OpenPie | πCloudDB

PieCloudDB Community Edition

社区版管理员指南

Cloud Native Virtual Data Warehouse PieCloudDB Administrator's Guide

www.openpie.com

目录

声明

法律说明

本手册仅为所涉及的软件产品的信息性介绍和帮助手册。本手册不可成为全面判断该软件产品具备或不具备某些功能和/或技术参数的依据，亦不可用于判定该软件产品是否符合某些要求、技术任务和/或参数以及其他第三方规范文件。

商标声明

免责声明

任何情况下 OpenPie 对本文档可能的任何错误和/或遗漏以及因其导致的产品购买方损失（直接或间接损失，包括未得利益）均不承担任何责任。

注意

本手册描述的产品及其附件的某些特性和功能，取决于当地网络的设计和性能，以及您安装的环境；本手册中产品经过详细的测试，但不能保证与所有的软硬件系统完全兼容，不能保证本产品完全没有错误。

本手册仅供参考，不构成任何形式的承诺，产品相关内容请以实际使用时的呈现为准。OpenPie 保留随时修改本手册中任何信息的权利，无需提前通知且不承担任何责任。

前言

文档概述

本文档旨在为数据库管理员和其他 IT 专业人员提供关于 PieCloudDB 数据库技术的深入了解、管理方法和最佳实践。云原生数据库管理员需要掌握的关键技能和工具包括数据库设计、配置、监控、备份与恢复、性能优化和故障排除，以确保数据的安全性、可用性和高性能。

本文档全面覆盖了 PieCloudDB 的多个方面，包括但不限于总体架构、虚拟数仓管理、数据库对象的访问和定义、数据的增删改查操作、数据的导入导出、安装部署以及数据库配置等。这些内容旨在为数据库管理员提供全面的知识储备，帮助他们在多角度和多维度上理解和管理 PieCloudDB。

适用读者

本文档专为 PieCloudDB 数据库系统的管理者设计，包括数据库管理员和数据库运维人员，提供必要的参考信息，以帮助他们更高效地管理和运维 PieCloudDB 数据库系统。

在阅读本文档时，我们假设读者已经具备了 Linux/Unix 系统管理、数据管理系统、数据库管理以及 SQL 语言等方面的基础知识。

通用约定

约定	含义
危险	指示会出现严重的故障或问题，可能导致系统完全崩溃或无法正常运行。
警告	指示会出现重要的问题或异常情况，可能会对系统的功能或性能产生明显的影响，但系统仍然可以正常运行。
注意	指示可能会出现问题或异常情况，但不会对系统的正常运行产生严重影响。此类告警需要用户加以关注。
提示	提供附加信息、指导或补充说明。
>	多级菜单路径。
粗体	UI 页面的按键或菜单等元素。
####	命令、URL、示例中的代码、屏幕输出的文本或输入的文本等。
##	参数或者变量等。

1 PieCloudDB 社区版说明

PieCloudDB 社区版提供了一种全容器化的一键式部署方案，该方案基于 PieCloudDB 内核和 PieCloudDB 集群管理器（PieCloudDB Cluster Controller，简称 PDBCC）。它支持在线和离线两种部署方式，使用户能够迅速体验 PieCloudDB 内核的核心功能。这些功能包括高弹性、多租户支持、多集群管理、存算分离架构以及丰富的生态组件等。

PieCloudDB 社区版可以免费下载，适用于体验产品功能、个人学习、概念验证（PoC）等场景，让社区用户能够方便地体验先进的数仓虚拟化技术。

功能差异性说明

PieCloudDB 社区版作为免费提供的产品，不包含 PieCloudDB 部分高阶功能，具体包括但不限于：

- 图形化虚拟数仓管理和数据开发治理功能
- 高级数据导入导出功能，如 Dataflow、Flink CDC 等
- 监控、告警、计量、计费等运维管理功能
- PieProxy 可视化外部接入工具
- 系统和用户数据的备份与恢复功能
- 所有组件的高可用性和容灾支持
- 虚拟数仓节点级别的 CPU、内存和 IO 的资源配额管理和资源隔离
- 跨物理机器的部署能力

如果您希望深度体验 PieCloudDB 的完整功能，欢迎访问 PieCloudDB 云上云版（app.piecloudb.com）进行免费试用，或联系我们以获取关于 PieCloudDB 企业版的更多信息。

文档导览

了解 PieCloudDB，请参见如下文档：

- PieCloudDB 简介
- 系统架构概览

PieCloudDB 社区版的部署和管理功能，请参见如下文档：

- 部署 PieCloudDB 社区版
  - PieCloudDB 社区版在线部署
  - PieCloudDB 社区版离线部署
- 管理 PieCloudDB
  - 添加新虚拟数仓
  - 启动虚拟数仓
  - 停止虚拟数仓

- 修改虚拟数仓执行器数量
- 修改虚拟数仓配置参数
- 查看虚拟数仓信息
- 使用客户端连接虚拟数仓

PieCloudDB 社区版的内核功能，请参见如下文档：

- 管理数据库对象
- 管理数据
- 查询数据
- 查询性能优化
- 导入与导出数据

2 关于 PieCloudDB

2.1 PieCloudDB 简介

拓数派旗下的大模型数据计算系统（PieDataComputing System，简称 πDataCS）推出了首款数据计算引擎 PieCloudDB，这是一款基于云计算和容器化技术的数据库系统。PieCloudDB 采用先进的数仓虚拟化技术，将多个数仓统一整合到一个高可用的云虚拟数仓中，打通多云的数据管道，实现数据计算资源的按需扩缩容，提升数仓的敏捷性和弹性，帮助企业降低数仓管理的复杂度，具有可扩展性、灵活性和可靠性等优点。

PieCloudDB 利用 eMPP 分布式专利技术、服务无感知（Serverless）以及透明数据加密（TDE）等多项核心技术，为企业构建了一个高安全、高可靠、高可用的“坚如磐石”的云原生虚拟数仓，助力企业实现数据价值的最大化，更好地赋能业务发展，推动企业走向绿色计算，成为新一代 AI 数据计算基础设施的典范。

PieCloudDB 目前提供云上云版、社区版和企业版。有关产品版本信息及差异性，请参阅 OpenPie 官网。

核心特性

- 多维度弹性扩缩容

  采用存算分离的 eMPP（elastic Massive Parallel Processing）弹性大规模并行计算架构，PieCloudDB 实现了多维度弹性扩缩容。存储侧支持标准对象存储，充分利用云计算平台的优势，提供接近无限的存储容量；计算侧的无状态设计允许计算节点充分利用云环境中的海量计算节点池，实现按需扩容和缩容。这样，系统能够根据用户业务需求和数据量的变化，进行横向、纵向以及集群级别的弹性伸缩。
- 高可用能力

  依托于元数据、计算、数据三层分离的高可用架构，PieCloudDB 实现了云上存储资源与计算资源的独立管理，以及云上计算资源的弹性分配。当计算节点发生故障时，系统会自动快速发现并调度新的节点来替代故障节点。每份元数据将以多副本形式分散到多个服务节点，并进行定时备份，以避免数据丢失。存储数据的完整性、一致性和可靠性由分布式存储的多副本、EC（Erasure Coding）以及灾备能力提供保障，从而降低了因人为或自然原因导致的数据丢失概率，并确保单个硬件设备的故障不会影响业务连续性。
- 湖仓一体化分析

  通过分布式计算引擎和数据接口，PieCloudDB 实现了湖仓一体化架构，打破了数据湖与数据仓库之间的割裂体系。内置的 Foreign-Data Wrapper（FDW）模块使用户能够访问 HDFS、MySQL、Oracle 等外部数据源；同时，支持用户利用过程语言 PL（Procedural Language）自行开发模块进行数据分析，并兼容主流的大数据处理框架，如 Spark 和 Flink。
- 全面的 SQL 兼容度

高度兼容 SQL:2016 标准，完全支持 SQL:1992 标准，以及大部分的 SQL:1999 标准和部分 SQL:2003 标准（主要支持其中的 OLAP 特性）。此外，兼容 PostgreSQL 协议，并支持标准数据库接口，例如 ODBC、JDBC 等。

