TDSQL-A PostgreSQL 版

产品文档

文档目录

产品简介
产品概述

最近更新时间: 2024-10-17 17:10:00

TDSQL-A PostgreSQL版 是腾讯自主研发的无共享架构的分布式分析型数据库系统，支持 SQL2011 标准，全面兼容 PostgreSQL 语法，高度兼容 Oracle 语法。自研列式存储引擎，支持行存储和列存储，支持混合存储，支持高压缩比。新一代向量化执行引擎能提供高性能海量数据实时高效复杂查询分析能力。

同时，支持完整的分布式事务处理，支持多级容灾以及多维度资源隔离，还提供强大的多级安全体系，提供弹性扩缩容能力，提供完善的企业级管理能力，为用户提供容灾、备份、恢复、监控、安全、审计等全套解决方案，适用于 GB级 - PB级 的海量联机分析处理（OLAP）场景，是具有市场竞争力的企业级数仓产品。

产品特点

行列混合存储

为更好地提供 OLAP 能力，TDSQL-A PostgreSQL版 在兼容 PostgreSQL 生态的行式存储基础上，还自研了列式存储引擎，提供完整的列存储能力，业务可以根据需要对写入数据库中的数据选择对应存储格式，提供高效的行列混合查询能力。

列存储支持强大的压缩能力，包括透明压缩和轻量级压缩，透明压缩支持 zlib，zstd 等压缩算法，轻量级压缩算法支持 delta，rle，bitpack 算法，可根据数据的特征自动调整优化算法进行高效压缩，最高压缩比达400:1。

高效复杂查询

TDSQL-A PostgreSQL版 自研新一代向量化执行引擎，对于复杂查询有高效的处理能力，能实现万亿数据关联分析秒级响应，性能相比开源和传统的数据仓库提升数倍至数百倍；具备强大的 OLAP 分析能力。

业务平滑迁移

支持 SQL2011 语法规范，语法完整兼容 PostgreSQL，高度兼容 Oracle 语法，且配备有腾讯 DBbridge 迁移工具，支持业务系统尽可能平滑地迁移到 TDSQL-A PostgreSQL版。

企业级数据安全

支持安全管理员、审计管理员、数据管理员三权分立体系，提供数据存储加密、数据脱敏访问、强制访问控制、数据审计等多个层级的策略保障数据安全。

完整分布式事务

支持完整的事务 ACID 能力，并且支持全局事务一致性；通过全局事务管理节点来管理分布式事务，通过拥有自主专利的分布式事务一致性技术，来保证数据在分布式架构下的一致性和高效性。

丰富的生态支持

TDSQL-A PostgreSQL版 具有丰富的周边生态：

- 支持强大的地理信息系统（GIS）。通过集群化的 PostGis 插件，支持存储空间地理数据，使 TDSQL-A PostgreSQL版 成为一个空间数据库，能够通过 SQL 语言高效的进行空间数据管理、数量测量和几何拓扑分析。
- TDSQL-A PostgreSQL版 不仅是一个分布式关系型数据库系统，同时还支持非关系数据类型 JSON。
- 支持 Foreign Data Wrappers（FDW）功能，该功能实现了部分的 SQL/MED 规定，允许用户使用普通 SQL 查询来访问位于 PostgreSQL 之外的数据。

  FDW 功能提供一套编程接口，用户可进行插件式的二次开发，建立外部数据源和数据库间的数据通道。大多数情况下用户可用 oracle_fdw、mysql_fdw、postgres_fdw，非关系型数据库的 redis_fdw、mongodb_fdw，以及大数据的 hive_fdw、hdfs_fdw 等。基于 FDW 功能和已有插件，TDSQL-A PostgreSQL版 提供强大的数据库联邦能力，通过 TDSQL-A PostgreSQL版 能够访问已有的多个数据源的数据。
- 支持通过数据迁移和同步服务及工具，便捷地将不同部署方式的源端数据同步至 TDSQL-A PostgreSQL版，包括腾讯云、自建或其他云数据库。同步功能稳定，性能优异，让您获得一体化数据体验。

产品功能

最近更新时间: 2024-10-17 17:10:00

支持列式存储和多种压缩算法

TDSQL-A PostgreSQL版 支持列式存储，客户可以根据自己的业务需求把表定义为列存表，一般建议对于大宽表及有高压缩需求的表可以设置为列存表。

列存表支持多种压缩算法，包括 delta，zlib，zstd，rle，bitpack 压缩算法，不同压缩算法支持不同的压缩级别，详见 开发指南，TDSQL-A PostgreSQL版 支持新一代列存存储向量化执行引擎，对行列混合存储和查询能提供很高效的查询性能。

高效分布式 JOIN 计算

业务分析场景，通常会有2个或多个表关联（JOIN）的逻辑，此逻辑在单机模式中是一个简单的操作，但在集群模式下，由于数据分布在1个或多个物理节点中，处理会相对复杂。在很多分布式解决方案中，JOIN 会把数据拉取到一个节点，进行关联计算，不仅耗费了大量网络资源，且语句的执行耗时会很高。

TDSQL-A PostgreSQL版 通过如下方式对分布式 JOIN 进行高效计算，基于高效的全局查询计划和数据重分布的技术支撑，TDSQL-A PostgreSQL版 能很好地发挥并行计算的优势，高效完成 JOIN 过程。

- 在执行方式上，协调节点接收到用户的 SQL 请求，根据收集的集群统计信息，生成最优的集群级分布式查询计划，并下发到参与计算的数据节点上进行执行，即协调节点下发的是执行计划，数据节点负责执行该计划。
- 在数据交互上，数据节点之间建立了高效数据交换通道，可以高效的交换数据，数据交换的过程在 TDSQL-A PostgreSQL版 里称之数据重分布（Data Redistribution）。

多核并行计算

TDSQL-A PostgreSQL版 在节点内部采用了并行计算，同时启动多个进程来协同完成一个查询，可充分利用服务器的多核处理能力来快速、高效地完成查询。通常情况下，TDSQL-A PostgreSQL版 会启动多个进程来完成查询，查询时间会大大缩短，如果有更多的资源可供使用，查询时间则会呈线性优化。

TDSQL-A PostgreSQL版 会根据查询表大小来决定是否进行并行查询，表的数据量超过阈值后，才会采用并行计算，当需要并行计算时，会根据表大小得出并行度，即需要的进程个数。

数据安全保障功能

数据加密

TDSQL-A PostgreSQL版 提供两种数据加密方式：

- 业务侧加密：业务调用 TDSQL-A PostgreSQL版 内置的加密函数，将加密结果写入数据库，正常读取的也是加密后的数据，然后在应用里执行解密。
- TDSQL-A PostgreSQL版 内置加密：加密过程对业务侧透明，优点如下：
  - 加密操作（函数调用）与业务侧解耦合，业务只负责写入原始数据到数据库内核，后续的加密计算在数据库内部完成，从而业务侧操作上无感知。
  - 加密算法由数据库维护，包括加密算法的选择、密钥管理，都由安全员独立操作完成。

    内核加密计算支持异步加密，保证系统在吞吐不变的情况下，达成数据加密。支持的加密算法有 AES128、AES192、AES256、国密SM4。

数据脱敏

TDSQL-A PostgreSQL版 支持透明数据脱敏功能，在用户无感知的情况下，对非授权用户返回被脱敏的数据。

从以上两个维度实现更细粒度的数据访问控制，增强对现有访问的控制，且对现有业务系统无感知。

全方位审计

TDSQL-A PostgreSQL版 从多个维度提供全方位的审计能力，审计采用旁路检测方式，对数据库运行影响极小。审计类型如下：

- 语句审计：针对某一种特定的语句进行审计。

- 对象审计：针对某个数据库对象的操作进行审计。
- 用户审计：针对某个数据库用户的操作进行审计。
- 细粒度审计（Fine-Grained Audit，FGA）：高级审计选项，使用表达式来作为审计条件，可设置审计被触发时的动作，例如，发邮件打电话等。

冷热数据分离

内核原生支持数据的冷热分离，业务无需感知底层存储介质的不同，对外提供统一的数据库视图。

- 冷热数据使用不同的节点 group 存储，节点组内部使用的物理机型配置不同，从而达到冷热分离节省成本的目的。
- 后台定时任务根据用户配置的冷热数据规则，自动进行数据迁移，系统即可实现自动的冷热分离，业务无需关心集群的冷热数据存储情况。

  此功能目前在私有云版本已有，在公有云目前还未提供。

多级容灾功能

TDSQL-A PostgreSQL版 在多个维度保证集群的容灾能力：

强同步复制

TDSQL-A PostgreSQL版 支持强同步复制，在节点级保证每个节点的主从数据完全一致，是整个容灾体系的基础，当主节点（Master）故障发生时，数据库可切换到从节点（Slave）提供服务且无任何数据丢失。强同步机制要求用户请求发生，从节点写入日志成功后，才给用户返回成功，保证主从节点的数据时刻一致。

主从高可用

TDSQL-A PostgreSQL版 主从高可用方案主要通过每组节点的多副本冗余来实现服务不中断或中断时间很短，当一组节点的主节点出现故障不可恢复，将自动从对应的备节点中选出新的主节点工作。在主从高可用基础上 TDSQL-A PostgreSQL版 支持：

- 故障自动转移：集群中主节点故障时，系统自动从对应的从节点中选出新的主节点，故障节点自动被集群隔离，基于强同步复制策略，主从切换保证主从数据完全一致，可满足金融级数据一致性要求。
- 故障恢复：备节点因磁盘故障导致数据丢失时，数据库管理员（DBA）可以通过重做备机来恢复备机，可选择在新的物理节点上添加备机来恢复主从备份关系，保证系统可靠性。
- 副本切换：每组主从节点（可以是1主 N 从）的每个节点都包含完整的数据副本，DBA 可根据需求进行切换。
- 设置禁止切换：即可设置在某一特殊时期，不处理故障转移。
- 跨可用区部署：主节点和从节点分处于不同机房，数据之间通过专线网络进行实时的数据复制。本地为主节点，远程为从节点，首先访问本地节点，若本地主节点发生故障或访问不可达，则远程的从节点升为主节点提供服务。
- TDSQL-A PostgreSQL版 支持基于强同步的高可用方案，主节点故障时将自动选出最优从节点立即顶替工作，切换过程对用户透明，且不改变访问 IP。TDSQL-A PostgreSQL版 对系统组件支持7 * 24小时持续监控，发生故障时，TDSQL-A PostgreSQL版 将自动重启节点或者隔离节点，从从节点选出新主节点提供服务。

支持全量增量备份

TDSQL-A PostgreSQL版 支持基于备份在事务一致性的时间点恢复数据，防止误操作带来的数据丢失。备份分为全量备份（冷备）和增量备份（xlog 备份）：

- 全量备份：指备份数据库的全部数据（除了运行日志和 xlog 之外），全量备份通常是周期性，如一天、一周或 N 天。
- 增量备份：指增量数据的备份，一般通过 xlog 文件实现，当数据库系统产生新的 xlog 文件后，系统将 xlog 文件备份到备份服务器上，增量备份通常是实时的行为。

  当发生事故或灾难后，用户可以利用备份数据来恢复系统。

产品优势

最近更新时间: 2024-10-17 17:10:00

分布式事务全局一致性

TDSQL-A PostgreSQL版 引入全局事务管理节点（Global Transaction Manager，GTM）来专门处理分布式事务一致性，通过即两阶段提交（Two Phase Commit）和全局事务管理策略来保证在全分布式环境下的事务一致性。同时 TDSQL-A PostgreSQL版 提供了分布式事务可靠性保证机制来避免资源阻塞、数据不一致和协调节点宕机等问题。

SQL 高兼容度

TDSQL-A PostgreSQL版 兼容 SQL2011 规范， 在 SQL 兼容性上具备很大优势，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，可满足大部分企业用户的需求。同时 TDSQL-A PostgreSQL版 还兼容大部分的 Oracle 数据类型、函数，此特性可方便 Oracle 数仓业务迁移到 TDSQL-A PostgreSQL版 数据库。

行列混合存储

TDSQL-A PostgreSQL版 在支持兼容 PostgreSQL 生态的行式存储基础上，还自研了列式存储，提供完整的列存储能力。业务可以根据需要对写入数据库中的数据选择对应存储格式。

TDSQL-A PostgreSQL版 列存储支持强大的压缩能力，包括透明压缩和轻量级压缩，透明压缩支持 zlib，zstd 等压缩算法，轻量级压缩算法支持 delta，rle，bitpack 算法，可根据数据的特征进行高效压缩，压缩比高达400+。

高效复杂查询

TDSQL-A PostgreSQL版 自研新一代向量化执行引擎，对于复杂查询有高效的处理能力，能实现万亿数据关联分析秒级响应，性能相比开源和传统的数据仓库提升数倍至数百倍；具备强大的 OLAP 分析能力。

多级安全策略

传统数据库系统的超级用户权限极大，不容易受到制约，也不利于数据库安全体系的建立，TDSQL-A PostgreSQL版 的三权分立体系，将传统数据库管理员 DBA 的角色分解为安全管理员、审计管理员、数据管理员三个相互独立的角色。安全管理员可以针对业务需求，配置数据加密规则对数据进行加密，保证数据不被泄露。支持数据透明加密，数据脱敏等安全特性。

丰富的周边生态

全面拥抱 PostgreSQL 生态，持续跟进社区发展。支持丰富的生态工具，包括 PostGis 组件，支持非结构化数据类型 JSON；支持 FDW 外表能力和其他数据源进行互通。

支持通过数据迁移服务或产品将其他数据源的数据同步到 TDSQL-A PostgreSQL版。

产品架构

最近更新时间: 2024-10-17 17:10:00

TDSQL-A PostgreSQL版 采用分布式无共享（share nothing）架构，节点之间相应独立，各自处理自己的数据，处理后的结果可能向上层汇总或在节点间流转，各处理单元之间通过网络协议进行通信，并行处理和扩展能力更好，这也意味着只需要简单的 x86 或 arm 服务器就可以部署 TDSQL-A PostgreSQL版 数据库集群。架构图如下：

各个模块说明如下：

- Coordinator：协调节点（简称 CN），对外提供接口，负责数据的分发和查询规划，多个节点位置对等，每个节点都提供相同的数据库视图；在功能上 CN 只存储系统的全局元数据，并不存储实际的业务数据。
- Datanode：数据节点（简称 DN），处理存储本节点相关的元数据，每个节点还存储业务数据的分片。在功能上，DN 负责完成执行协调节点分发的执行请求。
- GTM：全局事务管理器（Global Transaction Manager，GTM），负责管理集群事务信息，同时管理集群的全局对象，如序列等。
- Data Forward Bus：集群数据交互总线，集群交互总线由各个服务器上的 FN（Forward Node）节点组成，加入 FN 的主要目的在于减少 DN 之间、CN 和 DN 之间数据交换时创建的连接，从而保证大规模集群下连接不是瓶颈。

在此架构下，集群具有下面几个能力：

- 多活/多主：每个 CN 提供相同的集群视图，可以从任何一个 CN 进行写入，业务无需感知集群拓扑。
- 读/写扩展：数据被分片存储在了不同的 DN，集群的读/写能力，随着集群规模的扩大而得到提升。
- 集群写一致：业务在一个 CN 节点发生的写事务会一致性的呈现在其他 CN 节点，就像这些事务是本 CN 节点发生的一样。
- 集群结构透明：数据位于不同的数据库节点中，当查询数据时，不必关心数据位于具体的节点。

TDSQL-A PostgreSQL版 的 share nothing 集群架构方便业务接入，降低了业务接入的门槛。

应用场景

最近更新时间: 2024-10-17 17:10:00

数据仓库

TDSQL-A PostgreSQL版 借助 share nothing 架构，可在线线性平滑地扩展集群规模，具备 GB级 - PB级 数据支撑能力，全并行架构和向量化执行引擎可以高效处理百亿行多表连接查询，适用于操作数据存储 ODS（Operational Data Store）、企业数据仓库 EDW（Enterprise Data Warehouse）、数据集市 DM（Data Mart）等。

海量存储在线实时分析

互联网化的用户激增，伴随着系统的长期运行，数据累积越来越多，给部分行业（如支付业务，因为监管要求，数据必须永久保存）带来的存储成本，以及大数据量场景的复杂关联查询性能问题等亟待解决。

TDSQL-A PostgreSQL版 的在线线性扩容能力，能够按需扩充集群，保证集群可以支撑到 PB 级别的存储，同时结合业务历史数据不常被访问的特点，可将历史数据自动转移到低廉的存储设备上，兼顾性能和成本。

数据高安全依赖型系统

在民生、金融等行业里，存储了非常多的个人基本信息和金融交易数据，保障数据的安全性是首要考虑的问题，一旦发生数据丢失或者泄露，会造成不可估量的损失，因此该类业务对于存储核心数据的数据库系统安全非常依赖，包括数据查询结果加密，数据存储加密，以及事后审计需求。

TDSQL-A PostgreSQL版 能够提供多级安全策略来保障该类高安全依赖型系统的数据安全。

多点汇聚业务系统

银行、大型国企的组织架构通常采用总部-分部-分支的架构，其某些核心 IT 系统建设也采用总部-分部-分支模式，且各个分支采用的数据库不同，随着业务互通、人员互通、信息互通等需求越来越强烈，业务逐渐向总部聚合，因此能否高效的进行数据汇聚，是系统一个很重要的考量指标。

TDSQL-A PostgreSQL版 具备高效的异构数据库复制能力，让数据能够很好的在多个数据库中实现共享。

去 O 场景

近些年来各行各业的 IT 系统兴起了去 IOE 的浪潮，其中去 O 是相对较难的一项。

TDSQL-A PostgreSQL版 作为高扩展性的数据库集群，同时兼容 PostgreSQL 和大部分 Oracle 语法，另外 TDSQL-A PostgreSQL版 也支持存储过程、窗口函数、非结构化数据等诸多企业级特性，使得 TDSQL-A PostgreSQL版 成为去 O 的极佳选择。在替换 Oracle 数仓应用的场景下，使用 TDSQL-A PostgreSQL版 可以很好地减少迁移成本。

产品规格

最近更新时间: 2024-10-17 17:10:00

TDSQL-A PostgreSQL版 目前测试实例规格固定。规格如下：

节点	规格	主从配置
GTM	1 Core vCPU，4GB内存，20GB磁盘	1组，1主1从
CN	1 Core vCPU，4GB内存，20GB磁盘	2组，每组1主1从
DN	1 Core vCPU，8GB内存，300GB磁盘	4组，每组1主1从
FN	1 Core vCPU，4GB内存，20GB磁盘	每个实例所在机器会有一个

快速入门
创建实例

最近更新时间: 2024-10-17 17:10:00

本文为您介绍通过 TDSQL-A PostgreSQL版 控制台创建实例的操作。

操作步骤

1. 登录 TDSQL-A PostgreSQL版 控制台，在实例列表，单击新建。

2. 在购买页根据需求，选择地域、网络、字符集等，并单击立即购买。
   - 计费模式：目前支持包年包月。
   - 地域可用区：建议您选择与云服务器同一个地域，处在不同地域的云产品网络无法互通。
   - 网络类型：私有网络（默认选项）、选择 VPC 及子网。
     - 如果当前需要新建私有网络，可在控制台新建 私有网络、新建子网。
     - VPC 网络选择后不可更改，VPC 相关操作请参见 管理私有网络。
   - 安全组：默认为空。如果业务需要放通其他端口，请 自定义安全组。
   - 标签：标签用于从不同维度对资源分类管理。
   - 选择字符集：支持 UTF8、LATIN1 、EUC_CN、SQL_ASCII 字符集。
   - 数据复制方式：针对 DN 节点的数据同步方式设置，默认强同步（可退化）。
     - 当设置为强同步（可退化）时，DN 备机故障，系统自动修改数据方式为异步，业务不阻塞。
     - 当设置为强同步时，DN 备机故障，业务将被阻塞。
     - 当设置为异步时，DN 备机故障时，业务无影响。

3. 提交开通后，返回实例列表，待实例状态变为运行中，即可进行连接实例操作。

连接实例

最近更新时间: 2024-10-17 17:10:00

本文为您介绍通过云服务器连接 TDSQL-A PostgreSQL版 实例的操作。云服务器和数据库须是同一账号，且同一个 VPC 内（保障同一个地域）

操作步骤

本文以腾讯云服务器中 CentOS 7.2 64 位系统为例，云服务器购买请参见 购买方式。

1. 登录 Linux 云服务器，执行如下命令下载 PostgreSQL。

```
yum install -y http://imgcache.finance.cloud.tencent.com:80download.postgresql.org/pub/repos/yum/reporpms/EL-7-x86_64/pgdg-redhat-repo-latest.noarch.rpm
```

说明：

请根据实际操作系统，替换对应的 PostgreSQL 下载地址。

2. 执行如下命令安装 PostgreSQL。

```
yum install -y postgresql10-server postgresql10
```

3. 执行如下命令连接到 TDSQL-A PostgreSQL版。

```
psql -h 实例地址 -p 端口 -U dbadmin -d postgres
```

说明：

实例地址和端口可在 TDSQL-A 控制台 的实例详情页查看。

操作指南
查看实例详情

最近更新时间: 2024-10-17 17:10:00

本文为您介绍如何通过控制台查看 TDSQL-A PostgreSQL版 实例详情。

操作步骤

1. 登录 TDSQL-A PostgreSQL版 控制台，在实例列表，单击实例 ID 或操作列的管理，进入实例详情页。

2. 在实例详情页，可查看实例的基本信息、配置信息。

销毁实例

最近更新时间: 2024-10-17 17:10:00

本文为您介绍如何通过控制台销毁 TDSQL-A PostgreSQL版 实例。

操作场景

根据业务需求，您可以在控制台自助销毁实例。自助销毁后，实例的状态一旦变为已隔离，隔离7天后彻底销毁，隔离中实例无法恢复。

> **注意：**
>
> 实例销毁后数据将无法找回，备份文件会同步销毁，无法在云上进行数据恢复。
>
> 实例销毁后 IP 资源同时释放。

操作步骤

1. 登录 TDSQL-A PostgreSQL版 控制台，在实例列表选择所需实例，在操作列选择更多 > 销毁。

2. 在弹出的对话框，阅读并勾选已阅读并同意销毁规则后，单击确定。

3. 确定销毁实例后，实例状态变更为已隔离。

4. 在实例列表选择所需实例，在操作列选择更多 > 立即下线。

> **注意：**
>
> 执行下线操作后，实例会彻底销毁，数据将无法找回，请提前备份实例数据。

5. 在弹出的对话框，确认无误后，单击确定。

6. 实例下线成功后，实例列表右上角会弹出下线成功提示框。

备份数据库

最近更新时间: 2024-10-17 17:10:00

为防止数据丢失或损坏，您可以使用自动备份的方式来备份数据库。

自动备份

1. 登录 TDSQL-A 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，选择备份 > 自动备份设置页，单击编辑。

3. 在编辑页面，根据备份开始时间的提示，输入目标值，单击确定。

> **说明：**
>
> 目前只可修改备份开始时间。
>
> 自动备份无法手动删除，备份保留时间到期后会自动删除。

设置项	值
数据备份保留	7天
备份时间间隔	每24小时备份一次
备份开始时间	00:00:00-02:00:00
是否开启日志备份	否

监控功能

最近更新时间: 2024-10-17 17:10:00

本文为您介绍如何通过控制台查看导出 TDSQL-A PostgreSQL版 监控信息。

查看监控

1. 登录 TDSQL-A 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，选择系统管理页，选择时间，可以查看监控数据信息及负载。

- 在系统管理页面，单击重启节点，选择节点信息，可以选择重启 GTM、CN、DN 等节点。

> 注意：
>
> 节点重启期间实例将无法提供正常服务，本操作为高危操作，请您提前做好准备，以免造成影响。

- 查看告警及监控信息，根据时间查看监控信息粒度，单击近24小时可查看24小时监控数据。

- 单击右侧的导出数据，选择需要导出的信息，可导出监控数据。

监控指标

腾讯云云监控从实例维度为 TDSQL-A PostgreSQL版 实例提供以下监控指标：

指标中文名	指标英文名	单位	指标说明
CPU 利用率	cpu_used_pct	%	实例的 CN、DN、GTM 的 CPU 使用率的最大值
内存利用率	mem_used_pct	%	实例的 CN、DN、GTM 的内存使用率的最大值
IO 吞吐量	iops	次/s	实例的 CN/DN 主备，磁盘吞吐率
缓存命中率	cache_hit_pct	%	数据缓存命中率
连接数	connections	个	实例的活跃连接
最小 TOP10 执行耗时	sql_runtime_min	ms	执行时间最短的 TOP10 的 SQL 的平均值
最大 TOP10 执行耗时	sql_runtime_max	ms	执行时间最长的 TOP10 的 SQL 的平均值
平均 SQL 执行耗时	sql_runtime_avg	ms	所有 SQL 请求的平均执行时间，不包含事务里面的 SQL
总请求数	total_requests	次	所有 CN、DN 主备节点的请求之和，按分钟累加
业务请求数	user_requests	次	所有 CN、DN 主备节点的业务请求之和（已去除系统请求），按分钟累加
读请求数	read_requests	次	读请求每分钟总数
更新请求数	update_requests	次	更新请求每分钟总数
插入请求数	insert_requests	次	插入请求每分钟总数

指标中文名	指标英文名	单位	指标说明
删除请求数	delete_requests	次	删除请求每分钟总数
写请求数	write_requests	次	写请求每分钟总数
其他请求数	other_requests	次	除了读和写以外的请求总数，按分钟累加
错误请求数	error_requests	次	实例上记录的所有错误的请求数之和，按分钟累加
残留两阶段事务数目	two_phase_commit_trxs	个	实例中所有 CN、DN 的主备节点中 prepared 阶段在10分钟之前的事务数之和
容量使用率	capacity_used_pct	%	实例的容量使用率
容量已使用	capacity_usage	GBytes	实例的已使用容量
剩余 XID 数量	xid_remain	Count	实例的所有 CN、DN 上剩余 XID 的最小值

操作日志

最近更新时间: 2024-10-17 17:10:00

本文为您介绍如何通过控制台查看 TDSQL-A PostgreSQL版 慢日志和错误日志明细。

慢日志明细

超过指定时间的 SQL 语句查询称为“慢查询”，对应语句称为“慢查询语句”，数据库管理员（DBA）对慢查询语句进行分析并找到慢查询出现原因的过程称为“慢查询分析”。

1. 登录 TDSQL-A 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，选择操作日志页，单击慢日志明细，选择时间，可以查看慢日志信息。

