  3 |      9 | 345     | 2022-07-13 00:00:00
  4 |     11 | 567     | 2022-07-11 00:00:00
  5 |     12 | 678     | 2022-05-08 00:00:00
  6 |     10 | 456     | 2022-07-03 00:00:00
```

普通索引

```
postgres=# create index t_appoint_id_idx on t_appoint_col using btree(id);
CREATE INDEX
```

唯一索引

- 创建唯一索引。

```
postgres=# create unique index t_first_col_share_id_uidx on t_first_col_share using btree(id);
CREATE INDEX
```

- 非shard key字段不能建立唯一索引。

```
postgres=# create unique index t_first_col_share_nickname_uidx on t_first_col_share using btree(nickna
me);
ERROR:  Unique index of partitioned table must contain the hash/modulo distribution column.
```

表达式索引

```
postgres=# create table t_upper(id int,mc text);
postgres=# create index t_upper_mc on t_upper(mc);
postgres=# insert into t_upper select t,md5(t::text) from generate_series(1,10000) as t;
postgres=# analyze t_upper;
ANALYZE

postgres=# explain select * from t_upper where upper(mc)=md5('1');
                          QUERY PLAN
------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Seq Scan on t_upper  (cost=0.00..135.58 rows=25 width=37)
         Filter: (upper(mc) = 'c4ca4238a0b923820dcc509a6f75849b'::text)
(4 rows)

postgres=# create index t_upper_mc on t_upper(upper(mc));
CREATE INDEX
postgres=# explain select * from t_upper where upper(mc)=md5('1');
                               QUERY PLAN
-------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Bitmap Heap Scan on t_upper  (cost=2.48..32.43 rows=25 width=37)
         Recheck Cond: (upper(mc) = 'c4ca4238a0b923820dcc509a6f75849b'::text)
         ->  Bitmap Index Scan on t_upper_mc  (cost=0.00..2.47 rows=25 width=0)
               Index Cond: (upper(mc) = 'c4ca4238a0b923820dcc509a6f75849b'::text)
(6 rows)
```

条件索引

```
postgres=# create table t_sex(id int,sex text) ;
postgres=# create index t_sex_sex_idx on t_sex (sex);
postgres=# insert into t_sex select t,'男' from generate_series(1,1000000) as t;
postgres=# insert into t_sex select t,'女' from generate_series(1,100) as t;
postgres=# analyze t_sex ;
ANALYZE
postgres=#

postgres=# explain  select * from t_sex where sex ='女';
                              QUERY PLAN
-------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Index Scan using t_sex_sex_idx on t_sex  (cost=0.42..5.81 rows=67 width=8)
         Index Cond: (sex = '女'::text)
(4 rows)

#索引对于条件为男的情况下无效

postgres=# explain  select * from t_sex where sex ='男';
                         QUERY PLAN
---------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Seq Scan on t_sex  (cost=0.00..9977.58 rows=500539 width=8)
         Filter: (sex = '男'::text)
(4 rows)

#连接dn节点查看索引点用空间大，而且度数也高

postgres=# \di+
                                 List of relations
 Schema |     Name      | Type  | Owner |  Table  | Size  | Allocated Size | Description
--------+---------------+-------+-------+---------+-------+----------------+-------------
 tbase  | t_sex_sex_idx | index | tbase | t_sex   | 14 MB | 14 MB          |
 tbase  | t_upper_mc    | index | tbase | t_upper | 14 MB | 14 MB          |
(2 rows)

postgres=# \q

postgres=# drop index t_sex_sex_idx;
DROP INDEX
postgres=#  create index t_sex_sex_idx on t_sex (sex) where sex='女';
```

```
CREATE INDEX
postgres=# analyze t_sex;
ANALYZE
postgres=# explain  select * from t_sex where sex ='女';
                              QUERY PLAN
-------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Index Scan using t_sex_sex_idx on t_sex  (cost=0.14..6.69 rows=33 width=8)
(3 rows)

postgres=# explain  select * from t_sex where sex ='男';
                         QUERY PLAN
------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Seq Scan on t_sex  (cost=0.00..9977.58 rows=500573 width=8)
         Filter: (sex = '男'::text)
(4 rows)

postgres=# \di+
                                 List of relations
 Schema |     Name      | Type  | Owner |  Table  | Size  | Allocated Size | Description
--------+---------------+-------+-------+---------+-------+----------------+-------------
 tbase  | t_sex_sex_idx | index | tbase | t_sex   | 16 kB | 16 kB          |
 tbase  | t_upper_mc    | index | tbase | t_upper | 14 MB | 14 MB          |
(2 rows)
```

gist索引

```
postgres=# create table t_trgm (id int,trgm text,no_trgm text) ;
postgres=# create index t_trgm_trgm_idx on t_trgm using gist(trgm gist_trgm_ops);
```

> 注意：
>
> 使用gist索引需要预先安装插件。

```
create extension pg_trgm;
```

gin索引

- pg_trgm索引。

```
postgres=# drop index t_trgm_trgm_idx;
DROP INDEX
Time: 55.954 ms
postgres=# create index t_trgm_trgm_idx on t_trgm using gin(trgm gin_trgm_ops);
```

- jsonb索引。

```
postgres=# create table t_jsonb(id int,f_jsonb jsonb);
postgres=# create index t_jsonb_f_jsonb_idx on t_jsonb using gin(f_jsonb);
```

- 数组索引。

```
postgres=# create table t_array(id int, mc text[]);
postgres=# insert into t_array select t,('{'||md5(t::text)||'}')::text[] from generate_series(1,1000000) as t;
postgres=# analyze;
ANALYZE
postgres=# \timing
Timing is on.
postgres=# explain select * from t_array where mc @> ('{'||md5('1')||'}')::text[];
                                QUERY PLAN
---------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Gather  (cost=1000.00..12060.25 rows=2503 width=61)
         Workers Planned: 2
         ->  Parallel Seq Scan on t_array  (cost=0.00..10809.95 rows=1043 width=61)
               Filter: (mc @> ('{c4ca4238a0b923820dcc509a6f75849b}'::cstring)::text[])
(6 rows)
Time: 4.105 ms
postgres=# select * from t_array where mc @> ('{'||md5('1')||'}')::text[];
 id |                mc
----+----------------------------------
  1 | {c4ca4238a0b923820dcc509a6f75849b}
(1 row)
Time: 494.371 ms
postgres=# create index t_array_mc_idx on t_array using gin(mc);
CREATE INDEX
Time: 8195.387 ms (00:08.195)
postgres=# explain select * from t_array where mc @> ('{'||md5('1')||'}')::text[];
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Bitmap Heap Scan on t_array  (cost=29.40..3172.64 rows=2503 width=61)
         Recheck Cond: (mc @> ('{c4ca4238a0b923820dcc509a6f75849b}'::cstring)::text[])
         ->  Bitmap Index Scan on t_array_mc_idx  (cost=0.00..28.78 rows=2503 width=0)
               Index Cond: (mc @> ('{c4ca4238a0b923820dcc509a6f75849b}'::cstring)::text[])
(6 rows)
Time: 1.716 ms
postgres=# select * from t_array where mc @> ('{'||md5('1')||'}')::text[];
 id |                mc
----+----------------------------------
  1 | {c4ca4238a0b923820dcc509a6f75849b}
(1 row)
Time: 2.980 ms
```

- Btree_gin任意字段索引。

```
postgres=# create table gin_mul(f1 int, f2 int, f3 timestamp, f4 text, f5 numeric, f6 text);
postgres=# insert into gin_mul select random()*5000, random()6000, now()+((30000-60000random())||'
sec')::interval , md5(random()::text), round((random()*100000)::numeric,2), md5(random()::text) from ge
nerate_series(1,1000000);
postgres=# create extension btree_gin;
postgres=# create index gin_mul_gin_idx on gin_mul using gin(f1,f2,f3,f4,f5,f6);
#单字段查询
postgres=# explain select * from gin_mul where f1=10;
                                  QUERY PLAN
-------------------------------------------------------------------------------------

 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn002
   ->  Bitmap Heap Scan on gin_mul  (cost=11.51..369.70 rows=194 width=90)
         Recheck Cond: (f1 = 10)
         ->  Bitmap Index Scan on gin_mul_gin_idx  (cost=0.00..11.46 rows=194 width=0)
               Index Cond: (f1 = 10)
(6 rows)
postgres=#
postgres=# explain select * from gin_mul where f3='2019-02-18 23:01:01';
                                  QUERY PLAN
-------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Bitmap Heap Scan on gin_mul  (cost=10.01..12.02 rows=1 width=90)
         Recheck Cond: (f3 = '2019-02-18 23:01:01'::timestamp without time zone)
         ->  Bitmap Index Scan on gin_mul_gin_idx  (cost=0.00..10.01 rows=1 width=0)
```

```
            Index Cond: (f3 = '2019-02-18 23:01:01'::timestamp without time zone)
(6 rows)

postgres=# explain select * from gin_mul where f4='2364d9969c8b66402c9b7d17a6d5b7d3';
                              QUERY PLAN
-----------------------------------------------------------------------------------

 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Bitmap Heap Scan on gin_mul  (cost=10.01..12.02 rows=1 width=90)
         Recheck Cond: (f4 = '2364d9969c8b66402c9b7d17a6d5b7d3'::text)
         ->  Bitmap Index Scan on gin_mul_gin_idx  (cost=0.00..10.01 rows=1 width=0)
               Index Cond: (f4 = '2364d9969c8b66402c9b7d17a6d5b7d3'::text)
(6 rows)

postgres=# explain select * from gin_mul where f5=85375.30;
                              QUERY PLAN
-----------------------------------------------------------------------------------

 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Bitmap Heap Scan on gin_mul  (cost=10.01..12.02 rows=1 width=90)
         Recheck Cond: (f5 = 85375.30)
         ->  Bitmap Index Scan on gin_mul_gin_idx  (cost=0.00..10.01 rows=1 width=0)
               Index Cond: (f5 = 85375.30)
(6 rows)

#二个字段组合
postgres=# explain select * from gin_mul where f1=2 and f3='2019-02-18 16:59:52.872523';
                                    QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001
   ->  Bitmap Heap Scan on gin_mul  (cost=18.00..20.02 rows=1 width=90)
         Recheck Cond: ((f1 = 2) AND (f3 = '2019-02-18 16:59:52.872523'::timestamp without time zone))
         ->  Bitmap Index Scan on gin_mul_gin_idx  (cost=0.00..18.00 rows=1 width=0)
               Index Cond: ((f1 = 2) AND (f3 = '2019-02-18 16:59:52.872523'::timestamp without time zone))
(6 rows)
#三字段组合查询
postgres=# explain select * from gin_mul where f1=2 and f3='2019-02-18 16:59:52.872523' and f6='fa
627dc16c2bd026150afa0453a0991d';
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------

 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001
   ->  Bitmap Heap Scan on gin_mul  (cost=26.00..28.02 rows=1 width=90)
```

```
        Recheck Cond: ((f1 = 2) AND (f3 = '2019-02-18 16:59:52.872523'::timestamp without time zone)
AND (f6 = 'fa627dc16c2bd026150afa0453a0991d'::text))
        ->  Bitmap Index Scan on gin_mul_gin_idx  (cost=0.00..26.00 rows=1 width=0)
            Index Cond: ((f1 = 2) AND (f3 = '2019-02-18 16:59:52.872523'::timestamp without time zone)
AND (f6 = 'fa627dc16c2bd026150afa0453a0991d'::text))
(6 rows)
postgres=#
```

多字段索引

```
postgres=# create table t_mul_idx (f1 int,f2 int,f3 int,f4 int);
CREATE TABLE
Time: 308.109 ms
postgres=# create index t_mul_idx_idx on t_mul_idx(f1,f2,f3);
CREATE INDEX
Time: 108.734 ms
```

多字段使用注意事项：

- or查询条件bitmapscan最多支持两个不同字段条件。

```
 postgres=# insert into t_mul_idx select t,t,t,t from generate_series(1,1000000) as t;
INSERT 0 1000000
postgres=# analyze ;
ANALYZE

postgres=# explain select * from t_mul_idx where f1=1 or f2=2;
                                QUERYPLAN
-------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
  Node/s: dn001, dn002
  ->  Bitmap Heap Scan ont_mul_idx  (cost=7617.08..7621.07 rows=2width=16)
      Recheck Cond: ((f1 = 1) OR (f2 = 2))
      ->  BitmapOr  (cost=7617.08..7617.08 rows=2 width=0)
            ->  Bitmap Index Scan on t_mul_idx_idx  (cost=0.00..2.43 rows=1 width=0)
      Index Cond: (f1 = 1)
            ->  Bitmap Index Scan on t_mul_idx_idx  (cost=0.00..7614.65 rows=1 width=0)
                Index Cond: (f2 = 2)
(9 rows)

Time: 3.655 ms
postgres=# explain select * from t_mul_idxwhere f1=1 or f2=2 or f1=3;
                                QUERYPLAN
-------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
  Node/s: dn001, dn002
  ->  Bitmap Heap Scan ont_mul_idx  (cost=7619.51..7625.49 rows=3width=16)
      Recheck Cond: ((f1 = 1) OR (f2 = 2) OR (f1 = 3))
      ->  BitmapOr (cost=7619.51..7619.51 rows=3 width=0)
            ->  Bitmap Index Scan on t_mul_idx_idx  (cost=0.00..2.43 rows=1 width=0)
      Index Cond: (f1 = 1)
            ->  Bitmap Index Scan on t_mul_idx_idx  (cost=0.00..7614.65 rows=1 width=0)
```

```
                Index Cond: (f2 = 2)
             ->  Bitmap Index Scan on t_mul_idx_idx  (cost=0.00..2.43 rows=1 width=0)
                Index Cond: (f1 = 3)
(11 rows)

Time: 3.429 ms
postgres=# explain select * from t_mul_idxwhere f1=1 or f2=2 or f3=3;
                                QUERYPLAN
-------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
  Node/s: dn001, dn002
  ->  Seq Scan on t_mul_idx  (cost=0.00..12979.87 rows=3 width=16)
       Filter: ((f1 = 1) OR (f2 = 2) OR (f3 = 3))
(4 rows)

Time: 1.679 ms
```

- 如果返回字段全部在索引文件中，则只需要扫描索引，io开销会更少。

```
postgres=# explain select f1,f2,f3 from t_mul_idx where f1=1;
                                QUERYPLAN
-------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
  Node/s: dn001
  ->  Index Only Scan usingt_mul_idx_idx on t_mul_idx (cost=0.42..4.44 rows=1 width=12)
       Index Cond: (f1 = 1)
(4 rows)

Time: 1.564 ms
```

- 更新性能比单字段多索引文件要好。

```
--多字段
 postgres=# insert into t_simple_idx select t,t,t,t from generate_series(1,1000000) as t;
INSERT 0 1000000
Time: 7143.754 ms (00:07.144)
--单字段
 postgres=# insert into t_mul_idx select t,t,t,t from generate_series(1,1000000) as t;
INSERT 0 1000000
Time: 4034.208 ms (00:04.034)
```

- 多字段索引走非第一字段查询时性能比独立的单字段差。
    - 多字段

```
postgres=# select * from t_mul_idx where f1=1;
 f1 |f2 | f3 | f4
----+----+----+----
  1|  1 | 1 |  1
(1 row)

Time: 1.769 ms
postgres=# select * from t_mul_idx where f2=1;
 f1 |f2 | f3 | f4
----+----+----+----
  1|  1 | 1 |  1
(1 row)

Time: 25.423 ms
postgres=# select * from t_mul_idx where f3=1;
 f1 |f2 | f3 | f4
----+----+----+----
  1|  1 | 1 |  1
(1 row)

Time: 27.791 ms

--独立字段

postgres=# select * from t_simple_idx where f1=1;
 f1 |f2 | f3 | f4
----+----+----+----
  1|  1 | 1 |  1
(1 row)

Time: 1.530 ms
postgres=# select * from t_simple_idx where f2=1;
 f1 |f2 | f3 | f4
----+----+----+----
  1|  1 | 1 |  1
(1 row)

Time: 2.315 ms
postgres=# select * from t_simple_idx where f3=1;
 f1 |f2 | f3 | f4
----+----+----+----
  1|  1 | 1 |  1
(1 row)

Time: 2.390 ms
```

删除索引

```
postgres=# drop index t_appoint_id_idx;
DROP INDEX
```

修改表
修改表结构场景
修改表名

```
postgres=# alter table t rename to tbase;
ALTER TABLE
```

给表或字段添加注释

```
postgres=# comment on table tbase is 'TDSQL PG分布式关系型数据库系统';
COMMENT
postgres=# \dt+
                          List of relations
 Schema |       Name        | Type  | Owner | Size  |         Description
--------+-------------------+-------+-------+-------+-----------------------------
 public | t_appoint_col     | table | pgxz  | 16 kB |
 public | t_first_col_share | table | pgxz  | 16 kB |
 public | TBase             | table | pgxz  | 24 kB | TDSQLPG分布式关系型数据库系统
(3 rows)
postgres=# comment on column tbase.nickname is 'TDSQL PG昵称是大象';
COMMENT
postgres=# \d+ tbase
                                    Table "public.tbase"
  Column  |  Type   |                 Modifiers                  | Storage  | Stats target |   Description
----------+---------+--------------------------------------------+----------+--------------
+-----------------
 id       | integer | not null default nextval('t_id_seq'::regclass) | plain    |              |
 nickname | text    |                                            | extended |              | TDSQL PG昵称是大象
Has OIDs: no
Distribute By SHARD(id)
        Location Nodes: ALL DATANODES

postgres=#
```

给表增加字段

```
postgres=# alter table tbase add column age integer;
ALTER TABLE
postgres=# \d+ tbase
                                    Table "public.tbase"
  Column  |  Type   |                 Modifiers                 | Storage  | Stats target |   Description
----------+---------+-------------------------------------------+----------+--------------+-----------------
 id       | integer | not null default nextval('t_id_seq'::regclass) | plain    |              |
 nickname | text    |                                           | extended |              | TDSQL PG呢称是大象
 age      | integer |                                           | plain    |              |
Has OIDs: no
Distribute By SHARD(id)
        Location Nodes: ALL DATANODES
```

修改字段类型

```
postgres=# alter table tbase alter column age type float8;
ALTER TABLE
postgres=#  \d+ tbase
                                        Table "public.tbase"
 Column  |       Type       |                  Modifiers                  | Storage  | Stats target |   Description
---------+------------------+---------------------------------------------+----------+--------------+-----------------
 id      | integer          | not null default nextval('t_id_seq'::regclass) | plain    |              |
 nickname | text            |                                             | extended |              | TDSQL PG呢称是大象
 age     | double precision |                                             | plain    |              |
Has OIDs: no
Distribute By SHARD(id)
        Location Nodes: ALL DATANODES
```

修改字段默认值

```
postgres=# alter table tbase alter column age set default 0.0;
ALTER TABLE
postgres=#  \d+ tbase
                                        Table "public.tbase"
  Column  |       Type       |                  Modifiers                   | Storage  | Stats target |   Description
----------+------------------+----------------------------------------------+----------+--------------+-----------------
 id       | integer          | not null default nextval('t_id_seq'::regclass) | plain    |              |
 nickname | text             |                                              | extended |              | TDSQL PG昵称是大象
 age      | double precision | default 0.0                                  | plain    |              |
Has OIDs: no
Distribute By SHARD(id)
        Location Nodes: ALL DATANODES
```

删除字段

```
postgres=# alter table tbase drop column age;
ALTER TABLE
postgres=#  \d+ tbase
                                        Table "public.tbase"
  Column  |  Type   |                 Modifiers                  | Storage  | Stats target |   Description
----------+---------+--------------------------------------------+----------+--------------
+-----------------
 id       | integer | not null default nextval('t_id_seq'::regclass) | plain    |              |
 nickname | text    |                                            | extended |              | TDSQL PG昵称是大象
Has OIDs: no
Distribute By SHARD(id)
        Location Nodes: ALL DATANODES
```

添加主键

```
postgres=# ALTER TABLE t ADD CONSTRAINT t_id_pkey PRIMARY KEY (id) ;
ALTER TABLE
postgres=# \d+ t
                                Table "tbase.t"
 Column |  Type   | Collation | Nullable | Default | Storage  | Stats target | Description
--------+---------+-----------+----------+---------+----------+--------------+-------------
 id     | integer |           | not null |         | plain    |              |
 mc     | text    |           |          |         | extended |              |
Indexes:
    "t_id_pkey" PRIMARY KEY, btree (id)
Distribute By: SHARD(id)
Location Nodes: ALL DATANODES
```

删除主键

```
postgres=# ALTER TABLE t DROP CONSTRAINT t_id_pkey ;
ALTER TABLE
postgres=# \d+ t
                                Table "tbase.t"
 Column |  Type   | Collation | Nullable | Default | Storage  | Stats target | Description
--------+---------+-----------+----------+---------+----------+--------------+-------------
 id     | integer |           | not null |         | plain    |              |
 mc     | text    |           |          |         | extended |              |
Distribute By: SHARD(id)
Location Nodes: ALL DATANODES
```

如果是分区表，则删除主键要加上cascade，强制删除关联的子表主键。

重建主键

```
postgres=# \d t
              Table "public.t"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           | not null |
 mc     | text    |           |          |
Indexes:
"t_pkey" PRIMARY KEY, btree (id)

postgres=# CREATE UNIQUE INDEX CONCURRENTLY t_id_temp_idx ON t (id);
CREATE INDEX
postgres=#
postgres=# \d t
              Table "public.t"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           | not null |
 mc     | text    |           |          |
Indexes:
    "t_pkey" PRIMARY KEY, btree (id)
    "t_id_temp_idx" UNIQUE, btree (id)

postgres=#

postgres=# ALTER TABLE t DROP CONSTRAINT t_pkey, ADD CONSTRAINT t_pkey PRIMARY KEY USING
INDEX t_id_temp_idx;
NOTICE:  ALTER TABLE / ADD CONSTRAINT USING INDEX will rename index "t_id_temp_idx" to "t_pke
y"
ALTER TABLE
postgres=#
postgres=# \d t
              Table "public.t"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           | not null |
 mc     | text    |           |          |
Indexes:
    "t_pkey" PRIMARY KEY, btree (id)

postgres=#
```

添加外键

```
postgres=# create table t_p(f1 int not null,f2 int ,primary key(f1));
postgres=# create table t_f(f1 int not null,f2 int );
postgres=# ALTER TABLE t_f ADD CONSTRAINT t_f_f1_fkey FOREIGN KEY (f1) REFERENCES t_p (f1);
ALTER TABLE
postgres=# \d+ t_f
                                Table "public.t_f"
 Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------+-------------
 f1     | integer |           | not null |         | plain   |              |
 f2     | integer |           |          |         | plain   |              |
Foreign-key constraints:
    "t_f_f1_fkey" FOREIGN KEY (f1) REFERENCES t_p(f1)
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES
```

外键使用限制：

- 外键只是同一个节点内约束有效果，所以外键字段和对应主键字段必需都是表的分布键，否则由于数据分布于不同的节点内会导致更新失败。
- 分区表和冷热分区表也不支持外键，数据分区后位于不同的物理文件中，无法约束。

删除外键

```
postgres=# ALTER TABLE t_f DROP CONSTRAINT t_f_f1_fkey;
ALTER TABLE
```

修改表所属模式

```
postgres=# \dt t
       List of relations
 Schema | Name | Type  | Owner
--------+------+-------+-------
 tbase  | t    | table | tbase
(1 row)

postgres=# alter table t set schema public;
ALTER TABLE

postgres=# \dt t
       List of relations
 Schema | Name | Type  | Owner
--------+------+-------+-------
 public | t    | table | tbase
(1 row)
```

修改表所属用户

```
postgres=# \dt tbase
       List of relations
 Schema | Name  | Type  | Owner
--------+-------+-------+-------
 public | tbase | table | tbase
(1 row)

postgres=# alter table tbase owner to pgxz;
ALTER TABLE

postgres=# \dt tbase
       List of relations
 Schema | Name  | Type  | Owner
--------+-------+-------+-------
 public | tbase | table | pgxz
(1 row)
```

修改字段名

```
postgres=# \d+ tbase
                                Table "public.tbase"
 Column |         Type          | Collation | Nullable | Default | Storage  | Stats target | Description
--------+-----------------------+-----------+----------+---------+----------+--------------+-------------
 id     | integer               |           | not null |         | plain    |              |
 city   | character varying(50) |           |          |         | extended |              |
Distribute By: SHARD(id)
Location Nodes: dn01

postgres=# alter table tbase rename city to cityname;
ALTER TABLE

postgres=# \d+ tbase
                                Table "public.tbase"
  Column  |         Type          | Collation | Nullable | Default | Storage  | Stats target | Description
----------+-----------------------+-----------+----------+---------+----------+--------------+-------------
 id       | integer               |           | not null |         | plain    |              |
 cityname | character varying(50) |           |          |         | extended |              |
Distribute By: SHARD(id)
Location Nodes: dn01
```

修改表的填充率

一个表的填充率(fillfactor)默认值是100。INSERT 操作仅按照填充率指定的百分率填充表页。每个页上的剩余空间将用于在该页上的行更新，这就使得 UPDATE 有机会在同一页上放置同一条记录的新版本，这比把新版本放置在其它页上更有效。对于一个从不更新的表将填充因子设为 100 效率最高，但是对于频繁更新的表，较小的填充率则更加有效，但小的填充率会占用更大有空间。
在PostgreSQL中，当更新一行数据时，实际上旧行并没有删除，只是插入了一行新数据。如果这个表其他列上有索引，而更新的列上没有索引，因为新行的物理位置发生变化，因此需要更新索引，这将导致性能下降。为了解决这个问题，PostgreSQL引入了Heap Only Tuple（HOT）技术，如果更新后的新行和旧行位于同一个数据块内，则旧行会有一个指针指向新行，这样就不用更新索引了，通过索引访问到旧行数据，进而访问到新行数据。

```
postgres=# \d+ t1
                          Table "tbase_pg_proc.t1"
 Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------+-------------
 f1     | integer |           | not null |         | plain   |              |
 f2     | integer |           |          |         | plain   |              |
Indexes:
    "t1_f1_idx" btree (f1)
Distribute By: SHARD(f1)
Location Nodes: dn002, dn001

postgres=# alter table t1 set(fillfactor=80);
ALTER TABLE
postgres=#  \d+ t1
                          Table "tbase_pg_proc.t1"
 Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------+-------------
 f1     | integer |           | not null |         | plain   |              |
 f2     | integer |           |          |         | plain   |              |
Indexes:
    "t1_f1_idx" btree (f1)
Distribute By: SHARD(f1)
Location Nodes: dn002, dn001
Options: fillfactor=80

postgres=#
```

添加触发器

- INSERT触发器。

```
postgres=# create table t_trigger(f1 int,f2 int);
CREATE TABLE
postgres=# CREATE OR REPLACE FUNCTION t_trigger_insert_trigger_func () RETURNS trigger AS
$body$
BEGIN
  if NEW.f2
        NEW.f2=0;
     endif;
  RETURN NEW;
 END;
$body$
LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# CREATE TRIGGER t_trigger_insert_trigger BEFORE INSERT ON t_trigger FOR EACH ROW EX
ECUTE PROCEDURE t_trigger_insert_trigger_func();
CREATE TRIGGER
```

上面触发器的作用是在插入记录时，如果f2字段值小于0时，则将f2的值修改成0，效果如下

```
postgres=# insert into t_trigger values(1,-1);
INSERT 0 1
postgres=# select * from t_trigger;
 f1 |f2
----+----
  1|  0
```

- UPDATE触发器。

```
postgres=# CREATE OR REPLACE FUNCTION t_trigger_update_trigger_func () RETURNS trigger AS
$body$
BEGIN
  if NEW.f2
      NEW.f2=OLD.f2;
     endif;
  RETURN NEW;
END;
$body$
LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# CREATE TRIGGER t_trigger_update_trigger BEFORE UPDATE ON t_trigger FOR EACH ROW E
XECUTE PROCEDURE t_trigger_update_trigger_func();
```

```
CREATE TRIGGER
postgres=#
```

上面触发器的作用是在修改记录时，如果f2字段值小于0时，则将f2的值修改成原值，效果如下

```
postgres=# update t_trigger set f2=-1 where f1=1;
UPDATE 1
postgres=# select * from t_trigger;
 f1 |f2
----+----
  1|  0

postgres=# update t_trigger set f2=1 where f1=1;
UPDATE 1
postgres=# select * from t_trigger;
 f1 |f2
----+----
  1|  1
```

- DELETE触发器。

```
postgres=# CREATE OR REPLACE FUNCTION t_trigger_delete_trigger_func () RETURNS trigger AS
$body$
BEGIN
    if OLD.f2=0 then
        RETURN NULL;
       endif;
    RETURN OLD;
END;
$body$
LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# CREATE TRIGGER t_trigger_delete_trigger BEFORE DELETE ON t_trigger FOR EACH ROW EX
ECUTE PROCEDURE t_trigger_delete_trigger_func();
CREATE TRIGGER
```

上面触发器的作用是在删除记录时，如果f2字段值等于0时，则不删除记录，效果如下

```
postgres=# delete from t_trigger where f2=0;
DELETE 0
postgres=# select * from t_trigger;
 f1 |f2
----+----
  1|  0

postgres=# update t_trigger set f2=10;
```

```
UPDATE 1
postgres=# select * from t_trigger;
 f1 |f2
----+----
  1 |10
(1 row)

postgres=# delete from t_trigger where f1=1;
DELETE 1
postgres=# select * from t_trigger;
 f1 |f2
----+----
(0 rows)
```

- 多个事件。

```
postgres=# create table t_trigger_mulevent(f1 int,f2 int);
CREATE TABLE
CREATE OR REPLACE FUNCTION t_trigger_mulevent_func () RETURNS trigger AS
$body$
BEGIN
   if NEW.f2
          NEW.f2=0;
       endif;
   RETURN NEW;
END;
$body$
LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#

postgres=# CREATE TRIGGER t_trigger_insert_update_trigger BEFORE INSERT OR UPDATE ON t_trigger
_mulevent FOR EACH ROW EXECUTE PROCEDURE t_trigger_mulevent_func();
CREATE TRIGGER
```

上面触发器的作用是在插入和更新记录时，如果f2字段值小于0时，则f2值重置为0，效果如下

```
postgres=# insert into t_trigger_mulevent values(1,-10);
INSERT 0 1
postgres=# select * from t_trigger_mulevent;
 f1 |f2
----+----
  1|  0
(1 row)

postgres=# update t_trigger_mulevent setf2=-10;
```

```
UPDATE 1
postgres=# select * from t_trigger_mulevent;
 f1 |f2
----+----
  1|  0
(1 row)
```

删除触发器

```
#删除触发器
postgres=# DROP TRIGGER t_trigger_insert_update_trigger ON t_trigger_mulevent;
DROP TRIGGER
postgres=#
#删除触发器函数
postgres=# drop function t_trigger_mulevent_func();
DROP FUNCTION
postgres=#
```

truncate普通表

```
postgres=# truncate table t1;
TRUNCATE TABLE
#也可以一次truncate多个数据表
postgres=# truncate table t1,t2;
TRUNCATE TABLE
```

删除表

使用命令drop table删除表
示例：
当表存在时删除

```
drop table if exists tbase;
```

表管理场景
不用指定shard key 建表方式

不指定shard key建表方法，系统默认使用第一个字段做为表的shard key。

```
postgres=# create table t_first_col_share(id serial not null,nickname text);
CREATE TABLE
postgres=# \d+ t_first_col_share
                                    Table "public.t_first_col_share"
 Column  |  Type   |                       Modifiers                       | Storage  | Stats target | Description
---------+---------+-------------------------------------------------------+----------+--------------+-------------
 id       | integer | not null default nextval('t_first_col_share_id_seq'::regclass) | plain    |              |
 nickname | text    |                                                       | extended |              |
Has OIDs: no
Distribute By SHARD(id)
        Location Nodes: ALL DATANODES
```

分布键选择原则：

- 分布键只能选择一个字段。
- 如果有主键，则选择主键做分布键。
- 如果主键是复合字段组合，则选择字段值选择性多的字段做分布键。
- 也可以把复合字段拼接成一个新的字段来做分布键。
- 没有主键的可以使用UUID来做分布键。
- 总之一定要让数据尽可能的分布得足够散。

指定shard key 建表方式

```
postgres=# create table t_appoint_col(id serial not null,nickname text) distribute by shard(nickname);
CREATE TABLE
postgres=# \d+ t_appoint_col
                                    Table "public.t_appoint_col"
  Column  |  Type   |                       Modifiers                        | Storage  | Stats target | Description
----------+---------+------------------------------------------------------------+----------+--------------+-------------
 id       | integer | not null default nextval('t_appoint_col_id_seq'::regclass) | plain    |              |
 nickname | text    |                                                            | extended |              |
Has OIDs: no
Distribute By SHARD(nickname)
        Location Nodes: ALL DATANODES
```

指定group 建表方式

```
postgres=# create table t (id integer,nc text) distribute by shard (id) to group default_group;
CREATE TABLE

postgres=# \d+ t
                    Table "public.t"
 Column |  Type   | Modifiers | Storage  | Stats target | Description
--------+---------+-----------+----------+--------------+-------------
 id     | integer |           | plain    |              |
 nc     | text    |           | extended |              |
Has OIDs: no
Distribute By SHARD(id)
        Location Nodes: ALL DATANODES
```

复制表

复制表是所有dn节点都存储一份相同的数据

```
postgres=# create table t_rep (id int,mc text) distribute by replication to group default_group;
CREATE TABLE
postgres=# insert into t_rep values(1,'TBase'),(2,'pgxz');
INSERT 0 2
postgres=# EXECUTE DIRECT ON (dn001) 'select * from t_rep';
 id |  mc
----+-------
  1 | TBase
  2 | pgxz
(2 rows)

postgres=# EXECUTE DIRECT ON (dn002) 'select * from t_rep';
 id |  mc
----+-------
  1 | TBase
  2 | pgxz
(2 rows)
```

我们可以看到所有节点都保存了一份相同的数据。

列存表管理

创建列存表

```
postgres=# create table t_col_test(f1 int,f2 varchar(32),f3 date) with(orientation='column');
CREATE TABLE
postgres=# \d+ t_col_test
                                Table "public.t_col_test"
 Column |         Type          | Collation | Nullable | Default | Storage  | Stats target | Description
--------+-----------------------+-----------+----------+---------+----------+--------------+-------------
 f1     | integer               |           |          |         | plain    |              |
 f2     | character varying(32) |           |          |         | extended |              |
 f3     | date                  |           |          |         | plain    |              |
Options: orientation=column
```

指定压缩类型

```
postgres=# create table t_col_compress
(
f1 int encoding(compress_method='delta'),
f2 varchar(32)ENCODING(compress_method='zlib',compress_level=9) ,
f3 date ENCODING(compress_method='zlib',compress_level=9)
) with(orientation='column');
CREATE TABLE
postgres=# \d+ t_col_compress
                                Table"public.t_col_compress"
 Column |        Type         | Collation | Nullable | Default |Storage  | Stats target | Description
--------+---------------------+-----------+----------+---------+----------+--------------+-------------
 f1     |integer              |           |          |         | plain    |              |
 f2     |character varying(32) |           |          |         | extended |              |
 f3     |date                 |           |          |         | plain    |              |
Options: orientation=column

postgres=#
```

指定表列字段的压缩方式及压缩等级创建表
目前支持压缩方式compress_method包括（delta、zstd、zlib、rle、bitpack），compress_level表示缩级别；压缩级别越大，压缩比越大，压缩时间越长，压缩级别越高越消耗cpu。
轻量压缩为自研实现算法，包括rle、delta、bitpack，他们均只有一种压缩级别，且只能压缩数值类型。
其中rle主要针对大量重复的数据，比如1111112222233311111进行压缩。delta主要针对递增或者递减的数据，比如固定的时间类型(今天明天后天)，或者数值类型12345678进行压缩。bitpack针对数值比较小的数据，比如创建表的时候是integer，但实际存储的数值只是1-100。
创建表的时候指定压缩类型，但是实际存储的时候有没有用到指定的压缩类型是会自动适应调整的，有写场景可能导致最终存储没有使用压缩，如：压缩算法使用不当，比如针对text类型使用delta。
约束数据对比，100万行记录。

```
postgres=# \dt+
                          List ofrelations
 Schema |          Name           | Type  | Owner |   Size    | Description
--------+-------------------------+-------+-------+-----------+-------------
 public | t_col_compress          | table | tbase | 104 MB    |
 public | t_col_test              | table | tbase | 119 MB    |
 public | t_row_test              | table | tbase | 97 MB     |
```

下面对比一下重复率比较高的，测试表如下：

- 表定义
  表定义

行存	列存	列存（压缩）
create table t_row_normal (f1 int, f2 text , f3 text , f4 text , f5 text , f6 text , f7 text , f8 text , f9 text , f10 text) ;	create table t_col_normal (f1 int, f2 text , f3 text , f4 text , f5 text , f6 text , f7 text , f8 text , f9 text , f10 text) with(orientation='column');	create table t_col_compress (f1 int, f2 text ENCODING(compress_method='zstd',compress_level=1) , f3 text ENCODING(compress_method='zstd',compress_level=1) , f4 text ENCODING(compress_method='zstd',compress_level=1) , f5 text ENCODING(compress_method='zstd',compress_level=1) , f6 text ENCODING(compress_method='zstd',compress_level=1) , f7 text ENCODING(compress_method='zstd',compress_level=1) , f8 text ENCODING(compress_method='zstd',compress_level=1) , f9 text ENCODING(compress_method='zstd',compress_level=1) , f10 text ENCODING(compress_method='zstd',compress_level=1)) with(orientation='column');

- 测试脚本

insert into t_row_normalselectt,repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3)from generate_series(1,1000000) as t;

insert into t_col_normalselectt,repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3)from generate_series(1,1000000) as t;

insert into t_col_compressselectt,repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3)from generate_series(1,1000000) as t;

- 用时及容量对比
  对比

项目	行存	列存	列存（带压缩）
大小	521MB	896MB	480MB
生成数据时间	8.029	12.801	11.818

下面对比一下重复率非常高的，测试表如下：

- 测试脚本

insert into t_row_normalselectt,repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3)from generate_series(1,1000000) as t;

insert into t_col_normalselectt,repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3)from generate_series(1,1000000) as t;

insert into t_col_compressselect t,repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3),repeat(random()::text,3)from generate_series(1,1000000) as t;

- 用时及容量对比
  对比

项目	行存	列存	列存（带压缩）
大小	1898MB	1976MB	502MB

项目	行存	列存	列存（带压缩）
生成数据时间	12.680	26.019	19.261

tdx外表管理

tdx服务

- 部署tdx服务

[tbase@master ~]$ mkdir -p/data/tbase/tdx/tbase_pgxz
[tbase@master ~]$ mkdir -p/data/tbase/tdx/data
[tbase@master ~]$
#复制一份TDSQL PG二进制程序到/data/tbase/tdx/tbase_pgxz,然后配置环境变量如下
export PGXZ_HOME=/data/tbase/tdx/tbase_pgxz
export PATH=$PGXZ_HOME/bin:$PATH
exportLD_LIBRARY_PATH=$PGXZ_HOME/lib:${LD_LIBRARY_PATH}

- 启动服务

  tdx -d /data/tbase/tdx/data -p 8088 -l/data/tbase/tdx/tdx.log

  nohup tdx -d /data/tbase/tdx/data -p 8088-l /data/tbase/tdx/tdx.log & # 放后面运行

  参数说明

参数	说明
-d	工作目录，后面的数据文件存放地方
-p	服务port
-l	日志文件名称

- 停止服务

[tbase@master ~]$ ps -ewf |grep tdx |grep -v color
tbase 20279 13747 0 14:55 pts/1 00:00:00 tdx -d /data/tbase/tdx/data -p8088 -l /data/tbase/tdx/td
x.log
[tbase@master ~]$ kill 20279

创建exttable_fdw

```
postgres=# create extension exttable_fdw;
CREATE EXTENSION
```

创建外表

- Csv格式

```
postgres=# CREATE EXTERNAL TABLE t_table_csv
(
f1 int,
f2 text ,
f3 text ,
f4 text ,
f5 text ,
f6 text ,
f7 text ,
f8 text ,
f9 text ,
f10 text
)
LOCATION ('tdx://172.16.0.23:8088/t_table.csv')
FORMAT 'csv';
```

> 说明：
>
> (header DELIMITER '|') -- csv文件第一行为列的名称，列分隔符为竖线(默认为逗号)。

- text格式

```
postgres=# CREATE EXTERNAL TABLE t_table_text
(
f1 int,
f2 text ,
f3 text ,
f4 text ,
f5 text ,
f6 text ,
f7 text ,
f8 text ,
f9 text ,
f10 text
)
LOCATION ('tdx://172.16.0.23:8088/t_table.txt')
FORMAT 'text'  (delimiter E'\x0E' null '' eol E'\x0F\n');
```

> 说明：

> delimiter E'\x0E' 表示列分隔符为音乐符，空串''转换为null，行分隔符为太阳符加回车。

删除外表

```
postgres=# drop EXTERNAL table t_table_csv;
DROP FOREIGN TABLE
```

将外表数据导入到物理表中

```
postgres=# insert into t_col_compress select * from t_table_text;
INSERT 0 1000000
Time: 19241.301 ms (00:19.241)
```

导入数据对比，导入文件大小8.6G，行数1700万

对比项目

对比项目	行存	列存	列存（带压缩）
从tdx外表导入	100s	158s	137s
copy	129s	198	169s
大小	9.9G	12G	2.3G

外表管理
postgres_fdw
创建postgres_fdw extension

```
postgres=# create extension postgres_fdw ;
CREATE EXTENSION
postgres=#
```

创建一个fdw server

```
postgres=# CREATE SERVER postgres_fdw_postgres_db FOREIGN DATA WRAPPER postgres_fdw  OPTI
ONS (host '172.16.64.14', port '11345', dbname 'postgres');
CREATE SERVER
```

这里指定了外部服务节点的连接信息。

配置user mapping

```
postgres=# CREATE USER MAPPING FOR tbase server postgres_fdw_postgres_db OPTIONS (user 'tbase
', password '');
CREATE USER MAPPING
```

创建外表

```
postgres=# CREATE FOREIGN TABLE public.t1_fdw
(
  f1 int,
  f2 int
)
SERVER postgres_fdw_postgres_db OPTIONS (schema_name 'public', table_name 't1');
CREATE FOREIGN TABLE
```

访问外表

```
postgres=# select * from t1_fdw;
 f1 | f2
----+----
  1 |  1
(1 row)

#执行计划，数据是直接在cn上面访问fdw的外表
postgres=# explain select * from t1_fdw;
                         QUERY PLAN
------------------------------------------------------------------
 Foreign Scan on t1_fdw  (cost=100.00..153.86 rows=1462 width=8)
(1 row)

Time: 0.777 ms
postgres=#
```

oracle_fdw
创建oracle_fdw extension

```
postgres=# create extension oracle_fdw;
CREATE EXTENSION
```

创建一个fdw server

postgres=# create server oradb_215 foreign data wrapper oracle_fdw options(dbserver '172.16.0.17:1521/ora12c');
CREATE SERVER
postgres=#

配置user mapping

```
postgres=# create user mapping for tbase server oradb_215 options(user 'tbase',password 'tbase201
8');
CREATE USER MAPPING
postgres=#
```

创建外表

```
postgres=# create foreign table t1
(
 f1 integer,
 f2 varchar(50)
) server oradb_215 options(schema 'TBASE',table 'T1');
CREATE FOREIGN TABLE
```

访问外表

- 查看外表结构

```
postgres=# \d+ t1
                                    Foreign table "public.t1"
 Column |         Type          | Collation | Nullable | Default | FDW options | Storage  | Stats target | Descrip
tion
--------+-----------------------+-----------+----------+---------+-------------+----------+--------------
+-------------
 f1     | integer               |           | not null |         |             | plain    |              |
 f2     | character varying(50) |           |          |         |             | extended |              |
Server: oradb_215
FDW options: (schema 'TBASE', "table" 'T1')
```

- 访问数据

```
postgres=# select * from t1;
 f1 |  f2
----+-------
  1 | TDSQL PG
  1 | pgxz
(2 rows)
```

dblink外表
oracle_dblink
创建oracle_fdw

```
postgres=# create extension oracle_fdw ;
CREATE EXTENSION
```

创建dblink

```
postgres=# CREATE DATABASE LINK abc CONNECT TO "TBASE" IDENTIFIED BY 'tbase2018' USING '(DE
SCRIPTION =
(ADDRESS = (PROTOCOL = TCP)(HOST = 172.16.0.17)(PORT = 1521))
(CONNECT_DATA =
(SERVER = DEDICATED)
(SERVICE_NAME = ora12c)
)
)';
CREATE DATABASE LINK

#这里用户名需要使用大写
```

查询dblink系统表

```
postgres=# select * from pg_dblink ;
 dblinkname | dblinkowner | username |           created           |    host     | port | dblinkkind |   dblink_fo
reign_server
------------+-------------+----------+-----------------------------+-------------+------+------------
+---------------------------
 abc        | tbase       | TBASE    | 2021-07-08 16:38:38.016606+08 | 172.16.0.17 | 1521 |          0 | ora_172_
16_0_17_1521_tbase
(1 row)
```

访问远程表

```
postgres=# select * from t1@abc;
 f1 | f2 | f3
----+----+----
  1 |  1 |  1
(1 row)
```

删除dblink

```
postgres=# drop DATABASE LINK abc;
```

tbase_dblink
创建postgres_fdw

```
postgres=# create extension postgres_fdw ;
CREATE EXTENSION
```

创建dblink

```
#配置要创建的dblink类型为postgresql dblink
postgres=# set dblink_types = 'postgresql';
SET
postgres=# CREATE DATABASE LINK tbase_dblink CONNECT TO "tbase" IDENTIFIED BY 'tbase@2017' U
SING
'(DESCRIPTION =
      (ADDRESS = (PROTOCOL = TCP)(HOST = 172.16.0.23)(PORT = 11379))
      (CONNECT_DATA =(SERVER = DEDICATED)(SERVICE_NAME = postgres)(SCHEMA_NAME = publi
c))
      (COLLATE = YES)(DEFAULT = YES)(NOT_NULL = YES)
)';
CREATE DATABASE LINK
postgres=#
```

查询dblink系统表

```
postgres=# select * from pg_dblink where dblinkname='tbase_dblink';
  dblinkname  | dblinkowner | username |            created            |    host     | port  | dblinkkind |         dbl
ink_foreign_server
--------------+-------------+----------+-------------------------------+-------------+-------+------------
+----------------------------------------
 tbase_dblink | tbase       | tbase    | 2021-08-05 15:23:28.478796+08 | 172.16.0.23 | 11379 |          0 | pos
tgres_public_172_16_0_23_11379_tbase
(1 row)
```

访问远程表

```
postgres=# select * from t2@tbase_dblink;
 f1 | f2
----+----
  1 |  1
(1 row)
```

删除dblink

```
postgres=# drop DATABASE LINK tbase_dblink;
```

指定模式创建表

```
postgres=# create table public.t(id int,mc text);
```

使用将查询结果创建数据表

```
postgres=# create table t(id int,mc text) distribute by shard(mc);
CREATE TABLE
postgres=# insert into t values(1,'TDSQL PG');
INSERT 0 1
postgres=#  create table t_as as select * from t;
INSERT 0 1
postgres=# select * from t_as;
 id |  mc
----+-------
  1 | TDSQL PG
(1 row)

postgres=# \d+ t
                                Table "tbase.t"
 Column |  Type   | Collation | Nullable | Default | Storage  | Stats target | Description
--------+---------+-----------+----------+---------+----------+--------------+-------------
 id     | integer |           |          |         | plain    |              |
 mc     | text    |           |          |         | extended |              |
Distribute By: SHARD(mc)
Location Nodes: ALL DATANODES

postgres=# \d+ t_as
                               Table "tbase.t_as"
 Column |  Type   | Collation | Nullable | Default | Storage  | Stats target | Description
--------+---------+-----------+----------+---------+----------+--------------+-------------
 id     | integer |           |          |         | plain    |              |
 mc     | text    |           |          |         | extended |              |
Distribute By: SHARD(id)
Location Nodes: ALL DATANODES

postgres=#
```

删除数据表

- 删除当前模式下的数据表

```
postgres=# drop table t;
DROP TABLE
```

- 删除某个模式下数据表

```
postgres=# drop table public.t;
DROP TABLE
```

- 删除数据表，不存在时不执行，不报错

```
postgres=# drop table IF EXISTS t;
NOTICE:  table "t" does not exist, skipping
DROP TABLE
```

- 使用CASCADE无条件删除数据表

```
postgres=# create view tbase_schema.t1_view as select * from tbase_schema.t1 ;
CREATE VIEW
postgres=# drop table tbase_schema.t1 ;
ERROR:  cannot drop table tbase_schema.t1 because other objects depend on it
DETAIL:  view tbase_schema.t1_view depends on table tbase_schema.t1
HINT:  Use DROP ... CASCADE to drop the dependent objects too.

postgres=# drop table tbase_schema.t1 CASCADE;
NOTICE:  drop cascades to view tbase_schema.t1_view
DROP TABLE
postgres=#
```

copy的使用

实验表结构及数据

```
postgres=# \d+ t
                                Table "public.t"
 Column |            Type             |   Modifiers   | Storage  | Stats target | Description
--------+-----------------------------+---------------+----------+--------------+-------------
 f1     | integer                     | not null      | plain    |              |
 f2     | character varying(32)       | not null      | extended |              |
 f3     | timestamp without time zone | default now() | plain    |              |
 f4     | integer                     |               | plain    |              |
Has OIDs: yes
Distribute By SHARD(f1)
        Location Nodes: ALL DATANODES
#数据测试过程可以行再录入修改
postgres=# select * from t;
 f1 |  f2   |           f3            | f4
----+-------+-------------------------+----
  3 | pgxz  | 2017-10-28 18:24:05.645691 |
  1 | TDSQL PG |                         |  7
  2 |       | 2017-10-28 18:24:05.643102 |  3
(3 rows)
```

copy to用法详解--复制数据到文件中
导出所有列

```
postgres=# copy public.t to '/data/pgxz/t.txt';
COPY 3
postgres=# \! cat /data/pgxz/t.txt
1       TDSQL PG    \N      7
2               2017-10-28 18:24:05.643102      3
3       pgxz    2017-10-28 18:24:05.645691      \N
```

默认生成的文件内容为表的所有列，列与列之间使用tab分隔开来。NULL值生成的值为\N。

导出部分列

```
postgres=# copy public.t(f1,f2) to '/data/pgxz/t.txt';
COPY 3
postgres=# \! cat /data/pgxz/t.txt
1	TDSQL PG
2
3	pgxz

#只导出f1和f2列
```

导出查询结果

```
postgres=# copy (select f2,f3 from public.t order by f3) to  '/data/pgxz/t.txt';
COPY 3
postgres=# \! cat /data/pgxz/t.txt
        2017-10-28 18:24:05.643102
pgxz    2017-10-28 18:24:05.645691
Tbase   \N
postgres=#
#查询可以是任何复杂查询
```

指定生成文件格式

- 生成csv格式。

```
postgres=# copy public.t to '/data/pgxz/t.txt' with csv;
COPY 3
postgres=# \! cat /data/pgxz/t.txt
1,TDSQL PG,,7
2,pgxc,2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
```

- 生成二进制格式。

```
postgres=# copy public.t to '/data/pgxz/t.txt' with binary;
COPY 3
postgres=# \1
postgres=# \! cat /data/pgxz/t.txt
PGCOPY
TDSQL PG
```

默认为TEXT格式。

使用delimiter指定列与列之间的分隔符

```
postgres=# copy public.t to '/data/pgxz/t.txt' with delimiter '@';
COPY 3
postgres=# \! cat /data/pgxz/t.txt
1@TDSQL PG@\N@7
2@pgxc@2017-10-28 18:24:05.643102@3
3@pgxz@2017-10-28 18:24:05.645691@\N
postgres=# copy public.t to '/data/pgxz/t.txt' with csv delimiter '@';
COPY 3
postgres=# \! cat /data/pgxz/t.txt
1@TDSQL PG@@7
2@pgxc@2017-10-28 18:24:05.643102@3
3@pgxz@2017-10-28 18:24:05.645691@
postgres=# copy public.t to '/data/pgxz/t.txt' with csv delimiter '@@';
ERROR:  COPY delimiter must be a single one-byte character
postgres=# copy public.t to '/data/pgxz/t.txt' with binary delimiter '@';
ERROR:  cannot specify DELIMITER in BINARY mode
```

指定分隔文件各列的字符。文本格式中默认是一个制表符， 而CSV格式中默认是一个逗号。分隔符必须是一个单一的单字节字符，即汉字是不支持的。使用binary格式时不允许这个选项。

NULL值的处理

```
postgres=# copy public.t to '/data/pgxz/t.txt' with NULL 'NULL';
COPY 3
postgres=# \! cat /data/pgxz/t.txt
1       TDSQL PG    NULL    7
2       pgxc    2017-10-28 18:24:05.643102      3
3       pgxz    2017-10-28 18:24:05.645691      NULL
postgres=# copy public.t to '/data/pgxz/t.txt' with CSV NULL 'NULL';
COPY 3
postgres=# \! cat /data/pgxz/t.txt
1,TDSQL PG,NULL,7
2,pgxc,2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,NULL
postgres=# copy public.t to '/data/pgxz/t.txt' with binary NULL 'NULL';
ERROR:  cannot specify NULL in BINARY mode
postgres=#
```

记录值为NULL时导出为NULL字符。使用binary格式时不允许这个选项。

生成列标题名

```
postgres=# copy public.t to '/data/pgxz/t.txt' with csv HEADER;
COPY 3
postgres=# \! cat /data/pgxz/t.txt
f1,f2,f3,f4
1,TDSQL PG,,7
2,pgxc,2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
postgres=# copy public.t to '/data/pgxz/t.txt' with  HEADER;
ERROR:  COPY HEADER available only in CSV mode
```

只有使用 CSV格式时才允许这个选项。

导出oids系统列

```
postgres=# drop table t;
DROP TABLE
postgres=# CREATE TABLE t (
postgres(#     f1 integer NOT NULL,
postgres(#     f2 text NOT NULL,
postgres(#     f3 timestamp without time zone,
postgres(#     f4 integer
postgres(# )
postgres-# with oids DISTRIBUTE BY SHARD (f1);
CREATE TABLE
postgres=# copy t from '/data/pgxz/t.txt' with csv ;
COPY 3
postgres=# select * from t;
 f1 |      f2       |           f3            | f4
----+---------------+-------------------------+----
  1 | TDSQL PG      |                         |  7
  2 | pg'",      xc | 2017-10-28 18:24:05.643102 |  3
  3 | pgxz          | 2017-10-28 18:24:05.645691 |
(3 rows)
postgres=# copy t to  '/data/pgxz/t.txt' with oids ;
COPY 3
postgres=# \! cat /data/pgxz/t.txt
35055   1       TDSQL PG    \N      7
35056   2       pg'",       xc    2017-10-28 18:24:05.643102      3
35177   3       pgxz     2017-10-28 18:24:05.645691      \N
```

创建表时使用了with oids才能使用oids 选项。

使用quote自定义引用字符

```
postgres=# copy t to  '/data/pgxz/t.txt' with csv;
COPY 3
postgres=#  \! cat /data/pgxz/t.txt
1,TDSQL PG,,7
2,"pg'"",      xc%",2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
#默认引用字符为“双引号”
postgres=# copy t to  '/data/pgxz/t.txt' with quote '%' csv;
COPY 3
postgres=# \! cat /data/pgxz/t.txt
1,TDSQL PG,,7
2,%pg'",      xc%%%,2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
#上面指定了引用字符为百分号，系统自动把字段值为%的字符替换为双个%
postgres=# copy t to  '/data/pgxz/t.txt' with quote '%';
ERROR:  COPY quote available only in CSV mode
#只有使用 CSV格式时才允许这个选项。
postgres=# copy t to  '/data/pgxz/t.txt' with quote '%%' csv;
ERROR:  COPY quote must be a single one-byte character
postgres=#
#引用字符必须是一个单一的单字节字符，即汉字是不支持的。
```

使用escape自定义逃逸符

```
postgres=# copy t to  '/data/pgxz/t.txt' with quote '%' csv;
COPY 3
postgres=# \! cat /data/pgxz/t.txt
1,TDSQL PG,,7
2,%pg'",      xc%%%,2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
#不指定escape逃逸符，默认和QUOTE值一样
postgres=# copy t to  '/data/pgxz/t.txt' with quote '%' escape '@' csv;
COPY 3
postgres=# \! cat /data/pgxz/t.txt
1,TDSQL PG,,7
2,%pg'",      xc@%%,2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
#指定逃逸符为'@'
postgres=# copy t to  '/data/pgxz/t.txt' with quote '%' escape '@@' csv;
ERROR:  COPY escape must be a single one-byte character
#这必须是一个单一的单字节字符。
postgres=# copy t to  '/data/pgxz/t.txt' with quote '%' escape '@';
ERROR:  COPY quote available only in CSV mode
```

强制给某个列添加引用字符

```
postgres=# copy t to  '/data/pgxz/t.txt' (format 'csv',force_quote (f1,f2));
COPY 3
postgres=# \! cat /data/pgxz/t.txt
"1","TDSQL PG",,7
"2","pg'"",      xc%",2017-10-28 18:24:05.643102,3
"3","pgxz",2017-10-28 18:24:05.645691,
#指定2列强制添加引用字符
postgres=# copy t to  '/data/pgxz/t.txt' (format 'csv',force_quote (f1,f4,f3,f2));
COPY 3
postgres=# \! cat /data/pgxz/t.txt
"1","TDSQL PG",,"7"
"2","pg'"",      xc%","2017-10-28 18:24:05.643102","3"
"3","pgxz","2017-10-28 18:24:05.645691",
#指定4列强制添加引用字符，字段的顺序可以任意排列
postgres=# copy t to  '/data/pgxz/t.txt' (format 'text',force_quote (f1,f2,f3,f4));
ERROR:  COPY force quote available only in CSV mode
postgres=#
#只有使用CSV格式时才允许这个选项。
```

使用encoding指定导出文件内容编码

```
postgres=# copy t to  '/data/pgxz/t.csv' (encoding utf8);
COPY 3
#导出文件编码为UTF8
postgres=# copy t to  '/data/pgxz/t.csv' (encoding gbk);
COPY 3
postgres=#
#导出文件编码为gbk
#使用set_client_encdoing to gbk;也可以将文件的内容设置为需要的编码，如下所示
postgres=# set client_encoding to gbk;
SET
postgres=# copy t to '/data/pgxz/t.csv' with csv ;
COPY 4
```

copy from用法详解--复制文件内容到数据表中

导入所有列

```
postgres=# \! cat  /data/pgxz/t.txt
1       TDSQL PG   \N      7
2       pg'",      xc%  2017-10-28 18:24:05.643102      3
3       pgxz    2017-10-28 18:24:05.645691      \N
postgres=# truncate table t;
TRUNCATE TABLE
postgres=# copy t from '/data/pgxz/t.txt';
COPY 3
postgres=# select * from t;
 f1 |       f2       |            f3            | f4
----+----------------+--------------------------+----
  1 | TDSQL PG       |                          |  7
  2 | pg'",      xc% | 2017-10-28 18:24:05.643102 |  3
  3 | pgxz           | 2017-10-28 18:24:05.645691 |
(3 rows)
```

导入部分指定列

```
postgres=#  copy t(f1,f2) to '/data/pgxz/t.txt';
postgres=#  \! cat /data/pgxz/t.txt
1       TDSQL PG
2       pg'",      xc%
3       pgxz
postgres=# truncate table t;
TRUNCATE TABLE
postgres=# copy t(f1,f2) from  '/data/pgxz/t.txt';
COPY 3
postgres=# select * from t;
 f1 |       f2       |            f3             | f4
----+----------------+---------------------------+----
  1 | TDSQL PG       | 2017-10-30 11:54:16.559579 |
  2 | pg'",      xc% | 2017-10-30 11:54:16.559579 |
  3 | pgxz           | 2017-10-30 11:54:16.560283 |
(3 rows)
#有默认值的字段在没有导入时，会自动的将默认值付上
postgres=# \! cat /data/pgxz/t.txt
1       \N      TDSQL PG
2       2017-10-28 18:24:05.643102      pg'",      xc%
3       2017-10-28 18:24:05.645691      pgxz
postgres=# truncate table t;
TRUNCATE TABLE
postgres=# copy t(f1,f3,f2) from '/data/pgxz/t.txt';
COPY 3
postgres=# select * from t;
 f1 |       f2       |            f3             | f4
----+----------------+---------------------------+----
  1 | TDSQL PG       |                           |
  2 | pg'",      xc% | 2017-10-28 18:24:05.643102 |
  3 | pgxz           | 2017-10-28 18:24:05.645691 |
(3 rows)
#字段的顺序可以任意调整，但需要与导放文件的存放顺序一致
postgres=# \! cat /data/pgxz/t.txt;
1       TDSQL PG   \N      7
2       pg'",      xc%  2017-10-28 18:24:05.643102      3
3       pgxz    2017-10-28 18:24:05.645691      \N
postgres=# copy t (f1,f2) from  '/data/pgxz/t.txt';
ERROR:  extra data after last expected column
CONTEXT:  COPY t, line 1: "1    TDSQL PG   \N      7"
```

数据文件的列表不能多于要导入的列数，否则会出错。

指定导入文件格式

```
postgres=# \! cat /data/pgxz/t.txt
1	TDSQL PG	\N	7
2	pg'",	xc%	2017-10-28 18:24:05.643102	3
3	pgxz	2017-10-28 18:24:05.645691	\N
postgres=# copy t from '/data/pgxz/t.txt' (format 'text');
COPY 3
TRUNCATE TABLE
postgres=# \! cat /data/pgxz/t.csv
1,TDSQL PG,,7
2,"pg'"",	xc%",2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
postgres=#  copy t from '/data/pgxz/t.csv' (format 'csv');
COPY 3
postgres=# truncate table t;
TRUNCATE TABLE
postgres=# \! od -c /data/pgxz/t.bin
0000000   P   G   C   O   P   Y  \n 377  \r  \n  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0 004  \0  \0  \0 004  \0  \0  \0 001  \0  \0  \0
0000040 005   T   b   a   s   e 377 377 377 377  \0  \0  \0 004  \0  \0
0000060  \0  \a  \0 004  \0  \0  \0 004  \0  \0  \0 002  \0  \0  \0 016
0000100   p   g   '   "   ,                       x   c   %  \0  \0
0000120  \0  \b  \0 001 377 236   G   w 213   ^  \0  \0  \0 004  \0  \0
0000140  \0 003  \0 004  \0  \0  \0 004  \0  \0  \0 003  \0  \0  \0 004
0000160   p   g   x   z  \0  \0  \0  \b  \0 001 377 236   G   w 225   {
0000200 377 377 377 377 377 377
0000206
postgres=# copy t from '/data/pgxz/t.bin' (format 'binary');
COPY 3
postgres=# select * from t;
 f1 |       f2       |            f3            | f4
----+----------------+--------------------------+----
  1 | TDSQL PG       |                          |  7
  2 | pg'",      xc% | 2017-10-28 18:24:05.643102 |  3
  3 | pgxz           | 2017-10-28 18:24:05.645691 |
(3 rows)
```

使用delimiter指定列与列之间的分隔符

```
postgres=#  \! cat /data/pgxz/t.txt
1%TDSQL PG%\N%7
2%pg'",      xc\%%2017-10-28 18:24:05.643102%3
3%pgxz%2017-10-28 18:24:05.645691%\N
postgres=# copy t from  '/data/pgxz/t.txt' (format 'text',delimiter '%');
COPY 3
postgres=# select * from t;
 f1 |       f2       |            f3            | f4
----+----------------+--------------------------+----
  1 | TDSQL PG       |                          |  7
  2 | pg'",      xc% | 2017-10-28 18:24:05.643102 |  3
  3 | pgxz           | 2017-10-28 18:24:05.645691 |
(3 rows)

postgres=# \! cat /data/pgxz/t.csv
1%TDSQL PG%%7
2%"pg'"",      xc%"%2017-10-28 18:24:05.643102%3
3%pgxz%2017-10-28 18:24:05.645691%
postgres=# truncate table t;
TRUNCATE TABLE
postgres=# copy t from  '/data/pgxz/t.csv' (format 'csv',delimiter '%');
COPY 3
postgres=# select * from t;
 f1 |       f2       |            f3            | f4
----+----------------+--------------------------+----
  1 | TDSQL PG       |                          |  7
  2 | pg'",      xc% | 2017-10-28 18:24:05.643102 |  3
  3 | pgxz           | 2017-10-28 18:24:05.645691 |
(3 rows)
```

NULL值处理

```
postgres=#  \! cat /data/pgxz/t.txt
1       TDSQL PG    NULL     7
2       pg'",       xc%  2017-10-28 18:24:05.643102       3
3       pgxz     2017-10-28 18:24:05.645691       NULL
postgres=# copy t from '/data/pgxz/t.txt' (null 'NULL');
COPY 3
postgres=# select * from t;
 f1 |       f2       |            f3             | f4
----+----------------+---------------------------+----
  1 | TDSQL PG       |                           |  7
  2 | pg'",       xc% | 2017-10-28 18:24:05.643102 |  3
  3 | pgxz           | 2017-10-28 18:24:05.645691 |
(3 rows)
```

#将文件中的NULL字符串当成NULL值处理,SQL SERVER导出来的文件中把NULL值替换成字符串NULL,所以入库时可以这样处理一下，注意字符串是区分大小写，如下面语句导入数据就会出错

```
postgres=# copy t from '/data/pgxz/t.txt' (null 'null');
ERROR:  invalid input syntax for type timestamp: "NULL"
CONTEXT:  COPY t, line 1, column f3: "NULL"
```

自定义quote字符

```
postgres=# \! cat /data/pgxz/t.csv
1,TDSQL PG,,7
2,%pg'",      xc%%%,2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,

#如果不配置quote字符则无法导入文件

postgres=# copy t from '/data/pgxz/t.csv' (format 'csv');
ERROR:  unterminated CSV quoted field
CONTEXT:  COPY t, line 4: "2,%pg'",      xc%%%,2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,

"postgres=# copy t from '/data/pgxz/t.csv' (format 'csv',quote '%');
COPY 3
postgres=#
postgres=# copy t from '/data/pgxz/t.csv' (format 'text',quote '%');
ERROR:  COPY quote available only in CSV mode

#只有csv格式导入时才能配置quote字符
```

自定义escape字符

```
postgres=#  \! cat  /data/pgxz/t.csv
1,TDSQL PG,,7
2,"pg'@",      xc%",2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
postgres=#  copy t from '/data/pgxz/t.csv' (format 'csv');
ERROR:  unterminated CSV quoted field
CONTEXT:  COPY t, line 4: "2,"pg'@",      xc%",2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
"
#不指定escape字符时，系统默认就是双写的quote字符，如双倍的“双引号”
postgres=#  copy t from '/data/pgxz/t.csv' (format 'csv',escape '@');
COPY 3
postgres=# select * from t;
 f1 |       f2       |            f3            | f4
----+----------------+--------------------------+----
  1 | TDSQL PG       |                          |  7
  2 | pg'",      xc% | 2017-10-28 18:24:05.643102 |  3
  3 | pgxz           | 2017-10-28 18:24:05.645691 |
(3 rows)
postgres=#
```

csv header忽略首行

```
postgres=# \! cat /data/pgxz/t.csv;
f1,f2,f3,f4
1,TDSQL PG,,7
2,"pg'"",      xc%",2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
postgres=# copy t from '/data/pgxz/t.csv' (format 'csv');
ERROR:  invalid input syntax for integer: "f1"
CONTEXT:  COPY t, line 1, column f1: "f1"
postgres=# copy t from '/data/pgxz/t.csv' (format 'csv',header true);
COPY 3
postgres=# select * from t;
 f1 |       f2       |            f3            | f4
----+----------------+--------------------------+----
  1 | TDSQL PG       |                          |  7
  2 | pg'",      xc% | 2017-10-28 18:24:05.643102 |  3
  3 | pgxz           | 2017-10-28 18:24:05.645691 |
(3 rows)
```

如果不忽略首行，则系统会把首行当成数据，造成导入失败。

导入oid列值

```
postgres=# \! cat /data/pgxz/t.txt
35242   1       TDSQL PG    \N      7
35243   2       pg'",       xc%  2017-10-28 18:24:05.643102      3
35340   3       pgxz    2017-10-28 18:24:05.645691      \N
postgres=# truncate table t;
TRUNCATE TABLE
postgres=# copy t from '/data/pgxz/t.txt' (oids true);
COPY 3
postgres=# select oid,* from t;
  oid  | f1 |       f2       |            f3            | f4
-------+----+----------------+--------------------------+----
 35242 |  1 | TDSQL PG       |                          |  7
 35243 |  2 | pg'",      xc% | 2017-10-28 18:24:05.643102 |  3
 35340 |  3 | pgxz           | 2017-10-28 18:24:05.645691 |
(3 rows)
```

使用FORCE_NOT_NULL把某列中空值变成长度为0的字符串，而不是NULL值

```
postgres=#truncate table t;
TRUNCATETABLE
postgres=#\! cat '/data/pgxz/t.csv' ;
1,TDSQL PG,,7
2,"pg'"",      xc%",2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-2818:24:05.645691,
4,,2017-10-30 16:14:14.954213,4
postgres=#copy t from '/data/pgxz/t.csv' (format 'csv');
ERROR:  node:16386, error null value in column"f2" violates not-null constraint
DETAIL:  Failing row contains (4, null, 2017-10-3016:14:14.954213, 4).
postgres=#select * from t where f2='';
 f1 | f2 |            f3             | f4
----+----+----+----
(0 rows)
```

不使用FORCE_NOT_NULL处理的话就变成NULL值。

```
postgres=#truncate table t;
TRUNCATETABLE
postgres=#copy t from '/data/pgxz/t.csv' (format 'csv' ,FORCE_NOT_NULL (f2));
COPY 4
postgres=#select * from t where f2='';
 f1 | f2 |            f3             | f4
----+----+---------------------------+----
  4 |   | 2017-10-30 16:14:14.954213 |  4
row)
```

使用FORCE_NOT_NULL处理就变成长度为0的字符串。

encoding指定导入文件的编码

```
postgres=# \! enca -L zh_CN /data/pgxz/t.txt
Simplified Chinese National Standard; GB2312
postgres=# copy t from '/data/pgxz/t.txt' ;
COPY 4
postgres=# select * from t;
 f1 |       f2       |            f3            | f4
----+----------------+--------------------------+----
  1 | TDSQL PG       |                          |  7
  2 | pg'",      xc% | 2017-10-28 18:24:05.643102 |  3
  3 | pgxz           | 2017-10-28 18:24:05.645691 |
  4 |                | 2017-10-30 16:41:09.157612 |  4
(4 rows)
#不指定导入文件的编码格式，则无法正确导入中文字符
postgres=# truncate table t;
TRUNCATE TABLE
postgres=# copy t from '/data/pgxz/t.txt' (encoding gbk) ;
COPY 4
postgres=# select * from t;
 f1 |       f2       |            f3            | f4
----+----------------+--------------------------+----
  1 | TDSQL PG       |                          |  7
  2 | pg'",      xc% | 2017-10-28 18:24:05.643102 |  3
  3 | pgxz           | 2017-10-28 18:24:05.645691 |
  4 | 平台           | 2017-10-30 16:41:09.157612 |  4
(4 rows)
#使用encoding gbk后便可以正确导入文件的内容，你也可以使用下面的方式转换导入文件的编码后再导入数据
postgres=# truncate table t;
TRUNCATE TABLE
postgres=# \! enconv -L zh_CN -x UTF-8 /data/pgxz/t.txt
postgres=# copy t from '/data/pgxz/t.txt';
COPY 4
postgres=# select * from t;
 f1 |       f2       |            f3            | f4
----+----------------+--------------------------+----
  1 | TDSQL PG       |                          |  7
  2 | pg'",      xc% | 2017-10-28 18:24:05.643102 |  3
  3 | pgxz           | 2017-10-28 18:24:05.645691 |
  4 | 平台           | 2017-10-30 16:41:09.157612 |  4
(4 rows)
postgres=#
```

position指定字段长度导入

```
postgres=# \! cat  /data/tbase/t_position_copy.txt
123阿弟2021
postgres=#
postgres=# copy t_position_copy (f1 position(1:3),f2 position(4:9) ,f3 position(10:13)) from '/data/tbas
e/t_position_copy.txt';
COPY 1
postgres=#
```

position指定字段长度+函数处理导入

```
postgres=# \! cat  /data/tbase/t_position_copy.txt
123阿弟2020010120200101121212
postgres=# copy t_position_copy (f1 position(1:3),f2 position(4:9) ,f3 position(10:17) to_date($3,'yyyym
mdd'),f4 position(18:31) to_timestamp($4,'yyyymmddhh24miss')) from '/data/tbase/t_position_copy.tx
t';
COPY 1
```

copy支持多字节分隔符

TDSQL PG支持多字节分隔符入库，限制10字节以内，支持中文分隔符，支持多字符集utf8以及gbk。

示例：

1、csv格式入库

--创建表

```
postgres=#
create table test_copy_base_1(c0 int,c1 macaddr,c2 point,c3 text,c4 numeric,c5 char,c6 oidvector,c7 floa
t4,c8 abstime,c9 reltime,c10 polygon,c11 bool,c12 tinterval,c13 money,c14 tid,c15 circle,c16 macaddr8,
c17 inet,c18 varbit(5),c19 varchar,c20 timestamp,c21 oid,c22 line,c23 int4,c24 int2,c25 pg_oracle.date,c2
6 path,c27 bit varying(5),c28 int8,c29 box,c30 lseg,c31 name,c32 interval,c33 timestamptz,c34 bpchar,c
35 time,c36 float8,c37 timetz,c38 bytea,c39 varchar2,c40 number,c41 serial,c42 bigserial) distribute by s
hard(c0);
```

---插入数据

```
create or REPLACE procedure insert_data_start(starts int, table_num int, table_name varchar2)
as
    v_sql varchar2;
    region varchar[];
BEGIN
    region := array['城市1','城市2','城市3','城市4','城市5','城市6','城市7','城市8'];
    v_sql := 'insert into '||table_name||' select i, concat_ws(''-'',to_hex((i%192)+1),to_hex((i%168)+1),to_he
x((i%200)+1),to_hex((i%100)+1),to_hex((i%98)+1),to_hex((i%255)+1))::macaddr,point(ceil((i%68)+1),ceil
((i%77)+1)),''南山''||(i%100)||''号'',round((0.356*500*i)%876,3)::numeric,(i%10)::char,(i*987)::varchar::oidv
ector,round(i*5.301,3)::float4,abstime(''2022-06-07 12:12:12''::timestamp+(i%256)*interval ''1 day''),relti
me(i*542),polygon (box(point(i%550,i%360),point(i%50,i%22))),bool(i%2),tinterval(abstime(''2022-06-0
7 12:12:12''::timestamp+(i%300)*interval ''1 day''),abstime(''2022-06-07 12:12:12''::timestamp+i*interva
l ''1 hour'')),((i*3.156)%9999)::money,(i%798,i%666)::varchar::tid,circle(point(i%250,i%160),i%76),concat_
ws(''-'',to_hex((i%192)+16),to_hex((i%168)+16),to_hex((i%200)+16),to_hex((i%100)+16),to_hex((i%9
8)+16),to_hex((i%239)+16))::macaddr8,concat(concat_ws(''.'',(i%100)+100,(i%100)+1,(i%100)+1,(i%10
0)+100),''/25'')::inet,((i+1)%98799)::int::bit(5)::varbit(5),(''开发''||(i%20)+1||''组'')::varchar,(date ''2022-06-0
7 12:12:12''+i * interval ''1 day''+(i%24) * interval ''1 hour''+(i%60) * interval ''1 minute''+(i%60) * interv
al ''1 second'')::timestamp,(i%999)::bigint::oid,line(point(i%93+1,i%120),point(i%450+2,i%36)),int4(i%10
00),int2(i%150),date ''2022-06-07 12:12:12''+(i%378)*interval ''1 day'',path (polygon(box(point(i%19
2,i%120),point(i%913,i%120)))),int4(i%10000)::bit(5),int8(i*999),box(point(i%292+1,i%20),point(i%92
+2,i%24)),lseg(point(i%92+1,i%220),point(i%238+2,i%120)),name(''平台''||(i%12)),i * interval ''1 day'',ti
mestamptz(date(date ''2022-06-07 12:12:12'' + (i%777) * interval ''1 day''),(timestamp ''2022-06-07 1
2:12:12''+(i%100) * interval ''1 hour'')::time without time zone),bpchar((''你好''||(i%98))::text),(i%300)*int
erval ''1 hour 1 minute 1 second'',float8((i*3.122333)%998),timetz(timestamptz ''2022-06-07 10:56:00
+08'' + (i%200) * interval ''1 hour''),repeat(''深圳''||(i%1000),3)::bytea,(''tbase in 深圳''||(i%900))::varchar
2,((i*3.1123)%998)::number(10,2) from generate_series('||starts||','||table_num||') i';
    -- RAISE notice '%', v_sql;
```

```
    execute v_sql;
end;
/

postgres=#
call insert_data_start(1, 5000, 'test_copy_base_1');
postgres=#
analyze test_copy_base_1;
```

--出库

```
COPY (select * from test_copy_base_1 order by c0) to '/tmp/test_copy_base_1.data' with CSV delimiter
'$^$';
```

--入库

```
COPY test_copy_base_1 from '/tmp/test_copy_base_1.data' with CSV delimiter '$^$';
```

--查询入库结果

```
postgres=# select count(1) from test_copy_base_1;
 count
-------
  5000
(1 row)

postgres=# select c0,c22,c25 from test_copy_base_1 order by c0,c25 desc nulls last limit 5;
 c0 |   c22    |        c25
----+----------+--------------------
  1 | {0,-1,1} | 2022-06-08 12:12:12
  2 | {0,-1,2} | 2022-06-09 12:12:12
  3 | {0,-1,3} | 2022-06-10 12:12:12
  4 | {0,-1,4} | 2022-06-11 12:12:12
  5 | {0,-1,5} | 2022-06-12 12:12:12
(5 rows)
```

2、支持中文分隔符

--出库

```
postgres=#
COPY (select * from test_copy_base_1 order by c0) to '/tmp/test_copy_base_2.data' (format TEXT, delim
iter '你好');
postgres=#
delete from test_copy_base_1 where 1=1;
```

--入库

```
postgres=#
-----使用错误分隔符会报错
COPY test_copy_base_1 from '/tmp/test_copy_base_2.data' with (format TEXT,delimiter '//');
ERROR:  invalid input syntax for type numeric: "1你好02:02:02:02:02:02你好(2,2)你好南山1号你好178你好1你好987你好5.301你好2022-06-08 12:12:12 +08:00你好00:09:02你好((1,1),(1,1),(1,1),(1,1))你好t你好["2022-06-07 13:12:12 +08:00" "2022-06-08 12:12:12 +08:00"]你好￥3.16你好(1,1)你好
CONTEXT:  COPY test_copy_base_1, line 1, column c0: "1你好02:02:02:02:02:02你好(2,2)你好南山1号你好178你好1你好987你好5.301你好2022-06...", nodetype:1(1:cn,0:dn)
postgres=#
COPY test_copy_base_1 from '/tmp/test_copy_base_2.data' with (format TEXT,delimiter '你好');
```

--查询入库结果

```
postgres=# select count(1) from test_copy_base_1;
 count
-------
  5000
(1 row)
```

创建和管理分区表
范围分区
创建时间范围分区表

```
postgres=#  create table t_time_range
(f1 bigint, f2 timestamp ,f3 bigint)
partition by range (f2) begin (timestamp without time zone '2017-09-01 0:0:0')
step (interval '1 month')
partitions (12) distribute by shard(f1)
to group default_group;
CREATE TABLE
postgres=# \d+ t_time_range
                          Table "public.t_time_range"
 Column |            Type             | Modifiers | Storage | Stats target | Description
--------+-----------------------------+-----------+---------+--------------+-------------
 f1     | bigint                      |           | plain   |              |
 f2     | timestamp without time zone |           | plain   |              |
 f3     | bigint                      |           | plain   |              |
Has OIDs: no
Distribute By SHARD(f1)
       Location Nodes: ALL DATANODES
Partition By: RANGE(f2)
   # Of Partitions: 12
   Start With: 2017-09-01
Interval Of Partition: 1 MONTH
```

范围分区类型

- 创建主分区

```
postgres=# create table t_native_range (f1 bigint,f2 timestamp default now(), f3 integer) partition by ra
nge ( f2 )distribute by shard(f1) to group default_group;
CREATE TABLE
```

- 建立两个子表

```
postgres=# create table t_native_range_201709 partition of t_native_range (f1 ,f2 , f3 ) for values from
('2017-09-01') to ('2017-10-01');
CREATE TABLE

postgres=# create table t_native_range_201710 partition of t_native_range (f1 ,f2 , f3 ) for values from
('2017-10-01') to ('2017-11-01');
CREATE TABLE
```

- 默认分区表

```
--没有默认分区表时插入会出错
postgres=# insert into t_native_range values(2,'2017-08-01',2);
ERROR:  node:dn001, backend_pid:32123, nodename:dn001,backend_pid:32123,message:no partition o
f relation "t_native_range" found for row
DETAIL:  Partition key of the failing row contains (f2) = (2017-08-01 00:00:00).

--添加默认分区表

postgres=# CREATE TABLE t_native_range_default PARTITION OF t_native_range DEFAULT;
CREATE TABLE
postgres=# insert into t_native_range values(2,'2017-08-01',2);
INSERT 0 1
```

- MAXVALUE分区

```
postgres=# CREATE TABLE t_native_range_maxvalue PARTITION OF t_native_range for values from ('20
17-11-01') to (maxvalue);
CREATE TABLE
postgres=# insert into t_native_range values(1,'2017-11-01',1);
INSERT 0 1
```

```
postgres=# select * from t_native_range where f2='2017-11-01';
 f1 |         f2          | f3
----+---------------------+----
  1 | 2017-11-01 00:00:00 |  1
(1 row)

postgres=# explain select * from t_native_range where f2='2017-11-01';
                                QUERY PLAN
---------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Append  (cost=0.00..24.12 rows=6 width=20)
         ->  Seq Scan on t_native_range_maxvalue  (cost=0.00..24.12 rows=6 width=20)
               Filter: (f2 = '2017-11-01 00:00:00'::timestamp without time zone)
(5 rows)

postgres=# insert into t_native_range values(1,'2018-1-01',1);
INSERT 0 1
postgres=# explain select * from t_native_range where f2='2018-1-01';
                                QUERY PLAN
---------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Append  (cost=0.00..24.12 rows=6 width=20)
         ->  Seq Scan on t_native_range_maxvalue  (cost=0.00..24.12 rows=6 width=20)
               Filter: (f2 = '2018-01-01 00:00:00'::timestamp without time zone)
(5 rows)

postgres=#
```

所有比2017-11-1大的数据都存储到子表t_native_range_maxvalue

- MINVALUE分区

```
postgres=# CREATE TABLE t_native_range_minvalue PARTITION OF t_native_range for values from (min
value) to ('2017-09-01');
CREATE TABLE
postgres=# insert into t_native_range values(1,'2017-08-01',1);
INSERT 0 1
postgres=# explain SELECT * FROM t_native_range where f2='2017-08-01';
                                QUERY PLAN
---------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Append  (cost=0.00..24.12 rows=6 width=20)
```

```
        ->  Seq Scan on t_native_range_minvalue  (cost=0.00..24.12 rows=6 width=20)
            Filter: (f2 = '2017-08-01 00:00:00'::timestamp without time zone)
(5 rows)
```

- 查看表结构

```
postgres=# \d+ t_native_range
                                Table"tbase_pg_proc.t_native_range"
 Column |            Type             | Collation | Nullable | Default |Storage | Stats target | Description
--------+-----------------------------+-----------+----------+---------+---------+--------------
+-------------
 f1     |bigint                       |           |          |         | plain   |              |
 f2     |timestamp without time zone |           |          | now()   | plain  |              |
 f3     |integer                      |           |          |         | plain   |              |
Partition key: RANGE (f2)
Partitions: t_native_range_201709 FORVALUES FROM ('2017-09-01 00:00:00') TO ('2017-10-01 00:00:0
0'),
            t_native_range_201710 FOR VALUES FROM ('2017-10-01 00:00:00') TO('2017-11-01 00:00:00'),
            t_native_range_maxvalue FOR VALUES FROM ('2017-11-01 00:00:00') TO(MAXVALUE),
            t_native_range_minvalue FOR VALUES FROM (MINVALUE) TO ('2017-09-0100:00:00'),
            t_native_range_default DEFAULT
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES创建主分区

postgres=# create table t_native_range (f1 bigint,f2 timestamp default now(), f3 integer) partition by ra
nge ( f2 )distribute by shard(f1) to group default_group;
CREATE TABLE
```

范围分区表使用

```
postgres=#create table t_range (f1 bigint,f2 timestamp default now(), f3 integer) partition by range (f3) begin (1) step (50) partitions (3) distribute by shard(f1) to group default_group;
CREATE TABLE
postgres=# insert into t_range(f1,f3) values(1,1),(2,50),(3,100),(2,110);
INSERT 0 4

#这样的语句执行会出错,提示超出范围
postgres=# insert into t_range(f1,f3) values(1,151);
ERROR:  node:16385, error inserted value is not in range of partitioned table, please check the value of paritition key

#解决办法再扩展一些分区后就可以使用了
postgres=# ALTER TABLE t_range ADD PARTITIONS 2;
ALTER TABLE
postgres=# insert into t_range(f1,f3) values(1,151);
INSERT 0 1
postgres=#
```

列表分区
列表分区使用

以下示例说明列表分区使用

- 创建主分区

```
postgres=# create table t_native_list(f1 bigserial not null,f2 text, f3 integer,f4 date) partition by list( f2 ) distribute by shard(f1) to group default_group;
CREATE TABLE
```

- 建立两个子表,分别存入“城市1”和“城市2”

```
postgres=# create table t_list_gd partition of t_native_list(f1 ,f2 , f3,f4) for values in ('城市1');
CREATE TABLE

postgres=# create table t_list_bj partition of t_native_list(f1 ,f2 , f3,f4) for values in ('城市2');
CREATE TABLE
```

- 查看表结构

```
postgres=# \d+ t_native_list
                                        Table "tbase.t_native_list"
 Column | Type    | Collation | Nullable|                 Default                 | Storage  | Stats target | Description
--------+---------+-----------+---------+-----------------------------------------+----------+--------------+-------------
 f1     |bigint  |           | not null |nextval('t_native_list_f1_seq'::regclass) | plain    |              |
 f2     |text    |           |          |                                          |extended |              |
 f3     |integer |           |          |                                           |plain    |              |
 f4     |date    |           |          |                                          |plain    |              |
Partition key: LIST (f2)
Partitions: t_list_bj FOR VALUES IN ('城市2'),
            t_list_gd FOR VALUES IN ('城市1')
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES

postgres=#
```

- 创建default分区

#没有default分区情况下会出错，插入会出错

```
postgres=# insert into t_native_list values(1,'城市3',1,current_date);
ERROR:  node:dn001, backend_pid:31664, nodename:dn001,backend_pid:31664,message:no partition o
f relation "t_native_list" found for row
DETAIL:  Partition key of the failing row contains (f2) = (城市3).

postgres=# CREATE TABLE t_native_list_default PARTITION OF t_native_list DEFAULT;
CREATE TABLE

#创建后就能正常插入
postgres=# insert into t_native_list values(1,'城市3',1,current_date);
INSERT 0 1
postgres=#
```

散列分区
散列分区使用

以下示例说明散列分区使用

```
postgres=# create table t_hash_partition(f1 int,f2 int) partition by hash(f2);
create table t_hash_partition_1 partition of t_hash_partition FOR VALUES WITH(MODULUS 4, REMAIND
ER 0);
create table t_hash_partition_2 partition of t_hash_partition FOR VALUES WITH(MODULUS 4, REMAIND
ER 1);
create table t_hash_partition_3 partition of t_hash_partition FOR VALUES WITH(MODULUS 4, REMAIND
ER 2);
create table t_hash_partition_4 partition of t_hash_partition FOR VALUES WITH(MODULUS 4, REMAIND
ER 3);

#上面创建4个子分区的hash表，hash分区表需要指定分区的个数，因为使用分区数做为算子来计算每条数据所在分区，所以目前hash分区不支持删除添加和删除。

postgres=# insert into t_hash_partition values(1,1),(2,2),(3,3);
COPY 3
postgres=# select * from t_hash_partition;
 f1 | f2
----+----
  1 |  1
  3 |  3
  2 |  2
(3 rows)

#TDSQL pg自动根据分区值进行剪枝

postgres=# explain select * from t_hash_partition where f2=2;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Append  (cost=0.00..26.88 rows=7 width=8)
         ->  Seq Scan on t_hash_partition_3  (cost=0.00..26.88 rows=7 width=8)
               Filter: (f2 = 2)
(5 rows)
```

多级分区
多级分区表使用

- 创建主表

```
postgres=# create table t_native_mul_list(f1 bigserial not null,f2 integer,f3 text,f4 text, f5 date) partition by list ( f3 ) distribute by shard(f1) to group default_group;
NOTICE:  Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
```

- 创建二级表

```
postgres=# create table t_native_mul_list_gd partition of t_native_mul_list for values in ('城市1') partition by range(f5);
NOTICE:  Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE

postgres=# create table t_native_mul_list_bj partition of t_native_mul_list for values in ('城市2') partition by range(f5);
NOTICE:  Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE

postgres=# create table t_native_mul_list_sh partition of t_native_mul_list for values in ('城市3');
NOTICE:  Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
```

- 创建三级表

```
postgres=# create table t_native_mul_list_gd_201701 partition of t_native_mul_list_gd(f1,f2,f3,f4,f5) for values from ('2017-01-01') to ('2017-02-01');
NOTICE:  Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE

postgres=# create table t_native_mul_list_gd_201702 partition of t_native_mul_list_gd(f1,f2,f3,f4,f5) for values from ('2017-02-01') to ('2017-03-01');
NOTICE:  Replica identity is needed for shard table, please add to this table through "alter table" command.
```

```
CREATE TABLE

postgres=# create table t_native_mul_list_bj_201701 partition of t_native_mul_list_bj(f1,f2,f3,f4,f5) for v
alues from ('2017-01-01') to ('2017-02-01');
NOTICE:  Replica identity is needed for shard table, please add to this table through "alter table" comm
and.
CREATE TABLE

postgres=# create table t_native_mul_list_bj_201702 partition of t_native_mul_list_bj(f1,f2,f3,f4,f5) for v
alues from ('2017-02-01') to ('2017-03-01');
NOTICE:  Replica identity is needed for shard table, please add to this table through "alter table" comm
and.
CREATE TABLE
```

TDSQL PG支持存在1级和2级分区混用，大家不需要都平级。

分区拆分与合并
分区拆分

TDSQL PG支持分区表拆分，支持RANGE、LIST的一级和多级分区拆分。分区拆分通常用于将DEFAULT分区拆分到新的分区，或者将一个较大的分区拆分成两个分区。
暂不支持自研分区、HASH分区拆分。
范围（RANGE）分区拆分：
ALTER TABLE table_name SPLIT PARTITION part_3 AT ({ numeric_literal | string_literal }) INTO (PARTITION part_1, PARTITION part_2);

列表（LIST）分区拆分：
ALTER TABLE table_name SPLIT PARTITION part_3 VALUES ({ numeric_literal | string_literal | NULL } , { numeric_literal | string_literal | NULL }) INTO (PARTITION part_1, PARTITION part_2);

示例

```
postgres=# \d+ part_range_t2
                                Table "public.part_range_t2"
 Column |            Type             | Collation | Nullable | Default | Storage  | Stats target | Description
--------+-----------------------------+-----------+----------+---------+----------+--------------
+-------------
 f1     | integer                     |           | not null |         | plain    |              |
 f2     | timestamp without time zone |           | not null | now()   | plain    |              |
 f3     | character varying(50)       |           |          |         | extended |              |
 f4     | integer                     |           |          |         | plain    |              |
Partition key: RANGE (f2)
Partitions: part_range_t2_part_0 FOR VALUES FROM ('2020-06-01 00:00:00') TO ('2020-07-01 00:00:00'),
            part_range_t2_part_1 FOR VALUES FROM ('2020-07-01 00:00:00') TO ('2020-08-01 00:00:00'),
            part_range_t2_part_2 FOR VALUES FROM ('2020-08-01 00:00:00') TO ('2020-09-01 00:00:00')
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES
```

当前分区表有part_range_t2_part_0、part_range_t2_part_1、part_range_t2_part_2 3个分区。

下面的sql将part_range_t2_part_2中 ('2020-08-01 00:00:00') 到 ('2020-08-20 00:00:00')的数据拆分到part_range_t2_part_21。

```
postgres=# ALTER TABLE part_range_t2 SPLIT PARTITION part_range_t2_part_2 AT ( '2020-08-20') INTO (PARTITION part_range_t2_part_21, PARTITION part_range_t2_part_2);
ALTER TABLE

postgres=# \d+ part_range_t2
                                Table "public.part_range_t2"
```

```
 Column |            Type             | Collation | Nullable | Default | Storage  | Stats target | Description
--------+-----------------------------+-----------+----------+---------+----------+--------------
+-------------
 f1     | integer                     |           | not null |         | plain    |              |
 f2     | timestamp without time zone |           | not null | now()   | plain    |              |
 f3     | character varying(50)       |           |          |         | extended |              |
 f4     | integer                     |           |          |         | plain    |              |
Partition key: RANGE (f2)
Partitions: part_range_t2_part_0 FOR VALUES FROM ('2020-06-01 00:00:00') TO ('2020-07-01 00:00:00'),
            part_range_t2_part_1 FOR VALUES FROM ('2020-07-01 00:00:00') TO ('2020-08-01 00:00:00'),
            part_range_t2_part_2 FOR VALUES FROM ('2020-08-20 00:00:00') TO ('2020-09-01 00:00:00'),
            part_range_t2_part_21 FOR VALUES FROM ('2020-08-01 00:00:00') TO ('2020-08-20 00:00:00')
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODE
```

确认分区已拆分。

```
postgres=# select * from part_range_t2_part_2;
 f1 |         f2          |  f3  | f4
----+---------------------+------+----
  4 | 2020-08-21 00:00:00 | four |  4
(1 row)
postgres=# select * from part_range_t2_part_21;
 f1 |         f2          |  f3   | f4
----+---------------------+-------+----
  3 | 2020-08-01 00:00:00 | three |  4
(1 row)
```

确认数据已拆分到新的分区子表中。

分区合并

TDSQL PG支持分区表合并。通过分区合并，可以将两个分区的数据合并到另一个分区中。分区合并通常用于将一些较小的列表分区，或相邻的较小的范围分区合并，减少分区数量，提高效率。
不能用于散列（HASH）分区和子分区的合并。范围（RANGE）分区合并时，允许合并两临近范围的分区到另外分区，不相邻分区无法合并。合并结果分区继承两个源分区的最大边界。暂不支持UPDATE [GLOBAL] INDEXES。
语法

```
ALTER TABLE table_name MERGE PARTITIONS ( part1, part2 ) INTO PARTITION part3 ;
```

示例
当前分区有part_range_t2_part_0、part_range_t2_part_1、part_range_t2_part_2、part_range_t2_part_21 四个。

```
postgres=# \d+  part_range_t2
                                Table "public.part_range_t2"
 Column |            Type             | Collation | Nullable | Default | Storage  | Stats target | Description
--------+-----------------------------+-----------+----------+---------+----------+--------------
+-------------
 f1     | integer                     |           | not null |         | plain    |              |
 f2     | timestamp without time zone |           | not null | now()   | plain    |              |
 f3     | character varying(50)       |           |          |         | extended |              |
 f4     | integer                     |           |          |         | plain    |              |
Partition key: RANGE (f2)
Partitions: part_range_t2_part_0 FOR VALUES FROM ('2020-06-01 00:00:00') TO ('2020-07-01 00:00:00'),
            part_range_t2_part_1 FOR VALUES FROM ('2020-07-01 00:00:00') TO ('2020-08-01 00:00:00'),
            part_range_t2_part_2 FOR VALUES FROM ('2020-08-20 00:00:00') TO ('2020-09-01 00:00:00'),
            part_range_t2_part_21 FOR VALUES FROM ('2020-08-01 00:00:00') TO ('2020-08-20 00:00:00')
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES
```

下面的sql将part_range_t2_part_2、part_range_t2_part_21两个分区的数据合并到part_range_t2_part_22中。

```
postgres=# ALTER TABLE part_range_t2 MERGE PARTITIONS ( part_range_t2_part_2, part_range_t2_part
_21) INTO PARTITION part_range_t2_part_22;
ALTER TABLE
postgres=# \d+  part_range_t2
                                Table "public.part_range_t2"
 Column |            Type             | Collation | Nullable | Default | Storage  | Stats target | Description
--------+-----------------------------+-----------+----------+---------+----------+--------------
+-------------
 f1     | integer                     |           | not null |         | plain    |              |
 f2     | timestamp without time zone |           | not null | now()   | plain    |              |
 f3     | character varying(50)       |           |          |         | extended |              |
 f4     | integer                     |           |          |         | plain    |              |
```

```
Partition key: RANGE (f2)
Partitions: part_range_t2_part_0 FOR VALUES FROM ('2020-06-01 00:00:00') TO ('2020-07-01 00:00:00'),
            part_range_t2_part_1 FOR VALUES FROM ('2020-07-01 00:00:00') TO ('2020-08-01 00:00:00'),
            part_range_t2_part_22 FOR VALUES FROM ('2020-08-01 00:00:00') TO ('2020-09-01 00:00:00')
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES
postgres=#
```

确认分区合并后，分区从4个变为3个。

```
postgres=# select * from part_range_t2_part_22;
 f1 |         f2          |  f3   | f4
----+---------------------+-------+----
  4 | 2020-08-21 00:00:00 | four  |  4
  3 | 2020-08-01 00:00:00 | three |  4
(2 rows)
```

确认数据迁移至合并后的分区。

添加分区子表

```
postgres=# \d+ t1_pt
                                  Table "public.t1_pt"
 Column |            Type             | Collation | Nullable | Default | Storage  | Stats target | Description
--------+-----------------------------+-----------+----------+---------+----------+--------------
+-------------
 f1     | integer                     |           | not null |         | plain    |              |
 f2     | timestamp without time zone |           | not null |         | plain    |              |
 f3     | character varying(20)       |           |          |         | extended |              |
Indexes:
    "t1_pt_pkey" PRIMARY KEY, btree (f1)
Distribute By: SHARD(f1)
Location Nodes: dn01
Partition By: RANGE(f2)
        # Of Partitions: 3
        Start With: 2019-01-01
        Interval Of Partition: 1 MONTH

postgres=# ALTER TABLE t1_pt ADD PARTITIONS 2;

ALTER TABLE
postgres=# \d+ t1_pt
                                  Table "public.t1_pt"
 Column |            Type             | Collation | Nullable | Default | Storage  | Stats target | Description
--------+-----------------------------+-----------+----------+---------+----------+--------------
+-------------
 f1     | integer                     |           | not null |         | plain    |              |
 f2     | timestamp without time zone |           | not null |         | plain    |              |
 f3     | character varying(20)       |           |          |         | extended |              |
Indexes:
    "t1_pt_pkey" PRIMARY KEY, btree (f1)
Distribute By: SHARD(f1)
Location Nodes: dn01
Partition By: RANGE(f2)
        # Of Partitions: 5
        Start With: 2019-01-01
        Interval Of Partition: 1 MONTH
```

访问子分区

使用parent_table PARTITION (part_name) 直接访问子分区
示例：
访问范围分区

```
select * from part_range_t2partition(part_3) where rownum<=3;
```

访问列表分区

```
select * from part_list_t1 partition(part_3) where rownum<=3;
```

truncate分区

使用 alter table … truncate…删除分区
示例：

```
alter table part_range_t2 truncate partition(part_3) update global indexes;
```

查询数据

```
select * from part_range_t2 partition (part_3);
```

自研分区

在多分区表情况下，自研分区表性能更好。自研分区表随着分区的增多，性能不能不会下降，在2048个分区的情况下，自研分区表的剪技访问性能是普通分区（范围、列表、散列）表的30倍。
示例：
创建自研分区表（2048个分区）：

```
create table t_time_range
(
  f1 bigint not null, f2 timestamp ,f3 bigint
)
partition by range (f2) begin (timestamp without time zone '2017-09-01 0:0:0')
step (interval '1 month')
partitions (2048) distribute by shard(f1)
to group default_group;
```

向分区表中插入数据

```
select 'insert into t_time_range values(1,'''||startyf||''',1);' from (select ('2017-09-01'::date + (t::text||' mo
nths')::interval)::date::text as startyf from generate_series(0,2047) as t) as t;
```

冷热分区表

以下说明冷热分区的场景及使用。

```
postgres=#  create table t_cold_hot_table(f1 bigint, f2 timestamp ,f3 bigint) partition by range (f2) begi
n (timestamp without time zone '2017-09-01 0:0:0') step (interval '1 month') partitions (12) distribute b
y shard(f1,f2) to group default_group ext_group;
CREATE TABLE

postgres=# \d+ t_cold_hot_table
                          Table "pgxz.t_cold_hot_table"
 Column |            Type             | Modifiers | Storage | Stats target | Description
--------+-----------------------------+-----------+---------+--------------+-------------
 f1     | bigint                      |           | plain   |              |
 f2     | timestamp without time zone |           | plain   |              |
 f3     | bigint                      |           | plain   |              |
Has OIDs: no
Distribute By SHARD(f1,f2)
       Hotnodes:dn02, dn01 Coldnodes:dn03, dn04
Partition By: RANGE(f2)
   # Of Partitions: 12
   Start With: 2017-09-01
Interval Of Partition: 1 MONTH
```

> 注意：
>
> 创建时间范围冷热分区表需要有两个group，冷数据的ext_group对应的节点dn03和dn04需要标识为冷节点，如下所示：

```
postgres=#  select pg_set_node_cold_access();
 pg_set_node_cold_access
-------------------------
 success
(1 row)

postgres=# \q
[pgxz@VM_0_29_centos install]$ psql -p 23004
psql (PGXC , based on PG 9.4beta1)
Type "help" for help.

postgres=#  select pg_set_node_cold_access();
 pg_set_node_cold_access
-------------------------
```

```
 success
(1 row)

postgres=#
```

> 注意：
>
> 使用冷热分区表需要在postgresql.conf中配置冷热分区时间参数，如下所示：

```
manual_hot_date = '2017-12-01
```

truncate分区表

- truncate一个时间分区表

```
postgres=# \d+ t_time_range

                          Table"pgxz.t_time_range"
 Column |           Type              |  Modifiers   | Storage | Statstarget | Description

--------+-----------------------------+-----------+----------+-------------+-------------

 f1     |bigint                       |           | plain    |             |

 f2     |timestamp without time zone |           |plain     |             |

 f3     |character varying(20)        |           | extended |             |

Has OIDs: no

Distribute By SHARD(f1)

      Location Nodes: dn001, dn002

Partition By: RANGE(f2)

    #Of Partitions: 12

   Start With: 2017-09-01

   Interval Of Partition: 1 MONTH
postgres=# select * from t_time_range;

 f1|           f2            | f3

----+---------------------+-------

  1 |2017-09-01 00:00:00 | TDSQL PG

  2 |2017-10-01 00:00:00 | pgxz

(2 rows)
 postgres=# truncate t_time_range  partition for  ('2017-09-01' ::timestamp without time zone);
TRUNCATE TABLE
postgres=# select * from t_time_range;
```

```
 f1|           f2           | f3
----+--------------------+------
  2 |2017-10-01 00:00:00 | pgxz
(1 row)
 postgres=#
```

- truncate一个数字分区表

```
postgres=# \d+ t_range
                                Table"pgxz.t_range"
 Column |           Type            |  Modifiers   | Storage | Statstarget | Description
--------+---------------------------+--------------+---------+-------------+-------------
 f1     |integer                    |              | plain   |             |
 f2     |timestamp without time zone | default now() | plain   |             |
 f3     |integer                    |              | plain   |             |
Has OIDs: no
Distribute By SHARD(f1)
      Location Nodes: dn01, dn02
Partition By: RANGE(f3)
    #Of Partitions: 3
   Start With: 1
   Interval Of Partition: 50
postgres=# select * from t_range ;
 f1|           f2           | f3
----+--------------------------+-----
  1 |2017-12-22 11:47:39.153234 |   1
```

```
  2 |2017-12-22 11:47:39.153234 |  50

  2 |2017-12-22 11:47:39.153234 | 110

  3 |2017-12-22 11:47:39.153234 | 100

(4 rows)
postgres=# truncatet_range  partition for  (1);
TRUNCATE TABLE
postgres=# select * from t_range ;
 f1|             f2              | f3

----+-----------------------------+-----

  2 |2017-12-22 11:47:39.153234 | 110

  3 |2017-12-22 11:47:39.153234 | 100

(2 rows)
 postgres=#
```

部分分区表示例
partition访问子分区

```
postgres=# create table t_native_range (f1 bigint,f2 timestamp default now(), f3 integer) partition by ra
nge ( f2 ) distribute by shard(f1) to group default_group;
CREATE TABLE
postgres=# create table t_native_range_201709 partition of t_native_range  for values from ('2017-09-0
1') to ('2017-10-01');
CREATE TABLE
postgres=# create table t_native_range_201710 partition of t_native_range  for values from ('2017-10-0
1') to ('2017-11-01');
CREATE TABLE
postgres=# insert into t_native_range values(1,'2017-09-01',1);
INSERT 0 1
postgres=# insert into t_native_range values(2,'2017-09-01',2);
INSERT 0 1
postgres=# select * from t_native_range partition(t_native_range_201709);
 f1 |         f2          | f3
----+---------------------+----
  1 | 2017-09-01 00:00:00 |  1
  2 | 2017-09-01 00:00:00 |  2
(2 rows)
postgres=#
```

支持创建分区表时将默认分区中属于新分区的数据转移到新的分区表中

```
postgres=# CREATE TABLE t_native_range_default PARTITION OF t_native_range DEFAULT;
CREATE TABLE
postgres=#  insert into t_native_range values(2,'2017-08-01',2);
INSERT 0 1
postgres=# select * from t_native_range_default;
 f1 |         f2          | f3
----+---------------------+----
  2 | 2017-08-01 00:00:00 |  2
(1 row)
postgres=# create table t_native_range_201708 partition of t_native_range  for values from ('2017-08-0
1') to ('2017-09-01');
CREATE TABLE
postgres=# select * from t_native_range_201708;
 f1 |         f2          | f3
----+---------------------+----
  2 | 2017-08-01 00:00:00 |  2
(1 row)
postgres=# select * from t_native_range_default;
 f1 | f2 | f3
----+----+----
(0 rows)
postgres=#
```

TDSQ了PG实例自动转换。

truncate partition子分区

```
postgres=# alter table t_native_range truncate partition (t_native_range_201708) update indexes;
TRUNCATE TABLE
postgres=# select * from t_native_range_201708;
 f1 | f2 | f3
----+----+----
(0 rows)
```

此语法支持自研分区表、社区分区表（range , list, hash ）。

分区拆分

```
postgres=# \d+ t_native_range
                                Table "tbase_pg_proc.t_native_range"
 Column |            Type             | Collation | Nullable | Default | Storage | Stats target | Description
--------+-----------------------------+-----------+----------+---------+---------+--------------+-------------
 f1     | bigint                      |           |          |         | plain   |              |
 f2     | timestamp without time zone |           |          | now()   | plain   |              |
 f3     | integer                     |           |          |         | plain   |              |
Partition key: RANGE (f2)
Partitions: t_native_range_201708 FOR VALUES FROM ('2017-08-01 00:00:00') TO ('2017-09-01 00:00:00'),
            t_native_range_201709 FOR VALUES FROM ('2017-09-01 00:00:00') TO ('2017-10-01 00:00:00'),
            t_native_range_201710 FOR VALUES FROM ('2017-10-01 00:00:00') TO ('2017-11-01 00:00:00'),
            t_native_range_minvalue FOR VALUES FROM (MINVALUE) TO ('2017-08-01 00:00:00'),
            t_native_range_default DEFAULT
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES
将t_native_range_minvalue拆分成
(MINVALUE) TO ('2017-07-01 00:00:00')
('2017-07-01 00:00:00') TO ('2017-08-01 00:00:00')
postgres=# alter table t_native_range split partition t_native_range_minvalue at ('2017-07-01') into (partition t_native_range_minvalue_201706, partition t_native_range_201707);
ALTER TABLE
postgres=# \d+ t_native_range
                                Table "tbase_pg_proc.t_native_range"
 Column |            Type             | Collation | Nullable | Default | Storage | Stats target | Description
--------+-----------------------------+-----------+----------+---------+---------+--------------+-------------
 f1     | bigint                      |           |          |         | plain   |              |
 f2     | timestamp without time zone |           |          | now()   | plain   |              |
 f3     | integer                     |           |          |         | plain   |              |
Partition key: RANGE (f2)
Partitions: t_native_range_201707 FOR VALUES FROM ('2017-07-01 00:00:00') TO ('2017-08-01 00:00:00'),
            t_native_range_201708 FOR VALUES FROM ('2017-08-01 00:00:00') TO ('2017-09-01 00:00:00'),
            t_native_range_201709 FOR VALUES FROM ('2017-09-01 00:00:00') TO ('2017-10-01 00:00:00'),
            t_native_range_201710 FOR VALUES FROM ('2017-10-01 00:00:00') TO ('2017-11-01 00:00:00'),
            t_native_range_minvalue_201706 FOR VALUES FROM (MINVALUE) TO ('2017-07-01 00:00:00'),
            t_native_range_default DEFAULT
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES
```

分区合并

```
postgres=# alter table t_native_range merge partitions (t_native_range_minvalue_201706, t_native_ran
ge_201707) into partition t_native_range_minvalue_201707;
ALTER TABLE
postgres=# \d+ t_native_range
                                Table "tbase_pg_proc.t_native_range"
 Column |            Type             | Collation | Nullable | Default | Storage | Stats target | Description
--------+-----------------------------+-----------+----------+---------+---------+--------------
+-------------
 f1     | bigint                      |           |          |         | plain   |              |
 f2     | timestamp without time zone |           |          | now()   | plain   |              |
 f3     | integer                     |           |          |         | plain   |              |
Partition key: RANGE (f2)
Partitions: t_native_range_201708 FOR VALUES FROM ('2017-08-01 00:00:00') TO ('2017-09-01 00:00:0
0'),
            t_native_range_201709 FOR VALUES FROM ('2017-09-01 00:00:00') TO ('2017-10-01 00:00:00'),
            t_native_range_201710 FOR VALUES FROM ('2017-10-01 00:00:00') TO ('2017-11-01 00:00:00'),
            t_native_range_minvalue_201707 FOR VALUES FROM (MINVALUE) TO ('2017-08-01 00:00:00'),
            t_native_range_default DEFAULT
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES
```

支持不同父表下相同的子表名称

- 分区表1。

```
postgres=# create table t_same_child_partition_name_1 (f1 bigint,f2 timestamp default now(), f3 intege
r) partition by range ( f2 ) distribute by shard(f1) to group default_group;
CREATE TABLE
postgres=# create table t_same_child_partition_name_1_201709 partition of t_same_child_partition_na
me_1  for values from ('2017-09-01') to ('2017-10-01') as part_1;
CREATE TABLE
postgres=# insert into t_same_child_partition_name_1 values(1,'2017-09-01',1);
INSERT 0 1
```

- 分区表2。

```
postgres=# create table t_same_child_partition_name_2 (f1 bigint,f2 timestamp default now(), f3 intege
r) partition by range ( f2 ) distribute by shard(f1) to group default_group;
CREATE TABLE
postgres=#
postgres=# create table t_same_child_partition_name_2_201709 partition of t_same_child_partition_na
me_2  for values from ('2017-09-01') to ('2017-10-01') as part_1;
CREATE TABLE
postgres=#
postgres=# insert into t_same_child_partition_name_2 values(1,'2017-09-01',1);
INSERT 0 1
postgres=#
```

- 查询。

```
postgres=# select * from t_same_child_partition_name_1 partition(part_1);
 f1 |         f2          | f3
----+---------------------+----
  1 | 2017-09-01 00:00:00 |  1
(1 row)

postgres=# select * from t_same_child_partition_name_2 partition(part_1);
 f1 |         f2          | f3
----+---------------------+----
  1 | 2017-09-01 00:00:00 |  1
(1 row)
postgres=#
```

创建和管理视图
创建视图
创建视图示例

```
postgres=# create view t_range_view as select * from t_range;
CREATE VIEW
postgres=# select * from t_range_view;
 f1|            f2             | f3  | f4
----+----------------------------+-----+----
  1 |2017-09-27 23:17:39.674318 |   1 |
  2 |2017-09-27 23:17:39.674318 |  50 |
  2 |2017-09-27 23:17:39.674318 | 110 |
  1 |2017-09-27 23:39:45.841093 | 151 |
  3 |2017-09-27 23:17:39.674318 | 100 |
(5 rows)
```

- 数据类型重定义。

```
postgres=# create view t_range_view as select f1,f2::date from t_range;
CREATE VIEW
postgres=# select * from t_range_view;
 f1|     f2
----+------------
  1 |2017-09-27
  2 |2017-09-27
  2 |2017-09-27
  1 |2017-09-27
  3 |2017-09-27
(5 rows)
```

- 数据类型重定义,以及取别名。

```
postgres=# create view t_range_view as select f1,f2::date as mydate from t_range;
CREATE VIEW
postgres=# select * from t_range_view;
 f1|   mydate
----+------------
  1 |2017-09-27
  2 |2017-09-27
  2 |2017-09-27
  1 |2017-09-27
```

```
  3 |2017-09-27
(5 rows)
```

- tbase支持视图引用表或字段改名联动，不受影响。

```
 postgres=# \d+ t_view
                        View"tbase.t_view"
 Column | Type   | Collation | Nullable |Default | Storage  | Description
--------+---------+-----------+----------+---------+----------+-------------
 id     |integer |           |          |         | plain    |
 mc     |text    |           |          |         | extended |
View definition:
 SELECT t.id,
   t.mc
  FROM t;

postgres=# alter table t rename to t_new;
ALTER TABLE
Time: 62.875 ms
postgres=# alter table t_new rename mc tomc_new;
ALTER TABLE
Time: 22.081 ms
 postgres=# \d+ t_view
                        View"tbase.t_view"
 Column | Type   | Collation | Nullable |Default | Storage  | Description
--------+---------+-----------+----------+---------+----------+-------------
 id     |integer |           |          |         | plain    |
 mc     |text    |           |          |         | extended |
View definition:
 SELECT t_new.id,
   t_new.mc_new AS mc
  FROM t_new;
```

修改视图

通过create or replace view命令修改视图。

示例：

1）修改原指向t_range的视图指向t_list

```
create or replace view t_range_view as select * from t_list;
```

删除视图

删除视图示例

```
postgres=# drop table t;
DROP TABLE
postgres=# create table t (id int,mc text);
CREATE TABLE
postgres=# create view t_view as select * from t;
CREATE VIEW
postgres=# create view t_view_1 as select * from t_view;
CREATE VIEW
postgres=# create view t_view_2 as select * from t_view;
CREATE VIEW
postgres=# drop view t_view_2;
DROP VIEW
#使用cascade强制删除依赖对象
postgres=# drop view t_view;
ERROR:  cannot drop view t_view because other objects depend on it
DETAIL:  view t_view_1 depends on view t_view
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
postgres=# drop view t_view cascade;
NOTICE:  drop cascades to view t_view_1
DROP VIEW
```

物化视图
创建物化视图

物化视图是一种特殊的物理表，“物化”(Materialized)视图是相对普通视图而言的。普通视图是虚拟表，应用的局限性大，任何对视图的查询，都实际上转换为视图SQL语句的查询。这样对整体查询性能的提高，并没有实质上的好处。而物化视图是一张实际存在的表，是占有数据库磁盘空间的。
物化视图并不像普通视图那样，只有在使用的时候才去读取数据，而是预先计算并保存表连接或者聚集等比较耗时操作的结果，这样大大提高了读取的速度，特别适合抽取大数据量表的某些信息。
通过create MATERIALIZED VIEW创建物化视图
示例：

```
postgres=# CREATE MATERIALIZED VIEW t_range_mv AS select f1,f2::date from t_range;
SELECT 5
postgres=# select * from t_range_mv;
 f1 |     f2
----+------------
  1 | 2017-09-27
  2 | 2017-09-27
  2 | 2017-09-27
  1 | 2017-09-27
  3 | 2017-09-27
(5 rows)
```

访问物化视图

```
postgres=# select * from t_range_mv;
 f1 |     f2
----+------------
  1 | 2017-09-27
  2 | 2017-09-27
  2 | 2017-09-27
  1 | 2017-09-27
  3 | 2017-09-27
(5 rows)

postgres=# insert into t_range(f1,f3) values(5,10);
INSERT 0 1

postgres=# select * from t_range;
 f1 |            f2            | f3  | f4
----+--------------------------+-----+----
  1 | 2017-09-27 23:17:39.674318 |   1 |
  2 | 2017-09-27 23:17:39.674318 |  50 |
  5 | 2017-09-27 23:50:51.576173 |  10 |
  2 | 2017-09-27 23:17:39.674318 | 110 |
  1 | 2017-09-27 23:39:45.841093 | 151 |
  3 | 2017-09-27 23:17:39.674318 | 100 |
(6 rows)
```

增量数据刷新

```
postgres=# select * from t_range_mv ;
 f1 |     f2
----+------------
  1 | 2017-09-27
  2 | 2017-09-27
  2 | 2017-09-27
  1 | 2017-09-27
  3 | 2017-09-27
(5 rows)
postgres=# REFRESH MATERIALIZED VIEW t_range_mv;
REFRESH MATERIALIZED VIEW
postgres=# select * from t_range_mv ;
 f1 |     f2
----+------------
  1 | 2017-09-27
  2 | 2017-09-27
  5 | 2017-09-27
  2 | 2017-09-27
  1 | 2017-09-27
  3 | 2017-09-27
(6 rows)
```

> 注意：
>
> 物化视图数据存储在cn节点上面，每个cn节点各有一份相同的数据。

视图触发器创建与删除

创建视图触发器

```
#创建视图
postgres=# create or replace view t1_view as select * from t1;
CREATE VIEW
postgres=#

#创建触发器函数
postgres=# CREATE OR REPLACE FUNCTION t1_view_insert_trigger_func () RETURNS trigger AS
$body$
BEGIN
   INSERT INTO t1 values(NEW.ID,NEW.nickname);
   RETURN NEW;
END;
$body$
LANGUAGE 'plpgsql';
CREATE FUNCTION
postgres=#
#创建触发器
postgres=# CREATE TRIGGER t1_view_insert_trigger INSTEAD OF INSERT ON t1_view FOR EACH ROW
EXECUTE PROCEDURE t1_view_insert_trigger_func();
CREATE TRIGGER
#验证
postgres=# INSERT INTO t1_view values(2,'TDSQL PG');
INSERT 0 1
postgres=# select * from t1;
 id | nickname
----+----------
  1 | TDSQL PG
  2 | TDSQL PG
(2 rows)
```

删除视图触发器

```
postgres=# drop trigger t1_view_insert_trigger on t1_view ;
DROP TRIGGER
postgres=#
```

创建和管理序列
自增列与序列的用法
序列的创建和访问

1. 建立序列。

```
postgres=# create sequence tbase_seq;
CREATE SEQUENCE
```

2. 建立序列，不存在时才创建。

```
postgres=# create sequence IF NOT EXISTS tbase_seq;
NOTICE:  relation "tbase_seq" already exists, skipping
CREATE SEQUENCE
```

3. 查看序列当前的使用状况。

```
postgres=# \x
Expanded display is on.
postgres=# select * from tbase_seq ;
-[ RECORD 1 ]-
last_value | 1
log_cnt    | 0
is_called | f
```

4. 获取序列的下一个值。

```
postgres=# select nextval('tbase_seq');
 nextval
---------
       1
```

5. 获取序列的当前值，这个需要在访问nextval()后才能使用。

```
postgres=# select currval('tbase_seq');
 currval
---------
       1
```

6. 你可以后下面的方式来获取序列当前使用到那一个值。

```
postgres=# select last_value from tbase_seq;
 last_value
------------
          3
```

7. 设置序列当前值。

```
postgres=# select setval('tbase_seq',1);
 setval
--------
      1
```

序列在DML中使用

```
postgres=# create table t(id int ,nickname varchar(15));
postgres=# INSERT INTO t (id,nickname) VALUES(nextval('tbase_seq'),'TBase好');
postgres=# select * from t;
 id | nickname
----+-----------
  1 | TDSQL PG
  2 | TDSQL PG好
(2 rows)
```

序列做为字段的默认值使用

```
postgres=# alter table t alter column id set default nextval('tbase_seq');
postgres=# INSERT INTO t (nickname) VALUES('hello TDSQL PG');
INSERT 0 1
postgres=# select * from t;
 id |  nickname
----+-------------
  3 | hello TDSQL PG
  1 | TDSQL PG
  2 | TDSQL PG好
(3 rows)
```

序列做为字段类型使用

```
postgres=# drop table t;
DROP TABLE
postgres=# create table t (id serial not null,nickname text);
CREATE TABLE
postgres=# INSERT INTO t (nickname) VALUES('hello TDSQL PG');
INSERT 0 1
postgres=# select * from t;
 id |  nickname
----+-------------
  1 | hello TDSQL PG
(1 row)
```

删除序列

```
postgres=# drop sequence tbase_seq;
DROP SEQUENCE
```

1. 删除序列，不存在时跳过。

```
postgres=# drop sequence  IF EXISTS  tbase_seq;
NOTICE:  sequence "tbase_seq" does not exist, skipping
DROP SEQUENCE
```

查询gtm中的序列对象列表

```
postgres=# select *  from pg_list_storage_sequence();
```

创建和管理同义词
部分同义词示例
创建同义词

```
postgres=# create table public.syn(f1 int,f2 text);
CREATE TABLE
postgres=# insert into  public.syn values(1,'TDSQL PG');
INSERT 0 1
postgres=# create synonym syn_copy for public.syn;
CREATE SYNONYM
```

同义词访问

```
postgres=# select * from syn_copy;
 f1 |  f2
----+-------
  1 | TDSQL PG
(1 row)
postgres=# insert into syn_copy values(2,'pg');
INSERT 0 1
postgres=# update syn_copy set f2='postgres' where f1=2;
UPDATE 1
postgres=# delete from syn_copy where f1=2;
DELETE 1
```

同义词系统表

```
postgres=# select * from pg_synonym ;
 synname  | synnamespace | objnamespace | objname | objdblink | synowner
----------+--------------+--------------+---------+-----------+----------
 syn_copy |        19509 | public       | syn     |           |       10
(1 row)
postgres=#
```

关于PL/SQL
PL/SQL介绍

PL / SQL是SQL的过程扩展，是一种可移植的高性能事务处理语言，PL/SQL把数据操作和查询语句组织在过程化代码中，通过逻辑判断、循环等操作实现高级复杂功能。
PL/SQL具有高性能、可移植、可扩展、便于管理等优点。

PL/SQL控制语句
条件选择语句

以下几个示例，说明条件选择语句的类型以及使用方法。

IF...THEN...END IF

```
postgres=# CREATE OR REPLACE FUNCTION f26() RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     IF random()>0.5 THEN
postgres$#         RAISE NOTICE '随机数大于0.5';
postgres$#     END IF;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# select f26();
NOTICE:  随机数大于0.5
 f26
-----

(1 row)
```

IF...THEN...ELSE...END IF

```
postgres=# CREATE OR REPLACE FUNCTION f26() RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     IF random()>0.99 THEN
postgres$#         RAISE NOTICE '随机数大于0.99';
postgres$#     ELSE
postgres$#         RAISE NOTICE '随机数小于或等于0.99';
postgres$#     END IF;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
```

```
CREATE FUNCTION
postgres=# select f26();
NOTICE:  随机数小于或等于0.99
 f26
-----

(1 row)
```

IF...THEN...ELSIF...THEN...ELSE...END IF

```
postgres=# CREATE OR REPLACE FUNCTION f26() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_float8 float8 := random();
postgres$# BEGIN
postgres$#     IF v_float8>0.99 THEN
postgres$#         RAISE NOTICE '随机数大于0.99';
postgres$#     ELSIF v_float8>0.5 THEN
postgres$#         RAISE NOTICE '随机数大于0.50';
postgres$#     ELSIF v_float8>0.25 THEN
postgres$#         RAISE NOTICE '随机数大于0.25';
postgres$#     ELSE
postgres$#         RAISE NOTICE '随机数小于或等于0.25';
postgres$#     END IF;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f26();
NOTICE:  随机数大于0.50
 f26
-----

(1 row)
```

CASE语句

```
postgres=# CREATE OR REPLACE FUNCTION f26() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_float8 float8 := random();
postgres$# BEGIN
postgres$#     CASE
postgres$#     WHEN v_float8>0.99 THEN
postgres$#         RAISE NOTICE '随机数大于0.99';
postgres$#     WHEN v_float8>0.5 THEN
postgres$#         RAISE NOTICE '随机数大于0.50';
postgres$#     WHEN v_float8>0.25 THEN
postgres$#         RAISE NOTICE '随机数大于0.25';
postgres$#     ELSE
postgres$#         RAISE NOTICE '随机数小于或等于0.25';
postgres$#     END CASE;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f26();
NOTICE:  随机数大于0.50
 f26
-----

(1 row)
```

循环语句

以下几个示例，说明循环语句的类型以及使用方法。

LOOP循环

```
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#      v_id INTEGER := 1;
postgres$# BEGIN
postgres$#     LOOP
postgres$#         RAISE NOTICE '%',v_id;
postgres$#         EXIT WHEN random()>0.8;
postgres$#         v_id := v_id + 1;
postgres$#     END LOOP ;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  1
NOTICE:  2
 f27
-----


(1 row)
```

使用EXIT退出循环

```
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_id INTEGER := 1;
postgres$#     v_random float8 ;
postgres$# BEGIN
postgres$#     LOOP
postgres$#       RAISE NOTICE '%',v_id;
postgres$#       v_id := v_id + 1;
postgres$#       v_random := random();
postgres$#       IF v_random > 0.8 THEN
postgres$#           RETURN;
```

```
postgres$#         END IF;
postgres$#     END LOOP ;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  1
NOTICE:  2
NOTICE:  3
NOTICE:  4
NOTICE:  5
 f27
-----

(1 row)

postgres=#
```

使用RETURN退出循环返回

WHILE循环

```
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_id INTEGER := 1;
postgres$#     v_random float8 := random() ;
postgres$# BEGIN
postgres$#     WHILE v_random > 0.8 LOOP
postgres$#         RAISE NOTICE '%',v_id;
postgres$#         v_id := v_id + 1;
postgres$#         v_random = random();
postgres$#     END LOOP;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  1
 f27
-----

(1 row)
```

FOR循环

```
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     FOR i IN 1..3 LOOP
postgres$#         RAISE NOTICE 'i = %',i;
postgres$#     END LOOP;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  i = 1
NOTICE:  i = 2
NOTICE:  i = 3
 f27
-----

(1 row)
```

```
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     FOR i IN REVERSE 3..1 LOOP
postgres$#         RAISE NOTICE 'i = %',i;
postgres$#     END LOOP;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  i = 3
NOTICE:  i = 2
NOTICE:  i = 1
 f27
-----

(1 row)
```

使用REVERSE递减

```
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     FOR i IN 1..8 BY 2 LOOP
postgres$#         RAISE NOTICE 'i = %',i;
postgres$#     END LOOP;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  i = 1
NOTICE:  i = 3
NOTICE:  i = 5
NOTICE:  i = 7
 f27
-----

(1 row)
```

使用BY设置步长

FOR循环查询结果

```
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_rec RECORD;
postgres$# BEGIN
postgres$#     FOR v_rec IN SELECT * FROM public.t LOOP
postgres$#         RAISE NOTICE '%',v_rec;
postgres$#     END LOOP;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  (1,TDSQL PG)
NOTICE:  (2,pgxz)
 f27
-----

(1 row)
```

FOREACH循环一个数组

```
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_random_arr float8[]:=ARRAY[random(),random()];
postgres$#     v_random float8;
postgres$# BEGIN
postgres$#     FOREACH v_random IN ARRAY v_random_arr LOOP
postgres$#         RAISE NOTICE '%',v_random ;
postgres$#     END LOOP;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  0.452758576720953
NOTICE:  0.975814974401146
 f27
-----
```

```
(1 row)

postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_random_arr float8[][]:=ARRAY[ARRAY[random(),random()],ARRAY[random(),random()]];
postgres$#     v_random float8;
postgres$# BEGIN
postgres$#     FOREACH v_random SLICE 0 IN ARRAY v_random_arr LOOP
postgres$#         RAISE NOTICE '%',v_random ;
postgres$#     END LOOP;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  0.0588191924616694
NOTICE:  0.368828620761633
NOTICE:  0.813376842066646
NOTICE:  0.415377039927989
 f27
-----

(1 row)
```

循环会通过计算expression得到的数组的个体元素进行迭代

```
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_random_arr float8[][]:=ARRAY[ARRAY[random(),random()],ARRAY[random(),random()]];
postgres$#     v_random float8[];
postgres$# BEGIN
postgres$#     FOREACH v_random SLICE 1 IN ARRAY v_random_arr LOOP
postgres$#         RAISE NOTICE '%',v_random ;
postgres$#     END LOOP;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  {0.578366641886532,0.78098024148494}
NOTICE:  {0.783956411294639,0.450278480071574}
 f27
-----
```

(1 row)

通过一个正SLICE值，FOREACH通过数组的切片而不是单一元素迭代

其他控制语句

以下几个示例，说明其他控制语句的类型以及使用方法。

执行一个没有结果的命令

```
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS void AS
postgres-# $$
postgres$# BEGIN
postgres$#     perform f27(1);
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
 f27
-----

(1 row)
```

获取执行结果

```
postgres=# DROP FUNCTION f27(INTEGER);
DROP FUNCTION
postgres=# CREATE OR REPLACE FUNCTION f27(a_id INTEGER) RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_mc TEXT;
postgres$# BEGIN
postgres$#     SELECT mc INTO v_mc FROM t WHERE id=a_id;
postgres$#     IF FOUND THEN
postgres$#         RAISE NOTICE '查询到记录，值为%',v_mc;
postgres$#     ELSE
postgres$#         RAISE NOTICE '查不到记录' ;
postgres$#     END IF;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
```

CREATE FUNCTION
postgres=# SELECT f27(1);
NOTICE: 查询到记录，值为TDSQLPG
 f27

(1 row)

postgres=# SELECT f27(3);
NOTICE: 查不到记录
 f27

(1 row)

获取影响行数

```
postgres=# CREATE OR REPLACE FUNCTION f27(a_id INTEGER) RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_mc TEXT;
postgres$#     v_row_count BIGINT;
postgres$# BEGIN
postgres$#     SELECT mc INTO v_mc FROM t WHERE id=a_id;
postgres$#     GET DIAGNOSTICS v_row_count = ROW_COUNT;
postgres$#     RAISE NOTICE '查询到的记录数为 % ',v_row_count;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION

postgres=# SELECT f27(1);
NOTICE:  查询到的记录数为 1
 f27
-----

(1 row)
postgres=# SELECT f27(3);
NOTICE:  查询到的记录数为 0
 f27
-----

(1 row)
```

PL/SQL集合和记录

集合概述

集合是存储过程复合变量，可以按顺序存储多个类型相同的元素，类似于一维数组。整个集合可以作为子程序的参数进行传递（如果发送或者接收的子程序都不是独立的子程序）。
集合中的元素有唯一的下标描述它在集合中的位置。访问集合的元素，可以使用下标命名方式：集合名(下标)。
集合方法是内建的存储过程，可以返回集合的信息或者对集合进行操作。需要使用“.”表示调用集合，格式为：集合名.方法名。例如，集合名.COUNT 方法用于返回该集合元素的数量。
TDSQL PG支持三种集合类型：关联数组，可变数组和嵌套表。
目前，关联数组，嵌套表和可变数组只能在plpgsql定义的function、procdure、package、block中使用。支持select bulk collect into 到对应的变量中。

关联数组
关联数组定义和初始化

示例：

由不同类型索引的关联数组类型

```
set  enable_oracle_compatible=on;
drop table dept cascade;
CREATE TABLE DEPT
(DEPTNO NUMBER(2) CONSTRAINT PK_DEPT PRIMARY KEY,
DNAME VARCHAR2(14) ,
LOC VARCHAR2(13) ) ;
drop table emp cascade;
CREATE TABLE EMP
(EMPNO NUMBER(4) CONSTRAINT PK_EMP PRIMARY KEY,
ENAME VARCHAR2(10),
JOB VARCHAR2(9),
MGR NUMBER(4),
HIREDATE DATE,
SAL NUMBER(7,2),
COMM NUMBER(7,2),
DEPTNO NUMBER(2));
INSERT INTO DEPT VALUES
(10,'ACCOUNTING','NEW YORK');
INSERT INTO DEPT VALUES (20,'RESEARCH','DALLAS');
INSERT INTO DEPT VALUES
(30,'SALES','CHICAGO');
INSERT INTO DEPT VALUES
(40,'OPERATIONS','BOSTON');
INSERT INTO EMP VALUES
(7369,'SMITH','CLERK',7902,to_date('17-12-1980','dd-mm-yyyy'),800,NULL,20);
INSERT INTO EMP VALUES
(7499,'ALLEN','SALESMAN',7698,to_date('20-2-1981','dd-mm-yyyy'),1600,300,30);
INSERT INTO EMP VALUES
(7521,'WARD','SALESMAN',7698,to_date('22-2-1981','dd-mm-yyyy'),1250,500,30);
INSERT INTO EMP VALUES
(7566,'JONES','MANAGER',7839,to_date('2-4-1981','dd-mm-yyyy'),2975,NULL,20);
INSERT INTO EMP VALUES
(7654,'MARTIN','SALESMAN',7698,to_date('28-9-1981','dd-mm-yyyy'),1250,1400,30);
INSERT INTO EMP VALUES
(7698,'BLAKE','MANAGER',7839,to_date('1-5-1981','dd-mm-yyyy'),2850,NULL,30);
INSERT INTO EMP VALUES
(7782,'CLARK','MANAGER',7839,to_date('9-6-1981','dd-mm-yyyy'),2450,NULL,10);
INSERT INTO EMP VALUES
```

```
(7788,'SCOTT','ANALYST',7566,to_date('19-04-1987','dd-mm-yyyy'),3000,NULL,20);
INSERT INTO EMP VALUES
(7839,'KING','PRESIDENT',NULL,to_date('17-11-1981','dd-mm-yyyy'),5000,NULL,10);
INSERT INTO EMP VALUES
(7844,'TURNER','SALESMAN',7698,to_date('8-9-1981','dd-mm-yyyy'),1500,0,30);
INSERT INTO EMP VALUES
(7876,'ADAMS','CLERK',7788,to_date('23-05-1987', 'dd-mm-yyyy'),1100,NULL,20);
INSERT INTO EMP VALUES
(7900,'JAMES','CLERK',7698,to_date('3-12-1981','dd-mm-yyyy'),950,NULL,30);
INSERT INTO EMP VALUES
(7902,'FORD','ANALYST',7566,to_date('3-12-1981','dd-mm-yyyy'),3000,NULL,20);
INSERT INTO EMP VALUES
(7934,'MILLER','CLERK',7782,to_date('23-1-1982','dd-mm-yyyy'),1300,NULL,10);
drop table bonus cascade;
CREATE TABLE BONUS
(
ENAME VARCHAR2(10)      ,
JOB VARCHAR2(9)  ,
SAL NUMBER,
COMM NUMBER
) ;
drop table salgrade cascade;
CREATE TABLE SALGRADE
( GRADE NUMBER,
LOSAL NUMBER,
HISAL NUMBER );
INSERT INTO SALGRADE VALUES (1,700,1200);
INSERT INTO SALGRADE VALUES (2,1201,1400);
INSERT INTO SALGRADE VALUES (3,1401,2000);
INSERT INTO SALGRADE VALUES (4,2001,3000);
INSERT INTO SALGRADE VALUES (5,3001,9999);

-- TestPoint :  Associative array with different index type

DECLARE
  TYPE aa_type IS TABLE OF INTEGER INDEX BY PLS_INTEGER;
  TYPE bb_type IS TABLE OF number(6,2) INDEX BY VARCHAR2(2);
  TYPE cc_type IS TABLE OF char(5) INDEX BY VARCHAR(2);
  TYPE dd_type IS TABLE OF date INDEX BY long;

  aa aa_type;                    -- associative array
  bb bb_type;
  cc cc_type;
  dd dd_type;
BEGIN
  aa(1):=3;
  aa(2):=6;
```

```
  perform dbms_output.serveroutput('t');
  --raise notice '%,%',aa.count,aa.limit;
   perform DBMS_OUTPUT.PUT('aa.COUNT = ');
   perform DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa.COUNT), 'NULL'));

   perform DBMS_OUTPUT.PUT('aa.LIMIT = ');
   perform DBMS_OUTPUT.PUT_LINE(NVL(to_char(aa.LIMIT), 'NULL'));

  bb('a'):=1.1;
  bb('b'):=2.222;
  raise notice '%',bb.count;
  raise notice '%,%',bb('a'),bb('b');
  cc('a'):='chqin';
  cc('b'):='tbase';
  raise notice '%',bb.count;
  raise notice '%,%',cc('a'),cc('b');
  dd('a'):=to_date('2021-09-23 15:12:55','yyyy-mm-dd hh24:mi:ss');
  dd('b'):=to_date('2021-09-24 15:12:55','yyyy-mm-dd hh24:mi:ss');
  raise notice '%',bb.count;
  raise notice '%,%',dd('a'),dd('b');

end;
/

-- TestPoint : Associative Array index type long

declare
    TYPE population IS TABLE OF long INDEX BY long;
    myvar  population;        -- Associative array variable
    i  long;                  -- Scalar variable

BEGIN
    raise notice 'Begin Count=%', myvar.COUNT;
    myvar('v1')  := 'v1';
    myvar('v2')  := 'v2';
    raise notice 'Initialed Count=%', myvar.COUNT;

    raise notice 'Print first to last';
    i := myvar.FIRST();
    WHILE i IS NOT NULL LOOP
        raise notice 'Loop[%] is %',i, myvar(i);
        i := myvar.NEXT(i);
    END LOOP;
end;
/

-- TestPoint : Associative Array index type varchar
```

```
declare
    TYPE population IS TABLE OF long raw INDEX BY varchar(20);
    myvar  population;        -- Associative array variable
    i  varchar(20);                  -- Scalar variable
BEGIN
    raise notice 'Begin Count=%', myvar.COUNT;

    myvar('1')  := '2000';
    myvar('2')  := '750000';
    myvar('3')  := '1000000';
    myvar('4')  := '2001';
    myvar('5')  := '20210617';
    myvar('10') := '20210618';

    raise notice 'Initialed Count=%', myvar.COUNT;

    raise notice 'Print first to last';
    i := myvar.FIRST();
    WHILE i IS NOT NULL LOOP
        raise notice 'Loop[%] is %',i, myvar(i);
        i := myvar.NEXT(i);
    END LOOP;

    IF myvar.EXISTS('12') THEN
        raise notice 'myvar(''12'') is exist';
    ELSE
        raise notice 'myvar(''12'') is not exist';
    END IF;

    myvar('12') := '1001';
    raise notice 'INSERT (''12'') = 1001, COUNT=%', myvar.COUNT;

    IF myvar.EXISTS('12') THEN
        raise notice 'myvar(''12'') is exist';
    ELSE
        raise notice 'myvar(''12'') is not exist';
    END IF;

    myvar('20') := '30006';

    raise notice 'Print last to first';
    i := myvar.LAST;
    WHILE i IS NOT NULL LOOP
        raise notice 'Loop[%] is %',i, myvar(i);
        i := myvar.PRIOR(i);
    END LOOP;
```

```
    raise notice 'End Count=%', myvar.COUNT;
END;
/

-- TestPoint :  Associative Array : nclob type index of PLS_INTEGER

declare
    TYPE population IS TABLE OF nclob INDEX BY PLS_INTEGER;
    myvar  population;        -- Associative array variable
    i  PLS_INTEGER;                  -- Scalar variable
BEGIN
    raise notice 'Begin Count=%', myvar.COUNT;

    myvar(1)  := '2000';
    myvar(2)  := '750000';
    myvar(3)  := '1000000';
    myvar(4)  := '2001';
    myvar(5)  := '20210617';
    myvar(10) := '20210618';

    raise notice 'Initialed Count=%', myvar.COUNT;

    raise notice 'Print first to last';
    i := myvar.FIRST();
    WHILE i IS NOT NULL LOOP
        raise notice 'Loop[%] is %',i, myvar(i);
        i := myvar.NEXT(i);
    END LOOP;

    IF myvar.EXISTS(12) THEN
        raise notice 'myvar(''12'') is exist';
    ELSE
        raise notice 'myvar(''12'') is not exist';
    END IF;

    myvar(12) := '1001';
    raise notice 'INSERT (''12'') = 1001, COUNT=%', myvar.COUNT;

    IF myvar.EXISTS(12) THEN
        raise notice 'myvar(''12'') is exist';
    ELSE
        raise notice 'myvar(''12'') is not exist';
    END IF;

    myvar(20) := '30006';
```

```
    raise notice 'Print last to first';
    i := myvar.LAST;
    WHILE i IS NOT NULL LOOP
        raise notice 'Loop[%] is %',i, myvar(i);
        i := myvar.PRIOR(i);
    END LOOP;

    raise notice 'End Count=%', myvar.COUNT;
END;
/
declare
v1 string(20);
begin
  v1:='v';
end;
/

-- TestPoint : Associative Array : emp.ename%type type index of PLS_INTEGER

declare
    TYPE population IS TABLE OF emp.ename%type INDEX BY PLS_INTEGER;
    myvar  population;        -- Associative array variable
    i  PLS_INTEGER;                  -- Scalar variable
    v1 emp.ename%type;
BEGIN
   select ename into v1 from emp where empno=7788;
   myvar(1):=v1;
   select ename into v1 from emp where empno=7521;
   myvar(2):=v1;
   raise notice '%',myvar(1);
      i := myvar.FIRST();
    WHILE i IS NOT NULL LOOP
        raise notice 'Loop[%] is %',i, myvar(i);
        i := myvar.NEXT(i);
    END LOOP;

end;
/

-- TestPoint : Associative Array : emp.hiredate%type type index of varchar2(20)
-- ERROR:  input value length is 40; too long for type varchar2(10)

declare
    TYPE population IS TABLE OF emp.hiredate%type  INDEX BY varchar2(10);
    myvar  population;        -- Associative array variable
--      i  varchar2(10);                  -- Scalar variable
    i emp.ename%type;
```

```
    v1 emp.ename%type;
    v2 emp.hiredate%type;
BEGIN
    raise notice 'Begin Count=%', myvar.COUNT;
   select ename,hiredate into v1,v2 from emp where empno=7788;
    myvar(v1)  := v2;
   select ename,hiredate into v1,v2 from emp where empno=7521;
    myvar(v1)  := v2;
    raise notice 'Initialed Count=%', myvar.COUNT;

    raise notice 'Print first to last';
    i := myvar.FIRST();
    WHILE i IS NOT NULL LOOP
        raise notice 'Loop[%] is %',i, myvar(i);
        i := myvar.NEXT(i);
    END LOOP;

    IF myvar.EXISTS('12') THEN
        raise notice 'myvar(''12'') is exist';
    ELSE
        raise notice 'myvar(''12'') is not exist';
    END IF;

    raise notice 'Print last to first';
    i := myvar.LAST;
    WHILE i IS NOT NULL LOOP
        raise notice 'Loop[%] is %',i, myvar(i);
        i := myvar.PRIOR(i);
    END LOOP;

END;
/

-- TestPoint : Associative Array with table columns: varchar2 index by varchar2

drop table if exists chqin;
create table chqin (f1 varchar2(10), f2 varchar2(20));
insert into chqin values('1','tbase1');
insert into chqin values('2','tbase2');

declare
    TYPE population IS TABLE OF chqin.f2%type  INDEX BY varchar2(20);
    myvar  population;        -- Associative array variable
    i  varchar2(10);                  -- Scalar variable
    v1 chqin.f1%type;
    v2 chqin.f2%type;
BEGIN
```

```
    raise notice 'Begin Count=%', myvar.COUNT;
   select f1,f2 into v1,v2 from chqin where f1='1';
    myvar(v1)  := v2;
   select f1,f2 into v1,v2 from chqin where f1='2';
    myvar(v1)  := v2;
    raise notice 'Initialed Count=%', myvar.COUNT;

    raise notice 'Print first to last';
    i := myvar.FIRST();
    WHILE i IS NOT NULL LOOP
       raise notice 'Loop[%] is %',i, myvar(i);
       i := myvar.NEXT(i);
    END LOOP;

    IF myvar.EXISTS('12') THEN
       raise notice 'myvar(''12'') is exist';
    ELSE
       raise notice 'myvar(''12'') is not exist';
    END IF;

    raise notice 'Print last to first';
    i := myvar.LAST;
    WHILE i IS NOT NULL LOOP
       raise notice 'Loop[%] is %',i, myvar(i);
       i := myvar.PRIOR(i);
    END LOOP;

END;
/
drop table if exists chqin;

-- TestPoint : Associative Array with table columns

drop table if exists chqin;
create table chqin (f1 varchar2(10), f2 long,f3 nclob,f4 interval year to month,f5 timestamp(6) with local
time zone,f6 raw(8),f7 nchar(6));
insert into chqin values('1','tbase1','tbase1',interval '3-2' year to month, '2021-06-22 16:02:07.067',utl_r
aw.cast_to_raw('raw1'),'tbase1');
insert into chqin values('2','tbase2','tbase2',interval '4-2' year to month, '2021-07-22 16:02:07.067',utl_r
aw.cast_to_raw('raw2'),'tbase2');