应用场景

- 多样化的数据处理

  灵活扩展的数据引擎将关系型数据库 SQL、Spark/Flink 等流批一体处理、LLM 的向量数据库，以及 GIS 地理数据库等不同类型的数据库和数据处理方式集成为一体，用户可以根据具体需求，选择适合的数据存储和处理方式。
- 即席查询和实时分析

  支持即席查询和实时分析技术，PieCloudDB 能够根据用户需求进行灵活的数据探索和分析，适用于需要实时监测和决策支持的场景。
- 高级分析和机器学习

  PieCloudDB 提供了一系列高级分析函数，例如窗口函数和高级聚合函数等，支持复杂的数据分析和计算。用户可以利用这些功能进行数据探索、特征工程和模型训练，以支持机器学习和数据驱动的决策制定。
- 数据共享和分析

  提供统一的数据管理和分析环境，支持多模态数据处理（包括结构化、半结构化和非结构化数据）、数据共享与分析，使企业和组织能够更有效地利用各种类型的数据支持决策和创新。

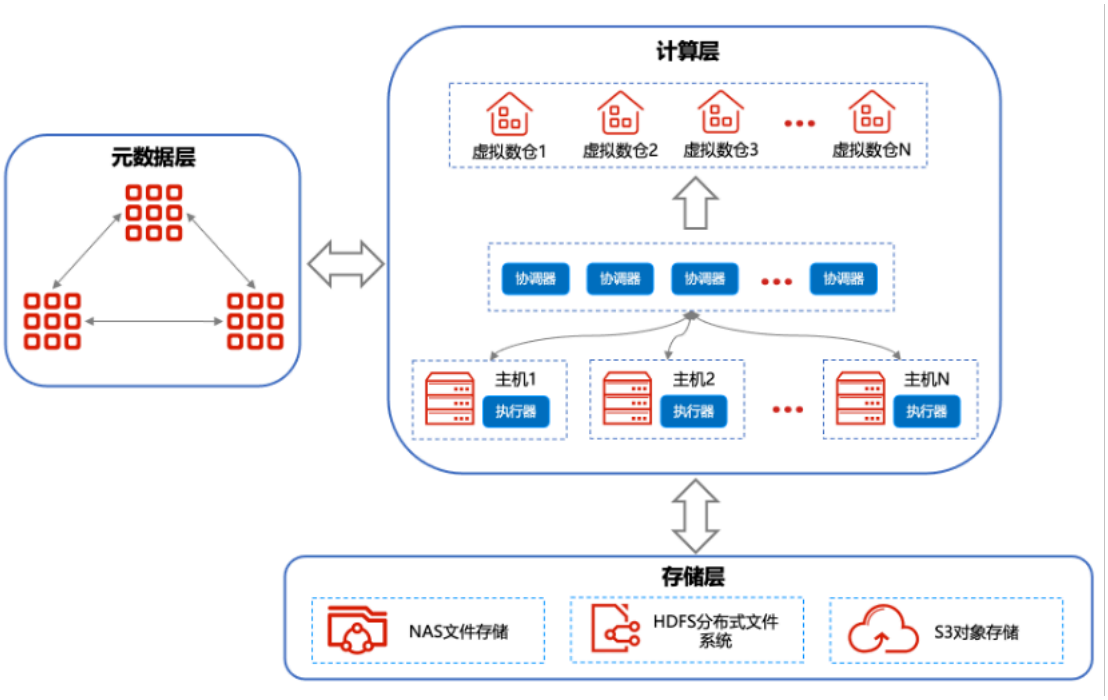
2.2 系统架构概览

PieCloudDB 采用了 eMPP（elastic Massive Parallel Processing）弹性大规模并行计算架构，这是在传统的 MPP 架构基础上增加了“Elastic”这一新维度。基于 eMPP，PieCloudDB 能够实现多个虚拟数仓的并发任务执行，并能够灵活地对计算集群进行扩缩容，以适应负载变化，实现高效伸缩。这种设计使得计算规模能够灵活适应数据规模，轻松处理 PB 级的海量数据。

与传统的 OLAP 数据库相比，云原生分布式数据库 PieCloudDB 具有以下优势：

- 采用存算分离的模式，具有良好的伸缩性，并支持快速弹性扩缩容。
- 采用共享存储架构，支持 Amazon S3、HDFS 和 NAS 等协议，实现数据存储的高可用性。
- 智能且高效地生成统计信息与查询计划，支持聚集下推、预计算、Data Skipping 等高级特性，全面满足各种复杂的分析查询需求。
- 采用高效的行列混存格式和向量化引擎，显著提升运算速率。
- 兼容 PostgreSQL 的协议、语法和生态。

PieCloudDB 的整体架构如下图所示。



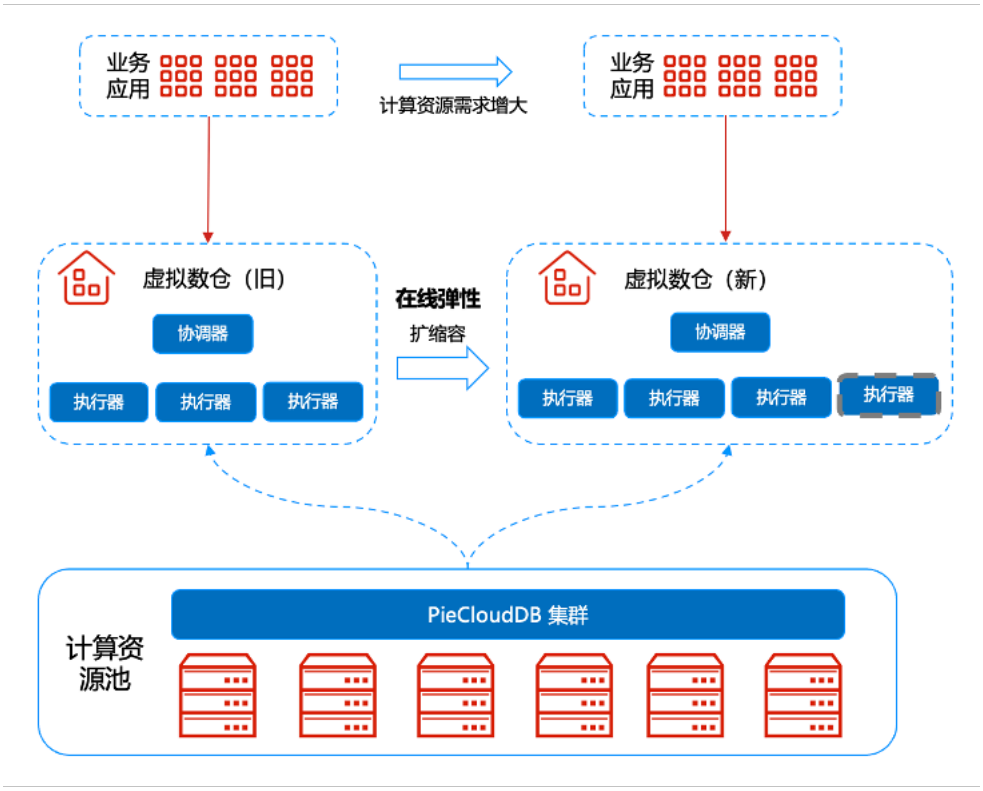
PieCloudDB 采用元数据、计算、数据分离的三层架构，实现了云上存储资源与计算资源的独立管理。云上计算资源可以弹性分配，根据查询计算任务的需求动态启动，并按照使用时间和规模来计算成本。

计算层

PieCloudDB 的计算层依托于 eMPP 弹性大规模并行计算技术，把所有的计算节点组成一个大的计算资源池，用户可以在计算资源池上创建云原生虚拟数仓。多个虚拟仓库可以同时运行，对数据执行多个并发处理，PieCloudDB 还可以根据工作负载构建多个虚拟仓库以满足不同的业务需求。

虚拟数仓由执行器和协调器等资源组成，每个虚拟数仓都拥有独立的计算集群，且不与其他虚拟数仓交互，形成无共享架构。在执行具体的业务应用请求时，协调器负责统筹整个计算资源，将优化后的 SQL 语句下发至执行器。执行器负责具体的 SQL 执行，多个执行器之间进行并行计算，并将结果返回给协调器，最终由协调器将结果反馈给业务应用。