- 可以根据时间来查询慢日志信息。

- 也可以根据数据库名称进行搜索。

错误日志明细

在数据库运行过程中的运行日志中出现错误信息被记录在错误日志明细中。

1. 登录 TDSQL-A 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，选择操作日志页，单击错误日志明细，选择时间，可以查看错误日志信息。

安全组

最近更新时间: 2024-10-17 17:10:00

安全组 是一种有状态的包含过滤功能的虚拟防火墙，用于设置单台或多台云数据库的网络访问控制，是腾讯云提供的重要的网络安全隔离手段。安全组是一个逻辑上的分组，您可以将同一地域内具有相同网络安全隔离需求的云数据库实例加到同一个安全组内。云数据库与云服务器等共享安全组列表，安全组内基于规则匹配，具体规则与限制请参见 安全组详细说明。

说明：

由于云数据库没有主动出站流量，因此出站规则对云数据库不生效。

配置安全组

1. 登录 TDSQL-A 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，选择安全组页，单击配置安全组。

3. 在弹出的对话框，选择需要绑定的安全组，单击确定，即可完成安全组绑定云数据库的操作。

4. 安全组配置成功后，可在安全组页面下方查看新安全组信息。

删除安全组

1. 登录 TDSQL-A 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，选择安全组页，单击操作列的，删除安全组同时删除规则。

设置实例参数

最近更新时间: 2024-10-17 17:10:00

本文为您介绍如何通过 TDSQL-A PostgreSQL版 控制台查看和修改部分参数，及在控制台查询参数修改记录。

> **注意：**
>
> 为保证实例的稳定，控制台仅开放部分参数的修改，控制台的参数配置页面展示的参数即为用户可以修改的参数。

批量修改参数

1. 登录 TDSQL-A 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。

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。

> **说明：**
>
> CN 节点和 DN 节点可分开单独修改。

查看参数修改记录

1. 登录 TDSQL-A 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 > 最近修改记录页，可查看近期参数修改记录。

访问管理
访问管理概述

最近更新时间: 2024-10-17 17:10:00

存在问题

如果您在腾讯云中使用到了云服务器、私有网络、云数据库等多项服务，这些服务由不同的人管理，但都共享您的云账号密钥，将存在如下问题：

- 您的密钥由多人共享，泄密风险高。
- 您无法限制其它人的访问权限，易产生误操作造成安全风险。

解决方案

您可以通过子帐号实现不同的人管理不同的服务来规避以上的问题。默认情况下，子帐号没有使用云服务的权利或者相关资源的权限。因此，我们就需要创建策略来允许子帐号使用他们所需要的资源或权限。

访问管理（Cloud Access Management，CAM）是腾讯云提供的一套 Web 服务，它主要用于帮助客户安全管理腾讯云账户下的资源的访问权限。通过 CAM，您可以创建、管理和销毁用户（组），并通过身份管理和策略管理控制指定用户可以使用指定腾讯云资源。

当您使用 CAM 的时候，可以将策略与一个用户或一组用户关联起来，策略能够授权或者拒绝用户使用指定资源完成指定任务。有关 CAM 策略的更多相关基本信息，请参见策略语法，更多使用信息，请参见 策略。

若您不需要对子账户进行云数据库相关资源的访问管理，您可以跳过此章节。跳过这些部分不会影响您对文档中其余部分的理解和使用。

快速入门

CAM 策略必须授权使用一个或多个 TDSQL-A PostgreSQL版 操作，或者必须拒绝使用一个或多个 TDSQL-A PostgreSQL版 操作，同时还必须指定可以用于操作的资源（可以是全部资源，某些操作也可以是部分资源），策略还可以包含操作资源所设置的条件。

> **说明：**
> - 建议用户使用 CAM 策略来管理 TDSQL-A PostgreSQL版 资源和授权 TDSQL-A PostgreSQL版 操作，对于存量分项目权限的用户体验不变，但不建议再继续使用分项目权限来管理资源与授权操作。
> - TDSQL-A PostgreSQL版 暂时不支持相关生效条件设置。

- 建议用户使用 CAM 策略来管理 TDSQL-A PostgreSQL版 资源和授权 TDSQL-A PostgreSQL版 操作，对于存量分项目权限的用户体验不变，但不建议再继续使用分项目权限来管理资源与授权操作。
- TDSQL-A PostgreSQL版 暂时不支持相关生效条件设置。

授权策略语法

最近更新时间: 2024-10-17 17:10:00

CAM 策略语法

```
{
"version":"2.0",
"statement":
[
{
"effect":"effect",
"action":["action"],
"resource":["resource"],
"condition": {"key":{"value"}}
}
]
}
```

- **版本 version**：必填项，目前仅允许值为"2.0"。
- **语句 statement**：用来描述一条或多条权限的详细信息。该元素包括 effect、action、resource、condition 等多个其他元素的权限或权限集合。一条策略有且仅有一个 statement 元素。
  - **影响 effect**：必填项，描述声明产生的结果是"允许"还是"显式拒绝"。包括 allow（允许）和 deny（显式拒绝）两种情况。
  - **操作 action**：必填项，用来描述允许或拒绝的操作。操作可以是 API 或者功能集（一组特定的 API ，以 permid 前缀描述）。
  - **资源 resource**：必填项，描述授权的具体数据。资源是用六段式描述，每款产品的资源定义详情会有所区别。
  - **生效条件 condition**：必填项，描述策略生效的约束条件。条件包括操作符、操作键和操作值组成。条件值可包括时间、IP 地址等信息，有些服务允许您在条件中指定其他值。

TDSQL-A PostgreSQL版 的操作

在 CAM 策略语句中，您可以从支持 CAM 的任何服务中指定任意的 API 操作。对于 TDSQL-A PostgreSQL版，请使用以 name/tdapg: 为前缀的 API 。如果您要在单个语句中指定多个操作的时候，请使用逗号将它们隔开，如下所示：

```
"action":["name/tdapg:action1","name/tdapg:action2"]
```

您也可以使用通配符指定多项操作。例如，您可以指定名字以单词 "Describe" 开头的所有操作，如下所示：

```
"action":["name/tdapg:Describe*"]
```

如果您要指定 TDSQL-A PostgreSQL版 中所有操作，请使用 * 通配符，如下所示：

```
"action"：["name/tdapg:*"]
```

TDSQL-A PostgreSQL版 的资源路径

每个 CAM 策略语句都有适用于自己的资源。

资源路径的一般形式如下：

```
qcs:project_id:service_type:region:account:resource
```

- **qcs**：qcloud service 的简称，表示是腾讯云的云资源。该字段是必填项。

- **project_id**：描述项目信息，仅为了兼容 CAM 早期逻辑，无需填写。
- **service_type**：产品简称，如 tdapg。
- **region**：地域信息，如 bj。
- **account**：资源拥有者的主帐号信息，如 uin/12345678。
- **resource**：各产品的具体资源详情，如 instance/instance_id 或者 instance/*。

可授权的资源类型

最近更新时间: 2024-10-17 17:10:00

TDSQL-A PostgreSQL版 支持资源级授权，您可以指定子账号拥有特定资源的接口权限。

支持资源级授权的接口列表如下：

说明：

表中未列出的云数据库 API 操作，即表示该 TDSQL-A PostgreSQL版 API 操作不支持资源级权限。针对不支持资源级权限的 TDSQL-A PostgreSQL版 API 操作，您仍可以向用户授予使用该操作的权限，但策略语句的资源元素必须指定为 *。

API 名	API 描述	资源六段式示例
DescribeAccounts	查询云数据库实例帐号	qcs::tdapg:gz:uin/2113345772:instance/tdapg-i8edslnn
DescribeBackupDetails	查询云数据库实例备份详情	qcs:: tdapg:gz:uin/2113345772:instance/tdapg-i8edslnn
DescribeBackupLists	查询云数据库实例备份列表	qcs:: tdapg:gz:uin/2113345772:instance/tdapg-i8edslnn
DescribeBackupRules	查询云数据库实例备份规则	qcs:: tdapg:gz:uin/2113345772:instance/tdapg-i8edslnn
DescribeInstanceDetails	查询实例详细信息	qcs:: tdapg:gz:uin/2113345772:instance/tdapg-i8edslnn
DescribeInstances	查询实例列表	qcs:: tdapg:gz:uin/2113345772:instance/tdapg-i8edslnn
ModifyInstanceName	修改云数据库实例名称	qcs:: tdapg:gz:uin/2113345772:instance/tdapg-i8edslnn
ResetAccountPassword	重置云数据库帐号密码	qcs:: tdapg:gz:uin/2113345772:instance/tdapg-i8edslnn
SetBackupRules	设置云数据库实例备份规则	qcs:: tdapg:gz:uin/2113345772:instance/tdapg-i8edslnn

开发指南
设计规范

最近更新时间: 2024-10-17 17:10:00

命名规范

- 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"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。
- 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。
- 临时或备份的 DB object:table、view 等，建议加上日期，如 dbaa_ops.b2c_product_summay_2014_07_12 (其中 dba_ops 为 DBA 专用 schema)。
- index 命名规则：普通索引为表名_列名_idx，唯一索引为表名_列名_uidx，如 student_name_idx,student_id_uidx。

COLUMN 设计

- 建议使用数值类型，不用字符类型。
- 建议使用 varchar(N) ，不用 char(N)，以节省存储空间。
- 建议使用 varchar(N) ，不用 text、varchar。
- 建议使用 default NULL，不用default ''，以节省存储空间。
- 建议国际业务时，使用 timestamp with time zone(timestamptz)，不用 timestamp without time zone，避免时间函数在不同时区的时间点返回值不同，也为业务国际化扫清障碍。
- 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，不建议使用 real、double precision。

Constraints 设计

- 建议每个 table 都使用 shard key 作为主键或者唯一索引，主键或唯一索引必须带着分布键。
- 建议建表时一步到位，一起建立主键或者唯一索引。
- 目前列存表暂不支持外键

Index 设计

- TDSQL-A for PostgreSQL版 提供的 index 类型：B-tree、Hash、GiST (Generalized Search Tree)、SP-GiST (space-partitioned GiST)、GIN (Generalized Inverted Index)、BRIN (Block Range Index)，目前不建议使用 Hash，通常情况下使用 B-tree。目前列存表只支持B-tree和Hash索引。

- 建议 create 或 drop index 时，加 CONCURRENTLY 参数，可以达到与写入数据并发的效果。
- 建议对频繁 update、delete 的包含于 index 定义中的 column 的 table，使用 create index CONCURRENTLY、drop index CONCURRENTLY 的方式进行维护其对应 index。
- 建议用 unique index 代替 unique constraints，便于后续维护。
- 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。
- 建议对固定条件的（一般有特定业务含义）且数据占比低的 query，建议带 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;
```

- 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) );
```

- 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

- NULL 的判断：IS NULL、IS NOT NULL。
- 注意 boolean 类型取值 true、false、NULL。
- 注意 NOT IN 集合中带有 NULL 元素。

```
postgres=# select * from tdapg;
id | nickname
----+------------------------------
1 | hello TDSQL-A for PostgreSQL
2 | TDSQL-A for PostgreSQL好
3 | TDSQL-A for PostgreSQL好
4 | TDSQL-A for PostgreSQL default
(4 rows)
postgres=# select * from tdapg where id not in (null);
id | nickname
----+----------
(0 rows)
```

- 建议对字符串型 NULL 值处理后，进行 || 操作。

```
postgres=# select id,nickname from tdapg limit 1;
id | nickname
----+------------------------------
1 | hello TDSQL-A for PostgreSQL
(1 row)
postgres=# select id,nickname||null from tdapg limit 1;
id | ?column?
----+----------
1 |
(1 row)
postgres=# select id,nickname||coalesce(null,'') from tdapg limit 1;
id | ?column?
----+------------------------------
1 | hello TDSQL-A for PostgreSQL
(1 row)
```

- 建议使用 count(1) 或 count(*) 来统计行数，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 tdapg ;
id | nickname
----+------------------------------
1 | hello TDSQL-A for PostgreSQL
2 | TDSQL-A for PostgreSQL好
5 |
3 | TDSQL-A for PostgreSQL好
4 | TDSQL-A for PostgreSQL default
(5 rows)
postgres=# select count(1) from tdapg;
count
\-------
5
(1 row)
postgres=# select count(*) from tdapg;
count
\-------
5
(1 row)
postgres=# select count(nickname) from tdapg;
count
\-------
4
(1 row)
postgres=# select count((id,nickname)) from tdapg;
count
\-------
5
(1 row)
```

- 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 tdapg;
count
\-------
3
(1 row)
postgres=# select count(distinct (id,nickname)) from tdapg;
count
\-------
5
(1 row)
两个 NULL 的对比方法。
postgres=# select null is not distinct from null as Tdapgnull;
Tdapgnull
\-----------
t
(1 row)
```

开发相关规范

1) 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：

```
postgres=# \d+ TDAPG_main
Table "public.tdapg_main"
Column | Type | Modifiers | Storage | Stats target | Description
--------+---------+-----------+----------+--------------+-------------
id | integer | | plain | |
mc | text | | extended | |
Indexes:
"TDAPG_main_id_uidx" UNIQUE, btree (id)
Has OIDs: no
Distribute By SHARD(id)
Location Nodes: ALL DATANODES
postgres=# comment on column TDAPG_main.id is 'id号';
COMMENT
postgres=# comment on column TDAPG_main.mc is '产品名称';
COMMENT
postgres=# \d+ TDAPG_main
Table "public.tdapg_main"
Column | Type | Modifiers | Storage | Stats target | Description
--------+---------+-----------+----------+--------------+-------------
id | integer | | plain | | id号
mc | text | | extended | | 产品名称
Indexes:
"TDAPG_main_id_uidx" UNIQUE, btree (id)
Has OIDs: no
Distribute By SHARD(id)
Location Nodes: ALL DATANODES
```

2) 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 tdapg_main where id=1;
QUERY PLAN
\-----------------------------------------------------------------------------------------------
Index Scan using Tdapg_main_id_uidx on public.tdapg_main (cost=0.15..8.17 rows=1 width=36)
Output: id, mc
Index Cond: (Tdapg_main.id = 1)
(3 rows)
postgres=# explain (verbose) select tableoid from tdapg_main where id=1;
QUERY PLAN
\-----------------------------------------------------------------------------------------------
Index Scan using Tdapg_main_id_uidx on public.tdapg_main (cost=0.15..8.17 rows=1 width=4)
Output: tableoid
Index Cond: (Tdapg_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。

3) 建议 update 时，尽量做 <> 判断，如 update table_a set column_b = c where column_b <> c ;

```
postgres=# update tdapg_main set mc='TDSQL-A for PostgreSQL' ;
UPDATE 1
postgres=# select xmin,* from tdapg_main;
xmin | id | mc
------+----+-----------------------
2562 | 1 | TDSQL-A for PostgreSQL
(1 row)
postgres=# update tdapg_main set mc='TDSQL-A for PostgreSQL' ;
UPDATE 1
postgres=# select xmin,* from tdapg_main;
xmin | id | mc
------+----+-----------------------
2564 | 1 | TDSQL-A for PostgreSQL
(1 row)
postgres=# update tdapg_main set mc='TDSQL-A for PostgreSQL' where mc!='TDSQL-A for PostgreSQL';
UPDATE 0
postgres=# select xmin,* from tdapg_main;
xmin | id | mc
------+----+-----------------------
2564 | 1 | TDSQL-A for PostgreSQL
(1 row)
```

以上效果是一样的，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。

4) 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。

#session1 把所有数据都更新而不提交，一下子锁了2000千万条记录。

```
postgres=# begin;
BEGIN
postgres=# update tdapg_main set mc='TDAPG_1.3';
UPDATE 20000000
```

#session2 等待。

```
postgres=# update tdapg_main set mc='TDAPG_1.4' where id=1;
```

#session3 等待。

```
postgres=# update tdapg_main set mc='TDAPG_1.5' where id=2;
```

如果#session1分批更新的话，如下所示：

```
postgres=# begin;
BEGIN
postgres=# update tdapg_main set mc='TDAPG_1.3' where id>0 and id <=100000;
UPDATE 100000
postgres=#COMMIT;
postgres=# begin;
BEGIN
postgres=# update tdapg_main set mc='TDAPG_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。

5) 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。如下，性能相差5倍。

```
postgres=# insert into tdapg_main select t,'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx' from generate_series(1,100000) as t;
INSERT 0 100000
Time: 9511.755 ms
postgres=# copy TDAPG_main to '/data/pgxz/TDAPG_main.txt';
COPY 100002
Time: 179.428 ms
postgres=# copy TDAPG_main from '/data/pgxz/TDAPG_main.txt';
COPY 100002
Time: 1625.803 ms
postgres=#
```

6) 建议对报表类的或生成基础数据的查询，使用物化视图 (MATERIALIZED VIEW) 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 TDAPG_main 的总记录数，可参考如下：

```
postgres=# select count(1) from tdapg_main;
count
\--------
200004
(1 row)
Time: 27.948 ms
postgres=# create MATERIALIZED VIEW TDAPG_main_count as select count(1) as num from tdapg_main;
SELECT 1
Time: 322.372 ms
postgres=# select num from TDAPG_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 tdapg_main from '/data/pgxz/TDAPG_main.txt';
COPY 100002
Time: 1201.774 ms
postgres=# select count(1) from tdapg_main;
count
\--------
300006
(1 row)
Time: 23.164 ms
postgres=# REFRESH MATERIALIZED VIEW TDAPG_main_count;
REFRESH MATERIALIZED VIEW
Time: 49.486 ms
postgres=# select num from tdapg_main_count ;
num
\--------
300006
(1 row)
Time: 0.301 ms
```

7) 建议复杂的统计查询可以尝试窗口函数。

请参见 窗口函数应用。

8) 两表 join 时，尽量的使用分布 key 进行 join。

在建立业务的主表、明细表时，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 (TDAPG 2.01))
Type "help" for help.
postgres=# create table tdapg_main(id integer,mc text) distribute by shard(id);
CREATE TABLE
postgres=# create table tdapg_detail(id integer,tdapg_main_id integer,mc text) distribute by shard(TDAPG_main_id);
CREATE TABLE
postgres=# explain select TDAPG_detail.* from tdapg_main,TDAPG_detail where TDAPG_main.id=TDAPG_detail.TDAPG_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 TDAPG_detail.* from tdapg_main,TDAPG_detail where TDAPG_main.id=TDAPG_detail.TDAPG_main_id;
QUERY PLAN
\------------------------------------------------------------------------------------------------------------------------------------------------------------
----------
Data Node Scan on "__REMOTE_FQS_QUERY__" (cost=0.00..0.00 rows=0 width=0)
Output: TDAPG_detail.id, TDAPG_detail.TDAPG_main_id, TDAPG_detail.mc
Node/s: dn001, dn002
Remote query: SELECT TDAPG_detail.id, TDAPG_detail.TDAPG_main_id, TDAPG_detail.mc FROM public.TDAPG_main, public.TDAPG_detail WHERE (TDAPG_
main.id = TDAPG_detail.TDAPG_main_id)
(4 rows)
postgres=#
```

9) 分布键用唯一索引代替主键。

```
postgres=# create unique index TDAPG_main_id_uidx on TDAPG_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。

10) 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 TDAPG_detail_TDAPG_main_id_idx on TDAPG_detail using btree(TDAPG_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。

11) 不对字段建立外键。

目前 T 不支持多 dn 外键约束，除非能确定数据关联键的数据全部落在同一个 dn 上面。

数据库对象管理

最近更新时间: 2024-10-17 17:10:00

表空间

表空间允许在文件系统中定义用来存放表示数据库对象的文件的位置。在 TDSQL-A PostgreSQL版 中表空间实际上就是给表指定一个存储目录。

通过使用表空间，管理员可以控制一个TDSQL-A PostgreSQL版 安装的磁盘布局，有如下两个用处：

- 如果初始化集簇所在的分区或者卷用光了空间，而又不能在逻辑上扩展或者做别的操作，那么表空间可以被创建在一个不同的分区上，直到系统可以被重新配置。
- 表空间允许管理员根据数据库对象的使用模式来优化性能。例如，一个很频繁使用的索引可以被放在非常快并且非常可靠的磁盘上，如一种非常贵的固态设备。同时，一个很少使用的或者对性能要求不高的存储归档数据的表可以存储在一个便宜但比较慢的磁盘系统上。

系统自带表空间：

1. 表空间 pg_default 是用来存储系统目录对象、用户表、用户表 index、和临时表、临时表 index、内部临时表的默认空间，对应存储目录 $PADATA/base/。

2. 表空间 pg_global 用来存放系统字典表，对应存储目录 $PADATA/global/。

列出现有的表空间：

```
postgres-# \db
List of tablespaces
Name | Owner | Location
------------+--------+----------
pg_default | dbadmin |
pg_global | dbadmin |
(2 rows)
```

数据库

数据库（Database）是一个长期存储在计算机内的、有组织的、有共享的、统一管理的数据集合。

列出现有的数据库：

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

模式

模式本质上是一个名字空间，Oracle 里一般叫用户，SQL Server 中叫框架，MySQL 中叫数据库，模式里面包含表、数据类型、函数以及操作符，对象名称可以与在其他模式中存在的对象重名，访问某个模式中的对象时可以使用"模式名.对象名"。

表

关系型数据库中的一个表是一张二维表：它由行和列组成。

- 列的数量和顺序是固定的，并且每一列拥有一个名字。
- 行的数目是变化的，它反映了在一个给定时刻表中存储的数据量。SQL并不保证表中行的顺序。当一个表被读取时，表中的行将以非特定顺序出现，除非明确地指定需要排序。

每一列都有一个数据类型。数据类型约束着一组可以分配给列的可能值，并且它为列中存储的数据赋予了语义，这样它可以用于计算。

例如，一个被声明为数字类型的列将不会接受任何文本串，而存储在这样一列中的数据可以用来进行数学计算。反过来，一个被声明为字符串类型的列将接受几乎任何一种的数据，它可以进行如字符串连接的操作但不允许进行数学计算。

TDSQL-A PostgreSQL版 包括了相当多的内建数据类型，可以适用于很多应用。用户也可以定义他们自己的数据类型。大部分内建数据类型有着显而易见的名称和语义，详细解释参见 数据类型。

一些常用的数据类型是：用于整数的 integer、可以用于分数的 numeric、用于字符串的 text、用于日期的 date、用于一天内时间的 time，以及可以同时包含日期和时间的 timestamp。

索引

索引是提高数据库性能的常用方法。索引可以令数据库服务器以比没有索引快得多的速度查找和取回特定的行。不过索引也在总体上增加了数据库系统的负荷，因此建议恰当地使用它们。

TDSQL-A PostgreSQL版 提供了几种索引类型：普通索引、唯一索引、表达式索引、B-tree、Hash、GiST 和 GIN。每种索引类型都比较适合某些特定的查询类型，因为它们用了不同的算法。缺省时，CREATE INDEX命令将创建 B-tree 索引，它适合大多数情况。

- 普通索引是最基本的索引，没有任何限制。
- 唯一索引可用于约束索引属性值的唯一性，或者属性组合值的唯一性。
- 表达式索引建立在一个函数或者从表中一个或多个属性计算出来的表达式上。表达式索引只有在查询时使用与创建时相同的表达式才会起作用。
- B-tree 适合处理那些能够按顺序存储的数据之上的等于和范围查询。特别是在一个建立了索引的字段涉及到使用<、>=、=、>、>=操作符之一进行比较的时候，TDSQL-A PostgreSQL版 的查询规划器都会考虑使用 B-tree 索引。等效于这些操作符组合的构造，如BETWEEN 和 IN，也可以用搜索 B-tree 索引实现。
- 同样，索引列中的 IS NULL 或 IS NOT NULL 条件可以和 B-tree 索引一起使用。B-tree 索引也可以用来按照排序顺序检索数据。这并不总是比一个简单的扫描和排序快，但通常是有帮助的。
- Hash 索引只能处理简单的等于比较。当一个索引的列涉及到使用=操作符进行比较的时候，查询规划器会考虑使用 Hash 索引。
- GiST 索引不是单独一种索引类型，而是一种架构，可以在这种架构上实现很多不同的索引策略。因此可以使用 GiST 索引的操作符高度依赖于索引策略（操作符类）作为示例，TDSQL-A PostgreSQL版 的标准发布中包含用于二维几何数据类型的GiST 操作符类，它支持 <<、&<、&>、>>、<<|、&<|、|&>、|>>、&>、<&、~+、&& 操作符的索引查询。
- GIN 索引是通用倒排索引，它可以处理包含多个键的值（如数组）。与 GiST 类似，GIN 支持用户定义的索引策略可以使用 GIN 索引的操作符根据索引策略的不同而不同。

视图

视图是从一个或者多个表中导出的，它的行为与表非常相似，但是视图是一个虚拟表。数据库中仅存放视图的定义，而不存放视图对应的数据，这些数据仍存放在原来的基本表中。若基本表中的数据发生变化，从视图中查询出的数据也随之改变。

查看视图列表：

```
postgres-# \dv
List of relations
Schema | Name | Type | Owner
--------+--------------------+------+-------
public | pg_stat_statements | view | dbadmin
(1 row)
```

序列

序列对象（也叫序列生成器）就是用 CREATE SEQUENCE 创建的特殊单行表。一个序列对象通常用于为行或者表生成唯一的标识符。

查看序列列表：

```
postgres=# \ds
List of relations
Schema | Name | Type | Owner
--------+--------+----------+-------
public | serial | sequence | dbadmin
(1 row)
```

权限管理

最近更新时间: 2024-10-17 17:10:00

用户

使用 CREATE USER和ALTER USER 可以创建和管理数据库用户。数据库集群包含一个或多个已命名数据库。用户和角色在整个集群范围内是共享的，但是其数据并不共享。即用户可以连接任何数据库，但当连接成功后，任何用户都只能访问连接请求里声明的那个数据库。

```
postgres=# CREATE USER tdapg_user1 PASSWORD 'tdapg@123';
CREATE ROLE
```

如果创建有“创建数据库”权限的用户，则需要加 CREATEDB 关键字。

```
postgres=# CREATE USER tdapg_user2 CREATEDB PASSWORD 'tdapg@123';
CREATE ROLE
```

将用户 tdapg_user2 的登录密码由 tdapg@123 修改为 Abcd@123。

```
postgres=# ALTER USER tdapg_user2 with password 'Abcd@123';
ALTER ROLE
```

- 为用户 tdapg_user2 追加 CREATEROLE 权限。

```
ALTER USER tdapg_user2 CREATEROLE;
```

- 删除用户。

```
DROP USER tdapg_user2;
```

角色

角色是一组权限的集合。通过 GRANT 把角色授予用户后，用户即具有了角色的所有权限。推荐使用角色进行高效权限分配。例如，可以为设计、开发和维护人员创建不同的角色，将角色 GRANT 给用户后，再向每个角色中的用户授予其工作所需数据的差异权限。在角色级别授予或撤消权限时，这些更改将作用到角色下的所有成员。

创建一个可以登录的角色，但是不给他设置口令：