-- TestPoint : Associative Array: varchar2 index by long --core dump
declare
    TYPE population IS TABLE OF chqin.f1%type  INDEX BY long;
    myvar  population;        -- Associative array variable
    i  long;                  -- Scalar variable
    v1 chqin.f1%type;
```

```
    v2 chqin.f2%type;
BEGIN
    raise notice 'Begin Count=%', myvar.COUNT;
   select f1,f2 into v1,v2 from chqin where f1='1';
    myvar(v2)  := v1;
   select f1,f2 into v1,v2 from chqin where f1='2';
    myvar(v2)  := v1;
    raise notice 'Initialed Count=%', myvar.COUNT;

    raise notice 'Print first to last';
    i := myvar.FIRST();
    WHILE i IS NOT NULL LOOP
        raise notice 'Loop[%] is %',i, myvar(i);
        i := myvar.NEXT(i);
    END LOOP;
end;
/

-- TestPoint : Associative Array: nclob index by long  --core

declare
    TYPE population IS TABLE OF chqin.f3%type  INDEX BY long;
    myvar  population;        -- Associative array variable
    i  long;                  -- Scalar variable
    v1 chqin.f1%type;
    v2 chqin.f2%type;
BEGIN
    raise notice 'Begin Count=%', myvar.COUNT;
   select f3,f2 into v1,v2 from chqin where f1='1';
    myvar(v2)  := v1;
   select f3,f2 into v1,v2 from chqin where f1='2';
    myvar(v2)  := v1;
    raise notice 'Initialed Count=%', myvar.COUNT;

    raise notice 'Print first to last';
    i := myvar.FIRST();
    WHILE i IS NOT NULL LOOP
        raise notice 'Loop[%] is %',i, myvar(i);
        i := myvar.NEXT(i);
    END LOOP;
end;
/
```

关联数组的使用

示例：

关联数组在包中的使用

```
drop package My_Types;
CREATE OR REPLACE PACKAGE My_Types AUTHID CURRENT_USER IS
TYPE My_AA IS TABLE OF VARCHAR2(20) INDEX BY PLS_INTEGER;
FUNCTION Init_My_AA RETURN My_AA;
END My_Types;
/
CREATE OR REPLACE PACKAGE BODY My_Types IS
FUNCTION Init_My_AA RETURN My_AA IS
Ret My_AA;
BEGIN
Ret(-10) := '-ten';
Ret(0) := 'zero';
Ret(1) := 'one';
Ret(2) := 'two';
Ret(3) := 'three';
Ret(4) := 'four';
Ret(9) := 'nine';
RETURN Ret;
END Init_My_AA;
END My_Types;
/

DECLARE
v CONSTANT My_Types.My_AA := My_Types.Init_My_AA();
BEGIN
DECLARE
Idx PLS_INTEGER := v.FIRST();
BEGIN
WHILE Idx IS NOT NULL LOOP
raise notice '%,%',Idx,v(Idx);

-- DBMS_OUTPUT.PUT_LINE(TO_CHAR(Idx, '999')||LPAD(v(Idx), 7));
Idx := v.NEXT(Idx);
END LOOP;
END;
END;
/
```

可变数组

可变数组是一个其元素数可以从零（空）到声明的最大大小变化的数组。当创建可变数组时，必须指定可变数组的最大长度。
要访问varray变量的元素，请使用语法variable_name(index)。index的下限是1;上限是当前的元素数。
示例：

```
set enable_oracle_compatible=on;
--匿名块中使用varray，使用dbms_output 输出array中的数据
declare
   type v_arr is varray(10) of varchar2(30);
   my_arr v_arr:= v_arr('1','2','3');
begin
   for i in 1..my_arr.count
   loop
   perform dbms_output.serveroutput('t');
   perform         dbms_output.put_line(my_arr[i]);
   end loop;
end;
/

--初始化1个字符后，输出溢出值和正常值
declare
   type v_arr is varray(3) of varchar2(30);
   my_arr v_arr:= v_arr(null);
begin
   my_arr[1]:='插入';
      perform dbms_output.serveroutput('t');
   perform DBMS_OUTPUT.PUT_LINE(my_arr[1]);
      perform dbms_output.serveroutput(my_arr[2]);
      exception WHEN others THEN
    BEGIN
       perform dbms_output.serveroutput('t');
    perform DBMS_OUTPUT.PUT_LINE('error ...');
       my_arr.extend;
       perform DBMS_OUTPUT.PUT_LINE(my_arr[1]||'--'||my_arr[2]);
       end;
end;
/
--输出溢出值
declare
   type v_arr is varray(3) of varchar2(30);
   my_arr v_arr:= v_arr(null);
begin
   my_arr[4]:='插入';
```

```
    perform DBMS_OUTPUT.PUT_LINE(my_arr[4]);
end;
/

declare
    type v_arr is varray(3) of varchar2(30);
    my_arr v_arr:= v_arr(null);
begin
    my_arr[2]:='插入';
    perform DBMS_OUTPUT.PUT_LINE(my_arr[2]);
end;
/

--验证nesttable 类型
declare
  type dateArray is table of varchar2(10);
  date_val dateArray:=dateArray('20201231','20200930');
  begin
    for i in 1 .. date_val.count loop
      perform dbms_output.serveroutput('t');
      perform  dbms_output.put_line('结果：'||date_val[i]);
    end loop;
end;
/
```

嵌套表

嵌套表是一种列类型，它以无特定顺序存储未指定数量的行。当一个嵌套表值从数据库检索到一个PL/SQL 嵌套表变量中时，PL/SQL会为这些行提供从 1 开始的连续索引。使用这些索引，可以访问嵌套表变量的各个行。语法是variable_name(index)。
支持的方法如下：

方法	描述
exists（index）	索引处的元素是否存在
count	当前集合中的元素总个数
limit	集合元素索引的最大值，返回null
first	返回集合第一个元素索引
last	返回集合最后一个元素索引
prior	当前元素的前一个
next	当前元素的后一个
extend	扩展集合的容量
trim	从集合的尾部删除元素
delete	按索引删除集合元素

目前TDSQL PG不支持使用外部定义的嵌套表类型，只能使用select bulk collect into
示例：
嵌套表读取

```
drop table if exists emp;
CREATE TABLE EMP(EMPNO NUMBER(4) CONSTRAINT PK_EMP PRIMARY KEY,ENAME VARCHAR2(20),J
OB VARCHAR2(9),MGR NUMBER(4), HIREDATE DATE,SAL NUMBER(7,2),COMM NUMBER(7,2), DEPTNO
NUMBER(4));
INSERT INTO EMP VALUES(7369,'SMITH','CLERK',7902,to_date('17-12-1980','dd-mm-yyyy'),800,NULL,2
00);
INSERT INTO EMP VALUES(7499,'ALLEN','SALESMAN',7698,to_date('20-2-1981','dd-mm-yyyy'),1600,30
0,300);
INSERT INTO EMP VALUES(7521,'WARD','SALESMAN',7698,to_date('22-2-1981','dd-mm-yyyy'),1250,50
0,300);
INSERT INTO EMP VALUES(7566,'JONES','MANAGER',7839,to_date('2-4-1981','dd-mm-yyyy'),2975,NUL
L,200);
INSERT INTO EMP VALUES(7654,'MARTIN','SALESMAN',7698,to_date('28-9-1981','dd-mm-yyyy'),1250,1
```

```
400,300);
INSERT INTO EMP VALUES(7698,'BLAKE','MANAGER',7839,to_date('1-5-1981','dd-mm-yyyy'),2850,NUL
L,300);
INSERT INTO EMP VALUES(7782,'CLARK','MANAGER',7839,to_date('9-6-1981','dd-mm-yyyy'),2450,NUL
L,100);
INSERT INTO EMP VALUES(7788,'SCOTT','ANALYST',7566,to_date('19-04-87','dd-mm-rr'),3000,NULL,20
0);
INSERT INTO EMP VALUES(7839,'KING','PRESIDENT',NULL,to_date('17-11-1981','dd-mm-yyyy'),5000,N
ULL,100);
INSERT INTO EMP VALUES(7844,'TURNER','SALESMAN',7698,to_date('8-9-1981','dd-mm-yyyy'),150
0,0,300);
INSERT INTO EMP VALUES(7876,'ADAMS','CLERK',7788,to_date('23-05-87', 'dd-mm-rr'),1100,NULL,20
0);
INSERT INTO EMP VALUES(7900,'JAMES','CLERK',7698,to_date('3-12-1981','dd-mm-yyyy'),950,NULL,30
0);
INSERT INTO EMP VALUES(7902,'FORD','ANALYST',7566,to_date('3-12-1981','dd-mm-yyyy'),3000,NUL
L,200);
INSERT INTO EMP VALUES(7934,'MILLER','CLERK',7782,to_date('23-1-1982','dd-mm-yyyy'),1300,NULL,1
00);

drop table IF EXISTS dept;
create table dept (DEPTNO NUMBER(4), DNAME VARCHAR2(14), LOC VARCHAR2(13));
insert into dept VALUES(100, 'ACCOUNTING', 'NEW YORK');
insert into dept VALUES(200, 'RESEARCH', 'DALLAS');
insert into dept VALUES(300, 'SALES', 'CHICAGO');
insert into dept VALUES(300, 'SALES', 'NEW YORK');
insert into dept VALUES(200, 'ACCOUNTING', 'NEW YORK');
insert into dept VALUES(200, 'RESEARCH', 'CHICAGO');
insert into dept VALUES(100, 'ACCOUNTING', 'BOSTON');
insert into dept VALUES(400, 'OPERATIONS', 'BOSTON');

-- TestPoint：存储过程里使用动态数组的相关函数extend/first/count/last/exists/prior/limit/next
create or replace procedure nested_pro
as
declare
    cursor emp_cursor is  select ename from emp;
    type emp_table is table of emp.ename%type;
    empTable emp_table:=emp_table();
    keyValue integer:=0;
begin
    empTable.extend;
    perform dbms_output.serveroutput('y');
    for tmp in emp_cursor
    loop
        keyValue := keyValue+1;
        --EXTEND:追加1个空元素到集合
        -- bug,已经有一个空间了，还是报数据越界
```

```
        empTable.extend;
        empTable[keyValue] := tmp.ename;
    end loop;

    --FIRST：返回在使用整数下标集合的第一个（最小的）索引号
    keyValue := empTable.FIRST;

    raise notice '%',keyValue;
    raise notice '%',empTable.last;
    raise notice '%',empTable.count;

    WHILE keyValue IS NOT null
    LOOP
        perform dbms_output.put_line('Emp name: ' || TO_CHAR(empTable[keyValue] || ' The emp index: '
|| keyValue));
        --返回索引keyValue的下一个索引号
        keyValue := empTable.NEXT(keyValue);
    END LOOP;

    --LAST：返回在使用整数下标集合的最后一个（最大的）索引号
    -- bug,first=0
    keyValue:= empTable.LAST;
    WHILE keyValue IS NOT null
    LOOP
        perform dbms_output.put_line('Emp name: ' || TO_CHAR(empTable[keyValue] || ' The emp index: '
|| keyValue));
        --返回索引keyValue的上一个索引号
        keyValue := empTable.prior(keyValue);
    END LOOP;

    -- 打印最大值
    -- bug
    -- perform dbms_output.put_line('limit values:'||empTable.limit);
    raise notice 'limit values %', empTable.limit;

    -- 判断元素是否存在
    raise notice 'exists:%', empTable.exists(2);
end;
/
call nested_pro();
```

记录类型

要创建记录变量，可以先定义RECORD类型，然后创建该类型的变量，也可以使用%ROWTYPE或%TYPE声明变量类型。
示例：record类型使用binary名

```
declare
type empinfo is record (empno number,ename varchar2(50));
v_one_row empinfo;
begin
select eno,ename into v_one_row from emp where rownum<=1; DBMS_OUTPUT.PUT_LINE(v_one_row.e
mpno||' '||v_one_row.ename);
end;
/
```

PL/SQL静态SQL
静态SQL概述

静态SQL在编译时是已知的。一般情况下静态 SQL 语句与相应的 SQL 语句具有相同的语法。

- SELECT（查询语句）
- 数据操纵语言（DML）语句：INSERT、UPDATE、DELETE 和 MERGE。
- 事务控制语言（TCL）语句：COMMIT、ROLLBACK、SAVEPOINT和SET TRANSACTION。

游标

显示游标

用户构造和管理的游标是显式游标

显示游标支持以下属性

%FOUND	只有在DML语句影响一行或者多行时，此属性才会返回true
%NOTFOUND	与%FOUND作用相反，如果DML语句没有影响任何行，则此属性返回true
%ROWCOUNT	返回DML语句影响的行数。如果DML语句没有影响任何行，则返回0
%ISOPEN	游标打开时返回true，游标关闭时返回false。隐式游标，此属性一直返回false

示例：显示游标属性

```
DECLARE
v_empno EMPLOYEES.EMPLOYEE_ID%TYPE;
v_sal EMPLOYEES.Salary%TYPE;
CURSOR c_cursor IS SELECT EMPLOYEE_ID, Salary FROM EMPLOYEES;
BEGIN
OPEN c_cursor;
LOOP
FETCH c_cursor INTO v_empno, v_sal;
EXIT WHEN c_cursor%NOTFOUND;
IF v_sal<=2000 THEN
UPDATE EMPLOYEES SET Salary=Salary+50 WHERE EMPLOYEE_ID=v_empno;
DBMS_OUTPUT.PUT_LINE('编码为'||v_empno||'工资已更新!');
END IF;
DBMS_OUTPUT.PUT_LINE('记录数:'|| c_cursor%ROWCOUNT);
END LOOP;
DBMS_OUTPUT.PUT_LINE('总记录数:'|| c_cursor%ROWCOUNT);
CLOSE c_cursor;
if c_cursor%ISOPEN then
DBMS_OUTPUT.PUT_LINE('关闭游标失败');
else
DBMS_OUTPUT.PUT_LINE('关闭游标成功');
end if;
END;
/
```

隐式游标

由数据库系统隐含创建游标是隐式游标
隐示游标支持以下属性

属性	值	SELECT	INSERT	UPDATE	DELETE
SQL%ISOPEN		FALSE	FALSE	FALSE	FALSE
SQL%FOUND	TRUE	有结果	成功	成功	失败
SQL%FOUND	FALSE	无结果	失败	失败	失败
SQL%NOTFOUND	TRUE	无结果	失败	失败	失败
SQL%NOTFOUND	FALSE	有结果	成功	成功	成功
SQL%ROWCOUNT		返回行数	插入行数	修改行数	删除行数

示例：隐式游标属性

```
DECLARE
V_deptno departments.department_id%TYPE :=1;
BEGIN
DELETE FROM departments WHERE department_id=v_deptno;
IF SQL%NOTFOUND THEN
DBMS_OUTPUT.PUT_LINE('没有发现 department_i='||v_deptno);
insert into departments values('a'||v_deptno, v_deptno*10, v_deptno);
DBMS_OUTPUT.PUT_LINE('插入数据行数='||SQL%ROWCOUNT);
elsif SQL%FOUND then
DBMS_OUTPUT.PUT_LINE('删除数据行数='||SQL%ROWCOUNT);
END IF;

if SQL%ISOPEN then
DBMS_OUTPUT.PUT_LINE('ISOPEN 为TRUE！！！有问题');
end if;
END;
/
```

使用for循环遍历游标

示例：

```
---创建表
drop table if exists employees;
create table employees(last_name varchar2(20), employee_id int, job_id varchar2(30), salary number(1
0, 3));
insert into employees values('a', 1, 'ACADFIMKSA_MANGR', 10005.45);
insert into employees values('b', 2, 'b_M_MANGR', 47956);
insert into employees values('c', 3, 'c_cursor', 1457.79);
insert into employees values('d', 4, 'deptcurtyp', 1458.256);192ye
insert into employees values('e', 5, 'SHT_CLERK', 1458.256);
insert into employees values('f', 6, 'SH_CLERK', 1458.256);
insert into employees values('g', 7, 'SA_REP', 1458.256);
insert into employees values('h', 8, 'AD_REP', 1458.256);
---定义游标，使用for循环遍历游标
DECLARE
   cursor c_cursor is select job_id||'--'||last_name||'--'||to_char(salary) as l from employees where rownu
m<=10 order by last_name;
   v_ename employees.last_name%type;
begin
   perform dbms_output.serveroutput('t');
   for v_ename in c_cursor
   loop
      perform DBMS_OUTPUT.PUT_LINE(c_cursor%ROWCOUNT||'---'||v_ename.l);
   end loop;
end;
/
```

游标的使用

定义一个游标

```
postgres=# begin;
BEGIN
postgres=#  DECLARE tbase_cur SCROLL CURSOR FOR SELECT * from tbase ORDER BY id;
DECLARE CURSOR
```

> 注意：
>
> 游标需要放在一个事务中使用。

提取下一行数据

```
postgres=# DECLARE tbase_cur SCROLL CURSOR FOR SELECT * from tbase ORDER BY id;
DECLARE CURSOR
postgres=# FETCH NEXT from tbase_cur ;
 id |  nickname
----+-------------
  1 | hello TDSQL PG
(1 row)
postgres=# FETCH NEXT from tbase_cur ;
 id | nickname
----+-----------
  2 | TDSQL PG好
(1 row)
```

提取前一行数据

```
postgres=# FETCH PRIOR from tbase_cur ;
 id |  nickname
----+-------------
  1 | hello TDSQL PG
```

提取最后一行

```
postgres=# FETCH LAST from tbase_cur ;
 id |  nickname
----+------------
  5 | TDSQL PG swap
(1 row)
```

提取第一行

```
postgres=# FETCH FIRST from tbase_cur ;
 id |  nickname
----+-------------
  1 | hello TDSQL PG
(1 row)
```

提取该查询的第x行

```
postgres=# FETCH ABSOLUTE 2 from tbase_cur ;
 id | nickname
----+----------
  2 | TDSQL PG好
(1 row)
postgres=# FETCH ABSOLUTE -1 from tbase_cur ;
 id |  nickname
----+------------
  5 | TDSQL PG swap
(1 row)
postgres=# FETCH ABSOLUTE -2 from tbase_cur ;
 id |   nickname
----+--------------
  4 | TDSQL PG default
(1 row)
```

X为负数时则从尾部向上提。

提取当前位置后的第x行

```
postgres=#  FETCH ABSOLUTE 1 from tbase_cur ;
 id |  nickname
----+-------------
  1 | hello TDSQL PG
(1 row)
postgres=# FETCH RELATIVE 2 from tbase_cur ;
 id | nickname
----+-----------
  3 | TDSQL PG好
(1 row)
postgres=# FETCH RELATIVE 2 from tbase_cur ;
 id |  nickname
----+------------
  5 | TDSQL PG swap
(1 row)
```

每提取一次数据，游标的位置都是会前行。

向前提取x行数据

```
postgres=# FETCH FORWARD 2 from tbase_cur ;
 id |  nickname
----+-------------
  1 | hello TDSQL PG
  2 | TDSQL PG好
(2 rows)
postgres=# FETCH FORWARD 2 from tbase_cur ;
 id |   nickname
----+---------------
  3 | TDSQL PG好
  4 | TDSQL PG default
(2 rows)
```

向前提取剩下的所有数据

```
postgres=# FETCH FORWARD 2 from tbase_cur ;
 id |  nickname
----+-------------
  1 | hello TDSQL PG
  2 | TDSQL PG好
(2 rows)
postgres=# FETCH FORWARD all from tbase_cur ;
 id |   nickname
----+---------------
  3 | TDSQL PG好
  4 | TDSQL PG default
  5 | TDSQL PG swap
(3 rows)
```

向后提取x行数据

```
postgres=# FETCH BACKWARD 2 from tbase_cur ;
 id |   nickname
----+---------------
  5 | TDSQL PG swap
  4 | TDSQL PG default
(2 rows)
```

向后提取剩下的所有数据

```
postgres=# FETCH BACKWARD all from tbase_cur ;
 id |  nickname
----+-------------
  3 | TDSQL PG好
  2 | TDSQL PG好
  1 | hello TDSQL PG
(3 rows)
```

事务处理与控制

TDSQL PG支持事务处理，允许多个用户同时处理数据库，并确保每个用户看到一致的数据版本，并确保以正确的顺序应用所有更改。
事务是Oracle数据库作为一个单元处理的一个或多个SQL语句的序列：要么执行所有语句，要么都不执行。
详细见章节 关于事务

自治事务

自治事务是由另一个事务（主事务）启动的独立事务。
自治事务执行SQL操作并提交或回滚，而无需提交或回滚主事务。
支持匿名块或者存储过程中使用子事务。
示例：

```
create table t3(f1 int);

do
$$
begin
    insert into t3 values(1);
    insert into t3 values(2);
    commit;
    insert into t3 values(3);
    rollback;
end;
$$ ;
create or replace procedure demo11(a_varchar in varchar)
as
$$
begin
   insert into t3 values(1);
   insert into t3 values(2);
   commit;
   insert into t3 values(3);
   rollback;
end;
$$
LANGUAGE plpgsql ;
call demo11(null);
select * from t3;
```

PL/SQL动态SQL

动态SQL是一种在PL/SQL运行时生成和执行SQL语句的编程方法，为PL/SQL编程提供了很大的灵活性。主要应用于一些静态SQL无法支持的操作场景，例如DDL，又或者是一些需要传入参数的SQL语句。
动态 SQL 使用 EXECUTE IMMEDIATE 语句处理大多数动态 SQL 语句。如果动态 SQL 语句是返回多行的 SELECT 语句，将 EXECUTE IMMEDIATE 语句与 BULK COLLECT INTO 子句一起使用。动态SQL传入参数使用USING子句。
示例：
1、使用USING字句给动态SQL传入参数

```
CREATE TABLE EMP
(EMPNO NUMBER(4) CONSTRAINT PK_EMP PRIMARY KEY,
ENAME VARCHAR2(10),
JOB VARCHAR2(9),
MGR NUMBER(4),
HIREDATE DATE,
SAL NUMBER(7,2),
COMM NUMBER(7,2),
DEPTNO NUMBER(2));

INSERT INTO EMP VALUES
(7369,'SMITH','CLERK',7902,to_date('17-12-1980','dd-mm-yyyy'),800,NULL,20);
INSERT INTO EMP VALUES
(7499,'ALLEN','SALESMAN',7698,to_date('20-2-1981','dd-mm-yyyy'),1600,300,30);
INSERT INTO EMP VALUES
(7521,'WARD','SALESMAN',7698,to_date('22-2-1981','dd-mm-yyyy'),1250,500,30);
INSERT INTO EMP VALUES
(7566,'JONES','MANAGER',7839,to_date('2-4-1981','dd-mm-yyyy'),2975,NULL,20);

create or replace procedure p_sql is
   v_sql varchar(500);
   vret number;
   vin number:=1111;
begin
   perform dbms_output.serveroutput('t');
   v_sql:='select sal from emp where empno=:id';
   execute immediate v_sql into vret using 7521;
   perform dbms_output.put_line(vret);
  -- raise notice 'vret=%',vret;
end;
/
call p_sql();
```

2、动态执行CREATE TABLE

```
create table emp1 as select * from emp;
create table dept_tmp (deptno number(2),dname varchar2(14),loc varchar2(13));
create or replace package pkg1 is
procedure raise_salary(v1 number,v2 number);
end;
/
create or replace package body pkg1 is
procedure raise_salary(v1 number,v2 number) is
begin
  update emp1 set sal=sal+v2 where empno=v1;
end;
end;
/
DECLARE
  sql_stmt    VARCHAR2(200);
  plsql_block VARCHAR2(500);
  emp_id      NUMBER(4) := 7566;
  salary      NUMBER(7,2);
  dept_id     NUMBER(2) := 50;
  dept_name   VARCHAR2(14) := 'PERSONNEL';
  location    VARCHAR2(13) := 'DALLAS';
  emp_rec     emp%ROWTYPE;
BEGIN
  EXECUTE IMMEDIATE 'CREATE TABLE using_t (id NUMBER, amt NUMBER)';
  sql_stmt := 'INSERT INTO dept_tmp VALUES (:1, :2, :3)';
  EXECUTE IMMEDIATE sql_stmt USING dept_id, dept_name, location;
  sql_stmt := 'SELECT * FROM emp WHERE empno = :id';
  EXECUTE IMMEDIATE sql_stmt INTO emp_rec USING emp_id;
 -- plsql_block := 'BEGIN pkg1.raise_salary(:id, :amt); END;';
  plsql_block:='call pkg1.raise_salary(:id, :amt)';
  EXECUTE IMMEDIATE plsql_block USING 7788, 500;
 -- execute immediate v_sql using in vin, in vret;
--    sql_stmt := 'UPDATE emp SET sal = 2000 WHERE empno = :1
--       RETURNING sal INTO :2';
--    EXECUTE IMMEDIATE sql_stmt USING emp_id RETURNING INTO salary;
  EXECUTE IMMEDIATE 'DELETE FROM dept_tmp WHERE deptno = :num'
     USING dept_id;
  EXECUTE IMMEDIATE 'ALTER table dept_tmp add  c1 number';
END;
/
```

PL/SQL子程序
过程

使用CREATEPROCEDURE 创建过程

示例：支持using out用法

```
set enable_oracle_compatible to on;

begin
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('t');
end;
/

create or replace procedure using_out_p1(a_ret out number) is
begin
    a_ret:=10;
end;
/

create or replace procedure using_out_p2(a_ret out number, b_ret out varchar) is
begin
    a_ret:=10;
    b_ret:='ok';
end;
/

create or replace procedure using_out_p3(a_ret out number, b_ret out varchar, c_out in out numeric) is
begin
    a_ret:=10;
    b_ret:='ok';
    c_out := c_out+10;
end;
/

-- TestPoint : 一个out参数
create or replace procedure p_1 is
    vsql varchar(500);
    v_ret number;
begin
    vsql:='call using_out_p1(:1)';
    execute immediate vsql using out v_ret;
    perform dbms_output.put_line(v_ret);
end;
```

```
/

call p_1();
```

函数

使用CREATE[OR REPLACE] FUNCTION创建函数

示例：

有无参数的函数与有参数的函数，从block调用函数，从函数调用函数

```
create or replace function get_log_id return number is
  Result number;
begin
  select LOG_ID.nextval into Result from dual;
  return(Result);
end get_log_id;
/

create or replace function get_log_id(v1 number) return number is
  Result number;
begin
  Result:=v1;
  return(Result);
end get_log_id;
/

declare
  v number;
begin
  v:= get_log_id;
  raise notice '%',v;
exception
  when others then
    raise;
end;
/

declare
  v number;
begin
  v:= get_log_id + get_log_id(100);
  raise notice '%',v;
exception
  when others then
    raise;
end;
/

drop function if exists f1;
```

```
create or replace function f1 return number is
  v number;
begin
  v:= get_log_id + get_log_id(100);
  raise notice '%',v;
  return v;
exception
  when others then
    return null;
end;
/

select f1 from dual;
```

⚠️TDSQL支持包内子程序重载

PL/SQL触发器

触发器是一个定义好的的PL/ SQL单元，它存储在数据库中并运行以响应数据库中发生的事件。
您可以指定事件，触发器在事件之前或之后触发，以及触发器是针对每个事件还是针对受事件影响的每一行运行。例如，您可以创建一个每次INSERT语句影响EMPLOYEES表时运行的触发器。
详细见章节触发器

PL/SQL错误处理

PL / SQL使检测和处理错误变得容易。
发生错误时，PL/ SQL会引发异常。正常执行停止并控制传输到PL/ SQL块的异常处理部分。通过以下几个示例，说明错误处理类型。

RAISE NOTICE

```
postgres=# CREATE OR REPLACE FUNCTION f28() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_int INTEGER := 1;
postgres$# BEGIN
postgres$#     RAISE NOTICE 'v_int = %, 随机数 = %',v_int,random();
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f28();
NOTICE:  v_int = 1, 随机数 = 0.236714988015592
```

结果为

```
 f28
-----

(1 row)
```

使用raise notice 向终端输出一个消息,也有可能写到日志中(需要调整日志的保存级别)

RAISE EXCEPTION

```
postgres=# CREATE OR REPLACE FUNCTION f28() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_int INTEGER := 1;
postgres$# BEGIN
postgres$#     RAISE EXCEPTION '程序EXCEPTION ';
postgres$#     --下面的语句不会再执行
```

```
postgres$#     RAISE NOTICE 'v_int = %, 随机数 = %',v_int,random();
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f28();
ERROR:  程序EXCEPTION
```

如果在事务中执行这个函数,则事务会中止(abort)

RAISE EXCEPTION 自定义ERRCODE

```
postgres=# CREATE OR REPLACE FUNCTION f28() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_int INTEGER := 1;
postgres$# BEGIN
postgres$#     RAISE EXCEPTION ' 程序EXCEPTION ' USING ERRCODE = '23505';
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f28();
ERROR:  程序EXCEPTION
```

日志中会记录这个ERRCODE

```
2017-10-03 18:40:16.710 CST,"pgxz","postgres",15072,"[local]",59d33b65.3ae0,225,"idle",2017-10-03 1
5:25:25 CST,4/367159,0,LOG,00000,"statement: SELECT f28();",,,,,,,,,"psql"
2017-10-03 18:40:16.710 CST,"pgxz","postgres",15072,"[local]",59d33b65.3ae0,226,"SELECT",2017-10-0
3 15:25:25 CST,4/367159,0,ERROR,23505," 程序EXCEPTION ",,,,,,,,,"psql"
```

参数详细介绍
参数模式
IN模式

IN模式指的是执行函数时需要输入参数值，如下所示：

```
postgres=# CREATE OR REPLACE FUNCTION f1(IN a_xm text) RETURNS TEXT AS
postgres-# $$
postgres$# BEGIN
postgres$#     RETURN a_xm;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT f1('TDSQL PG');
  f1
------
 TDSQL PG
(1 行记录)
postgres=# CREATE OR REPLACE FUNCTION f1(a_xm text) RETURNS TEXT AS
postgres-# $$
postgres$# BEGIN
postgres$#     RETURN a_xm;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f1('TDSQL PG');
  f1
------
 TDSQL PG
(1 行记录)
#上面两种方式定义的参数效果是一样的
```

OUT模式

OUT模式参数是指定了函数执行时返回的字段名及类型。

```
postgres=# CREATE OR REPLACE FUNCTION f1(OUT a_xm TEXT) RETURNS TEXT AS
postgres-# $$
postgres$# BEGIN
postgres$#      a_xm:='TDSQL PG';
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=# SELECT * FROM f1();
 a_xm
------
 TDSQL PG
(1 行记录)
#采用OUT模式参数不能用RETURN返回，而是要对返回的OUT参数直接付值。返回值类型与参数的数据类型必需一致。参数名就是返回的字段名
```

INOUT模式

INOUT模式是指参数即传入，同时又指定了返回值的字段名和类型。

```
postgres=# CREATE OR REPLACE FUNCTION f1(INOUT a_xm text) RETURNS TEXT AS
postgres-# $$
postgres$# BEGIN
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f1('TDSQL PG');
 a_xm
------
 TDSQL PG
(1 行记录)
#值得注意的是，上面的函数跟下面的函数是相同的，即重新定义会覆盖掉
postgres=# CREATE OR REPLACE FUNCTION f1(IN a_xm text) RETURNS TEXT AS
postgres-# $$
postgres$# BEGIN
postgres$#     RETURN 'TDSQL PG';
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=# SELECT * FROM f1('TDSQL PG');
  f1
------
 TDSQL PG
(1 行记录)
```

VARIADIC模式

VARIADICS模式是参数个数可变模式，系统用一个数组对传入的参数进行处理，VARIADIC参数必需是所有最后一个声明，如下所示：

```
postgres=# CREATE OR REPLACE FUNCTION f1(VARIADIC a_int integer[])  RETURNS void AS
postgres-# $$
postgres$# BEGIN
postgres$#     RAISE NOTICE 'a_int = %',a_int;
postgres$#     RAISE NOTICE 'a_int[1] = %',a_int[1];
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT f1(1);
NOTICE:  a_int = {1}
NOTICE:  a_int[1] = 1
 f1
----
(1 行记录)
postgres=# SELECT f1(1,2);
NOTICE:  a_int = {1,2}
NOTICE:  a_int[1] = 1
 f1
----
(1 行记录)
postgres=# CREATE OR REPLACE FUNCTION f1(a_xm TEXT,VARIADIC a_int integer[])  RETURNS void AS
postgres-# $$
postgres$# BEGIN
postgres$#     RAISE NOTICE 'a_int = %',a_int;
postgres$#     RAISE NOTICE 'a_int[1] = %',a_int[1];
postgres$#     RAISE NOTICE 'a_xm = %',a_xm;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT f1('TDSQL PG',1,2);
NOTICE:  a_int = {1,2}
NOTICE:  a_int[1] = 1
NOTICE:  a_xm = TDSQL PG
 f1
----
(1 行记录)
```

参数引用
无命名参数

```
postgres=# CREATE OR REPLACE FUNCTION f2(text) RETURNS TEXT AS
postgres-# $$
postgres$# BEGIN
postgres$#     RETURN $1;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f2('TDSQL PG');
  f2
------
 TDSQL PG
(1 行记录)
```

给标识符指定别名

```
postgres=# CREATE OR REPLACE FUNCTION f2(text) RETURNS TEXT AS
postgres-# $$
postgres$# DECLARE
postgres$#     a_xm ALIAS FOR $1; --a_xm是$1的别名
postgres$# BEGIN
postgres$#     RETURN a_xm;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f2('TDSQL PG');
  f2
------
 TDSQL PG
(1 行记录)
```

命名参数

```
postgres=# CREATE OR REPLACE FUNCTION f2(a_xm text) RETURNS TEXT AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_xm ALIAS FOR $1;
postgres$# BEGIN
postgres$#     RAISE NOTICE 'a_xm = % ; v_xm = % ; $1 = %',a_xm,v_xm,$1;
postgres$#     RETURN $1;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f2('TDSQL PG');
NOTICE:  a_xm = TDSQL PG ; v_xm = TDSQL PG ; $1 = TDSQL PG
  f2
------
 TDSQL PG
(1 行记录)
```

参数数据类型
基本类型

```
postgres=# CREATE OR REPLACE FUNCTION f3 (a_int integer,a_str text) RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     RAISE NOTICE 'a_int = % ; a_str = %',a_int,a_str;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=# SELECT * FROM f3(1,'TDSQL PG');
NOTICE:  a_int = 1 ; a_str = TDSQL PG
 f3
----
(1 行记录)
postgres=# CREATE OR REPLACE FUNCTION f3 (a_int integer[],a_str text[]) RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     RAISE NOTICE 'a_int = % ; a_str = %',a_int,a_str;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=# SELECT f3(ARRAY[1,2,3],ARRAY['TDSQL PG','pgxz']);
NOTICE:  a_int = {1,2,3} ; a_str = {TDSQL PG,pgxz}
 f3
----
(1 行记录)
```

复合类型

```
postgres=# CREATE TYPE t_per AS
postgres-# (
postgres(#     id integer,
postgres(#     mc text
postgres(# );
ERROR:  type "t_per" already exists
postgres=# CREATE OR REPLACE FUNCTION f3 (a_row public.t_per) RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     RAISE NOTICE 'id = % ; mc = %',a_row.id,a_row.mc;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT f3(ROW(1,'TDSQL PG')::public.t_per);
NOTICE:  id = 1 ; mc = TDSQL PG
 f3
----
(1 行记录)
postgres=# CREATE OR REPLACE FUNCTION f3 (a_rec public.t_per[]) RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     RAISE NOTICE 'a_rec = %',a_rec;
postgres$#     RAISE NOTICE 'a_rec[1].id = %',a_rec[1].id;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT f3(ARRAY[ROW(1,'TDSQL PG'),ROW(1,'pgxz')]::public.t_per[]);
NOTICE:  a_rec = {"(1,TDSQL PG)","(1,pgxz)"}
NOTICE:  a_rec[1].id = 1
 f3
----
(1 行记录)
```

行类型

```
postgres=# \d t
   资料表 "public.t"
 栏位 |  型别   | 修饰词
------+---------+--------
 id   | integer |
 mc   | text    |
postgres=# CREATE OR REPLACE FUNCTION f3 (a_row public.t) RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     RAISE NOTICE 'id = % ; mc = %',a_row.id,a_row.mc;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT f3(ROW(1,'TDSQL PG'));
NOTICE:  id = 1 ; mc = TDSQL PG
 f3
----
(1 行记录)
postgres=# SELECT f3(t.*) FROM t LIMIT 1;
NOTICE:  id = 1 ; mc = TDSQL PG
 f3
----
(1 行记录)
postgres=# CREATE OR REPLACE FUNCTION f3 (a_rec public.t[]) RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     RAISE NOTICE 'a_rec = %',a_rec;
postgres$#     RAISE NOTICE 'a_rec[1].id = %',a_rec[1].id;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT f3(array[row(1,'TDSQL PG'),row(1,'pgxz')]::public.t[]);
NOTICE:  a_rec = {"(1,TDSQL PG)","(1,pgxz)"}
NOTICE:  a_rec[1].id = 1
 f3
----
(1 行记录)
postgres=# SELECT f3(array[t.*,t.*]::public.t[]) FROM t LIMIT 2;
NOTICE:  a_rec = {"(1,TDSQL PG)","(1,TDSQL PG)"}
```

```
NOTICE:  a_rec[1].id = 1
NOTICE:  a_rec = {"(2,pgxz)","(2,pgxz)"}
NOTICE:  a_rec[1].id = 2
 f3
----
(2 行记录)
```

域类型

```
postgres=# CREATE DOMAIN xb AS TEXT CHECK
postgres-# (
postgres(#     VALUE = '男'
postgres(#     OR VALUE ='女'
postgres(#     OR VALUE = ''
postgres(# );
CREATE DOMAIN
postgres=#
postgres=# CREATE OR REPLACE FUNCTION f4 (a_xb public.xb) RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     RAISE NOTICE 'a_xb = %',a_xb;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f4('男');
NOTICE:  a_xb = 男
 f4
----
(1 行记录)
postgres=# SELECT * FROM f4('她');
ERROR:  value for domain xb violates check constraint "xb_check"
postgres=#
#域类型输入参数值时会检查是否违反规则
```

游标类型

```
postgres=# CREATE OR REPLACE FUNCTION f5 (a_ref refcursor) RETURNS void AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_rec record;
postgres$# BEGIN
postgres$#     OPEN a_ref FOR SELECT * FROM t LIMIT 1;
postgres$#     FETCH a_ref INTO v_rec;
postgres$#     RAISE NOTICE 'v_rec = % ',v_rec;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f5('a');
NOTICE:  v_rec = (1,TDSQL PG)
 f5
----
(1 行记录)
postgres=# CREATE OR REPLACE FUNCTION f6 (a_ref refcursor) RETURNS refcursor AS
postgres-# $$
postgres$# BEGIN
postgres$#     OPEN a_ref FOR SELECT * FROM t LIMIT 1;
postgres$#     RETURN a_ref;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
#注意这里需要开启一个事务
postgres=# BEGIN;
BEGIN
postgres=# SELECT * FROM f6('a');
 f6
----
 a
(1 行记录)
postgres=# FETCH ALL FROM a;
 id |  mc
----+------
  1 | TDSQL PG
(1 行记录)
```

多态类型

```
postgres=# CREATE OR REPLACE FUNCTION f_any(a_arg anyelement) RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     RAISE NOTICE '%',a_arg;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT f_any(1::integer);
NOTICE:  1
 f_any
-------
(1 行记录)
postgres=# SELECT f_any('TDSQL PG'::TEXT);
NOTICE:  TDSQL PG
 f_any
-------
(1 行记录)
postgres=# SELECT f_any(ROW(1,'TDSQL PG')::public.t_rec);
NOTICE:  (1,TDSQL PG)
 f_any
-------
(1 行记录)
postgres=# SELECT f_any(ARRAY[1,2]::INTEGER[]);
NOTICE:  {1,2}
 f_any
-------
(1 行记录)
postgres=# SELECT f_any(ARRAY[[1,2],[3,4],[5,6]]::INTEGER[][]);
NOTICE:  {{1,2},{3,4},{5,6}}
 f_any
-------
(1 行记录)
#注意多态类型参数函数调用时最好直接声明参数类型，否则有可能出错
postgres=# CREATE OR REPLACE FUNCTION f_any_array(a_arg anyarray) RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     RAISE NOTICE '%',a_arg;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
```

```
postgres=#
postgres=# SELECT f_any_array(ARRAY['TDSQL PG','pgxz']::TEXT[]);
NOTICE:  {TDSQL PG,pgxz}
 f_any_array
-------------
(1 行记录)
postgres=# SELECT f_any_array(ARRAY[ARRAY['TDSQL PG','pgxz'],ARRAY['TDSQL PG','xx']]::TEXT[][]);
NOTICE:  {{TDSQL PG,pgxz},{TDSQL PG,xx}}
 f_any_array
-------------
(1 行记录)
```

> 注意：
>
> Anyelement参数如果写成数组，其意义就跟anyarray参数一致，所以 f_any(a_arg anyelement)与f_any(a_arg anyarray)在调用f_any(ARRAY[1,2])时就会出现函数不是唯一化的错误(ERROR: function f_any(...) is not unique)提示。

参数默认值

PL/pgsql扩展语言函数支持给参数设置默认值。

```
postgres=# CREATE OR REPLACE FUNCTION f7 (a_int INTEGER DEFAULT 1) RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     RAISE NOTICE 'a_int = %',a_int;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f7();
NOTICE:  a_int = 1
 f7
----
(1 行记录)
#备注：如果原来存在一个f7()这样的函数，则上面的执行就会出错，因为系统无法清楚到你到底要执行那个函数，如下所示
postgres=# CREATE OR REPLACE FUNCTION f7()  RETURNS void AS
postgres-# $$
postgres$# BEGIN
postgres$#     RAISE NOTICE '无参数';
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql ;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f7();
ERROR:  function f7() is not unique
第1行SELECT * FROM f7();
                  ^
#提示:  Could not choose a best candidate function. You might need to add explicit type casts.
postgres=#
```

出错提示，f7()函数不是唯一的，这是使用上一个需要特别注意的地方。

返回值详细介绍

返回值介绍

返回值可以是一个简单数据类型、复合类型、RECORD、已经存在的表行类型、表字段类型、游标、另外还可以返回一个记录集、如果不需要返回值，则可以用RETURN void。返回值的字段名及类型可以在参数在用OUT,INOUT模式中声明。

返回值类型介绍

没有返回值

```
postgres=# CREATE OR REPLACE FUNCTION f8()  RETURNS void AS
postgres-# $$
postgres$# BEGIN
postgres$#     RAISE NOTICE '不用返回值，函数体可以有或没有return语句';
postgres$#     RETURN ;            #这一句可以有，也可以没有
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f8();
NOTICE:  不用返回值，函数体可以有或没有return语句
 f8
----
(1 行记录)
```

返回简单类型

```
postgres=# CREATE OR REPLACE FUNCTION f9()  RETURNS TEXT AS
postgres-# $$
postgres$# BEGIN
postgres$#     RETURN 'TDSQL PG';
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f9() t(a_xm);
 a_xm
------
 TDSQL PG
(1 行记录)
postgres=#
postgres=# CREATE OR REPLACE FUNCTION f9(OUT a_xm TEXT)  RETURNS TEXT AS
postgres-# $$
postgres$# BEGIN
postgres$#     a_xm:='TDSQL PG';
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f9();
 a_xm
------
 TDSQL PG
(1 行记录)
#上面两个函数其实就是同一个函数，建立时如果不加OR REPLACE则会提示已经存在
postgres=# CREATE OR REPLACE FUNCTION f10()  RETURNS TEXT[] AS
postgres-# $$
postgres$# BEGIN
postgres$#     RETURN ARRAY['TDSQL PG','pgxz'];
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f10();
      f10
---------------
 {TDSQL PG,pgxz}
```

(1 行记录)

返回一个复合类型

```
postgres=# CREATE TYPE t_rec AS
postgres-# (
postgres(#     id integer,
postgres(#     mc text
postgres(# );
CREATE TYPE
postgres=#
postgres=# CREATE OR REPLACE FUNCTION f11()  RETURNS t_rec AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_rec public.t_rec;
postgres$# BEGIN
postgres$#     v_rec.id:=1;
postgres$#     v_rec.mc='TDSQL PG';
postgres$#     RETURN v_rec;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f11();
 id |  mc
----+------
  1 | TDSQL PG
(1 行记录)
postgres=# CREATE OR REPLACE FUNCTION f12()  RETURNS t_rec[] AS
postgres-# $$
postgres$# BEGIN
postgres$#     RETURN ARRAY[ROW(1,'TDSQL PG'),ROW(1,'pgxz')]::t_rec[];
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f12();
           f12
---------------------------
 {"(1,TDSQL PG)","(1,pgxz)"}
(1 行记录)
```

返回行类型

```
postgres=# \d t
   资料表 "public.t"
 栏位 |  型别   | 修饰词
------+---------+--------
 id   | integer |
 mc   | text    |
postgres=# CREATE OR REPLACE FUNCTION f13()  RETURNS public.t AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_rec public.t%ROWTYPE;
postgres$# BEGIN
postgres$#     SELECT * INTO v_rec FROM public.t LIMIT 1;
postgres$#     RETURN v_rec;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f13();
 id |  mc
----+------
  1 | TDSQL PG
(1 行记录)
postgres=# CREATE OR REPLACE FUNCTION f14() RETURNS public.t[] AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_rec public.t[];
postgres$# BEGIN
postgres$#     SELECT ARRAY[ROW(t.*),ROW(t.*)]::public.t[] INTO v_rec FROM public.t LIMIT 1;
postgres$#     RETURN v_rec;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f14();
          f14
------------------------
 {"(1,TDSQL PG)","(1,TDSQL PG)"}
(1 行记录)
```

返回TABLE类型

```
postgres=# CREATE OR REPLACE FUNCTION f15() RETURNS RECORD AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_rec RECORD;
postgres$# BEGIN
postgres$#     v_rec:=ROW(1::integer,'TBase'::text,'pgxz'::text);
postgres$#     RETURN v_rec;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f15();
      f15
-----------------
 (1,TDSQL PG,pgxz)
(1 行记录)
postgres=# SELECT * FROM f15() t(id integer,xm text,xl text);
 id |  xm  |   xl
----+------+--------
  1 | TDSQL PG | pgxz
(1 行记录)
```

返回RECORD类型

```
postgres=# CREATE OR REPLACE FUNCTION f15() RETURNS RECORD AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_rec RECORD;
postgres$# BEGIN
postgres$#     v_rec:=ROW(1::integer,'TBase'::text,'pgxz'::text);
postgres$#     RETURN v_rec;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f15();
      f15
-----------------
 (1,TDSQL PG,pgxz)
(1 行记录)
postgres=# SELECT * FROM f15() t(id integer,xm text,xl text);
 id |  xm  |   xl
----+------+--------
  1 | TDSQL PG | pgxz
(1 行记录)
```

返回一个游标

```
postgres=# CREATE OR REPLACE FUNCTION f16() RETURNS refcursor AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_ref refcursor;
postgres$# BEGIN
postgres$#     OPEN v_ref FOR SELECT * FROM public.t;
postgres$#     RETURN v_ref;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#
postgres=# BEGIN;
BEGIN
postgres=#
postgres=# SELECT * FROM f16();
       f15
--------------------
 <unnamed portal 1>
(1 行记录)
postgres=# FETCH ALL FROM "<unnamed portal 1>";
 id |   mc
----+--------
  1 | TDSQL PG
  2 | pgxz
(2 行记录)
postgres=# END;
postgres=# CREATE OR REPLACE FUNCTION f16(a_ref refcursor) RETURNS refcursor AS
postgres-# $$
postgres$# BEGIN
postgres$#     OPEN a_ref FOR SELECT * FROM public.t;
postgres$#     RETURN a_ref;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#
postgres=# BEGIN;
BEGIN
postgres=# SELECT * FROM f16('a');
 f15
-----
 a
```

```
(1 行记录)
postgres=# FETCH ALL FROM a;
 id |   mc
----+--------
  1 | TDSQL PG
  2 | pgxz
(2 行记录)
postgres=# END;
COMMIT
```

返回记录集

```
postgres=# CREATE OR REPLACE FUNCTION f17() RETURNS SETOF TEXT AS
postgres-# $$
postgres$# BEGIN
postgres$#     RETURN NEXT 'TDSQLPG::text;
postgres$#     RETURN NEXT 'pgxz'::text;
postgres$#     RETURN ;                    #最后的RETURN可以加，也可以不加上去
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f17();
  f17
--------
 TDSQLPG
 pgxz
(2 行记录)
postgres=#
postgres=#
postgres=# CREATE OR REPLACE FUNCTION f18() RETURNS SETOF public.t AS
postgres-# $$
postgres$# DECLARE
postgres$#                          #使用行类型返回
postgres$#     v_rec public.t%ROWTYPE;
postgres$# BEGIN
postgres$#     FOR v_rec IN SELECT * FROM t ORDER BY id LOOP
postgres$#         RETURN NEXT v_rec;
postgres$#     END LOOP;
postgres$#     RETURN ;                    #最后的RETURN可以加，也可以不加上去
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f18();
 id |   mc
----+--------
  1 | TDSQLPG
  2 | pgxz
(2 行记录)
postgres=# \d t1
      资料表 "public.t1"
 栏位 |     型别      | 修饰词
```

```
------+--------------+--------
 id   | integer      | 非空
 yhm  | text         | 非空
 nc   | text         | 非空
 mm   | character(32) | 非空
索引：
    "t1_pkey" PRIMARY KEY, btree (id)
    "t1_yhm_key" UNIQUE CONSTRAINT, btree (yhm)
postgres=# CREATE OR REPLACE FUNCTION f19() RETURNS SETOF public.t_rec AS
postgres-# $$
postgres$# DECLARE
postgres$#     #使用已经定义的结构类型返回
postgres$#     v_rec public.t_rec;
postgres$# BEGIN
postgres$#     FOR v_rec IN SELECT id,yhm FROM t1 ORDER BY id LOOP
postgres$#         RETURN NEXT v_rec;
postgres$#     END LOOP;
postgres$#     RETURN ;                    #最后的RETURN可以加，也可以不加上去
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f19();
 id |   mc
----+--------
  1 | TDSQLPG
  2 | pgxc
  3 | pgxz
(3 行记录)
postgres=# CREATE OR REPLACE FUNCTION f20(a_int integer) RETURNS SETOF record AS
postgres-# $$
postgres$# DECLARE
postgres$#                          #a_int定义返回的字段数，实现动态列返回
postgres$#     v_rec record;
postgres$#     v_sql text;
postgres$# BEGIN
postgres$#     IF a_int = 2 THEN
postgres$#         v_sql:='SELECT id,yhm FROM t1 ORDER BY id ';
postgres$#     ELSE
postgres$#         v_sql:='SELECT id,yhm,nc FROM t1 ORDER BY id';
postgres$#     END IF;
postgres$#     FOR v_rec IN EXECUTE v_sql LOOP
postgres$#         RETURN NEXT v_rec;
postgres$#     END LOOP;
postgres$#     RETURN ;                    #最后的RETURN可以加，也可以不加上去
postgres$# END;
```

```
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=# SELECT * FROM f20(2) t(id integer,yhm text);
 id |  yhm
----+--------
  1 | TDSQLPG
  2 | pgxc
  3 | pgxz
(3 行记录)
postgres=# SELECT * FROM f20(3) t(id integer,yhm text,nc text);
 id |  yhm   |   nc
----+--------+--------
  1 | TDSQLPG   | TDSQLPG
  2 | pgxc  | pgxc
  3 | pgxz | pgxz
(3 行记录)
postgres=# CREATE OR REPLACE FUNCTION f21(OUT a_id integer,OUT a_yhm TEXT) RETURNS SETOF r
ecord AS
postgres-# $$
postgres$# DECLARE
postgres$#                                  #使用out返回
postgres$#     v_rec record;
postgres$# BEGIN
postgres$#     FOR v_rec IN SELECT id,yhm FROM t1 LOOP
postgres$#         a_id:=v_rec.id;
postgres$#         a_yhm:=v_rec.yhm;
postgres$#         RETURN NEXT;
postgres$#     END LOOP;
postgres$#     RETURN ;                     #最后的RETURN可以加，也可以不加上去
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f21();
 a_id | a_yhm
------+--------
    1 | TDSQLPG
    2 | pgxc
    3 | pgxz
(3 行记录)
postgres=# CREATE OR REPLACE FUNCTION f22() RETURNS SETOF refcursor AS
postgres-# $$
postgres$# DECLARE
postgres$#     #返回游标集
postgres$#     v_ref1 REFCURSOR;
```

```
postgres$#     v_ref2 REFCURSOR;
postgres$# BEGIN
postgres$#     OPEN v_ref1 FOR SELECT * FROM t;
postgres$#     OPEN v_ref2 FOR SELECT * FROM t1;
postgres$#     RETURN NEXT v_ref1;
postgres$#     RETURN NEXT v_ref2;
postgres$#     RETURN ;                    #最后的RETURN可以加，也可以不加上去
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# BEGIN;
BEGIN
postgres=# SELECT * FROM f22();
        f22
--------------------
 <unnamed portal 13>
 <unnamed portal 14>
(2 行记录)
postgres=# FETCH ALL FROM "<unnamed portal 13>";
 id |   mc
----+--------
  1 | TDSQLPG
  2 | pgxz
(2 行记录)
postgres=# FETCH ALL FROM "<unnamed portal 14>";
 id |  yhm   |   nc   |              mm
----+--------+--------+----------------------------------
  1 | TDSQLPG   | TDSQLPG   | 202cb962ac59075b964b07152d234b70
  2 | pgxc  | pgxc | 202cb962ac59075b964b07152d234b70
  3 | pgxz | pgxz | 202cb962ac59075b964b07152d234b70
(3 行记录)
postgres=# COMMIT;
COMMIT
postgres=# CREATE OR REPLACE FUNCTION f22(a_ref1 refcursor,a_ref2 refcursor) RETURNS SETOF refc
ursor AS
postgres-# $$
postgres$# BEGIN
postgres$#     #指定游标名称
postgres$#     OPEN a_ref1 FOR SELECT * FROM t;
postgres$#     OPEN a_ref2 FOR SELECT * FROM t1;
postgres$#     RETURN NEXT a_ref1;
postgres$#     RETURN NEXT a_ref2;
postgres$#     RETURN ;                    #最后的RETURN可以加，也可以不加上去
postgres$# END;
postgres$# $$
```

```
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# BEGIN;
BEGIN
postgres=#
postgres=# SELECT * FROM f22('a','b');
 f22
-----
 a
 b
(2 行记录)
postgres=# FETCH ALL FROM "a";
 id |   mc
----+--------
  1 | TDSQLPG
  2 | pgxz
(2 行记录)
postgres=# FETCH ALL FROM "b";
 id |  yhm   |   nc   |               mm
----+--------+--------+----------------------------------
  1 | TDSQLPG   | TDSQLPG   | 202cb962ac59075b964b07152d234b70
  2 | pgxc  | pgxc | 202cb962ac59075b964b07152d234b70
  3 | pgxz | pgxz | 202cb962ac59075b964b07152d234b70
(3 行记录)
postgres=# COMMIT;
COMMIT
```

返回多态类型

```
postgres=# CREATE OR REPLACE FUNCTION f23(a_arg anyelement) RETURNS anyelement AS
postgres-# $$
postgres$# BEGIN
postgres$#     RETURN a_arg;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f23('TDSQLPG'::text);
 f23
------
 TDSQLPG
(1 行记录)
postgres=# SELECT * FROM f23(1::integer);
 f23
-----
   1
(1 行记录)

postgres=# SELECT * FROM f23(ARRAY['TDSQLPG','pgxz']);
     f23
---------------
 {TDSQLPG,pgxz}
(1 行记录)
postgres=# SELECT * FROM f23(ROW(1,'TDSQLPG')::public.t_rec);
 id |  mc
----+------
  1 | TDSQLPG
(1 行记录)
postgres=# CREATE OR REPLACE FUNCTION f24(a_arg ANYARRAY) RETURNS anyarray AS
postgres-# $$
postgres$# BEGIN
postgres$#     RETURN a_arg;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=#
postgres=# SELECT * FROM f24(ARRAY[1,2]::INTEGER[]);
  f24
-------
 {1,2}
```

```
(1 行记录)
postgres=# SELECT f24(ARRAY[t1.*]) FROM t1;
                        f24
-----------------------------------------------------
 {"(1,TDSQLPG,TDSQLPG,202cb962ac59075b964b07152d234b70)"}
 {"(2,pgxc,pgxc,202cb962ac59075b964b07152d234b70)"}
 {"(3,pgxz,pgxz,202cb962ac59075b964b07152d234b70)"}
(3 行记录)
```

返回数据类型如果是多态，则函数最少需要定义一个多态参数。

变量使用
变量使用介绍

在一个块中使用的所有变量必须在该块的声明小节中事先进行声明,PL/pgSQL变量可以是任意 SQL 数据类型，可以是一个简单数据类型、复合类型、RECORD、已经存在的表行类型、表字段类型、游标。

变量使用实例

变量声明语法

name [CONSTANT] type [COLLATE collation_name] [NOT NULL] [{ DEFAULT | := | = }expression];
如果给定DEFAULT子句，它会指定进入该块时分配给该变量的初始值。如果没有给出DEFAULT子句， 则该变量被初始化为SQL空值。CONSTANT选项阻止该变量在初始化之后被赋值， 这样它的值在块的持续期内保持不变。
COLLATE 选项指定用于该变量的一个排序规则（见 第41.3.6 节）。如果指 定了NOTNULL，对该变量赋值为空值会导致一个 运行时错误。所有被声明为NOT NULL的变量必须 被指定一个非空默认值。 等号（ = ）可以被用来代替 PL/SQL-兼容的 :=。

定义一个普通变量

```
postgres=# CREATE OR REPLACE FUNCTION f25() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     #所有变量的声明都要放在这里,建议变量以v_开头,参数以a_开头
postgres$#     v_int integer := 1;
postgres$#     v_text text;
postgres$# BEGIN
postgres$#     v_text = 'TDSQL PG';
postgres$#     RAISE NOTICE 'v_int = %',v_int;
postgres$#     RAISE NOTICE 'v_text = %',v_text;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f25();
NOTICE:  v_int = 1
NOTICE:  v_text = TDSQL PG
 f25
-----

(1 row)
postgres=#
```

定义CONSTANT变量

```
postgres=# CREATE OR REPLACE FUNCTION f25() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_int CONSTANT integer := 1;
postgres$# BEGIN
postgres$#     RAISE NOTICE 'v_int = %',v_int;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# select f25();
NOTICE:  v_int = 1
 f25
-----
(1 row)
#CONSTANT 不能再次赋值
postgres=# CREATE OR REPLACE FUNCTION f25() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_int CONSTANT integer := 1;
postgres$# BEGIN
postgres$#     RAISE NOTICE 'v_int = %',v_int;
postgres$#     v_int = 10;
postgres$#     RAISE NOTICE 'v_int = %',v_int;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
ERROR:  "v_int" is declared CONSTANT
```

定义NOT NULL变量

```
postgres=# CREATE OR REPLACE FUNCTION f25() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_int integer NOT NULL := 1;
postgres$# BEGIN
postgres$#     RAISE NOTICE 'v_int = %',v_int;
postgres$#     SELECT NULL INTO v_int;
postgres$#     RAISE NOTICE 'v_int = %',v_int;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f25();
NOTICE:  v_int = 1
ERROR:  null value cannot be assigned to variable "v_int" declared NOT NULL
CONTEXT:  PL/pgSQL function f25() line 6 at SQL statement
postgres=#
```

定义为NOT NULL变量，则该变量受NOT NULL约束。

定义COLLATE变量

按unicode值对比大小。

```
postgres=# CREATE OR REPLACE FUNCTION f25() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_txt1 TEXT COLLATE "C" := '严';
postgres$#     v_txt2 TEXT COLLATE "C" := '丰';
postgres$# BEGIN
postgres$#     IF v_txt1 > v_txt2 THEN
postgres$#         RAISE NOTICE ' % -> % ',v_txt1,v_txt2;
postgres$#     ELSE
postgres$#         RAISE NOTICE ' % -> % ',v_txt2,v_txt1;
postgres$#     END IF;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f25();
NOTICE:   丰 -> 严
 f25
-----
(1 row)
postgres=# select '严'::bytea;
  bytea
----------
 \xe4b8a5
(1 row)
postgres=# select '丰'::bytea;
  bytea
----------
 \xe4b8b0
(1 row)
#按汉字的拼音对比大小
postgres=# CREATE OR REPLACE FUNCTION f25() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_txt1 TEXT COLLATE "zh_CN.utf8" := '严';
postgres$#     v_txt2 TEXT COLLATE "zh_CN.utf8" := '丰';
postgres$# BEGIN
postgres$#     IF v_txt1 > v_txt2 THEN
postgres$#         RAISE NOTICE ' % -> % ',v_txt1,v_txt2;
postgres$#     ELSE
postgres$#         RAISE NOTICE ' % -> % ',v_txt2,v_txt1;
```

```
postgres$#     END IF;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f25();
NOTICE:  严 -> 丰
 f25
-----
(1 row)
```

变量赋值

```
postgres=# CREATE OR REPLACE FUNCTION f25() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     #定义时赋值
postgres$#     v_int1 integer = 1;
postgres$#     #使用 :=兼容于plsql
postgres$#     v_int2 integer := 1;
postgres$#     v_txt1 text;
postgres$#     v_float float8;
postgres$#     #使用查询赋值
postgres$#     v_relname text = (select relname FROM pg_class LIMIT 1);
postgres$#     v_relpages integer;
postgres$#     v_rec RECORD;
postgres$# BEGIN
postgres$#     #在函数体中赋值
postgres$#     v_txt1 = 'TDSQL PG';
postgres$#     v_float = random();
postgres$#     #使用查询赋值的另一种方式
postgres$#     SELECT relname,relpages INTO v_relname,v_relpages FROM  pg_class ORDER BY random
() LIMIT 1;
postgres$#     RAISE NOTICE 'v_relname = % , relpages = %',v_relname,v_relpages;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT * FROM f25();
NOTICE:  v_relname = pg_ts_parser , relpages = 1
 f25
-----

(1 row)
```

PL/SQL函数实战
批量设置表owner的函数

```
CREATE OR REPLACE FUNCTION public.alter_owner(a_schema_name varchar,a_role_name varchar) RET
URNS TEXT AS
$$
DECLARE
    -- a_schema_name:指定某个模式下，不对定是对数据库的所有表
    -- a_role_name：表所有者
    v_rec RECORD;
    v_sql TEXT;
BEGIN
    IF  a_schema_name != '' THEN  --如果指模式，检查模式是否存
        PERFORM 1 FROM pg_namespace WHERE nspname = a_schema_name;
        IF NOT FOUND THEN
            RETURN '指定的模式 ' || a_schema_name || ' 不存在!';
        END IF;
    END IF;
    PERFORM 1 FROM pg_roles WHERE rolname = a_role_name ;
    IF NOT FOUND THEN  --检查用户是否存在
        RETURN '指定的用户 ' || a_role_name || ' 不存在!';
    END IF;
    IF  a_schema_name != '' THEN    --指定了模式
            v_sql:='SELECT schemaname,tablename FROM pg_tables WHERE schemaname=''' || a_sche || '''
';
    ELSE
            v_sql:='SELECT schemaname,tablename FROM pg_tables WHERE schemaname!=''pg_catalog''
AND schemaname!=''information_schema'' ';
    END IF;
    FOR v_rec IN EXECUTE v_sql LOOP
        EXECUTE 'ALTER TABLE "'||v_rec.schemaname||'"."'||v_rec.tablename||'" OWNER TO '||a_role_name;
    END LOOP;
    RETURN 'ok';
END;
$$
LANGUAGE PLPGSQL;
COMMENT ON FUNCTION  public.alter_owner(a_schema_name varchar,a_role_name varchar)  IS '批量
设置表所有者';
CREATE OR REPLACE FUNCTION public.alter_owner(a_role_name varchar) RETURNS TEXT AS
$$
BEGIN
    RETURN public.alter_owner('',a_role_name);
END;
$$
```

```
LANGUAGE PLPGSQL;
COMMENT ON FUNCTION  public.alter_owner(a_role_name varchar)  IS '批量设置表所有者重载';
```

批量设置表的加密规则函数

```
create or replace function MLS_TRANSPARENT_CRYPT_ALGORITHM_BIND_ALL_TABLE(a_schema text,a_
algo_id int) returns text as
$$
declare
   v_rec record;
   v_algorithm_name text;
   v_raise_notice text;
   v_sqlstate text;
   v_context text;
   v_message_text text;
begin
   perform 1 from pg_namespace where nspname=a_schema ;
   if not found then
      return '模式['||a_schema||']不存在';
   end if;
   --显示使用的加密算法
   select algorithm_name INTO v_algorithm_name from pg_transparent_crypt_policy_algorithm where
algorithm_id=a_algo_id;
   if not found then
      return '加密算法id['||a_algo_id::text||']不存在';
   else
      raise notice '你使用的密码算法为--%',v_algorithm_name;
   end if;
   for v_rec in select pg_tables.schemaname,pg_tables.tablename,pg_transparent_crypt_policy_map.tbl
name from pg_tables left outer join pg_transparent_crypt_policy_map on pg_tables.schemaname=pg_t
ransparent_crypt_policy_map.nspname and pg_tables.tablename=pg_transparent_crypt_policy_map.tbl
name where pg_tables.schemaname=a_schema and pg_transparent_crypt_policy_map.tblname is null l
oop
      begin
        PERFORM MLS_TRANSPARENT_CRYPT_ALGORITHM_BIND_TABLE(v_rec.schemaname,v_rec.table
name, a_algo_id);
        EXCEPTION WHEN OTHERS THEN
        GET STACKED DIAGNOSTICS v_sqlstate = RETURNED_SQLSTATE,
                        v_message_text = MESSAGE_TEXT,
                        v_context = PG_EXCEPTION_CONTEXT;
        RAISE NOTICE '出错信息 : %',v_message_text;
      end;
   end loop;
   return '配置表加密完成';
end;
$$
language plpgsql;
```

oracle to_date函数的实现

TDSQL PG可以使用to_timestamp函数代替，如果应用程序中已经大量的使用的oracle to_date函数并且是精细到时:分:秒，那我们可以自定义一个to_date函数存放到一个优先访问的schema中，函数的内容为to_timestamp，这样就可以兼容原来的oracle应用了，如下所示：

```
postgres=# create schema oracle;
CREATE SCHEMA
postgres=# CREATE OR REPLACE FUNCTION oracle.to_date(a_date text,a_style text ) RETURNS timesta
mp with time zone AS
postgres-# $$
postgres$# BEGIN
postgres$#     RETURN to_timestamp(a_date,a_style);
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=# set search_path = oracle ,"$user", public,pg_catalog;;
SET
postgres=# select to_date('2028-01-01 13:14:20','yyyy-MM-dd HH24:MI');
       to_date
-----------------------
 2028-01-01 13:14:00+08
(1 row)
```

关于包(PACKAGE)
语法说明
包结构

包有两个组成部分：包说明（Specification）与包体（Body）。
包说明由公共变量、游标、对象、过程、函数和异常的声明组成，是一个应用程序接口 (API)，包含客户端程序调用其子程序所需的所有信息，但没有关于其实现的信息。
包体定义了对已声明游标的查询，以及在包说明中声明的子程序的代码（因此，既没有声明游标也没有子程序的包不需要主体）。包体还可以定义局部子程序，这些子程序在包说明中没有声明，只能被包中的其他子程序调用（内部调用）。

创建包

创建包说明

```
CREATE [OR REPLACE] PACKAGE 包名IS

#定义数据类型
type xxx is 数据类型。

#定义函数
function 函数名(参数 参数类型...) return 返回类型;

#定义存储过程
procedure 存储过程(参数 参数类型...) ;

END
/
```

创建包体

```
CREATE [OR REPLACE] PACKAGE BODY 包名IS
#函数实现
function 函数名(参数 参数类型...) return 返回类型 is

#定义变量
begin
----
end

#存储过程实现
procedure 存储过程(参数 参数类型...)  is

begin
----
end;

END
/
```

删除包

---删除包体
DROP PACKAGE BODY 包名;
---删除包
DROP PACKAGE 包名;

包的使用
函数在包中的用法

```
#定义
postgres=#CREATE OR REPLACE package b_1 IS
   #定义函数
   function addnum(a_1 number,a_2 number) return number;
end;
/
CREATE OR REPLACE package body b_1 is
   #实现方法
   function addnum(a_1 number,a_2 number) return number is
      num number;
   begin
      num:=a_1+a_2;
      return num;
   end;
end;
/
#调用
postgres=# select b_1.addnum(1,1);
 addnum
--------
      2
(1 row)
```

存储过程在包中的用法

```
#定义
postgres=#
drop table t1;
create table t1(f1 int ,f2 varchar);
CREATE OR REPLACE package b_2 IS
   #定义过程
   procedure insert_record(a_f1 int,a_f2 varchar);
end;
/
CREATE OR REPLACE package body b_2 is
   #实现过程
   procedure insert_record(a_f1 int,a_f2 varchar) is
   begin
      insert into t1 values(a_f1,a_f2);
   end;
end;
/
#调用
postgres=# call b_2.insert_record(1,'tdsql pg');
CALL
postgres=# select * from t1;
 f1 |  f2
----+-------
  1 | tdsql pg
(1 row)
```

函数与存储过程一起使用

```
postgres=#
drop table t1;
create table t1(f1 int ,f2 varchar);
CREATE OR REPLACE package b_3 IS
   #定义函数
   function addnum(a_1 number,a_2 number) return number;
   #定义过程
   procedure insert_record(a_f1 int,a_f2 varchar);
end;
/
CREATE OR REPLACE package body b_3 is
   #实现方法
   function addnum(a_1 number,a_2 number) return number is
      num number;
   begin
      num:=a_1+a_2;
      return num;
   end;

   #实现过程
   procedure insert_record(a_f1 int,a_f2 varchar) is
   begin
      insert into t1 values(a_f1,a_f2);
   end;
end;
/
#调用
postgres=# select b_3.addnum(1,1);
 addnum
--------
      2
(1 row)
postgres=# call b_3.insert_record(1,'tdsql pg');
CALL
postgres=# select * from t1;
 f1 |  f2
----+-------
  1 | tdsql pg
(1 row)
```

变量使用方法

```
postgres=# CREATE OR REPLACE package b_4 IS
    #定义变量
    v_1 int default 10;
    #定义结构体
    type v_struct is record(
        f1 int,
        f2 varchar
    );
    v_struct_1 v_struct;
    #定义游标,但不能定义游标变量
    type v_ref is ref cursor;

    #定义存储过程
    procedure setval();
    procedure echoval();
end;
/
CREATE OR REPLACE package body b_4 is
    #实现过程
    procedure setval() is
    begin
        v_1:=random()*10;
        v_struct_1.f1:=random()*10;
        v_struct_1.f2:=md5(random()::text);
    end;

    procedure echoval() is
        v_rec record;
        v_ref_1 v_ref;
        type myvarchar is table of varchar;
        v_myvarchar_1 myvarchar:='{"tbase","pgxz"}';
    begin

        raise notice 'v_1 = %',v_1;
        raise notice 'v_struct_1 = %',v_struct_1;
        raise notice 'v_myvarchar = %',v_myvarchar_1;
        open v_ref_1 for select schemaname,tablename from pg_tables limit 2;
        loop
            fetch v_ref_1 into v_rec;
            exit when v_ref_1%notfound;
            raise notice 'v_rec = %',v_rec;
        end loop;
    end;
```

```
end;
/
#调用
postgres=# call b_4.setval();
CALL
postgres=# call b_4.echoval();
NOTICE:  v_1 = 8
NOTICE:  v_struct_1 = (9,618ad5bdc38b67af927e46d5ba2fca60)
NOTICE:  v_myvarchar = {tbase,pgxz}
NOTICE:  v_rec = (public,t1)
NOTICE:  v_rec = (squeeze,tables_internal)
CALL
postgres=#
postgres=# declare
    v_1 int:=b_4.v_1;   #定义包中变量
begin
    raise notice 'v_1 = %',v_1;
b_4.v_1:=11;
raise notice 'b_4.v_1 = %',b_4.v_1;
end;
/
NOTICE:  v_1 = 11
NOTICE:  b_4.v_1 = 11
DO
```

其他常见使用

- 查看已经存在的package。

```
postgres=# select pkgname from pg_package;
```

- 查看已经存在的公共变量和函数列表。

```
postgres=# select pkgheadersrc from pg_package;
```

- 查看函数列表。

```
postgres=# \df b_3.*
```

- 查看函数内容。

```
postgres=# \df+ b_3.addnum
postgres=# \df+ b_3.insert_record
```

关于触发器
创建触发器

使用命令
CREATE TRIGGER... ON ... EXECUTE PROCEDURE ... 创建触发器
触发事件可以是INSERT，UPDATE [OF column_name [, ...]]，DELETE或者TRUNCATE
示例：触发事件，插入记录时，如果f2字段值小于0时，则将f2的值修改成0
创建表

```
create table t_trigger(f1 int,f2 int);
```

创建函数

```
CREATE OR REPLACE FUNCTION t_trigger_insert_trigger_func () RETURNS trigger AS
$body$
BEGIN
   if NEW.f2
      NEW.f2=0;
  end if;
  RETURN NEW;
END;
$body$
LANGUAGE plpgsql;
```

创建触发器

```
CREATE TRIGGER t_trigger_insert_trigger BEFORE INSERT ON t_trigger FOR EACH ROW EXECUTE PROC
EDURE t_trigger_insert_trigger_func();
```

插入数据

```
insert into t_trigger values(1,-1);
```

查询数据

```
select * from t_trigger;
```

⚠查询结果f2为0，插入数据事件触发了触发器，直接将f2的值修改为0

表触发器创建与删除

创建触发器

```
#创建测试表
postgres=# CREATE SEQUENCE s1;
CREATE SEQUENCE
postgres=#  CREATE TABLE t1 (id INT NOT NULL,nickname varchar(20));
CREATE TABLE
postgres=#
#创建触发器函数
postgres=# CREATE OR REPLACE FUNCTION t1_insert_trigger_func () RETURNS trigger AS
$body$
BEGIN
    NEW.ID=NEXTVAL('s1');
    RETURN NEW;
END;
$body$
LANGUAGE 'plpgsql';
CREATE FUNCTION
#创建触发器
postgres=# CREATE TRIGGER t1_insert_trigger BEFORE INSERT ON t1 FOR EACH ROW EXECUTE PROCE
DURE t1_insert_trigger_func();
CREATE TRIGGER
postgres=#
#测试
postgres=# insert into t1(nickname) values('tbase');
INSERT 0 1
postgres=# select * from t1;
 id | nickname
----+----------
  1 | tbase
(1 row)
postgres=#
```

删除触发器

```
postgres=# drop trigger t1_insert_trigger on t1;
DROP TRIGGER
postgres=#
```

删除触发器

使用DROP TRIGGER 删除触发器
示例：

```
DROP TRIGGER t_trigger_insert_trigger ON t_trigger;
```

触发器函数
INSERT事件触发器函数

函数功能实现字段值t_trigger.nc值重写。

```
postgres=# CREATE TABLE t_trigger
postgres-# (
postgres(#     id integer NOT NULL,
postgres(#     nc text NOT NULL
postgres(# );
CREATE TABLE
postgres=# CREATE OR REPLACE FUNCTION t_trigger_insert_trigger_func() RETURNS trigger AS
postgres-# $$
postgres$# BEGIN
postgres$#     IF NEW.nc = '' THEN
postgres$#         NEW.nc = 'TDSQLPG_' || random()::text;
postgres$#     END IF;
postgres$#     RETURN NEW;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# CREATE TRIGGER t_trigger_insert_trigger BEFORE INSERT ON t_trigger FOR EACH ROW EX
ECUTE PROCEDURE t_trigger_insert_trigger_func();
CREATE TRIGGER
postgres=# INSERT INTO t_trigger values(1,'');
INSERT 0 1
postgres=# SELECT * FROM t_trigger ;
 id |           nc
----+------------------------
  1 | TDSQLPG_0.426093454472721
(1 row)
```

> 注意：
>
> 使用BEFORE，不能使用AFTER，否则重写失效。

UPDATE事件触发器函数

不准许更新t_trigger.nc字段值为 TDSQL PG。

```
postgres=# CREATE OR REPLACE FUNCTION t_trigger_update_trigger_func() RETURNS trigger AS
postgres-# $$
postgres$# BEGIN
postgres$#     --不准许t_trigger.nc值为 TBase
postgres$#     IF NEW.nc = 'TDSQL PG' THEN
postgres$#         NEW.nc = OLD.nc ;
postgres$#     END IF;
postgres$#     RETURN NEW;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# CREATE TRIGGER t_trigger_update_trigger BEFORE UPDATE ON t_trigger FOR EACH ROW E
XECUTE PROCEDURE t_trigger_update_trigger_func();
CREATE TRIGGER
postgres=# UPDATE t_trigger SET nc='TDSQL PG' WHERE id=1;
UPDATE 1
postgres=# SELECT * FROM t_trigger ;
 id |          nc
----+-----------------------
  1 | TDSQLPG_0.426093454472721
(1 row)
postgres=#
```

DELETE事件触发器函数

限制TDSQL PG记录不能被删除。

```
postgres=# CREATE OR REPLACE FUNCTION t_trigger_delete_trigger_func() RETURNS trigger AS
postgres-# $$
postgres$# BEGIN
postgres$#     #不准许t_trigger.nc值为 TDSQL PG
postgres$#     IF OLD.nc = 'TDSQL PG' THEN
postgres$#         RETURN NULL;
postgres$#         --RAISE EXCEPTION 'TDSQL PG不能被删除';
postgres$#     END IF;
postgres$#     RETURN OLD;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# CREATE TRIGGER t_trigger_delete_trigger BEFORE DELETE ON t_trigger FOR EACH ROW EX
ECUTE PROCEDURE t_trigger_delete_trigger_func();
CREATE TRIGGER
postgres=# INSERT INTO t_trigger VALUES(2,'TDSQL PG');
INSERT 0 1
postgres=# SELECT * t_trigg
postgres=# SELECT * FROM t_trigger ;
 id |          nc
----+-----------------------
  1 | TDSQLPG_0.426093454472721
  2 | TDSQL PG
(2 rows)
postgres=# DELETE FROM t_trigger WHERE id=2;
DELETE 0
postgres=# SELECT * FROM t_trigger ;
 id |          nc
----+-----------------------
  1 | TDSQLPG_0.426093454472721
  2 | TDSQL PG
(2 rows)
```

删除触发器

```
postgres=# drop TRIGGER t_trigger_insert_trigger on t_trigger;
DROP TRIGGER
```

触发器使用限制

分区表，冷热分区表和复制表不支持使用触发器。

ORACLE兼容性特性与SQL参考
启用Oracle兼容性

启用Oracle兼容特性，需要进行参数 enable_oracle_compatible 设置。

> 注意：
>
> TDSQL PG Oracle兼容版v5.0x版本该参数目前不可关闭。

- session中生效
  set enable_oracle_compatible to on;
- 配置某个库默认生效
  alter database postgres set enable_oracle_compatible to on;
- 配置某个账号生效
  alter role tbase_user set enable_oracle_compatible to on;

系统package
ALL_ARGUMENTS

列出了当前用户可以访问的函数和过程的参数。

名称	类型	说明
owner	varchar2(128)	对象所有者
object_name	varchar2(128)	过程或函数名称
package_name	varchar2(128)	包名称
object_id	number	对象的对象编号
overload	varchar2(40)	无效
subprogram_id	number	无效
argument_name	varchar2(128)	参数名。空参数名称用于表示函数返回
position	number	表示在参数列表中的位置，或 0 表示函数返回值
sequence	number	定义参数的顺序。 参数序列从 1 开始。返回类型在前，然后是每个参数
data_level	number	复合类型参数的嵌套深度
data_type	varchar2(30)	参数数据类型
defaulted	varchar2(1)	指定参数是否默认
default_value	long	保留
default_length	number	保留
in_out	varchar2(9)	参数方向:IN/OUT/INOUT
data_length	number	无效
data_precision	number	无效
data_scale	number	无效
radix	number	无效
character_set_name	varchar2(44)	无效

名称	类型	说明
type_owner	varchar2(128)	无效
type_name	varchar2(128)	无效
type_subname	varchar2(128)	无效
type_link	varchar2(128)	无效
type_object_type	varchar2(7)	无效
pls_type	varchar2(128)	无效
char_length	number	无效
char_used	varchar2(1)	无效
origin_con_id	varchar2(256)	无效

示例：

```
create or replace FUNCTION f3f_all_argument_20221111_1 (a int, b out number, c INOUT varchar2) retu
rn int is
begin
return a;
end;
/
create or replace procedure p3p_all_argument_20221111_1(e int, f out number, g INOUT varchar2) is
aa int;
begin
aa := e;
end;
/
create or replace function f1f_all_argument_20221111_2(a int, b number default 123) return int is
begin
return a;
end;
/
create or replace procedure p5p_all_argument_20221111_2(a int, b varchar2 default 'abc', c number de
fault 123, d number default NULL) is
e int;
begin
e := a;
end;
/
create or replace procedure pDp_all_argument_20221111_3(a int) is
```

```
begin
null;
end;
/
create or replace type obj_all_argument_20221111_1 as object(a int);
/
CREATE OR REPLACE TYPE nt_all_argument_20221111_2 AS TABLE OF VARCHAR2(30);
/
create table tbl_all_arguments_20221111_1 (f1 int, f2 varchar2(60));
create or replace function argTYPes_all_argument_20221111_3(a int[], obj obj_all_argument_20221111_
1, nt nt_all_argument_20221111_2, rt tbl_all_arguments_20221111_1%rowtype) return int is
begin
return 2;
end;
/
select owner, object_name, package_name, overload,
      SUBPROGRAM_ID, ARGUMENT_NAME, position, sequence,
      DEFAULTED, DEFAULT_VALUE, IN_OUT, data_length, radix
from all_arguments where object_name = upper('argTYPes_all_argument_20221111_3') order by object
_name, position;
```

DBMS_JOB

DBMS_JOB主要用来对在JOB QUEUE中的JOB做管理。该包已经被DBMS_SCHEDULER 包所取代. 特别是,如果你正在管理JOB以控制系统负载，那么你应该考虑通过收回用户在该 包上的执行权限，以禁和DBMS_JOB包。

示例：

```
declare
 jobno number;
 begin dbms_job.submit(
  job=>jobno,
    what=>'proc_job(2);',
    next_date=>sysdate,
   interval=>'sysdate+1/1440');
    COMMIT;
   dbms_output.put_line(jobno);
 end;
```

DBMS_ASSERT

DBMS_ASSERT包提供了一个验证输入值属性的接口。

存储过程/函数	描述
ENQUOTE_LITERAL	引用字符串文字
ENQUOTE_NAME	确保字符串用引号括起来，然后检查结果是否为有效的SQL标识符。
QUALIFIED_SQL_NAME	验证输入字符串是否为合格的SQL名称
SCHEMA_NAME	验证输入字符串是否为现有模式名称
SIMPLE_SQL_NAME	验证输入字符串是否为简单的SQL名称
SQL_OBJECT_NAME	验证输入参数字符串是现有SQL对象的限定SQL标识符

示例：

```
select dbms_assert.enquote_literal('tbase''') from dual;
ERROR:  numeric or value error
select dbms_assert.enquote_name('TBase') from dual;
select dbms_assert.qualified_sql_name('create table') from dual;
ERROR:  string is not qualified SQL name
select dbms_assert.schema_name('TBASE') from dual;
select dbms_assert.simple_sql_name('2tb') from dual;
ERROR:  string is not simple SQL name
select dbms_assert.sql_object_name('T1111') from dual;
ERROR:  invalid object name
```

DBMS_LOB

DBMS_LOB用于在大对象上进行操作。DBMS_LOB包提供了子程序可以在BLOB、CLOB、NCLOB、BFILE和临时LOB上操作的子程序。使用DBMS_LOB可以访问和处理LOB的特定部分或全部

存储过程/函数	描述
APPEND(dest_lob IN OUT,src_lob)	Appends one large object to another.
COMPARE(lob_1, lob_2 [, amount[, offset_1 [, offset_2]]])	Compares two large objects.
CONVERTOBLOB(dest_lob IN OUT,src_clob, amount, dest_offsetIN OUT, src_offset IN OUT,blob_csid, lang_context IN OUT,warning OUT)	Converts character data to binary.
CONVERTTOCLOB(dest_lob IN OUT,src_blob, amount, dest_offsetIN OUT, src_offset IN OUT,blob_csid, lang_context IN OUT,warning OUT)	Converts binary data to character.
COPY(dest_lob IN OUT, src_lob,amount [, dest_offset [,src_offset]])	Copies one large object to another.
ERASE(lob_loc IN OUT, amount IN OUT [, offset])	Erase a large object.
GET_STORAGE_LIMIT(lob_loc)	Get the storage limit for large objects.
GETLENGTH(lob_loc)	Get the length of the large object.
INSTR(lob_loc, pattern [,offset [, nth]])	Get the position of the nth occurrence of a pattern in the large object starting atoffset.
READ(lob_loc, amount IN OUT,offset, buffer OUT)	Read a large object.
SUBSTR(lob_loc [, amount [,offset]])	Get part of a large object.
TRIM(lob_loc IN OUT, newlen)	Trim a large object to the specified length.
WRITE(lob_loc IN OUT, amount,offset, buffer)	Write data to a large object.
WRITEAPPEND(lob_loc IN OUT,amount, buffer)	Write data from the buffer to the end of a large object.

示例：

```
DBMS_LOB.WRITEAPPEND (
lob_loc IN OUT NOCOPY BLOB,
amount  IN           INTEGER,
buffer  IN           RAW);

DBMS_LOB.WRITEAPPEND (
lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,
amount  IN           INTEGER,
buffer  IN           VARCHAR2 CHARACTER SET lob_loc%CHARSET);

declare
   v_clob1 clob;
begin
   v_clob1:=to_clob('123456');
   dbms_output.put_line(v_clob1);
   DBMS_LOB.WRITEAPPEND(v_clob1,3,'789');
   dbms_output.put_line(v_clob1);
end;
```

DBMS_OUTPUT

DBMS_OUTPUT用于向消息缓冲区发送消息（文本行的形式出现），或者从消息缓冲区中获取消息。

存储过程/函数	描述
DISABLE	Disable the capability to send and receive messages.
ENABLE(buffer size)	Enable the capability to send and receive messages.
GET LINE(line OUT, status OUT)	Get a line from the message buffer.
GET LINES(lines OUT, numlines IN OUT)	Get multiple lines from the message buffer.
NEW LINE	Puts an end-of-line character sequence.
PUT(item)	Puts a partial line without an end-of-line character sequence.
PUT LINE(item)	Puts a complete line with an end-of-line character sequence.
SERVEROUTPUT(stdout)	Direct messages from PUT, PUT LINE, or NEW_LINE to either standard output or the message buffer.

示例：

```
\set ECHO none
SET client_min_messages = warning;
SET DATESTYLE TO ISO;
SET client_encoding = utf8;
\pset null '<NULL>'
\set ECHO all

DROP FUNCTION dbms_output_test();
DROP TABLE dbms_output_test;

-- DBMS_OUTPUT.DISABLE [0]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
   buff      VARCHAR(20);
   stts      INTEGER;
BEGIN
   PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
   SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
   INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
```

```
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.PUT_LINE [1]
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
    buff1 VARCHAR(20) := 'orafce';
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('t');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE');
    PERFORM DBMS_OUTPUT.PUT_LINE (buff1);
    PERFORM DBMS_OUTPUT.PUT ('ABC');
    PERFORM DBMS_OUTPUT.PUT_LINE ('');
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.PUT_LINE [2]
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('t');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORA
F
CE');
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.PUT [1]
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
    buff1    VARCHAR(20) := 'ora';
    buff2    VARCHAR(20) := 'f';
    buff3    VARCHAR(20) := 'ce';
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('t');
```

```
    PERFORM DBMS_OUTPUT.PUT ('ORA');
    PERFORM DBMS_OUTPUT.PUT ('F');
    PERFORM DBMS_OUTPUT.PUT ('CE');
    PERFORM DBMS_OUTPUT.PUT_LINE ('');
    PERFORM DBMS_OUTPUT.PUT ('ABC');
    PERFORM DBMS_OUTPUT.PUT_LINE ('');
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.PUT [2]
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('t');
    PERFORM DBMS_OUTPUT.PUT ('ORA
F
CE');
    PERFORM DBMS_OUTPUT.PUT_LINE ('');
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.GET_LINE [1]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
    buff      VARCHAR(20);
    stts      INTEGER;
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 2');
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
```

```
SELECT * FROM dbms_output_test order by buff;
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.GET_LINE [2]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
   buff      VARCHAR(20);
   stts      INTEGER;
BEGIN
   PERFORM DBMS_OUTPUT.DISABLE();
   PERFORM DBMS_OUTPUT.ENABLE();
   PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
   PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
   PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 2');
   SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
   INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
   PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 3');
   SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
   INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
   SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
   INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
   SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
   INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
   PERFORM DBMS_OUTPUT.DISABLE();
   PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.GET_LINE [3]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
   buff      VARCHAR(20);
   stts      INTEGER;
BEGIN
   PERFORM DBMS_OUTPUT.DISABLE();
   PERFORM DBMS_OUTPUT.ENABLE();
   PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
   PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
   PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 2');
   SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
   INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
```

```
    PERFORM DBMS_OUTPUT.PUT ('ORA');
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.GET_LINE [4]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
    buff      VARCHAR(20);
    stts      INTEGER;
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 2');
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    PERFORM DBMS_OUTPUT.NEW_LINE();
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.GET_LINE [5]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
    buff      VARCHAR(20);
    stts      INTEGER;
BEGIN
```

```
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1
');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 2');
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.GET_LINE [6]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
    buff      VARCHAR(20);
    stts      INTEGER;
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORA
F
CE');
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.GET_LINES [1]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
    buff      VARCHAR(20);
    buff1     VARCHAR(20);
```

```
    buff2     VARCHAR(20);
    buff3     VARCHAR(20);
    stts      INTEGER := 10;
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 2');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 3');
    SELECT INTO buff1,buff2,buff3,stts lines[1],lines[2],lines[3],numlines FROM DBMS_OUTPUT.GET_LINE
S(stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff1, stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff2, stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff3, stts);
    SELECT INTO buff,stts lines[1],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.GET_LINES [2]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
    buff      VARCHAR(20);
    buff1     VARCHAR(20);
    buff2     VARCHAR(20);
    stts      INTEGER := 2;
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 2');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 3');
    SELECT INTO buff1,buff2,stts lines[1],lines[2],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff1, stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff2, stts);
    SELECT INTO buff,stts lines[1],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
```

```
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.GET_LINES [3]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
   buff      VARCHAR(20);
   stts      INTEGER := 1;
BEGIN
   PERFORM DBMS_OUTPUT.DISABLE();
   PERFORM DBMS_OUTPUT.ENABLE();
   PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
   PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
   PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 2');
   SELECT INTO buff,stts lines[1],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
   INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
   PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 3');
   SELECT INTO buff,stts lines[1],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
   INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
   SELECT INTO buff,stts lines[1],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
   INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
   PERFORM DBMS_OUTPUT.DISABLE();
   PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.GET_LINES [4]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
   buff      VARCHAR(20);
   stts      INTEGER := 1;
BEGIN
   PERFORM DBMS_OUTPUT.DISABLE();
   PERFORM DBMS_OUTPUT.ENABLE();
   PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
   PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
   PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 2');
   SELECT INTO buff,stts lines[1],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
   INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
```

```
    PERFORM DBMS_OUTPUT.PUT ('ORA');
    SELECT INTO buff,stts lines[1],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    SELECT INTO buff,stts lines[1],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.GET_LINES [5]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
    buff      VARCHAR(20);
    stts      INTEGER := 1;
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 2');
    SELECT INTO buff,stts lines[1],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    PERFORM DBMS_OUTPUT.NEW_LINE();
    SELECT INTO buff,stts lines[1],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    SELECT INTO buff,stts lines[1],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.GET_LINES [6]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
    buff      VARCHAR(20);
    stts      INTEGER := 1;
BEGIN
```

```
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORA
F
CE');
    SELECT INTO buff,stts lines[1],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.NEW_LINE [1]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
    buff1    VARCHAR(20);
    buff2    VARCHAR(20);
    stts     INTEGER := 10;
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.PUT ('ORA');
    PERFORM DBMS_OUTPUT.NEW_LINE();
    PERFORM DBMS_OUTPUT.PUT ('FCE');
    PERFORM DBMS_OUTPUT.NEW_LINE();
    SELECT INTO buff1,buff2,stts lines[1],lines[2],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff1, stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff2, stts);
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.NEW_LINE [2]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(3000), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
    buff1    VARCHAR(3000);
```

```
    stts       INTEGER := 10;
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.ENABLE(2000);
    FOR j IN 1..1999 LOOP
        PERFORM DBMS_OUTPUT.PUT ('A');
    END LOOP;
    PERFORM DBMS_OUTPUT.NEW_LINE();
    SELECT INTO buff1,stts lines[1],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff1, stts);
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.DISABLE [1]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
    buff       VARCHAR(20);
    stts       INTEGER;
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);

    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 2');
    PERFORM DBMS_OUTPUT.ENABLE();
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);

    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 3');
```

```
    PERFORM DBMS_OUTPUT.DISABLE();
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    PERFORM DBMS_OUTPUT.ENABLE();

    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.PUT ('ORAFCE TEST 4');
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.NEW_LINE();
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);

    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.PUT ('ORAFCE TEST 5');
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.NEW_LINE();
    PERFORM DBMS_OUTPUT.ENABLE();
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);

    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.DISABLE [2]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
    buff      VARCHAR(20);
    stts      INTEGER := 10;
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
    SELECT INTO buff,stts lines[1],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    PERFORM DBMS_OUTPUT.ENABLE();
END;
```

```
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.ENABLE [1]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
   buff      VARCHAR(20);
   status    INTEGER;
   num            INTEGER := 2000;
BEGIN
   PERFORM DBMS_OUTPUT.DISABLE();
   PERFORM DBMS_OUTPUT.ENABLE();
   PERFORM DBMS_OUTPUT.SERVEROUTPUT ('t');
   PERFORM DBMS_OUTPUT.ENABLE(2000);
   PERFORM DBMS_OUTPUT.PUT ('ORAFCE TEST 1');
   PERFORM DBMS_OUTPUT.NEW_LINE();
   PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.ENABLE [2]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
   buff      VARCHAR(20);
   stts      INTEGER;
BEGIN
   PERFORM DBMS_OUTPUT.DISABLE();
   PERFORM DBMS_OUTPUT.ENABLE();
   PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
   PERFORM DBMS_OUTPUT.DISABLE();
   PERFORM DBMS_OUTPUT.ENABLE();
   PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
   SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
   INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
   PERFORM DBMS_OUTPUT.PUT ('ORAFCE TEST 2');
   PERFORM DBMS_OUTPUT.NEW_LINE();
   SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
   INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
   PERFORM DBMS_OUTPUT.ENABLE();
END;
```

```
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.ENABLE [3]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
   buff      VARCHAR(20);
   stts      INTEGER := 10;
BEGIN
   PERFORM DBMS_OUTPUT.DISABLE();
   PERFORM DBMS_OUTPUT.ENABLE();
   PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
   PERFORM DBMS_OUTPUT.DISABLE();
   PERFORM DBMS_OUTPUT.ENABLE();
   PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
   SELECT INTO buff,stts lines[1],numlines FROM DBMS_OUTPUT.GET_LINES(stts);
   INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);

END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.ENABLE [4]
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
BEGIN
   PERFORM DBMS_OUTPUT.DISABLE();
   PERFORM DBMS_OUTPUT.ENABLE();
   PERFORM DBMS_OUTPUT.SERVEROUTPUT ('t');
   PERFORM DBMS_OUTPUT.DISABLE();
   PERFORM DBMS_OUTPUT.ENABLE();
   FOR j IN 1..2000 LOOP
      PERFORM DBMS_OUTPUT.PUT ('A');
   END LOOP;
   PERFORM DBMS_OUTPUT.NEW_LINE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.ENABLE [5]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
```

```
DECLARE
    buff      VARCHAR(20);
    stts      INTEGER;
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE(NULL);
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- DBMS_OUTPUT.ENABLE [6]
CREATE TABLE dbms_output_test (id serial,buff VARCHAR(20), status INTEGER);
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
DECLARE
    buff      VARCHAR(20);
    stts      INTEGER;
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
    PERFORM DBMS_OUTPUT.ENABLE();
    SELECT INTO buff,stts line,status FROM DBMS_OUTPUT.GET_LINE();
    INSERT INTO dbms_output_test(buff,status) VALUES (buff, stts);
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP TABLE dbms_output_test;
DROP FUNCTION dbms_output_test();

-- SERVEROUTPUT [1]
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
```

```
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP FUNCTION dbms_output_test();

CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
BEGIn
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('t');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 2');
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP FUNCTION dbms_output_test();

-- SERVEROUTPUT [2]
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.PUT ('ORAFCE TEST 1');
    PERFORM DBMS_OUTPUT.NEW_LINE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP FUNCTION dbms_output_test();

CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('t');
    PERFORM DBMS_OUTPUT.PUT ('ORAFCE TEST 2');
    PERFORM DBMS_OUTPUT.NEW_LINE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP FUNCTION dbms_output_test();

-- SERVEROUTPUT [3]
CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
```

```
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('f');
    PERFORM DBMS_OUTPUT.DISABLE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP FUNCTION dbms_output_test();

CREATE FUNCTION dbms_output_test() RETURNS VOID AS $$
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT ('t');
    PERFORM DBMS_OUTPUT.PUT_LINE ('ORAFCE TEST 1');
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_test();
DROP FUNCTION dbms_output_test();
```

DBMS_PIPE

DBMS_PIPE包用于在同一实例的不同会话之间进行通信；注意，如果用户要执行包dbms_pipe中的过程和函数,则必须要为用户授权。DBMS_PIPE包含以下接口：

接口	描述
*CREATE_PIPE	用于建立公用管道或私有管道。如果将参数private设置为TRUE，则建立私有管道；如果设置为FALSE，则建立公用管道。
*PACK_MESSAGE	用于将消息写入到本地消息缓冲区，包含类型number，bytea，date，string，timestamp, record
*SEND_MESSAGE	用于将本地消息缓冲区中的内容发送到管道
*RECEIVE_MESSAGE	用于接收管道消息
NEXT_ITEM_TYPE	用于确定本地消息缓冲区下一项的数据类型。 如果该函数返回0,则表示管道没有任何消息
*UNPACK_MESSAGE	用于将消息缓冲区的内容写入到变量中
*REMOVE_PIPE	用于删除已经建立的管道
PUGER	用于清除管道中的内容
RESET_BUFFER	用于复位管道缓冲区
UNIQUE_SESSION_NAME	用于为特定会话返回惟一的名称,并且名称的最长度为30字节

示例：

```
create or replace procedure dbmspipe_crtpipe_pro(in_cno int) as
declare
v_no int;
v_bigint bigint;
v_bytea bytea;
v_date date;
v_int int;
v_num numeric;
v_text text;
v_tmptz timestamp with time zone;
begin
  select c,c_bigint,c_bytea,c_date,c_int,c_num,c_text,c_tmptz
  from dbmspipe_tbl
  where c=in_cno
```

```
into v_no,v_bigint,v_bytea,v_date,v_int,v_num,v_text,v_tmptz;
perform dbms_pipe.create_pipe('bigint_pipe1'||v_no);
perform dbms_pipe.pack_message(v_bigint);
perform dbms_pipe.send_message('bigint_pipe1'||v_no);
perform dbms_pipe.create_pipe('bigint_pipe2'||v_no,50);
perform dbms_pipe.pack_message(v_bigint);
perform dbms_pipe.send_message('bigint_pipe2'||v_no);
perform dbms_pipe.create_pipe('bigint_pipe3'||v_no,100,true);
perform dbms_pipe.pack_message(v_bigint);
perform dbms_pipe.send_message('bigint_pipe3'||v_no);
perform dbms_pipe.create_pipe('bigint_pipe4'||v_no,200,false);
perform dbms_pipe.pack_message(v_bigint);
perform dbms_pipe.send_message('bigint_pipe4'||v_no);

perform dbms_pipe.create_pipe('bytea_pipe1'||v_no);
perform dbms_pipe.pack_message(v_bytea);
perform dbms_pipe.send_message('bytea_pipe1'||v_no);
perform dbms_pipe.create_pipe('bytea_pipe2'||v_no,50);
perform dbms_pipe.pack_message(v_bytea);
perform dbms_pipe.send_message('bytea_pipe2'||v_no);
perform dbms_pipe.create_pipe('bytea_pipe3'||v_no,100,true);
perform dbms_pipe.pack_message(v_bytea);
perform dbms_pipe.send_message('bytea_pipe3'||v_no);
perform dbms_pipe.create_pipe('bytea_pipe4'||v_no,200,false);
perform dbms_pipe.pack_message(v_bytea);
perform dbms_pipe.send_message('bytea_pipe4'||v_no);

perform dbms_pipe.create_pipe('date_pipe1'||v_no);
perform dbms_pipe.pack_message(v_date);
perform dbms_pipe.send_message('date_pipe1'||v_no);
perform dbms_pipe.create_pipe('date_pipe2'||v_no,50);
perform dbms_pipe.pack_message(v_date);
perform dbms_pipe.send_message('date_pipe2'||v_no);
perform dbms_pipe.create_pipe('date_pipe3'||v_no,100,true);
perform dbms_pipe.pack_message(v_date);
perform dbms_pipe.send_message('date_pipe3'||v_no);
perform dbms_pipe.create_pipe('date_pipe4'||v_no,200,false);
perform dbms_pipe.pack_message(v_date);
perform dbms_pipe.send_message('date_pipe4'||v_no);

perform dbms_pipe.create_pipe('int_pipe1'||v_no);
perform dbms_pipe.pack_message(v_int);
perform dbms_pipe.send_message('int_pipe1'||v_no);
perform dbms_pipe.create_pipe('int_pipe2'||v_no,50);
perform dbms_pipe.pack_message(v_int);
perform dbms_pipe.send_message('int_pipe2'||v_no);
perform dbms_pipe.create_pipe('int_pipe3'||v_no,100,true);
```

```
  perform dbms_pipe.pack_message(v_int);
  perform dbms_pipe.send_message('int_pipe3'||v_no);
  perform dbms_pipe.create_pipe('int_pipe4'||v_no,200,false);
  perform dbms_pipe.pack_message(v_int);
  perform dbms_pipe.send_message('int_pipe4'||v_no);

  perform dbms_pipe.create_pipe('num_pipe1'||v_no);
  perform dbms_pipe.pack_message(v_num);
  perform dbms_pipe.send_message('num_pipe1'||v_no);
  perform dbms_pipe.create_pipe('num_pipe2'||v_no,50);
  perform dbms_pipe.pack_message(v_num);
  perform dbms_pipe.send_message('num_pipe2'||v_no);
  perform dbms_pipe.create_pipe('num_pipe3'||v_no,100,true);
  perform dbms_pipe.pack_message(v_num);
  perform dbms_pipe.send_message('num_pipe3'||v_no);
  perform dbms_pipe.create_pipe('num_pipe4'||v_no,200,false);
  perform dbms_pipe.pack_message(v_num);
  perform dbms_pipe.send_message('num_pipe4'||v_no);

  perform dbms_pipe.create_pipe('text_pipe1'||v_no);
  perform dbms_pipe.pack_message(v_text);
  perform dbms_pipe.send_message('text_pipe1'||v_no);
  perform dbms_pipe.create_pipe('text_pipe2'||v_no,50);
  perform dbms_pipe.pack_message(v_text);
  perform dbms_pipe.send_message('text_pipe2'||v_no);
  perform dbms_pipe.create_pipe('text_pipe3'||v_no,100,true);
  perform dbms_pipe.pack_message(v_text);
  perform dbms_pipe.send_message('text_pipe3'||v_no);
  perform dbms_pipe.create_pipe('text_pipe4'||v_no,200,false);
  perform dbms_pipe.pack_message(v_text);
  perform dbms_pipe.send_message('text_pipe4'||v_no);

  perform dbms_pipe.create_pipe('tmptz_pipe1'||v_no);
  perform dbms_pipe.pack_message(v_tmptz);
  perform dbms_pipe.send_message('tmptz_pipe1'||v_no);
  perform dbms_pipe.create_pipe('tmptz_pipe2'||v_no,50);
  perform dbms_pipe.pack_message(v_tmptz);
  perform dbms_pipe.send_message('tmptz_pipe2'||v_no);
  perform dbms_pipe.create_pipe('tmptz_pipe3'||v_no,100,true);
  perform dbms_pipe.pack_message(v_tmptz);
  perform dbms_pipe.send_message('tmptz_pipe3'||v_no);
  perform dbms_pipe.create_pipe('tmptz_pipe4'||v_no,200,false);
  perform dbms_pipe.pack_message(v_tmptz);
  perform dbms_pipe.send_message('tmptz_pipe4'||v_no);
end;
/
```

```
--创建接收pipe message，打印message子存储过程
create or replace procedure rec_subpro(in_pipename varchar) as
declare
v_num numeric;
v_bytea bytea;
v_date date;
v_str varchar;
v_tmptz timestamp with time zone;
begin
  perform dbms_output.disable();
  perform dbms_output.enable();
  perform dbms_output.serveroutput ('t');
  perform dbms_pipe.receive_message(in_pipename);
  if in_pipename like '%int%' or in_pipename like 'num' then
    v_num := dbms_pipe.unpack_message_number();
    perform dbms_output.put_line(in_pipename || ' message: '||v_num);
  elsif in_pipename like '%bytea%' then
    v_bytea := dbms_pipe.unpack_message_bytea();
    perform dbms_output.put_line(in_pipename || ' message: '||v_bytea);
  elsif in_pipename like '%date%' then
    v_date := dbms_pipe.unpack_message_date();
    perform dbms_output.put_line(in_pipename || ' message: '||v_date::text);
  elsif in_pipename like '%text%' then
    v_str := dbms_pipe.unpack_message_text();
    perform dbms_output.put_line(in_pipename || ' message: '||v_str);
  elsif in_pipename like '%tmptz%' then
    v_tmptz := dbms_pipe.unpack_message_timestamp();
    perform dbms_output.put_line(in_pipename || ' message: '||v_tmptz::text);
  end if;
  perform dbms_pipe.remove_pipe(in_pipename);
end;
/
--unpack_message_*测试：接收pipe中各种数据类型的message（record除外），接收后remove_pipe，打印出message
create or replace procedure dbmspipe_rec_pro(in_cno int) as
begin
  call rec_subpro('bigint_pipe1'||in_cno);
  call rec_subpro('bigint_pipe2'||in_cno);
  call rec_subpro('bigint_pipe3'||in_cno);
  call rec_subpro('bigint_pipe4'||in_cno);

  call rec_subpro('bytea_pipe1'||in_cno);
  call rec_subpro('bytea_pipe2'||in_cno);
  call rec_subpro('bytea_pipe3'||in_cno);
  call rec_subpro('bytea_pipe4'||in_cno);

  call rec_subpro('date_pipe1'||in_cno);
```

```
  call rec_subpro('date_pipe2'||in_cno);
  call rec_subpro('date_pipe3'||in_cno);
  call rec_subpro('date_pipe4'||in_cno);

  call rec_subpro('int_pipe1'||in_cno);
  call rec_subpro('int_pipe2'||in_cno);
  call rec_subpro('int_pipe3'||in_cno);
  call rec_subpro('int_pipe4'||in_cno);

  call rec_subpro('num_pipe1'||in_cno);
  call rec_subpro('num_pipe2'||in_cno);
  call rec_subpro('num_pipe3'||in_cno);
  call rec_subpro('num_pipe4'||in_cno);

  call rec_subpro('text_pipe1'||in_cno);
  call rec_subpro('text_pipe2'||in_cno);
  call rec_subpro('text_pipe3'||in_cno);
  call rec_subpro('text_pipe4'||in_cno);

  call rec_subpro('tmptz_pipe1'||in_cno);
  call rec_subpro('tmptz_pipe2'||in_cno);
  call rec_subpro('tmptz_pipe3'||in_cno);
  call rec_subpro('tmptz_pipe4'||in_cno);
end;
/
--sessionA:
call dbmspipe_crtpipe_pro(1);
--sessionB:
call dbmspipe_rec_pro(1);
--sessionA:
call dbmspipe_crtpipe_pro(2);
--sessionB:
call dbmspipe_rec_pro(2);
--sessionA:
call dbmspipe_crtpipe_pro(3);
--sessionB:
call dbmspipe_rec_pro(3);
--sessionA:
call dbmspipe_crtpipe_pro(4);
--sessionB:
call dbmspipe_rec_pro(4);
--sessionA:
call dbmspipe_crtpipe_pro(5);
--sessionB:
call dbmspipe_rec_pro(5);
```

DBMS_RANDOM value()函数示例

返回两值之间的随机数。

```
postgres=#select dbms_random.value(1,100);
      value
------------------
 40.3055268009193
```

DBMS_SQL

DBMS_SQL可以在应用的运行时间构建查询和其它的命令。DBMS_SQL中可以使用的存储过程及函数如下表所示：

存储过程/函数	描述
BIND_VARIABLE(c, name, value [, out_value_size])	Bind a value to a variable
BIND_VARIABLE_CHAR(c, name, value [, out_value_size])	Bind a CHAR value to a variable
BIND_VARIABLE_RAW(c, name, value [, out_value_size])	Bind a RAW value to a variable
CLOSE_CURSOR(c IN OUT)	Close a cursor
COLUMN_VALUE(c, position, value OUT [, column_error OUT [, actual_length OUT]])	Return a column value into a variable.
COLUMN_VALUE_CHAR(c, position, value OUT [, column_error OUT [, actual_length OUT]])	Return a CHAR column value into a variable.
COLUMN_VALUE_RAW(c, position, value OUT [, column_error OUT [, actual_length OUT]])	Return a RAW column value into a variable.
DEFINE_COLUMN(c, position, column [, column_size])	Define a column in the SELECT list.
DEFINE_COLUMN_CHAR(c, position, column, column_size)	Define a CHAR column in the SELECT list.
DEFINE_COLUMN_RAW(c, position, column, column_size)	Define a RAW column in the SELECT list.
DESCRIBE_COLUMNS	Defines columns to hold a cursor result set.
EXECUTE(c)	Execute a cursor.
EXECUTE_AND_FETCH(c [, exact])	Execute a cursor and fetch a single row.
FETCH_ROWS(c)	Fetch rows from the cursor.
IS_OPEN(c)	Check if a cursor is open.
LAST_ROW_COUNT	Return cumulative number of rows fetched.
OPEN_CURSOR	Open a cursor.

PARSE(c, statement, language_flag)	Parse a statement.

示例：

```
set client_min_messages TO error;
CREATE EXTENSION IF NOT EXISTS dbms_sql;
set client_min_messages TO default;

do
$$
  declare
    c      int;
    strval varchar;
    intval int;
    nrows  int default 30;
  begin
    c := dbms_sql.open_cursor();
    call dbms_sql.parse(c, 'select ''ahoj'' || i, i from generate_series(1, :nrows) g(i)');
    call dbms_sql.bind_variable(c, 'nrows', nrows);
    call dbms_sql.define_column(c, 1, strval);
    call dbms_sql.define_column(c, 2, intval);
    perform dbms_sql.execute(c);
    while dbms_sql.fetch_rows(c) > 0
      loop
        call dbms_sql.column_value(c, 1, strval);
        call dbms_sql.column_value(c, 2, intval);
        raise notice 'c1: %, c2: %', strval, intval;
      end loop;
    call dbms_sql.close_cursor(c);
  end;
$$;

do
$$
  declare
    c      int;
    strval varchar;
    intval int;
    nrows  int default 30;
  begin
    c := dbms_sql.open_cursor();
    call dbms_sql.parse(c, 'select ''ahoj'' || i, i from generate_series(1, :nrows) g(i)');
    call dbms_sql.bind_variable(c, 'nrows', nrows);
    call dbms_sql.define_column(c, 1, strval);
    call dbms_sql.define_column(c, 2, intval);
```

```
        perform dbms_sql.execute(c);
        while dbms_sql.fetch_rows(c) > 0
            loop
                strval := dbms_sql.column_value_f(c, 1, strval);
                intval := dbms_sql.column_value_f(c, 2, intval);
                raise notice 'c1: %, c2: %', strval, intval;
            end loop;
        call dbms_sql.close_cursor(c);
    end;
$$;

create table foo
(
    a int,
    b varchar,
    c numeric
);

do
$$
    declare
        c int;
    begin
        c := dbms_sql.open_cursor();
        call dbms_sql.parse(c, 'insert into foo values(:a, :b, :c)');
        for i in 1..100
            loop
                call dbms_sql.bind_variable(c, 'a', i);
                call dbms_sql.bind_variable(c, 'b', 'Ahoj ' || i);
                call dbms_sql.bind_variable(c, 'c', i + 0.033);
                perform dbms_sql.execute(c);
            end loop;
    end;
$$;

select *
from foo
order by a;
truncate foo;

do
$$
    declare
        c int;
    begin
        c := dbms_sql.open_cursor();
        call dbms_sql.parse(c, 'insert into foo values(:a, :b, :c)');
```

```
        for i in 1..100
            loop
                perform dbms_sql.bind_variable_f(c, 'a', i);
                perform dbms_sql.bind_variable_f(c, 'b', 'Ahoj ' || i);
                perform dbms_sql.bind_variable_f(c, 'c', i + 0.033);
                perform dbms_sql.execute(c);
            end loop;
    end;
$$;

select *
from foo
order by a;
truncate foo;

do
$$
    declare
        c  int;
        a  int[];
        b  varchar[];
        ca numeric[];
    begin
        c := dbms_sql.open_cursor();
        call dbms_sql.parse(c, 'insert into foo values(:a, :b, :c)');
        a := ARRAY [1, 2, 3, 4, 5];
        b := ARRAY ['Ahoj', 'Nazdar', 'Bazar'];
        ca := ARRAY [3.14, 2.22, 3.8, 4];

        call dbms_sql.bind_array(c, 'a', a);
        call dbms_sql.bind_array(c, 'b', b);
        call dbms_sql.bind_array(c, 'c', ca);
        raise notice 'inserted rows %', dbms_sql.execute(c);
    end;
$$;

select *
from foo
order by a;
truncate foo;

do
$$
    declare
        c  int;
        a  int[];
        b  varchar[];
```

```
    ca numeric[];
  begin
    c := dbms_sql.open_cursor();
    call dbms_sql.parse(c, 'insert into foo values(:a, :b, :c)');
    a := ARRAY [1, 2, 3, 4, 5];
    b := ARRAY ['Ahoj', 'Nazdar', 'Bazar'];
    ca := ARRAY [3.14, 2.22, 3.8, 4];

    call dbms_sql.bind_array(c, 'a', a, 2, 3);
    call dbms_sql.bind_array(c, 'b', b, 3, 4);
    call dbms_sql.bind_array(c, 'c', ca);
    raise notice 'inserted rows %', dbms_sql.execute(c);
  end;
$$;

select *
from foo
order by a;
truncate foo;

do
$$
  declare
    c  int;
    a  int[];
    b  varchar[];
    ca numeric[];
  begin
    c := dbms_sql.open_cursor();
    call dbms_sql.parse(c, 'select i, ''Ahoj'' || i, i + 0.003 from generate_series(1, 35) g(i)');
    call dbms_sql.define_array(c, 1, a, 10, 1);
    call dbms_sql.define_array(c, 2, b, 10, 1);
    call dbms_sql.define_array(c, 3, ca, 10, 1);

    perform dbms_sql.execute(c);
    while dbms_sql.fetch_rows(c) > 0
      loop
        call dbms_sql.column_value(c, 1, a);
        call dbms_sql.column_value(c, 2, b);
        call dbms_sql.column_value(c, 3, ca);
        raise notice 'a = %', a;
        raise notice 'b = %', b;
        raise notice 'c = %', ca;
      end loop;
    call dbms_sql.close_cursor(c);
  end;
$$;
```

DBMS_STATS

DBMS_STATS能良好地估计统计数据(尤其是针对较大的分区表)，并能获得更好的统计结果,最终制定出速度更快的SQL执行计划。包含以下接口：

接口	描述
GATHER_DATABASE_STATS	分析数据库，包括所有用户对象和系统对象
GATHER_TABLE_STATS	分析表
GET_COLUMN_STATS	取得列的统计信息
GET_INDEX_STATS	取得索引的统计信息
GET_TABLE_STATS	取得表的统计信息

示例：

```
CREATE EXTENSION IF NOT EXISTS tbase_oracle_package_function;

create user godlike_dbms_stats superuser;
create user joe;
create user no_privilege;

\c postgres godlike_dbms_stats
grant usage on schema dbms_stats to no_privilege;
grant usage on schema dbms_stats to joe;

\c postgres joe
-- table joe_t
create table joe_t (id integer not null PRIMARY KEY, test integer);
create index joe_t_test_idx on joe_t(test);
insert into joe_t SELECT generate_series(1,1000) as key, (random()*(10^3))::integer;

\c postgres joe
exec dbms_stats.gather_table_stats(ownname => 'joe',tabname => 'joe_t');
exec dbms_stats.get_table_stats(ownname => 'joe',tabname => 'joe_t');
exec dbms_stats.get_column_stats('joe', 'joe_t', 'test');
exec dbms_stats.get_index_stats('joe', 'joe_t_test_idx');

-- table joe_t_p
create table joe_t_p (id integer not null PRIMARY KEY, test integer) partition by range (id) begin (1) ste
p (5) partitions (200) distribute by shard(id);
create index joe_t_p_test_idx on joe_t_p(test);
```

```
insert into joe_t_p SELECT generate_series(1,1000) as key, (random()*(10^3))::integer;

exec dbms_stats.gather_database_stats();
exec dbms_stats.get_table_stats(ownname => 'joe',tabname => 'joe_t_p');
exec dbms_stats.get_column_stats('joe', 'joe_t_p', 'test');
exec dbms_stats.get_index_stats('joe', 'joe_t_p_test_idx');
exec dbms_stats.get_table_stats('joe', 'joe_t_p', 'joe_t_p_part_0');
exec dbms_stats.get_index_stats('joe', 'joe_t_p_test_idx', 'joe_t_p_test_idx_part_0');

-- clean
\c postgres godlike_dbms_stats
drop table joe_t;
drop table joe_t_p;
REVOKE usage ON schema dbms_stats FROM joe;
REVOKE usage ON schema dbms_stats FROM no_privilege;
drop user joe;
drop user no_privilege;
```

DBMS_UTILITY

FORMAT_CALL_STACK

这个内置函数返回一个格式化的字符串，它显示了执行调用堆栈，直至此函数的调用点处的所有过程或者函数的调用顺序。
示例：

```
create or replace function dbms_uti_func() returns text as
declare
v_str1 text;
v_str2 text;
v_str3 text;
v_str4 text;
begin
  v_str1 := dbms_utility.format_call_stack();
  select regexp_replace(v_str1,'[ 0-9a-fA-F]{4}[0-9a-fA-F]{4}','      0','g') into v_str1;
  select regexp_replace(v_str1,'[45()]','','g') into v_str1;
  v_str2 := dbms_utility.format_call_stack('o');
  select regexp_replace(v_str2,'[ 0-9a-fA-F]{4}[0-9a-fA-F]{4}','      0','g') into v_str2;
  select regexp_replace(v_str2,'[45()]','','g') into v_str2;
  v_str3 := dbms_utility.format_call_stack('p');
  select regexp_replace(v_str3,'[ 0-9a-fA-F]{4}[0-9a-fA-F]{4}','      0','g') into v_str3;
  select regexp_replace(v_str3,'[45()]','','g') into v_str3;
  v_str4 := dbms_utility.format_call_stack('s');
  select regexp_replace(v_str4,'[ 0-9a-fA-F]{4}[0-9a-fA-F]{4}','      0','g') into v_str4;
  select regexp_replace(v_str4,'[45()]','','g') into v_str4;
  return 'v_str1:
'||v_str1 || '
v_str2:
'|| v_str2|| '
v_str3:
'|| v_str3|| '
v_str4:
'|| v_str4;
end;
/
select dbms_uti_func();
```

GET_HASH_VALUE

这个内置函数用于计算给定字符串的散列值。
使用方法：DBMS_UTILITY.GET_HASH_VALUE (name VARCHAR2, base NUMBER, hash_size NUMBER) RETURN NUMBER;

参数	描述
name	需要计算其散列值的字符串。
base	需要生成散列值的起始值。
hash_size	所需散列表的散列值的数量。

返回值：计算出来的哈希值
例如，要获取哈希值应在1000到3047之间的字符串上的哈希值，请使用1000作为基值，2048作为hash_size值。
使用场景：对于指定的字符串，返回范围在[base, base+hase_size-1]的散列值
示例：

```
postgres=# select dbms_utility.get_hash_value('hello tdsql pg',1,10000) from dual;

 get_hash_value
----------------
           9930
(1 行记录)
```

> 注意：
>
> - name为null值无法计算哈希，约定name参数为null时且其他参数正常，则直接返回base值
> - base不可为null，可为负值。返回值的最小值为base；返回hash值的最大值bash+hash_size-1。
> - hash_size不可为null或者0，若hash_size为负值则返回的哈希值不在哈希范围内，哈希值可能为正，也可能为负（内部运算时会把hash_size转成正值）。
> - base和hash_size的最大最小值兼容oracle，因哈希算法不同，因此哈希值不兼容oracle的哈希值。

UTL_FILE

UTL_FILE包提供文本文件输入和输出功能。

UTL_FILE包含以下接口：

接口	描述
*FOPEN	用于打开文件
IS_OPEN	用于确定文件是否已经被打开
FCLOSE	用于关闭已经打开的文件
FCLOSE_ALL	该过程用于关闭当前打开的所有文件
GET_LINE	用于从已经打开的文件中读取行内容，行内容会被读取到输出缓冲区
GET_NEXTLINE	用于从已经打开的文件中读取下一条行内容
*PUT	用于将缓冲区内容写入到文件中。当使用PUT过程的时候，文件必须以写方式打开，在写入缓冲区之后，如果要结束行，那么可以使用NEW_LINE过程
NEW_LINE	该过程用于为文件增加行终止符
*PUT_LINE	该过程用于将文本缓冲区内容写入到文件中。当使用该过程为文件追加内容时，会自动在文件的尾部追加行终止符。
FFLUSH	用于将数据强制性写入到文件中，正常情况下，当给文件写入数据的时候，数据会被暂时的放到缓存中。过程FFLUSH用于强制将数据写入到文件中。
FREMOVE	用于删除磁盘文件
*FCOPY	用于将源文件的全部或者部分内容复制到目标文件中。
*FRENAME	该过程用于修改已经存在的文件名字，其作用于UNIX的MV命令完全相同，在修改文件名字的时候，通过指定overwrite参数可以覆盖已经存在的文件
FGETATTR	读取磁盘上的文件并返回文件的属性
FRENAME	将一个存在的文件重命名
PUTF	写入格式化的内容到文件中

示例：

```
create orreplace procedure utlfile_open_prowa() as
declare
v_count int;
```

```
v_fileintinteger;
begin
  --open_node=w
  v_fileint :=utl_file.fopen('/data1/tbasev5_autotest/tbaseTest/TbaseV5/pro_package/data/utl_file_dir','file_w.txt','w');
  perform utl_file.put_line(v_fileint,'writefile test.');
  if utl_file.is_open(v_fileint) then
    perform utl_file.fclose(v_fileint);
  end if;
  v_fileint :=utl_file.fopen('/data1/tbasev5_autotest/tbaseTest/TbaseV5/pro_package/data/utl_file_dir','file_w.txt','w',50);
  perform utl_file.new_line(v_fileint);
  perform utl_file.put_line(v_fileint,'一二三四五六七八九十');
  if utl_file.is_open(v_fileint) then
    perform utl_file.fclose(v_fileint);
  end if;
  v_fileint :=utl_file.fopen('/data1/tbasev5_autotest/tbaseTest/TbaseV5/pro_package/data/utl_file_dir','file_w.txt','w',18,'SQL_ASCII');
  perform utl_file.new_line(v_fileint);
  perform utl_file.put_line(v_fileint,'nice tomeet you');
  if utl_file.is_open(v_fileint) then
    perform utl_file.fclose(v_fileint);
  end if;
  v_fileint :=utl_file.fopen('/data1/tbasev5_autotest/tbaseTest/TbaseV5/pro_package/data/utl_file_dir','file_w.txt','w',31,'UTF-8');
  perform utl_file.new_line(v_fileint);
  perform utl_file.put_line(v_fileint,'abcdefghijklmn opq rst uvw xyz');
  if utl_file.is_open(v_fileint) then
    perform utl_file.fclose(v_fileint);
  end if;
  v_fileint :=utl_file.fopen('/data1/tbasev5_autotest/tbaseTest/TbaseV5/pro_package/data/utl_file_dir','file_w.txt','w',88,'GBK');
  perform utl_file.new_line(v_fileint);
  perform utl_file.put_line(v_fileint,'hello');
  if utl_file.is_open(v_fileint) then
    perform utl_file.fclose(v_fileint);
  end if;
  --open_mode=a
  v_fileint := utl_file.fopen('/data1/tbasev5_autotest/tbaseTest/TbaseV5/pro_package/data/utl_file_dir','file_w.txt','a');
  perform utl_file.put_line(v_fileint,'writefile test.');
  if utl_file.is_open(v_fileint) then
    perform utl_file.fclose(v_fileint);
  end if;
  v_fileint :=utl_file.fopen('/data1/tbasev5_autotest/tbaseTest/TbaseV5/pro_package/data/utl_file_dir','file_w.txt','a',50);
  perform utl_file.new_line(v_fileint);
```

```
  perform utl_file.put_line(v_fileint,'一二三四五六七八九十');
  if utl_file.is_open(v_fileint) then
    perform utl_file.fclose(v_fileint);
  end if;
  v_fileint :=utl_file.fopen('/data1/tbasev5_autotest/tbaseTest/TbaseV5/pro_package/data/utl_file_dir','file_w.txt','a',18,'SQL_ASCII');
  perform utl_file.new_line(v_fileint);
  perform utl_file.put_line(v_fileint,'nice tomeet you');
  if utl_file.is_open(v_fileint) then
    perform utl_file.fclose(v_fileint);
  end if;
  v_fileint :=utl_file.fopen('/data1/tbasev5_autotest/tbaseTest/TbaseV5/pro_package/data/utl_file_dir','file_w.txt','a',31,'UTF-8');
  perform utl_file.new_line(v_fileint);
  perform utl_file.put_line(v_fileint,'abcdefghijklmn opq rst uvw xyz');
  if utl_file.is_open(v_fileint) then
    perform utl_file.fclose(v_fileint);
  end if;
  v_fileint := utl_file.fopen('/data1/tbasev5_autotest/tbaseTest/TbaseV5/pro_package/data/utl_file_dir','file_w.txt','a',88,'GBK');
  perform utl_file.new_line(v_fileint);
  perform utl_file.put_line(v_fileint,'hello');
  if utl_file.is_open(v_fileint) then
    perform utl_file.fclose(v_fileint);
  end if;
exception
  when others then
    raise notice 'EXP: something wrong.';
end;
/
callutlfile_open_prowa();
```

UTL_RAW

UTL_RAW 系统包提供用于操作 RAW 数据类型的 SQL 函数。普通的 SQL 函数不在RAW 数据上操作，而 PL 不允许在 RAW 和 CHAR 数据类型之间重载。

存储过程/函数	描述
BIT_AND	对 RAW r1 和 RAW r2 中的值执行按位逻辑“与”，并返回原始的“与”结果。
BIT_COMPLEMENT	对 RAW r 中的值执行按位逻辑”补码”，并返回”补码”结果 RAW。
BIT_OR	对 RAW r1 和 RAW r2 中的值执行按位逻辑“或”，并返回原始的“或”结果。
BIT_XOR	对 RAW r1 和 RAW r2 中的值执行逐位逻辑“异或”，并返回原始的“异或”结果。
CAST_TO_RAW	将 VARCHAR2 值转换为 RAW 值。
CAST_TO_VARCHAR2	将RAW 值转换为 VARCHAR2 值。
COPIES	返回串联在一起的 r 的 n 个副本。
COMPARE	比较 RAW r1 和 RAW r2。
CONCAT	将最多 12 个 RAW 串连到一个 RAW 中。
LENGTH	返回 RAW r 的字节长度。
REVERSE	从端到端反转 RAW r 中的字节序列。
SUBSTR	返回 len 字节数，从 RAW r 的 pos 位开始计算。

示例：

```
select utl_raw.bit_and('ABC', 'ABC') fromdual;
select utl_raw.bit_and('ABC', '') fromdual;
select utl_raw.bit_and('', 'ABC') fromdual;
select utl_raw.bit_and('', '') from dual;
select utl_raw.bit_and(null, '') from dual;
select utl_raw.bit_and(null, null) fromdual;
select utl_raw.bit_and('f000', 'EE10') fromdual;
select utl_raw.bit_and('000', '111') fromdual;
select utl_raw.bit_and('1111', '111') fromdual;

select utl_raw.bit_or('ABC', 'ABC') fromdual;
select utl_raw.bit_or('ABC', '') from dual;
select utl_raw.bit_or('', 'ABC') from dual;
select utl_raw.bit_or('', '') from dual;
```

```
select utl_raw.bit_or(null, '') from dual;
select utl_raw.bit_or(null, null) fromdual;
select utl_raw.bit_or('f000', 'EE10') fromdual;
select utl_raw.bit_or('000', '111') fromdual;
select utl_raw.bit_or('1111', '111') fromdual;

select utl_raw.bit_xor('ABC', 'ABC') fromdual;
select utl_raw.bit_xor('ABC', '') fromdual;
select utl_raw.bit_xor('', 'ABC') fromdual;
select utl_raw.bit_xor('', '') from dual;
select utl_raw.bit_xor(null, '') from dual;
select utl_raw.bit_xor(null, null) fromdual;
select utl_raw.bit_xor('f000', 'EE10') fromdual;
select utl_raw.bit_xor('000', '111') fromdual;
select utl_raw.bit_xor('1111', '111') fromdual;

SELECT utl_raw.bit_complement('0102F3')FROM dual;
SELECTutl_raw.bit_complement('00000000000000000000000000') FROM dual;
SELECTutl_raw.bit_complement('FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF') FROM dual;
SELECT utl_raw.bit_complement('0246813579')FROM dual;
SELECTutl_raw.bit_complement('ABCDEF13579') FROM dual;
SELECT utl_raw.bit_complement('') FROMdual;
SELECT utl_raw.bit_complement(null) FROMdual;


select utl_raw.substr('1234567810', -4, 3)from dual;
select utl_raw.substr('1234567810', -5, 5)from dual;
select utl_raw.substr('1234567810', -5)from dual;
select utl_raw.substr('1', 0) from dual;
select utl_raw.substr('1', '1') from dual;
select utl_raw.substr('ABCD', '1E0') fromdual;
select utl_raw.substr('ABCD', '1E-1') fromdual;
select utl_raw.substr('ABCD', '0.2') fromdual;

select utl_raw.length(repeat('AB', 32768))from dual;
select utl_raw.length('') from dual;
select utl_raw.length(1E10||'') from dual;
select utl_raw.length(null) from dual;


select utl_raw.reverse('ABC123') from dual;
select utl_raw.reverse('ABCD') from dual;
select utl_raw.reverse('A') from dual;
select utl_raw.reverse(123E3||'') fromdual;

select utl_raw.concat('ABC') from dual;
select utl_raw.concat('ABC', '') from dual;
```

```
select utl_raw.concat('ABC', 99E2||'') fromdual;
select utl_raw.concat('', 'ABC') from dual;
select utl_raw.concat('', '') from dual;
select utl_raw.concat('') from dual;
select utl_raw.concat() from dual;
select utl_raw.concat('', null) from dual;
select utl_raw.concat(null, null) fromdual;
select utl_raw.concat('', '') from dual;
select utl_raw.concat(repeat('ABCD', 2000),repeat('ABCD', 2000)) from dual;

select utl_raw.compare('912', '') fromdual;
select utl_raw.compare('912', null) fromdual;
select utl_raw.compare('', null) from dual;
select utl_raw.compare(null, null) from dual;
select utl_raw.compare('912', '91245') fromdual;
select utl_raw.compare('ABC', 'ABCDEFF')from dual;
select utl_raw.compare('ABCD', 'ABCDEFF')from dual;


select utl_raw.COPIES('AB', 10) from dual;
select utl_raw.COPIES('AB123', 10) fromdual;


select utl_raw.cast_to_varchar2('1') fromdual;
select utl_raw.cast_to_varchar2('') fromdual;
select utl_raw.cast_to_varchar2(null) fromdual;
select utl_raw.cast_to_varchar2('6666')from dual;
selectutl_raw.cast_to_varchar2('E4BDA0E5A5BD') from dual;

select utl_raw.CAST_TO_RAW('ff') from dual;
select utl_raw.CAST_TO_RAW(' ') from dual;
select utl_raw.CAST_TO_RAW('') from dual;
select utl_raw.CAST_TO_RAW(null) from dual;
select utl_raw.CAST_TO_RAW(123) from dual;
select utl_raw.CAST_TO_RAW(123E4) fromdual;
select utl_raw.CAST_TO_RAW('嗯嗯ABC123.5') from dual;
```

系统视图
all_arguments

列出了当前用户可以访问的函数和过程的参数。

名称	类型	说明
owner	varchar2(128)	对象所有者
object_name	varchar2(128)	过程或函数名称
package_name	varchar2(128)	包名称
object_id	number	对象的对象编号
overload	varchar2(40)	无效
subprogram_id	number	无效
argument_name	varchar2(128)	参数名。空参数名称用于表示函数返回
position	number	表示在参数列表中的位置，或 0 表示函数返回值
sequence	number	定义参数的顺序。 参数序列从 1 开始。返回类型在前，然后是每个参数
data_level	number	复合类型参数的嵌套深度
data_type	varchar2(30)	参数数据类型
defaulted	varchar2(1)	指定参数是否默认
default_value	long	保留
default_length	number	保留
in_out	varchar2(9)	参数方向 - IN - OUT - INOUT
data_length	number	无效
data_precision	number	无效
data_scale	number	无效
radix	number	无效

名称	类型	说明
character_set_name	varchar2(44)	无效
type_owner	varchar2(128)	无效
type_name	varchar2(128)	无效
type_subname	varchar2(128)	无效
type_link	varchar2(128)	无效
type_object_type	varchar2(7)	无效
pls_type	varchar2(128)	无效
char_length	number	无效
char_used	varchar2(1)	无效
origin_con_id	varchar2(256)	无效

all_users

列出了对当前用户可见的数据库的所有用户。

名称	类型	说明
username	name	用户名
user_id	oid	用户 ID
created	text	创建用户的时间

all_tab_columns

所有表和视图中的所有列的信息

名称	类型	说明
owner	name	所属用户
table_schema	name	表或视图所在 schema 的名称。
table_name	name	列所在的表或视图的名称。
column_name	name	列的名称。
data_type	text	列的数据类型。
data_type_owner	name	数据类型所有者
data_length	integer	文本列的长度。
data_precision	integer	NUMBER 列的精度（位数）。
data_scale	integer	NUMBER 列的小数位数。
nullable	text	列是否可为空，可能值包括： Y：列可为空。 N：列不可为空。
column_id	numeric	表中列的相对位置。
data_default	character varying	分配给列的默认值。
num_distinct	numeric(38,0)	表格每列中不同值的数量
low_value	text	对于超过三行的表，第二低和第二高的值
high_value	text	N/A
density	text	N/A
num_nulls	real	列中的空值数
num_buckets	integer	列的直方图中的桶数
last_analyzed	text	最近分析此列的日期
sample_size	text	用于分析此列的样本量
character_set_name	text	字符集的名称

char_col_decl_length	integer	字符集的长度
global_stats	text	N/A
user_stats	text	N/A
avg_col_len	integer	列的平均长度（以字节为单位）
char_length	integer	以字符显示列的长度
char_used	text	N/A
v80_fmt_image	text	N/A
data_upgraded	text	N/A
histogram	text	N/A

all_col_comments

ALL_COL_COMMENTS所有表的字段注释 。

all_col_comments名称说明

名称	类型	说明
OWNER	NAME	所有者的名称。
SCHEMA_NAME	NAME	模式名称。
TABLE_NAME	NAME	对象名。
COLUMN_NAME	NAME	列名。
COMMENT	TEXT	列注释。

all_constraints

所有表中所有约束信息。

all_constraints名称说明

名称	类型	说明
owner	TEXT	约束所有者的名称。
constraint_schema	TEXT	约束所属 schema 的名称。
constraint_name	TEXT	约束的名称。
constraint_type	TEXT	约束类型。可能值包括： - C：检查约束 - F：外键约束 - P：主键约束 - U：唯一键约束 - R：引用完整性约束 - V：视图上的约束 - O：具有只读属性，在视图上
table_schema	NAME	约束所属表所在schema。
table_name	TEXT	约束所属表的名称。
search_condition	TEXT	应用于检查约束的搜索条件。
r_owner	TEXT	引用约束引用的表的所有者。
r_constraint_schema	-	引用表的约束定义的模式名称。
r_constraint_name	TEXT	引用表的约束定义的名称。
delete_rule	TEXT	引用约束的删除规则。可能值包括： - C：级联 - R：限制 - N：无操作
is_deferrable	BOOLEAN	指定了约束是否可延迟（T 或 F）。
deferred	BOOLEAN	指定约束是否已延迟（T 或 F）。

all_synonyms

描述了当前用户可以访问的同义词。

名称	类型	说明
owner	name	同义词的拥有者
synonym_name	name	同义词名字
table_owner	text	同义词引用对象的名字
table_name	name	同义词引用的对象名字
db_link	name	如果引用的是 DB Link，则表示 DB Link 的名字

all_indexes

描述了当前用户可访问的表上的索引

名称	类型	说明
owner	name	所有者
index_nsp	name	
index_name	name	索引名称
index_type	name	索引类型
table_owner	name	索引所属表的所有者
table_nsp	name	
table_name	name	表名
table_type	text	索引所属表名称
compression	text	索引的压缩类型
prefix_length	text	索引前缀长度
tablespace_name	name	索引的表空间的名称
ini_trans	text	初始交易数量
max_trans	text	最大交易次数
initial_extent	text	初始范围的大小
next_extent	text	辅助范围的大小
min_extents	text	段中允许的最小区数
max_extents	text	段中允许的最大区数
pct_increase	text	范围大小的百分比增加
pct_threshold	integer	每个索引条目允许的块空间的阈值百分比
include_column	text	要包含在索引组织表主键（非溢出）索引中的最后一列的列ID
freelists	text	分配给此段的进程空闲列表的数量
freelist_groups	text	分配给此段的空闲列表组的数量

pct_free	text	块中可用空间的最小百分比
logging	text	表示对索引的修改是否被记录了日志
blevel	integer	表示 B*-Tree 的深度
leaf_blocks	integer	索引中的叶块数
distinct_keys	integer	不同索引值的数量
avg_leaf_blocks_per_key	integer	索引中每个不同值出现的平均叶块数
avg_data_blocks_per_key	integer	表中由索引中的不同值指向的平均数据块数舍入为最接近的整数
clustering_factor	integer	表示基于索引值的表中行的顺序数
status	text	状态
num_rows	real	索引中的行数
sample_size	integer	索引的样本的大小
last_analyzed	text	最近分析此索引的日期
degree	text	扫描索引的每个实例的线程数
instances	text	要扫描索引的实例数
partitioned	text	指示索引是否已分区
temporary	text	指示索引是否在临时表上
generated	text	指示索引的名称是否是系统生成的
secondary	text	指示索引是否是辅助对象
buffer_pool	text	用于索引块的缓冲池
user_stats	text	统计信息是否由用户直接输入
duration	text	指示临时表的持续时间
pct_direct_access	integer	对于索引组织表上的二级索引
ityp_owner	text	对于域索引，indextype的所有者
ityp_name	text	对于域索引，indextype的名称
parameters	text	对于域索引，参数字符串

global_stats	text	对于分区索引，指定是通过整体分析索引收集统计信息，还是从基础索引分区和子分区的统计信息中估算统计信息
domidx_status	text	域索引的状态
domidx_opstatus	text	域索引上的操作状态
funcidx_status	text	基于函数的索引的状态
join_index	text	指示索引是否为连接索引
iot_redundant_pkey_elim	text	指示是否从索引组织表上的二级索引中删除冗余主键列
dropped	text	指示索引是否已被删除并且是否在回收站中

all_ind_columns

描述当前用户可访问的所有表的索引列

名称	类型	说明
index_owner	name	索引列的所有者
Index_nsp	name	
index_name	name	索引名称
table_owner	name	表的所有者
table_nsp	name	
table_name	text	表名
column_name	text	列名
column_position	smallint	列的编号
column_length	smallint	列长
char_length	integer	列的数据类型如果为char，表示字节数。否则为0
descend	text	默认DESC。指示列是按降序（DESC）还是按升序（ASC）排序

all_cons_columns

提供所有表中，约束包含的所有列的信息。

all_cons_columns名称说明

名称	类型	说明
owner	TEXT	约束所有者的用户名。
schema_name	TEXT	约束所属 schema 的名称。
constraint_name	TEXT	约束的名称。
table_name	TEXT	约束所属表的名称。
column_name	TEXT	约束中引用的列的名称。
position	SMALLINT	列在对象定义中的位置。

dba_ind_columns

描述了数据库中所有表和集群的所有索引的列。它的列与 ALL_IND_COLUMNS 中的列相同。

名称	类型	说明
index_owner	name	索引列的所有者
index_nsp	name	
index_name	name	索引名称
table_owner	name	表的所有者
table_nsp	name	
table_name	name	表名
column_name	name	列名
column_position	smallint	列的编号
column_length	smallint	列长
char_length	integer	列的数据类型如果为char，表示字节数。否则为0
descend	text	默认DESC。指示列是按降序（DESC）还是按升序（ASC）排序

dba_constraints

描述了数据库中所有表的所有约束定义。它的列与 ALL_CONSTRAINTS 中的列相同。

名称	类型	说明
owner	name	约束的拥有者
constraint_schema	name	
constraint_name	name	约束名字
constraint_type	text	索引类型
table_schema	name	
table_name	name	约束所在的表的名字
r_owner	name	被引用约束的拥有者
r_constraint_schema	name	
r_constraint_name	name	被引用约束的名字
delete_rule	text	外键约束的级联删除规则
is_deferrable	boolean	表示约束是 DEFERRABLE 还是 NOT DEFERRABLE
deferred	boolean	表示约束是 DEFERRED 还是 IMMEDIATE

dba_indexes

DBA_INDEXES描述了数据库中的所有索引。要收集此视图的统计信息，请使用 DBMS_STATS 包。此视图支持并行分区索引扫描。它的列与 ALL_INDEXES中的列相同。

名称	类型	说明
owner	name	所有者
index_nsp	name	
index_name	name	索引名称
index_type	name	索引类型
table_owner	name	索引所属表的所有者
table_nsp	name	
table_name	name	表名
table_type	text	索引所属表名称
compression	text	索引的压缩类型
prefix_length	text	索引前缀长度
tablespace_name	name	索引的表空间的名称
ini_trans	text	初始交易数量
max_trans	text	最大交易次数
initial_extent	text	初始范围的大小
next_extent	text	辅助范围的大小
min_extents	text	段中允许的最小区数
max_extents	text	段中允许的最大区数

pct_increase	text	范围大小的百分比增加
pct_threshold	integer	每个索引条目允许的块空间的阈值百分比
include_column	text	要包含在索引组织表主键（非溢出）索引中的最后一列的列ID
freelists	text	分配给此段的进程空闲列表的数量
freelist_groups	text	分配给此段的空闲列表组的数量
pct_free	text	块中可用空间的最小百分比
logging	text	表示对索引的修改是否被记录了日志
blevel	integer	表示 B*-Tree 的深度
leaf_blocks	integer	索引中的叶块数
distinct_keys	integer	不同索引值的数量
avg_leaf_blocks_per_key	integer	索引中每个不同值出现的平均叶块数
avg_data_blocks_per_key	integer	表中由索引中的不同值指向的平均数据块数舍入为最接近的整数
clustering_factor	integer	表示基于索引值的表中行的顺序数
status	text	状态
num_rows	real	索引中的行数
sample_size	integer	索引的样本的大小
last_analyzed	text	最近分析此索引的日期
degree	text	扫描索引的每个实例的线程数
instances	text	要扫描索引的实例数
partitioned	text	指示索引是否已分区
temporary	text	指示索引是否在临时表上
generated	text	指示索引的名称是否是系统生成的
secondary	text	指示索引是否是辅助对象
buffer_pool	text	用于索引块的缓冲池
user_stats	text	统计信息是否由用户直接输入
duration	text	指示临时表的持续时间

pct_direct_access	integer	对于索引组织表上的二级索引
ityp_owner	text	对于域索引，indextype的所有者
ityp_name	text	对于域索引，indextype的名称
parameters	text	对于域索引，参数字符串
global_stats	text	对于分区索引，指定是通过整体分析索引收集统计信息，还是从基础索引分区和子分区的统计信息中估算统计信息
domidx_status	text	域索引的状态
domidx_opstatus	text	域索引上的操作状态
funcidx_status	text	基于函数的索引的状态
join_index	text	指示索引是否为连接索引
iot_redundant_pkey_elim	text	指示是否从索引组织表上的二级索引中删除冗余主键列
dropped	text	指示索引是否已被删除并且是否在回收站中

dba_synonyms

描述了数据库中的所有同义词。它的列与 ALL_SYNONYMS 中的列相同

名称	类型	说明
owner	name	同义词的拥有者
synonym_name	name	同义词名字
table_owner	text	同义词引用对象的名字
table_name	name	同义词引用的对象名字
db_link	name	如果引用的是 DB Link，则表示 DB Link 的名字

dba_tab_columns

DBA_TAB_COLUMNS描述了数据库中所有表、视图和集群的列。

名称	类型	说明
owner	name	所属用户
table_schema	name	表或视图所在 schema 的名称。
table_name	name	列所在的表或视图的名称。
column_name	name	列的名称。
data_type	text	列的数据类型。
data_type_owner	name	数据类型所有者
data_length	integer	文本列的长度。
data_precision	integer	NUMBER 列的精度（位数）。
data_scale	integer	NUMBER 列的小数位数。
nullable	text	列是否可为空，可能值包括： Y：列可为空。 N：列不可为空。
column_id	smallint	表中列的相对位置。
data_default	text	分配给列的默认值。
num_distinct	numeric(38,0)	表格每列中不同值的数量
low_value	text	对于超过三行的表，第二低和第二高的值
high_value	text	N/A
density	text	N/A
num_nulls	real	列中的空值数
num_buckets	integer	列的直方图中的桶数
last_analyzed	text	最近分析此列的日期
sample_size	text	用于分析此列的样本量
character_set_name	text	字符集的名称

char_col_decl_length	integer	字符集的长度
global_stats	text	N/A
user_stats	text	N/A
avg_col_len	integer	列的平均长度（以字节为单位）
char_length	integer	以字符显示列的长度
char_used	text	N/A
v80_fmt_image	text	N/A
data_upgraded	text	N/A
histogram	text	N/A

dba_users

DBA_USERS描述了数据库的所有用户。

名称	类型	说明
username	name	用户名
user_id	oid	用户的身份证号
password	text	此列不建议,使用 AUTHENTICATION_TYPE 列
account_status	text	帐户状态
lock_date	timestamp with time zone	如果帐户状态被锁定，则锁定帐户的日期
expiry_date	timestamp with time zone	账户到期日期
default_tablespace	text	数据的默认表空间
temporary_tablespace	text	临时表的默认表空间的名称或表空间组的名称
created	text	用户创建日期
profile	name	用户资源配置文件名称
initial_rsrc_consumer_group	text	用户的初始资源使用者组
external_name	text	用户外部名称

user_indexes

USER_INDEXES描述当前用户拥有的索引。要收集此视图的统计信息，请使用 DBMS_STATS 包。此视图支持并行分区索引扫描。它的列（除了 OWNER）与 ALL_INDEXES 中的列相同。

名称	类型	说明
index_nsp	name	
index_name	name	索引名称
index_type	name	索引类型
table_owner	name	索引所属表的所有者
table_nsp	name	
table_name	name	表名
table_type	text	索引所属表名称
compression	text	索引的压缩类型
prefix_length	text	索引前缀长度
tablespace_name	name	索引的表空间的名称
ini_trans	text	初始交易数量
max_trans	text	最大交易次数
initial_extent	text	初始范围的大小
next_extent	text	辅助范围的大小
min_extents	text	段中允许的最小区数
max_extents	text	段中允许的最大区数
pct_increase	text	范围大小的百分比增加
pct_threshold	integer	每个索引条目允许的块空间的阈值百分比

include_column	text	要包含在索引组织表主键（非溢出）索引中的最后一列的列ID
freelists	text	分配给此段的进程空闲列表的数量
freelist_groups	text	分配给此段的空闲列表组的数量
pct_free	text	块中可用空间的最小百分比
logging	text	表示对索引的修改是否被记录了日志
blevel	integer	表示 B*-Tree 的深度
leaf_blocks	integer	索引中的叶块数
distinct_keys	integer	不同索引值的数量
avg_leaf_blocks_per_key	integer	索引中每个不同值出现的平均叶块数
avg_data_blocks_per_key	integer	表中由索引中的不同值指向的平均数据块数舍入为最接近的整数
clustering_factor	integer	表示基于索引值的表中行的顺序数
status	text	状态
num_rows	real	索引中的行数
sample_size	integer	索引的样本的大小
last_analyzed	text	最近分析此索引的日期
degree	text	扫描索引的每个实例的线程数
instances	text	要扫描索引的实例数
partitioned	text	指示索引是否已分区
temporary	text	指示索引是否在临时表上
generated	text	指示索引的名称是否是系统生成的
secondary	text	指示索引是否是辅助对象
buffer_pool	text	用于索引块的缓冲池
user_stats	text	统计信息是否由用户直接输入
duration	text	指示临时表的持续时间
pct_direct_access	integer	对于索引组织表上的二级索引
ityp_owner	text	对于域索引，indextype的所有者

ityp_name	text	对于域索引，indextype的名称
parameters	text	对于域索引，参数字符串
global_stats	text	对于分区索引，指定是通过整体分析索引收集统计信息，还是从基础索引分区和子分区的统计信息中估算统计信息
domidx_status	text	域索引的状态
domidx_opstatus	text	域索引上的操作状态
funcidx_status	text	基于函数的索引的状态
join_index	text	指示索引是否为连接索引
iot_redundant_pkey_elim	text	指示是否从索引组织表上的二级索引中删除冗余主键列
dropped	text	指示索引是否已被删除并且是否在回收站

user_synonyms

描述了私有同义词（当前用户拥有的同义词）。它的列（除了 OWNER）与 ALL_SYNONYMS 中的列相同。

名称	类型	说明
synonym_name	name	同义词的拥有者
table_owner	text	同义词名字
table_name	name	同义词引用对象的名字
db_link	name	同义词引用的对象名字

user_cons_columns

提供当前用户拥有的表中，约束包含的所有列的信息。

名称	类型	说明
constraint _schema	name	约束所属 schema 的名称。
constraint_name	name	约束的名称。
table_schema	name	约束所属表的模式名称
table_name	name	约束所属表的名称。
column_name	name	约束中引用的列的名称。
position	smallint	列在对象定义中的位置。

user_tab_columns

提供当前用户拥有的表和视图中的所有列的信息。

名称	类型	说明
table_schema	name	表或视图所在 schema 的名称。
table_name	name	列所在的表或视图的名称。
column_name	name	列的名称。
data_type	text	列的数据类型。
data_type_owner	name	数据类型所有者
data_length	integer	文本列的长度。
data_precision	integer	NUMBER 列的精度（位数）。
data_scale	integer	NUMBER 列的小数位数。
nullable	text	列是否可为空，可能值包括： Y：列可为空。 N：列不可为空。
column_id	smallint	表中列的相对位置。
data_default	text	分配给列的默认值。
num_distinct	numeric(38,0)	表格每列中不同值的数量
low_value	text	对于超过三行的表，第二低和第二高的值
high_value	text	N/A
density	text	N/A
num_nulls	real	列中的空值数
num_buckets	integer	列的直方图中的桶数
last_analyzed	text	最近分析此列的日期
sample_size	text	用于分析此列的样本量
character_set_name	text	字符集的名称
char_col_decl_length	integer	字符集的长度

global_stats	text	N/A
user_stats	text	N/A
avg_col_len	integer	列的平均长度（以字节为单位）
char_length	integer	以字符显示列的长度
char_used	text	N/A
v80_fmt_image	text	N/A
data_upgraded	text	N/A
histogram	text	N/A

user_ind_columns

描述了当前用户拥有的索引的列以及当前用户拥有的表的索引列。

名称	类型	说明
index_owner	name	索引列的所有者
index_nsp	name	
index_name	name	索引名称
table_owner	name	表的所有者
table_nsp	name	
table_name	name	表名
column_name	name	列名
column_position	smallint	列的编号
column_length	smallint	列长
char_length	integer	列的数据类型如果为char，表示字节数。否则为0
descend	text	默认DESC。指示列是按降序（DESC）还是按升序（ASC）排序

user_constraints

用户表约束提供当前用户拥有的表中放置的所有约束的信息。。

user_constraints名称说明

名称	类型	说明
constraint_schema	TEXT	约束所属 schema 的名称。
constraint_name	TEXT	约束的名称。
constraint_type	TEXT	约束类型。可能值包括： - C：检查约束 - F：外键约束 - P：主键约束 - U：唯一键约束 - R：引用完整性约束 - V：视图上的约束 - O：具有只读属性，在视图上
table_schema	NAME	约束所属表所在schema。
table_name	TEXT	约束所属表的名称。
search_condition	TEXT	应用于检查约束的搜索条件。
r_owner	TEXT	引用约束引用的表的所有者。
r_constraint_schema	-	引用表的约束定义的模式名称。
r_constraint_name	TEXT	引用表的约束定义的名称。
delete_rule	TEXT	引用约束的删除规则。可能值包括： - C：级联 - R：限制 - N：无操作
is_deferrable	BOOLEAN	指定了约束是否可延迟（T 或 F）。
Deferred	BOOLEAN	指定约束是否已延迟（T 或 F）。

user_col_comments

用户表的字在当前用户拥有的表和视图的列上显示注释。其列（OWNER除外）与ALL_COL_COMMENTS中的列相同。注释。

user_col_comments名称说明

名称	类型	说明
SCHEMA_NAME	NAME	模式名称。
TABLE_NAME	NAME	对象名。
COLUMN_NAME	NAME	列名。
COMMENTS	TEXT	列注释。

user_users

描述当前用户信息

名称	类型	说明
username	name	用户名
user_id	oid	用户身份证号
account_status	text	帐户状态
lock_date	Timestamp with time zone	如果帐户状态为锁定，则帐户被锁定的日期
expiry_date	Timestamp with time zone	账户到期日期
default_tablespace	text	数据的默认表空间
temporary_tablespace	text	临时表的默认表空间的名称或表空间组的名称
created	text	用户创建日期
initial_rsrc_consumer_group	text	用户的初始资源使用者组
external_name	text	用户外部名称

函数
单行函数
数值函数

函数名	功能描述
ABS	返回指定数值表达式的绝对值（正值）的数学函数。
ACOS	返回以弧度表示的角，其余弦为指定 NUMBER表达式，也称为反余弦。
ASIN	求反正弦值。
ATAN	用于求反正切值。
ATAN2	返回 y 和 x 的反正切值，即返回的是原点至点 (x,y) 与 x 轴的夹角。
BITAND	运算符按位进行“与”操作。输入和输出类型均为 INT 整型，且类型一致。
CEIL	返回值大于等于数值 numeric_expression 的最小整数。
COS	用于计算参数角度的余弦值。
EXP	返回 e 的 numeric_expression 次幂。
FLOOR	返回小于等于数值 numeric_expression 的最大整数。
MOD	返回 x 除以 y 的余数。
NANVL	用于判断输入值参数 n1 是不是 NaN（表示非数字），并返回结果。
POWER	返回 x 的 y 次幂。
REMAINDER	返回 x 除以 y 的余数。
ROUND	返回 numberic 四舍五入后的值。
SIGN	返回数字 n 的符号，大于 0 返回 1，小于 0 返回 -1 ，等于 0 返回 0。
SIN	返回参数角度的正弦值。
SQRT	返回 n 的平方根。
TAN	返回角度的正切值。
TANH	返回数值参数的双曲正切值。
TRUNC	返回 numeric 按精度 precision 截取后的值。

返回字符值的字符函数
部分返回字符值的字符函数示例
regexp_substr

- string：需要进行正则处理的字符串 。
- pattern：进行匹配的正则表达式 。
- position：起始位置，从字符串的第几个字符开始正则表达式匹配（默认为1） 注意：字符串最初的位置是1而不是0。
- occurrence：获取第几个分割出来的组（分割后最初的字符串会按分割的顺序排列成组）。
- modifier：模式（‘i’不区分大小写进行检索；‘c’区分大小写进行检索。默认为’c’）针对的是正则表达式里字符大小写的匹配。