随着业务应用的计算任务数量增加或需要处理的数据量变大，相应的计算资源（主要为 CPU 和内存）也会随之增加。根据计算资源的需求，虚拟数仓可以共享计算资源并实现在线弹性扩缩容。如下图所示，执行器可以从原来的 3 个扩容到 4 个，缩容的原理同样适用。



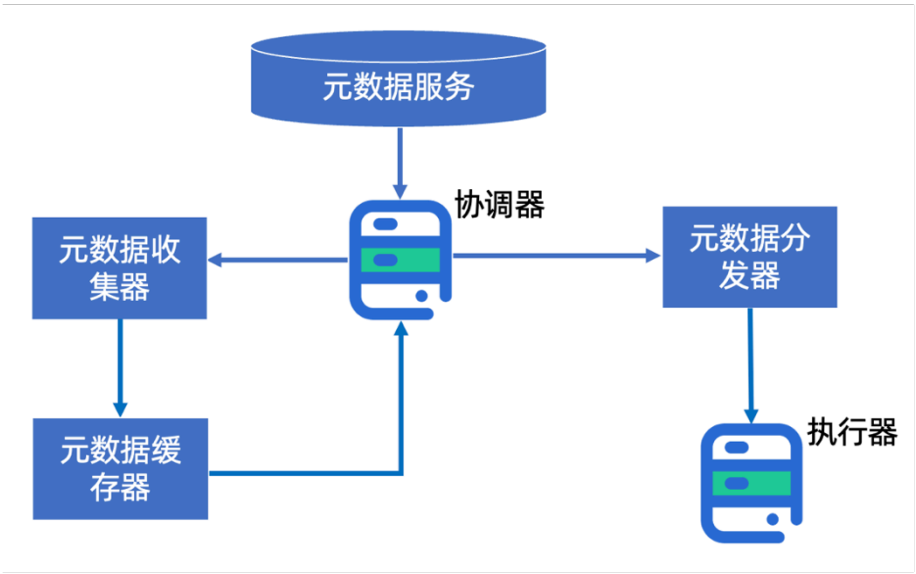
PieCloudDB 的执行器都是无状态的。当多个执行器并行处理数据时，如果其中一个执行器突然发生故障，PieCloudDB 会通过“快速发现故障 --> 快速重建执行器”的机制进行故障修复。整个过程耗时极短，从而最大程度地减少对业务侧 SQL 命令执行的影响，确保服务的高可用性。

此外，计算层的各个计算节点针对元数据和用户数据都设计了多层缓存结构。这种设计可以降低网络延迟并减少数据移动，不仅提高了计算效率，也满足了用户的实时性需求。

元数据层

PieCloudDB 的元数据主要用于描述、管理、控制数据库的关键功能和用户数据，并通过分布式 KV 存储进行管理。PieCloudDB 提供的独立元数据服务（Catalog Service）确保了元数据存取的效率和高可用性。

PieCloudDB 的元数据服务主要依赖于三个核心模块：元数据收集器（位于协调器）、元数据缓存器（位于执行器）以及元数据分发器（位于协调器）。在数据分发过程中，协调器负责分析用户指令、收集数据并返回结果；执行器则负责处理用户指令并生成计算结果。协调器和执行器都需要元数据来确保它们能够正确分析和处理用户请求。



PieCloudDB 元数据服务的工作流程通常如下：元数据收集器负责从协调器收集执行器的数据；元数据缓存器负责缓存已收集的元数据，并分析当前用户指令所需的元数据；元数据分发器负责将元数据载入执行器以供运算。

元数据缓存器根据查询请求的类型选择相应的缓存模式。对于某些特定类型的元数据请求，缓存器会智能地切换到预计算模式，提前计算结果并进行缓存，以减少向元数据分发器推送的数据量。对于其他类型的请求，缓存器则切换回标准的数据缓存模式。此外，元数据缓存器还会根据访问频率对元数据进行冷热分层，将频繁访问的热数据保留在内存中，从而提高缓存命中率。

同时，元数据收集器和缓存器与协调器紧密绑定，每个协调器的处理会话都会启动一个对应的收集器和缓存器。分发器与执行器同样紧密绑定，每个协调器的每个会话都会启动一个分发器。

作为 PieCloudDB 架构的关键部分，元数据层支持分布式和高可用性，使云数据仓库能够轻松应对分布式环境下的各种挑战。它为数据共享、用户权限管理、多虚拟数仓并发、分布式锁等服务提供了强有力的支撑。

存储层

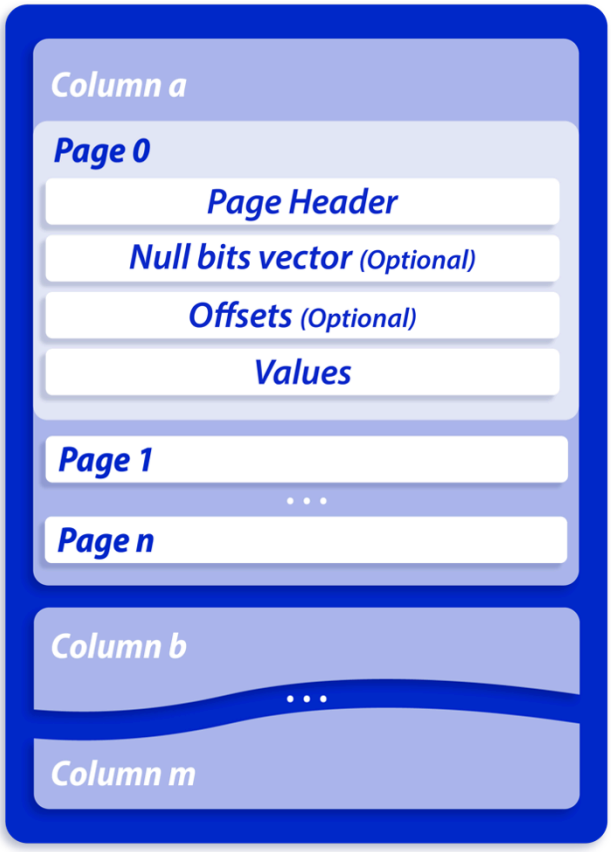
PieCloudDB 架构采用存算分离模式，使得存储层相对独立，其存储容量能够独立于计算层进行弹性伸缩。PieCloudDB 支持对象存储、HDFS、NAS 等分布式存储解决方案，为用户提供云端可扩展的共享存储服务，以存储用户数据和元数据。

在底层数据存储之上，PieCloudDB 计算引擎依托的数据存储底座是简墨 JANM。简墨可兼容不同的云环境，主要设计用于对象存储，并将对象存储作为持久化存储层。

针对 PieCloudDB eMPP 架构下数据的分布式特性和弹性特点，简墨采用一致性 Hash 算法来确保分布式系统中每个计算节点对数据访问的一致性，即使在扩缩容过程中也能尽可能减少数据缓存迁移。在数据安全性方面，简墨结合 PieCloudDB 中的云原生虚拟数仓透明加密技术，在数据存储落盘时即完成加密。该透明加密技术采用三级密钥系统，确保数据的绝对安全。