```
postgres=# CREATE ROLE Tdapg_role1 LOGIN;
CREATE ROLE
```

创建一个可以创建数据库和管理角色的角色：

```
postgres=# CREATE ROLE Tdapg_role2 WITH CREATEDB CREATEROLE;
CREATE ROLE
```

将角色的权限赋予用户。

```
postgres=# GRANT Tdapg_role1 , Tdapg_role2 TO tdapg_user1;
GRANT ROLE
```

Schema

Schema 又称作模式。通过管理 Schema，允许多个用户使用同一数据库而不相互干扰，可以将数据库对象组织成易于管理的逻辑组，同时便于将第三方应用添加到相应 Schema 下而不引起冲突。

每个数据库包含一个或多个 Schema。数据库中的每个 Schema 包含表和其他类型的对象。数据库创建初始，默认具有一个名为 public 的 Schema，且所有用户都拥有此 Schema 的权限。可以通过 Schema 分组数据库对象。Schema 类似于操作系统目录，但 Schema 不能嵌套。

用户权限设置

库级权限管理

使用超级用户创建用户 user1，并创建测试数据库。

```
postgres=# create user user1 password 'tdapg@123';
CREATE ROLE
postgres=# create database testdb;
CREATE DATABASE
```

赋予用户 user1 测试库增删查改权限。

```
postgres=# grant create,connect on database testdb to user1;
GRANT
```

使用用户 user1 进行测试库的增删查改操作。

```
[tdapg@TENCENT64 ~]$ psql -d testdb -p 11347 -U user1
psql (PostgreSQL 10.0 TDSQL-A for PostgreSQL)
Type "help" for help.


testdb=> create schema schm_user1;
CREATE SCHEMA
testdb=> create table schm_user1.t1(id int);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
testdb=> INSERT INTO schm_user1.t1 VALUES(1);
INSERT 0 1
testdb=> select * from schm_user1.t1;
id
----


1
(1 row)


testdb=> delete from schm_user1.t1 ;
DELETE 1
```

使用用户 user1 进行登录，回收 user1 用户的权限。

```
postgres=# revoke all on database testdb from user1;
REVOKE
```

使用 user1 连接 testdb 数据库，执行创建 schema 无权限，权限回收成功。

```
[tdapg@TENCENT64 ~]$ psql -d testdb -p 11347 -U user1
psql (PostgreSQL 10.0 TDSQL-A for PostgreSQL)
Type "help" for help.


testdb=> create schema schema_user1;
ERROR: permission denied for database testdb
```

表级权限管理

授权用户可以增删改查某个表记录。 使用管理员 dbadmin 连接到某个 CN 节点，下面操作相同。

授权用户可以访问某个模式。

```
postgres=# CREATE SCHEMA mysch;
CREATE SCHEMA
postgres=# CREATE USER user1 PASSWORD 'tdapg@123';
CREATE ROLE
postgres=# CREATE TABLE mysch.t2(id int,name text);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
```

```
postgres=# GRANT usage ON SCHEMA mysch TO user1;
GRANT
```

说明： 默认情况下，普通用户是无法访问没授权的 schema，所以要授权用户访问某个表的访问权限，则需要先将表所在的 schema 使用权分配给用户。 如果模式访问无权限，则提示 ERROR: permission denied for schema mysch。

授权用户可以增删改查某个表记录。

```
postgres=# GRANT SELECT ON mysch.t2 TO user1;
GRANT
postgres=# GRANT ALL ON mysch.t2 TO user1;
GRANT
```

说明： 增，删，改，查分别对应 INSERT/DELETE/UPDATE/SELECT。 如果需要全部权限，则可以写成 all。

移除用户的访问权限。

```
postgres=# REVOKE ALL ON mysch.t2 FROM user1;
REVOKE
postgres=# REVOKE SELECT ON mysch.t2 FROM user1;
REVOKE
```

将某个模式下所有表访问权限分配给某个用户。

```
postgres=# CREATE TABLE mysch.t4(f1 serial,f2 int);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# CREATE TABLE mysch.t5(f1 serial,f2 int);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# GRANT ALL ON ALL TABLES IN SCHEMA mysch TO user1;
GRANT
postgres=# \c - user1
You are now connected to database "postgres" as user "user1".
postgres=> SELECT * FROM mysch.t4;
f1 | f2
----+----
(0 rows)


postgres=> SELECT * FROM mysch.t5;
f1 | f2
----+----
(0 rows)
postgres=> \q
```

使用管理员 dbadmin 连接到某个 CN 节点，下面操作相同。 取消某个模式下所有表访问权限：

```
postgres=# REVOKE ALL ON ALL TABLES IN SCHEMA mysch FROM user1;
REVOKE
```

字段级权限管理

使用超级用户创建不同的用户 user1，并创建测试数据库及表 t0(id int ，name varchar(10)，num varchar(20))；。

```
postgres=# create user user1 password 'tdapg@123';
CREATE ROLE
postgres=# create database testdb;
CREATE DATABASE
postgres=# \c testdb
You are now connected to database "testdb" as user "dbadmin".
testdb=# create table t0(id int,name varchar(10),num varchar(20)) distribute by replication; CREATE TABLE
```

赋予用户 user1 表 t0 的 ID 字段增删查改权限。

```
testdb=# grant select(id),insert(id),update(id) on t0 to user1;
GRANT
```

使用用户 user1 对 t0 表的增删查改操作，在单独进行 ID 列的操作。

```
[tdapg@TENCENT64 ~]$ psql -d testdb -p 11347 -U user1
psql (PostgreSQL 10.0 TDSQL-A for PostgreSQL)
Type "help" for help.


testdb=> insert into t0 values(1,1,1);
ERROR: permission denied for relation t0
testdb=> update t0 set name=2;
ERROR: permission denied for relation t0
testdb=> select * from t0;
ERROR: permission denied for relation t0
testdb=> insert into t0(id) values(1);
INSERT 0 1
testdb=> update t0 set id=2;
UPDATE 1
testdb=> select id from t0;
id
----


2
(1 row)
```

使用超级用户连接 testdb 数据库，收回 user1 全部权限并查看权限信息。

```
[tdapg@VM-16-32-tlinux ~]$ psql -d testdb -U dbadmin -p 11347
psql (PostgreSQL 10.0 TDSQL-A for PostgreSQL)
Type "help" for help.


testdb=# revoke all(id) on t0 from user1;
REVOKE
```

使用用户 user1 对 t0 表的增删查改操作，无操作权限，权限回收成功。

```
[tdapg@TENCENT64 ~]$ psql -d testdb -p 11347 -U user1
psql (PostgreSQL 10.0 TDSQL-A for PostgreSQL)
Type "help" for help.


testdb=> insert into t0 values(1,1,1);
ERROR: permission denied for relation t0
testdb=> update t0 set id=2;
ERROR: permission denied for relation t0
testdb=> select * from t0;
ERROR: permission denied for relation t0
testdb=> insert into t0(id) values(1);
ERROR: permission denied for relation t0
testdb=> select id from t0;
ERROR: permission denied for relation t0
```

帐号安全

帐号安全策略

TDSQL-A PostgreSQL版 把传统数据库系统 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，这个三个角色之间相互制约，消除系统中的超级权限，从系统角色设计上了解决了数据安全问题 。

安全员职责：

- 定义强制访问、脱敏、加密策略。
- 安全员独立完成安全策略制定，不受管理员约束 。
- 安全策略，管理员也需要遵守，不例外。

审计员职责：

- 所有操作都可以被审计。
- 审计员独立完成审计策略制定，不受管理员约束 。
- 审计员操作被强制记录，不可更改。

原系统管理员：

- 仍具备自主访问控制权限、运维权限。
- 不可干预安全员、审计员操作。

帐号有效期

```
postgres=# create role user1 with login password 'user1@123' VALID UNTIL '2023-09-30 23:59:59';
CREATE ROLE
```

VALID UNTIL '2023-09-30 23:59:59' 表示用户密码到期时间戳，VALID UNTIL 'infinity' 让一个口令永远有效。

密码安全管理

TDSQL-A PostgreSQL版 中密码始终以加密方式存储在系统目录中。加密方式可以通过 password_encryption 参数配置。

```
postgres=# show password_encryption;
password_encryption
---------------------


md5
(1 row)
postgres=# select passwd from pg_shadow where usename='user1';
passwd
-------------------------------------


md57a215dcdf258f57c76edeec46243f85b
(1 row)
```

设置用户密码过时时间戳。

```
postgres=# alter role user1 with VALID UNTIL '2023-09-30 23:59:59';
ALTER ROLE
postgres=# alter role user1 with VALID UNTIL 'infinity';
ALTER ROLE
```

VALID UNTIL '2023-09-30 23:59:59' 表示用户密码到期时间戳，VALID UNTIL 'infinity' 让一个口令永远有效 。

数据类型

最近更新时间: 2024-10-17 17:10:00

数字类型

数字类型由2、4或8字节的整数以及4或8字节的浮点数和可选精度小数组成。TDSQL-A PostgreSQL版 列存表支持如下数字类型：

名字	存储尺寸	描述	范围
smallint	2字节	小范围整数	-32768 到 +32767
integer	4字节	整数的典型选择	-2147483648 到 +2147483647
bigint	8字节	大范围整数	-9223372036854775808 到 +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

货币类型

money 类型存储固定小数精度的货币数字：

名字	存储尺寸	描述	范围
money	8字节	货币额	-92233720368547758.08 到 +92233720368547758.07

字符类型

TDSQL-A PostgreSQL版 列存表支持如下字符类型：

- character varying(n) 和 character(n) 类型， 其中 n 是一个正整数。两种类型都可以存储最多 n 个字符长的串。
- text 类型，可以存储任何长度的串。
- name 类型，用于在内部系统目录中存储标识符并且不是给一般用户使用的。该类型长度当前定为64字节（63可用字符加结束符），但在 C 源代码应该使用常量 NAMEDATALEN 引用。这个长度是在编译的时候设置的（因而可以为特殊用途调整），缺省的最大长度在以后的版本可能会改变。
- "char"（注意引号）类型，和 char(1) 不一样，它只用了一个字节的存储空间。它在系统内部用于系统目录当做简化的枚举类型用。

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
"char"	1字节，单字节内部类型
numeric	64字节，用于对象名单内部类型

二进制数据类型

bytea 数据类型允许存储二进制串:

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

TDSQL-A PostgreSQL版 列存表支持 SQL 中所有的日期和时间类型：

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

TDSQL-A PostgreSQL版 提供标准的 SQL 类型 boolean，boolean 可以有多个状态：“true（真）”、“false（假）”和第三种状态“unknown（未知）”，未知状态由 SQL 空值表示。

名字	存储字节	描述
boolean	1字节	状态为真或假

几何类型

几何数据类型表示二维的空间物体。TDSQL-A PostgreSQL版 列存表支持如下几种几何类型：

名字	存储尺寸	表示	描述
point	16字节	平面上的点	(x,y)
line	32字节	无限长的线	{A,B,C}
lseg	32字节	有限线段	((x1,y1),(x2,y2))
box	32字节	矩形框	((x1,y1),(x2,y2))
path	16+16n字节	封闭路径（类似于多边形）	((x1,y1),...)
path	16+16n字节	开放路径	[(x1,y1),...]
circle	24字节	圆	<(x,y),r>(中心点和半径)

网络地址类型

网络地址类型是用于存储 IPv4、IPv6 和 MAC 地址的数据类型，用这些数据类型存储网络地址比用纯文本类型好，因为这些类型提供了输入错误检查以及特殊的操作符和函数。

名字	存储尺寸	描述

名字	存储尺寸	描述
cidr	7或19字节	IPv4 和 IPv6 网络
inet	7或19字节	IPv4 和 IPv6 主机及网络
macaddr	6字节	MAC 地址
macaddr8	8字节	MAC 地址（EUI-64 格式）

位串类型

位串就是1和0的串。它们可以用于存储和可视化位掩码。

支持两种类型的 SQL 位类型：bit(n) 和 bit varying(n)，其中 n 是一个正整数。

- bit 类型的数据必须准确匹配长度 n；试图存储短些或者长一些的位串都是错误的。
- bit varying 数据是最长 n 的变长类型，更长的串会被拒绝。

写一个没有长度的 bit 等效于 bit(1)，没有长度的 bit varying 则意味着没有长度限制。

示例：

```
postgres=# CREATE TABLE bit_test (col1 int,col2 BIT(3), col3 BIT VARYING(5))WITH(ORIENTATION=column);
CREATE TABLE
postgres=# INSERT INTO bit_test VALUES(1,B'101', B'00');
INSERT 0 1
postgres=# INSERT INTO bit_test VALUES(2,B'10'::bit(3), B'101');
INSERT 0 1
postgres=# SELECT * FROM bit_test;
col1 | col2 | col3
------+------+------
1 | 101 | 00
2 | 100 | 101
(2 rows)
```

文本搜索类型

TDSQL-A PostgreSQL版 提供两种数据类型：tsvector 和 tsquery，用来支持全文搜索。全文搜索是一种在自然语言的文档集合中，搜索以定位那些最匹配一个查询的文档的活动。

tsvector 类型表示一个为文本搜索优化的形式下的文档，一个 tsvector 值是一个排序的可区分词位列表，词位是被正规化合并了同一个词的不同变种的词，排序和去重是在输入期间自动完成的。

示例：

```
postgres=# SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
tsvector
----------------------------------------------------
'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
(1 row)
postgres=# SELECT $$the lexeme ' ' contains spaces$$::tsvector;
tsvector
-------------------------------------------
' ' 'contains' 'lexeme' 'spaces' 'the'
(1 row)
```

tsquery 类型表示一个文本查询。一个 tsquery 值存储要用于搜索的词位，并且使用布尔操作符&（AND）、|（OR）和!（NOT）来组合它们，还有短语搜索操作符<->（FOLLOWED BY）。也有一种 FOLLOWED BY 操作符的变体，其中 N 是一个整数常量，它指定要搜索的两个词位之间的距离。<->等效于<1>。

圆括号可以被用来强制对操作符分组。如果没有圆括号，!（NOT）的优先级最高，其次是<->（FOLLOWED BY），然后是&（AND），最后是|（OR）。

示例：

```
postgres=# SELECT 'fat & rat'::tsquery;
tsquery
---------------
'fat' & 'rat'
(1 row)
postgres=# SELECT 'fat & (rat | cat)'::tsquery;
tsquery
---------------------------
'fat' & ( 'rat' | 'cat' )
(1 row)
postgres=# SELECT 'fat:ab & cat'::tsquery;
tsquery
------------------
'fat':AB & 'cat'
(1 row)
```

UUID 类型

数据类型 uuid 存储由 RFC 4122、ISO/IEC 9834-8:2005 以及相关标准定义的通用唯一标识符（UUID）（某些系统将这种数据类型引用为全局唯一标识符 GUID）。这种标识符是一个128位的量，它由一个精心选择的算法产生，该算法能保证在已知空间中，任何其他使用相同算法的人能够产生同一个标识符的可能性非常非常小。因此，对于分布式系统，这些标识符相比序列生成器而言提供了一种很好的唯一性保障，序列生成器只能在一个数据库中保证唯一。

一个 UUID 被写成一个小写十六进制位的序列，该序列被连字符分隔成多个组：首先是一个8位组，接下来是三个4位组，最后是一个12位组。总共的32位（十六进制位）表示了128个二进制位。一个标准形式的 UUID 类似于：

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

XML 类型

XML 数据类型可以被用来存储 XML 数据。相比直接在一个 text 域中存储 XML 数据的优势在于，它会检查输入值的结构是不是良好，并且有支持函数用于在其上执行类型安全的操作。使用这种数据类型要求在安装时用 configure --with-libxml 选项编译。

XML 类型可以存储结构良好（如 XML 标准所定义）的“文档”，以及“内容”片段，它们由 XML 标准中的 XMLDecl? content 产品所定义。粗略地看，这意味着内容片段中可以有多于一个的顶层元素或字符节点。表达式 xmlvalue IS DOCUMENT 可以被用来评估一个特定的 XML 值是一个完整文档或者仅仅是一个文档片段。

JSON 类型

JSON 数据类型是用来存储 JSON（JavaScript Object Notation）数据。这种数据也可以被存储为 text，但是 JSON 数据类型的优势在于，能强制要求每个被存储的值符合 JSON 规则。也有很多 JSON 相关的函数和操作符可以用于存储在这些数据类型中的数据。

JSON 数据类型有两种：json 和 jsonb。它们几乎接受完全相同的值集合作为输入，主要的实际区别之一是效率。

json 数据类型存储输入文本的精准拷贝，处理函数必须在每次执行时必须重新解析该数据。

jsonb 数据被存储在一种分解好的二进制格式中，它在输入时要稍慢一些，因为需要做附加的转换。但是 jsonb 在处理时要快很多，因为不需要解析。jsonb 也支持索引。

由于 json 类型存储的是输入文本的准确拷贝，其中可能会保留在语法上不明显的、存在于记号之间的空格，还有 JSON 对象内部的键的顺序。还有，如果一个值中的 JSON 对象包含同一个键超过一次，所有的键/值对都会被保留（处理函数会把最后的值当作有效值）。相反，jsonb 不保留空格、不保留对象键的顺序并且不保留重复的对象键。如果在输入中指定了重复的键，只有最后一个值会被保留。

通常，除非有特别特殊的需要（如遗留的对象键顺序假设），大多数应用应该更愿意把 JSON 数据存储为 jsonb。

示例：

```
postgres=# SELECT '5'::json;
json
------
5
(1 row)
postgres=# SELECT '[1, 2, "foo", null]'::json;
json
----------------------
[1, 2, "foo", null]
```

```
(1 row)
postgres=# SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
json
-------------------------------------------------------
{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}
(1 row)
postgres=# SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"foo": {}}'::jsonb;
?column?
----------
t
(1 row)
postgres=# SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb;
?column?
----------
f
(1 row)
```

数组类型

允许一个表中的列定义为变长多维数组。可以创建任何内建或用户定义的基类、枚举类型或组合类型的数组。

一个数组数据类型可以通过在数组元素的数据类型名称后面加上方括号（[]）来命名。

示例：

```
postgres=# CREATE TABLE sal_emp (
postgres(# name text,
postgres(# pay_by_quarter integer[],
postgres(# schedule text[][]
postgres(# )with(orientation=column);
CREATE TABLE
postgres=# INSERT INTO sal_emp
postgres-# VALUES ('Bill',
postgres(# '{10000, 10000, 10000, 10000}',
postgres(# '{{"meeting", "lunch"}, {"training", "presentation"}}');
INSERT 0 1
postgres=#
postgres=# INSERT INTO sal_emp
postgres-# VALUES ('Carol',
postgres(# '{20000, 25000, 25000, 25000}',
postgres(# '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
INSERT 0 1
postgres=# SELECT * FROM sal_emp;
name | pay_by_quarter | schedule
-------+---------------------------+-------------------------------------------
Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
Bill | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
(2 rows)
```

另一种符合 SQL 标准的语法是使用关键词 ARRAY，可以用来定义一维数组。

如上示例中 sal_emp 表中 pay_by_quarter 字段可定义为：

```
pay_by_quarter integer ARRAY[4],
```

或者不指定数组尺寸：

```
pay_by_quarter integer ARRAY,
```

枚举类型

枚举（enum）类型是由一个静态、值的有序集合构成的数据类型。它们等效于很多编程语言所支持的 enum 类型。枚举类型的一个例子可以是一周中的日期，或者一个数据的状态值集合。

枚举类型可以使用 CREATE TYPE 命令创建，创建后，枚举类型可以像其他类型 一样，在表和函数定义中使用：

示例：

```
postgres=# CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TYPE
postgres=# CREATE TABLE person (
postgres(# name text,
postgres(# current_mood mood
postgres(# )WITH(ORIENTATION=column);
CREATE TABLE
postgres=# INSERT INTO person VALUES ('Moe', 'happy');
INSERT 0 1
postgres=# SELECT * FROM person WHERE current_mood = 'happy';
name | current_mood
------+--------------
Moe | happy
(1 row)
```

复合类型

一个复合类型表示一行或一个记录的结构，它本质上就是一个域名和它们数据类型的列表。

示例：

```
postgres=# CREATE TYPE inventory_item AS (
postgres(# name text,
postgres(# supplier_id integer,
postgres(# price numeric
postgres(# );
CREATE TYPE
postgres=# CREATE TABLE on_hand (
postgres(# item inventory_item,
postgres(# count integer
postgres(# )WITH(ORIENTATION=column);
CREATE TABLE
postgres=# INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
INSERT 0 1
postgres=# SELECT * FROM on_hand;
item | count
-----------------------+-------
("fuzzy dice",42,1.99) | 1000
(1 row)
```

范围类型

范围类型是表达某种元素类型（称为范围的 subtype）一个值的范围的数据类型。例如，timestamp 的范围可以被用来表达一个会议室被保留的时间范围。在这种情况下，数据类型是 tsrange（“timestamp range” 的简写，而 timestamp 是 subtype。subtype 必须具有一种总体的顺序，这样对于元素值是在一个范围值之内、之前或之后就是界线清楚的）。

范围类型非常有用，因为它们可以表达一种单一范围值中的多个元素值，并且可以很清晰地表达诸如范围重叠等概念。用于时间安排的时间和日期范围是最清晰的例子，但是价格范围、一种仪器的量程等也都有用。

TDSQL-A PostgreSQL版 列存表支持如下内建范围类型：

- int4range — integer 的范围
- int8range — bigint 的范围
- numrange — numeric 的范围
- tsrange — 不带时区的 timestamp 的范围
- tstzrange — 带时区的 timestamp 的范围

- daterange — date 的范围

示例：

```
postgres=# CREATE TABLE reservation (room int, during tsrange)WITH(ORIENTATION=column);
CREATE TABLE
postgres=# INSERT INTO reservation VALUES
postgres-# (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');
INSERT 0 1
postgres=# SELECT * FROM reservation;
room | during
------+-------------------------------------------------
1108 | ["2010-01-01 14:30:00","2010-01-01 15:30:00")
(1 row)
```

函数和操作符
逻辑操作符

最近更新时间: 2024-10-17 17:10:00

常用的逻辑操作符有：AND，OR，NOT，结果包括真，假和 null，null 表示“未知”。运算规则如下：

a	b	a AND b	a OR b	NOT a
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	NULL	NULL	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	NULL	FALSE	NULL	TRUE
NULL	NULL	NULL	NULL	NULL

比较操作符

最近更新时间: 2024-10-17 17:10:00

比较操作符可以用于所有可以比较的数据类型（两个数据类型必须是相同的数据类型或者是可以进行隐式转换的类型），所有比较操作符都是双目操作符，它们返回 boolean 类型。

TDSQL-A PostgreSQL版 对于常见的比较操作符都可用：

操作符	描述
<	小于
>	大于
<=	小于等于
>=	大于等于
=	等于
<>或者!=	不等于

数学函数和操作符号

最近更新时间: 2024-10-17 17:10:00

数学操作符

操作符	描述	示例	结果
+	加	3 + 4	7
-	减	10 - 9	1
*	乘	3 * 5	15
/	除（整数除法截断结果）	10 / 4	2
%	模（取余）	6 % 4	2
^	指数	2 ^ 3	8
|/	平方根	|/ 16	4
||/	立方根	||/ 8	2
!	阶乘	5 !	120
!!	阶乘（前缀操作符）	!! 5	120
@	绝对值	@ (-5)	5
&	按位与	2 & 3	2
|	按位或	2 | 3	3
#	按位异或	2 # 3	1
~	按位求反	~ 2	-3
<<	按位左移	2 << 3	16
>>	按位右移	8 >> 2	2

其中，按位操作操作符只能用于整数数据类型，而其它的操作符可以用于全部数字数据类型。按位操作的操作符还可以用于位串类型 bit 和 bit varying。

数学函数

函数	描述	示例	结果
abs(x)	绝对值	abs(-17.4)	17.4
cbrt(dp)	立方根	cbrt(27.0)	3
ceil(dp or numeric)	不小于参数的最小整数	ceil(-42.8)	-42
ceiling(dp or numeric)	不小于参数的最小整数（ceil 的别名）	ceiling(-95.3)	-95
degrees(dp)	把弧度转为角度	degrees(0.5)	28.6478897565412
div(y numeric, x numeric)	y/x 的整数商	div(9,4)	2
exp(dp or numeric)	指数	exp(1.0)	2.7182818284590452
floor(dp or numeric)	不大于参数的最大整数	floor(-42.8)	-43
ln(dp or numeric)	自然对数	ln(2.0)	0.6931471805599453
log(dp or numeric)	以10为底的对数	log(100.0)	2.0000000000000000

函数	描述	示例	结果
log(b numeric, x numeric)	以b为底的对数	log(2.0, 64.0)	6.0000000000000000
mod(y, x)	y/x 的余数	mod(9,4)	1
pi()	“π” 常数	pi()	3.14159265358979
power(a dp, b dp)	求 a 的 b 次幂	power(9.0, 3.0)	729.0000000000000000
power(a numeric, b numeric)	求 a 的 b 次幂	power(9.0, 3.0)	729.0000000000000000
radians(dp)	把角度转为弧度	radians(45.0)	0.785398163397448
round(dp or numeric)	圆整为最接近的整数	round(42.4)	42
round(v numeric, s int)	圆整为s位小数数字	round(42.4382, 2)	42.44
scale(numeric)	参数的精度（小数点后的位数）	scale(8.41)	2
sign(dp or numeric)	参数的符号（-1, 0, +1）	sign(-8.4)	-1
sqrt(dp or numeric)	平方根	sqrt(2.0)	1.414213562373095
trunc(dp or numeric)	截断（向零靠近）	trunc(42.8)	42
trunc(v numeric, s int)	截断为s位小数位置的数字	trunc(42.4382, 2)	42.43
width_bucket(operand dp, b1dp, b2 dp, count int)	返回一个桶，这个桶是在一个有 count 个桶， 上界为 b1，下界为 b2 的柱图中 operand 将被赋予的那个桶。为外部范围输入返回0或者 count+1	width_bucket(5.35, 0.024, 10.06, 5)	3
width_bucket(operand numeric, b1 numeric, b2 numeric, countint)	返回一个桶，这个桶是在一个有 count 个桶，上界为 b1，下界为 b2 的柱图中 operand 将被赋予的那个桶； 为范围外的输入返回0或者 count+1	width_bucket(5.35, 0.024, 10.06, 5)	3
width_bucket(operandanyelement, thresholdsanyarray)	返回一个桶，它是给定数组列出桶的下限 operand 将被赋予的那个桶， 为了输入低于第一下界返回0；thresholds 数组必须被存储，首先最小值，或者获取意想不到的结果	width_bucket(now(), array['yesterday', 'today', 'tomorrow']::timestamptz[])	2

随机函数

函数	返回类型	描述
random()	dp	范围 0.0 <= x < 1.0 中的随机值
setseed(dp)	viod	为后续的 random() 调用设置种子（值位于-1.0和1.0之间，包括边界值）

三角函数

所有三角函数都有类型为 double precision 的参数和返回类型。

三角函数参数表示为弧度。

反函数返回表示为弧度的值。

函数（弧度）	函数（角度）	描述
acos(x)	acosd(x)	反余弦
asin(x)	asind(x)	反正弦
atan(x)	atand(x)	反正切
atan2(y, x)	atan2d(y, x)	y/x 的反正切
cos(x)	cosd(x)	余弦

函数（弧度）	函数（角度）	描述
cot(x)	cotd(x)	余切
sin(x)	sind(x)	正弦
tan(x)	tand(x)	正切

字符串函数和操作符

最近更新时间: 2024-10-17 17:10:00

TDSQL-A PostgreSQL版 支持类型多样、功能丰富的字符串函数和操作符，在此列举部分常用的字符串操作符和函数。

操作符/函数	返回类型	描述
string || string	text	串接
char_length(string) or character_length(string)	int	串中字符数
lower(string)	text	将字符串转换为小写形式
overlay(string placing stringfrom int [for int])	text	替换子串
position(substring in string)	int	定位指定子串
substring(string [from int] [for int])	text	提取子串
trim([leading | trailing | both] [characters] fromstring)	text	从 string 的开头、结尾、两端（默认是 both ）删除最长的只包含来自 characters 字符（默认是一个空格）的串
upper(string)	text	将字符串转换成大写形式
ascii(string)	int	参数第一个字符的 ASCII 代码。对于 UTF8 返回该字符的Unicode 代码点。对于其他多字节编码，该参数必须是一个 ASCII 字符
btrim(string text [, characterstext])	text	从 string 的开头或结尾删除最长的只包含 characters（默认是一个空格）的串
concat(str "any" [, str "any" [, ...]])	text	串接所有参数的文本表示。NULL 参数被忽略
decode(string text, format text)	bytea	从 string 中的文本表达解码二进制数据。format 的选项和 encode 中的一样
left(str text, n int)	text	返回字符串中的前 n 个字符。当 n 为负时，将返回除了最后|n|个字符之外的所有字符
length(string)	int	string 中的字符数
lpad(string text, length int [, fill text])	text	将 string 通过前置字符 fill（默认是一个空格）填充到长度 length。如果 string 已经长于 length，则它被（从右边）截断
ltrim(string text [, characterstext])	text	从 string 的开头删除最长的只包含 characters（默认是一个空格）的串
repeat(string text, number int)	text	重复 string 指定的 number 次

示例：