```
postgres=# SELECT REGEXP_SUBSTR('17,20,23','[^,]+',1,1,'i') AS STR FROM DUAL;
 str
-----
 17
(1 row)
postgres=#
```

nlssort

指定排序规则。

```
postgres=# create table t_nlssort(f1 integer,f2 varchar2(10));
CREATE TABLE
postgres=#
postgres=# insert into t_nlssort values(1,'平台');
INSERT 0 1
postgres=#
postgres=# insert into t_nlssort values(2,'深圳');
INSERT 0 1
postgres=#
postgres=# insert into t_nlssort values(3,'中国');
INSERT 0 1
postgres=# SELECT * FROM t_nlssort ORDER BY NLSSORT(f2,'NLS_SORT = SCHINESE_PINYIN_M');
 f1 |  f2
----+------
  2 | 城市1
  1 | 平台
  3 | 全球
(1 rows)
```

目前tbase只能支持按拼音。

REGEXP_INSTR

寻找字符第一个出现位置。

```
create or replace function regexp_instr(str text,str2 text,str3 INTEGER) returns text as
$$
declare
    res text;
begin
    res = str3 - 1 + position(substring(substring(str, str3) from str2) in str);
    return res;
end;
$$
language plpgsql;
create or replace function regexp_instr(str text,str2 text) returns text as $$
declare
    res text;
begin
    res = position(substring(str from str2) in str);
    return res;
end;
$$
language plpgsql;
```

调用结果

```
#搜索第一次出现的位置
postgres=# select regexp_instr('abc123abc156', '[1-2]+')  from  dual;
 regexp_instr
--------------
 4
(1 row)
#搜索第一次出现的位置
postgres=# select regexp_instr('hello Tbase','e');
 regexp_instr
--------------
 2
(1 row)
#从第7个开始搜索
postgres=# select regexp_instr('abc123abc156', '[1-2]+', 7)  from  dual;
 regexp_instr
--------------
 10
(1 row)
```

regexp_replace

regexp_replace(1,2,3,4,5,6)

语法说明：

1：字段

2：替换的字段

3：替换成什么

4：起始位置（默认从1开始）

5：替换的次数（0是无限次）

6：不区分大小写。

```
postgres=# select regexp_replace('tbase_Tbase','s','ee',1,1) from dual;
regexp_replace
----------------
 tbaeee_Tbase
(1 row)
postgres=#
```

nchr

给出一个数字代码，返回其对应字符。

```
postgres=# select NCHR(116) from dual;
nchr
------
 t
(1 row)
```

nls_upper

将字符转换为大写。

```
postgres=# select NLS_UPPER('tbase','nls_sort= SCHINESE_PINYIN_M')  from dual;
nls_upper
-----------
 TBASE
(1 row)
```

返回数值的字符函数
部分返回数值的字符函数示例
ASCIISTR

ASCIISTR函数，参数是一个字符串，如果这个字符在ASCII码表中有，则转成ASCII表中的字符。如果没有，则转成\xxxx格式，xxxx是UTF8的编码。

```
create or replace function asciistr(str text) returns text as
$$
declare
    mid text;
    res text:='';
begin
    foreach mid in array regexp_split_to_array(str, '')
    loop
        if ascii(mid)<256 then
            res := res || mid;
        else
            res := res || convert_to(mid,'UTF8')::text;
        end if;
    end loop;
    return res;
end;
$$
language plpgsql strict;
```

LENGTHB

返回字符长度。

```
postgres=# select LENGTHB('测试') from dual;
 lengthb
---------
       6
(1 row)
 postgres=# select LENGTH('测试') from dual;
 length
--------
      2
(1 row)
```

length

获取字符长度。

```
postgres=# select length(1);
length
--------
      1
(1 row)
postgres=# select length('tbase');
length
--------
      5
(1 row)
postgres=# select length('阿弟');
length
--------
      2
(1 row)
postgres=# select length(12.12::numeric(10,2));
length
--------
      5
(1 row)
```

instr

instr函数返回要截取的字符串在源字符串中的位置。

```
postgres=# select instr('helloworld','l') from dual;
instr
-------
     3
(1 row)
postgres=#
```

regexp_count

REGEXP_COUNT 返回pattern 在source_char 串中出现的次数。

```
postgres=# select REGEXP_COUNT('tbase_TBase','se') from DUAL;
regexp_count
--------------
           2
(1 row)
```

日期时间函数
部分日期时间函数示例
NUMTODSINTERVAL

numtodsinterval(,),x是一个数字,c是一个字符串，表明x的单位,这个函数把x转为interval day to second数据类型。

```
postgres=# select sysdate,sysdate+numtodsinterval(2,'hour') as res from dual;
    orcl_sysdate     |         res
---------------------+---------------------
 2020-08-02 21:55:35 | 2020-08-02 23:55:35
(1 row)
 postgres=#
```

DBTIMEZONE

```
postgres=# select DBTIMEZONE from dual;
 dbtimezone
------------
 08:00:00
(1 row)
postgres=#
```

MONTHS_BETWEEN

MONTHS_BETWEEN(DATE1,DATE2) #函数返回两个日期之间的月份数

```
postgres=# select months_between(to_date('20210331', 'yyyymmdd'), to_date('20200131', 'yyyymmd
d')) as months from dual;
 months
--------
     14
(1 row)
postgres=#
```

LAST_DAY

LAST_DAY函数返回指定日期对应月份的最后一天。

```
postgres=#  SELECT last_day('2020-05-01') FROM dual;
       last_day
------------------------
 2020-05-31 00:00:00+08
(1 row)
```

ADD_MONTHS

ADD_MONTHS(x,y)

x值为日期，y值为数量, 用于计算某个日期向前或者向后y个月后的时间。

```
postgres=#  select add_months(sysdate,1) from dual;
    add_months
---------------------
 2020-09-04 16:08:11
(1 row)
postgres=#  select add_months(sysdate,-1) from dual;
    add_months
---------------------
 2020-07-04 16:08:15
(1 row)
postgres=#
```

比较函数

函数名	功能描述
REGEXP_LIKE	比较字符串和正则表达式，返回t or f。
GREATEST	返回一个或多个表达式列表中的最大值。
LEAST	返回一个或多个表达式列表中的最小值。

转换函数
部分转换函数示例
to_number

将 expr 转换为数值数据类型的值。
示例：
1、to_number(?,'xxxx')：将指定的16进制数转化成10进制数显示

```
SELECT to_number(substr(dbms_session.unique_session_id,1,4),'xxxx') FROM dual;
```

> 注意：
>
> 支持字符串类型的16进制数转10进制数，不支持数字类型的16进制数转10进制数。

to_clob

转换字符为clob类型。

```
postgres=#  select to_clob('tbase') from dual;
to_clob
---------
 tbase
(1 row)
```

ROWIDTOCHAR

转换rowid值为varchar2类型。

```
postgres=# \d+ t_rowid
                            Table "public.t_rowid"
 Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------+-------------
 f1     | integer |           | not null |         | plain   |              |
 f2     | integer |           |          |         | plain   |              |
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES
postgres=# SELECT ROWIDTOCHAR(rowid),rowid from t_rowid;
     rowidtochar      |        rowid
----------------------+----------------------
 XPK3fw==AAAAAA==AQA= | XPK3fw==AAAAAA==AQA=
(1 row)
postgres=#
```

CHARTOROWID

CHARTOROWID(c1)

转换varchar2类型为rowid值,c1,字符串，长度为20的字符串，字符串必须符合rowid格式，ROWID。

```
postgres=# select CHARTOROWID('AAAFd1AAFAAAABSACCAA') a1 from dual;
         a1
----------------------
 AAAFdw==FAAAAA==CCA=
(1 row)
postgres=#
```

XML函数
部分XML函数示例
extractvalue

xml文本解释

```
postgres=# SELECT extractvalue(XMLTYPE('
<AAA>
<BBB>
<BBB>
<BBB>
</AAA>
'),'/AAA/BBB[2]') FROM dual;
extractvalue
--------------
 好转
(1 row)
```

extract

extract(xmltype类型,节点)
create table xmlexample(ID varchar(100),name varchar(20),data xmltype);
insert into xmlexample(id,name,data) values('xxxxxxxxxxxxxxx','my document','<?xml version="1.0" enc
oding="UTF-8" ?>
<collection xmlns="">
<record>
<leader>-----nam0-22-----^^^450-</leader>
<datafield tag="200" ind1="1" ind2=" ">

</datafield>
<datafield tag="209" ind1=" " ind2=" ">

<subfield code="c">10001</subfield>
<subfield code="d">2005-07-09</subfield>
</datafield>
<datafield tag="610" ind1="0" ind2=" ">

</datafield>
</record>
</collection>'::xmltype);

postgres=# select id,name, extract(x.data,'/collection/record/datafield/subfield') as A from xmlexample
x;
id | name |
a

----------+----------
+----------

---
xxxxxxxxxxxxxxx | my document |
d code="b">计算机
de="a">笔记本
(1 row)

编码和解码函数
部分编码和解码函数示例
vsize

返回传入参数的字节数。

```
postgres=# select vsize('TBase合理');
vsize
-------
    11
(1 row)
#一个汉字三个字节
postgres=# select vsize(1);
vsize
-------
     4
(1 row)
postgres=# select vsize(10000);
vsize
-------
     4
(1 row)
#整形是4个节点
```

NULL相关函数

部分NULL相关函数示例

lnnvl

传入表达式为true 返回false；传入为false 返回true。

```
postgres=# create table t_lnnvl(f1 integer, f2 integer);
CREATE TABLE
postgres=#
postgres=# insert into t_lnnvl values(1,1);
INSERT 0 1
postgres=#
postgres=# insert into t_lnnvl values(1,2);
INSERT 0 1
postgres=#
postgres=# insert into t_lnnvl values(1,3);
INSERT 0 1
postgres=#
postgres=# insert into t_lnnvl values(1,4);
INSERT 0 1
postgres=#
postgres=# insert into t_lnnvl values(1,null);
INSERT 0 1
postgres=# select * from t_lnnvl where lnnvl(f2>2);
 f1 | f2
----+----
  1 |  1
  1 |  2
  1 |
(1 rows)
```

nvl2

NVL2(E1, E2, E3)

如果E1为NULL，则函数返回E3，若E1不为null，则返回E2。

```
postgres=#  select NVL2('tbase', 'tbase1'::text, 'tbase2'::text) from dual;
nvl2
--------
 tbase1
(1 row)
postgres=#
postgres=#  select NVL2(NULL, 'tbase1'::text, 'tbase2'::text) from dual;
nvl2
--------
 tbase2
(1 row)
```

环境和标识符函数
说明以及示例

函数XS_SYS_CONTEXT

参数名	返回值
CREATED_BY	创建当前应用程序会话的所有者。
CREATE_TIME	当前应用程序会话的创建时间。
COOKIE	作为参数传递的安全会话 cookie，可用于在以后的调用中识别新创建的 Real Application Security 应用程序会话，直到更改 cookie 值或会话被销毁。
CURRENT_XS_USER	权限当前处于活动状态的 Real Application Security 会话应用程序用户的名称。
CURRENT_XS_USER_GUID	权限当前处于活动状态的 Real Application Security 会话应用程序用户的标识符。
INACTIVITY_TIMEOUT	当前应用程序会话的指定非活动超时值（以分钟为单位）。
LAST_ACCESS_TIME	会话应用程序用户最后一次访问会话的时间。
LAST_AUTHENTICATION_TIME	上次对会话应用程序用户进行身份验证的时间。
LAST_UPDATED_BY	上次更新应用程序会话的时间。
PROXY_GUID	代表 SESSION_XS_USER 打开当前会话的 Real Application Security 会话应用程序用户的标识符。
SESSION_ID	应用程序会话的会话标识符。
SESSION_SIZE	应用程序会话的大小（以字节为单位）。
SESSION_XS_USER	登录时 Real Application Security 会话应用程序用户的名称。
SESSION_XS_USER_GUID	登录时 Real Application Security 会话应用程序用户的标识符。
USERNAME	会话应用程序用户名。
USER_ID	会话应用程序用户的标识符。

示例：

```
SELECT DECODE(USER, 'XS$NULL', XS_SYS_CONTEXT('XS$SESSION','USERNAME'), USER) FROM DUA
L ;
```

函数USERENV

参数名	返回值
SCHEMAID	返回当前默认的 Schema ID。
LANG	返回 ISO 缩写的语言名称，比 LANGUAGE 参数更短的格式。
LANGUAGE	返回当前会话的语言、地域和字符集。
SID	返回会话 ID。

> 注意：
>
> language参数返回值，Oracle为全称组合，TDSQL PG是缩写组合

示例：

```
select 'USER_LANGUAGE' as Lang,userenv('language') from dual;
```

函数UID

示例

```
SELECT USER, UID FROM DUAL;
```

聚合函数

函数名称	功能描述
APPROX_COUNT_DISTINCT	返回包含expr的不同值的近似行数为COUNT (DISTINCT expr)函数的替代方法
AVG	返回expr平均值
COUNT	返回查询返回的行数，可以用作聚合或分析函数
KEEP	以某种规则排序后，取同一个分组下的第一个或最后一个的值
LISTAGG	根据查询条件中的一个或多个表达式将查询结果集划分为组，可用于列转行
MAX	返回expr最大值。可用作聚合或分析函数
MIN	返回expr最小值。可用作聚合或分析函数
ROLLUP	用于分组统计，返回小组合计以及总计
STDDEV	返回expr的样本标准差，即一组数字。可用作聚合和分析函数
SUM	返回expr值的总和。可用作聚合或分析函数。
VARIANCE	返回expr的方差。可用作聚合或分析函数。
WMSYS.WM_CONCAT/ WM_CONCAT	可以把列值以'，'分割开来，并显示成一行，用于列转行

分析函数

函数名称	功能描述
AVG	返回expr平均值
COUNT	返回查询返回的行数，可以用作聚合或分析函数
KEEP	以某种规则排序后，取同一个分组下的第一个或最后一个的值
SUM	返回expr值的总和。可用作聚合或分析函数。
MAX	返回expr最大值。可用作聚合或分析函数
MIN	返回expr最小值。可用作聚合或分析函数
LISTAGG	根据查询条件中的一个或多个表达式将查询结果集划分为组，可用于列转行
STDDEV	返回expr的样本标准差，即一组数字。可用作聚合和分析函数
VARIANCE	返回expr的方差。可用作聚合或分析函数。
RANK	计算一组值中的值的等级。返回类型为NUMBER
LEAD	在查询中取出同一字段的后 offset 行的数据作为独立的列存在表中。可以代替表的自联接
LAG	查询中取出同一字段的前 offset 行的数据作为独立的列存在表中。可以代替表的自联接
FIRST_VALUE	返回有序值集中的第一个值。如果零，则返回NULL ，除非指定IGNORE NULLS
LAST_VALUE	返回有序值集中的最后一个值
NTH_VALUE	返回analytic_clause定义的窗口中第n行的measure_expr值
CUME_DIST	计算一组值中值的累积分布。返回的值范围> 0到
DENSE_RANK	计算有序行组中行的排序位置
NTILE	将有序数据集划分为若干个组，并为每一行分配适当的组号。组编号为 1 到 expr。
PERCENT_RANK	类似于CUME_DIST （累积分布）函数。 PERCENT_RANK返回的值范围是0到1（包括0和1）
RATIO_TO_REPORT	计算值与一组值之和的比率

函数名称	功能描述
ROW_NUMBER	为每一行（分区中的每一行或查询返回的每一行）分配一个唯一的编号
WMSYS.WM_CONCAT/ WM_CONCAT	可以把列值以'，'分割开来，并显示成一行，用于列转行

二进制操作函数

empty_clob

初始化CLOB字段。

```
postgres=# select empty_clob();
empty_clob
------------
(1 row)
postgres=# create table t1 (f1 int,f2 clob);
CREATE TABLE
postgres=# insert into t1(f1,f2) values (1,empty_clob());
INSERT 0 1
postgres=#
```

统计函数
listagg

listagg (filedname,',') WITHIN GROUP (ORDER BY filedname)

行转列函数

```
postgres=# create table person
 (
 deptno varchar2(10),
ename  varchar(20)
);
CREATE TABLE
postgres=# insert into person values('20','aaa');
INSERT 0 1
postgres=#
postgres=# insert into person values('20','bbb');
INSERT 0 1
postgres=#
postgres=# insert into person values('20','ccc');
INSERT 0 1
postgres=#
postgres=# insert into person values('21','ddd');
INSERT 0 1
postgres=#
postgres=# insert into person values('21','eee');
INSERT 0 1
postgres=# select
postgres-#     deptno,
postgres-#     listagg (ename,',') WITHIN GROUP (ORDER BY ENAME)
postgres-# from
postgres-#     person
postgres-# group by
postgres-#     deptno ;
 deptno |   listagg
--------+-------------
 20     | aaa,bbb,ccc
 21     | ddd,eee
(2 rows)
```

其他函数的使用

sys_guid()

sys_guid 生成并返回一个由 16 个字节组成的全局唯一标识符（RAW 值）。 生成的标识符由主机标识符、调用函数的进程或线程的进程或线程标识符以及该进程或线程的非重复值（字节序列）组成。

```
postgres=# select sys_guid();
            sys_guid
---------------------------------
 B975096BCDCCBA209F74A69DDA83DE7B
(1 行记录)
```

数据表准备

```
drop table if  exists bills ;
create table bills
(
  id serial not null,
  goodsdesc text not null,
  beginunit text not null,
  begincity text not null,
  pubtime timestamp not null,
  amount float8 not null default 0,
  primary key (id)
);
COMMENT ON TABLE bills is '运单记录';
COMMENT ON COLUMN bills.id IS 'id号';
COMMENT ON COLUMN bills.goodsdesc IS '货物名称';
COMMENT ON COLUMN bills.beginunit IS '启运城市';
COMMENT ON COLUMN bills.begincity IS '启运地区';
COMMENT ON COLUMN bills.pubtime IS '发布时间';
COMMENT ON COLUMN bills.amount IS '运费';
INSERT INTO bills(id,goodsdesc,beginunit,begincity,pubtime,amount)
VALUES(default,'衣服','城市1','地区1','2015-10-05 09:32:01',ROUND((random()*10000)::NUMERIC,2));
INSERT INTO bills(id,goodsdesc,beginunit,begincity,pubtime,amount)
VALUES(default,'建筑设备','城市2','地区2','2015-10-05 07:21:22',ROUND((random()*10000)::NUMERIC,2));
INSERT INTO bills(id,goodsdesc,beginunit,begincity,pubtime,amount)
VALUES(default,'设备','城市2','地区2','2015-10-05 11:21:54',ROUND((random()*10000)::NUMERIC,2));
INSERT INTO bills(id,goodsdesc,beginunit,begincity,pubtime,amount)
VALUES(default,'普货','城市2','地区2','2015-10-05 15:19:17',ROUND((random()*10000)::NUMERIC,2));
INSERT INTO bills(id,goodsdesc,beginunit,begincity,pubtime,amount)
VALUES(default,'5 0铲车，后八轮翻斗车','城市3','地区3','2015-10-05 07:53:13',ROUND((random()*1000
0)::NUMERIC,2));
INSERT INTO bills(id,goodsdesc,beginunit,begincity,pubtime,amount)
VALUES(default,'鲜香菇2000斤','城市3','地区3','2015-10-05 10:38:29',ROUND((random()*10000)::NUMERI
C,2));
INSERT INTO bills(id,goodsdesc,beginunit,begincity,pubtime,amount)
VALUES(default,'旋挖附件38吨','城市3','地区3','2015-10-05 10:48:38',ROUND((random()*10000)::NUMERI
C,2));
INSERT INTO bills(id,goodsdesc,beginunit,begincity,pubtime,amount)
VALUES(default,'旋挖附件35吨','城市3','地区3','2015-10-05 10:48:38',ROUND((random()*10000)::NUMERI
C,2));
INSERT INTO bills(id,goodsdesc,beginunit,begincity,pubtime,amount)
VALUES(default,'旋挖附件39吨','城市3','地区3','2015-10-05 11:38:38',ROUND((random()*10000)::NUMERI
C,2));
INSERT INTO bills(id,goodsdesc,beginunit,begincity,pubtime,amount)
VALUES(default,'设备','城市4','城市4','2015-10-05 07:59:35',ROUND((random()*10000)::NUMERIC,2));
```

```
INSERT INTO bills(id,goodsdesc,beginunit,begincity,pubtime,amount)
VALUES(default,'普货40吨需13米半挂一辆','城市4','城市4','2015-10-05 08:13:59',ROUND((random()*1000
0)::NUMERIC,2));
```

row_number() --返回行号，不分组

```
postgres=# select row_number() over(),* from bills limit 2;
 row_number | id | goodsdesc | beginunit | begincity |       pubtime       | amount
------------+----+-----------+-----------+-----------+---------------------+---------
          1 |  1 | 衣服      | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
          2 |  2 | 建筑设备  | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
(2 rows)
postgres=# select row_number() over(),* from bills limit 2 offset 2;
 row_number | id |       goodsdesc       | beginunit | begincity |       pubtime       | amount
------------+----+-----------------------+-----------+-----------+---------------------+---------
          3 |  6 | 5 0铲车，后八轮翻斗车 | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
          4 |  8 | 旋挖附件38吨          | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
(2 rows)
```

row_number() --返回行号，按amount排序

```
postgres=# select row_number() over(order by amount),* from bills;
 row_number | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
------------+----+------------------------+-----------+-----------+---------------------+---------
          1 | 11 | 设备                   | 城市4     | 城市4     | 2015-10-05 07:59:35 |  971.54
          2 |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
          3 |  4 | 普货                   | 城市2     | 地区2     | 2015-10-05 15:19:17 | 1316.27
          4 |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
          5 |  2 | 建筑设备               | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
          6 |  7 | 鲜香菇2000斤           | 城市3     | 地区3     | 2015-10-05 10:38:29 | 4182.68
          7 |  8 | 旋挖附件38吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
          8 | 10 | 旋挖附件39吨           | 城市3     | 地区3     | 2015-10-05 11:38:38 |  8290.5
          9 |  3 | 设备                   | 城市2     | 地区2     | 2015-10-05 11:21:54 | 8771.11
         10 |  9 | 旋挖附件35吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 9621.37
         11 |  5 | 普货40吨需13米半挂一辆 | 城市4     | 城市4     | 2015-10-05 08:13:59 | 9886.15
(11 rows)
```

row_number() --返回行号，按begincity分组，pubtime排序

```
postgres=# select row_number() over(partition by begincity order by pubtime),* from bills;
 row_number | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
------------+----+------------------------+-----------+-----------+---------------------+---------
          1 |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
          1 |  2 | 建筑设备               | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
          2 |  3 | 设备                   | 城市2     | 地区2     | 2015-10-05 11:21:54 | 8771.11
          3 |  4 | 普货                   | 城市2     | 地区2     | 2015-10-05 15:19:17 | 1316.27
          1 |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
          2 |  7 | 鲜香菇2000斤           | 城市3     | 地区3     | 2015-10-05 10:38:29 | 4182.68
          3 |  8 | 旋挖附件38吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
          4 |  9 | 旋挖附件35吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 9621.37
          5 | 10 | 旋挖附件39吨           | 城市3     | 地区3     | 2015-10-05 11:38:38 |  8290.5
          1 | 11 | 设备                   | 城市4     | 城市4     | 2015-10-05 07:59:35 |  971.54
          2 |  5 | 普货40吨需13米半挂一辆 | 城市4     | 城市4     | 2015-10-05 08:13:59 | 9886.15
(11 rows)
```

rank()--返回行号,对比值重复时行号重复并间断，即返回1,2,2,4...

```
postgres=# select rank() over(partition by begincity order by pubtime),* from bills;
 rank | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
------+----+------------------------+-----------+-----------+---------------------+---------
    1 |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
    1 |  2 | 建筑设备               | 城市2     | 地区1     | 2015-10-05 07:21:22 | 2022.31
    2 |  3 | 设备                   | 城市2     | 地区1     | 2015-10-05 11:21:54 | 8771.11
    3 |  4 | 普货                   | 城市2     | 地区1     | 2015-10-05 15:19:17 | 1316.27
    1 |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区1     | 2015-10-05 07:53:13 |  1030.9
    2 |  7 | 鲜香菇2000斤           | 城市3     | 地区1     | 2015-10-05 10:38:29 | 4182.68
    3 |  8 | 旋挖附件38吨           | 城市3     | 地区1     | 2015-10-05 10:48:38 | 5365.04
    3 |  9 | 旋挖附件35吨           | 城市3     | 地区1     | 2015-10-05 10:48:38 | 9621.37
    5 | 10 | 旋挖附件39吨           | 城市3     | 地区1     | 2015-10-05 11:38:38 |  8290.5
    1 | 11 | 设备                   | 城市4     | 城市4     | 2015-10-05 07:59:35 |  971.54
    2 |  5 | 普货40吨需13米半挂一辆 | 城市4     | 城市4     | 2015-10-05 08:13:59 | 9886.15
(11 rows)
```

dense_rank() --返回行号,对比值重复时行号重复但不间断，即返回1,2,2,3...

```
postgres=# select dense_rank() over(partition by begincity order by pubtime),* from bills;
 dense_rank | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
------------+----+------------------------+-----------+-----------+---------------------+---------
          1 |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
          1 |  2 | 建筑设备               | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
          2 |  3 | 设备                   | 城市2     | 地区2     | 2015-10-05 11:21:54 | 8771.11
          3 |  4 | 普货                   | 城市2     | 地区2     | 2015-10-05 15:19:17 | 1316.27
          1 |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
          2 |  7 | 鲜香菇2000斤           | 城市3     | 地区3     | 2015-10-05 10:38:29 | 4182.68
          3 |  8 | 旋挖附件38吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
          4 |  9 | 旋挖附件35吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 9621.37
          5 | 10 | 旋挖附件39吨           | 城市3     | 地区3     | 2015-10-05 11:38:38 |  8290.5
          1 | 11 | 设备                   | 城市4     | 城市4     | 2015-10-05 07:59:35 |  971.54
          2 |  5 | 普货40吨需13米半挂一辆 | 城市4     | 城市4     | 2015-10-05 08:13:59 | 9886.15
(11 rows)
```

percent_rank()从当前开始，计算在分组中的比例 (行号-1)*(1/(总记录数-1))

```
postgres=# select percent_rank() over(partition by begincity order by id),* from bills;
 percent_rank | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
--------------+----+------------------------+-----------+-----------+---------------------+---------
            1 |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
            1 |  2 | 建筑设备               | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
            2 |  3 | 设备                   | 城市2     | 地区2     | 2015-10-05 11:21:54 | 8771.11
            3 |  4 | 普货                   | 城市2     | 地区2     | 2015-10-05 15:19:17 | 1316.27
            1 |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
            2 |  7 | 鲜香菇2000斤           | 城市3     | 地区3     | 2015-10-05 10:38:29 | 4182.68
            3 |  8 | 旋挖附件38吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
            4 |  9 | 旋挖附件35吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 9621.37
            5 | 10 | 旋挖附件39吨           | 城市3     | 地区3     | 2015-10-05 11:38:38 |  8290.5
            2 |  5 | 普货40吨需13米半挂一辆 | 城市4     | 城市4     | 2015-10-05 08:13:59 | 9886.15
            1 | 11 | 设备                   | 城市4     | 城市4     | 2015-10-05 07:59:35 |  971.54
(11 rows)
```

cume_dist() --返回行数除以记录数值

```
postgres=# select ROUND((cume_dist() over(partition by begincity order by id))::NUMERIC,2) AS cume
_dist,* from bills;
 cume_dist | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
-----------+----+------------------------+-----------+-----------+---------------------+---------
      1.00 |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
      0.33 |  2 | 建筑设备               | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
      0.67 |  3 | 设备                   | 城市2     | 地区2     | 2015-10-05 11:21:54 | 8771.11
      1.00 |  4 | 普货                   | 城市2     | 地区2     | 2015-10-05 15:19:17 | 1316.27
      0.20 |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
      0.40 |  7 | 鲜香菇2000斤           | 城市3     | 地区3     | 2015-10-05 10:38:29 | 4182.68
      0.60 |  8 | 旋挖附件38吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
      0.80 |  9 | 旋挖附件35吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 9621.37
      1.00 | 10 | 旋挖附件39吨           | 城市3     | 地区3     | 2015-10-05 11:38:38 |  8290.5
      0.50 |  5 | 普货40吨需13米半挂一辆 | 城市4     | 城市4     | 2015-10-05 08:13:59 | 9886.15
      1.00 | 11 | 设备                   | 城市4     | 城市4     | 2015-10-05 07:59:35 |  971.54
(11 rows)
```

ntile(分组数量)--让所有记录尽可以的均匀分布

```
postgres=# select ntile(2) over(partition by begincity order by id),* from bills;
 ntile | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
-------+----+------------------------+-----------+-----------+---------------------+---------
     1 |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
     1 |  2 | 建筑设备               | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
     2 |  3 | 设备                   | 城市2     | 地区2     | 2015-10-05 11:21:54 | 8771.11
     3 |  4 | 普货                   | 城市2     | 地区2     | 2015-10-05 15:19:17 | 1316.27
     1 |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
     2 |  7 | 鲜香菇2000斤           | 城市3     | 地区3     | 2015-10-05 10:38:29 | 4182.68
     3 |  8 | 旋挖附件38吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
     4 |  9 | 旋挖附件35吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 9621.37
     5 | 10 | 旋挖附件39吨           | 城市3     | 地区3     | 2015-10-05 11:38:38 |  8290.5
     2 |  5 | 普货40吨需13米半挂一辆 | 城市4     | 城市4     | 2015-10-05 08:13:59 | 9886.15
     1 | 11 | 设备                   | 城市4     | 城市4     | 2015-10-05 07:59:35 |  971.54
(11 rows)

postgres=# select ntile(3) over(partition by begincity order by id),* from bills;
 ntile | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
-------+----+------------------------+-----------+-----------+---------------------+---------
     1 |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
     1 |  2 | 建筑设备               | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
     2 |  3 | 设备                   | 城市2     | 地区2     | 2015-10-05 11:21:54 | 8771.11
     3 |  4 | 普货                   | 城市2     | 地区2     | 2015-10-05 15:19:17 | 1316.27
     1 |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
     2 |  7 | 鲜香菇2000斤           | 城市3     | 地区3     | 2015-10-05 10:38:29 | 4182.68
     3 |  8 | 旋挖附件38吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
     4 |  9 | 旋挖附件35吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 9621.37
     5 | 10 | 旋挖附件39吨           | 城市3     | 地区3     | 2015-10-05 11:38:38 |  8290.5
     2 |  5 | 普货40吨需13米半挂一辆 | 城市4     | 城市4     | 2015-10-05 08:13:59 | 9886.15
     1 | 11 | 设备                   | 城市4     | 城市4     | 2015-10-05 07:59:35 |  971.54
(11 rows)
```

lag(value any [, offset integer [, default any]])--返回偏移量值

#offset integer是偏移值，正数时取前值，负数时取后值，没有取到值时用default代替

```
postgres=# select lag(amount,1,null) over(partition by begincity order by id),* from bills;
   lag   | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
---------+----+------------------------+-----------+-----------+---------------------+---------
         |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
         |  2 | 建筑设备               | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
 2022.31 |  3 | 设备                   | 城市2     | 地区2     | 2015-10-05 11:21:54 | 8771.11
 8771.11 |  4 | 普货                   | 城市2     | 地区2     | 2015-10-05 15:19:17 | 1316.27
         |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
  1030.9 |  7 | 鲜香菇2000斤           | 城市3     | 地区3     | 2015-10-05 10:38:29 | 4182.68
 4182.68 |  8 | 旋挖附件38吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
 5365.04 |  9 | 旋挖附件35吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 9621.37
 9621.37 | 10 | 旋挖附件39吨           | 城市3     | 地区3     | 2015-10-05 11:38:38 |  8290.5
         |  5 | 普货40吨需13米半挂一辆 | 城市4     | 地区4     | 2015-10-05 08:13:59 | 9886.15
 9886.15 | 11 | 设备                   | 城市4     | 地区4     | 2015-10-05 07:59:35 |  971.54
(11 rows)
postgres=# select lag(amount,2,0::float8) over(partition by begincity order by id),* from bills;
   lag   | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
---------+----+------------------------+-----------+-----------+---------------------+---------
       0 |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
       0 |  2 | 建筑设备               | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
       0 |  3 | 设备                   | 城市2     | 地区2     | 2015-10-05 11:21:54 | 8771.11
 2022.31 |  4 | 普货                   | 城市2     | 地区2     | 2015-10-05 15:19:17 | 1316.27
       0 |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
       0 |  7 | 鲜香菇2000斤           | 城市3     | 地区3     | 2015-10-05 10:38:29 | 4182.68
  1030.9 |  8 | 旋挖附件38吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
 4182.68 |  9 | 旋挖附件35吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 9621.37
 5365.04 | 10 | 旋挖附件39吨           | 城市3     | 地区3     | 2015-10-05 11:38:38 |  8290.5
       0 |  5 | 普货40吨需13米半挂一辆 | 城市4     | 地区4     | 2015-10-05 08:13:59 | 9886.15
       0 | 11 | 设备                   | 城市4     | 地区4     | 2015-10-05 07:59:35 |  971.54
(11 rows)
postgres=# select lag(amount,-2,0::float8) over(partition by begincity order by id),* from bills;
   lag   | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
---------+----+------------------------+-----------+-----------+---------------------+---------
       0 |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
 1316.27 |  2 | 建筑设备               | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
       0 |  3 | 设备                   | 城市2     | 地区2     | 2015-10-05 11:21:54 | 8771.11
       0 |  4 | 普货                   | 城市2     | 地区2     | 2015-10-05 15:19:17 | 1316.27
 5365.04 |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
 9621.37 |  7 | 鲜香菇2000斤           | 城市3     | 地区3     | 2015-10-05 10:38:29 | 4182.68
  8290.5 |  8 | 旋挖附件38吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
```

```
      0 |  9 | 旋挖附件35吨          | 城市3    | 地区3  | 2015-10-05 10:48:38 | 9621.37
      0 | 10 | 旋挖附件39吨          | 城市3    | 地区3  | 2015-10-05 11:38:38 |  8290.5
      0 |  5 | 普货40吨需13米半挂一辆 | 城市4    | 地区4  | 2015-10-05 08:13:59 | 9886.15
      0 | 11 | 设备                  | 城市4    | 地区4  | 2015-10-05 07:59:35 |  971.54
(11 rows)
```

lead(value any [,offset integer [, default any]])--返回偏移量值

#offset integer是偏移值，正数时取后值，负数时取前值，没有取到值时用default代替

```
postgres=# select lead(amount,2,null) over(partition by begincity order by id),* from bills;
  lead   | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
---------+----+------------------------+-----------+-----------+---------------------+---------
       0 |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
 1316.27 |  2 | 建筑设备               | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
       0 |  3 | 设备                   | 城市2     | 地区2     | 2015-10-05 11:21:54 | 8771.11
       0 |  4 | 普货                   | 城市2     | 地区2     | 2015-10-05 15:19:17 | 1316.27
 5365.04 |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
 9621.37 |  7 | 鲜香菇2000斤           | 城市3     | 地区3     | 2015-10-05 10:38:29 | 4182.68
  8290.5 |  8 | 旋挖附件38吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
       0 |  9 | 旋挖附件35吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 9621.37
       0 | 10 | 旋挖附件39吨           | 城市3     | 地区3     | 2015-10-05 11:38:38 |  8290.5
       0 |  5 | 普货40吨需13米半挂一辆 | 城市4     | 地区4     | 2015-10-05 08:13:59 | 9886.15
       0 | 11 | 设备                   | 城市4     | 地区4     | 2015-10-05 07:59:35 |  971.54
(11 rows)
postgres=# select lead(amount,-2,null) over(partition by begincity order by id),* from bills;
  lead   | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
---------+----+------------------------+-----------+-----------+---------------------+---------
       0 |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
       0 |  2 | 建筑设备               | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
       0 |  3 | 设备                   | 城市2     | 地区2     | 2015-10-05 11:21:54 | 8771.11
 2022.31 |  4 | 普货                   | 城市2     | 地区2     | 2015-10-05 15:19:17 | 1316.27
       0 |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
       0 |  7 | 鲜香菇2000斤           | 城市3     | 地区3     | 2015-10-05 10:38:29 | 4182.68
  1030.9 |  8 | 旋挖附件38吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
 4182.68 |  9 | 旋挖附件35吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 9621.37
 5365.04 | 10 | 旋挖附件39吨           | 城市3     | 地区3     | 2015-10-05 11:38:38 |  8290.5
       0 |  5 | 普货40吨需13米半挂一辆 | 城市4     | 地区4     | 2015-10-05 08:13:59 | 9886.15
       0 | 11 | 设备                   | 城市4     | 地区4     | 2015-10-05 07:59:35 |  971.54
(11 rows)
```

first_value(value any)返回第一值

```
postgres=# select first_value(amount) over(partition by begincity order by  id),* from bills;
 first_value | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
-------------+----+------------------------+-----------+-----------+---------------------+---------
     1915.86 |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
     2022.31 |  2 | 建筑设备               | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
     2022.31 |  3 | 设备                   | 城市2     | 地区2     | 2015-10-05 11:21:54 | 8771.11
     2022.31 |  4 | 普货                   | 城市2     | 地区2     | 2015-10-05 15:19:17 | 1316.27
      1030.9 |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
      1030.9 |  7 | 鲜香菇2000斤           | 城市3     | 地区3     | 2015-10-05 10:38:29 | 4182.68
      1030.9 |  8 | 旋挖附件38吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
      1030.9 |  9 | 旋挖附件35吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 9621.37
      1030.9 | 10 | 旋挖附件39吨           | 城市3     | 地区3     | 2015-10-05 11:38:38 |  8290.5
     9886.15 |  5 | 普货40吨需13米半挂一辆 | 城市4     | 地区4     | 2015-10-05 08:13:59 | 9886.15
     9886.15 | 11 | 设备                   | 城市4     | 地区4     | 2015-10-05 07:59:35 |  971.54
(11 rows)
```

last_value(value any)返回最后值

```
postgres=# select last_value(amount) over(partition by begincity order by pubtime),* FROM bills;
 last_value | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
------------+----+------------------------+-----------+-----------+---------------------+---------
    1915.86 |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
    2022.31 |  2 | 建筑设备               | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
    8771.11 |  3 | 设备                   | 城市2     | 地区2     | 2015-10-05 11:21:54 | 8771.11
    1316.27 |  4 | 普货                   | 城市2     | 地区2     | 2015-10-05 15:19:17 | 1316.27
     1030.9 |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
    4182.68 |  7 | 鲜香菇2000斤           | 城市3     | 地区3     | 2015-10-05 10:38:29 | 4182.68
    9621.37 |  8 | 旋挖附件38吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
    9621.37 |  9 | 旋挖附件35吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 9621.37
     8290.5 | 10 | 旋挖附件39吨           | 城市3     | 地区3     | 2015-10-05 11:38:38 |  8290.5
     971.54 | 11 | 设备                   | 城市4     | 城市4     | 2015-10-05 07:59:35 |  971.54
    9886.15 |  5 | 普货40吨需13米半挂一辆 | 城市4     | 城市4     | 2015-10-05 08:13:59 | 9886.15
(11 rows)

postgres=# select last_value(amount) over(partition by begincity),* FROM bills;
 last_value | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
------------+----+------------------------+-----------+-----------+---------------------+---------
    1915.86 |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
    1316.27 |  2 | 建筑设备               | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
    1316.27 |  3 | 设备                   | 城市2     | 地区2     | 2015-10-05 11:21:54 | 8771.11
    1316.27 |  4 | 普货                   | 城市2     | 地区2     | 2015-10-05 15:19:17 | 1316.27
    9621.37 |  7 | 鲜香菇2000斤           | 城市3     | 地区3     | 2015-10-05 10:38:29 | 4182.68
    9621.37 | 10 | 旋挖附件39吨           | 城市3     | 地区3     | 2015-10-05 11:38:38 |  8290.5
    9621.37 |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
    9621.37 |  8 | 旋挖附件38吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
    9621.37 |  9 | 旋挖附件35吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 9621.37
     971.54 |  5 | 普货40吨需13米半挂一辆 | 城市4     | 城市4     | 2015-10-05 08:13:59 | 9886.15
     971.54 | 11 | 设备                   | 城市4     | 城市4     | 2015-10-05 07:59:35 |  971.54
(11 rows)
```

> 注意：
>
> 不要加上order by id，默认情况下，带了order by 参数会从分组的起始值开始一直叠加，直到当前值（不是当前记录）不同为止，当忽略order by 参数则是整个分组。下面通过修改分组的统计范围就可以实现order by参数取最后值 。

```
postgres=# select last_value(amount) over(partition by begincity order by id range between unbounde
d preceding and unbounded following),* FROM bills;
 last_value | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
------------+----+------------------------+-----------+-----------+---------------------+---------
```

```
   1915.86 |  1 | 衣服                | 城市1   | 地区1   | 2015-10-05 09:32:01 | 1915.86
   1316.27 |  2 | 建筑设备            | 城市2   | 地区2   | 2015-10-05 07:21:22 | 2022.31
   1316.27 |  3 | 设备                | 城市2   | 地区2   | 2015-10-05 11:21:54 | 8771.11
   1316.27 |  4 | 普货                | 城市2   | 地区2   | 2015-10-05 15:19:17 | 1316.27
    8290.5 |  6 | 5 0铲车，后八轮翻斗车 | 城市3   | 地区3   | 2015-10-05 07:53:13 |  1030.9
    8290.5 |  7 | 鲜香菇2000斤        | 城市3   | 地区3   | 2015-10-05 10:38:29 | 4182.68
    8290.5 |  8 | 旋挖附件38吨        | 城市3   | 地区3   | 2015-10-05 10:48:38 | 5365.04
    8290.5 |  9 | 旋挖附件35吨        | 城市3   | 地区3   | 2015-10-05 10:48:38 | 9621.37
    8290.5 | 10 | 旋挖附件39吨        | 城市3   | 地区3   | 2015-10-05 11:38:38 |  8290.5
    971.54 |  5 | 普货40吨需13米半挂一辆 | 城市4   | 城市4   | 2015-10-05 08:13:59 | 9886.15
    971.54 | 11 | 设备                | 城市4   | 城市4   | 2015-10-05 07:59:35 |  971.54
(11 rows)
```

nth_value(value any, nth integer)：返回窗口框架中的指定值

```
postgres=# select nth_value(amount,2) over(partition by begincity order by id),* from bills;
 nth_value | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
-----------+----+------------------------+-----------+-----------+---------------------+---------
           |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
           |  2 | 建筑设备               | 城市2     | 地区2     | 2015-10-05 07:21:22 | 2022.31
   8771.11 |  3 | 设备                   | 城市2     | 地区2     | 2015-10-05 11:21:54 | 8771.11
   8771.11 |  4 | 普货                   | 城市2     | 地区2     | 2015-10-05 15:19:17 | 1316.27
           |  6 | 5 0铲车，后八轮翻斗车  | 城市3     | 地区3     | 2015-10-05 07:53:13 |  1030.9
   4182.68 |  7 | 鲜香菇2000斤           | 城市3     | 地区3     | 2015-10-05 10:38:29 | 4182.68
   4182.68 |  8 | 旋挖附件38吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 5365.04
   4182.68 |  9 | 旋挖附件35吨           | 城市3     | 地区3     | 2015-10-05 10:48:38 | 9621.37
   4182.68 | 10 | 旋挖附件39吨           | 城市3     | 地区3     | 2015-10-05 11:38:38 |  8290.5
           |  5 | 普货40吨需13米半挂一辆 | 城市4     | 城市4     | 2015-10-05 08:13:59 | 9886.15
    971.54 | 11 | 设备                   | 城市4     | 城市4     | 2015-10-05 07:59:35 |  971.54
(11 rows)
```

统计各个城市的总运费及平均每单的运费

```
postgres=# select sum(amount) over(partition by begincity),avg(amount) over(partition by begincity),b
egincity,amount from bills;
   sum    |       avg        | begincity | amount
----------+------------------+-----------+---------
  1915.86 |          1915.86 | 地区1     | 1915.86
 12109.69 | 4036.56333333333 | 地区2     | 2022.31
 12109.69 | 4036.56333333333 | 地区2     | 8771.11
 12109.69 | 4036.56333333333 | 地区2     | 1316.27
 28490.49 |         5698.098 | 地区3     | 4182.68
 28490.49 |         5698.098 | 地区3     |  8290.5
 28490.49 |         5698.098 | 地区3     |  1030.9
 28490.49 |         5698.098 | 地区3     | 5365.04
 28490.49 |         5698.098 | 地区3     | 9621.37
 10857.69 |         5428.845 | 地区4     | 9886.15
 10857.69 |         5428.845 | 地区4     |  971.54
(11 rows)
```

窗口函数别名使用

```
postgres=# select sum(amount) over w,avg(amount) over w,begincity,amount from bills window w as (partition by begincity);
   sum    |       avg        | begincity | amount
----------+------------------+-----------+---------
  1915.86 |         1915.86 | 地区1     | 1915.86
 12109.69 | 4036.56333333333 | 地区2     | 2022.31
 12109.69 | 4036.56333333333 | 地区2     | 8771.11
 12109.69 | 4036.56333333333 | 地区2     | 1316.27
 28490.49 |         5698.098 | 地区3     | 4182.68
 28490.49 |         5698.098 | 地区3     |  8290.5
 28490.49 |         5698.098 | 地区3     |  1030.9
 28490.49 |         5698.098 | 地区3     | 5365.04
 28490.49 |         5698.098 | 地区3     | 9621.37
 10857.69 |         5428.845 | 地区4     | 9886.15
 10857.69 |         5428.845 | 地区4     |  971.54
(11 rows)
```

获取每个城市运费前两名订单

```
postgres=# select * from (select row_number() over(partition by begincity order by amount desc),* fro
m bills) where row_number<3;
 row_number | id |       goodsdesc        | beginunit | begincity |       pubtime       | amount
------------+----+------------------------+-----------+-----------+---------------------+---------
          1 |  1 | 衣服                   | 城市1     | 地区1     | 2015-10-05 09:32:01 | 1915.86
          1 |  3 | 设备                   | 城市2     | 地区2 2015-10-05 11:21:54 | 8771.11
          2 |  2 | 建筑设备               | 城市2     | 地区2 2015-10-05 07:21:22 | 2022.31
          1 |  9 | 旋挖附件35吨           | 城市3     | 地区3  | 2015-10-05 10:48:38 | 9621.37
          2 | 10 | 旋挖附件39吨           | 城市3     | 地区3  | 2015-10-05 11:38:38 |  8290.5
          1 |  5 | 普货40吨需13米半挂一辆 | 城市4     | 城市4     | 2015-10-05 08:13:59 | 9886.15
          2 | 11 | 设备                   | 城市4     | 城市4     | 2015-10-05 07:59:35 |  971.54
(7 rows)
```

连续百分率：返回一个对应于排序中指定分数的值

```
postgres=# select * from bills where begincity='地区1';
 id | goodsdesc | beginunit | begincity |       pubtime       | amount
----+-----------+-----------+-----------+---------------------+---------
 12 | 建筑设备  | 城市2     | 地区1     | 2015-10-05 07:21:22 | 2022.31
 13 | 设备      | 城市2     | 地区1     | 2015-10-05 11:21:54 | 8771.11
 14 | 普货      | 城市2     | 地区1     | 2015-10-05 15:19:17 | 1316.27
(3 rows)
postgres=# select
    min(amount),
    PERCENTILE_CONT(0) WITHIN GROUP(order by amount) AS RATE_0 ,
    PERCENTILE_CONT(0.25) WITHIN GROUP(order by amount) AS RATE_25,
    PERCENTILE_CONT(0.5) WITHIN GROUP(order by amount) AS RATE_50,
    PERCENTILE_CONT(0.75) WITHIN GROUP(order by amount) AS RATE_75,
    PERCENTILE_CONT(1) WITHIN GROUP(order by amount) AS RATE_100
from bills
where
    begincity='地区1'
;
   min   | rate_0  | rate_25 | rate_50 | rate_75 | rate_100
---------+---------+---------+---------+---------+----------
 1316.27 | 1316.27 | 1669.29 | 2022.31 | 5396.71 |  8771.11
(1 row)
postgres=#
```

伪列
ROWNUM

ROWNUM是Oracle中对结果集加的一个伪列，即先查到结果集之后再加上去的一个列 (强调：先要有结果集)。简单的说 ROWNUM是对符合条件结果的序列号。它总是从1开始排起的。
TDSQL PG支持ROWNUM伪列用法。

使用限制
使用ROWNUM时需要注意事项：

- ROWNUM是数据库从数据文件或缓冲区中读取数据的顺序，所以rownum>n（n>=1）、rownum between n(n>1) and m是得不到数据的。
- 不能使用ROWNUM做为字段名称。

语法
```
SELECT [ROWNUM,]column1,... FROM table1...WHERE ROWNUM [< |<= ] n [ and ...];
```

示例

```
postgres=# create table t1(f1 int,f2 varchar2(16));
CREATE TABLE
postgres=# insert into t1 values (1,'ONE');
INSERT 0 1
postgres=# insert into t1 values (2,'TWO');
INSERT 0 1
postgres=# insert into t1 values (3,'THREE');
INSERT 0 1
postgres=# insert into t1 values (4,'FOUR');
INSERT 0 1
postgres=# insert into t1 values (5,'FIVE');
INSERT 0 1
postgres=# insert into t1 values (6,'SIX');
INSERT 0 1
postgres=# insert into t1 values (7,'SEVEN');
INSERT 0 1
postgres=# select * from t1 order by f1;
 f1 |  f2
----+-------
  1 | ONE
  2 | TWO
  3 | THREE
  4 | FOUR
  5 | FIVE
```

```
  6 | SIX
  7 | SEVEN
(7 rows)

postgres=# select rownum,f1,f2 from t1 where f1>5;
 rownum | f1 |  f2
--------+----+-------
      1 |  6 | SIX
      2 |  7 | SEVEN
(2 rows)
```

ROWNUM从1开始。

```
postgres=# select rownum,f1,f2 from t1 where rownum<2;
 rownum | f1 | f2
--------+----+-----
      1 |  1 | ONE
(1 row)
```

只显示一行数据。

```
postgres=# select rownum,f1,f2 from t1 where rownum>=1;
 rownum | f1 |  f2
--------+----+-------
      1 |  1 | ONE
      2 |  2 | TWO
      3 |  5 | FIVE
      4 |  6 | SIX
      5 |  3 | THREE
      6 |  4 | FOUR
      7 |  7 | SEVEN
(7 rows)
```

显示全部数据。

```
postgres=# select rownum,f1,f2 from t1 where rownum between 1 and 2;
 rownum | f1 | f2
--------+----+-----
      1 |  1 | ONE
      2 |  2 | TWO
(2 rows)

postgres=# select rownum,f1,f2 from t1 where rownum between 2 and 3;
 rownum | f1 | f2
--------+----+----
(0 rows)
```

rownumbetween 如果大于1，查询结果为空，因为ROWNUM总是从1开始。

ROWID

Oracle数据库的表中的每一行数据都有一个唯一的标识符，或者称为ROWID，在Oracle内部通常就是使用它来访问数据的。ROWID需要 10个字节的存储空间，并用18个字符来显示。该值表明了该行在Oracle数据库中的物理具体位置。可以在一个查询中使用ROWID来表明查询结果中包含该值。
保存ROWID需要10个字节或者是80个位二进制位。这80个二进制位分别是：

1. 数据对象编号，表明此行所属的数据库对象的编号，每个数据对象在数据库建立的时候都被唯一分配一个编号，并且此编号唯一。数据对象编号占用大约32位。
2. 对应文件编号，表明该行所在文件的编号，表空间的每一个文件标号都是唯一的。文件编号所占用的位置是10位。
3. 块编号，表明改行所在文件的块的位置块编号需要22位。
4. 行编号，表明该行在行目录中的具体位置行编号需要16位。
   这样加起来就是80位。

Oracle的物理扩展ROWID有18位，每位采用64位编码，分别用A~~Z、a~~z、0~9、+、/共64个字符表示。A表示0，B表示1，……Z表示25，a表示26，……z表示51，0表示52，……，9表示61，+表示62，/表示63。

TDSQL PG 兼容Oracle的ROWID，由

使用限制
关于使用ROWID的一些注意事项：

- 参数default_with_rowid可以设置默认建表时是否带ROWID，默认该值为off；可以根据需要修改该参数。
- default_with_rowid=OFF时，也可以通过create table 时加上with (rowids)来让表带ROWID列。
- ROWID列默认没有索引，所以基于该列的sql可能存在性能问题，可能通过创建索引来提高sql效率，建索引语法与普通字段建索引语法相同。
- 不支持Oracle的DBMS_ROWID包调用。
- cluster table、vacuum full table重写表数据后，ROWID会变，与Oracle一致。

语法
default_with_rowid=off，建表时带ROWID语法，示例：

```
CREATE TABLE[ IF NOT EXISTS ] table_name
( [
  { column_name data_type [ COLLATE collation] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ PARTITION BY { RANGE | LIST | HASH } ( {column_name | ( expression ) } [ COLLATE collation ] [ opclas
```

```
s ] [, ... ] ) ]
 [ WITH ( storage_parameter [= value] [, ... ]) | WITH OIDS | WITHOUT OIDS ]
 WITH ( ROWIDS ) ;
```

示例：

```
postgres=#create table t_rowid(f1 int,f2 varchar2(32)) with (rowids);
CREATE TABLE
postgres=#insert into t_rowid(f1,f2) values(1,'ONE');
INSERT 0 1
postgres=#insert into t_rowid(f1,f2) values(2,'TWO');
INSERT 0 1
postgres=#select rowid,* from t_rowid;
       rowid        | f1 | f2
--------------------+----+-----
 XPK3fw==AQAAAAAAAAA= |  1 | ONE
 XPK3fw==AgAAAAAAAAA= |  2 | TWO

postgres=#select rowid,* from t_rowid where rowid='XPK3fw==AgAAAAAAAAA=';
       rowid        | f1 | f2
--------------------+----+-----
 XPK3fw==AgAAAAAAAAA= |  2 | TWO
(1 row)

postgres=#update t_rowid set f2='TWO1' where f1=2;
UPDATE 1
postgres=#select rowid,* from t_rowid where rowid='XPK3fw==AgAAAAAAAAA=';
       rowid        | f1 | f2
--------------------+----+------
 XPK3fw==AgAAAAAAAAA= |  2 | TWO1
(1 row)

#数据更新，ROWID不变。

postgres=#create index idx_t_rowid_rowid on t_rowid(rowid);
CREATE INDEX
postgres=#explain select rowid,* from t_rowid where rowid='XPK3fw==AgAAAAAAAAA=';
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   -> Index Scan using idx_t_rowid_rowid on t_rowid  (cost=0.13..4.15 rows=1 width=98)
        Index Cond: (rowid ='XPK3fw==AgAAAAAAAAA='::rowid)
(4 rows)
```

支持ROWID字段上创建索引。

hint

加载插件

```
postgres=# create extension pg_hint_plan ;
CREATE EXTENSION
postgres=# alter role all set session_preload_libraries='pg_hint_plan';
ALTER ROLE
postgres=# \c
```

部分hint用法示例

full全表扫描

示例：/*+full(table_name) */强制选择全表扫描

创建表并插入数据，对比使用hint全表扫描前后执行计划差异。

```
postgres=# create table hint_t1(f1 integer,f2 integer) ;
CREATE TABLE
postgres=# create index hint_t1_f1_idx on hint_t1(f1);
CREATE INDEX
postgres=# insert into hint_t1 select t as f1,t as f2 from generate_series(1,10000) as t;
INSERT 0 10000
postgres=# explain select /*+SeqScan(hint_t1) */ * from hint_t1 where f1=1;
                          QUERY PLAN
------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001)  (cost=0.00..205.00 rows=1 width=8)
   ->  Seq Scan on hint_t1  (cost=0.00..205.00 rows=1 width=8)
         Filter: (f1 = 1)
(4 rows)
postgres=#  explain select  * from hint_t1 where f1=1;
                             QUERY PLAN
---------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001)  (cost=0.16..4.18 rows=1 width=8)
   ->  Index Scan using hint_t1_f1_idx on hint_t1  (cost=0.16..4.18 rows=1 width=8)
         Index Cond: (f1 = 1)
(3 rows)
postgres=# explain select /*+full(hint_t1) */ * from hint_t1 where f1=1;
                          QUERY PLAN
------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001)  (cost=0.00..205.00 rows=1 width=8)
   ->  Seq Scan on hint_t1  (cost=0.00..205.00 rows=1 width=8)
         Filter: (f1 = 1)
(3 rows)
```

index全表扫描

示例：*/+index(table_name)/*对于指定表的查询强制选择索引

```
postgres=# create table hint_t2(f1 integer,f2 integer) ;
CREATE TABLE
postgres=# create index hint_t2_f1_idx on hint_t2(f1);
CREATE INDEX
postgres=# insert into hint_t2 select t as f1,t as f2 from generate_series(1,1000) as t;
INSERT 0 1000
postgres=# vacuum ANALYZE hint_t2;
VACUUM
postgres=#
postgres=# explain select /*+index(hint_t2) */ * from hint_t2 where f1>1;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001)  (cost=0.15..28.65 rows=1000 width=8)
   ->  Index Scan using hint_t2_f1_idx on hint_t2  (cost=0.15..28.65 rows=1000 width=8)
         Index Cond: (f1 > 1)
(3 rows)
postgres=# explain select /*+index(hint_t2 hint_t2_f1_idx1) */ * from hint_t2 where f1>1;
                                  QUERY PLAN
-----------------------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001)  (cost=10000000000.00..10000000020.50 rows=1000 width=8)
   ->  Seq Scan on hint_t2  (cost=10000000000.00..10000000020.50 rows=1000 width=8)
         Filter: (f1 > 1)
(3 rows)
```

no_index全表扫描

示例：*/+no_index(table_name)/*强制SQL不走特定索引的方法

```
postgres=# create table hint_t7(f1 integer,f2 integer) ;
CREATE TABLE
postgres=# insert into hint_t7 select t as f1,t as f2 from generate_series(1,10000) as t;
INSERT 0 10000
postgres=# create index hint_t7_f1_idx on hint_t7(f1);
CREATE INDEX
postgres=# vacuum ANALYZE hint_t7;
VACUUM
postgres=# explain select /*+no_index(hint_t7) */ * from hint_t7 where f1=1 ;
                          QUERY PLAN
------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001)  (cost=0.00..200.00 rows=1 width=8)
   ->  Seq Scan on hint_t7  (cost=0.00..200.00 rows=1 width=8)
         Filter: (f1 = 1)
(3 rows)
postgres=# explain select  * from hint_t7 where f1=1 ;
                             QUERY PLAN
---------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001)  (cost=0.16..4.18 rows=1 width=8)
   ->  Index Scan using hint_t7_f1_idx on hint_t7  (cost=0.16..4.18 rows=1 width=8)
         Index Cond: (f1 = 1)
(3 rows)
postgres=#
```

别名使用

```
postgres=# create table hint_t6(f1 integer,f2 integer) ;
CREATE TABLE
postgres=# insert into hint_t6 select t as f1,t as f2 from generate_series(1,10000) as t;
INSERT 0 10000
postgres=# create index hint_t6_f1_idx on hint_t6(f1);
CREATE INDEX
postgres=# vacuum ANALYZE hint_t6;
VACUUM
postgres=# select /*+full(hint_t6) */ * from hint_t6 a where f1=1;
 f1 | f2
----+----
  1 |  1
(1 row)
postgres=# explain select /*+full(hint_t6) */ * from hint_t6 a where f1=1;;
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001)  (cost=0.16..4.18 rows=1 width=8)
   ->  Index Scan using hint_t6_f1_idx on hint_t6 a  (cost=0.16..4.18 rows=1 width=8)
         Index Cond: (f1 = 1)
(3 rows)
postgres=# explain select /*+full(a) */ * from hint_t6 a where f1=1;
                            QUERY PLAN
------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001)  (cost=0.00..200.00 rows=1 width=8)
   ->  Seq Scan on hint_t6 a  (cost=0.00..200.00 rows=1 width=8)
         Filter: (f1 = 1)
(3 rows)
```

强制走nest loop

示例：/*+use_nl(table1,table2) */强制选择嵌套循环连接

```
postgres=# create table hint_t6(f1 integer,f2 integer) ;
CREATE TABLE
postgres=# insert into hint_t6 select t as f1,t as f2 from generate_series(1,10000) as t;
INSERT 0 10000
postgres=# vacuum ANALYZE hint_t6;
VACUUM
postgres=#
postgres=# create table hint_t7(f1 integer,f2 integer) ;
CREATE TABLE
postgres=# insert into hint_t7 select t as f1,t as f2 from generate_series(1,10000) as t;
INSERT 0 10000
postgres=# vacuum ANALYZE hint_t7;
VACUUM
postgres=#
postgres=# explain select * from hint_t6 t,hint_t7 t1 where t.f1=t1.f1 and t1.f1>9999 ;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001)  (cost=200.01..412.52 rows=1 width=16)
   ->  Hash Join  (cost=200.01..412.52 rows=1 width=16)
         Hash Cond: (t.f1 = t1.f1)
         ->  Seq Scan on hint_t6 t  (cost=0.00..175.00 rows=10000 width=8)
         ->  Hash  (cost=200.00..200.00 rows=1 width=8)
               ->  Seq Scan on hint_t7 t1  (cost=0.00..200.00 rows=1 width=8)
                     Filter: (f1 > 9999)
(7 rows)
postgres=# explain select /*+use_nl(t,t1) */ * from hint_t6 t,hint_t7 t1 where t.f1=t1.f1 and t.f1>999 ;
                               QUERY PLAN
-------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001)  (cost=0.00..1350547.50 rows=9001 width=16)
   ->  Nested Loop  (cost=0.00..1350547.50 rows=9001 width=16)
         Join Filter: (t.f1 = t1.f1)
         ->  Seq Scan on hint_t7 t1  (cost=0.00..175.00 rows=10000 width=8)
         ->  Materialize  (cost=0.00..245.00 rows=9001 width=8)
               ->  Seq Scan on hint_t6 t  (cost=0.00..200.00 rows=9001 width=8)
                     Filter: (f1 > 999)
(7 rows)
postgres=#
```

强制走merge join

示例：/*+use_merge(table1,table2) */强制选择合并连接

```
postgres=# create table hint_t6(f1 integer,f2 integer) ;
CREATE TABLE
postgres=# insert into hint_t6 select t as f1,t as f2 from generate_series(1,10000) as t;
INSERT 0 10000
postgres=# create index hint_t6_f1_idx on hint_t6(f1);
CREATE INDEX
postgres=# vacuum ANALYZE hint_t6;
VACUUM
postgres=# create table hint_t7(f1 integer,f2 integer) ;
CREATE TABLE
postgres=# insert into hint_t7 select t as f1,t as f2 from generate_series(1,10000) as t;
INSERT 0 10000
postgres=# create index hint_t7_f1_idx on hint_t7(f1);
CREATE INDEX
postgres=# vacuum ANALYZE hint_t7;
VACUUM
postgres=#
postgres=# explain select * from hint_t6 t,hint_t7 t1 where t.f1=t1.f1 and t.f1>9999 ;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001)  (cost=0.32..8.37 rows=1 width=16)
   ->  Nested Loop  (cost=0.32..8.37 rows=1 width=16)
         ->  Index Scan using hint_t6_f1_idx on hint_t6 t  (cost=0.16..4.18 rows=1 width=8)
               Index Cond: (f1 > 9999)
         ->  Index Scan using hint_t7_f1_idx on hint_t7 t1  (cost=0.16..4.18 rows=1 width=8)
               Index Cond: (f1 = t.f1)
(6 rows)
postgres=# explain select /*+use_merge(t,t1) */ * from hint_t6 t,hint_t7 t1 where t.f1=t1.f1 and t.f1>99
99 ;
                                   QUERY PLAN
-----------------------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001)  (cost=0.32..257.35 rows=1 width=16)
   ->  Merge Join  (cost=0.32..257.35 rows=1 width=16)
         Merge Cond: (t.f1 = t1.f1)
         ->  Index Scan using hint_t6_f1_idx on hint_t6 t  (cost=0.16..4.18 rows=1 width=8)
               Index Cond: (f1 > 9999)
         ->  Index Scan using hint_t7_f1_idx on hint_t7 t1  (cost=0.16..228.16 rows=10000 width=8)
(6 rows)
postgres=#
```

强制走hash join

示例：/*+use_hash(table1,table2) */强制选择哈希连接

```
postgres=# create table hint_t6(f1 integer,f2 integer) ;
CREATE TABLE
postgres=# insert into hint_t6 select t as f1,t as f2 from generate_series(1,10000) as t;
INSERT 0 10000
postgres=# create index hint_t6_f1_idx on hint_t6(f1);
CREATE INDEX
postgres=# vacuum ANALYZE hint_t6;
VACUUM
postgres=# create table hint_t7(f1 integer,f2 integer) ;
CREATE TABLE
postgres=# insert into hint_t7 select t as f1,t as f2 from generate_series(1,10000) as t;
INSERT 0 10000
postgres=# create index hint_t7_f1_idx on hint_t7(f1);
CREATE INDEX
postgres=# vacuum ANALYZE hint_t7;
VACUUM
postgres=#
postgres=#  explain select * from hint_t6 t,hint_t7 t1 where t.f1=t1.f1 and t.f1>9999 ;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001)  (cost=0.32..8.37 rows=1 width=16)
   ->  Nested Loop  (cost=0.32..8.37 rows=1 width=16)
         ->  Index Scan using hint_t6_f1_idx on hint_t6 t  (cost=0.16..4.18 rows=1 width=8)
               Index Cond: (f1 > 9999)
         ->  Index Scan using hint_t7_f1_idx on hint_t7 t1  (cost=0.16..4.18 rows=1 width=8)
               Index Cond: (f1 = t.f1)
(6 rows)
postgres=# explain select /*+use_hash(t,t1) */  * from hint_t6 t,hint_t7 t1 where t.f1=t1.f1 and t.f1>999
9 ;
                                   QUERY PLAN
-----------------------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001)  (cost=4.19..216.70 rows=1 width=16)
   ->  Hash Join  (cost=4.19..216.70 rows=1 width=16)
         Hash Cond: (t1.f1 = t.f1)
         ->  Seq Scan on hint_t7 t1  (cost=0.00..175.00 rows=10000 width=8)
         ->  Hash  (cost=4.18..4.18 rows=1 width=8)
               ->  Index Scan using hint_t6_f1_idx on hint_t6 t  (cost=0.16..4.18 rows=1 width=8)
                     Index Cond: (f1 > 9999)
(7 rows)
```

并行执行

示例：parallel(n)指定SQL的并发数

```
postgres=# create table hint_t3(f1 integer,f2 integer) ;
CREATE TABLE
postgres=# insert into hint_t3 select t as f1,t as f2 from generate_series(1,1000000) as t;
INSERT 0 1000000
postgres=# vacuum ANALYZE hint_t3;
VACUUM
postgres=#
postgres=#  explain select count(1) from hint_t3;
                              QUERY PLAN
-----------------------------------------------------------------------------------
 Finalize Aggregate  (cost=20020.01..20020.02 rows=1 width=8)
   ->  Remote Subquery Scan on all (dn001)  (cost=20020.00..20020.01 rows=1 width=0)
         ->  Partial Aggregate  (cost=19920.00..19920.01 rows=1 width=8)
               ->  Seq Scan on hint_t3  (cost=0.00..17420.00 rows=1000000 width=0)
(4 rows)

postgres=# explain select /*+parallel(hint_t3 2) */ count(1) from hint_t3;
                                  QUERY PLAN
-----------------------------------------------------------------------------------------------
 Parallel Finalize Aggregate  (cost=13728.35..13728.36 rows=1 width=8)
   ->  Parallel Remote Subquery Scan on all (dn001)  (cost=13728.33..13728.35 rows=1 width=0)
         ->  Gather  (cost=13628.33..13628.34 rows=1 width=8)
             Workers Planned: 2
               ->  Partial Aggregate  (cost=12628.33..12628.34 rows=1 width=8)
                     ->  Parallel Seq Scan on hint_t3  (cost=0.00..11586.67 rows=416667 width=0)
(6 rows)
postgres=#
```

运算符
算术运算符

可以使用算数运算符对TDSQL PG所支持数据类型的值进行操作。
下面的表显示了允许使用的算数运算符。

运算符	描述	示例	结果
+	加法	4 + 4	8
-	减法	5 - 2	3
*	乘法	4 * 4	16
/	除法	16/4	4.0000000000000000

这里除法与Oracle运算结果精度稍有差异，TDSQL PG运算结果小数点后为16-20位，Oracle运算结果小数点后最多9位，自动省略末尾0，小于1时自动省略个位数。

示例：

```
TDSQL PG：
postgres=#select 16/4 from dual;
       ?column?
--------------------
 4.0000000000000000
(1 row)
postgres=#select 2/3 from dual;
       ?column?
-----------------------
 0.66666666666666666667
postgres=#select 4/4 from dual;
       ?column?
-----------------------
 1.00000000000000000000
(1 row)

Oracle：
SQL>select 16/4 from dual;

      16/4
----------
         4
SQL>select 2/3 from dual;
```

```
      2/3
----------
.666666667
SQL>select 4/4 from dual;

      4/4
----------
         1
SQL>select 20/8 from dual;

     20/8
----------
       2.5
```

逻辑运算符

AND OR NOT

比较运算符

> ,<,>=,<=,<>,!=

连接运算符

TDSQL PG兼容Oracle的sql连接符"||"，支持字符与字符、字符与数字、数字与数字类型的连接运算。

描述	示例	结果
字符与字符	'a'||'b'	ab
字符与数字	'a'||1	a1
数字与数字	1||2	12
1||2.1	12.1	
1.1||2.1	1.12.1	

示例：

```
postgres=# select 'a'||'b' from dual;
 ?column?
----------
 ab
(1 row)

postgres=# select 'a'||1 from dual;
 ?column?
----------
 a1
(1 row)

postgres=# select 1||2 from dual;
 ?column?
----------
 12
(1 row)

postgres=# select 1||2.1 from dual;
 ?column?
----------
 12.1
(1 row)

postgres=# select 1.1||2.1 from dual;
 ?column?
----------
 1.12.1
```

(1 row)

集合运算符

集合运算符

TDSQL PG兼容Oracle的集合运算符UNION、UNION ALL、INTERSECT、MINUS。

运算符	描述
UNION	多个sql结果求并集，结果集去重
UNION ALL	多个sql结果求并集，结果集不去重
INTERSECT	多个sql结果求交集，结果集去重
MINUS	多个sql结果求差集，结果集去重

语法：

```
SELECTSTATEMENT
UNION |UNION ALL | INTERSECT | MINUS
SELECTSTATEMENT
[
UNION |UNION ALL | INTERSECT | MINUS
SELECTSTATEMENT
...
]
[ ORDER BY [number [ ASC | DESC ] , ... ]
```

示例：

```
create table student
(
id int primary key,
namen varchar2(50) not null,
score number not null
);
insert into student values(1,'Aaron',78);
insert into student values(2,'Bill',76);
insert into student values(3,'Cindy',89);
insert into student values(4,'Damon',90);
insert into student values(5,'Ella',73);
insert into student values(6,'Frado',61);
insert into student values(7,'Gill',99);
insert into student values(8,'Hellen',56);
insert into student values(9,'Ivan',93);
insert into student values(10,'Jay',90);
```

UNION示例：

```
postgres=#select * from student where id < 4 union select * from student where id >2 and id < 6;
 id | name | score
----+-------+-------
  5 | Ella |    73
  3 | Cindy |   89
  1 | Aaron |   78
  2 | Bill |    76
  4 | Damon |   90
(5 rows)
```

UNION ALL示例：

```
postgres=#select * from student where id < 4 union all select * from student where id> 2 and id < 6;
 id | name | score
----+-------+-------
  1 | Aaron |   78
  2 | Bill |    76
  5 | Ella |    73
  3 | Cindy |   89
  3 | Cindy |   89
  4 | Damon |   90
(6 rows)
```

INTERSECT示例:

```
postgres=#select * from student where id < 4 intersect select * from student where id> 2 and id < 6;
 id | name | score
----+-------+-------
  3 | Cindy |   89
(1 row)
```

MINUS示例：

```
postgres=#select * from student where id < 4 minus select * from student where id >2 and id < 6;
 id | name | score
----+-------+-------
  1 | Aaron |   78
  2 | Bill |    76
(2 rows)
```

其他运算符

（+）

TDSQL PG兼容Oracle的sql（左/右）外连接"(+)"写法，等同于left | right join。

使用限制

关于使用(+)的一些注意事项：

- (+)操作符只能出现在where子句中，且不能与outer join语法同时使用。
- 使用(+)操作符进行关联操作时，如果where子句中包含多个条件，则关联的两张表的其它所有条件上关联表的字段上必须都包含(+)操作符。
- (+)操作符只能用于列，不能用于表达式。
- (+)操作符不能与or和in操作符一起使用。

语法

左外连接（left join）：

```
... WHERE table1.column1=table2.column1(+) [ AND table1.column2 =table2.column2(+) ...]
```

右外连接（right join）：

```
... WHERE table1.column1(+)=table2.column1[ AND table1.column2(+) =table2.column2 ...]
```

示例

```
create table a(a1 number(12),a2 varchar2(50));
create table b(b1 number(12),b2 varchar2(50));
insert into a values (1,'a'),(2,'ba'),(3,'ca'),(4,'da');
insert into b values (1,'a'),(2,'bba'),(3,'ca'),(5,'dda'),(6,'eda');
```

左外连接（left join）：

```
postgres=# select a.a1,b.b1 from a,b where a.a1=b.b1(+) ;
 a1 | b1
----+----
  1 |  1
  2 |  2
  4 |
  3 |  3
(4 rows)
```

右外连接（right join）：

```
postgres=# select a.a1,b.b1 from a,b where a.a1(+)=b.b1;
 a1 | b1
```

```
----+----
  1 |  1
  3 |  3
  2 |  2
    |  5
    |  6
(5 rows)
```

空值与NULL

空值：即空字符串，两个单引号，中间无空格等任何内容。
NULL：SQL关键字，代表没有。
Oracle中的空值被当做NULL处理，TDSQL PG（ORACLE兼容版）则保持与Oracle一致。
下表是空值与NULL在不同数据库中的对比情况。

描述	Oracle	TDSQL PG
空值与NULL是否一致	一致	一致
NULL与数值运算	NULL	NULL
空值与数值运算	NULL	NULL
空值排序	空值最大	空值最大
NULL排序	NULL值最大	NULL值最大

使用限制

- 空值、NULL与任何数值运算，结果为NULL。
- SQL中空值、NULL不能用=''、!=''来判断，应使用is null、is not null。
- 对于有空值、NULL的情况，需要注意排序时的场景，如有需要可以用nvl转换为

示例

```
postgres=#select nvl2('','1','2') from dual ;
 nvl2
------
 2
(1 row)

postgres=#select 1 from dual where '' is null ;
 ?column?
----------
        1
(1 row)

postgres=#select null||'tbase' from dual;
 ?column?
----------
 tbase
(1 row)
```

```
postgres=#select null+1 from dual;
 ?column?
----------

(1 row)

postgres=#drop table t_null;
DROP TABLE
postgres=#  create table t_null(f1 int,f2 varchar2(10));
CREATE TABLE
postgres=#  insert into t_null values(1,null);
INSERT 0 1
postgres=#  insert into t_null values(2,'');
INSERT 0 1
postgres=#select * from t_null where f2 is null;
 f1 | f2
----+----
  1 |
  2 |
(2 rows)

postgres=#select * from t_null where f2 ='' ;
 f1 | f2
----+----
(0 rows)

postgres=#insert into t_null values(3,'1');
INSERT 0 1
postgres=#insert into t_null values(4,'2');
INSERT 0 1
postgres=#select * from t_null order by f2;
 f1 | f2
----+----
  3 | 1
  4 | 2
  1 |
  2 |
(4 rows)
```

字符串拼接

示例

```
set enable_oracle_compatible=on;
set transform_null_equals=on;

select 'a'|| ascii(null) || 'b' from dual;
select ascii(null) || ascii('1') || ascii(null) from dual;
select ascii(null) || null || ascii(null) from dual;
select 'a'|| 'b' || 'c' from dual;
select'a'||nvl2('', 1, 2)||'b' from dual;

create table sale (month char(6),sell number);
insert into sale values('200001',1000);
insert into sale values('200002',1100);
insert into sale values('200003',null);
insert into sale values('200004', null);
select month||sell from sale;
```

GOTO

语法：

GOTO lable

示例：

```
create tablegoto_test(id int,name varchar);

create orreplace function goto_func(v_maxnum integer) returns void as
declare
maxnuminteger;
begin
maxnum :=v_maxnum;
for i in1..maxnum loop
if i=3 then
goto label;
end if;
insert intogoto_test values (i,'hasai');
end loop;
<<label>>
updategoto_test set name = 'user1' where id = 2;
end;
/

select goto_func(5);


select *from goto_test;
```

其他常用特性
table函数

利用table()函数，我们可以将PL/SQL返回的结果集代替table。
TABLE 表达式可用于查询任何集合值表达式，包括变量和参数等瞬态值。
TABLE 表达式中的子查询具有以下限制：

- 子查询必须返回一个集合类型。
- 子查询的 SELECT 列表必须只包含一项。
- 子查询必须只返回一个集合； 它不能返回多行的集合。

示例：

```
postgres=# select * from table(string_to_array('ab,cd,ef', ','))t;

 column_value
--------------
 ab
 cd
 ef

(3 行记录)
```

sqlcode

在异常处理程序中，SQLCODE 返回正在处理的异常的数字代码。

SQLCODE取值范围	场景
[1]	不绑定直接抛出场景
[100]	NO_DATA_FOUND
[-20999, -20000]	返回RAISE_APPLICATION_ERROR绑定的sqlcode
[-99999, -21000] or [-20001, -1]	其他异常无法对齐，SQLCODE统一为-1

示例：

```
 *  4.1 taped show sqlcode -1 and type is number
 *  4.2 user defined exception
 *  4.3 user defined exception without assigning error code value
 *  4.4 NO_DATA_FOUND
 *  4.5 others
 *  4.6 type test include out and assignment （type:INTEGER,NUMBER,NUMERIC)
 *  4.7 compare with sqlcode

drop extension if exists tbase_oracle_package_function;
create extension tbase_oracle_package_function;
--  4.1 taped show sqlcode -1 and type is number
declare
  x NUMBER := 9;
  y NUMBER := 0;
  z NUMBER;
  o_return_code NUMBER;
begin
  z := x/y;
exception
  when others then
    o_return_code := sqlcode;
    raise notice '%',o_return_code;
end;
/
--  4.2 user defined exception
DECLARE
  past_due  EXCEPTION;                    -- declare exception
  PRAGMA EXCEPTION_INIT (past_due, -20999);  -- assign error code to exception
BEGIN
```

```
  RAISE_APPLICATION_ERROR(-20999, 'Account past due.', false);
EXCEPTION
  when past_due then
   raise notice '%',sqlstate;
   raise notice '%',sqlcode;
END;
/
--  4.3 user defined exception without assigning error code value
DECLARE
  xxx  EXCEPTION;
BEGIN
  raise xxx;
EXCEPTION
  WHEN xxx THEN
    raise notice '%',sqlstate;
    raise notice '%',sqlcode;
END;
/
--  4.4 NO_DATA_FOUND
create table t_test_no_data_found(id int, name varchar2(20));
DECLARE
  name1 varchar2(20);
BEGIN
  select name into name1 from t_test_no_data_found where id = 1;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    raise notice '%',sqlstate;
    raise notice '%',sqlcode;
END;
/
drop table t_test_no_data_found;
--  4.5 others
create or replace function f_test_err return int as
DECLARE
  x NUMBER := 9;
  y NUMBER := 0;
  z NUMBER;
BEGIN
  z := x/y;
  return z;
EXCEPTION
  WHEN division_by_zero then
    raise notice 'sqlstate:%',sqlstate;
    raise notice 'sqlcode:%',sqlcode;
    return 0;
END;
/
```

```
select f_test_err() from dual;
drop function f_test_err;
--  4.6 type test include out and assignment  ( type:INTEGER,NUMBER,NUMERIC)
CREATE or replace PROCEDURE pro_test_error_type_out_integer (
 O_RETURN_CODE OUT INTEGER
) AUTHID DEFINER
IS
  var_sql_code_number NUMBER;--type_test
  var_sql_code_numeric NUMERIC;
  var_sql_code_integer INTEGER;

BEGIN
  RAISE_APPLICATION_ERROR(-20999, 'Account past due.', false);
EXCEPTION
  WHEN OTHERS THEN
  O_RETURN_CODE := sqlcode;
  var_sql_code_number := sqlcode;
  var_sql_code_numeric := sqlcode;
  var_sql_code_integer := sqlcode;
  raise notice 'number:%, numeric:%, integer:%',var_sql_code_number,var_sql_code_numeric,var_sql_cod
e_integer;
END;
/

CREATE or replace PROCEDURE pro_test_error_type_out_numeric (
 O_RETURN_CODE OUT NUMERIC
) AUTHID DEFINER
IS
  var_sql_code_number NUMBER;--type_test
  var_sql_code_numeric NUMERIC;
  var_sql_code_integer INTEGER;

BEGIN
  RAISE_APPLICATION_ERROR(-20000, 'Account past due.', false);
EXCEPTION
  WHEN OTHERS THEN
  O_RETURN_CODE := sqlcode;
  var_sql_code_number := sqlcode;
  var_sql_code_numeric := sqlcode;
  var_sql_code_integer := sqlcode;
  raise notice 'number:%, numeric:%, integer:%',var_sql_code_number,var_sql_code_numeric,var_sql_cod
e_integer;
END;
/

CREATE or replace PROCEDURE pro_test_error_type_out_number (
 O_RETURN_CODE OUT NUMBER
```

```
) AUTHID DEFINER
IS
  var_sql_code_number NUMBER;--type_test
  var_sql_code_numeric NUMERIC;
  var_sql_code_integer INTEGER;
  z INTEGER;
BEGIN
  z := 9/130618222416261120;
EXCEPTION
  WHEN OTHERS THEN
  O_RETURN_CODE := sqlcode;
  var_sql_code_number := sqlcode;
  var_sql_code_numeric := sqlcode;
  var_sql_code_integer := sqlcode;
  raise notice 'number:%, numeric:%, integer:%',var_sql_code_number,var_sql_code_numeric,var_sql_cod
e_integer;
END;
/

declare
  O_RETURN_CODE_INTEGER INTEGER;
  O_RETURN_CODE_NUMERIC NUMERIC;
  O_RETURN_CODE_NUMBER NUMBER;

begin
  pro_test_error_type_out_integer(O_RETURN_CODE_INTEGER);
  raise notice 'return type integer:%',O_RETURN_CODE_INTEGER;
  pro_test_error_type_out_numeric(O_RETURN_CODE_NUMERIC);
  raise notice 'return type numeric:%',O_RETURN_CODE_NUMERIC;
  pro_test_error_type_out_number(O_RETURN_CODE_NUMBER);
  raise notice 'return type number:%',O_RETURN_CODE_NUMBER;
end;
/

drop PROCEDURE pro_test_error_type_out_number;
drop PROCEDURE pro_test_error_type_out_numeric;
drop PROCEDURE pro_test_error_type_out_integer;
--  4.7 compare with sqlcode
declare
  x NUMBER := 9;
  y NUMBER := 0;
  z NUMBER;
  var_number NUMBER := -1;
  var_integer INTEGER := -1;
  var_numeric NUMERIC := -1;
  o_return_code NUMBER;
begin
```

```
  z := x/y;
exception
  when others then
    raise notice '%', sqlcode;
    if sqlcode = -1 then
      raise notice 'support compare with -1';
    end if;
    if sqlcode = var_number then
      raise notice 'support compare with number';
    end if;
    if sqlcode = var_integer then
      raise notice 'support compare with integer';
    end if;
    if sqlcode = var_numeric then
      raise notice 'support compare with numeric';
    end if;
end;
/
drop extension if exists tbase_oracle_package_function;
```

select for update wait

支持oracle语法select for update

语法描述

[FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [OF table_name [, ...]] [NOWAIT | SKIP LOCKED | WAIT Iconst] [...]]

功能说明

1、锁冲突矩阵

锁模式	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

2、如果针对一个表指定了多个加锁策略，则采用如下规则进行最终策略选择：

- 选择最强加锁强度：LCS_NONE
- 选择最强等待策略：LockWaitBlock
- 选择最短等待时间，如果指定了wait字段，选择值最小的等待时间，如果值为0，则视为无穷大

  3、如果同时设置了StatementTimeout、LockTimeout和waitTimeout, 则报错信息以最先到的超时时间为准，不同超时时间报错信息如下：
- StatementTimeout: canceling statement due to statement timeout
- LockTimeout: canceling statement due to lock timeout
- waitTimeout: canceling statement due to row lock timeout

  4、当同一个语句中对从属于多个表的行加锁时，可以分别指定不同的加锁策略和等待规则

  5、不支持cte和join同时出现

  6、行锁不支持集合操作：UNION/INTERSECT/EXCEPT

  7、行锁语句不支持DISTINCT去重

8、行锁不支持GROUP BY、HAVING等子句
9、行锁不支持聚合函数
10、行锁不支持窗口函数
11、行锁语句不支持返回集合类型的函数

示例：

```
create table tb_struc
(
id int,
dt date,
time_str timestamp,
chr char(50),
str varchar(50),
str2 varchar2(50),
bb blob,
cb clob,
xml xmltype
);

select id from tb_struc for update wait 3;
```

自治事务

自治事务(AUTONOMOUS TRANSACTION，即AT)由主事务调用，但是又独立于主事务。MT(Main Transaction)在执行到 AT的时候，MT 就会被挂起，在 AT执行的时候内部程序可以进行任何 COMMIT 和ROLLBACK 操作，这些操作不会影响到 MT 的执行。在AT 执行完成之后 MT 又会获得程序的控制权，继续向下执行，在 MT 执行的过程中任何的 COMMIT 和 ROLLBACK 也不会影响到 AT 的执行。

自治事务主要用于2个场景：一是程序日志，二是触发器。在PLSQL程序开发过程中，一般都需要记录一些中间数据或者执行时间等，但是在一个大的事务中间，需要把日志保存下来而不影响整个事务的提交或回退，就只能用自治事务了。 有一个触发器，在table 执行UPDATE 之后被触发，以记录审计数据。假设你希望即使主事务失败，仍旧能够记录审计数据（即触发器操作仍然有效），这样可以通过自治事务实现。

自治事务允许离开调用事务的上下文，执行独立事务并返回到调用事务，而不会影响其状态。自治事务没有与调用事务的链接，因此两个事务只能共享已提交的数据。可以将以下类型的PL / SQL块定义为自治事务：
—存储过程、函数
—局部存储过程、函数
—包内存储过程、函数
— 类型方法
—最顶层的匿名块

示例：

```
set enable_oracle_compatible to on;

set client_min_messages to warning;

-- 公共插入函数
drop procedure if exists insert_data_start(starts int,table_num int, table_name varchar2);
create or REPLACE procedure insert_data_start(starts int,table_num int, table_name varchar2)
as
   v_sql varchar2;
BEGIN
   v_sql := 'insert into '||table_name||' select i, repeat((i%365)::text,5)::bytea, (i%10)::char, int8(i%550), int2(i%127), int4(i%789),round(i%20*2.5866,3)::numeric, bpchar(md5((i%300)::text)), (''工号:''||(i%360))::varchar, ''南山''||(i%400)||''号''::NVARCHAR2, md5((i%300)::TEXT), (i%666)::bigint::oid, ((i%200)*2.35)::float4, float8((i%300)*2.215), ''tbase ''||(i%765),''abstime''||(i%35),''tinterval''||(i%44), concat(concat_ws(''.'',(i+100)%200, i%100,i%100,(i+100)%200),''/25'')::inet, date(abstime(''2022-02-17 19:12:40+08''::timestamp + (i%500+1))), ''00:00:00'' + (i%400+1) * interval ''1 minute'', (date(''2022-02-18'')+(i%24) * interval ''1 day''+(i%24) * interval ''1 minute''+(i%24) * interval ''1 second'')::timestamp,timestamptz(date(date(''2022-02-18'')+(i%24) * interval ''1 day'')),(i%500) * interval ''1 day'',timetz(''2022-02-16 17:29:13.9+08''::timestamp + (i%500)*interval ''1 second''),concat(concat_ws(''.'',(i+100)%200, i%100,i%100,(i+100)%200),''/25'')::inet,int4(i%10)::bit(5), (i%100+1)::int::bit(5)::varbit(5), bool(i%2), concat_ws(''-'',to_hex((i%192)+1),to_hex((i%168)+1),to_hex((i%200)+1),to_hex((i%100)+1),to_hex((i%98)+1),to_hex((i%255)+1))::macaddr, point(ceil((i%68)+1),ceil((i%77)+1)),(i*987)::varchar::oidvector,polygon(box(point(i%550,i%360),p
```

```
oint(i%50,i%22))),((i*3.156)%9999)::money,(i%798,i%666)::varchar::tid,circle(point(i%250,i%160),i%76),c
oncat_ws(''-'',to_hex((i%192)+16),to_hex((i%168)+16),to_hex((i%200)+16),to_hex((i%100)+16),to_hex
((i%98)+16),to_hex((i%239)+16))::macaddr8,line(point(i%93+1,i%120),point(i%450+2,i%36)),path(polyg
on(box(point(i%192,i%120),point(i%913,i%120)))),box(point(i%292+1,i%20),point(i%92+2,i%24)),lseg(p
oint(i%92+1,i%220),point(i%238+2,i%120)),name(''平台''||(i%12)),(''tbase in 城市1''||(i%900))::varchar2,((i
*3.1123)%998)::number(10,2),''生活如此多娇''||(i%120),''南山''||(i%30),(i%50)/13, (i%20)/17, to_hex(i*90
0%500),''公司员工''||(i%40) from generate_series('||starts||','||table_num||') i';
    -- RAISE notice '%', v_sql;
    execute v_sql;
end;
/

drop table if exists test_at_mt_pro_shard_20220801_1 cascade;
create table test_at_mt_pro_shard_20220801_1(c0 serial,c1 BYTEA default '',c2 CHAR default 'a',c3 INT8
default 1999,c4 INT2 default -127,c5 INT4 default -1,c6 numeric default 3.14156,c7 BPCHAR default '',c
8 VARCHAR default '',c9 NVARCHAR2 default 'NVARCHAR2',c10 TEXT default '',c11 OID default '2',c12
FLOAT4 default 10.8,c13 FLOAT8 default 0.999,c14 text default 'tbase',c15 varchar default '00:05:04',c16
varchar default 'tinterval(abstime(now()), abstime(now()))',c17 INET default '148.85.65.189/25',c18 DATE
default '2022-07-26 14:25:25',c19 TIME default '14:25:03',c20 TIMESTAMP default '2022-07-26 12:12:12
',c21 TIMESTAMPTZ default '2022-07-26 12:12:12+08',c22 INTERVAL default interval '1 day',c23 TIMET
Z default '14:36:04+08',c24 CIDR default '198.24.10.0/24',c25 bit varying(5) default '',c26 VARBIT defaul
t '00011',c27 bool default false,c28 macaddr default 'bf:77:bf:5b:5b:92',c29 point default '(10,10)',c30 oi
dvector default '',c31 polygon default '((2,2),(2,2),(2,2),(2,2))',c32 money default 10,c33 tid default
'(2,2)',c34 circle default '<(2,2),2>',c35 macaddr8 default '12:12:12:ff:fe:12:12:12',c36 line default
'{0,-1,2}',c37 path default '((2,2),(2,2),(2,2),(2,2))',c38 box default '(4,2),(3,2)',c39 lseg default '[(3,2),
(4,2)]',c40 name default '',c41 varchar2 default '',c42 number default 1.99,c43 clob default 'clob', c44 N
CLOB default 'NCLOB',c45 BINARY_FLOAT default 3.141596, c46 BINARY_DOUBLE default 1.963489, c47
RAW(100) default '',c48 BLOB default 'BLOB',c49 serial,c50 bigserial,c51 int[] default '{1,2,3}',c52 rowid
default '') distribute by replication;
create unique index on test_at_mt_pro_shard_20220801_1(c0);
create index on test_at_mt_pro_shard_20220801_1(c20,c19);
-- 导入数据
call insert_data_start(1, 5000, 'test_at_mt_pro_shard_20220801_1');
analyze test_at_mt_pro_shard_20220801_1;
select count(1) from test_at_mt_pro_shard_20220801_1;

drop table if exists test_at_mt_proc_self_par_20220801_1 cascade;
create table test_at_mt_proc_self_par_20220801_1(c0 serial,c1 BYTEA default '',c2 CHAR default 'a',c3 IN
T8 default 1999,c4 INT2 default -127,c5 INT4 default -1,c6 numeric default 3.14156,c7 BPCHAR default
'',c8 VARCHAR default '',c9 NVARCHAR2 default 'NVARCHAR2',c10 TEXT default '',c11 OID default '2',c
12 FLOAT4 default 10.8,c13 FLOAT8 default 0.999,c14 text default 'tbase',c15 varchar default '00:05:04',
c16 varchar default 'tinterval(abstime(now()), abstime(now()))',c17 INET default '148.85.65.189/25',c18
DATE default '2022-07-26 14:25:25',c19 TIME default '14:25:03',c20 TIMESTAMP default '2022-07-26 12
:12:12',c21 TIMESTAMPTZ default '2022-07-26 12:12:12+08',c22 INTERVAL default interval '1 day',c23 T
IMETZ default '14:36:04+08',c24 CIDR default '198.24.10.0/24',c25 bit varying(5) default '',c26 VARBIT d
efault '00011',c27 bool default false,c28 macaddr default 'bf:77:bf:5b:5b:92',c29 point default '(10,10)',c
30 oidvector default '',c31 polygon default '((2,2),(2,2),(2,2),(2,2))',c32 money default 10,c33 tid default
```

```
'(2,2)',c34 circle default '<(2,2),2>',c35 macaddr8 default '12:12:12:ff:fe:12:12:12',c36 line default
'{0,-1,2}',c37 path default '((2,2),(2,2),(2,2),(2,2))',c38 box default '(4,2),(3,2)',c39 lseg default '[(3,2),
(4,2)]',c40 name default '',c41 varchar2 default '',c42 number default 1.99,c43 clob default 'clob', c44 N
CLOB default 'NCLOB',c45 BINARY_FLOAT default 3.141596, c46 BINARY_DOUBLE default 1.963489, c47
RAW(100) default '',c48 BLOB default 'BLOB',c49 serial,c50 bigserial,c51 int[] default '{1,2,3}',c52 rowid
default '') partition by range(c18) begin(timestamp '2022-02-18 00:00:00') step(interval '1 month') parti
tions(18);
-- 导入数据
call insert_data_start(1, 5000, 'test_at_mt_proc_self_par_20220801_1');
analyze test_at_mt_proc_self_par_20220801_1;
select count(1) from test_at_mt_proc_self_par_20220801_1;

-- AT事务DDL了表的数据commit,MT事务查询AT事务DDL的数据
drop procedure if exists test_at_pro_ddl_1(create_num int) cascade;
create or replace procedure test_at_pro_ddl_1(create_num int)
as
    pragma autonomous_transaction;
    vsql varchar;
    tbl_name varchar;
    myexp exception;
begin
    perform dbms_output.serveroutput('t');
    for i in 1..create_num
    loop
        tbl_name := 'test_at_mt_temp_'||i;
        vsql := 'drop table if exists '||tbl_name;
        execute immediate(vsql);
        vsql := 'create table '||tbl_name||'(id int, names name, address varchar, dt date, score number(7,2))
'||(array['distribute by shard(dt)','distribute by replication'])[((i%2)+1)];
        execute immediate vsql;
        vsql := 'insert into '||tbl_name||' select i,''tbase''||i,''南山''||i,sysdate, 3.954*i from generate_series(1,
1000) i';
        execute immediate vsql;
        commit;
    end loop;

    exception
        when myexp then
            perform dbms_output.put_line('失败原因:'||SQLERRM);
            rollback;
end;
/

drop procedure if exists test_mt_pro_ddl_1(create_num int) cascade;
create or replace procedure test_mt_pro_ddl_1(create_num int)
as
    vsql varchar;
```

```
    my_cursor REFCURSOR;
    num int := 0;
    v_tbl_name varchar;
    func_num int;
begin
    perform dbms_output.serveroutput('t');
    test_at_pro_ddl_1(create_num);
    open my_cursor for select tablename from pg_tables where tablename like 'test_at_mt_temp_%';
    loop
        fetch my_cursor into v_tbl_name;
        exit when my_cursor%notfound;
        num := num + 1;
    end loop;
    perform dbms_output.put_line('num:'||num||',func_num:'||func_num);
    select 1/101981939864477696 into num;
    exception
        when others then
            perform dbms_output.put_line('失败原因:'||SQLERRM);
            open my_cursor for select tablename from pg_tables where tablename like 'test_at_mt_temp
_%';
            loop
                fetch my_cursor into v_tbl_name;
                exit when my_cursor%notfound;
                vsql := 'drop table if exists '||v_tbl_name||' cascade';
                execute immediate vsql;
            end loop;
            close my_cursor;
end;
/
call test_mt_pro_ddl_1(10);

declare
    pragma autonomous_transaction;
begin
    test_at_pro_ddl_1(10);
    test_mt_pro_ddl_1(10);
    commit;
end;
/

start TRANSACTION ISOLATION level REPEATABLE READ;
    call test_mt_pro_ddl_1(10);
    call test_at_pro_ddl_1(10);
commit;
```

dual

```
postgres=# select 1 as f1 from dual;
 f1
----
  1
(1 row)
```

sysdate

```
postgres=# select sysdate from dual;
```

结果为

```
     orcl_sysdate
---------------------
 2022-05-07 16:00:52
(1 row)
```

systimestamp

```
postgres=# select systimestamp from dual;
```

结果为：

```
        orcl_systimestamp
---------------------------------
 2022-05-07 15:55:59.318227 +08:00
(1 row)
```

merge into

MERGE INTO基于源表和目标表的关联结果，对目标表进行插入/更新操作。源表可以是表、视图或子查询。
关联条件支持非分布键。
示例1、

```
DROP TABLE IF EXISTS test1;
create table test1(id int primary key,name varchar2(10));
insert into test1 values(1,'test1');
insert into test1 values(2,'test1');
insert into test1 values(3,'test1');

DROP TABLE IF EXISTS test2;
create table test2(id int primary key,name varchar2(10));
insert into test2 values(2,'test2');
insert into test2 values(3,'test2');
insert into test2 values(4,'test2');
insert into test2 values(5,'test2');
#执行merge join
MERGE INTO test1 t
USING (
select * from test2
) t2 ON (t.id = t2.id)
WHEN MATCHED THEN UPDATE SET t.name =t2.name WHERE t.id = t2.id
WHEN NOT MATCHED THEN INSERT (id,name)VALUES (t2.id, t2.name) ;

postgres=# select * from test1;
```

结果为

```
 id |name
----+-------
  1 |test1
  2 |test2
  3 |test2
  4 |test2
  5 |test2
(5 rows)
```

connect by

CONNECT BY级联查询，常用于对具有树状结构的记录查询某一节点的所有子孙节点或所有祖辈节点，或通过CONNECT BY 结合LEVEL来构造数据。
LEVEL关键字，代表树形结构中的层级编号；第一层是数字1，第二层数字2，依次递增。

- 使用level实现1到5的序列

```
postgres=# select level from dual connect by level<=5;
 level
-------
     1
     2
     3
     4
     5
（4 rows）
```

- bom表

```
create table bom(fid number,cid number);
insert into bom values(0,1);
insert into bom values(1,2);
insert into bom values(1,3);
insert into bom values(2,4);
insert into bom values(4,5);

postgres=# select fid,cid,ltrim(sys_connect_by_path(cid,'->'),'->')
from  bom
start with fid = 0
connect by prior cid=fid;
```

结果为：

```
 fid| cid |   ltrim
-----+-----+------------
   0|   1 | 1
   1|   2 | 1->2
   1|   3 | 1->3
   2|   4 | 1->2->4
   4|   5 | 1->2->4->5
(5 rows)
```

- 多表join

```
create table cn_tb3(
        id integer ,
        id1 int ,
        emp_id number(18),
        lead_id number(18),
        col1 numeric,
        col2 numeric,
        col3 interval year to month,
        col4 interval year to month,
        col5 varchar2(20),
        col6 varchar2(20),
        col7 char(20) ,
        col8 char(20),
        col9 text,
        col10 text,
        col11 raw(20),
        col12 raw(20),
        col15 float,
        col16 float,
        emp_name varchar2(200),
        salary number(10,2),
        dept_no varchar2(8)
) partition by range (id) begin (1) step (5) partitions (50) distribute by shard(id) ;

-- 添加数据
insert into cn_tb3 values('1',0,'1',0,'1.3',0.0,INTERVAL '12-2' YEAR(2) TO MONTH,INTERVAL '1-2' YEAR(1) TO MONTH,'b1a','b0a','陕西省','宝鸡市','河南省','郑州市',utl_raw.cast_to_raw('1'),utl_raw.cast_to_raw('0'),'1.3',0.0,'king','1000000.00','001');
insert into cn_tb3 values('2',1,'2',1,'2.3',1.3,INTERVAL '1-2' YEAR(1) TO MONTH,INTERVAL '12-2' YEAR(2) TO MONTH,'b2a','b1a','新城区','陕西省','高新区','河南省',utl_raw.cast_to_raw('2'),utl_raw.cast_to_raw('1'),'2.3',1.3,'jack','50500.00','002');
insert into cn_tb3 values('3',1,'3',1,'3.3',1.3,INTERVAL '12' YEAR,INTERVAL '1-2' YEAR(1) TO MONTH,'b3a','b1a','大明宫','新城区','大明宫','高新区',utl_raw.cast_to_raw('3'),utl_raw.cast_to_raw('1'),'3.3',1.3,'arise','60000.00','003');
insert into cn_tb3 values('4',2,'4',2,'4.3',2.3,INTERVAL '12' MONTH,INTERVAL '12' YEAR ,'b4a','b2a','陕西省','大明宫','西安市','大明宫',utl_raw.cast_to_raw('4'),utl_raw.cast_to_raw('2'),'4.3',2.3,'scott','30000.00','002');
insert into cn_tb3 values('5',2,'5',2,'5.3',2.3,INTERVAL '2' MONTH(1),INTERVAL '12' MONTH,'b5a','b2a','碑林区','西安市','碑林区','西安市',utl_raw.cast_to_raw('5'),utl_raw.cast_to_raw('2'),'5.3',2.3,'tiger','25000.00','002');
insert into cn_tb3 values('6',3,'6',3,'6.3',3.3,INTERVAL '9' YEAR(1),INTERVAL '2' MONTH(1),'b6a','b3a','黄岩村','碑林区','黄岩村','碑林区',utl_raw.cast_to_raw('6'),utl_raw.cast_to_raw('3'),'6.3',3.3,'wudde','23000.00','003');
insert into cn_tb3 values('7',3,'7',3,'7.3',3.3,INTERVAL '10' YEAR(5),INTERVAL '9' YEAR(1),'b7a','b3a','陕西
```

```
省','黄岩村','陕西省','西安市',utl_raw.cast_to_raw('7'),utl_raw.cast_to_raw('3'),'7.3',3.3,'joker','21000.00','00
3');
insert into cn_tb3 values('1',6,'1',6,'1.3',6.3,INTERVAL '12-2' YEAR(2) TO MONTH,INTERVAL '10' YEAR
(5),'b1a','b6a','新城区','西安市','新城区','西安市',utl_raw.cast_to_raw('1'),utl_raw.cast_to_raw('6'),'1.3',6.3,'w
udde','23000.00','003');
insert into cn_tb3 values('1',5,'1',5,'1.3',5.3,INTERVAL '12-2' YEAR(2) TO MONTH,INTERVAL '200' MONT
H(9),'b1a','b5a','陕西省','新城区','河南省','新城区',utl_raw.cast_to_raw('1'),utl_raw.cast_to_raw('5'),'1.3',5.3,'j
oker','21000.00','001');

insert into cn_tb3 values('11',2,'11',3  ,'1.13', 1.3,INTERVAL '1-2' YEAR(1) TO MONTH,INTERVAL '200' M
ONTH(9),'b11a','b2a','陕西省','西安市', '陕西省','西安市',utl_raw.cast_to_raw('11'),utl_raw.cast_to_raw('10'),
'1.13', 1.3,'king','1000000.00','001');
insert into cn_tb3 values('21',11,'21',11,'3.13',1.13,'1-2',INTERVAL '1-2' YEAR(1) TO MONTH,'b21a','b11
a','宝鸡市','陕西省','新城区','西安市',utl_raw.cast_to_raw('21'),utl_raw.cast_to_raw('11'),'2.13',1.13,'jack','505
00.00','002');
insert into cn_tb3 values('31',11,'31',11,'4.13',3.13,'0-0','1-2','b31a','b11a','大明宫','宝鸡市','大明宫','新城
区',utl_raw.cast_to_raw('31'),utl_raw.cast_to_raw('11'),'3.13',1.13,'arise','60000.00','003');
insert into cn_tb3 values('41',21,'41',21,'5.13',3.13,'0-1','0-0','b41a','b21a','陕西省','西安市','陕西省','西安
市',utl_raw.cast_to_raw('41'),utl_raw.cast_to_raw('21'),'4.13',2.13,'scott','30000.00','002');
insert into cn_tb3 values('51',21,'51',21,'6.13',4.13,INTERVAL '200' MONTH(9),'0-1','b51a','b21a','碑林
区','大明宫','碑林区','西安市',utl_raw.cast_to_raw('51'),utl_raw.cast_to_raw('21'),'5.13',2.13,'tiger','25000.0
0','002');
insert into cn_tb3 values('61',31,'61',31,'7.13',5.13,INTERVAL '2' MONTH(1),INTERVAL '200' MONTH
(9),'b61a','b31a','黄岩村','碑林区','黄岩村','碑林区',utl_raw.cast_to_raw('61'),utl_raw.cast_to_raw('31'),'6.1
3',3.13,'wudde','23000.00','003');
insert into cn_tb3 values('71',31,'71',31,'8.13',6.13,INTERVAL '10' YEAR(5),INTERVAL '2' MONTH(1),'b71
a','b31a','陕西省','西安市','陕西省','西安市',utl_raw.cast_to_raw('71'),utl_raw.cast_to_raw('31'),'7.13',3.13,'jo
ker','21000.00','003');
insert into cn_tb3 values('11',61,'11',61,'1.13',6.13,'1-2',INTERVAL '10' YEAR(5),'b11a','b61a','新城区','黄岩
村','新城区','西安市',utl_raw.cast_to_raw('11'),utl_raw.cast_to_raw('61'),'1.13',6.13,'wudde','23000.00','00
3');
insert into cn_tb3 values('11',51,'11',51,'1.13',5.13,'1-2',INTERVAL '9' YEAR(1),'b11a','b51a','陕西省','新城
区','陕西省','新城区',utl_raw.cast_to_raw('11'),utl_raw.cast_to_raw('51'),'1.13',5.13,'joker','21000.00','001');
insert into cn_tb3 values('11',null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,nu
ll,null,null);

create table cn_tb4 as select * from cn_tb3;

select
   cn_tb3.id,
   cn_tb4.id1,
    level le
from
   cn_tb3 inner join cn_tb4
   on cn_tb3.id = cn_tb4.emp_id
   start with case when cn_tb3.id1 is null then 3 else cn_tb3.id1 end = 3
   connect by nocycle  cn_tb3.id=cn_tb4.id1 order by 1,2,3;
```

- connect by isleaf

```
create table cn_tb5(
    emp_id number,
    lead_id number,
    id integer ,
    id1 int ,
    col1 numeric,
    col2 numeric,
    col3 date,
    col4 date,
    col5 varchar2(20),
    col6 varchar2(20),
    col7 char(20) ,
    col8 char(20),
    col9 clob,
    col10 clob,
    col11 nclob,
    col12 nclob,
    col13 text,
    col14 text,
    col15 float,
    col16 float,
    emp_name varchar2(200),
    salary number(10,2),
    dept_no varchar2(8)
);

-- 添加数据
insert into cn_tb5 values('1',0,'1',0,'1.3',0.0,'2020-10-1','1999-12-1','b1a','b0a','陕西省','宝鸡市','b1a','b0a','河南省','郑州市','
insert into cn_tb5 values('2',1,'2',1,'2.3',1.3,'2020-10-2','2020-10-1','b2a','b1a','新城区','陕西省','b2a','b1a','高新区','河南省','
insert into cn_tb5 values('3',1,'3',1,'3.3',1.3,'2020-10-3','2020-10-2','b3a','b1a','大明宫','新城区','b3a','b1a','大明宫','高新区','
insert into cn_tb5 values('4',2,'4',2,'4.3',2.3,'2020-10-4','2020-10-3','b4a','b2a','陕西省','大明宫','b4a','b2a','西安市','大明宫','
insert into cn_tb5 values('5',2,'5',2,'5.3',2.3,'2020-10-5','2020-10-4','b5a','b2a','碑林区','西安市','b5a','b2a','碑林区','西安市','
insert into cn_tb5 values('6',3,'6',3,'6.3',3.3,'2020-10-6','2020-10-5','b6a','b3a','黄岩村','碑林区','b6a','b3a','黄岩村','碑林区','
insert into cn_tb5 values('7',3,'7',3,'7.3',3.3,'2020-10-7','2020-10-6','b7a','b3a','陕西省','黄岩村','b7a','b3a','陕西省','西安市','
insert into cn_tb5 values('1',6,'1',6,'1.3',6.3,'2020-10-1','2020-10-7','b1a','b6a','新城区','西安市','b1a','b6a','新城区','西安市','
```

```
insert into cn_tb5 values('1',5,'1',5,'1.3',5.3,'2020-10-1','2020-10-8','b1a','b5a','陕西省','新城区','b1a','b5a','河南省','新城区','

insert into cn_tb5 values('11',2,'11',3  ,'1.13', 1.3,'2020-10-11','1999-12-10','b11a','b2a','陕西省','西安市','b11a','b2a','陕西省','西安市','
insert into cn_tb5 values('21',11,'21',11,'3.13',1.13,'2020-11-12','2020-10-11','b21a','b11a','宝鸡市','陕西省','b31a','b11a','新城区','西安市','
insert into cn_tb5 values('31',11,'31',11,'4.13',3.13,'2020-11-13','2020-11-12','b31a','b11a','大明宫','宝鸡市','b31a','b11a','大明宫','新城区','
insert into cn_tb5 values('41',21,'41',21,'5.13',3.13,'2020-11-14','2020-11-13','b41a','b21a','陕西省','西安市','b31a','b31a','陕西省','西安市','
insert into cn_tb5 values('51',21,'51',21,'6.13',4.13,'2020-11-15','2020-11-14','b51a','b21a','碑林区','大明宫','b31a','b31a','碑林区','西安市','
insert into cn_tb5 values('61',31,'61',31,'7.13',5.13,'2020-11-16','2020-11-15','b61a','b31a','黄岩村','碑林区','b31a','b31a','黄岩村','碑林区','
insert into cn_tb5 values('71',31,'71',31,'8.13',6.13,'2020-11-17','2020-11-16','b71a','b31a','陕西省','西安市','b31a','b31a','陕西省','西安市','
insert into cn_tb5 values('11',61,'11',61,'1.13',6.13,'2020-11-11','2020-11-17','b11a','b61a','新城区','黄岩村','b31a','b31a','新城区','西安市','
insert into cn_tb5 values('11',51,'11',51,'1.13',5.13,'2020-11-11','2020-11-18','b11a','b51a','陕西省','新城区','b11a','b31a','陕西省','新城区','
insert into cn_tb5 values('11',null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null);

create temp table cn_tb6 as select * from cn_tb5;

select emp_id,emp_name,lead_id,salary,connect_by_isleaf
    from cn_tb5
    start with lead_id=0
    connect by nocycle prior emp_id=lead_id;
```

pivot

pivot行转列

示例：

1、单聚合函数

```
create table scores(student varchar(10) not null, course varchar(10) not null,score int not null);
insert into scores values('张三','语文',78);
insert into scores values('张三','语文',98);
insert into scores values('张三','数学',79);
insert into scores values('张三','英语',80);
insert into scores values('张三','物理',81);
insert into scores values('李四','语文',65);
insert into scores values('李四','数学',75);
insert into scores values('李四','英语',85);
insert into scores values('李四','物理',95);

select
    t.*,
    (t.语+t.数+t.外+t.物) as total
from
(
    select
        *
    from
        scores pivot
        (
            max(score) for course in ('语文' as 语 , '数学' as 数, '英语' as 外,'物理' as 物)
        )
) t;
```

结果为：

```
 student | 语 | 数 | 外 | 物 | total
---------+----+----+----+----+-------
 李四    | 65 | 75 | 85 | 95 |   320
 张三    | 98 | 79 | 80 | 81 |   338
(1 rows)
```

2、多聚合函数

```
drop table if exists t_usr_20221226;
create table t_usr_20221226
```

```
(name varchar2(20),
 score int,
 rank int,
 class varchar2(20)
);
insert into t_usr_20221226 values('a',20,1,'math');
insert into t_usr_20221226 values('a',22,2,'phy');
insert into t_usr_20221226 values('b',23,3,'phy');
insert into t_usr_20221226 values('b',21,4,'math');
insert into t_usr_20221226 values('c',22,3,'phy');
insert into t_usr_20221226 values('c',24,2,'math');
insert into t_usr_20221226 values('d',25,5,'math');
insert into t_usr_20221226 values('d',23,4,'phy');
drop table if exists t_usr2_20221226;
create table t_usr2_20221226
(sid int,
 name varchar2(20),
 school varchar2(20)
);
insert into t_usr2_20221226 values(1, 'a', 'a school');
insert into t_usr2_20221226 values(2, 'b', 'b school');
insert into t_usr2_20221226 values(3, 'c', 'c school');
insert into t_usr2_20221226 values(4, 'd', 'd school');
```

结果为：

```
name | rank | 'math'_s | 'math'_c | 'phy'_s | 'phy'_c
------+------+----------+----------+---------+---------
 a   |    1 |       20 |        1 |         |       0
 a   |    2 |          |        0 |      22 |       1
 b   |    3 |          |        0 |      23 |       1
 b   |    4 |       21 |        1 |         |       0
 c   |    2 |       24 |        1 |         |       0
 c   |    3 |          |        0 |      22 |       1
 d   |    4 |          |        0 |      23 |       1
 d   |    5 |       25 |        1 |         |       0

(8 行记录)
```

unpivot

```
postgres=# create table t_unpivot(f1 int,f2 int,f3 int,f4 int);
CREATE TABLE
postgres=# insert into t_unpivot values(1,12,13,14);
INSERT 0 1
postgres=# insert into t_unpivot values(2,22,23,24);
INSERT 0 1

postgres=# select * from t_unpivot unpivot ( value for fields in(f2,f3,f4));
 f1 | value | fields
----+-------+--------
  1 |   12 | F2
  1 |   13 | F3
  1 |   14 | F4
  2 |   22 | F2
  2 |   23 | F3
  2 |   24 | F4
(6 rows)
```

offset x rows fetch next x rows only

```
postgres=# create table t1(f1 int,f2 int);
CREATE TABLE
postgres=# insert into t1 values(1,1),(2,2);
COPY 2

postgres=# select * from t1 order by f1 offset 0 rows fetch next 1 rows only;
 f1 |f2
----+----
  1|  1
(1 row)

postgres=# select * from t1 order by f1 offset 1 rows fetch next 1 rows only;
 f1 |f2
----+----
  2|  2
(1 row)
```

只读事务获取序列值

```
postgres=# CREATE SEQUENCE seq_userid;
CREATE SEQUENCE
postgres=# select nextval('seq_userid');
 nextval
---------
       1
(1 row)

postgres=# begin transaction read only;
BEGIN
postgres=# select nextval('seq_userid');
 nextval
---------
       2
(1 row)
```

''等同于null

```
postgres=# select '' is null as empty_is_null;
 empty_is_null
---------------
 t
(1 row)
```

创建view使用with read only

```
postgres=# create table t1(f1 int,f2 int);
CREATE TABLE
postgres=# insert into t1 values(1,1);
INSERT 0 1
postgres=# create or replace view t1_view as select * from t1 with read only;
CREATE VIEW
postgres=# \d+ t1_view
                    View"public.t1_view"
 Column | Type   | Collation | Nullable |Default | Storage | Description
--------+---------+-----------+----------+---------+---------+-------------
 f1     |integer |           |          |        | plain   |
 f2     |integer |           |          |        | plain   |
View definition:
 SELECT t1.f1,
   t1.f2
  FROM t1;
Options: readonly=true
```

insert all

```
postgres=# create table t2 (f1 int,f2 int);
create table t3(f1 int,f2 int);
CREATE TABLE
CREATE TABLE
postgres=# insert all into t2 values(f1,f2) into t3 values(f1,f2) select 1 as f1,1 as f2;
INSERT 0 2
```

q’转义字符

一个Q-quote的表达式，用来简化SQL或PLSQL中字符串的表示。

示例：

```
 select q'qMy "good" Name''q' names from dual;
```

匿名块

没有名称的块是匿名块。匿名块仅供一次性使用。
示例：

```
create tablet5(f1 integer,f2 varchar(10));
insert intot5 values(1,'tbase1');
insert intot5 values(2,'tbase2');

DECLARE
TYPEname_tbl IS TABLE OF VARCHAR2(50);
emp_name name_tbl;
BEGIN
SELECT f2
    BULK COLLECT INTO emp_name
    FROM t5;
FOR i INemp_name.FIRST .. emp_name.LAST
   LOOP
   perform Dbms_output.put_line(i);
   END LOOP;

END;
```

关于插件
插件查看，添加和删除
查看数据库加载了那些插件

```
postgres=# SELECT e.extname AS "Name", e.extversion AS "Version", n.nspname AS "Schema", c.descri
ption AS "Description" FROM pg_catalog.pg_extension e LEFT JOIN pg_catalog.pg_namespace n ON n.
oid = e.extnamespace LEFT JOIN pg_catalog.pg_description c ON c.objoid = e.oid AND c.classoid = 'pg
_catalog.pg_extension'::pg_catalog.regclass ORDER BY 1;
       Name        | Version |   Schema   |                    Description
-------------------+---------+------------+------------------------------------------------------------
 pageinspect       | 1.1     | public     | inspect the contents of database pages at a low level
 pg_errcode_stat   | 1.1     | public     | track error code of all processes
 pg_stat_statements | 1.1     | public     | track execution statistics of all SQL statements executed
 plpgsql           | 1.0     | pg_catalog | PL/pgSQL procedural language
 shard_statistic   | 1.0     | public     | tools for get shard statistic
(5 rows)
```

添加插件

```
postgres=#create extension "uuid-ossp" with schema tbase;
CREATEEXTENSION
```

上面的语句把"uuid-ossp"创建到模式tbase下。

删除插件

```
postgres=# drop extension "uuid-ossp" ;
DROP EXTENSION
```

插件uuid-ossp使用
uuid-ossp功能介绍及添加方法

功能介绍

uuid-ossp模块提供函数使用几种标准算法之一产生通用唯一标识符（UUID）。还提供产生某些特殊 UUID 常量的函数。该模块提供的功能函数可用于替代序列的，而且性能比序列更好。

操作步骤

1. 给数据库添加uuid-ossp插件。

```
postgres=# create extension "uuid-ossp" with schema tbase;
CREATE EXTENSION
```

uuid常量函数

uuid_nil()

这是一个"nil" UUID 常量。

```
postgres=# select uuid_nil();
              uuid_nil
--------------------------------------
 00000000-0000-0000-0000-000000000000
(1 row)
```

uuid_ns_dns()

为 UUID 指定 DNS 名字空间的常量。

```
postgres=# select uuid_ns_dns();
             uuid_ns_dns
--------------------------------------
 6ba7b810-9dad-11d1-80b4-00c04fd430c8
(1 row)
```

uuid_ns_url()

为 UUID 指定 URL 名字空间的常量。

```
postgres=# select uuid_ns_url();
             uuid_ns_url
--------------------------------------
 6ba7b811-9dad-11d1-80b4-00c04fd430c8
(1 row)
```

uuid_ns_oid()

为 UUID 指定 ISO 对象标识符（OID）名字空间的常量。

```
postgres=# select uuid_ns_oid();
              uuid_ns_oid
--------------------------------------
 6ba7b812-9dad-11d1-80b4-00c04fd430c8
(1 row)
```

uuid_ns_x500()

为 UUID 指定 X.500 可识别名（DN）名字空间的常量。

```
postgres=# select uuid_ns_x500();
              uuid_ns_x500
--------------------------------------
 6ba7b814-9dad-11d1-80b4-00c04fd430c8
(1 row)
```

uuid生成函数
uuid_generate_v1()

这个函数产生一个版本 1 的 UUID。这涉及到计算机的 MAC 地址和一个时间戳。

```
postgres=# select uuid_generate_v1(),clock_timestamp();
          uuid_generate_v1           |       clock_timestamp
-------------------------------------+-------------------------------
 e909c7f0-c36b-11e7-984d-525400efe0ca | 2017-11-07 11:29:49.67391+08
(1 row)

postgres=# select uuid_generate_v1(),clock_timestamp();
          uuid_generate_v1           |       clock_timestamp
-------------------------------------+-------------------------------
 e9f24458-c36b-11e7-9f81-525400efe0ca | 2017-11-07 11:29:51.197516+08
(1 row)
```

uuid_generate_v1mc()

这个函数产生一个版本 1 的 UUID，但是使用一个随机广播 MAC 地址而不是该计算机真实的 MAC 地址。

```
postgres=# select uuid_generate_v1mc();
         uuid_generate_v1mc
--------------------------------------
 1538d848-c36c-11e7-9043-5749d89bc820
(1 row)
```

uuid_generate_v3(namespace uuid, name text)

这个函数使用指定的输入名称在给定的名字空间中产生一个版本 3 的 UUID，名称参数将使用 MD5 进行哈希，因此从产生的 UUID 中得不到明文。

```
postgres=# SELECT uuid_generate_v3(uuid_ns_url(), '');
          uuid_generate_v3
--------------------------------------
 29ba82b2-955d-3401-8cc3-bb88bf063a22
(1 row)

postgres=# SELECT uuid_generate_v3(uuid_ns_dns(), '');
          uuid_generate_v3
--------------------------------------
 29ba82b2-955d-3401-8cc3-bb88bf063a22
(1 row)
```

uuid_generate_v4()

这个函数产生一个版本 4 的 UUID，它完全从随机数产生。

```
postgres=# select uuid_generate_v4();
          uuid_generate_v4
--------------------------------------
 dfc68e17-6b97-496e-a992-531aca74ef18
(1 row)

postgres=# select uuid_generate_v4();
          uuid_generate_v4
--------------------------------------
 e96b64c2-cf1e-423b-845c-541e9b3901a3
(1 row)
```

uuid_generate_v5(namespace uuid, name text)

这个函数产生一个版本 5 的 UUID，它和版本 3 的 UUID 相似，但是采用的是 SHA-1 作为哈希方法。版本 5 比版本 3 更好，因为 SHA-1 被认为比 MD5 更安全。

```
postgres=# SELECT uuid_generate_v5(uuid_ns_url(), '');
          uuid_generate_v5
--------------------------------------
 d1bcb48f-3572-568e-8b1c-8b26f6b776a7
(1 row)

postgres=# SELECT uuid_generate_v5(uuid_ns_dns(), '');
          uuid_generate_v5
--------------------------------------
 a4e01443-3f13-56a8-a470-72340d900d30
(1 row)
```

在数据表中使用uuid默认值

```
postgres=# create table t_uuid(f1 uuid not null default uuid_generate_v1(),f2 varchar(256) );
CREATE TABLE
postgres=# \d+ t_uuid
                                  Table "public.t_uuid"
 Column |         Type          |              Modifiers              | Storage  | Stats target | Description
--------+-----------------------+-------------------------------------+----------+--------------
+-------------
 f1     | uuid                  | not null default uuid_generate_v1() | plain    |              |
 f2     | character varying(256) |                                     | extended |              |
Has OIDs: no
Distribute By SHARD(f2)
        Location Nodes: ALL DATANODES
postgres=# insert into t_uuid (f2) values(uuid_generate_v4());
INSERT 0 1
postgres=# select * from t_uuid;
                f1                |                f2
----------------------------------+----------------------------------
 75dfb6b0-c382-11e7-9f82-525400efe0ca | 2b99799c-d77e-4023-81db-3e5c6ad901b0
(1 row)
```

uuid各函数性能对比

- 数据表结构。

```
postgres=# \d+ t_uuid
                          Table "public.t_uuid"
 Column |         Type          |              Modifiers              | Storage  | Stats target | Description
--------+-----------------------+-------------------------------------+----------+--------------
+-------------
 f1     | uuid                  | not null default uuid_generate_v1() | plain    |              |
 f2     | character varying(256) |                                     | extended |              |
Has OIDs: no
Distribute By SHARD(f2)
       Location Nodes: ALL DATANODES
```

- uuid_generate_v3函数。

```
postgres=# insert into t_uuid(f1,f2) select uuid_generate_v3(uuid_ns_dns(), ''),uuid_generate_v3(uuid_n
s_dns(), 'http://tbase.qq.com') from generate_series(1,10000) as t;
INSERT 0 10000
Time: 58.407 ms
```

- uuid_generate_v5函数。

```
postgres=# insert into t_uuid(f1,f2) select uuid_generate_v5(uuid_ns_dns(), ''),uuid_generate_v5(uuid_n
s_dns(), 'http://tbase.qq.com') from generate_series(1,10000) as t;
INSERT 0 10000
Time: 47.049 ms
```

- uuid_generate_v1函数。

```
postgres=# insert into t_uuid(f1,f2) select uuid_generate_v1(),uuid_generate_v1() from generate_series
(1,10000) as t;
INSERT 0 10000
Time: 111.204 ms
```

- uuid_generate_v4函数。

```
postgres=# insert into t_uuid(f1,f2) select uuid_generate_v4(),uuid_generate_v4() from generate_series
(1,10000) as t;
INSERT 0 10000
```

Time: 137.062 ms

- uuid_generate_v1mc函数。

postgres=# insert into t_uuid(f1,f2) select uuid_generate_v1mc(),uuid_generate_v1mc() from generate_series(1,10000) as t;
INSERT 0 10000
Time: 125.663 ms

- 测试数据对比

V1	v1mc	V4	V3	V5
111.204 ms	125.663 ms	137.062 ms	58.407 ms	47.049 ms

可以看出V5的性能是最好的。

uuid与serial 在TDSQL PG中性能对比

- uuid插入1万条数据。

```
postgres=#  \d+ t_uuid
                                  Table "public.t_uuid"
 Column |         Type          |           Modifiers            | Storage  | Stats target | Description
--------+-----------------------+--------------------------------+----------+--------------+-------------
 f1     | uuid                  | not null default uuid_generate_v1() | plain    |              |
 f2     | character varying(256) |                                | extended |              |
Has OIDs: no
Distribute By SHARD(f2)
        Location Nodes: ALL DATANODES

postgres=# insert into t_uuid(f2) select t from generate_series(1,1000000) as t;
INSERT 0 1000000
Time: 4869.140 ms (00:04.869)
```

- serial插入1万条数据。

```
postgres=# \d+ t_serial
                                     Table "public.t_serial"
 Column |         Type          |                     Modifiers                     | Storage  | Stats target | Description
--------+-----------------------+---------------------------------------------------+----------+--------------+-------------
 f1     | integer               | not null default nextval('t_serial_f1_seq'::regclass) | plain    |              |
 f2     | character varying(256) |                                                   | extended |              |
Has OIDs: no
Distribute By SHARD(f1)
        Location Nodes: ALL DATANODES

postgres=#  insert into t_serial(f2) select t from generate_series(1,1000000) as t;
INSERT 0 1000000
Time: 3683.533 ms (00:03.684)
```

- uuid pgbench结果。

```
pghost: 172.16.0.42 pgport: 11016 nclients: 32 duration: 600 dbName: postgres
transaction type: insert_t_uuid.sql
scaling factor: 1
query mode: prepared
number of clients: 32
```

```
number of threads: 1
duration: 600 s
number of transactions actually processed: 531667
latency average = 36.166 ms
tps = 884.807291 (including connections establishing)
tps = 884.813103 (excluding connections establishing)
script statistics:
 - statement latencies in milliseconds:
         0.013  \set id random(1, 10000000)
        36.094  insert into t_uuid(f2) values(:id::text) ;
```

- erial pgbench结果。

```
transaction type: insert_t_serial.sql
scaling factor: 1
query mode: prepared
number of clients: 32
number of threads: 1
duration: 600 s
number of transactions actually processed: 493799
latency average = 38.889 ms
tps = 822.864578 (including connections establishing)
tps = 822.869984 (excluding connections establishing)
script statistics:
 - statement latencies in milliseconds:
         0.013  \set id random(1, 10000000)
        38.861  insert into t_serial(f2) values(:id::text) ;
```

- 测试数据对比 。

uuid用时	Serial用时	百分比	Uuid pgbench	Serial pgbench	百分比
4.869s	3.684s	75.66%	884	822	92.98%

占用空间对比

- UUID表。

```
postgres=# \d+ t_uuid
                                    Table "public.t_uuid"
 Column |         Type          | Collation | Nullable |      Default       | Storage  | Stats target | Description
--------+-----------------------+-----------+----------+--------------------+----------+--------------+-------------
 f1     | uuid                  |           | not null | uuid_generate_v1() | plain    |              |
 f2     | character varying(256) |           |          |                    | extended |              |
Distribute By: SHARD(f2)
Location Nodes: ALL DATANODES

postgres=# select count(1),pg_size_pretty(pg_table_size('t_uuid')) from t_uuid;
  count  | pg_size_pretty
---------+----------------
 1000000 | 73 MB
(1 row)
```

- Serial表。

```
postgres=# \d+ t_serial
                                      Table "public.t_serial"
 Column |         Type          | Collation | Nullable |              Default               | Storage  | Stats target | Des
cription
--------+-----------------------+-----------+----------+------------------------------------+----------+--------------+----------
 id     | integer               |           | not null | nextval('t_serial_id_seq'::regclass) | plain    |              |
 f2     | character varying(256) |           |          |                                    | extended |              |
Distribute By: SHARD(id)
Location Nodes: ALL DATANODES

postgres=# select count(1),pg_size_pretty(pg_table_size('t_serial')) from t_serial;
  count  | pg_size_pretty
---------+----------------
 1000000 | 66 MB
(1 row)

Time: 186.737 ms
```

- 测试数据对比。

Uuid/10000条记录	Serial/10000条记录	占用比
73MB	66MB	1.1倍

使用uuid做为分布列的方案

目前uuid类型字段不能做为分布列，提示如下：

```
postgres=# create table t_uuid(f1 uuid );
ERROR:  No appropriate column can be used as distribute key because of data type.
```

解决方案如下：

```
postgres=# create table t_uuid(f1 varchar(36) default uuid_nil()::text );
CREATE TABLE
postgres=#
```

使用建议

性能上序列和uuid相关不大，但serial每次都需要与gtm通信，会加大gtm的通信压力，所以尽可能使用uuid来代替serial。

插件pg_stat_statements使用
pg_stat_statements功能介绍及添加方法

功能介绍

pg_stat_statements模块提供一种方法追踪一个服务器所执行的所有 SQL 语句的执行统计信息。该模块必须通过在postgresql.conf的shared_preload_libraries中增加pg_stat_statements来载入，因为它需要额外的共享内存。增加或移除该模块需要服务器重启才能生效。当pg_stat_statements被载入时，它会跟踪该服务器的所有数据库的统计信息。该模块提供了一个视图pg_stat_statements以及函数pg_stat_statements_reset 和pg_stat_statements用于访问和操纵这些统计信息。

操作步骤

1. 给数据库添加pg_stat_statements插件。

```
postgres=#create extension pg_stat_statements with schema tbase;
CREATE EXTENSION
```

获取执行次数最多的语句

```
postgres=# select
    pg_authid.rolname as rolname,
    pg_database.datname as datname,
    pg_stat_statements.query,
    pg_stat_statements.calls,
    pg_stat_statements.total_time,
    round((pg_stat_statements.total_time/pg_stat_statements.calls)::numeric,6) as runtime_of_once
from
    pg_stat_statements
    inner join pg_authid on pg_authid.oid=pg_stat_statements.userid
    inner join pg_database on pg_database.oid= pg_stat_statements.dbid
order by
    pg_stat_statements.calls desc
limit 10;
```

获取执行总时间最长的语句

```
postgres=# select
    pg_authid.rolname as rolname,
    pg_database.datname as datname,
    pg_stat_statements.query,
    pg_stat_statements.calls,
    pg_stat_statements.total_time,
    pg_stat_statements.total_time/pg_stat_statements.calls AS onestime
from
    pg_stat_statements
    inner join pg_authid on pg_authid.oid=pg_stat_statements.userid
    inner join pg_database on pg_database.oid= pg_stat_statements.dbid
order by
    pg_stat_statements.total_time desc
limit 10;
```

获取每句平均执行时间最长的语句

```
postgres=# select
    pg_authid.rolname as rolname,
    pg_database.datname as datname,
    pg_stat_statements.query,
    pg_stat_statements.calls,
    pg_stat_statements.total_time,
    pg_stat_statements.total_time/pg_stat_statements.calls as onestime
from
    pg_stat_statements
    inner join pg_authid on pg_authid.oid=pg_stat_statements.userid
    inner join pg_database on pg_database.oid= pg_stat_statements.dbid
order by
    pg_stat_statements.total_time/pg_stat_statements.calls desc
limit 10;
```

获取buffer读最多的语句

```
postgres=# select
    pg_authid.rolname as rolname,
    pg_database.datname as datname,
    pg_stat_statements.query,
    pg_stat_statements.shared_blks_hit,
    pg_stat_statements.shared_blks_read,
    (pg_stat_statements.shared_blks_hit+pg_stat_statements.shared_blks_read) as all_blks,
    pg_stat_statements.calls,
    pg_stat_statements.total_time,
    pg_stat_statements.total_time/pg_stat_statements.calls
from
    pg_stat_statements
    inner join pg_authid on pg_authid.oid=pg_stat_statements.userid
    inner join pg_database on pg_database.oid= pg_stat_statements.dbid
order by
    pg_stat_statements.shared_blks_hit+pg_stat_statements.shared_blks_read desc
limit 10;
```

插件pg_trgm使用
pg_trgm功能介绍及添加方法

功能介绍

前模糊，后模糊，前后模糊，正则匹配都属于文本搜索领域常见的需求。TDSQL PG在文本搜索领域除了全文检索，还有TRGM。 对于前模糊和后模糊，TDSQL PG与其他数据库一样，可以使用btree来加速。 对于前后模糊和正则匹配，则可以使用TRGM，TRGM是一个非常强的插件，对这类文本搜索场景性能提升非常有效，100万左右的数据量，性能提升有100倍以上。

添加方法

```
create extension pg_trgm;
```

测试环境准备

```
postgres=# create table t_trgm (id int,trgm text,no_trgm text) ;
CREATE TABLE

postgres=# \timing
Timing is on.

postgres=#  insert into t_trgm select t,md5(t::text),md5(t::text) from generate_series(1,1000000) as t;
INSERT 0 1000000
Time: 4239.598 ms (00:04.240)postgres=#
```

gist索引测试
创建索引消耗时间