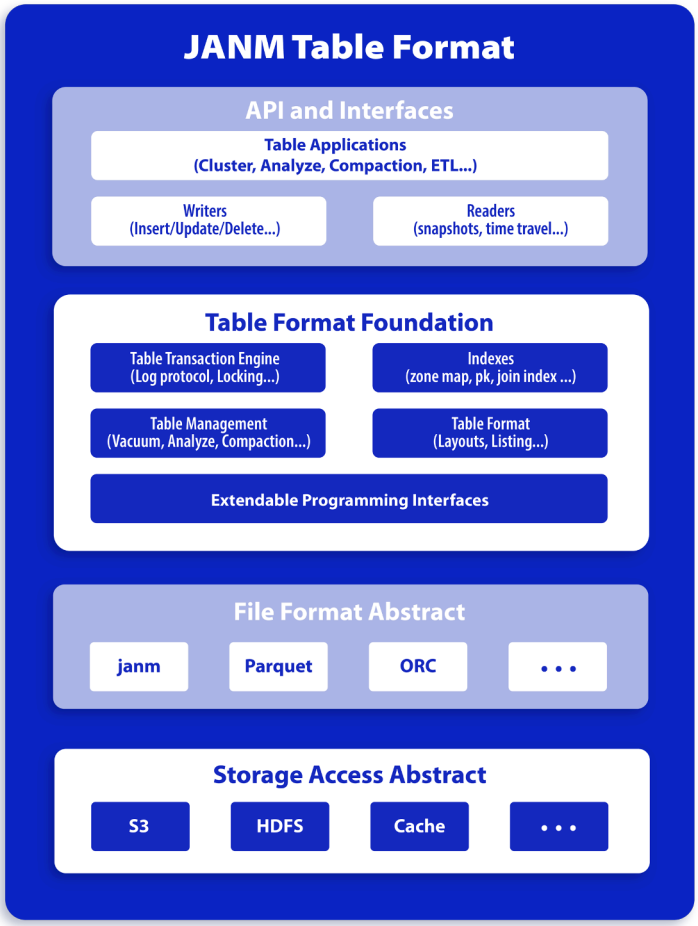
简墨存储系统旨在构建一个能够满足各种云场景下高性能计算系统的数据管理和存储需求的底座，并计划通过三个阶段逐步实现这一目标。具体来说，第一阶段是成为新一代的云原生存储系统；第二阶段是发展为大数据计算系统的云存储底座；第三阶段则是演化为统一的大数据计算系统存储引擎。目前，简墨已经成功完成了第一阶段和第二阶段的演进。

简墨在其发展第一阶段的目标是成为以 janm 文件格式为核心的新一代云原生虚拟数仓 PieCloudDB 的云原生存储系统。PieCloudDB 的自研 janm 文件格式采用了行列混合存储结构，这一设计不仅保留了行存储的高效性能，还结合了列存储的高压缩比和 Cache Line 优化等优势。此外，在重组数据时，简墨会重新定义数据在磁盘和内存中的格式，确保表中的数据在磁盘与内存之间的转换无需额外开销。janm 存储格式的示意图如下所示。



在 janm 文件格式中，简墨不仅收集文件内的数据统计信息以加速查询，还支持向量化（SIMD）计算、并行计算和预计算等性能优化特性。为了进一步提升 I/O 性能，janm 文件格式内置了多种压缩算法，如 zstd 和 lz 等。此外，简墨在数据读取与查询方面进行了大量优化，实现了数据裁剪（Data Skipping）、预计算加速聚集查询、Smart Analyze、TOAST 等功能，显著提升了数据加载和查询的效率。

简墨在其演进的第二阶段旨在成为大数据计算系统的云存储底座。简墨存储系统以 JANM Table Format 为核心，其架构主要分为四个层次：存储访问抽象层（Storage Access Abstraction）、数据文件格式抽象层（File Format Abstraction）、表格式基础层（Table Format Foundation）和可扩展的编程接口层（API and Interfaces）。每个层级都依赖于直接下方的层级，并从中获取所需的功能。JANM Table Format 的层级结构如下图所示。



JANM Table Format 的最底层是存储访问抽象层。简墨通过抽象 API 实现了与各种类型存储（包括云对象存储，例如 S3、HDFS 等）的交互，其强大的存储适配接口确保了与所有存储引擎的兼容性。此外，简墨还提供文件系统监控和多种读写策略，以进一步优化存储功能。

JANM Table Format 的数据文件格式抽象层支持多种文件格式，具备适配多种文件格式的能力，并提供统一的访问接口以简化数据访问操作。这使用户能够自由选择不同的文件格式来存储数据于简墨。同时，简墨采用独特的文件布局设计，记录每个文件的所有更改，并创建一个独立的 redo 日志，这有助于实现更多丰富的功能。

JANM Table Format 的表格式核心层负责提供各项特性的功能封装和实现，主要包含表的事务引擎、索引、表数据的自适应管理和表格式相关操作与控制封装等。这一层的主要功能包括：

- 表的事务引擎实现了文件级别的 MVCC（多版本并发控制），支持根据隔离级别进行数据库的可见性判断，确保了一定程度的并发控制。
- 索引功能有助于数据库优化查询规划，减少 I/O 总量，并提供更快的响应时间。目前，JANM 支持构建索引以实现数据裁剪（Data Skipping）。在 OLAP 分析型数据库的应用场景中，文件列表和列的索引信息使 OLAP 引擎能够快速生成高效的查询计划。
- 自适应数据管理功能主要包括数据清理（Vacuum）、数据分布信息采样（Smart Analyze）和小文件合并（Compaction），这些功能可以提升 I/O 效率。

- 对表组成和布局进行控制和功能封装，可以快速进行文件遍历和数据大小统计等操作，同时降低对象存储中对文件进行 List 操作的开销。

JANM Table Format 还支持可扩展的编程接口，提供统一的 API 作为与外部服务交互和数据访问的入口，以便于第三方应用的接入。针对表的应用服务，简墨提供了无状态的数据管理应用，这些应用可以注册到任意服务中，从而实现自适应的数据管理。

2.3 体验新特性

2.3.1 LocalCache

PieCloudDB 的 LocalCache 功能用于管理本地文件缓存，支持文件上传、下载、错误处理、缓存查询以及缓存清理等功能。通过有效管理本地缓存，LocalCache 能够显著减少从 S3 等云存储服务读取文件时产生的网络通信成本，并降低查询延迟。

> **提示:**
>
> PieCloudDB 内核 V2.13.0 及以上版本支持此功能。

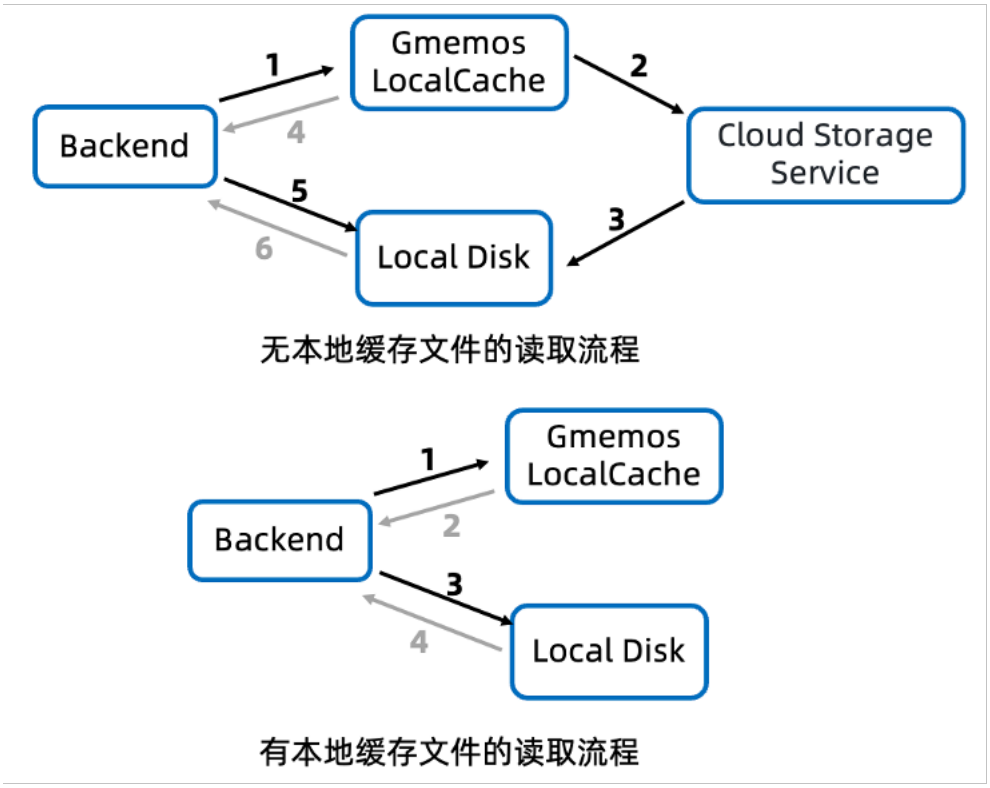
应用场景

LocalCache 会缓存文件的历史查询记录，这适用于频繁对同一张或几张表进行多次查询的场景。然而，对于涉及大量不同表的查询场景，历史缓存可能会被替换和清理，这种情况下 LocalCache 可能无法有效提升查询速率。