```
postgres=# SELECT 'T'||'dapg';
?column?
----------
Tdapg
(1 row)

postgres=# SELECT char_length('jose');
char_length
-------------
4
(1 row)

postgres=# SELECT overlay('Txxxxas' placing 'hom' from 2 for 4);
overlay
---------
Thomas
(1 row)

postgres=# SELECT position('om' in 'Thomas');
position
----------
3
(1 row)

postgres=# SELECT substring('Thomas' from 2 for 3);
```

```
substring
-----------
hom
(1 row)

postgres=# SELECT trim(both from 'yxTomxx', 'xyz');
btrim
-------
Tom
(1 row)

postgres=# SELECT ascii('x');
ascii
-------
120
(1 row)

postgres=# SELECT btrim('xyxtrimyyx', 'xyz');
btrim
-------
trim
(1 row)

postgres=# SELECT concat('abcde', 2, NULL, 22);
concat
----------
abcde222
(1 row)
```

二进制串函数和操作符

最近更新时间: 2024-10-17 17:10:00

函数	返回类型	描述
string || string	bytea	串连接
octet_length(string)	int	二进制串中的字节数
overlay(string placing stringfrom int [for int])	bytea	替换子串
position(substring in string)	int	指定子串的位置
substring(string [from int] [for int])	bytea	提取子串
trim([both] bytes from string)	bytea	从 string 的开头或结尾删除只包含出现在 bytes 中的字节的最长串
btrim(stringbytea, bytesbytea)	bytea	从 string 的开头或结尾删除只包含出现在 bytes 中的字节的最长串
decode(stringtext, formattext)	bytea	从 string 中的文本表示解码二进制数据
encode(databytea, formattext)	text	将二进制数据编码为一个文本表示
get_bit(string, offset)	int	从串中抽取位
get_byte(string, offset)	int	从串中抽取字节
length(string)	int	二进制串的长度
md5(string)	text	计算 string 的 MD5 哈希码，十六进制形式返回结果
set_bit(string,offset, newvalue)	bytea	设置串中的位
set_byte(string,offset, newvalue)	bytea	设置串中的字节

位串函数和操作符

最近更新时间: 2024-10-17 17:10:00

操作符	描述	示例	结果
||	连接	B'10001' || B'011'	10001011
&	按位与	B'10001' & B'01101'	00001
|	按位或	B'10001' | B'01101'	11101
#	按位异或	B'10001' # B'01101'	11100
~	按位求反	~ B'10001'	01110
<<	按位左移	B'10001' << 3	01000
>>	按位右移	B'10001' >> 2	00100

模式匹配

最近更新时间: 2024-10-17 17:10:00

LIKE

```
string LIKE pattern [ESCAPE escape-character]
string NOT LIKE pattern [ESCAPE escape-character]
```

如果该 string 匹配了提供的 pattern，那么 LIKE 表达式返回真，和预期的一样，如果 LIKE 返回真，那么 NOT LIKE 表达式返回假，反之亦然。

一个等效的表达式是 NOT (string !IKE pattern)。

示例：

```
postgres=# SELECT 'abc' LIKE 'abc';
?column?
----------
t
(1 row)

postgres=# SELECT 'abc' LIKE 'a%';
?column?
----------
t
(1 row)

postgres=# SELECT 'abc' LIKE '_b_';
?column?
----------
t
(1 row)

postgres=# SELECT 'abc' LIKE 'c';
?column?
----------
f
(1 row)
```

SIMILAR TO 正则表达式

```
string SIMILAR TO pattern [ESCAPE escape-character]
string NOT SIMILAR TO pattern [ESCAPE escape-character]
```

SIMILAR TO 操作符根据自己的模式是否匹配给定串而返回真或者假。它和 LIKE 非常类似，只不过它使用 SQL 标准定义的正则表达式理解模式。

SIMILAR 除了从 LIKE 借用的功能外，还支持下面这些从 POSIX 正则表达式借用的模式匹配元字符：

- | 表示选择（两个候选之一）。
- * 表示重复前面的项零次或更多次。
- + 表示重复前面的项一次或更多次。
- ? 表示重复前面的项零次或一次。
- {m} 表示重复前面的项刚好 m 次。
- {m,} 表示重复前面的项 m 次或更多次。
- {m,n} 表示重复前面的项至少 m 次并且不超过 n 次。
- 可以使用圆括号 () 把多个项组合成一个逻辑项。

- 一个方括号表达式 [...] 声明一个字符类，就像 POSIX 正则表达式一样。
- 注意点号（.）不是 SIMILAR TO 的一个元字符。

示例：

```
postgres=# SELECT 'abc' SIMILAR TO 'abc';
?column?
----------
t
(1 row)

postgres=# SELECT 'abc' SIMILAR TO 'a';
?column?
----------
f
(1 row)

postgres=# SELECT 'abc' SIMILAR TO '%(b|d)%';
?column?
----------
t
(1 row)
```

POSIX 正则表达式

POSIX 正则表达式比 LIKE 和 SIMILAR TO 操作符更强大。

POSIX 正则表达式匹配操作符：

操作符	描述
~	匹配正则表达式，大小写敏感
~*	匹配正则表达式，大小写不敏感
!~	不匹配正则表达式，大小写敏感
!~*	不匹配正则表达式，大小写不敏感

示例：

```
postgres=# SELECT 'abc' ~ 'abc' ;
?column?
----------
t
(1 row)

postgres=# SELECT 'abc' ~ '^a' ;
?column?
----------
t
(1 row)

postgres=# SELECT 'abc' ~ '^(b|c)';
?column?
----------
f
(1 row)
```

数据类型格式化函数

最近更新时间: 2024-10-17 17:10:00

格式化函数用于把各种数据类型（日期/时间、整数、浮点、数字）转换成格式化的字符串，以及反过来从格式化的字符串转换成指定的数据类型。

下表列出了这些函数。这些函数都遵循一个公共的调用习惯：第一个参数是待格式化的值，而第二个是一个定义输出或输入格式的模版。

函数	返回类型	描述
to_char(timestamp,text)	text	把时间戳转成字符串
to_char(interval,text)	text	把间隔转成字符串
to_char(int,text)	text	把整数转成字符串
to_char(double precision,text)	text	把实数或双精度转成字符串
to_char(numeric,text)	text	把数字转成字符串
to_date(text,text)	date	把字符串转成日期
to_number(text,text)	numeric	把字符串转成数字
to_timestamp(text,text)	timestamp with time zone	把字符串转成时间戳

示例：

```
postgres=# SELECT to_char(interval '15h 2m 12s', 'HH24:MI:SS');
to_char
----------
15:02:12
(1 row)

postgres=# SELECT to_char(125, '999');
to_char
---------
125
(1 row)

postgres=# SELECT to_char(125.8::real, '999D9');
to_char
---------
125.8
(1 row)

postgres=# SELECT to_char(-125.8, '999D99S');
to_char
---------
125.80-
(1 row)

postgres=# SELECT to_date('05 Dec 2000', 'DD Mon YYYY');
to_date
------------
2000-12-05
(1 row)
```

时间日期函数和操作符

最近更新时间: 2024-10-17 17:10:00

日期/时间操作符

操作符	示例	结果
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00:00'
+	time '01:00' + interval '3 hours'	time '04:00:00'
-	- interval '23 hours'	interval '-23:00:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3' (days)
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00:00'
-	time '05:00' - time '03:00'	interval '02:00:00'
-	time '05:00' - interval '2 hours'	time '03:00:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00:00'
-	interval '1 day' - interval '1 hour'	interval '1 day -01:00:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00:00'
*	900 * interval '1 second'	interval '00:15:00'
*	21 * interval '1 day'	interval '21 days'
*	double precision '3.5' * interval '1 hour'	interval '03:30:00'
/	interval '1 hour' / double precision '1.5'	interval '00:40:00'

日期/时间函数

函数	返回值类型	描述
age(timestamp,timestamp)	interval	减去参数后的"符号化"结果，使用年和月，不只是使用天
age(timestamp)	interval	从 current_date 减去参数后的结果（在午夜）
clock_timestamp()	timestamp with time zone	实时时钟的当前时间戳（在语句执行时变化）
current_date	date	当前日期
current_time	time with time zone	当前时间
current_timestamp	timestamp with time zone	当前日期和时间

函数	返回值类型	描述
date_part(text,timestamp)	double precision	获得子域
date_part(text,interval)	double precision	获得子域
date_trunc(text,timestamp)	timestamp	截断到指定精度
date_trunc(text,interval)	interval	截断到指定精度
extract(field from timestamp)	double precision	获得子域
extract(field from interval)	double precision	获得子域
isfinite(date)	bool	测试有限日期
isfinite(timestamp)	bool	测试有限时间戳
isfinite(interval)	bool	测试有限间隔
justify_days(interval)	interval	调整间隔，30天时间周期可以表示为月
justify_hours(interval)	interval	调整间隔，24小时时间周期可以表示为日
justif_interval(interval)	interval	调整间隔
localtime	time	当前时间
localtimestamp	timestamp	当前日期和时间
make_date(year int, month int, day int)	date	从年、月、日创建日期
make_interval(years int DEFAULT 0, months int DEFAULT 0, weeksint DEFAULT 0, days int DEFAULT 0, hours int DEFAULT 0, minsint DEFAULT 0, secs double precision DEFAULT 0.0)	interval	从年、月、周、日、时、分、秒创建 interval
make_time(hour int, min int, sec double precision)	time	从时、分、秒创建时间
make_timestamp(year int, monthint, day int, hour int, minint, sec double precision)	timestamp	从年、月、日、时、分、秒创建时间戳
make_timestamptz(year int, month int, day int, hour int, min int, sec double precision, [timezone text])	timestamp with time zone	从年、月、日、时、分、秒创建带时区的时间戳，如果没有指定 timezone，则使用当前时区
now()	timestamp with time zone	当前日期和时间
statement_timestamp()	timestamp with time zone	当前日期和时间
timeofday()	text	当前日期和时间
transaction_timestamp()	timestamp with time zone	当前日期和时间
to_timestamp(double precision)	timestamp with time zone	把 UNIX 时间转换成 timestamp

OVERLAPS

```
(start1, end1) OVERLAPS (start2, end2)
(start1, length1) OVERLAPS (start2, length2)
```

操作符 OVERLAPS 通过如上两个表达式，在两个时间域重叠的时候得到真，不重叠时得到假。

示例：

```
postgres=# SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
postgres-# (DATE '2001-10-30', DATE '2002-10-30');
overlaps
----------
t
(1 row)

postgres=# SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
postgres-# (DATE '2001-10-30', DATE '2002-10-30');
overlaps
----------
f
(1 row)
```

EXTRACT

```
EXTRACT(field FROM source)
```

EXTRACT 函数从日期/时间值中抽取子域。

- source 必须是一个类型 timestamp、time 或 interval 的值表达式。
- field 是一个标识符或者字符串，它指定从源值中抽取的域。

EXTRACT 函数返回类型为 double precision 的值。

示例：

```
postgres=# SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----------
16
(1 row)

postgres=# SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----------
200
(1 row)

postgres=# SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----------
47
(1 row)

postgres=# SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----------
20
(1 row)
```

DATE_TRUNC

```
date_trunc('field', source)
```

DATE_TRUNC 函数在概念上和用于数字的 trunc 函数类似。

source 是类型 timestamp 或 interval 的值表达式。

field 指定对输入值选用什么样的精度进行截断，有效值为：microseconds、milliseconds、second、minute、hour、day、week、month、quarter、year、decade、century、millennium。

返回值是 timestamp 类型或者所有小于选定的精度的域都设置为零的 interval。

示例：

```
postgres=# SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
date_trunc
---------------------
2001-02-16 20:00:00
(1 row)

postgres=# SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
date_trunc
---------------------
2001-01-01 00:00:00
(1 row)
```

AT TIME ZONE

AT TIME ZONE 结构允许把时间戳转换成不同的时区。

示例：

```
postgres=# SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'MST';
timezone
------------------------
2001-02-17 11:38:40+08
(1 row)

postgres=# SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'MST';
timezone
---------------------
2001-02-16 18:38:40
(1 row)
```

第一个例子接受一个无时区的时间，戳然后把它解释成 MST 时间（UTC-7），然后这个时间转换为 PST（UTC-8）来显示。 第二个例子接受一个指定为 EST（UTC-5）的时间戳，然后把它转换成 MST（UTC-7）的当地时间。

函数 timezone(zone,timestamp) 等效于 SQL 兼容的结构 timestamp AT TIME ZONE zone。

当前日期/时间

如下函数可以获取当前日期和时间：

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(precision)
CURRENT_TIMESTAMP(precision)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(precision)
LOCALTIMESTAMP(precision)
```

延时执行

通过如下函数可以让服务器进程延时执行：

```
pg_sleep(seconds)
pg_sleep_for(interval)
pg_sleep_until(timestamp with time zone)
```

pg_sleep 让当前的会话进程休眠 seconds 秒以后再执行。

seconds 是一个 double precision 类型的值，所以可以指定带小数的秒数。

pg_sleep_for 是针对用 interval 指定的较长休眠时间的函数。

pg_sleep_until 则可以用来休眠到一个指定的时刻唤醒。

示例：

```
SELECT pg_sleep(1.5);
SELECT pg_sleep_for('5 minutes');
SELECT pg_sleep_until('tomorrow 03:00');
```

枚举支持函数

最近更新时间: 2024-10-17 17:10:00

枚举类型支持如下函数：

函数	描述
enum_first(anyenum)	返回输入枚举类型的第一个值
enum_last(anyenum)	返回输入枚举类型的最后一个值
enum_range(anyenum)	将输入枚举类型的所有值作为一个有序的数组返回
enum_range(anyenum, anyenum)	以一个数组返回在给定两个枚举值之间的范围，值必须来自相同的枚举类型。如果第一个参数为空，其结果将从枚举类型的第一个值开始；如果第二参数为空，其结果将以枚举类型的最后一个值结束

示例：

```
postgres=# CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue', 'purple');
CREATE TYPE
postgres=# SELECT enum_first(null::rainbow);
enum_first
------------
red
(1 row)

postgres=# SELECT enum_last(null::rainbow);
enum_last
-----------
purple
(1 row)
postgres=# SELECT enum_range(null::rainbow);
enum_range
---------------------------------------
{red,orange,yellow,green,blue,purple}
(1 row)

postgres=# SELECT enum_range('orange'::rainbow, 'green'::rainbow);
enum_range
----------------------
{orange,yellow,green}
(1 row)
postgres=# SELECT enum_range(NULL, 'green'::rainbow);
enum_range
--------------------------
{red,orange,yellow,green}
(1 row)

postgres=# SELECT enum_range('orange'::rainbow, NULL);
enum_range
---------------------------------
{orange,yellow,green,blue,purple}
(1 row)
```

几何函数和操作符

最近更新时间: 2024-10-17 17:10:00

几何类型 point、box、lseg、line、path、polygon 和 circle 有许多本地支持函数和操作符，具体可参考 官网。

网络地址函数和操作符

最近更新时间: 2024-10-17 17:10:00

cidr 和 inet 操作符

cidr 和 inet 类型支持如下操作符：

操作符	描述
<	小于
<=	小于等于
=	等于
>=	大于等于
>	大于
<>	不等于
<<	被包含在内
<<=	被包含在内或等于
>>	包含
>>=	包含或等于
&&	包含或被包含
~	按位 NOT
&	按位 AND
|	按位 OR
+	加
-	减

示例：

```
postgres=# SELECT inet '192.168.1.5' <= inet '192.168.1.5';
?column?
----------
t
(1 row)

postgres=# SELECT inet '192.168.1.5' << inet '192.168.1/24';
?column?
----------
t
(1 row)

postgres=# SELECT inet '192.168.1/24' && inet '192.168.1.80/28';
?column?
----------
t
(1 row)

postgres=# SELECT inet '192.168.1.6' + 25;
?column?
--------------
192.168.1.31
(1 row)

postgres=# SELECT inet '192.168.1.6' & inet '0.0.0.255';
?column?
----------
0.0.0.6
(1 row)
```

cidr 和 inet 函数

函数	返回类型	描述
abbrev(inet)	text	缩写显示格式文本
abbrev(cidr)	text	缩写显示格式文本
broadcast(inet)	inet	网络广播地址
family(inet)	int	抽取地址族：4为 IPv4，6为 IPv6
host(inet)	text	抽取 IP 地址为文本
hostmask(inet)	inet	为网络构造主机掩码
masklen(inet)	int	抽取网络掩码长度
netmask(inet)	inet	为网络构造网络掩码
network(inet)	cidr	抽取地址的网络部分
set_masklen(inet, int)	inet	为 inet 值设置网络掩码长度
set_masklen(cidr, int)	cidr	为 cidr 值设置网络掩码长度
text(inet)	text	抽取 IP 地址和网络掩码长度为文本
inet_same_family(inet, inet)	bool	地址是否来自同一个地址族
inet_merge(inet, inet)	cidr	最小的网络包括给定的两个网络

示例：

```
postgres=# SELECT abbrev(inet '10.1.0.0/16');
abbrev
------------
10.1.0.0/16
(1 row)
postgres=# SELECT family('::1');
family
--------
6
(1 row)

postgres=# SELECT hostmask('192.168.23.20/30');
hostmask
----------
0.0.0.3
(1 row)

postgres=# SELECT set_masklen('192.168.1.5/24', 16);
set_masklen
----------------
192.168.1.5/16
(1 row)

postgres=# SELECT netmask('192.168.1.5/24');
netmask
---------------
255.255.255.0
(1 row)
```

文本搜索函数和操作符

最近更新时间: 2024-10-17 17:10:00

文本搜索操作符

操作符	返回类型	描述
@@	bool	tsvector 是否匹配 tsquery
@@@	bool	同 @@
||	tsvector	连接 tsvector
&&	tsquery	将 tsvector 用 AND 连接起来
||	tsquery	将 tsquery 用 OR 连接起来
!!	tsquery	对一个 tsquery 取反
<->	tsquery	tsquery 后面跟着 tsquery
@>	bool	tsquery 包含另一个 tsquery
<@	bool	tsquery 是否被包含

示例：

```
postgres=# SELECT to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat');
?column?
----------
f
(1 row)

postgres=# SELECT 'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector;
?column?
----------------------------
'a':1 'b':2,5 'c':3 'd':4
(1 row)

postgres=# SELECT 'fat | rat'::tsquery && 'cat'::tsquery;
?column?
---------------------------
( 'fat' | 'rat' ) & 'cat'
(1 row)

postgres=# SELECT !! 'cat'::tsquery;
?column?
----------
!'cat'
(1 row)

postgres=# SELECT to_tsquery('fat') <-> to_tsquery('rat');
?column?
-----------------
'fat' <-> 'rat'
(1 row)

postgres=# SELECT 'cat'::tsquery @> 'cat & rat'::tsquery;
?column?
----------
f
(1 row)
```

文本搜索函数

函数	返回类型	描述
array_to_tsvector(text[])	tsvector	把词位数组转换成 tsvector
get_current_ts_config()	regconfig	获得默认文本搜索配置
length(tsvector)	integer	tsvector 中的词位数
numnode(tsquery)	integer	tsquery 中词位外加操作符的数目
querytree(query tsquery)	text	获得一个 tsquery 的可索引部分
setweight(tsvector, "char")	tsvector	为 tsvector 的每一个元素分配权重
to_tsquery([config regconfig ,] query text)	tsvector	规范化词并转换成 tsquery
to_tsvector([config regconfig,] document text)	tsvector	缩减文档文本成 tsvector
ts_delete(vector tsvector, lexeme text)	tsvector	从 vector 中移除给定的 lexeme
ts_filter(vector tsvector, weights "char"[])	tsvector	从 vector 中只选择带有给定权重的元素
ts_rewrite(query tsquery, targettsquery, substitute tsquery)	tsquery	在查询内用 substitute 替换 target
ts_rewrite(query tsquery, selecttext)	tsquery	使用来自一个 SELECT 的目标和替换者进行替换
tsvector_to_array(tsvector)	text[]	把 tsvector 转换为词位数组

示例：

```
postgres=# SELECT array_to_tsvector('{fat,cat,rat}'::text[]);
array_to_tsvector
-------------------
'cat' 'fat' 'rat'
(1 row)

postgres=# SELECT get_current_ts_config();
get_current_ts_config
-----------------------
simple
(1 row)

postgres=# SELECT length('fat:2,4 cat:3 rat:5A'::tsvector);
length
--------
3
(1 row)

postgres=# SELECT querytree('foo & ! bar'::tsquery);
querytree
-----------
'foo'
(1 row)

postgres=# SELECT to_tsquery('english', 'The & Fat & Rats');
to_tsquery
---------------
'fat' & 'rat'
(1 row)

postgres=# SELECT ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, 'fat');
ts_delete
------------------
'cat':3 'rat':5A
(1 row)

postgres=# SELECT tsvector_to_array('fat:2,4 cat:3 rat:5A'::tsvector);
tsvector_to_array
-------------------
{cat,fat,rat}
(1 row)
```

XML 函数

最近更新时间: 2024-10-17 17:10:00

支持 XML 类型和使用 XML 函数时，要求安装时使用 configure –with-libxml 进行编译。

XML 函数主要用于：产生 XML 内容、XML 谓词、处理 XML、将表映射到 XML。

JSON 函数和操作符

最近更新时间: 2024-10-17 17:10:00

JSON 和 JSONB 操作符

操作符	描述
->	获得 JSON 数组元素/通过键获得 JSON 对象域
->>	以文本形式获得 JSON 数组元素/对象域
#>	获取在指定路径的 JSON 对象
#>>	以文本形式获取在指定路径的 JSON 对象

示例：

```
postgres=# SELECT '[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json->2;
?column?
-------------
{"c":"baz"}
(1 row)

postgres=# SELECT '{"a": {"b":"foo"}}'::json->'a';
?column?
-------------
{"b":"foo"}
(1 row)

postgres=# SELECT '[1,2,3]'::json->>2;
?column?
----------
3
(1 row)

postgres=# SELECT '{"a":1,"b":2}'::json->>'b';
?column?
----------
2
(1 row)

postgres=# SELECT '{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}';
?column?
--------------
{"c": "foo"}
(1 row)
```

JSON 创建函数

函数	描述
to_json(anyelement) to_jsonb(anyelement)	把值返回为 json 或者 jsonb。数组和组合被（递归地）转换成数组和对象；否则，如果有从该类型到 json 的投影，将使用该投影函数来执行转换； 否则将产生一个标量值。对任何一个数值、布尔量或空值的标量类型， 将使用其文本表达，以这样一种方式使其成为有效的 json 或者 jsonb 值
array_to_json (anyarray [, pretty_bool])	把数组作为一个 JSON 数组返回。一个多维数组会成为一个数组 的 JSON 数组。如果 pretty_bool 为真，将在 第1维度的元素之间增加换行
row_to_json (record [, pretty_bool])	把行作为一个 JSON 对象返回。如果 pretty_bool 为真，将在第1层元素之间增加换行
json_build_array (VARIADIC "any") jsonb_build_array (VARIADIC "any")	从一个可变参数列表构造一个可能包含异质类型的 JSON 数组

函数	描述
json_build_object (VARIADIC "any") jsonb_build_object (VARIADIC "any")	从一个可变参数列表构造一个 JSON 对象。通过转换，该参数列表由交替出现的键和值构成
json_object(text[]) jsonb_object(text[])	从一个文本数组构造一个 JSON 对象。该数组必须可以是具有偶数个成员的一维数组（成员被当做交替出现的键/值对），或者是一个二维数组（每一个 内部数组刚好有2个元素，可以被看做是键/值对）
json_object (keys text[], values text[]) jsonb_object (keys text[], values text[])	json_object 的这种形式从两个独立的数组得到键/值对，在其他方面和一个参数的形式相同