```
postgres=# create index t_trgm_trgm_idx on t_trgm using gist(trgm gist_trgm_ops);
CREATE INDEX
Time: 24974.600 ms (00:24.975)
postgres=#
postgres=# vacuum ANALYZE t_trgm;
VACUUM
Time: 551.989 ms
```

索引占用空间

```
postgres=# select pg_size_pretty(pg_indexes_size('t_trgm'));
 pg_size_pretty
----------------
 178 MB
(1 row)
```

模糊查询测试

- 返回记录数多。

```
postgres=# explain analyze select * from t_trgm where trgm ilike '%67%';
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0) (actual time=126.483..1172.196
rows=114475 loops=1)
   Node/s: dn001, dn002
 Planning time: 0.055 ms
 Execution time: 1197.320 ms
(4 rows)

postgres=# explain analyze select * from t_trgm where no_trgm ilike '%67%';
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0) (actual time=6.679..965.383 ro
ws=114475 loops=1)
   Node/s: dn001, dn002
 Planning time: 0.056 ms
 Execution time: 989.840 ms
(4 rows)
```

使用gist索引开销反而更大。

- 返回记录比较少。

```
postgres=# explain analyze select * from t_trgm where trgm ilike '%67a5%';
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0) (actual time=106.522..118.693 r
ows=481 loops=1)
   Node/s: dn001, dn002
 Planning time: 0.074 ms
 Execution time: 118.831 ms
(4 rows)
postgres=# explain analyze select * from t_trgm where no_trgm ilike '%67a5%';
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0) (actual time=190.056..502.639 r
ows=481 loops=1)
   Node/s: dn001, dn002
```

```
 Planning time: 0.063 ms
 Execution time: 502.756 ms
(4 rows)
Time: 503.887 ms
```

过滤返回记录少，使用gist索引提高性能比较明显。

数据导入时间测试

```
postgres=# truncate table t_trgm ;
TRUNCATE TABLE
Time: 78.705 ms
postgres=# insert into t_trgm select t,md5(t::text),md5(t::text) from generate_series(1,1000000) as t;
INSERT 0 1000000
Time: 29061.725 ms (00:29.062)
```

pgbench并发查询测试

```
[tbase@VM_0_37_centos pgbench]$ cat select_t_trgm.sql
\set id random(1, 1000000)
select * from t_trgm where trgm ilike '%'||md5(:id)||'%';
[tbase@VM_0_37_centos pgbench]$ pgbench -h 127.0.0.1 -p 11008 -d postgres -U tbase -c 4 -j 1 -n -
M prepared -T 60 -r -f select_t_trgm.sql > select_t_trgm.txt 2>&1
[tbase@VM_0_37_centos pgbench]$ tail -20 select_t_trgm.txt
client 0 receiving
client 1 receiving
client 2 receiving
client 0 receiving
client 3 receiving
pghost: 127.0.0.1 pgport: 11008 nclients: 4 duration: 60 dbName: postgres
transaction type: select_t_trgm.sql
scaling factor: 1
query mode: prepared
number of clients: 4
number of threads: 1
duration: 60 s
number of transactions actually processed: 14150
latency average = 16.965 ms
tps = 235.773228 (including connections establishing)
tps = 235.794290 (excluding connections establishing)
script statistics:
 - statement latencies in milliseconds:
         0.013  \set id random(1, 1000000)
        16.944  select * from t_trgm where trgm ilike '%'||md5(:id)||'%';
[tbase@VM_0_37_centos pgbench]$ cat select_t_trgm.sql
\set id random(1, 1000000)
select * from t_trgm where trgm ilike '%'||substring(md5(:id) from 3 for 6)||'%';
[tbase@VM_0_37_centos pgbench]$ pgbench -h 127.0.0.1 -p 11008 -d postgres -U tbase -c 4 -j 1 -n -
M prepared -T 60 -r -f select_t_trgm.sql > select_t_trgm.txt 2>&1
[tbase@VM_0_37_centos pgbench]$ tail -20  select_t_trgm.txt
client 3 receiving
client 2 receiving
client 1 receiving
client 0 receiving
client 3 receiving
pghost: 127.0.0.1 pgport: 11008 nclients: 4 duration: 60 dbName: postgres
transaction type: select_t_trgm.sql
scaling factor: 1
query mode: prepared
number of clients: 4
number of threads: 1
```

```
duration: 60 s
number of transactions actually processed: 2365
latency average = 101.642 ms
tps = 39.353883 (including connections establishing)
tps = 39.357547 (excluding connections establishing)
script statistics:
 - statement latencies in milliseconds:
         0.015  \set id random(1, 1000000)
       101.528  select * from t_trgm where trgm ilike '%'||substring(md5(:id) from 3 for 6)||'%';
```

pgbench并发写入测试

```
[tbase@VM_0_37_centos pgbench]$ cat insert_t_trgm.sql
\set id random(1, 1000000)
insert into t_trgm values(:id,:id,:id) ;
[tbase@VM_0_37_centos pgbench]$ pgbench -h 127.0.0.1 -p 11008 -d postgres -U tbase -c 4 -j 1 -n -
M prepared -T 60 -r -f insert_t_trgm.sql > insert_t_trgm.txt 2>&1
[tbase@VM_0_37_centos pgbench]$ tail -20 insert_t_trgm.txt
client 0 receiving
client 1 receiving
client 3 receiving
client 2 receiving
client 0 receiving
pghost: 127.0.0.1 pgport: 11008 nclients: 4 duration: 60 dbName: postgres
transaction type: insert_t_trgm.sql
scaling factor: 1
query mode: prepared
number of clients: 4
number of threads: 1
duration: 60 s
number of transactions actually processed: 58884
latency average = 4.077 ms
tps = 981.188175 (including connections establishing)
tps = 981.285101 (excluding connections establishing)
script statistics:
 - statement latencies in milliseconds:
        0.011  \set id random(1, 1000000)
        4.063  insert into t_trgm values(:id,:id,:id) ;
```

gin索引测试
创建索引消耗时间

```
postgres=# drop index t_trgm_trgm_idx;
DROP INDEX
Time: 55.954 ms
postgres=# create index t_trgm_trgm_idx on t_trgm using gin(trgm gin_trgm_ops);
CREATE INDEX
Time: 16648.549 ms (00:16.649)
postgres=#
```

索引占用空间

```
postgres=# select pg_size_pretty(pg_indexes_size('t_trgm'));
 pg_size_pretty
----------------
 74 MB
(1 row)
```

模糊查询测试

- 返回记录数多。

```
postgres=# explain analyze select * from t_trgm where trgm ilike '%67%';
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0) (actual time=3.603..958.097 rows=114475 loops=1)
   Node/s: dn001, dn002
 Planning time: 0.061 ms
 Execution time: 985.647 ms
(4 rows)

Time: 986.631 ms

postgres=# explain analyze select * from t_trgm where no_trgm ilike '%67%';
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0) (actual time=2.890..947.736 rows=114475 loops=1)
   Node/s: dn001, dn002
 Planning time: 0.066 ms
 Execution time: 973.220 ms
(4 rows)
Time: 974.374 ms
```

使用gin索引与不使用性能不相上下。

- 返回记录比较少。

```
postgres=# explain analyze select * from t_trgm where trgm ilike '%67a5%';
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0) (actual time=3.001..4.178 rows=481 loops=1)
   Node/s: dn001, dn002
 Planning time: 0.067 ms
 Execution time: 4.300 ms
(4 rows)

Time: 5.212 ms
```

```
postgres=# explain analyze select * from t_trgm where no_trgm ilike '%67a5%';
                                                  QUERY PLAN
---------------------------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0) (actual time=174.435..524.049 r
ows=481 loops=1)
   Node/s: dn001, dn002
 Planning time: 0.069 ms
 Execution time: 524.207 ms
(4 rows)
Time: 525.226 ms
```

过滤返回记录少，使用gin索引提高性能100倍，效果非常的好。

数据导入时间测试

```
postgres=# truncate table t_trgm ;
TRUNCATE TABLE
Time: 43.750 ms
postgres=#  insert into t_trgm select t,md5(t::text),md5(t::text) from generate_series(1,1000000) as t;
INSERT 0 1000000
Time: 36371.813 ms (00:36.372)
```

pgbench并发查询测试

```
[tbase@VM_0_37_centos pgbench]$ cat select_t_trgm.sql
\set id random(1, 1000000)
select * from t_trgm where trgm ilike '%'||md5(:id)||'%';
[tbase@VM_0_37_centos pgbench]$ pgbench -h 127.0.0.1 -p 11008 -d postgres -U tbase -c 4 -j 1 -n -
M prepared -T 60 -r -f select_t_trgm.sql > select_t_trgm.txt 2>&1
[tbase@VM_0_37_centos pgbench]$ tail -20  select_t_trgm.txt
client 2 receiving
client 1 receiving
client 3 receiving
client 0 receiving
client 2 receiving
pghost: 127.0.0.1 pgport: 11008 nclients: 4 duration: 60 dbName: postgres
transaction type: select_t_trgm.sql
scaling factor: 1
query mode: prepared
number of clients: 4
number of threads: 1
duration: 60 s
number of transactions actually processed: 13352
latency average = 17.982 ms
tps = 222.445724 (including connections establishing)
tps = 222.468993 (excluding connections establishing)
script statistics:
 - statement latencies in milliseconds:
         0.012  \set id random(1, 1000000)
        17.959  select * from t_trgm where trgm ilike '%'||md5(:id)||'%';
```

pgbench并发写入测试

```
[tbase@VM_0_37_centos pgbench]$ cat select_t_trgm.sql
\set id random(1, 1000000)
select * from t_trgm where trgm ilike '%'||substring(md5(:id) from 3 for 6)||'%';
[tbase@VM_0_37_centos pgbench]$ pgbench -h 127.0.0.1 -p 11008 -d postgres -U tbase -c 4 -j 1 -n -
M prepared -T 60 -r -f select_t_trgm.sql > select_t_trgm.txt 2>&1
[tbase@VM_0_37_centos pgbench]$ tail -20 select_t_trgm.txt
client 1 receiving
client 0 receiving
client 3 receiving
client 1 receiving
client 2 receiving
pghost: 127.0.0.1 pgport: 11008 nclients: 4 duration: 60 dbName: postgres
transaction type: select_t_trgm.sql
scaling factor: 1
query mode: prepared
number of clients: 4
number of threads: 1
duration: 60 s
number of transactions actually processed: 57280
latency average = 4.190 ms
tps = 954.570838 (including connections establishing)
tps = 954.650960 (excluding connections establishing)
script statistics:
 - statement latencies in milliseconds:
         0.011  \set id random(1, 1000000)
         4.177  select * from t_trgm where trgm ilike '%'||substring(md5(:id) from 3 for 6)||'%';
```

无索引字段测试
pgbench并发查询测试

```
[tbase@VM_0_37_centos pgbench]$ cat select_t_trgm.sql
\set id random(1, 1000000)
select * from t_trgm where no_trgm ilike '%'||substring(md5(:id) from 3 for 6)||'%';
[tbase@VM_0_37_centos pgbench]$ pgbench -h 127.0.0.1 -p 11008 -d postgres -U tbase -c 4 -j 1 -n -
M prepared -T 60 -r -f select_t_trgm.sql > select_t_trgm.txt 2>&1
[tbase@VM_0_37_centos pgbench]$ tail -20 select_t_trgm.txt
client 3 receiving
client 2 receiving
client 1 receiving
client 3 receiving
client 0 receiving
pghost: 127.0.0.1 pgport: 11008 nclients: 4 duration: 60 dbName: postgres
transaction type: select_t_trgm.sql
scaling factor: 1
query mode: prepared
number of clients: 4
number of threads: 1
duration: 60 s
number of transactions actually processed: 212
latency average = 1147.939 ms
tps = 3.484507 (including connections establishing)
tps = 3.484812 (excluding connections establishing)
script statistics:
 - statement latencies in milliseconds:
         0.016  \set id random(1, 1000000)
      1144.441  select * from t_trgm where no_trgm ilike '%'||substring(md5(:id) from 3 for 6)||'%';
[tbase@VM_0_37_centos pgbench]$ cat select_t_trgm.sql
\set id random(1, 1000000)
select * from t_trgm where no_trgm ilike '%'||md5(:id)||'%';
[tbase@VM_0_37_centos pgbench]$ pgbench -h 127.0.0.1 -p 11008 -d postgres -U tbase -c 4 -j 1 -n -
M prepared -T 60 -r -f select_t_trgm.sql > select_t_trgm.txt 2>&1
[tbase@VM_0_37_centos pgbench]$ tail -20 select_t_trgm.txt
client 3 receiving
client 1 receiving
client 0 receiving
client 2 receiving
client 3 receiving
pghost: 127.0.0.1 pgport: 11008 nclients: 4 duration: 60 dbName: postgres
transaction type: select_t_trgm.sql
scaling factor: 1
query mode: prepared
```

```
number of clients: 4
number of threads: 1
duration: 60 s
number of transactions actually processed: 175
latency average = 1384.472 ms
tps = 2.889189 (including connections establishing)
tps = 2.889430 (excluding connections establishing)
script statistics:
 - statement latencies in milliseconds:
         0.015  \set id random(1, 1000000)
      1379.714  select * from t_trgm where no_trgm ilike '%'||md5(:id)||'%';
```

pgbench并发写入测试

```
[tbase@VM_0_37_centos pgbench]$  cat insert_t_trgm.sql
\set id random(1, 1000000)
insert into t_trgm values(:id,:id,:id) ;
[tbase@VM_0_37_centos pgbench]$ pgbench -h 127.0.0.1 -p 11008 -d postgres -U tbase -c 4 -j 1 -n -
M prepared -T 60 -r -f insert_t_trgm.sql > insert_t_trgm.txt 2>&1
[tbase@VM_0_37_centos pgbench]$ tail -20 insert_t_trgm.txt
client 0 receiving
client 2 receiving
client 3 receiving
client 0 receiving
client 1 receiving
pghost: 127.0.0.1 pgport: 11008 nclients: 4 duration: 60 dbName: postgres
transaction type: insert_t_trgm.sql
scaling factor: 1
query mode: prepared
number of clients: 4
number of threads: 1
duration: 60 s
number of transactions actually processed: 57719
latency average = 4.159 ms
tps = 961.874383 (including connections establishing)
tps = 961.961361 (excluding connections establishing)
script statistics:
 - statement latencies in milliseconds:
         0.011  \set id random(1, 1000000)
         4.145  insert into t_trgm values(:id,:id,:id) ;
```

数据对比总结及例外
数据对比

数据对比

项目	GIST	GIN	无索引
创建索引消耗时间	24.9s	16.6s	-
索引占用空间	178M	74M	-
数据导入时间测试	29s	36s	4.2s
极短字符串查询	1197.320 ms	986.631 ms	989.840 ms
中字符串查询	118.831 ms	4.300 ms	502.756 ms
pgbench长字符串查询，返回单一记录	235tps	222tps	2.8tps
pgbench短字符串查询，返回多条记录	39tps	954tps	3.4tps
pgbench并发写入测试	981tps	928tps	961tpc

一般情况下使用gin索引即可。

插件postgis使用

postgis添加方法

连接cn节点。

```
postgres=# set enable_sampling_analyze to off;
postgres=# create extension postgis;
postgres=# set enable_sampling_analyze to on;
```

关于存储过程
存储过程语法介绍
建立存储语法

语法概述

```
CREATE [ ORREPLACE ] PROCEDURE [ schema. ]procedure
  [ (argument [ { IN | OUT | IN OUT } ]
              [ NOCOPY ]
              datatype [ DEFAULT expr ]
    [, argument [ { IN | OUT | IN OUT } ]
              [ NOCOPY ]
              datatype [ DEFAULT expr ]
    ]...
  )
  ]
  [ invoker_rights_clause ]
  { IS | AS }
  { pl/sql_subprogram_body | call_spec } ;
```

示例

```
create orreplace procedure exe_imed_pro() as
begin
 perform dbms_output.put_line('tbase');
end;

callexe_imed_pro();
```

- IN：指定IN表示在调用过程时必须为参数提供一个值。
- OUT：指定OUT以指示该过程在执行后将该参数的值传递回其调用环境。
- IN OUT：指定IN OUT表示在调用过程时必须为参数提供一个值，并且该过程在执行后将值传递回其调用环境。
- 如果省略IN，OUT和IN OUT，则参数默认为IN。

示例：

```
create or replace procedure demo4(a_int in integer)
as
begin
  a_int:=1;
end;
```

/

[OR REPLACE] 更新存储介绍

带OR REPLACE的作用就存储过程存在时则替换的功能，建立存储时不带OR REPLACE关键字， 则遇到函数已经存系统则会报错，如下所示：

```
postgres=#  select prosrc from pg_proc where proname='proc_1';
            prosrc
-------------------------------
                          +
 begin                      +
    raise notice 'Hello TBase';+
 end;                       +

(1 row)
postgres=# CREATE OR REPLACE PROCEDURE proc_1() AS
$$
begin
raise notice 'Hello，TBase';
end;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# select prosrc from pg_proc where proname='proc_1';
            prosrc
--------------------------------
                           +
 begin                      +
    raise notice 'Hello，TBase';+
 end;                       +
 (1 row)
postgres=#
postgres=# call proc_1();
NOTICE:  Hello，TBase
CALL
postgres=#
```

另外也可以写成这样：

```
postgres=# CREATE OR REPLACE PROCEDURE proc_1() AS
begin
raise notice 'Hello，TBase';
end;
/
```

即不要$$开始和结束符号，最后使用/结束，这个语法更接近于oralce。

[模式名.]存储过程名介绍

建立存储过程，模式名可以指定，也可以不指定，不指存则存放在当前模式下，如上面例子就没有指定模式名，则就存放在当前模式下，如下所示：

```
postgres=# select * from pg_namespace;
     nspname       | nspowner |              nspacl
-------------------+----------+----------------------------------
 pg_toast          |       10 |
 pg_temp_1         |       10 |
 pg_toast_temp_1   |       10 |
 pg_catalog        |       10 | {postgres=UC/postgres,=U/postgres}
 public            |       10 | {postgres=UC/postgres,=UC/postgres}
 information_schema |       10 | {postgres=UC/postgres,=U/postgres}
(6 行记录)
postgres=# show search_path;
  search_path
----------------
 "$user",public
(1 行记录)
postgres=# select pg_namespace.nspname,pg_proc.prosrc from pg_proc,pg_namespace where
pg_proc.pronamespace=pg_namespace.oid and pg_proc.proname='proc_1';
 nspname |            prosrc
---------+-------------------------------
 public  |                              +
         | begin                        +
         |    raise notice 'Hello，TBase';+
         | end;                         +
         |
(1 row)
```

因为$user模式不存在，所以存在public模式下。

存储过程与函数不能同名

如创建一个与函数同名的存储过程会提示function xxx already exists with same argument types。

```
postgres=# CREATE OR REPLACE FUNCTION proc_1() RETURNS void AS
$$
begin
  raise notice 'Hello TBase';
end;
$$
LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=# CREATE  PROCEDURE proc_1() AS
$$
begin
  raise notice 'Hello TBase';
end;
$$
LANGUAGE PLPGSQL;
ERROR:  function "proc_1" already exists with same argument types
postgres=#
#如果你要替换，则提示
postgres=# CREATE OR REPLACE PROCEDURE proc_1() AS
$$
begin
  raise notice 'Hello TBase';
end;
$$
LANGUAGE PLPGSQL;
ERROR:  cannot change routine kind
DETAIL:  "proc_1" is a function.
postgres=#
```

删除存储过程

删除不带参数的存储过程

```
postgres=# CREATE OR REPLACE PROCEDURE proc_1() AS
$$
begin
  raise notice 'Hello TBase';
end;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# drop procedure proc_1 ( );
DROP PROCEDURE
postgres=#
```

删除带参数的存储过程

```
postgres=# CREATE OR REPLACE PROCEDURE proc_1(a_int int) AS
$$
begin
  raise notice '%',a_int;
end;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# drop procedure proc_1 ( a_int int);
DROP PROCEDURE
postgres=#
#也可以只指定参数的类型即可
postgres=# drop procedure proc_1 (int);
DROP PROCEDURE
postgres=#
```

存储过程修改名称

修改不带参数的存储过程名称

```
postgres=# CREATE OR REPLACE PROCEDURE proc_1() AS
$$
begin
  raise notice 'Hello TBase';
end;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# alter procedure proc_1() rename to proc_1_1;
ALTER PROCEDURE
postgres=#
```

修改带参数的存储过程名称

```
postgres=# CREATE OR REPLACE PROCEDURE proc_1(a_int int) AS
$$
begin
  raise notice '%',a_int;
end;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# alter procedure proc_1 (a_int int) rename to proc_1_1;
ALTER PROCEDURE
```

修改存储过程所属schema
修改不带参数的存储过程schema

```
postgres=# CREATE OR REPLACE PROCEDURE public.proc_1() AS
$$
begin
  raise notice 'Hello TBase';
end;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=#
postgres=# alter procedure public.proc_1() set schema myche;
ALTER PROCEDURE
postgres=#
```

修改带参数的存储过程schema

```
postgres=# CREATE OR REPLACE PROCEDURE public.proc_1(a_int int) AS
$$
begin
  raise notice '%',a_int;
end;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# alter procedure public.proc_1 (int) set schema myche;
ALTER PROCEDURE
postgres=#
```

修改存储过程所属用户

修改不带参数的存储过程所属用户

```
postgres=# CREATE OR REPLACE PROCEDURE public.proc_1() AS
$$
begin
  raise notice 'Hello TBase';
end;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# alter procedure proc_1() owner to tbase_01_admin;
ALTER PROCEDURE
postgres=#
```

修改带参数的存储过程所属用户

```
postgres=# CREATE OR REPLACE PROCEDURE proc_1(a_int int) AS
$$
begin
  raise notice '%',a_int;
end;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# alter procedure proc_1 (int) owner to tbase_01_admin;
ALTER PROCEDURE
postgres=#
```

存储过程执行
不带参数

```
postgres=# create or replace procedure p_no_para() as
$$
begin
    raise notice 'no_para procedure';
end;
$$
language plpgsql;
CREATE PROCEDURE
postgres=# call p_no_para();
NOTICE:  no_para procedure
CALL
postgres=#
```

带参数

```
postgres=# create or replace procedure p_para(a_int integer) as
$$
begin
    raise notice 'a_int = %',a_int;
end;
$$
language plpgsql;
CREATE PROCEDURE
postgres=# call p_para(1);
NOTICE:  a_int = 1
CALL
postgres=#
```

中途return返回

过程没有返回值。因此，过程的结束可以不用RETURN语句。 如果想用一个RETURN语句提前退出代码，只需写一个没有表达式的RETURN。

```
postgres=# CREATE OR REPLACE PROCEDURE p_return(a_return integer) AS
$$
BEGIN
    if a_return > 1 then
    raise notice '提前返回';
    return;
    end if;
    raise notice '结束返回';
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_return(2);
NOTICE:  提前返回
CALL
postgres=#
```

参数详细介绍
参数模式
IN模式

IN模式指的是执行函数时需要输入参数值，如下所示：

```
postgres=# CREATE OR REPLACE PROCEDURE p_in(IN a_xm text) AS
$$
BEGIN
    RAISE NOTICE 'a_xm=%',a_xm;
END;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# CALL p_in('Tbase');
NOTICE:  a_xm=Tbase
CALL
postgres=#

postgres=# CREATE OR REPLACE PROCEDURE p_in_default(a_xm text) AS
$$
BEGIN
    RAISE NOTICE 'a_xm=%',a_xm;
END;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# CALL p_in_default('Tbase');
NOTICE:  a_xm=Tbase
CALL
```

上面两种方式定义的参数效果是一样的。

INOUT模式

INOUT模式是指参数即传入，同时又指定了返回值的字段名和类型。

```
postgres=# CREATE OR REPLACE PROCEDURE p_inout(INOUT a_xm text) AS
$$
BEGIN
    RAISE NOTICE 'a_xm=%',a_xm;
END;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# CALL p_inout('Tbase');
NOTICE:  a_xm=Tbase
 a_xm
-------
 Tbase
(1 row)
```

参数引用
无命名参数

```
postgres=# CREATE OR REPLACE PROCEDURE p_unname(text) AS
$$
BEGIN
    raise notice '$1=%',$1;
END;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# call ps_unname('Tbase');
NOTICE:  $1=Tbase
CALL
postgres=#
```

给标识符指定别名

```
postgres=# CREATE OR REPLACE PROCEDURE p_specify_name(text) AS
$$
DECLARE
    a_xm ALIAS FOR $1; #a_xm是$1的别名
BEGIN
    raise notice '$1=%',a_xm;
END;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# CALL p_specify_name('Tbase');
NOTICE:  $1=Tbase
CALL
postgres=#
```

命名参数

```
postgres=# CREATE OR REPLACE PROCEDURE p_name(a_xm text) AS
$$
BEGIN
    raise notice '$1=%',a_xm;
END;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# call p_name('Tbase');
NOTICE:  $1=Tbase
CALL
postgres=#
```

参数数据类型

基本类型

```
postgres=# CREATE OR REPLACE PROCEDURE p_base_para (a_int integer,a_str text) AS
$$
BEGIN
   RAISE NOTICE 'a_int = % ; a_str = %',a_int,a_str;
END;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# CALL  p_base_para(1,'Tbase');
NOTICE:  a_int = 1 ; a_str = Tbase
CALL
postgres=#

postgres=# CREATE OR REPLACE PROCEDURE p_base_array (a_int integer[],a_str text[]) AS
$$
BEGIN
   RAISE NOTICE 'a_int = % ; a_str = %',a_int,a_str;
END;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# CALL p_base_array(ARRAY[1,2,3],ARRAY['TBase','pgxz']);
NOTICE:  a_int = {1,2,3} ; a_str = {TBase,pgxz}
CALL
postgres=#
```

复合类型

```
postgres=# CREATE TYPE public.t_per AS
(
    id integer,
    mc text
);
CREATE TYPE
postgres=# CREATE OR REPLACE PROCEDURE p_type (a_row public.t_per) AS
$$
BEGIN
    RAISE NOTICE 'id = % ; mc = %',a_row.id,a_row.mc;
END;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# CALL p_type(ROW(1,'TBase')::public.t_per);
NOTICE:  id = 1 ; mc = TBase
CALL
postgres=#
```

- 复合数组。

```
postgres=# CREATE OR REPLACE PROCEDURE p_type_array (a_rec public.t_per[]) AS
$$
BEGIN
    RAISE NOTICE 'a_rec = %',a_rec;
    RAISE NOTICE 'a_rec[1].id = %',a_rec[1].id;
END;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# CALL p_type_array (ARRAY[ROW(1,'TBase'),ROW(1,'pgxz')]::public.t_per[]);
NOTICE:  a_rec = {"(1,TBase)","(1,pgxz)"}
NOTICE:  a_rec[1].id = 1
CALL
postgres=#
```

行类型

```
postgres=# create table public.t(id int,mc text);
CREATE TABLE
postgres=#
postgres=# CREATE OR REPLACE PROCEDURE p_row (a_row public.t)  AS
$$
BEGIN
   RAISE NOTICE 'id = % ; mc = %',a_row.id,a_row.mc;
END;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# CALL p_row(ROW(1,'TBase'));
NOTICE:  id = 1 ; mc = TBase
CALL
postgres=#
```

- 行数组。

```
postgres=# CREATE OR REPLACE PROCEDURE p_row_array (a_rec public.t[]) AS
$$
BEGIN
   RAISE NOTICE 'a_rec = %',a_rec;
   RAISE NOTICE 'a_rec[1].id = %',a_rec[1].id;
END;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# CALL p_row_array(array[row(1,'TBase'),row(1,'pgxz')]::public.t[]);
NOTICE:  a_rec = {"(1,TBase)","(1,pgxz)"}
NOTICE:  a_rec[1].id = 1
CALL
postgres=#
```

游标类型

```
postgres=# CREATE OR REPLACE PROCEDURE p_refcursor (a_ref refcursor) AS
$$
DECLARE
   v_rec record;
BEGIN
   OPEN a_ref FOR SELECT * FROM t LIMIT 1;
   FETCH a_ref INTO v_rec;
   RAISE NOTICE 'v_rec = % ',v_rec;
END;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# CALL p_refcursor('a');
NOTICE:  v_rec = (1,Tbase)
CALL
postgres=#
```

多态类型

```
postgres=# CREATE OR REPLACE PROCEDURE f_any(a_arg anyelement) AS
$$
BEGIN
   RAISE NOTICE '%',a_arg;
END;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# CALL f_any(1);
NOTICE:  1
CALL
postgres=# CALL f_any('Tbase'::varchar);
NOTICE:  Tbase
CALL
postgres=#
postgres=# SELECT f_any('TBase'::TEXT);
NOTICE:  TBase
 f_any
-------
(1 行记录)
postgres=# CALL f_any(ROW(1,'TBase')::public.t);
NOTICE:  (1,TBase)
CALL
postgres=#
postgres=# CALL f_any(ARRAY[1,2]::INTEGER[]);
NOTICE:  {1,2}
CALL
postgres=#
postgres=# CALL  f_any(ARRAY[[1,2],[3,4],[5,6]]::INTEGER[][][]);
NOTICE:  {{1,2},{3,4},{5,6}}
CALL
postgres=#
#注意多态类型参数调用时最好直接声明参数类型，否则有可能出错
postgres=# CREATE OR REPLACE PROCEDURE f_any(a_arg anyarray) AS
$$
BEGIN
   RAISE NOTICE '%',a_arg;
END;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# call f_any(ARRAY['TBase','pgxz']::TEXT[]);
ERROR:  procedure f_any(text[]) is not unique
```

```
LINE 1: call f_any(ARRAY['TBase','pgxz']::TEXT[]);
             ^
HINT:  Could not choose a best candidate procedure. You might need to add explicit type casts.
postgres=#
```

> 注意：
>
> Anyelement参数如果写成数组，其意义就跟anyarray参数一致，所以 f_any(a_arg anyelement)与f_any(a_arg anyarray)在调用f_any(ARRAY[1,2])时就会出现函数不是唯一化的错误(ERROR: function f_any(...) is not unique)提示。

参数默认值

```
postgres=# CREATE OR REPLACE PROCEDURE p_default_value (a_int INTEGER DEFAULT 1) AS
$$
BEGIN
   RAISE NOTICE 'a_int = %',a_int;
END;
$$
LANGUAGE PLPGSQL;
CREATE PROCEDURE
postgres=# CALL p_default_value(2);
NOTICE:  a_int = 2
CALL
postgres=# CALL p_default_value();
NOTICE:  a_int = 1
CALL
postgres=#
#备注：如果原来存在一个p_default_value ()这样的存储过程，则上面的执行就会出错，因为系统无法清楚到你到底要执行那个函数，如下所示
postgres=# CREATE OR REPLACE PROCEDURE p_default_value() AS
$$
BEGIN
   RAISE NOTICE '无参数';
END;
$$
LANGUAGE plpgsql ;
CREATE PROCEDURE
postgres=# CALL p_default_value();
ERROR:  procedure p_default_value() is not unique
LINE 1: CALL p_default_value();
             ^
HINT:  Could not choose a best candidate procedure. You might need to add explicit type casts.
postgres=#
```

出错提示，p_default_value ()存储过程不是唯一的，这是使用上一个需要特别注意的地方。

变量使用
变量使用介绍
%TYPE

语法：

```
{ collection_name
| cursor_variable_name
| object name
| record_name
| record_name . field_name
| db_table_name . column_name
| variable_name
}
%TYPE
```

示例：

```
create table t4 (f1 integer,f2 varchar2(10));
insert into t4 values(1,'tbase');
commit;
DECLARE
v_t4_f2 t4.f2%type;
begin
 select f2 into v_t4_f2 from t4 where f1=1;
 perform dbms_output.put_line('v_t4_f2='||v_t4_f2);
end;
/
```

%ROWTYPE

示例：%rowtype当作参数和返回值

```
-- 9 rowtype as parameter
set enable_oracle_compatible to on;

drop table if exists t_rowtype_20221111_1;
create table t_rowtype_20221111_1(a int, b int);
insert into t_rowtype_20221111_1 values(generate_series(1, 10), generate_series(1, 10));

create table t_rowtype_20221111_2(a int);
insert into t_rowtype_20221111_2 values(generate_series(1, 10));

create table t_rowtype_20221111_3(a text, b text);
insert into t_rowtype_20221111_3 values(generate_series(1, 10), generate_series(1, 10));

drop table if exists t_rowtype_20221111_4;
drop view if exists tv_rowtype_20221111_1;
drop table if exists t_rowtype_20221111_5;
create table t_rowtype_20221111_4(a int, b varchar2(50), c number);
create or replace view tv_rowtype_20221111_1 as select a, b from t_rowtype_20221111_4;
insert into t_rowtype_20221111_4 values(generate_series(1, 10), chr(generate_series(1,10) + 96), genera
te_series(1, 10));
create table t_rowtype_20221111_5(key1 int, des1 varchar2(50));

drop function if exists f2f_rowtype_20221111_1;
create or replace function f2f_rowtype_20221111_1(e int, b t_rowtype_20221111_4%rowtype) return int
is
res t_rowtype_20221111_4%rowtype;
begin
res := b;
select * into res from t_rowtype_20221111_4 where t_rowtype_20221111_4.a = e;
insert into t_rowtype_20221111_5 values(res.a, res.b);
return b.b;
end;
/

drop function if exists p2p_rowtype_20221111_1;
create or replace procedure p2p_rowtype_20221111_1(e int, b t_rowtype_20221111_4%rowtype) is
res t_rowtype_20221111_4%rowtype;
begin
res := b;
select * into res from t_rowtype_20221111_4 where t_rowtype_20221111_4.a = e;
insert into t_rowtype_20221111_5 values(res.a, res.b);
```

```
end;
/

drop function if exists p4p;
create or replace procedure p4p(e1 int, e2 int, f out t_rowtype_20221111_4%rowtype, g IN OUT t_rowty
pe_20221111_4%rowtype) is
res t_rowtype_20221111_4%rowtype;
begin
res := g;
insert into t_rowtype_20221111_5 values(res.a, 'inout :' || res.b);
select * into res from t_rowtype_20221111_4 where t_rowtype_20221111_4.a = e1;
f := res;
select * into res from t_rowtype_20221111_4 where t_rowtype_20221111_4.a = e2;
g := res;
end;
/

create or replace package pkg_rowtype_20221111_1 is
dft_rt t_rowtype_20221111_4%rowtype;
end;
/

create or replace package pkg_rowtype_20221111_2 is
dft_rt t_rowtype_20221111_4%rowtype;
CURSOR c1 is select * from t_rowtype_20221111_4;
dft_rt1 c1%rowtype;
function pkg_f_rowtype_20221111_3(a int, b c1%rowtype) return c1%rowtype;
end;
/

create or replace package body pkg_rowtype_20221111_2 is
function pkg_f_rowtype_20221111_3(a int, b c1%rowtype) return c1%rowtype is
val c1%rowtype;
begin
val := b;
select * into val from t_rowtype_20221111_4;
return val;
end;
end;
/

begin
select * into pkg_rowtype_20221111_2.dft_rt1 from t_rowtype_20221111_4;
end;
/

insert into t_rowtype_20221111_4 values(1, 'pkg_f', 2);
```

```
declare
v1 pkg_rowtype_20221111_2.c1%rowtype;
v2 pkg_rowtype_20221111_2.c1%rowtype;
a int := 3;
begin
v1 := pkg_rowtype_20221111_2.pkg_f_rowtype_20221111_3(a, v2);
end;
/

drop function if exists p3p_rowtype_20221111_2;
-- pl/sql not support using package var as default parameter
-- create or replace procedure p3p_rowtype_20221111_2(e int, a t_rowtype_20221111_4%rowtype defa
ult pkg_rowtype_20221111_1.dft_rt, b t_rowtype_20221111_4%rowtype default null) is
create or replace procedure p3p_rowtype_20221111_2(e int, b t_rowtype_20221111_4%rowtype default
null) is
buf t_rowtype_20221111_4%rowtype;
begin
if b is null then
select * into buf from t_rowtype_20221111_4 where t_rowtype_20221111_4.a = e;
insert into t_rowtype_20221111_5 values(buf.a, 'null test, defualt val:' || buf.b);
else
insert into t_rowtype_20221111_5 values(-1, 'not support null test of record type');
end if;
end;
/

declare
p1 t_rowtype_20221111_4%rowtype;
p2 t_rowtype_20221111_4%rowtype;
res int;
begin
res := f2f_rowtype_20221111_1(5, p1);
call p2p_rowtype_20221111_1(3, p1);
p2.a := 12;
p2.b := 'z';
p2.c := 12;
call p4p(6,7,p1,p2);
insert into t_rowtype_20221111_5 values(p1.a, 'OUT: ' || p1.b);
insert into t_rowtype_20221111_5 values(p2.a, 'OUT: ' || p2.b);
pkg_rowtype_20221111_1.dft_rt.a := 13;
pkg_rowtype_20221111_1.dft_rt.b := 'x';
pkg_rowtype_20221111_1.dft_rt.c := 13;
call p3p_rowtype_20221111_2(8);
end;
/
select * from t_rowtype_20221111_5 order by key1, des1;
```

```
-- 10 rowtype as return
drop function if exists f3f_rowtype_20221111_2;
create or replace function f3f_rowtype_20221111_2(e int, b int, c int) return t_rowtype_20221111_4%ro
wtype is
res t_rowtype_20221111_4%rowtype;
begin
select * into res from t_rowtype_20221111_4 where t_rowtype_20221111_4.a = e;
return res;
end;
/
declare
p1 t_rowtype_20221111_4%rowtype;
begin
p1 := f3f_rowtype_20221111_2(6, 2, 2);
insert into t_rowtype_20221111_5 values(p1.a, p1.b);
end;

/
select * from t_rowtype_20221111_5 order by key1, des1;
```

> 注意：
>
> 暂不支持游标使用%rowtype。

记录类型

语法

TYPE rec_type IS RECORD (fields)

示例：

```
CREATE SCHEMA IF NOT EXISTS test_oracle;
set search_path to test_oracle;
create table tbl_city(id int, population int,  nation varchar(32), city varchar(32), gdp int, others text);

insert into tbl_city(id, population,  nation, city, gdp) values(1,500, 'China','Guangzhou', 23000);
insert into tbl_city(id, population,  nation, city, gdp) values(2,1500, 'China', 'Shanghai', 29000);
insert into tbl_city(id, population,  nation, city, gdp) values(3,1500, 'China', 'Beijing', 25000);
insert into tbl_city(id, population,  nation, city, gdp) values(4,1000, 'China', 'Shenzhen', 24000);
insert into tbl_city(id, population,  nation, city, gdp) values(5,1000,'USA','New York', 35000);
insert into tbl_city(id, population,  nation, city, gdp) values(6,500, 'USA', 'Bostom', 15000);
insert into tbl_city(id, population,  nation, city, gdp) values(7,500, 'Japan','Tokyo', 40000);
insert into tbl_city(id, population,  nation, city, gdp) values(8,800, 'China', 'Hongkong', 23500);
insert into tbl_city(id, population,  nation, city, gdp) values(9,800, 'China', 'Hangzhou', 15500);
insert into tbl_city(id, population,  nation, city, gdp) values(10,100, 'USA', 'Los Angele', 15500);

set search_path to test_oracle;
DROP FUNCTION catcity;
CREATE FUNCTION catcity(nation_name text) RETURNS text AS $$
DECLARE
    TYPE mycitytype IS RECORD(city text);
    trow tbl_city%ROWTYPE;
    rlt text;
    lt mycitytype;
BEGIN
    rlt = ':' || nation_name;
    FOR trow IN SELECT * FROM tbl_city WHERE nation = nation_name
    LOOP

        lt.city = trow.city;
        rlt = lt.city ||'+' || rlt;
    END LOOP;
    return rlt;
END;
$$ LANGUAGE plpgsql;

select catcity('China');
```

变量使用示例

变量声明语法

name [CONSTANT] type [COLLATE collation_name] [NOT NULL] [{ DEFAULT | := | = }expression];
如果给定DEFAULT子句，它会指定进入该块时分配给该变量的初始值。如果没有给出DEFAULT子句， 则该变量被初始化为SQL空值。CONSTANT选项阻止该变量在初始化之后被赋值， 这样它的值在块的持续期内保持不变。
COLLATE 选项指定用于该变量的一个排序规则（见 第41.3.6 节）。如果指 定了NOTNULL，对该变量赋值为空值会导致一个 运行时错误。所有被声明为NOT NULL的变量必须 被指定一个非空默认值。 等号（ = ）可以被用来代替 PL/SQL-兼容的 :=。

定义一个普通变量

```
postgres=# CREATE OR REPLACE PROCEDURE ordinary_var() AS
$$
DECLARE
    #所有变量的声明都要放在这里,建议变量以v_开头,参数以a_开头
    v_int integer := 1;
    v_text text;
BEGIN
    v_text = 'TBase';
    RAISE NOTICE 'v_int = %',v_int;
    RAISE NOTICE 'v_text = %',v_text;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL ordinary_var();
NOTICE:  v_int = 1
NOTICE:  v_text = TBase
CALL
postgres=#
```

定义CONSTANT变量

```
postgres=# CREATE OR REPLACE PROCEDURE p_constant()  AS
$$
DECLARE
   v_int CONSTANT integer := 1;
BEGIN
   RAISE NOTICE 'v_int = %',v_int;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_constant();
NOTICE:  v_int = 1
CALL
#CONSTANT 不能再次赋值
postgres=# CREATE OR REPLACE PROCEDURE p_constant()  AS
$$
DECLARE
   v_int CONSTANT integer := 1;
BEGIN
   RAISE NOTICE 'v_int = %',v_int;
v_int = 10;
END;
$$
LANGUAGE plpgsql;
ERROR:  "v_int" is declared CONSTANT
LINE 7: v_int = 10;
        ^
postgres=#
```

定义NOT NULL变量

```
postgres=# CREATE OR REPLACE PROCEDURE p_not_null_var() AS
$$
DECLARE
   v_int integer NOT NULL := 1;
BEGIN
   RAISE NOTICE 'v_int = %',v_int;
   SELECT NULL INTO v_int;
   RAISE NOTICE 'v_int = %',v_int;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_not_null_var();
NOTICE:  v_int = 1
ERROR:  null value cannot be assigned to variable "v_int" declared NOT NULL
CONTEXT:  PL/pgSQL function p_not_null_var() line 6 at SQL statement
postgres=#
```

定义为NOT NULL变量，则该变量受NOT NULL约束。

定义COLLATE变量

- 按unicode值对比大小。

```
postgres=# CREATE OR REPLACE PROCEDURE p_collate_unicode() AS
$$
DECLARE
   v_txt1 TEXT COLLATE "C" := '严';
   v_txt2 TEXT COLLATE "C" := '丰';
BEGIN
IF v_txt1 > v_txt2 THEN
   RAISE NOTICE ' % > % ',v_txt1,v_txt2;
ELSE
   RAISE NOTICE ' % > % ',v_txt2,v_txt1;
END IF;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_collate_unicode();
NOTICE:  丰 > 严
CALL
postgres=#

postgres=# select '严'::bytea;
  bytea
----------
 \xe4b8a5
(1 row)

postgres=# select '丰'::bytea;
  bytea
----------
 \xe4b8b0
(1 row)
```

- 按汉字的拼音对比大小。

```
postgres=# CREATE OR REPLACE PROCEDURE p_collate_pinyin() AS
$$
DECLARE
   v_txt1 TEXT COLLATE "zh_CN.utf8" := '严';
   v_txt2 TEXT COLLATE "zh_CN.utf8" := '丰';
BEGIN
```

```
    IF v_txt1 > v_txt2 THEN
    RAISE NOTICE ' % -> % ',v_txt1,v_txt2;
    ELSE
    RAISE NOTICE ' % -> % ',v_txt2,v_txt1;
    END IF;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_collate_pinyin();
NOTICE:   严 -> 丰
CALL
postgres=#
```

变量赋值

```
postgres=# CREATE OR REPLACE PROCEDURE p_setval()  AS
$$
DECLARE
    #定义时赋值
    v_int1 integer = 1;
    --使用 :=兼容于plsql
    v_int2 integer := 1;
    v_txt1 text;
    v_float float8;
    --使用查询赋值
    v_relname text = (select relname FROM pg_class LIMIT 1);
    v_relpages integer;
    v_rec RECORD;
BEGIN
    #在函数体中赋值
    v_txt1 = 'TBase';
    v_float = random();
    #使用查询赋值的另一种方式
    SELECT relname,relpages INTO v_relname,v_relpages FROM pg_class ORDER BY random() LIMIT 1;
    RAISE NOTICE 'v_relname = % , relpages = %',v_relname,v_relpages;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_setval();
NOTICE:  v_relname = pg_toast_17220_index , relpages = 1
CALL
postgres=#
```

BULK COLLECT

BULK COLLECT将一个查询结果集保存起来。

- 用例1。

```
postgres=# create table t5(f1 integer,f2 varchar(10));
CREATE TABLE
postgres=# insert into t5 values(1,'tbase1');
INSERT 0 1
postgres=# insert into t5 values(2,'tbase2');
INSERT 0 1

postgres=# create or replace procedure p_bulk_collect()
AS
$$
declare
   TYPE t5list IS TABLE OF t5.f2%TYPE;
   t5s t5list;
BEGIN
   SELECT f2 BULK COLLECT INTO t5s FROM t5;
   FOR i IN t5s.FIRST .. t5s.LAST
   LOOP
      raise notice '%',t5s[i];
   END LOOP;
END
$$language plpgsql;
NOTICE:  type reference t5.f2%TYPE converted to character varying
CREATE PROCEDURE
postgres=# CALL p_bulk_collect();
NOTICE:  tbase1
NOTICE:  tbase2
CALL
```

- 用例2。

```
postgres=# create table tbl_person(id integer, name text, tdd int);
CREATE TABLE
postgres=# insert into tbl_person values(1,'tbase',1);
insert into tbl_person values(2,'pgxz',1);
INSERT 0 1
postgres=# insert into tbl_person values(2,'pgxz',1);
INSERT 0 1
postgres=# insert into tbl_person values(3,'pg',2);
```

```
INSERT 0 1

postgres=# create or replace procedure p_bulkcollect_select_into(a_tdd integer) AS
$$
declare
  type TAPersonlist is table of tbl_person%rowtype;
  vpa TAPersonlist;
  rp tbl_person%rowtype;
begin
  select * bulk collect into vpa from tbl_person where tdd = a_tdd;
  raise notice 'count=%', vpa.count;
  for i in vpa.first..vpa.last loop
     rp = vpa[i];
     raise notice 'loop=%', i;
     raise notice 'vname=%', rp.name;
  end loop;
  raise notice 'vpa.count=%',vpa.count;
end;
$$language plpgsql;
CREATE PROCEDURE
postgres=# CALL p_bulkcollect_select_into(1);
NOTICE:  count=2
NOTICE:  loop=1
NOTICE:  vname=tbase
NOTICE:  loop=2
NOTICE:  vname=pgxz
NOTICE:  vpa.count=2
CALL
```

控制结构
判断语句
IF...THEN...END IF

```
postgres=# CREATE OR REPLACE PROCEDURE p_if() AS
$$
BEGIN
   IF random()>0.5 THEN
      RAISE NOTICE '随机数大于0.5';
   END IF;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_if();
NOTICE:  随机数大于0.5
CALL
postgres=#
```

IF...THEN...ELSE...END IF

```
postgres=# CREATE OR REPLACE PROCEDURE p_if_else() AS
$$
BEGIN
   IF random()>0.99 THEN
      RAISE NOTICE '随机数大于0.99';
   ELSE
      RAISE NOTICE '随机数小于或等于0.99';
   END IF;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_if_else();
NOTICE:  随机数小于或等于0.99
CALL
postgres=#
```

IF...THEN...ELSIF...THEN...ELSE...END IF

```
postgres=# CREATE OR REPLACE PROCEDURE p_if_elsif()  AS
$$
DECLARE
   v_float8 float8 := random();
BEGIN
   IF v_float8>0.99 THEN
      RAISE NOTICE '随机数大于0.99';
   ELSIF v_float8>0.5 THEN
      RAISE NOTICE '随机数大于0.50';
   ELSIF v_float8>0.25 THEN
      RAISE NOTICE '随机数大于0.25';
   ELSE
      RAISE NOTICE '随机数小于或等于0.25';
   END IF;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_if_elsif();
NOTICE:  随机数大于0.50
CALL
```

CASE语句

```
postgres=# CREATE OR REPLACE PROCEDURE p_case()  AS
$$
DECLARE
   v_float8 float8 := random();
BEGIN
   CASE
   WHEN v_float8>0.99 THEN
      RAISE NOTICE '随机数大于0.99';
   WHEN v_float8>0.5 THEN
      RAISE NOTICE '随机数大于0.50';
   WHEN v_float8>0.25 THEN
      RAISE NOTICE '随机数大于0.25';
   ELSE
      RAISE NOTICE '随机数小于或等于0.25';
   END CASE;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_case();
NOTICE:  随机数小于或等于0.25
CALL
postgres=#
```

循环语句

LOOP循环

```
postgres=# CREATE OR REPLACE PROCEDURE p_loop()  AS
$$
DECLARE
    v_id INTEGER := 1;
BEGIN
   LOOP
      RAISE NOTICE '%',v_id;
      EXIT WHEN random()>0.8;
      v_id := v_id + 1;
   END LOOP ;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_loop();
NOTICE:  1
NOTICE:  2
NOTICE:  3
CALL
postgres=#
```

WHILE循环

```
postgres=# CREATE OR REPLACE PROCEDURE p_while()  AS
$$
DECLARE
   v_id INTEGER := 1;
   v_random float8 ;
BEGIN
   LOOP
      RAISE NOTICE '%',v_id;
      v_id := v_id + 1;
      v_random := random();
      IF v_random > 0.8 THEN
         RETURN;
      END IF;
   END LOOP ;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_while();
NOTICE:  1
CALL
```

FOR循环

```
postgres=# CREATE OR REPLACE PROCEDURE p_for()  AS
$$
BEGIN
   FOR i IN 1..3 LOOP
      RAISE NOTICE 'i = %',i;
   END LOOP;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_for();
NOTICE:  i = 1
NOTICE:  i = 2
NOTICE:  i = 3
CALL
postgres=# CREATE OR REPLACE PROCEDURE p_for_reverse()  AS
$$
BEGIN
   FOR i IN REVERSE 3..1 LOOP
      RAISE NOTICE 'i = %',i;
   END LOOP;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_for_reverse();
NOTICE:  i = 3
NOTICE:  i = 2
NOTICE:  i = 1
CALL
使用REVERSE递减
postgres=# CREATE OR REPLACE PROCEDURE p_for_by()  AS
$$
BEGIN
   FOR i IN 1..8 BY 2 LOOP
      RAISE NOTICE 'i = %',i;
   END LOOP;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_for_by();
NOTICE:  i = 1
```

```
NOTICE:  i = 3
NOTICE:  i = 5
NOTICE:  i = 7
CALL
postgres=#
#使用BY设置步长
```

FOR循环查询结果

```
postgres=# CREATE OR REPLACE PROCEDURE p_for_record()  AS
$$
DECLARE
   v_rec RECORD;
BEGIN
   FOR v_rec IN SELECT relname,relkind FROM pg_class limit 2 LOOP
      RAISE NOTICE '%',v_rec;
   END LOOP;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_for_record();
NOTICE:  (pg_stat_statements,v)
NOTICE:  (pg_proc,v)
CALL
postgres=#
```

FOREACH循环一个数组

```
postgres=# CREATE OR REPLACE PROCEDURE p_foreach()  AS
$$
DECLARE
   v_random_arr float8[]:=ARRAY[random(),random()];
   v_random float8;
BEGIN
   FOREACH v_random IN ARRAY v_random_arr LOOP
      RAISE NOTICE '%',v_random ;
   END LOOP;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_foreach();
NOTICE:  0.744417542591691
NOTICE:  0.804096563253552
CALL
postgres=#
postgres=# CREATE OR REPLACE PROCEDURE p_foreach_slice()  AS
$$
DECLARE
   v_random_arr float8[][]:=ARRAY[ARRAY[random(),random()],ARRAY[random(),random()]];
   v_random float8;
BEGIN
   FOREACH v_random SLICE 0 IN ARRAY v_random_arr LOOP
      RAISE NOTICE '%',v_random ;
   END LOOP;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_foreach_slice();
NOTICE:  0.0220407997258008
NOTICE:  0.898449067492038
NOTICE:  0.190678883343935
NOTICE:  0.103653562255204
CALL
postgres=#
#循环会通过计算expression得到的数组的个体元素进行迭代
postgres=# CREATE OR REPLACE PROCEDURE p_foreach_slice_1()  AS
$$
DECLARE
   v_random_arr float8[][]:=ARRAY[ARRAY[random(),random()],ARRAY[random(),random()]];
```

```
    v_random float8[];
BEGIN
    FOREACH v_random SLICE 1 IN ARRAY v_random_arr LOOP
        RAISE NOTICE '%',v_random ;
    END LOOP;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_foreach_slice_1();
NOTICE:  {0.248282201588154,0.757913041394204}
NOTICE:  {0.0194511725567281,0.43799454299733}
CALL
#通过一个正SLICE值，FOREACH通过数组的切片而不是单一元素迭代
```

其它控制语句
动态执行

```
postgres=# CREATE OR REPLACE PROCEDURE p_execute() AS
$$
DECLARE
   v_sql TEXT;
   v_relname TEXT;
BEGIN
   v_sql := 'SELECT relname FROM pg_class limit 1';
   EXECUTE v_sql INTO v_relname;
   RAISE NOTICE 'relname = %',v_relname;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_execute();
NOTICE:  relname = pg_stat_statements
CALL
postgres=#
#也可以使用immediate
postgres=# CREATE OR REPLACE PROCEDURE p_execute() AS
$$
DECLARE
   v_sql TEXT;
   v_relname TEXT;
BEGIN
   v_sql := 'SELECT relname FROM pg_class limit 1';
   EXECUTE immediate v_sql INTO v_relname;
   RAISE NOTICE 'relname = %',v_relname;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_execute();
NOTICE:  relname = s1
CALL
postgres=#
#动态执行就是拼sql语句,然后使用EXECUTE命令执行
```

执行一个没有结果的命令

```
postgres=# CREATE OR REPLACE PROCEDURE p_perform()  AS
$$
BEGIN
    perform md5(random()::text);
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# call p_perform();
CALL
postgres=#
```

获取执行结果

```
postgres=# CREATE OR REPLACE PROCEDURE p_found()  AS
$$
DECLARE
   v_relname TEXT;
BEGIN
   SELECT relname INTO v_relname FROM pg_class limit 1;
   IF FOUND THEN
      RAISE NOTICE '查询到记录，值为%',v_relname;
   ELSE
      RAISE NOTICE '查不到记录' ;
   END IF;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_found();
NOTICE:  查询到记录，值为pg_stat_statements
CALL
```

获取影响行数

```
postgres=# CREATE OR REPLACE PROCEDURE p_row_count()  AS
$$
DECLARE
   v_row_count BIGINT;
BEGIN
   delete from t1;
   GET DIAGNOSTICS v_row_count = ROW_COUNT;
   RAISE NOTICE '查询到的记录数为 % ',v_row_count;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# call p_row_count();
NOTICE:  查询到的记录数为 3
CALL
postgres=#
```

GOTO

```
postgres=# create or replace procedure p_goto(v_maxnum integer) as
$$
declare
   maxnum integer;
begin
   maxnum := v_maxnum;
for i in 1..maxnum loop
   if i=3 then
      goto label;
   end if;
   raise notice 'i=%',i;
end loop;
<<label>>
   raise notice 'goto end';
end;
$$
language plpgsql;
CREATE PROCEDURE
postgres=# call p_goto(5);
NOTICE:  i=1
NOTICE:  i=2
NOTICE:  goto end
CALL
postgres=#
```

go用于跳转到某个标签下。

俘获错误
错误俘获处理

```
postgres=# CREATE OR REPLACE PROCEDURE p_exception(a_id integer,a_nc text) AS
$$
BEGIN
   INSERT INTO t_exception VALUES(a_id,a_nc);
   RETURN ;
   EXCEPTION WHEN OTHERS THEN
   RAISE NOTICE '执行出错';
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=#
postgres=# CALL p_exception(1,'Tbase');
CALL
postgres=# CALL p_exception(1,'Tbase');
NOTICE:  执行出错
CALL
```

获取错误相关信息

```
postgres=# CREATE OR REPLACE PROCEDURE p_exception_error(a_id integer,a_nc text) AS
$$
DECLARE
   v_sqlstate text;
   v_context text;
   v_message_text text;
BEGIN
   INSERT INTO t_exception VALUES(a_id,a_nc);
   RETURN ;
   EXCEPTION WHEN OTHERS THEN
   GET STACKED DIAGNOSTICS v_sqlstate = RETURNED_SQLSTATE,
                           v_message_text = MESSAGE_TEXT,
                           v_context = PG_EXCEPTION_CONTEXT;
   RAISE NOTICE '错误代码 : %',v_sqlstate;
   RAISE NOTICE '出错信息 : %',v_message_text;
   RAISE NOTICE '发生异常语句 : %',v_context;
   raise notice '错误代码 : % \n出错信息 : % 发生异常语句 : %',v_sqlstate ,v_message_text,v_context;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_exception_error(2,'Tbase');
CALL
postgres=# CALL p_exception_error(2,'Tbase');
NOTICE:  错误代码 : 23505
NOTICE:  出错信息 : node:dn001, backend_pid:16204, nodename:dn001,backend_pid:16204,message:du
plicate key value violates unique constraint "t_exception_id_uidx"
NOTICE:  发生异常语句 : SQL statement "INSERT INTO t_exception VALUES(a_id,a_nc)"
PL/pgSQL function p_exception_error(integer,text) line 7 at SQL statement
NOTICE:  错误代码 : 23505 \n出错信息 : node:dn001, backend_pid:16204, nodename:dn001,backend_pid
:16204,message:duplicate key value violates unique constraint "t_exception_id_uidx" 发生异常语句 : SQL
statement "INSERT INTO t_exception VALUES(a_id,a_nc)"
PL/pgSQL function p_exception_error(integer,text) line 7 at SQL statement
CALL
postgres=#
```

自治事务
在存储过程中commit和rollback

```
postgres=# create table t_autonomous_transaction( f1 int,f2 int);
CREATE TABLE
postgres=# create or replace procedure t_autonomous_transaction_isnert() as
$$
begin
    insert into t_autonomous_transaction values(1,1);
insert into t_autonomous_transaction values(2,2);
commit;
insert into t_autonomous_transaction values(3,3);
rollback;
end;
$$
language plpgsql;
CREATE PROCEDURE
postgres=# call t_autonomous_transaction_isnert();
CALL
postgres=# select * from t_autonomous_transaction ;
 f1 | f2
----+----
  1 |  1
  2 |  2
(2 rows)
#在同一个存储过程中，记录1，2 commit，记录3 rollback
#另外，如果存储过程中使用了自治事务，则这个存储过程就不能放在一个事务中调用，如下
postgres=# begin;
BEGIN
postgres=# call t_autonomous_transaction_isnert();
ERROR:  invalid transaction termination
CONTEXT:  PL/pgSQL function t_autonomous_transaction_isnert() line 5 at COMMIT
postgres=# rollback;
#用例，每插入10条记录提交一次
postgres=# create or replace procedure t_autonomous_transaction_isnert() as
$$
begin
    for i in 1 .. 100 loop
       insert into t_autonomous_transaction values(i,i);
   if mod(i,10)=0 then
       commit;
   end if;
   insert into t_autonomous_transaction values(i,i);
end loop;
```

```
end;
$$
language plpgsql;
```

游标使用限制

```
postgres=# create table t_ref(f1 int,f2 int);
CREATE TABLE
postgres=# insert into t_ref values(1,1),(2,2),(3,3);
COPY 3
postgres=# CREATE OR REPLACE procedure p_refcursor() AS
$$
DECLARE
    v_ref refcursor;
v_rec record;
BEGIN
    OPEN v_ref FOR SELECT * FROM t_ref;
    fetch next from v_ref into v_rec ;
commit;
fetch next from v_ref into v_rec ;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# call p_refcursor();
ERROR:  cursor "<unnamed portal 5>" does not exist
CONTEXT:  PL/pgSQL function p_refcursor() line 9 at FETCH
```

在存储过程中如果使用了游标，则游标在commit后不可使用。

自治事务与exception的限制

```
postgres=# CREATE OR REPLACE PROCEDURE p_exception_error(a_id integer,a_nc text) AS
$$
DECLARE
   v_sqlstate text;
   v_context text;
   v_message_text text;
BEGIN
   INSERT INTO t_exception VALUES(a_id,a_nc);
COMMIT;
   EXCEPTION WHEN OTHERS THEN
   GET STACKED DIAGNOSTICS v_sqlstate = RETURNED_SQLSTATE,
                           v_message_text = MESSAGE_TEXT,
                           v_context = PG_EXCEPTION_CONTEXT;
   RAISE NOTICE '错误代码 : %',v_sqlstate;
   RAISE NOTICE '出错信息 : %',v_message_text;
   RAISE NOTICE '发生异常语句 : %',v_context;
   raise notice '错误代码 : % \n出错信息 : % 发生异常语句 : %',v_sqlstate ,v_message_text,v_context;
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# call p_exception_error(1,'tbase');
NOTICE:  错误代码 : 2D000
NOTICE:  出错信息 : cannot commit while a subtransaction is active
NOTICE:  发生异常语句 : PL/pgSQL function p_exception_error(integer,text) line 8 at COMMIT
NOTICE:  错误代码 : 2D000 \n出错信息 : cannot commit while a subtransaction is active 发生异常语句 : P
L/pgSQL function p_exception_error(integer,text) line 8 at COMMIT
CALL
postgres=#
```

如果存储过程中使用了exception，就不能再使用自治事务，在5.06版本中兼容这个问题。

消息及异常处理
RAISE NOTICE

```
postgres=# CREATE OR REPLACE PROCEDURE p_raise() AS
$$
DECLARE
   v_int INTEGER := 1;
BEGIN
   RAISE NOTICE 'v_int = %, 随机数 = %',v_int,random();
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_raise();
NOTICE:  v_int = 1, 随机数 = 0.668172102887183
CALL
postgres=#
```

使用raise notice 向终端输出一个消息,也有可能写到日志中(需要调整日志的保存级别)。

RAISE EXCEPTION

```
postgres=# CREATE OR REPLACE PROCEDURE p_exception() AS
$$
DECLARE
   v_int INTEGER := 1;
BEGIN
   RAISE EXCEPTION '程序EXCEPTION ';
   --下面的语句不会再执行
   RAISE NOTICE 'v_int = %, 随机数 = %',v_int,random();
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_exception();
ERROR:  程序EXCEPTION
CONTEXT:  PL/pgSQL function p_exception() line 5 at RAISE
postgres=#
```

如果在事务中执行这个存储过程,则事务会中止(abort)。

RAISE EXCEPTION 自定义ERRCODE

```
postgres=# CREATE OR REPLACE PROCEDURE p_exception_errcode() AS
$$
DECLARE
   v_int INTEGER := 1;
BEGIN
   RAISE EXCEPTION ' 程序EXCEPTION ' USING ERRCODE = '23505';
END;
$$
LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=# CALL p_exception_errcode();
ERROR:  程序EXCEPTION
CONTEXT:  PL/pgSQL function p_exception_errcode() line 5 at RAISE
postgres=#
#日志中会记录这个ERRCODE
2021-04-25 11:25:25.958 CST,"tbase","postgres",30486,coord(30486,135876),"172.16.64.8:62252",6084c
50a.7716,coord(30486,135876),91,"CALL",2021-04-25 09:25:30 CST,11/135876,0,ERROR,23505," 程序EX
CEPTION ",,,,,"PL/pgSQL function p_exception_errcode() line 5 at RAISE","CALL p_exception_errcode
();",,,"psql"
```

oracle存储过程兼容性
不带参数的存储过程不需要括号

```
postgres=#  set enable_oracle_compatible to on;
SET
```

- 带括号创建存储过程。

```
postgres=# create or replace procedure no_para() is
begin
   raise notice 'no_para';
end;
/
CREATE PROCEDURE
```

- 不带括号创建存储过程。

```
postgres=# create or replace procedure no_para is
begin
   raise notice 'no_para';
end;
/
CREATE PROCEDURE
postgres=# call no_para();
NOTICE:  no_para
CALL
```

支持使用is语法

- as语法。

```
postgres=# create or replace procedure p_is() as
begin
   raise notice '兼容 is 语法';
end;
/
CREATE PROCEDURE
```

- is 语法。

```
postgres=# create or replace procedure p_is() is
begin
   raise notice '兼容 is 语法';
end;
/
CREATE PROCEDURE
postgres=# call p_is();
NOTICE:  兼容 is 语法
CALL
postgres=#
```

支持不使用$$语法

- 使用$$符号。

```
postgres=# create or replace procedure p_dollor() is
$$
begin
    raise notice '使用美元符号';
end;
$$
/
CREATE PROCEDURE
```

- 不使用$$符号。

```
postgres=# create or replace procedure p_dollor() is
begin
    raise notice '不使用美元符号';
end;
/
CREATE PROCEDURE
postgres=# call p_dollor();
NOTICE:  不使用美元符号
CALL
postgres=#
```

支持使用”end存储过程名称“结束

```
postgres=# create or replace procedure p_end_procedure() is
begin
    raise notice '使用 "end 存储过程名称" 结束 ';
end p_end_procedure;
/
CREATE PROCEDURE
postgres=# call p_end_procedure();
NOTICE:  使用 "end 存储过程名称" 结束
CALL
postgres=#
```

使用“/“结束函数定义

- 使用language plpgsql。

```
postgres=# create or replace procedure p_end_line() is
$$
begin
    raise notice '使用斜线结束定义';
end ;
$$
language plpgsql;
CREATE PROCEDURE
postgres=#
```

- 使用斜线结束。

```
postgres=# create or replace procedure p_end_line() is
begin
    raise notice '使用斜线结束定义';
end ;
/
CREATE PROCEDURE
postgres=# call p_end_line();
NOTICE:  使用斜线结束定义
CALL
postgres=#
```

定义变量无需要declare

- declare用法。

```
postgres=# create or replace procedure p_declare() is
declare
   v_int int;
begin
   v_int:=1;
   raise notice 'v_int = %',v_int;
end ;
/
CREATE PROCEDURE
```

- 无declare用法。

```
postgres=# create or replace procedure p_declare() is
   v_int int;
begin
   v_int:=1;
   raise notice 'v_int = %',v_int;
end ;
/
CREATE PROCEDURE
postgres=# call p_declare();
NOTICE:  v_int = 1
CALL
postgres=#
```

调用存储过程不需要call

```
postgres=# create or replace procedure p_call_test() is
begin
    raise notice '存储过程调用';
end ;
/
CREATE PROCEDURE
postgres=# create or replace procedure p_call() is
begin
    call p_call_test();
    p_call_test();
end ;
/
CREATE PROCEDURE
postgres=#
postgres=# call p_call();
NOTICE:  存储过程调用
NOTICE:  存储过程调用
CALL
postgres=#
```

支持存储过程out返回值

```
postgres=# create or replace procedure triple(x out number) is
begin
    x:=10;
end ;
/
CREATE PROCEDURE
postgres=# declare
  test number:=1;
begin
  triple(test);
  raise notice 'test = %',test;
end;
/
NOTICE:  test = 10
DO
postgres=#
postgres=# call triple(1);
 x
----
 10
(1 row)
```

oracle语法改写
forall改写

- Oracle语法。

```
create table t5(f1 int,f2 int);
insert into t5 values(1,1);
insert into t5 values(2,2);
commit;

create table t6(f1 int,f2 int);

create or replace procedure p_forall is
   TYPE t5list IS TABLE OF t5.f2%TYPE;
   t5s t5list;
BEGIN
   SELECT f2 BULK COLLECT INTO t5s FROM t5;
   FORALL i IN t5s.FIRST .. t5s.LAST
      insert into t6 values(t5s(i),t5s(i));
   commit;
END;
/
```

- TDSQL-PG改写。

```
create or replace procedure p_forall is
   TYPE t5list IS TABLE OF t5.f2%TYPE;
   t5s t5list;
BEGIN
   SELECT f2 BULK COLLECT INTO t5s FROM t5;
   FOR i IN t5s.FIRST .. t5s.LAST
   LOOP
      insert into t6 values(t5s[i],t5s[i]);
   END LOOP;
   commit;
END;
/
```

table函数

- Oracle用法。

```
create table t_table(f1 varchar2(36),f2 varchar2(36),f3 varchar2(36));
insert into t_table values('tbase','tbase','tbase');
insert into t_table values('Tbase','Tbase','Tbase');
commit;
create or replace type ty_row as object
(
  col1 varchar2(36),
  col2 varchar2(36),
  col3 varchar2(36)
);
/
create or replace type ty_table as table of ty_row;
create or replace function f_table  return ty_table as
  v_ty_table ty_table;
begin
  select ty_row(f1,f2,f3) bulk collect  into v_ty_table from t_table;
  return v_ty_table;
end;
/
#调用，使用table返回一个表格
select * from table(f_table());
```

- TDSQL-PG改写。

```
create table t_table(f1 varchar2(36),f2 varchar2(36),f3 varchar2(36));
insert into t_table values('tbase','tbase','tbase');
insert into t_table values('Tbase','Tbase','Tbase');
create or replace function f_table  returns setof t_table as
  v_rec record;
begin
  for v_rec in select * from t_table loop
      return next v_rec;
  end loop;
  return;
end;
/
#调用
select * from f_table();
```

应用程序样例

java

创建数据表

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
public class createtable {
  public static void main( String args[] )
    {
      Connection c = null;
      Statement stmt = null;
      try {
        Class.forName("org.postgresql.Driver");
        c = DriverManager.getConnection("jdbc:postgresql://127.0.0.1:15432/postgres?binaryTransfer=fa
lse","tbase", "tbase");
        System.out.println("Opened database successfully");
        stmt = c.createStatement();
        String sql = "create table public.tbase(id int,nickname text) distribute by shard(id) to group  defa
ult_group" ;
        stmt.executeUpdate(sql);
        stmt.close();
        c.close();
      } catch ( Exception e ) {
        System.err.println( e.getClass().getName()+": "+ e.getMessage() );
        System.exit(0);
      }
      System.out.println("Table created successfully");
    }
}
```

编译

```
javac createtable.java
```

执行

java createtable

> 注意：
>
> 连接url中禁止使用currentSchema=xxxx来指定当前搜索路径。

插入数据

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
public class insert {
  public static void main(String args[]) {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.postgresql.Driver");
      c = DriverManager.getConnection("jdbc:postgresql://127.0.0.1:15432/postgres?binaryTransfer=fa
lse","tbase", "tbase");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");
      stmt = c.createStatement();
      String sql = "INSERT INTO public.tbase (id,nickname) "
           + "VALUES (1,'tbase');";
      stmt.executeUpdate(sql);

      sql = "INSERT INTO tbase (id,nickname) "
           + "VALUES (2, 'pgxz' ),(3,'pgxc');";
      stmt.executeUpdate(sql);
      stmt.close();
      c.commit();
      c.close();
    } catch (Exception e) {
      System.err.println( e.getClass().getName()+": "+ e.getMessage() );
      System.exit(0);
    }
    System.out.println("Records created successfully");
  }
}
```

扩展协议插入数据

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.*;
import java.util.Random;
public class insert_prepared_return {
    public static void main(String args[]) {
        Connection c = null;
        PreparedStatement stmt;
        try {
            Class.forName("org.postgresql.Driver");
            c = DriverManager.getConnection("jdbc:postgresql://172.16.0.23:11345/postgres","adichen", "t
base");
            c.setAutoCommit(true);
            System.out.println("Opened database successfully");
            //插入数据
            String sql = "INSERT INTO public.t1 (f1,f2) VALUES (?,?) returning *";
            stmt = c.prepareStatement(sql,Statement.RETURN_GENERATED_KEYS);
            stmt.setInt(1, 1);
            stmt.setString(2, "111");
            stmt.executeUpdate();

            ResultSet generatedKeys = stmt.getGeneratedKeys();
            if (generatedKeys.next()) {
            long f1 = generatedKeys.getLong(1);
            String f2 = generatedKeys.getString(2);
            System.out.println(f1);
            System.out.println(f2);
            }
            generatedKeys.close();
            stmt.close();
            //c.commit();
            c.close();
        } catch (Exception e) {
            System.err.println( e.getClass().getName()+": "+ e.getMessage() );
            System.exit(0);
        }
        System.out.println("Records created successfully");
    }
}
```

扩展协议插入返回数据

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.*;
import java.util.Random;
public class insert_prepared_return {
   public static void main(String args[]) {
      Connection c = null;
      PreparedStatement stmt;
      try {
         Class.forName("org.postgresql.Driver");
         c = DriverManager.getConnection("jdbc:postgresql://172.16.0.23:11345/postgres","adichen", "t
base");
         c.setAutoCommit(true);
         System.out.println("Opened database successfully");
         //插入数据
         String sql = "INSERT INTO public.t1 (f1,f2) VALUES (?,?) returning *";
         stmt = c.prepareStatement(sql,Statement.RETURN_GENERATED_KEYS);
         stmt.setInt(1, 1);
         stmt.setString(2, "111");
         stmt.executeUpdate();

         ResultSet generatedKeys = stmt.getGeneratedKeys();
         if (generatedKeys.next()) {
         long f1 = generatedKeys.getLong(1);
         String f2 = generatedKeys.getString(2);
         System.out.println(f1);
         System.out.println(f2);
         }
         generatedKeys.close();
         stmt.close();
         //c.commit();
         c.close();
      } catch (Exception e) {
         System.err.println( e.getClass().getName()+": "+ e.getMessage() );
         System.exit(0);
      }
      System.out.println("Records created successfully");
   }
}
```

查询数据

```
/*
表及数据
create table select_normal(f1 int not null ,f2 date not null,f2 varchar(10));
insert into select_normal values(1,'2020-01-01','tbase'),(1,'2020-02-01','tbase');
*/
import java.sql.*;

public class select_bind {
    public static void main(String args[]) {
    Connection c = null;
    Statement stmt = null;
    try {
        Class.forName("org.postgresql.Driver");
        c = DriverManager.getConnection("jdbc:postgresql://172.16.0.30:11345/postgres","tbase", "tbas
e");
        c.setAutoCommit(false);
        System.out.println("Opened database successfully");
        long start_time = System.currentTimeMillis();
        PreparedStatement preparedStatement = null;
        try {
            stmt = c.createStatement();
            ResultSet resultSet = stmt.executeQuery("select * from public.select_normal where f2 >= '202
0-01-01' and f2< '2020-02-01'");
            while(resultSet.next()){
            System.out.println(resultSet.getString(1));
            System.out.println(resultSet.getString(2));
            System.out.println(resultSet.getString(3));
            }
            resultSet.close();
            stmt.close();
            c.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    } catch (Exception e) {
        System.err.println( e.getClass().getName()+": "+ e.getMessage() );
        System.exit(0);
    }
        System.out.println("Records created successfully");
    }
}
```

扩展协议查询数据

```
import java.sql.*;
import java.text.SimpleDateFormat;
public class select_normal {
   public static void main(String args[]) {
   Connection c = null;
   Statement stmt = null;
   try {
      Class.forName("org.postgresql.Driver");
      c = DriverManager.getConnection("jdbc:postgresql://172.16.0.30:11345/postgres","tbase", "tbas
e");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");
      long start_time = System.currentTimeMillis();
      PreparedStatement preparedStatement = null;
      try {
         String sql = "select * from public.select_normal where f2 >= ? and f2 < ? " ;
         preparedStatement = c.prepareStatement(sql);
         String deskdate_start = "2020-01-01";
         String deskdate_end = "2020-02-01";
         Date deskdate_start_date = new java.sql.Date(new SimpleDateFormat("yyyy-MM-dd").parse(de
skdate_start).getTime());
         Date deskdate_end_date = new java.sql.Date(new SimpleDateFormat("yyyy-MM-dd").parse(des
kdate_end).getTime());
         preparedStatement.setDate(1,deskdate_start_date);
         preparedStatement.setDate(2,deskdate_end_date);
         ResultSet resultSet = preparedStatement.executeQuery();
         System.out.println("执行查询耗时："+(System.currentTimeMillis()-start_time)+"ms");
         while(resultSet.next()){
         System.out.println(resultSet.getString(1));
         System.out.println(resultSet.getString(2));
         System.out.println(resultSet.getString(3));
      }

      } catch (SQLException e) {
         e.printStackTrace();
      }

   } catch (Exception e) {
      System.err.println( e.getClass().getName()+": "+ e.getMessage() );
      System.exit(0);
   }
      System.out.println("Records created successfully");
   }
```

```
}
```

copy from 入库

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import org.postgresql.copy.CopyManager;
import org.postgresql.core.BaseConnection;
import java.io.*;
public class copyfrom {
  public static void main( String args[] )
    {
      Connection c = null;
      Statement stmt = null;
      FileInputStream fs = null;
      try {
        Class.forName("org.postgresql.Driver");
        c = DriverManager.getConnection("jdbc:postgresql://127.0.0.1:15432/postgres?binaryTransfer=fa
lse","tbase", "tbase");
        System.out.println("Opened database successfully");
        CopyManager cm = new CopyManager((BaseConnection) c);
        fs = new FileInputStream("/data/tbase/tbase.csv");
        String sql = "COPY public.tbase FROM STDIN DELIMITER AS ','";
        cm.copyIn(sql, fs);
        c.close();
        fs.close();
      } catch ( Exception e ) {
        System.err.println( e.getClass().getName()+": "+ e.getMessage() );
        System.exit(0);
      }
      System.out.println("Copy data successfully");
    }
}
```

copy from 数据流入库

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import org.postgresql.copy.CopyManager;
import org.postgresql.core.BaseConnection;
import java.io.*;
public class copy_from_buff {
  public static void main( String args[] )
    {
      Connection c = null;
      Statement stmt = null;
      FileInputStream fs = null;
      try {
        Class.forName("org.postgresql.Driver");
        c = DriverManager.getConnection("jdbc:postgresql://127.0.0.1:11379/postgres","tbase", "tbase");
        System.out.println("Opened database successfully");
        CopyManager cm = new CopyManager((BaseConnection) c);
        String delimiter = new String(";");
        StringBuilder buf = new StringBuilder();
        //create table t_buff(f1 int,f2 varchar(10));
        for (int i=0; i<10000; i++){
           buf.append(i + delimiter+ "tbase" + "\r\n");
        }
        StringReader reader = new StringReader(buf.toString());
        cm.copyIn("COPY public.t_buff FROM STDIN WITH DELIMITER ';'", reader);
        c.close();
      } catch ( Exception e ) {
        System.err.println( e.getClass().getName()+": "+ e.getMessage() );
        System.exit(0);
      }
      System.out.println("Copy data successfully");
    }
}
```

copy to出库

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import org.postgresql.copy.CopyManager;
import org.postgresql.core.BaseConnection;
import java.io.*;
public class copyto {
  public static void main( String args[] )
    {
      Connection c = null;
      Statement stmt = null;
      FileOutputStream fs = null;
      try {
        Class.forName("org.postgresql.Driver");
        c = DriverManager.getConnection("jdbc:postgresql://127.0.0.1:15432/postgres?binaryTransfer=fa
lse","tbase", "tbase");
        System.out.println("Opened database successfully");
        CopyManager cm = new CopyManager((BaseConnection) c);
        fs = new FileOutputStream("/data/tbase/tbase.csv");
        String sql = "COPY public.tbase TO STDOUT DELIMITER AS ','";
        cm.copyOut(sql, fs);
        c.close();
        fs.close();
      } catch ( Exception e ) {
        System.err.println( e.getClass().getName()+": "+ e.getMessage() );
        System.exit(0);
      }
      System.out.println("Copy data successfully");
    }
}
```

兼容oracle字段大写

```
import java.sql.*;
public class oracle_test {
  public static void main(String args[]) {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.postgresql.Driver");
      c = DriverManager.getConnection("jdbc:postgresql://172.16.0.30:11345/postgres?oracle_compile
=true","tbase", "tbase");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");

      PreparedStatement preparedStatement = null;
      try {
        preparedStatement = c.prepareStatement("select * from public.select_normal");
        ResultSetMetaData metaData =  preparedStatement.getMetaData();
        for (int i = 0; i < metaData.getColumnCount(); i++) {
          // resultSet数据下标从1开始
          String columnName = metaData.getColumnName(i + 1);
          int type = metaData.getColumnType(i + 1);
          System.out.println("列名："+columnName);
        }
      } catch (SQLException e) {
        e.printStackTrace();
      }
    } catch (Exception e) {
      System.err.println( e.getClass().getName()+": "+ e.getMessage() );
      System.exit(0);
    }
    System.out.println("Records created successfully");
  }
}
```

配置多个cn负载

```
import java.sql.*;
public class select_bind {
    public static void main(String args[]) {
    Connection c = null;
    Statement stmt = null;
    try {
        Class.forName("org.postgresql.Driver");
        c = DriverManager.getConnection("jdbc:postgresql://172.16.0.61:11345,172.16.0.30:11347/postgr
es?loadBalanceHosts=true","tbase", "tbase");
        c.setAutoCommit(false);
        System.out.println("Opened database successfully");
        long start_time = System.currentTimeMillis();
        PreparedStatement preparedStatement = null;
        try {
            stmt = c.createStatement();
            ResultSet resultSet = stmt.executeQuery("select * from public.select_normal where f2 >= '202
0-01-01' and f2< '2020-02-01'");
            while(resultSet.next()){
            System.out.println(resultSet.getString(1));
            System.out.println(resultSet.getString(2));
            System.out.println(resultSet.getString(3));
            }
            resultSet.close();
            stmt.close();
            c.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    } catch (Exception e) {
        System.err.println( e.getClass().getName()+": "+ e.getMessage() );
        System.exit(0);
    }
        System.out.println("Records created successfully");
    }
}
```

合并多条insert

```
import java.sql.*;
import java.util.Random;
import java.util.UUID;
public class batch {
    public static void main(String args[]) {
        Connection conn = null;
        Statement stmt = null;
        try {
            Class.forName("org.postgresql.Driver");
            conn = DriverManager.getConnection("jdbc:postgresql://172.16.0.30:11345/postgres?reWriteBa
tchedInserts=true","tbase", "tbase");
            conn.setAutoCommit(true);
            System.out.println("Opened database successfully");
            //开始执行批量插入
            String insert_sql = "INSERT INTO t1 VALUES (?,?)";
            try {
                assert conn != null;
                PreparedStatement preparedStatement = conn.prepareStatement(insert_sql);
                for (int i = 0; i < 100000; i++) {
                    preparedStatement.setInt(1, new Random().nextInt(1000));
                    preparedStatement.setString(2, UUID.randomUUID().toString());
                    preparedStatement.addBatch();
                    if (i % 500 == 0) {
                        preparedStatement.executeBatch();
                        preparedStatement.clearBatch();
                    }
                }
                preparedStatement.executeBatch();
                preparedStatement.clearBatch();
            } catch (SQLException e) {
                e.printStackTrace();
            }

        } catch (Exception e) {
            System.err.println( e.getClass().getName()+": "+ e.getMessage() );
            System.exit(0);
        }
        System.out.println("Records created successfully");
    }
}
```

jdbc驱动包

参见JDBC驱动包下载

C程序

连接服务

```
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"
int
main(int argc, char **argv){
    const char *conninfo;
    PGconn     *conn;
    if (argc > 1){
        conninfo = argv[1];
    }else{
        conninfo = "dbname = postgres";
    }
    conn = PQconnectdb(conninfo);
    if (PQstatus(conn) != CONNECTION_OK){
        fprintf(stderr, "连接数据库失败: %s",PQerrorMessage(conn));
    }else{
        printf("连接数据库成功！\n");
    }
    PQfinish(conn);
    return 0;
}
```

编译

```
gcc -c -I /usr/local/install/tbase_pgxz/include/ conn.c
gcc -o conn conn.o -L /usr/local/install/tbase_pgxz/lib/ -lpq
```

运行

```
./conn "host=172.16.0.3 dbname=postgres port=11000"
#连接数据库成功！
./conn "host=172.16.0.3 dbname=postgres port=15432 user=tbase"
#连接数据库成功！
```

建立数据表

```
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"
int
main(int argc, char **argv){
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    const char *sql = "create table tbase(id int,nickname text) distribute by shard(id) to group  default_g
roup";
    if (argc > 1){
        conninfo = argv[1];
    }else{
        conninfo = "dbname = postgres";
    }
    conn = PQconnectdb(conninfo);
    if (PQstatus(conn) != CONNECTION_OK){
        fprintf(stderr, "连接数据库失败: %s",PQerrorMessage(conn));
    }else{
        printf("连接数据库成功！\n");
    }
    res = PQexec(conn,sql);
    if(PQresultStatus(res) != PGRES_COMMAND_OK){
        fprintf(stderr, "建立数据表失败: %s",PQresultErrorMessage(res));
    }else{
        printf("建立数据表成功！\n");
    }
    PQclear(res);
    PQfinish(conn);
    return 0;
}
```

编译

```
gcc -c -I /usr/local/install/tbase_pgxz/include/ createtable.c
gcc -o createtable createtable.o -L /usr/local/install/tbase_pgxz/lib/ -lpq
```

运行

```
./createtable "port=11000 dbname=postgres"
#连接数据库成功！
#建立数据表成功！
```

插入数据

```
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"
int
main(int argc, char **argv){
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    const char *sql = "INSERT INTO tbase (id,nickname) values(1,'tbase'),(2,'pgxz')";
    if (argc > 1){
        conninfo = argv[1];
    }else{
        conninfo = "dbname = postgres";
    }
    conn = PQconnectdb(conninfo);
    if (PQstatus(conn) != CONNECTION_OK){
        fprintf(stderr, "连接数据库失败: %s",PQerrorMessage(conn));
    }else{
        printf("连接数据库成功！\n");
    }
    res = PQexec(conn,sql);
    if(PQresultStatus(res) != PGRES_COMMAND_OK){
        fprintf(stderr, "插入数据失败: %s",PQresultErrorMessage(res));
    }else{
        printf("插入数据成功！\n");
    }
    PQclear(res);
    PQfinish(conn);
    return 0;
}
```

编译

```
gcc -c -I /usr/local/install/tbase_pgxz/include/ insert.c
gcc -o insert insert.o -L /usr/local/install/tbase_pgxz/lib/ -lpq
```

运行

```
./insert "dbname=postgres port=15432"
#连接数据库成功！
#插入数据成功！
```

查询数据

```
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"
int
main(int argc, char **argv){
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    const char *sql = "select * from tbase";
    if (argc > 1){
        conninfo = argv[1];
    }else{
        conninfo = "dbname = postgres";
    }
    conn = PQconnectdb(conninfo);
    if (PQstatus(conn) != CONNECTION_OK){
        fprintf(stderr, "连接数据库失败: %s",PQerrorMessage(conn));
    }else{
        printf("连接数据库成功！\n");
    }
    res = PQexec(conn,sql);
    if(PQresultStatus(res) != PGRES_TUPLES_OK){
        fprintf(stderr, "插入数据失败: %s",PQresultErrorMessage(res));
    }else{
        printf("查询数据成功！\n");
        int rownum = PQntuples(res) ;
        int colnum = PQnfields(res);
        for(int j = 0;j< colnum; ++j){
            printf("%s\t",PQfname(res,j));
        }
        printf("\n");
        for(int i = 0;i< rownum; ++i){
            for(int j = 0;j< colnum; ++j){
                printf("%s\t",PQgetvalue(res,i,j));
            }
            printf("\n");
        }
    }
    PQclear(res);
    PQfinish(conn);
    return 0;
}
```

编译

```
gcc -std=c99 -c -I /usr/local/install/tbase_pgxz/include/ select.c
gcc -o select select.o -L /usr/local/install/tbase_pgxz/lib/ -lpq
```

运行

```
./select "dbname=postgres port=15432"
#连接数据库成功！
#查询数据成功！
id      nickname
1       tbase
2       pgxz
```

copy入库

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"
int
main(int argc, char **argv){
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    const char *buffer = "1,tbase\n2,pgxz\n3,Tbase牛";
    if (argc > 1){
        conninfo = argv[1];
    }else{
        conninfo = "dbname = postgres";
    }
    conn = PQconnectdb(conninfo);
    if (PQstatus(conn) != CONNECTION_OK){
        fprintf(stderr, "连接数据库失败: %s",PQerrorMessage(conn));
    }else{
        printf("连接数据库成功！\n");
    }
    res=PQexec(conn,"COPY tbase FROM STDIN DELIMITER ',';");
    if(PQresultStatus(res) != PGRES_COPY_IN){
        fprintf(stderr, "copy数据出错1: %s",PQresultErrorMessage(res));
    }else{
        int len = strlen(buffer);
        if(PQputCopyData(conn,buffer,len) == 1){
            if(PQputCopyEnd(conn,NULL) == 1){
                res = PQgetResult(conn);
                if(PQresultStatus(res) == PGRES_COMMAND_OK){
                    printf("copy数据成功！\n");
                }else{
                    fprintf(stderr, "copy数据出错2: %s",PQerrorMessage(conn));
                }
            }else{
                fprintf(stderr, "copy数据出错3: %s",PQerrorMessage(conn));
            }
        }else{
            fprintf(stderr, "copy数据出错4: %s",PQerrorMessage(conn));
        }
    }
    PQclear(res);
    PQfinish(conn);
```

```
    return 0;
}
```

编译

```
gcc -c -I /usr/local/install/tbase_pgxz/include/ copy.c
gcc -o copy copy.o -L /usr/local/install/tbase_pgxz/lib/ -lpq
```

运行

```
./copy "dbname=postgres port=15432"
#连接数据库成功！
copy数据成功！
```

shell程序

```
#!/bin/sh
if [ $# -ne 0 ]
then
   echo "usage: $0 exec_sql"
   exit 1
fi
exec_sql=$1
masters=`psql -h 172.16.0.29 -d postgres -p 15432 -t -c "select string_agg(node_host, ' ') from (select *
from pgxc_node where node_type = 'D' order by node_name) t"`
port_list=`psql -h 172.16.0.29 -d postgres -p 15432 -t -c "select string_agg(node_port::text, ' ') from (sel
ect * from pgxc_node where node_type = 'D' order by node_name) t"`
node_cnt=`psql -h 172.16.0.29 -d postgres -p 15432 -t -c "select count(*) from pgxc_node where node
_type = 'D'"`
masters=($masters)
ports=($port_list)
echo $node_cnt
flag=0
for((i=0;i<$node_cnt;i++));
do
   seq=$(($i+1))
   master=${masters[$i]}
   port=${ports[$i]}
   echo $master
   echo $port

   psql -h $master -p $port  postgres -c "$exec_sql"
done
```

python程序
安装psycopg2模块

[root@VM_0_29_centos ~]# yum install python-psycopg2

连接服务

```
#coding=utf-8
#!/usr/bin/python
import psycopg2
try:
    conn = psycopg2.connect(database="postgres", user="tbase", password="", host="172.16.0.29", por
t="15432")
    print "连接数据库成功"
    conn.close()
except psycopg2.Error,msg:
    print "连接数据库出错，错误详细信息： %s" %(msg.args[0])
```

运行

```
[tbase@VM_0_29_centos python]$ python conn.py
#连接数据库成功
```

创建数据表

```
#coding=utf-8
#!/usr/bin/python
import psycopg2
try:
    conn = psycopg2.connect(database="postgres", user="tbase", password="", host="172.16.0.29", por
t="15432")
    print "连接数据库成功"
    cur = conn.cursor()
    sql = """
        create table tbase
        (
            id int,
            nickname varchar(100)
        )distribute by shard(id) to group default_group
        """
    cur.execute(sql)
    conn.commit()
    print "建立数据表成功"
    conn.close()
except psycopg2.Error,msg:
    print "TBase Error %s" %(msg.args[0])
```

运行

```
[tbase@VM_0_29_centos python]$ python createtable.py
#连接数据库成功
#建立数据表成功
```

新增数据

```
#coding=utf-8
#!/usr/bin/python
import psycopg2
try:
    conn = psycopg2.connect(database="postgres", user="tbase", password="", host="172.16.0.29", por
t="15432")
    print "连接数据库成功"
    cur = conn.cursor()
    sql = "insert into tbase values(1,'tbase'),(2,'tbase');"
    cur.execute(sql)
    sql = "insert into tbase values(%s,%s)"
    cur.execute(sql,(3,'pg'))
    conn.commit()
    print "插入数据成功"
    conn.close()
except psycopg2.Error,msg:
    print "操作数据库出库 %s" %(msg.args[0])
```

运行

```
[tbase@VM_0_29_centos python]$ python insert.py
#连接数据库成功
#插入数据成功
```

查询数据

```
#coding=utf-8
#!/usr/bin/python
import psycopg2
try:
    conn = psycopg2.connect(database="postgres", user="tbase", password="", host="172.16.0.29", por
t="15432")
    print "连接数据库成功"
    cur = conn.cursor()
    sql = "select * from tbase"
    cur.execute(sql)
    rows = cur.fetchall()
    for row in rows:
        print "ID = ", row[0]
        print "NICKNAME = ", row[1],"\n"
    conn.close()
except psycopg2.Error,msg:
    print "操作数据库出库 %s" %(msg.args[0])
```

运行

```
[tbase@VM_0_29_centos python]$ python select.py
#连接数据库成功
ID =  1
NICKNAME =  tbase
ID =  2
NICKNAME =  pgxz
ID =  3
NICKNAME =  pg
```

copy from方法

```
#coding=utf-8
#!/usr/bin/python
import psycopg2
try:
    conn = psycopg2.connect(database="postgres", user="tbase", password="", host="172.16.0.29", por
t="15432")
    print "连接数据库成功"
    cur = conn.cursor()
    filename = "/data/tbase/tbase.txt"
    cols = ('id','nickname')
    tablename="public.tbase"
    cur.copy_from(file=open(filename),table=tablename,columns=cols,sep=',')
    conn.commit()
    print "导入数据成功"
    conn.close()
except psycopg2.Error,msg:
    print "操作数据库出库 %s" %(msg.args[0])
```

运行

```
[tbase@VM_0_29_centos python]$ python copy_from.py
#连接数据库成功
#导入数据成功
```

php程序
连接服务

```
<?php
$host="172.16.0.29";
$port="15432";
$dbname="postgres";
$user="tbase" ;
$password="";
//连接数据库
$conn=@pg_connect("host=$host port=$port dbname=$dbname user=$user password=$password");
if (!$conn){
   $error_msg=@pg_errormessage($conn);
   echo "连接数据库出错，详情：".$error_msg."\n
   exit;
}else{
   echo "连接数据库成功"."\n
}
//关闭连接
pg_close($conn);
?>
```

执行

```
[root@VM_0_47_centos test]# curl http://127.0.0.1:8080/dbsta/test/conn.php
#连接数据库成功
```

创建数据表

```
<?php
$host="172.16.0.29";
$port="15432";
$dbname="postgres";
$user="tbase" ;
$password="";
//连接数据库
$conn=@pg_connect("host=$host port=$port dbname=$dbname user=$user password=$password");
if (!$conn){
   $error_msg=@pg_errormessage($conn);
   echo "连接数据库出错，详情：".$error_msg."\n"; ;
   exit;
}else{
   echo "连接数据库成功"."\n";
}

//建立数据表
$sql="create table public.tbase(id integer,nickname varchar(100)) distribute by shard(id) to group defa
ult_group;";
$result = @pg_exec($conn,$sql) ;
if (!$result){
   $error_msg=@pg_errormessage($conn);
   echo "创建数据表出错，详情：".$error_msg."\n"; ;
   exit;
}else{
   echo "创建数据表成功"."\n";
}
//关闭连接
pg_close($conn);
?>
```

执行

```
[root@VM_0_47_centos test]# curl http://127.0.0.1:8080/dbsta/test/createtable.php
#连接数据库成功
#创建数据表成功
```

插入数据

```
<?php
$host="172.16.0.29";
$port="15432";
$dbname="postgres";
$user="tbase" ;
$password="";
//连接数据库
$conn=@pg_connect("host=$host port=$port dbname=$dbname user=$user password=$password");
if (!$conn){
   $error_msg=@pg_errormessage($conn);
   echo "连接数据库出错，详情：".$error_msg."\n"; ;
   exit;
}else{
   echo "连接数据库成功"."\n";
}
//插入数据
$sql="insert into public.tbase values(1,'tbase'),(2,'pgxz');";
$result = @pg_exec($conn,$sql) ;
if (!$result){
   $error_msg=@pg_errormessage($conn);
   echo "插入数据出错，详情：".$error_msg."\n";
   exit;
}else{
   echo "插入数据成功"."\n";
}
//关闭连接
pg_close($conn);
?>
```

执行

```
[tbase@VM_0_47_centos test]$ curl http://127.0.0.1:8080/dbsta/test/insert.php
#连接数据库成功
#插入数据成功
```

查询记录

```
<?php
$host="172.16.0.29";
$port="15432";
$dbname="postgres";
$user="tbase" ;
$password="";
//连接数据库
$conn=@pg_connect("host=$host port=$port dbname=$dbname user=$user password=$password");
if (!$conn){
   $error_msg=@pg_errormessage($conn);
   echo "连接数据库出错，详情：".$error_msg."\n"; ;
   exit;
}else{
   echo "连接数据库成功"."\n";
}
//查询数据
$sql="select id,nickname from public.tbase";
$result = @pg_exec($conn,$sql) ;
if (!$result){
   $error_msg=@pg_errormessage($conn);
   echo "查询数据出错，详情：".$error_msg."\n";
   exit;
}else{
   echo "插入数据成功"."\n";
}
$record_num = pg_numrows($result);
echo "返回记录数".$record_num."\n";
$rec=pg_fetch_all($result);
for($i=0;$i<$record_num;$i++){
   echo "记录数#".strval($i+1)."\n";
   echo "id：".$rec[$i]["id"]."\n";
   echo "nickname：".$rec[$i]["nickname"]."\n\n";
}
//关闭连接
pg_close($conn);
?>
```

调用方法

```
[root@VM_0_47_centos ~]# curl http://127.0.0.1:8080/dbsta/test/select.php
连接数据库成功
插入数据成功
返回记录数2
记录数#1
id：1
```

nickname：tbase
记录数#2
id：2
nickname：pgxz

copy from 方法

把一个php数组导入到数据表中。

```
<?php
$host="172.16.0.29";
$port="15432";
$dbname="postgres";
$user="tbase" ;
$password="";
//连接数据库
$conn=@pg_connect("host=$host port=$port dbname=$dbname user=$user password=$password");
if (!$conn){
   $error_msg=@pg_errormessage($conn);
   echo "连接数据库出错，详情：".$error_msg."\n"; ;
   exit;
}else{
   echo "连接数据库成功"."\n";
}
$row=ARRAY("1,TBase","2,pgxz");
$flag=pg_copy_from($conn,"public.tbase",$row,",");

if (!$flag){
   $error_msg=@pg_errormessage($conn);
   echo "copy出错，详情：".$error_msg."\n";
}else{
   echo "copy成功"."\n";
}
//关闭连接
pg_close($conn);
?>
#调用方法
curl http://127.0.0.1/dbsta/cron/php_copy_from.php
#连接数据库成功
#copy成功
```

copy to 方法

将一个表的记录复制到一个php数据中。

```
<?php
$host="172.16.0.29";
$port="15432";
$dbname="postgres";
$user="tbase" ;
$password="";
//连接数据库
$conn=@pg_connect("host=$host port=$port dbname=$dbname user=$user password=$password");
if (!$conn){
   $error_msg=@pg_errormessage($conn);
   echo "连接数据库出错，详情：".$error_msg."\n"; ;
   exit;
}else{
   echo "连接数据库成功"."\n";
}
$row=pg_copy_to($conn,"public.tbase",",");
if (!$row){
   $error_msg=@pg_errormessage($conn);
   echo "copy出错，详情：".$error_msg."\n";
}else{
   print_r($row);
}
//关闭连接
pg_close($conn);
?>
#调用方法
curl http://127.0.0.1/dbsta/cron/php_copy_to.php
#连接数据库成功
Array
(
   [0] => 1,TBase
   [1] => 2,pgxz
)
```

入库去重方法

```
<?php
error_reporting(E_ALL && ~E_NOTICE);
ini_set('display_errors', '1');
set_time_limit(0);
$host="172.16.0.29";
$port="15432";
$dbname="postgres";
$user="tbase" ;
$password="";
/*
CREATE TABLE mydata (
    mpid text NOT NULL,
    datatype text NOT NULL,
    datatime timestamp without time zone NOT NULL,
    datavalue text,
    inputtime timestamp without time zone
)
DISTRIBUTE BY SHARD (mpid);

CREATE unique INDEX mydata_uidx ON mydata USING btree (mpid, datatype, datatime);
*/
//连接数据库
$conn=@pg_connect("host=$host port=$port dbname=$dbname user=$user password=$password");
if (!$conn){
    $error_msg=@pg_errormessage($conn);
    echo "连接数据库出错，详情：".$error_msg."\n"; ;
    exit;
}else{
    echo "连接数据库成功"."\n";
}
//建立临时表
$tmp_table_name="mydata_tmp_".strval(rand(100000000, 999999999));
$k=0;
do {
      $sql="
      CREATE TABLE ".$tmp_table_name." (
          mpid text NOT NULL,
          datatype text NOT NULL,
          datatime timestamp without time zone NOT NULL,
          datavalue text,
          inputtime timestamp without time zone
      )
      WITH OIDS DISTRIBUTE BY SHARD (mpid);
```

```
    ";
    $result = @pg_exec($conn,$sql) ;
    if ($result){
        //建议临时数据表成功,退出
        $sql="CREATE INDEX ".$tmp_table_name."_idx ON ".$tmp_table_name." USING btree (mpid, dat
atype, datatime); ";
        $result = @pg_exec($conn,$sql) ;
        if (!$result){
            $error_msg=@pg_errormessage($conn);
            echo "建立临时表索引失败，详情：".$error_msg."\n
            exit;
        }
        break;
    }else{
        $k++;
        if($k>10){
            $error_msg=@pg_errormessage($conn);
            echo "多次建立数据表失败，详情：".$error_msg."\n
            exit;
        }
    }
}while (true);
$sql="";

$mydata_sql="
INSERT INTO mydata (mpid, datatype,datatime,datavalue,inputtime) VALUES
";
$mydata_tmp_sql="
INSERT INTO ".$tmp_table_name." (mpid, datatype,datatime,datavalue,inputtime) VALUES
";

for ($i=0;$i<100;$i++){
   $insertnum = 250;
   for ($j=0;$j<$insertnum;$j++){
       $sql=$sql."
       ('".strval(rand(100000000, 999999999))."','10129f14','2018-03-15 02:00:00','004390.44','2018-03-1
5 11:05:55')
       ";
       if (($j+1)!=$insertnum){
           $sql=$sql.",";
       }
   }
   $execsql=$mydata_sql.$sql;
   $result = @pg_exec($conn,$execsql) ;

   if (!$result){
      ECHO "执行失败\n";
```

```
        //将数据导入到临时表
        $execsql=$mydata_tmp_sql.$sql;
        $result = @pg_exec($conn,$execsql) ;
        if (!$result){
            $error_msg=@pg_errormessage($conn);
            echo "数据插入临时表失败，详情：".$error_msg."\n
            exit;
        }
        //删除临时表中重复的数据,这一步最好在应用程序中去重，减少数据的负担
        $execsql="DELETE FROM ".$tmp_table_name." WHERE oid NOT IN (select min(oid) from ".$tmp_ta
ble_name." group by mpid, datatype, datatime)";
        $result = @pg_exec($conn,$execsql) ;
        if (!$result){
            $error_msg=@pg_errormessage($conn);
            echo "删除临时表重复数据失败，详情：".$error_msg."\n";
            exit;
        }
        $k=0;
        do {
            //删除重复数据
            $execsql="DELETE FROM ".$tmp_table_name." USING mydata WHERE mydata.mpid=".$tmp_
table_name.".mpid AND mydata.datatype=".$tmp_table_name.".datatype AND mydata.datatime=".$tm
p_table_name.".datatime" ;
            //将数据导入到正式表
            $execsql=$execsql.";INSERT INTO mydata SELECT * FROM ".$tmp_table_name;
            $result = @pg_exec($conn,$execsql) ;
            if ($result){
                //直到操作成功退出
                //退出前清理数据
                $execsql="truncate table ".$tmp_table_name;
                $result = @pg_exec($conn,$execsql) ;
                if (!$result){
                    $error_msg=@pg_errormessage($conn);
                    echo "truncate 临时表数据出错，详情：".$error_msg."\n";
                    exit;
                }
                ECHO "内层去重成功\n";
                break;
            }else{
                $k++;
                if($k>10){
                    $error_msg=@pg_errormessage($conn);
                    echo "多次删除重复数据失败，详情：".$error_msg."\n";
                    exit;
                }
            }
        } while (true);
```

```
    }else{
        ECHO "执行成功\n";
    }
    $sql="";
}

//退出前删除临时表
$execsql="drop table ".$tmp_table_name;
$result = @pg_exec($conn,$execsql) ;
if (!$result){
    $error_msg=@pg_errormessage($conn);
    echo "删除临时表数据出错，详情：".$error_msg."\n";
    exit;
}
//关闭连接
pg_close($conn); ;
?>
```

golang程序连接服务

```
package main
import (
    "fmt"
    "time"

    "github.com/jackc/pgx"
)
func main() {
    var error_msg string

    //连接数据库
    conn, err := db_connect()
    if err != nil {
        error_msg = "连接数据库失败，详情：" + err.Error()
        write_log("Error", error_msg)
        return
    }
    //程序运行结束时关闭连接
    defer conn.Close()
    write_log("Log", "连接数据库成功")
}
/*
功能描述：写入日志处理
参数说明：
log_level -- 日志级别，只能是是Error或Log
error_msg -- 日志内容
返回值说明：无
*/
func write_log(log_level string, error_msg string) {
    //打印错误信息
    fmt.Println("访问时间：", time.Now().Format("2006-01-02 15:04:05"))
    fmt.Println("日志级别：", log_level)
    fmt.Println("详细信息：", error_msg)
}
/*
功能描述：连接数据库
参数说明：无
返回值说明：
conn *pgx.Conn -- 连接信息
err error --错误信息
*/
```

```
func db_connect() (conn *pgx.Conn, err error) {
    var config pgx.ConnConfig
    config.Host = "127.0.0.1"    //数据库主机host或ip
    config.User = "tbase"        //连接用户
    config.Password = "pgsql"    //用户密码
    config.Database = "postgres" //连接数据库名
    config.Port = 15432          //端口号
    conn, err = pgx.Connect(config)
    return conn, err
}
[root@VM_0_29_centos tbase]# go run conn.go
访问时间： 2018-04-03 20:40:28
日志级别： Log
详细信息： 连接数据库成功
编译后运行
[root@VM_0_29_centos tbase]# go build conn.go
[root@VM_0_29_centos tbase]# ./conn
访问时间： 2018-04-03 20:40:48
日志级别： Log
详细信息： 连接数据库成功
```

建立数据表

```
package main
import (
    "fmt"
    "time"
    "github.com/jackc/pgx"
)
func main() {
    var error_msg string
    var sql string
    //连接数据库
    conn, err := db_connect()
    if err != nil {
        error_msg = "连接数据库失败，详情：" + err.Error()
        write_log("Error", error_msg)
        return
    }
    //程序运行结束时关闭连接
    defer conn.Close()
    write_log("Log", "连接数据库成功")

    //建立数据表
    sql = "create table public.tbase(id varchar(20),nickname varchar(100)) distribute by shard(id) to grou
p  default_group;"
    _, err = conn.Exec(sql)
    if err != nil {
        error_msg = "创建数据表失败,详情：" + err.Error()
        write_log("Error", error_msg)
        return
    } else {
        write_log("Log", "创建数据表成功")
    }
}
/*
功能描述：写入日志处理
参数说明：
log_level -- 日志级别，只能是是Error或Log
error_msg -- 日志内容
返回值说明：无
*/
func write_log(log_level string, error_msg string) {
    //打印错误信息
    fmt.Println("访问时间：", time.Now().Format("2006-01-02 15:04:05"))
    fmt.Println("日志级别：", log_level)
```

```
    fmt.Println("详细信息：", error_msg)
}
/*
功能描述：连接数据库
参数说明：无
返回值说明：
conn *pgx.Conn -- 连接信息
err error --错误信息
*/
func db_connect() (conn *pgx.Conn, err error) {
    var config pgx.ConnConfig
    config.Host = "127.0.0.1"    //数据库主机host或ip
    config.User = "tbase"        //连接用户
    config.Password = "pgsql"    //用户密码
    config.Database = "postgres" //连接数据库名
    config.Port = 15432          //端口号
    conn, err = pgx.Connect(config)
    return conn, err
}
[root@VM_0_29_centos tbase]# go run createtable.go
访问时间： 2018-04-03 20:50:24
日志级别： Log
详细信息： 连接数据库成功
访问时间： 2018-04-03 20:50:24
日志级别： Log
详细信息： 创建数据表成功
```

插入数据

```
package main
import (
    "fmt"
    "strings"
    "time"
    "github.com/jackc/pgx"
)
func main() {
    var error_msg string
    var sql string
    var nickname string
    //连接数据库
    conn, err := db_connect()
    if err != nil {
        error_msg = "连接数据库失败，详情：" + err.Error()
        write_log("Error", error_msg)
        return
    }
    //程序运行结束时关闭连接
    defer conn.Close()
    write_log("Log", "连接数据库成功")
    //插入数据
    sql = "insert into public.tbase values('1','tbase'),('2','pgxz');"
    _, err = conn.Exec(sql)
    if err != nil {
        error_msg = "插入数据失败,详情：" + err.Error()
        write_log("Error", error_msg)
        return
    } else {
        write_log("Log", "插入数据成功")
    }
    //绑定变量插入数据,不需要做防注入处理
    sql = "insert into public.tbase values($1,$2),($1,$3);"
    _, err = conn.Exec(sql, "3", "postgresql", "postgres")
    if err != nil {
        error_msg = "插入数据失败,详情：" + err.Error()
        write_log("Error", error_msg)
        return
    } else {
        write_log("Log", "插入数据成功")
    }
    //拼接sql语句插入数据,需要做防注入处理
    nickname = "TBase is ' good!"
```

```
    sql = "insert into public.tbase values('1','" + sql_data_encode(nickname) + "')"
    _, err = conn.Exec(sql)
    if err != nil {
        error_msg = "插入数据失败,详情：" + err.Error()
        write_log("Error", error_msg)
        return
    } else {
        write_log("Log", "插入数据成功")
    }
}
/*
功能描述：sql查询拼接字符串编码
参数说明：
str -- 要编码的字符串
返回值说明：
返回编码过的字符串
*/
func sql_data_encode(str string) string {
    return strings.Replace(str, "'", "''", -1)
}
/*
功能描述：写入日志处理
参数说明：
log_level -- 日志级别，只能是是Error或Log
error_msg -- 日志内容
返回值说明：无
*/
func write_log(log_level string, error_msg string) {
    //打印错误信息
    fmt.Println("访问时间：", time.Now().Format("2006-01-02 15:04:05"))
    fmt.Println("日志级别：", log_level)
    fmt.Println("详细信息：", error_msg)
}
/*
功能描述：连接数据库
参数说明：无
返回值说明：
conn *pgx.Conn -- 连接信息
err error --错误信息
*/
func db_connect() (conn *pgx.Conn, err error) {
    var config pgx.ConnConfig
    config.Host = "127.0.0.1"    //数据库主机host或ip
    config.User = "tbase"        //连接用户
    config.Password = "pgsql"    //用户密码
    config.Database = "postgres" //连接数据库名
    config.Port = 15432          //端口号
```

```
    conn, err = pgx.Connect(config)
    return conn, err
}
[root@VM_0_29_centos tbase]# go run insert.go
访问时间： 2018-04-03 21:05:51
日志级别： Log
详细信息： 连接数据库成功
访问时间： 2018-04-03 21:05:51
日志级别： Log
详细信息： 插入数据成功
访问时间： 2018-04-03 21:05:51
日志级别： Log
详细信息： 插入数据成功
访问时间： 2018-04-03 21:05:51
日志级别： Log
详细信息： 插入数据成功
```

查询数据

```
package main
import (
    "fmt"
    "strings"
    "time"
    "github.com/jackc/pgx"
)
func main() {
    var error_msg string
    var sql string
    //连接数据库
    conn, err := db_connect()
    if err != nil {
        error_msg = "连接数据库失败，详情：" + err.Error()
        write_log("Error", error_msg)
        return
    }
    //程序运行结束时关闭连接
    defer conn.Close()
    write_log("Log", "连接数据库成功")
    sql = "SELECT id,nickname FROM public.tbase LIMIT 2"
    rows, err := conn.Query(sql)
    if err != nil {
        error_msg = "查询数据失败,详情：" + err.Error()
        write_log("Error", error_msg)
        return
    } else {
        write_log("Log", "查询数据成功")
    }
    var nickname string
    var id string

    for rows.Next() {
        err = rows.Scan(&id, &nickname)
        if err != nil {
            error_msg = "执行查询失败，详情：" + err.Error()
            write_log("Error", error_msg)
            return
        }
        error_msg = fmt.Sprintf("id：%s nickname：%s", id, nickname)
        write_log("Log", error_msg)
    }
    rows.Close()
```

```
    nickname = "tbase"
    sql = "SELECT id,nickname FROM public.tbase WHERE nickname ='" + sql_data_encode(nickname) +
"' "
    rows, err = conn.Query(sql)
    if err != nil {
        error_msg = "查询数据失败,详情：" + err.Error()
        write_log("Error", error_msg)
        return
    } else {
        write_log("Log", "查询数据成功")
    }
    defer rows.Close()

    for rows.Next() {
        err = rows.Scan(&id, &nickname)
        if err != nil {
            error_msg = "执行查询失败，详情：" + err.Error()
            write_log("Error", error_msg)
            return
        }
        error_msg = fmt.Sprintf("id：%s nickname：%s", id, nickname)
        write_log("Log", error_msg)
    }
}
/*
功能描述：sql查询拼接字符串编码
参数说明：
str -- 要编码的字符串
返回值说明：
返回编码过的字符串
*/
func sql_data_encode(str string) string {
    return strings.Replace(str, "'", "''", -1)
}
/*
功能描述：写入日志处理
参数说明：
log_level -- 日志级别，只能是是Error或Log
error_msg -- 日志内容
返回值说明：无
*/
func write_log(log_level string, error_msg string) {
    //打印错误信息
    fmt.Println("访问时间：", time.Now().Format("2006-01-02 15:04:05"))
    fmt.Println("日志级别：", log_level)
    fmt.Println("详细信息：", error_msg)
}
```

```
/*
功能描述：连接数据库
参数说明：无
返回值说明：
conn *pgx.Conn -- 连接信息
err error --错误信息
*/
func db_connect() (conn *pgx.Conn, err error) {
    var config pgx.ConnConfig
    config.Host = "127.0.0.1"    //数据库主机host或ip
    config.User = "tbase"        //连接用户
    config.Password = "pgsql"    //用户密码
    config.Database = "postgres" //连接数据库名
    config.Port = 15432          //端口号
    conn, err = pgx.Connect(config)
    return conn, err
}
[root@VM_0_29_centos tbase]# go run select.go
访问时间： 2018-04-09 10:35:50
日志级别： Log
详细信息： 连接数据库成功
访问时间： 2018-04-09 10:35:50
日志级别： Log
详细信息： 查询数据成功
访问时间： 2018-04-09 10:35:50
日志级别： Log
详细信息： id：2 nickname：tbase
访问时间： 2018-04-09 10:35:50
日志级别： Log
详细信息： id：3 nickname：postgresql
访问时间： 2018-04-09 10:35:50
日志级别： Log
详细信息： 查询数据成功
访问时间： 2018-04-09 10:35:50
日志级别： Log
详细信息： id：1 nickname：tbase
```

copy from 方法

```
package main
import (
    "fmt"
    "math/rand"
    "time"
    "github.com/jackc/pgx"
)
func main() {
    var error_msg string
    //连接数据库
    conn, err := db_connect()
    if err != nil {
        error_msg = "连接数据库失败，详情：" + err.Error()
        write_log("Error", error_msg)
        return
    }
    //程序运行结束时关闭连接
    defer conn.Close()
    write_log("Log", "连接数据库成功")
    //构造5000行数据
    inputRows := [][]interface{}{}
    var id string
    var nickname string
    for i := 0; i < 5000; i++ {
        id = fmt.Sprintf("%d", rand.Intn(10000))
        nickname = fmt.Sprintf("%d", rand.Intn(10000))
        inputRows = append(inputRows, []interface{}{id, nickname})
    }
    copyCount, err := conn.CopyFrom(pgx.Identifier{"tbase"}, []string{"id", "nickname"}, pgx.CopyFromRo
ws(inputRows))
    if err != nil {
        error_msg = "执行copyFrom失败,详情：" + err.Error()
        write_log("Error", error_msg)
        return
    }
    if copyCount != len(inputRows) {
        error_msg = fmt.Sprintf("执行copyFrom失败，copy行数：%d 返回行数为：%d", len(inputRows), cop
yCount)
        write_log("Error", error_msg)
        return
    } else {
        error_msg = "Copy 记录成功"
        write_log("Log", error_msg)
```

```
    }
}
/*
功能描述：写入日志处理
参数说明：
log_level -- 日志级别，只能是是Error或Log
error_msg -- 日志内容
返回值说明：无
*/
func write_log(log_level string, error_msg string) {
    //打印错误信息
    fmt.Println("访问时间：", time.Now().Format("2006-01-02 15:04:05"))
    fmt.Println("日志级别：", log_level)
    fmt.Println("详细信息：", error_msg)
}
/*
功能描述：连接数据库
参数说明：无
返回值说明：
conn *pgx.Conn -- 连接信息
err error --错误信息
*/
func db_connect() (conn *pgx.Conn, err error) {
    var config pgx.ConnConfig
    config.Host = "127.0.0.1"    //数据库主机host或ip
    config.User = "tbase"        //连接用户
    config.Password = "pgsql"    //用户密码
    config.Database = "postgres" //连接数据库名
    config.Port = 15432          //端口号
    conn, err = pgx.Connect(config)
    return conn, err
}
[root@VM_0_29_centos tbase]# go run copy_from.go
访问时间： 2018-04-09 10:36:40
日志级别： Log
详细信息： 连接数据库成功
访问时间： 2018-04-09 10:36:40
日志级别： Log
详细信息： Copy 记录成功
```

go相关资源包

需要git的资源包：

jackc/pgx

pkg/errors

关于运维
数据库管理
创建数据库

要创建一个数据库，你必须是一个超级用户或者具有特殊的CREATEDB特权，默认情况下，新数据库将通过克隆标准系统数据库template1被创建。可以通过写TEMPLATE name指定一个不同的模板。特别地，通过写TEMPLATE template0你可以创建一个干净的数据库，它将只包含你的TDSQL PG所预定义的标准对象。

- 默认参数创建数据库。

```
postgres=# create database tbase_db;
CREATE DATABASE
```

- 指定克隆库。

```
postgres=# create database tbase_db_template TEMPLATE template0;
CREATE DATABASE
```

- 指定所有者。

```
postgres=# create role pgxz with login;
CREATE ROLE
postgres=# create database tbase_db_owner owner pgxz;
CREATE DATABASE
postgres=# \l+ tbase_db_owner
                                    List of databases
    Name      | Owner | Encoding|  Collate   |  Ctype    | Access privileges |Size  | Tablespace | Description
---------------+-------+----------+------------+------------+-------------------+-------+------------
+-------------
 tbase_db_owner | pgxz  | UTF8     | en_US.utf8 | en_US.utf8 |                   | 18 MB | pg_default |
(1 row)
```

- 指定编码。

```
postgres=# create database tbase_db_encoding ENCODING UTF8;
CREATE DATABASE
postgres=# \l+  tbase_db_encoding
                                    List of databases
    Name      | Owner | Encoding|  Collate   |  Ctype    | Access privileges |Size  | Tablespace | Description
------------------+-------+----------+------------+------------+-------------------+-------+------------
```

```
+-------------
 tbase_db_encoding | tbase | UTF8     | en_US.utf8 | en_US.utf8 |                | 18 MB | pg_default |
(1 row)
```

- 创建gbk编码。

```
postgres=# CREATE DATABASE db_gbk template template0 encoding = gbk LC_COLLATE = 'zh_CN.gbk
' LC_CTYPE = 'zh_CN.gbk';
CREATE DATABASE
postgres=# \l+
                                    List of databases
    Name      | Owner | Encoding|  Collate   |  Ctype    | Access privileges |Size  | Tablespace | Description
-----------+-------+----------+------------+------------+-------------------+-------+------------
+-----------------------------------------
 db_gbk    | tbase | GBK      | zh_CN.gbk  | zh_CN.gbk  |                   | 19 MB | pg_default |
 postgres  | tbase | UTF8     | zh_CN.utf8 | zh_CN.utf8 |                   | 26 MB | pg_default | default administ
rative connection database
 template0 | tbase | UTF8     | zh_CN.utf8 | zh_CN.utf8 | =c/tbase          +| 19 MB | pg_default | unmodifia
ble empty database
           |       |          |            |            | tbase=CTc/tbase   |       |            |
 template1 | tbase | UTF8     | zh_CN.utf8 | zh_CN.utf8 | =c/tbase          +| 24 MB | pg_default | default te
mplate for new databases
           |       |          |            |            | tbase=CTc/tbase   |       |            |
(4 rows)
```

- 创建gb18030编码。

```
postgres=# create database db_gb18030 template template0 encoding=gb18030 LC_COLLATE = 'zh_C
N.gb18030' LC_CTYPE = 'zh_CN.gb18030';
CREATE DATABASE
postgres=# \l
                                    List of databases
   Name    | Owner | Encoding |    Collate    |     Ctype     | Access privileges
-----------+-------+----------+---------------+---------------+-------------------
 db_gb18030 | tbase | GB18030  | zh_CN.gb18030 | zh_CN.gb18030 |
 db_gbk     | tbase | GBK      | zh_CN.gbk     | zh_CN.gbk     |
 postgres   | tbase | UTF8     | zh_CN.utf8    | zh_CN.utf8    |
 template0  | tbase | UTF8     | zh_CN.utf8    | zh_CN.utf8    | =c/tbase        +
            |       |          |               |               | tbase=CTc/tbase
 template1  | tbase | UTF8     | zh_CN.utf8    | zh_CN.utf8    | =c/tbase        +
            |       |          |               |               | tbase=CTc/tbase
 test       | tbase | UTF8     | zh_CN.utf8    | zh_CN.utf8    |
(6 rows)

postgres=#
```

- 指定排序规则。

```
postgres=# create database tbase_db_lc_collate lc_collate 'C';
CREATE DATABASE
postgres=# \l+ tbase_db_lc_collate
                                    List of databases
    Name      | Owner | Encoding|  Collate   |  Ctype    | Access privileges |Size  | Tablespace | Description
--------------------+-------+----------+---------+-----------+-------------------+-------+------------
+-------------
 tbase_db_lc_collate | tbase | UTF8     | C       | en_US.utf8 |                   | 18 MB | pg_default |
(1 row)
```

- 指定分组规则。

```
postgres=# create database tbase_db_lc_ctype LC_CTYPE 'C' ;
CREATE DATABASE
postgres=# \l+  tbase_db_lc_ctype
                                    List of databases
    Name      | Owner | Encoding|  Collate   |  Ctype    | Access privileges |Size  | Tablespace | Description
------------------+-------+----------+------------+-------+-------------------+-------+------------
+-------------
 tbase_db_lc_ctype | tbase | UTF8     | en_US.utf8 | C     |                   | 18 MB | pg_default |
(1 row)
```

- 配置数据可连接。

```
postgres=# create database tbase_db_allow_connections ALLOW_CONNECTIONS true;
CREATE DATABASE
postgres=# select datallowconn from pg_database where datname='tbase_db_allow_connections';
 datallowconn
--------------
 t
```

- 配置连接数。

```
postgres=# create database tbase_db_connlimit CONNECTION LIMIT 100;
CREATE DATABASE
postgres=# select datconnlimit  from pg_database where datname='tbase_db_connlimit';
 datconnlimit
--------------
          100
(1 row)
```

- 配置数据库可以被复制。

```
postgres=# create database tbase_db_istemplate is_template true;
CREATE DATABASE
postgres=# select datistemplate from pg_database where datname='tbase_db_istemplate';
 datistemplate
---------------
 t
(1 row)
```

- 多个参数一起配置。

```
postgres=# create database tbase_db_mul owner pgxz CONNECTION LIMIT 50 template template0 en
coding 'utf8'  lc_collate 'C';
CREATE DATABASE
```

修改数据库配置

- 修改数据库名称。

```
postgres=# alter database tbase_db rename to tbase_db_new;
ALTER DATABASE
```

- 修改连接数。

```
postgres=# alter database tbase_db_new connection limit 50;
ALTER DATABASE
```

- 修改数据库所有者。

```
postgres=# alter database tbase_db_new owner to tbase;
ALTER DATABASE
```

- 配置数据默认运行参行。

```
postgres=# alter database tbase_db_new set search_path to public,pg_catalog,pg_oracle;
ALTER DATABASE
```

- Alter database不支持的项目。
  不支持项目

项目	备注
encoding	编码
lc_collate	排序规则
lc_ctype	分组规则

删除数据库

```
postgres=# drop database tbase_db_new;
DROP DATABASE
```

删除数据库时，如果该数据库已经有session连接上来，则会提示如下错误：

```
postgres=# drop database tbase_db_template;
ERROR: database "tbase_db_template" is being accessed by other users
DETAIL: There is 1 other session using the database.