工作原理

PieCloudDB 将文件上传和下载功能从 Backend 中分离出来，统一交由 LocalCache 模块管理。因此，Backend 无需直接与 S3 等云存储服务器交互，而是向 LocalCache 发送查询请求。LocalCache 根据 Backend 的请求内容，执行相应的文件操作。

下图分别展示了“无本地缓存文件”和“有本地缓存文件”两种场景下的文件读取流程。



启用 LocalCache 功能后，即使在数据库重启的情况下，已有的本地缓存文件也不会被清除，因此可以继续使用。

使用说明

要使用 LocalCache 功能，需要配置以下参数：

- *pdb_janm_enable_disk_cache*：用于控制是否启用 LocalCache 功能。设置为 “ON” 以启用，设置为 “OFF” 以禁用。禁用 LocalCache 时，文件将直接载入内存，而不通过本地磁盘。以下示例展示了如何启用 LocalCache 功能：

  ```
  SET pdb_janm_enable_disk_cache='on';
  ```

- *gmemos.local_cache_usage*：设置 LocalCache 的缓存空间大小，其默认值为 10240。建议的最小缓存空间大小为 “BlockSize * 并发数” 。如果设置的值低于此建议值，可能会导致某些线程进入等待状态，影响查询性能。

  缓存空间越大，可缓存的文件数量越多，缓存命中率越高，性能提升越明显。如果由于本地磁盘空间不足或其他原因导致无法分配足够的磁盘空间给本地缓存，建议关闭 LocalCache 功能。当缓存空间已满时，新的查询请求将触发清理操作。

- *gmemos.single_table_constrict_percentage*：设置 LocalCache 中单表数据的最大缓存占用比例。为了避免在读取大表数据时大量占用缓存空间，应确保 “single_table_constrict_percentage * local_cache_usage” 的计算结果大于 BlockSize。

此外，*pdb_get_local_cache_hit_rate()* 函数可用于查询当前各 Segment 的缓存命中率等信息。参考命令如下：

```
=> CREATE EXTENSION pdb_internal_tools;

=> SELECT * FROM pdb_get_local_cache_hit_rate();
 seg_id | cache_size | cache_file_count | hit_query_count | total_query_count | hit_rate
--------+------------+------------------+-----------------+-------------------+----------
      0 |        847 |               43 |              50 |               354 |  14.12%
      1 |        836 |               42 |              38 |               354 |  10.73%
      2 |        856 |               43 |              40 |               354 |  11.30%
(3 rows)
```

pdb_get_local_cache_hit_rate() 函数的字段含义如下：

字段	数据类型	含义
seg_id	smallint	Segment ID
cache_size	bigint	已有的缓存大小
cache_file_count	bigint	已有的缓存文件数量
hit_query_count	bigint	缓存命中数量
total_query_count	bigint	缓存请求数量
hit_rate	text	命中率

2.3.2 多进程 ic-proxy

PieCloudDB 支持 ic-proxy 的多进程工作模式，这有助于增强分布式系统的负载均衡能力，提升 Motion 数据的传输效率，以及提高数据库的并行查询性能。

> **提示：**
>
> PieCloudDB 内核 V2.13.0 及以上版本支持此功能。

应用场景

在并行执行过程中，如果涉及的 Motion 数据量很大，单个进程的 ic-proxy 可能会导致 CPU 使用率飙升至 100%，从而引起过重的系统负载，影响查询性能。相比之下，多进程 ic-proxy 能够更有效地实现负载均衡和优化网络性能。在面对大数据量的场景时，它有助于防止在单进程执行并行查询时遇到的性能瓶颈。

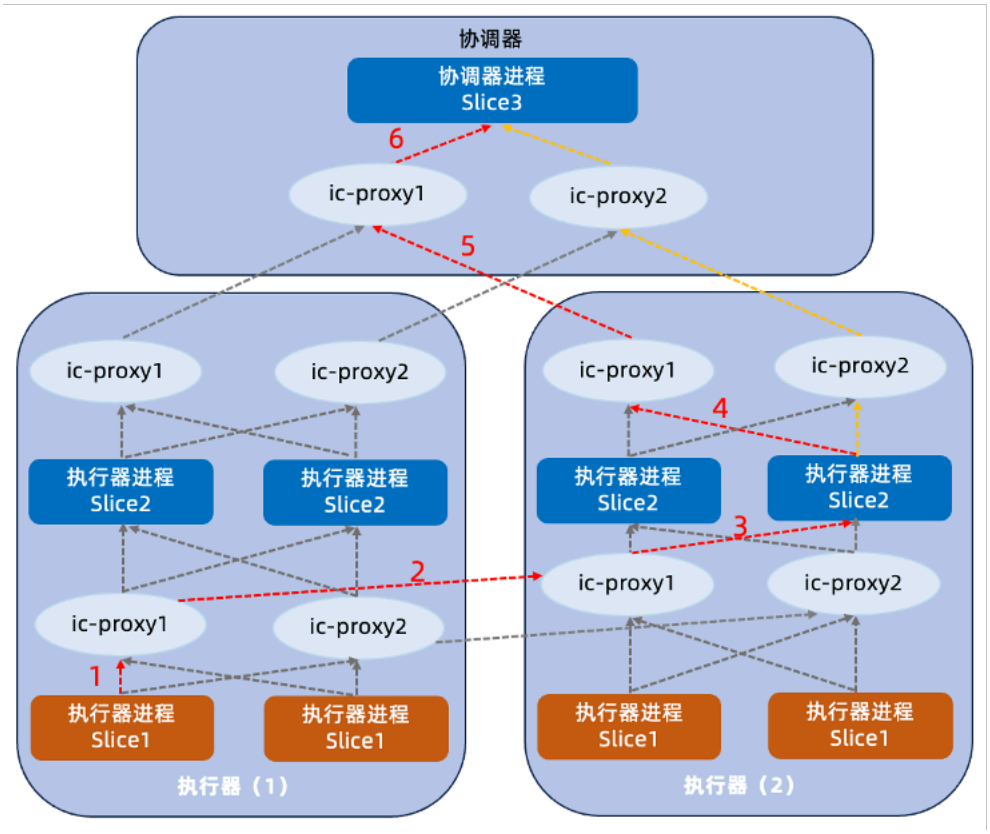
工作原理

PieCloudDB 的并行查询是由协调器以及一个或多个执行器共同完成的。协调器负责接收、解析并优化查询请求，而执行器则负责执行协调器分配的任务。当存在多个执行器时，它们可以并行处理同一个 SQL 查询任务。

在查询处理过程中，PieCloudDB 会生成多个进程以处理并行查询任务。如果查询涉及 Motion 操作，即数据移动，那么就需要通过 Interconnect 的 ic-proxy 在执行器之间传输数据。当执行计划中包含数据移动操作时，该查询计划会被分割成多个片段（Slice），并分布在数据传输的两端。在执行并行查询时，每个执行器将并行运行多个

进程，每个查询计划的片段至少分配给一个工作进程，每个工作进程独立处理分配给它的查询计划部分。

下图展示了 PieCloudDB 多进程 ic-proxy 的数据传输原理。



在并行查询的场景中，本文以 Slice1 的执行器（1）向 Slice2 的执行器（2）传输数据为例，阐述多进程 ic-proxy 如何传输 Motion 数据。

1. Slice1 中执行器（1）将数据发送到对应的 ic-proxy1。
2. 执行器（1）的 ic-proxy1 在接收到数据后，将其路由到 Slice1 中执行器（2）上的 ic-proxy1。
3. Slice1 中执行器（2）上的 ic-proxy1 根据 ic-proxy Key 路由数据到执行器 2 所对应的 Slice2 中的执行器进程。
4. Slice2 中的执行器进程随后将数据发送至其对应的 ic-proxy1（或 ic-proxy2）。
5. Slice2 中的执行器（2）上的 ic-proxy1（或 ic-proxy2）再将数据发送到协调器上的 ic-proxy1（或 ic-proxy2）。
6. 最终，协调器上的 ic-proxy1（或 ic-proxy2）将执行结果返回至 Slice3 的协调器进程。