示例：

```
postgres=# SELECT '{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}';
?column?
--------------
{"c": "foo"}
(1 row)

postgres=# SELECT to_json('Fred said "Hi."'::text);
to_json
----------------------
"Fred said \"Hi.\""
(1 row)

postgres=# SELECT array_to_json('{{1,5},{99,100}}'::int[]);
array_to_json
------------------
[[1,5],[99,100]]
(1 row)

postgres=# SELECT row_to_json(row(1,'foo'));
row_to_json
----------------------
{"f1":1,"f2":"foo"}
(1 row)

postgres=# SELECT json_build_array(1,2,'3',4,5);
json_build_array
-------------------
[1, 2, "3", 4, 5]
(1 row)
postgres=# SELECT json_build_object('foo',1,'bar',2);
json_build_object
------------------------
{"foo" : 1, "bar" : 2}
(1 row)
postgres=# SELECT json_object('{a, 1, b, "def", c, 3.5}');
json_object
---------------------------------------
{"a" : "1", "b" : "def", "c" : "3.5"}
(1 row)
postgres=# SELECT json_object('{{a, 1},{b, "def"},{c, 3.5}}');
json_object
---------------------------------------
{"a" : "1", "b" : "def", "c" : "3.5"}
(1 row)

postgres=# SELECT json_object('{a, b}', '{1,2}');
json_object
------------------------
{"a" : "1", "b" : "2"}
(1 row)
```

JSON 处理函数

函数	描述
json_array_length(json) jsonb_array_length(jsonb)	返回最外层 JSON 数组中的元素数量
json_each(json) jsonb_each(jsonb)	扩展最外层的 JSON 对象成为一组键/值对
json_each_text(json) jsonb_each_text(jsonb)	扩展最外层的 JSON 对象成为一组键/值对。返回值将是文本类型
json_object_keys(json) jsonb_object_keys(jsonb)	返回最外层 JSON 对象中的键集合
json_array_elements(json) jsonb_array_elements(jsonb)	把一个 JSON 数组扩展成一个 JSON 值的集合
json_array_elements_text(json) jsonb_array_elements_text(jsonb)	把一个 JSON 数组扩展成一个 text 值集合
json_typeof(json) jsonb_typeof(jsonb)	把最外层的 JSON 值的类型作为一个文本字符串返回。可能的类型是：object、array、string、number、boolean以及null
jsonb_insert(target jsonb, path text[], new_value jsonb, [insert_after boolean])	返回被插入了 new_value 的 target。如果 path 指定的 target 节在一个 JSONB 数组中，new_value 将被插入到目标之前（insert_after 为 false，默认情况）或者之后（insert_after 为真）。如果 path 指定的 target 节在一个 JSONB 对象内，则只有当 target 不存在时才插入 new_value。 对于面向路径的操作符来说，出现在 path 中的负整数表示从 JSON 数组的末尾开始计数。

示例：

```
postgres=# SELECT jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"');
jsonb_insert
------------------------------
{"a": [0, "new_value", 1, 2]}
(1 row)

postgres=# SELECT json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]');
json_array_length
-------------------
5
(1 row)

postgres=# SELECT * FROM json_each('{"a":"foo", "b":"bar"}');
key | value
-----+-------
a | "foo"
b | "bar"
(2 rows)
postgres=# SELECT * FROM json_each_text('{"a":"foo", "b":"bar"}');
key | value
-----+-------
a | foo
b | bar
(2 rows)

postgres=# SELECT json_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}}');
json_object_keys
------------------
f1
f2
(2 rows)

postgres=# SELECT * FROM json_array_elements('[1,true, [2,false]]');
value
-----------
1
true
[2,false]
(3 rows)
```

```
postgres=# SELECT * FROM json_array_elements_text('["foo", "bar"]');
value
-------
foo
bar
(2 rows)
```

序列操作函数

最近更新时间: 2024-10-17 17:10:00

序列对象被称为序列生成器或序列，使用 CREATE SEQUENCE 创建的特殊单行表，通常用于为表的行生成唯一标识符。

序列是非事务的，setval 造成的改变不会因事务的回滚而撤销。

常用的序列函数如下：

函数	返回值	描述
currval(regclass)	bigint	返回最近一次用 nextval 获取的指定序列的值
lastval()	bigint	返回最近一次用 nextval 获取的任何序列的值
nextval(regclass)	bigint	递增序列并返回新值
setval(regclass, bigint)	bigint	设置序列的当前值
setval(regclass, bigint, boolean)	bigint	设置序列的当前值以及 is_called 标志

示例：

```
postgres=# CREATE SEQUENCE seq;
CREATE SEQUENCE
postgres=# SELECT nextval('seq');
nextval
---------
1
(1 row)

postgres=# SELECT nextval('seq');
nextval
---------
2
(1 row)

postgres=# SELECT currval('seq');
currval
---------
2
(1 row)

postgres=# SELECT setval('seq',188);
setval
--------
188
(1 row)

postgres=# SELECT currval('seq');
currval
---------
188
(1 row)

postgres=# SELECT lastval();
lastval
---------
188
(1 row)
```

条件表达式

最近更新时间: 2024-10-17 17:10:00

CASE

CASE 表达式是一种通用的条件表达式，类似其他编程语言中的 if...else 语句，语法格式如下：

```
CASE WHEN condition THEN result
[WHEN ...]
[ELSE result]
END
```

或者：

```
CASE expression
WHEN value THEN result
[WHEN ...]
[ELSE result]
END
```

示例：

```
postgres=# CREATE TABLE test(a INT);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# INSERT INTO test VALUES(1),(2),(3);
COPY 3
postgres=# SELECT * FROM test;
a
---
1
3
2
(3 rows)

postgres=# SELECT a,
postgres-# CASE WHEN a=1 THEN 'one'
postgres-# WHEN a=2 THEN 'two'
postgres-# ELSE 'other'
postgres-# END
postgres-# FROM test;
a | case
---+-------
1 | one
3 | other
2 | two
(3 rows)
postgres=# SELECT a,
postgres-# CASE a WHEN 1 THEN 'one'
postgres-# WHEN 2 THEN 'two'
postgres-# ELSE 'other'
postgres-# END
postgres-# FROM test;
a | case
---+-------
1 | one
3 | other
2 | two
(3 rows)
```

COALESCE

COALESCE 函数返回它的第一个非空参数的值，仅当所有参数都为空时才返回空。它通常用于为显示目的检索数据时用缺省值替换空值。

语法格式如下：

```
COALESCE(value [, ...])
```

示例：

```
postgres=# SELECT COALESCE(null,'a',null,'hello');
coalesce
----------
a
(1 row)
```

NULLIF

NULLIF 语法格式如下：

```
NULLIF(value1, value2)
```

当 value1 和 value2 相等时，NULLIF 返回一个空值，否则返回 value1。

示例：

```
postgres=# SELECT NULLIF('hello','hello');
nullif
--------
(1 row)

postgres=# SELECT NULLIF(null,'hello');
nullif
--------
(1 row)
postgres=# SELECT NULLIF('hello',null);
nullif
--------
hello
(1 row)
```

GREATEST 和 LEAST

GREATEST 和 LEAST 函数从一个任意的数字表达式列表中选取最大或最小的数值，语法格式如下：

```
GREATEST(value [, ...])
LEAST(value [, ...])
```

示例：

```
postgres=# SELECT GREATEST(1,2,3,4,5);
greatest
----------
5
(1 row)

postgres=# SELECT LEAST(1,2,3,4,5);
least
-------
1
(1 row)
```

数组函数和操作符

最近更新时间: 2024-10-17 17:10:00

数组操作符

TDSQL-A PostgreSQL版 支持如下用于数组类型的操作符：

操作符	描述	示例	结果
=	等于	ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3]	t
<>	不等于	ARRAY[1,2,3] <> ARRAY[1,2,4]	t
<	小于	ARRAY[1,2,3] < ARRAY[1,2,4]	t
>	大于	ARRAY[1,4,3] > ARRAY[1,2,4]	t
<=	小于等于	ARRAY[1,2,3] <= ARRAY[1,2,3]	t
>=	大于等于	ARRAY[1,4,3]>=ARRAY[1,4,3]	t
@>	包含	ARRAY[1,4,3] @> ARRAY[3,1]	t
<@	被包含	ARRAY[2,7] <@ ARRAY[1,7,4,2,6]	t
&&	重叠	ARRAY[1,4,3] && ARRAY[2,1]	t
\|\| | 数组和数组串接 | ARRAY[1,2,3] \|\| ARRAY[4,5,6] | {1,2,3,4,5,6} |
\|\| | 数组和数组串接 | ARRAY[1,2,3]\|\|ARRAY[[4,5,6],[7,8,9]] | {{1,2,3},{4,5,6},{7,8,9}} |
\|\| | 元素到数组串接 | 3 \|\| ARRAY[4,5,6] | {3,4,5,6} |
\|\| | 数组到元素串接 | ARRAY[4,5,6] \|\| 7 | {4,5,6,7} |

数组函数

函数	返回类型	描述
array_append(anyarray, anyelement)	anyarray	向一个数组的末端追加一个元素
array_cat(anyarray, anyarray)	anyarray	连接两个数组
array_ndims(anyarray)	int	返回数组的维度数
array_length(anyarray, int)	int	返回被请求的数组维度的长度
array_lower(anyarray, int)	int	返回被请求的数组维度的下界
array_position(anyarray, anyelement [, int])	int	返回数组中第二个参数第一次出现的下标。 起始于第三个参数或第一个元素指示的元素位置（数组必须是一维的）
array_prepend(anyelement, anyarray)	anyarray	向一个数组的首部追加一个元素
array_remove(anyarray, anyelement)	anyarray	从数组中移除所有等于给定值的所有元素（数组必须是一维的）
array_replace(anyarray, anyelement, anyelement)	anyarray	将每一个等于给定值的数组元素替换成一个新值
array_to_string(anyarray, text [, text])	text	使用提供的定界符和可选的空串连接数组元素
array_upper(anyarray, int)	int	返回被请求的数组维度的上界
string_to_array(text, text [, text])	text[]	使用提供的定界符和可选的空串将字符串划分成数组元素

示例：

```
postgres=# SELECT array_append(ARRAY[1,2], 3);
array_append
--------------
{1,2,3}
(1 row)
postgres=# SELECT array_cat(ARRAY[1,2,3], ARRAY[4,5]);
array_cat
-------------
{1,2,3,4,5}
(1 row)
postgres=# SELECT array_ndims(ARRAY[[1,2,3], [4,5,6]]);
array_ndims
-------------
2
(1 row)
postgres=# SELECT array_length(array[1,2,3], 1);
array_length
--------------
3
(1 row)
postgres=# SELECT array_lower('[0:2]={1,2,3}'::int[], 1);
array_lower
-------------
0
(1 row)
postgres=# SELECT array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon');
array_position
----------------
2
(1 row)
postgres=# SELECT array_prepend(1, ARRAY[2,3]);
array_prepend
---------------
{1,2,3}
(1 row)

postgres=# SELECT array_remove(ARRAY[1,2,3,2], 2);
array_remove
--------------
{1,3}
(1 row)
postgres=# SELECT array_replace(ARRAY[1,2,5,4], 5, 3);
array_replace
---------------
{1,2,3,4}
(1 row)
postgres=# SELECT array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*');
array_to_string
-----------------
1,2,3,*,5
(1 row)
postgres=# SELECT array_upper(ARRAY[1,8,3,7], 1);
array_upper
-------------
4
(1 row)
postgres=# SELECT string_to_array('xx~^~yy~^~zz', '~^~', 'yy');
string_to_array
-----------------
{xx,NULL,zz}
(1 row)
```

范围函数和操作符

最近更新时间: 2024-10-17 17:10:00

范围操作符

范围类型可用的操作符：

操作符	描述
=	等于
<>	不等于
<	小于
>	大于
<=	小于等于
>=	大于等于
@>	包含范围/元素
<@	范围/元素被包含
&&	重叠
<<	严格左部
>>	严格右部
&<	不超过右部
&>	不超过左部
-|-	相邻
+	并
*	交
-	差

示例：

```
postgres=# SELECT int4range(1,5) = '[1,4]'::int4range;
?column?
----------
t
(1 row)

postgres=# SELECT numrange(1.1,2.2) <> numrange(1.1,2.3);
?column?
----------
t
(1 row)

postgres=# SELECT int4range(2,4) @> int4range(2,3);
?column?
----------
t
(1 row)
postgres=# SELECT numrange(5,15) + numrange(10,20);
?column?
----------
[5,20)
(1 row)
postgres=# SELECT int8range(5,15) * int8range(10,20);
```

```
?column?
----------
[10,15)
(1 row)
```

范围函数

函数	返回类型	描述
lower(anyrange)	范围的元素类型	范围的下界
upper(anyrange)	范围的元素类型	范围的上界
isempty(anyrange)	bool	范围是否为空
lower_inc(anyrange)	bool	下界是否包含在内
upper_inc(anyrange)	bool	上界是否包含在内
lower_inf(anyrange)	bool	下界是否无限
upper_inf(anyrange)	bool	上界是否无限
range_merge(anyrange, anyrange)	anyrange	最小范围其中包含两个给定范围

示例：

```
postgres=# SELECT lower(numrange(1.1,2.2));
lower
-------
1.1
(1 row)
postgres=# SELECT upper(numrange(1.1,2.2));
upper
-------
2.2
(1 row)
postgres=# SELECT isempty(numrange(1.1,2.2));
isempty
---------
f
(1 row)
postgres=# SELECT lower_inc(numrange(1.1,2.2));
lower_inc
-----------
t
(1 row)
postgres=# SELECT lower_inf('(,)'::daterange);
lower_inf
-----------
t
(1 row)
postgres=# SELECT range_merge('[1,2)'::int4range, '[3,4)'::int4range);
range_merge
-------------
[1,4)
(1 row)
```

聚集函数

最近更新时间: 2024-10-17 17:10:00

聚集函数用于从一个输入值的集合计算出一个单一结果。包括内建的通用聚集函数、统计信息聚合函数。

聚集函数类型很多，最常用的聚集函数如 sum、max、avg、count、array_agg 等。

窗口函数

最近更新时间: 2024-10-17 17:10:00

窗口函数提供在与当前查询行相关的行集合上执行计算的能力，通用的窗口函数如下：

函数	返回类型	描述
row_number()	bigint	当前行在其分区中的行号，从1计
rank()	bigint	带间隙的当前行排名；与该行的第一个同等行的 row_number 相同
dense_rank()	bigint	不带间隙的当前行排名； 这个函数计数同等组
percent_rank()	double precision	当前行的相对排名： (rank- 1) / (总分区行数 - 1)
cume_dist()	double precision	累积分配： (当前行前面的分区行数 或 与当前行同等的行的分区行数) / (总分区行数)
first_value(value any)	same type as value	返回在窗口帧中第一行上计算的 value
last_value(value any)	same type as value	返回在窗口帧中最后一行上计算的 value
nth_value(value any, nthinteger)	same type as value	返回在窗口帧中第 nth 行（行从1计数）上计算的 value；没有这样的行则返回空值

子查询表达式

最近更新时间: 2024-10-17 17:10:00

EXISTS

EXISTS 语法格式如下：

```
EXISTS (subquery)
```

它的参数是一个任意的 SELECT 语句或者子查询。系统对子查询进行运算以判断它是否返回行。

- 如果它至少返回一行，那么 EXISTS 的结果为真。
- 如果子查询没有返回行，那么它的结果是假。

示例：

```
postgres=# SELECT EXISTS(SELECT 1<0);
exists
--------
t
(1 row)


postgres=# SELECT EXISTS(SELECT 1 where 1<0);
exists
--------
f
(1 row)
```

IN

IN 的语法格式为：

```
expression IN (subquery)
```

右手边是一个圆括弧括起来的子查询，它必须正好只返回一个列。左手边表达式将被计算并与子查询结果逐行进行比较。

- 如果找到任何等于子查询行的情况，那么 IN 的结果就是“真”。
- 如果没有找到相等行，那么结果是“假”（包括子查询没有返回任何行的情况）。

示例：

```
postgres=# SELECT (1>0) IN('true','false');
?column?
----------
t
(1 row)


postgres=# SELECT (2+3) IN(3,4);
?column?
----------
f
(1 row)
```

NOT IN

NOT IN 的语法格式如下：

```
expression NOT IN (subquery)
```

右手边是一个用圆括弧包围的子查询，它必须返回正好一个列。左手边表达式将被计算并与子查询结果逐行进行比较。

- 如果只找到不相等的子查询行（包括子查询不返回行的情况），那么 NOT IN 的结果是“真”。
- 如果找到任何相等行，则结果为“假”。

示例：

```
postgres=# SELECT (1>0) NOT IN('true','false');
?column?
----------
f
(1 row)
postgres=# SELECT (2+3) NOT IN(3,4);
?column?
----------
t
(1 row)
```

ANY/SOME

ANY/SOME语法格式如下：

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

这种形式的右手边是一个圆括弧括起来的子查询，它必须返回正好一个列。左手边表达式将被计算并使用给出的操作符对子查询结果逐行进行比较。

- 如果获得任何真值结果，那么 ANY 的结果就是“真”。
- 如果没有找到真值结果，那么结果是“假”（包括子查询没有返回任何行的情况）。

SOME 是 ANY的同义词。IN 等价于 = ANY。

示例：

```
postgres=# CREATE TABLE any_test(a int);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# INSERT INTO any_test VALUES(1),(2),(3),(4),(5);
COPY 5
postgres=# SELECT * FROM any_test where a < ANY(SELECT AVG(a) FROM any_test);
a
---
1
2
(2 rows)
```

ALL

ALL 的语法格式如下：

```
expression operator ALL (subquery)
```

ALL 这种形式的右手边是一个圆括弧括起来的子查询，它必须只返回一列。左手边表达式将被计算并使用给出的操作符对子查询结果逐行进行比较。该操作符必须生成布尔结果。

- 如果所有行得到真（包括子查询没有返回任何行的情况），ALL的结果就是“真”。
- 如果没有存在任何假值结果，那么结果是“假”。
- 如果比较为任何行都不返回假并且对至少一行返回 NULL，则结果为 NULL。

NOT IN 等价于 <> ALL。

示例：

```
postgres=# CREATE TABLE all_test(a INT);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# INSERT INTO all_test VALUES(1),(2),(3),(4),(5);
COPY 5
postgres=# SELECT * FROM all_test WHERE a < ALL(SELECT * FROM all_test);
a
---
(0 rows)
postgres=# SELECT * FROM all_test WHERE 6 > ALL(SELECT * FROM all_test) order by 1;
a
---
1
2
3
4
5
(5 rows)
```

行和数组比较

最近更新时间: 2024-10-17 17:10:00

行和数组比较支持如下几种表达式：IN、NOT IN、ANY/SOME(array)、ALL(array)、行构造器比较和组合类型比较。

它们都返回布尔类型的结果。

SQL 语法参考

数据库操作

最近更新时间: 2024-10-17 17:10:00

数据库创建

要创建一个数据库，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。通过写 TEMPLATE template0 您可以创建一个干净的数据库，它将只包含您的 TDSQL-A PostgreSQL版 所预定义的标准对象。

默认参数创建数据库

```
postgres=# CREATE DATABASE tdapg_db;
CREATE DATABASE
```

指定克隆库

```
postgres=# CREATE DATABASE tdapg_db_template TEMPLATE template0;
CREATE DATABASE
```

指定所有者

```
postgres=# CREATE ROLE pgxz WITH LOGIN;
CREATE ROLE
postgres=# CREATE DATABASE tdapg_db_owner OWNER pgxz;
CREATE DATABASE
postgres=# \l+ tdapg_db_owner
List of databases
Name | Owner | Encoding | Collate | Ctype | Access privileges | Size | Tablespace | Description
----------------+-------+----------+------------+------------+-------------------+-------+------------+-------------
tdapg_db_owner | pgxz | UTF8 | en_US.utf8 | en_US.utf8 | | 18 MB | pg_default |
(1 row)
```

指定编码

```
postgres=# CREATE DATABASE tdapg_db_encoding ENCODING UTF8;
CREATE DATABASE
postgres=# \l+ tdapg_db_encoding
List of databases
Name | Owner | Encoding | Collate | Ctype | Access privileges | Size | Tablespace | Description
-------------------+-------+----------+------------+------------+-------------------+-------+------------+-------------
tdapg_db_encoding | dbadmin | UTF8 | en_US.utf8 | en_US.utf8 | | 18 MB | pg_default |
(1 row)
```

指定排序规则

```
postgres=# CREATE DATABASE tdapg_db_lc_collate LC_COLLATE 'zh_CN.utf8';
CREATE DATABASE
postgres=# \l+ tdapg_db_lc_collate
List of databases
Name | Owner | Encoding | Collate | Ctype | Access privileges | Size | Tablespace | Description
---------------------+-------+----------+------------+------------+-------------------+-------+------------+-------------
tdapg_db_lc_collate | dbadmin | UTF8 | zh_CN.utf8 | zh_CN.utf8 | | 18 MB | pg_default |
```

指定分组规则

```
postgres=# CREATE DATABASE tdapg_db_lc_ctype LC_CTYPE 'zh_CN.utf8';
CREATE DATABASE
postgres=# \l+ tdapg_db_lc_ctype
List of databases
Name | Owner | Encoding | Collate | Ctype | Access privileges | Size | Tablespace | Description
-------------------+-------+----------+------------+------------+-------------------+-------+------------+-------------
tdapg_db_lc_ctype | dbadmin | UTF8 | zh_CN.utf8 | zh_CN.utf8 | | 18 MB | pg_default |
(1 row)
```

配置数据可连接

```
postgres=# CREATE DATABASE tdapg_db_allow_connections ALLOW_CONNECTIONS true;
CREATE DATABASE
postgres=# SELECT datallowconn FROM pg_database WHERE datname='tdapg_db_allow_connections';
datallowconn
--------------
t
(1 row)
```

配置连接数

```
postgres=# CREATE DATABASE tdapg_db_connlimit CONNECTION LIMIT 100;
CREATE DATABASE
postgres=# SELECT datconnlimit FROM pg_database WHERE datname='tdapg_db_connlimit';
datconnlimit
--------------
100
(1 row)
```

配置数据库可以被复制

```
postgres=# CREATE DATABASE tdapg_db_istemplate IS_TEMPLATE true;
CREATE DATABASE
postgres=# SELECT datistemplate FROM pg_database WHERE datname='tdapg_db_istemplate';
datistemplate
---------------
t
(1 row)
```

多个参数一起配置

```
postgres=# CREATE DATABASE tdapg_db_mul OWNER pgxz CONNECTION LIMIT 50 TEMPLATE template0 ENCODING 'utf8' LC_COLLATE 'C';
CREATE DATABASE
```

数据库修改

修改数据库名称

```
postgres=# ALTER DATABASE tdapg_db RENAME TO tdapg_db_new;
ALTER DATABASE
```

修改数据库时，如果该数据库已经有 session 连接上来，则会提示如下错误：

```
ERROR: database "tdapg_db" is being accessed by other users
DETAIL: There are 6 other sessions using the database.
```

使用下面方法可以把 session 断开，然后再修改：

```
postgres=# SELECT pg_terminate_backend(pid) FROM pg_stat_activity WHERE datname='tdapg_db_template';
pg_terminate_backend
----------------------
t
(1 row)
```

修改连接数

```
postgres=# ALTER DATABASE tdapg_db_new CONNECTION LIMIT 50;
ALTER DATABASE
```

修改数据库所有者

```
postgres=# ALTER DATABASE tdapg_db_new OWNER TO dbadmin;
ALTER DATABASE
```

配置数据默认运行参行

```
postgres=# ALTER DATABASE tdapg_db_new SET search_path TO public,pg_catalog,pg_oracle;
ALTER DATABASE
```

ALTER DATABASE 不支持的项目：

项目	备注
encoding	编码
lc_collate	排序规则
lc_ctype	分组规则

数据库删除

```
postgres=# DROP DATABASE tdapg_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 tdapg_db_template;
ERROR: database "tdapg_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='tdapg_db_template';
pg_terminate_backend
----------------------
t
(1 row)
postgres=# DROP DATABASE tdapg_db_template;
DROP DATABASE
```

模式操作

最近更新时间: 2024-10-17 17:10:00

模式创建

标准语句：

```
postgres=# CREATE SCHEMA tdapg;
CREATE SCHEMA
```

扩展语法，不存在时才创建：

```
postgres=# CREATE SCHEMA IF NOT EXISTS tdapg ;
NOTICE: schema "tdapg" already exists, skipping
CREATE SCHEMA
```

指定所属用户：

```
postgres=# CREATE SCHEMA tdapg_pgxz AUTHORIZATION pgxz;
CREATE SCHEMA
postgres=# \dn tdapg_pgxz
List of schemas
Name | Owner
------------+-------
tdapg_pgxz | pgxz
(1 row)
```

模式修改

修改模式名：

```
postgres=# ALTER SCHEMA tdapg RENAME TO tdapg_new;
ALTER SCHEMA
```

修改所有者：

```
postgres=# ALTER SCHEMA tdapg_pgxz OWNER TO tdapg;
ALTER SCHEMA
```

模式删除

```
postgres=# DROP SCHEMA tdapg_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 tdapg_pgxz.t(id int);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# DROP SCHEMA tdapg_pgxz;
ERROR: cannot drop schema tdapg_pgxz because other objects depend on it
DETAIL: table tdapg_pgxz.t depends on schema tdapg_pgxz
HINT: Use DROP ... CASCADE to drop the dependent objects too.
```

使用 CASCADE 强制删除：

```
postgres=# DROP SCHEMA tdapg_pgxz CASCADE;
NOTICE: drop cascades to table tdapg_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：

```
[tdapg@VM_0_37_centos root]$ psql -U dbadmin
psql (PostgreSQL 10.0 TDSQL-A for PostgreSQL)
Type "help" for help.
postgres=# CREATE SCHEMA tdapg;
CREATE SCHEMA
postgres=# CREATE TABLE tdapg.t(id int);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
```

授权用户 pgxz 可以查询 tdapg.t 表：

```
postgres=# GRANT SELECT ON tdapg.t TO pgxz;
GRANT
postgres=# \q
[tdapg@VM_0_37_centos root]$ psql -U pgxz
psql (PostgreSQL 10.0 TDSQL-A for PostgreSQL)
Type "help" for help.
```

在没授权用户可以使用 tdapg 模式前，还是无法访问：

```
postgres=> SELECT * FROM tdapg.t;
ERROR: permission denied for schema tdapg
LINE 1: SELECT * FROM tdapg.t;
^
postgres=> \q
[tdapg@VM_0_37_centos root]$ psql -U dbadmin
psql (PostgreSQL 10.0 TDSQL-A for PostgreSQL)
Type "help" for help.
postgres=# GRANT USAGE ON SCHEMA tdapg TO pgxz;
GRANT
postgres=# \q
[tdapg@VM_0_37_centos root]$ psql -U pgxz
psql (PostgreSQL 10.0 TDSQL-A for PostgreSQL)
Type "help" for help.
```

授权用户可以使用 tdapg 模式后，可以访问 tdapg.t 表：