#使用下面方法可以把session断开，然后再删除

postgres=# select pg_terminate_backend(pid)from pg_stat_activity where datname='tbase_db_templat
e';
 pg_terminate_backend
----------------------
 t
(1 row)
postgres=# drop database tbase_db_template;
DROP DATABASE
```

会话及锁管理
查看当前会话的PID

```
postgres=# Select pg_backend_pid();
```

查看当前节点有那些会话

select * from pg_stat_activity;

- 查看活跃超时会话。

select pid,client_addr,state_change,query_start,state_change,EXTRACT(EPOCH FROM(now()-query_start)),query,state,usename,application_name from pg_stat_activity where EXTRACT(EPOCH FROM (now()-query_start))>60 and state!='idle';

- 按用户统计活跃连接数。

select count(1),usename from pg_stat_activity where EXTRACT(EPOCH FROM (now()-query_start))>60 and state!='idle' group by usename;

- 按客户端统计活跃连接数。

select count(1),client_addr from pg_stat_activity where EXTRACT(EPOCH FROM (now()-query_start))>60 and state!='idle' group by client_addr;

杀掉连接

```
postgres=# select pg_terminate_backend(pid);
```

⚠️pid为进程号

查看会话持锁情况

- 排它锁阻塞。

```
postgres=# select pid,pg_blocking_pids(pid),EXTRACT(EPOCH FROM(now()-query_start)),wait_event_type,query from pg_stat_activity where wait_event_type = 'Lock' and pid!=pg_backend_pid();
```

- 同步阻塞。

```
postgres=# select query,wait_event_type,wait_event from pg_stat_activity where wait_event='SyncRep';
```

配置会话锁超时

- 当前会话。

postgres=# set lock_timeout to '3s';

- 数据库默认锁超时。

postgres=# alter database postgres set lock_timeout to '3s';

- 用户默认锁超时。

postgres=# alter role tbase set lock_timeout to '3s';

问题定位及性能优化
访问日志管理
配置日志

TDSQL PG运行参数配置文件Postgresql.conf中涉及日志配置参数如下所示：

```
#日志相关配置
#用户访问日志格式支持多种方法来记录服务器消息，包括stderr、csvlog
#如果csvlog被包括在log_destination中，日志项会以"逗号分隔值" （CSV）格式被输出，这样可以很方便地把日志载入到程序中
log_destination = 'csvlog'
#启用用户访问日志收集器
logging_collector = on
#日志存放目录，可以是相对路径（相对了data目录）或绝对路径
log_directory = 'pg_log'
#设置日志文件的权限
log_file_mode = 0600
#按我先前经常是只有log_rotation_age这个值有效果
log_truncate_on_rotation = 'on'
#120分钟切换一次
log_rotation_age = 120
#64MB切换一次
log_rotation_size = 64MB
#会话的当前执行命令保留长度，用于在字段pg_stat_activity.query中显示
track_activity_query_size = 4096
#配置sql语句执行超过多少毫秒数时，语句将被记录,值为-1时禁用，0时记录所有语句
log_min_duration_statement = 1000
#是否记录checkpoint
log_checkpoints = on
#是否记录连接
log_connections = on
#是否记录断开连接
log_disconnections = on
#是否记录autovacuum执行日志
log_autovacuum_min_duration = 0
#是否记录所有语句的执行时间，值为on时将单独记录所有语句的执行时间，这里不记录语句
log_duration = off
#stderr配置日志记录的内容
log_line_prefix = '%h'
#控制那些类型的语句被记录，有效值是 none (off)、ddl、mod和 all（所有语句）。
#如果log_min_duration_statement 值为0时，则log_statement什么值的效果都一样
#如果log_min_duration_statement 值大于0，并且log_statement为ddl则ddl语句全部表被记录，为两条log
#dml超时才被记录，为一条记录
```

```
log_statement = 'none'
#设置在服务器日志中写入的时间戳的时区
log_timezone = 'PRC'
log_filename = 'postgresql-%A-%H.log'
#是否记录autovacuum执行日志
log_autovacuum_min_duration = 0
#控制被发送给客户端的消息级别
#其值有debug5,debug4,debug3,debug2,debug1,log,notice,warning,error
client_min_messages = notice
#控制哪些消息级别被写入到服务器日
#其值有debug5,debug4,debug3,debug2,debug1,log,notice,warning,error
log_min_messages = warning
#控制哪些导致一个错误情况的 SQL 语句被记录在服务器日志中
#默认值是ERROR,它表示导致错误、日志消息、致命错误或恐慌错误的语句将被记录在日志中
log_min_error_statement = error
```

日志格式说明

执行正确的语句产生的日志。

```
2017-10-11 16:23:55.178 CST,"pgxz","postgres",11499,"127.0.0.1:2329",59ddd50c.2ceb,1,"idle",2017-10-11 16:23:40 CST,3/26053,0,LOG,00000,"statement: select * from pg_class limit 1;",,,,,,,,,"psql"
执行时间              | 2017-10-11 16:23:55.178
用户名              | pgxz
数据库          | postgres
进程id              | 11499
客户端id        | 127.0.0.1:2329
会话 ID             | 59ddd50c.2ceb
每个会话的行号       | 1
命令标签            | idle
登录时间     | 2017-10-11 16:23:40
虚拟事务 ID| 3/26053
普通事务 ID、          | 0
级别         | LOG
SQLSTATE 代码         | 00000
执行信息              | statement: select * from pg_class limit 1;
详情                |
提示                |
导致错误的内部查询         |
错误位置所在的字符计数             |
错误上下文              |
导致错误的用户查询（如果有且被log_min_error_statement启用）                 |
错误位置所在的字符计数             |
在 PostgreSQL 源代码中错误的位置（如果log_error_verbosity被设置为verbose）               |
应用名       | psql
错误日志解释
2017-10-11 16:24:10.233 CST,"pgxz","postgres",11499,"127.0.0.1:2329",59ddd50c.2ceb,2,"idle",2017-10-11 16:23:40 CST,3/26054,0,LOG,00000,"statement: select * from pgxc_nodes limit 1;",,,,,,,,,"psql"
2017-10-11 16:24:10.233 CST,"pgxz","postgres",11499,"127.0.0.1:2329",59ddd50c.2ceb,3,"SELECT",2017-10-11 16:23:40 CST,3/26054,0,ERROR,42P01,"relation ""pgxc_nodes"" does not exist",,,,,,,,,"psql"
```

执行错误的语句会产生两条日志，一条是执行语句，一条提示出错的原因。

对日志进行分析

创建日志表

```
CREATE table tbase_log
(
  log_time timestamp without time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp without time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text
);
```

导入日志数据

日志文件默认存储在"数据目录 /pg_log 目录下面。

```
postgres=# COPY tbase_log FROM '/data/pgxz/data/pgxz/dn001/pg_log/postgresql-Tuesday-16.csv'
WITH csv;
COPY 10790
```

统计日志数据

- 按照session连接及操作时间排序

select * from tbase_log order by process_id,log_time;

- 查询错误日志

SELECT * FROM TBase_log WHERE error_severity='ERROR' limit 1;

- 统计session操作数统计

```
postgres=#  select count(1),process_id,user_name,database_name from tbase_log group by process_id,
user_name,database_name order by count(1) desc limit 10;
 count | process_id | user_name | database_name
-------+------------+-----------+---------------
  2770 |      48067 | pgxz      | postgres
    10 |      22143 | pgxz      | postgres
    10 |      28778 | pgxz      | postgres
     9 |      28367 | pgxz      | postgres
     9 |      44280 | pgxz      | postgres
     8 |      32442 | pgxz      | postgres
     7 |      17911 | pgxz      | postgres
     7 |      21865 | pgxz      | postgres
     7 |      26159 | pgxz      | postgres
     7 |      45471 | pgxz      | postgres
(10 rows)
```

- 用户操作统计

```
postgres=#  select count(1),user_name from tbase_log group by user_name order by count(1) desc limi
t 10;
 count | user_name
-------+-----------
 10790 | pgxz
```

- 数据库访问次数统计

```
postgres=#  select count(1),database_name from tbase_log group by database_name order by count
(1) desc limit 10;
 count | database_name
```

```
-------+---------------
 10790 | postgres
(1 row)
```

- 错误信息统计

```
postgres=# select count(1),user_name,database_name from tbase_log where error_severity='ERROR' gr
oup by user_name,database_name order by count(1) desc limit 10;
 count | user_name | database_name
-------+-----------+---------------
  1390 | pgxz      | postgres
(1 row)
```

配置只收集慢的sql语句

```
#用户访问日志格式
log_destination = 'csvlog'
#启用用户访问日志收集器
logging_collector = on
#配置sql语句执行超过多少毫秒数时，语句将被记录,值为-1时禁用，0时记录所有语句
#下面配置只收集运行超过1秒的语句
log_min_duration_statement = 1000
#默认不记录任何日志
log_statement = 'none'
```

收集到的日志文件内容如下所示：

```
2017-10-15 10:25:54.106 CST,"postgres","postgres",43799,"127.0.0.1:17899",59e2c65b.ab17,4,"SELEC
T",2017-10-15 10:22:19 CST,2/91749387678543872,0,LOG,00000,"duration: 1338.366 ms  statement: sel
ect * from t where id=20000 or id=2000000;",,,,,,,,,"psql"
```

系统记录运行的语句及运行时间。

如何查询数据是否倾斜

连接上不同的 dn 节点，分别查询数据表的容量大小，如果大小偏差较大就可以判断存在数据倾斜。

```
postgres=# select pg_size_pretty(pg_table_size('TBase_1'));
 pg_size_pretty
----------------
 2408 kB
(1 row)
postgres=# select pg_size_pretty(pg_table_size('TBase_2'));
 pg_size_pretty
----------------
 896 kB
(1 row)
```

如何优化有问题的Sql语句

查看是否为分布键查询

```
postgres=# explain select * from tbase_1 where f1=1;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Gather  (cost=1000.00..7827.20 rows=1 width=14)
         Workers Planned: 2
         ->  Parallel Seq Scan on tbase_1  (cost=0.00..6827.10 rows=1 width=14)
               Filter: (f1 = 1)
(6 rows)
postgres=# explain select * from tbase_1 where f2=1;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001
   ->  Gather  (cost=1000.00..7827.20 rows=1 width=14)
         Workers Planned: 2
         ->  Parallel Seq Scan on tbase_1  (cost=0.00..6827.10 rows=1 width=14)
               Filter: (f2 = 1)
(6 rows)
```

上面第一个查询为非分布键查询，需要发往所有节点，这样最慢的节点决定了整个业务的速度，需要保持所有节点的响应性能一致，业务设计查询时尽可能带上分布键。

查看是否使用上索引

```
postgres=# create index tbase_2_f2_idx on tbase_2(f2);
CREATE INDEX
postgres=# explain select * from tbase_2 where f2=1;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Index Scan using tbase_2_f2_idx on tbase_2  (cost=0.42..4.44 rows=1 width=14)
         Index Cond: (f2 = 1)
(4 rows)
postgres=# explain select * from tbase_2 where f3='1';
                               QUERY PLAN
-------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Gather  (cost=1000.00..7827.20 rows=1 width=14)
         Workers Planned: 2
         ->  Parallel Seq Scan on tbase_2  (cost=0.00..6827.10 rows=1 width=14)
               Filter: (f3 = '1'::text)
(6 rows)
postgres=#
```

第一个查询使用了索引，第二个没有使用索引，通常情况下，使用索引可以加速查询速度，但要记住索引也会增加更新的开销。

查看是否为分布key join

```
postgres=# explain  select tbase_1.* from tbase_1,tbase_2 where tbase_1.f1=tbase_2.f1 ;
                                  QUERY PLAN
-------------------------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001,dn002)  (cost=29.80..186.32 rows=3872 width=40)
   ->  Hash Join  (cost=29.80..186.32 rows=3872 width=40)
         Hash Cond: (tbase_1.f1 = tbase_2.f1)
         ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.00..158.40 rows=880 width=40)
               Distribute results by S: f1
               ->  Seq Scan on tbase_1  (cost=0.00..18.80 rows=880 width=40)
         ->  Hash  (cost=18.80..18.80 rows=880 width=4)
               ->  Seq Scan on tbase_2  (cost=0.00..18.80 rows=880 width=4)
(8 rows)
postgres=# explain  select tbase_1.* from tbase_1,tbase_2 where tbase_1.f2=tbase_2.f1 ;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Hash Join  (cost=18904.69..46257.08 rows=500564 width=14)
         Hash Cond: (tbase_1.f2 = tbase_2.f1)
         ->  Seq Scan on tbase_1  (cost=0.00..9225.64 rows=500564 width=14)
         ->  Hash  (cost=9225.64..9225.64 rows=500564 width=4)
               ->  Seq Scan on tbase_2  (cost=0.00..9225.64 rows=500564 width=4)
(7 rows)
```

第一查询需要数据重分布，而第二个是不需要，分布键join查询性能会更高。

查看join发生的节点

```
postgres=#  explain select tbase_1.* from tbase_1,tbase_2 where tbase_1.f1=tbase_2.f1 ;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Hash Join  (cost=29.80..186.32 rows=3872 width=40)
   Hash Cond: (tbase_1.f1 = tbase_2.f1)
   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.00..158.40 rows=880 width=40)
         ->  Seq Scan on tbase_1  (cost=0.00..18.80 rows=880 width=40)
   ->  Hash  (cost=126.72..126.72 rows=880 width=4)
         ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.00..126.72 rows=880 width=4)
               ->  Seq Scan on tbase_2  (cost=0.00..18.80 rows=880 width=4)
(7 rows)
postgres=# set prefer_olap to on;
SET
postgres=# explain select tbase_1.* from tbase_1,tbase_2 where tbase_1.f1=tbase_2.f1 ;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn001,dn002)  (cost=29.80..186.32 rows=3872 width=40)
   ->  Hash Join  (cost=29.80..186.32 rows=3872 width=40)
         Hash Cond: (tbase_1.f1 = tbase_2.f1)
         ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.00..158.40 rows=880 width=40)
               Distribute results by S: f1
               ->  Seq Scan on tbase_1  (cost=0.00..18.80 rows=880 width=40)
         ->  Hash  (cost=18.80..18.80 rows=880 width=4)
               ->  Seq Scan on tbase_2  (cost=0.00..18.80 rows=880 width=4)
(8 rows)
```

上面join在cn节点执行，下面的在dn上重分布后再join，业务上设计一般oltp类业务在cn上进行少数据量join性能会更好。

查看并行的worker数

```
postgres=# explain select count(1) from tbase_1;
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=118.81..118.83 rows=1 width=8)
   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=118.80..118.81 rows=1 width=0)
         ->  Partial Aggregate  (cost=18.80..18.81 rows=1 width=8)
               ->  Seq Scan on tbase_1  (cost=0.00..18.80 rows=880 width=0)
(4 rows)
postgres=# analyze tbase_1;
ANALYZE
postgres=# explain select count(1) from tbase_1;
                                      QUERY PLAN
---------------------------------------------------------------------------------------------------
 Parallel Finalize Aggregate  (cost=14728.45..14728.46 rows=1 width=8)
   ->  Parallel Remote Subquery Scan on all (dn001,dn002)  (cost=14728.33..14728.45 rows=1 width=0)
         ->  Gather  (cost=14628.33..14628.44 rows=1 width=8)
             Workers Planned: 2
             ->  Partial Aggregate  (cost=13628.33..13628.34 rows=1 width=8)
                   ->  Parallel Seq Scan on tbase_1  (cost=0.00..12586.67 rows=416667 width=0)
(6 rows)
```

上面第一个查询没走并行，analyze后走并行才是正确的，建议大数据量更新再执行analyze。

检查各个节点的执行计划是否一致

```
./tbase_run_sql_dn_master.sh "explain select * from tbase_2 where f2=1"
dn006 --- psql -h 172.16.0.13 -p 11227 -d postgres -U tbase -c "explain select * from tbase_2 where f2
=1"
                              QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tbase_2  (cost=2.18..7.70 rows=4 width=40)
   Recheck Cond: (f2 = 1)
   ->  Bitmap Index Scan on tbase_2_f2_idx  (cost=0.00..2.18 rows=4 width=0)
         Index Cond: (f2 = 1)
(4 rows)
dn002 --- psql -h 172.16.0.42 -p 11012 -d postgres -U tbase -c "explain select * from tbase_2 where f2
=1"
                               QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using tbase_2_f2_idx on tbase_2  (cost=0.42..4.44 rows=1 width=14)
   Index Cond: (f2 = 1)
(2 rows)
```

这两个dn的执行计划不一致，最大可能可以是数据倾斜或者是执行计划给禁用。

如果有可能的话，可以配置在系统空闲时执行全库analyze和vacuum。

优化实例
count(distinct xx)优化

```
postgres=# CREATE TABLE t1(f1 serial not null unique,f2 text,f3 text,f4 text,f5 text,f6 text,f7 text,f8 text,f
9 text,f10 text,f11 text,f12 text) distribute by shard(f1);
postgres=# insert into t1 select t,md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::te
xt),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text) from generate_series(1,1000000) as t;
postgres=# analyze t1;
ANALYZE
postgres=# explain (verbose) select count(distinct f2) from t1;
                              QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=103320.00..103320.01 rows=1 width=8)
   Output: count(DISTINCT f2)
   ->  Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cost=10
0.00..100820.00 rows=1000000 width=33)
         Output: f2
         ->  Seq Scan on public.t1  (cost=0.00..62720.00 rows=1000000 width=33)
               Output: f2
(6 rows)
Time: 0.748 ms
postgres=# select count(distinct f2) from t1;
  count
---------
 1000000
(1 row)

Time: 6274.684 ms (00:06.275)

postgres=# select count(distinct f2) from t1 where f1 <10;
 count
-------
     9
(1 row)
Time: 19.261 ms
```

上面发现count(distinct f2)是发生在cn节点，对于TP类业务，需要操作的数据量少的情况下，性能开销是没有问题的，而且往往比下推执行的性能开销还要小。但如果一次要操作的数据量比较大的ap类业务，则网络传输就会成功瓶颈，下面看看改写后的执行计划。

```
postgres=# explain (verbose) select count(1) from (select f2 from t1 group by f2) as t ;
                                 QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
```

```
 Finalize Aggregate  (cost=355600.70..355600.71 rows=1 width=8)
   Output: count(1)
   ->  Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cost=35
5600.69..355600.70 rows=1 width=0)
         Output: PARTIAL count(1)
         ->  Partial Aggregate  (cost=355500.69..355500.70 rows=1 width=8)
               Output: PARTIAL count(1)
               ->  Group  (cost=340500.69..345500.69 rows=1000000 width=33)
                     Output: t1.f2
                     Group Key: t1.f2
                     ->  Sort  (cost=340500.69..343000.69 rows=1000000 width=0)
                           Output: t1.f2
                           Sort Key: t1.f2
                           ->  Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,d
n10)  (cost=216192.84..226192.84 rows=1000000 width=0)
                                 Output: t1.f2
                                 Distribute results by S: f2
                                 ->  Group  (cost=216092.84..221092.84 rows=1000000 width=33)
                                       Output: t1.f2
                                       Group Key: t1.f2
                                       ->  Sort  (cost=216092.84..218592.84 rows=1000000 width=33)
                                             Output: t1.f2
                                             Sort Key: t1.f2
                                             ->  Seq Scan on public.t1  (cost=0.00..62720.00 rows=1000000 width=33)
                                                   Output: t1.f2
(23 rows)
```

改写后，并行推到dn去执行，现在看看执行的效果。

```
postgres=# select count(1) from (select f2 from t1 group by f2) as t ;
  count
---------
 1000000
(1 row)
Time: 1328.431 ms (00:01.328)
postgres=# select count(1) from (select f2 from t1 where f1<10 group by f2) as t ;
 count
-------
     9
(1 row)
Time: 24.991 ms
postgres=#
```

我们可以看到对于大量数据计算的AP类业务，性能提高了5倍。

增大work_mem减少io访问

```
postgres=# CREATE TABLE t1(f1 serial not null unique,f2 text,f3 text,f4 text,f5 text,f6 text,f7 text,f8 text,f
9 text,f10 text,f11 text,f12 text) distribute by shard(f1);

CREATE TABLE
Time: 70.545 ms

postgres=# CREATE TABLE t2(f1 serial not null unique,f2 text,f3 text,f4 text,f5 text,f6 text,f7 text,f8 text,f
9 text,f10 text,f11 text,f12 text) distribute by shard(f1);
CREATE TABLE
Time: 61.913 ms

postgres=# insert into t1 select t,md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::te
xt),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text) from generate_series(1,1000) as t;
INSERT 0 1000
Time: 48.866 ms

postgres=# insert into t2 select t,md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::te
xt),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text) from generate_series(1,50000) as t;
INSERT 0 50000
Time: 792.858 ms

postgres=# analyze t1;
ANALYZE
Time: 175.946 ms

postgres=# analyze t2;
ANALYZE
Time: 318.802 ms

postgres=# explain  select * from t1 where f2 not in (select f2 from t2);
                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cost=0.00..20
76712.50 rows=500 width=367)
   ->  Seq Scan on t1  (cost=0.00..2076712.50 rows=500 width=367)
         Filter: (NOT (SubPlan 1))
         SubPlan 1
           ->  Materialize  (cost=0.00..4028.00 rows=50000 width=33)
                 ->  Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=0.00..3240.00 rows=50000 width=33)
                       ->  Seq Scan on t2  (cost=0.00..3240.00 rows=50000 width=33)
(7 rows)
Time: 0.916 ms
```

```
postgres=# select * from t1 where f2 not in (select f2 from t2);
 f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 | f9 | f10 | f11 | f12
----+----+----+----+----+----+----+----+----+-----+-----+-----
(0 rows)
Time: 4226.825 ms (00:04.227)
postgres=# set work_mem to '8MB';
SET
Time: 0.289 ms
postgres=# explain  select * from t1 where f2 not in (select f2 from t2);
                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cost=3365.0
0..3577.50 rows=500 width=367)
   ->  Seq Scan on t1  (cost=3365.00..3577.50 rows=500 width=367)
         Filter: (NOT (hashed SubPlan 1))
         SubPlan 1
           ->  Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cos
t=0.00..3240.00 rows=50000 width=33)
                 ->  Seq Scan on t2  (cost=0.00..3240.00 rows=50000 width=33)
(6 rows)
Time: 0.890 ms
postgres=# select * from t1 where f2 not in (select f2 from t2);
 f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 | f9 | f10 | f11 | f12
----+----+----+----+----+----+----+----+----+-----+-----+-----
(0 rows)

Time: 105.249 ms
postgres=#
```

增大work_mem后，性能提高了40倍，因为work_mem足够放下filter的数据，不需要再做 Materialize物化，filter由原来的subplan变成了hash subplan，直接在内存hash表中filter，性能就上去了。

> 注意：
>
> work_mem默认不宜过大，建议在某个具体的查询语句中再根据需要进行调整即可。

not in改写为anti join

增大work_mem减少io访问通过增大计算内存达到提高性能，但内存不可能无限扩大，下面通过改写语句也可以达到提高查询的性能。

```
postgres=#  explain select * from t1 left outer join t2 on t1.f2 = t2.f2 where t2.f2 is null;
                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cost=6405.0
0..9260.75 rows=1 width=734)
   ->  Hash Anti Join  (cost=6405.00..9260.75 rows=1 width=734)
         Hash Cond: (t1.f2 = t2.f2)
         ->  Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cost
=100.00..682.00 rows=1000 width=367)
               Distribute results by S: f2
               ->  Seq Scan on t1  (cost=0.00..210.00 rows=1000 width=367)
         ->  Hash  (cost=21940.00..21940.00 rows=50000 width=367)
               ->  Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (c
ost=100.00..21940.00 rows=50000 width=367)
                     Distribute results by S: f2
                     ->  Seq Scan on t2  (cost=0.00..3240.00 rows=50000 width=367)
(10 rows)

Time: 1.047 ms
postgres=# select * from t1 left outer join t2 on t1.f2 = t2.f2 where t2.f2 is null;
 f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 | f9 | f10 | f11 | f12 | f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 | f9 | f10 | f11 | f12
----+----+----+----+----+----+----+----+----+-----+-----+-----+----+----+----+----+----+----+----+----+----+----+----+----
+----+----+-----+-----+-----
(0 rows)
Time: 107.233 ms
postgres=#
#也可以修改not exists
postgres=# explain select * from t1 where not exists( select 1 from t2 where t1.f2=t2.f2);
                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cost=3865.0
0..4078.75 rows=1 width=367)
   ->  Hash Anti Join  (cost=3865.00..4078.75 rows=1 width=367)
         Hash Cond: (t1.f2 = t2.f2)
         ->  Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cost
=100.00..682.00 rows=1000 width=367)
               Distribute results by S: f2
               ->  Seq Scan on t1  (cost=0.00..210.00 rows=1000 width=367)
         ->  Hash  (cost=5240.00..5240.00 rows=50000 width=33)
               ->  Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (c
```

```
ost=100.00..5240.00 rows=50000 width=33)
                Distribute results by S: f2
                ->  Seq Scan on t2  (cost=0.00..3240.00 rows=50000 width=33)
(10 rows)
Time: 0.974 ms
postgres=# select * from t1 where not exists( select 1 from t2 where t1.f2=t2.f2);
 f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 | f9 | f10 | f11 | f12
----+----+----+----+----+----+----+----+----+-----+-----+-----
(0 rows)
Time: 42.944 ms
postgres=#
```

分布key jon+limit优化

数据准备。

```
postgres=# CREATE TABLE t1(f1 serial not null unique,f2 text,f3 text,f4 text,f5 text,f6 text,f7 text,f8 text,f
9 text,f10 text,f11 text,f12 text) distribute by shard(f1);
CREATE TABLE

postgres=# CREATE TABLE t2(f1 serial not null unique,f2 text,f3 text,f4 text,f5 text,f6 text,f7 text,f8 text,f
9 text,f10 text,f11 text,f12 text) distribute by shard(f1);
CREATE TABLE

postgres=# insert into t1 select t,md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::te
xt),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text) from generate_series(1,1000000) as t;
INSERT 0 1000000

postgres=# insert into t2 select t,md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::te
xt),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text) from generate_series(1,1000000) as t;
INSERT 0 1000000

postgres=# analyze t1;
ANALYZE
postgres=# analyze t2;
ANALYZE
postgres=#
postgres=# \timing
Timing is on.
postgres=# explain  select t1.* from t1,t2 where t1.f1=t2.f1 limit 10;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.25..1.65 rows=10 width=367)
   ->  Merge Join  (cost=0.25..140446.26 rows=1000000 width=367)
         Merge Cond: (t1.f1 = t2.f1)
         ->  Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cost
=100.12..434823.13 rows=1000000 width=367)
               ->  Index Scan using t1_f1_key on t1  (cost=0.12..62723.13 rows=1000000 width=367)
         ->  Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cost
=100.12..71823.13 rows=1000000 width=4)
               ->  Index Only Scan using t2_f1_key on t2  (cost=0.12..62723.13 rows=1000000 width=4)
(7 rows)

Time: 1.372 ms
postgres=# explain analyze  select t1.* from t1,t2 where t1.f1=t2.f1 limit 10;
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
```

```
 Limit  (cost=0.25..1.65 rows=10 width=367) (actual time=2675.437..2948.199 rows=10 loops=1)
   ->  Merge Join  (cost=0.25..140446.26 rows=1000000 width=367) (actual time=2675.431..2675.508 r
ows=10 loops=1)
         Merge Cond: (t1.f1 = t2.f1)
         ->  Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cost
=100.12..434823.13 rows=1000000 width=367) (actual time=1.661..1.704 rows=10 loops=1)
         ->  Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cost
=100.12..71823.13 rows=1000000 width=4) (actual time=2673.761..2673.783 rows=10 loops=1)
 Planning time: 0.358 ms
 Execution time: 2973.948 ms
(7 rows)
Time: 2976.008 ms (00:02.976)
postgres=#
```

看执行计划是在cn上面执行，merge join需要把要join的数据拉回cn再排序，然后再join，这里主切的开销在于网络，优化的话方法就是让语句其推下去计算。

```
postgres=# set prefer_olap to on;
SET
Time: 0.291 ms
postgres=#  explain  select t1.* from t1,t2 where t1.f1=t2.f1 limit 10;
                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 Limit  (cost=100.25..101.70 rows=10 width=367)
   ->  Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cost=10
0.25..101.70 rows=10 width=367)
         ->  Limit  (cost=0.25..1.65 rows=10 width=367)
               ->  Merge Join  (cost=0.25..140446.26 rows=1000000 width=367)
                   Merge Cond: (t1.f1 = t2.f1)
                   ->  Index Scan using t1_f1_key on t1  (cost=0.12..62723.13 rows=1000000 width=367)
                   ->  Index Only Scan using t2_f1_key on t2  (cost=0.12..62723.13 rows=1000000 width=4)
(7 rows)
Time: 1.061 ms
postgres=# explain analyze  select t1.* from t1,t2 where t1.f1=t2.f1 limit 10;
                                     QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 Limit  (cost=100.25..101.70 rows=10 width=367) (actual time=1.527..3.899 rows=10 loops=1)
   ->  Remote Subquery Scan on all (dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cost=10
0.25..101.70 rows=10 width=367) (actual time=1.525..1.529 rows=10 loops=1)
 Planning time: 0.360 ms
 Execution time: 18.193 ms
(4 rows)
Time: 19.921 ms
```

相差150倍的性能，一般情况下，如果需要拉大量的数据回cn计算，则下推执行的效率会更好。

非分布key join使用hash join性能一般最好

为了提高tp类业务查询的性能，我们经常需要对一些字段建立索引，使用有索引字段join时系统往往也会使用Merge Cond和nestloop。

```
mydb=# CREATE TABLE t1(f1 serial not null,f2 text,f3 text,f4 text,f5 text,f6 text,f7 text,f8 text,f9 text,f10 t
ext,f11 text,f12 text) distribute by shard(f1);
CREATE TABLE
Time: 481.042 ms

mydb=# create index t1_f1_idx on t1(f2);
CREATE INDEX
Time: 85.521 ms

mydb=# CREATE TABLE t2(f1 serial not null,f2 text,f3 text,f4 text,f5 text,f6 text,f7 text,f8 text,f9 text,f10 t
ext,f11 text,f12 text) distribute by shard(f1);
CREATE TABLE
Time: 75.973 ms

mydb=# create index t2_f1_idx on t2(f2);
CREATE INDEX
Time: 29.890 ms

mydb=# insert into t1 select t,md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::tex
t),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text) from generate_series(1,1000000) as t;
INSERT 0 1000000
Time: 16450.623 ms (00:16.451)

mydb=# insert into t2 select t,md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::tex
t),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text) from generate_series(1,1000000) as t;
INSERT 0 1000000
Time: 17218.738 ms (00:17.219)
mydb=# analyze t1;
ANALYZE
Time: 2219.341 ms (00:02.219)
mydb=# analyze t2;
ANALYZE
Time: 1649.506 ms (00:01.650)
mydb=#
--merge join
mydb=# explain select t1.* from t1,t2 where t1.f2=t2.f2 limit 10;
                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Limit  (cost=100.25..102.78 rows=10 width=367)
   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.25..102.78 rows=10 width=367)
```

```
      ->  Limit  (cost=0.25..2.73 rows=10 width=367)
            ->  Merge Join  (cost=0.25..248056.80 rows=1000000 width=367)
                  Merge Cond: (t1.f2 = t2.f2)
                  ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.12..487380.85 rows=1000000
width=367)
                        Distribute results by S: f2
                        ->  Index Scan using t1_f1_idx on t1  (cost=0.12..115280.85 rows=1000000 width=36
7)
                  ->  Materialize  (cost=100.12..155875.95 rows=1000000 width=33)
                        ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.12..153375.95 rows=1000
000 width=33)
                              Distribute results by S: f2
                              ->  Index Only Scan using t2_f1_idx on t2  (cost=0.12..115275.95 rows=1000000 w
idth=33)
(12 rows)

Time: 4.183 ms
mydb=# explain analyze select t1.* from t1,t2 where t1.f2=t2.f2 limit 10;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=100.25..102.78 rows=10 width=367) (actual time=6555.346..6556.296 rows=10 loops=1)
   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.25..102.78 rows=10 width=367) (actual ti
me=6555.343..6555.349 rows=10 loops=1)
 Planning time: 0.473 ms
 Execution time: 6569.828 ms
(4 rows)

Time: 6614.439 ms (00:06.614)

--nested loop

mydb=# set enable_mergejoin to off;
SET
Time: 0.422 ms
mydb=# explain select t1.* from t1,t2 where t1.f2=t2.f2 limit 10;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=100.12..103.57 rows=10 width=367)
   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.12..103.57 rows=10 width=367)
         ->  Limit  (cost=0.12..3.52 rows=10 width=367)
               ->  Nested Loop  (cost=0.12..339232.00 rows=1000000 width=367)
                     ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.00..434740.00 rows=1000000
width=367)
                           Distribute results by S: f2
                           ->  Seq Scan on t1  (cost=0.00..62640.00 rows=1000000 width=367)
                     ->  Materialize  (cost=100.12..100.31 rows=1 width=33)
                           ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.12..100.30 rows=1 width=
```

```
33)
                        Distribute results by S: f2
                        ->  Index Only Scan using t2_f1_idx on t2  (cost=0.12..0.27 rows=1 width=33)
                            Index Cond: (f2 = t1.f2)
(12 rows)

Time: 1.033 ms
mydb=# explain analyze select t1.* from t1,t2 where t1.f2=t2.f2 limit 10;
                                    QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Limit  (cost=100.12..103.57 rows=10 width=367) (actual time=5608.326..5609.571 rows=10 loops=1)
   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.12..103.57 rows=10 width=367) (actual ti
me=5608.323..5608.349 rows=10 loops=1)
 Planning time: 0.347 ms
 Execution time: 5669.901 ms
(4 rows)

Time: 5672.584 ms (00:05.673)

mydb=# set enable_nestloop to off;
SET
Time: 0.436 ms
mydb=#  explain select t1.* from t1,t2 where t1.f2=t2.f2 limit 10;
                                    QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Limit  (cost=85983.00..85984.94 rows=10 width=367)
   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=85983.00..85984.94 rows=10 width=367)
       ->  Limit  (cost=85883.00..85884.89 rows=10 width=367)
           ->  Hash Join  (cost=85883.00..274580.00 rows=1000000 width=367)
               Hash Cond: (t1.f2 = t2.f2)
               ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.00..434740.00 rows=1000000
width=367)
                   Distribute results by S: f2
                   ->  Seq Scan on t1  (cost=0.00..62640.00 rows=1000000 width=367)
               ->  Hash  (cost=100740.00..100740.00 rows=1000000 width=33)
                   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.00..100740.00 rows=1000
000 width=33)
                       Distribute results by S: f2
                       ->  Seq Scan on t2  (cost=0.00..62640.00 rows=1000000 width=33)
(12 rows)
Time: 1.141 ms
mydb=#  explain analyze select t1.* from t1,t2 where t1.f2=t2.f2 limit 10;
                                            QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Limit  (cost=85983.00..85984.94 rows=10 width=367) (actual time=1083.691..1085.962 rows=10 loops
=1)
   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=85983.00..85984.94 rows=10 width=367) (act
```

```
ual time=1083.688..1083.699 rows=10 loops=1)
 Planning time: 0.530 ms
 Execution time: 1108.830 ms
(4 rows)
Time: 1117.713 ms (00:01.118)
mydb=#
```

exists的优化

exists在数据量比较大情况下，一般使用的是Semi Join ，在work_mem足够大的情况下走的是hash join，性能会更好。

```
postgres=# show work_mem;
 work_mem
----------
 4MB
(1 row)
Time: 0.298 ms
postgres=# explain select count(1) from t1 where exists(select 1 from t2 where t2.t1_f1=t1.f1);
                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=242218.32..242218.33 rows=1 width=8)
   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=242218.30..242218.32 rows=1 width=0)
         ->  Partial Aggregate  (cost=242118.30..242118.31 rows=1 width=8)
               ->  Hash Semi Join  (cost=110248.00..242118.30 rows=505421 width=0)
                     Hash Cond: (t1.f1 = t2.t1_f1)
                     ->  Seq Scan on t1  (cost=0.00..17420.00 rows=1000000 width=4)
                     ->  Hash  (cost=79340.00..79340.00 rows=3000000 width=4)
                           ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.00..79340.00 rows=30000
00 width=4)
                                 Distribute results by S: t1_f1
                                 ->  Seq Scan on t2  (cost=0.00..52240.00 rows=3000000 width=4)
(10 rows)
Time: 1.091 ms
postgres=# select count(1) from t1 where exists(select 1 from t2 where t2.t1_f1=t1.f1);
 count
--------
 500000
(1 row)
Time: 3779.401 ms (00:03.779)
postgres=# set work_mem to '128MB';
SET
Time: 0.368 ms
postgres=# explain select count(1) from t1 where exists(select 1 from t2 where t2.t1_f1=t1.f1);
                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=101763.76..101763.77 rows=1 width=8)
   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=101763.75..101763.76 rows=1 width=0)
         ->  Partial Aggregate  (cost=101663.75..101663.76 rows=1 width=8)
               ->  Hash Join  (cost=89660.00..101663.75 rows=505421 width=0)
                     Hash Cond: (t2.t1_f1 = t1.f1)
                     ->  Remote Subquery Scan on all (dn001,dn002)  (cost=59840.00..69443.00 rows=505421
```

```
width=4)
                    Distribute results by S: t1_f1
                    ->  HashAggregate  (cost=59740.00..64794.21 rows=505421 width=4)
                        Group Key: t2.t1_f1
                        ->  Seq Scan on t2  (cost=0.00..52240.00 rows=3000000 width=4)
                ->  Hash  (cost=17420.00..17420.00 rows=1000000 width=4)
                    ->  Seq Scan on t1  (cost=0.00..17420.00 rows=1000000 width=4)
(12 rows)
Time: 4.739 ms
postgres=# select count(1) from t1 where exists(select 1 from t2 where t2.t1_f1=t1.f1);
 count
--------
 500000
(1 row)
Time: 1942.037 ms (00:01.942)
postgres=#
```

大约有一倍性能的提升。

重新聚簇表

重新聚簇表可以减少扫描走某个索引值扫描表的页数。

```
psql -h 172.16.0.61 -p 11002 -d postgres -U tbase -c "explain (analyze,buffers) select count(1) from t1 where f2=1;"
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=7208.99..7209.00 rows=1 width=8) (actual time=25.505..25.505 rows=1 loops=1)
   Buffers: shared hit=5869
   ->  Bitmap Heap Scan on t1  (cost=124.87..7185.62 rows=9348 width=0) (actual time=3.567..23.002 rows=10051 loops=1)
         Recheck Cond: (f2 = 1)
         Heap Blocks: exact=5838
         Buffers: shared hit=5869
         ->  Bitmap Index Scan on t1_f2_idx  (cost=0.00..122.53 rows=9348 width=0) (actual time=2.405..2.405 rows=10051 loops=1)
               Index Cond: (f2 = 1)
               Buffers: shared hit=31
 Planning time: 0.626 ms
 Execution time: 25.659 ms
(11 rows)
postgres=# CLUSTER t1 USING t1_f2_idx;
CLUSTER
postgres=#
dn001 --- psql -h 172.16.0.61 -p 11002 -d postgres -U tbase -c "explain (analyze,buffers) select count(1) from t1 where f2=1;"
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=7201.23..7201.24 rows=1 width=8) (actual time=9.808..9.808 rows=1 loops=1)
   Buffers: shared hit=116
   ->  Bitmap Heap Scan on t1  (cost=124.87..7177.86 rows=9348 width=0) (actual time=1.312..7.348 rows=10051 loops=1)
         Recheck Cond: (f2 = 1)
         Heap Blocks: exact=85
         Buffers: shared hit=116
         ->  Bitmap Index Scan on t1_f2_idx  (cost=0.00..122.53 rows=9348 width=0) (actual time=1.219..1.219 rows=10051 loops=1)
               Index Cond: (f2 = 1)
               Buffers: shared hit=31
 Planning time: 0.696 ms
 Execution time: 9.969 ms
(11 rows)
```

聚簇前要扫描的block数为5869，聚簇后为116，性能提高2倍。

jdbc访问数据库优化

- 正确的数据类型绑定。

```
import java.sql.*;
import java.text.SimpleDateFormat;
public class select_normal {
   public static void main(String args[]) {
   Connection c = null;
   Statement stmt = null;
   try {
      Class.forName("org.postgresql.Driver");
      c = DriverManager.getConnection("jdbc:postgresql://172.16.0.30:11345/postgres","tbase", "tbas
e");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");
      long start_time = System.currentTimeMillis();
      PreparedStatement preparedStatement = null;
      try {
         String sql = "select * from public.select_normal where f2 >= ? and f2 < ? " ;
         preparedStatement = c.prepareStatement(sql);
         String deskdate_start = "2020-01-01";
         String deskdate_end = "2020-02-01";
         Date deskdate_start_date = new java.sql.Date(new SimpleDateFormat("yyyy-MM-dd").parse(de
skdate_start).getTime());
         Date deskdate_end_date = new java.sql.Date(new SimpleDateFormat("yyyy-MM-dd").parse(des
kdate_end).getTime());
         preparedStatement.setDate(1,deskdate_start_date);
         preparedStatement.setDate(2,deskdate_end_date);
         ResultSet resultSet = preparedStatement.executeQuery();
         System.out.println("执行查询耗时："+(System.currentTimeMillis()-start_time)+"ms");
         while(resultSet.next()){
         System.out.println(resultSet.getString(1));
         System.out.println(resultSet.getString(2));
         System.out.println(resultSet.getString(3));
      }
      } catch (SQLException e) {
         e.printStackTrace();
      }
   } catch (Exception e) {
      System.err.println( e.getClass().getName()+": "+ e.getMessage() );
      System.exit(0);
   }
      System.out.println("Records created successfully");
   }
```

```
}
```

- 错误的数据类型绑定。

```
import java.sql.*;
import java.text.SimpleDateFormat;
public class select_normal {
   public static void main(String args[]) {
   Connection c = null;
   Statement stmt = null;
   try {
      Class.forName("org.postgresql.Driver");
      c = DriverManager.getConnection("jdbc:postgresql://172.16.0.30:11345/postgres","tbase", "tbas
e");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");
      long start_time = System.currentTimeMillis();
      PreparedStatement preparedStatement = null;
      try {
         String sql = "select * from public.select_normal where f2 >= ? and f2 < ? " ;
         preparedStatement = c.prepareStatement(sql);
         String deskdate_start = "2020-01-01";
         String deskdate_end = "2020-02-01";
         preparedStatement.setDate(1,deskdate_start);
         preparedStatement.setDate(2,deskdate_end);
         ResultSet resultSet = preparedStatement.executeQuery();
         System.out.println("执行查询耗时：" +(System.currentTimeMillis()-start_time)+"ms");
         while(resultSet.next()){
         System.out.println(resultSet.getString(1));
         System.out.println(resultSet.getString(2));
         System.out.println(resultSet.getString(3));
      }
      } catch (SQLException e) {
         e.printStackTrace();
      }
   } catch (Exception e) {
      System.err.println( e.getClass().getName()+": "+ e.getMessage() );
      System.exit(0);
   }
      System.out.println("Records created successfully");
   }
}
```

日期类型如果使用String类型绑定变量，则相当

```
select * from select_normal from f2::text>'2020-01-10'::text
```

这样导致查询无法走索引。

分区表now()剪枝问题

```
postgres=# create table t_time_range
(f1 bigint, f2 timestamp ,f3 bigint)
partition by range (f2) begin (timestamp without time zone '2021-06-01 0:0:0')
step (interval '1 month')
partitions (12) distribute by shard(f1)
to group default_group;
CREATE TABLE
```

- 使用原生的now()函数是剪不了枝。

```
postgres=# explain select * from t_time_range where f2<now();
                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002, dn003
   ->  Append  (cost=0.00..0.00 rows=0 width=0)
         ->  Seq Scan on t_time_range (partition sequence: 0, name: t_time_range_part_0)  (cost=0.00..2.1
7 rows=30 width=24)
               Filter: (f2 < now())
         ->  Seq Scan on t_time_range (partition sequence: 1, name: t_time_range_part_1)  (cost=0.00..2.1
7 rows=30 width=24)
               Filter: (f2 < now())
         ->  Seq Scan on t_time_range (partition sequence: 2, name: t_time_range_part_2)  (cost=0.00..2.1
7 rows=30 width=24)
               Filter: (f2 < now())
         ->  Seq Scan on t_time_range (partition sequence: 3, name: t_time_range_part_3)  (cost=0.00..2.1
7 rows=30 width=24)
               Filter: (f2 < now())
         ->  Seq Scan on t_time_range (partition sequence: 4, name: t_time_range_part_4)  (cost=0.00..2.1
7 rows=30 width=24)
               Filter: (f2 < now())
         ->  Seq Scan on t_time_range (partition sequence: 5, name: t_time_range_part_5)  (cost=0.00..2.1
7 rows=30 width=24)
               Filter: (f2 < now())
         ->  Seq Scan on t_time_range (partition sequence: 6, name: t_time_range_part_6)  (cost=0.00..2.1
7 rows=30 width=24)
               Filter: (f2 < now())
         ->  Seq Scan on t_time_range (partition sequence: 7, name: t_time_range_part_7)  (cost=0.00..2.1
7 rows=30 width=24)
               Filter: (f2 < now())
         ->  Seq Scan on t_time_range (partition sequence: 8, name: t_time_range_part_8)  (cost=0.00..2.1
7 rows=30 width=24)
```

```
            Filter: (f2 < now())
      ->  Seq Scan on t_time_range (partition sequence: 9, name: t_time_range_part_9)  (cost=0.00..2.1
7 rows=30 width=24)
            Filter: (f2 < now())
      ->  Seq Scan on t_time_range (partition sequence: 10, name: t_time_range_part_10)  (cost=0.0
0..2.17 rows=30 width=24)
            Filter: (f2 < now())
      ->  Seq Scan on t_time_range (partition sequence: 11, name: t_time_range_part_11)  (cost=0.0
0..2.17 rows=30 width=24)
            Filter: (f2 < now())
(27 rows)
```

- 定义一个稳定函数。

```
postgres=# create or replace function get_current_timestamp() returns timestamp as
$$
begin
   return current_timestamp;
end;
$$
language plpgsql IMMUTABLE;
CREATE FUNCTION
postgres=# explain select * from t_time_range where f2<get_current_timestamp();
                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002, dn003
   ->  Append  (cost=0.00..0.00 rows=0 width=0)
      ->  Seq Scan on t_time_range (partition sequence: 0, name: t_time_range_part_0)  (cost=0.00..1
1.69 rows=178 width=24)
            Filter: (f2 < '2021-07-12 11:31:33.639119'::timestamp without time zone)
      ->  Seq Scan on t_time_range (partition sequence: 1, name: t_time_range_part_1)  (cost=0.00..1
1.69 rows=178 width=24)
            Filter: (f2 < '2021-07-12 11:31:33.639119'::timestamp without time zone)
(7 rows)
postgres=#
```

这样优化后剪枝就正常了。

使用TDSQL PG自研分区表
插入数据性能测试对比
自研分区表

#创建测试表2048个子表
create table t_time_range
(
  f1 bigint not null, f2 timestamp ,f3 bigint
)
partition by range (f2) begin (timestamp without time zone '2017-09-01 0:0:0')
step (interval '1 month')
partitions (2048) distribute by shard(f1)
to group default_group;

#pgbench sql脚本

[tbase@eg-9-117-182-250 pgbench]$ cat insert_t_time_range.sql
\set f1 random(1, 10000000)
\set f2 random(1, 50000)
insert into t_time_range values(:f1,('2017-09-01'::date+:f2::integer)::timestamp,:f1);

#测试方法

[tbase@eg-9-117-182-250 pgbench]$ pgbench -h 9.117.183.12 -p 15432 -d pgbench -U tbase -c 64 -j 1 -n -M prepared -T 60 -r -f insert_t_time_range.sql > insert_t_time_range.log 2>&1
#测试结果
scaling factor: 1
query mode: prepared
number of clients: 64
number of threads: 1
duration: 60 s
number of transactions actually processed: 1039672
latency average = 3.695 ms
tps = 17319.573296 (including connections establishing)
tps = 17320.178973 (excluding connections establishing)
script statistics:
 - statement latencies in milliseconds:
        0.007 \set f1 random(1, 10000000)
        0.004 \set f2 random(1, 50000)
        3.673 insert into t_time_range values(:f1,('2017-09-01'::date+:f2::integer)::timestamp,:f1);

PG社区分区表

#创建测试表2048个子表
[tbase@TENCENT64 shell]$ psql -d pgbench
psql (PostgreSQL 10.0 TBase V2)
Type "help" for help.
pgbench=#create table t_native_range(f1 bigint,f2 timestamp default now(),f3 integer) partition by ran
ge (f2);
\q
[tbase@TENCENT64 shell]$ psql -At -c "select 'create table t_native_range_'||num::text||' partition of t_
native_range(f1,f2,f3) for values from ('''||startyf||''') to ('''||endyf||'''); ' from (select t::text as num, ('201
7-09-01'::date + (t::text||' months')::interval)::date::text as startyf,('2017-10-01'::date + (t::text||' month
s')::interval)::date::text as endyf from generate_series(0,2047) as t) as t;"|psql -d pgbench
#pgbench sql脚本
[tbase@eg-9-117-182-250 pgbench]$ cat insert_t_native_range.sql
\set f1 random(1, 10000000)
\set f2 random(1, 50000)
insert into t_native_range values(:f1,('2017-09-01'::date+:f2::integer)::timestamp,:f1);
#测试方法
[tbase@eg-9-117-182-250 pgbench]$ pgbench -h 9.117.183.12 -p 15432 -d pgbench -U tbase -c 64 -j
1 -n -M prepared -T 60 -r -f insert_t_native_range.sql > insert_t_native_range.log 2>&1
#测试结果
scaling factor: 1
query mode: prepared
number of clients: 64
number of threads: 1
duration: 60 s
number of transactions actually processed: 781613
latency average = 4.913 ms
tps = 13025.700700 (including connections establishing)
tps = 13026.161416 (excluding connections establishing)
script statistics:
 - statement latencies in milliseconds:
        0.007 \set f1 random(1, 10000000)
        0.004 \set f2 random(1, 50000)
        4.850 insert into t_native_range values(:f1,('2017-09-01'::date+:f2::integer)::timestamp,:f1);

查询数据性能测试对比
自研分区表

```
#创建测试表2048个子表
drop table t_time_range;
create table t_time_range
(
  f1 bigint not null, f2 timestamp ,f3 bigint
)
partition by range (f2) begin (timestamp without time zone '2017-09-01 0:0:0')
step (interval '1 month')
partitions (2048) distribute by shard(f1)
to group default_group;
#每个分区插入一条数据
psql -At  -c "select 'insert into t_time_range values(1,'''||startyf||''',1);' from (select ('2017-09-01'::date +
(t::text||' months')::interval)::date::text as startyf from generate_series(0,2047) as t) as t"|psql -d pgbenc
h
#编写测试存储过程
create or replace function get_t_time_range() returns t_time_range as
$$
declare
    v_start timestamp;
    v_end timestamp;
    v_random integer;
    v_rec t_time_range%rowtype;
begin
    v_random:=(random()*50000)::integer;
    v_start:=(to_char('2017-09-01'::date+v_random,'YYYY-MM')||'-01')::timestamp;
    v_end:=(to_char('2017-09-01'::date+v_random,'YYYY-MM')||'-01')::DATE+'1 months'::interval;
    select * into v_rec from  t_time_range where f2>=v_start and f2<v_end;
    return v_rec;
end;
$$
LANGUAGE plpgsql;
#pgbench sql脚本
[tbase@eg-9-117-182-250 pgbench]$ cat select_t_time_range.sql
select * from get_t_time_range();
#pgbench方法
pgbench -h 9.117.183.12 -p 15432 -d pgbench -U tbase -c 64 -j 1 -n -M prepared -T 60 -r -f select_t_ti
me_range.sql > select_t_time_range.log 2>&1
#测试结果
scaling factor: 1
query mode: prepared
number of clients: 64
```

```
number of threads: 1
duration: 60 s
number of transactions actually processed: 1002843
latency average = 3.829 ms
tps = 16712.957914 (including connections establishing)
tps = 16713.547319 (excluding connections establishing)
script statistics:
 - statement latencies in milliseconds:
         3.813  select * from get_t_time_range();
```

PG社区分区表

```
#创建测试表2048个子表
[tbase@TENCENT64 shell]$ psql -d pgbench
psql (PostgreSQL 10.0 TBase V2)
Type "help" for help.
pgbench=#drop table t_native_range; create table t_native_range(f1 bigint,f2 timestamp default now
(),f3 integer) partition by range ( f2 );
\q
[tbase@TENCENT64 shell]$ psql -At  -c "select 'create table t_native_range_'||num::text||' partition of t_
native_range(f1,f2,f3) for values from ('''||startyf||''') to ('''||endyf||'''); ' from (select t::text as num, ('201
7-09-01'::date + (t::text||' months')::interval)::date::text as startyf,('2017-10-01'::date + (t::text||' month
s')::interval)::date::text as endyf from generate_series(0,2047) as t) as t;"|psql -d pgbench
#每个分区插入一条数据
psql -At  -c "select 'insert into t_native_range values(1,'''||startyf||''',1);' from (select ('2017-09-01'::date
+ (t::text||' months')::interval)::date::text as startyf from generate_series(0,2047) as t) as t"|psql -d pgben
ch
#编写测试存储过程
create or replace function get_t_native_range() returns t_native_range as
$$
declare
    v_start timestamp;
    v_end timestamp;
    v_random integer;
    v_rec t_native_range%rowtype;
begin
    v_random:=(random()*50000)::integer;
    v_start:=(to_char('2017-09-01'::date+v_random,'YYYY-MM')||'-01')::timestamp;
    v_end:=(to_char('2017-09-01'::date+v_random,'YYYY-MM')||'-01')::DATE+'1 months'::interval;
    select * into v_rec from  t_native_range where f2>=v_start and f2<v_end;
    return v_rec;
end;
$$
LANGUAGE plpgsql;
#pgbench sql脚本
[tbase@eg-9-117-182-250 pgbench]$ cat select_t_native_range.sql
select * from get_t_native_range();
#pgbench方法
pgbench -h 9.117.183.12 -p 15432 -d pgbench -U tbase -c 64 -j 1 -n -M prepared -T 60 -r -f select_t_n
ative_range.sql > select_t_native_range.log 2>&1
#测试结果
scaling factor: 1
query mode: prepared
number of clients: 64
number of threads: 1
```

```
duration: 60 s
number of transactions actually processed: 35978
latency average = 106.832 ms
tps = 599.068955 (including connections establishing)
tps = 599.090129 (excluding connections establishing)
script statistics:
 - statement latencies in milliseconds:
         105.706  select * from get_t_native_range();
```

数据库开发规范
分布键设计规范
分布键约束规则

- 分布键字段无法更新，更新分布键需要先删除记录，再插入新的记录。
- 分布键字段类型不能修改。
- 分布键字段的长度不能修改。
- 分布键数据类型只能支持char,varchar,varchar2,text,date,timestamp,int,bigint,float8,number,numberic。
- 分布键只能选择一个字段。

分布键选择规范

- 分布键关系到数据分布是否均衡，原则就是不能由于各个分布键值数据不均产生数据倾斜，出现木桶效应。
- 如果有主键，则选择主键做分布键。
- 如果是复合主键，则可选择数据重复率低的字段来做分布键。
- 没有主键的可以使用JAVA生成UUID来做分布键。
- 如果涉及数据表JOIN,则选择JOIN字段来做分布键。
- 也可以按业务类型，地区或者分公司来做分布键。

分布键对其它约束影响

- 主键必需包含分布键字段。
- 唯一索引必需包含分布键字段。
- 外键必需是分布键字段。

命名规范

1. DB object: database, schema, table, column, view, index, sequence,function, trigger等名称：
    - 建议使用小写字母、数字、下划线的组合。
    - 建议不使用双引号即"包围，除非必须包含大写字母或空格等特殊字符。
    - 长度不能超过63个字符。
    - 不建议以pg_开头或者pgxc_（避免与系统DB object混淆），不建议以数字开头。
    - 禁止使用 SQL 关键字例如 type, order 等。
2. table能包含的column数目,根据字段类型的不同，数目在 250 到 1600 之间；
3. 临时或备份的DB object:table,view 等,建议加上日期, 如dba_ops.b2c_product_summary_2014_07_12(其中dba_ops 为DBA专用schema)；
4. index命名规则为: 普通索引为 表名_列名_idx，唯一索引表名_列名_uidx,如student_name_idx,student_id_uidx。

COLUMN设计

1. 建议能用数值类型的，就不用字符类型。
2. 建议能用varchar(N) 就不用char(N),以利于节省存储空间。
3. 建议能用varchar(N) 就不用text,varchar。
4. 建议使用default NULL,而不用default '',以节省存储空间。
5. 建议如有国际货业务的话，使用timestamp with time zone(timestamptz),而不用timestamp without time zone,避免时间函数在对于不同时区的时间点返回值不同,也为业务国际化扫清障碍。
6. 建议使用NUMERIC(precision, scale)来存储货币金额和其它要求精确计算的数值, 而不建议使用real,double precision。

Constraints 设计

1. 建议每个table都使用shard key做为主键或者唯一索引。
2. 建议建表时一步到位把主键或者唯一过引也一起建立。

> 注意：
>
> 非shard key不可以建立primary key或者 unique index。

Index 设计

1. TDSQL PG提供的index类型: B-tree, Hash, GiST (GeneralizedSearch Tree), SP-GiST (space-partitioned GiST), GIN (Generalized InvertedIndex), BRIN (Block Range Index),目前不建议使用Hash，通常情况下使用B-tree；
2. 建议create 或 drop index 时,加 CONCURRENTLY参数,这是个好习惯，达到与写入数据并发的效果；
3. 建议对于频繁update, delete的包含于index 定义中的column的table, 用create index CONCURRENTLY，dropindex CONCURRENTLY 的方式进行维护其对应index；
4. 建议用unique index 代替unique constraints,便于后续维护；
5. 建议对where 中带多个字段and条件的高频 query，参考数据分布情况，建多个字段的联合index；
6. 建议对固定条件的（一般有特定业务含义）且选择比好（数据占比低）的query，建带where的Partial Indexes；

   select * from test where status=1 and col=?; -- 其中status=1为固定的条件。

   create index on test (col) where status=1;

   7.建议对经常使用表达式作为查询条件的query，可以使用表达式或函数索引加速query；

   select * from test where exp(xxx);

   create index on test (exp(xxx));

   8.建议不要建过多index，一般不要超过6个，核心table（产品，订单）可适当增加index个数。

关于NULL

1. NULL 的判断：IS NULL ，IS NOT NULL。

> 注意：
>
> boolean 类型取值 true，false， NULL；

2. 小心NOT IN 集合中带有NULL元素。

```
postgres=# select * from tbase;
 id |   nickname
----+--------------
  1 | hello TDSQL PG
  2 | TDSQL PG好
  3 | TDSQL PG好
  4 | TDSQL PG default
(4 rows)

postgres=# select * from tbase where id not in (null);
 id | nickname
----+----------
(0 rows)
```

3. 建议对字符串型NULL值处理后，进行 || 操作。

```
postgres=# select id,nickname from tbase limit 1;
 id |  nickname
----+-------------
  1 | hello TDSQL PG
(1 row)

postgres=# select id,nickname||null from tbase limit 1;
 id | ?column?
----+----------
  1 |
(1 row)

postgres=# select id,nickname||coalesce(null,'') from tbase limit 1;
 id |  ?column?
----+-------------
  1 | hello TDSQL PG
(1 row)
```

4. 建议使用count(1) 或count(*) 来统计行数，而不建议使用count(col) 来统计行数，因为NULL值不会计入。

> 注意：
>
> count(多列列名)时，多列列名必须使用括号，例如count((col1,col2,col3)); 注意多列的count，即使所有列都为NULL，该行也被计数，所以效果与count(*) 一致；

```
postgres=# select * from tbase ;
 id |   nickname
----+---------------
  1 | hello TDSQL PG
  2 | TDSQL PG好
  5 |
  3 | TDSQL PG好
  4 | TDSQL PG default
(5 rows)

postgres=# select count(1) from tbase;
 count
-------
     5
(1 row)

postgres=# select count(*) from tbase;
 count
-------
     5
(1 row)

postgres=# select count(nickname) from tbase;
 count
-------
     4
(1 row)

postgres=# select count((id,nickname)) from tbase;
 count
-------
     5
(1 row)
```

5. count(distinct col) 计算某列的非NULL不重复数量，NULL不被计数。

> 注意：
>
> count(distinct (col1,col2,...)) 计算多列的唯一值时，NULL会被计数，同时NULL与NULL会被认为是相同的；

```
postgres=# select count(distinct nickname)from tbase;
 count
-------
     3
(1 row)
postgres=# select count(distinct(id,nickname)) from tbase;
 count
-------
     5
(1 row)
```

6. 两个null的对比方法。

```
postgres=# select null is not  distinct from null as TBasenull;
 TBasenull
-----------
 t
(1 row)
```

开发相关规范

- 建议对DB object 尤其是COLUMN 加COMMENT，便于后续新人了解业务及维护。
  注释前后的数据表可读性对比，有注释的一看就明白。

```
postgres=# \d+ TBase_main
                    Table "public.tbase_main"
 Column |  Type   | Modifiers | Storage  | Stats target | Description
--------+---------+-----------+----------+--------------+-------------
 id     | integer |           | plain    |              |
 mc     | text    |           | extended |              |
Indexes:
    "TBase_main_id_uidx" UNIQUE, btree (id)
Has OIDs: no
Distribute By SHARD(id)
        Location Nodes: ALL DATANODES

postgres=# comment on column TBase_main.id is 'id号';
COMMENT
postgres=# comment on column TBase_main.mc is '产品名称';
COMMENT
postgres=# \d+ TBase_main
                    Table "public.tbase_main"
 Column |  Type   | Modifiers | Storage  | Stats target | Description
--------+---------+-----------+----------+--------------+-------------
 id     | integer |           | plain    |              | id号
 mc     | text    |           | extended |              | 产品名称
Indexes:
    "TBase_main_id_uidx" UNIQUE, btree (id)
Has OIDs: no
Distribute By SHARD(id)
        Location Nodes: ALL DATANODES
```

- 建议非必须时避免select *，只取所需字段，以减少包括不限于网络带宽消耗。

```
postgres=#  explain (verbose) select * from tbase_main where id=1;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using TBase_main_id_uidx on public.tbase_main  (cost=0.15..8.17 rows=1 width=36)
   Output: id, mc
   Index Cond: (TBase_main.id = 1)
(3 rows)
```

```
postgres=#  explain (verbose) select tableoid from tbase_main where id=1;
                                  QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using TBase_main_id_uidx on public.tbase_main  (cost=0.15..8.17 rows=1 width=4)
   Output: tableoid
   Index Cond: (TBase_main.id = 1)
(3 rows)
```

*是返回36个字符，而另一个一条记录只能4个字段的长度。

- 建议update 时尽量做

```
postgres=# update tbase_main set mc='TBase' ;
UPDATE 1
postgres=# select xmin,* from tbase_main;
 xmin | id |  mc
------+----+-------
 2562 |  1 | TBase
(1 row)

postgres=# update tbase_main set mc='TBase' ;
UPDATE 1
postgres=# select xmin,* from tbase_main;
 xmin | id |  mc
------+----+-------
 2564 |  1 | TDSQL PG
(1 row)

postgres=# update tbase_main set mc='TBase' where mc!='TBase';
UPDATE 0
postgres=# select xmin,* from tbase_main;
 xmin | id |  mc
------+----+-------
 2564 |  1 | TDSQL PG
(1 row)
```

上面的效果是一样的，但带条件的更新不会产生一个新的版本记录，不需要系统执行vacuum回收垃圾数据。

- 建议将单个事务的多条SQL操作,分解、拆分，或者不放在一个事务里，让每个事务的粒度尽可能小，尽量lock少的资源，避免lock 、dead lock的产生。

```
postgres=# begin;
BEGIN
postgres=# update tbase_main set mc='TBase_1.3';
UPDATE 200000000
```

```
#sesseion2 等待
postgres=# update tbase_main set mc='TBase_1.4'  where id=1;

#sesseion3 等待
postgres=# update tbase_main set mc='TBase_1.5'  where id=2;
```

如果#sesseion1分布批更新的话，如下所示

```
postgres=# begin;
BEGIN
postgres=# update tbase_main set mc='TBase_1.3' where id>0 and id <=100000;
UPDATE 100000
postgres=#COMMIT;

postgres=# begin;
BEGIN
postgres=# update tbase_main set mc='TBase_1.3' where id>100000 and id <=200000;
UPDATE 100000
postgres=#COMMIT;
```

则session2和session3中就能部分提前完成，这样可以避免大量的锁等待和出现大量的session占用系统资源，在做全表更新时请使用这种方法来执行：

- 建议大批量的数据入库时，使用copy ，不建议使用insert，以提高写入速度；

```
postgres=# insert into tbase_main select t,'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx' from generate_seri
es(1,100000) as t;
INSERT 0 100000
Time: 9511.755 ms

postgres=# copy  TBase_main to '/data/pgxz/TBase_main.txt';
COPY 100002
Time: 179.428 ms

postgres=# copy  TBase_main from  '/data/pgxz/TBase_main.txt';
COPY 100002
Time: 1625.803 ms
postgres=#
```

性能相差5倍。

- 建议对报表类的或生成基础数据的查询，使用物化视图(MATERIALIZEDVIEW)定期固化数据快照，避免对多表（尤其多写频繁的表）重复跑相同的查询，且物化视图支持REFRESH MATERIALIZED VIEW CONCURRENTLY, 支持并发更新；
  如有一个程序需要不断查询TBase_main的总记录数，那么我们这样做：

```
postgres=# select count(1) from tbase_main;
 count
--------
 200004
(1 row)

Time: 27.948 ms

postgres=# create MATERIALIZED VIEW TBase_main_count as select count(1) as num from tbase_main;
SELECT 1
Time: 322.372 ms
postgres=# select num from  TBase_main_count ;
  num
--------
 200004
(1 row)

Time: 0.421 ms
```

性能提高上百倍。

有数据变化时刷新方法 。

```
postgres=#  copy  TBase_main from  '/data/pgxz/TBase_main.txt';
COPY 100002
Time: 1201.774 ms
postgres=# select count(1) from tbase_main;
 count
--------
 300006
(1 row)

Time: 23.164 ms
postgres=# REFRESH MATERIALIZED VIEW TBase_main_count;
REFRESH MATERIALIZED VIEW
Time: 49.486 ms
postgres=# select num from tbase_main_count ;
  num
--------
 300006
(1 row)

Time: 0.301 ms
```

- 建议复杂的统计查询可以尝试窗口函数 。

- 两表join时尽量的使用分布key进行join。
  所以在建立业务的主表，明细表时，就需要使用他们的关联键来做分布键，如下所示：

```
[pgxz@VM_0_29_centos pgxz]$ psql -p 15001
psql (PostgreSQL 10 (TBase 2.01))
Type "help" for help.

postgres=# create table tbase_main(id integer,mc text) distribute by shard(id);
CREATE TABLE
postgres=# create table tbase_detail(id integer,TBase_main_id integer,mc text) distribute by shard(TBas
e_main_id);
CREATE TABLE
postgres=# explain select TBase_detail.* from tbase_main,TBase_detail where TBase_main.id=TBase_de
tail.TBase_main_id;
                              QUERY PLAN
-----------------------------------------------------------------------
 Data Node Scan on "__REMOTE_FQS_QUERY__"  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
(2 rows)

postgres=# explain (verbose) select TBase_detail.* from tbase_main,TBase_detail where TBase_main.id
=TBase_detail.TBase_main_id;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Data Node Scan on "__REMOTE_FQS_QUERY__"  (cost=0.00..0.00 rows=0 width=0)
   Output: TBase_detail.id, TBase_detail.TBase_main_id, TBase_detail.mc
   Node/s: dn001, dn002
   Remote query: SELECT TBase_detail.id, TBase_detail.TBase_main_id, TBase_detail.mc FROM public.tba
se_main, public.tbase_detail WHERE (TBase_main.id = TBase_detail.TBase_main_id)
(4 rows)

postgres=#
```

- 分布键用唯一索引代替主键。

```
postgres=# create unique index TBase_main_id_uidx on TBase_main using btree(id);
CREATE INDEX
```

因为唯一索引后期的维护成本比主键要低很多。

- 分布键无法建立唯一索引则要建立普通索引，提高查询的效率。

```
postgres=# create index TBase_detail_TBase_main_id_idx on TBase_detail using btree(TBase_main_id);
CREATE INDEX
```

这样两表在join查询时返回少量数据时的效率才会高。

- 不要对字段建立外键。
  目前TDSQL PG还不支持多dn外键约束，除非你能确定数据关联键的数据全部落在同一个dn上面。

数据库开发基础
TDSQL-PG的数据类型
数字类型

数字类型

名字	存储尺寸	描述	范围
smallint	2字节	小范围整数	-32768 to +32767
integer	4字节	整数的典型选择	-2147483648 to +2147483647
bigint	8字节	大范围整数	-9223372036854775808 to +9223372036854775807
decimal	可变	用户指定精度，精确	最高小数点前131072位，以及小数点后16383位
numeric	可变	用户指定精度，精确	最高小数点前131072位，以及小数点后16383位
real	4字节	可变精度，不精确	6位十进制精度
double precision	8字节	可变精度，不精确	15位十进制精度
smallserial	2字节	自动增加的小整数	1到32767
serial	4字节	自动增加的整数	1到2147483647
bigserial	8字节	自动增长的大整数	1到9223372036854775807

字符类型

字符类型

名字	描述
character varying(n), varchar(n)	有限制的变长
character(n), char(n)	定长，空格填充
text	1G

二进制数据类型

二进制数据类型

名字	存储尺寸	描述
bytea	1或4字节外加真正的二进制串	变长二进制串

日期类型

日期类型

名字	存储尺寸	描述	最小值	最大值	解析度
timestamp [(p)] [without time zone]	8字节	包括日期和时间（无时区）	4713 BC	294276 AD	1微秒 / 14位
timestamp [(p)] with time zone	8字节	包括日期和时间，有时区	4713 BC	294276 AD	1微秒 / 14位
date	4字节	日期（没有一天中的时间）	4713 BC	5874897 AD	1日
time [(p)] [without time zone]	8字节	一天中的时间（无日期）	0:00:00	24:00:00	1微秒 / 14位
time [(p)] with time zone	12字节	仅仅是一天中的时间，带有时区	00:00:00+1459	24:00:00-1459	1微秒 / 14位
interval [fields] [(p)]	16字节	时间间隔	-178000000年	178000000年	1微秒 / 14位

布尔类型

布尔类型

名字	存储字节	描述
boolean	1字节	状态为真或假

更多的数据类型介绍

请单击 postgresql官网数据类型说明 查看更多的数据类型说明。

消息及异常输出
RAISE NOTICE

```
postgres=# CREATE OR REPLACE FUNCTION f28() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_int INTEGER := 1;
postgres$# BEGIN
postgres$#     RAISE NOTICE 'v_int = %, 随机数 = %',v_int,random();
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f28();
NOTICE:  v_int = 1, 随机数 = 0.236714988015592
 f28
-----
(1 row)
```

使用raise notice 向终端输出一个消息,也有可能写到日志中(需要调整日志的保存级别)。

RAISE EXCEPTION

```
postgres=# CREATE OR REPLACE FUNCTION f28() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_int INTEGER := 1;
postgres$# BEGIN
postgres$#     RAISE EXCEPTION '程序EXCEPTION ';
postgres$#     #下面的语句不会再执行
postgres$#     RAISE NOTICE 'v_int = %, 随机数 = %',v_int,random();
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f28();
ERROR:  程序EXCEPTION
```

如果在事务中执行这个函数,则事务会中止(abort)。

RAISE EXCEPTION 自定义ERRCODE

```
postgres=# CREATE OR REPLACE FUNCTION f28() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_int INTEGER := 1;
postgres$# BEGIN
postgres$#     RAISE EXCEPTION ' 程序EXCEPTION ' USING ERRCODE = '23505';
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f28();
ERROR:  程序EXCEPTION
#日志中会记录这个ERRCODE
2017-10-03 18:40:16.710 CST,"pgxz","postgres",15072,"[local]",59d33b65.3ae0,225,"idle",2017-10-03 1
5:25:25 CST,4/367159,0,LOG,00000,"statement: SELECT f28();",,,,,,,,,"psql"
2017-10-03 18:40:16.710 CST,"pgxz","postgres",15072,"[local]",59d33b65.3ae0,226,"SELECT",2017-10-0
3 15:25:25 CST,4/367159,0,ERROR,23505," 程序EXCEPTION ",,,,,,,,,"psql"
```

控制结构
判断语句
IF...THEN...END IF

```
postgres=# CREATE OR REPLACE FUNCTION f26() RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     IF random()>0.5 THEN
postgres$#         RAISE NOTICE '随机数大于0.5';
postgres$#     END IF;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# select f26();
NOTICE:  随机数大于0.5
 f26
-----

(1 row)
postgres=#
```

IF...THEN...ELSE...END IF

```
postgres=# CREATE OR REPLACE FUNCTION f26() RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     IF random()>0.99 THEN
postgres$#         RAISE NOTICE '随机数大于0.99';
postgres$#     ELSE
postgres$#         RAISE NOTICE '随机数小于或等于0.99';
postgres$#     END IF;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# select f26();
NOTICE:  随机数小于或等于0.99
 f26
-----
(1 row)
postgres=#
```

IF...THEN...ELSIF...THEN...ELSE...END IF

```
postgres=# CREATE OR REPLACE FUNCTION f26() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_float8 float8 := random();
postgres$# BEGIN
postgres$#     IF v_float8>0.99 THEN
postgres$#         RAISE NOTICE '随机数大于0.99';
postgres$#     ELSIF v_float8>0.5 THEN
postgres$#         RAISE NOTICE '随机数大于0.50';
postgres$#     ELSIF v_float8>0.25 THEN
postgres$#         RAISE NOTICE '随机数大于0.25';
postgres$#     ELSE
postgres$#         RAISE NOTICE '随机数小于或等于0.25';
postgres$#     END IF;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f26();
NOTICE:  随机数大于0.50
 f26
-----

(1 row)
```

CASE语句

```
postgres=# CREATE OR REPLACE FUNCTION f26() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_float8 float8 := random();
postgres$# BEGIN
postgres$#     CASE
postgres$#     WHEN v_float8>0.99 THEN
postgres$#         RAISE NOTICE '随机数大于0.99';
postgres$#     WHEN v_float8>0.5 THEN
postgres$#         RAISE NOTICE '随机数大于0.50';
postgres$#     WHEN v_float8>0.25 THEN
postgres$#         RAISE NOTICE '随机数大于0.25';
postgres$#     ELSE
postgres$#         RAISE NOTICE '随机数小于或等于0.25';
postgres$#     END CASE;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f26();
NOTICE:  随机数大于0.50
 f26
-----

(1 row)
```

循环语句

LOOP循环

```
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#      v_id INTEGER := 1;
postgres$# BEGIN
postgres$#     LOOP
postgres$#         RAISE NOTICE '%',v_id;
postgres$#         EXIT WHEN random()>0.8;
postgres$#         v_id := v_id + 1;
postgres$#     END LOOP ;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  1
NOTICE:  2
 f27
-----
(1 row)
#使用EXIT退出循环
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_id INTEGER := 1;
postgres$#     v_random float8 ;
postgres$# BEGIN
postgres$#     LOOP
postgres$#         RAISE NOTICE '%',v_id;
postgres$#         v_id := v_id + 1;
postgres$#         v_random := random();
postgres$#         IF v_random > 0.8 THEN
postgres$#             RETURN;
postgres$#         END IF;
postgres$#     END LOOP ;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  1
```

```
NOTICE:  2
NOTICE:  3
NOTICE:  4
NOTICE:  5
 f27
-----
(1 row)
postgres=#
#使用RETURN退出循环返回
```

WHILE循环

```
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_id INTEGER := 1;
postgres$#     v_random float8 := random() ;
postgres$# BEGIN
postgres$#     WHILE v_random > 0.8 LOOP
postgres$#         RAISE NOTICE '%',v_id;
postgres$#         v_id := v_id + 1;
postgres$#         v_random = random();
postgres$#     END LOOP;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  1
 f27
-----
(1 row)
```

FOR循环

```
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     FOR i IN 1..3 LOOP
postgres$#         RAISE NOTICE 'i = %',i;
postgres$#     END LOOP;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  i = 1
NOTICE:  i = 2
NOTICE:  i = 3
 f27
-----
(1 row)
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     FOR i IN REVERSE 3..1 LOOP
postgres$#         RAISE NOTICE 'i = %',i;
postgres$#     END LOOP;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  i = 3
NOTICE:  i = 2
NOTICE:  i = 1
 f27
-----
(1 row)
#使用REVERSE递减
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     FOR i IN 1..8 BY 2 LOOP
postgres$#         RAISE NOTICE 'i = %',i;
postgres$#     END LOOP;
postgres$# END;
postgres$# $$
```

```
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  i = 1
NOTICE:  i = 3
NOTICE:  i = 5
NOTICE:  i = 7
 f27
-----
(1 row)
#使用BY设置步长
```

FOR循环查询结果

```
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_rec RECORD;
postgres$# BEGIN
postgres$#     FOR v_rec IN SELECT * FROM public.t LOOP
postgres$#         RAISE NOTICE '%',v_rec;
postgres$#     END LOOP;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  (1,TBase)
NOTICE:  (2,pgxz)
 f27
-----
(1 row)
```

FOREACH循环一个数组

```
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_random_arr float8[]:=ARRAY[random(),random()];
postgres$#     v_random float8;
postgres$# BEGIN
postgres$#     FOREACH v_random IN ARRAY v_random_arr LOOP
postgres$#         RAISE NOTICE '%',v_random ;
postgres$#     END LOOP;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  0.452758576720953
NOTICE:  0.975814974401146
 f27
-----
(1 row)
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_random_arr float8[][]:=ARRAY[ARRAY[random(),random()],ARRAY[random(),random()]];
postgres$#     v_random float8;
postgres$# BEGIN
postgres$#     FOREACH v_random SLICE 0 IN ARRAY v_random_arr LOOP
postgres$#         RAISE NOTICE '%',v_random ;
postgres$#     END LOOP;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  0.0588191924616694
NOTICE:  0.368828620761633
NOTICE:  0.813376842066646
NOTICE:  0.415377039927989
 f27
-----
(1 row)
#循环会通过计算expression得到的数组的个体元素进行迭代
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS VOID AS
postgres-# $$
```

```
postgres$# DECLARE
postgres$#     v_random_arr float8[][]:=ARRAY[ARRAY[random(),random()],ARRAY[random(),random()]];
postgres$#     v_random float8[];
postgres$# BEGIN
postgres$#     FOREACH v_random SLICE 1 IN ARRAY v_random_arr LOOP
postgres$#         RAISE NOTICE '%',v_random ;
postgres$#     END LOOP;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
NOTICE:  {0.578366641886532,0.78098024148494}
NOTICE:  {0.783956411294639,0.450278480071574}
 f27
-----
(1 row)
#通过一个正SLICE值，FOREACH通过数组的切片而不是单一元素迭代
```

其它控制语句
动态执行

```
postgres=# CREATE OR REPLACE FUNCTION f27(a_id INTEGER) RETURNS text AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_sql TEXT;
postgres$#     v_mc TEXT;
postgres$# BEGIN
postgres$#     v_sql := 'SELECT mc FROM t WHERE id='||a_id::TEXT;
postgres$#     EXECUTE v_sql INTO v_mc;
postgres$#     RETURN v_mc;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27(1);
  f27
-------
 TBase
(1 row)
```

动态执行就是拼sql语句,然后使用EXECUTE命令执行。

执行一个没有结果的命令

```
postgres=# CREATE OR REPLACE FUNCTION f27() RETURNS void AS
postgres-# $$
postgres$# BEGIN
postgres$#     perform f27(1);
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27();
 f27
-----
(1 row)
postgres=#
```

获取执行结果

```
postgres=# DROP FUNCTION f27(INTEGER);
DROP FUNCTION
postgres=# CREATE OR REPLACE FUNCTION f27(a_id INTEGER) RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_mc TEXT;
postgres$# BEGIN
postgres$#     SELECT mc INTO v_mc FROM t WHERE id=a_id;
postgres$#     IF FOUND THEN
postgres$#         RAISE NOTICE '查询到记录，值为%',v_mc;
postgres$#     ELSE
postgres$#         RAISE NOTICE '查不到记录' ;
postgres$#     END IF;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27(1);
NOTICE:  查询到记录，值为TBase
 f27
-----
(1 row)
postgres=# SELECT f27(3);
NOTICE:  查不到记录
 f27
-----
(1 row)
```

获取影响行数

```
postgres=# CREATE OR REPLACE FUNCTION f27(a_id INTEGER) RETURNS VOID AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_mc TEXT;
postgres$#     v_row_count BIGINT;
postgres$# BEGIN
postgres$#     SELECT mc INTO v_mc FROM t WHERE id=a_id;
postgres$#     GET DIAGNOSTICS v_row_count = ROW_COUNT;
postgres$#     RAISE NOTICE '查询到的记录数为 % ',v_row_count;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27(1);
NOTICE:  查询到的记录数为 1
 f27
-----
(1 row)
postgres=# SELECT f27(3);
NOTICE:  查询到的记录数为 0
 f27
-----
(1 row)
```

俘获错误
错误俘获处理

```
postgres=# CREATE TABLE t_exception (id integer not null,nc text);
CREATE TABLE
postgres=# create unique index t_exception_id_uidx on t_exception using btree(id);
CREATE INDEX
postgres=# CREATE OR REPLACE FUNCTION f27(a_id integer,a_nc text) RETURNS TEXT AS
postgres-# $$
postgres$# BEGIN
postgres$#     INSERT INTO t_exception VALUES(a_id,a_nc);
postgres$#     RETURN '';
postgres$#     EXCEPTION WHEN OTHERS THEN
postgres$#     RETURN '执行出错';
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27(1,'TBase');
 f27
-----
(1 row)
postgres=# SELECT f27(1,'TBase');
   f27
----------
 执行出错
(1 row)
```

获取错误相关信息

```
postgres=# CREATE OR REPLACE FUNCTION f27(a_id integer,a_nc text) RETURNS TEXT AS
postgres-# $$
postgres$# DECLARE
postgres$#     v_sqlstate text;
postgres$#     v_context text;
postgres$#     v_message_text text;
postgres$# BEGIN
postgres$#     INSERT INTO t_exception VALUES(a_id,a_nc);
postgres$#     RETURN '';
postgres$#     EXCEPTION WHEN OTHERS THEN
postgres$#     GET STACKED DIAGNOSTICS v_sqlstate = RETURNED_SQLSTATE,
postgres$#                             v_message_text = MESSAGE_TEXT,
postgres$#                             v_context = PG_EXCEPTION_CONTEXT;
postgres$#     RAISE NOTICE '错误代码 : %',v_sqlstate;
postgres$#     RAISE NOTICE '出错信息 : %',v_message_text;
postgres$#     RAISE NOTICE '发生异常语句 : %',v_context;
postgres$#     RETURN '错误代码 : '||v_sqlstate || '\n出错信息 : '||v_message_text|| '发生异常语句 : '||v_cont
ext;
postgres$# END;
postgres$# $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# SELECT f27(1,'TBase');
NOTICE:  错误代码 : 23505
NOTICE:  出错信息 : node:16385, error duplicate key value violates unique constraint "t_exception_id_ui
dx"
NOTICE:  发生异常语句 : SQL statement "INSERT INTO t_exception VALUES(a_id,a_nc)"
PL/pgSQL function f27(integer,text) line 7 at SQL statement
                                                                                                    f27
-------------------------------------------------------------------------------------------------------------------------
 错误代码 : 23505\n出错信息 : node:16385, error duplicate key value violates unique constraint "t_except
ion_id_uidx"发生异常语句 : SQL statement "INSERT INTO t_exception VALUES(a_id,a_nc)"+
 PL/pgSQL function f27(integer,text) line 7 at SQL statement
(1 row)
```

应用程序语法介绍
建立函数语法

```
CREATE [OR REPLACE] FUNCTION [模式名.]函数名 ([参数模式 [参数名] 数据类型 [default 默认值] [,...]]) RET
RUNS [SETOF] 数据类型 AS
[标签]
[DECLARE
    --变量定义]
BEGIN
--注释
/注释/
    --语句执行
END;
[标签]
LANGUAGE PLPGSQL;
```

[OR REPLACE] 更新函数介绍

带OR REPLACE的作用就函数存在时则替换的功能，建立PL/pgsql函数时不带OR REPLACE关键字， 则遇到函数已经存系统则会报错，如下所示：

```
postgres=# select prosrc from pg_proc where proname='f';
         prosrc
-------------------------
                     +
 BEGIN                 +
    RAISE NOTICE 'TBase';+
 END;                  +

(1 行记录)

postgres=# CREATE FUNCTION f() RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     RAISE NOTICE 'Hello ,TBase';
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
ERROR:  function "f" already exists with same argument types
postgres=# CREATE OR REPLACE FUNCTION f() RETURNS VOID AS
postgres-# $$
postgres$# BEGIN
postgres$#     RAISE NOTICE 'Hello ,TBase';
postgres$# END;
postgres$# $$
postgres-# LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=# select prosrc from pg_proc where proname='f';
           prosrc
--------------------------------
                           +
 BEGIN                     +
    RAISE NOTICE 'Hello ,TBase';+
 END;                      +

(1 行记录)

postgres=# SELECT f();
NOTICE:  Hello ,TBase
 f
---
```

(1 行记录)

[模式名.]函数名介绍

建立函数名称，模式名可以指定，也可以不指定，不指存则存放在当前模式下，如上面例子就没有指定模式名，则就存放在当前模式下，如下所示：

```
postgres=# select * from pg_namespace;
     nspname       | nspowner |              nspacl
-------------------+----------+----------------------------------
 pg_toast          |       10 |
 pg_temp_1         |       10 |
 pg_toast_temp_1   |       10 |
 pg_catalog        |       10 | {postgres=UC/postgres,=U/postgres}
 public            |       10 | {postgres=UC/postgres,=UC/postgres}
 information_schema |       10 | {postgres=UC/postgres,=U/postgres}
(6 行记录)

postgres=# show search_path;
  search_path
----------------
 "$user",public
(1 行记录)

postgres=# select pg_namespace.nspname,pg_proc.prosrc from pg_proc,pg_namespace where
pg_proc.pronamespace=pg_namespace.oid and pg_proc.proname='f';
 nspname |            prosrc
---------+-------------------------------
 public  |                              +
         | BEGIN                         +
         |     RAISE NOTICE 'Hello ,TBase';+
         | END;                          +
         |
(1 行记录)
```

因为$user模式不存在，所以存在public模式下

连接到数据库（replace）
shell交互客户端psql
连接到一个数据库

- 使用参数连接。

```
[tbase@VM_0_29_centos root]$ psql -h 172.16.0.29 -p 15432 -U tbase -d postgres
psql (PostgreSQL 10 (TBase 2.01))
Type "help" for help.
postgres=#
```

- 使用conninfo字符串或者一个URI。

```
[tbase@VM_0_29_centos root]$ psql postgresql://&#x74;&#x62;&#97;&#x73;&#x65;&#x40;&#49;&#55;&#x32;&#46;&#x31;&#54;&#46;&#48;&#46;&#x32;&#x39;:15432/postgres
psql (PostgreSQL 10 (TBase 2.01))
Type "help" for help.
postgres=#
```

- 配置证书连接。

```
[tbase@VM_0_29_centos ~]$ psql 'host=172.16.0.29 port=15432 dbname=postgres user=tbase sslmode=verify-ca sslcert=postgresql.crt sslkey=postgresql.key sslrootcert=root.crt'
Password:
psql (PostgreSQL 10 (TBase 2.01))
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.
postgres=> \q
```

- 配置环境变量后快捷连接。

```
[tbase@VM_0_29_centos root]$ PGHOST=127.0.0.1
[tbase@VM_0_29_centos root]$ PGDATABASE=postgres
[tbase@VM_0_29_centos root]$ PGPORT=15432
[tbase@VM_0_29_centos root]$ export PGHOST PGUSER PGDATABASE PGPORT
[tbase@VM_0_29_centos root]$ psql
psql (PostgreSQL 10 (TBase 2.01))
Type "help" for help.
postgres=#
```

也可以配置在用户环境变量中。

```
[tbase@VM_0_29_centos root]$ cat ~/.bashrc
.bashrc
PGUSER=tbase
PGHOST=127.0.0.1
PGDATABASE=postgres
PGPORT=15432
export PGHOST PGUSER PGDATABASE PGPORT
```

建立一个新连接

- 连接到另外一个库（也可以是当前库）。

```
 postgres=# select pg_backend_pid();
 pg_backend_pid
----------------
         2408
(1 row)
 postgres=# \c
You are now connected to database"postgres" as user "tbase".
 postgres=# select pg_backend_pid();
 pg_backend_pid
----------------
         2426
(1 row)
 postgres=# \c template1
You are now connected to database"template1" as user "tbase".
template1=#
```

- 连接到外一台服务。

```
postgres=# \c postgres tbase 172.16.0.4715432
You are now connected to database"postgres" as user "tbase" on host "172.16.0.47"at port "15432".
```

显示和设置该连接当前运行参数

- 显示当前连接的运行参数。

```
 postgres=# SELECT CURRENT_USER;
 current_user
--------------
 tbase
(1 row)

postgres=# show search_path ;
 search_path
----------------
 "$user",public
(1 row)

postgres=# show work_mem ;
 work_mem
----------
 4MB
(1 row)
```

- 设置当前连接的运行参数。

```
postgres=# set search_path ="$user",public,pg_catalog;
SET
postgres=# set work_mem = '8MB';
SET
```

- 打开和关闭显示每个sql语句执行的时间。

```
postgres=# \timing on
Timing is on.
postgres=# select count(1) from tbase;
 count
-------
 10000
(1 row)

Time: 5.139 ms
postgres=# \timing off
Timing is off.
postgres=# select count(1) from tbase;
```

```
 count
-------
 10000
(1 row)
```

- 打开和关闭显示每个快捷操作符实际运行的sql语句。

```
postgres=# \set ECHO_HIDDEN on

postgres=# \dt

********* QUERY **********

SELECT n.nspname as "Schema",


c.relname as "Name",


CASE c.relkind WHEN 'r' THEN 'table' WHEN 'v' THEN 'view' WHEN 'm' THEN 'materialized view' WHEN
'i' THEN 'index' WHEN 'S' THEN 'sequence' WHEN 's' THEN 'special' WHEN 'f' THEN 'foreign table' END
as "Type",


pg_catalog.pg_get_userbyid(c.relowner) as "Owner"

FROM pg_catalog.pg_class c


LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace

WHERE c.relkind IN ('r','')


AND n.nspname <> 'pg_catalog'


AND n.nspname <> 'information_schema'


AND n.nspname !~ '^pg_toast'


AND pg_catalog.pg_table_is_visible(c.oid)

ORDER BY 1,2;
```

****

List of relations

Schema | Name | Type | Owner

--------+--------------+-------+-------

public | t_time_range | table | tbase

public | tbase | table | tbase

(2 rows)

postgres=# \set ECHO_HIDDEN off

postgres=# \dt

List of relations

Schema | Name | Type | Owner

--------+--------------+-------+-------

public | t_time_range | table | tbase

public | tbase | table | tbase

(2 rows)

- 配置输出结果为HTML格式

```
postgres=# \pset format html
Output format is html.
postgres=# \d tbase
<table border="1">
<caption>Table "public.tbase"</caption>
<tr>
<th align="center">Column</th>
<th align="center">Type</th>
<th align="center">Modifiers</th>
</tr>
<tr valign="top">
<td align="left">id</td>
<td align="left">integer</td>
<td align="left">
</tr>
<tr valign="top">
<td align="left">mc</td>
```

<td align="left">text</td>

<td align="left">

</tr>

</table>

恢复为对齐模式

postgres=# \pset format aligned

Output format is aligned.

postgres=# \d tbase

Table "public.tbase"

Column | Type | Modifiers

--------+---------+-----------

id | integer |

mc | text |

- 配置行列显示格式

postgres=# \x on
Expanded display is on.
postgres=# select * from tbase where id=1;
-[RECORD 1]
id | 1
mc | 1
-[RECORD 2]

```
id | 1
mc | 2
-[ RECORD 3 ]
id | 1
mc | 2

postgres=# \x off
Expanded display is off.
postgres=# select * from tbase where id=1;
 id | mc
----+----
  1 | 1
  1 | 2
  1 | 2
(3 rows)
```

- 显示和配置客户端编码

```
postgres=# \encoding
UTF8
#配置客户端编码为SQL_ASCII
postgres=# \encoding sql_ascii
postgres=# \encoding
SQL_ASCII
```

退出连接

```
postgres=# \q
```

psql执行一个sql命令

- 显示标题

```
[tbase@VM_0_29_centos root]$ psql -h 172.16.0.29 -p 15432 -U tbase -d postgres -c "select count(1) fr
om pg_class"
 count
-------
   317
(1 row)
```

- 不显示标题

```
[tbase@VM_0_29_centos root]$ psql -h 172.16.0.29 -p 15432 -U tbase -d postgres -t -c  "select count
(1) from pg_class"
   317
```

psql执行一个sql文件中所有命令

- 在外部执行。

```
[tbase@VM_0_29_centos ~]$ cat /data/tbase/tbase.sql
set search_path = public;
insert into tbase values(1,2);
select count(1) from tbase;

[tbase@VM_0_29_centos ~]$ psql -h 172.16.0.29 -p 15432 -U tbase -d postgres -f /data/tbase/tbase.sq
l
SET
INSERT 0 1
 count
-------
 10001
(1 row)
```

- 在内部执行。

```
[tbase@VM_0_29_centos ~]$ psql -h 172.16.0.29 -p 15432 -U tbase -d postgres
psql (PostgreSQL 10 (TBase 2.01))
Type "help" for help.

postgres=# \i  /data/tbase/tbase.sql
SET
INSERT 0 1
 count
-------
 10002
(1 row)
```

调用编辑器编写sql脚本

```
postgres=# \e
"/tmp/psql.edit.5532.sql" 2L, 35Cwritten
```

编辑完成后保证退出执行。

调用外部命令

```
postgres=# \! cat /data/tbase/tbase.sql
set search_path = public;
insert into tbase values(1,2);
select count(1) from tbase;
postgres=# \! ls
data1  data2  data3  install
postgres=# \! ls -l
total 0
drwxrwxr-x 3 tbase tbase 28 Sep 18 11:05 data1
drwxrwxr-x 3 tbase tbase 28 Sep 18 11:05 data2
drwxrwxr-x 3 tbase tbase 28 Sep 18 11:17 data3
drwxr-xr-x 3 tbase tbase 23 Sep 18 10:55 install
postgres=#
```

将执行的结果保存到文件

```
postgres=# \o /data/tbase/log.txt
postgres=# select * from tbase where id=1;
postgres=# \! cat /data/tbase/log.txt
 id | mc
----+----
  1 | 1
  1 | 2
  1 | 2
(3 rows)

postgres=# \o
postgres=# select * from tbase where id=1;
 id | mc
----+----
  1 | 1
  1 | 2
  1 | 2
(3 rows)
```

改变当前的工作目录

```
postgres=# \! ls
backup  data  install  java  log.txt  pgbench  tbase  tbaseconf  shell  tbase.sql
postgres=# \cd /data
postgres=# \! ls
package  pgsql  tbase  tbaseConfdb
postgres=#
```

插件管理

- 查看当前库加载了那些插件。

```
postgres=# \dx
                                List of installedextensions
      Name        | Version |   Schema   |                   Description
------------------+---------+------------+-----------------------------------------------------------
 pg_stat_statements | 1.1     | public     | track execution statistics of all SQLstatements executed
 plpgsql            | 1.0     | pg_catalog | PL/pgSQL procedurallanguage
(2 rows)
```

- 给当前库加载插件。

```
postgres=# create extension pg_stat_statements;

CREATE EXTENSION
```

- 删除当前库某个插件。

```
postgres=# drop extension pg_stat_statements;

DROP EXTENSION
```

数据库相关操作

- \l显示当前集群中所有数据库。

```
postgres=# \l
                                  List of databases
  Name    | Owner | Encoding |  Collate   |   Ctype    | Access privileges
-----------+-------+----------+------------+------------+-------------------
 postgres | tbase  | UTF8     | en_US.utf8 | en_US.utf8 |
 template0 | tbase  | UTF8    | en_US.utf8 | en_US.utf8 | =c/tbase         +
          |      |          |            |            | tbase=CTc/tbase
 template1 | tbase  | UTF8    | en_US.utf8 | en_US.utf8 | =c/tbase         +
          |      |          |            |            | tbase=CTc/tbase
(3 rows)
```

- \l+显示当前当前集群中所有数据库（包含库大小及注释）。

> 注意：
>
> 如果节点特别多，数据表特别多，使用\l+时统计比较耗时。

```
postgres=# \l+
                                  List of databases
  Name    | Owner | Encoding |  Collate   |   Ctype    | Access privileges | Size  | Tablespace |          Descri
ption
-----------+-------+----------+------------+------------+-------------------+-------+------------
+-----------------------------------
 postgres | tbase  | UTF8     | en_US.utf8 | en_US.utf8 |                   | 17 MB | pg_default |
 template0 | tbase  | UTF8    | en_US.utf8 | en_US.utf8 | =c/tbase          +| 14 MB | pg_default | unmodifia
bleempty database
          |      |          |            |            | tbase=CTc/tbase     |      |          |
 template1 | tbase  | UTF8    | en_US.utf8 | en_US.utf8 | =c/tbase          +| 14 MB | pg_default | defaultte
mplate for new databases
          |      |          |            |            | tbase=CTc/tbase     |      |          |
(3 rows)
```

- 创建一个新库。

```
postgres=# create database mydb;
CREATE DATABASE
postgres=#
```

模式相关操作

- \dn显示当前库所有模式。

```
postgres=# \dn
                  List of schemas
 Name  | Owner
--------+-------
 pgxc   |tbase
 public | tbase
(2 rows)
```

- \dn+显示当前库所有模式（包含注释）。

```
 postgres=# \dn+
                  List of schemas
 Name  | Owner | Access privileges|      Description
--------+-------+------------------+-----------------------
 pgxc   |tbase  |                  |
 public | tbase | tbase=UC/tbase     +| standardpublic schema
      |      | =UC/tbase          |
(2 rows)
```

- 创建一个新模式。

```
postgres=# create schema mysche;
CREATE SCHEMA
```

用户相关操作

- \du显示当前集群中所有数据库用户。

```
postgres=# \du

                             List of roles
 Role name   |                         Attributes                         | Member of
-------------+------------------------------------------------------------+-----------
 audit_admin | No inheritance                                             |{}
 mls_admin   | No inheritance                                             | {}
 tbase       | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 tbase01_admin | Superuser, Create role, CreateDB                         | {}
```

- \du+显示当前集群中所有数据库用户（包含注释）。

```
postgres=# \du+
                             List of roles
 Role name   |                         Attributes                         | Member of |Description
-------------+------------------------------------------------------------+-----------+-------------
 audit_admin | No inheritance                                             |{}         |
 mls_admin   | No inheritance                                             |{}         |
 tbase       | Superuser, Create role, Create DB, Replication, Bypass RLS | {}        |
 tbase01_admin | Superuser, Create role, CreateDB                         | {}        |
```

- 创建一个新的用户。

```
postgres=# create role pgxc with login ;
CREATE ROLE
```

- 配置用户密码。

```
postgres=# \password pgxc
Enter new password:
Enter it again:
```

表相关操作

- 建立数据表。

```
postgres=# create table tbase(id int,mc text) distribute by shard(id);
CREATE TABLE
```

- \d查看表结构，包括使用的触发器。

```
postgres=# \d tbase
                    Table"public.tbase"
 Column | Type   | Modifiers
--------+---------+-----------
 id     |integer |
 mc     |text    |
```

- \d+查看表结构（包含注释），表类型，分布节点。

```
 postgres=# \d+ tbase
                    Table"public.tbase"
 Column |  Type   |Modifiers | Storage  | Stats target |Description
--------+---------+-----------+----------+--------------+-------------
 id     |integer |          | plain    |              |
 mc     |text    |          | extended |              |
Has OIDs: no
Distribute By SHARD(id)
      Location Nodes: ALL DATANODES
```

- \dt查看表列表。

```
postgres=# \dt
                      List of relations
 Schema |    Name     | Type  | Owner
--------+--------------+-------+-------
 public | t_time_range | table | tbase
 public | tbase        | table | tbase
(2 rows)
```

- \dt+查看表列表详细信息，包含表大小和注释。
  这里连接的节点如果是cn的话，表大小为所有dn节点大小之和，否则为只是该节点的表大小。

```
postgres=# \dt+
                    List of relations
 Schema|     Name     | Type | Owner |  Size   |   Description
--------+--------------+-------+-------+---------+-------------------
 public | t_time_range | table | tbase  | 0 bytes | 这是一个日期分区表
 public | tbase        | table | tbase  | 576 kB |
(2 rows)
```

- \dt+显示某个模式下的所有表。

```
postgres=# \dt+ pgxc.*
                    List of relations
 Schema|     Name     | Type | Owner |  Size   |   Description
--------+------------+-------+-------+---------+-------------
 pgxc   |order_main | table | tbase  | 0 bytes |
(1 row)
```

- \dt+表名显示某个表的详细信息。

```
postgres=# \dt+ tbase
                    List of relations
 Schema|     Name     | Type | Owner |  Size   |   Description
--------+-------+-------+-------+--------+-------------
 public | tbase        | table | tbase  | 576 kB |
(1 row)
```

- \dt+通配符列出适配的表。

```
postgres=# \dt+ t*
                    List of relations
 Schema|     Name     | Type | Owner |  Size   |   Description
--------+--------------+-------+-------+---------+-------------------
 public | t_time_range | table | tbase  | 0 bytes | 这是一个日期分区表
 public | tbase        | table | tbase  | 576 kB |
(2 rows)

postgres=# \dt+ t_*
                    List of relations
 Schema|     Name     | Type | Owner |  Size   |   Description
--------+--------------+-------+-------+---------+-------------------
 public | t_time_range | table | tbase  | 0 bytes | 这是一个日期分区表
(1 row)
```

- \det 列出外部表 。

```
postgres=# \det
        List of foreign tables
 Schema |   Table    |     Server
--------+------------+----------------
 public | t_table_csv | exttable_server
(1 row)
```

视图相关操作

- 建立视图。

```
postgres=# create or replace view tbase_view as select * from tbase;
CREATE VIEW
```

- \d查视图结构。

```
postgres=# \d tbase
    Table "public.tbase"
 Column | Type    | Modifiers
--------+---------+-----------
 id     |integer |
 mc     |text    |
```

- \d+查看视图结构（包含注释），包含创建视图的sql语句。

```
postgres=# \d+ tbase_view
              View "public.tbase_view"
 Column | Type    | Modifiers | Storage  | Description
--------+---------+-----------+----------+-------------
 id     |integer |           | plain    |
 mc     |text    |           | extended |
View definition:
 SELECT tbase.id,
   tbase.mc
  FROM tbase;
```

- \dv查看视图列表。

```
postgres=# \dv
                    List of relations
 Schema |      Name        | Type | Owner
--------+------------------+------+-------
 pgxc   |t_time_range_view | view | tbase
 public | tbase_view       | view | tbase
(2 rows)
```

- \dv+查看视图列表详细信息（包含注释）。

```
postgres=# \dv+
                    List of relations
 Schema |   Name    | Type | Owner |  Size  | Description
--------+-----------------+------+-------+---------+-------------
 pgxc   |t_time_range_view | view | tbase  | 0bytes |
 public | tbase_view | view | tbase  | 0 bytes | 我的视图
(2 rows)
```

- \dv+显示某个模式下的所有视图。

```
postgres=# \dv+ pgxc.*
                    List of relations
 Schema |   Name    | Type | Owner |  Size  | Description
--------+-----------------+------+-------+---------+-------------
 pgxc   |t_time_range_view | view | tbase  | 0bytes |
(1 row)
```

- \dv+视图名显示某个视图的详细信息。

```
postgres=# \dv+ tbase_view
                    List of relations
 Schema |   Name    | Type | Owner |  Size  | Description
--------+-----------+------+-------+---------+-------------
 public | tbase_view | view | tbase  | 0 bytes | 我的视图
(1 row)
```

- \dv+通配符列出适配的视图。

```
postgres=# \dv+ t*
                    List of relations
 Schema |   Name    | Type | Owner |  Size  | Description
--------+-----------------+------+-------+---------+-------------
 pgxc   |t_time_range_view | view | tbase  | 0bytes |
 public | tbase_view | view | tbase  | 0 bytes | 我的视图
(2 rows)

postgres=# \dv+ tb*
                    List of relations
 Schema |   Name    | Type | Owner |  Size  | Description
--------+-----------+------+-------+---------+-------------
 public | tbase_view | view | tbase  | 0 bytes | 我的视图
(1 row)
```

物化视图相关操作

- 建立物化视图。

```
postgres=# create MATERIALIZED VIEW tbase_count as select count(1) as num from tbase;
SELECT 1
postgres=# select * from tbase_count;
  num
-------
 10000
(1 row)
```

- \d查物化视图结构。

```
postgres=# \d tbase_count
              Materialized view"public.tbase_count"
 Column | Type  | Modifiers
--------+--------+-----------
 num    |bigint |
```

- \d+查看物化视图结构（包含注释），包含创建物化视图的sql语句。

```
postgres=# \d+ tbase_count
              Materialized view"public.tbase_count"
 Column | Type  | Modifiers | Storage |Stats target | Description
--------+--------+-----------+---------+--------------+-------------
 num    |bigint |           | plain   |              |
View definition:
 SELECT count(1) AS num
  FROM tbase;
```

- \dm查看视图列表。

```
postgres=# \dm
              List of relations
 Schema |   Name     |       Type        | Owner
--------+-------------+-------------------+-------
 pgxc   |tbase_sum   | materialized view | tbase
 public | tbase_count | materialized view |tbase
(2 rows)
```

- \dm+查看物化视图列表详细信息（包含注释），占用空间大小。

```
postgres=# \dm+
                    List of relations
 Schema |   Name     |       Type        | Owner |    Size    |  Description
--------+------------+-------------------+-------+------------+---------------
 pgxc   |tbase_sum | materialized view | tbase  |8192 bytes |
 public | tbase_count | materialized view |tbase  | 8192 bytes | tbase总记录数
(2 rows)
```

- \dm+显示某个模式下的所有物化视图。

```
postgres=# \dm+ pgxc.*
                    List of relations
 Schema |   Name     |       Type        | Owner |    Size    |  Description
--------+-----------+-------------------+-------+------------+-------------
 pgxc   |tbase_sum | materialized view | tbase  |8192 bytes |
(1 row)
```

- \dm+视图名显示某个物化视图的详细信息。

```
postgres=# \dm+ tbase_count
                    List of relations
 Schema |   Name     |       Type        | Owner |    Size    |  Description
--------+------------+-------------------+-------+------------+---------------
 public | tbase_count | materialized view |tbase  | 8192 bytes | tbase总记录数
(1 row)
```

- \dm+通配符列出适配的物化视图。

```
postgres=# \dm t*
                    List of relations
 Schema |   Name     |       Type        | Owner
--------+------------+-------------------+-------
 pgxc   |tbase_sum   | materialized view | tbase
 public | tbase_count | materialized view |tbase
(2 rows)

postgres=# \dm tbase_c*
                    List of relations
 Schema |   Name     |       Type        | Owner
--------+------------+-------------------+-------
 public | tbase_count | materialized view |tbase
(1 row)
```

序列相关操作

- 建立序列。

```
postgres=# create sequence tbase_seq;
CREATE SEQUENCE
postgres=# create sequence pgxc.tbase_seq;
CREATE SEQUENCE
```

- \d查看序列定义和使用情况。

```
postgres=# \d tbase_seq
        Sequence "public.tbase_seq"
  Column   |  Type  | Value
-----------+---------+-------
 last_value | bigint  | 1
 log_cnt    | bigint  | 0
 is_called | boolean | f
```

- \ds查看序列列表。

```
postgres=# \ds
          List of relations
 Schema |  Name    |   Type  | Owner
--------+-----------+----------+-------
 pgxc    |tbase_seq  | sequence | tbase
 public | tbase_seq | sequence | tbase
(2 rows)
```

- \ds+查看序列列表详细信息（包含注释），占用空间大小。

```
postgres=# \ds+
          List of relations
 Schema |  Name    |   Type  | Owner |    Size    | Description
--------+-----------+----------+-------+------------+-------------
 pgxc    |tbase_seq | sequence | tbase  | 8192bytes |
 public | tbase_seq | sequence | tbase  | 8192 bytes | tbase序列
(2 rows)
```

- \ds+显示某个模式下的所有序列。

```
postgres=# \ds+ pgxc.*
        List of relations
 Schema |  Name    |   Type  | Owner |    Size    | Description
--------+----------+----------+-------+------------+-------------
 pgxc   |tbase_seq | sequence | tbase  | 8192bytes |
(1 row)
```

- \ds+序列名显示某个序列的详细信息。

```
postgres=# \ds+ tbase_seq
        List of relations
 Schema |  Name    |   Type  | Owner |    Size    | Description
--------+-----------+----------+-------+------------+-------------
 public | tbase_seq | sequence | tbase  | 8192 bytes | tbase序列
(1 row)
```

- \ds+通配符列出适配的序列。

```
postgres=# \ds *_seq
        List of relations
 Schema |  Name    |   Type  | Owner
--------+-----------+----------+-------
 pgxc   |tbase_seq  | sequence | tbase
 public | tbase_seq | sequence | tbase
(2 rows)

postgres=# \ds t*_seq
        List of relations
 Schema |  Name    |   Type  | Owner
--------+-----------+----------+-------
 public | tbase_seq | sequence | tbase
(1 row)
```

索引相关操作

- 建立索引。

```
postgres=# create unique index tbase_id_uidx on tbase(id);
CREATE INDEX
postgres=# create index tbase_mc_idx on tbase(mc);
CREATE INDEX
postgres=#
```

- \di查看索引列表。

```
postgres=# \di
                        List of relations
 Schema |    Name      | Type | Owner | Table
--------+--------------+-------+-------+-------
 public | tbase_id_uidx | index | tbase  | tbase
 public | tbase_mc_idx  | index | tbase  | tbase
(2 rows)
```

- \di+查看索引列表详细信息（包含注释），占用空间大小--只能在dn上面查看索引大小。

```
postgres=# \di+
                        List of relations
 Schema |    Name      | Type  | Owner | Table |    Size    |  Description
--------+--------------+-------+-------+-------+------------+---------------
 public | tbase_id_uidx | index | tbase  | tbase | 8192 bytes | tbase唯一索引
 public | tbase_mc_idx      | index | tbase  | tbase     | 8192 bytes |
(2 rows)
```

- \di+显示某个模式下的所有索引--只能在dn上面查看索引大小。

```
postgres=# \di+ public.*
                        List of relations
 Schema |    Name      | Type  | Owner | Table |    Size    |  Description
--------+--------------+-------+-------+-------+------------+---------------
 public | tbase_id_uidx | index | tbase  | tbase | 8192 bytes | tbase唯一索引
 public | tbase_mc_idx      | index | tbase  | tbase     | 8192 bytes |
(2 rows)
```

- \di+索引名显示某个索引的详细信息--只能在dn上面查看索引大小。

```
postgres=# \di+ public.tbase_id_uidx
                        List of relations
 Schema |    Name      | Type  | Owner | Table |    Size    |  Description
--------+--------------+-------+-------+-------+------------+--------------
 public | tbase_id_uidx | index | tbase  | tbase | 8192 bytes | tbase唯一索引
(1 row)
```

- \di+通配符列出适配的索引--只能在dn上面查看索引大小。

```
postgres=# \di+ *idx
                        List of relations
 Schema |    Name      | Type  | Owner | Table |    Size    |  Description
--------+------------------+-------+-------+------------+------------+--------------
 pgxc   |order_main_id_idx | index | tbase  |order_main | 8192 bytes |
 public | tbase_id_uidx | index | tbase  | tbase | 8192 bytes | tbase唯一索引
 public | tbase_mc_idx      | index | tbase  | tbase     | 8192 bytes |
(3 rows)

postgres=# \di+ *uidx--#只能在dn上面查看索引大小
                        List of relations
 Schema |    Name      | Type  | Owner | Table |    Size    |  Description
--------+--------------+-------+-------+-------+------------+--------------
 public | tbase_id_uidx | index | tbase  | tbase | 8192 bytes | tbase唯一索引
(1 row)
```

函数相关操作

- 建立函数。

```
postgres=# CREATE OR REPLACE FUNCTION tbase_f1(a_1 text) returns text as
postgres-# $$
postgres$# begin
postgres$#     return a_1;
postgres$# end;
postgres$# $$
postgres-# language plpgsql;
CREATE FUNCTION
postgres=# CREATE OR REPLACE FUNCTION pgxc.tbase_f2(a_2 text) returns text as
postgres-# $$
postgres$# begin
postgres$#     return a_2;
postgres$# end;
postgres$# $$
postgres-# language plpgsql;
CREATE FUNCTION
postgres=#
```

- \df查看函数列表。

```
postgres=# \df
                        List of functions
 Schema |  Name   | Result data type |Argument data types |  Type
--------+---------+------------------+--------------------+--------
 pgxc   |tbase_f2 | text             | a_2 text           | normal
 public | tbase_f1 | text            | a_1 text           | normal
(2 rows)
```

- \df+查看函数列表详细信息（包含注释），定义。

```
postgres=# \x
Expanded display is on.
postgres=# \df+ tbase*
                        List of functions
-[ RECORD 1 ]-------+--------------------
Schema             | pgxc
Name               | tbase_f2
Result data type   | text
Argument data types | a_2 text
```

```
Type               | normal
Volatility         | volatile
Parallel           | unsafe
Owner              | tbase
Security           | invoker
Access privileges  |
Language           | plpgsql
Source code        |                 +
                   |     begin       +
                   |     return a_2;  +
                   |     end;        +
                   |
Description        |
-[ RECORD 2 ]-------+--------------------
Schema             | public
Name               | tbase_f1
Result data type   | text
Argument data types | a_1 text
Type               | normal
Volatility         | volatile
Parallel           | unsafe
Owner              | tbase
Security           | invoker
Access privileges  |
Language           | plpgsql
Source code        |                 +
                   |     begin       +
                   |       return a_1;+
                   |     end;        +
                   |
Description        |
```

- \df+函数名显示某个函数的详细信息。

```
postgres=# \df+ tbase_f1
                        List of functions
-[ RECORD 1 ]-------+--------------------
Schema             | public
Name               | tbase_f1
Result data type   | text
Argument data types | a_1 text
Type               | normal
Volatility         | volatile
Parallel           | unsafe
Owner              | tbase
Security           | invoker
```

```
Access privileges   |
Language            | plpgsql
Source code         |                 +
                    |     begin         +
                    |       return a_1;+
                    |     end;          +
                    |
Description         |
```

- \df+通配符列出适配的函数。

```
postgres=# \df+ tbase*
                                  List of functions
-[ RECORD 1 ]-------+--------------------
Schema              | pgxc
Name                | tbase_f2
Result data type    | text
Argument data types | a_2 text
Type                | normal
Volatility          | volatile
Parallel            | unsafe
Owner               | tbase
Security            | invoker
Access privileges   |
Language            | plpgsql
Source code         |                 +
                    |     begin         +
                    |     return a_2;   +
                    |     end;          +
                    |
Description         |
-[ RECORD 2 ]-------+--------------------
Schema              | public
Name                | tbase_f1
Result data type    | text
Argument data types | a_1 text
Type                | normal
Volatility          | volatile
Parallel            | unsafe
Owner               | tbase
Security            | invoker
Access privileges   |
Language            | plpgsql
Source code         |                 +
                    |     begin         +
                    |       return a_1;+
```

```
                |    end;          +
               |
Description        |

postgres=# \df+ *f1
                          List of functions
-[ RECORD 1 ]-------+--------------------
Schema              | public
Name                | tbase_f1
Result data type    | text
Argument data types | a_1 text
Type                | normal
Volatility          | volatile
Parallel            | unsafe
Owner               | tbase
Security            | invoker
Access privileges   |
Language            | plpgsql
Source code         |                  +
                |    begin         +
                |        return a_1;+
                |    end;          +
               |
Description
```

自定义数据类型相关操作

- 建立数据类型。

```
postgres=# CREATE TYPE bug_status AS ENUM('new', 'open', 'closed');
CREATE TYPE
```

- \dT查看自定义数据类型列表。

```
postgres=# \dT
                          List ofdata types
 Schema   |    Name   |      Description
-----------+-----------+-------------------------
 pg_oracle | nvarchar2  | oracle nvarchar2(length)
 pg_oracle | varchar2   | oracle varchar2(length)
 public   | bug_status |
```

- \dT+查看自定义数据类型列表详细信息（包含enum类型的值）。

```
postgres=# \dT+
                                 List ofdata types
 Schema |   Name    | Internal name | Size |Elements | Access privileges | Description
-----------+-----------+---------------+-------+----------+-------------------+-------------------------
 pg_oracle | nvarchar2  | nvarchar2     | var  |         |                 | oracle nvarchar2(length)
 pg_oracle | varchar2   | varchar2      | var  |         |                | oracle varchar2(length)
 public | bug_status | bug_status    | 4    | new     +|                 |
       |           |               |      | open    +|                 |
       |           |               |      | closed  |                 |
```

- \dT+显示某个模式下的所有自定义类型。

```
postgres=# \dT+ public.*
                                 List ofdata types
 Schema |   Name    | Internal name | Size |Elements | Access privileges | Description
--------+-----------+---------------+-------+----------+-------------------+-------------
 public | bug_status | bug_status    | 4    | new     +|                 |
       |           |               |      | open    +|                 |
       |           |               |      | closed  |                 |
```

- \dT+自定义数据类型显示某个数据类型的详细信息。

```
postgres=# \dT+ bug_status
                                List ofdata types
 Schema |   Name    | Internal name | Size |Elements | Access privileges | Description
--------+-----------+---------------+------+---------+-------------------+-------------
 public | bug_status | bug_status    | 4    | new     +|                  |
        |           |               |      | open    +|                  |
        |           |               |      | closed  |                   |
(1 row)
```

- \dT+通配符列出适配的数据类型。

```
postgres=# \dT+ bug_*
                                List ofdata types
 Schema |   Name    | Internal name | Size |Elements | Owner | Access privileges | Description
--------+-----------+---------------+------+---------+-------+-------------------+-------------
 public | bug_status | bug_status    | 4    | new     +| tbase  |                  |
        |           |               |      | open    +|       |                  |
        |           |               |      | closed  |       |                  |
```

存储过程语句相关操作

- 加载某个存储过程语言。

```
postgres=# create language plpgsql;
CREATE LANGUAGE
```

- \dL查看存储过程语言列表。

```
postgres=# \dL
                List of languages
 Name   | Owner | Trusted |        Description
---------+-------+---------+----------------------------
 plpgsql | tbase  | t      | PL/pgSQL procedural language
(1 row)
```

列出表、视图和序列和它们相关的访问权限

- 显示所有对象的访问权限。

```
postgres=# \dp
                            Accessprivileges
 Schema |       Name        |   Type  | Access privileges | Column privileges | Policies
--------+-------------------+---------+-------------------+-------------------+----------
 public | pg_stat_statements | view    | tbase=arwdDxt/tbase+|                   |
        |                   |         | =r/tbase          |                   |
 public | tbase             | table   | tbase=arwdDxt/tbase+|                   |
        |                   |         | pgxc=r/tbase      |                   |
 public | tbase_seq         | sequence | tbase=rwU/tbase    +|                   |
        |                   |         | pgxc=rwU/tbase    |                   |
(3 rows)
```

- \dp+某个对象名，只显示匹配的对象。

```
postgres=# \dp tbase
                            Accessprivileges
 Schema |       Name        |   Type  | Access privileges | Column privileges | Policies
--------+-------+-------+-------------------+-------------------+----------
 public | tbase             | table   | tbase=arwdDxt/tbase+|                   |
        |                   |         | pgxc=r/tbase      |                   |
(1 row)

postgres=# \dp public.*
                            Accessprivileges
 Schema |       Name        |   Type  | Access privileges | Column privileges | Policies
--------+-------------------+---------+-------------------+-------------------+----------
 public | pg_stat_statements | view    | tbase=arwdDxt/tbase+|                   |
        |                   |         | =r/tbase          |                   |
 public | tbase             | table   | tbase=arwdDxt/tbase+|                   |
        |                   |         | pgxc=r/tbase      |                   |
 public | tbase_seq         | sequence | tbase=rwU/tbase    +|                   |
        |                   |         | pgxc=rwU/tbase    |                   |
(3 rows)
```

列出库或用户定义的配置

```
postgres=# \drds
                List of settings
 Role | Database |             Settings
------+----------+----------------------------------
 pgxc |          | log_statement=none
      | postgres | search_path="$user", public, pgxc
(2 rows)
```

copy命令的使用

- \copy to 将数据复制到本地文件中。

```
postgres=# \copy tbase to '/data/tbase/tbase.txt';
postgres=# \! cat /data/tbase/tbase.txt
1	tbase
```

- \copy from 将本地文件复制到数据表中。

```
postgres=# \copy tbase from '/data/tbase/tbase.txt';
```

copy FROM stdin使用方法

```
postgres=# COPY tbase (id, mc) FROM stdin;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1  tbase
>> 2  \N
>> 3  pgxc
>> \.
postgres=# select * from tbase;
 id |  mc
----+-------
  1 | tbase
  2 |
  3 | pgxc
(3 rows)
```

注：1 和tbase 之间不是空格，而是tab符

打印当前查询缓冲区到标准输出

```
postgres=# select * from tbase;
 id |  mc
----+-------
  1 | tbase
(1 row)

postgres=# \p
select * from tbase;
```

自定义显示格式

- 配置null的显示代替字符串。

```
postgres=# \pset null '(null)'
Null display is "(null)".
postgres=# insert into tbase values(2,null);
INSERT 0 1
postgres=# select * from tbase;
 id |   mc
----+--------
  1 | tbase
  2 | (null)
(2 rows)
```

- 不显示边框。

```
postgres=# \pset border 0
Border style is 0.
postgres=# select * from tbase;
id  mc
-- ------
 1tbase
 2(null)
(2 rows)
```

- 只显示记录。

```
postgres=# \pset tuples_only
Showing only tuples.
postgres=# select * from tbase;
 1tbase
 2(null)
```

- 列之间使用逗号分隔。

```
postgres=# \pset format unaligned
Output format is unaligned.
postgres=# \pset fieldsep ,
Field separator is ",".
postgres=# select * from tbase;
1,tbase
```

2,(null)

显示psql内部操作

```
postgres=# \set ECHO_HIDDEN on
postgres=# \dt+ t
********* QUERY **********
SELECT n.nspname as "Schema",
  c.relname as "Name",
  CASE c.relkind WHEN 'r' THEN 'table' WHEN 'v' THEN 'view' WHEN 'm' THEN 'materialized view' WHE
N 'i' THEN 'index' WHEN 'S' THEN 'sequence' WHEN 's' THEN 'special' WHEN 'f' THEN 'foreign table' W
HEN 'p' THEN 'table' END as "Type",
  pg_catalog.pg_get_userbyid(c.relowner) as "Owner",
  pg_catalog.pg_size_pretty(pg_catalog.pg_table_size(c.oid)) as "Size",
  pg_catalog.pg_size_pretty(pg_catalog.pg_allocated_table_size(c.oid)) as "Allocated Size",
  pg_catalog.obj_description(c.oid, 'pg_class') as "Description"
FROM pg_catalog.pg_class c
     LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkind IN ('r','p','s','')
      AND n.nspname !~ '^pg_toast'
  AND c.relname ~ '^(t)$'
  AND pg_catalog.pg_table_is_visible(c.oid)
ORDER BY 1,2;
**************************

                        List of relations
 Schema | Name | Type  | Owner | Size  | Allocated Size | Description
--------+------+-------+-------+-------+----------------+-------------
 public | t    | table | tbase | 16 kB | 0 bytes        |
(1 row)

#禁用显示psql内部操作

postgres=# \set ECHO_HIDDEN off
```

重复执行上一条语句

```
postgres=# select count(1) from pg_stat_activity where state!='idle';
 count
-------
     1
(1 row)

postgres=# \watch 1
Fri 28 Sep 2018 04:58:51 PM CST (every 1s)

 count
-------
     1
(1 row)

Fri 28 Sep 2018 04:58:52 PM CST (every 1s)

 count
-------
     1
(1 row)
```

上面的语句为每次自动查询活跃的进程数，相当于临时监控作用。

sql命令帮助查看

```
postgres=# \h
Available help:
  ABORT                      ALTER USER                CREATE ROLE               DROP INDEX
LOCK
  ALTER AGGREGATE            ALTER USER MAPPING        CREATE RULE               DROP LANG
UAGE                MOVE
  ALTER COLLATION            ALTER VIEW                CREATE SCHEMA             DROP MATERIA
LIZED VIEW          NOTIFY
。 。 。

postgres=# \h begin;
Command:     BEGIN
Description: start a transaction block
Syntax:
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]

where transaction_mode is one of:

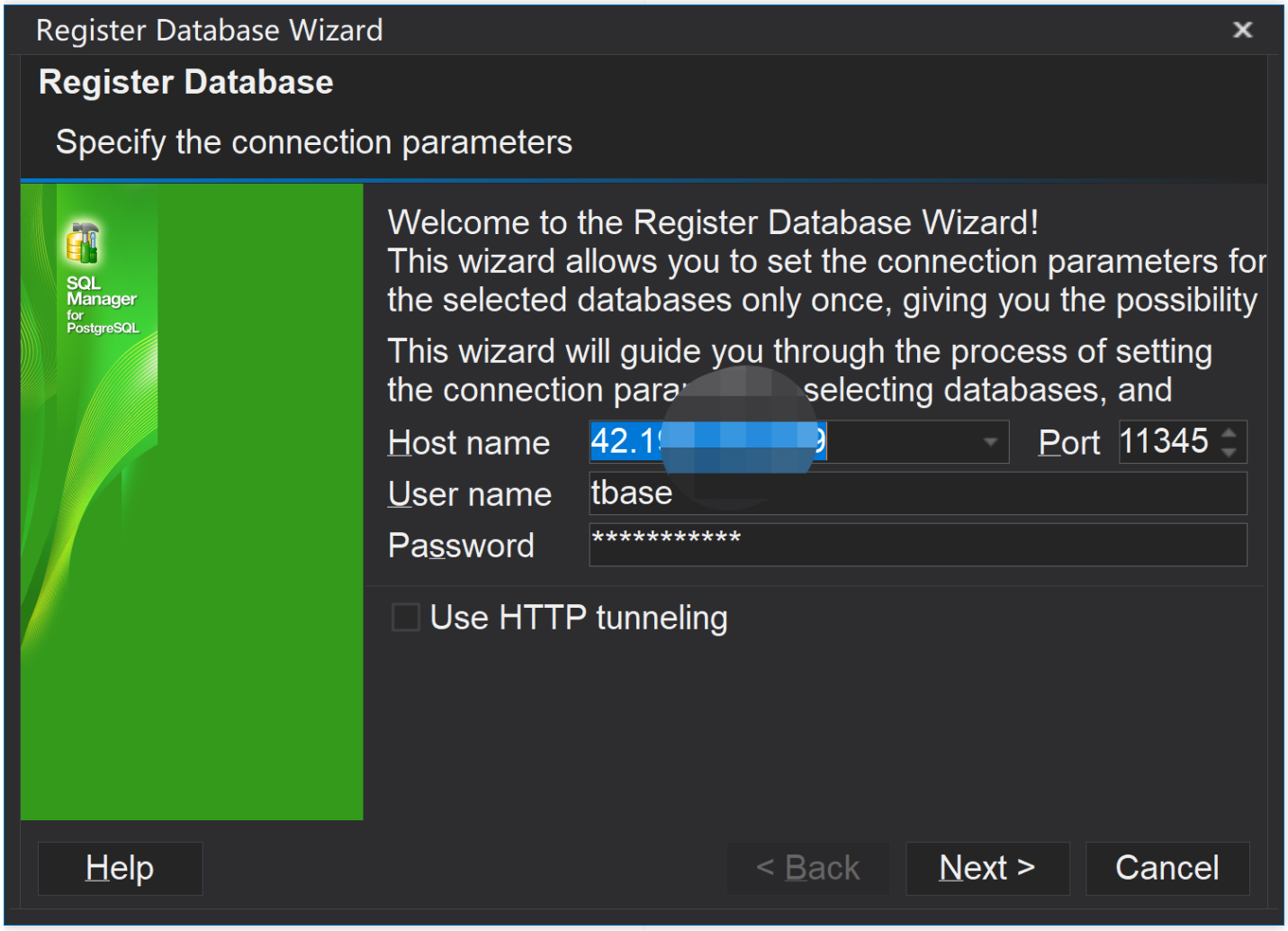
    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE

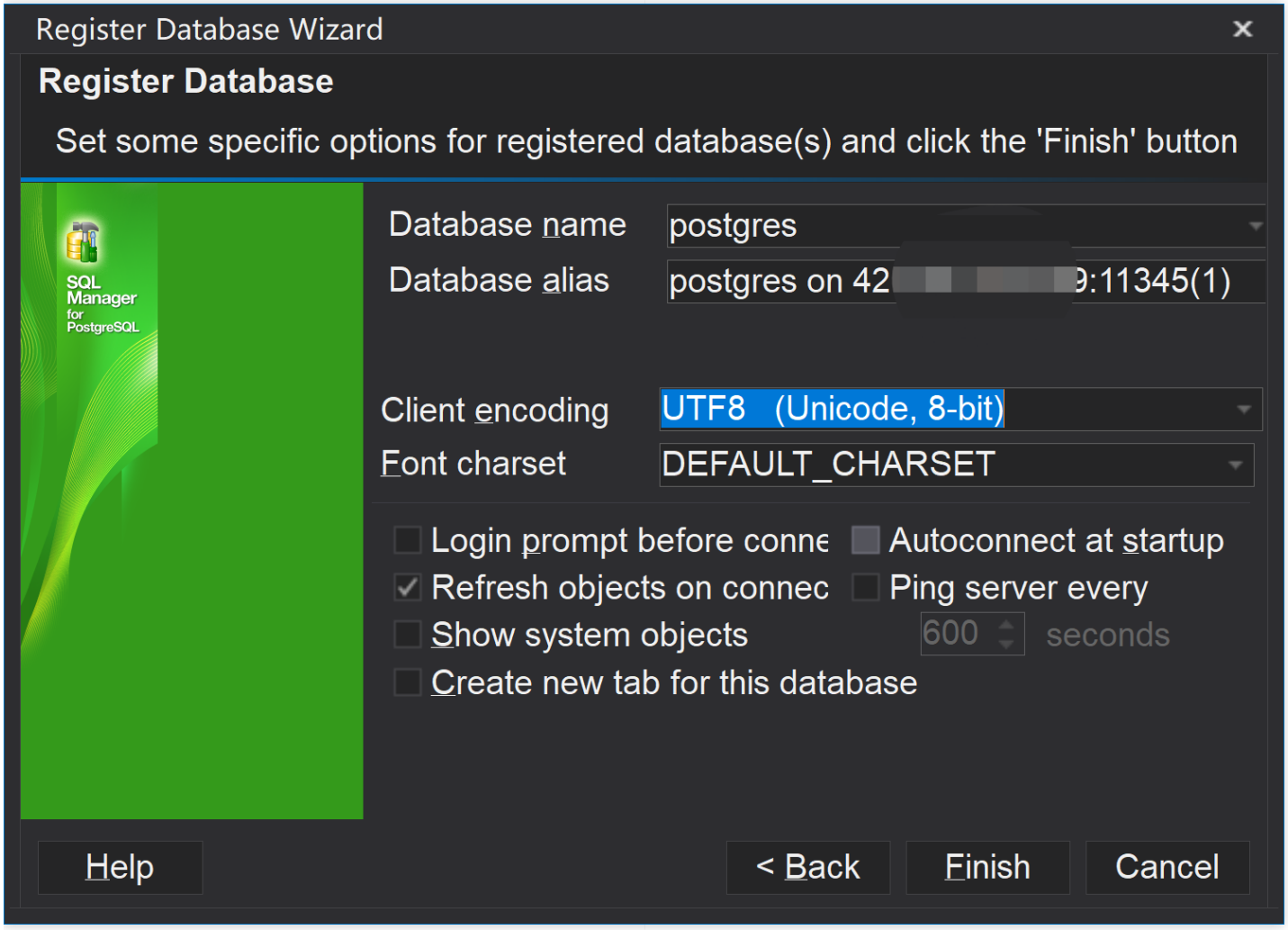
postgres=#
```

ems postgresql manager
ems postgresql manage介绍

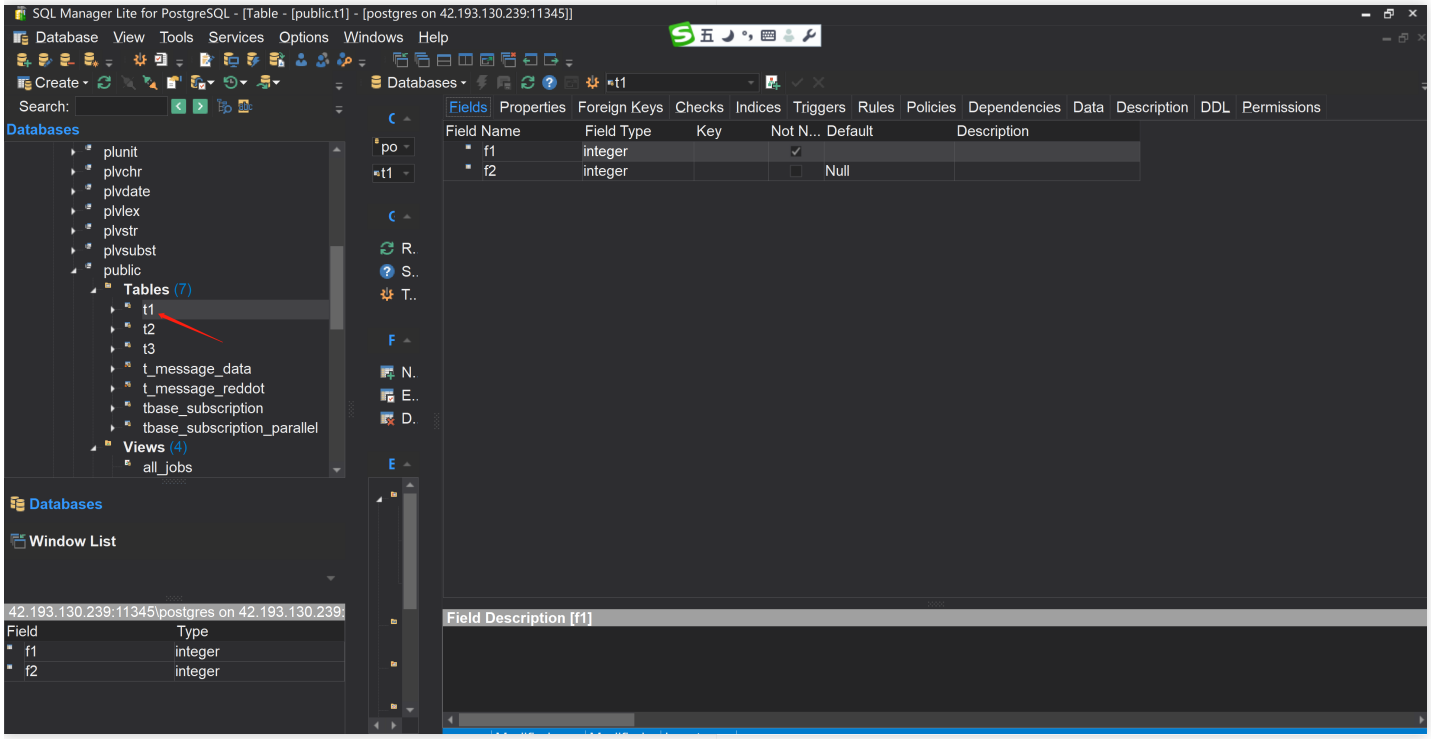
这是一款商业的postgresql的客户端管理工具，可以用来创建对象，执行sql，ems postgresql manage分为收费版本和免费的lite版本，使用上lite版本基本足够使用。

连接到tbase服务


Register Database Wizard
Register Database
Specify the connection parameters
SQL Manager for PostgreSQL
Welcome to the Register Database Wizard!
This wizard allows you to set the connection parameters for the selected databases only once, giving you the possibility
This wizard will guide you through the process of setting the connection para selecting databases, and
Host name
Port 11345
User name tbase
Password **********
Use HTTP tunneling
Help
< Back
Next >
Cancel

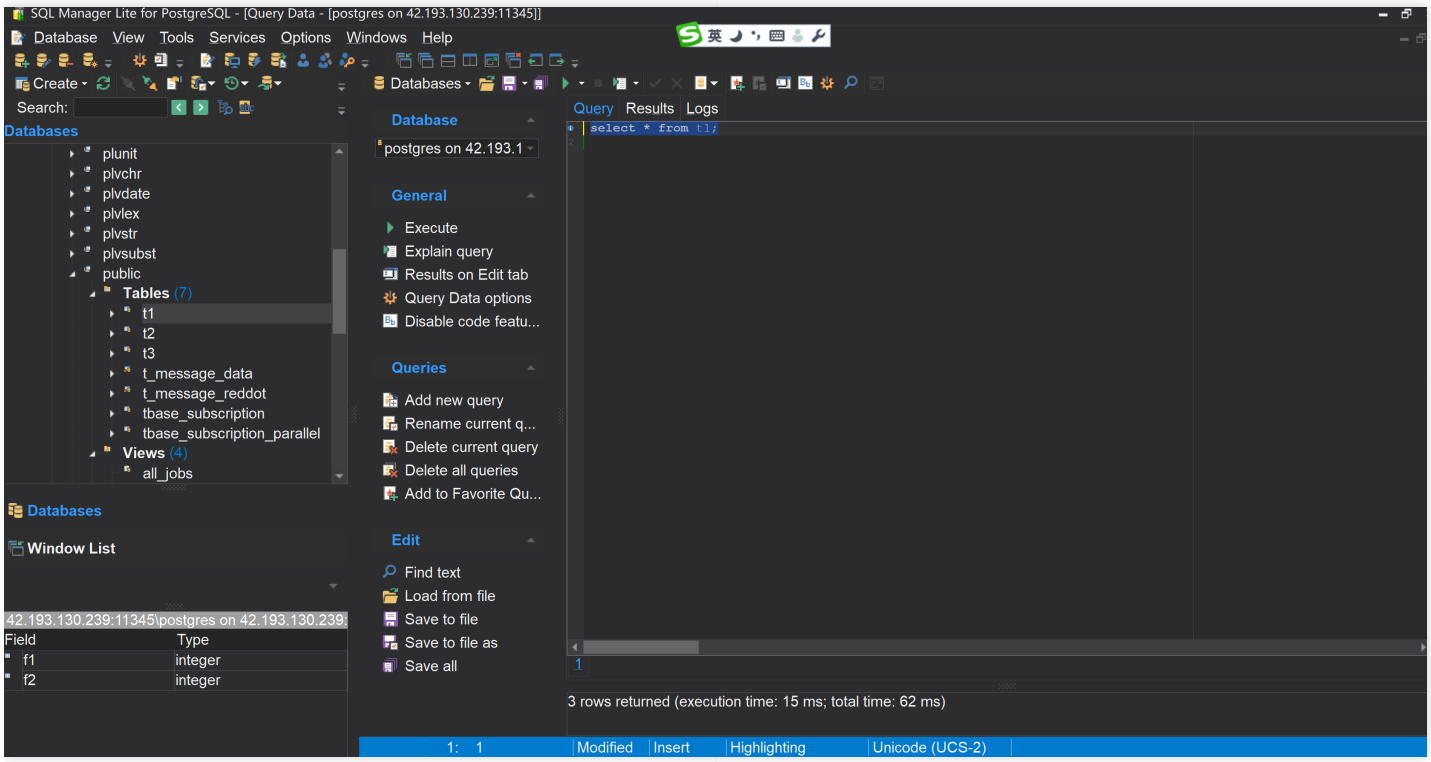

Register Database Wizard
Register Database
Set some specific options for registered database(s) and click the 'Finish' button
SQL Manager for PostgreSQL
Database name
postgres
Database alias
postgres on 42 9:11345(1)
Client encoding
UTF8 (Unicode, 8-bit)
Font charset
DEFAULT_CHARSET
Login prompt before conne
Autoconnect at startup
Refresh objects on connec
Ping server every
Show system objects
600
seconds
Create new tab for this database
Help
< Back
Finish
Cancel

查询表结构



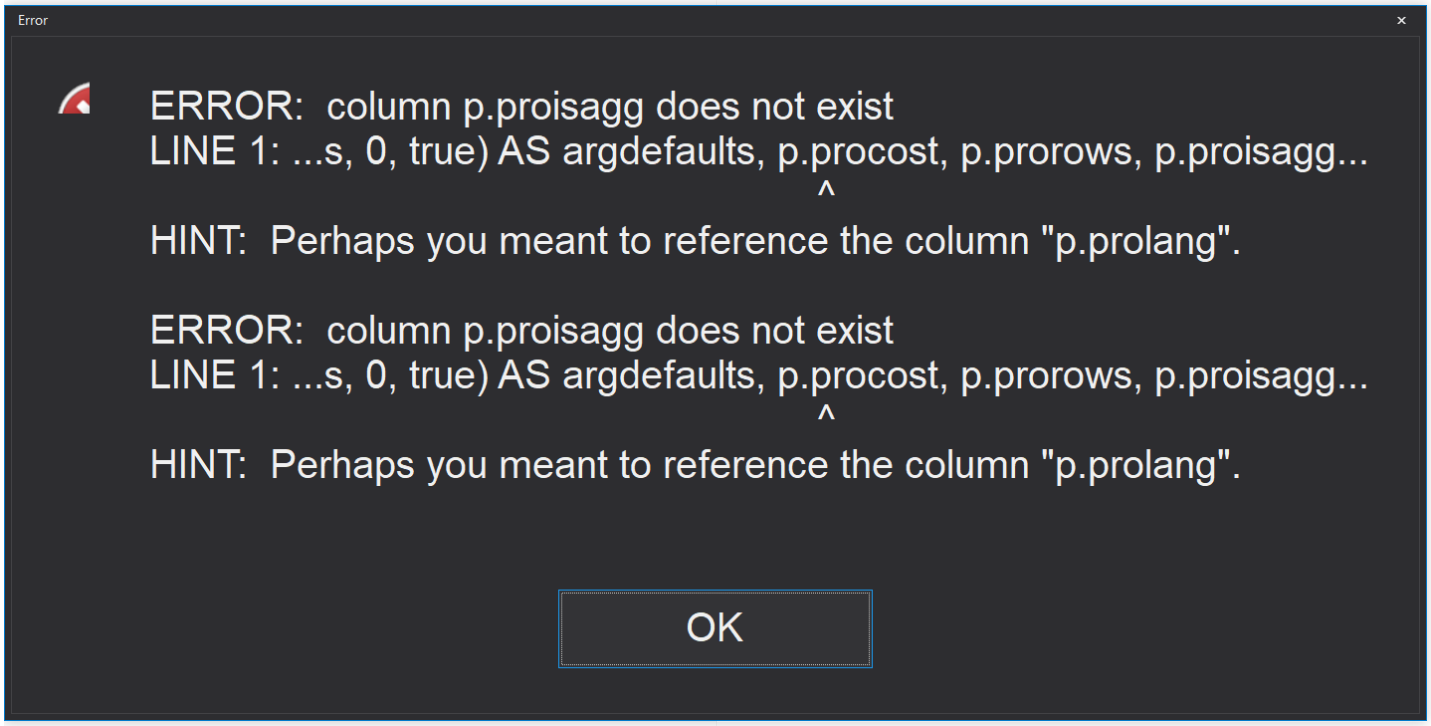
双击数据表即可显示表结构，也可增减字段。

执行SQL语句



单击菜单tools > query data，输入要执行的sql，然后单击execute即可。

连接tbase-v5出错处理



这是因为tbase merge了一些postgresql11存储过程的一些特性，我们可以通过定义一个新的pg_proc视图来达到兼容访问，如下所示。

- 连接到数据库后创建一个兼容的专用schema。

```
postgres=# create schema tbase_pg_proc;
CREATE SCHEMA

postgres=# CREATE OR REPLACE VIEW tbase_pg_proc.pg_proc as
select
   *,
   case when prokind='a' then true else false end as proisagg,
   case when prokind='w' then true else false end as proiswindow,
   oid as oid,xmin as xmin,tableoid tableoid
from
   pg_catalog.pg_proc;
CREATE VIEW
postgres=#
```

- 配置用户的搜索路径。

```
postgres=# alter role tbase set search_path to tbase_pg_proc,pg_catalog,"$user", public;
```

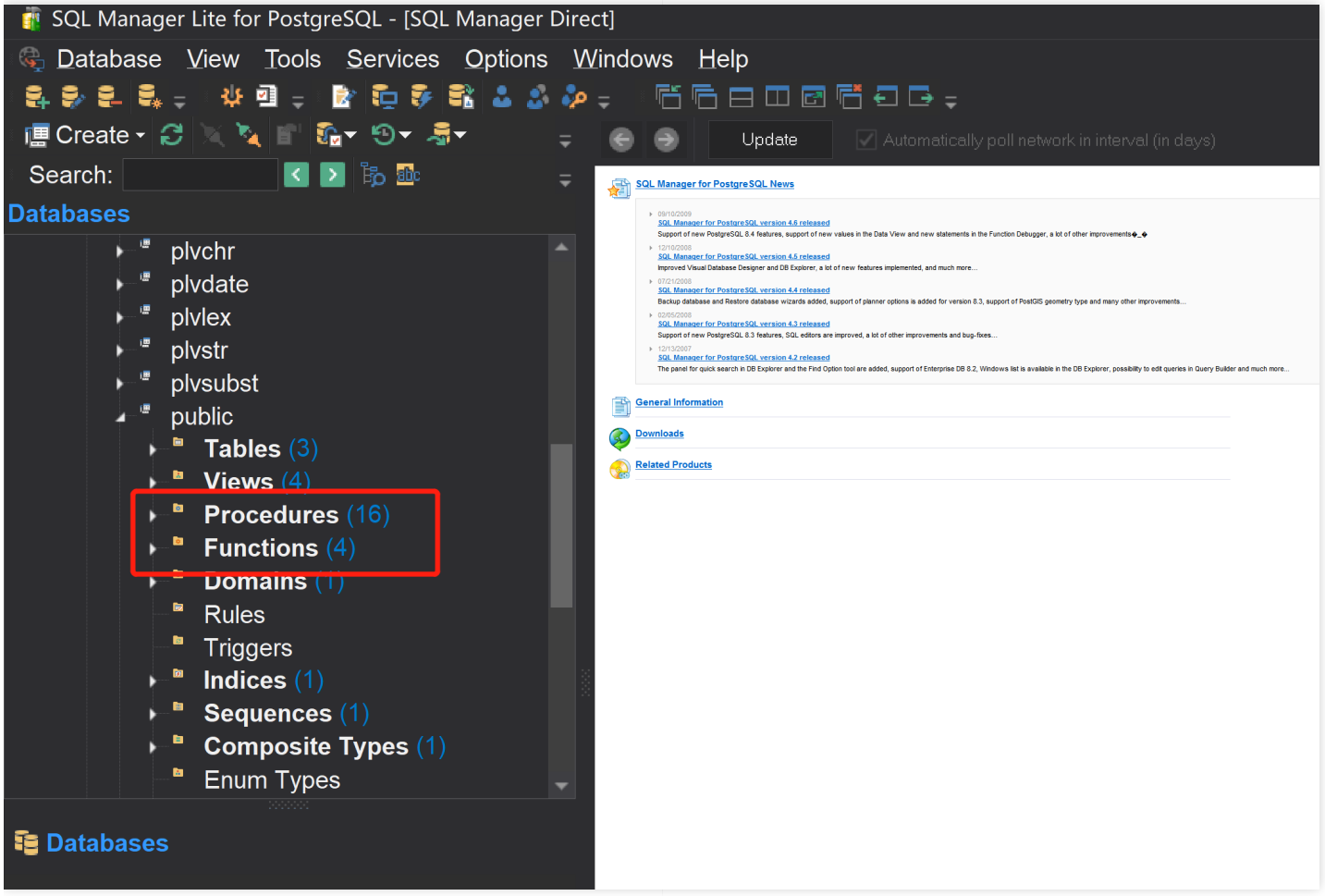
```
ALTER ROLE
postgres=#
```

再次断开，连接就可以正常访问function了。

- 适配function和procedure分开。

```
postgres=# create or replace function tbase_pg_proc.version() returns text as
$$
begin
    return 'PostgreSQL 11.0 TBase V5 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 4.8.5 20150623 (R
ed Hat 4.8.5-39), 64-bit';
end;
$$
language plpgsql;
```

返回postgresql11版本号就能将procedure和function 分别返回。



开发指南（V5.21）
概述

本文档涵盖TDSQL PostgreSQL连接方式，SQL语言开发编写，Oracle兼容特性等内容，目的是指导应用开发。
本文档适用于使用TDSQL PostgreSQL的应用开发人员、数据库开发设计人员、数据库管理员等。
本文档适用于TDSQL PostgreSQL V5.21.8及以上版本。

连接TDSQL PG
使用psql连接TDSQL PG

本节主要介绍如何使用 psql 连接 TDSQL PG。

前置条件

通过 psql 连接 TDSQL PG：

- 通过浏览器访问一体化管控平台，在左侧导航栏选择实例管理，点击需要连接的实例ID， 选择节点管理页面。
- 如果是集中式实例，点击数据节点；如果是分布式实例，点击计算节点。



- 将鼠标移动到节点IP地址后的 号上，弹出的提示框中显示了该节点配置信息，支持快捷复制。
  - bin 目录：数据库的可执行程序所在的目录
  - data 目录：数据库的数据文件目录
  - 环境变量配置：登录此节点，psql连接数据库前需要配置的环境变量。
  - psql 连接：psql连接此数据库节点的参数信息
- 根据节点IP地址，登录到相应的服务器，执行上一步环境变量配置项中的命令。

连接方式

psql可以通过四种方式连接实例，分别是：使用参数连接、使用conninfo字符串或者URI连接、配置SSL连接、配置环境变量后快捷连接。

使用参数连接

下面列出了参数连接需要配置的内容，建议直接查看一体化管控平台，复制节点的psql 连接信息：

- -h 数据库连接地址，分布式实例通常是计算节点的IP
- -p 数据库服务的端口号
- -U 数据库用户名
- -d 需要访问的数据库名称

命令格式如下：

```
psql -h <数据库连接地址> -p <端口号> -U <用户名> -d <数据库名称>
```

示例

以 tbase 身份登录 TDSQL 数据库：

```
psql -h 10.211.55.39 -p 11345 -U tbase -d postgres
```

使用conninfo字符串或者URI连接

psql通过conninfo字符串或者URI连接数据库，同样需要在conninfo或URI中指定数据库连接地址等参数。

```
conninfo: psql "host=<数据库连接地址> port=<端口号> dbname=<数据库名> user=用户名"

URI: postgresql://[用户名[:密码]@][数据库连接地址][:端口号][/数据库名][?param1=value1&...]
```

示例

示例1：通过conninfo连接数据库：

```
psql "host=10.211.55.39 port=11345 dbname=postgres user=tbase"
```

示例2：通过URI连接数据库：

```
psql postgresql://tbase@10.211.55.39:11345/postgres
```

配置SSL连接

TDSQL支持通过配置单/双向认证，开启SSL连接加密后，通过psql连接数据库。

配置单项认证连接

单项认证仅需要服务端持有证书，客户端需要验证服务端证书的有效性，确认后开启通信。

创建服务端证书：

示例采用自签名证书进行演示，实际生产环境中应该使用权威证书机构签署的证书。

> 注意：
>
> 1. 请登录一体化管控平台，在实例的节点管理页面，查看该节点配置的data 目录，确认data 目录的完整路径。
> 2. root.crt、server.crt和server.key需要拷贝到节点data 目录下，并通过chown命令，将文件的所有者修改成与postgresql.conf一致，保证数据库可执行程序有权限读取。
> 3. 配置完成后需要通过pg_ctl reload命令让数据库加载新配置的参数，也可以重启数据库使得参数生效。

1. 通过下述命令生成证书

```
# 1. 创建证书签名请求和密钥文件
openssl req -new -nodes -text -out root.csr -keyout root.key -subj "/CN=root"
# 2. 创建根证书颁发机构
openssl x509 -req -in root.csr -text -days 3650 -extfile /etc/pki/tls/openssl.cnf  -extensions v3_ca -sign
key root.key -out root.crt
# 3. 创建由根证书颁发机构签名的服务器证书
openssl req -new -nodes -text -out server.csr -keyout server.key -subj "/CN=postgres"
chmod og-rwx server.key # 确认只有文件拥有者有读写权限，移除其他用户和组用户的权限
openssl x509 -req -in server.csr -text -days 365 -CA root.crt -CAkey root.key -CAcreateserial -out server
.crt
```

2. 将上述命令生成的root.crt、server.crt和server.key拷贝到节点数据目录下，并通过chown命令，将文件的所有者修改成与postgresql.conf文件相同。
3. 修改该节点data 目录下的postgresql.conf文件，配置ssl参数。

```
ssl = on
ssl_cert_file = 'server.crt'
ssl_key_file = 'server.key'
```

4. 修改该节点data 目录下的pg_hba.conf，配置访问控制，通过hostssl指定SSL连接。

```
hostssl     all    all    0.0.0.0/0    md5
```

5. 通过pg_ctl reload命令，让数据库加载新配置的参数；如果pg_ctl命令找不到，请参照前置条件进行环境变量配置；请将下述的$data_dir替换为实际的数据目录。

```
pg_ctl reload -D $data_dir
```

6. psql连接节点，通过提示信息 SSL connection 确认该连接为SSL连接。

```
psql "host=10.211.55.39 port=11345 dbname=postgres user=tbase"

SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression:
off)
Type "help" for help.

postgres=#
```

配置双向认证连接

配置双向认证连接需要在单向认证的基础上，配置客户端证书，服务端和客户端都需要确认彼此证书的有效性，确认后开启通信。

请参考“配置单项认证连接”中的示例，创建服务端证书。

> 注意：
>
> 请登录一体化管控平台，在实例的“节点管理”页面，查看该节点配置的“data 目录”，确认数据目录的完整路径。

1. 修改该节点data 目录下的postgresql.conf文件，配置ssl参数。

```
ssl = on
ssl_cert_file = 'server.crt'
ssl_key_file = 'server.key'
ssl_ca_file = 'root.crt'
```

2. 修改该节点data 目录下的pg_hba.conf，配置访问控制，将认证方法设置为cert，SSL客户端证书认证。

```
hostssl     all    all    0.0.0.0/0    cert
```

3. 通过pg_ctl reload命令，让数据库加载新配置的参数；如果pg_ctl命令找不到，请参照前置条件进行“环境变量配置”；请将下述的$data_dir替换为实际的数据目录。

```
pg_ctl reload -D $data_dir
```

4. 创建客户端证书，示例采用自签名证书进行演示，实际生产环境中应该使用权威证书机构签署的证书.

> 注意：
>
> 请将下述的tbase用户名替换为实际连接此数据库节点的用户名。

```
openssl req -new -nodes -text -out postgresql.csr -keyout postgresql.key  -subj "/CN=tbase"
chmod og-rwx postgresql.key
openssl x509 -req -in postgresql.csr -text -days 365 -CA root.crt -CAkey root.key -CAcreateserial -out p
ostgresql.crt
```

5. 使用psql连接

   示例1：通过conninfo字符串连接

   在客户端机器创建 ~/.postgresql 目录，并将上方创建的root.crt，postgresql.crt，postgresql.key复制到 ~/.postgresql 目录。

```
psql "host=10.211.55.39 port=11345 dbname=postgres user=tbase"

SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression:
off)
```

```
Type "help" for help.

postgres=#
```

示例2：通过conninfo字符串连接，分别通过sslcert、sslkey和sslrootcert参数设置postgresql.crt、postgresql.key和root.crt的文件路径。

```
psql "host=10.211.55.39 port=11345 dbname=postgres user=tbase sslmode=require
sslcert=/PATH/TO/postgresql.crt sslkey=/PATH/TO/postgresql.key sslrootcert=/PATH/TO/root.crt"

psql (PostgreSQL 10.0 @ xxx 2024-02-21 10:27:15)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression:
off)
Type "help" for help.

postgres=#
```

示例3: 分别通过环境变量PGSSLROOTCERT、PGSSLCERT、PGSSLKEY设置root.crt、postgresql.crt和postgresql.key的文件路径。

```
export PGSSLROOTCERT=/PATH/TO/root.crt
export PGSSLCERT=/PATH/TO/postgresql.crt
export PGSSLKEY=/PATH/TO/postgresql.key

psql "host=10.211.55.39 port=11345 dbname=postgres user=tbase sslmode=require"

SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression:
off)
Type "help" for help.

postgres=#
```

配置环境变量后快捷连接

TDSQL PG 支持通过在环境变量中指定主机名称，端口地址，数据库以及数据库用户后通过命令psql即可快速连接。相关环境变量分别为 PGHOST ， PGPORT ， PGDATABASE ， PGUSER 。

示例

```
export PGUSER=tbase
export PGHOST=10.211.55.39
export PGDATABASE=postgres
export PGPORT=11345

psql
```

psql常用命令

命令	功能	命令	功能
\password	设置密码	\dt[+]	显示表格列表或表格的详细信息
\q \quit	退出psql程序	\dv[+]	显示视图列表或视图的详细信息
\conninfo	输出有关当前数据库连接的信息	\df[+]	显示函数/过程列表或函数/过程的详细信息
\timing	显示SQL语句打开和关闭的时间	\ds[+]	显示序列列表或序列的详细信息
\i	从指定文件中读取输入，并执行	\do[+]	显示操作符列表或操作符的详细信息
\o	指定结果输出文件	\du[+]	显示用户列表或用户的详细信息
\l	列出可用的数据库	\dn[+]	显示模式列表或模式的详细信息
\copy	执行一次前端拷贝	\di[+]	显示索引列表或索引的详细信息
\x	切换扩展表格格式化模式	\dx[+]	显示插件列表或插件的详细信息

更多信息

如果在通过 psql 客户端连接 TDSQL PG的过程中遇到本文档未涉及的问题，您可以查阅PostgreSQL 客户端官方的使用文档。

使用数据库连接池

本节主要介绍如何使用数据连接池连接 TDSQL PG 数据库。

简介

创建和关闭连接是一个比较昂贵的操作，为了避免频繁地创建和销毁 JDBC 连接，通常应用会选择使用数据库连接池来复用连接。Java 中常用的连接池很多 (HikariCP、c3p0、druid等)，应用可以根据业务特点选择使用合适的连接池。本文档以 HikariCP 为例，介绍如何使用数据连接池连接 TDSQL PG 数据库。

连接数据库

准备 TDSQL PG JDBC驱动包

内核Oracle模式下建议使用自研驱动，请联系云平台技术支持工程师获取软件包，可根据开发环境选择合适的驱动包：

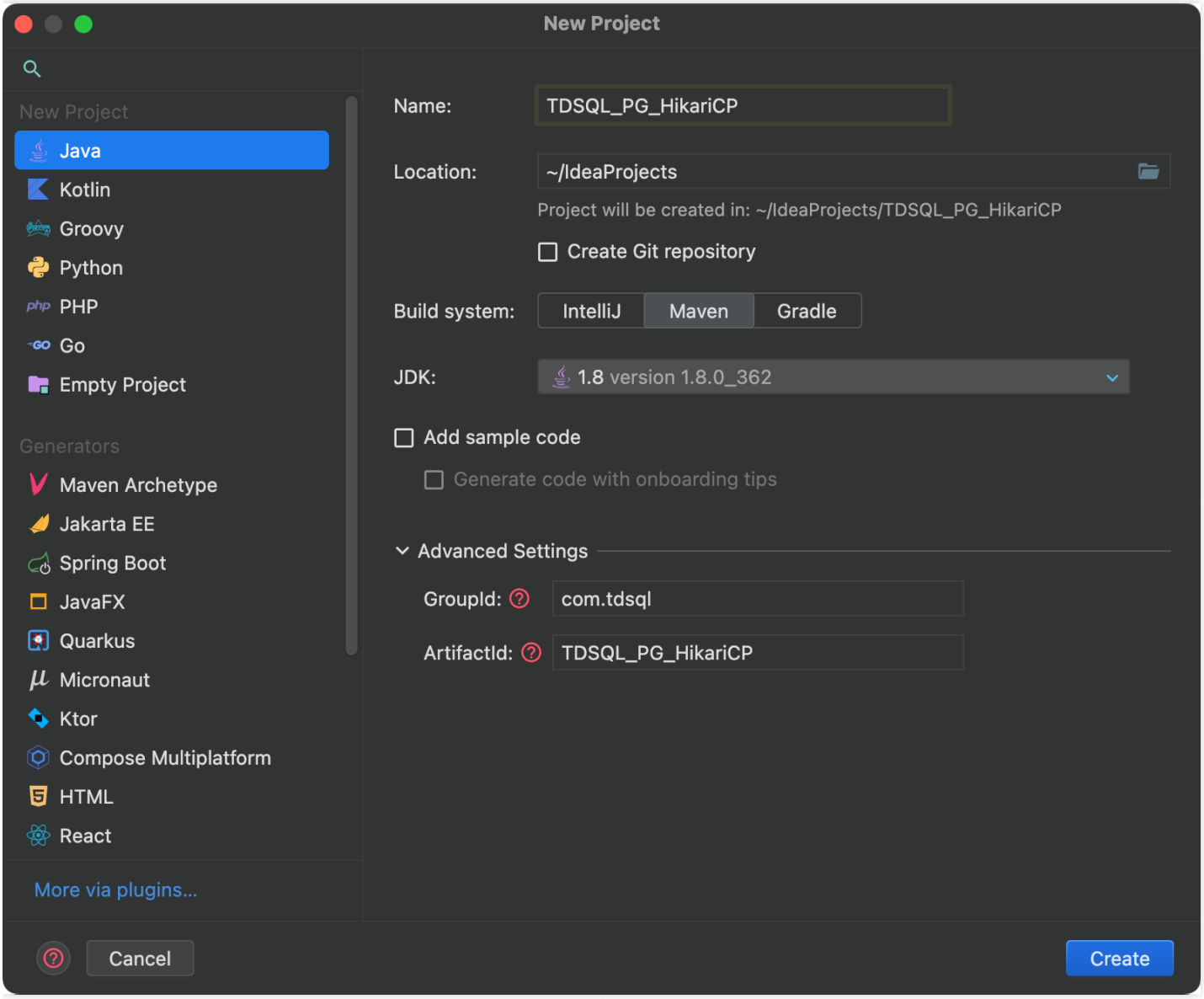
- postgresql-connector-java8-x.x.x.jar 对应 JAVA 8 及以上版本的JDK
- postgresql-connector-java7-x.x.x.jar 对应 JAVA 7版本的JDK
- postgresql-connector-java6-x.x.x.jar 对应 JAVA 6版本的JDK

从管理员获取数据库相关信息：包括数据库IP、端口、实例名、用户、密码等。驱动名和连接URL如下：

- 驱动名 org.postgresql.Driver
- 连接URL（Oracle模式下需打开oracle_compile参数） jdbc:postgresql://host:port/database?oracle_compile=true
- 更多JDBC相关配置请参考“使用JDBC连接TDSQL PG”章节

创建 Maven 项目

创建 Maven 项目，如下：



> 注意
>
> 本文档示例项目使用的 IDE 是 IntelliJ IDEA 2024.3.5 (Ultimate Edition)，也可根据喜好选择合适的 IDE。

配置 pom.xml 文件，导入项目依赖

配置文件 pom.xml 如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.tdsql</groupId>
```

```
    <artifactId>TDSQL_PG_HikariCP</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
      <maven.compiler.source>8</maven.compiler.source>
      <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <dependencies>
      <!-- HikariCP -->
      <dependency>
        <groupId>com.zaxxer</groupId>
        <artifactId>HikariCP</artifactId>
        <version>4.0.3</version>
      </dependency>
      <!-- TDSQL-PG JDBC Driver -->
      <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>1.1.4</version>
        <scope>system</scope>
        <systemPath>${project.basedir}/lib/postgresql-connector-java8-1.1.4.jar</systemPath>
      </dependency>
    </dependencies>
  </project>
```