需要注意的是，不同执行器之间或者执行器与协调器之间 ic-proxy_i 的接收端和发送端是一一对应的关系，例如，ic-proxy_1 仅能接受来自 ic-proxy_1 的数据，或者将数据发送给 ic-proxy_1。

协调器和执行器与 ic-proxy 之间的通信通过 Domain Socket（域套接字）进行。例如，当协调器或执行器向 ic-proxy 发送数据时，作为 Server 端的 ic-proxy 进程在启动

时即监听此连接，等待协调器或执行器建立连接并发送数据。由于各 ic-proxy 进程不位于同一个执行器（物理机）上，它们之间通过网络 Socket 进行数据传输。

使用说明

pdb_ic_proxy_num_worker 参数用于配置协调器和执行器上启动的 ic-proxy 进程数量，且协调器和执行器将启动相同数量的 ic-proxy 进程。此参数的取值应根据物理机的网络带宽来决定，其最低取值为 1，这也是其默认值，表示不启用多进程 ic-proxy。

pdb_ic_proxy_num_worker 参数默认在初始化部署的时候进行配置，需要注意的是，该参数值被修改后，需要重新启动虚拟数仓才能生效。

2.3.3 动态配置查询簇

PieCloudDB 支持动态配置查询簇的大小，以提升单个查询并行使用底层资源的能力，同时加快查询响应速度。

> **提示:**
>
> PieCloudDB 内核 V2.14.0 及以上版本支持此功能。

应用场景

动态配置查询簇的功能主要适用于单个 SELECT 查询的场景。

工作原理

pdb_parallel_factor 参数用于设置查询簇的大小，它决定了执行器（Backend）的数量因子，即总计算进程数量为“执行器数量 * pdb_parallel_factor”，由此实现可伸缩的并行化查询。

下面以具体示例介绍该功能的工作原理。首先对三张表 t1、t2 和 t3 执行 SELECT 查询，并使用 EXPLAIN 分析其执行计划。

```
test=# \d
          List of relations
 Schema | Name | Type  |  Owner  | Storage
--------+------+-------+---------+---------
 public | t1   | table | gpadmin |
 public | t2   | table | gpadmin |
 public | t3   | table | gpadmin |
(3 rows)

test=# explain select sum(t1.c1) from t1 where c1 not in (select t3.c1 from t2, t3 where t3.c2 = t2.c2);
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=968981.15..968981.16 rows=1 width=8)
   ->  Gather Motion 2:1  (slice1; segments: 2)  (cost=968981.10..968981.14 rows=2 width=8)
         ->  Partial Aggregate  (cost=968981.10..968981.11 rows=1 width=8)
               ->  Hash Left Anti Semi (Not-In) Join  (cost=312730.48..906482.58 rows=24999409 width=4)
                     Hash Cond: (t1.c1 = t3.c1)
                     ->  Seq Scan on t1  (cost=0.00..250008.00 rows=25000000 width=4)
                     ->  Hash  (cost=312715.60..312715.60 rows=1191 width=4)
                           ->  Broadcast Motion 2:2  (slice2; segments: 2)  (cost=40.00..312715.60 rows=1191 width=4)
                                 ->  Hash Join  (cost=40.00..312697.74 rows=595 width=4)
                                       Hash Cond: (t3.c2 = t2.c2)
                                       ->  Seq Scan on t3  (cost=0.00..250008.00 rows=25000000 width=8)
                                       ->  Hash  (cost=25.00..25.00 rows=1200 width=4)
                                             ->  Broadcast Motion 2:2  (slice3; segments: 2)  (cost=0.00..25.00 rows=1200 width=4)
                                                   ->  Seq Scan on t2  (cost=0.00..7.00 rows=600 width=4)
 Optimizer: Postgres query optimizer
(15 rows)
```

接下来，通过将 *pdb_parallel_factor* 设置为 3 来增加查询执行的并发数量。设置完成后，再次执行 EXPLAIN 以分析查询的执行计划。

```
test=# set pdb_parallel_factor to 3;
SET
test=# explain select sum(t1.c1) from t1 where c1 not in (select t3.c1 from t2, t3 where t3.c2 = t2.c2);
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=968981.15..968981.16 rows=1 width=8)
   ->  Gather Motion 6:1  (slice1; segments: 6)  (cost=968981.10..968981.14 rows=2 width=8)
         ->  Partial Aggregate  (cost=968981.10..968981.11 rows=1 width=8)
               ->  Hash Left Anti Semi (Not-In) Join  (cost=312730.48..906482.58 rows=24999409 width=4)
                     Hash Cond: (t1.c1 = t3.c1)
                     ->  Seq Scan on t1  (cost=0.00..250008.00 rows=25000000 width=4)
                     ->  Hash  (cost=312715.60..312715.60 rows=1191 width=4)
                           ->  Broadcast Motion 6:6  (slice2; segments: 6)  (cost=40.00..312715.60 rows=1191 width=4)
                                 ->  Hash Join  (cost=40.00..312697.74 rows=595 width=4)
                                       Hash Cond: (t3.c2 = t2.c2)
                                       ->  Seq Scan on t3  (cost=0.00..250008.00 rows=25000000 width=8)
                                       ->  Hash  (cost=25.00..25.00 rows=1200 width=4)
                                             ->  Broadcast Motion 6:6  (slice3; segments: 6)  (cost=0.00..25.00 rows=1200 width=4)
                                                   ->  Seq Scan on t2  (cost=0.00..7.00 rows=600 width=4)
 Optimizer: Postgres query optimizer
(15 rows)
```

通过比较上述示例在开启并行化查询前后的 EXPLAIN 输出结果，可以观察到以下主要变化：

- Gather Motion 从 2:1 增加到 6:1，这表明每个执行器上有 3 个进程在同时并行工作。
- Broadcast Motion 从 2:2 增加到 6:6，这表明在广播每一份数据时，使用了 6 个虚拟执行器。

如果将 *pdb_parallel_factor* 参数分别设置为 1、2、3 和 4，并执行相同的 SELECT 查询，会发现查询执行时间依次减少，这表明查询性能随着并行度的提高而线性提升，如下所示。

```
test=# set pdb_parallel_factor to 1;
SET
test=# \timing
Timing is on.
test=# select sum(t1.c1) from t1 where c1 not in (select t3.c1 from t2, t3 where t3.c2 = t2.c2);
       sum
-----------------
 1250000024279400
(1 row)

Time: 16370.020 ms (00:16.370)
test=# set pdb_parallel_factor to 2;
SET
Time: 4.859 ms
test=# select sum(t1.c1) from t1 where c1 not in (select t3.c1 from t2, t3 where t3.c2 = t2.c2);
       sum
-----------------
 1250000024279400
(1 row)

Time: 9146.481 ms (00:09.146)
test=# set pdb_parallel_factor to 3;
SET
Time: 5.114 ms
test=# select sum(t1.c1) from t1 where c1 not in (select t3.c1 from t2, t3 where t3.c2 = t2.c2);
       sum
-----------------
 1250000024279400
(1 row)

Time: 6607.457 ms (00:06.607)
test=# set pdb_parallel_factor to 4;
SET
Time: 5.038 ms
test=# select sum(t1.c1) from t1 where c1 not in (select t3.c1 from t2, t3 where t3.c2 = t2.c2);
       sum
-----------------
 1250000024279400
(1 row)

Time: 4584.176 ms (00:04.584)
```

使用说明

pdb_parallel_factor 参数的值是动态生效的，并且必须是整数。该参数的取值范围受到限制，即“执行器数量与 pdb_parallel_factor 的乘积不能超过 CPU 的总核数（而非总线程数）”。参考命令如下：

```
=> SET pdb_parallel_factor to int_value;
```

3 部署 PieCloudDB 社区版

3.1 部署概述

一个典型且完整的 PieCloudDB 集群通常由以下核心组件构成：计算服务（包括一个协调器和若干执行器）、元数据服务以及对象存储服务。PieCloudDB 社区版的部署方案提供了一个全面的镜像，该镜像包含了所有这些基础服务，支持用户一键式快速部署 PieCloudDB 集群。