```
postgres=> SELECT * FROM tdapg.t;
id
----
(0 rows)
```

配置访问模式的顺序

TDSQL-A PostgreSQL版 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:

查看当前连接用户：

```
[tdapg@VM_0_37_centos root]$ psql -U dbadmin
psql (PostgreSQL 10.0 TDSQL-A for PostgreSQL)
Type "help" for help.
postgres=# SELECT current_user;
current_user
--------------
dbadmin
(1 row)
```

总共两个模式：

```
postgres=# \dn
List of schemas
Name | Owner
```

```
--------------+-------
public | dbadmin
tdapg | dbadmin
(2 rows)
```

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 “dbadmin”：

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

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_schema(id int,mc text);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# \dt t_schema
List of relations
Schema | Name | Type | Owner
--------+----------+-------+-------
dbadmin| t_schema | table | dbadmin
(1 row)
```

指定表位于某个模式下，不同模式下表名可以相同：

```
postgres=# CREATE SCHEMA tdapg_schema;
CREATE SCHEMA
postgres=# CREATE TABLE tdapg_schema.t_schema (id int,mc text);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# \dt tdapg_schema.t_schema
List of relations
Schema | Name | Type | Owner
--------------+----------+-------+-------
tdapg_schema | t_schema | table | dbadmin
(1 row)
```

访问不在搜索路径对象时，需要写全路径：

```
postgres=# CREATE TABLE tdapg_schema.t2 (id int,mc text);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# SELECT * FROM t2;
ERROR: relation "t2" does not exist
LINE 1: SELECT * FROM t2;
^
postgres=# SELECT * FROM tdapg_schema.t2;
id | mc
----+----
(0 rows)
```

上面出错是因为模式 tdapg_schema 没有配置在 search_path 搜索路径中。

表操作

最近更新时间: 2024-10-17 17:10:00

表创建

创建一个表，要用到 CREATE TABLE 命令。在这个命令中需要为新表至少指定一个名字、列的名字及数据类型。例如：

```
CREATE TABLE datatypetest (
col1 integer,
col2 character varying(20),
col3 date,
col4 jsonb,
col5 smallint
) WITH (orientation='row');
```

指定表的储存方式

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。

指定行存创建表

```
CREATE TABLE coltest1 (
id integer NOT NULL,
name character(16) NOT NULL,
start_date date,
subject character(1)
) WITH (orientation = 'row');
```

指定列存创建表

```
CREATE TABLE coltest1 (
id integer NOT NULL,
name character(16) NOT NULL,
start_date date,
subject character(1)
) WITH (orientation = 'column');
```

指定资源组创建表

```
CREATE TABLE t3(id integer,nc text) with (orientation ='column') DISTRIBUTE BY shard (id) TO GROUP default_group;
```

指定 schema 创建表

```
CREATE SCHEMA myschema;
CREATE TABLE myschema.t4(
col1 integer,
col2 char,
col3 bigint,
clo4 varchar(10)
) WITH (orientation ='column');
```

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。

- rle 主要针对大量重复的数据，如1111112222233311111进行压缩。
- delta 主要针对递增或者递减的数据，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，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，如针对 text 类型使用 delta。

原表大小15GB：

```
CREATE TABLE coltest2 (
c1 int encoding(compress_method='delta'),
c2 char(30),
c3 float4 ENCODING(compress_method='zstd', compress_level=19),
c4 date encoding(compress_method='zlib', compress_level=9),
c5 int encoding(compress_method='rle'),
c6 int encoding(compress_method='bitpack')
) WITH (orientation = 'column');
```

说明：

- delta 压缩，无压缩级别限制，压缩后大小14GB。
- zlib 压缩，压缩级别1-9，压缩级别为9时，压缩后大小4234MB。
- zstd 压缩，压缩级别1-19，压缩级别为19时，压缩后大小3691MB。
- rle 压缩，无压缩级别，压缩后大小14GB。
- bitpack 压缩，无压缩级别，压缩后大小14GB。

表的分布方式

复制表：表的每一行存在所有数据节点 DN 中，即每个数据节点都有完整的表数据。

```
CREATE TABLE t_rep (id int,mc text) DISTRIBUTE BY REPLICATION TO GROUP default_group;
INSERT INTO t_rep(id,mc) VALUES
(1,'ReplicationTableTest1'),(2,'ReplicationTableTest2'),(3,'ReplicationTableTest3');
```

shard 表：对指定的列进行 shard，把数据分布到指定 DN。不指定 shard key 建表方式，默认使用主键作为 shardkey，如果没有主键，则系统默认使用第一个字段作为表的 shard key。

```
CREATE TABLE t_shard(
f1 bigserial not null,
f2 integer,
f3 text,
f4 text,
f5 date) WITH (orientation = 'column')
DISTRIBUTE BY SHARD(f1) TO GROUP default_group;
```

继承表

单表继承

```
CREATE TABLE cities(
name text,
population float,
altitude int
) WITH (orientation ='column');
CREATE TABLE capitals(
state char(2)
) INHERITS (cities) WITH (orientation = 'column');
```

插入数据

```
INSERT INTO cities VALUES('Las Vegas', 1.53, 2174);
INSERT INTO cities VALUES('Mariposa',3.30,1953);
INSERT INTO capitals VALUES('Madison',4.34,845,'WI');
```

查询

```
SELECT name, altitude FROM cities WHERE altitude > 500;
SELECT name, altitude FROM capitals WHERE altitude > 500;
SELECT name,altitude FROM ONLY cities WHERE altitude > 500;
SELECT * FROM capitals;
```

多表继承

```
CREATE TABLE parent1 (FirstCol integer) WITH (orientation = 'column');
CREATE TABLE parent2 (FirstCol integer, SecondCol varchar(20)) WITH (orientation = 'column');
CREATE TABLE parent3 (FirstCol varchar(200)) WITH (orientation = 'column');
--子表 child1 将同时继承自 parent1 和 parent2 表，而这两个父表中均包含 integer 类型的 FirstCol 字段，因此 child1 可以创建成功
CREATE TABLE child1 (MyCol timestamp) INHERITS (parent1,parent2) WITH (orientation = 'column');
--子表 child2 将不会创建成功，因为其两个父表中均包含 FirstCol 字段，但是它们的类型不相同
CREATE TABLE child2 (MyCol timestamp) INHERITS (parent1,parent3) WITH (orientation = 'column');
--子表 child3 同样不会创建成功，因为它和其父表均包含 FirstCol 字段，但是它们的类型不相同
CREATE TABLE child3 (FirstCol varchar(20)) INHERITS(parent1) WITH (orientation = 'column');
```

使用IF NOT EXISTS

如果已经存在相同名称的表，不会抛出一个错误，而会发出一个通知，告知表关系已存在：

```
postgres=# CREATE TABLE t(id int,mc text);
CREATE TABLE
postgres=# CREATE TABLE t(id int,mc text);
ERROR: relation "t" already exists
postgres=# CREATE TABLE IF NOT EXISTS t(id int,mc text);
NOTICE: relation "t" already exists, skipping
CREATE TABLE
postgres=#
```

指定模式创建表

```
postgres=# CREATE TABLE public.t1(id int,mc text);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
```

使用查询结果创建数据表

```
postgres=# CREATE TABLE t(id int,mc text) DISTRIBUTE BY shard(mc);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# INSERT INTO t VALUES(1,'tdapg');
INSERT 0 1
postgres=# CREATE TABLE t_as as SELECT * FROM t;
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
INSERT 0 1
postgres=# SELECT * FROM t_as;
id | mc
----+-------
1 | tdapg
(1 row)
postgres=# \d+ t
Table "tdapg.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 "tdapg.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
```

表修改

创建表

```
CREATE TABLE t(
col1 bigint,
col2 char,
col3 text,
col4 varchar(5)
) WITH(ORIENTATION=column);
```

增加列

```
ALTER TABLE t ADD column_name varchar(20);
```

删除列

```
ALTER TABLE t DROP column_name;
```

修改列名

```
ALTER TABLE t RENAME col2 TO col5;
```

修改列的默认值

```
ALTER TABLE t ALTER col3 SET DEFAULT 'test';
```

修改表名

```
ALTER TABLE t RENAME TO t1;
```

修改列字段数据类型

```
ALTER TABLE t1 ALTER COLUMN col5 type TEXT;
```

目前列存模式下不支持修改列字段数据类型。

添加主键

```
ALTER TABLE t1 ADD CONSTRAINT t_id_pkey PRIMARY KEY (col1) ;
```

添加唯一约束

```
ALTER TABLE t1 ADD CONSTRAINT t1_CONSTR_clo3 CHECK (col3 IS NOT NULL);
```

添加检查约束

```
ALTER TABLE t1 ADD CONSTRAINT t1_CONSTR_clo3 CHECK (col3 IS NOT NULL);
```

外键创建

```
postgres=# CREATE TABLE t_p(f1 int not null,f2 int ,primary key(f1));
CREATE TABLE t_f(f1 int not null,f2 int );NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# CREATE TABLE t_f(f1 int not null,f2 int );
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# ALTER TABLE t_f ADD CONSTRAINT t_f_f1_fkey FOREIGN KEY (f1) REFERENCES t_p (f1);
ALTER TABLE
```

目前列存表不支持创建外键。

删除外键

```
ALTER TABLE t_f DROP CONSTRAINT t_f_f1_fkey;
```

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。

表删除

删除当前模式下的数据表

```
CREATE TABLE t(
col1 bigint
);
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 SCHEMA if not exists tdapg_schema;
CREATE SCHEMA
postgres=# CREATE TABLE tdapg_schema.t1( col1 bigint);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# CREATE VIEW tdapg_schema.t1_view AS SELECT * FROM tdapg_schema.t1 ;
CREATE VIEW
postgres=# DROP TABLE tdapg_schema.t1 ;
ERROR: cannot DROP TABLE tdapg_schema.t1 because other objects depend on it
DETAIL: view tdapg_schema.t1_view depends on table tdapg_schema.t1
HINT: Use DROP ... CASCADE to drop the dependent objects too.
postgres=# DROP TABLE tdapg_schema.t1 CASCADE;
NOTICE: drop cascades to view tdapg_schema.t1_view
DROP TABLE
```

数据插入

单条插入

```
postgres=# DROP TABLE IF EXISTS t6;
DROP TABLE
postgres=# CREATE TABLE t6(id int,mc text) WITH (ORIENTATION='column');
CREATE TABLE
postgres=# INSERT INTO t6 VALUES(1,'tdapg01');
INSERT 0 1
```

批量插入

```
postgres=# INSERT INTO t6 VALUES(1,'tdapg01'),(1,'tdapg02');
COPY 2
```

插入返回

```
postgres=# INSERT INTO t6 VALUES(1,'tdapg03') RETURNING id ;
id
----
1
```

```
(1 row)
INSERT 0 1
```

数据更新

更新语句

```
postgres=# DROP TABLE IF EXISTS t1;
DROP TABLE
postgres=# CREATE TABLE t1(name text, price integer, id serial) WITH (ORIENTATION ='column');
CREATE TABLE
postgres=# INSERT INTO t1(name, price) VALUES ('potato',4),('tomato',5),('chicken',20),('beef',50);
COPY 4
postgres=# UPDATE t1 SET price = price * 1.10 WHERE price <= 99.99;
UPDATE 4
```

更新返回

```
postgres=# DROP TABLE IF EXISTS t1;
DROP TABLE
postgres=# CREATE TABLE t1(name text, price integer, id serial) WITH (ORIENTATION ='column');
CREATE TABLE
postgres=# insert into t1(name, price) values ('potato',4),('tomato',5),('chicken',20),('beef',50);
COPY 4
postgres=# UPDATE t1 SET price = price * 1.10 WHERE price <= 99.99 RETURNING name, price AS new_price;
name | new_price
---------+-----------
potato | 4
tomato | 6
beef | 55
chicken | 22
(4 rows)
UPDATE 4
```

数据删除

删除数据 delete 操作：

```
postgres=# DROP TABLE IF EXISTS t1;
DROP TABLE
CREATE TABLE t1(
id integer,
nickname text
) WITH (ORIENTATION ='column');
INSERT INTO t1(id,nickname)
VALUES(1,'tdapgCTest1'),(2,'tdapgCTest2'),(3,'tdapgCTest3'),(33,'tdapgCTest3');
postgres=# DELETE FROM t1 WHERE id=1;
DELETE 1
```

数据清空

TRUNCATE 功能用于对表数据进行快速清除，TRUNCAT E属于 DDL 级别，会给 TRUNCATE 表加上 ACCESS EXCLUSIVE 最高级别的锁。

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,t6;
TRUNCATE TABLE
postgres=#
```

truncate 分区表

truncate 一个时间分区表：

```
CREATE TABLE t_time_range(
f1 bigint,
f2 timestamp,f3 varchar(20))
PARTITION BY range (f2) begin (timestamp without time zone '2017-09-01 0:0:0')
step (interval '1 month')
PARTITIONS (12)
WITH (orientation = 'column') ;
postgres=# INSERT INTO t_time_range VALUES(1,'2017-09-01','tdapg');
INSERT 0 1
postgres=# INSERT INTO t_time_range VALUES(2,'2017-10-01','pgxz');
INSERT 0 1
postgres=# \d+ t_time_range
Column oriented table "public.t_time_range"
Column | Type | TC method | TC level | LWC method | Collation | Nullable | Default | Storage | Stats target | Description
--------+------------------------------+-------------+----------+-------------+-----------+----------+---------+----------+--------------+-------------
f1 | bigint | no compress | | no compress | | | | plain | |
f2 | timestamp without time zone | no compress | | no compress | | | | plain | |
f3 | character varying(20) | no compress | | no compress | | | | extended | |
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
Options: orientation=column
postgres=# SELECT * FROM t_time_range;
f1 | f2 | f3
----+---------------------+-------
1 | 2017-09-01 00:00:00 | tdapg
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)
```

truncate 一个数值范围分区表：

```
CREATE TABLE t_range(
f1 bigint,
f2 timestamp default now(),
f3 integer
) PARTITION BY range (f3) begin (1) step (50) partitions (3) WITH (orientation = 'column') DISTRIBUTE BY shard(f1) TO GROUP default_group;
INSERT INTO t_range(f1,f3) VALUES (1,1);
INSERT INTO t_range(f1,f3) VALUES(2,50);
INSERT INTO t_range(f1,f3) VALUES(2,110);
INSERT INTO t_range(f1,f3) VALUES(3,100);
postgres=# \d+ t_range
Column oriented table "public.t_range"
Column | Type | TC method | TC level | LWC method | Collation | Nullable | Default | Storage | Stats target | Description
--------+------------------------------+-------------+----------+-------------+-----------+----------+---------+----------+--------------+-------------
f1 | bigint | no compress | | no compress | | | | plain | |
f2 | timestamp without time zone | no compress | | no compress | | | now() | plain | |
f3 | integer | no compress | | no compress | | | | plain | |
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES
PARTITION BY: RANGE(f3)
\# Of Partitions: 3
Start With: 1
Interval Of Partition: 50
Options: orientation=column
postgres=# SELECT * FROM t_range ;
f1 | f2 | f3
----+----------------------------+-----
```

```
3 | 2021-01-19 19:47:31.060817 | 100
1 | 2021-01-19 19:47:30.258023 | 1
2 | 2021-01-19 19:47:30.37273 | 50
2 | 2021-01-19 19:47:30.387583 | 110
(4 rows)
postgres=# TRUNCATE t_range partition for (1);
TRUNCATE TABLE
postgres=# SELECT * FROM t_range ;
f1 | f2 | f3
----+----------------------------+-----
3 | 2021-01-19 19:47:31.060817 | 100
2 | 2021-01-19 19:47:30.387583 | 110
(2 rows)
postgres=#
```

空间回收

数据库操作中，那些已经被 DELETE 的行并没有做物理删除，在完成 VACUUM 之前，这部分垃圾数据依然存在，所以如果对表进行了大量的更新或删除行，会产生大量磁盘页面碎片，降低查询效率，此时应运行 VACUUM FULL 命令来清理垃圾数据回收空间。

示例：

建表导数后，并对全表数据做delete删除，此时表大小不变，垃圾数据只是逻辑上做了删除：

```
postgres=# CREATE TABLE vacuum_test(a INT,b VARCHAR) WITH(ORIENTATION=column);
CREATE TABLE
postgres=# INSERT INTO vacuum_test VALUES(generate_series(1,10000),'hello world');
INSERT 0 10000
postgres=# DELETE FROM vacuum_test;
DELETE 10000
postgres=# \dt+ vacuum_test
List of relations
Schema | Name | Type | Owner | Size | Allocated Size | Description
--------+-------------+-------+-------+---------+----------------+-------------
public | vacuum_test | table | dbadmin | 144 MB | 144 MB |
(1 row)
```

列存表需要连接各个 DN 来执行 VACUUM 操作：

```
postgres=# EXECUTE DIRECT ON(dn001) 'VACUUM FULL vacuum_test;';
EXECUTE DIRECT
postgres=# EXECUTE DIRECT ON(dn002) 'VACUUM FULL vacuum_test;';
EXECUTE DIRECT
postgres=# EXECUTE DIRECT ON(dn003) 'VACUUM FULL vacuum_test;';
EXECUTE DIRECT
postgres=# EXECUTE DIRECT ON(dn004) 'VACUUM FULL vacuum_test;';
EXECUTE DIRECT
postgres=# EXECUTE DIRECT ON(dn005) 'VACUUM FULL vacuum_test;';
EXECUTE DIRECT
postgres=# EXECUTE DIRECT ON(dn006) 'VACUUM FULL vacuum_test;';
EXECUTE DIRECT
postgres=# \dt+ vacuum_test
List of relations
Schema | Name | Type | Owner | Size | Allocated Size | Description
--------+-------------+-------+-------+---------+----------------+-------------
public | vacuum_test | table | dbadmin | 0 bytes | 0 bytes |
(1 row)
```

VACUUM FULL 操作完成后，表中垃圾数据被清理，空间回收。

除 VACUUM 之外，系统自动清理进程（autovacuum）会自动执行 VACUUM 命令，回收被标识为删除状态的记录空间。

分区表

最近更新时间: 2024-10-17 17:10:00

int 型范围 range 分区

```
DROP TABLE if exists t_range;
CREATE TABLE t_range(
f1 bigint,
f2 timestamp default now(),
f3 integer
) PARTITION BY range (f3) begin (1) step (50) partitions (3) with (orientation = 'column') DISTRIBUTE BY shard(f1) TO GROUP default_group;
INSERT INTO t_range(f1,f3) VALUES(1,150);
```

timestamp 类型 range 分区

```
DROP TABLE IF EXISTS t_time_range;
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') **st
ep** (interval '1 month') **partitions** (12) with (orientation = 'column') DISTRIBUTE BY shard(f1) TO GROUP default_group;
```

list 分区

```
CREATE TABLE t_native_list(
f1 bigserial not null,
f2 text,
f3 integer,
f4 date
) PARTITION BY list(f2) with (orientation = 'column') DISTRIBUTE BY shard(f1) TO GROUP default_group;
--two child tables
CREATE TABLE t_native_list_gd partition of t_native_list(f1 ,f2 , f3,f4) for VALUES in ('guangdong');
CREATE TABLE t_native_list_bj partition of t_native_list(f1 ,f2 , f3,f4) for VALUES in ('beijing');
```

二级分区

先 list 分区，再 range 分区：

```
CREATE TABLE t_native_mul_list(
f1 bigserial not null,
f2 integer,
f3 text,
f4 text,
f5 date) PARTITION BY list ( f3 ) WITh (orientation = 'column') DISTRIBUTE BY shard(f1) TO GROUP default_group;
--secondary tables
CREATE TABLE t_native_mul_list_gd partition of t_native_mul_list for VALUES in ('guangdong') PARTITION BY range(f5) with (orientation = 'column');
CREATE TABLE t_native_mul_list_bj partition of t_native_mul_list for VALUES in ('beijing') PARTITION BY range(f5) with (orientation = 'column');
CREATE TABLE t_native_mul_list_sh partition of t_native_mul_list for VALUES in ('shanghai') WITh (orientation = 'column');
--three level tables
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') WITh (orientat
ion = 'column');
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') WITh (orientat
ion = 'column');
CREATE TABLE t_native_mul_list_bj_201701 partition of t_native_mul_list_bj(f1,f2,f3,f4,f5) for VALUES FROM ('2017-01-01') to ('2017-02-01') WITh (orientatio
n = 'column');
CREATE TABLE t_native_mul_list_bj_201702 partition of t_native_mul_list_bj(f1,f2,f3,f4,f5) for VALUES FROM ('2017-02-01') to ('2017-03-01') WITh (orientatio
n = 'column');
DROP TABLE t_native_mul_list_bj_201702;
DROP TABLE t_native_mul_list_bj_201701;
DROP TABLE t_native_mul_list_gd_201702;
DROP TABLE t_native_mul_list_gd_201701;
```

```
DROP TABLE t_native_mul_list_sh;
DROP TABLE t_native_mul_list_gd;
DROP TABLE t_native_mul_list;
```

索引操作

最近更新时间: 2024-10-17 17:10:00

索引创建

普通索引

```
postgres=# CREATE TABLE t_appoint(id int,name text) WITh (orientation = 'column');
CREATE TABLE
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 TABLE t_first_col_share (id int,nickname text) with (orientation = 'column');
CREATE TABLE
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。

唯一索引必须带着分布键

```
postgres=# CREATE UNIQUE INDEX t_first_col_share_nickname_uidx on t_first_col_share using btree(nickname);
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) WITh (orientation = 'column');
NOTICE: Replica identity is needed for shard table, please add to this table through "" command.
CREATE TABLE
postgres=# INSERT INTO t_upper SELECT t,md5(t::text) FROM generate_series(1,10000) as t;
INSERT 0 10000
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, dn003
-> Index Scan using t_upper_mc on t_upper (cost=0.28..32.58 rows=17 width=36)
Index Cond: (upper(mc) = 'c4ca4238a0b923820dcc509a6f75849b'::text)
(4 rows)
```

条件索引

```
postgres=# CREATE TABLE t_sex(id int,sex text) WITH (orientation = 'column');
NOTICE: Replica identity is needed for shard table, please add to this table through "" command.
CREATE TABLE
postgres=# CREATE INDEX t_sex_sex_idx on t_sex (sex);
CREATE INDEX
postgres=# INSERT INTO t_sex SELECT t,'男' FROM generate_series(1,1000000) as t;
INSERT 0 1000000
postgres=# INSERT INTO t_sex SELECT t,'女' FROM generate_series(1,100) as t;
INSERT 0 100
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=0.42..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
[tdapg@VM_0_37_centos shell]$ psql -p 11010
psql (PostgreSQL 10.0 TDSQL-A for PostgreSQL)
Type "help" for help.
postgres=# \di+
List of relations
Schema | Name | Type | Owner | Table | Size | Allocated Size | Description
--------+---------------+-------+-------+---------+-------+----------------+-------------
dbadmin | t_sex_sex_idx | index | dbadmin | t_sex | 14 MB | 14 MB |
dbadmin | t_upper_mc | index | dbadmin | t_upper | 14 MB | 14 MB |
(2 rows)
postgres=# \q
[tdapg@VM_0_37_centos shell]$ psql
psql (PostgreSQL 10.0 TDSQL-A for PostgreSQL)
Type "help" for help.
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=# \q
[tdapg@VM_0_37_centos shell]$ psql -p 11010
psql (PostgreSQL 10.0 TDSQL-A for PostgreSQL)
Type "help" for help.
postgres=# \di+
List of relations
Schema | Name | Type | Owner | Table | Size | Allocated Size | Description
--------+---------------+-------+-------+---------+-------+----------------+-------------
dbadmin | t_sex_sex_idx | index | dbadmin | t_sex | 16 kB | 16 kB |
dbadmin | t_upper_mc | index | dbadmin| t_upper | 14 MB | 14 MB |
(2 rows)
postgres=#
```

gist 索引

```
postgres=# CREATE TABLE t_trgm (id int,trgm text,no_trgm text) ;
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# CREATE INDEX t_trgm_trgm_idx ON t_trgm USING gist(trgm gist_trgm_ops);
CREATE INDEX
```

列存模式不支持 gist 索引。

gin 索引

```
postgres=# DROP INDEX t_trgm_trgm_idx;
DROP INDEX
postgres=# CREATE INDEX t_trgm_trgm_idx ON t_trgm USING gin(trgm gin_trgm_ops);
CREATE INDEX
```

列存模式不支持 gin 索引。

jsonb 索引

```
postgres=# CREATE TABLE t_jsonb(id int,f_jsonb jsonb);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# CREATE INDEX t_jsonb_f_jsonb_idx ON t_jsonb using gin(f_jsonb);
CREATE INDEX
```

列存模式不支持 jsonb 索引。

数组索引

```
postgres=# CREATE TABLE t_array(id int, mc text[]);
NOTICE: Replica identity is needed for shard table, please add to this table through "" command.
CREATE TABLE
postgres=# INSERT INTO t_array SELECT t,('{'||md5(t::text)||'}')::text[] FROM generate_series(1,1000000) as t;
INSERT 0 1000000
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
```

列存模式不支持数组索引。

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);
NOTICE: Replica identity is needed for shard table, please add to this table through "" command.
CREATE TABLE
postgres=# INSERT INTO gin_mul SELECT random()*5000, random()*6000, now()+((30000-60000*random())||' sec')::interval , md5(random()::text), round((ra
ndom()*100000)::numeric,2), md5(random()::text) FROM generate_series(1,1000000);
INSERT 0 1000000
postgres=# CREATE EXTENSION btree_gin;
CREATE EXTENSION
postgres=# CREATE INDEX gin_mul_gin_idx ON gin_mul using gin(f1,f2,f3,f4,f5,f6);
CREATE INDEX
```

#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=# 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='fa627dc16c2bd026150afa0453a0991d';
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)
```