通过 <dependencies> 定义项目所依赖的组件，主要包括：

1. HikariCP 依赖包
2. 下载 TDSQL-PG JDBC 驱动包，并在 TDSQL_PG_HikariCP 目录下新建 lib 目录，将驱动包拷贝到 lib 目录下。

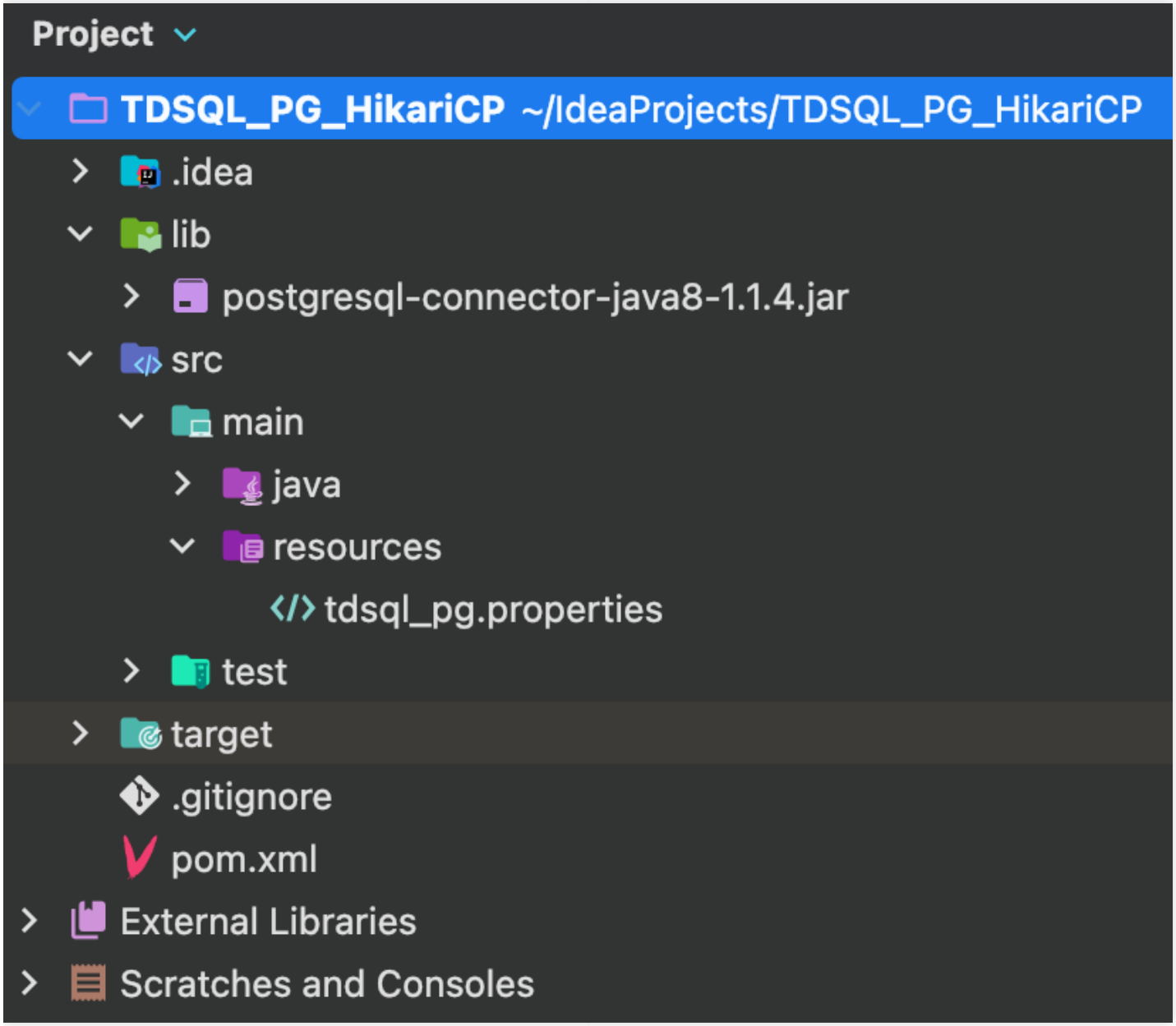
> 注意
>
> 本文所有示例需在启用Oracle兼容内核的情况下执行

配置 HikariCP

在 TDSQL_PG_HikariCP/src/main/resources 目录下新建 tdsql_pg.properties 文件，用于配置 JDBC 连接信息，如下：

```
# 配置JDBC连接信息：host、port、database、user、password 需要替换为数据库管理员提供的数据库连接信息。
jdbcUrl=jdbc:postgresql://host:port/database?oracle_compile=true
username=user
password=password
```

至此，已完成 HikariCP 项目的基本配置，完整的项目目录结构如下图所示：



开发示例

测试

- 开发使用的数据表为 STUDENT 表

```
CREATE TABLE STUDENT(ID INT PRIMARY KEY, NAME VARCHAR(50));
```

- 在 TDSQL_PG_HikariCP/src/main/java/ 下创建 Package: com.tdsql ，
- 在 TDSQL_PG_HikariCP/src/main/java/com/tdsql 下添加测试类 Main ，展示了基本的增删查改功能，如下：

```
package com.tdsql;

import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import javax.sql.DataSource;

public class Main {
    public static void main(String args[]) {
        try {
            // Load HikariCP configuration
            HikariConfig config = new HikariConfig("/tdsql_pg.properties");
            DataSource ds = new HikariDataSource(config);
            // Get connection
            Connection conn = ds.getConnection();
            Statement stmt = conn.createStatement();

            // Insert data
            System.out.println("------Insert data------");
            stmt.execute("insert into STUDENT values (1,'Alex'),(2,'Bob'),(3,'John')");
            System.out.println("Insert 3 records");

            // Query data
            System.out.println("------Query data------");
            ResultSet rs1 = stmt.executeQuery("select * from STUDENT order by id");
            while (rs1.next()) {
                System.out.println(rs1.getInt(1) + " " + rs1.getString(2));
            }

            // Update data
            System.out.println("------Update data------");
            stmt.execute("update STUDENT set name = 'Alice' where id = 1");
            System.out.println("Update the data where id is 1");

            // Query data
            System.out.println("------Query data------");
            ResultSet rs2 = stmt.executeQuery("select * from STUDENT order by id");
            while (rs2.next()) {
                System.out.println(rs2.getInt(1) + " " + rs2.getString(2));
            }

            // Delete data
            System.out.println("------Delete data------");
```

```
            stmt.execute("delete from STUDENT where id = 2");
            System.out.println("Delete the data where id is 2");

            // Query data
            System.out.println("------Query data------");
            ResultSet rs3 = stmt.executeQuery("select * from STUDENT order by id");
            while (rs3.next()) {
                System.out.println(rs3.getInt(1) + " " + rs3.getString(2));
            }

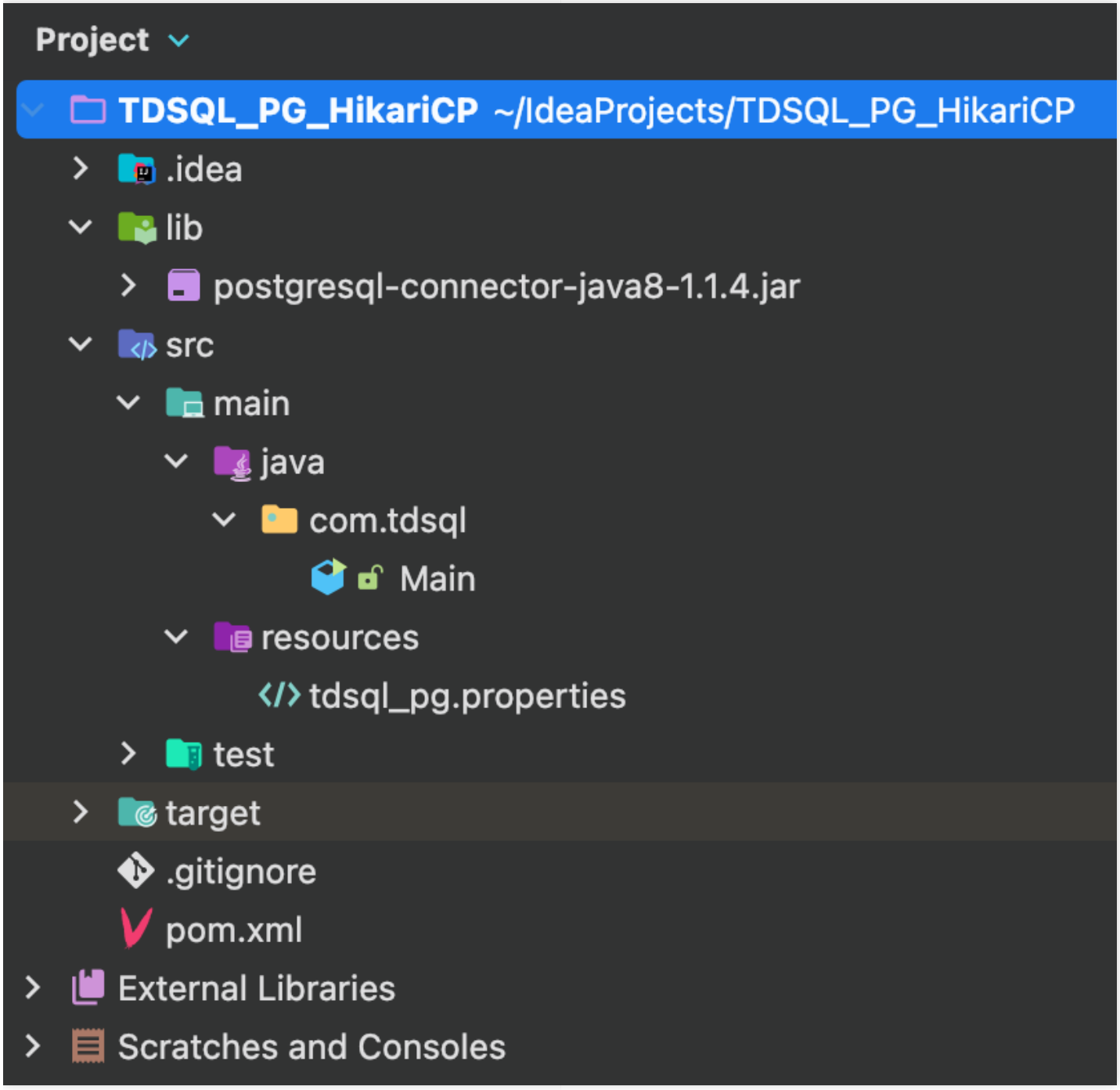
            // Close the object
            rs1.close();
            rs2.close();
            rs3.close();
            stmt.close();
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- 运行结果如下：

```
------Insert data------
Insert 3 records
------Query data------
1 Alex
2 Bob
3 John
------Update data------
Update the data where id is 1
------Query data------
1 Alice
2 Bob
3 John
------Delete data------
Delete the data where id is 2
------Query data------
1 Alice
3 John
```

附：项目结构

至此，完成了 HikariCP 项目的配置和开发示例，项目的整体结构如下图所示：


Project
TDSQL_PG_HikariCP ~/IdeaProjects/TDSQL_PG_HikariCP
.idea
lib
postgresql-connector-java8-1.1.4.jar
src
main
java
com.tdsql
Main
resources
tdsql_pg.properties
test
target
.gitignore
pom.xml
External Libraries
Scratches and Consoles

使用Java框架连接TDSQL PG
使用JDBC连接TDSQL PG

本节主要介绍如何使用 TDSQL PG JDBC 连接 TDSQL PG 数据库，实现数据增删查改等基本操作。

简介

The Java Database Connectivity (JDBC) API 是 Java 编程语言与各种数据库、SQL 数据库和其他表格数据源（如电子表格或平面文件）之间独立于数据库的连接的行业标准。JDBC API 提供了基于 SQL 数据库访问的调用级 API。TDSQL PG Connector 驱动程序是云平台独立更新和维护的、商业化的、纯 Java编写的 JDBC 驱动程序；使用 PostgreSQL原生网络协议进行通信，并特别针对 TDSQL PG 数据库的使用语法和功能特性进行了开发和适配。

连接数据库

准备 TDSQL PG JDBC驱动包

内核PG模式下建议使用社区驱动：驱动下载，并准备好java环境。
内核Oracle模式下建议使用自研驱动，请联系云平台技术支持工程师获取软件包，可根据开发环境选择合适的驱动包：

- postgresql-connector-java8-x.x.x.jar 对应 JAVA 8 及以上版本的JDK
- postgresql-connector-java7-x.x.x.jar 对应 JAVA 7版本的JDK
- postgresql-connector-java6-x.x.x.jar 对应 JAVA 6版本的JDK

配置包名和连接串

TDSQL PG JDBC 驱动名为 org.postgresql.Driver 。
用于连接 TDSQL PG 服务器的 JDBC URL 的一般格式如下，方括号（[ ]）中的选项为可选内容：
jdbc:postgresql://host[:port]/][database][?property1=value1[&property2=value2]...] ，其中：

- jdbc:postgresql: (固定)：TDSQL PG Connector 驱动程序固定使用的URL前缀。
- host (可选)：服务器地址。可以是 DNS 或 IP 地址，也可以是本地计算机的 localhost 或 127.0.0.1。默认值为 localhost。
- port (可选)：服务器上监听的端口号。默认值为 5432。
- database (可选)：数据库名。默认为连接中使用的用户名。
- propertyX (可选)：是一个或多个连接参数。

oracle_compile

- **参数说明：**JDBC驱动是否开启 Oracle 兼容模式
- 参数值： true 或 false ，默认为 false
- **前置条件：**无

```
// connection string，replace host, port, database, user and password to real settings
private static final String URL = "jdbc:postgresql://host:port/database?oracle_compile=true";
```

escapeSyntaxCallMode

- **参数说明：**是社区版本的参数，指定驱动程序如何将 JDBC 调用语法转换为底层 SQL，以调用存储过程或函数。 在 select 模式下，驱动程序始终将调用语法转换为 SELECT 语句（仅允许函数调用）。在 call 模式下，驱动程序始终将调用语法转换为CALL语句（仅允许存储过程调用）。在 callIfNoReturn 模式下，如果没有指定返回参数，则驱动程序使用 CALL 语句，否则驱动程序使用 SELECT 语句。
- 参数值： select 或 callIfNoReturn 或 call ，默认为 select
- **前置条件：**无

本文所有示例需在启用Oracle兼容内核的情况下执行

> 说明
>
> 出于演示目的，本文例子中需要连接Oracle兼容模式的数据库，如果数据库管理员尚未创建这类数据库，可以要求数据库管理员通过以下命令创建。

```
createdb -p [port] --template template0_ora -U tbase -w [password] [database name]
```

连接示例

示例程序： testConnect.java

```
import java.sql.Connection;
import java.sql.DriverManager;

public class testConnect {
   public static void main(String args[]) {
      try {
         Class.forName("org.postgresql.Driver");
         // replace host, port, database, user and password to real settings
         Connection conn = DriverManager.getConnection("jdbc:postgresql://host:port/database?oracle
_compile=true","user", "password");
```

```
            if (conn != null) {
                System.out.println("connect to database successfully");
                conn.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

> 注意
>
> host 、 port 、 database 、 user 、 password 需要改为实际使用的参数。

编译Java文件并执行，这里以 postgresql-connector-java8-1.1.4.jar 为例，具体驱动版本需要与Java安装环境匹配，下同不再重复。

```
javac -cp ".:/data/postgresql-connector-java8-1.1.4.jar" testConnect.java
java -cp ".:/data/postgresql-connector-java8-1.1.4.jar" testConnect
```

运行结果

```
connect to database successfully
```

开发示例

基础增删查改示例

示例程序： testInsert.java

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class testInsert {
    public static void main(String args[]) {
        try {
            Class.forName("org.postgresql.Driver");
            // replace host, port, database, user and password to real settings
            Connection conn = DriverManager.getConnection("jdbc:postgresql://host:port/database?oracle
_compile=true","user", "password");
            Statement stmt = conn.createStatement();
```

```
// create table
System.out.println("------create table------");
stmt.execute("drop table if exists studentTab");
stmt.execute("create table studentTab(id int, name varchar)");
System.out.println("create table studentTab");

// insert data
System.out.println("------insert data------");
stmt.execute("insert into studentTab values (1,'Alex'),(2,'Bob'),(3,'John')");
System.out.println("insert 3 rows");

// lookup data
System.out.println("------lookup data------");
ResultSet rs1 = stmt.executeQuery("select * from studentTab order by id");
while (rs1.next()) {
   System.out.println(rs1.getInt(1) + " " + rs1.getString(2));
}

// update data
System.out.println("------update data------");
stmt.execute("update studentTab set name = 'Alice' where id = 1");
System.out.println("update row id == 1");

// lookup data
System.out.println("------lookup data------");
ResultSet rs2 = stmt.executeQuery("select * from studentTab order by id");
while (rs2.next()) {
   System.out.println(rs2.getInt(1) + " " + rs2.getString(2));
}

// delete data
System.out.println("------delete data------");
stmt.execute("delete from studentTab where id = 2");
System.out.println("delete row id == 2");

// lookup data
System.out.println("------lookup data------");
ResultSet rs3 = stmt.executeQuery("select * from studentTab order by id");
while (rs3.next()) {
   System.out.println(rs3.getInt(1) + " " + rs3.getString(2));
}

// close connection
rs1.close();
rs2.close();
rs3.close();
stmt.close();
```

```
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

编译Java文件并执行。

```
javac -cp ".:/data/postgresql-connector-java8-1.1.4.jar" testInsert.java
java -cp ".:/data/postgresql-connector-java8-1.1.4.jar" testInsert
```

运行结果

```
------create table------
create table studentTab
------insert data------
insert 3 rows
------lookup data------
1 Alex
2 Bob
3 John
------update data------
update row id == 1
------lookup data------
1 Alice
2 Bob
3 John
------delete data------
deleta row id == 2
------lookup data------
1 Alice
3 John
```

prepare语法示例

示例程序： testPrepare.java

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;

public class testPrepare {
    public static void main(String args[]) {
```

```
        try {
            Class.forName("org.postgresql.Driver");
            // replace host, port, database, user and password to real settings
            Connection conn = DriverManager.getConnection("jdbc:postgresql://host:port/database?oracle
_compile=true","user", "password");
            Statement stmt = conn.createStatement();

            // resue studentTab table from previous example
            // prepare insert data
            System.out.println("------prepare insert data------");
            PreparedStatement pstmt = conn.prepareStatement("insert into studentTab values (?, ?)");
            pstmt.setInt(1, 4);
            pstmt.setString(2, "Luna");
            pstmt.execute();
            System.out.println("insert 1 row");

            // lookup data
            System.out.println("------lookup data------");
            ResultSet rs = stmt.executeQuery("select * from studentTab order by id");
            while (rs.next()) {
                System.out.println(rs.getInt(1) + " " + rs.getString(2));
            }

            // close connection
            rs.close();
            pstmt.close();
            stmt.close();
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

> 注意
>
> host 、 port 、 database 、 user 、 password 需要改为实际使用的参数。

编译Java文件并执行。

```
javac -cp ".:/data/postgresql-connector-java8-1.1.4.jar" testPrepare.java
java -cp ".:/data/postgresql-connector-java8-1.1.4.jar" testPrepare
```

运行结果

```
------prepare insert data------
```

```
insert 1 row
------lookup data------
1 Alice
3 John
4 Luna
```

调用存储过程和函数示例

示例程序： testCall.java

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.Types;

public class testCall {
   public static void main(String args[]) {
      try {
         Class.forName("org.postgresql.Driver");
         // replace host, port, database, user and password to real settings
         // add escapeSyntaxCallMode=callIfNoReturn to ensure jdbc driver knowing how to call proced
ure or function
         Connection conn = DriverManager.getConnection(
               "jdbc:postgresql://host:port/database?oracle_compile=true&escapeSyntaxCallMode=callIf
NoReturn","user", "password");
         Statement stmt = conn.createStatement();

         // create stored procedure
         stmt.execute("create or replace procedure testProc(p1 in int, p2 out int) is\n"
               + "begin\n"
               + "  p2 := p1;\n"
               + "end;");

         // call stored procedure
         System.out.println("------call stored procedure------");
         CallableStatement proc = conn.prepareCall("{call testProc(?, ?)}");
         proc.setInt(1, 100);
         proc.registerOutParameter(2, Types.INTEGER);
         proc.execute();
         System.out.println("result: " + proc.getInt(2));
         proc.close();

         // create function
         stmt.execute("create or replace function testFunc(p1 varchar) return varchar\n"
               + "is\n"
```

```
                + "begin\n"
                + "  return p1;\n"
                + "end;");

            // call function
            System.out.println("------call function------");
            CallableStatement func = conn.prepareCall("{? = call testFunc(?)}");
            func.registerOutParameter(1, Types.VARCHAR);
            func.setString(2, "test");
            func.execute();
            System.out.println("result: " + func.getString(1));
            func.close();

            // close connection
            stmt.close();
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

> 注意
>
> host 、 port 、 database 、 user 、 password 需要改为实际使用的参数。
>
> 需要添加escapeSyntaxCallMode=callIfNoReturn参数，以确保驱动程序正确调用存储过程或者函数。

编译Java文件并执行。

```
javac -cp ".:/data/postgresql-connector-java8-1.1.4.jar" testCall.java
java -cp ".:/data/postgresql-connector-java8-1.1.4.jar" testCall
```

运行结果

```
------call stored procedure------
result: 100
------call function------
result: test
```

使用Hibernate框架连接TDSQL PG

本节主要介绍如何使用 Hibernate框架 连接 TDSQL PG 数据库，实现数据的增删查改等基本操作。

简介

Hibernate是一个开放源代码的对象关系映射框架，它对JDBC进行了非常轻量级的对象封装，它将POJO与数据库表建立映射关系，是一个全自动的orm框架，hibernate可以自动生成SQL语句，自动执行，使得Java程序员可以随心所欲的使用对象编程思维来操纵数据库。

引入和配置Hibernate框架

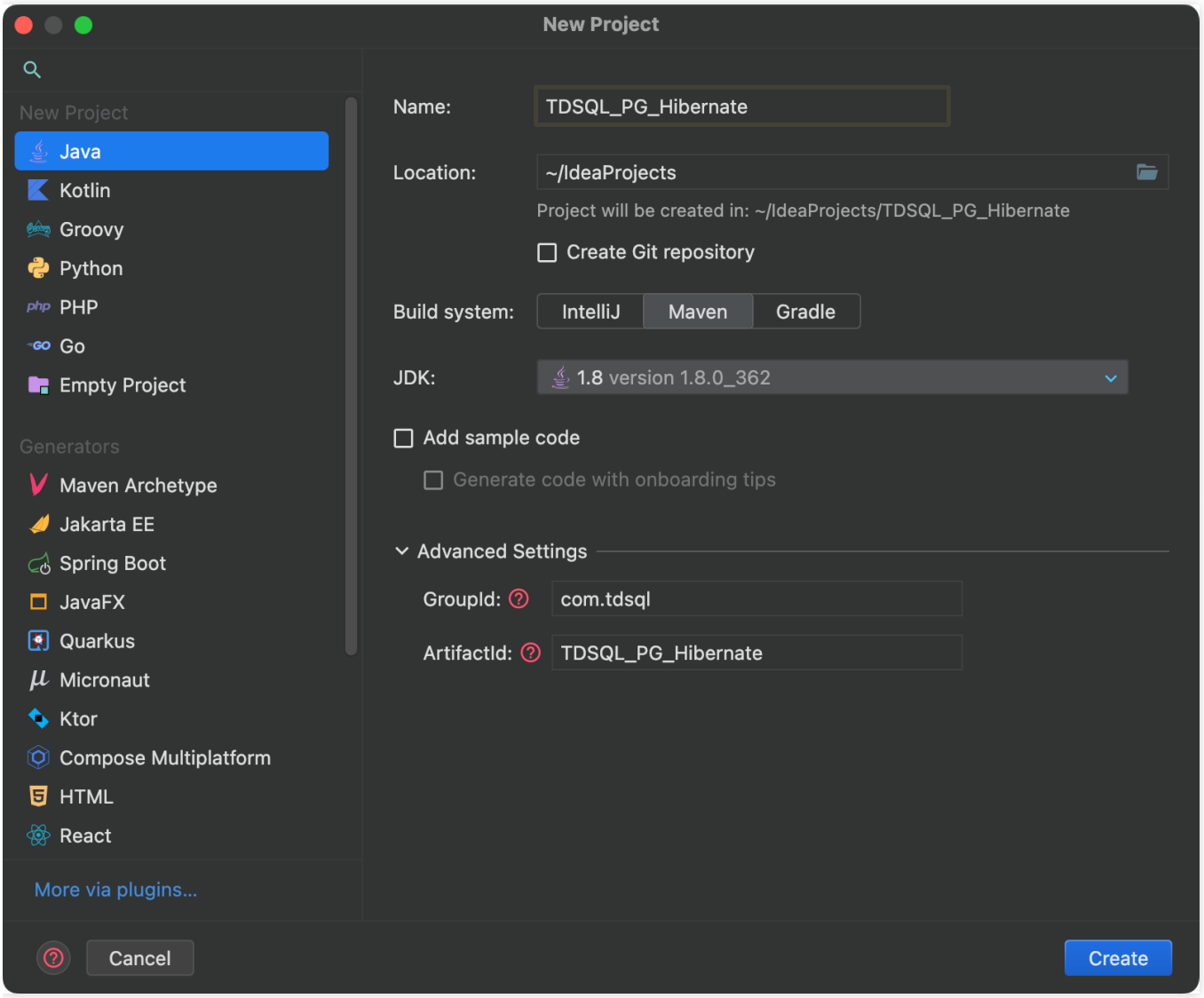
前置依赖

在配置之前需要保证你的机器上已安装对应版本的JDK，本例是 JDK 8。
如内核运行在Oracle兼容模式下，建议使用自研TDSQL PG JDBC驱动，请联系云技术支持工程师获取。联系数据库管理员获取数据库相关信息：包括数据库IP、端口、实例名、用户、密码等。驱动名和连接URL如下：

- 驱动名 org.postgresql.Driver
- 连接URL（Oracle模式下需打开oracle_compile参数） jdbc:postgresql://host:port/database?oracle_compile=true

创建 Maven 项目

创建 Maven 项目，如下：



> 注意
>
> 本文档示例项目使用的 IDE 是 IntelliJ IDEA 2022.3 (Ultimate Edition)，也可根据喜好选择合适的 IDE。

配置 pom.xml 文件，导入项目依赖

配置文件 pom.xml 如下：
配置dependencies部分。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.tdsql</groupId>
    <artifactId>TDSQL_PG_Hibernate</artifactId>
```

```
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- orm framework Hibernate -->
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>5.6.15.Final</version>
        </dependency>
        <!-- Junit5 -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.10.0</version>
            <scope>test</scope>
        </dependency>
        <!-- TDSQL-PG JDBC driver -->
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>1.1.4</version>
            <scope>system</scope>
            <systemPath>${project.basedir}/lib/postgresql-connector-java8-1.1.4.jar</systemPath>
        </dependency>
    </dependencies>

</project>
```

通过 <dependencies> 定义项目所依赖的组件，主要包括：

1. Hibernate 依赖包，填写好后会从公共库自动下载。
2. Junit5 测试框架依赖包，填写好后会从公共库自动下载。
3. 下载 TDSQL PG JDBC 驱动包，并在 TDSQL_PG_Hibernate 目录下新建 lib 目录，将驱动包拷贝到 lib 目录下。

> 注意
>
> 1.TDSQL PG JDBC驱动包请联系云技术支持工程师获取
>
> 2.本文所有示例需在启用Oracle兼容内核的情况下执行
>
> 3.示例 pom.xml 中的 TDSQL_PG JDBC 驱动版本，应与下载驱动包的版本相同（以1.1.4版本为例）。
>
> 4.部分版本的IntelliJ IDEA使用上述 pom.xml 报错Failed to resolve org.junit.platform:junit-platform-launcher:1.10.0，解决方法是在 pom.xml 中增加以下内容:
>
> ```
>     <groupId>org.junit.platform</groupId>
> ```

```
<artifactId>junit-platform-launcher</artifactId>

<version>1.10.0</version>

<scope>test</scope>
```

配置 Hibernate

在 TDSQL_PG_Hibernate/src/main/resources 目录下新建 hibernate.cfg.xml 文件，用于配置 Hibernate， hibernate.cfg.xml 配置如下：

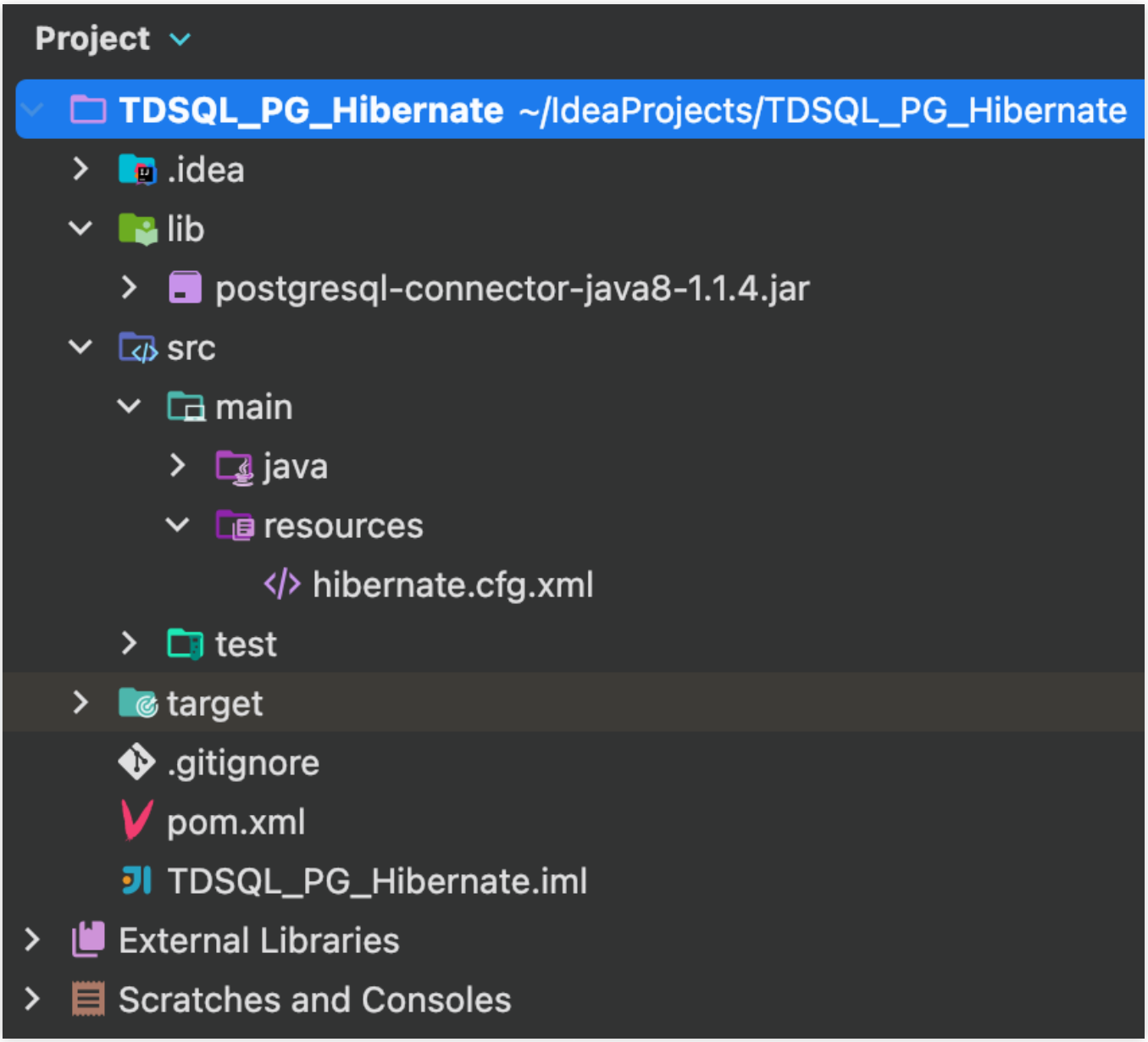
```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- JDBC connection info：replace host、port、database、user and password to real settings。 -->
        <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
        <property name="hibernate.connection.url">jdbc:postgresql://host:port/database?oracle_compile=true
</property>
        <property name="hibernate.connection.username">user</property>
        <property name="hibernate.connection.password">password</property>

        <!-- configure database dialect -->
        <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQL10Dialect</property>
    </session-factory>
</hibernate-configuration>
```

其中，需要配置 JDBC 连接信息和数据库方言：

- hibernate.connection.driver_class ：指定驱动包名为 org.postgresql.Driver 。
- hibernate.connection.url ：指定连接URL，包括数据库地址、端口、数据库名、连接属性等信息。
- hibernate.connection.username ：指定用户名。
- hibernate.connection.password ：指定相应用户的密码。
- hibernate.dialect ：指定数据库方言为PostgreSQL的数据库方言，使 Hibernate 生成适当的 SQL。

至此，已完成 Hibernate 项目的基本配置，完整的项目目录结构如下图所示：



开发示例

添加数据表对应的实体类

首先，开发使用的数据表为 STUDENT 表（ CREATE TABLE STUDENT(ID INT PRIMARY KEY, NAME VARCHAR(50)) ），
对应地，添加实体类 Student ，先在 TDSQL_PG_Hibernate/src/main/java/ 下创建 Package: com.tdsql.pojo 再在 TDSQL_PG_Hibernate/src/main/java/com/tdsql/pojo 目录下新建 Student.java 文件：

```
package com.tdsql.pojo;

public class Student {
    private int id;
```

```
    private String name;

    public Student() {
    }

    public Student(int id, String name) {
       this.id = id;
       this.name = name;
    }

    public int getId() {
       return id;
    }

    public void setId(int id) {
       this.id = id;
    }

    public String getName() {
       return name;
    }

    public void setName(String name) {
       this.name = name;
    }

    @Override
    public String toString() {
       return "Student [id=" + id + ", name=" + name + "]";
    }
}
```

配置实体类与数据表之间的映射

其次，需要将 Java 对象 Student 映射到数据库中的表 STUDENT ，先在 TDSQL_PG_Hibernate/src/main/resources/ 下创建文件夹: com/tdsql/pojo 在 TDSQL_PG_Hibernate/src/main/resources/com/tdsql/pojo 下新建 Student.hbm.xml 映射文件，如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
      "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.tdsql.pojo">
   <!-- configure table and attributes -->
   <class name="Student" table="STUDENT">
      <id name="id" column="ID" type="java.lang.Integer">
         <generator class="org.hibernate.id.Assigned" />
      </id>
      <property name="name" column="NAME" length="50" type="java.lang.String"/>
```

```
    </class>
</hibernate-mapping>
```

同时，需要将该映射文件的路径写入到 Hibernate 配置文件 hibernate.cfg.xml 中，如下：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- configure database info：replace host、port、database、user and password to real settings。 -->
        <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
        <property name="hibernate.connection.url">jdbc:postgresql://host:port/database?oracle_compile=true
</property>
        <property name="hibernate.connection.username">user</property>
        <property name="hibernate.connection.password">password</property>

        <!-- configure database dialect -->
        <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQL10Dialect</property>

        <!-- add mapping -->
        <mapping resource="com/tdsql/pojo/Student.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

封装数据库表的增删查改操作

然后，在 TDSQL_PG_Hibernate/src/main/java/ 下创建Package com.tdsql.dao ，在 TDSQL_PG_Hibernate/src/main/java/com/tdsql/dao 下新建 StudentDao 类，定义与数据库交互相关的方法，以实现数据的增删查改，如下：

```
package com.tdsql.dao;

import com.tdsql.pojo.Student;
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.query.Query;

public class StudentDao {
    private Session session = null;
    {
        // read hibernate.cfg.xml
        Configuration cfg = new Configuration().configure();
        SessionFactory sessionFactory = cfg.buildSessionFactory();
        // open session
        session = sessionFactory.openSession();
    }
```

```
    // do insert
    public int insertStudent(Student student) {
        session.beginTransaction();
        session.save(student);
        session.getTransaction().commit();
        return 1;
    }

    // do lookup
    public List<Student> selectAllStudent() {
        String sql = "from Student order by id";
        Query query = session.createQuery(sql);
        List<Student> students = query.list();
        return students;
    }

    // do update
    public int updateStudentById(Student student) {
        session.beginTransaction();
        Student student1 = session.get(Student.class, student.getId());
        session.merge(student);
        session.getTransaction().commit();
        return 1;
    }

    // do delete
    public int deleteStudentById(int id) {
        session.beginTransaction();
        Student student = session.get(Student.class, id);
        session.delete(student);
        session.getTransaction().commit();
        return 1;
    }
}
```

更多 Hibernate 操作，请查询 Hibernate 官网。

测试

最后，在 TDSQL_PG_Hibernate/src/test/java 下添加测试类 TestStudent ，展示了基本的增删查改功能，如下：

```
import com.tdsql.dao.StudentDao;
import com.tdsql.pojo.Student;

import java.util.List;
import org.junit.jupiter.api.Test;

public class TestStudent {
    @Test
```

```
    public void test() {
        StudentDao studentDao = new StudentDao();
        System.out.println("------insert data------");
        studentDao.insertStudent(new Student(1, "Alex"));
        studentDao.insertStudent(new Student(2, "Bob"));
        studentDao.insertStudent(new Student(3, "John"));
        System.out.println("insert 3 rows");

        System.out.println("------lookup data------");
        List<Student> students1 = studentDao.selectAllStudent();
        for (Student student: students1) {
            System.out.println(student);
        }

        System.out.println("------update data------");
        studentDao.updateStudentById(new Student(1, "Alice"));
        System.out.println("change row id == 1");

        System.out.println("------lookup data------");
        List<Student> students2 = studentDao.selectAllStudent();
        for (Student student: students2) {
            System.out.println(student);
        }

        System.out.println("------delete data------");
        studentDao.deleteStudentById(2);
        System.out.println("delete row id == 2");

        System.out.println("------lookup data------");
        List<Student> students3 = studentDao.selectAllStudent();
        for (Student student: students3) {
            System.out.println(student);
        }
    }
}
```

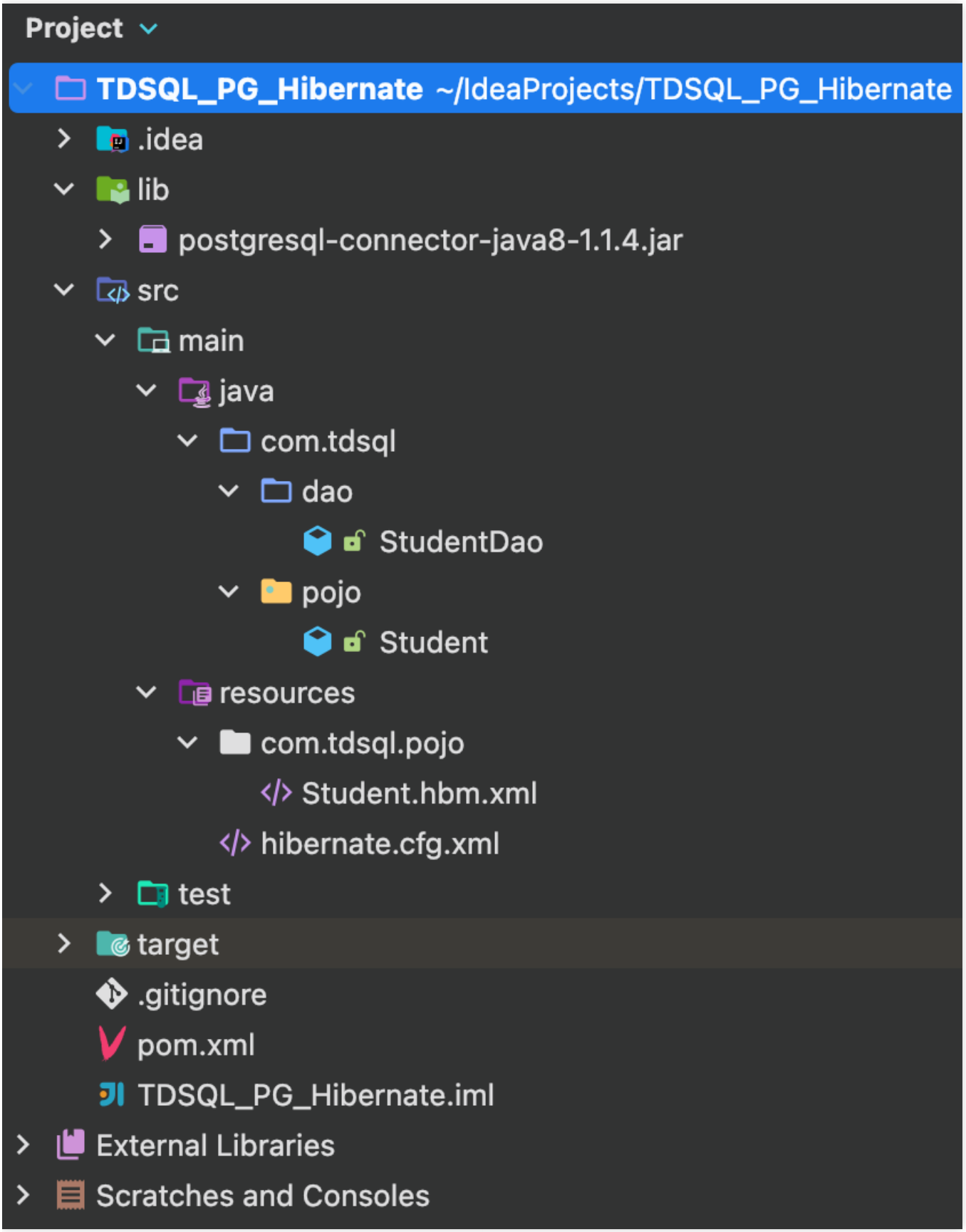
运行结果如下：

```
------insert data------
insert 3 rows
------lookup data------
Student [id=1, name=Alex]
Student [id=2, name=Bob]
Student [id=3, name=John]
------update data------
change row id == 1
------lookup data------
Student [id=1, name=Alice]
Student [id=2, name=Bob]
```

```
Student [id=3, name=John]
------delete data------
delete row id == 2
------lookup data------
Student [id=1, name=Alice]
Student [id=3, name=John]
```

附：项目结构

至此，完成了 Hibernate 项目的配置和开发示例，项目的整体结构如下图所示：


Project
TDSQL_PG_Hibernate ~/IdeaProjects/TDSQL_PG_Hibernate
.idea
lib
postgresql-connector-java8-1.1.4.jar
src
main
java
com.tdsql
dao
StudentDao
pojo
Student
resources
com.tdsql.pojo
Student.hbm.xml
hibernate.cfg.xml
test
target
.gitignore
pom.xml
TDSQL_PG_Hibernate.iml
External Libraries
Scratches and Consoles

使用MyBatis框架连接TDSQL PG

本节主要介绍如何使用 MyBatis 框架 连接 TDSQL PG 数据库，实现数据的增删查改等基本操作。

简介

MyBatis 是一款优秀的持久层框架，它支持自定义 SQL、存储过程以及高级映射。MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。MyBatis 可以通过简单的 XML 或注解来配置和映射原始类型，将接口和 Java POJO（Plain Old Java Objects，普通的 Java 对象）映射成数据库中的记录。

引入和配置MyBatis框架

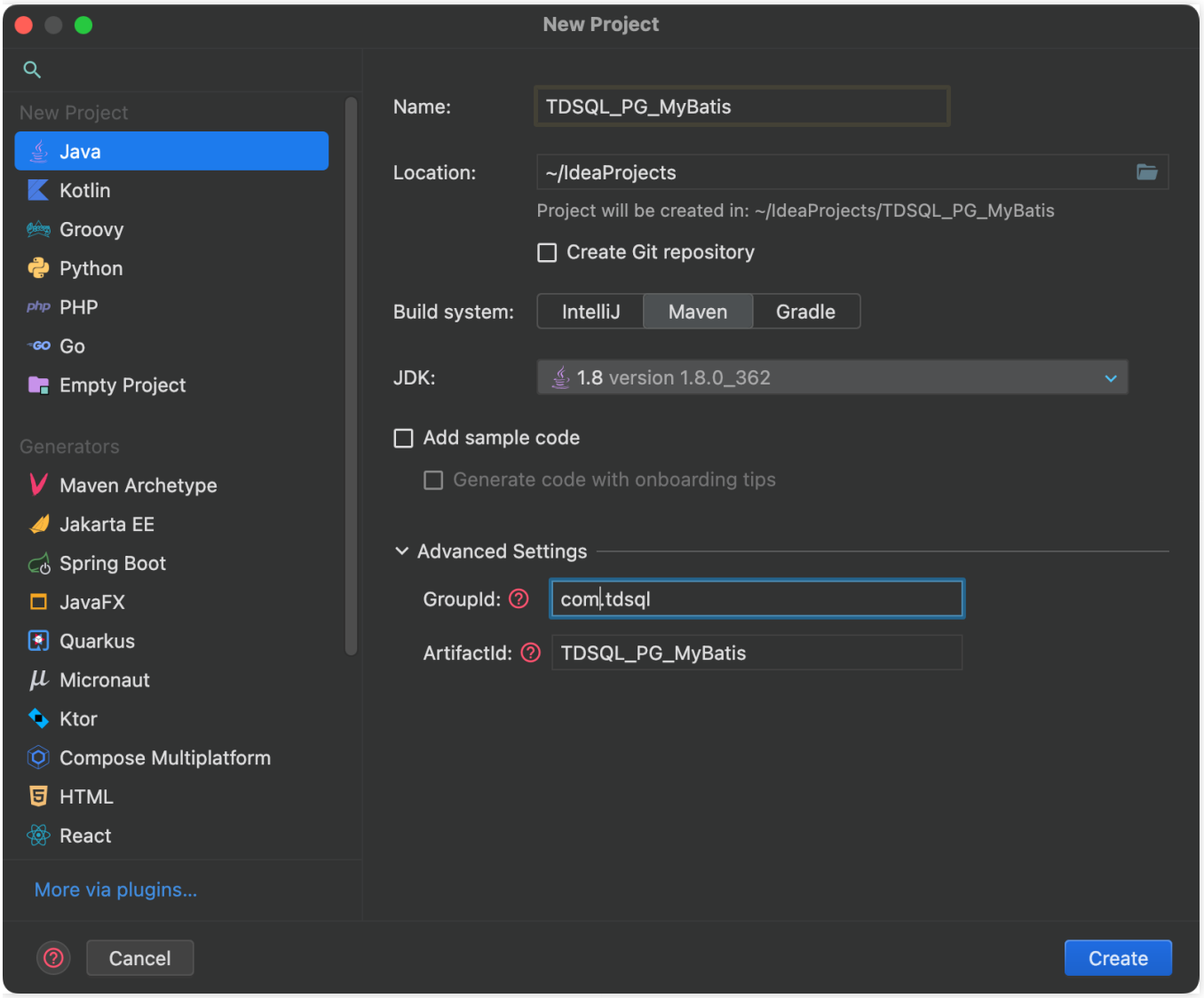
前置依赖

在配置之前需要保证你的机器上已安装对应版本的JDK，本例是 JDK 8。
内核Oracle模式下建议使用自研TDSQL PG JDBC驱动，请联系云技术支持工程师获取。联系数据库管理员获取数据库相关信息：包括数据库IP、端口、实例名、用户、密码等。驱动名和连接URL如下：

- 驱动名 org.postgresql.Driver
- 连接URL（Oracle模式下需打开oracle_compile参数） jdbc:postgresql://host:port/database?oracle_compile=true

创建 Maven 项目

创建 Maven 项目，如下：



> 注意
>
> 本文档示例项目使用的 IDE 是 IntelliJ IDEA 2022.3 (Ultimate Edition)，也可根据喜好选择合适的 IDE。

配置 pom.xml 文件，导入项目依赖

配置文件 pom.xml 如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.tdsql</groupId>
    <artifactId>TDSQL_PG_MyBatis</artifactId>
    <version>1.0-SNAPSHOT</version>
```

```
    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- MyBatis -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.11</version>
        </dependency>
        <!-- Junit5 -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.10.0</version>
            <scope>test</scope>
        </dependency>
        <!-- TDSQL-PG JDBC driver -->
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>1.1.4</version>
            <scope>system</scope>
            <systemPath>${project.basedir}/lib/postgresql-connector-java8-1.1.4.jar</systemPath>
        </dependency>
    </dependencies>

</project>
```

通过 <dependencies> 定义项目所依赖的组件，主要包括：

1. MyBatis 依赖包，填写好后会从公共库自动下载。
2. Junit5 测试框架依赖包，填写好后会从公共库自动下载。
3. 下载 TDSQL PG JDBC 驱动包，并在 TDSQL_PG_MyBatis 目录下新建 lib 目录，将驱动包拷贝到 lib 目录下。

> 注意
>
> 1.TDSQL PG JDBC驱动包请联系云技术支持工程师获取。
>
> 2.本文所有示例需在启用Oracle兼容内核的情况下执行。
>
> 3.示例 pom.xml 中的 TDSQL_PG JDBC 驱动版本，应与下载驱动包的版本相同（以1.1.4版本为例）。
>
> 4.部分版本的IntelliJ IDEA使用上述 pom.xml 报错Failed to resolve org.junit.platform:junit-platform-launcher:1.10.0，解决方法是在 pom.xml 中增加以下内容:
>
> ```
> <groupId>org.junit.platform</groupId>
> ```

```
    <artifactId>junit-platform-launcher</artifactId>

    <version>1.10.0</version>

    <scope>test</scope>
```

配置 MyBatis

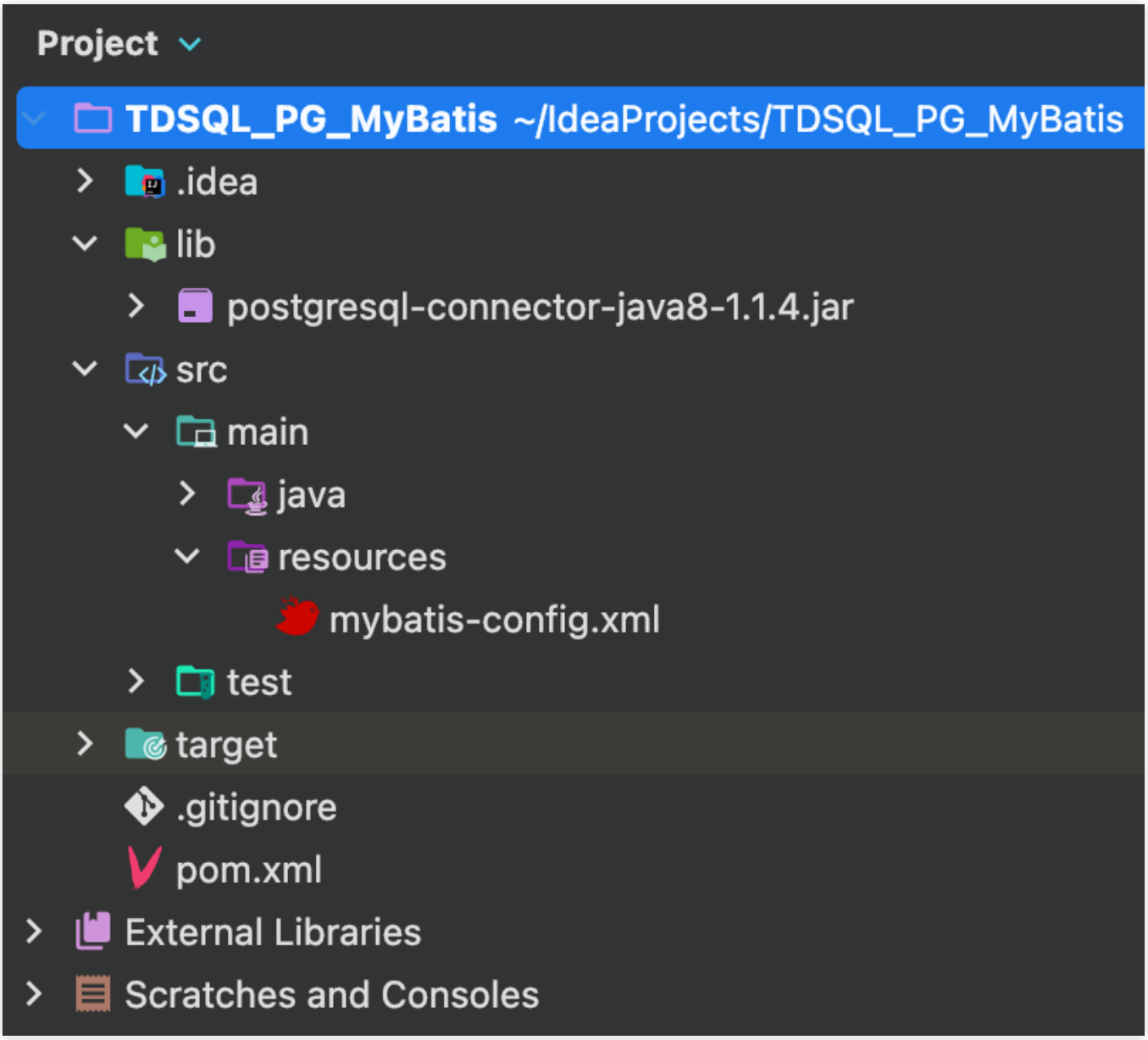
在 TDSQL_PG_MyBatis/src/main/resources 目录下新建 mybatis-config.xml 文件，用于配置 MyBatis， mybatis-config.xml 配置如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
      PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
      "https://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
   <!-- configure MyBatis runtime -->
   <environments default="development">
      <environment id="development">
         <transactionManager type="JDBC"/>
         <dataSource type="POOLED">
            <!-- configure JDBC connect info：replace host, port, database, user and password to real settings
-->
            <property name="driver" value="org.postgresql.Driver"/>
            <property name="url" value="jdbc:postgresql://host:port/database?oracle_compile=true"/>
            <property name="username" value="user"/>
            <property name="password" value="password"/>
         </dataSource>
      </environment>
   </environments>
</configuration>
```

其中，需要配置 JDBC 连接信息，仅修改 value 的值：

- driver ：指定驱动包名为 org.postgresql.Driver 。
- url ：指定连接URL，包括数据库地址、端口、数据库名、连接属性等信息。
- username ：指定用户名。
- password ：指定相应用户的密码。

至此，已完成 MyBatis 项目的基本配置，完整的项目目录结构如下图所示：



开发示例

添加数据表对应的实体类

首先，开发使用的数据表为 STUDENT 表（ CREATE TABLE STUDENT(ID INT PRIMARY KEY, NAME VARCHAR(50)) ），
对应地，添加实体类 Student ，先在 TDSQL_PG_MyBatis/src/main/java/ 下创建 Package: com.tdsql.pojo 再在 TDSQL_PG_MyBatis/src/main/java/com/tdsql/pojo 下新建 Student.java 文件：

```
package com.tdsql.pojo;

public class Student {
    private int id;
```

```
    private String name;

    public Student() {
    }

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Student [id=" + id + ", name=" + name + "]";
    }
}
```

定义数据库操作接口

其次，在 TDSQL_PG_MyBatis/src/main/java/ 下创建 Package：com.tdsql.mapper 再在 TDSQL_PG_MyBatis/src/main/java/com/tdsql/mapper 下新建 StudentMapper.java 文件，定义数据库的增删查改接口，如下：

```
package com.tdsql.mapper;

import com.tdsql.pojo.Student;
import java.util.List;

public interface StudentMapper {
    // insert
    Integer insertStudent(Student student);

    // lookup
    List<Student> selectAllStudent();
```

```
    // update
    Integer updateStudentById(Student student);

    // delete
    Integer deleteStudentById(Integer id);
}
```

配置操作接口与对应SQL语句的映射关系

然后，需要将数据库的增删查改接口映射到对应的 SQL 语句，在 TDSQL_PG_MyBatis/src/main/resources/ 下创建目录 com/tdsql/mapper ，
再在 TDSQL_PG_MyBatis/src/main/resources/com/tdsql/mapper 下新建 StudentMapper.xml 映射文件，如下：

```
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "https://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!-- configure namespace to interface StudentMapper -->
<mapper namespace="com.tdsql.mapper.StudentMapper">
    <insert id="insertStudent" parameterType="com.tdsql.pojo.Student">
        insert into STUDENT (ID, NAME) values (#{id}, #{name})
    </insert>
    <select id="selectAllStudent" resultType="com.tdsql.pojo.Student">
        select * from STUDENT order by ID
    </select>
    <update id="updateStudentById" parameterType="com.tdsql.pojo.Student">
        update STUDENT set NAME = #{name} where ID = #{id}
    </update>
    <delete id="deleteStudentById" parameterType="java.lang.Integer">
        delete from STUDENT where ID = #{id};
    </delete>
</mapper>
```

其中，定义四个 SQL 语句，分别对应了 StudentMapper 接口类的增删查改接口：

- insertStudent ：向 STUDENT 表中插入一条记录，包括 ID 和 NAME 两个字段，值分别为 #{id} 和 #{name} ,传入的参数类型则为 com.tdsql.pojo.Student 。
- selectAllStudent ：从 STUDENT 表中查询所有记录，返回的结果类型为 com.tdsql.pojo.Student 。
- updateStudentById ：根据 ID 字段去更新相应记录的 NAME 字段，值分别为 #{id} 和 #{name} ，传入的参数类型为 com.tdsql.pojo.Student 。
- deleteStudentById ：根据 ID 字段删除相应的记录，值为 #{id} ,传入的参数类型为 java.lang.Integer 。

同时，需要将 StudentMapper.xml 映射文件的路径写入到 MyBatis 配置文件 mybatis-config.xml 中，如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "https://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
```

```
    <!-- configure MyBatis runtime -->
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <!-- configure JDBC connection info：replace host, port, database, user and password to real setti
ngs -->
                <property name="driver" value="org.postgresql.Driver"/>
                <property name="url" value="jdbc:postgresql://host:port/database?oracle_compile=true"/>
                <property name="username" value="user"/>
                <property name="password" value="password"/>
            </dataSource>
        </environment>
    </environments>
    <!-- add mapper -->
    <mappers>
        <mapper resource="com/tdsql/mapper/StudentMapper.xml"/>
    </mappers>
</configuration>
```

测试

最后，在 TDSQL_PG_MyBatis/src/test/java 下添加测试类 TestStudent ，展示了基本的增删查改功能，如下：

```
import com.tdsql.mapper.StudentMapper;
import com.tdsql.pojo.Student;

import java.io.IOException;
import java.io.InputStream;
import java.util.List;

import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.jupiter.api.Test;

public class TestStudent {
    @Test
    public void test() throws IOException {
        // load MyBatis configuration
        InputStream inputStream = Resources.getResourceAsStream("mybatis-config.xml");
        // create factory instance to spawn SqlSession
        SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(inputStream);
        // create SqlSession instance，run SQL statement
        SqlSession session = factory.openSession();
        // get StudentMapper instance
        StudentMapper studentMapper = session.getMapper(StudentMapper.class);
        System.out.println("------insert data------");
```

```
        studentMapper.insertStudent(new Student(1, "Alex"));
        studentMapper.insertStudent(new Student(2, "Bob"));
        studentMapper.insertStudent(new Student(3, "John"));
        // commit
        session.commit();
        System.out.println("insert 3 rows");

        System.out.println("------lookup data------");
        List<Student> students1 = studentMapper.selectAllStudent();
        for (Student student: students1) {
            System.out.println(student);
        }

        System.out.println("------update data------");
        studentMapper.updateStudentById(new Student(1, "Alice"));
        // commit
        session.commit();
        System.out.println("update row id == 1");

        System.out.println("------lookup data------");
        List<Student> students2 = studentMapper.selectAllStudent();
        for (Student student: students2) {
            System.out.println(student);
        }

        System.out.println("------delete data------");
        studentMapper.deleteStudentById(2);
        // commit
        session.commit();
        System.out.println("delete row id == 2");

        System.out.println("------lookup data------");
        List<Student> students3 = studentMapper.selectAllStudent();
        for (Student student: students3) {
            System.out.println(student);
        }

        session.close();
    }
}
```

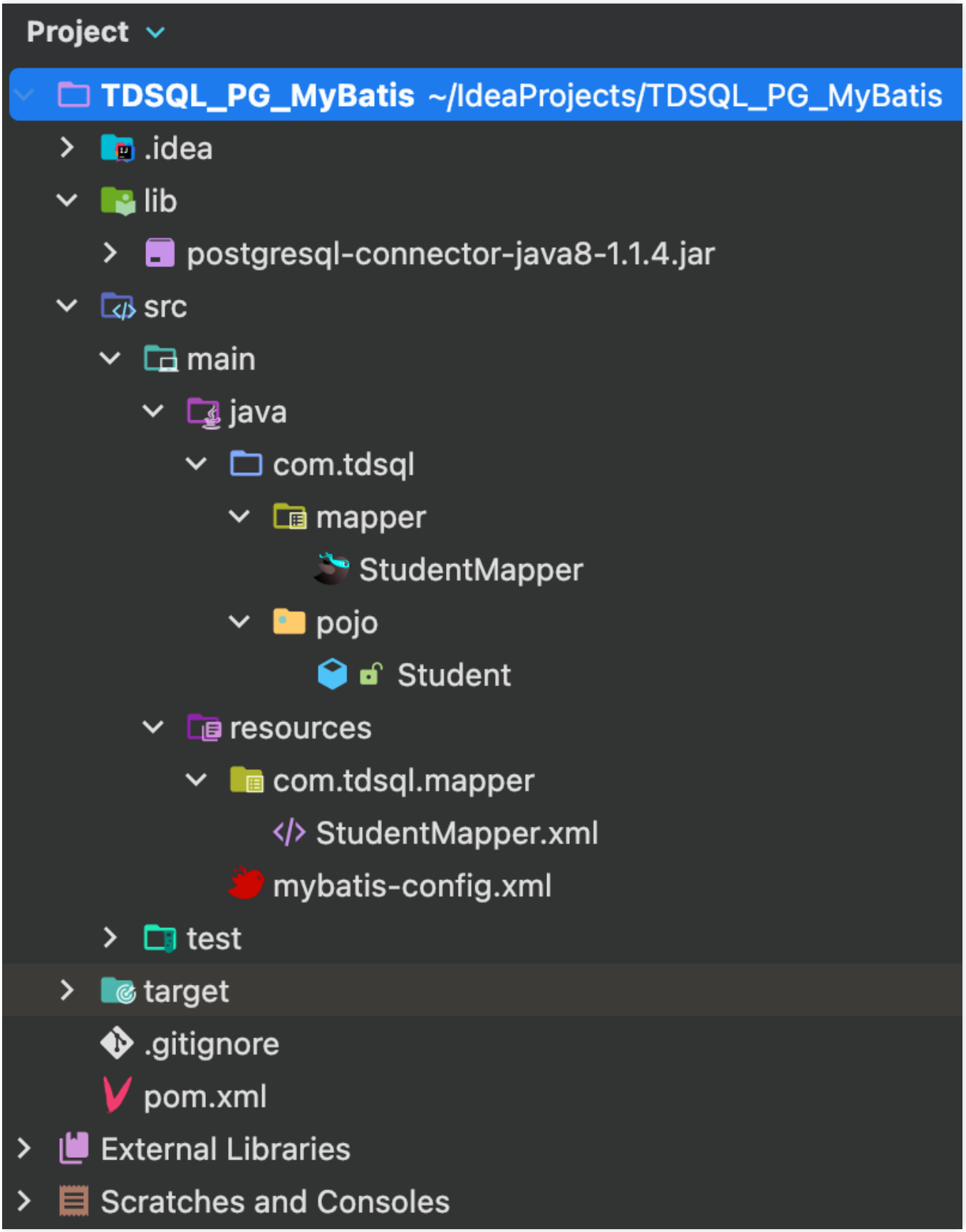
运行结果如下：

```
------insert data------
insert 3 rows
------lookup data------
Student [id=1, name=Alex]
Student [id=2, name=Bob]
Student [id=3, name=John]
```

```
------update data------
update row id == 1
------lookup data------
Student [id=1, name=Alice]
Student [id=2, name=Bob]
Student [id=3, name=John]
------delete data------
delete row id == 2
------lookup data------
Student [id=1, name=Alice]
Student [id=3, name=John]
```

附：项目结构

至此，完成了 MyBatis 项目的配置和开发示例，项目的整体结构如下图所示：


Project
TDSQL_PG_MyBatis ~/IdeaProjects/TDSQL_PG_MyBatis
.idea
lib
postgresql-connector-java8-1.1.4.jar
src
main
java
com.tdsql
mapper
StudentMapper
pojo
Student
resources
com.tdsql.mapper
StudentMapper.xml
mybatis-config.xml
test
target
.gitignore
pom.xml
External Libraries
Scratches and Consoles

使用Spring Boot框架连接TDSQL PG

本节主要介绍如何使用 Spring Boot 框架连接 TDSQL PG 数据库，实现数据的增删查改等基本操作。

简介

Spring Boot 是一个基于 Spring 的套件，可以简化 Spring 应用的创建及部署。它提供了丰富的 Spring 模块化支持，可以帮助开发者更轻松快捷地构建出企业级应用。Spring Boot 通过自动配置功能，降低了复杂性，同时支持基于 JVM 的多种开源框架，可以缩短开发时间，使开发更加简单和高效。

引入和配置Spring Boot框架

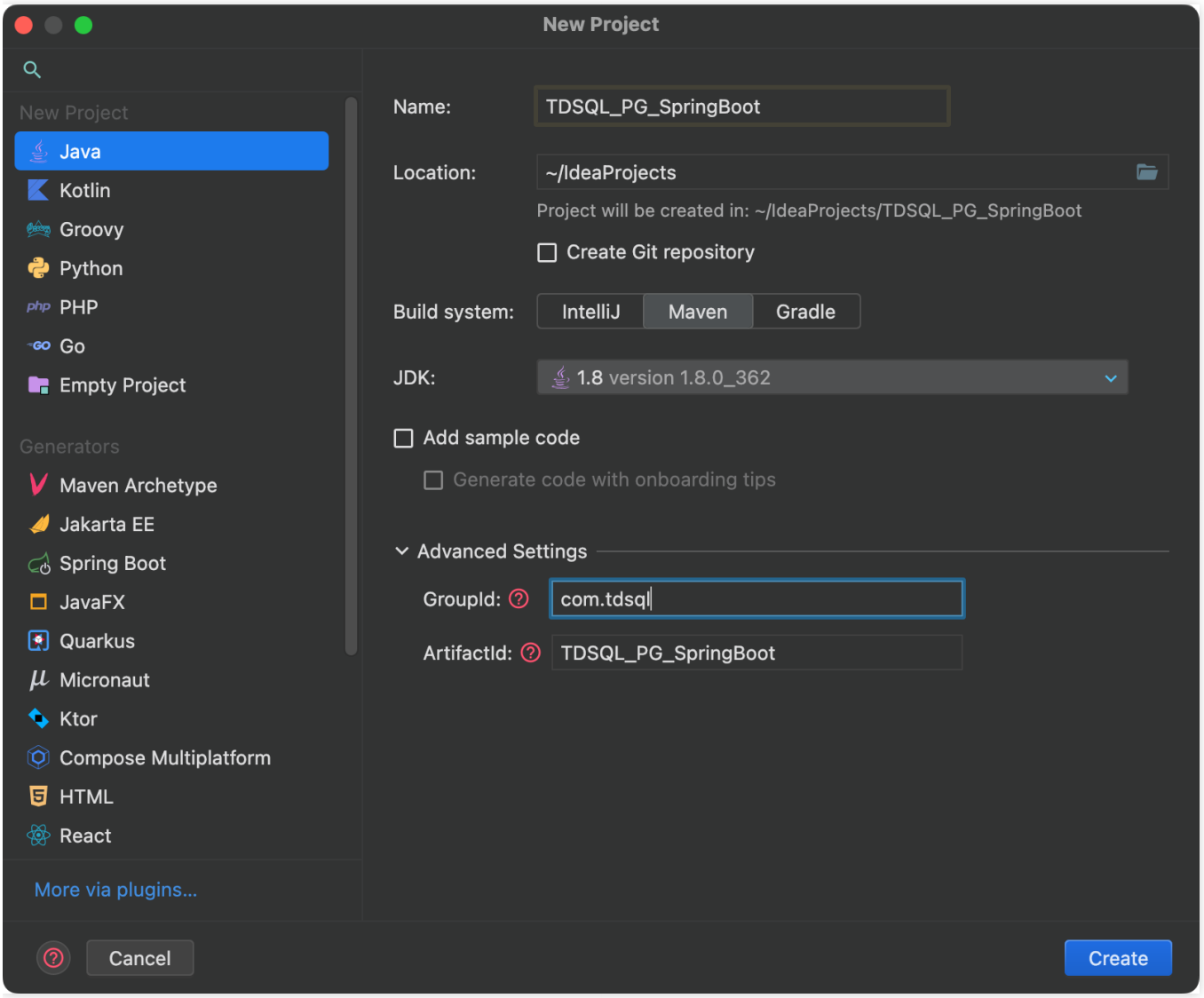
前置依赖

在配置之前需要保证你的机器上已安装对应版本的JDK，本例是 JDK 8（推荐配置）。
内核Oracle模式下建议使用自研驱动，请联系云技术支持工程师获取。 联系数据库管理员获取数据库相关信息：包括数据库IP、端口、实例名、用户、密码等。驱动名和连接URL如下：

- 驱动名 org.postgresql.Driver
- 连接URL（Oracle模式下需打开oracle_compile参数） jdbc:postgresql://host:port/database?oracle_compile=true

创建 Maven 项目

创建 Maven 项目，如下：



> 注意
>
> 本文档示例项目使用的 IDE 是 IntelliJ IDEA 2022.3 (Ultimate Edition)，也可根据喜好选择合适的 IDE。

配置 pom.xml 文件，导入项目依赖

配置文件 pom.xml 如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <!-- Configuring parent dependencies -->
    <parent>
        <groupId>org.springframework.boot</groupId>
```

```
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.7.18</version>
  <relativePath/>
</parent>

<groupId>com.tdsql</groupId>
<artifactId>TDSQL_PG_SpringBoot</artifactId>
<version>1.0-SNAPSHOT</version>

<properties>
  <java.version>1.8</java.version>
  <maven.compiler.source>8</maven.compiler.source>
  <maven.compiler.target>8</maven.compiler.target>
</properties>

<dependencies>
  <!-- Spring Boot -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
  <!-- TDSQL-PG JDBC Driver -->
  <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>1.1.4</version>
    <scope>system</scope>
    <systemPath>${project.basedir}/lib/postgresql-connector-java8-1.1.4.jar</systemPath>
  </dependency>
</dependencies>

<!-- Configuring Maven Plugin -->
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

```
</project>
```

1. 通过 <parent> 配置父项依赖：配置父项目依赖 spring-boot-starter-parent ，引入Spring Boot的预置配置。
2. 通过 <dependencies> 定义项目所依赖的组件，主要包括：
   - spring-boot-starter ：Spring Boot 的核心启动器。
   - spring-boot-starter-jdbc ：Spring JDBC 依赖包。
   - spring-boot-starter-test ：Spring Boot 单元测试依赖包。
   - spring-boot填写好后会从公共库自动下载。
   - 下载 TDSQL PG JDBC 驱动包，并在 TDSQL_PG_SpringBoot 目录下新建 lib 目录，将驱动包拷贝到 lib 目录下。
3. 配置 Maven 插件 spring-boot-maven-plugin ，配合 spring-boot-starter-parent 可以把Spring Boot应用打包成JAR来直接运行。

> 注意
>
> 1.TDSQL PG JDBC驱动包请联系云技术支持工程师获取
>
> 2.本文所有示例需在启用Oracle兼容内核的情况下执行

配置 Spring Boot

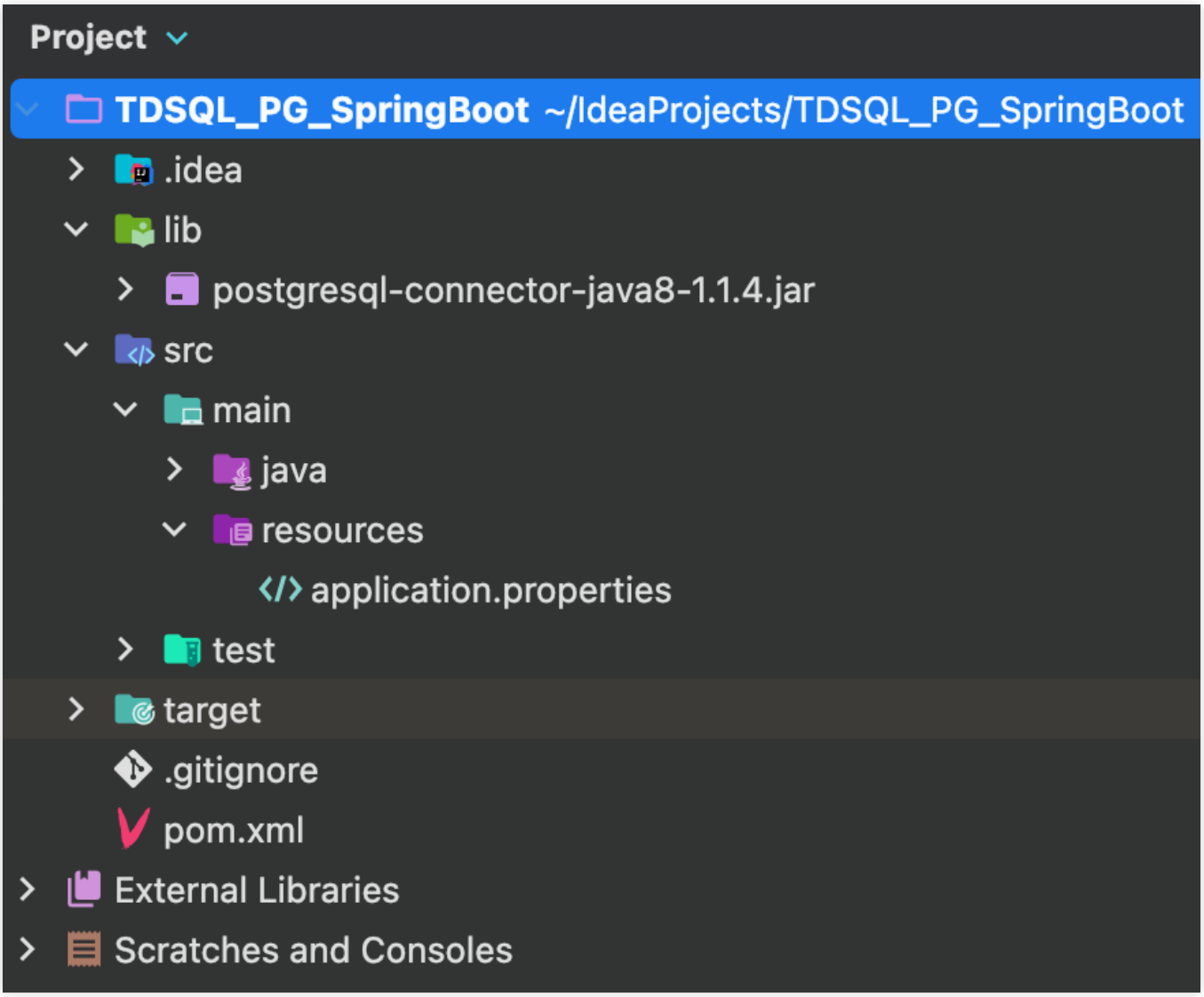
在 TDSQL_PG_SpringBoot/src/main/resources 目录下新建 application.properties 文件，用于配置 Spring Boot， application.properties 配置如下：
配置JDBC连接信息：host、port、database、user、password 需要替换为数据库管理员提供的数据库相关信息。

```
# Configure JDBC connection information: host, port, database, user, and password
# need to be replaced with the database-related information provided by the database administrator.
spring.datasource.driverClassName=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://host:port/database?oracle_compile=true
spring.datasource.username=user
spring.datasource.password=password
# Set the log printing level to WARN
logging.level.root=WARN
```

其中，需要配置 JDBC 连接信息：

- spring.datasource.driverClassName ：指定驱动包名为 org.postgresql.Driver 。
- spring.datasource.url ：指定连接URL，包括数据库地址、端口、数据库名、连接属性等信息。
- spring.datasource.username ：指定用户名。
- spring.datasource.password ：指定相应用户的密码。

至此，已完成 Spring Boot 项目的基本配置，完整的项目目录结构如下图所示：



开发示例

添加数据表对应的实体类

首先，开发使用的数据表为 STUDENT 表（ CREATE TABLE STUDENT(ID INT PRIMARY KEY, NAME VARCHAR(50)) ），
对应地，添加实体类 Student ，先在 TDSQL_PG_SpringBoot/src/main/java/ 下创建 Package: com.tdsql.pojo ，在 TDSQL_PG_SpringBoot/src/main/java/com/tdsql/pojo 下新建 Student.java 文件：

```
package com.tdsql.pojo;

public class Student {
    private int id;
    private String name;

    public Student() {
```

```
    }

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Student [id=" + id + ", name=" + name + "]";
    }
}
```

定义数据库的增删查改操作

其次，在 TDSQL_PG_SpringBoot/src/main/java/ 下创建Package com.tdsql.dao ，在 TDSQL_PG_SpringBoot/src/main/java/com/tdsql/dao 下新建 StudentDao.java 文件，定义数据库的增删查改操作，如下：

```
package com.tdsql.dao;

import com.tdsql.pojo.Student;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

@Repository
public class StudentDao {
  @Autowired
  private JdbcTemplate jdbcTemplate;

  // Insert
```

```
    public Integer insertStudent(Student student) {
      return jdbcTemplate.update(
            "insert into STUDENT values(?, ?)",
            new Object[]{student.getId(), student.getName()});
    }

    // select
    public List<Student> selectAllStudent() {
      return jdbcTemplate.query(
            "select * from STUDENT order by ID",
            new BeanPropertyRowMapper<>(Student.class));
    }

    // update
    public Integer updateStudentById(Student student) {
      return jdbcTemplate.update(
            "update STUDENT set NAME = ? where ID = ?",
            new Object[]{student.getName(), student.getId()});
    }

    // delete
    public Integer deleteStudentById(Integer id) {
      return jdbcTemplate.update("delete from STUDENT where ID = ?", id);
    }
  }
```

StudentDao 类通过 JdbcTemplate 对象执行 SQL 语句，实现了对 STUDENT 表的插入、查询、更新和删除操作。

构建启动类

然后，在 TDSQL_PG_SpringBoot/src/main/java/com/tdsql 下新建 Application.java 文件，用于启动Spring Boot应用程序，如下：

```
  package com.tdsql;

  import org.springframework.boot.SpringApplication;
  import org.springframework.boot.autoconfigure.SpringBootApplication;

  @SpringBootApplication
  public class Application {

     public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
     }
  }
```

测试

最后，在 TDSQL_PG_SpringBoot/src/test/java/com/tdsql 下添加测试类 TestStudent ，展示了基本的增删查改功能，如

下：

```
package com.tdsql;

import com.tdsql.dao.StudentDao;
import com.tdsql.pojo.Student;
import java.util.List;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
public class TestStudent {
    @Autowired
    private StudentDao studentDao;

    @Test
    void contextLoads() {
        System.out.println("------Insert Data------");
        studentDao.insertStudent(new Student(1, "Alex"));
        studentDao.insertStudent(new Student(2, "Bob"));
        studentDao.insertStudent(new Student(3, "John"));
        System.out.println("Insert 3 pieces of data");

        System.out.println("------Select data------");
        List<Student> students1 = studentDao.selectAllStudent();
        for (Student student: students1) {
            System.out.println(student);
        }

        System.out.println("------Update data------");
        studentDao.updateStudentById(new Student(1, "Alice"));
        System.out.println("Update the data id 1");

        System.out.println("------Select data------");
        List<Student> students2 = studentDao.selectAllStudent();
        for (Student student: students2) {
            System.out.println(student);
        }

        System.out.println("------Delete data------");
        studentDao.deleteStudentById(2);
        System.out.println("Delete the data id 2");

        System.out.println("------Select data------");
        List<Student> students3 = studentDao.selectAllStudent();
        for (Student student: students3) {
            System.out.println(student);
        }
```

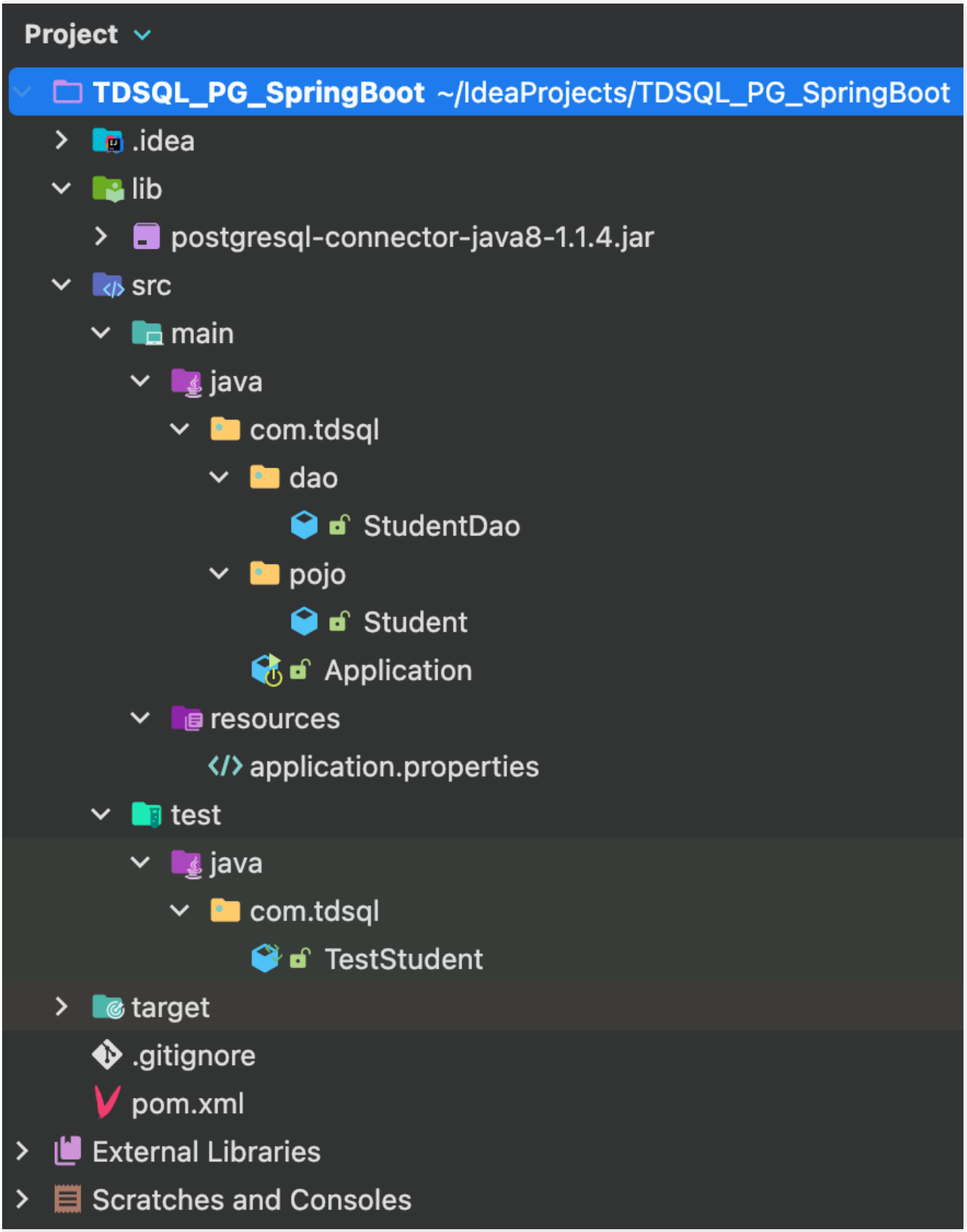
```
    }
}
```

运行结果如下：

```
------Insert Data------
Insert 3 pieces of data
------Select data------
Student [id=1, name=Alex]
Student [id=2, name=Bob]
Student [id=3, name=John]
------Update data------
Update the data id 1
------Select data------
Student [id=1, name=Alice]
Student [id=2, name=Bob]
Student [id=3, name=John]
------Delete data------
Delete the data id 2
------Select data------
Student [id=1, name=Alice]
Student [id=3, name=John]
```

附：项目结构

至此，完成了 Spring Boot 项目的配置和开发示例，项目的整体结构如下图所示：


Project
TDSQL_PG_SpringBoot ~/IdeaProjects/TDSQL_PG_SpringBoot
.idea
lib
postgresql-connector-java8-1.1.4.jar
src
main
java
com.tdsql
dao
StudentDao
pojo
Student
Application
resources
application.properties
test
java
com.tdsql
TestStudent
target
.gitignore
pom.xml
External Libraries
Scratches and Consoles

使用C/C++连接TDSQL PG
使用ODBC连接TDSQL PG

本文介绍如何使用ODBC示例程序连接及操作数据库。

获取数据库连接信息

联系数据库管理员获取数据库连接命令：

```
psql -h [host] -p [port] -U [username] -d [dbname]
```

参数说明：

- -h 数据库IP，通常是CN的IP
- -p 数据库端口
- -U 数据库用户名称
- -d 需要访问的数据库名称

安装驱动依赖

说明，操作系统上使用ODBC驱动依赖于UnixODBC管理，需要先在系统中安装UnixODBC。

UnixODBC安装

假设安装路径为UNIXODBC_PATH。

> 注意
>
> UnixODBC版本必须为2.3.12，否则会导致无法连接，可参考源码包安装。

参考：
1.下载unixODBC

```
unixODBC官网下载地址：https://www.unixodbc.org/unixODBC-2.3.12.tar.gz
```

2.安装
2.1将tar包上传到服务器，执行解压操作：

```
tar -xvf unixODBC*.tar.gz
```

2.2编译（进入到解压后的目录，依次执行以下命令）

```
./configure --prefix=$UNIXODBC_PATH
make
make install
```

2.3配置环境变量，编辑~/.bashrc文件。

```
export PATH=$UNIXODBC_PATH/bin:$PATH
export LD_LIBRARY_PATH=$UNIXODBC_PATH/lib:$LD_LIBRARY_PATH
```

2.4 环境变量生效。

```
source ~/.bashrc
```

2.5 测试。

```
odbcinst -j
    unixODBC 2.3.12
    DRIVERS.............: /etc/odbcinst.ini
    SYSTEM DATA SOURCES: /etc/odbc.ini
    FILE DATA SOURCES..: /etc/ODBCDataSources
    USER DATA SOURCES..: /etc/odbc.ini
    SQLULEN Size.......: 8
    SQLLEN Size........: 8
    SQLSETPOSIROW Size.: 8
```

ODBC驱动安装和配置

参考：

1. 下载驱动：联系云平台技术支持工程师获取软件包。
2. 安装：
   2.1 将tar包上传到服务器解压

```
tar -xvf tdsql_postgresql_odbc_*.tar.gz
```

解压TDSQL ODBC驱动包，假设解压路径为 $PG_ODBC_PATH

2.2 配置环境变量，编辑~/.bashrc文件。

```
export LD_LIBRARY_PATH=$PG_ODBC_PATH:$LD_LIBRARY_PATH
```

2.3 环境变量生效

```
source ~/.bashrc
```

3. 手动配置ODBC数据源

3.1 通过odbcinst -j命令获取/etc/odbcinst.ini的实际路径，编辑/etc/odbcinst.ini，输入如下内容：

```
[TDSQL_PG]
Description     = ODBC for TDSQL PostgreSQL
Driver          = $PG_ODBC_PATH/psqlodbcw.so
FileUsage       = 1
```

> 注意
>
> /etc/odbcinst.ini是通过odbcinst -j命令获取的路径，以实际查询配置文件为准。

3.2 编辑/etc/odbc.ini，输入如下内容：

```
[PG_CON]
Driver=TDSQL_PG
Servername=
Port=
Username=
Password=
Database=
```

参数说明：

- Driver 驱动名称，在文件/etc/odbcinst.ini中声明，比如上述文件中的TDSQL_PG。
- Servername 服务器名称/IP
- Port 数据库端口
- Username 数据库用户
- Password 用户密码
- Database 需要访问的数据库名称

  > 注意
  >
  > 请按照实际情况进行字段值的修改，驱动动态库路径依据实际路径填写。

校验驱动库已经包含依赖：

```
ldd $PG_ODBC_PATH/psqlodbcw.so
```

链接不出现not found则为成功。

4. 测试连接

可以使用unixODBC自带的isql工具进行检测，执行：

```
isql PG_CON -v

   +--------------------------------------+
   | Connected!
   |
   | sql-statement
   | help [tablename]
   | quit
   |
   +--------------------------------------+
SQL>
```

使用OCI连接TDSQL PG

本文介绍如何使用OCI示例程序连接及操作数据库

> 注意
>
> 该 OCI 驱动暂未对内核 Oracle 模式的语法进行适配

获取数据库连接信息

联系数据库管理员获取数据库连接信息，可以用下面命令行工具测试连接信息：

```
psql -h <host> -p <port> -U <username> -d <database>
```

参数说明：

- -h 指定数据库服务的主机信息，带有一个host参数，host参数可以是IP地址或者hostname，通常是指定CN节点。
- -p 指定数据库服务的端口信息，带有一个port参数，port参数是一个2字节无符号整数。
- -U 指定数据库服务的用户信息，带有一个username参数，username参数是用于连接数据库的用户名。
- -d 指定要访问的数据库名称，带有一个database参数，database参数是数据库中存在的数据库名。
  以用tbase用户连接本地数据库服务的postgres数据库为例，应该执行以下命令就可以连接数据库：

```
psql -h 127.0.0.1 -p 5432 -U tbase -d postgres
```

OCI驱动获取

OCI驱动目前主要支持RedHat x86和KylinV10 ARM两个平台，联系云平台技术支持工程师获取软件包。
以Linux的X86平台为例，获取的驱动安装包名称为tdsql_psqloci_tlinux4_x86_64.tar.gz，可以将此安装包放在用户家目录下，可以用下面命令确认：

```
ls ~/tdsql_psqloci*
```

ARM平台除了文件名不同外，其它都类似，这里不再说明。

OCI驱动安装

设置环境变量 PG_OCI_PATH 并创建指定目录，用于安装驱动文件，具体命令如下所示：

```
export PG_OCI_PATH=/opt/ocidriver
mkdir -p ${PG_OCI_PATH}
tar -xvf ~/tdsql_psqloci_tlinux4_x86_64.tar.gz --strip-components=1 -C ${PG_OCI_PATH}
```

解压TDSQL OCI驱动包，解压后文件会保存在 ${PG_OCI_PATH} 目录中，各文件和目录说明如下：

- include 依赖的头文件，OCI函数声明。
- lib 依赖库，OCI函数实现。
- oci_test.c demo，提供参考。
- pg_service.conf 数据库连接信息配置（数据源设置）。
- README demo操作说明。
- run.sh demo 执行脚本。

OCI驱动配置

1. 配置连接文件，编辑 ${PG_OCI_PATH}/pg_service.conf ：
   其中host、port和database为前面说明的联系数据库管理员获取的数据库连接信息，最好先通过psql命令验证正确以后再进行配置。

```
[db_test] # Data Source Name
# Database server IP
hostaddr=<host>
# Database server port
port=<port>
# The database name to connect
dbname=<database>
```

> 注意
>
> OCI应用程序连接数据库进行dbname设置时，使用的是 pg_service.conf 中的数据源名称而不是实际访问的数据库名称。
> 在OCI应用程序中配置数据源、用户名和密码。在 pg_service.conf 配置数据库IP、端口、实际访问的数据库名。

下面是 pg_service.conf 文件的具体示例：

```
[db_test] # Data Source Name
# Database server IP
hostaddr=127.0.0.1
# Database server port
port=5432
```

```
# The database name to connect
dbname=postgres
```

2. 配置环境变量，编辑~/.bashrc文件，追加如下内容：

```
export LD_LIBRARY_PATH=${PG_OCI_PATH}/lib:${LD_LIBRARY_PATH}
export PGSERVICEFILE=${PG_OCI_PATH}/pg_service.conf
```

> 说明
>
> PGSERVICEFILE：pg_services.conf文件环境变量，依据实际路径填写。

3. 环境变量生效

```
source ~/.bashrc
```

4. 校验驱动库依赖完整性：

```
ldd $PG_OCI_PATH/lib/liboci.so
```

链接不出现not found则为成功。

OCI测试示例

示例程序： ${PG_OCI_PATH}/oci_test.c ，请用下面代码覆盖整个文件，并修改用户名、密码、数据源信息即可。

```
#include "oci.h"
#include "oratypes.h"
#include "xa.h"
#include <unistd.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/signal.h>
#include <unistd.h>
#include <pthread.h>

#define MAX_SIZE 1024
#define MAX_STR_LENTH 256
#define MAX_COUNT 3
```

```
typedef struct __OCI_GLOBAL_CONTEXT_TYPE
{
   OCIEnv*     envhp;
   OCIServer*  srvhp;
   OCISvcCtx*  svchp;
   OCIError*   errhp;
   OCISession* authp;
   OCIStmt*    stmthp;
} OCI_GLOBAL_CONTEXT;

// Prepare test stage
sword oci_alloc_connect(bool useLogon, OCI_GLOBAL_CONTEXT* ctxptr, char* uname, char* passwd, c
har* dbname);
sword oci_free_disconnect(bool useLogon, OCI_GLOBAL_CONTEXT* ctxptr);
void oci_checkerr(OCIError* errhp, sword status);
sword oci_exec_sample_sql(OCI_GLOBAL_CONTEXT* ctxptr, char* sql, ub4 mode);

sword case0_create(OCI_GLOBAL_CONTEXT *ctxptr);
sword case1_select(OCI_GLOBAL_CONTEXT *ctxptr);
sword case2_update(OCI_GLOBAL_CONTEXT *ctxptr);
sword case3_insert(OCI_GLOBAL_CONTEXT *ctxptr);
sword case4_delete(OCI_GLOBAL_CONTEXT *ctxptr);
sword case5_vacuum(OCI_GLOBAL_CONTEXT *ctxptr);

// Bind parameters and define result set tests based on location.
sword fetch_by_define(void* ptr, char* sql);
void test();

sword oci_free_handle(OCI_GLOBAL_CONTEXT* ctxptr, char* sql, ub4 mode);
sword oci_alloc_handle(OCI_GLOBAL_CONTEXT* ctxptr, char* sql, ub4 mode);

/*
The functions to be tested：OCIInitialize()、OCIEnvInit()、OCIAttrSet()、OCIStmtPrepare()、OCIStmtEx
ecute()、OCITransCommit()、OCIDescriptorAlloc()、OCIDefineByPos()、OCIStmtFetch()、OCIErrorGet()
   1、OCIInitialize() initializes the OCI application environment.
   2、OCIEnvInit() initializes the environment handle.
   3、OCIDescriptorAlloc() allocates handles.
   4、OCIAttrSet() sets the session handle username and password.
   5、OCIAttrSet() sets the environment handle attributes.
   6、Create and start two sessions for one user.
   7、Disconnect one session.
   8、OCIErrorGet() retrieves error messages.
   9、Create a table and perform the following operations:
     a. OCIStmtPrepare() prepares the SQL statement.
     b. OCIStmtExecute() executes the SQL statement.
     c. OCITransCommit() commits the transaction.
     d. OCIDefineByPos() defines output variables.
```

```
    e. OCIStmtFetch() fetches data.
  10、Conclude by releasing resources, disconnecting the database connection, and freeing handles.
*/

/* It is necessary to modify the username, password, and the data source name configured in pg_servic
e.conf;
 * none of these three variables can be left empty.
 */
char*      uname     = "tbase";
char*      passwd    = "1234@abcd";
char*      dbname    = "db_test";

int main(int argc, char* argv[])
{
  test();
  return 0;
}

void test()
{
  OCI_GLOBAL_CONTEXT ctxptr = {NULL};
  sword              err    = OCI_SUCCESS;

  do {
    err = oci_alloc_connect(true, &ctxptr, uname, passwd, dbname);
    if (err != OCI_SUCCESS) {
      break;
    }

    err = case0_create(&ctxptr);
    if (err != OCI_SUCCESS) {
      break;
    }

    err = case3_insert(&ctxptr);
    if (err != OCI_SUCCESS) {
      break;
    }

    err = case1_select(&ctxptr);
    if (err != OCI_SUCCESS) {
      break;
    }

    oci_alloc_handle(&ctxptr, NULL, OCI_COMMIT_ON_SUCCESS);
    err = case2_update(&ctxptr);
    if (err != OCI_SUCCESS) {
```

```
            break;
        }
        err = case5_vacuum(&ctxptr);
        if (err != OCI_SUCCESS) {
            break;
        }

        err = case4_delete(&ctxptr);
        if (err != OCI_SUCCESS) {
            break;
        }
        oci_free_handle(&ctxptr, NULL, OCI_COMMIT_ON_SUCCESS);

        oci_free_disconnect(false, &ctxptr);

    } while (0);
}

sword case0_create(OCI_GLOBAL_CONTEXT *ctxptr)
{
    char*   sql = NULL;
    sword   err = OCI_ERROR;

    do {
        // Drop and create table
        sql = "DROP TABLE IF EXISTS case1_bindbypos_define;";
        err = oci_exec_sample_sql(ctxptr, sql, OCI_COMMIT_ON_SUCCESS);

        sql = "CREATE TABLE case1_bindbypos_define(id INT, name CHARACTER VARYING(255), created C
HARACTER VARYING(12), attr TEXT);";
        err = oci_exec_sample_sql(ctxptr, sql, OCI_COMMIT_ON_SUCCESS);

    } while (0);
    return err;
}

sword case1_select(OCI_GLOBAL_CONTEXT *ctxptr)
{
    char*   sql = NULL;
    sword   err = OCI_ERROR;

    do {

        // Get the result set
        sql = "SELECT * FROM case1_bindbypos_define WHERE id >10";
        err = fetch_by_define((void*)ctxptr, sql);
        if (err != OCI_SUCCESS) {
```

```
            break;
        }

    } while (0);
    return err;
}

sword case2_update(OCI_GLOBAL_CONTEXT *ctxptr)
{
    char*   sql = NULL;
    sword   err = OCI_ERROR;

    do {

        // Update the data
        sql = "UPDATE case1_bindbypos_define SET name='YYYYYYYYY' WHERE id=100;";
        err = oci_exec_sample_sql(ctxptr, sql, OCI_COMMIT_ON_SUCCESS);

    } while (0);
    return err;
}

sword case3_insert(OCI_GLOBAL_CONTEXT *ctxptr)
{
    char*   sql = NULL;
    sword   err = OCI_ERROR;

    do {
        // Insert data
        sql = "INSERT INTO case1_bindbypos_define VALUES (generate_series(1, 10000), 'AAAAAAAAAA
AAAA', 'BBBBBBBBB', 'CCCCCCCCCCCCCC');";
        err = oci_exec_sample_sql(ctxptr, sql, OCI_COMMIT_ON_SUCCESS);

    } while (0);
    return err;
}

sword case4_delete(OCI_GLOBAL_CONTEXT *ctxptr)
{
    char*   sql = NULL;
    sword   err = OCI_ERROR;

    do {

        // Delete data
        sql = "DELETE FROM case1_bindbypos_define WHERE id < 10001;";
        err = oci_exec_sample_sql(ctxptr, sql, OCI_COMMIT_ON_SUCCESS);
```

```
    } while (0);
    return err;
}

sword case5_vacuum(OCI_GLOBAL_CONTEXT *ctxptr)
{
    char*   sql = NULL;
    sword   err = OCI_ERROR;

    do {
        sql = "VACUUM FULL case1_bindbypos_define";
        err = oci_exec_sample_sql(ctxptr, sql, OCI_COMMIT_ON_SUCCESS);

    } while (0);
    return err;
}

sword fetch_by_define(void* ptr, char* sql)
{
#define max_one_size 5
    sword               status  = OCI_SUCCESS;
    OCI_GLOBAL_CONTEXT* ctxptr  = (OCI_GLOBAL_CONTEXT*)ptr;
    OCIDefine*          dnfp[4] = {NULL};

    int ret = -1;
    int j = 0;

    do {
        int  id[max_one_size]       = {0};
        char name[max_one_size][15] = {{'\0'}};
        char date[max_one_size][12] = {{'\0'}};
        char attr[max_one_size][20] = {{'\0'}};

        // 1 Allocate an execution statement
        if (ctxptr->stmthp == NULL) {
            if ((status = OCIHandleAlloc((dvoid*)ctxptr->envhp, (dvoid**)&ctxptr->stmthp, (ub4)OCI_HTYP
E_STMT,
                                         (CONST size_t)0, (dvoid**)0)) != OCI_SUCCESS) {
                oci_checkerr(ctxptr->errhp, status);
                break;
            }
        }

        // 2 Prepare query sql
        status = OCIStmtPrepare(ctxptr->stmthp, ctxptr->errhp, (text*)sql, (ub4)strlen((char*)sql), OCI_NT
V_SYNTAX,
```

```
                        OCI_DEFAULT);
    if (status != OCI_SUCCESS) {
        oci_checkerr(ctxptr->errhp, status);
        break;
    }
    // 3. Bind the result set memory to the corresponding columns based on the query position.
    if (OCIDefineByPos(ctxptr->stmthp, &dnfp[0], ctxptr->errhp, 1, (void*)&id[0], sizeof(int), SQLT_INT,
(void*)0,
                       (ub2*)0, (ub2*)0, OCI_DEFAULT) != OCI_SUCCESS) {
        break;
    }
    printf("SUCCESS: OCIDefineByPos ()\n");

    if (OCIDefineByPos(ctxptr->stmthp, &dnfp[1], ctxptr->errhp, 2, (void*)name[0], 15, SQLT_STR, (voi
d*)0, (ub2*)0,
                       (ub2*)0, OCI_DEFAULT) != OCI_SUCCESS) {
        break;
    }

    if (OCIDefineByPos(ctxptr->stmthp, &dnfp[2], ctxptr->errhp, 3, (void*)date[0], 12, SQLT_STR, (void
*)0, (ub2*)0,
                       (ub2*)0, OCI_DEFAULT) != OCI_SUCCESS) {
        break;
    }

    if (OCIDefineByPos(ctxptr->stmthp, &dnfp[3], ctxptr->errhp, 4, (void*)attr[0], 20, SQLT_STR, (void
*)0, (ub2*)0,
                       (ub2*)0, OCI_DEFAULT) != OCI_SUCCESS) {
        break;
    }
    // Since the entire column in the result set resides in a single block of memory, it is necessary to s
pecify the segmentation size.
    if (OCIDefineArrayOfStruct(dnfp[0], ctxptr->errhp, sizeof(int), 0, 0, 0) != OCI_SUCCESS) {
        break;
    }
    if (OCIDefineArrayOfStruct(dnfp[1], ctxptr->errhp, 15, 0, 0, 0) != OCI_SUCCESS) {
        break;
    }
    if (OCIDefineArrayOfStruct(dnfp[2], ctxptr->errhp, 12, 0, 0, 0) != OCI_SUCCESS) {
        break;
    }
    if (OCIDefineArrayOfStruct(dnfp[3], ctxptr->errhp, 20, 0, 0, 0) != OCI_SUCCESS) {
        break;
    }
    // Execute SQL
    status = OCIStmtExecute(ctxptr->svchp, ctxptr->stmthp, ctxptr->errhp, (ub4)max_one_size, (ub
4)0, NULL, NULL,
```

```
                        OCI_DEFAULT);
        if (status != OCI_SUCCESS) {
            oci_checkerr(ctxptr->errhp, status);
            break;
        }

        // Fetch data
        status = OCIStmtFetch(ctxptr->stmthp, ctxptr->errhp, (ub4)max_one_size, (ub4)OCI_FETCH_NEXT,
(ub4)OCI_DEFAULT);
        printf("SUCCESS: OCIStmtFetch() fetch data\n");
        if (status == OCI_SUCCESS) {
            for (j = 0; j < max_one_size; j++) {
                printf("%d %d %s %s %s\n", status, id[j], name[j], date[j], attr[j]);
            }
        }
        else if (status == OCI_NO_DATA) {
            ub4 ret_val = 0, ret_size = 0;

            // After retrieving the attribute values, the variable values will be validated by printing them in
a loop.
            if ((OCIAttrGet((void*)ctxptr->stmthp, (ub4)OCI_HTYPE_STMT, (void*)&ret_val, (ub4*)&ret_size,
                    (ub4)OCI_ATTR_ROW_COUNT, ctxptr->errhp)) != OCI_SUCCESS) {
                printf("DCIAttrGet fail\n");
                break;
            }
            printf("SUCCESS: OCIAttrGet()\n");
            for (j = 0; j < ret_val; j++) {
                printf("%d %d %s %s %s\n", status, id[j], name[j], date[j], attr[j]);
            }
        }

        ret = 0;
    } while (0);
    // Free the handle of stmt
    if (ctxptr->stmthp != NULL) {
        status = OCIHandleFree((dvoid*)ctxptr->stmthp, (ub4)OCI_HTYPE_STMT);
        if (status != OCI_SUCCESS) {
            printf("OCIHandleFree OCI_HTYPE_STMT failed\n");
            return OCI_ERROR;
        }
        ctxptr->stmthp = NULL;
    }

    if (ret == -1) {
        return OCI_ERROR;
    }
    return OCI_SUCCESS;
```

```
}

sword oci_alloc_connect(bool useLogon, OCI_GLOBAL_CONTEXT* ctxptr, char* uname, char* passwd, c
har* dbname)
{
   sword err = OCI_SUCCESS;
   do{
      // Initialization, which is part of the preparation process, does not yield actual operational results.
      err = OCIInitialize((ub4)OCI_DEFAULT, (dvoid*)0, (dvoid * (*)(dvoid*, size_t))0,
                  (dvoid * (*)(dvoid*, dvoid*, size_t))0, (void (*)(dvoid*, dvoid*))0);
      if (err != OCI_SUCCESS) {
         printf("FAILED: OCIInitialize()\n");
         break;
      }

      printf("SUCCESS: OCIInitialize() Initialize the OCI application environment.\n");

      // Applying for an env handle is part of the preparation process and does not yield any actual exe
cution results.
      err = OCIEnvInit((OCIEnv**)&ctxptr->envhp, OCI_DEFAULT, 0, NULL);
      if (err != OCI_SUCCESS) {
         printf("FAILED: OCIEnvInit()\n");
         break;
      }
      printf("SUCCESS: OCIEnvInit() Initialize the environment handle.\n");

      // Requesting an error handle is part of the preparation process and does not produce any actual
execution results.
      if (OCIHandleAlloc((dvoid*)ctxptr->envhp, (dvoid**)&ctxptr->errhp, (ub4)OCI_HTYPE_ERROR, (size
_t)0,
                  (dvoid**)0)) {
         printf("FAILED: OCIHandleAlloc() on ctxptr->errhp\n");
         break;
      }

      // Requesting an srv handle is part of the preparation process and does not yield any actual execu
tion results.
      if (OCIHandleAlloc((dvoid*)ctxptr->envhp, (dvoid**)&ctxptr->srvhp, (ub4)OCI_HTYPE_SERVER, (siz
e_t)0,
                  (dvoid**)0)) {
         printf("FAILED: OCIHandleAlloc() on ctxptr->srvhp\n");
         break;
      }

      // Requesting an svch handle is part of the preparation process and does not yield any actual exe
cution results.
      if (OCIHandleAlloc((dvoid*)ctxptr->envhp, (dvoid**)&ctxptr->svchp, (ub4)OCI_HTYPE_SVCCTX, (siz
```

```
e_t)0,
                    (dvoid**)0)) {
        printf("FAILED: OCIHandleAlloc() on ctxptr->svchp\n");
        break;
    }

    // Requesting an auth handle is part of the preparation process and does not yield any actual exe
cution results.
    if (OCIHandleAlloc((dvoid*)ctxptr->envhp, (dvoid**)&ctxptr->authp, (ub4)OCI_HTYPE_SESSION, (si
ze_t)0,
                    (dvoid**)0)) {
        printf("FAILED: OCIHandleAlloc() on ctxptr->authp\n");
        break;
    }
    printf("SUCCESS: OCIHandleAlloc() Allocate a handle.\n");

    // Request an execution statement.
    if (OCIHandleAlloc((dvoid*)ctxptr->envhp, (dvoid**)&ctxptr->stmthp, (ub4)OCI_HTYPE_STMT,
                        (CONST size_t)0, (dvoid**)0)) {
        printf("FAILED: OCIHandleAlloc() on ctxptr->stmthp\n");
        break;
    }

    // Non-test interface, which is an auxiliary interface.
    if (OCIServerAttach(ctxptr->srvhp, ctxptr->errhp, (text*)dbname, (sb4)strlen((char*)dbname),
                    (ub4)OCI_DEFAULT)) {
        printf("FAILED: OCIServerAttach()\n");
        break;
    }

    // Setting the srvh attribute to the svch handle is part of the preparation process and does not yie
ld any actual execution results.
    if (OCIAttrSet((dvoid*)ctxptr->svchp, (ub4)OCI_HTYPE_SVCCTX, (dvoid*)ctxptr->srvhp, (ub4)0,
                (ub4)OCI_ATTR_SERVER, ctxptr->errhp)) {
        printf("FAILED: OCIAttrSet() server attribute\n");
        break;
    }
    printf("SUCCESS: OCIAttrSet() Set the environment handle attributes.\n"); // Please note that durin
g the process of OCIHandleAlloc requesting other handles, they have already been saved to the enviro
nment handle. The process of OCIAttrSet saving the srvhp handle attribute to svchp is equivalent to si
multaneously saving it to env.

    // Setting the uname to the auth handle is part of the preparation process and does not yield any
actual execution results.
    if (OCIAttrSet((dvoid*)ctxptr->authp, (ub4)OCI_HTYPE_SESSION, (dvoid*)uname, (ub4)strlen((char
*)uname),
                (ub4)OCI_ATTR_USERNAME, ctxptr->errhp)) {
```

```
        printf("FAILED: OCIAttrSet() userid\n");
        break;
    }
    printf("SUCCESS: OCIAttrSet() Set the username for the session handle.\n");

    // Setting the passwd to the auth handle is part of the preparation process and does not yield any actual execution results.
    if (OCIAttrSet((dvoid*)ctxptr->authp, (ub4)OCI_HTYPE_SESSION, (dvoid*)passwd, (ub4)strlen((char*)passwd),
                (ub4)OCI_ATTR_PASSWORD, ctxptr->errhp)) {
        printf("FAILED: OCIAttrSet() passwd\n");
        break;
    }
    printf("SUCCESS: OCIAttrSet() Set the password for the session handle.\n");

    // Open a session; subsequent code contains database operations, which can prove that this interface is effective.
    if (OCISessionBegin((dvoid*)ctxptr->svchp, ctxptr->errhp, ctxptr->authp, (ub4)OCI_CRED_RDBMS,
                (ub4)OCI_DEFAULT)) {
        text sqlstate[6] = "", msg[256] = "";
        sb4  errcode = 0;
        OCIErrorGet(ctxptr->errhp, 1, sqlstate, &errcode, msg, 256, OCI_HTYPE_ERROR);
        printf("SUCCESS: OCIErrorGet() Read the error message.\n");
        printf("initialize failed: %d %s\n", errcode, msg);
        break;
    }
    printf("SUCCESS: Created and initiated a user session.\n");

    // Setting the auth to the svchp handle is part of the preparation process and does not yield any actual execution results.
    if (OCIAttrSet((dvoid*)ctxptr->svchp, (ub4)OCI_HTYPE_SVCCTX, (dvoid*)ctxptr->authp, (ub4)0,
                (ub4)OCI_ATTR_SESSION, ctxptr->errhp)) {
        printf("FAILED: OCIAttrSet() session\n");
        break;
    }

    return OCI_SUCCESS;
  } while (0);
  return OCI_ERROR;
}

void oci_checkerr(OCIError* errhp, sword status)
{
  text errbuf[512] = {'\0'};
  sb4  errcode;

  if (status == OCI_SUCCESS) {
```

```
        return;
    }

    switch (status) {
        case OCI_SUCCESS_WITH_INFO: {
            printf("Error - OCI_SUCCESS_WITH_INFO\n");
            OCIErrorGet((void*)errhp, (ub4)1, (text*)NULL, &errcode, errbuf, (ub4)sizeof(errbuf), (ub4)OCI_H
TYPE_ERROR);
            printf("Error %d - %s\n", errcode, errbuf);
            break;
        }
        case OCI_NEED_DATA: {
            printf("Error - OCI_NEED_DATA\n");
            break;
        }
        case OCI_NO_DATA: {
            printf("Error - OCI_NO_DATA\n");
            break;
        }
        case OCI_ERROR: {
            OCIErrorGet((void*)errhp, (ub4)1, (text*)NULL, &errcode, errbuf, (ub4)sizeof(errbuf), (ub4)OCI_H
TYPE_ERROR);
            printf("OCI_ERROR %d - %s\n", errcode, errbuf);
            break;
        }
        case OCI_INVALID_HANDLE: {
            printf("Error - OCI_INVALID_HANDLE\n");
            break;
        }
        case OCI_STILL_EXECUTING: {
            printf("Error - OCI_STILL_EXECUTING\n");
            break;
        }
        case OCI_CONTINUE: {
            printf("Error - OCI_CONTINUE\n");
            break;
        }
        default: {
            printf("Error - %d\n", status);
            break;
        }
    }
}
// Exit and release the corresponding handle resources.
sword oci_free_disconnect(bool useLogon, OCI_GLOBAL_CONTEXT* ctxptr)
{
    sword err = OCI_SUCCESS;
```

```
    printf("SUCCESS: Conclude by releasing resources, disconnecting from the database, and freeing the
handles.\n");
    // End the session.
    err = OCILogoff(ctxptr->svchp, ctxptr->errhp);
    if (err != OCI_SUCCESS) {
        printf("FAILED: OCILogoff()\n");
    }

    // Handle release.
    if (ctxptr->svchp != NULL) {
        err = OCIHandleFree((dvoid*)ctxptr->svchp, (ub4)OCI_HTYPE_SVCCTX);
        if (err != OCI_SUCCESS) {
            printf("FAILED: OCIHandleFree() on ctxptr->svchp\n");
        }
        ctxptr->svchp = NULL;
    }
    if (ctxptr->errhp) {
        err          = OCIHandleFree((dvoid*)ctxptr->errhp, (ub4)OCI_HTYPE_ERROR);
        ctxptr->errhp = NULL;
    }

    if (ctxptr->envhp) {
        err          = OCIHandleFree((dvoid*)ctxptr->envhp, (ub4)OCI_HTYPE_ENV);
        ctxptr->envhp = NULL;
    }

    return err;
}

sword oci_exec_sample_sql(OCI_GLOBAL_CONTEXT* ctxptr, char* sql, ub4 mode)
{
    sword err      = OCI_SUCCESS;
    bool  new_stmt = false;

    do {
        // SQL preparation, with no actual execution results.
        err = OCIStmtPrepare(ctxptr->stmthp, ctxptr->errhp, (const text*)sql, (ub4)strlen((char*)sql),
                             (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT);
        if (err != OCI_SUCCESS) {
            text sqlstate[6] = "", msg[256] = "";
            sb4  errcode = 0;

            OCIErrorGet(ctxptr->errhp, 1, sqlstate, &errcode, msg, 256, OCI_HTYPE_ERROR);
            printf("OCIStmtPrepare failed: %d %s\n", errcode, msg);
            break;
        }
```

```
        // SQL execution, not yet committed; it requires executing OCITransCommit to verify.
        err = OCIStmtExecute(ctxptr->svchp, ctxptr->stmthp, ctxptr->errhp, 1, 0, (OCISnapshot*)0, (OCISn
apshot*)0, mode);
        if (err != OCI_SUCCESS) {
            text sqlstate[6] = "", msg[256] = "";
            sb4  errcode = 0;

            OCIErrorGet(ctxptr->errhp, 1, sqlstate, &errcode, msg, 256, OCI_HTYPE_ERROR);
            printf("OCIStmtExecute failed: %d %s\n", errcode, msg);
            break;
        }

        // Commit transaction
        // Specifically, executing the SQL command: \d case1_bindbypos_define, allows verification that th
e table already exists.
        if (mode == OCI_DEFAULT) {
            err = OCITransCommit(ctxptr->svchp, ctxptr->errhp, (ub4)OCI_DEFAULT);
            if (err != OCI_SUCCESS) {
                printf("OCITransCommit failed\n");
                break;
            }
        }
    } while (0);

    if (new_stmt && ctxptr->stmthp != NULL) {
        // Release the handle, with no actual execution results.
        err = OCIHandleFree((dvoid*)ctxptr->stmthp, (ub4)OCI_HTYPE_STMT);
        if (err != OCI_SUCCESS) {
            printf("OCIHandleFree failed\n");
        }
        ctxptr->stmthp = NULL;
    }

    return err;
}

sword oci_alloc_handle(OCI_GLOBAL_CONTEXT* ctxptr, char* sql, ub4 mode)
{
    sword err      = OCI_SUCCESS;
    bool  new_stmt = false;

    do {
        // Request an execution statement.
        if (ctxptr->stmthp == NULL) {
            if (OCIHandleAlloc((dvoid*)ctxptr->envhp, (dvoid**)&ctxptr->stmthp, (ub4)OCI_HTYPE_STMT,
(CONST size_t)0,
```

```
                    (dvoid**)0)) {
            printf("FAILED: alloc statement handle\n");
            break;
        }
        else {
            new_stmt = true;
        }
    }
  } while (0);

  return err;
}

sword oci_free_handle(OCI_GLOBAL_CONTEXT* ctxptr, char* sql, ub4 mode)
{
  sword err      = OCI_SUCCESS;

  if ( ctxptr->stmthp != NULL) {
    // Free the handle, with no actual running results
    err = OCIHandleFree((dvoid*)ctxptr->stmthp, (ub4)OCI_HTYPE_STMT);
    if (err != OCI_SUCCESS) {
      printf("OCIHandleFree failed\n");
    }
    ctxptr->stmthp = NULL;
  }

  return err;
}
```

1. 修改示例代码中的 uname和passwd 两个变量的值为要连接环境的数据库用户名和密码， dbname 指定为 pg_service.conf中配置的数据源名称，不是要访问的数据库名称
2. 运行

```
cd $PG_OCI_PATH
./run.sh
```

结果

```
SUCCESS: OCIInitialize() Initialize the OCI application environment.
SUCCESS: OCIEnvInit() Initialize the environment handle.
SUCCESS: OCIHandleAlloc() Allocate a handle.
SUCCESS: OCIAttrSet() Set the environment handle attributes.
SUCCESS: OCIAttrSet() Set the username for the session handle.
SUCCESS: OCIAttrSet() Set the password for the session handle.
SUCCESS: Created and initiated a user session.
```

SUCCESS: OCIDefineByPos ()
SUCCESS: OCIStmtFetch() fetch data
0 12 AAAAAAAAAAAAAA BBBBBBBB CCCCCCCCCCCCCC
0 13 AAAAAAAAAAAAAA BBBBBBBB CCCCCCCCCCCCCC
0 15 AAAAAAAAAAAAAA BBBBBBBB CCCCCCCCCCCCCC
0 17 AAAAAAAAAAAAAA BBBBBBBB CCCCCCCCCCCCCC
0 19 AAAAAAAAAAAAAA BBBBBBBB CCCCCCCCCCCCCC
SUCCESS: Conclude by releasing resources, disconnecting from the database, and freeing the handles.

使用Pro*C连接TDSQL PG

本文介绍如何使用Pro*C示例程序连接及操作数据库。

> 注意
>
> 该 Pro*C 驱动暂未对内核 Oracle 模式的语法进行适配。

获取数据库连接信息

联系数据库管理员获取数据库连接命令，例如：

```
psql -h <host> -p <port> -U <username> -d <database>
```

参数说明：

- -h 数据库IP，通常是CN的IP
- -p 数据库端口
- -U 数据库用户
- -d 需要访问的数据库名称

以上参数将用于 Pro/C 连接配置。

安装驱动依赖

1.下载驱动：联系云平台技术支持工程师获取软件包。
2.安装

```
rpm -ivh tdsql_proc-1.1.0-1.x86_64.rpm
```

软件包会默认安装在/usr/local/tdsql_proc_pg目录下，目录中有以下文件：

- 预编译程序/usr/local/tdsql_proc_pg/bin/tdsql_proc
- 头文件/usr/local/tdsql_proc_pg/include
- 运行库/usr/local/tdsql_proc_pg/lib/libtdsqlproc.so 包含依赖库libssl.so.10和libcrypto.so.10
- 连接配置模板/usr/local/tdsql_proc_pg/tdsql_proc.cnf

3.设置环境变量，在操作用户目录添加环境变量，编辑~/.bashrc。

```
export PATH=/usr/local/tdsql_proc_pg/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/tdsql_proc_pg/lib:$LD_LIBRARY_PATH
```

4.环境变量生效

```
source ~/.bashrc
```

5.连接配置文件
修改 /usr/local/tdsql_proc_pg/tdsql_proc.cnf 文件，保存数据库连接信息。例如：

```
[db_test]
host=10.211.55.30
port=11345
db=tdsql
```

- db_test CONNECT命令中指定的服务名
- host tdsql数据库的ip地址
- port tdsql数据库的端口号
- db tdsql数据库的数据库名

编写Pro/C测试程序

示例程序： demo.pc

```
/*---------------------------------------------------------------------
 * demo.pc is an example Pro*C program
*---------------------------------------------------------------------
*/

#include <stdio.h>
#include <sqlca.h>
#include <string.h>
#include <setjmp.h>
#include <stdlib.h>
#include <sqlcpr.h>
#include <sqlca.h>

#define MAX_VAR_LEN 255
#define MAX_NAME_LEN 31

void sqlerror();

/* 变量声明 */
EXEC SQL BEGIN DECLARE SECTION;
char *username = "tbase";
char *password = "1234abcd";
```

```
char *service = "db_test";
EXEC SQL END DECLARE SECTION;
EXEC SQL WHENEVER SQLERROR DO sqlerror();

/* 宿主变量类型 */

void variable_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    int a;
    unsigned int b;
    char c = 'a';
    unsigned char d = 'b';
    char *e = "hello";
    char f[20] = "world";
    long g = 3;
    unsigned long h = 4;
    float i = 3.14;
    double j = 5.4321;
    varchar k[100];
    EXEC SQL END DECLARE SECTION;
}

/* 创建表 */
void create_table_demo_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char *sql1 = "DROP TABLE IF EXISTS demo_test";
    char *sql2 = "CREATE TABLE demo_test (id int, name varchar(100), address varchar(100))";
    EXEC SQL END DECLARE SECTION;

    printf("**********create_table_demo_test start**********\n");
    EXEC SQL PREPARE stat FROM :sql1;
    EXEC SQL EXECUTE stat;
    EXEC SQL PREPARE stat FROM :sql2;
    EXEC SQL EXECUTE stat;
    EXEC SQL COMMIT;
    printf("**********create_table_demo_test success**********\n");
    printf ("\n");
}

/* SELECT */
void select_table_demo_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    long id;
    int test_id = 2;
```

```
    varchar name[100];
    char address[100];
    short ind;
    EXEC SQL END DECLARE SECTION;

    printf("***********select_table_demo_test start***********\n");
    EXEC SQL SELECT id,name,address INTO :id, :name, :address:ind from demo_test where id = :test_id;
    name.arr[name.len] = '\0';
    printf("id:%d, name:%s, address:%s\n", id, name.arr, ind != -1 ? address : "NULL");
    printf("***********select_table_demo_test success***********\n");
    printf ("\n");
}

/* INSERT */
void insert_table_demo_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    int test_id = 2;
    char name[100] = "FirstMan";
    char address[100] = "EARTH";
    EXEC SQL END DECLARE SECTION;

    printf("***********insert_table_demo_test start***********\n");
    EXEC SQL INSERT INTO demo_test values(:test_id, :name, :address);
    EXEC SQL COMMIT;
    printf("insert demo_test values(%d,%s,%s)\n", test_id, name, address);
    printf("***********insert_table_demo_test success***********\n");
    printf ("\n");
}

/* UPDATE */
void update_table_demo_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    int test_id = 2;
    char address[100] = "AAAAA";
    short ind = -1;
    EXEC SQL END DECLARE SECTION;
    printf("***********update_table_demo_test start***********\n");
    EXEC SQL UPDATE demo_test SET address = :address where id = :test_id;
    EXEC SQL COMMIT;
    printf("update demo_test address to %s\n",address);
    printf("***********update_table_demo_test success***********\n");
    printf ("\n");
}

/* DELETE */
```

```
void delete_table_demo_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    int test_id = 2;
    EXEC SQL END DECLARE SECTION;
    printf("***********delete_table_demo_test start***********\n");
    EXEC SQL DELETE FROM demo_test where id = :test_id;
    EXEC SQL COMMIT;
    printf("***********delete_table_demo_test success***********\n");
    printf ("\n");
}

/* COMMIT已经包含在其他用例，ROLLBACK */
void rollback_demo_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    int test_id = 2;
    char name[100] = "ROLLBACK";
    char address[100] = "ROLLBACK";
    EXEC SQL END DECLARE SECTION;
    printf("***********rollback_demo_test start***********\n");
    printf("insert demo_test values(%d, %s, %s) rollback\n",test_id, name, address);
    EXEC SQL INSERT INTO demo_test values(:test_id, :name, :address);
    EXEC SQL ROLLBACK;
    printf("***********rollback_demo_test success***********\n");
    printf ("\n");
}

/* 使用游标，将结果集保存在变量中 */
void select_table_with_cursor_demo_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    int id;
    varchar name[50];
    char address[50];
    EXEC SQL END DECLARE SECTION;

printf("***********select_table_with_cursor_demo_test start***********\n");
    EXEC SQL DECLARE cur CURSOR for select id,name,address from demo_test;
    EXEC SQL OPEN cur;
    EXEC SQL FETCH cur INTO :id, :name, :address;
    name.arr[name.len] = '\0';
    printf("id:%d, name:%s, address:%s\n", id, name.arr, address);
    EXEC SQL CLOSE cur;
printf("***********select_table_with_cursor_demo_test success***********\n");
    printf ("\n");
}
```

```
/* 简单动态SQL */
void simple_dynamic_SQL_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char * sql = "insert into simple_dynamic_table(c1,c2) values (1, 2)";
    char *sql1 = "DROP TABLE IF EXISTS simple_dynamic_table";
    char *sql2 = "CREATE TABLE simple_dynamic_table (c1 int,c2 int)";
    EXEC SQL END DECLARE SECTION;

    EXEC SQL PREPARE stat FROM :sql1;
    EXEC SQL EXECUTE stat;
    EXEC SQL PREPARE stat FROM :sql2;
    EXEC SQL EXECUTE stat;

    printf("***********simple_dynamic_SQL_test start***********\n");
    printf("execute sql is %s\n", sql);
    EXEC SQL EXECUTE IMMEDIATE :sql;
    EXEC SQL COMMIT;
    printf("***********simple_dynamic_SQL_test success***********\n");
    printf ("\n");
}

/* ANSI动态SQL，使用游标将结果集保存在DESCRIPTOR中 */
void ANSI_dynamic_SQL_test()
{
    int i;
    EXEC SQL BEGIN DECLARE SECTION;
    char *sql1 = "DROP TABLE IF EXISTS ANSI_dynamic_table";
    char *sql2 = "CREATE TABLE ANSI_dynamic_table (c1 int, c2 int, c3 int)";
    int c1val = 1;
    int type = 3;
    int len = sizeof(int);
    int input_count, output_count, occurs;
    char name[31];
    int out_data;
    char out_name[MAX_NAME_LEN] = {0};
    short indi;
    char dyn_statement[1024] = "select c2,c3 from ANSI_dynamic_table where c1=:c1val";
    EXEC SQL END DECLARE SECTION;

    EXEC SQL PREPARE stat FROM :sql1;
    EXEC SQL EXECUTE stat;
    EXEC SQL PREPARE stat FROM :sql2;
    EXEC SQL EXECUTE stat;

    printf("***********ANSI_dynamic_SQL_test start***********\n");
```

```
EXEC SQL ALLOCATE DESCRIPTOR 'in_desc';
EXEC SQL ALLOCATE DESCRIPTOR 'out_desc';
EXEC SQL INSERT INTO ANSI_dynamic_table VALUES ( 1, 2, 3);
EXEC SQL INSERT INTO ANSI_dynamic_table VALUES ( 1, 2, 2);
EXEC SQL INSERT INTO ANSI_dynamic_table VALUES ( 2, 1, 1);
EXEC SQL INSERT INTO ANSI_dynamic_table(c1) VALUES (1);
EXEC SQL COMMIT;

/* 通过DESCRIPTOR给SQL传入变量（c1val） */
EXEC SQL PREPARE S from :dyn_statement;
EXEC SQL DECLARE C CURSOR for S;
EXEC SQL DESCRIBE INPUT S USING SQL DESCRIPTOR 'in_desc';
EXEC SQL GET DESCRIPTOR 'in_desc' :input_count = COUNT;
for (i=0; i < input_count; i++)
{
   occurs = i +1;    /* occurence is 1 based */
   EXEC SQL SET DESCRIPTOR 'in_desc' VALUE :occurs TYPE = :type, LENGTH = :len, DATA = :c1val;
}
EXEC SQL OPEN C USING DESCRIPTOR 'in_desc';

/* 通过DESCRIPTOR设定SQL输出变量属性 */
EXEC SQL DESCRIBE OUTPUT S USING DESCRIPTOR 'out_desc';
EXEC SQL GET DESCRIPTOR 'out_desc' :output_count = COUNT;
for (i = 0; i < output_count; i++)
{
   occurs = i + 1;
   EXEC SQL GET DESCRIPTOR 'out_desc' VALUE :occurs
   :out_name = NAME;
   EXEC SQL SET DESCRIPTOR 'out_desc' VALUE :occurs
   TYPE = :type, LENGTH = :len;
   printf("%-*.*s", 9,9, out_name);
}
printf("\n");
/* 通过DESCRIPTOR获取SQL输出变量值 */
EXEC SQL WHENEVER NOT FOUND GOTO end_select_loop;

for (;;)
{
  EXEC SQL FETCH C INTO DESCRIPTOR 'out_desc';
  for (i=0; i < output_count; i++)
{
   occurs = i + 1;
   EXEC SQL GET DESCRIPTOR 'out_desc' VALUE :occurs
   :out_data = DATA, :indi = INDICATOR;
   if (indi == -1)
      printf("%-*.*s", 9,9, "NULL");
   else
```

```
            printf("%-*d", 9, out_data);
        }
        printf ("\n");
        }
    end_select_loop:
        EXEC SQL CLOSE C;
        EXEC SQL DEALLOCATE DESCRIPTOR 'in_desc';
        EXEC SQL DEALLOCATE DESCRIPTOR 'out_desc';
        printf("***********ANSI_dynamic_SQL_test success***********\n");
        printf ("\n");
        return;
    }

    void sqlerror() {
        printf("Stop Error:\t%25i, %s\n", sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
        EXEC SQL WHENEVER SQLERROR CONTINUE;
        EXEC SQL ROLLBACK WORK RELEASE;
        exit(-1);
    }

    void main() {
        /* 建立连接 */
        EXEC SQL CONNECT :username identified by :password using :service;
        if (sqlca.sqlcode == 0) {
            printf("connect success\n");
            printf ("\n");
        } else {
            sqlerror();  // 触发详细错误输出
        }
        /* 测试 */
        variable_test();
        create_table_demo_test();
        insert_table_demo_test();
        select_table_demo_test();
        update_table_demo_test();
        select_table_demo_test();
        rollback_demo_test();
        select_table_with_cursor_demo_test();
        delete_table_demo_test();
        simple_dynamic_SQL_test();
        ANSI_dynamic_SQL_test();
        return;
    }
```

1.修改示例代码中 username 、 password 、 service 三个变量为用户名、密码、在 tdsql_proc.cnf 配置的数据源名称。

2.预编译

```
/usr/local/tdsql_proc_pg/bin/tdsql_proc iname=demo.pc oname=demo.c conn_config=/usr/local/tdsql_proc_pg/tdsql_proc.cnf
```

3.编译

```
gcc demo.c -o test -I /usr/local/tdsql_proc_pg/include -L /usr/local/tdsql_proc_pg/lib -ltdsqlproc
```

4.执行

```
./test
```

5.结果

```
connect success

**********create_table_demo_test start**********
**********create_table_demo_test success**********

**********insert_table_demo_test start**********
insert demo_test values(2,FirstMan,EARTH)
**********insert_table_demo_test success**********

**********select_table_demo_test start**********
id:2, name:FirstMan, address:EARTH
**********select_table_demo_test success**********

**********update_table_demo_test start**********
update demo_test address to AAAAA
**********update_table_demo_test success**********

**********select_table_demo_test start**********
id:2, name:FirstMan, address:AAAAA
**********select_table_demo_test success**********

**********rollback_demo_test start**********
insert demo_test values(2, ROLLBACK, ROLLBACK) rollback
**********rollback_demo_test success**********

**********select_table_with_cursor_demo_test start**********
id:2, name:FirstMan, address:AAAAA
**********select_table_with_cursor_demo_test success**********

**********delete_table_demo_test start**********
**********delete_table_demo_test success**********
```

```
***********simple_dynamic_SQL_test start***********
execute sql is insert into simple_dynamic_table(c1,c2) values (1, 2)
***********simple_dynamic_SQL_test success***********

***********ANSI_dynamic_SQL_test start***********
c2       c3
2        3
2        3
NULL     NULL
***********ANSI_dynamic_SQL_test success***********
```

创建数据库对象
创建数据库

本节介绍如何在TDSQL PG通过 SQL 命令创建数据库。

数据库简介

在TDSQL PG中，一个数据库是一个或多个模式的集合，而模式包含表、函数等。

前置条件

- 必须由超级用户或者拥有 CREATEDB 权限的用户来执行命令。

数据库命名规范

- 在TDSQL PG中，每个数据库的名称必须保证全局唯一。
- 数据库名长度限制小于等于 63 个字符。
- 避免使用保留关键字作为数据库名。
- 数据库名称包含大小写字母、数字、下划线，不要以数字，下划线开头。
- 为避免与系统表，系统视图名称混淆，数据库名称不建议以pg_、pgxc或sys_开头。

数据库创建建议

- 建议给数据库起一个意义明确的名字，使其能够反映其用途和内容。避免使用拼音缩写作为数据库名称。
- 建议使用 tbase 用户创建数据库和相关用户，并且只赋予必要的权限，以确保数据库的安全性和可控性。
- 建议在创建数据库时使用 UTF-8 字符集编码，以确保能够存储绝大多数字符。数据库支持的字符集编码相关内容请参见 数据库级字符集。

使用命令行创建数据库

请使用 CREATE DATABASE 语句创建数据库。默认情况下将创建 postgresql 类型的数据库，除此之外，通过指定 SQL MODE 参数为 oracle 则可以创建 oracle 兼容的数据库。
默认情况下，postgresql 类型的新数据库将通过克隆模板数据库 template1 来创建，oracle 类型的新数据库通过克隆模板数据库 template1_ora 来创建。同时，可以通过 TEMPLATE name 指定一个不同的模板。通过

TEMPLATE template0 或 TEMPLATE template0_ora 可以创建一个干净的 postgresql 或 oracle 数据库，它将只包含你的TDSQL PG所预定义的标准对象。

CREATE DATABASE 语法如下：

```
CREATE DATABASE name
    [ [ WITH ] [ OWNER [=] user_name ]
           [ TEMPLATE [=] template ]
           [ ENCODING [=] encoding ]
           [ LC_COLLATE [=] lc_collate ]
           [ LC_CTYPE [=] lc_ctype ]
           [ TABLESPACE [=] tablespace_name ]
           [ ALLOW_CONNECTIONS [=] allowconn ]
           [ CONNECTION LIMIT [=] connlimit ]
           [ IS_TEMPLATE [=] istemplate ]
           [SQL MODE [=] sqlmode]]
```

参数说明：

- name ：即将创建的数据库名称。
- user_name ：即将拥有该数据库的用户的角色名，默认为执行该命令的用户。
- encoding ：新数据库中使用的字符集编码,默认使用模板数据库的字符集。
- template ：新数据库的模板数据库。创建 postgresql 类型数据库，默认模板数据库为 template1；oracle 类型数据库的默认模板库为 template1_ora。通过指定模板库为 template0/template0_ora 可以创建一个原始的标准的 postgresql/oracle 类型数据库。
- lc_collate ：新数据库中使用的排序规则顺序，默认使用模板数据库的排序规则顺序。
- lc_ctype ：新数据库中使用的字符分类，默认使用模板数据库的字符分类。
- tablespace_name ：新数据库相关的表空间名称，默认为模板数据库的表空间。
- allowconn ：是否允许连接到该数据库。
- connlimit ：数据库允许的并发连接数量，默认为-1，表示没有限制。
- istemplate ：如果为真，则任何具有CREATEDB特权的用户都可以从这个数据库克隆。如果为假（默认），则只有超级用户或者该数据库的拥有者可以克隆它。
- sqlmode ：数据库类型。若为oracle，则表示创建oracle兼容的数据库；若为postgresql（默认），则表示创建postgres数据库。

示例

使用默认参数创建数据库

```
tdsql=# create database tbase_db;
CREATE DATABASE
```

```
tdsql=# \l+ tbase_db
                                                List of databases
   Name    | Owner | Encoding | Sqlmode  |  Collate   |   Ctype    | Access privileges | Size  | Tablespace | D
escription
-----------+-------+----------+----------+------------+------------+-------------------+-------+------------
+-------------
 tbase_db | tbase | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 |                   | 30 MB | pg_default |
(1 row)
```

从指定模板数据库中创建

```
tdsql=# create database tbase_db_template TEMPLATE template0;
CREATE DATABASE
tdsql=# \l+ tbase_db_template
                                                List of databases
      Name        | Owner | Encoding | Sqlmode  |  Collate   |   Ctype    | Access privileges | Size  | Tablespa
ce | Description
------------------+-------+----------+----------+------------+------------+-------------------+-------
+------------+-------------
 tbase_db_template | tbase | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 |                   | 29 MB | pg_defa
ult |
(1 row)
```

指定数据库所有者

```
tdsql=# create role pgxz with login;
CREATE ROLE
tdsql=# create database tbase_db_owner owner 'pgxz';
CREATE DATABASE
tdsql=# \l+ tbase_db_owner
                                                List of databases
     Name      | Owner | Encoding | Sqlmode  |  Collate   |   Ctype    | Access privileges | Size  | Tablespace
| Description
---------------+-------+----------+----------+------------+------------+-------------------+-------
+------------+-------------
 tbase_db_owner | pgxz  | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 |                   | 30 MB | pg_default
|
(1 row)
```

指定数据库字符集编码

```
tdsql=# create database tbase_db_encoding ENCODING UTF8;
CREATE DATABASE
tdsql=# \l+ tbase_db_encoding
```

```
                                                  List of databases
      Name       | Owner | Encoding | Sqlmode  |  Collate   |   Ctype    | Access privileges | Size  | Tablespa
ce | Description
------------------+-------+----------+----------+------------+------------+-------------------+-------
+------------+-------------
 tbase_db_encoding | tbase | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 |                   | 30 MB | pg_defa
ult |
(1 row)
```

创建gbk编码

```
tdsql=# CREATE DATABASE db_gbk template template0 encoding = gbk LC_COLLATE = 'zh_CN.gbk' LC
_CTYPE = 'zh_CN.gbk';
CREATE DATABASE
tdsql=# \l+ db_gbk
                                                List of databases
  Name  | Owner | Encoding | Sqlmode  |  Collate   |   Ctype    | Access privileges | Size  | Tablespace | Des
cription
--------+-------+----------+----------+------------+------------+-------------------+-------+------------
+-------------
 db_gbk | tbase | GBK      | postgres | zh_CN.gbk | zh_CN.gbk |                   | 29 MB | pg_default |
(1 row)
```

创建gb18030编码

```
tdsql=# create database db_gb18030 template template0 encoding=gb18030 LC_COLLATE = 'zh_CN.g
b18030' LC_CTYPE = 'zh_CN.gb18030';
CREATE DATABASE
tdsql=# \l+ db_gb18030
                                                  List of databases
    Name    | Owner | Encoding | Sqlmode  |    Collate    |     Ctype     | Access privileges | Size  | Tablespac
e | Description
------------+-------+----------+----------+---------------+---------------+-------------------+-------
+------------+-------------
 db_gb18030 | tbase | GB18030  | postgres | zh_CN.gb18030 | zh_CN.gb18030 |                   | 29 MB | pg
_default |
(1 row)
```

指定排序规则

```
tdsql=# create database tbase_db_lc_collate lc_collate 'C' template template0;
CREATE DATABASE
tdsql=# \l+ tbase_db_lc_collate
                                                  List of databases
      Name       | Owner | Encoding | Sqlmode  | Collate |   Ctype    | Access privileges | Size  | Tablespac
```

```
e | Description
-------------------+-------+----------+----------+---------+------------+------------------+-------
+------------+-------------
 tbase_db_lc_collate | tbase | UTF8     | postgres | C       | zh_CN.utf8 |                  | 29 MB | pg_default |
(1 row)
```

指定分组规则

```
tdsql=# create database tbase_db_lc_ctype LC_CTYPE 'C' template template0;
CREATE DATABASE
tdsql=# \l+ tbase_db_lc_ctype
                                        List of databases
      Name       | Owner | Encoding | Sqlmode  |  Collate   | Ctype | Access privileges | Size  | Tablespace |
Description
------------------+-------+----------+----------+------------+-------+-------------------+-------
+------------+-------------
 tbase_db_lc_ctype | tbase | UTF8     | postgres | zh_CN.utf8 | C     |                   | 29 MB | pg_default |
(1 row)
```

配置数据可连接

```
tdsql=# create database tbase_db_allow_connections ALLOW_CONNECTIONS true;
CREATE DATABASE
tdsql=# select datallowconn from pg_database where datname='tbase_db_allow_connections';
 datallowconn
--------------
 t
(1 row)
```

配置连接数

```
tdsql=# create database tbase_db_connlimit CONNECTION LIMIT 100;
CREATE DATABASE
tdsql=# select datconnlimit  from pg_database where datname='tbase_db_connlimit';
 datconnlimit
--------------
          100
(1 row)
```

配置数据库可以被复制

```
tdsql=# create database tbase_db_istemplate is_template true;
CREATE DATABASE
```

```
tdsql=# select datistemplate from pg_database where datname='tbase_db_istemplate';
 datistemplate
---------------
 t
(1 row)
```

创建oracle兼容数据库

```
tdsql=# create database db_oracle sql mode oracle;
CREATE DATABASE
tdsql=# \l+ db_oracle
                                        List of databases
   Name    | Owner | Encoding | Sqlmode |  Collate   |   Ctype    | Access privileges | Size  | Tablespace | D
escription
-----------+-------+----------+---------+------------+------------+-------------------+-------+------------
+-------------
 db_oracle | tbase | UTF8     | oracle  | zh_CN.utf8 | zh_CN.utf8 |                   | 42 MB | pg_default |
(1 row)
```

多参数同时配置

```
tdsql=# create database tbase_db_mul owner 'pgxz' CONNECTION LIMIT 50 template template0 enco
ding 'utf8'  lc_collate 'C';
CREATE DATABASE
tdsql=# \l+ tbase_db_mul
                                        List of databases
    Name     | Owner | Encoding | Sqlmode  | Collate |   Ctype    | Access privileges | Size  | Tablespace |
Description
-------------+-------+----------+----------+---------+------------+-------------------+-------+------------
+-------------
 tbase_db_mul | pgxz  | UTF8     | postgres | C       | zh_CN.utf8 |                   | 29 MB | pg_default |
(1 row)
```

创建模式

本文介绍如何使用 SQL 语句来创建模式，同时介绍创建模式的前提条件、模式使用的限制和建议等，并提供一些示例。

模式（Schema），一个模式本质上是一个名字空间：它包含命名对象（表、数据类型、函数以及操作符），对象可以与在其他模式中存在的对象重名。可以通过用模式名作为一个前缀“限定”命名对象的名称“模式名.对象名”来访问它们，或者通过把要求的模式包括在搜索路径中来访问命名对象。

创建模式

CREATE SCHEMA 语法如下：

```
CREATE SCHEMA schema_name [ AUTHORIZATION role_specification ] [ schema_element [ ... ] ]
CREATE SCHEMA AUTHORIZATION role_specification [ schema_element [ ... ] ]
CREATE SCHEMA IF NOT EXISTS schema_name [ AUTHORIZATION role_specification ]
CREATE SCHEMA IF NOT EXISTS AUTHORIZATION role_specification

where role_specification can be:

    user_name
  | CURRENT_USER
  | SESSION_USER
```

参数说明：

- schema_name ：将要创建的模式名称，不建议以pg_开头。
- user_name ：拥有该模式的用户的角色名，默认为执行该命令的用户。
- schema_element ：SQL语句，创建属于该模式的对象。目前，只接受CREATE TABLE、 CREATE VIEW、CREATE INDEX、CREATE SEQUENCE、 CREATE TRIGGER和GRANT子句。其它类型的对象需要在创建完模式之后，使用独立的命令创建。
- IF NOT EXISTS ：如果同名模式已经存在，那么跳过创建操作，通知用户；否则创建模式。使用该选项时，不能指定schema_element子句。

示例

使用默认参数创建模式

```
tdsql=# create schema tbase_1;
CREATE SCHEMA
tdsql=# \dn tbase_1
```

```
List of schemas
 Name  | Owner
-------+-------
 tbase_1 | tbase
(1 row)
```

扩展语法，不存在时才创建

```
tdsql=# create schema if not exists tbase;
NOTICE:  (42P06) schema "tbase" already exists, skipping
CREATE SCHEMA
```

创建时指定所属用户

指定所属用户为CURRENT_USER

```
tdsql=# create user cu with superuser;
CREATE ROLE
tdsql=# set role cu;
SET
tdsql=# create schema tbase_cu AUTHORIZATION current_user;
CREATE SCHEMA
tdsql=# \dn tbase_cu
 List of schemas
   Name   | Owner
----------+-------
 tbase_cu | cu
(1 row)
```

指定所属用户为SESSION_USER

```
tdsql=# create user cu with superuser;
CREATE ROLE
tdsql=# set role cu;
SET
tdsql=# create schema tbase_su AUTHORIZATION session_user;
CREATE SCHEMA
tdsql=# \dn tbase_su
 List of schemas
   Name   | Owner
----------+-------
 tbase_su | tbase
(1 row)
```

指定所属用户为当前存在的用户

```
tdsql=# create user pgxc;
CREATE ROLE
tdsql=# create schema tbase_pgxc AUTHORIZATION pgxc;
CREATE SCHEMA
tdsql=# \dn tbase_pgxc
  List of schemas
    Name    | Owner
------------+-------
 tbase_pgxc | pgxc
(1 row)
```

指定schema_element

```
CREATE SCHEMA hollywood
    CREATE TABLE films (title text, release date, awards text[])
    CREATE VIEW winners AS
        SELECT title, release FROM films WHERE awards IS NOT NULL;
tdsql=# \d hollywood.films
              Table "hollywood.films"
 Column  |  Type  | Collation | Nullable | Default
---------+--------+-----------+----------+---------
 title   | text   |           |          |
 release | date   |           |          |
 awards  | text[] |           |          |

tdsql=# \d hollywood.winners
            View "hollywood.winners"
 Column  | Type | Collation | Nullable | Default
---------+------+-----------+----------+---------
 title   | text |           |          |
 release | date |           |          |
```

创建表
创建表概述

本文介绍如何通过 SQL 命令来创建表。
在 TDSQL PG 中，一个表是一系列二维数组的集合，用来代表和储存数据对象之间的关系。

前置条件

拥有 CREATE 权限才能创建表。

数据库表命名规范

创建表时，需要先为表命名，以下是定义表名时需要遵循的要求：

- 在 TDSQL PG 数据库中，每个表的名称必须保证在 SCHEMA 内唯一。
- 表名称不能超过 63 个字符（超过63个字符表名称会被截断，保留前63个字符）。
- 建议给表起一个意义明确的名字，不要使用 t1 、 table1 这样的表名。

创建表语法

使用 CREATE TABLE 语句创建表。 CREATE TABLE 将在当前数据库中创建一个新的、初始为空的表。该表将由执行该命令的用户所拥有。
语法如下:

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_nam
e ( [
  { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
] )
[
  DISTRIBUTE BY { REPLICATION | { [ SHARD ] ( column_name ) } }
]
```