PieCloudDB 社区版的部署镜像具体提供了以下服务和组件：计算服务（包括一个协调器和三个执行器）、FoundationDB 作为元数据服务、MinIO 作为对象存储服务、元数据缓存服务，以及 PieCloudDB 集群管理器（PieCloudDB Cluster Controller，简称 PDBCC）和两个 PieCloudDB 集群管理代理（PieCloudDB Cluster Agent，简称 PDBCA）组件。

PieCloudDB Database 社区版支持如下两种容器化部署方式：

- PieCloudDB 社区版在线部署
- PieCloudDB 社区版离线部署

3.2 PieCloudDB 社区版在线部署

-本章节主要介绍通过在线方式安装部署 PieCloudDB 社区版的流程。在线部署方式将通过 Docker Hub 获取镜像一键部署。

环境准备

- 准备 x86 架构下 Linux 系统的安装环境，并确保安装 Docker。
- 建议至少为 Docker 预留 8 GB 内存以及 10 GB 磁盘存储空间。
- 安装 Postgres 命令行交互式客户端工具 psql。安装命令参考如下：
  - Ubuntu 操作系统

    ```
    sudo apt install postgresql-client
    ```

  - Centos 操作系统

    ```
    sudo yum install postgresql
    ```

  执行 `psql --help` 命令，如果提示 psql 的帮助信息则说明安装成功。

操作步骤

以具有 sudo 权限的用户登陆安装 Docker 的部署环境后，参考如下步骤部署 PieCloudDB 社区版：

1. 使用 Docker 命令启动 PieCloudDB 社区版镜像。

```
sudo docker run -p 5432-5442:5432-5442 --name piecloudb -itd openpietsp/
piecloudb-ce-allin1:latest
```

2. 查看日志，确认已经成功部署 PieCloudDB。

```
sudo docker logs -f piecloudb

# 输出信息的片段示例
2024/08/12 03:12:10 Cluster created
2024/08/12 03:12:10 clusterid:1  port:6000  hostname:"127.0.0.1"  address:"127.0.0.1"
 datadir:"/workspace/apps/pdb_data/ca-segment/openpie/1/6000"
2024/08/12 03:12:10 clusterid:1  contentid:1  port:6001  hostname:"127.0.0.1"
 address:"127.0.0.1"  datadir:"/workspace/apps/pdb_data/ca-segment/openpie/1/6001"
2024/08/12 03:12:10 clusterid:1  contentid:2  port:6002  hostname:"127.0.0.1"
 address:"127.0.0.1"  datadir:"/workspace/apps/pdb_data/ca-segment/openpie/1/6002"
2024/08/12 03:12:10 clusterid:1  contentid:-1  port:5432  hostname:"127.0.0.1"
  address:"127.0.0.1"  datadir:"/workspace/apps/pdb_data/ca-coordinator/
openpie/1/5432"
```

3. 使用 psql 连接 PieCloudDB 并测试。

   如下连接示例使用默认的用户“openpie”和默认的数据库“openpie”。

```
psql -h127.0.0.1 -p5432 -Uopenpie -dopenpie
```

   执行一些简单的 SQL 语句进行测试，示例如下：

```
openpie=# CREATE DATABASE test;
CREATE DATABASE

openpie=# \c test;
You are now connected to database "test" as user "openpie".

test=# CREATE TABLE t1(id int);
NOTICE:  distribution policy forced to random for current relation with access method:
 janm
CREATE TABLE

test=# INSERT INTO t1 VALUES(1),(2),(3);
INSERT 0 3

test=# SELECT * FROM t1;
 id
----
  2
  1
  3
(3 rows)
```

3.3 PieCloudDB 社区版离线部署

本章节主要介绍通过离线方式安装部署 PieCloudDB 社区版的流程。离线部署方式主要适用于部署环境未连接互联网的使用场景。

环境准备

- 准备 x86 架构下 Linux 系统的安装环境，并确保安装 Docker。
- 建议至少为 Docker 预留 8 GB 内存以及 10 GB 磁盘存储空间。
- 安装 Postgres 命令行交互式客户端工具 psql。安装命令参考如下：
  - Ubuntu 操作系统

    ```
    sudo apt install postgresql-client
    ```

  - Centos 操作系统

    ```
    sudo yum install postgresql
    ```

  执行 `psql --help` 命令，如果提示 psql 的帮助信息则说明安装成功。

操作步骤

以具有 sudo 权限的用户登陆安装 Docker 的部署环境后，参考如下步骤部署 PieCloudDB 社区版：

1. 下载 PieCloudDB 社区版离线镜像压缩包。

   ```
   wget https://piecloudb.oss-cn-beijing.aliyuncs.com/community/piecloudb-ce-allin1.tar.gz
   ```

2. 解压所下载的 PieCloudDB 社区版离线镜像压缩包。

   ```
   tar -zxvf piecloudb-ce-allin1.tar.gz

   # 输出信息示例
   ./piecloudb-ce-allin1/
   ./piecloudb-ce-allin1/start-piecloudb.sh
   ./piecloudb-ce-allin1/piecloudb-ce-allin1.tar
   ```

3. 进入镜像压缩包解压后的目录，执行启动脚本 start-piedb.sh 初始化 PieCloudDB。

   ```
   cd piecloudb-ce-allin1
   sudo bash start-piecloudb.sh

   # 输出信息的片段示例
   Loaded image: reg.dev.openpie.com/release/piecloudb-ce-allin1:latest
   22ee1964782651b3d7abaea3df129b2f0a561b360d842ce4b972a51c14fc7582
   piecloudb is starting
   print log: docker logs -f piecloudb
   ```

4. 查看日志，确认已经成功部署 PieCloudDB。

```
sudo docker logs -f piecloudb

# 输出信息的片段示例
2024/08/12 03:29:42 Cluster created
2024/08/12 03:29:42 clusterid:1 port:6000 hostname:"127.0.0.1" address:"127.0.0.1"
 datadir:"/workspace/apps/pdb_data/ca-segment/openpie/1/6000"
2024/08/12 03:29:42 clusterid:1 contentid:1 port:6001 hostname:"127.0.0.1"
 address:"127.0.0.1" datadir:"/workspace/apps/pdb_data/ca-segment/openpie/1/6001"
2024/08/12 03:29:42 clusterid:1 contentid:2 port:6002 hostname:"127.0.0.1"
 address:"127.0.0.1" datadir:"/workspace/apps/pdb_data/ca-segment/openpie/1/6002"
2024/08/12 03:29:42 clusterid:1 contentid:-1 port:5432 hostname:"127.0.0.1"
 address:"127.0.0.1" datadir:"/workspace/apps/pdb_data/ca-coordinator/
openpie/1/5432"
```

5. 使用 psql 连接 PieCloudDB 并测试。

   如下连接示例使用默认的用户“openpie”和默认的数据库“openpie”。

```
psql -h127.0.0.1 -p5432 -Uopenpie -dopenpie
```

   执行一些简单的 SQL 语句进行测试，示例如下：

```
openpie=# CREATE DATABASE test;
CREATE DATABASE

openpie=# \c test;
You are now connected to database "test" as user "openpie".

test=# CREATE TABLE t1(id int);
NOTICE:  distribution policy forced to random for current relation with access method:
 janm
CREATE TABLE

test=# INSERT INTO t1 VALUES(1),(2),(3);
INSERT 0 3

test=# SELECT * FROM t1;
 id
----
  2
  1
  3
(3 rows)
```

4 管理 PieCloudDB

4.1 管理 PieCloudDB 概述

PieCloudDB 社区版镜像提供了 pdbcli 命令行工具来协助管理虚拟数仓，该工具已打包在社区版安装镜像（allInOne.tar.gz）中，不需要用户单独安装。

> **注意:**
>
> 在 Docker 容器中执行 pdbcli 命令时，需要用户具有 sudo 权限。

4.2 添加新虚拟数仓

pdbcli cluster create 命令用于创建并添加一个新的虚拟数仓。*--cluster-size* 用于指定虚拟数仓的执行器数量，*--tenant* 用于指定用户。需要注意的是，新虚拟数仓的端口是基于初始端口号逐一递增的（协调器的默认初始端口为 5433，三个执行器的默认初始端口为 6003、6004 和 6005）。