多字段索引

```
postgres=# CREATE TABLE t_mul_idx (f1 int,f2 int,f3 int,f4 int);
NOTICE: Replica identity is needed for shard table, please add to this table through "" command.
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 bitmap scan 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 ;
QUERY PLAN
-----------------------------------------------------------------------------------------
Remote Fast Query Execution (cost=0.00..0.00 rows=0 width=0)
Node/s: dn001, dn002
-> Bitmap Heap Scan on t_mul_idx (cost=7617.08..7621.07 rows=2 width=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_idx WHERE f1=1 or f2=2 or f1=3 ;
QUERY PLAN
-----------------------------------------------------------------------------------------
Remote Fast Query Execution (cost=0.00..0.00 rows=0 width=0)
Node/s: dn001, dn002
-> Bitmap Heap Scan on t_mul_idx (cost=7619.51..7625.49 rows=3 width=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_idx WHERE f1=1 or f2=2 or f3=3 ;
QUERY PLAN
-----------------------------------------------------------------------
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 ;
QUERY PLAN
------------------------------------------------------------------------------------------
Remote Fast Query Execution (cost=0.00..0.00 rows=0 width=0)
Node/s: dn001
-> Index Only Scan using t_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。

单字段：

```
postgres=# CREATE TABLE t_simple_idx (f1 int,f2 int,f3 int,f4 int);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
Time: 25.926 ms
postgres=# CREATE INDEX t_simple_idx1 ON t_simple_idx(f1);
CREATE INDEX
Time: 31.568 ms
postgres=# CREATE INDEX t_simple_idx2 ON t_simple_idx(f2);
CREATE INDEX
Time: 26.315 ms
postgres=# CREATE INDEX t_simple_idx3 ON t_simple_idx(f3);
CREATE INDEX
Time: 23.997 ms
postgres=# INSERT INTO t_simple_idx SELECT t,t,t,t FROM generate_series(1,1000000) as t;
INSERT 0 1000000
Time: 4860.602 ms (00:04.861)
```

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
```

索引修改

修改索引名称

```
postgres=# ALTER INDEX t_simple_idx1 RENAME TO t_simple_idx1_new;
ALTER INDEX
```

重建索引

```
postgres=# REINDEX INDEX t_simple_idx1_new;
REINDEX
```

删除索引

```
postgres=# DROP INDEX t_appoint_id_idx;
DROP INDEX
```

视图操作

最近更新时间: 2024-10-17 17:10:00

视图修改

创建视图

```
DROP TABLE if exists t_range;
CREATE TABLE t_range(
f1 bigint,
f2 timestamp default now(),
f3 integer
) PARTITION BY range (f3) begin (1) step (50) partitions (3) WITh (orientation = 'column') DISTRIBUTE BY shard(f1) TO GROUP default_group;
INSERT INTO t_range(f1,f3) VALUES(1,1);
INSERT INTO t_range(f1,f3) VALUES(2,50);
INSERT INTO t_range(f1,f3) VALUES(2,110);
INSERT INTO t_range(f1,f3) VALUES(3,100);
postgres=# CREATE VIEW t_range_view as SELECT * FROM t_range;
CREATE VIEW
postgres=# SELECT * FROM t_range_view;
f1 | f2 | f3
----+----------------------------+-----
3 | 2021-01-19 20:22:14.392501 | 100
1 | 2021-01-19 20:22:13.585137 | 1
2 | 2021-01-19 20:22:13.685255 | 50
2 | 2021-01-19 20:22:13.702273 | 110
(4 rows)
```

数据类型重定义

```
postgres=# create or replace 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)
```

TDSQL-A PostgreSQL版 支持视图引用表或字段改名联动，不受影响：

```
DROP TABLE IF EXISTS t;
CREATE TABLE t(id int,mc text);
create view t_view as SELECT * FROM t;
postgres=# \d+ t_view
View "tdapg.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 to mc_new;
ALTER TABLE
Time: 22.081 ms
postgres=# \d+ t_view
View "tdapg.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;
```

视图删除

```
postgres=# DROP TABLE if exists t;
DROP TABLE
postgres=# drop view if exists t_view;
DROP VIEW
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

```
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)
```

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。

序列操作

最近更新时间: 2024-10-17 17:10:00

序列创建

建立序列

```
postgres=# CREATE SEQUENCE tdapg_seq;
CREATE SEQUENCE
```

建立序列，不存在时才创建：

```
postgres=# CREATE SEQUENCE IF NOT EXISTS tdapg_seq;
NOTICE: relation "tdapg_seq" already exists, skipping
CREATE SEQUENCE
```

查看序列当前的使用状况

```
postgres=# \x
Expanded display is on.
postgres=# SELECT * FROM tdapg_seq ;
-[ RECORD 1 ]-
last_value | 1
log_cnt | 0
is_called | f
```

获取序列的下一个值

```
postgres=# SELECT nextval('tdapg_seq');
-[ RECORD 1 ]
nextval | 1
```

获取序列的当前值

需要在访问 nextval() 后才能使用：

```
postgres=# SELECT currval('tdapg_seq');
-[ RECORD 1 ]
currval | 1
```

也可以下面的方式来获取序列当前使用到那一个值：

```
postgres=# SELECT last_value FROM tdapg_seq ;
-[ RECORD 1 ]-
last_value | 1
```

设置序列当前值

```
postgres=# SELECT setval('tdapg_seq',1);
-[ RECORD 1 ]
setval | 1
postgres=# \x
Expanded display is off.
```

序列使用

```
postgres=# CREATE TABLE t (id int, nickname text);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# INSERT INTO t (id,nickname) VALUES(nextval('tdapg_seq'),'tdapg好');
```

```
INSERT 0 1
postgres=# SELECT * FROM t;
id | nickname
----+----------
2 | tdapg好
(1 row)
```

序列作为字段的默认值使用

```
postgres=# ALTER TABLE t alter column id set default nextval('tdapg_seq');
postgres=# INSERT INTO t (nickname) VALUES('hello tdapg');
INSERT 0 1
postgres=# SELECT * FROM t;
id | nickname
----+--------------
3 | hello tdapg
2 | tdapg好
(2 rows)
```

序列作为字段类型使用

```
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 tdapg');
INSERT 0 1
postgres=# SELECT * FROM t;
id | nickname
----+--------------
1 | hello tdapg
(1 row)
```

序列删除

```
postgres=# DROP SEQUENCE tdapg_seq;
DROP SEQUENCE
```

删除序列，不存在时跳过：

```
postgres=# DROP SEQUENCE IF EXISTS tdapg_seq;
NOTICE: sequence "tdapg_seq" does not exist, skipping
DROP SEQUENCE
```

查询操作

最近更新时间: 2024-10-17 17:10:00

按某一列排序

```
postgres=# DROP TABLE if exists tdapg;
DROP TABLE
postgres=# CREATE TABLE tdapg (id int, nickname text);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# INSERT INTO tdapg (nickname) VALUES('tdapg数据库好');
INSERT 0 1
postgres=# INSERT INTO tdapg (id,nickname) VALUES(1,'tdapg数据库的时代来了');
INSERT 0 1
postgres=# INSERT INTO tdapg (id,nickname) VALUES(2,'hello tdapg ');
INSERT 0 1
postgres=# SELECT * FROM tdapg ORDER BY id;
id | nickname
----+-----------------------
1 | tdapg 数据库的时代来了
2 | hello tdapg
| tdapg 数据库好
(3 rows)
```

随机排序

```
postgres=# SELECT * FROM tdapg ORDER BY random();
id | nickname
----+-----------------------
1 | tdapg 数据库的时代来了
| tdapg 数据库好
2 | hello tdapg
(3 rows)
```

计算排序

```
postgres=# SELECT * FROM tdapg ORDER BY md5(nickname);
id | nickname
----+-----------------------
2 | hello tdapg
| tdapg数据库好
1 | tdapg数据库的时代来了
(3 rows)
```

子查询排序

```
postgres=# SELECT * FROM tdapg ORDER BY (SELECT id FROM tdapg ORDER BY random() limit 1);
id | nickname
----+-----------------------
1 | tdapg 数据库的时代来了
2 | hello tdapg
| tdapg 数据库好
(3 rows)
```

null 值排序结果处理

```
postgres=# INSERT INTO tdapg VALUES(4,null);
INSERT 0 1
```

null 值记录排在最前面：

```
postgres=# SELECT * FROM tdapg ORDER BY nickname nulls first;
id | nickname
----+-----------------------
4 |
2 | hello tdapg
1 | tdapg 数据库的时代来了
```

```
| tdapg 数据库好
(4 rows)
```

null 值记录排在最后：

```
postgres=# SELECT * FROM tdapg ORDER BY nickname nulls last;
id | nickname
----+----------------------
2 | hello tdapg
1 | tdapg 数据库的时代来了
| tdapg 数据库好
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，不符合使用习惯：

```
postgres=# SELECT * FROM (VALUES ('张三'), ('李四'),('陈五')) t(myname) ORDER BY convert(myname::bytea,'UTF-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,'GBK');
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.utf8";
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 tdapg WHERE id=1;
id | nickname
----+----------------------
1 | tdapg 数据库的时代来了
(1 row)
```

多条件 and

```
postgres=# SELECT * FROM tdapg WHERE id=2 and nickname like '%h%';
id | nickname
----+----------
1 | hello tdapg
(1 row)
```

多条件 or

```
postgres=# SELECT * FROM tdapg WHERE id=1 or nickname like '%h%';
id | nickname
----+------------------------
1 | tdapg 数据库的时代来了
2 | hello tdapg
(2 rows)
```

ilike 不区分大小写匹配

```
postgres=# CREATE TABLE t_ilike(id int,mc text);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# INSERT INTO t_ilike VALUES(1,'tdapg'),(2,'tdapg');
COPY 2
postgres=# SELECT * FROM t_ilike WHERE mc ilike '%tb%';
id | mc
----+-------
1 | tdapg
2 | tdapg
(2 rows)
```

WHERE 条件也能支持子查询

```
postgres=# SELECT * FROM tdapg WHERE id=(SELECT (random()*2)::integer FROM tdapg ORDER BY random() limit 1);
id | nickname
----+------------------------
1 | tdapg数据库的时代来了
(1 row)
postgres=# SELECT * FROM tdapg WHERE id=(SELECT (random()*2)::integer FROM tdapg ORDER BY random() limit 1);
id | nickname
----+----------
(0 rows)
```

null 值查询方法

```
postgres=# SELECT * FROM tdapg WHERE nickname is null;
id | nickname
----+----------
4 |
(1 row)
postgres=# SELECT * FROM tdapg WHERE nickname is not null;
id | nickname
----+------------------------
| tdapg 数据库好
1 | tdapg 数据库的时代来了
2 | hello tdapg
(3 rows)
```

exists 只要有记录返回就为真

```
postgres=# CREATE TABLE t_exists1(id int,mc text);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# INSERT INTO t_exists1 VALUES(1,'tdapg'),(2,'tdapg');
COPY 2
postgres=# CREATE TABLE t_exists2(id int,mc text);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# INSERT INTO t_exists2 VALUES(1,'tdapg'),(1,'tdapg');
COPY 2
postgres=# SELECT * FROM t_exists1 WHERE exists(SELECT 1 FROM t_exists2 WHERE t_exists1.id=t_exists2.id) ;
```

```
id | mc
----+-------
1 | tdapg
(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.id;
id | mc
----+-------
1 | tdapg
(1 row)
```

分页查询

默认从第一条开始，返回一条记录：

```
postgres=# SELECT * FROM tdapg limit 1;
id | nickname
----+---------------
| tdapg 数据库好
(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 tdapg limit 1 offset 0;
id | nickname
----+-------------
1 | hello tdapg
(1 row)
```

从第3条开始，返回二条记录：

```
postgres=# SELECT * FROM tdapg limit 1 offset 2;
id | nickname
----+--------------
2 | hello tdapg
(1 row)
```

ORDER BY 可以获得一个有序的结果：

```
postgres=# SELECT * FROM tdapg ORDER BY id limit 1 offset 2;
id | nickname
----+----------
4 |
(1 row)
```

合并多个查询结果

不过滤重复的记录

```
postgres=# create table u1(a int,b varchar)with(orientation=column);
CREATE TABLE
postgres=# create table u2(a int,b varchar)with(orientation=column);
CREATE TABLE
postgres=# insert into u1 values(1,'hi'),(2,'hello');
COPY 2
postgres=# insert into u2 values(1,'hi'),(2,'hello'),(3,'nihao');
COPY 3
postgres=# select * from u1 union all select * from u2 order by 1;
a | b
---+-------
1 | hi
1 | hi
```

```
2 | hello
2 | hello
3 | nihao
(5 rows)
```

过滤重复的记录

```
postgres=# select * from u1 union select * from u2 order by 1;
a | b
---+-------
1 | hi
2 | hello
3 | nihao
(3 rows)
```

每个子查询分布在合并结果中的使用

```
postgres=# SELECT * FROM ( SELECT * FROM tdapg limit 1) as t union all SELECT * FROM (SELECT * FROM t_appoint_col limit 1) as t ;
id | nickname
----+---------------
| tdapg 数据库好
2 | hello tdapg
(2 rows)
```

返回两个结果的交集

```
postgres=# CREATE TABLE t_intersect1(id int,mc text);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# INSERT INTO t_intersect1 VALUES(1,'tdapg'),(2,'tdapg');
COPY 2
postgres=# CREATE TABLE t_intersect2(id int,mc text);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# INSERT INTO t_intersect2 VALUES(1,'tdapg'),(3,'tdapg');
COPY 2
postgres=# SELECT * FROM t_intersect1 INTERSECT SELECT * FROM t_intersect2;
id | mc
----+-------
1 | tdapg
(1 row)
```

返回两个结果的差集

```
postgres=# CREATE TABLE t_except1(id int,mc text);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# INSERT INTO t_except1 VALUES(1,'tdapg'),(2,'tdapg');
COPY 2
postgres=# CREATE TABLE t_except2(id int,mc text);
NOTICE: Replica identity is needed for shard table, please add to this table through "alter table" command.
CREATE TABLE
postgres=# INSERT INTO t_except2 VALUES(1,'tdapg'),(3,'tdapg');
COPY 2
postgres=# SELECT * FROM t_except1 except SELECT * FROM t_except2;
id | mc
----+-------
2 | tdapg
(1 row)
```

多表关联

内连接

```
postgres=# SELECT * FROM tdapg inner join t_appoint_col on tdapg.id=t_appoint_col.id;
id | nickname | id | nickname
----+--------------+----+-------------
2 | hello tdapg | 2 | hello tdapg
(1 row)
```

左外连接

```
postgres=# SELECT * FROM tdapg left join t_appoint_col on tdapg.id=t_appoint_col.id;
id | nickname | id | nickname
----+----------------------+----+-------------
| tdapg 数据库好 | |
1 | tdapg 数据库的时代来了 | |
2 | hello tdapg | 2 | hello tdapg
4 | | |
(4 rows)
```

右外连接

```
postgres=# SELECT * FROM tdapg right join t_appoint_col on tdapg.id=t_appoint_col.id;
id | nickname | id | nickname
----+--------------+----+-------------
2 | hello tdapg | 2 | hello tdapg
(1 row)
```

全连接

```
postgres=# SELECT * FROM tdapg full join t_appoint_col on tdapg.id=t_appoint_col.id;
id | nickname | id | nickname
----+----------------------+----+-------------
| tdapg 数据库好 | |
1 | tdapg 数据库的时代来了 | |
2 | hello tdapg | 2 | hello tdapg
4 | | |
(4 rows)
```

事务控制

最近更新时间: 2024-10-17 17:10:00

事务（transaction）是访问并可能操作各种数据项的一个数据库操作序列，这些操作要么全部执行，要么全部不执行，是一个不可分割的工作单位。事务由事务开始与事务结束之间执行的全部数据库操作组成。

事务的存在包含了两个目的：

- 为数据库操作序列提供了一个从失败中恢复到正常状态的方法，同时提供数据库即使在异常状态下仍能保持一致性的方法。
- 当多个应用程序在并发访问数据库时，可以在这些应用程序之间提供一个隔离方法，以防止彼此的操作互相干扰。

事务特性

事务具有 ACID 特性，具体如下：

- 原子性(Atomicity)：事务中的全部操作在数据库中是不可分割的，要么全部完成，要么全部不执行。
- 一致性(Consistency)：几个并行执行的事务，其执行结果必须与按某一顺序串行执行的结果相一致。
- 隔离性(Isolation)：事务的执行不受其他事务的干扰，事务执行的中间结果对其他事务必须是透明的。
- 持久性(Durability)：对于任意已提交事务，系统必须保证该事务对数据库的改变不被丢失，即使数据库出现故障。

事务的 ACID 特性是由关系数据库系统（DBMS）来实现的，DBMS 采用日志来保证事务的原子性、一致性和持久性。日志记录了事务对数据库所作的更新，如果某个事务在执行过程中发生错误，就可以根据日志撤销事务对数据库已做的更新，使得数据库回滚到执行事务前的初始状态。

对于事务的隔离性，DBMS 是采用锁机制来实现的。当多个事务同时更新数据库中相同的数据时，只允许持有锁的事务能更新该数据，其他事务必须等待，直到前一个事务释放了锁，其他事务才有机会更新该数据。

事务隔离

SQL 标准定义了四种隔离级别。最严格的是可序列化，在标准中用了一整段来定义它，其中说到一组可序列化事务的任意并发执行被保证效果和以某种顺序一个一个执行这些事务一样。其他三种级别使用并发事务之间交互产生的现象来定义，每一个级别中都要求必须不出现一种现象。注意由于可序列化的定义，在该级别上这些现象都不可能发生。

在各个级别上被禁止出现的现象是：

脏读

一个事务读取了另一个并行未提交事务写入的数据。

不可重复读

一个事务重新读取之前读取过的数据，发现该数据已经被另一个事务（在初始读之后提交）修改。

幻读

一个事务重新执行一个返回符合一个搜索条件的行集合的查询，发现满足条件的行集合因为另一个最近提交的事务而发生了改变。

序列化异常

成功提交一组事务的结果与这些事务所有可能的串行执行结果都不一致。

隔离级别	脏读	不可重复读	幻读	序列化异常
读未提交	允许	可能	可能	可能
读已提交	不可能	可能	可能	可能
可重复读	不可能	不可能	允许	可能
可序列化	不可能	不可能	不可能	不可能

声明事务时，隔离级别可以声明为四种标准事务隔离级别中的任意一种，目前 TDSQL-A PostgreSQL版 读未提交和读已提交模式相同，实现了读已提交和可重复读。

事务控制

启动事务

启动事务可以使用 START TRANSACTION 或者 BEGIN 语法，可以在启动事务的同时，声明该事务的隔离级别、读写模式。

示例：

postgres=# START TRANSACTION;

START TRANSACTION

或者：

postgres=# BEGIN;

BEGIN

或者：

postgres=# START TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION

或者：

postgres=# BEGIN WORK ISOLATION LEVEL READ COMMITTED;

BEGIN

提交事务

进程#1访问：

postgres=# BEGIN;

BEGIN

postgres=# DELETE FROM tdapg WHERE id=5;

DELETE 1

postgres=#

postgres=# SELECT * FROM tdapg ORDER BY id;

id | nickname

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

1 | hello tdapg

2 | tdapg好

3 | tdapg好

4 | tdapg default

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

进程#2访问：

postgres=# SELECT * FROM tdapg ORDER BY id;

id | nickname

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

1 | hello tdapg

2 | tdapg好

3 | tdapg好

4 | tdapg default

5 | tdapg swap

(5 rows)

进程#1提交数据：

postgres=# COMMIT;

COMMIT

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“读已提交”：

postgres=# SELECT * FROM tdapg ORDER BY id;

id | nickname

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

1 | hello tdapg

2 | tdapg好

3 | tdapg好

4 | tdapg default

(4 rows)

回滚事务

postgres=# BEGIN;

BEGIN

postgres=# DELETE FROM tdapg WHERE id IN (3,4);

DELETE 2

postgres=# SELECT * FROM tdapg;

id | nickname

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

1 | hello tdapg

2 | tdapg好

(2 rows)

postgres=# ROLLBACK;

ROLLBACK

ROLLBACK 后数据又恢复回事务开始前的状态：

postgres=# SELECT * FROM tdapg;

id | nickname

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

1 | hello tdapg

2 | tdapg好

3 | tdapg好

4 | tdapg default

(4 rows)

锁管理

最近更新时间: 2024-10-17 17:10:00

LOCK table 可获取表级锁，该语句只能在事务块中使用，没有 UNLOCK table 语句，事务结束时锁将自动释放。如果使用LOCK 命令对表进行加锁，缺省为最严格的模式 ACCESS EXCLUSIVE。

```
v3=# begin;
BEGIN
v3=# lock table t10;
LOCK TABLE
v3=# select relation::regclass,mode from pg_locks;
relation | mode
----------+---------------------
t10 | AccessExclusiveLock
```

表级锁

ACCESS SHARE

SELECT 命令在被引用的表上获得一个这种模式的锁。通常，任何只读取表而不修改它的查询都将获得这种锁模式。

ROW SHARE

SELECT FOR UPDATE 和 SELECT FOR SHARE 命令在目标表上取得一个这种模式的锁（加上在被引用但没有选择 FOR UPDATE/FOR SHARE 的任何其他表上的 ACCESS SHARE 锁）。

ROW EXCLUSIVE

命令 UPDATE、DELETE 和 INSERT 在目标表上取得这种锁模式（加上在任何其他被引用表上的 ACCESS SHARE 锁）。通常，这种锁模式将被任何修改表中数据的命令取得。

SHARE UPDATE EXCLUSIVE

由 VACUUM（不带 FULL）、ANALYZE、CREATE INDEX CONCURRENTLY、CREATE STATISTICS 和 ALTER TABLE VALIDATE 以及其他 ALTER TABLE 的变体获得。

SHARE

由 CREATE INDEX（不带 CONCURRENTLY）取得。

SHARE ROW EXCLUSIVE

由 CREATE COLLATION、CREATE TRIGGER 和很多 ALTER TABLE 的很多形式所获得。

EXCLUSIVE

由 REFRESH MATERIALIZED VIEW CONCURRENTLY 获得。

ACCESS EXCLUSIVE

由 ALTER TABLE、DROP TABLE、TRUNCATE、REINDEX、CLUSTER、VACUUM FULL 和 REFRESH MATERIALIZED VIEW（不带 CONCURRENTLY）命令获取。ALTER TABLE 的很多形式也在这个层面上获得锁（见 ALTER TABLE）。这也是未显式指定模式的 LOCK TABLE 命令的默认锁模式。

冲突的锁模式

锁之间会互相阻塞、冲突。

请求的锁模式	当前的锁模式	当前的锁模式	当前的锁模式	当前的锁模式	当前的锁模式	当前的锁模式	当前的锁模式	当前的锁模式
-	ACCESSSHARE	ROWSHARE	ROWEXCLUSIVE	SHAREUPDATE EXCLUSIVE	SHARE	SHAREROW EXCLUSIVE	EXCLUSIVE	ACCESSEXCLUSIV
ACCESS SHARE	-	-	-	-	-	-	-	X
ROW SHARE	-	-	-	-	-	-	X	X
ROW EXCLUSIVE	-	-	-	-	X	X	X	X

请求的锁模式	当前的锁模式	当前的锁模式	当前的锁模式	当前的锁模式	当前的锁模式	当前的锁模式	当前的锁模式	当前的锁模式
SHARE UPDATE EXCLUSIVE	-	-	-	X	X	X	X	X
SHARE	-	-	X	X	-	X	X	X
SHARE ROW EXCLUSIVE	-	-	X	X	X	X	X	X
EXCLUSIVE	-	X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

除了表级锁以外，还有行级锁，行级锁不影响数据查询，它们只阻塞对同一行的写入者和加锁者。

死锁

TDSQL-A PostgreSQL版 中，并发执行的事务由于竞争同一个资源会导致死锁。要检查在一个数据库中所有被锁的对象，可以查看 pg_locks 系统视图。

显式锁定的使用可能会增加死锁的可能性，死锁是指两个（或多个）事务相互持有对方想要的锁。可以手动或者以其他方式结束其中一个事务来解决死锁的问题。

查询当前存在的锁：

```
select pid,relation::regclass,mode from pg_locks where relation<>'pg_locks'::regclass::oid;
pid | relation | mode
-----------+------------+-----------------
27577 | t4 | AccessShareLock
```

可通过系统视图 pg_locks 的 pid 字段和系统视图 pg_stat_activity 关联查询出当前的持有锁的事务信息。

用户自定义函数

最近更新时间: 2024-10-17 17:10:00

用户可以通过语句 CREATE FUNCTION 定义一个新函数。CREATE OR REPLACE FUNCTION 将创建一个新函数或者替换一个现有的函数。要定义一个函数，用户必须具有该语言上的 USAGE 权限。新函数的名称不能匹配同一个模式中具有相同输入参数类型的任何现有函数，支持重载。

PL/pgSQL 语言函数

PL/pgSQL 类似于 Oracle 的 PL/SQL，是一种可载入的过程语言。

用 PL/pgSQL 创建的函数可以被用在任何可以使用内建函数的地方。SQL 被大多数数据库用作查询语言。它是可移植的并且容易学习。但是每一个 SQL 语句必须由数据库服务器单独执行。这意味着客户端应用必须发送每一个查询到数据库服务器、等待它被处理、接收并处理结果、做一些计算，然后发送更多查询给服务器。如果客户端和数据库服务器不在同一台机器上，所有这些会引起进程间通信并且将带来网络负担。

通过 PL/pgSQL，可以将一整块计算和一系列查询分组在数据库服务器内部，这样就有了一种过程语言的能力，并且使 SQL 更易用，同时能节省的客户端/服务器通信开销。

- 客户端和服务器之间的额外往返通信被消除。
- 客户端不需要的中间结果不必被整理或者在服务器和客户端之间传送。
- 多轮的查询解析可以被避免。

PL/pgSQL 可以使用 SQL 中所有的数据类型、操作符和函数。其应用方法与存储过程相似，只是存储过程无返回值，函数有返回值。

示例：

```
CREATE FUNCTION one() RETURNS integer AS $$
SELECT 1 AS result;
$$ LANGUAGE SQL;
直接 select 进行调用
SELECT one();
```

C/C++ 语言函数

用户定义的函数可以用 C\C++ 编写， 这类函数被编译成共享库（即 so 库文件），在创建之前上传到服务器上。动态载入是把“C 语言”函数和 “内部”函数区分开的特性，两者真正的编码习惯实际上是一样的。

编写代码

编写和编译 C 函数的基本规则如下：

在分配内存时，使用函数 palloc 和 pfree，而不是使用对应的 C 库函数 malloc 和 free。在每个事务结束时会自动释放通过palloc 分配的内存，以免内存泄露。

大部分的内部 TDSQL-A PostgreSQL版 的类型都声明在 postgres.h 中，不过函数管理器接口在 fmgr.h中，为了移植性，最好在其他系统或者用户头文件之前，先包括 postgres.h 和 fmgr.h。