参数说明:

- TEMPORARY or TEMP
  如果指定，该表被创建为一个临时表。临时表会被在会话结束时自动被删除，或者也可以选择在当前事务结束

时删除（见下文的ON COMMIT）。当临时表存在时，已有的同名持久表将对于当前会话不可见，不过可以使用模式限定的名称进行引用。在一个临时表上创建的任何索引也自动地变为临时的。自动清理守护进程不能访问并且因此也不能清理或分析临时表。由于这个原因，应该通过会话的 SQL 命令执行合适的清理和分析操作。例如，如果一个临时表将要被用于复杂的查询，最好在插入数据完成后在其上运行ANALYZE。

- UNLOGGED
  如果指定，那么该表将不被日志记录，其中的数据不会写入到预写式日志中，也不会被复制到后备服务器中。表上创建的索引也默认不进行日志记录。这让它们比普通表快非常多。但是，该类型的表在数据库发生崩溃时，也不会被记录（在崩溃或非正常关闭之后会被自动地截断）。请谨慎使用。
- IF NOT EXISTS
  表不存在时创建，当表存在时会给出提示。
- table_name
  要被创建的表名（可以选择用schema限定）。
- column_name
  要在新表中创建的一列的名称。
- data_type
  该列的数据类型。这可以包括数组说明符。
- COLLATE collation
  COLLATE子句为该列（必须是一种可排序数据类型）赋予一个排序规则。如果没有指定，将使用该列数据类型的默认排序规则。
- column_constraint
  列级约束。
- table_constraint
  表级约束。
- DISTRIBUTE BY
  该子句指定表如何在 Datanode 之间分布或复制。
  - REPLICATION
    表的每一行都会被复制到TDSQL PG数据库集群的所有Datanode上。
  - SHARD（column_name）
    指定shard key建表。表的每一行会根据shard key对应的值计算出shardid，然后再根据系统中的shardmap中存储的Datanode与shardid的映射关系，将数据存储到对应的Datanode中。

> 说明：
>
> 您可以使用 \dt 语句查看数据库中表的信息。

创建分布式表

在TDSQL PG分布式实例的数据库中，如不指定 shard key，系统默认使用第一个字段作为表的 shard key。
shard key选择原则：

- v5.21.x支持多列分布键（v5.06.x和v2.x只支持单列分布键）
- 如果有主键，则选择主键做分布键。
- 如果主键是复合字段组合，则选择字段值选择性多的字段做分布键。
- 也可以把复合字段拼接成一个新的字段来做分布键，复合字段分布键的数据分布跟单字段一样，只是单字段只有一个字段值求hash值，多字段的分布键则由多个字段值合一起求hash值。
- 没有主键的可以使用UUID来做分布键
- 让数据尽可能地分布得足够散

> 注意：
>
> 使用复合字段作为分片键时，需要注意：
>
> - 由于在数据分片时会使用复合字段来计算hash值，应优先选择相互独立的字段进行组合，确保数据的均匀分布，避免出现数据倾斜问题。
> - SQL关联时需要将所有的分片键写在WHERE条件中，避免发生数据重分布，减少对OLTP的高频SQL执行性能的影响。
> - 如果主键为多个字段，建议直接选择主键字段作为分片键。

指定shard key 建表语法如下：

```
CREATE TABLE table_name column_definition DISTRIBUTE BY SHARD(column_name);
```

示例: 指定shard key创建表

```
tdsql=# CREATE TABLE t_test (c1 SERIAL NOT NULL,c2 INT) DISTRIBUTE BY SHARD(c1);
CREATE TABLE
tdsql=# \d+ t_test
                    Table "public.t_test"
 Column |  Type   | Collation | Nullable |            Default             | Storage | Stats target | Description
--------+---------+-----------+----------+--------------------------------+---------+--------------
+-------------
 c1     | integer |           | not null | nextval('t_test_c1_seq'::regclass) | plain   |              |
 c2     | integer |           |          |                                | plain   |              |
Distribute By: SHARD(c1)
Location Nodes: ALL DATANODES
```

创建复制表

复制表是 TDSQL PG 分布式实例的数据库中的一种特殊表。这种表的每一行都会被复制到TDSQL PG数据库集群的所有Datanode上。
创建复制表的 SQL 语法如下：

```
CREATE TABLE table_name column_definition DISTRIBUTE BY REPLICATION;
```

示例：使用下面 SQL 语句，创建复制表 t_test

```
tdsql=# CREATE TABLE t_test (c1 INT,c2 INT) DISTRIBUTE BY REPLICATION;
CREATE TABLE
tdsql=# insert into t_test values(1,1),(2,2);
INSERT 0 2
tdsql=# EXECUTE DIRECT ON(datanode_1) 'select * from t_test';
 c1 | c2
----+----
  1 |  1
  2 |  2
(2 rows)
tdsql=# EXECUTE DIRECT ON(datanode_2) 'select * from t_test';
 c1 | c2
----+----
  1 |  1
  2 |  2
(2 rows)
```

创建列存表

表的储存方式支持行存储和列存储。
ROW 行存储适合于 OLTP 业务，此类型的表上交互事务比较多，一次交互会涉及表中的多个列，用行存查询效率较高。
COLUMN 表示表的数据将以列式存储。列存储适合于数据仓库业务，此类型的表上会做大量的汇聚计算，且涉及的列操作较少。

```
CREATE TABLE coltest2 (
  id integer NOT NULL,
  name character(16) NOT NULL,
  start_date date,
  subject character(1)
) WITH (orientation = 'column');
```

> 说明：
>
> 不带WITH子句默认创建行存表
>
> 在Oracle兼容模式下，不支持创建列存储表。

创建Stash表

如果用户在使用列存表的过程中，经常需要执行点数据的操作，如单条insert或单条update等，那么可以选择打开此列存表的Stash功能。Stash功能使列存表的内部结构得到充分和高效的使用，避免数据过于碎片化可能导致的存储效率的下降问题。当Stash功能打开后列存表会优先将碎片化的增量数据存储在Stash中，然后根据用户的设置和操作将一段时间内Stash的数据存储至底层列存储结构中。 需要说明的是，Stash本身使用行存储，有着完整的事务、持久化以及故障恢复支持，是一份持久化存储，用户数据并不会因为开启Stash功能有丢失或者不一致的风险。对列存表的操作（如DML或查询）也会同时覆盖Stash，保证数据操作的一致性和完整性。

Stash功能默认为关闭状态。用户可以根据自己对表的操作以及使用特征判断是否需要打开此功能。目前有两种方式打开列存表Stash功能：

1. 直接创建一个Stach表

```
postgres=# create table t_stash(f1 int,f2 int) WITH (orientation=column, stash_enabled=on);
CREATE TABLE
```

```
postgres=# \d+ t_stash
                                Table "public.t_stash"
 Column |  Type   | compression | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-------------+-----------+----------+---------+---------+--------------+-------------
 f1     | integer | low         |           |          |         | plain   |              |
 f2     | integer | low         |           |          |         | plain   |              |
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES
Options: orientation=column, stash_enabled=on, compression=low
```

2. 修改列存表为 stash表

```
postgres=# create table t_col(f1 int,f2 int) WITH (orientation=column);
CREATE TABLE

postgres=# \d+ t_col
                                 Table "public.t_col"
 Column |  Type   | compression | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-------------+-----------+----------+---------+---------+--------------+-------------
 f1     | integer | low         |           |          |         | plain   |              |
 f2     | integer | low         |           |          |         | plain   |              |
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES
Options: orientation=column, compression=low

postgres=# alter table t_col set (stash_enabled=on);
ALTER TABLE

postgres=# \d+ t_col
                                 Table "public.t_col"
 Column |  Type   | compression | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-------------+-----------+----------+---------+---------+--------------+-------------
 f1     | integer | low         |           |          |         | plain   |              |
 f2     | integer | low         |           |          |         | plain   |              |
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES
Options: orientation=column, compression=low, stash_enabled=on
```

也可以修改已知stash表为非 stash表

```
postgres=# alter table t_stash set (stash_enabled=off);
ALTER TABLE
postgres=# \d+ t_stash
                                Table "public.t_stash"
 Column |  Type   | compression | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-------------+-----------+----------+---------+---------+--------------+-------------
```

```
 f1    | integer | low      |         |         |         | plain   |              |
 f2    | integer | low      |         |         |         | plain   |              |
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES
Options: orientation=column, compression=low, stash_enabled=off
```

创建分区表

在TDSQL PG数据库中，分区表 是一个逻辑上的大表, 通过 分区策略 或规则 关联1个或多个子表（称为 分区 ），分区表不存储数据，数据库引擎将根据 分区表定义的 策略 对数据进行处理，存储到指定的子表中。而子表还可以定义为分区表，称为子分区，这样就构成 多级分区表。使用分区表可提高各种应用程序的性能、可用性和管理性，并有助于降低存储数据的总拥有成本。TDSQL PG 提供丰富多样的分区策略和扩展 来满足不同业务需求。
分区特点:

1. 每个分区都有一个由其分区边界定义的数据子集。
2. 每个分区是相互独立的，具有相同的逻辑属性，例如列名、数据类型和约束.
3. 每个分区具有自己的名称和可选的存储特性, 从而能更精细的粒度级别管理和访问这些数据库对象。
4. 分区对应用是完全透明的，不影响应用的业务逻辑。

分区键

分区键 是分区表中一个或多个列的集合，用于确定分区表中的每一行所在的分区, 插入到表中的数据行将根据分区键中的列或表达式的值路由到分区。
例如，对于一个用户表，可以指定 user_id 列作为 Range 分区的键。数据库根据此列中的用户号是否在指定范围内，将行分配给分区。
TDSQL PG 使用分区键自动地将插入、更新和删除操作定向到合适的分区。

分区创建

TDSQL PG 分区语法 按 声明方式 创建，分2个步骤: 先建立分区表，然后建立分区子表。

创建分区表

语法如下：

```
CREATE TABLE table_name ( ... ) [ PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression ) }
```

参数说明：
PARTITION BY子句指定对表进行分区的策略，包括范围，列表，散列三种，带括号的列或表达式列表形成表的分区键。

创建分区子表

语法如下

```
/* -- 创建分区表语法 -- */
CREATE TABLE table_name PARTITION OF parent_table
    [ ( { column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ] | table_constraint } [, ... ] ) ]
    FOR VALUES partition_bound_spec
```

参数说明：

partition_bound_spec: FOR VALUES WITH '(' hash_partbound ')'

| FOR VALUES IN_P '(' partbound_datum_list ')'

| FOR VALUES FROM '(' range_datum_list ')' TO '(' range_datum_list ')'

| DEFAULT

hash_partbound: MODULUS x, REMAINDER y；

partbound_datum_list：常量，number，NULL，MINVALUE, MAXVALUE；

range_datum_list：常量，计算结果为常量的表达式，MAXVALUE, MINVALUE等；

> 说明
>
> - partition_bound_spec 必须对应父表的分区方法和分区键，且不能与父表现有分区重叠， VALUES WITH (xxx) 用于散列分区， VALUES IN_P (xxx) 用于列表分区， VALUES FROM (xxx) TO (xxx) 形式用于范围分区。
> - partition_bound_spec 中 partbound_datum_list 是一个文字、NULL、MINVALUE 或MAXVALUE，每个文字值必须是可对相应的分区键列类型强制的数字常量，或是该类型的有效输入的字符串文字。

示例

假如有一个大型的冰激凌公司，每天想记录测量最高温度以及每个区域的冰激凌销售情况，大部分查询只会访问上周的、上月的或者上季度的数据，该表的主要用途是为管理层提供在线报告，为减少旧数据量，只保留最近3年的数据，则可使用如下步骤使用分区技术：

1. 通过指定PARTITION BY子句 创建分区表(measurement表), 该子句包括分区方法（RANGE）以及用作分区键的列表(logdate)。

```
CREATE TABLE measurement (
 city_id         int not null,
 logdate         date not null,
 peaktemp        int,
 unitsales       int
) PARTITION BY RANGE (logdate);
```

2. 创建分区: 每个分区的定义必须指定对应父表的分区方法和分区键的边界, 分区子表以普通PostgreSQL表的方式创建, 可为每个分区单独指定存储参数。

> 注意：

> 如果指定边界使新分区的值与已有分区中值重叠，则会产生错误，即向父表中插入无法映射到任何现有分区的数据将会导致错误，这种情况需手工增加一个合适的分区。

```
CREATE TABLE measurement_y2006m02 PARTITION OF measurement FOR VALUES FROM ('200
6-02-01') TO ('2006-03-01');
CREATE TABLE measurement_y2006m03 PARTITION OF measurement FOR VALUES FROM ('200
6-03-01') TO ('2006-04-01');
CREATE TABLE measurement_y2007m11 PARTITION OF measurement FOR VALUES FROM ('200
7-11-01') TO ('2007-12-01');
CREATE TABLE measurement_y2007m12 PARTITION OF measurement FOR VALUES FROM ('200
7-12-01') TO ('2008-01-01');
CREATE TABLE measurement_y2008m01 PARTITION OF measurement FOR VALUES FROM ('200
8-01-01') TO ('2008-02-01')
    WITH (parallel_workers = 4);
```

3. 在分区表的键列上创建一个索引，会自动在每个分区上创建一个索引，且后续创建或绑定(attach)的分区也将包含该索引。

```
CREATE INDEX ON measurement (logdate);
```

4. 查看分区表相关信息

```
tdsql=# \d+ measurement
                          Table "public.MEASUREMENT"
  Column   |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
-----------+---------+-----------+----------+---------+---------+--------------+-------------
 CITY_ID   | INTEGER |           | not null |         | plain   |              |
 LOGDATE   | DATE    |           | not null |         | plain   |              |
 PEAKTEMP  | INTEGER |           |          |         | plain   |              |
 UNITSALES | INTEGER |           |          |         | plain   |              |
Partition key: RANGE (LOGDATE)
Indexes:
 "MEASUREMENT_LOGDATE_IDX" btree (LOGDATE)
Partitions: MEASUREMENT_Y2006M02 FOR VALUES FROM ('2006-02-01 00:00:00') TO ('2006-0
3-01 00:00:00'),
        MEASUREMENT_Y2006M03 FOR VALUES FROM ('2006-03-01 00:00:00') TO ('2006-04-01 0
0:00:00'),
        MEASUREMENT_Y2007M11 FOR VALUES FROM ('2007-11-01 00:00:00') TO ('2007-12-01 0
0:00:00'),
        MEASUREMENT_Y2007M12 FOR VALUES FROM ('2007-12-01 00:00:00') TO ('2008-01-01 0
0:00:00'),
        MEASUREMENT_Y2008M01 FOR VALUES FROM ('2008-01-01 00:00:00') TO ('2008-02-01 0
0:00:00')
```

```
Distribute By: SHARD(CITY_ID)
Location Nodes: ALL DATANODES

tdsql=# \d+ measurement_y2008m01
                    Table "public.MEASUREMENT_Y2008M01"
  Column   |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
-----------+---------+-----------+----------+---------+---------+--------------+-------------
 CITY_ID   | INTEGER |           | not null |         | plain   |              |
 LOGDATE   | DATE    |           | not null |         | plain   |              |
 PEAKTEMP  | INTEGER |           |          |         | plain   |              |
 UNITSALES | INTEGER |           |          |         | plain   |              |
Partition of: MEASUREMENT FOR VALUES FROM ('2008-01-01 00:00:00') TO ('2008-02-01 00:00:0
0')
Partition constraint: ((LOGDATE IS NOT NULL) AND (LOGDATE >= '2008-01-01 00:00:00'::DATE) A
ND (LOGDATE < '2008-02-01 00:00:00'::DATE))
Indexes:
 "MEASUREMENT_Y2008M01_LOGDATE_IDX" btree (LOGDATE)
Distribute By: SHARD(CITY_ID)
Location Nodes: ALL DATANODES
Options: parallel_workers=4
```

分区维护

通常创建完分区表的各个分区子表后并非保持静态不变，对于移除旧分区数据，为新数据周期性地增加新分区的需求比比皆是，所以需要分区维护操作包括如何删除分区，拆分分区，合并分区，添加新分区等操作 来管理分区。

删除分区

对于某个业务如不需要某个分区子表的数据，可通过 drop table ... 命令删除分区（包括该分区的表定义 和 数据 ），不过需注意:

1. 通过drop table 方式删除后，数据将清除，子表不能作为普通表独立存在，安全的做法使用 解绑分区 alter table ... detach partition ... 方式。
2. 要在父表上拿到 ACCESS EXCLUSIVE锁才能操作。
   相关操作示例如下：

```
tdsql=# DROP TABLE measurement_y2006m02;

/* -- 查看分区表信息 -- */
tdsql=# \d+ measurement
                    Table "public.MEASUREMENT"
  Column   |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
-----------+---------+-----------+----------+---------+---------+--------------+-------------
```

```
 CITY_ID   | INTEGER |          | not null |         | plain   |             |
 LOGDATE   | DATE    |          | not null |         | plain   |             |
 PEAKTEMP  | INTEGER |          |          |         | plain   |             |
 UNITSALES | INTEGER |          |          |         | plain   |             |
Partition key: RANGE (LOGDATE)
Indexes:
    "MEASUREMENT_LOGDATE_IDX" btree (LOGDATE)
Partitions: MEASUREMENT_Y2006M03 FOR VALUES FROM ('2006-03-01 00:00:00') TO ('2006-04-01 0
0:00:00'),
            MEASUREMENT_Y2007M11 FOR VALUES FROM ('2007-11-01 00:00:00') TO ('2007-12-01 00:0
0:00'),
            MEASUREMENT_Y2007M12 FOR VALUES FROM ('2007-12-01 00:00:00') TO ('2008-01-01 00:0
0:00'),
            MEASUREMENT_Y2008M01 FOR VALUES FROM ('2008-01-01 00:00:00') TO ('2008-02-01 00:0
0:00')
Distribute By: SHARD(CITY_ID)
Location Nodes: ALL DATANODES

/* -- 查看删除分区子表信息 -- */
tdsql=# \d+ MEASUREMENT_Y2006M02
没有找到任何名称为 "MEASUREMENT_Y2006M02" 的关联。          -- 表示已经无该关系表，无法使用SQL 命
令来操作该分区子表
```

清理分区数据

如果想清理某个分区子表的数据，而不删除该子分区子表的定义。

方法一：

可以通过 alter table ... truncate ... partition(分区子表名) 方式清除某个分区的数据

```
/* -- 写入数据 -- */
tdsql=# INSERT INTO measurement VALUES(1, '2006-03-01 12:23:32', 1, 100);
tdsql=# INSERT INTO measurement VALUES(1, '2007-11-01 12:23:32', 2, 200);
tdsql=# INSERT INTO measurement VALUES(1, '2007-12-01 12:23:32', 3, 300);
tdsql=# INSERT INTO measurement VALUES(1, '2008-01-01 12:23:32', 4, 400);

/* -- 验证各个分区数据 -- */
tdsql=# select "tableoid"::regclass, count(*) from measurement group by "tableoid" order by 1;
      tableoid       | COUNT
---------------------+-------
 MEASUREMENT_Y2006M03 |     1
 MEASUREMENT_Y2007M11 |     1
 MEASUREMENT_Y2007M12 |     1
 MEASUREMENT_Y2008M01 |     1
(4 rows)

/* -- Truncate 子表, 将只清理该分区数据 -- */
```

```
tdsql=# alter table measurement truncate partition(measurement_y2006m03);

/* -- 验证各个分区数据 -- */
tdsql=# select "tableoid"::regclass, count(*) from measurement group by "tableoid" order by 1;
      tableoid       | COUNT
---------------------+-------
 MEASUREMENT_Y2007M11 |     1
 MEASUREMENT_Y2007M12 |     1
 MEASUREMENT_Y2008M01 |     1
(3 rows)
```

方法二：

也可通过　delete from ... partition(分区子表名 ） 方式清除某个分区的数据。

```
/* -- delete分区., 将只清理该分区数据 -- */
tdsql=# delete from measurement partition(measurement_y2007m12);

/* -- 验证各个分区数据 -- */
tdsql=# select "tableoid"::regclass, count(*) from measurement group by "tableoid" order by 1;

      tableoid       | COUNT
---------------------+-------
 MEASUREMENT_Y2008M01 |     1
 MEASUREMENT_Y2007M11 |     1
(2 rows)
```

解绑 && 绑定分区

如果想把某个分区子表从当前分区表分离出去，而保留该分区子表的数据，使得 SQL（查询，修改，删除，统计等）操作分区表时，将不再考虑该分区子表的数据，则可通过 detach 方式 把某个分区子表从分区表中解绑，使它作为一个独立的普通表存在, 而往分区表读写数据时将不再访问该分区子表，不过可通过单独的命令操作该分区子表，而解绑后的分区子表可通过 attach 方式 重新加入到分区表中，具体操作示例如下：

```
/* -- 构造数据 -- */
tdsql=# INSERT INTO measurement VALUES(1, '2006-03-01 12:23:32', 1, 100);
tdsql=# INSERT INTO measurement VALUES(1, '2006-03-02 12:23:32', 2, 200);
tdsql=# select * from measurement;
 CITY_ID |       LOGDATE       | PEAKTEMP | UNITSALES
---------+---------------------+----------+-----------
       1 | 2006-03-01 12:23:32 |        1 |       100
       1 | 2006-03-02 12:23:32 |        2 |       200
(2 rows)

tdsql=# select * from measurement partition(measurement_y2006m03);
 CITY_ID |       LOGDATE       | PEAKTEMP | UNITSALES
```

```
---------+--------------------+----------+-----------
       1 | 2006-03-01 12:23:32 |        1 |       100
       1 | 2006-03-02 12:23:32 |        2 |       200
(2 rows)

/* -- 解绑分区子表 -- */
tdsql=# ALTER TABLE measurement DETACH PARTITION measurement_y2006m03;

/* -- 查看分区表信息 -- */
tdsql=# \d+ measurement
                              Table "public.MEASUREMENT"
  Column   |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
-----------+---------+-----------+----------+---------+---------+--------------+-------------
 CITY_ID   | INTEGER |           | not null |         | plain   |              |
 LOGDATE   | DATE    |           | not null |         | plain   |              |
 PEAKTEMP  | INTEGER |           |          |         | plain   |              |
 UNITSALES | INTEGER |           |          |         | plain   |              |
Partition key: RANGE (LOGDATE)
Indexes:
    "MEASUREMENT_LOGDATE_IDX" btree (LOGDATE)
Partitions: MEASUREMENT_Y2007M11 FOR VALUES FROM ('2007-11-01 00:00:00') TO ('2007-12-01 0
0:00:00'),
            MEASUREMENT_Y2007M12 FOR VALUES FROM ('2007-12-01 00:00:00') TO ('2008-01-01 00:0
0:00'),
            MEASUREMENT_Y2008M01 FOR VALUES FROM ('2008-01-01 00:00:00') TO ('2008-02-01 00:0
0:00')
Distribute By: SHARD(CITY_ID)
Location Nodes: ALL DATANODES

/* -- 查看解绑后该分区子表信息 -- */
tdsql=# \d+ measurement_y2006m03
                            Table "public.MEASUREMENT_Y2006M03"
  Column   |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
-----------+---------+-----------+----------+---------+---------+--------------+-------------
 CITY_ID   | INTEGER |           | not null |         | plain   |              |
 LOGDATE   | DATE    |           | not null |         | plain   |              |
 PEAKTEMP  | INTEGER |           |          |         | plain   |              |
 UNITSALES | INTEGER |           |          |         | plain   |              |
Indexes:
    "MEASUREMENT_Y2006M03_LOGDATE_IDX" btree (LOGDATE)
Distribute By: SHARD(CITY_ID)
Location Nodes: ALL DATANODES

/* -- 1. 查看解绑后分区表数据 -- */
tdsql=# select * from measurement;
 CITY_ID | LOGDATE | PEAKTEMP | UNITSALES
---------+---------+----------+-----------
```

```
(0 rows)

/* -- 2. 查看解绑后分区子表数据, 将出错 -- */
tdsql=# select * from measurement partition(measurement_y2006m03);
ERROR:  relation MEASUREMENT_Y2006M03 is not a child table of MEASUREMENT .

/* -- 3. 直接查看解绑后分区子表数据 -- */
tdsql=# select * from measurement_y2006m03;
 CITY_ID |       LOGDATE       | PEAKTEMP | UNITSALES
---------+---------------------+----------+-----------
       1 | 2006-03-01 12:23:32 |        1 |       100
       1 | 2006-03-02 12:23:32 |        2 |       200
(2 rows)

/* -- 4. 验证写入分区子表数据 -- */
tdsql=# insert into measurement_y2006m03 values(1, '2006-03-03 12:23:32', 3, 300);
tdsql=# select * from MEASUREMENT_Y2006M03;
 CITY_ID |       LOGDATE       | PEAKTEMP | UNITSALES
---------+---------------------+----------+-----------
       1 | 2006-03-01 12:23:32 |        1 |       100
       1 | 2006-03-02 12:23:32 |        2 |       200
       1 | 2006-03-03 12:23:32 |        3 |       300
(3 rows)

/* -- 5. 回归验证分区表，则无数据 -- */
tdsql=# select * from measurement;
 CITY_ID | LOGDATE | PEAKTEMP | UNITSALES
---------+---------+----------+-----------
(0 rows)

/* -- 6. 通过绑定操作可以把表重新绑定到分区表上 -- */
tdsql=# alter table measurement attach partition measurement_y2006m03 FOR VALUES FROM ('200
6-03-01') TO ('2006-04-01');

/* -- 7. 查看绑定的分区表信息 -- */
tdsql=# \d+ measurement;
                          Table "public.MEASUREMENT"
  Column   |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
-----------+---------+-----------+----------+---------+---------+--------------+-------------
 CITY_ID   | INTEGER |           | not null |         | plain   |              |
 LOGDATE   | DATE    |           | not null |         | plain   |              |
 PEAKTEMP  | INTEGER |           |          |         | plain   |              |
 UNITSALES | INTEGER |           |          |         | plain   |              |
Partition key: RANGE (LOGDATE)
Indexes:
    "MEASUREMENT_LOGDATE_IDX" btree (LOGDATE)
Partitions: MEASUREMENT_Y2006M03 FOR VALUES FROM ('2006-03-01 00:00:00') TO ('2006-04-01 0
```

```
0:00:00'),
        MEASUREMENT_Y2007M11 FOR VALUES FROM ('2007-11-01 00:00:00') TO ('2007-12-01 00:0
0:00'),
        MEASUREMENT_Y2007M12 FOR VALUES FROM ('2007-12-01 00:00:00') TO ('2008-01-01 00:0
0:00'),
        MEASUREMENT_Y2008M01 FOR VALUES FROM ('2008-01-01 00:00:00') TO ('2008-02-01 00:0
0:00')
Distribute By: SHARD(CITY_ID)
Location Nodes: ALL DATANODES

/* -- 8. 查看绑定的分区表数据 -- */
tdsql=# select * from measurement;
 CITY_ID |       LOGDATE       | PEAKTEMP | UNITSALES
---------+---------------------+----------+-----------
       1 | 2006-03-01 12:23:32 |        1 |       100
       1 | 2006-03-02 12:23:32 |        2 |       200
       1 | 2006-03-03 12:23:32 |        3 |       300
(3 rows)
```

拆分分区

当分区表内数据分布倾斜或不均衡时，可通过拆分分区使得数据分布在各个分区上保持均衡，TDSQL PG支持RANGE、LIST的一级和多级分区拆分，分区拆分通常用于将DEFAULT分区拆分到新的分区，或者将一个较大的分区拆分成两个分区。

范围（RANGE）分区拆分语法：

```
ALTER TABLE table_name SPLIT PARTITION part_3 AT ( { numeric_literal | string_literal } ) INTO (PARTITI
ON part_1, PARTITION part_2);
```

列表（LIST）分区拆分语法：

```
ALTER TABLE table_name SPLIT PARTITION part_3 VALUES ( { numeric_literal | string_literal | NULL } , { n
umeric_literal | string_literal | NULL } )
      INTO (PARTITION part_1, PARTITION part_2);
```

例子：把默认分区分成两个分区

```
/* -- 1. 创建分区表 -- */
tdsql=# CREATE TABLE tbl_split (
            id serial,
            create_time timestamp(0) without time zone
        ) PARTITION BY RANGE(create_time);
tdsql=# CREATE TABLE tbl_split_p1 PARTITION OF tbl_split FOR VALUES FROM ('2024-01-01') TO ('202
4-02-01');
tdsql=# CREATE TABLE tbl_split_p2 PARTITION OF tbl_split FOR VALUES FROM ('2024-02-01') TO ('202
```

```
4-03-01');
tdsql=# CREATE TABLE tbl_split_def PARTITION OF tbl_split DEFAULT;

/* -- 2. 查看分区信息 -- */
tdsql=# \d+ tbl_split;
                                        Table "public.TBL_SPLIT"
   Column    |            Type             | Collation | Nullable |              Default               | Storage | Stats tar
get | Description
-------------+-----------------------------+-----------+----------+------------------------------------
+---------+--------------+-------------
 ID          | INTEGER                     |           | not null | NEXTVAL('TBL_SPLIT_ID_SEQ'::REGCLASS) | plain   |
|
 CREATE_TIME | TIMESTAMP(0) WITHOUT TIME ZONE |           |          |                                    | plain   |
|
Partition key: RANGE (CREATE_TIME)
Partitions: TBL_SPLIT_P1 FOR VALUES FROM ('2024-01-01 00:00:00') TO ('2024-02-01 00:00:00'),
            TBL_SPLIT_P2 FOR VALUES FROM ('2024-02-01 00:00:00') TO ('2024-03-01 00:00:00'),
            TBL_SPLIT_DEF DEFAULT
Distribute By: SHARD(ID)
Location Nodes: ALL DATANODES

/* -- 3. 写入数据 -- */
INSERT INTO tbl_split (id, create_time) VALUES (1,'2024-01-01');
INSERT INTO tbl_split (id, create_time) VALUES (2,'2024-02-01');
INSERT INTO tbl_split (id, create_time) VALUES (3,'2024-03-01');
INSERT INTO tbl_split (id, create_time) VALUES (4,'2024-04-01');
INSERT INTO tbl_split (id, create_time) VALUES (5,'2024-05-01');

/* -- 4. 查看分区数据分布 -- */
tdsql=# select "tableoid"::regclass, count(*) from tbl_split group by "tableoid" order by 1;
   tableoid    | COUNT
---------------+-------
 TBL_SPLIT_P1  |     1
 TBL_SPLIT_P2  |     1
 TBL_SPLIT_DEF |     3
(3 rows)

/* -- 5. 拆分默认分区 -- */
tdsql=# alter table tbl_split split partition tbl_split_def at ('2024-04-01')
              into (partition tbl_split_p3, partition tbl_split_def);

/* -- 6. 查看拆分后的分区表信息 -- */
tdsql=# \d+ tbl_split
                                        Table "public.TBL_SPLIT"
   Column    |            Type             | Collation | Nullable |              Default               | Storage | Stats tar
get | Description
-------------+-----------------------------+-----------+----------+------------------------------------
```

```
+---------+-------------+-------------
 ID          | INTEGER                     |           | not null | NEXTVAL('TBL_SPLIT_ID_SEQ'::REGCLASS) | plain   |
|
 CREATE_TIME | TIMESTAMP(0) WITHOUT TIME ZONE |           |          |                                       | plain   |
|
Partition key: RANGE (CREATE_TIME)
Partitions: TBL_SPLIT_P1 FOR VALUES FROM ('2024-01-01 00:00:00') TO ('2024-02-01 00:00:00'),
            TBL_SPLIT_P2 FOR VALUES FROM ('2024-02-01 00:00:00') TO ('2024-03-01 00:00:00'),
            TBL_SPLIT_P3 FOR VALUES FROM ('2024-03-01 00:00:00') TO ('2024-04-01 00:00:00'),
            TBL_SPLIT_DEF DEFAULT
Distribute By: SHARD(ID)
Location Nodes: ALL DATANODES

/* -- 7. 查看拆分后的分区表数据分布 -- */
tdsql=# select "tableoid"::regclass, count(*) from tbl_split group by "tableoid" order by 1;
   tableoid    | COUNT
---------------+-------
 TBL_SPLIT_P1  |     1
 TBL_SPLIT_P2  |     1
 TBL_SPLIT_P3  |     1
 TBL_SPLIT_DEF |     2
(4 rows)
```

合并分区

TDSQL PG支持分区表合并，通过分区合并，可将两个分区的数据合并到另一个分区中。分区合并通常用于将一些较小的列表分区或相邻的较小的范围分区合并，减少分区数量，从而提高效率。

> 注意：
>
> - 不能用于散列（HASH）分区下的子分区的合并。
> - 范围（RANGE）分区合并时，允许合并两临近范围的分区到另外分区，不相邻分区无法合并。合并结果分区将继承两个源分区的最大边界。
> - 暂不支持UPDATE [GLOBAL] INDEXES。

语法：

```
ALTER TABLE table_name MERGE PARTITIONS ( part1, part2 ) INTO PARTITION part3 ;
```

例子：合并两个分区

```
/* -- 1. 创建分区信息 -- */
tdsql=# CREATE TABLE tbl_merge (
                id serial,
                create_time timestamp(0) without time zone
        ) PARTITION BY RANGE(create_time);
```

```
tdsql=# CREATE TABLE tbl_merge_p1 PARTITION OF tbl_merge FOR VALUES FROM ('2024-01-01') TO
('2024-02-01');
tdsql=# CREATE TABLE tbl_merge_p2 PARTITION OF tbl_merge FOR VALUES FROM ('2024-02-01') TO
('2024-03-01');
tdsql=# CREATE TABLE tbl_merge_p3 PARTITION OF tbl_merge FOR VALUES FROM ('2024-03-01') TO
('2024-04-01');
tdsql=# CREATE TABLE tbl_merge_def PARTITION OF tbl_merge DEFAULT;

/* -- 2. 查看分区信息 -- */
tdsql=# \d+ tbl_merge;
                                          Table "public.TBL_MERGE"
  Column    |            Type             | Collation | Nullable |              Default              | Storage | Stats tar
get | Description
------------+-----------------------------+-----------+----------+-----------------------------------
+---------+-------------+-------------
 ID         | INTEGER                     |           | not null | NEXTVAL('TBL_MERGE_ID_SEQ'::REGCLASS) | plain
|           |
 CREATE_TIME | TIMESTAMP(0) WITHOUT TIME ZONE |           |          |                                   | plain   |
|
Partition key: RANGE (CREATE_TIME)
Partitions: TBL_MERGE_P1 FOR VALUES FROM ('2024-01-01 00:00:00') TO ('2024-02-01 00:00:00'),
            TBL_MERGE_P2 FOR VALUES FROM ('2024-02-01 00:00:00') TO ('2024-03-01 00:00:00'),
            TBL_MERGE_P3 FOR VALUES FROM ('2024-03-01 00:00:00') TO ('2024-04-01 00:00:00'),
            TBL_MERGE_DEF DEFAULT
Distribute By: SHARD(ID)
Location Nodes: ALL DATANODES

/* -- 3. 写入数据 -- */
tdsql=# INSERT INTO tbl_merge (id, create_time) VALUES (1,'2024-01-01');
tdsql=# INSERT INTO tbl_merge (id, create_time) VALUES (2,'2024-02-01');
tdsql=# INSERT INTO tbl_merge (id, create_time) VALUES (3,'2024-03-01');
tdsql=# INSERT INTO tbl_merge (id, create_time) VALUES (4,'2024-04-01');
tdsql=# INSERT INTO tbl_merge (id, create_time) VALUES (5,'2024-05-01');

/* -- 4. 查看分区数据分布 -- */
tdsql=# select "tableoid"::regclass, count(*) from tbl_merge group by "tableoid" order by 1;
   tableoid    | COUNT
---------------+-------
 TBL_MERGE_P1  |     1
 TBL_MERGE_P2  |     1
 TBL_MERGE_P3  |     1
 TBL_MERGE_DEF |     2
(4 rows)

/* -- 5. 合并 tbl_merge_p3 和 tbl_merge_def 分区 -- */
tdsql=# alter table tbl_merge merge partitions(tbl_merge_p3, tbl_merge_def) into partition tbl_merge_
def;
```

```
/* -- 6. 查看合并后的分区表信息 -- */
tdsql=# \d+ tbl_merge
                          Table "public.TBL_MERGE"
  Column    |            Type             | Collation | Nullable |              Default              | Storage | Stats tar
get | Description
------------+-----------------------------+-----------+----------+-----------------------------------
+---------+--------------+-------------
 ID         | INTEGER                     |           | not null | NEXTVAL('TBL_MERGE_ID_SEQ'::REGCLASS) | plain
|            |
 CREATE_TIME | TIMESTAMP(0) WITHOUT TIME ZONE |           |          |                                   | plain   |
|
Partition key: RANGE (CREATE_TIME)
Partitions: TBL_MERGE_P1 FOR VALUES FROM ('2024-01-01 00:00:00') TO ('2024-02-01 00:00:00'),
            TBL_MERGE_P2 FOR VALUES FROM ('2024-02-01 00:00:00') TO ('2024-03-01 00:00:00'),
            TBL_MERGE_DEF DEFAULT
Distribute By: SHARD(ID)
Location Nodes: ALL DATANODES

/* -- 7. 查看合并后的分区表数据分布 -- */
tdsql=# select "tableoid"::regclass, count(*) from tbl_merge group by "tableoid" order by 1;
   tableoid    | COUNT
---------------+-------
 TBL_MERGE_P1 |     1
 TBL_MERGE_P2 |     1
 TBL_MERGE_DEF |     3
(3 rows)
```

分区策略

TDSQL PG 支持以下多种分区策略，用于控制数据库如何将数据放入分区：

- 范围（RANGE）分区:一种分区技术，根据分区键值的范围将数据映射到分区，最常见的是基于时间字段做分区。
- 列表（LIST）分区:一种分区技术，数据库使用离散值列表作为每个分区的分区键。
- 散列（HASH）分区:一种分区技术，散列分区根据指定的分区键的散列算法将数据映射到分区。
- 多级分区：一种结合两种或多种分区方法的分区技术，可使用范围（RANGE）、列表（LIST）、散列（HASH）根据业务特点进行组合。

范围分区

根据分区键值的范围将数据映射到分区，最常见的是基于时间字段做分区，把数据按不同时间段存储在不同的分区子表中。

范围分区格式

```
CREATE TABLE table_name (col1, col2, col3 ...) PARTITION BY RANGE ( col1 );
CREATE TABLE table_name_part1 PARTITION OF table_name FOR VALUES FROM ( MINVALUE ) TO ( exp
r );
CREATE TABLE table_name_part2 PARTITION OF table_name FOR VALUES FROM ( exp1 ) TO ( expr2 );
CREATE TABLE table_name_part3 PARTITION OF table_name FOR VALUES FROM ( exp1 ) TO ( expr2 );
...
CREATE TABLE table_name_part4 PARTITION OF table_name FOR VALUES FROM ( MAXVALUE ) TO ( ex
pr2 );
CREATE TABLE table_name_part5 PARTITION OF table_name DEFAULT;
```

> 说明
>
> - 分区键可以包含多个列或表达式（最多32个）
> - 创建范围分区时，由FROM指定下限是一个包含范围，而用TO指定的上限是排除范围。在FROM列表中指定的值是该分区的相应分区键列的有效值，而TO列表中的值不是。
> - 创建范围分区时，可以使用特殊值MINVALUE 和MAXVALUE，表示列值下限或上限，如 FROM (MINVALUE) TO (10) 表示允许任何小于10的值，FROM (10) TO (MAXVALUE)表示允许任何大于或等于10的值。
> - 默认分区：如写数据不在分区范围，会报告错误，可通过引入默认分区来存储超出范围的数据, 注意点：
>   - 一个分区表只能拥有一个 DEFAULT 分区；
>   - 对于已经存储在 DEFAULT 分区中的数据，不能再创建相应的分区；
>   - 如果将已有的表挂载为 DEFAULT 分区，将会检查该表中的所有数据；如果在已有的分区中存在相同的数据，将会产生一个错误；
>   - 散列分区表不支持 DEFAULT 分区。
>   - MAXVALUE && MINVALUE: MAXVALUE 和 MINVALUE 是 范围的 上限 和 下限，进一步划分分区的边界。

示例 - 创建时间范围分区，包括默认分区 && MAXVALUE && MINVALUE

创建一个日志分区表，存储2023年1月和2月的日志，使用创建时间作为分区键，按月分区。

```
/* -- 1. 创建分区表 --*/
create table tbl_log_2023 (id serial, create_time timestamp(0) without time zone, remark char(1)) partiti
on by RANGE(create_time);

/* -- 2. 创建分区子表 --*/
CREATE TABLE tbl_log_202301 PARTITION OF tbl_log_2023 FOR VALUES FROM ('2023-01-01') TO ('202
3-02-01');
CREATE TABLE tbl_log_202302 PARTITION OF tbl_log_2023 FOR VALUES FROM ('2023-02-01') TO ('202
3-03-01');

/* -- 3. 查看表信息 -- */
```

```
tdsql=# \d+ tbl_log_2023

                                                Table "public.TBL_LOG_2023"
  Column    |            Type             | Collation | Nullable |                Default                | Storage  | Stats t
arget | Description
------------+-----------------------------+-----------+----------
+---------------------------------------+----------+-------------+-------------
 ID          | INTEGER                     |           | not null | NEXTVAL('TBL_LOG_2023_ID_SEQ'::REGCLASS) | pla
in    |             |
 CREATE_TIME | TIMESTAMP(0) WITHOUT TIME ZONE |           |          |                                       | plain
|             |
 REMARK      | CHARACTER(1)                |           |          |                                       | extended |             |
Partition key: RANGE (CREATE_TIME)
Partitions: TBL_LOG_202301 FOR VALUES FROM ('2023-01-01 00:00:00') TO ('2023-02-01 00:00:00'),
            TBL_LOG_202302 FOR VALUES FROM ('2023-02-01 00:00:00') TO ('2023-03-01 00:00:00')
Distribute By: SHARD(ID)
Location Nodes: ALL DATANODES

/* -- 4. 验证分区 -- */
/* -- 4.1 写入2023 1月到 2月数据 -- */
tdsql=# insert into tbl_log_2023(create_time, remark) values('2023-01-01 10:55:01', 1);
tdsql=# insert into tbl_log_2023(create_time, remark) values('2023-01-02 10:55:01', 1);
tdsql=# insert into tbl_log_2023(create_time, remark) values('2023-02-01 10:55:01', 1);
tdsql=# insert into tbl_log_2023(create_time, remark) values('2023-02-02 10:55:01', 1);

/* -- 4.2 查看TBL_LOG_202301 分区数据 -- */
tdsql=# select * from tbl_log_2023 partition(TBL_LOG_202301) ORDER BY ID;
 ID |     CREATE_TIME     | REMARK
----+---------------------+--------
  1 | 2023-01-01 10:55:01 | 1
  2 | 2023-01-02 10:55:01 | 1
(2 rows)

/* -- 4.3 查看TBL_LOG_202302 分区数据 -- */
tdsql=# select * from tbl_log_2023 partition(TBL_LOG_202302) ORDER BY ID;
 ID |     CREATE_TIME     | REMARK
----+---------------------+--------
  3 | 2023-02-01 10:55:01 | 1
  4 | 2023-02-02 10:55:01 | 1
(2 rows)

/* -- 4.4 统计 TBL_LOG_2023 各个分区数据 -- */
tdsql=# select "tableoid"::regclass, count(*) from tbl_log_2023 group by "tableoid" order by 1;
RELNAME     | COUNT
----------------+-------
TBL_LOG_202301 |     2
TBL_LOG_202302 |     2
```

```
(2 rows)

/* -- 5. 使用默认分区  -- */
/* -- 5.1 当前无默认分区，写入范围外的记录将报告错误  -- */
tdsql=# insert into tbl_log_2023(create_time, remark) values('2022-12-01 10:55:01', 1);
ERROR:  no partition of relation "TBL_LOG_2023" found for row
DETAIL:  Partition key of the failing row contains (CREATE_TIME) = (2022-12-01 10:55:01).

tdsql=# insert into tbl_log_2023(create_time, remark) values('2023-03-01 10:55:01', 1);
ERROR:  no partition of relation "TBL_LOG_2023" found for row
DETAIL:  Partition key of the failing row contains (CREATE_TIME) = (2023-03-01 10:55:01).

/* -- 5.2 创建默认分区 -- */
tdsql=# CREATE TABLE tbl_log_2023_default PARTITION OF tbl_log_2023 default;
tdsql=# insert into tbl_log_2023(create_time, remark) values('2022-12-01 10:55:01', 1);
tdsql=# insert into tbl_log_2023(create_time, remark) values('2023-03-01 10:55:01', 1);

tdsql=# select * from tbl_log_2023 partition(tbl_log_2023_default) ORDER BY ID;
 ID |     CREATE_TIME     | REMARK
----+---------------------+--------
  5 | 2022-12-01 10:55:01 | 1
  6 | 2023-03-01 10:55:01 | 1
(2 rows)

/*
 * 6. 设置 MAXVALUE && MINVALUE 边界
 *
 * 如2023年之前的日志建立一个分区，以及超过2023年3月的日志建立一个分区
 */
/* -- 6.1 清理老表 -- */
tdsql=# drop table if exists tbl_log_2023;

/* -- 6.2 创建分区主表 -- */
tdsql=# create table tbl_log_2023(id serial, create_time timestamp(0) without time zone, remark char
(1)) partition by RANGE(create_time);

/* -- 6.3 创建分区子表 -- */
tdsql=# CREATE TABLE tbl_log_2023_min PARTITION OF tbl_log_2023 FOR VALUES FROM (MINVALUE)
TO ('2023-01-01');
tdsql=# CREATE TABLE tbl_log_202301 PARTITION OF tbl_log_2023 FOR VALUES FROM ('2023-01-01')
TO ('2023-02-01');
tdsql=# CREATE TABLE tbl_log_202302 PARTITION OF tbl_log_2023 FOR VALUES FROM ('2023-02-01')
TO ('2023-03-01');
tdsql=# CREATE TABLE tbl_log_2023_max PARTITION OF tbl_log_2023 FOR VALUES FROM ('2023-03-0
1') TO (MAXVALUE);

/* -- 6.4 查看分区表信息 -- */
```

```
tdsql=# \d+ tbl_log_2023
                                          Table "public.TBL_LOG_2023"
   Column    |            Type             | Collation | Nullable |                Default                | Storage  | Stats t
arget | Description
-------------+-----------------------------+-----------+----------
+--------------------------------------+----------+-------------+-------------
 ID          | INTEGER                     |           | not null | NEXTVAL('TBL_LOG_2023_ID_SEQ'::REGCLASS) | pla
in    |             |
 CREATE_TIME | TIMESTAMP(0) WITHOUT TIME ZONE |           |          |                                      | plain
|             |
 REMARK      | CHARACTER(1)                |           |          |                                      | extended |             |
Partition key: RANGE (CREATE_TIME)
Partitions: TBL_LOG_202301 FOR VALUES FROM ('2023-01-01 00:00:00') TO ('2023-02-01 00:00:00'),
            TBL_LOG_202302 FOR VALUES FROM ('2023-02-01 00:00:00') TO ('2023-03-01 00:00:00'),
            TBL_LOG_2023_MAX FOR VALUES FROM ('2023-03-01 00:00:00') TO (MAXVALUE),
            TBL_LOG_2023_MIN FOR VALUES FROM (MINVALUE) TO ('2023-01-01 00:00:00')
Distribute By: SHARD(ID)
Location Nodes: ALL DATANODES

/* -- 6.5 验证 MIN && MAX -- */
tdsql=# insert into tbl_log_2023(create_time, remark) values('2022-12-01 13:55:02', 1);
tdsql=# insert into tbl_log_2023(create_time, remark) values('2023-03-01 14:52:01', 1);

tdsql=# select * from tbl_log_2023 partition(TBL_LOG_2023_MIN) ORDER BY ID;
 ID |     CREATE_TIME     | REMARK
----+---------------------+--------
  1 | 2022-12-01 13:55:02 | 1
(1 row)

tdsql=# select * from tbl_log_2023 partition(TBL_LOG_2023_MAX) ORDER BY ID;
 ID |     CREATE_TIME     | REMARK
----+---------------------+--------
  2 | 2023-03-01 14:52:01 | 1
(1 row)
```

示例 - 整数范围分区

```
/* -- 1. 创建分区 -- */
CREATE TABLE t_prt_range (a int) PARTITION BY RANGE(a);
CREATE TABLE t_prt_range_p1 PARTITION OF t_prt_range FOR VALUES FROM (0) TO (250);
CREATE TABLE t_prt_range_p2 PARTITION OF t_prt_range FOR VALUES FROM (250) TO (500);
CREATE TABLE t_prt_range_p3 PARTITION OF t_prt_range DEFAULT;

/* -- 2. 查看分区信息 -- */
tdsql=# \d+ t_prt_range
                          Table "public.T_PRT_RANGE"
```

```
 Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------+-------------
 A      | INTEGER |           |          |         | plain   |              |
Partition key: RANGE (A)
Partitions: T_PRT_RANGE_P1 FOR VALUES FROM (0) TO (250),
            T_PRT_RANGE_P2 FOR VALUES FROM (250) TO (500),
            T_PRT_RANGE_P3 DEFAULT
Distribute By: SHARD(A)
Location Nodes: ALL DATANODES

/* -- 3. 写1000条数据 -- */
tdsql=# INSERT INTO t_prt_range SELECT gs FROM generate_series(1,1000) gs;

/* -- 4. 统计各分区数据 -- */
tdsql=# select "tableoid"::regclass, count(*) from t_prt_range group by "tableoid" order by 1;
tableoid    | COUNT
----------------+-------
T_PRT_RANGE_P1 |   249
T_PRT_RANGE_P2 |   250
T_PRT_RANGE_P3 |   501
(3 rows)
```

列表分区

列表分区通过为分区键指定离散值列表，通过该列表显式控制行映射到分区子表。

列表分区的优点是可以以自然的方式对无序和不相关的数据集进行分组和组织，如以地区列作为分区键的表，分区可能包含值城市a、城市b和城市c等。

语法：

```
CREATE TABLE table_name (col1, col2, col3 ...) PARTITION BY LIST ( col1 );
CREATE TABLE table_name_part_1 PARTITION OF table_name FOR VALUES IN ( exp1, exp2 ... );
...
CREATE TABLE table_name_part_n PARTITION OF table_name DEFAULT;
```

> 说明
>
> - 分区键由一个或多个列组成, 可指定NULL表示允许分区键列为空。
> - exp1 可以是常量，数字, NULL, TRUE, FALSE 等。

示例

```
/* -- 1. 创建分区表 -- */
tdsql=# CREATE TABLE tbl_city (id int, city varchar) PARTITION BY LIST (city);
tdsql=# CREATE TABLE tbl_city_super PARTITION OF tbl_city FOR VALUES IN ('城市a', '城市b', '城市c', '城市d');
tdsql=# CREATE TABLE tbl_city_others PARTITION OF tbl_city FOR VALUES IN (NULL);
```

```
tdsql=# CREATE TABLE tbl_city_def PARTITION OF tbl_city DEFAULT;

/* -- 2. 查看分区表信息 -- */
tdsql=# \d+ tbl_city
                                Table "public.TBL_CITY"
 Column |       Type        | Collation | Nullable | Default | Storage  | Stats target | Description
--------+-------------------+-----------+----------+---------+----------+--------------+-------------
 ID     | INTEGER           |           |          |         | plain    |              |
 CITY   | CHARACTER VARYING |           |          |         | extended |              |
Partition key: LIST (CITY)
Partitions: TBL_CITY_OTHERS FOR VALUES IN (NULL),
            TBL_CITY_SUPER FOR VALUES IN ('城市a', '城市b', '城市c', '城市d'),
            TBL_CITY_DEF DEFAULT
Distribute By: SHARD(ID)
Location Nodes: ALL DATANODES

/* -- 3. 验证分区 -- */
tdsql=# INSERT INTO tbl_city VALUES(1, '城市a');              -- 将写入 tbl_city_super 分区
tdsql=# INSERT INTO tbl_city VALUES(2, '城市b');              -- 将写入 tbl_city_super 分区
tdsql=# INSERT INTO tbl_city VALUES(3, '城市e');              -- '城市e' 未在列表范围内 将进入默认分区 tbl_
city_def
tdsql=# INSERT INTO tbl_city VALUES(4, '');                   -- '' 表示NULL 将写入 tbl_city_others 分区

/* -- 4. 查看各个分区的数据 -- */
tdsql=# select * from tbl_city partition(tbl_city_super);
 ID | CITY
----+------
  1 | 城市a
  2 | 城市b
(2 rows)

tdsql=# select * from tbl_city partition(tbl_city_others);
 ID | CITY
----+------
  4 | null
(1 row)

tdsql=# select * from tbl_city partition(tbl_city_def);
 ID | CITY
----+------
  3 | 城市e
(1 row)
```

散列分区

散列分区是跨设备均匀分布数据的理想方法，也是范围分区的一种易于使用的替代方案，尤其是当要分区的数据不是

历史数据或没有明显的分区键时。散列分区根据指定分区键通过散列算法将数据映射到指定的分区子表，散列算法在分区之间均匀分布行，使分区大小大致相同。通过为每个分区指定模数和余数来对表进行分区，每个分区所持有的行都满足：分区键的值除以其指定的模数将产生为其指定的余数。
语法：

```
CREATE TABLE table_name (col1, col2, col3 ...) PARTITION BY HASH( col1 );
CREATE TABLE table_name_part_1 PARTITION OF table_name FOR VALUES WITH ( MODULUS x, REMAI
NDER  y);
CREATE TABLE table_name_part_2 PARTITION OF table_name FOR VALUES WITH ( MODULUS x, REMAI
NDER  y);
...
```

> 说明
>
> - 当创建哈希分区时，必须指定模数MODULUS 和余数 REMAINER。模数必须是正整数，余数必须是小于模数的非负整数。
> - 通常情况下，当初始设置哈希分区表时，应选择一个与分区数相等的模数，并为每个表分配相同的模数和不同的余数。
> - 散列分区表不支持 DEFAULT 分区。

示例

```
/* -- 1. 创建分区表 -- */
tdsql=# create table tbl_hash(id integer, product integer) partition by hash (id);
tdsql=# create table tbl_hash_part_0 partition of tbl_hash for values with(modulus 2, remainder 0);
tdsql=# create table tbl_hash_part_1 partition of tbl_hash for values with(modulus 2, remainder 1);

/* -- 2. 查看分区表信息 -- */
tdsql=# \d+ tbl_hash
                              Table "public.TBL_HASH"
 Column  |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
---------+---------+-----------+----------+---------+---------+--------------+-------------
 ID      | INTEGER |           |          |         | plain   |              |
 PRODUCT | INTEGER |           |          |         | plain   |              |
Partition key: HASH (ID)
Partitions: TBL_HASH_PART_0 FOR VALUES WITH (modulus 2, remainder 0),
            TBL_HASH_PART_1 FOR VALUES WITH (modulus 2, remainder 1)
Distribute By: SHARD(ID)
Location Nodes: ALL DATANODES

/* -- 2.1 查看分区子表表信息 -- */
tdsql=# \d+ tbl_hash_part_0
                           Table "public.TBL_HASH_PART_0"
 Column  |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
---------+---------+-----------+----------+---------+---------+--------------+-------------
```

```
 ID      | INTEGER |           |          |         | plain   |              |
 PRODUCT | INTEGER |           |          |         | plain   |              |
Partition of: TBL_HASH FOR VALUES WITH (modulus 2, remainder 0)
Partition constraint: SATISFIES_HASH_PARTITION('17413'::OID, 2, 0, ID)
Distribute By: SHARD(ID)
Location Nodes: ALL DATANODES

/* -- 2.2 查看分区子表表信息 -- */
tdsql=# \d+ tbl_hash_part_1
                        Table "public.TBL_HASH_PART_1"
 Column  |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
---------+---------+-----------+----------+---------+---------+--------------+-------------
 ID      | INTEGER |           |          |         | plain   |              |
 PRODUCT | INTEGER |           |          |         | plain   |              |
Partition of: TBL_HASH FOR VALUES WITH (modulus 2, remainder 1)
Partition constraint: SATISFIES_HASH_PARTITION('17413'::OID, 2, 1, ID)
Distribute By: SHARD(ID)
Location Nodes: ALL DATANODES

/* -- 3. 验证哈希分区 -- */
tdsql=# insert into tbl_hash select generate_series(1,1000),generate_series(1,1000);

/* -- 4. 查看各个分区数据 -- */
tdsql=# select "tableoid"::regclass, count(*) from tbl_hash group by "tableoid" order by 1;
    tableoid     | COUNT
-----------------+-------
 TBL_HASH_PART_0 |   535
 TBL_HASH_PART_1 |   465
(2 rows)
```

多级分区

一种结合两种或多种分区方法的分区技术，TDSQL PG支持一级分区为RANGE、LIST，二级分区为RANGE、LIST、HASH组合。

示例：

两层分区，第一层按列表分区，第二层按范围分区。

```
/* -- 1. 建立分区主表 -- */
tdsql=# CREATE TABLE t_prt_list_range (a varchar2, b int) PARTITION BY LIST(a);

/* -- 2. 建立一级分区子表 -- */
tdsql=# CREATE TABLE t_prt_list_range_p1 PARTITION OF t_prt_list_range FOR VALUES IN ('a') PARTITIO
N BY RANGE(b);
tdsql=# CREATE TABLE t_prt_list_range_p2 PARTITION OF t_prt_list_range FOR VALUES IN ('b');
tdsql=# CREATE TABLE t_prt_list_range_p3 PARTITION OF t_prt_list_range DEFAULT;
```

```
/* -- 3. 建立二级分区子表 -- */
tdsql=# CREATE TABLE t_prt_list_range_p1_p1 PARTITION OF t_prt_list_range_p1 FOR VALUES FROM
(0) TO (200);
tdsql=# CREATE TABLE t_prt_list_range_p1_p2 PARTITION OF t_prt_list_range_p1 FOR VALUES FROM (2
00) TO (400);
tdsql=# CREATE TABLE t_prt_list_range_p1_p3 PARTITION OF t_prt_list_range_p1 DEFAULT;

/* -- 4. 查看分区信息 -- */
tdsql=# \d+ T_PRT_LIST_RANGE
                        Table "public.T_PRT_LIST_RANGE"
 Column |   Type   | Collation | Nullable | Default | Storage  | Stats target | Description
--------+----------+-----------+----------+---------+----------+--------------+-------------
 A      | VARCHAR2 |           |          |         | extended |              |
 B      | INTEGER  |           |          |         | plain    |              |
Partition key: LIST (A)
Partitions: T_PRT_LIST_RANGE_P1 FOR VALUES IN ('a'), PARTITIONED,
            T_PRT_LIST_RANGE_P2 FOR VALUES IN ('b'),
            T_PRT_LIST_RANGE_P3 DEFAULT
Distribute By: SHARD(A)
Location Nodes: ALL DATANODES

/* -- 4.1 查看子分区信息 -- */
tdsql=# \d+ T_PRT_LIST_RANGE_P1
                       Table "public.T_PRT_LIST_RANGE_P1"
 Column |   Type   | Collation | Nullable | Default | Storage  | Stats target | Description
--------+----------+-----------+----------+---------+----------+--------------+-------------
 A      | VARCHAR2 |           |          |         | extended |              |
 B      | INTEGER  |           |          |         | plain    |              |
Partition of: T_PRT_LIST_RANGE FOR VALUES IN ('a')
Partition constraint: ((A IS NOT NULL) AND ((A)::TEXT = 'a'::VARCHAR2))
Partition key: RANGE (B)
Partitions: T_PRT_LIST_RANGE_P1_P1 FOR VALUES FROM (0) TO (200),
            T_PRT_LIST_RANGE_P1_P2 FOR VALUES FROM (200) TO (400),
            T_PRT_LIST_RANGE_P1_P3 DEFAULT
Distribute By: SHARD(A)
Location Nodes: ALL DATANODES

/* -- 5. 验证多级分区 -- */
tdsql=# INSERT INTO t_prt_list_range SELECT 'a', gs FROM generate_series(1,500) gs;
tdsql=# INSERT INTO t_prt_list_range SELECT 'b', gs FROM generate_series(1,500) gs;
tdsql=# INSERT INTO t_prt_list_range SELECT 'c', gs FROM generate_series(1,500) gs;

/* -- 6. 查看各分区数据分布 -- */
tdsql=# select "tableoid"::regclass, count(*) from t_prt_list_range group by "tableoid" order by 1;
       tableoid       | COUNT
----------------------+-------
```

```
 T_PRT_LIST_RANGE_P2   |  500
 T_PRT_LIST_RANGE_P3   |  500
 T_PRT_LIST_RANGE_P1_P1 |  199
 T_PRT_LIST_RANGE_P1_P2 |  200
 T_PRT_LIST_RANGE_P1_P3 |  101
(5 rows)
```

分区表限制

- 无法创建跨越所有分区的排除约束，只能是每个子分区的单个约束。
- 在分区表上支持主键，但不支持引用分区表的外键。
- 当一个UPDATE导致一行从一个分区移动到另一个分区时，另一个并发的UPDATE或DELETE可能会产生一个串行化错误。假设会话1正在执行一个分区键上的UPDATE，同时一个并发的能看见这个行的会话2执行了对该行的UPDATE或者DELETE操作。在这种情况下，会话2的UPDATE或者DELETE会检测到行的移动，并抛出一个串行化的错误(将总是会返回一个SQLSTATE '40001')。 如果发生这种情况，应用程序可能希望重试该事务。在没有分区表或没有行移动的通常情况下，会话2将识别新更新的行并在新行上执行UPDATE/DELETE。
- 不允许在同一分区表中混用临时表和持久表，如果分区表是持久的，则其分区也必须是持久的，反之亦然。在使用临时表时，所有的分区都必须来自于同一个会话。

创建索引

本文介绍如何通过 SQL 命令来创建索引。
TDSQL PG 数据库采用的是聚集索引表模型，对于用户指定的主键，系统会自动生成主键索引。可以根据自身业务需要来决定在哪些字段上创建索引，以便加快在这些字段上的查询速度。

前置条件

- 拥有 CREATE 权限。

索引的限制

- 在 TDSQL PG 数据库中，索引名称必须在表对应的 SCHEMA 内保证唯一。
- 索引名称的长度不能超过 63 字节（超过63个字符表名称会被截断，保留前63个字符）。
- 在一个表中可以创建多个唯一索引，但是每个唯一索引所对应的列值都必须保持唯一。
- 如果一个表带shard key，那么唯一索引必须包含shard key。

索引使用的建议

- 索引名称应该包含对应表名与列名：
  - 主键索引：pk_table_column
  - 唯一索引：uk_table_column
  - 普通索引：idx_table_column
- 单个表索引数量不宜太多，索引数量多会占用大量磁盘空间，同时索引过多，也会增加计算成本。
- 频繁更新的表列不宜建索引。表列的频繁更新会导致索引的频繁更新，增加维护索引成本。
- 选择经常用于查询且区分度高的表列进行创建索引。

创建索引

创建索引语法如下

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
```

[TABLESPACE tablespace_name]
[WHERE predicate]

参数说明:

- UNIQUE
  系统会在索引被创建时（如果数据已经存在）或者插入数据时检查重复值。
- CONCURRENTLY
  一般情况下，索引创建时会把表锁住以阻止对表的写（但不阻塞读），这种锁定会持续到索引创建完毕。当使用了这个选项时，TDSQL PG在构建索引时不会取得任何会阻止该表上并发插入、更新或者删除的锁。
- IF NOT EXISTS
  当索引不存在时才创建，若存在则会给出提示跳过。若使用 IF NOT EXISTS，需要指定索引名。
- name
  要创建的索引名称。这里不能包括模式名，因为索引总是被创建在其基表所在的模式中。若不指定索引名，TDSQL PG将根据表名以及被索引的列名自动分配一个索引名称。
- table_name
  拥有该索引的表 （可以被模式限定）。
- method
  索引类型。可以选择 btree、hash、gist、spgist、gin以及brin。 默认为btree。
- column_name
  一个表列的名称。
- expression
  一个基于一个或者更多个表列的表达式。
- collation
  要用于该索引的排序规则。默认情况下，该索引使用被索引列的排序规则或者被索引表达式的结果排序规则。当查询涉及到使用非默认排序规则的表达式时，使用非默认排序规则的索引就能排上用场。
- opclass
  一个操作符类的名称。
- ASC
  指定上升排序（默认）。
- DESC
  指定下降排序。
- NULLS FIRST
  指定把空值排序在非空值前面。在指定DESC时， 这是默认行为。
- NULLS LAST
  指定把空值排序在非空值后面。在没有指定DESC时， 这是默认行为。
- storage_parameter
  索引方法相关的存储参数的名称。详见下方索引存储参数。
- tablespace_name
  在其中创建索引的表空间。如果没有指定，将会使用 default_tablespace。或者对临时表上的索引使用

temp_tablespaces。

- predicate

  部分索引的约束表达式。

  > 说明:可以使用 SELECT * FROM pg_indexes WHERE tablename = 'table_name'; 语句查看表中索引的信息。

示例

创建唯一索引

如果列上不存在重复的值，可以对该列创建唯一索引。使用以下 SQL 语句创建一个名为 t_test 的表，并为表 t_test 创建一个基于 col2 列的唯一索引。

1. 创建表 t_test 。

```
tdsql=# CREATE TABLE t_test(col1 INT, col2 INT, col3 VARCHAR(50), PRIMARY KEY (col1, col2)) DI
STRIBUTE BY SHARD(col2);
CREATE TABLE
```

2. 在表 t_test 上基于 col2 列创建一个名为 idx_tbl1_col2 的唯一索引。

```
tdsql=# CREATE UNIQUE INDEX idx_tbl1_col2 ON t_test(col2);
CREATE INDEX
```

3. 查看表 t_test 索引信息。

```
tdsql=# SELECT * FROM pg_indexes WHERE tablename = 't_test';
```

返回结果如下：

```
 schemaname | tablename |   indexname   | tablespace |                          indexdef
------------+-----------+---------------+------------
+-------------------------------------------------------------------------
 public     | t_test    | idx_tbl1_col2 |            | CREATE UNIQUE INDEX idx_tbl1_col2 ON public.t_test
USING btree (col2)
 public     | t_test    | t_test_pkey   |            | CREATE UNIQUE INDEX t_test_pkey ON public.t_test U
SING btree (col1, col2)
(2 rows)
```

4. 使用索引扫描

```
tdsql=# insert into t_test select i, i  from generate_series(1, 100000) i;
INSERT 0 100000
tdsql=# analyze t_test;
ANALYZE
tdsql=# explain select * from t_test where col2 = 1;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Remote Fast Query Execution
   Node/s: dn001
   ->  Index Scan using idx_tbl1_col2 on t_test  (cost=0.29..8.31 rows=1 width=126)
         Index Cond: (col2 = 1)
(4 rows)
```

创建非唯一索引

使用以下 SQL 语句创建一个名为 t_test 的表，并为表 t_test 创建一个基于 col2 列的索引。

1. 创建表 t_test 。

```
tdsql=# CREATE TABLE t_test(col1 INT, col2 INT, col3 VARCHAR(50), PRIMARY KEY (col1, col2));
CREATE TABLE
```

2. 在表 t_test 上基于 col2 列创建一个名为 idx_tbl2_col2 的索引。

```
tdsql=# CREATE INDEX idx_tbl2_col2 ON t_test(col2);
CREATE INDEX
```

3. 查看表 t_test 索引信息。

```
tdsql=# SELECT * FROM pg_indexes WHERE tablename = 't_test';
```

返回结果如下：

```
 schemaname | tablename |   indexname   | tablespace |                                indexdef
------------+-----------+---------------+------------+
+-------------------------------------------------------------------------
 public     | t_test    | idx_tbl2_col2 |            | CREATE INDEX idx_tbl2_col2 ON public.t_test USING b
tree (col2)
 public     | t_test    | t_test_pkey   |            | CREATE UNIQUE INDEX t_test_pkey ON public.t_test U
```

```
SING btree (col1, col2)
(2 rows)
```

4. 使用索引扫描

```
tdsql=# insert into t_test select i, i from generate_series(1,100000) i;
INSERT 0 100000
tdsql=# analyze t_test;
ANALYZE
tdsql=# explain select * from t_test where col2 = 1;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Remote Fast Query Execution
   Node/s: dn001, dn002
   ->  Index Scan using idx_tbl2_col2 on t_test  (cost=0.29..8.31 rows=1 width=126)
         Index Cond: (col2 = 1)
(4 rows)
```

创建空间索引

空间索引是一种用于处理和优化空间数据的数据库索引。它被广泛应用于地理信息系统（GIS）和位置数据的存储和查询, TDSQL PG 支持的空间索引类型有： gist， spgist， brin 。
使用以下 SQL 语句创建一个名为 t_test 的表，并为表 t_test 创建一个基于 c3 列的gist空间索引。

1. 创建表 t_test 。

```
tdsql=#  CREATE TABLE t_test(c1 int,c2 int, c3 box, c4 box);
CREATE TABLE
```

2. 在表 t_test 上基于 c3 列创建一个名为 idx_test_c3 的空间索引。

```
tdsql=# CREATE INDEX idx_test_c3 on t_test USING gist(c3);
CREATE INDEX
```

3. 查看表 t_test 索引信息。

```
tdsql=# SELECT * FROM pg_indexes WHERE tablename = 't_test';
```

返回结果如下：

```
tdsql=# SELECT * FROM pg_indexes WHERE tablename = 't_test';
 schemaname | tablename |  indexname  | tablespace |                    indexdef
------------+-----------+-------------+------------
+-------------------------------------------------------
 public     | t_test    | idx_test_c3 |            | CREATE INDEX idx_test_c3 ON public.t_test USING gist
(c3)
(1 row)
```

4. 使用索引扫描

```
tdsql=# insert into t_test select i, i, box(point(0.05*i, 0.05*i), point(0.05*i, 0.05*i)) from generate_series
(1,100000) i;
INSERT 0 100000
tdsql=# analyze t_test;
ANALYZE
tdsql=# explain select * from t_test where c3 <@ box(point(0,0), point(0.5, 0.5));
                              QUERY PLAN
-------------------------------------------------------------------------------
 Remote Fast Query Execution
   Node/s: dn001, dn002
   ->  Bitmap Heap Scan on t_test  (cost=4.67..155.43 rows=50 width=72)
         Recheck Cond: (c3 <@ '(0.5,0.5),(0,0)'::box)
         ->  Bitmap Index Scan on idx_test_c3  (cost=0.00..4.65 rows=50 width=0)
               Index Cond: (c3 <@ '(0.5,0.5),(0,0)'::box)
(6 rows)
```

创建函数索引

基于表中一列或多列的值进行计算后的结果建立的索引称为函数索引。函数索引是一种优化技术，使用函数索引可以在查询时快速定位匹配的函数值，从而避免重复计算，提高查询效率。

使用以下 SQL 语句创建一个名为 t_test 的表，并为表 t_test 创建一个基于 c_time 列的函数索引。

1. 创建表 t_test 。

```
tdsql=# CREATE TABLE t_test(id INT, name VARCHAR(18), c_time DATE);
CREATE TABLE
```

2. 在表 t_test 上创建一个名为 idx_t_test_c_time 的索引，该索引是基于 c_time 列的年份部分进行创建的。

```
tdsql=# CREATE INDEX idx_t_test_c_time ON t_test(EXTRACT(YEAR FROM c_time));
CREATE INDEX
```

3. 使用下面 SQL 语句可以查看创建的函数索引。

```
tdsql=# SELECT * FROM pg_indexes WHERE tablename = 't_test';
```

返回结果如下：

```
 schemaname | tablename |     indexname     | tablespace |                                                indexdef
------------+-----------+-------------------+------------
+-----------------------------------------------------------------------------------------------
 public     | t_test    | idx_t_test_c_time |            | CREATE INDEX idx_t_test_c_time ON public.t_test
USING btree (date_part('year'::text, c_time))
(1 row)
```

4. 使用索引扫描

```
tdsql=# insert into t_test select i, 'name', '2021-1-1'::date + interval '1 day' from generate_series(1, 100
000) i;
INSERT 0 100000
tdsql=# analyze t_test;
ANALYZE
tdsql=# explain select * from t_test where EXTRACT(YEAR FROM c_time) = 123;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Remote Fast Query Execution
   Node/s: dn001, dn002
   ->  Index Scan using idx_t_test_c_time on t_test  (cost=0.29..4.31 rows=1 width=13)
         Index Cond: (date_part('year'::text, (c_time)::timestamp without time zone) = '123'::double precisi
on)
(4 rows)
```

创建DBLink

TDSQL PG 数据库的 DBLink 功能实现了跨数据源访问的能力，您可以在本地数据库上访问远端数据库。
当前，对于 TDSQL PG 数据库的 Oracle 模式，支持创建：
1、TDSQL PG 数据库到 TDSQL PG 数据库下的 DBLink。
2、TDSQL PG 数据库到 Oracle 数据库的 DBlink。

> 说明：
>
> 仅支持oracle模式的库下支持创建dblink，且源端数据库和目标端数据库都是Oracle模式。

前提条件

创建 DBLink 的用户需要具备 CREATE DATABASE LINK 权限。

创建 TDSQL PG 数据库到 TDSQL PG 数据库的 DBLink

创建oracle模式的数据库db_ora。

```
create database db_ora sql mode oracle;
```

如果要访问远端的 TDSQL PG 数据库，可以在 TDSQL PG Oracle模式下的数据库创建对应的 DBLink。
创建 DBLink 需要指定 DBLink 名称并且提供远端数据库的用户名、密码、IP 地址、端口号及访问类型等信息。
DBLink 的 SQL 语法如下：

```
db_ora=# CREATE DATABASE LINK <dblink_name> CONNECT TO <username> IDENTIFIED BY '<passw
ord>' USING
  '(DESCRIPTION = (HOST = <ip>)(PORT = <port>)(SERVICE_NAME = <database_name>)(SCHEMA_N
AME=<schema_name>))';
```

相关参数说明如下：

- <dblink_name> ：DBLink 的名称，长度不超过 63 个字符。
- <username> ：远端 TDSQL PG 数据库的用户名。
- <password> ：远端 TDSQL PG 数据库用户名的登录密码。
- <ip> ：指定远端TDSQL PG 数据库的 IP 地址。
- <port> ：指定远端 TDSQL PG 数据库的端口号。

- <database_name> ：远端 TDSQL PG 数据库名称。
- <schema_name>：远端TDSQL PG 数据库下的模式。

  示例如下：
- 创建TDSQL PG 数据库连接到远端TDSQL PG 数据库的dblink

```
db_ora=# set dblink_types to 'postgresql';
SET
db_ora=# create extension "postgres_fdw" ;
CREATE EXTENSION
db_ora=# CREATE DATABASE LINK tbase_dblink_ora CONNECT TO <username> IDENTIFIED BY '<pass
word>' USING
'(DESCRIPTION = (HOST = <ip>)(PORT = <port>)(SERVICE_NAME = <database_name>)(SCHEMA_NA
ME=<schema_name>))';
CREATE DATABASE LINK
```

查看已经创建的dblink

```
db_ora=# select * from PG_DBLINK ;
```

表格字段说明如下：

- DBLINKNAME : dblink名称
- DBLINKOWNER : dblink的所有者
- USERNAME : dblink用于连接远端库的用户
- HOST : dblink连接远端库的host信息
- PORT : dblink连接远端库的端口号
- DBLINK_FOREIGN_SERVER : dblink对应的foreign server名称

通过dblink访问、修改远端数据库的数据

向远端数据库的 test_dblink_table 插入一条数据 (1,'tbase dblink test') ，并查询。

> 说明：
>
> 需要先在远端库对应的模式下创建 test_dblink_table 表

```
db_ora=# insert into test_dblink_table@tbase_dblink_ora values (1,'tbase dblink test');
INSERT 0 1
db_ora=# select * from test_dblink_table@tbase_dblink_ora ;
```

```
I |     CONTENT
---+------------------
1 | tbase dblink test
(1 row)
```

删除创建的dblink

删除dblink tbase_dblink_ora 。

```
db_ora=# drop database link tbase_dblink_ora;
 NOTICE:  (00000) drop cascades to 2 other objects
 drop cascades to foreign table ORA_172_17_0_2_1521_U1.TEST_DBLINK_TABLE
 DROP DATABASE LINK
```

创建 TDSQL PG 数据库到Oracle数据库的dblink

如果要访问远端的Oracle数据库，可以在 TDSQL PG Oracle模式下创建对应的 DBLink。
创建 DBLink 需要指定 DBLink 名称并且提供远端Oracle数据库的服务名、用户名、密码、IP 地址、端口号等信息。
DBLink 的 SQL 语法如下：

```
db_ora=# CREATE DATABASE LINK <dblink_name> CONNECT TO <username> IDENTIFIED BY '<passw
ord>' USING
 '(DESCRIPTION = (HOST = <ip>)(PORT = <port>)(SERVICE_NAME = <oracle_service_name>))';
```

相关参数说明如下：

- <dblink_name> ：DBLink 的名称，长度不超过 63 个字符。
- <username> ：远端Oracle数据库的用户名。
- <password> ：远端Oracle数据库用户名的登录密码。
- <ip> ：指定远端Oracle数据库的 IP 地址。
- <port> ：指定远端Oracle 数据库的端口号。
- <oracle_service_name> ：远端Oracle数据库的服务名称。

  示例如下：

  创建到Oracle数据库的Database Link

```
db_ora=# set dblink_types to 'oracle_x86';
SET
db_ora=# create extension "oracle_fdw" ;
CREATE EXTENSION
db_ora=# CREATE DATABASE LINK tbase_dblink_oracle CONNECT TO <username> IDENTIFIED BY '<pa
ssword>' USING
'(DESCRIPTION = (HOST = <ip>)(PORT = <port>)(SERVICE_NAME = <oracle_service_name>))';
CREATE DATABASE LINK
```

通过dblink访问远端Oracle数据库的对象

目前TDSQL PG远端为Oracle数据库的DBLINK支持访问远端Oracle数据库的表/视图/同义词，访问对象为远端同义词时，远端同义词的对象也只能是指代的表或视图

```
db_ora=# SELECT * from oracle_table@tbase_dblink_oracle; --access TABLE in oracle side
db_ora=# SELECT * from oracle_view@tbase_dblink_oracle; --access VIEW in oracle side
db_ora=# SELECT * from oracle_synonym@tbase_dblink_oracle; --access SYNONYM in oracle side
```

删除创建的dblink（远端为Oracle数据库）

删除DBLINK的语法同远端为TDSQL PG数据库的DBLINK

```
db_ora=# drop database link tbase_dblink_oracle;
```

在PG模式下访问远端数据库

在PG模式下，不支持DBLINK语法，如果要访问远端的 TDSQL PG 数据库或postgresql数据库（不支持远端为Oracle数据库），您可以直接使用postgres_fdw插件完成dblink类似的功能。
1.安装插件postgres_fdw
2.创建foreign data server，语法如下:

```
CREATE SERVER server_name FOREIGN DATA WRAPPER wrapper_name OPTIONS(opt_list);
```

相关参数说明如下：

- server_name：要创建的外部服务器（foreign server）名称

- wrapper_name : data wrapper名称，如: postgres_fdw
- opt_list: 可选项
  - host 'hostip' 指定foreign server连接的远端库ip
  - port 'portnumber' 指定foreign server连接的远端库端口号
  - dbname 'dbname' 指定foreign server连接的远端库名

  示例如下：

```
--安装postgres_fdw插件
db1=# create extension postgres_fdw;
CREATE EXTENSION

db1=# create server sv1 foreign data wrapper postgres_fdw options(host 'hostip',port'portnumber',db
name'dbname'); --创建foreign server
CREATE SERVER
```

3.创建foreign table对应远端数据库的table，语法如下:

```
CREATE FOREIGN TABLE table_name(column_list) SERVER server_name(opt_list)
```

相关参数说明:

- table_name: 表名，本地操作库的表名，之后用此表和远端库表建立关联。
- column_list: 列信息，必须和远端库的表的列信息对应。
- server_name: 要使用的已存在的外部服务器（foreign server）名
- opt_list: 可选项
  - schema_name 'schema' 对应远端库的模式名，必须一致。
  - table_name 'table' 对应远端库的表名，必须一致。

  4.创建user mapping用于访问远端数据库的用户名和密码，语法如下:

```
CREATE USER MAPPING FOR user_name SERVER server_name options(opt_list)
```

相关参数说明：

- user_name: 本地操作库的用户名
- server_name: 要使用的已存在的外部服务器（foreign server）名
- opt_list:可选项
  - user 'user_name': 对应远端库的用户名
  - password 'passwd': 对应远端库的用户登录密码
- 示例：

```
db1=# create foreign table test_fdw_tbase(i int, value text) server sv1 options(schema_name'public', ta
```

ble_name'test_fdw_tbase')--创建foreign table，将本地的表test_fdw_tbase和远端库的表public.test_fdw_tbase建立关联;
CREATE FOREIGN TABLE

db1=# create user MAPPING FOR tbase server sv1 options (user'user_name',password'passwd')--创建user mapping;
CREATE USER MAPPING

使用foreign table对远端数据库的表进行修改或查询

```
db1=# select * from test_fdw_tbase;
 i | value
---+-------
(0 rows)

db1=# insert into test_fdw_tbase values (1,'tbase fdw test');
INSERT 0 1
db1=# select * from test_fdw_tbase;
 i |     value
---+----------------
 1 | tbase fdw test
(1 row)
```

删除对应的foreign table

```
db1=# drop foreign table test_fdw_tbase;
DROP FOREIGN TABLE
```

删除对应的foreign server和user mapping

```
db1=# drop server sv1 cascade;
NOTICE:  (00000) drop cascades to user mapping for tbase on server sv1
DROP SERVER
```

数据写入
插入数据

数据库和数据表创建完成后，可以使用 INSERT INTO ... 语句向表中插入行记录。
INSERT 用于向一张表中插入数据，期望插入的数据可以是一条，多条或者是一个select查询的结果集。
语法如下：

```
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
```

参数说明：

- table_name : 一个已有表的名称（可以被模式限定）。
- alias : table_name 的替补名称。当提供了一个别名时，它会完全隐藏掉表的实际名称。
- column_name : table_name 表中的列名。如果需要，可以使用子字段名或数组下标限定列名。（如果只向复合列的某些字段插入数据，则其他字段将为空）。
- DEFAULT VALUES : 所有列都将被其默认值填充。
- expression : 要赋予给相应列的表达式或者值。
- DEFAULT : 相应的列将被其默认值填充。
- query : 提供要被插入行的查询（SELECT语句）。 其语法描述请参考SELECT语句。
  插入数据前要注意：
- 插入数据前，建议通过 \d table_name 语句查看表结构及列信息，包括数据类型、取值范围、列约束（如 NOT NULL 、 PRIMARY KEY 、 UNIQUE 、 FOREIGN KEY 、 CHECK 等）等，避免插入数据时报错。

```
tdsql=#\d tbase_insert;
          Table "public.tbase_insert"
Column   |  Type   | Collation | Nullable |  Default
-----------+---------+-----------+----------+------------
id        | numeric |           | not null |
name      | text    |           | not null |
value     | numeric |           |          |
create_ts | date    |           | not null | CURRENT_DATE
```

```
Indexes:
"tbase_insert_pkey" PRIMARY KEY, btree (id)
"uniq_idx_value" UNIQUE, btree (value)
```

- 如果列属性 Nullable 为 not null ，即不允许为 NULL 时，如果列属性 Default 有配置默认值，则可以在插入时不指定该列的值，系统会在该列上插入配置好的默认值。否则必须指定该列的值，不然会报错。
- 如果列属性 Nullable 为 NULL ，则可以在插入时不指定该列的值，系统会在该列上插入一个 NULL 值。

示例

通过 INSERT 语句可以插入单条数据。可以执行多个单条插入语句来插入多条数据。
表结构如下：

```
CREATE TABLE tbase_insert(
   id numeric NOT NULL PRIMARY KEY,
   name text NOT NULL,
   value numeric,
   create_ts DATE NOT NULL DEFAULT CURRENT_DATE
);
```

其中，表的 id 列、 name 列不能为空，且 id 列为主键，满足唯一性约束要求，不能有重复的值； create_ts 列指定了获取当前时间作为默认值。
1、使用多个单条插入语句插入多行数据。
由于 create_ts 列指定了默认值，在插入数据时可以不指定该列。

```
tdsql=# set nls_date_format to 'YYYY-MM-DD';
SET
tdsql=# INSERT INTO tbase_insert(id, name, value) VALUES (1,'tbase',100);
INSERT 0 1
tdsql=# INSERT INTO tbase_insert(id, name, value) VALUES (2,'tbase',101);
INSERT 0 1
```

> 注意：
>
> 如果 create_ts 列未指定默认值，则在插入数据时，必须指定值。

```
insert into tbase_insert values(3 ,'tbase', 102, '1999-1-1');
```

2、如果要批量插入多条记录，也可以用一个 INSERT 语句包含多个 VALUES 的内容来批量插入。批量插入语句比多个单条插入语句要快。

```
tdsql=# insert into tbase_insert values(4,'TDSQL PG', 103),(5,'TDSQL PG好', 104);
INSERT 0 2
tdsql=# select * from tbase_insert;
id |    name    | value | create_ts
----+------------+-------+------------
  1 | tbase      |   100 | 2024-03-02
  2 | tbase      |   101 | 2024-03-02
  3 | tbase      |   102 | 1999-01-01
  4 | TDSQL PG   |   103 | 2024-03-02
  5 | TDSQL PG好 |   104 | 2024-03-02
(5 rows)
```

此外，当需要备份表数据或者将一个表的全部记录复制到另一个表时，可以使用查询语句 INSERT INTO ... SELECT ... FROM 进行批量插入。

3、将表 tbase_insert 中的全部数据备份到 tbase_insert_bak 表中。

```
tdsql=# select * from tbase_insert;
id |    name    | value | create_ts
----+------------+-------+------------
  1 | tbase      |   100 | 2024-03-02
  2 | tbase      |   101 | 2024-03-02
  3 | tbase      |   102 | 1999-01-01
  4 | TDSQL PG   |   103 | 2024-03-02
  5 | TDSQL PG好 |   104 | 2024-03-02
(5 rows)

tdsql=# CREATE TABLE tbase_insert_bak(
        id numeric NOT NULL PRIMARY KEY,
        name text NOT NULL,
        value numeric,
        create_ts DATE NOT NULL DEFAULT CURRENT_DATE
);
CREATE TABLE

tdsql=# INSERT INTO tbase_insert_bak SELECT * FROM tbase_insert;
INSERT 0 5

tdsql=# select * from tbase_insert_bak;
id |    name    | value | create_ts
----+------------+-------+------------
  1 | tbase      |   100 | 2024-03-02
  2 | tbase      |   101 | 2024-03-02
  3 | tbase      |   102 | 1999-01-01
  4 | TDSQL PG   |   103 | 2024-03-02
  5 | TDSQL PG好 |   104 | 2024-03-02
```

```
(5 rows)
```

4、返回插入的数据。

```
tdsql=# create table ins(i int, j int);
CREATE TABLE
tdsql=# insert into ins values (1,1) returning *;
 i | j
---+---
 1 | 1
(1 row)

INSERT 0 1
```

5、 使用insert all一次性往多个表插入数据(oracle库下特有语法)

```
---将数据(1,1)插入ins1和ins2两张表
db_ora=# create table ins1(i int, j int);
CREATE TABLE
db_ora=# create table ins2(i int, j int);
CREATE TABLE
db_ora=# insert all into ins1 values (f1,f2) into ins2 values(f1,f2) select 1 as f1,1 as f2;
INSERT 0 2
db_ora=# select * from ins1;
I | J
---+---
 1 | 1
(1 row)

db_ora=# select * from ins2;
I | J
---+---
 1 | 1
(1 row)
```

使用INSERT...ON CONFLICT方式进行数据更新

在INSERT插入数据时候，如果存在主键或唯一索引冲突，可以使用ON CONFLICT DO UPDATE方式选择更新冲突部分数据，无冲突部分的数据将会正常插入。
示例，使用ON CONFLICT UPDATE更新数据,示例中插入id为3的value存在冲突，将name值更新为 'TDSQL PG Good'， id值为6的value正常插入：

```
tdsql=# select * from tbase_insert;
```

```
 id |    name    | value | create_ts
----+------------+-------+------------
  1 | tbase      |   100 | 2024-03-02
  2 | tbase      |   101 | 2024-03-02
  3 | tbase      |   102 | 1999-01-01
  4 | TDSQL PG   |   103 | 2024-03-02
  5 | TDSQL PG好 |   104 | 2024-03-02
(5 rows)

tdsql=# insert into tbase_insert values (3,'TDSQL PG Good', 102),(6,'tbase', 105) on conflict(id) do upda
te set name=excluded.name;
INSERT 0 2
tdsql=# select * from tbase_insert;
 id |     name      | value | create_ts
----+---------------+-------+------------
  1 | tbase         |   100 | 2024-03-02
  2 | tbase         |   101 | 2024-03-02
  4 | TDSQL PG      |   103 | 2024-03-02
  5 | TDSQL PG好    |   104 | 2024-03-02
  3 | TDSQL PG Good |   102 | 1999-01-01
  6 | tbase         |   105 | 2024-03-02
(6 rows)
```

可以看到 id 值为 3 的 name 被修改为了 TDSQL PG Good，增加了 id 为 6 的一条数据。

如果存在冲突也可以使用 ON CONFLICT DO NOTHING 选择冲突的 value 不做处理，也不报错。

```
tdsql=# select * from tbase_insert;
 id |     name      | value | create_ts
----+---------------+-------+------------
  1 | tbase         |   100 | 2024-03-02
  2 | tbase         |   101 | 2024-03-02
  4 | TDSQL PG      |   103 | 2024-03-02
  5 | TDSQL PG好    |   104 | 2024-03-02
  3 | TDSQL PG Good |   102 | 1999-01-01
  6 | tbase         |   105 | 2024-03-02
(6 rows)
tdsql=# insert into tbase_insert values (3,'TDSQL PG nice', 102) on conflict(id) do nothing;
INSERT 0 0
tdsql=# select * from tbase_insert;
 id |     name      | value | create_ts
----+---------------+-------+------------
  1 | tbase         |   100 | 2024-03-02
  2 | tbase         |   101 | 2024-03-02
  4 | TDSQL PG      |   103 | 2024-03-02
  5 | TDSQL PG好    |   104 | 2024-03-02
  3 | TDSQL PG Good |   102 | 1999-01-01
```

```
 6 | tbase     |  105 | 2024-03-02
(6 rows)
```

可以看到 id 值为 3 的 name 没有被更新。

更新数据

表中插入数据后，可以使用 UPDATE ... SET ... 语句更新表的行记录。
UPDATE 用于更新表数据，可以更新一个字段或者多个字段。
语法如下：

```
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    SET { column_name = { expression | DEFAULT } |
          ( column_name [, ...] ) = [ ROW ] ( { expression | DEFAULT } [, ...] ) |
          ( column_name [, ...] ) = ( sub-SELECT )
        } [, ...]
    [ FROM from_list ]
    [ WHERE condition ]
```

参数说明：

- table_name : 要更新的表的名称（可以是模式限定的）。如果在表名前指定了 ONLY ，只会更新所提及表中的匹配行。如果没有指定 ONLY ，会递归地更新所提及表以及子表中所匹配的行。
- alias : 目标表的一个替代名称。在提供了一个别名时，它会完全隐藏表的真实名称。例如，给定 UPDATE foo AS f ， UPDATE 语句的剩余部分必须用 f 而不是 foo 来引用该表。
- column_name : table_name 所指定的表的一列的名称。如果需要，该列名可以用一个子域名称或者数组下标限定。不要在目标列的说明中包括表的名称，例如 UPDATE table_name SET table_name.col = 1 是非法的。
- expression : 要被赋值给该列的一个表达式。该表达式可以使用该表中这一列或者其他列的旧值。
- DEFAULT : 将该列设置为它的默认值（如果没有为它指定默认值表达式，默认值将会为 NULL ）。
- sub-SELECT : 一个 SELECT 子查询，它产生和在它之前的圆括号中列列表中一样多的输出列。被执行时，该子查询必须得到不超过一行。如果它得到一行，其列值会被赋予给目标列。如果它得不到行， NULL 值将被赋予给目标列。该子查询可以引用被更新表中当前行的旧值。
- from_list : 表表达式的列表，允许来自其他表的列出现在 WHERE 条件和更新表达式中。这类似于可以在 SELECT 语句的 FROM 子句中指定的表列表。注意目标表不能出现在 from_list 中，除非你想做自连接（这种情况下它必须以别名出现在 from_list 中）。
- condition : 一个返回 boolean 类型值的表达式。让这个表达式返回 true 的行将会被更新。

示例

tbase_insert 表结构如下：

```
tdsql=#\d tbase_insert;
          Table "public.tbase_insert"
Column   |  Type   | Collation | Nullable |    Default
```

```
-----------+---------+-----------+----------+------------
id        | numeric |           | not null |
name      | text    |           | not null |
value     | numeric |           |          |
create_ts | date    |           | not null | CURRENT_DATE
Indexes:
"tbase_insert_pkey" PRIMARY KEY, btree (id)
"uniq_idx_value" UNIQUE, btree (value)
```

1、更新 tbase_insert 表中 create_ts 列的所有值。

```
tdsql=# set nls_date_format to 'YYYY-MM-DD';
SET
tdsql=# select * from tbase_insert;
id |     name      | value | create_ts
----+---------------+-------+------------
 1 | tbase         |   100 | 2024-03-02
 2 | tbase         |   101 | 2024-03-02
 4 | TDSQL PG      |   103 | 2024-03-02
 5 | TDSQL PG好    |   104 | 2024-03-02
 3 | TDSQL PG Good |   102 | 1999-01-01
 6 | tbase         |   105 | 2024-03-02
(6 rows)

tdsql=# UPDATE tbase_insert SET create_ts = '2022-1-1';
UPDATE 6

tdsql=# select * from tbase_insert;
id |     name      | value | create_ts
----+---------------+-------+------------
 1 | tbase         |   100 | 2022-01-01
 2 | tbase         |   101 | 2022-01-01
 4 | TDSQL PG      |   103 | 2022-01-01
 5 | TDSQL PG好    |   104 | 2022-01-01
 3 | TDSQL PG Good |   102 | 2022-01-01
 6 | tbase         |   105 | 2022-01-01
(6 rows)
```

2、 更新部分记录。

在更新表数据时，要注意控制更新量不要太大，否则可能会产生大事务导致执行失败。可以只更新部分记录，通过添加 WHERE 关键字来控制范围。

对 value 值为 100 的所有列，将其值更新为 value+10

```
tdsql=# UPDATE tbase_insert SET value = value+10 where value = 100;
UPDATE 1
```

```
tdsql=# select * from tbase_insert;
 id |     name      | value | create_ts
----+---------------+-------+------------
 2 | tbase         |   101 | 2022-01-01
 4 | TDSQL PG      |   103 | 2022-01-01
 5 | TDSQL PG好    |   104 | 2022-01-01
 3 | TDSQL PG Good |   102 | 2022-01-01
 6 | tbase         |   105 | 2022-01-01
 1 | tbase         |   110 | 2022-01-01
(6 rows)
```

tbase_insert 表中 更新 id = 3 所在行的 name 列的值为 tbase ok 。

```
tdsql=# UPDATE tbase_insert SET name = 'tbase ok' WHERE id = 3;
UPDATE 1

tdsql=# select * from tbase_insert;
id |    name    | value | create_ts
----+------------+-------+------------
2 | tbase      |   101 | 2022-01-01
4 | TDSQL PG   |   103 | 2022-01-01
5 | TDSQL PG好 |   104 | 2022-01-01
6 | tbase      |   105 | 2022-01-01
1 | tbase      |   110 | 2022-01-01
3 | tbase ok   |   102 | 2022-01-01
(6 rows)
```

3、 唯一性约束冲突。

当表上有唯一性约束时，更新为相同的记录，系统会报错，如， tbase_insert 表中 id 为主键。

```
tdsql=# select * from tbase_insert;
id |    name    | value | create_ts
----+------------+-------+------------
 2 | tbase      |   101 | 2022-01-01
 4 | TDSQL PG   |   103 | 2022-01-01
 5 | TDSQL PG好 |   104 | 2022-01-01
 6 | tbase      |   105 | 2022-01-01
 1 | tbase      |   110 | 2022-01-01
 3 | tbase ok   |   102 | 2022-01-01
(6 rows)

tdsql=# UPDATE tbase_insert SET id=3 where value = 105;
ERROR:  (23505) duplicate key value violates unique constraint "tbase_insert_pkey"
DETAIL:  Key (id)=(3) already exists.
```

> 说明：

> 在分布式TDSQL PG中，首列默认为分布键，因此报错信息为：
>
> ERROR: (42P10) Distributed column "ID" can't be updated in current version

将 value = 105 这一行记录的 id 改为 3，会和当前 id = 3 的记录冲突。

4、 多表关联更新数据。

示例中找到 update_table 中 i 列和 src_table 中 i 列相同的记录，将 update_table 的 j 列值更新为 src_table 对应记录的 j 列值。

```
tdsql=# create table update_table(i int, j int);
CREATE TABLE
tdsql=# insert into update_table values (1,1);
INSERT 0 1
tdsql=# create table src_table(i int, j int);
CREATE TABLE
tdsql=# insert into src_table values (1,2);
INSERT 0 1
tdsql=# select * from update_table;
 i | j
---+---
 1 | 1
(1 row)
tdsql=# update update_table set j=src_table.j from src_table where update_table.i=src_table.i;
UPDATE 1
tdsql=# select * from update_table;
i | j
---+---
 1 | 2
(1 row)
```

5、 update返回更新的数据

通过returning返回更新结果，可以减少与数据库的交互次数。示例：

```
tdsql=# update update_table set j=3 returning *;
 i | j
---+---
 1 | 3
(1 row)

UPDATE 1
```

删除数据

表中插入数据后，可以使用 DELETE 语句来删除表中的记录。
语法如下：

```
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING from_item [, ...] ]
    [ WHERE condition ]
```

参数说明：

- table_name : 要从其中删除行的表名（可以是模式限定的）。如果在表名前指定 ONLY ，只会从提到的表中删除匹配的行。如果没有指定 ONLY ，还会删除该表的任何继承表中的匹配行。可选地，可以在表名后面指定 *来显式指定要包括继承表。
- alias : 目标表的一个别名。提供别名时，它会完全隐藏该表的真实名称。例如，对于 DELETE FROM foo AS f ， DELETE 语句的剩余部分都会用 f 而不是 foo 来引用该表。
- from_item : 一个表的表达式列表，它允许在WHERE条件中出现来自其他表的列。这和SELECT语句的 FROM 子句中指定的表列表相似。例如，可以指定表的别名。除非想要进行自连接，否则不要在 from_item 再写上目标表。
- condition : 一个返回 boolean 类型值的表达式。只有让这个表达式返回 true 的行才将被删除。

示例

1、 删除部分数据
通过 DELETE 语句加 WHERE 条件，可以删除表中满足条件的部分数据。

```
tdsql=#\d tbase_insert;
          Table "public.tbase_insert"
Column   |  Type   | Collation | Nullable |  Default
-----------+---------+-----------+----------+------------
id        | numeric |           | not null |
name      | text    |           | not null |
value     | numeric |           |          |
create_ts | date    |           | not null | CURRENT_DATE
Indexes:
"tbase_insert_pkey" PRIMARY KEY, btree (id)
"uniq_idx_value" UNIQUE, btree (value)
```

删除 tbase_insert 表中符合 value = 101 的所有行记录，示例如下：

```
tdsql=#  DELETE FROM tbase_insert WHERE value = 101;
```

```
DELETE 1
```

筛选表 tbase_insert 中 value 列数据，分别执行多条语句分批删除 value < 100 ， value < 200 ， value < 300 的数据。

```
tdsql=# DELETE FROM tbase_insert WHERE value < 100;

tdsql=# DELETE FROM tbase_insert WHERE value < 200;

tdsql=# DELETE FROM tbase_insert WHERE value < 300;
```

2、 删除所有数据

当表中的数据量较小时，可以直接使用 DELETE 语句删除所有行记录。

删除 tbase_insert 表中的所有行数据。

```
tdsql=# DELETE FROM tbase_insert;
```

3、返回删除的数据

```
tdsql=# create table del1 (i int, j int);
CREATE TABLE
tdsql=# insert into del1 values (1,1);
INSERT 0 1
tdsql=# delete from del1 returning *;
 i | j
---+---
 1 | 1
(1 row)

DELETE 1
```

当表中数据记录过多，通过 DELETE 执行全部删除可能会出现性能问题，建议根据表的内容结合 WHERE 条件进行分批删除，或者直接使用 TRUNCATE TABLE 语句清空表数据。

4、多表关联删除数据

示例中找到 del_table 中 i 列和 src_table 中 i 列相同的记录，并将 del_table 中该行记录删除。

```
tdsql=# create table del_table (i int, j int);
CREATE TABLE
tdsql=# insert into del_table values (1,1),(2,2);
COPY 2
tdsql=# select * from del_table;
i | j
---+---
 1 | 1
 2 | 2
```

```
(2 rows)

tdsql=# select * from src_table;
i | j
---+---
 1 | 2
(1 row)
tdsql=# delete from del_table using src_table where del_table.i=src_table.i;
DELETE 1
tdsql=# select * from del_table;
i | j
---+---
 2 | 2
(1 rows)
```

使用 TRUNCATE TABLE 语句清空表数据

TRUNCATE 可以从一组表中快速地移除所有行。 它具有和在每个表上执行无条件 DELETE 相同的 效果，不过它会更快，因为它没有实际扫描表。此外，它会立刻回收磁盘空间， 而不是要求一个后续的 VACUUM 操作。
语法如下：

```
TRUNCATE [TABLE] table_name;
```

示例：

```
tdsql=# TRUNCATE TABLE tbase_insert;
TRUNCATE TABLE
```

替换数据

可以使用 MERGE INTO 语句来插入或者更新一条或多条数据。
语法如下：

```
MERGE INTO [ target_tablename ]
  USING [source_tablename]
  ON ( on_condition )
  [WHEN MATCHED THEN merge_update | merge_delete | DO NOTHING ]
  [WHEN NOT MATCHED THEN merge_insert | DO NOTHING ]
```

参数说明：

- target_tablename ：指定需要进行数据替换的表。
- source_tablename ：指定源数据表。
- on_condition ：指定目标表和源表的ON条件，当ON条件为true，走matched分支，否则走not matched分支。
- merge_update ：merge的update部分语句，matched时候执行，如 update set ... where
- merge_delete ：merge的delete部分语句，matched时候执行，必须跟在merge_update之后使用，如 delete ... where
- merge_insert ：merge的insert部分语句，not matched时候执行，如 insert values (...)

示例

1、替换表中数据，当源表数据和目标表数据不匹配时候进行 insert
当目标表中无数据记录，可以使用 MERGE INTO 语句替换数据，结果等价于通过 INSERT 语句插入数据。
创建 tbase_merge_insert 表后不插入数据，创建源表 tbase_source_table 并插入一行数据，使用 MERGE INTO 语句往目标表插入一行数据，
merge的on条件为源表第一列 i 等于目标表第一列 i ，在不匹配的条件下插入数据。

```
 tdsql=# create table tbase_merge_insert(i int, j int);
CREATE TABLE
 tdsql=# select * from tbase_merge_insert;
i | j
---+---
(0 rows)
 tdsql=# create table tbase_source_table(i int, j int);
CREATE TABLE
 tdsql=# insert into tbase_source_table values (1,1);
INSERT 0 1
 tdsql=# select * from tbase_source_table;
```

```
 i | j
---+---
1 | 1
(1 row)

 tdsql=# merge into tbase_merge_insert using tbase_source_table on tbase_merge_insert.i=tbase_sour
ce_table.i when not matched then insert values (2,2);
MERGE 1
 tdsql=# select * from tbase_merge_insert;
i | j
---+---
2 | 2
(1 row)
```

从示例结果可以看出， tbase_merge_insert 表创建后未插入数据，执行 MERGE INTO 语句后，表中插入了一条记录。

merge的on条件改为目标表第一列 i 不等于源表第一列 i ，在不匹配的条件下插入数据。

```
 tdsql=# select * from tbase_merge_insert;
i | j
---+---
2 | 2
(1 row)

 tdsql=# select * from tbase_source_table;
i | j
---+---
1 | 1
(1 row)
 tdsql=# merge into tbase_merge_insert using tbase_source_table on tbase_merge_insert.i!=tbase_sour
ce_table.i when not matched then insert values (3,3);
MERGE 0
 tdsql=# select * from tbase_merge_insert;
i | j
---+---
2 | 2
(1 row)
```

从示例结果可以看出， tbase_merge_insert 表中已有一条记录 (2,2) ，由于源表中的记录 (1,1) 满足 1 != 2 ,此时 on 结果为true，但后续判断是在 when not matched 插入，所以无法往目标表插入新数据。

2、 替换表中数据，当源表数据和目标表数据*匹配*时候进行update。

创建目标表 tbase_merge_update ，插入一条记录 (1,1) ，此时清空源表，并往源表 tbase_source_table 也插入一条数据(1,1)，merge语句的on条件改回为第一列 i 相等的条件。 再使用 update 去更新 tbase_merge_update 的记录的 j 列值为3。

```
 tdsql=# select * from tbase_source_table;
i | j
---+---
1 | 1
(1 row)

 tdsql=# create table tbase_merge_update (i int, j int);
CREATE TABLE
 tdsql=# insert into tbase_merge_update values (1,1);
INSERT 0 1
 tdsql=# select * from tbase_merge_update;
i | j
---+---
1 | 1
(1 row)
 tdsql=# merge into tbase_merge_update using tbase_source_table on tbase_source_table.i= tbase_me
rge_update.i when matched then update set j=3 ;
MERGE 1
 tdsql=# select * from tbase_merge_update;
i | j
---+---
1 | 3
(1 row)
```

从示例结果中可以看出， merge 的 on 条件为 true ，此时做 update 目标表将记录 (1,1) 更新为了 (1,3)
update 还可以加上 where 条件做更多的限制，这里我们基于上一步的结果去做 merge ， merge 语句 on 条件不变， update 之后加上 where 进行更多限制。

```
 tdsql=# select * from tbase_merge_update;
i | j
---+---
1 | 3
(1 row)

 tdsql=# select * from tbase_source_table;
i | j
---+---
1 | 1
(1 row)
 tdsql=# merge into tbase_merge_update using tbase_source_table on tbase_source_table.i= tbase_me
rge_update.i when matched then update set j=4 where tbase_merge_update.i > 2 ;
MERGE 0
 tdsql=# select * from tbase_merge_update;
i | j
---+---
```

```
 1 | 3
 (1 row)
```

结果中可以看出， update 之后 where 限制了当 tbase_merge_update 表的i列大于2时候才更新，因此 merge 结果为0行， tbase_merge_update 表中没有记录更新。

3、 替换表中数据，当源表数据和目标表数据*匹配*时候进行delete。

使用 MERGE INTO 语句的 delete 功能删除目标表中的记录，创建目标表 tbase_merge_delete ，插入一条记录 (1,1)，此时清空源表，并往源表 tbase_source_table 也插入一条数据 (1,1)，merge 语句的 on 条件改回为第一列 i 相等的条件。

```
 tdsql=# delete from tbase_source_table;
DELETE 1
 tdsql=# insert into tbase_source_table values(1,1);
INSERT 1
 tdsql=# select * from tbase_source_table;
i | j
---+---
1 | 1
(1 row)
 tdsql=# create table tbase_merge_delete (i int, j int);
CREATE TABLE
 tdsql=# insert into tbase_merge_delete values (1,1);
INSERT 0 1
 tdsql=# select * from tbase_merge_delete;
i | j
---+---
1 | 1
(1 row)
 tdsql=# merge into tbase_merge_delete using tbase_source_table on tbase_source_table.i= tbase_mer
ge_delete.i when matched then update set j=10 delete where j=1;
MERGE 1
 tdsql=# select * from tbase_merge_delete;
i | j
---+---
(0 rows)
```

从示例中可以看出， merge 的执行后将目标表 tbase_merge_delete 中的记录删除了(和Oracle实现有差异，TDSQL PG delete中的j列默认指代源表的j列)

4、 替换表中数据，在merge功能中同时使用insert、update去修改目标表。

创建新的目标表 tbase_merge_insup ,插入一行记录 (2,2,'update') ,清空源表数据，再往源表插入 (2,2) 和 (1,1) 两条记录。

```
 tdsql=# create table tbase_merge_insup(i int, j int, op text);
CREATE TABLE
 tdsql=# insert into tbase_merge_insup values (2,2,'update');
```

```
INSERT 0 1
 tdsql=# delete from tbase_source_table;
 DELETE 1
 tdsql=# insert into tbase_source_table values(1,1),(2,2);
 INSERT 2
 tdsql=# select * from tbase_source_table;
i | j
---+---
1 | 1
2 | 2
(2 rows)

 tdsql=# select * from tbase_merge_insup;
i | j |   op
---+---+--------
2 | 2 | update
(1 row)
 tdsql=# merge into tbase_merge_insup using tbase_source_table on tbase_source_table.i = tbase_mer
ge_insup.i when not matched then insert values (1,1,'insert') when matched then update set op='tbase
_update';
MERGE 2
 tdsql=# select * from tbase_merge_insup;
i | j |      op
---+---+--------------
1 | 1 | insert
2 | 2 | tbase_update
(2 rows)
```

通过以上的例子可以看出，源表的记录 (1,1) 走到not matched分支，往目标表插入数据 (1,1,insert) ,源表记录 (2,2) 走到matched分支，更新目标表插入的记录 (2,2,update) 为 (2,2,tbase_update) 。

5、 替换表中数据，在merge功能中同时使用insert、delete去修改目标表

创建新的目标表 tbase_merge_insdel ,插入一行记录 (2,2,'delete') ,清空源表数据，再往源表插入 (2,2) 和 (1,1) 两条记录。

```
 tdsql=# create table tbase_merge_insdel(i int, j int, op text);
CREATE TABLE
 tdsql=# insert into tbase_merge_insdel values (2,2,'delete');
INSERT 0 1
 tdsql=# select * from tbase_merge_insdel;
i | j |   op
---+---+--------
2 | 2 | delete
(1 row)
 tdsql=# delete from tbase_source_table;
 DELETE 2
 tdsql=# insert into tbase_source_table values(1,1),(2,2);
```

```
 INSERT 2
 tdsql=# select * from tbase_source_table;
i | j
---+---
1 | 1
2 | 2
(2 rows)
 tdsql=# merge into tbase_merge_insdel using tbase_source_table on tbase_source_table.i = tbase_mer
ge_insdel.i when not matched then insert values (1,1,'insert') when matched then update set j=10 delet
e where j=2;
MERGE 2
 tdsql=# select * from tbase_merge_insdel;
i | j |   op
---+---+--------
 1 |  1 | insert
(2 rows)
```

通过以上的例子可以看出，源表的记录 (1,1) 走到not matched分支，往目标表插入数据 (1,1,insert) ,源表记录 (2,2) 走到matched分支，删除了目标表插入的记录 (2,2,delete) 。(和Oracle实现有差异，TDSQL PG delete中的j列默认指代源表的j列)

6、 oracle兼容模式下，使用merge替换表数据，可以同时使用insert、update、delete修改目标表数据

创建新的目标表 tbase_merge_all ,插入两行记录 (2,2,'delete') , (2,2,update) ，清空源表数据，再往源表插入 (2,2) 和 (1,1) 两条记录。

```
 tdsql=# create table tbase_merge_all(i int, j int, op text);
CREATE TABLE
 tdsql=# insert into tbase_merge_all values(2,2,'delete');
INSERT 0 1
 tdsql=# insert into tbase_merge_all values(2,2,'update');
INSERT 0 1
 tdsql=# select * from tbase_merge_all;
i | j |   op
---+---+--------
2 | 2 | delete
2 | 2 | update
(3 rows)
 tdsql=# delete from tbase_source_table;
 DELETE 2
 tdsql=# insert into tbase_source_table values(1,1),(2,2);
 INSERT 2
 tdsql=# select * from tbase_source_table;
i | j
---+---
1 | 1
2 | 2
(2 rows)
```

```
 tdsql=# merge into tbase_merge_all using tbase_source_table on tbase_source_table.i = tbase_merge_
all.i when not matched then insert values (1,1,'insert') when matched then update set op='tbase_updat
e' where op='update' delete where op='tbase_update' or op='delete';
MERGE 2
 tdsql=# select * from tbase_merge_all;
i | j |   op
---+---+--------
2 | 2 | delete
1 | 1 | insert
(2 rows)
```

通过以上的例子可以看出：

- 源表的记录 (1,1) 走到not matched分支，往目标表插入数据 (1,1,insert)，
- 源表记录 (2,2) 走到matched分支，匹配了目标表2行记录 (2,2,'update')，(2,2,'delete'') ,先通过update操作更新了目标表记录 (2,2,'update') 为 (2,2,'tbase_update') ，之后delete待操作的集合就变为update操作之后的结果记录 (2,2,'tbase_update') 而不包括 (2,2,'delete') 这条记录，因此delete通过where条件只能把这条记录删除，记录 (2,2,'delete') 不受影响。

数据读取
单表查询

单表查询是指从 TDSQL PG 数据库的一个表或视图中检索一列或多列数据。TDSQL PG 支持对单表查询的结果数据进行过滤、去重、排序、分组等操作。

语法

SELECT 的单表查询语法如下：

```
SELECT [ ALL | DISTINCT ] column[, column]
    [ FROM table_clause  ]
    [ WHERE condition ]
    [ GROUP BY grouping_element [, ...] ]
    [ HAVING condition ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
```

参数说明：

- DISTINCT ： 从结果中消除重复的行
- ALL ： 默认，返回所有行
- table_clause ：可以是TDSQL PG中的表、视图和物化视图
- WHERE ：由一个或多个表达式组成的条件，如果是多个条件，根据需要可以用 AND/OR 连接
- GROUP BY ：合成由在一个或者多个值上匹配的行构成的分组，并且在其上计算聚集函数的结果
- HAVING ：消除不满足给定条件的分组
- LIMIT 和 OFFSET ：限制查询返回的行，实现分页查询功能。可以指定偏移量，以及要返回的行数或行百分比

示例

现有一个员工信息表 EMPLOYEES，包括员工ID（EMPLOYEE_ID）、员工姓名（NAME）、性别（GENDER）、出生日期（BIRTHDAY）、员工邮箱（EMAIL）、薪水（SALARY）、主管ID（MANAGER_ID）、部门ID（DEPARTMENT_ID）。
数据如下：

```
 EMPLOYEE_ID | NAME  | GENDER |  BIRTHDAY  |    EMAIL     | SALARY | MANAGER_ID | DEPARTMENT_I
D
-------------+-------+--------+------------+--------------+--------+------------+--------------
 0001        | john  | male   | 1983-03-04 | john@gmail.com |   6500 | 0081       | 003
```

```
0002    | andy  | male   | 1987-08-09 | andy@gmail.com  |   8000 | 0081      | 003
0003    | bob   | male   | 1985-01-04 | bob@gmail.com   |  12000 | 0082      | 005
0004    | eric  | male   | 1997-02-18 | eric@gmail.com  |   7000 | 0081      | 003
0005    | david | male   | 1990-06-12 | david@gmail.com |   9000 | 0082      | 005
0006    | frank | male   | 1982-12-19 | frank@gmail.com |   8800 | 0081      | 003
0007    | henry | male   | 1993-03-04 | henry@gmail.com |  11000 | 0082      | 005
0008    | emily | female | 1989-05-04 | emily@gmail.com |   8500 | 0081      | 003
0009    | lucy  | female | 1992-01-03 | lucy@gmail.com  |   6500 | 0081      | 003
0010    | julia | female | 1991-12-30 | julia@gmail.com |   7000 | 0082      | 005
0011    | leo   | male   | 1983-12-20 | leo@gmail.com   |  14000 | 0081      | 003
```

建表和插入数据语句如下：

```
CREATE TABLE EMPLOYEES(
EMPLOYEE_ID VARCHAR(100),
NAME VARCHAR(100),
GENDER VARCHAR(6),
BIRTHDAY DATE,
EMAIL VARCHAR(100),
SALARY NUMERIC,
MANAGER_ID VARCHAR(100),
DEPARTMENT_ID VARCHAR(100));

INSERT INTO EMPLOYEES VALUES('0001', 'john', 'male', '1983-3-4', 'john@gmail.com', 6500, '0081', '003');
INSERT INTO EMPLOYEES VALUES('0002', 'andy', 'male', '1987-8-9', 'andy@gmail.com', 8000, '0081', '003');
INSERT INTO EMPLOYEES VALUES('0003', 'bob',   'male', '1985-01-04', 'bob@gmail.com', 12000, '0082', '005');
INSERT INTO EMPLOYEES VALUES('0004', 'eric',  'male', '1997-02-18', 'eric@gmail.com', 7000, '0081', '003');
INSERT INTO EMPLOYEES VALUES('0005', 'david', 'male', '1990-6-12', 'david@gmail.com', 9000, '0082', '005');
INSERT INTO EMPLOYEES VALUES('0006', 'frank', 'male', '1982-12-19', 'frank@gmail.com', 8800, '0081', '003');
INSERT INTO EMPLOYEES VALUES('0007', 'henry', 'male', '1993-03-04', 'henry@gmail.com', 11000, '0082', '005');
INSERT INTO EMPLOYEES VALUES('0008', 'emily', 'female', '1989-5-4', 'emily@gmail.com', 8500, '0081', '003');
INSERT INTO EMPLOYEES VALUES('0009', 'lucy', 'female', '1992-1-3', 'lucy@gmail.com', 6500, '0081', '003');
INSERT INTO EMPLOYEES VALUES('0010', 'julia', 'female', '1991-12-30', 'julia@gmail.com', 7000, '0082', '005');
INSERT INTO EMPLOYEES VALUES('0011', 'leo', 'male', '1983-12-20', 'leo@gmail.com', 14000, '0081', '003');
```

示例1

查询所有员工信息

```
SELECT * FROM (employees);
```

返回结果如上。

示例2

从员工表中查询所有1990年之前出生的员工

```
set nls_date_format to 'YYYY-MM-DD HH24:MI:SS';
SELECT * FROM employees WHERE birthday < to_date('1990-1-1');
```

结果如下：

```
 EMPLOYEE_ID | NAME  | GENDER |      BIRTHDAY       |    EMAIL     | SALARY | MANAGER_ID | DEPARTM
ENT_ID
-------------+-------+--------+---------------------+--------------+--------+------------+--------------
 0001        | john  | male   | 1983-03-04 00:00:00 | john@gmail.com  |   6500 | 0081       | 003
 0002        | andy  | male   | 1987-08-09 00:00:00 | andy@gmail.com  |   8000 | 0081       | 003
 0003        | bob   | male   | 1985-01-04 00:00:00 | bob@gmail.com   |  12000 | 0082       | 005
 0006        | frank | male   | 1982-12-19 00:00:00 | frank@gmail.com |   8800 | 0081       | 003
 0008        | emily | female | 1989-05-04 00:00:00 | emily@gmail.com |   8500 | 0081       | 003
 0011        | leo   | male   | 1983-12-20 00:00:00 | leo@gmail.com   |  14000 | 0081       | 003
```

示例3

从员工表中查询所有员工信息，按薪水从高到低进行排序

```
SELECT * FROM employees ORDER BY salary DESC;
```

结果如下：

```
 EMPLOYEE_ID | NAME  | GENDER |  BIRTHDAY  |    EMAIL     | SALARY | MANAGER_ID | DEPARTMENT_I
D
-------------+-------+--------+------------+--------------+--------+------------+--------------
 0011        | leo   | male   | 1983-12-20 | leo@gmail.com   |  14000 | 0081       | 003
 0003        | bob   | male   | 1985-01-04 | bob@gmail.com   |  12000 | 0082       | 005
 0007        | henry | male   | 1993-03-04 | henry@gmail.com |  11000 | 0082       | 005
 0005        | david | male   | 1990-06-12 | david@gmail.com |   9000 | 0082       | 005
 0006        | frank | male   | 1982-12-19 | frank@gmail.com |   8800 | 0081       | 003
 0008        | emily | female | 1989-05-04 | emily@gmail.com |   8500 | 0081       | 003
 0002        | andy  | male   | 1987-08-09 | andy@gmail.com  |   8000 | 0081       | 003
 0004        | eric  | male   | 1997-02-18 | eric@gmail.com  |   7000 | 0081       | 003
```

```
0010       | julia | female | 1991-12-30 | julia@gmail.com |   7000 | 0082       | 005
0001       | john  | male   | 1983-03-04 | john@gmail.com  |   6500 | 0081       | 003
0009       | lucy  | female | 1992-01-03 | lucy@gmail.com  |   6500 | 0081       | 003
```

示例4

从员工表中查询各个部门的员工人数

```
SELECT department_id, count(*) FROM employees GROUP BY department_id;
```

结果如下：

```
 DEPARTMENT_ID | COUNT
---------------+-------
 003           |     7
 005           |     4
```

示例5

在示例 4 的基础上，只查询部门人数大于 5 人的部门ID

```
SELECT department_id, count(*) FROM employees GROUP BY department_id HAVING count(*) > 5;
```

结果如下：

```
 DEPARTMENT_ID | COUNT
---------------+-------
 003           |     7
```

示例6

查询工资最高的前3个员工信息

```
SELECT * FROM employees ORDER BY salary DESC LIMIT 3;
```

结果如下：

```
 EMPLOYEE_ID | NAME  | GENDER |  BIRTHDAY  |    EMAIL     | SALARY | MANAGER_ID | DEPARTMENT_I
D
-------------+-------+--------+------------+--------------+--------+------------+--------------
 0011        | leo   | male   | 1983-12-20 | leo@gmail.com   |  14000 | 0081       | 003
 0003        | bob   | male   | 1985-01-04 | bob@gmail.com   |  12000 | 0082       | 005
 0007        | henry | male   | 1993-03-04 | henry@gmail.com |  11000 | 0082       | 005
```

关联查询

关联查询是指从 TDSQL PG 数据库的两个或多个表/视图中检索数据。TDSQL PG 支持的关联方式的有内连接、（全/左/右）外连接、自然连接等。

内连接

内连接用于返回满足联接条件的多表数据。
INNER JOIN需要搭配联接条件（ON）一起使用，否则将返回多表的笛卡尔积。

语法

```
SELECT target_list FROM table_name1 [INNER] JOIN table_name2 ON join_condition
[ WHERE query_condition ]
[ ORDER BY column_list ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition ]
[ LIMIT { count | ALL } ]
[ OFFSET start [ ROW | ROWS ] ]
```

或者，直接在 FROM 子句后面写多个表（或者视图），使用逗号分开

```
SELECT target_list FROM table_name1，table_name2 [ WHERE query_condition ] ...
```

示例

查询有多少人比自己的薪水低

```
select e1.name, count(*) from EMPLOYEES e1 join EMPLOYEES e2 on e1.salary > e2.salary group by e
1.name;
 NAME  | COUNT
-------+-------
 henry |     8
 eric  |     2
 andy  |     4
 bob   |     9
 david |     7
 leo   |    10
 frank |     6
 julia |     2
 emily |     5
(9 rows)
```

另一种内联接写法

```
select e1.name, count(*) from EMPLOYEES e1, EMPLOYEES e2 WHERE e1.salary > e2.salary group by e
1.name;
 NAME  | COUNT
-------+-------
 henry |     8
 eric  |     2
 andy  |     4
 bob   |     9
 david |     7
 leo   |    10
 frank |     6
 julia |     2
 emily |     5
(9 rows)
```

全外连接

全外连接，FULL [OUTER] JOIN，返回多表的所有的行，包括满足联接条件的行和不满足的行，未匹配到的用 NULL 值填充。

语法

```
SELECT target_list FROM table_name1 FULL [OUTER] JOIN table_name2 ON join_condition
[ WHERE query_condition ]
[ ORDER BY column_list ]
```

示例

```
CREATE TABLE t1(a INT, b INT, c INT);
CREATE TABLE t2(x INT, y INT, z INT);

INSERT INTO t1 VALUES(1, 1, 1);
INSERT INTO t1 VALUES(2, 2, 2);
INSERT INTO t1 VALUES(3, 3, 3);
INSERT INTO t2 VALUES(2, 2, 2);
INSERT INTO t2 VALUES(3, 3, 3);
INSERT INTO t2 VALUES(5, 5, 5);

SELECT * FROM t1 FULL JOIN t2 ON a = x;
 A | B | C | X | Y | Z
---+---+---+---+---+---
```

```
 1 | 1 | 1 |   |   |
 2 | 2 | 2 | 2 | 2 | 2
 3 | 3 | 3 | 3 | 3 | 3
   |   |   | 5 | 5 | 5
```

左外连接

左外连接，LEFT [OUTER] JOIN，它返回左侧表中所有的行，并与右侧表进行匹配，未匹配到的用 NULL 值填充。

语法

```
SELECT target_list FROM table_name1 LEFT JOIN table_name2 ON join_condition
[ WHERE query_condition ]
[ ORDER BY column_list ]
```

示例

```
SELECT * FROM t1 LEFT JOIN t2 ON a = x;
 A | B | C | X | Y | Z
---+---+---+---+---+---
 1 | 1 | 1 |   |   |
 2 | 2 | 2 | 2 | 2 | 2
 3 | 3 | 3 | 3 | 3 | 3
(3 rows)
```

右外连接

右外连接又称为 RIGHT [OUTER] JOIN，它返回右侧表中所有的行，并与左侧表进行匹配，未匹配到的用 NULL 值填充

语法

```
SELECT target_list FROM table_name1 RIGHT [OUTER] JOIN table_name2 ON join_condition
[ WHERE query_condition ]
[ ORDER BY column_list ]
```

示例

```
SELECT * FROM t1 RIGHT JOIN t2 ON a = x;
 A | B | C | X | Y | Z
```

```
---+---+---+---+---+---
 2 | 2 | 2 | 2 | 2 | 2
 3 | 3 | 3 | 3 | 3 | 3
   |   |   | 5 | 5 | 5
(3 rows)
```

自然连接

自然连接， NATURAL JOIN，是一种特殊的等值连接，它要求两个表中进行比较的列必须是同名且可比较，并在结果中把同名的列去掉，如果列类型不匹配则会报错。

语法

```
SELECT target_list FROM table_name1 NATURAL JOIN table_name2
```

示例

```
CREATE TABLE tab1(a INT, b INT, c INT);
CREATE TABLE tab2(a INT, d INT, f INT);
INSERT INTO tab1 VALUES(1, 1, 1);
INSERT INTO tab1 VALUES(2, 2, 2);
INSERT INTO tab1 VALUES(3, 3, 3);
INSERT INTO tab2 VALUES(2, 2, 2);
INSERT INTO tab2 VALUES(3, 3, 3);
INSERT INTO tab2 VALUES(5, 5, 5);

SELECT * FROM tab1 NATURAL JOIN tab2;
 A | B | C | D | F
---+---+---+---+---
 2 | 2 | 2 | 2 | 2
 3 | 3 | 3 | 3 | 3
(2 rows)


--类型不匹配，报错
CREATE TABLE tab3(a INT, b INT, c INT);
CREATE TABLE tab4(a DATE, d INT, f INT);
SELECT * FROM tab3 NATURAL JOIN tab4;
ERROR:  (42804) JOIN/USING types integer and date cannot be matched
```

子查询

子查询是指嵌套在一个查询中的查询。TDSQL PG支持多层嵌套查询。在特定情况下，一个查询语句的条件需要从另一个查询语句中获取，内层查询（inner query）语句的查询结果，可以为外层查询（outer query）语句提供查询条件。
子查询按不同的维度，可以有不同的划分。按子查询和父查询的依赖关系划分，可以分为相关子查询和不相关子查询。
按子查询结果集划分，可以分为标量子查询，行子查询，列子查询和表子查询。

语法及关键字

语法说明：

- 子查询可以在大部分的DDL（如CREATE TABLE/VIEW）以及DML（SELECT、INSERT、UPDATE）语句中使用。
- 子查询需要放在小括号中。
- 子查询中的关键词可以使用 IN、ANY、SOME 和 ALL 等。

示例表

```
---职员信息表
CREATE TABLE EMPLOYEES(
EMPLOYEE_ID VARCHAR(100),
NAME VARCHAR(100),
GENDER VARCHAR(6),
BIRTHDAY DATE,
EMAIL VARCHAR(100),
SALARY NUMERIC,
MANAGER_ID VARCHAR(100),
DEPARTMENT_ID VARCHAR(100));
---插入数据
INSERT INTO EMPLOYEES VALUES('0001', 'john', 'male', '1983-3-4', 'john@gmail.com', 6500, '0081', '003');
INSERT INTO EMPLOYEES VALUES('0002', 'andy', 'male', '1987-8-9', 'andy@gmail.com', 8000, '0081', '003');
INSERT INTO EMPLOYEES VALUES('0003', 'bob',   'male', '1985-01-04', 'bob@gmail.com', 12000, '0082', '005');
INSERT INTO EMPLOYEES VALUES('0004', 'eric',  'male', '1997-02-18', 'eric@gmail.com', 7000, '0081', '003');
INSERT INTO EMPLOYEES VALUES('0005', 'david', 'male', '1990-6-12', 'david@gmail.com', 9000, '0082',
```

```
'005');
INSERT INTO EMPLOYEES VALUES('0006', 'frank', 'male', '1982-12-19', 'frank@gmail.com', 8800, '0081',
'003');
INSERT INTO EMPLOYEES VALUES('0007', 'henry', 'male', '1993-03-04', 'henry@gmail.com', 11000, '008
2', '005');
INSERT INTO EMPLOYEES VALUES('0008', 'emily', 'female', '1989-5-4', 'emily@gmail.com', 8500, '0081',
'003');
INSERT INTO EMPLOYEES VALUES('0009', 'lucy', 'female', '1992-1-3', 'lucy@gmail.com', 6500, '0081', '00
3');
INSERT INTO EMPLOYEES VALUES('0010', 'julia', 'female', '1991-12-30', 'julia@gmail.com', 7000, '0082',
'005');
INSERT INTO EMPLOYEES VALUES('0011', 'leo', 'male', '1983-12-20', 'leo@gmail.com', 14000, '0081', '00
3');
```

相关子查询

子查询对父查询有所依赖，子查询并不能独立运行的查询。

示例1

```
select * from employees e1 where DEPARTMENT_ID in (select DEPARTMENT_ID from employees e2 wh
ere e2.salary < e1.salary);
 EMPLOYEE_ID | NAME  | GENDER |      BIRTHDAY       |     EMAIL     | SALARY | MANAGER_ID | DEPARTM
ENT_ID
-------------+-------+--------+---------------------+---------------+--------+------------+---------------
 0002        | andy  | male   | 1987-08-09 00:00:00 | andy@gmail.com  |   8000 | 0081       | 003
 0003        | bob   | male   | 1985-01-04 00:00:00 | bob@gmail.com   |  12000 | 0082       | 005
 0004        | eric  | male   | 1997-02-18 00:00:00 | eric@gmail.com  |   7000 | 0081       | 003
 0005        | david | male   | 1990-06-12 00:00:00 | david@gmail.com |   9000 | 0082       | 005
 0006        | frank | male   | 1982-12-19 00:00:00 | frank@gmail.com |   8800 | 0081       | 003
 0007        | henry | male   | 1993-03-04 00:00:00 | henry@gmail.com |  11000 | 0082       | 005
 0008        | emily | female | 1989-05-04 00:00:00 | emily@gmail.com |   8500 | 0081       | 003
 0011        | leo   | male   | 1983-12-20 00:00:00 | leo@gmail.com   |  14000 | 0081       | 003
(8 rows)
```

不相关子查询

子查询对父查询没有依赖，可以独立执行的查询。

示例2

```
select * from employees where gender in (select 'male' from dual);
```

```
EMPLOYEE_ID | NAME  | GENDER |      BIRTHDAY       |    EMAIL     | SALARY | MANAGER_ID | DEPARTME
NT_ID
-------------+-------+--------+---------------------+--------------+--------+------------+--------------
 0001       | john  | male   | 1983-03-04 00:00:00 | john@gmail.com  |   6500 | 0081       | 003
 0002       | andy  | male   | 1987-08-09 00:00:00 | andy@gmail.com  |   8000 | 0081       | 003
 0003       | bob   | male   | 1985-01-04 00:00:00 | bob@gmail.com   |  12000 | 0082       | 005
 0004       | eric  | male   | 1997-02-18 00:00:00 | eric@gmail.com  |   7000 | 0081       | 003
 0005       | david | male   | 1990-06-12 00:00:00 | david@gmail.com |   9000 | 0082       | 005
 0006       | frank | male   | 1982-12-19 00:00:00 | frank@gmail.com |   8800 | 0081       | 003
 0007       | henry | male   | 1993-03-04 00:00:00 | henry@gmail.com |  11000 | 0082       | 005
 0011       | leo   | male   | 1983-12-20 00:00:00 | leo@gmail.com   |  14000 | 0081       | 003
(8 rows)
```

标量子查询

返回的结果集是 1 行 1 列

示例3

```
--查询薪水大于平均薪水的所有员工信息
select * from EMPLOYEES where salary > (select avg(salary) from EMPLOYEES);

 EMPLOYEE_ID | NAME  | GENDER |  BIRTHDAY  |    EMAIL     | SALARY | MANAGER_ID | DEPARTMENT_I
D
-------------+-------+--------+------------+--------------+--------+------------+--------------
 0003       | bob   | male   | 1985-01-04 | bob@gmail.com   |  12000 | 0082       | 005
 0005       | david | male   | 1990-06-12 | david@gmail.com |   9000 | 0082       | 005
 0007       | henry | male   | 1993-03-04 | henry@gmail.com |  11000 | 0082       | 005
 0011       | leo   | male   | 1983-12-20 | leo@gmail.com   |  14000 | 0081       | 003
(4 rows)
```

行子查询

返回的结果集是 1 行 N 列

示例4

```
---查询与eric属于同一部门，且性别相同的所有员工信息
select * from EMPLOYEES  where (department_id, gender) = (select department_id, gender from EMPL
OYEES where name = 'eric');
```

```
 EMPLOYEE_ID | NAME  | GENDER |  BIRTHDAY  |    EMAIL     | SALARY | MANAGER_ID | DEPARTMENT_I
D
-------------+-------+--------+------------+--------------+--------+------------+--------------
 0001        | john  | male   | 1983-03-04 | john@gmail.com  |   6500 | 0081       | 003
 0002        | andy  | male   | 1987-08-09 | andy@gmail.com  |   8000 | 0081       | 003
 0004        | eric  | male   | 1997-02-18 | eric@gmail.com  |   7000 | 0081       | 003
 0006        | frank | male   | 1982-12-19 | frank@gmail.com |   8800 | 0081       | 003
 0011        | leo   | male   | 1983-12-20 | leo@gmail.com   |  14000 | 0081       | 003
(5 rows)
```

列子查询

返回的结果集是 N 行 1 列

列子查询通常使用 IN, ANY, ALL 和 SOME 操作符

- IN

  返回布尔型，如果属于列表中任意一个，返回 true，否则返回 false
- ANY

  返回布尔型，需要与比较操作符配合使用，如果集合中任意一个满足条件，返回 true，否则返回 false
- SOME

  与 ANY 作用相同
- ALL

  返回布尔型，需要与比较操作符配合使用，集合中所有元素满足条件才返回 true，否则返回 false

示例5

```
---查询名字以 d 或 f 开头的员工信息
select * from EMPLOYEES where substr(name, 1, 1) in (select 'd' union select 'f');

 EMPLOYEE_ID | NAME  | GENDER |  BIRTHDAY  |    EMAIL     | SALARY | MANAGER_ID | DEPARTMENT_I
D
-------------+-------+--------+------------+--------------+--------+------------+--------------
 0005        | david | male   | 1990-06-12 | david@gmail.com |   9000 | 0082       | 005
 0006        | frank | male   | 1982-12-19 | frank@gmail.com |   8800 | 0081       | 003
(2 rows)
```

示例6

```
---查询工资大于 8000 或者 10000 的员工
select * from EMPLOYEES where salary > any (select 8000 union select 10000);

 EMPLOYEE_ID | NAME  | GENDER |  BIRTHDAY  |    EMAIL     | SALARY | MANAGER_ID | DEPARTMENT_I
```

```
D
------------+-------+--------+------------+-------------+--------+-----------+--------------
 0003       | bob   | male   | 1985-01-04 | bob@gmail.com   |  12000 | 0082      | 005
 0005       | david | male   | 1990-06-12 | david@gmail.com |   9000 | 0082      | 005
 0006       | frank | male   | 1982-12-19 | frank@gmail.com |   8800 | 0081      | 003
 0007       | henry | male   | 1993-03-04 | henry@gmail.com |  11000 | 0082      | 005
 0008       | emily | female | 1989-05-04 | emily@gmail.com |   8500 | 0081      | 003
 0011       | leo   | male   | 1983-12-20 | leo@gmail.com   |  14000 | 0081      | 003
(6 rows)
```

示例7

```
---查询工资大于 8000 和 10000 的员工
select * from EMPLOYEES where salary > all (select 8000 union select 10000);
 EMPLOYEE_ID | NAME  | GENDER |  BIRTHDAY  |    EMAIL     | SALARY | MANAGER_ID | DEPARTMENT_I
D
------------+-------+--------+------------+-------------+--------+-----------+--------------
 0003       | bob   | male   | 1985-01-04 | bob@gmail.com   |  12000 | 0082      | 005
 0007       | henry | male   | 1993-03-04 | henry@gmail.com |  11000 | 0082      | 005
 0011       | leo   | male   | 1983-12-20 | leo@gmail.com   |  14000 | 0081      | 003
(3 rows)
```

表子查询

返回的结果集是 N 行 N 列

表子查询与列子查询类似，区别是返回的是多列，需要与 IN, ANY, ALL 配合使用。

示例8

```
select * from employees where (gender, department_id) in (select 'male', 003 from dual);

 EMPLOYEE_ID | NAME  | GENDER |      BIRTHDAY       |    EMAIL     | SALARY | MANAGER_ID | DEPARTM
ENT_ID
------------+-------+--------+---------------------+-------------+--------+-----------+--------------
 0001       | john  | male   | 1983-03-04 00:00:00 | john@gmail.com  |   6500 | 0081      | 003
 0002       | andy  | male   | 1987-08-09 00:00:00 | andy@gmail.com  |   8000 | 0081      | 003
 0004       | eric  | male   | 1997-02-18 00:00:00 | eric@gmail.com  |   7000 | 0081      | 003
 0006       | frank | male   | 1982-12-19 00:00:00 | frank@gmail.com |   8800 | 0081      | 003
 0011       | leo   | male   | 1983-12-20 00:00:00 | leo@gmail.com   |  14000 | 0081      | 003
(5 rows)
```

CTE

CTE（Common Table Expressions，即公共表表达式）是一个可以命名的临时结果集，不实际作为对象存储；其作用范围是当前语句，仅可在查询执行期间使用。
CTE一般用于相同子查询的复用、简化递归查询或者可以将复杂的嵌套查询拆分为多个逻辑更清晰的子句。

语法

使用 WITH 语句创建公共表表达式，语法为

```
WITH [RECURSIVE]
  with_query_name [(col_name [, ...] )] AS ( select | values | insert | update | delete )
  [, ...]
  ...
```

参数说明

- RECURSIVE, 递归，如果指定了RECURSIVE，则允许一个 SELECT 子查询使用名称引用自身。
- with_query_name, 公共表表达式的名称，可被包含 WITH 子句的的表引用。
- column_name，公共表表达式的列名， 可在之后的语句中直接引用。

递归的CTE

递归CTE用于处理具有递归结构的数据，例如树形结构、图形等。其WITH子句必须以RECURSIVE 开头，同时AS之后的语句需要使用UNION或UNION ALL将未引用CTE的SELECT和引用了CTE的SELECT 联合起来, 且未引用CTE的SELECT要放在前面。

语法

```
WITH RECURSIVE cte(col) AS (
subquery1
UNION [ALL]
subquery2
)
```

这里，subquery1 用来产生初始结果集， 然后用该结果作为条件，递归执行 subquery2，当 subquery2 不再产生新的结果集时，执行结束。

示例1

```
WITH RECURSIVE cte (col) AS
(
  SELECT 1              -- 非递归部分，生成初始行集
  UNION ALL
  SELECT col + 3 FROM cte WHERE col < 20     -- 递归部分，col 初始为 1，然后 +3 递归执行，直到小于20，结束
)
SELECT * FROM cte;

--执行结果：
 COL
-----
   1
   4
   7
  10
  13
  16
  19
  22
(8 rows)
```

非递归的CTE

与递归CTE相对应的，如果一个CTE的AS子句中，没有引用自身，那么该CTE被称为非递归 CTE。

示例2

```
WITH cte AS (
SELECT gender, count(*) FROM employees GROUP BY gender
) SELECT * FROM cte;

--查询结果
 GENDER | COUNT
--------+-------
 female |     3
 male   |     8
(2 rows)
```

集合操作

TDSQL PG支持三种集合操作：并集 (UNION [all])，差集 (EXCEPT\MINUS) 和交集 (INTERSECT)。

语法

```
select_statement1
{ UNION | UNION ALL | EXCEPT  |  MINUS | INTERSECT}
select_statement2
...
```

参数说明：

- UNION ：计算所涉及的SELECT语句所返回的行的并集，并去除重复的行。
- UNION ALL ：计算所涉及的SELECT语句所返回的行的并集，保留重复的行。
- EXCEPT / MINUS ：计算位于左SELECT语句的结果中但不在右SELECT语句结果中的行集合。
- INTERSECT ：计算两个SELECT语句的结果集的交集。

> 说明
>
> 每个select结果集的列数和数据类型必须相同。

示例

数据表

```
select * from test_tb1;
 id | name
----+------
  1 | A
  2 | B
  5 | E
  6 | F
  3 | C
  4 | D
(6 rows)

select * from test_tb2;
 id | name
----+------
  1 | A
  2 | B
```

```
 8 | H
 9 | I
 3 | C
 7 | G
(6 rows)
```

示例1- UNION

```
---返回所有行，并去除重复行
select id, name from test_tb1
union
select id, name from test_tb2;
 id | name
----+------
 9 | I
 1 | A
 2 | B
 8 | H
 5 | E
 3 | C
 6 | F
 4 | D
 7 | G
(9 rows)
```

示例2- UNION ALL

```
---返回所有行，并保留重复行
select id, name from test_tb1
union all
select id, name from test_tb2;
 id | name
----+------
 1 | A
 2 | B
 5 | E
 6 | F
 1 | A
 2 | B
 8 | H
 9 | I
 3 | C
 4 | D
 3 | C
 7 | G
```

```
(12 rows)
```

示例3- EXCEPT

```
---返回左表中有而右表中没有的行
select id, name from test_tb1
except
select id, name from test_tb2;
 id | name
----+------
  4 | D
  5 | E
  6 | F
(3 rows)
```

示例4- MINUS

```
---返回左表中有而右表中没有的行
select id, name from test_tb1
minus
select id, name from test_tb2;
 ID | NAME
----+------
  4 | D
  5 | E
  6 | F
(3 rows)
```

> 说明
>
> 仅ORACLE兼容下支持 MINUS 语法

示例5- INTERSECT

```
---返回左表中有且右表中也有的行
select id, name from test_tb1
intersect
select id, name from test_tb2;
 id | name
----+------
  3 | C
  1 | A
  2 | B
(3 rows)
```

事务使用
事务概述

事务是一组数据库操作的集合，这些操作作为一个整体被执行，要么全部成功，要么全部失败，数据库会从一个一致性状态转换到下一个一致性状态。即使在系统故障的情况下，也能保证数据的一致性。
如果事务中的某个操作失败，那么整个事务将被回滚，数据库状态将恢复到事务开始之前的状态。
TDSQL PG事务系统通过使用多版本并发控制（MVCC）和快照隔离等技术，提供了高效的并发控制和严格的事务隔离，确保并发执行的事务不会互相干扰。在MVCC中查询数据获取的锁不会与写入数据获取的锁发生冲突，因此读取不会阻塞写入，写入也不会阻塞读取，可以为用户提供高性能并发读写能力。

事务隔离级别

TDSQL PG支持四种事务隔离级别，这些级别定义了一个事务可能会看到其他并发事务所做更改的程度。这些级别从低到高分别是：

- Read Uncommitted（读未提交）：这是最低的隔离级别，一个事务可能会看到其他事务未提交的更改。
- Read Committed（读已提交）：这是TDSQL PG默认隔离级别。在这个级别，一个事务能看到在该事务开始之前已经提交事务所做的更改，后续的查询也会看到其他事务所提交新的更改。
- Repeatable Read（可重复读）：在这个级别，一个事务在其整个过程中看到的是一个一致的快照。也就是说，它只能看到在该事务开始时已经提交的事务所做的更改，而不能看到在该事务进行期间其他事务所提交的更改。
- Serializable（可串行化）：这是最高的隔离级别。它提供了最严格的隔离级别，确保事务序列化执行，即使在并发执行时也能得到相同的结果。这个级别可以防止所有类型的并发问题。

锁

TDSQL PG还提供了表级和行级锁定，为应用提供显式管理互斥场景的能力；TDSQL PG支持咨询锁，提供了跨事务的锁机制；整体上为应用提供了更高的灵活度。

表级锁

TDSQL PG提供了多种锁模式来控制对表中数据的并发访问。在多版本并发控制（MVCC）无法提供所需行为的情况下，这些模式可以用于应用程序显式执行锁定动作。注意，TDSQL PG支持的命令会自动获取合适的锁，确保在命令执行期间操作的表不会被删除等情况发生（例如TRUNCATE不能与其他命令同时操作一张表，所以TRUNCATE需要拿到ACCESS EXCLUSIVE模式锁）。
Conflicting Lock Modes

Requested	ACCESS	ROW	ROW	SHARE	SHARE	SHARE	EXCL.	ACCESS

Lock/ Existing Lock	SHARE	SHARE	EXCL.	UPDATE EXCL.		ROW EXCL.		EXCL.
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCL.					X	X	X	X
SHARE UPDATE EXCL.				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCL.			X	X	X	X	X	X
EXCL.		X	X	X	X	X	X	X
ACCESS EXCL.	X	X	X	X	X	X	X	X

行级锁

TDSQL PG的行级锁主要用于控制对单数据行的并发访问。行级锁通常在事务中使用，以确保在事务执行期间，其他事务不能修改被锁定的行。
TDSQL PG提供了以下几种行级锁模式：

1. FOR UPDATE：这种锁定模式在读取数据行时获取一个排他锁。一旦一个事务获取了这种锁，其他事务就不能获取同一行的FOR UPDATE、FOR NO KEY UPDATE、FOR SHARE 或 FOR KEY SHARE锁，直到第一个事务完成。
2. FOR NO KEY UPDATE：这种锁定模式类似于FOR UPDATE，但它允许其他事务获取FOR KEY SHARE锁。这对于需要防止其他事务修改数据行，但允许它们并发读取数据行的情况很有用。
3. FOR SHARE：这种锁定模式在读取数据行时获取一个共享锁。这意味着其他事务可以获取同一行的FOR SHARE或FOR KEY SHARE锁，但不能获取FOR UPDATE或FOR NO KEY UPDATE锁。
4. FOR KEY SHARE：这是最轻量级的行级锁定模式。它允许其他事务获取FOR NO KEY UPDATE、FOR SHARE或FOR KEY SHARE锁。

咨询锁

咨询锁（Advisory Locks）是TDSQL PG提供的一种特殊类型的锁，它们不由系统自动管理，完全提供给应用程序使用。这种锁的主要用途是在应用程序级别实现复杂的业务规则或协调多个事务的行为。

Advisory Lock有两种级别：会话级别和事务级别。

1. 会话级别的Advisory Lock：这种锁在当前会话结束时自动释放，或者可以通过调用unlock函数手动释放。
2. 事务级别的Advisory Lock：这种锁在当前事务结束时自动释放，无论事务是提交还是回滚。

事务操作

事务操作包括BEGIN（开启事务），COMMIT（提交事务），ROLLBACK（回滚事务）还有SAVEPOINT（保存点），用于控制和管理事务行为。

开启事务

BEGIN开始一个事务块，BEGIN命令之后的所有语句将被在一个事务中执行，直到给出一个显式的COMMIT或ROLLBACK。

语法