创建的对象文件中定义的符号名不能和原库中的定义的符号名、操作系统的可执行程序中定义的符号互相冲突。

如：

```
extern "C"{
#include "postgres.h"
#include "fmgr.h"

PG_MODULE_MAGIC;
}
//加法
extern "C"{
PG_FUNCTION_INFO_V1(my_add_func);
Datum my_add_func(PG_FUNCTION_ARGS)
{
int32 a = PG_GETARG_INT32(0);
int32 b = PG_GETARG_INT32(1);
```

```
int64 result = a + b;
PG_RETURN_INT64(result);
}
}
```

编译和载入自定义函数

1、编译。

```
gcc -Wall -Werror -g3 -o my_func.o -fPIC -c my_func.cpp -I/data/tbase/user_1/tdata_02/tbase_v3_1/3.0.16/install/tbase_pgxz/include/postgresql/server/
```

2、生成 so 库文件。

```
gcc -shared -o my_func.so my_func.o
```

3、将库文件发送到每个 DN、CN 节点上，并赋予数据库用户角色权限，进行函数创建。

```
create function my_add_func(a integer,b integer) RETURNS integer
as '/data/tbase/my_func.so' ,'my_add_func' language c strict;
CREATE FUNCTION
进行函数查询
v3=# select my_add_func(1,2);
my_add_func
\-------------
3
(1 row)
```

PL/Python 函数

在 TDSQL-A PostgreSQL版 使用 PL/Python 函数之前需要提前的创建拓展 plpythonu：

```
create extension plpythonu;
```

语法声明

PL/Python 中的函数通过标准的 CREATE FUNCTION 语法声明：

```
CREATE FUNCTION funcname (argument-list)
RETURNS return-type
AS $$
\# PL/Python 函数体
$$ LANGUAGE plpythonu;
```

函数体就是一个 Python 脚本。当函数被调用时，它的参数被当做列表 args 的元素传递，命名参数也被作为普通变量传递给 Python 脚本。使用命名参数通常可读性更好。Python 代码会以通常的方式返回结果，即使用 return 或者 yield（在结果集合语句的情况中）。如果没有提供一个返回值，Python 会返回默认的None。PL/Python 会把 Python 的 None 翻译成 SQL 空值。

例如，一个返回两个整数中较大的整数的函数可以定义为：

```
CREATE FUNCTION pymax (a integer, b integer)
RETURNS integer
AS $$
if a > b:
return a
return b
$$ LANGUAGE plpythonu;
```

作为该函数定义给出的 Python 代码会被转换成一个 Python 函数。例如上面的代码会得到：

```
def __plpython_procedure_pymax_23456():
if a > b:
return a
return b
```

变量范围

参数被设置为全局变量。由于 Python 的可见范围规则，这会导致一种后果：在函数内不能把一个参数变量重新赋予给一个涉及该变量名称本身的表达式的值，除非在该代码块中重新声明该变量为全局的。例如，下面的代码无法工作：

```
CREATE FUNCTION pystrip(x text)
RETURNS text
AS $$
x = x.strip() # 错误
return x
$$ LANGUAGE plpythonu;
```

因为对 x 的赋值让 x 成为了整个代码块的一个局部变量，并且因此该赋值操作右边的 x 引用的是一个还未赋值的局部变量x，而不是 PL/Python 函数的参数。通过使用 global 语句，可以让上面的代码正常工作：

```
CREATE FUNCTION pystrip(x text)
RETURNS text
AS $$
global x
x = x.strip() # 现在好了
return x
$$ LANGUAGE plpythonu;
```

但是不建议依赖于这类 PL/Python 的实现细节，最好把函数参数当作是只读。

插件管理

最近更新时间: 2024-10-17 17:10:00

TDSQL-A PostgreSQL版 支持多种模块，开放模块扩展接口。用户可以使用语法 CREATE EXTENSION 将一个新加的扩展载入到数据库中。不能有同名的拓展被载入。

载入一个扩展本质上是运行该扩展的脚本文件。该脚本通常将创建新的 SQL 对象，例如函数、数据类型、操作符以及索引支持方法。

CREATE EXTENSION 会额外地记录所有被创建对象的标识，发出 DROP EXTENSION 时可以删除它们。在使用 CREATE EXTENSION 载入扩展到数据库之前，必须先安装好该扩展的支持文件。

当前可以用于载入的扩展，可以在系统视图 pg_available_extension_versions 中看到。

当前已经部署的拓展，可以通过系统表 pg_extension 查看，也可以通过 psql 的元命令 \dx 查看。

请勿删除系统默认已经存在的拓展，避免发生不可预知的集群故障。

插件管理

部分集群默认已有的插件如下：

- pageinspect：pageinspect 模块提供函数让用户从低层次观察数据库页面的内容，这对于调试目的很有用。
- pg_check：检查 catalog 一致性的工具。
- pg_clean：清理残留的二阶段事务提交的工具。
- pg_errcode_stat：跟踪集群进程的一场的插件。
- 可访问函数 pg_errcode_stat 查看当前集群异常，通过 pg_errcode_stat_reset 进行清理。
- pg_squeeze：回收部分无效空间资源的插件，相较于 vacuum full 不会有太长时间的锁表情况，导致无法进行读写。
- pg_stat_error：跟踪集群进程的一场的插件。
- pg_stat_log：跟踪所有 SQL 的执行信息。
- 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。
- pg_unlock：用于检测和解除死锁的工具。
- plpgsql：PL/pgSQL 函数语言插件。
- Plpythonu：PL/python 函数语言插件。

查看插件

```
postgres=# SELECT e.extname AS "Name", e.extversion AS "Version", n.nspname AS "Schema", c.description
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 tdapg;
CREATE EXTENSION
```

上面的语句把"uuid-ossp"创建到模式 tdapg 下。

删除插件

```
postgres=# drop extension "uuid-ossp" ;
DROP EXTENSION
```

插件 pg_trgm 使用

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-A PostgreSQL版 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-A PostgreSQL版 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。

创建插件：

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

创建需要进行全文检索的表：

```
postgres=# create table t_trgm (id int,trgm text,no_trgm text);
CREATE TABLE
```

插入测试数据：

```
postgres=# insert into t_trgm select t,md5(t::text),md5(t::text) from generate_series(1,1000000) as t;
INSERT 0 1000000
```

创建 trgm 索引：

```
postgres=# create index t_trgm_trgm_idx on t_trgm using gist(trgm gist_trgm_ops);
CREATE INDEX
```

数据导入导出

最近更新时间: 2024-10-17 17:10:00

数据导入

INSERT 直接写入数据

通过 INSERT 语句直接向 TDSQL-A PostgreSQL版 的表中写入数据，使用 psql 客户端工具连接数据库后，INSERT INTO ... VALUES 或者 INSERT INTO...SELECT 写入数据，TDSQL-A PostgreSQL版 也支持 JDBC、ODBC、Python、CAPI、ADO.net 等接口方式连接数据库，来执行 INSERT 语句向 TDSQL-A PostgreSQL 版 表中写入数据。

COPY FROM 导入数据

除 INSERT 外，TDSQL-A PostgreSQL版 支持通过 COPY 命令，在文件和表之间复制数据，其语法格式如下：

```
COPY table_name [ ( column_name [, ...] ) ]
FROM { 'filename' | PROGRAM 'command' | STDIN }
[ [ WITH ] ( option [, ...] ) ]
COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
TO { 'filename' | PROGRAM 'command' | STDOUT }
[ [ WITH ] ( option [, ...] ) ]
```

其中 option 可以是下列之一：

```
FORMAT format_name
OIDS [ boolean ]
FREEZE [ boolean ]
DELIMITER 'delimiter_character'
NULL 'null_string'
HEADER [ boolean ]
QUOTE 'quote_character'
ESCAPE 'escape_character'
FORCE_QUOTE { ( column_name [, ...] ) | * }
FORCE_NOT_NULL ( column_name [, ...] )
FORCE_NULL ( column_name [, ...] )
ENCODING 'encoding_name'
```

带一个文件名的 COPY 指示 TDSQL-A PostgreSQL版 服务器直接从一个文件读取或者写入到一个文件。该文件必须是运行TDSQL-A PostgreSQL版 服务器的用户可访问的并且应该以服务器的视角来指定其名称。

当指定了 PROGRAM 时，服务器执行给定的命令，并且从该程序的标准输出读取或者写入到该程序的标准输入。该程序必须以服务器的视角指定，并且必须是 tdapg 用户可执行的。

在指定 STDIN 或者 STDOUT 时，数据会通过客户端和服务器之间的连接传输。

另外，\COPY FROM/TO，用于将本地文件复制到数据表中或将数据表中数据复制到本地文件中。

多种数据源加载

- STDIN 标准输入：

```
postgres=# COPY tdapg (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 tdapg
>> 2 \N
>> 3 pgxc
>> \.
postgres=# select * from tdapg;
id | mc
----+-------
1 | tdapg
2 |
3 | pgxc
(3 rows)
```

- SFTP 数据源：

准备环境：安装 sftp 服务，将数据文件上传进 sftp 服务器。

```
postgres=# COPY copy_hdfs FROM PROGRAM 'hadoop fs -cat hdfs://10.183.208.190:9000/lineitem' DELIMITER '|';
COPY 797000000
```

- HTTP 数据源：

准备环境：安装 httpd 服务，并将数据文件上传到 HTTP 服务器。

```
postgres =# \COPY copy_http FROM PROGRAM 'curl -s http://imgcache.finance.cloud.tencent.com:8010.183.208.191:8081/lineitem.csv' WITH csv DELIMIT
ER ',';
COPY 797000000
```

- HDFS 数据源：

准备环境：安装 hadoop 服务并配置，将数据文件上传到 hadoop 服务上。

```
postgres =# COPY copy_hdfs FROM PROGRAM 'hadoop fs -cat hdfs://10.183.208.190:9000/lineitem' DELIMITER '|';
COPY 797000000
```

多种数据格式加载

COPY 支持 CSV，TXT，GZIP，SNAPPY 等多种数据文件格式导入。

- CSV 文件导入：

```
postgres =# \COPY copy_csv FROM '/data12/copy_test_data/lineitem.csv' WITH csv DELIMITER ',';
COPY 797000000
```

- TXT 文件导入：

```
postgres =# \COPY copy_csv FROM '/data12/copy_test_data/lineitem.txt' WITH (FORMAT 'text', DELIMITER '|');
COPY 797000000
```

- SNAPPY 文件导入：

```
postgres=# \COPY copy_snappy FROM PROGRAM 'snzip -dc /data12/copy_test_data/lineitem.snappy' DELIMITER '|';
COPY 797000000
```

- GZIP 文件导入：

```
postgrs=# \COPY copy_gzip FROM PROGRAM 'tar -zxvf /data12/copy_test_data/lineitem.tar.gz | xargs cat' DELIMITER '|';
COPY 797000000
```

- LZO 文件导入：

```
postgres=# \COPY copy_lzop FROM PROGRAM 'lzop -cd /data12/copy_test_data/lineitem.lzo | more' DELIMITER '|';
COPY 797000000
```

数据导出

使用 COPY TO 或者 \COPY TO 将表数据复制到文件：

```
COPY copyperf_txt TO '/data12/copy_test_data/cpto/cpto1.dat' WITH(FORMAT csv, DELIMITER ',');
```

将表中数据导入到多个数据文件：

```
COPY copyperf_txt TO PROGRAM 'cd /data12/copy_test_data/cpto && split -l 39850000 -d -a 4 && ls | grep ^x | xargs -n1 -i{} mv {} perfcopyto_{}.csv' WIT
H csv DELIMITER ',';
```

应用程序开发
基于 JDBC 开发

最近更新时间: 2024-10-17 17:10:00

JDBC 驱动包

可前往 官网 获取，或者向软件提供商获取。

驱动类

在代码中创建数据库连接之前，需要加载数据库驱动类“org.postgresql.Driver”。

连接参数

使用 JDBC 连接数据库，由 URL 指定具体连接的数据库，可采用以下的几种形式：

```
jdbc：postgresql:database
jdbc：postgresql://host/database
jdbc：postgresql://host:port/database
```

参数含义如下：

- host：服务器的主机名。默认为 localhost，要指定 IPv6 地址，您必须将 host 参数括在方括号中，例如 Jdbc:postgresql:// 127.0.0.1:5740/accounting。
- port：服务器正在监听的端口号。
- database：数据库名称。

  除标准连接参数外，驱动程序还支持许多其他属性，这些属性可用于指定特定于其他驱动程序行为。这些属性可以在连接 URL 或附加 Properties 对象参数中指定 DriverManager.getConnection。
- user：string 类型，表示创建连接的数据库用户。
- password：string 类型，表示数据库用户的密码。
- ssl：Boolean 类型，表示是否使用 SSL 连接。
- loggerLevel：string 类型，为 LogStream 或 LogWriter 设置记录进 DriverManager 当前值的日志信息量。目前支持 OFF、DEBUG 和 TRACE。
  - 值为 DEBUG 时，表示只打印 DEBUG 级别以上的日志，将记录非常少的信息。
  - 值等于 TRACE 时，表示打印 DEBUG 和 TRACE 级别的日志，将产生详细的日志信息。默认值为 OFF，表示不打印日志。
- prepareThreshold：integer类型，用于确定在转换为服务器端的预备语句之前，要求执行方法 PreparedStatement 的次数。缺省值是5。
- batchMode：boolean 类型，用于确定是否使用 batch 模式连接。
- Fetchsize：integer 类型，用于设置数据库链接所创建 statement 的默认 fetchsize。
- ApplicationName：string 类型，应用名称，在不做设置时，缺省值为 PostgreSQL JDBC Driver。
- allowReadOnly：boolean 类型，用于设置 connection 是否允许设置 readonly 模式，默认为 false，若该参数不被设置为 true，则执行 connnection.setReadOnly 不生效。

- blobMode：string 类型，用于设置 setBinaryStream 方法为不同的数据类型赋值，设置为 on 时表示为 blob 数据类型赋值，设置为 off 时表示为 bytea 数据类型赋值，默认为 on。
- connectionExtraInfo：boolean 类型，表示驱动是否上报当前驱动的部署路径、进程属主用户到数据库。

示例：

```
public static Connection GetConnection(String username, String passwd)
{
//驱动类
String driver = "org.postgresql.Driver";
//数据库连接描述符
String sourceURL = "jdbc:postgresql://10.10.0.13:11345/tdapg";
Connection conn = null;
try
{
//加载驱动
Class.forName(driver);
}
catch( Exception e )
{
e.printStackTrace();
return null;
}
try
{
//创建连接
conn = DriverManager.getConnection(sourceURL, username, passwd);
System.out.println("Connection succeed!");
}
catch(Exception e)
{
e.printStackTrace();
return null;
}
return conn;
};
```

示例代码

连接、建表、batch 插入、更新等 Java 连接 demo。

```
package Demo;
import java.sql.*;
public class tdapg_Conn {
public static Connection GetConnection(String username, String passwd) {
String driver = "org.postgresql.Driver";
String sourceURL = "jdbc:postgresql://100.1.1.1:11345/v3";
Connection conn = null;
try {
Class.forName(driver).newInstance();
} catch (Exception e) {
e.printStackTrace();
return null;
}
try {
conn = DriverManager.getConnection(sourceURL, username, passwd);
System.out.println("Connection succeed!");
} catch (Exception e) {
e.printStackTrace();
return null;
}
return conn;
};
public static void CreateTable(Connection conn) {
Statement stmt = null;
try {
```

```
stmt = conn.createStatement();
int rc = stmt.executeUpdate("CREATE TABLE user_table(user_id INTEGER, user_name VARCHAR(32));");
stmt.close();
System.out.println("TABLE customer_t1 create Successful!");
} catch (SQLException e) {
if (stmt != null) {
try {
stmt.close();
} catch (SQLException e1) {
e1.printStackTrace();
}
}
e.printStackTrace();
}
}


public static void BatchInsertData(Connection conn) {
PreparedStatement pst = null;
try {
pst = conn.prepareStatement("INSERT INTO user_table VALUES (?,?)");
for (int i = 0; i < 3; i++) {
pst.setInt(1, i);
pst.setString(2, "data " + i);
pst.addBatch();
}
pst.executeBatch();
pst.close();
System.out.println("Insert succeed!");


} catch (SQLException e) {
if (pst != null) {
try {
pst.close();
} catch (SQLException e1) {
e1.printStackTrace();
}
}
e.printStackTrace();
}
}
public static void ExecPreparedSQL(Connection conn) {
PreparedStatement pstmt = null;
try {
pstmt = conn.prepareStatement("UPDATE user_table SET user_name = ? WHERE user_id = 1");
pstmt.setString(1, "new Data");
int rowcount = pstmt.executeUpdate();
pstmt.close();
System.out.println("Update succeed!");
} catch (SQLException e) {
if (pstmt != null) {
try {
pstmt.close();


} catch (SQLException e1) {
e1.printStackTrace();
}
}
e.printStackTrace();
}
}


public static void main(String[] args) {
Connection conn = GetConnection("dbadmin", "tdapg@2021");
CreateTable(conn);
BatchInsertData(conn);
ExecPreparedSQL(conn);
try {
conn.close();
```

```
} catch (SQLException e) {
e.printStackTrace();
}
}
}
```

执行结果：

基于 ODBC 开发

最近更新时间: 2024-10-17 17:10:00

使用 ODBC 连接 TDSQL-A PostgreSQL版 进行数据库操作需要提前部署 ODBC 的驱动包，驱动包可前往 官网 获取，或者向软件提供商获取。

Linux 下进行 ODBC 开发

Linux 环境下，开发应用程序要用到 unixODBC 提供的头文件（sql.h、sqlext.h 等）和库 libodbc.so。这些头文件和库可从 ODBC 的安装包中获得。

Linux 部署 ODBC：

可使用 yum 命令直接进行 ODBC 和 unixODBC 的部署。

```
yum install -y unixODBC
yum install -y postgresql-odbc
```

部署完成后执行 isql 表明 ODBC 部署完成。

Linux ODBC 配置：

1. 创建 /etc/odbc.ini 文件，修改其中的 Servername、UserName、Password、Port 等参数。

```
vi /etc/odbc.ini
[tdapg]
Description = Test to tdapg
Driver = PostgreSQL
Database = postgres
Servername = localhost
UserName = dbadmin
Password = tdapg@tdapg
Port = 11345
ReadOnly = 0
ConnSettings = set client_encoding to UTF8
```

完成后保存退出。

2. 执行 isql -v tdapg(数据源名称) 命令。

   如果显示如下信息，表明配置正确，连接成功。

```
+---------------------------------------+
| Connected! |
| |
| sql-statement |
| help [tablename] |
| quit |
| |
+---------------------------------------+
SQL>
```

若显示 ERROR 信息，则表明配置错误，请检查上述配置是否正确。

Windows 下进行 ODBC 开发

Windows 环境下，开发应用程序用到的相关头文件和库文件都由系统自带。

Windows 配置数据源：

1. 解压 psqlodbc_13_00_0000-x64 .zip 获取 psqlodbc_x64.msi 驱动直接进行部署。

2. 打开驱动管理器，使用快捷键 Win+R 直接运行 odbcad32。

3. 配置数据源，在打开的驱动管理器上，选择【用户 DSN】>【添加】 > 【PostgreSQL Unicode】，然后进行配置。

4. 单击【Test】进行连接测试，测试成功后单击【Save】保存连接信息，完成数据源的配置，即可进行后续开发。

基于 libpq 开发

最近更新时间: 2024-10-17 17:10:00

libpq 是应用程序员使用 TDSQL-A PostgreSQL版 的 C 接口。libpq 是一个库函数的集合，它们允许客户端程序传递查询给TDSQL-A PostgreSQL版 后端服务器并且接收这些查询的结果。

libpq 也是很多其他 TDSQL-A PostgreSQL版 应用接口的底层引擎，包括为 C++、Perl、Python、Tcl 和 ECPG 编写的接口。

使用 libpq 的客户端程序必须包括头文件 libpq-fe.h，并必须与 libpq 库连接在一起。

示例代码

示例：进行数据库连接

1. 创建 conn_test.cpp 文件。

```
#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;
}
```

2. 进行编译。

```
gcc -c -I ~/user_1/tdata_02/tbase_v3_1/3.0.16/install/tbase_pgxz/include conn_test.cpp
```

3. 连接成可执行文件。

```
gcc -o conn conn_test.o -L ~/user_1/tdata_02/tbase_v3_1/3.0.16/install/tbase_pgxz/lib/ -lpq
```

4. 进行数据库连接测试。

```
./conn "host=localhost dbname=postgres port=11345"
```

连接数据库成功！

示例：创建表

1. 创建文件 createtable.c。

```
#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 tdapg(id int,nickname text) ";
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;
}
```

2. 进行编译。

```
gcc -c -I ~/user_1/tdata_02/tbase_v3_1/3.0.16/install/tbase_pgxz/include createtable .cpp
```

3. 连接成可执行文件。

```
gcc -o createtable createtable.o -L ~/user_1/tdata_02/tbase_v3_1/3.0.16/install/tbase_pgxz/lib/ -lpq
```

4. 进行数据库连接测试。

```
./createtable "host=localhost dbname=postgres port=11345"
```

示例：数据插入

1. 创建文件 insert.c。

```
#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 tdapg (id,nickname) values(1,'tdapg'),(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;
}
```

2. 进行编译。

```
gcc -c -I ~/user_1/tdata_02/tbase_v3_1/3.0.16/install/tbase_pgxz/include insert.cpp
```

3. 连接成可执行文件。

```
gcc -o insert insert.o -L ~/user_1/tdata_02/tbase_v3_1/3.0.16/install/tbase_pgxz/lib/ -lpq
```

4. 进行数据库连接测试。

```
./insert "host=localhost dbname=postgres port=11345"
```

示例：进行数据查询

1. 创建文件 select.c。

```
#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 tdapg";
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;
}
```

2. 进行编译。

```
gcc -std=c99 -c -I ~/user_1/tdata_02/tbase_v3_1/3.0.16/install/tbase_pgxz/include select.c
```

3. 连接成可执行文件。

```
gcc -o select select.o -L ~/user_1/tdata_02/tbase_v3_1/3.0.16/install/tbase_pgxz/lib/ -lpq
```

4. 进行数据库连接测试。

```
./select"host=localhost dbname=postgres port=11345"
```

基于 Python 开发

最近更新时间: 2024-10-17 17:10:00

Psycopg 是常用于 Python 编程语言的 PostgreSQL 数据库适配器，同样也可以用来连接TDSQL-A PostgreSQL版 进行数据库操作。

Psycopg2 需要提前进行部署，可使用 pip install psycopg2 命令进行部署。

示例使用的均是3.6的 Python 版本，若使用 python2.x 版本需进行代码兼容修改。

示例1：数据库连接

```
conn.py
#coding=utf-8
#!/usr/bin/python
import psycopg2
try:
conn = psycopg2.connect(database="v3", user="dbadmin", password="tdapg@tdapg", host="100.1.1.1", port="11345")
print ("连接数据库成功")
conn.close()
except psycopg2.Error as msg:
print ("连接数据库出错，错误详细信息： %s" %(msg.args[0]))
```

示例2：表创建

```
#coding=utf-8
#!/usr/bin/python
import psycopg2
try:
conn = psycopg2.connect(database="v3", user="dbadmin", password="tdapg@tdapg", host="100.1.1.1", port="11345")
print ("连接数据库成功")
cur = conn.cursor()
sql = """
create table tdapg
(
id int,
nickname varchar(100)
) """
cur.execute(sql)
conn.commit()
print ("建立数据表成功")
conn.close()
except psycopg2.Error as msg:
print ("Tdapg Error %s" %(msg.args[0]))
```

示例3：数据插入

```
#coding=utf-8
#!/usr/bin/python
import psycopg2
try:
conn = psycopg2.connect(database="v3", user="dbadmin", password="tdapg@Tdapg", host="100.1.1.1", port="11345")
print ("连接数据库成功")
cur = conn.cursor()
sql = "insert into tdapg values(1,'tdapg'),(2,'tdapg');"
cur.execute(sql)
sql = "insert into tdapg values(%s,%s)"
```

```
cur.execute(sql,(3,'pg'))
conn.commit()
print ("插入数据成功")
conn.close()
except psycopg2.Error as msg:
print ("操作数据库出库 %s" %(msg.args[0]))
```

示例4：数据查询

```
#coding=utf-8
#!/usr/bin/python
import psycopg2
try:
conn = psycopg2.connect(database="v3", user="dbadmin", password="tdapg@Tdapg", host="100.1.1.1", port="11345")
print ("连接数据库成功")
cur = conn.cursor()
sql = "select * from tdapg"
cur.execute(sql)
rows = cur.fetchall()
for row in rows:
print ("ID = %s NICKNAME = %s " %(row[0],row[1]))
conn.close()
except psycopg2.Error as msg:
print ("操作数据库出库 %s" %(msg.args[0]))
```

示例5：copy 数据插入

```
#coding=utf-8
#!/usr/bin/python
import psycopg2
try:
conn = psycopg2.connect(database="postgres", user="dbadmin",
password="", host="172.16.0.29", port="15432")
print ("连接数据库成功")
cur = conn.cursor()
filename = "/data/tbase/tdapg.txt"
cols = ('id','nickname')
tablename="public.tdapg"
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 as msg:
print ("操作数据库出库 %s" %(msg.args[0]))
```

基于 ADO.NET 开发

最近更新时间: 2024-10-17 17:10:00

用户也可以通过 Npgsql 来和 TDSQL-A PostgreSQL版 进行数据库交互。

Npgsql 是 PostgreSQL 的一个开源 ADO.NET 数据提供程序，它允许用 C＃、Visual Basic、F＃编写的程序访问 PostgreSQL 数据库服务器。它以100%的 C＃ 代码实现，是免费并且是开源的。

golang 语言开发

最近更新时间: 2024-10-17 17:10:00

用户可使用golang的pgx接口来连接TDSQL-A PostgreSQL版数据库，进行数据库交互。开发所需要的资源包可前往 http://imgcache.finance.cloud.tencent.com:80github.com/jackc/pgx进行下载。

示例1：连接数据库

```
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 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 = "dbadmin" //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
}
```

示例2：表创建

```
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.tdapg(id varchar(20),nickname varchar(100))
distribute by shard(id) to group 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
*/
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
*/
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 = "dbadmin" //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
}
```

示例3：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.tdapg values('1','tdapg'),('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.tdapg 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 = "Tdapg is ' good!"
sql = "insert into public.tdapg 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")
}
}
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 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 = "dbadmin" //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
}
```

示例4：数据查询

```
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.tdapg 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 = "tdapg"
sql = "SELECT id,nickname FROM public.tdapg 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 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 = "dbadmin" //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
}
```

示例5：copy 数据插入

```
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{"dbadmin"}, []string{"id",
"nickname"}, pgx.CopyFromRows(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), copyCount)
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 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 = "dbadmin" //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
}
```