```
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]
```

- transaction_mode 可以是:
  ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
  READ WRITE | READ ONLY
  [NOT] DEFERRABLE

  > 说明：
  >
  > 如果不指定transaction_mode，会使用默认隔离级别，READ COMMITTED（读已提交）。

提交事务

COMMIT提交当前事务，所有由该事务所作的更改将持久保存在数据库中。

语法

```
COMMIT [ WORK | TRANSACTION ]
```

示例

```
drop table t1;
create table t1(id int);

begin;
insert into t1 values(1);
select * from t1;
 ID
----
```

```
 1
(1 row)
commit;

-- after the commit is executed, the data is truly inserted into the database.
select * from t1;
 ID
----
 1
(1 row)
```

回滚事务

ROLLBACK中止一个事务，该事务的所有操作将撤销。

语法

```
ROLLBACK [ WORK | TRANSACTION ]
```

示例

```
drop table t1;
create table t1(id int);

begin;
insert into t1 values(1);
select * from t1;
 ID
----
 1
(1 row)
rollback;

-- after a rollback is performed, the data is not actually inserted into the database.
select * from t1;
 ID
----
(0 rows)
```

AUTOCOMMIT模式

注意，以上均为显示开启事务块的场景。如果不显示开启事务块，则默认采用autocommit模式，即自动提交。

示例

```
drop table t1;
create table t1(id int);

insert into t1 values(1);
select * from t1;
 ID
----
  1
(1 row)
```

SAVEPOINT

SAVEPOINT用于在事务中设置一个保存点，即在当前事务内做一个特殊标记，允许在后续操作回滚（ROLLBACK TO）到指定保存点，将事务状态恢复到保存点时的状态。

建立保存点

设置好一个保存点，在后续操作中就可以把数据撤销到保存点时的状态，做到只回滚部分数据，方便按需回滚，需要在事务块内执行。

语法

```
SAVEPOINT savepoint_name
```

示例

```
-- create table
CREATE TABLE table1 (id integer);


BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT sp1;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT sp1;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

上述示例中，回滚到SAVEPOINT的动作只会将 INSERT INTO table1 VALUES (2); 的结果回滚掉，不会影响其他写入。

回滚保存点

回滚保存点后执行的所有操作，回滚后，SAVEPOINT仍然有效，可以在需要时继续回滚。

语法

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name
```

示例

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT sp1;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT sp1;
    INSERT INTO table1 VALUES (3);
    ROLLBACK TO SAVEPOINT sp1;
COMMIT;
```

上述示例中，发生了两次ROLLBACK TO sp1，最终事务提交后只有 INSERT INTO table1 VALUES (1); 操作被提交。

释放保存点

释放指定的保存点，并会同时释放后续创建的所有保存点。

语法

```
RELEASE SAVEPOINT savepoint_name
```

示例

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
ROLLBACK;
```

释放保存点主要用于释放事务资源，并不会将事务提交，例如上述示例中，INSERT插入的数据会跟随最后主事务的回滚操作而被撤销。

Oracle兼容特性

Oracle兼容特性概述

本节介绍TDSQL PG数据库在Oracle兼容方面的特性。
TDSQL PG数据库基本兼容了Oracle的数据类型、SQL功能、数据库对象（如：同义词、包）。在PL（过程化语言）方面也做了很多细致兼容工作，用户应用迁移基本不需要改动业务代码。
TDSQL PG对如下功能做了兼容：

- SQL基本元素
- 伪列
- 操作符
- SQL表达式
- 条件
- 常见SQL语法
- PL/SQL
- 系统函数
- 系统视图
- 系统包
- Oracle兼容性配置参数

使用Oracle兼容模式

TDSQL PG内核引擎支持Oracle 和PostgreSQL 两种SQL兼容模式。
在创建数据库的时候使用 sql mode``<sql_mode> 即可指定创建数据库的兼容模式，其中 <sql_mode> 可选 oracle 或 postgresql 。不指定默认是PostgreSQL 兼容模式。SQL 兼容模式涉及底层底层存储和 SQL 行为，指定后无法改变。
示例：

```
tdsql=# create database ora sql mode oracle;
CREATE DATABASE

tdsql=# \l
                                 List of databases
    Name     | Owner | Encoding | Sqlmode  |  Collate   |   Ctype    | Access privileges
-------------+-------+----------+----------+------------+------------+-------------------
 ora         | tbase | UTF8     | oracle   | zh_CN.utf8 | zh_CN.utf8 |
 postgres    | tbase | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 |
 tbase_db    | tbase | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 |
 tdsql       | tbase | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 |
```

```
 template0     | tbase | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 | =c/tbase        +
               |       |          |          |            |            | tbase=CTc/tbase
 template0_ora | tbase | UTF8     | oracle   | zh_CN.utf8 | zh_CN.utf8 | =c/tbase        +
               |       |          |          |            |            | tbase=CTc/tbase
 template1     | tbase | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 | =c/tbase        +
               |       |          |          |            |            | tbase=CTc/tbase
 template1_ora | tbase | UTF8     | oracle   | zh_CN.utf8 | zh_CN.utf8 | =c/tbase        +
               |       |          |          |            |            | tbase=CTc/tbase
 user1         | user1 | UTF8     | postgres | zh_CN.utf8 | zh_CN.utf8 |
(9 rows)
```

可以看到 ora 数据库的 Sqlmode 列显示 oracle ，即表示 ora 数据库为 Oracle 兼容模式。
而 tdsql 数据库的 Sqlmode 列显示 postgres ，即表示 tdsql 数据库为 PostgreSQL 兼容模式。

SQL基本元素
SQL数据类型

本节主要介绍 TDSQL PG 数据库的 Oracle 兼容模式与原生 Oracle 数据库中 SQL 数据类型的兼容对比信息。
我们把 Oracle 数据库的数据类型大体分为几类，以下简单说明与 TDSQL PG 中数据类型的差异：

字符类型

序号	Oracle 数据类型	TDSQL PG 数据库是否支持	说明
1	CHAR [(size [CHAR])]	支持	固定字符长度的字符类型，TDSQL PG 目前仅支持指定 size 单位为CHAR（字符），即使指定 size BYTE 内部存储也是按照字符长度。
2	NCHAR[(size [CHAR])]	支持	固定字符长度的字符类型，TDSQL PG 目前不区分 CHAR 和NCHAR，NCHAR 类型默认会当成 CHAR 来处理。字符编码取决于数据库编码。
3	VARCHAR2[(size [CHAR | BYTE])]	支持	可变字节长度的字符类型，TDSQL PG 目前支持指定 size 单位为BYTE（字节长度）或 CHAR（字符长度），如果 size 后面不接单位默认是 BYTE。
4	NVARCHAR2[(size)]	支持	可变字符长度的字符类型，size 单位为字符长度。字符编码取决于数据库编码。
5	LONG	支持	可变长度的字符类型，Oracle 已废弃，仅供兼容历史版本使用。

数字类型

序号	Oracle数据类型	TDSQL PG 数据库是否支持	说明
1	NUMBER[(p[,s])]	支持	精度为 p 并且数值范围为 s 的数字类型，底层为 TDSQL PG 中的 NUMERIC
2	FLOAT[(p)]	支持	精度为 p 的浮点型，底层为 TDSQL PG 中的 NUMERIC
3	BINARY_FLOAT	支持	浮点型，底层为 TDSQL PG 中的 NUMERIC

4	BINARY_DOUBLE	支持	浮点型，底层为 TDSQL PG 中的 NUMERIC

时间类型

序号	Oracle数据类型	TDSQL PG 数据库是否支持	说明
1	DATE	支持	时间类型，显示格式取决于 NLS_DATE_FORMAT 格式
2	TIMESTAMP [(fractional_seconds_precision)]	支持	时间戳类型，fractional_seconds_precision 表示其中秒的小数位数，范围为[0,9]
3	TIMESTAMP [(fractional_seconds_precision)] WITH TIME ZONE	支持	带时区的时间戳类型，fractional_seconds_precision 同 TIMESTAMP
4	TIMESTAMP [(fractional_seconds_precision)] WITH LOCAL TIME ZONE	支持	带本地时区的时间戳，fractional_seconds_precision 同 TIMESTAMP
5	INTERVAL YEAR [(year_precision)] TO MONTH	支持	包含年、月的时间间隔类型，year_precision 是年的数字位数
6	INTERVAL DAY [(day_precision)] TO SECOND [(fractional_seconds_precision)]	支持	包含天、小时、分钟、秒的时间间隔类型，day_precision 表示天的数字位数，fractional_seconds_precision 同 TIMESTAMP

RAW 和 LONG RAW 类型

序号	Oracle数据类型	TDSQL PG 数据库是否支持	说明
1	RAW(size)	支持	可变字节长度的二进制数据类型
2	LONG RAW	支持	可变字节长度的二进制数据类型

大对象类型

序号	Oracle数据类型	TDSQL PG 数据库是否支持	说明
1	BLOB	支持	二进制大对象
2	CLOB	支持	字符大对象
3	NCLOB	支持	字符大对象，字符编码取决于数据库编码。

ROWID 和 UROWID 类型

序号	Oracle数据类型	TDSQL PG 数据库是否支持	说明
1	ROWID	支持	行标识符，可以唯一标识表中的一行。
2	UROWID	支持	逻辑行标识符。

XMLTYPE 类型

序号	Oracle数据类型	TDSQL PG 数据库是否支持	说明
1	XMLTYPE	支持	用于存储 XML 数据格式的类型，支持getStringVal、getClobVal、EXTRACT 等成员函数。

用户自定义类型

序号	Oracle数据类型	TDSQL PG 数据库是否支持	说明
1	对象类型	支持，有差异	支持定义构造函数和成员函数的对象类型，不支持继承语法，不支持map/order函数
2	可变数组	支持，有差异	支持包内定义可变数组，不支持直接 create type 定义可变数组
3	嵌套表	支持	无界的集合，它可以包含任意数量的元素，并且元素是无序的

伪类型

序号	Oracle数据类型	TDSQL PG数据库是否支持	说明
1	anytype	支持	代指其他所有的类型
2	anydata	支持	代指其他所有的对象类型

数据类型比较规则

数据类型比较（排序）规则是指相同数据类型的值之间发生比较（排序）时遵循的比较（排序）规则。

序号	Oracle数据库	TDSQL PG数据库	说明
1	数字值	支持	支持基于数值大小的比较， 位于数轴中相对靠近正方向的数字大于位于数轴中相对靠近反方向的数字，例如 1 < 2，3 > -4，其中 NAN 大于其他的数字值。
2	日期时间值	支持	支持基于日期早晚的比较， 较晚的日期或时间戳大于较早的日期或时间戳，如'2024-10-01 12:00:00' > '2023-10-01 00:00:00'。
3	二进制值	支持	支持 RAW 和 BLOB 二进制类型值的比较，在比较时会使用其字节流进行比较。
4	字符值	支持，有差异	TDSQL PG 和Oracle 的差异如下： 1. TDSQL PG 不支持指定表级别的比较规则，可以指定数据库和列的默认排序规则。 1. TDSQL PG 支持使用ENCODING、LC_CTYPE和LC_COLLATE 三个参数决定创建数据库或者列的默认排序规则。 TDSQL PG 和Oracle 无差异的内容： 1. 在 TDSQL PG 和 Oracle 中，对字符串的比较都遵循如下的基本规则：从字符串的第一个字符开始逐个比较，如果相等则继续比较下一个字符，直到出现不相等的字符或者有一个字符串已经结束为止。如果两个字符串的所有字符都相等，则认为它们相等；否则，以第一个不相等的字符的大的为更大的值，其中如果一个字符串包含另外一个字符串，则长的字符串更大。 1. CHAR 类型会自动填充空格，和其他的类型进行比较。 1. TDSQL PG 和 Oracle 都会根据字符集选择二进制或者单语言排序。
5	可变数组和嵌套表	支持	支持每个元素的比较和整体的数据比较。
6	数据类型优先级	支持	支持数据类型在比较时与 Oracle 相同的优先级。
7	数据类型转换	支持	支持数据类型在比较时与 Oracle 相同的隐式数据类型转换规则

字面量

序号	Oracle数据库	TDSQL　PG数据库
1	文本字面量	支持
2	数值字面量	支持
3	日期时间字面量	支持
4	时间间隔字面量	支持

格式模型

序号	Oracle数据库	TDSQL PG数据库
1	数字格式	支持
2	日期时间格式	支持
3	格式模型修饰符	支持
4	字符串到日期转换规则	支持

空值

序号	Oracle数据库	TDSQL PG数据库
1	空值函数	支持 NVL 函数
2	判空表达式	支持 IS NULL 和 IS NOT NULL
3	空值条件	支持 NULL 值和其他数值进行比较

注释

序号	Oracle数据库	TDSQL PG数据库
1	斜杠和星号（/*）	支持
2	两个连字符（--）	支持
3	COMMENT命令	支持
4	HINT	支持

数据库对象

TDSQL PG兼容的Oracle数据库对象如下：

模式对象

序号	Oracle数据库	TDSQL PG数据库
1	约束	支持CHECK、UNIQUE、NOT NULL约束。 支持PRIMARY KEY约束。 支持FOREIGN KEY约束。 支持ALTER TABLE对约束进行启用、禁用。
2	数据库链接	TDSQL PG支持创建、删除DBLINK。 支持用DBLINK读和写Oracle数据库。
3	数据库触发器	支持
5	索引	支持索引的创建和删除。
6	索引类型	支持BTREE、HASH、BRIN，GIST、GIN索引。 支持索引定义中包含表达式。
7	层级结构	不支持
8	Java 类	不支持
9	Java 资源	不支持
10	Java 源码	不支持
11	物化视图	不支持
12	对象表	支持自定义对象表
13	对象类型	支持自定义类型
14	对象视图	支持
15	操作符	支持，但是语法和 Oracle 不同，兼容 PostgreSQL 语法。
16	包	支持，具体可在 PL/SQL 章节查看。
17	序列	支持序列的创建和删除。 支持NOCACHE、NOORDER 属性。
18	存储函数	支持

序号	Oracle数据库	TDSQL PG数据库
19	存储过程	支持
20	同义词	支持同义词的创建和删除。 支持对表、视图、包、同义词等对象创建同义词。
21	表	支持创建表，包括临时表和全局临时表。创建表可以指定约束、索引等信息。 支持修改表定义，通过ALTER TABLE来添加列，修改列类型，增加或者删除默认值、约束等。 支持删除表。
22	分区表	支持TDSQL PG语法创建一级和多级分区。 支持RANGE、LIST、HASH 分区。 支持分区表的局部索引。 支持分区表的基本维护: - 分区表添加。 - 分区表删除。 - 分区表分裂、合并。
23	视图	支持创建和删除视图。 支持创建 FORCE VIEW。 支持简单视图的DML操作，不支持多表视图的DML。

非模式对象

序号	Oracle数据库	TDSQL PG数据库
1	版本	不支持
2	锁定配置文件	不支持
3	还原点	不支持
4	角色	支持
5	表空间	支持
6	表空间集	不支持
7	统一审计策略	不支持
8	用户	支持，有差异，创建用户时不会自动创建和用户同名的 SCHEMA。

数据库对象名称和限定符

序号	Oracle数据库	TDSQL PG数据库
1	数据库对象命名规则	支持，有差异，长度超过 63 会被截断。
2	模式对象命名规则	支持，有差异，长度超过 63 会被截断。

SQL 语句中模式对象及其组成部分的语法

序号	Oracle数据库	TDSQL PG数据库
1	引用对象的一般语法	支持
2	解析对象的引用规则	支持，有差异，TDSQL PG会按照 search_path 中 schema 的前后顺序来查找对象。
3	引用外部模式对象	支持
4	引用外部数据库对象	支持通过 dblink 访问远程的对象。
5	引用表和索引分区和子分区	支持

伪列

TDSQL PG 兼容层次查询伪列、序列伪列、COLUMN_VALUE 伪列、ROWNUM 伪列，具体如下。

层次查询伪列

序号	Oracle数据库	TDSQL PG数据库
1	CONNECT_BY_ISCYCLE	支持
2	CONNECT_BY_ISLEAF	支持
3	LEVEL	支持

序列伪列

序号	Oracle数据库	TDSQL PG数据库
1	CURRVAL	支持，有差异
2	NEXTVAL	支持，有差异

COLUMN_VALUE伪列

序号	Oracle数据库	TDSQL PG数据库
1	COLUMN_VALUE	支持，有差异

ROWNUM伪列

序号	Oracle数据库	TDSQL PG数据库
1	ROWNUM	支持

操作符

TDSQL PG 兼容绝大多数的Oracle 的运算符，包括算术运算符，连接运算符，集合运算符，其他运算符，具体如下。

SQL运算符

序号	Oracle数据库	TDSQL PG 数据库
1	一元运算符和二元运算符	支持
2	运算符优先级	支持

算术运算符

序号	Oracle 数据库运算符	TDSQL PG 数据库
1	正负一元运算符	支持
2	加减二元运算符	支持
3	乘除二元运算符	支持

COLLATE运算符

序号	Oracle数据库	TDSQL PG 数据库
1	COLLATE collation_name	支持

连接运算符

序号	Oracle 数据库运算符	TDSQL PG 数据库是否支持	描述
1	||	支持	连接运算

集合运算符

序号	Oracle 数据库运算符	TDSQL PG 数据库是否支持	描述
1	UNION	支持	集合求并集，结果集去重
2	UNION ALL	支持	集合求并集，结果集不去重
3	INTERSECT	支持	集合求交集，结果集去重
4	MINUS	支持	集合求差集，结果集去重

多集合运算符

序号	Oracle 数据库运算符	TDSQL PG 数据库是否支持	描述
1	MULTISET UNION ALL	支持	集合类型求并集，结果不去重
2	MULTISET UNION DISTINCT	支持	集合类型求并集，结果去重
3	MULTISET INTERSECT ALL	支持	集合类型求交集，结果不去重
4	MULTISET INTERSECT DISTINCT	支持	集合类型求交集，结果去重
5	MULTISET EXCEPT ALL	支持	集合类型求差集，如果有重复的元素，只移出第一个匹配项
6	MULTISET EXCEPT DISTINCT	支持	集合类型求差集，结果去重

> 注意：
>
> 多集合运算符左右两侧必须是元素类型为 SQL基础类型或嵌套表且不超过二维的 SCHEMA 对象，左右表达式返回的结果类型如果是嵌套表，那么必须是使用 create type 定义的。

比较操作符

序号	Oracle 数据库运算符	TDSQL PG 数据库是否支持
1	=	支持
2	>	支持

序号	Oracle 数据库运算符	TDSQL PG 数据库是否支持
3	<	支持
4	< >	支持
5	> =	支持
6	< =	支持
7	! =	支持

NULL判断操作符

序号	Oracle 数据库运算符	TDSQL PG 数据库是否支持	描述
1	IS NULL	支持	是否为 NULL
2	IS NOT NULL	支持	是否非 NULL

用户自定义操作符

序号	Oracle 数据库	TDSQL PG 数据库是否支持	描述
1	CREATE OPERATOR	支持	创建用户自定义操作符

SQL表达式

TDSQL PG 兼容绝大部分Oracle 的 SQL 表达式，具体如下：

序号	Oracle 数据库	TDSQL PG 数据库	说明
1	字面量	支持	支持字符串，数值，日期，时间戳和间隔字面量。
2	简单表达式	支持	支持常见的简单表达式，包括 ROWNUM，ROWID 伪列，序列，CURRVAL/NEXTVAL。
3	复合表达式	支持	支持多种表达式组成的复合表达式，支持 PRIOR 层次查询。
4	CASE 表达式	支持	支持 CASE WHEN ELSE END 表达式。
5	列表达式	支持	支持表，视图的列作为表达式或者表达式中的一个元素。
6	时间戳表达式	支持	支持时间戳表达式，支持带时区。
7	函数表达式	支持	支持函数作为表达式或者表达式中的一个元素。
8	时间间隔表达式	支持	支持时间间隔表达式。
9	JSON 对象访问表达式	部分支持，有差异	支持通过“->'key'”的方式从JSON对象中提取value。
10	对象表达式	支持	-
11	占位符表达式	部分支持，有差异	支持形如 $1,$2,$3 的占位符表达式。
12	标量子查询表达式	支持	支持标量子查询作为表达式或者表达式的一个元素。
13	类型构造函数表达式	支持	支持对象类型的构造函数作为表达式或者表达式的一个元素。
14	表达式列表	支持	-

条件

TDSQL PG 数据库兼容Oracle大部分条件。

序号	Oracle数据库	TDSQL PG数据库
1	比较条件	支持，有差异。
2	逻辑条件	支持。
3	模式匹配条件	支持。
4	NULL值条件	支持。
5	SQL/JSON条件	部分支持，有差异。
6	复合条件	支持。
7	BETWEEN条件	支持。
8	EXISTS条件	支持。
9	IN条件	支持。

常见SQL语法

DDL

TDSQL PG 兼容 Oracle 中常见的 SQL DDL，具体如下：

序号	Oracle 数据库	TDSQL PG 数据库	说明
1	CREATE DATEBASE	支持，有差异	支持创建数据库指定表空间，不支持指定用户和密码等
2	CREATE TABLE	支持，有差异	支持创建表，不支持 MEMOPTIMIZE FOR 选项
3	CREATE GLOBAL TEMP TABLE	支持	支持创建全局临时表
4	CREATE INDEX	部分支持	支持创建唯一索引和全局索引，不支持创建 BITMAP 索引
5	CREATE VIEW	支持，有差异	支持创建普通视图或者 FORCE 创建视图， 不支持 SHARING 选项
6	CREATE TABLE ... PARTITION BY	支持	支持创建LIST, RANGE, HASH分区表，且创建时可带分区定义和子分区定义
7	CREATE FUNCTION	支持	支持创建函数，可指定 IN/INOUT/OUT 参数
8	CREATE PROCEDURE	支持	支持创建存储过程，可指定 IN/INOUT/OUT 参数
9	CREATE ROLE/ USER	支持，有差异	支持创建角色，不支持 IDENTIFIED BY 子句
10	CREATE SCHEMA	支持	支持创建模式
11	CREATE SEQUENCE	部分支持	支持创建序列，支持 INCREMENT BY/MAXVALUE/ MINVALUE/CYCLE/CACHE/ORDER/NOORDER 选项
12	CREATE SYNONYM	支持	支持创建表，视图，函数，存储过程，包等的同义词
13	CREATE TABLESPACE	支持	支持创建表空间

序号	Oracle 数据库	TDSQL PG 数据库	说明
14	CREATE TRIGGER	支持	支持创建触发器
15	CREATE TYPE	支持	支持创建RECORD，嵌套表，TYPE OBJECT，关联数组等类型
16	CREATE TYPE BODY	支持	支持创建 TYPE OBJECT 的类型体包括构造函数和成员函数
17	CREATE MATERIALIZED VIEW	支持	支持创建物化视图
18	CREATE DATABASE LINK	支持	支持创建 DBLINK
19	CREATE OPERATOR	支持，有差异	支持创建操作符，兼容 PostgreSQL 语法，可以指定 FUNCTION，LEFTARG，COMMTATOR，RESTRICT，HASHES，MERGES 选项
20	CREATE PACKAGE	支持	支持创建包
21	CREATE PACKAGE BODY	支持	支持创建包体
22	ALTER DATABASE	支持，有差异	支持修改数据库的名字，拥有者，设置参数
23	ALTER TABLE	支持，有差异	支持修改表的列类型，增加/删除列，增加/删除约束，支持分区表增加/删除分区/TRUNCATE 分区/EXCHANGE 分区，不支持 HASH 分区表修改分区个数，不支持 ENABLE/DISABLE 选项等
24	ALTER INDEX	部分支持	支持修改索引的属性
25	ALTER MATERIALIZED VIEW	支持，有差异	支持修改物化视图的名字，列的名字和属性，所属的模式，表空间等
26	ALTER OPERATOR	支持，有差异	支持修改运算符的名字，拥有者，所属模式，以及上面 CREATE 语句中的其他选项
27	ALTER PACKAGE	支持	支持修改包的 EDITIONABLE/NONEDITIONABLE 属性，COMPILE 选项
28	ALTER PROCEDURE	部分支持	支持修改存储过程的名字，拥有者以及执行时的参数，不支持 EDITIONABLE/NONEDITIONABLE 属性和 COMPILE 选项

序号	Oracle 数据库	TDSQL PG 数据库	说明
29	ALTER SEQUENCE	部分支持	支持修改序列的 INCREMENT BY/MAXVALUE/MINVALUE/ CYCLE/CACHE/ORDER/NOORDER 选项
30	ALTER ROLE/USER	支持，有差异	支持修改用户的名字和密码，但是不支持 IDENTIFIED 关键字
31	ALTER SESSION	部分支持	只支持修改连接的参数
32	ALTER SYNONYM	部分支持	只支持修改同义词的拥有者
33	ALTER SYSTEM	部分支持	只支持修改全局的参数
34	ALTER TRIGGER	部分支持	只支持修改触发器的名字
35	ALTER TYPE	部分支持	只支持修改类型的名字，拥有者，域名称，删除和增加域
36	ALTER VIEW	部分支持	只支持修改视图的名字，列名和默认值，所属模式，参数值
37	DROP ...	支持	支持删除表，视图，函数，数据库，DBLINK 等上面列举的数据库对象
38	TRUNCATE	支持	支持清空表数据

其中具体涉及到的 SQL DDL 子句，说明如下：

序号	Oracle 数据库	TDSQL PG 数据库	说明
1	CONSTRAINT	支持	约束子句
2	LOGGING/NOLOGGING/ FILESYSTEM_LIKE_LOGGING	部分支持，有差异	支持对应的 LOG/UNLOG 子句，不支持 FILESYSTEM_LIKE_LOGGING
3	PCTFREE/PCTUSED/INITRANS/ MAXTRANS/STORAGE	支持	物理属性子句
4	STORAGE clause	部分支持，有差异	存储子句

DML

TDSQL PG 兼容 Oracle 中常见的 SQL DML，具体如下：

序号	Oracle 数据库	TDSQL PG 数据库	说明
1	SELECT	支持	支持单表和多表连接；支持多种JOIN连接方式（内连接、外连接）； 支持子查询；支持层次查询；支持多种窗口函数等 支持多种集合操作：UNION、UNION ALL、INTERSECT、 EXCEPT、MINUS等 支持EXPLAIN查看执行计划 支持PIVOT、UNPIVOT 支持 CONNECT BY 子句 支持 OFFSET 子句 支持 GROUP BY 子句 支持 OVER 子句 支持并行查询
2	INSERT	支持	支持VALUES插入一行或者多行 支持分区表指定分区插入数据 支持INSERT ... SELECT插入数据 支持同时插入多个目标表（INSERT ALL/FIRST） 支持并行插入
3	UPDATE	支持	SET支持一次更新多列数据 支持分区表指定分区更新数据 支持并行更新
4	DELETE	支持	支持删除表数据 支持分区表指定分区删除数据
5	MERGE INTO	支持	支持根据条件执行插入，删除或者更新操作

TDSQL PG 兼容常见 SQL 查询和子查询，说明如下：

序号	Oracle数据库	TDSQL PG数据库
1	创建简单查询	支持。
2	分层查询	支持。
3	UNION [ALL]，INTERSECT，减运算符	支持。
4	查询结果排序	支持，有差异。
5	Joins	支持，有差异。
6	使用子查询	支持。

序号	Oracle数据库	TDSQL PG数据库
7	嵌套子查询的解嵌套	支持，有差异。
8	分布式查询	支持，有差异。
9	聚集函数嵌套	支持。

其他常见的 SQL语法

TDSQL PG 兼容 Oracle 中其他常见的 SQL语法，具体如下：

序号	Oracle 数据库	TDSQL PG 数据库	说明
1	ANALYZE	部分支持	只支持更新表的统计信息
2	CALL	部分支持	支持调用存储过程，不支持 INTO 子句
3	COMMIT	支持	事务提交
4	ROLLBACK	支持	事务回滚
5	SAVEPOINT	支持	创建恢复点
6	GRANT	支持，有差异	赋予用户权限，兼容 PostgreSQL 语法
7	REVOKE	支持，有差异	取消用户权限，兼容 PostgreSQL 语法
8	SET TRANSACTION	部分支持，有差异	设置事务相关参数，支持 DEFERRABLE/NOT DEFERRABLE/ ISOLATION LEVEL/READ/SNAPSHOT，兼容 PostgreSQL 语法
9	Hint	支持	与 SQL 语句一起使用以更改执行计划

PL/SQL

TDSQL PG支持PL/SQL语法说明：

序号	Oracle 数据库	TDSQL PG 数据库	说明
1	基本语法	支持	支持 PL/SQL 操作符 支持逻辑运算符 支持比较表达式 支持条件表达式 支持变量声明相关参数
2	数据类型	支持	支持sql类型数据类型 支持集合和记录类型 支持object类型 支持oracle兼容的pls_integer和binary_integer
3	流程控制	支持	支持循环控制逻辑 支持条件控制逻辑 支持顺序执行逻辑
4	集合类型与记录类型	支持	支持嵌套表 支持关联数组 支持可变数组 支持记录类型
5	静态 SQL	支持	支持静态SQL
6	动态 SQL	支持	支持动态SQL
7	子过程	支持	支持包内子过程 支持OBJECT类型子过程 支持创建FUNCTION和PROCEDURE
8	触发器	支持	支持insert/update/delete/truncate 4种类型DML触发器 支持alter/drop/create且出发目标是database/schema的ddl触发器 支持before/after/instead of三种触发时机的触发器
9	异常处理	支持	PL/SQL中支持异常块，可以便捷地检查和处理错误
10	PACKAGE对象	支持	支持定义包对象，用来封装一组成员变量和成员函数
11	自定义类型	支持	支持使用CREATE TYPE创建自定义类型
12	游标	支持	支持隐式游标 支持显式游标

序号	Oracle 数据库	TDSQL PG 数据库	说明
			支持自定义游标类型 支持SYS_REFCURSOR
13	函数嵌套声明	支持	支持在函数/存储过程/匿名块中嵌套定义函数或者存储过程
14	PL 系统包	支持	包括 DBMS_OUTPUT、DBMS_LOB、DBMS_ASSERT、DBMS_ALERT、DBMS_LOCK、DBMS_METADATA、DBMS_RANDOM、DBMS_SESSION 等

> 注意：
>
> 不支持条件编译、PL的编译优化级别（PLSQL_OPTIMIZE_LEVEL）。

系统函数

本节主要介绍 TDSQL PG 与原生 Oracle 内建函数的兼容性对比信息。

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持
1	数值函数	支持
2	字符串函数	支持
3	日期函数	支持
4	通用比较函数	支持
5	转换函数	部分支持
6	编码解码函数	支持
7	环境和标识符函数	支持
8	空值相关函数	支持
9	JSON 函数	部分支持
10	XML 函数	部分支持
11	聚合函数	部分支持
12	分析函数	部分支持
13	对象引用函数	部分支持
14	关于用户自定义的函数	支持

数值函数

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
1	ABS(n)	支持	返回指定参数n的绝对值
2	ACOS(n)	支持	返回指定参数n的反余弦值
3	ASIN(n)	支持	返回指定参数n的反正弦值
4	ATAN(n)	支持	返回指定参数n的反正切值

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
5	ATAN2(n1, n2)	支持	返回指定的参数n1 和 参数n2的反正切
6	BITAND(expr1, expr2)	支持	返回两个参数expr1, expr2 按位与运算后的结果
7	CEIL(n)	支持	返回大于等于指定参数n的最小整数
8	COS(n)	支持	返回指定参数n的余弦值
9	COSH(n)	支持	返回指定参数n的双曲余弦值
10	EXP(n)	支持	返回自然常数 e 的指定参数n的次幂
11	FLOOR(n)	支持	返回小于或等于指定参数n的最大整数值
12	LN(n)	支持	返回指定参数n的自然对数
13	LOG(n2, n1)	支持	返回参数n2 基于 参数n1 的对数
14	MOD(n2, n1)	支持	返回参数n2 除以 参数n1 的余数
15	NANVL(n2, n1)	支持	如参数n2 为NaN 则返回 参数n1，否则返回 参数n2
16	POWER(n2, n1)	支持	返回 参数n2 的 参数n1 次幂
17	REMAINDER(n2, n1)	支持	返回 参数n2 除以 参数n1 的余数
18	ROUND(n, integer)	支持	将数字 n 四舍五入到指定的小数位数 integer
19	SIGN(n)	支持	返回指定参数n的符号，如-1，0 或 1，表示负数，零或正数
20	SIN(n)	支持	返回指定弧度值n的正弦值
21	SINH(n)	支持	返回指定数值n的双曲正弦值
22	SQRT(n)	支持	返回指定参数n的平方根
23	TAN(n)	支持	返回指定弧度n的正切
24	TANH(n)	支持	返回指定数值n的双曲正切
25	TRUNC(n1, n2)	支持	将指定数字n1截断到指定精度n2（默认为 0）并返回结果。

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
26	WIDTH_BUCKET(numeric , numeric , numeric , int)	支持	返回指定值 numeric 位于范围 [min_value, max_value]和个数 num_buckets 的桶中的位置。

字符串函数

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
1	1. CHR(n) 1. CHR(n USING NCHAR_CS)	支持	返回指定整数参数n对应的字符，如指定 USING NCHAR_CS，则返回国家字符集中的字符，注意当前国家字符集为数据库字符集。 注意： TDSQL PG 中 chr(0) 的返回结果为空格字符，与Oracle不同。
2	CONCAT(char1, char2)	支持	返回char1 拼接 char2 而成的字符串。
3	INITCAP(char)	支持	返回一个字符串，将指定字符串char中的每个单词的首字母转为大写，其他字母转为小写。
4	LOWER(char)	支持	返回指定字符串char的小写形式。
5	LPAD(expr1, n, expr2)	支持	返回一个字符串，即在字符表达式expr1左侧填充字符表达式expr2，使其长度达到n。
6	LTRIM(char, set)	支持	返回一个字符串，即在字符串char中删除前导字符（默认为空格）set后的字符串。
7	NCHAR(number)	支持	返回一个参数number 对应的 NVARCHAR2 类型的字符
8	NLS_INITCAP(char, 'nlsparam')	支持	返回一个字符串，将指定字符串char中的每个单词的首字母转为大写，其他字母转为小写。
9	NLS_LOWER(char, 'nlsparam')	支持	返回指定字符串char的小写形式。

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
10	NLS_UPPER(char, 'nlsparam')	支持	返回指定字符串char的大写形式。
11	NLSSORT(char, 'nlsparam')	支持	返回指定字符串char的排序规则键。
12	1. REGEXP_REPLACE(source_char, pattern) 1. REGEXP_REPLACE(source_char, pattern, replace_string) 1. REGEXP_REPLACE(source_char, pattern, replace_string, position) 1. REGEXP_REPLACE(source_char, pattern, replace_string, position, occurrence) 1. REGEXP_REPLACE(source_char, pattern, replace_string, position, occurrence, match_param)	支持	在一个源字符串source_char中，用新字符串replace_string替换pattern匹配的内容并返回替换后的内容。
13	1. REGEXP_SUBSTR(source_char, pattern) 1. REGEXP_SUBSTR(source_char, pattern, position) 1. REGEXP_SUBSTR(source_char, pattern, position, occurrence) 1. REGEXP_SUBSTR(source_char, pattern, position, occurrence, match_param) 1. REGEXP_SUBSTR(source_char, pattern, position, occurrence, match_param, subexpr)	支持	从一个给定的源字符串source_char中搜索并返回一个与给定的正则表达式pattern匹配的字符串。
14	REPLACE(char, search_string [, replacement_string])	支持	返回一个字符串，将源字符串中的所有的 search_string 用 replacement_string 替换。
15	RPAD(str, len [, padstr])	支持	返回一个字符串，在源字符串str的右侧填充字符序列padstr（默认为空格），使其具有指定的长度len
16	RTRIM(str [, set])	支持	返回一个字符串，其为删除str 指定的尾随字符set（默认为空格）的字符串。
17	SOUNDEX(str)	支持	将普通字符串str 转换为SOUNDEX字符串。

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
18	SUBSTR(char, position [, substring_length])	支持	从一个字符串char中返回一个从指定位置position开始的指定长度substring_length 的子字符串。
19	TRANSLATE(string, from_string, to_string)	支持	返回一个翻译后的字符串，将 string 中所有在 from_string 字符翻译成在 to_string 中对应的字符。
20	1. TRIM(str) 1. TRIM(trim_character FROM str) 1. TRIM({BOTH / LEADING / TRAILING} trim_character FROM str)	支持	返回一个字符串，其为源字符串str删除前导和尾随字符（默认为空格）set的字符串。
21	UPPER(str)	支持	返回指定字符串str的大写形式
22	ASCII(character)	支持	根据数据库字符集返回字符型数值character第一个字符的ascii码
23	INSTR(string , substring [, position [, occurrence]])	支持	返回一个整数，即子字符串 substring 在字符串 string 中的索引位置。
24	LENGTH(str)	支持	返回指定字符串str的长度
25	1. REGEXP_COUNT(source_char, pattern) 1. REGEXP_COUNT(source_char, pattern, position) 1. REGEXP_COUNT(source_char, pattern, position, match_param)	支持	返回正则表达式pattern匹配值在源字符串source_char中出现的次数。
26	1. REGEXP_INSTR(source_char, pattern) 1. REGEXP_INSTR(source_char, pattern, position) 1. REGEXP_INSTR(source_char, pattern, position, occurrence) 1. REGEXP_INSTR(source_char, pattern, position, occurrence, return_opt) 1. REGEXP_INSTR(source_char, pattern, position, occurrence, return_opt, match_param) 1. REGEXP_INSTR(source_char, pattern, position, occurrence, return_opt, match_param, subexpr)	支持	返回正则表达式pattern匹配值在源字符串source_char中的位置。

日期函数

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
1	ADD_MONTHS(date, months)	支持	在指定日期date上增加或减少指定月数months
2	CURRENT_DATE	支持	返回当前会话时区对应的当前日期
3	CURRENT_TIMESTAMP	支持	返回当前会话时区对应的日期和时间
4	DBTIMEZONE	支持	返回数据库时区
5	EXTRACT(field FROM expr)	支持	从日期时间或间隔表达式 expr 中提取日期字段field的值，field包括：YEAR, MONTH, DAY, HOUR, MINUTE, SECOND。
6	FROM_TZ(timestamp_value, time_zone_value)	支持	将一个时间戳 timestamp_value 和时区 time_zone_value 转换为带有时区的时间戳值。
7	LAST_DAY(date)	支持	返回指定日期date 所在月份的最后一天。
8	1. LOCALTIMESTAMP 1. LOCALTIMESTAMP(precision)	支持	返回当前会话时区中的当前日期和时间。
9	MONTHS_BETWEEN(date1, date2)	支持	返回两个指定日期date1和date2之间的月数
10	NEXT_DAY(date, char)	支持	返回晚于指定日期date 的第一个工作日的日期
11	NUMTODSINTERVAL(n, 'interval_unit')	支持	将指定数字n转为 INTERVAL DAY TO SECOND 文本。
12	NUMTOYMINTERVAL(n, 'interval_unit')	支持	将指定数字n转为 INTERVAL YEAR TO MONTH 文本。
13	1. ROUND(date) 1. ROUND(date, fmt)	支持	将指定日期date 四舍五入到指定的单位。

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
14	SESSIONTIMEZONE	支持	返回当前会话的时区的值，返回类型是时区偏移量或时区区域名称
15	SYS_EXTRACT_UTC(datetime_with_timezone)	支持	从具有时区偏移量或时区名的日期时间值 datetime_with_timezone 中提取 UTC。
16	SYSDATE	支持	返回数据库服务器所在的操作系统设置的当前日期和时间。
17	SYSTIMESTAMP	支持	返回数据库服务器所在的操作系统设置的当前日期和时间，包括小数秒和时区。
18	TO_CHAR(expr [, fmt [, 'nlsparam']])	支持	根据格式fmt 将指定的日期时间或间隔表达式expr 转为字符串。
19	TO_DSINTERVAL(str, [DEFAULT return_value ON CONVERSION ERROR])	支持	将指定的字符串str 转为 INTERVAL DAY TO SECOND 类型的值。
20	TO_TIMESTAMP(str [DEFAULT return_value ON CONVERSION ERROR] [, fmt [, 'nlsparam']])	支持	将指定的字符串str转为 TIMESTAMP 类型的值。
21	TO_TIMESTAMP_TZ(str [DEFAULT return_value ON CONVERSION ERROR] [, fmt [, 'nlsparam']])	支持	将指定字符串str 转为 TIMESTAMP WITH TIME ZONE 类型的值。
22	TO_YMINTERVAL(str, [DEFAULT return_value ON CONVERSION ERROR])	支持	将指定字符串str 转为 INTERVAL MONTH TO YEAR 类型的值。
23	TRUNC(date) TRUNC(date, fmt)	支持	将指定日期date 截断到指定单位。
24	TZ_OFFSET(tz)	支持	返回参数tz对应的时区偏移量。

通用比较函数

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
1	GREATEST(expr [, expr]...)	支持	返回给定参数列表中的最大值
2	LEAST(expr [, expr]...)	支持	返回给定参数列表中的最小值

转换函数

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
1	ASCIISTR(str)	支持	使用数据库字符集将任何字符集中的字符串转换为 ASCII 字符串，非 ASCII 字符转换为 \xxxx 格式，其中 xxxx 表示 UTF-16 代码单元，注意返回值与数据库字符集有关。
2	BIN_TO_NUM(expr [, expr]...)	支持	将指定参数位向量列表转换为数字
3	CAST({ expr } AS type_name)	支持	将指定参数expr从一种类型转换为另一种类型
4	CHARTOROWID(rowid_str)	支持，有差异	将指定字符串 rowid_str 转换为 ROWID 数据类型，注意 rowid_str 字符 与 oracle 格式不同
5	CONVERT(char, dest_char_set[, source_char_set])	支持	将指定字符串char从一个字符集 source_char_set(默认值为数据库字符集)转为另一个字符集 dest_char_set。
6	HEXTORAW(str)	支持	将指定参数str的十六进制值转为原始值。
7	NUMTODSINTERVAL(n, 'interval_unit')	支持	把参数 n 转为以参数 interval_unit 为单位的 INTERVAL DAY TO SECOND 数据类型的值。
8	NUMTOYMINTERVAL(n, 'interval_unit')	支持	把参数 n 转为以 interval_unit 为单位的 INTERVAL YEAR TO MONTH 数据类型的值。
9	RAWTOHEX(raw)	支持	将原始值raw转换为一个包含其十六进制表示的字符值。
10	ROWIDTOCHAR(rowid)	支持，有差异	将给定 rowid 值转为 VARCHAR2 类型，注意 rowid 与 oracle 格式不同。

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
11	TO_BINARY_DOUBLE(expr [DEFAULT return_value ON CONVERSION ERROR] [, fmt [, 'nlsparam']])	部分支持，有差异	将表达式expr 转换为双精度浮点数，注意差异包括： 1. nlsparam参数仅支持 "NLS_NUMERIC_CHARACTERS" 1. 存在输出结果表示方式不同，Oracle使用科学计数法表示。
12	TO_BINARY_FLOAT(expr [DEFAULT return_value ON CONVERSION ERROR] [, fmt [, 'nlsparam']])	部分支持，有差异	将表达式expr 转换为浮点数，注意差异包括： 1. 存在少部分fmt 格式与 Oracle 未匹配 1. nlsparam参数仅支持 "NLS_NUMERIC_CHARACTERS" 1. 存在输出结果表示方式不同，Oracle使用科学计数法表示。
13	TO_CHAR (character)	支持	将 NCHAR、 NVARCHAR2，或 CLOB 数据character转换为 VARCHAR2 数据类型。
14	TO_CHAR(date [, fmt [, 'nlsparam']])	部分支持	将日期时间类型或间隔类型date 根据格式fmt 转化为字符串，注意差异包括： 1. 存在部分 fmt 格式 未兼容，如'TS DS'，TZR 等 1. nlsparam 参数仅支持 'nls_date_language = ''American'' '
15	TO_CHAR(n [, fmt [, 'nlsparam']])	部分支持	将数值n 根据格式fmt 转换为 VARCHAR2 类型的值，注意 'nlsparam' 参数 仅支持 NLS_NUMERIC_CHARACTERS = ''dg''
16	TO_DATE(char [DEFAULT return_value ON CONVERSION ERROR] [, fmt [, 'nlsparam']])	部分支持	将日期字符串char 根据格式fmt 转为 日期 DATE 类型的值，注意 'nlsparam' 参数 仅支持 'NLS_DATE_LANGUAGE = American'
17	TO_DSINTERVAL(str, [DEFAULT return_value ON CONVERSION ERROR])	支持	将字符串 str 转为 INTERVAL DAY TO SECOND 类型的值。
18	TO_MULTI_BYTE(char)	支持	将指定参数char中的所有单字节字符转换为相应的多字节字符，并返回与参数相同数据类型的值。
19	TO_NUMBER(expr [DEFAULT return_value ON CONVERSION	部分支持	将指定参数expr 根据格式fmt 转换为 NUMBER 类型的值，注意 'nlsparam' 参

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
	ERROR] [, fmt [, 'nlsparam']])		数 仅支持 NLS_NUMERIC_CHARACTERS = ''dg''
20	TO_SINGLE_BYTE(char)	支持	将指定参数char中的所有多字节字符转换为相应的单字节字符，并返回与参数相同数据类型的值。
21	TO_TIMESTAMP(str [DEFAULT return_value ON CONVERSION ERROR] [, fmt [, 'nlsparam']])	部分支持	将指定字符串str 按格式fmt 转换为 TIMESTAMP 类型的值，注意 nlsparam 参数 仅支持 'nls_date_language = ''American'' '
22	TO_TIMESTAMP_TZ(str [DEFAULT return_value ON CONVERSION ERROR] [, fmt [, 'nlsparam']])	部分支持	将指定字符串str 按格式fmt 转为 TIMESTAMP WITH TIME ZONE 类型的值，注意nlsparam 参数 仅支持 'nls_date_language = ''American'' '
23	TO_YMINTERVAL(str, [DEFAULT return_value ON CONVERSION ERROR])	支持	将指定字符串str转为 INTERVAL MONTH TO YEAR 类型的值。

编码解码函数

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
1	DUMP(expr[, return_fmt [, start_position [, length]]])	支持	返回一个 字符串，其包含 expr 的数据类型代码、字节长度和内部表示。
2	VSIZE(expr)	支持	返回表达式expr 内部表示的字节数。
3	DECODE (expr, search, result [, search, result ...][, default])	支持	根据一个或多个指定的映射关系将指定的参数解码，然后返回解码后的值。
4	ORA_HASH	支持	计算指定表达式expr的哈希值

环境和标识符函数

序号	Oracle 数据库函数	TDSQL PG	描述

		数据库是否支持	
1	UID	支持	返回当前会话用户的唯一标识。
2	USER	支持	返回当前会话用户的名称
3	USERENV	支持	返回用户环境信息
4	SYS_CONTEXT('namespace', 'parameter' [, length])	部分支持	返回给定上下文相关联参数parameter的值，注意 'namespace' 参数 仅支持 'USERENV' && 'XS$SESSION'
5	SYS_GUID	支持	生成并返回一个由 16 个字节组成的全局唯一标识符。

空值相关的函数

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
1	COALESCE(expr1[, expr2 ...])	支持	返回参数列表中第一个非 NULL 值，最少必须指定两个参数。
2	LNNVL(condition)	支持	如果条件为 TRUE 则返回 FALSE； 如果条件为 FALSE 或者 UNKNOWN 则返回 TRUE。
3	NANVL(num1, num2)	支持	如果 num1 是 NaN， 返回 num2，否则返回 num1。
4	NULLIF(expr1, expr2)	支持	比较两个参数，如果两个参数相等则返回 NULL，否则返回第一个参数。
5	NVL(expr1, expr2)	支持	如果 expr1 为 NULL，则返回 expr2。 如果 expr1 不为 NULL，则 NVL 返回 expr1。
6	NVL2(expr1, expr2, expr3)	支持	如果 expr1 不为 NULL， 则返回 expr2，否则返回 expr3。

JSON 函数

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
1	JSON_OBJECT	支持	返回一个 JSON 对象，其成员为参数指定的键值对。

XML 函数

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
1	EXTRACT(XMLType_instance, XPath_string, namespace_string)	支持	按 XPath_string 提取 XMLType_instance 内容，返回类型是XMLtype
2	EXTRACTVALUE(XMLType_instance, XPath_string, namespace_string)	支持	按 XPath_string 提取 XMLType_instance 中的 XML 值
3	XMLSEQUENCE	支持	提取 XML 数据的所有顶层节点，将其转换为一个可迭代的序列。通常与 TABLE、EXTRACT 等函数配合使用。
4	XMLAGG	支持	XMLAgg 是一个聚合函数， 接受 XML 片段的集合并返回聚合的 XML 文档
5	XMLEXISTS	支持	检查给定的 XPath 表达式是否返回任意节点，如果是，返回 TRUE； 否则，返回 FALSE。
6	XMLELEMENT	支持	生成具有给定名称、属性和内容的 XML 元素。
7	XMLPARSE	支持	解析 XML 文档或片段，返回一个 XMLType。
8	XMLTABLE	支持	使用 XPATH 表达式提取输入的 XML 数据，并且将提取到的数据分解为表中的列，返回一张虚拟表。

聚合函数

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
1	APPROX_COUNT_DISTINCT(expr)	支持	对参数expr 列去重后的行进行计算，返回行数，且该值是近似值，用于估算一个列或表达式中不同值的数量。
2	AVG([DISTINCT | ALL] expr)	支持	用于计算expr 平均值。
3	CORR(expr1, expr2)	支持	返回一组数字对(expr1, expr2)的相关系数
4	COUNT({ * | [DISTINCT | ALL] expr })	支持	返回查询 expr 的行数
5	COVAR_POP(expr1, expr2)	支持	用于计算一组数值对的总体协方差
6	COVAR_SAMP(expr1, expr2)	支持	用于计算一组数值对的样本协方差
7	CUME_DIST	支持	用于计算一组值中的某个值的累积分布
8	DENSE_RANK	支持	计算有序行组中某一行的排名，并以数字形式返回排名
9	FIRST	支持	用于对一组行中的值进行操作，将这组行按照指定的排序规则排序后返回排在第一值
10	GROUPING_ID	支持	用于计算一组表达式的组合值，返回一个数字，代表行所处的 GROUP BY 层级。
11	GROUPING	支持	标识结果集中的行是否由 ROLLUP、CUBE 或 GROUPING SETS 生成的聚合行，通过返回 0 或 1 来区分普通分组行和超级聚合行（如小计、总计）
12	LAST	支持	用于对一组行中的值进行操作，将这组行按照指定的排序规则排序后返回排在最后的值
13	LISTAGG	支持	用于列转行，将多行数据合并为单个字符串 的聚合函数
14	MAX	支持	用于计算最大值
15	MEDIAN	支持	用于返回一组数值的中值，即将一组数值排序后返回居于中间的数值。

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
16	MIN	支持	用于计算最小值
17	PERCENT_RANK	支持	用来计算一组值中的某个值的累积分布，计算某一个数在一个集合中的对应位置百分比，返回值范围为 (0, 1]。
18	PERCENTILE_CONT	支持	一个假定连续分布模型的逆分布函数，根据指定百分比值和排序规范，返回一个在该排序规范的给定百分比值的内插值，在计算中忽略空值。
19	PERCENTILE_DISC	支持	根据一个百分位值和指定排序规范，返回大于或等于百分位值的最小累积分布值（相对于同一排序规范）的值，在计算中忽略空值。
20	RANK	支持	用于计算一个值在一组值的排名。
21	STDDEV_POP	支持	将数值型数据作为参数计算总体标准差。
22	STDDEV_SAMP	支持	将数值型数据作为参数计算样本标准差。
23	STDDEV	支持	用于计算一组数值型数据标准差。
24	SUM([DISTINCT | ALL] expr)	支持	返回expr值的总和。
25	VAR_POP	支持	返回一组数值集合的总体方差
26	VAR_SAMP	支持	返回一组数值集合的样本方差
27	VARIANCE	支持	返回指定列的方差
28	XMLAGG	支持	接收 XML 片段的集合并返回聚合的 XML 文档

分析函数

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
1	AVG	支持	计算指定列expr 的平均值

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
2	CORR(expr1, expr2)	支持	用于计算一组数字对(expr1, expr2)的皮尔逊相关系数
3	COUNT(expr)	支持	统计指定列 expr的行数
4	COVAR_POP	支持	用于计算一组数值对的总体协方差
5	COVAR_SAMP	支持	用于计算一组数值对的样本协方差
6	CUME_DIST	支持	用于计算一组值中的某个值的累积分布
7	DENSE_RANK	支持	计算有序行组中某一行的排名，并以数字形式返回排名。
8	FIRST_VALUE	支持	在分析语句analytic_clause 中的每个分组（窗口）内，按指定规则排序，返回第一行 expr 值
9	FIRST	支持	用于对一组行中的值进行操作，按照指定的排序规则进行排序，返回排在第一的值
10	LAG	支持	在某个分组（窗口）内，按指定规则排序，返回列 expr 往前偏移 offset 行的数据
11	LAST_VALUE	支持	在analytic_clause 分析语句中的每个分组（窗口）内，按指定规则排序，返回最后一行的 expr 值
12	LAST	支持	用于对一组行中的值进行操作，将这组行按照指定的规则排序，返回排在最后的值
13	LEAD	支持	在某个分组（窗口）内，按指定规则排序，返回列 expr 往后偏移 offset 行的数据
14	LISTAGG	支持	用于列转行，将多行数据合并为单个字符串 的聚合函数
15	MAX	支持	计算指定列expr 在某个分组（窗口）内的最大值
16	MEDIAN	支持	计算指定列expr 在某个分组（窗口）内的中间值
17	MIN	支持	计算指定列expr 在某个分组（窗口）内的最小值
18	NTH_VALUE	支持	返回 analytic_clause 定义的窗口中第 n 行的expr 值
19	NTILE	支持	将有序数据集划分为若干个组，并为每一行分配适当的组号
20	PERCENT_RANK	支持	计算一组值中的某个值的累积分布

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
21	RANK	支持	计算一个值在一组值的排名
22	RATIO_TO_REPORT	支持	计算一个值与一组值之和的比率
23	REGR_(Linear Regression)	支持	一组 REGR_ 开头的线性回归函数，用于计算简单线性回归模型的统计指标（如斜率、截距、决定系数等）。
24	ROW_NUMBER	支持	为查询结果的每一行分配一个唯一的序号，序号从 1 开始，根据指定的排序规则（ORDER BY）和分组规则（PARTITION BY）动态生成。
25	STDDEV_POP	支持	将数值型数据作为参数计算总体标准差，总体标准差是总体方差的算术平方根。
26	STDDEV_SAMP	支持	将数值型数据作为参数计算样本标准差，样本标准差是样本方差的算术平方根。
27	STDDEV	支持	用于计算一组数值型数据标准差
28	SUM	支持	计算指定列expr 在某个分组（窗口）内的和
29	VAR_POP	支持	返回一组数值集合的总体方差
30	VAR_SAMP	支持	返回一组数值集合的样本方差
31	VARIANCE	支持	返回指定列的方差

对象引用函数

序号	Oracle 数据库函数	TDSQL PG 数据库是否支持	描述
1	VALUE	支持	VALUE 函数通常与 对象类型（Object Types）或嵌套表（Nested Tables）相关，用于直接引用某个对象实例的值

系统视图

本节主要介绍 TDSQL PG 数据库与原生 Oracle 数据库中静态视图的详细兼容列表。

静态视图

TDSQL PG 数据库兼容 Oracle 数据库的静态视图列表说明如下：

序号	Oracle数据库	TDSQL PG数据库	说明
1	ALL_VIEWS	支持	描述当前用户可以访问的所有视图。需要确保安装tbase_oracle_package_function插件。
2	ALL_USERS	支持	当前用户可访问的所有数据库用户。
3	ALL_TABLES	支持	描述当前用户可访问的表。需要确保安装tbase_oracle_package_function插件。
4	ALL_TAB_COMMENTS	支持	当前用户可访问的表或视图的所有列注释。需要确保安装tbase_oracle_package_function插件。
5	ALL_TAB_COLUMNS	支持	当前用户可访问的表，视图列，过滤系统隐藏列和虚拟列
6	ALL_SYNONYMS	支持	描述当前用户可访问的所有同义词。
7	ALL_SEQUENCES	支持	描述当前用户可访问的所有序列。需要确保安装tbase_oracle_package_function插件。
8	ALL_INDEXES	支持	描述当前用户可访问的所有表上的索引信息。
9	ALL_IND_COLUMNS	支持	查看用户可访问的所有表的索引的索引列信息。
10	ALL_CONSTRAINTS	支持	显示用户可访问的所有表约束定义。
11	ALL_CONS_COLUMNS	支持	查看用户可访问的所有表的约束中的列信息。
12	ALL_COL_COMMENTS	支持	当前用户可访问的表或视图的列的注释。
13	ALL_ARGUMENTS	支持	当前用户可访问的函数和过程的参数列表。需要确保安装tbase_oracle_package_function插件。
14	DBA_VIEWS	支持	描述数据库中的所有视图。需要确保安装tbase_oracle_package_function插件。
15	DBA_USERS	支持	描述数据库的所有用户，并且比ALL_USERS包含更多的列。

序号	Oracle数据库	TDSQL PG数据库	说明
16	DBA_TABLES	支持	描述数据库中的所有关系表。需要确保安装tbase_oracle_package_function插件。
17	DBA_TAB_COMMENTS	支持	显示数据库中所有表和视图的注释。需要确保安装tbase_oracle_package_function插件。
18	DBA_TAB_COLUMNS	支持	描述数据库中所有表，视图的列，其列与“ALL_TAB_COLUMNS”中的列相同。
19	DBA_SYNONYMS	支持	描述数据库中的所有同义词。
20	DBA_SEQUENCES	支持	描述数据库中的所有序列。需要确保安装tbase_oracle_package_function插件。
21	DBA_INDEXES	支持	描述数据库中的所有索引。
22	DBA_IND_COLUMNS	支持	描述数据库中所有表上的所有索引列信息。
23	DBA_CONSTRAINTS	支持	显示当前数据库上所有约束
24	USER_VIEWS	支持	描述当前用户拥有的视图，不显示“OWNER”列。需要确保安装tbase_oracle_package_function插件。
25	USER_USERS	支持	描述当前用户，并且包含的列多于ALL_USERS。
26	USER_TABLES	支持	描述了当前用户拥有的关系表。需要确保安装tbase_oracle_package_function插件。
27	USER_TAB_COMMENTS	支持	当前用户拥有的表和视图上列的注释，不显示“OWNER”列。需要确保安装tbase_oracle_package_function插件。
28	USER_TAB_COLUMNS	支持	描述当前用户拥有的表，视图列，不显示“OWNER”列。
29	USER_SYNONYMS	支持	描述当前用户拥有的同义词，不显示OWNER列。
30	USER_SEQUENCES	支持	描述了当前用户拥有的所有序列，不显示SEQUENCE_OWNER列。需要确保安装tbase_oracle_package_function插件。
31	USER_INDEXES	支持	描述当前用户拥有的所有索引。
32	USER_IND_COLUMNS	支持	描述当前用户拥有的索引的列以及当前用户拥有的表的索引的列，不显示INDEX_OWNER或TABLE_OWNER列。
33	USER_CONSTRAINTS	支持	显示当前用户所拥有的所有约束信息

序号	Oracle数据库	TDSQL PG数据库	说明
34	USER_CONS_COLUMNS	支持	显示用户约束列信息
35	USER_COL_COMMENTS	支持	当前用户拥有的表或视图的列的注释，不显示OWNER 字段

系统包

系统包指的是 TDSQL PG 数据库提供的兼容Oracle功能的一系列内置PL/SQL 程序包，包含程序、过程、函数、类型和变量，用于执行各种数据库级别的任务，如数据访问、数据处理、安全性、性能监控和数据库管理等。
举例常见的 TDSQL PG 系统包如DBMS_OUTPUT，其用于在 PL/SQL 代码块或存储过程中向用户显示信息。
使用方法如下：

```
-- output 'Hello, TDSQL PG!' in client
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello, TDSQL PG!');
END;
/
```

下表列举TDSQL PG目前支持的兼容oracle系统包列表，具体每个包的使用方法见参考手册。其中，使用 DBMS_JOB 包需要确保已经安装 tbase_dbms_job 插件，使用列表中的其它包需要确保已经安装 tbase_oracle_package_function 插件。

序号	Oracle 数据库	TDSQL PG 数据库	说明
1	DBMS_APPLICATION_INFO	支持	该包允许应用程序为执行上下文设置特定的模块和客户端信息，方便后续的问题定位和性能跟踪。
2	DBMS_JOB	部分支持	使用该包来执行一些定时任务。
3	DBMS_OUTPUT	支持	使用该包来进行信息输出，方便调试和问题定位。
4	DBMS_ASSERT	支持	该包主要用来验证特定字符串是否是合乎语法语义规则，不同的字符串有不同的验证规则。
5	DBMS_ALERT	支持	该包用来订阅和发送消息，结合TRIGGER等使用，可以感知数据库发生的数据变化。
6	DBMS_PIPE	支持	使用该包来创建一个消息管道用来进行消息传递。
7	DBMS_RANDOM	支持	使用该包来生成一串符合正态分布的随机数，并能基于该随机数生成一些随机字符串。
8	DBMS_UTILITY	部分支持	该包提供了一些工具函数，如FORMAT STACK等。
9	DBMS_LOB	部分支持	该包提供了一些工具函数用来操作LOB数据类型。注意由于TDSQL PG和ORACLE底层实现的不同，某些函数可能未实现或行为和ORACLE不一致。

序号	Oracle 数据库	TDSQL PG 数据库	说明
10	DBMS_LOCK	部分支持	该包提供了一些工具函数，可以用来向lock manager获取锁，帮助在PL中写一些需要锁支持的逻辑。注意目前只支持sleep函数。
11	DBMS_SESSION	部分支持	该包提供一些工具函数用来操纵SESSION内的部分数据。注意TDSQL PG目前只支持UNIQUE_SESSION_ID，RESET_PACKAGE，FREE_UNUSED_USER_MEMORY三个函数。
12	DBMS_STATS	部分支持	该包提供一些工具函数用来进行统计信息搜集。
13	DBMS_METADATA	部分支持	该包提供一些工具函数，用来获取系统内部的一些SCHEMA信息。
14	DBMS_ROWID	部分支持	该包提供了一些工具函数，来创建和解析rowid。由于TDSQL PG和oracle底层实现的不同，某些函数行为不一定相同。
15	UTL_FILE	部分支持	该包提供一些工具函数，用来读写操作系统的文件。
16	UTL_RAW	部分支持	该包提供一些工具函数，用来操作RAW数据类型。
17	XMLDOM	部分支持	该包提供一些 XML 文档处理函数，目前支持 getElementsByTagName，getLength，item，getChildNodes，getFirstChild，getNodeValue，getNodeName，freeDocument 等。
18	XMLPARSER	部分支持	该包提供一些 XML 解析函数，目前支持parseClob，getDocument，freeParser 等。

Oracle兼容性配置参数

本节主要介绍TDSQL PG 数据库的 Oracle 模式中需要配置参数。
TDSQL PG的参数配置主要有以下几种参数级别， session级 ， server级 ， internal级 ：

- session级：
  - 普通用户配置：用户在当前会话设置，只在当前会话生效，不会影响其他session。使用方法： set parameter = value 。
  - session级（ superuser ）配置：session级的特殊情况，相关参数只能由superuser设置，普通用户设置时会报错。
- server级：
  - 方法1：superuser通过 alter system 关键字命令设置，使用方法： alter system set parameter = value ，修改后需要重新启动数据库服务器才能够生效。

    > 说明
    >
    > alter system 修改后只是将该参数写到 postgresql.auto.conf 文件里，故需要重启才可以生效。

  - 方法2：手动配置到 postgresql.conf 里，修改后重启数据库生效。

    > 说明
    >
    > 由于 postgresql.auto.conf 文件里的内容，在执行 alter system reset all; 会全部清除，故若需要手动配置，最好不要在 postgresql.auto.conf中修改。

- internal级：内部级别，不允许用户手动进行设置。
  TDSQL PG 数据库的 Oracle 模式中可配置参数参考下表：

序号	参数名称	参数类型	参数级别	取值范围	参数说明
1	allow_limit_ident	布尔	session级	on/off	是否允许 limit 做标识符，默认 off
2	allow_unmatched_block_comments	布尔	session级	on/off	是否允许c风格注释 /* */ 不配对出现，默认off
3	case_as_alias	布尔	session级	on/off	是否允许用case做别名，默认off
4	convert_sysview_to_sysclass	布尔	session级	on/off	在oracle 模式下是否支持 regclass 函数把 PG_CLASS 这种系统视图转化为 pg_class 系统表，默认on

序号	参数名称	参数类型	参数级别	取值范围	参数说明
5	dblink_types	枚举	session级	oracle_x86 oracle_arm postgresql tdsql_pg	DBLink连接的是oracle还是postgresql，需要在 CREATE DATABASE LINK之前设置，默认oracle_x86
6	default_sql_mode	枚举	session级	oracle/postgresql	CREATE DATABASE 不指定mode和template时的默认模式，默认postgresql
7	default_with_rowid	布尔	session级	on/off	创建的表是否带有ROWID列，默认值off
8	different_func_argret	布尔	session级	on/off	函数的out参数行为是否兼容oracle，默认off
9	disable_empty_to_null	布尔	session级	on/off	是否禁止空串('')转换为null，默认off
10	do_as_alias	布尔	session级	on/off	是否允许用do做别名，默认off
11	enable_create_package_rec_typ	布尔	session级	on/off	在包头中 用户定义 record类型是否允许定义到pg_type 中，这样包外也可以使用该record 类型，默认off
12	enable_distinct_order_by	布尔	session级	on/off	检查order by的列与distinct的列是否匹配，默认off
13	enable_function_overload	布尔	session级	on/off	是否允许函数重载，默认on
14	enable_inline_target_function	布尔	session级(superuser)	on/off	是否允许将targetlist中对应的sql function转化为子连接，默认值off

序号	参数名称	参数类型	参数级别	取值范围	参数说明
15	enable_nestedtable_check	布尔	session级	on/off	SQL中引用包函数，而该函数包含嵌套类型，将导致CachePlan失败，在循环中执行将比较耗时，通过该配置规避cacheplan问题，默认on
16	enable_oracle_column_name	布尔	session级	on/off	查询返回的列名是否完全兼容oracle，默认off
17	enable_oracle_dist_udaf	布尔	session级	on/off	是否允许在分布式环境创建oracle自定义聚合函数，默认off
18	enable_oracle_implicit_coercion	布尔	session级	on/off	是否使用oracle隐式转换，默认on
19	enable_oracle_pkg_nested_table	布尔	session级	on/off	包中嵌套表类型是否在pg_type中记录一条类型信息，默认off
20	enable_plsql_analysis	布尔	session级	on/off	是否在创建函数或者存储过程时支持PLSQL 静态检查，默认off
21	enable_secfunc_xact	布尔	session级(superuser)	on/off	是否允许在security definer函数中使用commit/rollback，默认on
22	enable_type_priority_in_ora_mode	布尔	session级	on/off	是否在oracle 模式下支持类型优先级判断，默认on
23	enable_typobj_func_verify	布尔	session级	on/off	检查type object的spec和body的函数是否匹配，默认on
24	function_xact_control	布尔	session级	on/off	函数内是否支持COMMIT/ROLLBACK，默

序号	参数名称	参数类型	参数级别	取值范围	参数说明
					认on
25	keywords_as_ident	布尔	session级	on/off	是否允许关键字做标识符，默认on
26	max_autonomous_transactions	整型	server级	0 ~ 536870911	支持的自治事务最大数，默认100
27	nls_date_format	字符串	session级	-	date的输入输出格式，默认YYYY-MM-DD HH24:MI:SS
28	nls_sort_locale	字符串	session级	SCHINESE_PINYIN_M SCHINESE_STROKE_M SCHINESE_RADICAL_M	nlssort函数使用的排序规则，默认空
29	nls_timestamp_format	字符串	session级	-	timestamp的输入输出格式，默认YYYY-MM-DD HH24:MI:SS.US
30	nls_timestamp_tz_format	字符串	session级	-	timestamptz的输入输出格式，默认YYYY-MM-DD HH24:MI:SS.US TZ
31	only_rename_typobj_metainfo	布尔	session级	on/off	alter type object rename的时候，是否只改名字，不改其它信息。调试使用，正常情况下不要使用该参数， 默认off
32	plpgsql_hold_cursor	布尔	session级	on/off	事务提交的时候是否保持游标，默认on
33	plsql_emptystring_to_blank	布尔	session级	on/off	PLSQL中空串仍然是空串，不会转换成null，默认on
34	profile_password_limit	布尔	session级 (superuser)	on/off	是否启用密码限制，默认off

序号	参数名称	参数类型	参数级别	取值范围	参数说明
35	profile_resource_limit	布尔	session级 (superuser)	on/off	是否启用资源限制，默认off
36	skip_check_same_relname	布尔	session级	on/off	是否跳过检测sql中存在相同的别名表，默认on
37	sql_mode	枚举	internal级	oracle/postgresql	该参数仅用于显示当前连接的数据库的模式，不允许修改，无默认值，依赖于连接的数据库类型
38	nls_date_language	字符串	session级	-	该参数用于指定日期和时间以哪种形式输出月份和星期几，默认是 American

API文档
TBase部署中心（tbase）
版本（2019-01-07）
API 概览

API版本

V3

TBase部署中心

接口名称	接口功能
DescribeAvailableRecoveryTime	获取回档时间
DescribeAvailableRegion	查询售卖地域信息

调用方式
接口签名v1

TCloudFinanceZone API 会对每个访问请求进行身份验证，即每个请求都需要在公共请求参数中包含签名信息（Signature）以验证请求者身份。
签名信息由安全凭证生成，安全凭证包括 SecretId 和 SecretKey；若用户还没有安全凭证，请前往云API密钥页面申请，否则无法调用云API接口。

1. 申请安全凭证

在第一次使用云API之前，请前往云API密钥页面申请安全凭证。
安全凭证包括 SecretId 和 SecretKey：

- SecretId 用于标识 API 调用者身份
- SecretKey 用于加密签名字符串和服务器端验证签名字符串的密钥。
- 用户必须严格保管安全凭证，避免泄露。

申请安全凭证的具体步骤如下：

1. 登录TCloudFinanceZone管理中心控制台。
2. 前往云API密钥的控制台页面
3. 在云API密钥页面，点击【新建】即可以创建一对SecretId/SecretKey

注意：开发商帐号最多可以拥有两对 SecretId / SecretKey。

2. 生成签名串

有了安全凭证SecretId 和 SecretKey后，就可以生成签名串了。以下是生成签名串的详细过程：

假设用户的 SecretId 和 SecretKey 分别是：

- SecretId: AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE
- SecretKey: Gu5t9xGARNpq86cd98joQYCN3EXAMPLE

注意：这里只是示例，请根据用户实际申请的 SecretId 和 SecretKey 进行后续操作！

以云服务器查看实例列表(DescribeInstances)请求为例，当用户调用这一接口时，其请求参数可能如下:

参数名称	中文	参数值

参数名称	中文	参数值
Action	方法名	DescribeInstances
SecretId	密钥Id	AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE
Timestamp	当前时间戳	1465185768
Nonce	随机正整数	11886
Region	实例所在区域	shjr
InstanceIds.0	待查询的实例ID	ins-09dx96dg
Offset	偏移量	0
Limit	最大允许输出	20
Version	接口版本号	2017-03-12

2.1. 对参数排序

首先对所有请求参数按参数名的字典序（ ASCII 码）升序排序。注意：1）只按参数名进行排序，参数值保持对应即可，不参与比大小；2）按 ASCII 码比大小，如 InstanceIds.2 要排在 InstanceIds.12 后面，不是按字母表，也不是按数值。用户可以借助编程语言中的相关排序函数来实现这一功能，如 php 中的 ksort 函数。上述示例参数的排序结果如下:

```
{
    'Action' : 'DescribeInstances',
    'InstanceIds.0' : 'ins-09dx96dg',
    'Limit' : 20,
    'Nonce' : 11886,
    'Offset' : 0,
    'Region' : 'shjr',
    'SecretId' : 'AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE',
    'Timestamp' : 1465185768,
    'Version': '2017-03-12',
}
```

使用其它程序设计语言开发时，可对上面示例中的参数进行排序，得到的结果一致即可。

2.2. 拼接请求字符串

此步骤生成请求字符串。
将把上一步排序好的请求参数格式化成“参数名称”=“参数值”的形式，如对 Action 参数，其参数名称为 "Action" ，参数值为 "DescribeInstances" ，因此格式化后就为 Action=DescribeInstances 。
注意：“参数值”为原始值而非url编码后的值。

然后将格式化后的各个参数用"&"拼接在一起，最终生成的请求字符串为:

```
Action=DescribeInstances&InstanceIds.0=ins-09dx96dg&Limit=20&Nonce=11886&Offset=0&Region=shjr&SecretId=AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE&Timestamp=1465185768&Version=2017-03-12
```

2.3. 拼接签名原文字符串

此步骤生成签名原文字符串。
签名原文字符串由以下几个参数构成:

1. 请求方法: 支持 POST 和 GET 方式，这里使用 GET 请求，注意方法为全大写。
2. 请求主机:查看实例列表(DescribeInstances)的请求域名为：cvm.finance.cloud.tencent.com。实际的请求域名根据接口所属模块的不同而不同，详见各接口说明。
3. 请求路径: 当前版本云API的请求路径固定为 / 。
4. 请求字符串: 即上一步生成的请求字符串。

签名原文串的拼接规则为: 请求方法 + 请求主机 +请求路径 + ? + 请求字符串

示例的拼接结果为：

```
GETcvm.finance.cloud.tencent.com/?Action=DescribeInstances&InstanceIds.0=ins-09dx96dg&Limit=20&Nonce=11886&Offset=0&Region=shjr&SecretId=AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE&Timestamp=1465185768&Version=2017-03-12
```

2.4. 生成签名串

此步骤生成签名串。
首先使用 HMAC-SHA1 算法对上一步中获得的签名原文字符串进行签名，然后将生成的签名串使用 Base64 进行编码，即可获得最终的签名串。

具体代码如下，以 PHP 语言为例:

```
$secretKey = 'Gu5t9xGARNpq86cd98joQYCN3EXAMPLE';
$srcStr = 'GETcvm.finance.cloud.tencent.com/?Action=DescribeInstances&InstanceIds.0=ins-09dx96dg&Limit=20&Nonce=11886&Offset=0&Region=shjr&SecretId=AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE&Timestamp=1465185768&Version=2017-03-12';
$signStr = base64_encode(hash_hmac('sha1', $srcStr, $secretKey, true));
echo $signStr;
```

最终得到的签名串为:

```
EliP9YW3pW28FpsEdkXt/+WcGeI=
```

使用其它程序设计语言开发时，可用上面示例中的原文进行签名验证，得到的签名串与例子中的一致即可。

3. 签名串编码

生成的签名串并不能直接作为请求参数，需要对其进行 URL 编码。

如上一步生成的签名串为 EliP9YW3pW28FpsEdkXt/+WcGeI= ，最终得到的签名串请求参数（Signature）为：EliP9YW3pW28FpsEdkXt%2f%2bWcGeI%3d，它将用于生成最终的请求 URL。

注意：如果用户的请求方法是 GET，或者请求方法为 POST 同时 Content-Type 为 application/x-www-form-urlencoded，则发送请求时所有请求参数的值均需要做 URL 编码，参数键和=符号不需要编码。非 ASCII 字符在 URL 编码前需要先以 UTF-8 进行编码。

注意：有些编程语言的 http 库会自动为所有参数进行 urlencode，在这种情况下，就不需要对签名串进行 URL 编码了，否则两次 URL 编码会导致签名失败。

注意：其他参数值也需要进行编码，编码采用 RFC 3986。使用 %XY 对特殊字符例如汉字进行百分比编码，其中“X”和“Y”为十六进制字符（0-9 和大写字母 A-F），使用小写将引发错误。

4. 签名失败

根据实际情况，存在以下签名失败的错误码，请根据实际情况处理

错误代码	错误描述
AuthFailure.SignatureExpire	签名过期
AuthFailure.SecretIdNotFound	密钥不存在
AuthFailure.SignatureFailure	签名错误
AuthFailure.TokenFailure	token 错误
AuthFailure.InvalidSecretId	密钥非法（不是云 API 密钥类型）

5. 签名演示

在实际调用 API 3.0 时，推荐使用配套的TCloudFinanceZone SDK 3.0 ，SDK 封装了签名的过程，开发时只关注产品提供的具体接口即可。详细信息参见 SDK 中心。当前支持的编程语言有：

- Python
- Java

- PHP
- Go
- Node

为了更清楚的解释签名过程，下面以实际编程语言为例，将上述的签名过程具体实现。请求的域名、调用的接口和参数的取值都以上述签名过程为准，代码只为解释签名过程，并不具备通用性，实际开发请尽量使用 SDK 。

最终输出的 url 可能为： https://cvm.finance.cloud.tencent.com/?Action=DescribeInstances&InstanceIds.0=ins-09dx96dg&Limit=20&Nonce=11886&Offset=0&Region=shjr&SecretId=AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE&Signature=EliP9YW3pW28FpsEdkXt%2F%2BWcGeI%3D&Timestamp=1465185768&Version=2017-03-12

注意：由于示例中的密钥是虚构的，时间戳也不是系统当前时间，因此如果将此 url 在浏览器中打开或者用 curl 等命令调用时会返回鉴权错误：签名过期。为了得到一个可以正常返回的 url ，需要修改示例中的 SecretId 和 SecretKey 为真实的密钥，并使用系统当前时间戳作为 Timestamp 。

注意：在下面的示例中，不同编程语言，甚至同一语言每次执行得到的 url 可能都有所不同，表现为参数的顺序不同，但这并不影响正确性。只要所有参数都在，且签名计算正确即可。

注意：以下代码仅适用于 API 3.0，不能直接用于其他的签名流程，即使是旧版的 API ，由于存在细节差异也会导致签名计算错误，请以对应的实际文档为准。

Java

```
import java.io.UnsupportedEncodingException;
import java.net.URLEncoder;
import java.util.Random;
import java.util.TreeMap;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import javax.xml.bind.DatatypeConverter;

public class CloudAPIDemo {
    private final static String CHARSET = "UTF-8";

    public static String sign(String s, String key, String method) throws Exception {
        Mac mac = Mac.getInstance(method);
        SecretKeySpec secretKeySpec = new SecretKeySpec(key.getBytes(CHARSET), mac.getAlgorithm());
        mac.init(secretKeySpec);
        byte[] hash = mac.doFinal(s.getBytes(CHARSET));
        return DatatypeConverter.printBase64Binary(hash);
    }

    public static String getStringToSign(TreeMap<String, Object> params) {
        StringBuilder s2s = new StringBuilder("GETcvm.finance.cloud.tencent.com/?");
```

```
        // 签名时要求对参数进行字典排序，此处用TreeMap保证顺序
        for (String k : params.keySet()) {
            s2s.append(k).append("=").append(params.get(k).toString()).append("&");
        }
        return s2s.toString().substring(0, s2s.length() - 1);
    }

    public static String getUrl(TreeMap<String, Object> params) throws UnsupportedEncodingException
{
        StringBuilder url = new StringBuilder("https://cvm.finance.cloud.tencent.com/?");
        // 实际请求的url中对参数顺序没有要求
        for (String k : params.keySet()) {
            // 需要对请求串进行urlencode，由于key都是英文字母，故此处仅对其value进行urlencode
            url.append(k).append("=").append(URLEncoder.encode(params.get(k).toString(), CHARSET)).app
end("&");
        }
        return url.toString().substring(0, url.length() - 1);
    }

    public static void main(String[] args) throws Exception {
        TreeMap<String, Object> params = new TreeMap<String, Object>(); // TreeMap可以自动排序
        // 实际调用时应当使用随机数，例如：params.put("Nonce", new Random().nextInt(java.lang.Intege
r.MAX_VALUE));
        params.put("Nonce", 11886); // 公共参数
        // 实际调用时应当使用系统当前时间，例如：  params.put("Timestamp", System.currentTimeMillis() /
1000);
        params.put("Timestamp", 1465185768); // 公共参数
        params.put("SecretId", "AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE"); // 公共参数
        params.put("Action", "DescribeInstances"); // 公共参数
        params.put("Version", "2017-03-12"); // 公共参数
        params.put("Region", "shjr"); // 公共参数
        params.put("Limit", 20); // 业务参数
        params.put("Offset", 0); // 业务参数
        params.put("InstanceIds.0", "ins-09dx96dg"); // 业务参数
        params.put("Signature", sign(getStringToSign(params), "Gu5t9xGARNpq86cd98joQYCN3EXAMPLE
", "HmacSHA1")); // 公共参数
        System.out.println(getUrl(params));
    }
}
```

Python

注意：如果是在 Python 2 环境中运行，需要先安装 requests 依赖包： pip install requests 。

```
# -*- coding: utf8 -*-
import base64
```

```
import hashlib
import hmac
import time

import requests

secret_id = "AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE"
secret_key = "Gu5t9xGARNpq86cd98joQYCN3EXAMPLE"

def get_string_to_sign(method, endpoint, params):
    s = method + endpoint + "/?"
    query_str = "&".join("%s=%s" % (k, params[k]) for k in sorted(params))
    return s + query_str

def sign_str(key, s, method):
    hmac_str = hmac.new(key.encode("utf8"), s.encode("utf8"), method).digest()
    return base64.b64encode(hmac_str)

if __name__ == '__main__':
    endpoint = "cvm.finance.cloud.tencent.com"
    data = {
        'Action' : 'DescribeInstances',
        'InstanceIds.0' : 'ins-09dx96dg',
        'Limit' : 20,
        'Nonce' : 11886,
        'Offset' : 0,
        'Region' : 'shjr',
        'SecretId' : secret_id,
        'Timestamp' : 1465185768, # int(time.time())
        'Version': '2017-03-12'
    }
    s = get_string_to_sign("GET", endpoint, data)
    data["Signature"] = sign_str(secret_key, s, hashlib.sha1)
    print(data["Signature"])
    # 此处会实际调用，成功后可能产生计费
    # resp = requests.get("https://" + endpoint, params=data)
    # print(resp.url)
```

接口签名v3

TCloudFinanceZone API 会对每个访问请求进行身份验证，即每个请求都需要在公共请求参数中包含签名信息（Signature）以验证请求者身份。
签名信息由安全凭证生成，安全凭证包括 SecretId 和 SecretKey；若用户还没有安全凭证，请前往云API密钥页面申请，否则无法调用云API接口。

1. 申请安全凭证

在第一次使用云API之前，请前往云API密钥页面申请安全凭证。
安全凭证包括 SecretId 和 SecretKey：

- SecretId 用于标识 API 调用者身份
- SecretKey 用于加密签名字符串和服务器端验证签名字符串的密钥。
- 用户必须严格保管安全凭证，避免泄露。

申请安全凭证的具体步骤如下：

1. 登录TCloudFinanceZone管理中心控制台。
2. 前往云API密钥的控制台页面
3. 在云API密钥页面，点击【新建】即可以创建一对SecretId/SecretKey

注意：开发商帐号最多可以拥有两对 SecretId / SecretKey。

2. TC3-HMAC-SHA256 签名方法

注意：对于GET方法，只支持 Content-Type: application/x-www-form-urlencoded 协议格式。对于POST方法，目前支持 Content-Type: application/json 以及 Content-Type: multipart/form-data 两种协议格式，json 格式默认所有业务接口均支持，multipart 格式只有特定业务接口支持，此时该接口不能使用 json 格式调用，参考具体业务接口文档说明。

下面以云服务器查询广州区实例列表作为例子，分步骤介绍签名的计算过程。我们仅用到了查询实例列表的两个参数：Limit 和 Offset，使用 GET 方法调用。

假设用户的 SecretId 和 SecretKey 分别是：AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE 和 Gu5t9xGARNpq86cd98joQYCN3EXAMPLE

2.1. 拼接规范请求串

按如下格式拼接规范请求串（CanonicalRequest）：

```
CanonicalRequest =
    HTTPRequestMethod + '\n' +
    CanonicalURI + '\n' +
    CanonicalQueryString + '\n' +
    CanonicalHeaders + '\n' +
    SignedHeaders + '\n' +
    HashedRequestPayload
```

- HTTPRequestMethod：HTTP 请求方法（GET、POST ），本示例中为 GET；
- CanonicalURI：URI 参数，API 3.0 固定为正斜杠（/）；
- CanonicalQueryString：发起 HTTP 请求 URL 中的查询字符串，对于 POST 请求，固定为空字符串，对于 GET 请求，则为 URL 中问号（?）后面的字符串内容，本示例取值为：Limit=10&Offset=0。注意：CanonicalQueryString 需要经过 URL 编码。
- CanonicalHeaders：参与签名的头部信息，至少包含 host 和 content-type 两个头部，也可加入自定义的头部参与签名以提高自身请求的唯一性和安全性。拼接规则：1）头部 key 和 value 统一转成小写，并去掉首尾空格，按照 key:value\n 格式拼接；2）多个头部，按照头部 key（小写）的字典排序进行拼接。此例中为： content-type:application/x-www-form-urlencoded\nhost:cvm.finance.cloud.tencent.com\n
- SignedHeaders：参与签名的头部信息，说明此次请求有哪些头部参与了签名，和 CanonicalHeaders 包含的头部内容是一一对应的。content-type 和 host 为必选头部。拼接规则：1）头部 key 统一转成小写；2）多个头部 key（小写）按照字典排序进行拼接，并且以分号（;）分隔。此例中为： content-type;host
- HashedRequestPayload：请求正文的哈希值，计算方法为 Lowercase(HexEncode(Hash.SHA256(RequestPayload)))，对 HTTP 请求整个正文 payload 做 SHA256 哈希，然后十六进制编码，最后编码串转换成小写字母。注意：对于 GET 请求，RequestPayload 固定为空字符串，对于 POST 请求，RequestPayload 即为 HTTP 请求正文 payload。

根据以上规则，示例中得到的规范请求串如下（为了展示清晰，\n 换行符通过另起打印新的一行替代）：

```
GET
/
Limit=10&Offset=0
content-type:application/x-www-form-urlencoded
host:cvm.finance.cloud.tencent.com

content-type;host
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
```

2.2. 拼接待签名字符串

按如下格式拼接待签名字符串：

```
StringToSign =
    Algorithm + \n +
```

```
RequestTimestamp + \n +
CredentialScope + \n +
HashedCanonicalRequest
```

- Algorithm：签名算法，目前固定为 TC3-HMAC-SHA256；
- RequestTimestamp：请求时间戳，即请求头部的 X-TC-Timestamp 取值，如上示例请求为 1539084154；
- CredentialScope：凭证范围，格式为 Date/service/tc3_request，包含日期、所请求的服务和终止字符串（tc3_request）。Date 为 UTC 标准时间的日期，取值需要和公共参数 X-TC-Timestamp 换算的 UTC 标准时间日期一致；service 为产品名，必须与调用的产品域名一致，例如 cvm。如上示例请求，取值为 2018-10-09/cvm/tc3_request；
- HashedCanonicalRequest：前述步骤拼接所得规范请求串的哈希值，计算方法为 Lowercase(HexEncode(Hash.SHA256(CanonicalRequest)))。

注意：

> 1. Date 必须从时间戳 X-TC-Timestamp 计算得到，且时区为 UTC+0。如果加入系统本地时区信息，例如东八区，将导致白天和晚上调用成功，但是凌晨时调用必定失败。假设时间戳为 1551113065，在东八区的时间是 2019-02-26 00:44:25，但是计算得到的 Date 取 UTC+0 的日期应为 2019-02-25，而不是 2019-02-26。
> 2. Timestamp 必须是当前系统时间，且需确保系统时间和标准时间是同步的，如果相差超过五分钟则必定失败。如果长时间不和标准时间同步，可能导致运行一段时间后，请求必定失败（返回签名过期错误）。

根据以上规则，示例中得到的待签名字符串如下（为了展示清晰，\n 换行符通过另起打印新的一行替代）：

```
TC3-HMAC-SHA256
1539084154
2018-10-09/cvm/tc3_request
91c9c192c14460df6c1ffc69e34e6c5e90708de2a6d282cccf957dbf1aa7f3a7
```

2.3. 计算签名

1）计算派生签名密钥，伪代码如下

```
SecretKey = "Gu5t9xGARNpq86cd98joQYCN3EXAMPLE"
SecretDate = HMAC_SHA256("TC3" + SecretKey, Date)
SecretService = HMAC_SHA256(SecretDate, Service)
SecretSigning = HMAC_SHA256(SecretService, "tc3_request")
```

- SecretKey：原始的 SecretKey；
- Date：即 Credential 中的 Date 字段信息，如上示例，为2018-10-09；
- Service：即 Credential 中的 Service 字段信息，如上示例，为 cvm；

2）计算签名，伪代码如下

```
Signature = HexEncode(HMAC_SHA256(SecretSigning, StringToSign))
```

- SecretSigning：即以上计算得到的派生签名密钥；
- StringToSign：即步骤2计算得到的待签名字符串；

2.4. 拼接 Authorization

按如下格式拼接 Authorization：

```
Authorization =
    Algorithm + ' ' +
    'Credential=' + SecretId + '/' + CredentialScope + ', ' +
    'SignedHeaders=' + SignedHeaders + ', '
    'Signature=' + Signature
```

- Algorithm：签名方法，固定为 TC3-HMAC-SHA256；
- SecretId：密钥对中的 SecretId；
- CredentialScope：见上文，凭证范围；
- SignedHeaders：见上文，参与签名的头部信息；
- Signature：签名值

根据以上规则，示例中得到的值为：

```
TC3-HMAC-SHA256 Credential=AKIDEXAMPLE/Date/service/tc3_request, SignedHeaders=content-type;host, Signature=5da7a33f6993f0614b047e5df4582db9e9bf4672ba50567dba16c6ccf174c474
```

最终完整的调用信息如下：

```
https://cvm.finance.cloud.tencent.com/?Limit=10&Offset=0

Authorization: TC3-HMAC-SHA256 Credential=AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE/2018-10-09/cvm/tc3_request, SignedHeaders=content-type;host, Signature=5da7a33f6993f0614b047e5df4582db9e9bf4672ba50567dba16c6ccf174c474
Content-Type: application/x-www-form-urlencoded
Host: cvm.finance.cloud.tencent.com
X-TC-Action: DescribeInstances
X-TC-Version: 2017-03-12
X-TC-Timestamp: 1539084154
X-TC-Region: shjr
```

3. 签名失败

根据实际情况，存在以下签名失败的错误码，请根据实际情况处理

错误代码	错误描述
AuthFailure.SignatureExpire	签名过期
AuthFailure.SecretIdNotFound	密钥不存在
AuthFailure.SignatureFailure	签名错误
AuthFailure.TokenFailure	token 错误
AuthFailure.InvalidSecretId	密钥非法（不是云 API 密钥类型）

4. 签名演示

Java

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Map;
import java.util.TimeZone;
import java.util.TreeMap;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import javax.net.ssl.HttpsURLConnection;
import javax.xml.bind.DatatypeConverter;

import org.apache.commons.codec.digest.DigestUtils;

public class CloudAPITC3Demo {
    private final static String CHARSET = "UTF-8";
    private final static String ENDPOINT = "cvm.finance.cloud.tencent.com";
    private final static String PATH = "/";
    private final static String SECRET_ID = "AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE";
    private final static String SECRET_KEY = "Gu5t9xGARNpq86cd98joQYCN3EXAMPLE";
    private final static String CT_X_WWW_FORM_URLENCODED = "application/x-www-form-urlencode
d";
    private final static String CT_JSON = "application/json";
```

```
    private final static String CT_FORM_DATA = "multipart/form-data";

    public static byte[] sign256(byte[] key, String msg) throws Exception {
        Mac mac = Mac.getInstance("HmacSHA256");
        SecretKeySpec secretKeySpec = new SecretKeySpec(key, mac.getAlgorithm());
        mac.init(secretKeySpec);
        return mac.doFinal(msg.getBytes(CHARSET));
    }

    public static void main(String[] args) throws Exception {
        String service = "cvm";
        String host = "cvm.finance.cloud.tencent.com";
        String region = "shjr";
        String action = "DescribeInstances";
        String version = "2017-03-12";
        String algorithm = "TC3-HMAC-SHA256";
        String timestamp = "1539084154";
        //String timestamp = String.valueOf(System.currentTimeMillis() / 1000);
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        // 注意时区，否则容易出错
        sdf.setTimeZone(TimeZone.getTimeZone("UTC"));
        String date = sdf.format(new Date(Long.valueOf(timestamp + "000")));

        // ************* 步骤 1：拼接规范请求串 *************
        String httpRequestMethod = "GET";
        String canonicalUri = "/";
        String canonicalQueryString = "Limit=10&Offset=0";
        String canonicalHeaders = "content-type:application/x-www-form-urlencoded\n" + "host:" + host
+ "\n";
        String signedHeaders = "content-type;host";
        String hashedRequestPayload = DigestUtils.sha256Hex("");
        String canonicalRequest = httpRequestMethod + "\n" + canonicalUri + "\n" + canonicalQueryStri
ng + "\n"
                + canonicalHeaders + "\n" + signedHeaders + "\n" + hashedRequestPayload;
        System.out.println(canonicalRequest);

        // ************* 步骤 2：拼接待签名字符串 *************
        String credentialScope = date + "/" + service + "/" + "tc3_request";
        String hashedCanonicalRequest = DigestUtils.sha256Hex(canonicalRequest.getBytes(CHARSET));
        String stringToSign = algorithm + "\n" + timestamp + "\n" + credentialScope + "\n" + hashedCan
onicalRequest;
        System.out.println(stringToSign);

        // ************* 步骤 3：计算签名 *************
        byte[] secretDate = sign256(("TC3" + SECRET_KEY).getBytes(CHARSET), date);
        byte[] secretService = sign256(secretDate, service);
        byte[] secretSigning = sign256(secretService, "tc3_request");
```

```
        String signature = DatatypeConverter.printHexBinary(sign256(secretSigning, stringToSign)).toLow
erCase();
        System.out.println(signature);

        // ************* 步骤 4：拼接 Authorization *************
        String authorization = algorithm + " " + "Credential=" + SECRET_ID + "/" + credentialScope + ", "
                + "SignedHeaders=" + signedHeaders + ", " + "Signature=" + signature;
        System.out.println(authorization);

        TreeMap<String, String> headers = new TreeMap<String, String>();
        headers.put("Authorization", authorization);
        headers.put("Host", host);
        headers.put("Content-Type", CT_X_WWW_FORM_URLENCODED);
        headers.put("X-TC-Action", action);
        headers.put("X-TC-Timestamp", timestamp);
        headers.put("X-TC-Version", version);
        headers.put("X-TC-Region", region);
    }
}
```

Python

```
# -*- coding: utf-8 -*-
import hashlib, hmac, json, os, sys, time
from datetime import datetime

# 密钥参数
secret_id = "AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE"
secret_key = "Gu5t9xGARNpq86cd98joQYCN3EXAMPLE"

service = "cvm"
host = "cvm.finance.cloud.tencent.com"
endpoint = "https://" + host
region = "shjr"
action = "DescribeInstances"
version = "2017-03-12"
algorithm = "TC3-HMAC-SHA256"
timestamp = 1539084154
date = datetime.utcfromtimestamp(timestamp).strftime("%Y-%m-%d")
params = {"Limit": 10, "Offset": 0}

# ************* 步骤 1：拼接规范请求串 *************
http_request_method = "GET"
canonical_uri = "/"
canonical_querystring = "Limit=10&Offset=0"
ct = "x-www-form-urlencoded"
```

```
payload = ""
if http_request_method == "POST":
    canonical_querystring = ""
    ct = "json"
    payload = json.dumps(params)
canonical_headers = "content-type:application/%s\nhost:%s\n" % (ct, host)
signed_headers = "content-type;host"
hashed_request_payload = hashlib.sha256(payload.encode("utf-8")).hexdigest()
canonical_request = (http_request_method + "\n" +
                     canonical_uri + "\n" +
                     canonical_querystring + "\n" +
                     canonical_headers + "\n" +
                     signed_headers + "\n" +
                     hashed_request_payload)
print(canonical_request)

# ************* 步骤 2：拼接待签名字符串 *************
credential_scope = date + "/" + service + "/" + "tc3_request"
hashed_canonical_request = hashlib.sha256(canonical_request.encode("utf-8")).hexdigest()
string_to_sign = (algorithm + "\n" +
                  str(timestamp) + "\n" +
                  credential_scope + "\n" +
                  hashed_canonical_request)
print(string_to_sign)

# ************* 步骤 3：计算签名 *************
# 计算签名摘要函数
def sign(key, msg):
    return hmac.new(key, msg.encode("utf-8"), hashlib.sha256).digest()
secret_date = sign(("TC3" + secret_key).encode("utf-8"), date)
secret_service = sign(secret_date, service)
secret_signing = sign(secret_service, "tc3_request")
signature = hmac.new(secret_signing, string_to_sign.encode("utf-8"), hashlib.sha256).hexdigest()
print(signature)

# ************* 步骤 4：拼接 Authorization *************
authorization = (algorithm + " " +
                 "Credential=" + secret_id + "/" + credential_scope + ", " +
                 "SignedHeaders=" + signed_headers + ", " +
                 "Signature=" + signature)
print(authorization)

# 公共参数添加到请求头部
headers = {
    "Authorization": authorization,
    "Host": host,
    "Content-Type": "application/%s" % ct,
```

```
    "X-TC-Action": action,
    "X-TC-Timestamp": str(timestamp),
    "X-TC-Version": version,
    "X-TC-Region": region,
}
```

请求结构

1. 服务地址

地域（Region）是指物理的数据中心的地理区域。TCloudFinanceZone交付验证不同地域之间完全隔离，保证不同地域间最大程度的稳定性和容错性。为了降低访问时延、提高下载速度，建议您选择最靠近您客户的地域。

您可以通过 API接口 查询地域列表 查看完成的地域列表。

2. 通信协议

TCloudFinanceZone API 的所有接口均通过 HTTPS 进行通信，提供高安全性的通信通道。

3. 请求方法

支持的 HTTP 请求方法:

- POST（推荐）
- GET

POST 请求支持的 Content-Type 类型：

- application/json（推荐），必须使用 TC3-HMAC-SHA256 签名方法。
- application/x-www-form-urlencoded，必须使用 HmacSHA1 或 HmacSHA256 签名方法。
- multipart/form-data（仅部分接口支持），必须使用 TC3-HMAC-SHA256 签名方法。

GET 请求的请求包大小不得超过 32 KB。POST 请求使用签名方法为 HmacSHA1、HmacSHA256 时不得超过 1 MB。POST 请求使用签名方法为 TC3-HMAC-SHA256 时支持 10 MB。

4. 字符编码

均使用UTF-8编码。

返回结果

正确返回结果

以云服务器的接口查看实例状态列表 (DescribeInstancesStatus) 2017-03-12 版本为例，若调用成功，其可能的返回如下为：

```
{
    "Response": {
        "TotalCount": 0,
        "InstanceStatusSet": [],
        "RequestId": "b5b41468-520d-4192-b42f-595cc34b6c1c"
    }
}
```

- Response 及其内部的 RequestId 是固定的字段，无论请求成功与否，只要 API 处理了，则必定会返回。
- RequestId 用于一个 API 请求的唯一标识，如果 API 出现异常，可以联系我们，并提供该 ID 来解决问题。
- 除了固定的字段外，其余均为具体接口定义的字段，不同的接口所返回的字段参见接口文档中的定义。此例中的 TotalCount 和 InstanceStatusSet 均为 DescribeInstancesStatus 接口定义的字段，由于调用请求的用户暂时还没有云服务器实例，因此 TotalCount 在此情况下的返回值为 0， InstanceStatusSet 列表为空。

错误返回结果

若调用失败，其返回值示例如下为：

```
{
    "Response": {
        "Error": {
            "Code": "AuthFailure.SignatureFailure",
            "Message": "The provided credentials could not be validated. Please check your signature is cor
rect."
        },
        "RequestId": "ed93f3cb-f35e-473f-b9f3-0d451b8b79c6"
    }
}
```

- Error 的出现代表着该请求调用失败。Error 字段连同其内部的 Code 和 Message 字段在调用失败时是必定返回的。
- Code 表示具体出错的错误码，当请求出错时可以先根据该错误码在公共错误码和当前接口对应的错误码列表里面查找对应原因和解决方案。

- Message 显示出了这个错误发生的具体原因，随着业务发展或体验优化，此文本可能会经常保持变更或更新，用户不应依赖这个返回值。
- RequestId 用于一个 API 请求的唯一标识，如果 API 出现异常，可以联系我们，并提供该 ID 来解决问题。

公共错误码

返回结果中如果存在 Error 字段，则表示调用 API 接口失败。 Error 中的 Code 字段表示错误码，所有业务都可能出现的错误码为公共错误码，下表列出了公共错误码。

错误码	错误描述
AuthFailure.InvalidSecretId	密钥非法（不是云 API 密钥类型）。
AuthFailure.MFAFailure	MFA 错误。
AuthFailure.SecretIdNotFound	密钥不存在。
AuthFailure.SignatureExpire	签名过期。
AuthFailure.SignatureFailure	签名错误。
AuthFailure.TokenFailure	token 错误。
AuthFailure.UnauthorizedOperation	请求未 CAM 授权。
DryRunOperation	DryRun 操作，代表请求将会是成功的，只是多传了 DryRun 参数。
FailedOperation	操作失败。
InternalError	内部错误。
InvalidAction	接口不存在。
InvalidParameter	参数错误。
InvalidParameterValue	参数取值错误。
LimitExceeded	超过配额限制。
MissingParameter	缺少参数错误。
NoSuchVersion	接口版本不存在。
RequestLimitExceeded	请求的次数超过了频率限制。
ResourceInUse	资源被占用。
ResourceInsufficient	资源不足。

错误码	错误描述
ResourceNotFound	资源不存在。
ResourceUnavailable	资源不可用。
UnauthorizedOperation	未授权操作。
UnknownParameter	未知参数错误。
UnsupportedOperation	操作不支持。
UnsupportedProtocol	http(s)请求协议错误，只支持 GET 和 POST 请求。
UnsupportedRegion	接口不支持所传地域。

公共参数

公共参数是用于标识用户和接口鉴权目的的参数，如非必要，在每个接口单独的接口文档中不再对这些参数进行说明，但每次请求均需要携带这些参数，才能正常发起请求。

签名方法 v3

使用 TC3-HMAC-SHA256 签名方法时，公共参数需要统一放到 HTTP Header 请求头部中，如下：

参数名称	类型	必选	描述
X-TC-Action	String	是	操作的接口名称。取值参考接口文档中输入参数公共参数 Action 的说明。例如云服务器的查询实例列表接口，取值为 DescribeInstances。
X-TC-Region	String	是	地域参数，用来标识希望操作哪个地域的数据。接口接受的地域取值参考接口文档中输入参数公共参数 Region 的说明。注意：某些接口不需要传递该参数，接口文档中会对此特别说明，此时即使传递该参数也不会生效。
X-TC-Timestamp	Integer	是	当前 UNIX 时间戳，可记录发起 API 请求的时间。例如 1529223702。注意：如果与服务器时间相差超过5分钟，会引起签名过期错误。
X-TC-Version	String	是	操作的 API 的版本。取值参考接口文档中入参公共参数 Version 的说明。例如云服务器的版本 2017-03-12。
Authorization	String	是	HTTP 标准身份认证头部字段，例如： TC3-HMAC-SHA256 Credential=AKIDEXAMPLE/Date/service/tc3_request, SignedHeaders=content-type;host, Signature=fe5f80f77d5fa3beca038a248ff027d0445342fe2855ddc963176630326f1024 其中， - TC3-HMAC-SHA256：签名方法，目前固定取该值； - Credential：签名凭证，AKIDEXAMPLE 是 SecretId；Date 是 UTC 标准时间的日期，取值需要和公共参数 X-TC-Timestamp 换算的 UTC 标准时间日期一致；service为产品名，必须与调用的产品域名一致，例如cvm； - SignedHeaders：参与签名计算的头部信息，content-type 和 host 为必选头部； - Signature：签名摘要。
X-TC-Token	String	否	临时证书所用的 Token ，需要结合临时密钥一起使用。临时密钥和 Token 需要到访问管理服务调用接口获取。长期密钥不需要 Token。

签名方法 v1

使用 HmacSHA1 和 HmacSHA256 签名方法时，公共参数需要统一放到请求串中，如下

参数名称	类型	必选	描述
Action	String	是	操作的接口名称。取值参考接口文档中输入参数公共参数 Action 的说明。例如云服务器的查询实例列表接口，取值为 DescribeInstances。
Region	String	是	地域参数，用来标识希望操作哪个地域的数据。接口接受的地域取值参考接口文档中输入参数公共参数 Region 的说明。注意：某些接口不需要传递该参数，接口文档中会对此特别说明，此时即使传递该参数也不会生效。

参数名称	类型	必选	描述
Timestamp	Integer	是	当前 UNIX 时间戳，可记录发起 API 请求的时间。例如1529223702，如果与当前时间相差过大，会引起签名过期错误。
Nonce	Integer	是	随机正整数，与 Timestamp 联合起来，用于防止重放攻击。
SecretId	String	是	在云API密钥上申请的标识身份的 SecretId，一个 SecretId 对应唯一的 SecretKey ，而 SecretKey 会用来生成请求签名 Signature。
Signature	String	是	请求签名，用来验证此次请求的合法性，需要用户根据实际的输入参数计算得出。具体计算方法参见接口鉴权文档。
Version	String	是	操作的 API 的版本。取值参考接口文档中入参公共参数 Version 的说明。例如云服务器的版本 2017-03-12。
SignatureMethod	String	否	签名方式，目前支持 HmacSHA256 和 HmacSHA1。只有指定此参数为 HmacSHA256 时，才使用 HmacSHA256 算法验证签名，其他情况均使用 HmacSHA1 验证签名。
Token	String	否	临时证书所用的 Token ，需要结合临时密钥一起使用。临时密钥和 Token 需要到访问管理服务调用接口获取。长期密钥不需要 Token 。

地域列表

地域（Region）是指物理的数据中心的地理区域。TCloudFinanceZone交付验证不同地域之间完全隔离，保证不同地域间最大程度的稳定性和容错性。为了降低访问时延、提高下载速度，建议您选择最靠近您客户的地域。

您可以通过 API接口 查询地域列表 查看完成的地域列表。

TBase部署中心
获取回档时间

1. 接口描述

接口请求域名： tbase.api3.finance.cloud.tencent.com。

数据恢复时，获取回档时间

默认接口请求频率限制：20次/秒。

接口更新时间：2025-06-13 06:59:28。

接口只验签名不鉴权。

2. 输入参数

以下请求参数列表仅列出了接口请求参数和部分公共参数，完整公共参数列表见公共请求参数。

参数名称	必选	允许NULL	类型	描述
Action	是	否	String	公共参数，本接口取值：DescribeAvailableRecoveryTime
Version	是	否	String	公共参数，本接口取值：2019-01-07
Region	是	否	String	公共参数，地域信息可通过DescribeRegions接口查看产品支持的地域列表
InstanceId	是	否	String	实例ID 示例值： "tdpg-ls6hihx7"
RecoveryExpiredBackup	否	否	Bool	是否从过期备份回档，默认为false 示例值： false

3. 输出参数

参数名称	类型	描述
StartTime	String	回档允许的最早时间 示例值： "2025-04-08 10:49:30"

参数名称	类型	描述
EndTime	String	回档允许的最晚时间 示例值： "2025-04-08 11:21:03"
RecoveryTaskId	Int64	本次回档的任务id 示例值： 0
RecoveryStatus	String	当前该实例的回档状态，success成功，init初始化中，running回档中，running状态下不允许再次发起回档请求 示例值： "init"
ExpiredBackupTimeRange	ExpiredBackupTimeRange	过期备份可回档时间范围，入参RecoveryExpiredBackup为true时返回 示例值：查看
RequestId	String	唯一请求 ID，每次请求都会返回。定位问题时需要提供该次请求的 RequestId。

4. 错误码

以下仅列出了接口业务逻辑相关的错误码，其他错误码详见公共错误码。

错误码	描述
UnauthorizedOperation.PermissionDenied	cam鉴权无权限。
InvalidParameterValue.InstanceNotFound	实例查询失败。
InvalidParameter.CheckParamNotPass	参数检查未通过。
InvalidParameter	参数错误

查询售卖地域信息

1. 接口描述

接口请求域名： tbase.api3.finance.cloud.tencent.com。

本接口(DescribeAvailableRegion)用于查询售卖地域信息

默认接口请求频率限制：20次/秒。

接口更新时间：2024-12-11 18:33:33。

接口只验签名不鉴权。

2. 输入参数

以下请求参数列表仅列出了接口请求参数和部分公共参数，完整公共参数列表见公共请求参数。

参数名称	必选	允许NULL	类型	描述
Action	是	否	String	公共参数，本接口取值：DescribeAvailableRegion
Version	是	否	String	公共参数，本接口取值：2019-01-07
Region	是	否	String	公共参数，地域信息可通过DescribeRegions接口查看产品支持的地域列表
EngineType	是	否	Array of String	引擎类型，TbaseV2/TbaseV3/TbaseV5 示例值： ["TbaseV2","TbaseV3"]

3. 输出参数

参数名称	类型	描述
Items	Array of RegionSaleInfo	可售卖地域详情 示例值：查看
RequestId	String	唯一请求 ID，每次请求都会返回。定位问题时需要提供该次请求的 RequestId。

4. 错误码

以下仅列出了接口业务逻辑相关的错误码，其他错误码详见公共错误码。

错误码	描述
FailedOperation.DescribeAvailableRegion	查询售卖地域失败。
InternalError.SystemError	异常错误。
MissingParameter.InvalidParameterValue	缺少参数。
UnauthorizedOperation.PermissionDenied	cam鉴权无权限。

数据结构

NodeBackupDetail

实例节点备份详情

被如下接口引用：DescribeBackupDetails

名称	必选	允许NULL	类型	描述
NodeName	是	否	String	节点名。 示例值： "cn0001"
NodeType	是	否	String	节点类型。 示例值： "coordinator"
LatestStatus	是	否	Int64	备份状态。1(备份中)，2(备份成功)，-1(备份失败) 示例值： 2
LatestStartTime	是	否	String	最近备份开始时间。 示例值： "2025-03-28 00:10:41"
LatestEndTime	是	否	String	最近备份结束时间。 示例值： "2025-03-28 00:11:23"

AccountInfo

账号信息。

被如下接口引用：DescribeAccounts

名称	必选	允许NULL	类型	描述
InstanceId	是	否	String	实例Id。 示例值： tdpg-ls6hihx7
Username	是	否	String	账号名。 示例值： dbadmin
Attributes	否	否	Array of String	账号属性。 示例值： ["Create role", "Create DB"]

RegionSellConf

描述单个地域的可选规格

被如下接口引用：DescribeAvailableZoneConfig

名称	必选	允许NULL	类型	描述
Region	是	否	String	地域英文名 示例值： region1
RegionName	是	否	String	地域名称 示例值： region name
Area	是	否	String	所属大区 示例值： source area
IsDefaultRegion	是	否	Bool	是否为默认地域 示例值： true
AvailableZones	是	否	Array of ZoneSellConf	可用区售卖配置 示例值：查看
IsSellOut	否	否	Bool	是否售罄 示例值： false

ParamDetail

实例参数详细信息。

被如下接口引用：DescribeDBParameters

名称	必选	允许NULL	类型	描述
ParameterName	是	否	String	参数名。 示例值： max_connections
DefaultValue	是	否	String	参数默认值。 示例值： 3000
NeedRestart	是	否	Bool	修改此参数是否需要重启。 示例值： true
RunningValue	是	否	String	当前运行值。 示例值： 2000
ValueRange	是	否	ParamConstraint	参数取值范围。 示例值：查看
Unit	是	否	String	参数单位。 示例值： ""
ShortDesc	是	否	String	参数英文简介。 示例值： Sets the maximum number of concurrent connections.

PGInstanceSet

Postgres实例信息

被如下接口引用：DescribePGInstances

名称	必选	允许NULL	类型	描述
InstanceId	是	否	String	实例Id 示例值： "postgres-eh4m6wfw"
InstanceName	是	否	String	实例名 示例值： "postgres-eh4m6wfw"
Region	是	否	String	地域信息 示例值： "region1"
RegionName	是	否	String	地域名称 示例值： "region1"
Zone	是	否	String	可用区信息 示例值： "zone1"
ZoneName	是	否	String	可用区名称 示例值： "zone1"
VpcId	是	否	String	私有网络Id 示例值： "vpc-299sadwp"
VpcName	是	否	String	私有网络名称 示例值： "tbase-test"
SubnetId	是	否	String	私有网络子网ID 示例值： "subnet-fzig2yxq"
SubnetName	是	否	String	私有网络子网名称 示例值： "tbase-subnet-az1"
CreatedAt	是	否	String	创建时间 示例值： "2024-10-24 10:10:08"
Status	是	否	String	实例状态 示例值： "online"
StatusDesc	是	否	String	实例状态描述 示例值： "运行中"
TaskType	是	否	String	任务类型 示例值： ""
TaskDesc	是	否	String	任务描述信息 示例值： ""
Vip	是	否	String	实例ip 示例值： "10.0.0.10"
Vport	是	否	Int64	实例端口 示例值： 5432

名称	必选	允许NULL	类型	描述
Cpu	是	否	Int64	cpu核数 示例值： 1
Memory	是	否	Int64	内存大小 示例值： 1
Storage	是	否	Int64	磁盘大小 示例值： 10
NodeCount	是	否	Int64	节点数量 示例值： 2
TradeType	是	否	String	付费类型，(后付费-postpaid，预付费-prepaid) 示例值： "postpaid"
AdminUserName	是	否	String	管理员账号名 示例值： "dbadmin"
RoVip	是	否	String	只读vip 示例值： ""
RoVport	是	否	Int64	只读port 示例值： 0
ProjectId	否	否	String	项目ID，为空表示没有所属项目 示例值： ""
ProjectName	否	否	String	项目名，为空表示没有所属项目 示例值： ""
IsolateOperator	否	否	String	实例隔离者 示例值： ""
IsArrears	否	否	Int64	是否欠费，1 - 欠费， 0 - 不欠费 示例值： 0
PoolAlias	否	否	String	资源池别名 示例值： "DefaultPool"
CpuArch	否	否	String	cpu架构 示例值： "x86_64"
PoolId	否	否	Int64	资源池id 示例值： 1
EngineVersion	否	否	String	数据库内核版本 示例值： 5.11.15.06110
EngineType	否	否	String	实例类型，引擎类型，当前只支持PostgreSQL、TbaseXC 示例值： TbaseXC
DestroyTime	否	否	String	销毁时间 示例值： "2024-10-24 10:10:08"

名称	必选	允许NULL	类型	描述
DisplayVersion	否	否	String	前端展示的数据库版本 示例值： 15.6

ZoneSellConf

描述单个可用区的可选规格

被如下接口引用：DescribeAvailableZoneConfig

名称	必选	允许NULL	类型	描述
Zone	是	否	String	可用区英文名 示例值： zone4
ZoneName	是	否	String	可用区名称 示例值： zone4
IsDefaultZone	是	否	Bool	是否为当前地域的默认可用区 示例值： fasle
AvailableSpec	是	否	Array of SellSpec	可选节点规格 示例值：查看
IsSellOut	否	否	Bool	是否售罄 示例值： false

PGSellSpec

描述Postgres售卖单条规格信息

被如下接口引用：DescribePGAvailableZoneConfig

名称	必选	允许NULL	类型	描述
SpecCode	是	否	String	规格码，唯一标识一个规格，创建实例时传该字段用于标识一个规格 示例值： pg.dn.480g
NodeType	是	否	String	节点类型: pg_dn（postgres实例存储节点） 示例值： pg_dn
Cpu	是	否	Int64	cpu核心数，单位：核 示例值： 1
Memory	是	否	Int64	内存大小，单位：GB 示例值： 1
MinStorage	是	否	Int64	磁盘最小规格，单位：GB 示例值： 10

名称	必选	允许NULL	类型	描述
MaxStorage	是	否	Int64	磁盘最大规格，单位：GB 示例值： 6000
StorageStep	是	否	Int64	磁盘调整步长，单位：GB 示例值： 10
NodeCount	是	否	Int64	磁盘调整步长，单位：GB 示例值： 1

SellSpec

描述单条规格信息

被如下接口引用：DescribeAvailableZoneConfig

名称	必选	允许NULL	类型	描述
SpecCode	是	否	String	规格码，唯一标识一个规格，创建实例时传该字段用于标识一个规格 示例值： tbase.cn.1g
NodeType	是	否	String	节点类型: cn（计算节点），dn（存储节点） 示例值： cn
Cpu	是	否	Int64	cpu核心数，单位：核 示例值： 1
Memory	是	否	Int64	内存大小，单位：GB 示例值： 1
MinStorage	是	否	Int64	磁盘最小规格，单位：GB 示例值： 10
MaxStorage	是	否	Int64	磁盘最大规格，单位：GB 示例值： 6000
StorageStep	是	否	Int64	磁盘调整步长，单位：GB 示例值： 10
NodeCount	是	否	Int64	磁盘调整步长，单位：GB 示例值： 1

BaseBackup

数据备份

被如下接口引用：DescribeBaseBackups、DescribeMetaBackups

名称	必选	允许NULL	类型	描述

名称	必选	允许NULL	类型	描述
InstanceId	是	否	String	实例ID 示例值： "tdpg-xxxx"
InstanceName	是	否	String	实例名称 示例值： "张三的实例"
BackupId	是	否	Uint64	备份任务ID 示例值： 3
FileDir	是	否	String	备份路径 示例值： "backup/tdpg-7a2l33zq/bkp_cluster_2022_08_22"
FileSize	是	否	String	备份文件大小 示例值： "235.99 GB"
BackupStrategy	是	否	String	备份类型， Manual/Auto 示例值： Manual
BackupType	是	否	String	备份方式, physical/logical/meta 示例值： physical
BackupPattern	是	否	String	备份方法, full/increment 示例值： full
StartTime	是	否	String	备份开始时间 示例值： "2022-08-22 18:51:13.532994"
EndTime	是	否	String	备份结束时间 示例值： "2022-08-22 18:51:13.532994"

ErrorLogDetail

错误日志详细信息。

被如下接口引用：DescribeErrorLog、DescribePGErrorLog

名称	必选	允许NULL	类型	描述
UserName	是	否	String	用户名。 示例值： dbadmin
Database	是	否	String	数据库名。 示例值： postgres
ErrorTime	是	否	String	错误日志发生时间。 示例值： 2025-04-08 11:33:49
ErrorMessage	是	否	String	错误详细信息。 示例值： connect failed

SlowLogDetail

慢查询日志详细信息。

被如下接口引用：DescribeSlowLog

名称	必选	允许NULL	类型	描述
TotalTime	是	否	Float	花费总时间。 示例值： 10
TotalCalls	是	否	Int64	调用总次数。 示例值： 1500
NormalQuerys	是	否	Array of NormQueryItem	慢查询sql详细信息。 示例值：查看

RecoveryInfo

创建回档实例的必传参数

被如下接口引用：CreateInstanceHour

名称	必选	允许NULL	类型	描述
RecoveryTimePoint	是	否	String	回档时间点，会在获取回档时间接口中返回 示例值： 2021-11-03 10:12:55
RecoveryTime	是	否	String	回档时间，会在获取回档时间接口中返回，需选择回档时间范围中的某一个时刻点 示例值： 2021-11-03 10:11:55
RecoveryInstanceId	是	否	String	要回档的实例ID 示例值： tdpg-tabskgw
RecoveryExpiredBackup	否	否	Bool	是否从过期备份回档，默认为false 示例值： false

VpcInfo

节点vpc信息

被如下接口引用：DescribePGInstanceDetails

名称	必选	允许NULL	类型	描述
VpcId	是	否	String	私有网络Id 示例值： vpc-3i5ribn3
SubnetId	是	否	String	私有网络子网Id 示例值： subnet-mmyjn1f6

名称	必选	允许NULL	类型	描述
VpcName	是	否	String	私有网络名称 示例值： test-vpc
SubnetName	是	否	String	私有网络子网名称 示例值： test-subnet
Vip	是	否	String	vip 示例值： 10.0.0.2
Vport	是	否	Int64	vport 示例值： 5432

Product

约束项用量

被如下接口引用：DescribeLicenseUsage

名称	必选	允许NULL	类型	描述
ProductCode	否	否	String	产品代码 示例值： tbase
ProductName	否	否	String	产品名称 示例值： TDSQL for PostgreSQL
LicenseItemList	否	否	Array of LicenseItem	云产品License用量列表 示例值：查看

TagInfo

Tag信息，包含TagKey和TagValue信息

被如下接口引用：CreateInstanceHour、CreatePGInstanceHour

名称	必选	允许NULL	类型	描述
TagKey	是	否	String	TagKey值 示例值： tagKey
TagValue	否	是	String	TagValue值 示例值： tagValue

Instance

描述实例的信息

被如下接口引用：DescribeInstances

名称	必选	允许NULL	类型	描述
InstanceId	是	否	String	实例ID 示例值： "tdpg-0go24zsp"
InstanceName	是	否	String	实例名称 示例值： "tdpg-0go24zsp"
Region	是	否	String	地域ID 示例值： "region1"
RegionName	是	否	String	地域名称 示例值： "region1"
Zone	是	否	String	可用区ID 示例值： "zone1"
ZoneName	是	否	String	可用区名称 示例值： "zone1"
VpcId	是	否	String	私有网络Id 示例值： "vpc-299sadwp"
VpcName	是	否	String	私有网络名称 示例值： "tbase-test"
SubnetId	是	否	String	私有网络子网ID 示例值： "subnet-fzig2yxq"
SubnetName	是	否	String	私有网络子网名称 示例值： "tbase-subnet-az1"
CreateTime	是	否	Datetime	创建时间 示例值： 2025-04-08 11:33:49
Status	是	否	String	状态 示例值： "online"
StatusDesc	是	否	String	状态描述 示例值： "运行中"
TaskType	是	否	String	任务类型 示例值： ""
TaskDesc	是	否	String	任务描述 示例值： ""
Vip	是	否	String	虚拟IP 示例值： "10.0.0.8"
Vport	是	否	Int64	虚拟端口 示例值： 5432

名称	必选	允许NULL	类型	描述
TradeType	是	否	String	付费类型，postpaid - 后付费，prepaid - 预付费 示例值： "postpaid"
CNNodes	是	否	Array of InstanceNode	计算节点列表 示例值：查看
DNNodes	是	否	Array of InstanceNode	数据节点列表 示例值：查看
AdminUserName	是	否	String	管理员账号名 示例值： "dbadmin"
ProjectId	否	否	String	项目ID，为空表示没有所属项目 示例值： ""
ProjectName	否	否	String	项目名，为空表示没有所属项目 示例值： ""
IsolateOperator	否	否	String	实例隔离者 示例值： user
PeriodEndTime	否	否	String	到期时间 示例值： 2025-05-08 11:33:49
Area	否	否	String	所属大区 示例值： area1
DestroyTime	否	否	String	销毁时间 示例值： 2025-04-08 11:33:49
IsArrears	否	否	Int64	是否欠费，1 - 欠费， 0 - 不欠费 示例值： 0
DisplayVersion	否	否	String	前端展示的引擎版本 示例值： 2.15.19.14
EngineType	否	否	String	实例类型，引擎类型，当前只支持TbaseV2、TbaseV5、TbaseV5_C、TbaseH、TbaseHC 示例值： TbaseH
GTMNodes	否	否	Array of InstanceNode	全局事务节点列表 示例值：查看
CpuArch	否	否	String	cpu架构 示例值： x86_64
Version	否	否	String	数据库版本 示例值： 2.15.19.14
AutoRenewFlag	否	否	Int64	自动续费标记，仅针对预付费实例，后付费实例不涉及 示例值： 0
AppId	否	否	Int64	用户的唯一ID 示例值： 1255000463

名称	必选	允许NULL	类型	描述
PoolAlias	否	否	String	资源池别名 示例值： DefaultPool
PoolId	否	否	Int64	资源池id 示例值： 1
Product	否	否	String	产品名 示例值： tbase
RegionId	否	否	Int64	地域ID 示例值： 50000001
ClusterName	否	否	String	集群名称 示例值： "tdpg-0go24zsp"

Filter

描述键值对过滤器，用于条件过滤查询。例如过滤ID、名称、状态等

- 若存在多个Filter时，Filter间的关系为逻辑与（AND）关系。
- 若同一个Filter存在多个Values，同一Filter下Values间的关系为逻辑或（OR）关系。

被如下接口引用：DescribeInstances、DescribePGInstances

名称	必选	允许NULL	类型	描述
Name	是	否	String	需要过滤的字段。 示例值： "product"
Values	是	否	Array of String	字段的过滤值。 示例值： ["tdpg-ls6hihx7"]

TimePoint

实例回档时可选择的时间

被如下接口引用：DescribeAvailableRecoveryTimePoint

名称	必选	允许NULL	类型	描述
Time	是	否	String	实例回档时可选择的时间 示例值： 2021-11-03 10:12:55

Node

节点信息

被如下接口引用：DescribePGInstanceDetails

名称	必选	允许NULL	类型	描述
Role	是	否	String	节点角色 示例值： "master"
Region	是	否	String	地域信息 示例值： "region1"
Zone	是	否	String	可用区信息 示例值： "zone1"
ZoneName	是	否	String	可用区名 示例值： "zone1"
Type	是	否	String	节点所在节点组中的节点类型 示例值： "master"
NodeName	是	否	String	节点名 示例值： "pg0000"

PGRegionSellConf

描述单个地域的可选规格

被如下接口引用：DescribePGAvailableZoneConfig

名称	必选	允许NULL	类型	描述
Region	是	否	String	地域英文名 示例值： region1
RegionName	是	否	String	地域名称 示例值： region1
Area	是	否	String	所属大区 示例值： "area1"
IsDefaultRegion	是	否	Bool	是否为默认地域 示例值： false
AvailableZones	是	否	Array of PGZoneSellConf	可用区售卖配置 示例值：查看

ConstraintRange

实例参数取值范围，标识最大最小值。

被如下接口引用：DescribeDBParameters

名称	必选	允许NULL	类型	描述
Min	是	否	String	区间的最小值。 示例值： 0
Max	是	否	String	区间的最大值。 示例值： 2147483647

Backup

实例备份信息

被如下接口引用：DescribeBackupLists

名称	必选	允许NULL	类型	描述
BackupUid	是	否	Int64	实例备份信息的唯一标识，查询备份详情时的入参。 示例值： 38271
StartTime	是	否	String	实例备份开始时间。 示例值： "2025-03-31 00:08:41"
EndTime	是	否	String	实例备份结束时间。 示例值： "2025-03-31 00:15:51"
StrategyType	是	否	String	备份策略类型 Auto(自动备份) 示例值： "Auto"
BackupType	是	否	String	备份类型 Physical(物理备份) 示例值： "Physical"
Status	是	否	Int64	备份状态。1(备份中)，2（备份成功），-1（备份失败） 示例值： -1

ExpiredBackupTimeRange

过期备份可回档时间范围。

被如下接口引用：DescribeAvailableRecoveryTime

名称	必选	允许NULL	类型	描述
StartTime	是	否	String	过期备份回档的最早时间 示例值： "2025-04-08 10:49:30"
EndTime	是	否	String	过期备份回档的最晚时间 示例值： "2025-04-15 11:21:03"

InstanceNode

描述实例的节点信息

被如下接口引用：DescribeInstanceDetails、DescribeInstances

名称	必选	允许NULL	类型	描述
NodeType	是	否	String	节点类型 示例值： "dn"
Cpu	是	否	Int64	节点cpu 示例值： 1
Memory	是	否	Int64	节点内存 示例值： 1
Storage	是	否	Int64	节点数据盘大小 示例值： 10
NodeId	是	否	String	节点唯一ID 示例值： "21"
UsedStorage	是	否	Int64	已用数据存储空间 示例值： 9
NodeName	是	否	String	节点名 示例值： "dn0001"
Type	是	否	String	所在节点组中的节点类型，master-主节点 示例值： "master"
Zone	否	是	String	可用区 示例值： "zone1"
ZoneName	否	是	String	可用区名称 示例值： "zone1"
SyncType	否	否	String	节点的复制类型，sync:同步，async:异步，sync2async:同步可退化成异步 示例值： sync2async
SpecCode	否	否	String	节点code码 示例值： dn.4.billing

NormQueryItem

慢查询详细信息

被如下接口引用：DescribeSlowLog

名称	必选	允许NULL	类型	描述
DatabaseName	是	否	String	数据库名。 示例值： postgres

名称	必选	允许NULL	类型	描述
NormalQuery	是	否	String	脱敏后的sql语句。 示例值： sellect * from test
MaxElapsedQuery	是	否	String	执行时间最长的语句。 示例值： sellect count(*) from test
Calls	是	否	Int64	调用次数。 示例值： 1000
CostTime	是	否	Float	花费总时间。 示例值： 1
ClientIp	是	否	String	客户端ip。 示例值： 127.0.0.1
UserName	是	否	String	用户名。 示例值： postgres
TotalCallsPercent	是	否	Float	在总的慢查询中的访问次数占比。 示例值： 21.2
TotalCostTimePercent	是	否	Float	在总的慢查询中的访问时间占比。 示例值： 0.12
MinCostTime	是	否	Float	花费的最小时间。 示例值： 1
MaxCostTime	是	否	Float	花费的最长时间。 示例值： 1
FirstTime	是	否	String	最早一条慢查询的sql时间。 示例值： 2020-02-02 16:02:02
LastTime	是	否	String	最近一条慢查询的sql时间。 示例值： 2020-02-02 17:02:02
SharedReadBlks	是	否	Int64	读共享内存块数。 示例值： 1024
SharedWriteBlks	是	否	Int64	写共享内存块数。 示例值： 1024
ReadCostTime	是	否	Float	读io总耗时。 示例值： 1
WriteCostTime	是	否	Float	写io总耗时。 示例值： 1

EngineVersion

描述可用的引擎版本

被如下接口引用：DescribeAvailableEngineVersions

名称	必选	允许NULL	类型	描述
Type	是	否	String	引擎类型，例如 TBaseV2,TBaseV3 示例值： "TbaseXC"
Version	是	否	String	引擎版本，例如：1.0.1 示例值： "5.11.15.06110"
DisplayVersion	否	否	String	前端展示的引擎版本 示例值： "15.6"

RecoveryNodeSpec

按恢复时间选择可恢复的实例规格

被如下接口引用：DescribeAvailableRecoveryNodeSpec

名称	必选	允许NULL	类型	描述
NodeType	否	否	Uint64	节点类型：0-GTM，1-CN，2-DN，3-FN 示例值： 1
NodeSet	否	否	Uint64	节点组数 示例值： 2
NodeCount	否	否	Uint64	节点个数 示例值： 4
Cpu	否	否	Uint64	CPU，单位：核 示例值： 1
Memory	否	否	Uint64	内存大小，单位：GB 示例值： 4
Storage	否	否	Uint64	磁盘大小，单位：GB 示例值： 100

RegionSaleInfo

售卖地域信息

被如下接口引用：DescribeAvailableRegion

名称	必选	允许NULL	类型	描述
TotalCount	是	否	Uint64	可售卖地域总数 示例值： 1

名称	必选	允许NULL	类型	描述
RegionSaleDetail	是	否	Array of RegionSaleDetail	可售卖地域详情 示例值：查看
EngineType	是	否	String	可售卖地域引擎类型TbaseV2/TbaseV3/TbaseV5 示例值： TbaseV2

PGZoneSellConf

描述Postgres实例单个可用区的可选规格

被如下接口引用：DescribePGAvailableZoneConfig

名称	必选	允许NULL	类型	描述
Zone	是	否	String	可用区英文名 示例值： zone1
ZoneName	是	否	String	可用区名称 示例值： zone1
IsDefaultZone	是	否	Bool	是否为当前地域的默认可用区 示例值： true
AvailableSpec	是	否	Array of PGSellSpec	可选节点规格 示例值：查看

ConfigParam

实例参数配置。

被如下接口引用：ModifyDBParameters

名称	必选	允许NULL	类型	描述
ParameterName	是	否	String	参数名。 示例值： lock_timeout
ParameterValue	是	否	String	参数值。 示例值： 30000

SyncTypeInfo

修改节点的复制类型。

被如下接口引用：ModifyInstanceSyncType

名称	必选	允许NULL	类型	描述
SourceSyncType	否	否	String	节点的原复制类型，可选项(sync:同步不可退化，async:异步，sync2async:同步可退化) 示例值： "sync2async"
SlaveNodeId	是	否	String	备节点的节点id。 示例值： "136"
TargetSyncType	是	否	String	节点的目标复制类型，可选项(sync:同步不可退化，async:异步，sync2async:同步可退化) 示例值： "sync"
MasterNodeId	是	否	String	主节点的节点id。 示例值： "135"

ParamConstraint

实例参数的类型取值范围。

被如下接口引用：DescribeDBParameters

名称	必选	允许NULL	类型	描述
Type	是	否	String	用来描述取值范围类型。section: 数值范围, enum: 枚举, string: 字符串。 示例值： section
Range	是	否	ConstraintRange	type取值为section时的数值范围。 示例值：查看
Enum	是	否	Array of String	type取值为enum时的枚举值。 示例值： ["utf8"]
String	是	否	String	type取值为string时的数值。 示例值： ""

PgNodeInfo

节点规格信息

被如下接口引用：CreatePGInstanceHour

名称	必选	允许NULL	类型	描述
SpecCode	是	否	String	节点规格代码 示例值： "pg.dn.1g"
Storage	是	否	Int64	磁盘单个节点大小 示例值： 10

ResourcePool

资源池信息

被如下接口引用：DescribeResourcePools

名称	必选	允许NULL	类型	描述
PoolName	否	否	String	资源池名 示例值： default_pool
PoolId	否	否	Int64	资源池id 示例值： 1
PoolDesc	否	否	String	资源池描述信息 示例值： default system defined pool
PoolAlias	否	否	String	资源池别名 示例值： DefaultPool
CpuArch	否	否	String	cpu架构 示例值： x86_64

ParamItem

各节点对应的参数。

被如下接口引用：DescribeDBParameters

名称	必选	允许NULL	类型	描述
NodeType	是	否	String	节点类型。 示例值： cn
NodeName	是	否	String	节点名。 示例值： cn0001
Details	是	否	Array of ParamDetail	节点的各个参数信息。 示例值：查看
TotalCount	是	否	Int64	参数个数。 示例值： 5

NodeInfo

创建tbase实例时的节点相关信息

被如下接口引用：CreateInstanceHour

名称	必选	允许NULL	类型	描述

名称	必选	允许NULL	类型	描述
NodeCount	是	否	Int64	要创建实例的某类型节点的总个数（节点组数与每组节点个数的乘积）。 示例值： 4
SetCount	是	否	Int64	要创建实例的某类型节点的组数。 示例值： 2
Storage	是	否	Int64	要创建实例的某类型节点的单节点磁盘大小。 示例值： 10
SpecCode	是	否	String	要创建实例的某类型节点的规格码，可以通过DescribeAvailableZoneConfig接口获得。 示例值： "dn.v5.1g"
SyncType	否	是	String	设置节点的复制类型，目前只有dn设置，可选参数为sync（同步），async（异步），sync2async（异步可退化），默认值为sync2async 示例值： "sync2async"

WALBackup

增量备份

被如下接口引用：DescribeWALBackups

名称	必选	允许NULL	类型	描述
InstanceId	是	否	String	实例 ID 示例值： tdpg-mdi2stas
InstanceName	是	否	String	实例名称 示例值： 啊啊啊
FileName	是	否	String	备份文件路径 示例值： backup/tdpg-ei1cg10y/xlog_bkp_2022_09_01
StartTime	是	否	String	备份开始时间 示例值： 2022-09-01 00:00:41.345604
EndTime	是	否	String	备份结束时间 示例值： 2022-09-01 00:11:43.317891
FileSize	是	否	String	备份文件大小 示例值： 125.04 GB
NodeName	是	否	String	节点名称 示例值： dn0001

ConfigHistory

参数修改历史记录

被如下接口引用：DescribeDBConfigHistory

名称	必选	允许NULL	类型	描述
Id	是	否	Int64	参数修改历史记录唯一id。 示例值： 60
InstanceId	是	否	String	实例id。 示例值： "tdpg-0go24zsp"
CreatedAt	是	否	String	参数修改的时间。 示例值： "2025-03-31T11:53:17+08:00"
UpdatedAt	是	否	String	参数修改的结束时间。 示例值： "2025-03-31T11:53:25+08:00"
NodeType	是	否	String	节点类型。 示例值： "dn"
ParamName	是	否	String	参数名。 示例值： "work_mem"
ParamNewValue	是	否	String	修改后的参数值。 示例值： "65536"
ParamOldValue	是	否	String	修改前的参数值。 示例值： "65535"
Status	是	否	String	参数修改状态：success, failed, doing 示例值： "success"

RegionSaleDetail

售卖地域详细信息

被如下接口引用：DescribeAvailableRegion

名称	必选	允许NULL	类型	描述
Region	否	否	String	地域英文名 示例值： region1
RegionName	否	否	String	地域名称 示例值： region name
Area	否	否	String	地域所属大区 示例值： source area
IsSellOut	否	否	Bool	是否售罄 示例值： true

NodeConfigParams

节点参数配置。

被如下接口引用：ModifyDBParameters

名称	必选	允许NULL	类型	描述
NodeType	是	否	String	节点类型，目前支持：cn、dn。 示例值： dn
ConfigParams	是	否	Array of ConfigParam	需要修改的参数配置。 示例值：查看

LicenseItem

云产品License用量

被如下接口引用：DescribeLicenseUsage

名称	必选	允许NULL	类型	描述
Usage	否	否	Float	当前DN节点数量 示例值： 4.0
RuleKey	否	否	String	分配的计费中心校验项 key 示例值： sv_tce_db_sw_tbase_lic

错误码

功能说明

如果返回结果中存在 Error 字段，则表示调用 API 接口失败。例如：

```
{
    "Response": {
        "Error": {
            "Code": "AuthFailure.SignatureFailure",
            "Message": "The provided credentials could not be validated. Please check your signature is cor
rect."
        },
        "RequestId": "ed93f3cb-f35e-473f-b9f3-0d451b8b79c6"
    }
}
```

Error 中的 Code 表示错误码，Message 表示该错误的具体信息。

错误码列表

公共错误码

错误码	说明
AuthFailure.InvalidSecretId	密钥非法（不是云 API 密钥类型）。
AuthFailure.MFAFailure	MFA 错误。
AuthFailure.SecretIdNotFound	密钥不存在。 请在控制台检查密钥是否已被删除或者禁用，如状态正常，请检查密钥是否填写正确，注意前后不得有空格。
AuthFailure.SignatureExpire	签名过期。Timestamp 和服务器时间相差不得超过五分钟，请检查本地时间是否和标准时间同步。
AuthFailure.SignatureFailure	签名错误。 签名计算错误，请对照调用方式中的接口鉴权文档检查签名计算过程。
AuthFailure.TokenFailure	token 错误。
AuthFailure.UnauthorizedOperation	请求未 CAM 授权。
DryRunOperation	DryRun 操作，代表请求将会是成功的，只是多传了 DryRun 参数。

错误码	说明
FailedOperation	操作失败。
InternalError	内部错误。
InvalidAction	接口不存在。
InvalidParameter	参数错误。
InvalidParameterValue	参数取值错误。
LimitExceeded	超过配额限制。
MissingParameter	缺少参数错误。
NoSuchVersion	接口版本不存在。
RequestLimitExceeded	请求的次数超过了频率限制。
ResourceInUse	资源被占用。
ResourceInsufficient	资源不足。
ResourceNotFound	资源不存在。
ResourceUnavailable	资源不可用。
UnauthorizedOperation	未授权操作。
UnknownParameter	未知参数错误。
UnsupportedOperation	操作不支持。
UnsupportedProtocol	http(s)请求协议错误，只支持 GET 和 POST 请求。
UnsupportedRegion	接口不支持所传地域。

业务错误码

错误码	说明
FailedOperation.QueryConfigHistoryFailed	查询参数修改历史记录失败。
InvalidParameterValue.InvalidInstanceId	无效的实例id。
MissingParameter.InvalidParameterValue	缺少参数。
FailedOperation.IsolateInstanceFailed	实例隔离失败。

错误码	说明
FailedOperation.QueryInstanceInfoFailed	查询实例相信信息失败。
InvalidParameterValue.VpcIpInvalid	vip状态：该vip不合规。
FailedOperation.DescribePublicKeyFailed	获取公钥失败
FailedOperation.QuerySupportedEngineFailed	查询引擎版本失败。
InvalidParameter.PasswordStrengthNotEnough	密码不符合规范。
FailedOperation.QueryServerFailed	查询服务器信息失败。
FailedOperation.QueryErrLogFailed	查询错误日志错误。
InternalError.SystemError	异常错误。
FailedOperation.ModifyDBParametersFailed	修改实例参数配置失败。
InvalidParameter.PermissionDenied	无权限操作。
ResourceUnavailable.UnexpectedInstanceStatus	实例状态有误。
InvalidParameterValue.CheckVipStatusFail	检查vip状态失败。
FailedOperation.OssCreatedResource	不能销毁oss创建的实例。
FailedOperation.AssociateSecurityGroupsFailed	绑定安全组失败。
InvalidParameterValue.VpcIpNotInSubnet	vip状态：本子网网段下不包含该vip。
FailedOperation.QueryInstanceFailed	查询实例信息失败。
InvalidParameterValue.GetSubnetInfoFail	获取vpc子网信息失败。
InvalidParameter.CheckParamNotPass	参数检查未通过。
FailedOperation.ModifyInstanceSyncTypeFailed	修改实例节点复制类型失败。
FailedOperation.CreateFlowFailed	创建异步流程失败。
FailedOperation.PostpaidCreateInstanceHourFailed	后付费实例创建失败。
FailedOperation.GetInstanceNetworkFailed	获取实例网络信息失败。
InvalidParameter	参数错误
InvalidParameterValue.SpecNotMatch	规格信息不匹配。
InvalidParameterValue.TypeNotSupport	不支持的类型。

错误码	说明
FailedOperation.ListAccountsFailed	查询账号列表失败。
MissingParameter.ValueNotFound	参数不合法。
FailedOperation.ActiveIsolateInstanceFailed	实例解隔离失败。
FailedOperation.QueryAvailableZoneConfigFailed	查询地域规格信息失败。
FailedOperation.DescribeAvailableRegion	查询售卖地域失败。
InvalidParameterValue.InstanceNotFound	实例查询失败。
FailedOperation.QueryDBNameListFailed	查询实例db名称失败。
FailedOperation.OperateDatabaseFailed	操作数据库时报错。
InvalidParameterValue.ZoneNotFound	查询可用区信息失败。
FailedOperation.QueryInstanceNodesFailed	查询实例节点信息失败。
FailedOperation.GetBackupRulesFailed	查询备份规则失败。
FailedOperation.ImportMetaBackupFailed	导入元数据备份失败。
FailedOperation.ListBackupsFailed	查询备份列表失败。
FailedOperation.GetBackupDetailsFailed	查询备份详情失败。
InvalidParameterValue.VpcCodeUnknown	vip状态：未知错误。
FailedOperation.QuerySlowLogFailed	查询慢查询日志失败。
FailedOperation.SetBackupRulesFailed	设置备份规则失败。
FailedOperation.QueryCharsetFailed	查询字符集失败。
UnauthorizedOperation.PermissionDenied	cam鉴权无权限。
InvalidParameterValue.VpcIpUsed	vip状态：已被使用。
InvalidParameterValue.RegionNotFound	查询地域信息失败。
FailedOperation.ActivateIsolateInstanceFailed	实例重新上线失败。
FailedOperation.QueryVpcAndSubnetFailed	vpc信息查询失败。
FailedOperation.DisassociateSecurityGroupsFailed	解绑安全组失败。