如下示例为，在 openpie 用户下创建一个拥有 3 个执行器的虚拟数仓。

```
sudo docker exec -it piedb pdbcli cluster create  --cluster-size 3

# 输出信息的示例
2024/08/08 09:11:47 Tenant openpie exist
2024/08/08 09:11:47 Allocate QD on coordinator
2024/08/08 09:11:47 running initdb for tenant openpie cluster 2
2024/08/08 09:11:48 initdb for tenant openpie cluster 2 finished
2024/08/08 09:11:48 PrepareSegment 2:-1 on coordinator
2024/08/08 09:11:48 PrepareSegment cluster 2 content 0 on segment
2024/08/08 09:11:48 PrepareSegment cluster 2 content 1 on segment
2024/08/08 09:11:48 PrepareSegment cluster 2 content 2 on segment
2024/08/08 09:11:48 cluster 2 registered
2024/08/08 09:11:48 qd: 127.0.0.1:5433
2024/08/08 09:11:48 Cluster created
2024/08/08 09:11:48 clusterid:2 port:6003 hostname:"127.0.0.1" address:"127.0.0.1"
 datadir:"/workspace/apps/pdb_data/ca-segment/openpie/2/6003"
2024/08/08 09:11:48 clusterid:2 contentid:1 port:6004 hostname:"127.0.0.1"
 address:"127.0.0.1" datadir:"/workspace/apps/pdb_data/ca-segment/openpie/2/6004"
2024/08/08 09:11:48 clusterid:2 contentid:2 port:6005 hostname:"127.0.0.1"
 address:"127.0.0.1" datadir:"/workspace/apps/pdb_data/ca-segment/openpie/2/6005"
2024/08/08 09:11:48 clusterid:2 contentid:-1 port:5433 hostname:"127.0.0.1"
 address:"127.0.0.1" datadir:"/workspace/apps/pdb_data/ca-coordinator/openpie/2/5433"
```

上述输出信息显示，新添加的虚拟数仓 ID 为 2，协调器（contentid 为 -1）将运行在 5433 端口，三个执行器（contentid 分别为 0、1 和 2）将运行在 6003、6004 和 6005 端口。

4.3 启动虚拟数仓

通过 *pdbcli cluster create* 命令添加的虚拟数仓并不会立即启动，只是完成了相关的初始化目录等操作。创建虚拟数仓之后需要使用 *pdbcli cluster start* 命令来启动。*--cluster* 用于指定虚拟数仓 ID。

如下示例为，启动 ID 为 2 的虚拟数仓。

```
sudo docker exec -it piedb pdbcli cluster start  --cluster 2

# 输出信息示例
2024/08/08 09:25:34 db 0 started by agent segment
2024/08/08 09:25:34 db 1 started by agent segment
2024/08/08 09:25:34 db 2 started by agent segment
2024/08/08 09:25:34 db -1 started by agent coordinator
```

4.4 停止虚拟数仓

pdbcli cluster stop 命令用于停止一个虚拟数仓，以便释放 CPU 与内存资源。

如下示例为，停止 ID 为 2 的虚拟数仓。

```
sudo docker exec -it piedb pdbcli cluster stop  --cluster 2

# 输出信息示例
2024/08/08 09:34:10 db 0 stopped by agent segment
2024/08/08 09:34:10 db -1 stopped by agent coordinator
2024/08/08 09:34:10 db 1 stopped by agent segment
2024/08/08 09:34:10 db 2 stopped by agent segment
```

4.5 修改虚拟数仓执行器数量

虚拟数仓可根据用户需求，通过增加或减少执行器来实现扩缩容（Scale Out/In），而虚拟数仓中的协调器保持不变，始终为 1。

使用 *pdbcli cluster resize* 命令可以修改虚拟数仓执行器的数量。*--cluster* 用于指定虚拟数仓 ID，*--cluster-size* 用于指定虚拟数仓的执行器数量。需要注意的是，在执行该命令修改虚拟数仓执行器数量之前需要保证目标虚拟数仓已经停止。

如下示例为将 ID 为 2 的虚拟数仓执行器数量设置为 4 个。

```
sudo docker exec -it piedb pdbcli cluster stop  --cluster 2
sudo docker exec -it piedb pdbcli cluster resize  --cluster 2 --cluster-size 4

# 输出信息示例
2024/08/08 09:40:25 PrepareSegment cluster 2 content 3 on segment
```

修改虚拟数仓执行器数量完成后，需要重新启动虚拟数仓。示例如下：

```
sudo docker exec -it piedb pdbcli cluster start  --cluster 2

# 输出信息示例
2024/08/08 09:42:28 db 0 started by agent segment
2024/08/08 09:42:28 db 1 started by agent segment
2024/08/08 09:42:28 db 2 started by agent segment
2024/08/08 09:42:28 db 3 started by agent segment
2024/08/08 09:42:28 db -1 started by agent coordinator
```

上述输出信息显示，该虚拟数仓有一个协调器（contentid 为 -1）和四个执行器（contentid 分别为 0、1、2 和 3）。

4.6 修改虚拟数仓配置参数

pdbcli cluster config 命令可以修改虚拟数仓的配置参数。*--cluster* 用于指定虚拟数仓 ID，*--name* 用于指定参数名称，*--value* 用于指定修改后的参数值。需要注意的是，在执行该命令修改参数值后需要重启虚拟数仓后才能生效。

如下示例为，将 ID 为 2 的虚拟数仓的 *pdb_parallel_factor* 参数值设置为“3”。

```
sudo docker exec -it piedb pdbcli cluster config  --cluster 2 --name  pdb_parallel_factor --
value 3
```

如果命令运行成功，则会输出：

```
Config success
```

该参数值被修改后需要重启虚拟数仓后生效，参考命令如下：

```
sudo docker exec -it piedb pdbcli cluster stop  --cluster 2
sudo docker exec -it piedb pdbcli cluster start  --cluster 2
```

连接数据库，通过 SHOW 语句检查变更是否生效：

```
openpie=# show pdb_parallel_factor;
 pdb_parallel_factor
---------------------
 3
(1 row)
```

4.7 查看虚拟数仓信息

pdbcli cluster list 命令用于查看当前所创建的虚拟数仓的信息。*--tenant* 用于指定用户。

如下示例为，查看默认用户 openpie 的全部虚拟数仓的信息。

```
sudo docker exec -it piedb pdbcli cluster list

# 输出信息的示例
clusterid:1 contentid:-1 port:5432 hostname:"127.0.0.1" address:"127.0.0.1" datadir:"/
workspace/apps/pdb_data/ca-coordinator/openpie/1/5432"
clusterid:2 contentid:-1 port:5433 hostname:"127.0.0.1" address:"127.0.0.1" datadir:"/
workspace/apps/pdb_data/ca-coordinator/openpie/2/5433"
```

如果需要查看指定虚拟数仓的详细信息，则需要在命令中指定 ID 信息。如下示例为，查看 ID 为 2 的虚拟数仓的详细信息。

```
sudo docker exec -it piedb pdbcli cluster list --cluster 2

# 输出信息的示例
clusterid:2 contentid:-1 port:5433 hostname:"127.0.0.1" address:"127.0.0.1" datadir:"/
workspace/apps/pdb_data/ca-coordinator/openpie/2/5433"
clusterid:2 contentid:0 port:6003 hostname:"127.0.0.1" address:"127.0.0.1" datadir:"/
workspace/apps/pdb_data/ca-segment/openpie/2/6003"
clusterid:2 contentid:1 port:6004 hostname:"127.0.0.1" address:"127.0.0.1" datadir:"/
workspace/apps/pdb_data/ca-segment/openpie/2/6004"
clusterid:2 contentid:2 port:6005 hostname:"127.0.0.1" address:"127.0.0.1" datadir:"/
workspace/apps/pdb_data/ca-segment/openpie/2/6005"
```

4.8 使用客户端连接虚拟数仓

目前 PieCloudDB 社区版支持 psql、DBeaver 等客户端连接虚拟数仓。

通过 Postgres Client PSQL 命令连接

通过 Postgres Client PSQL 命令连接虚拟数仓的命令格式如下：

```
psql -h 接入IP地址 -U 用户名 -p 端口号 -d 数据库名称
```

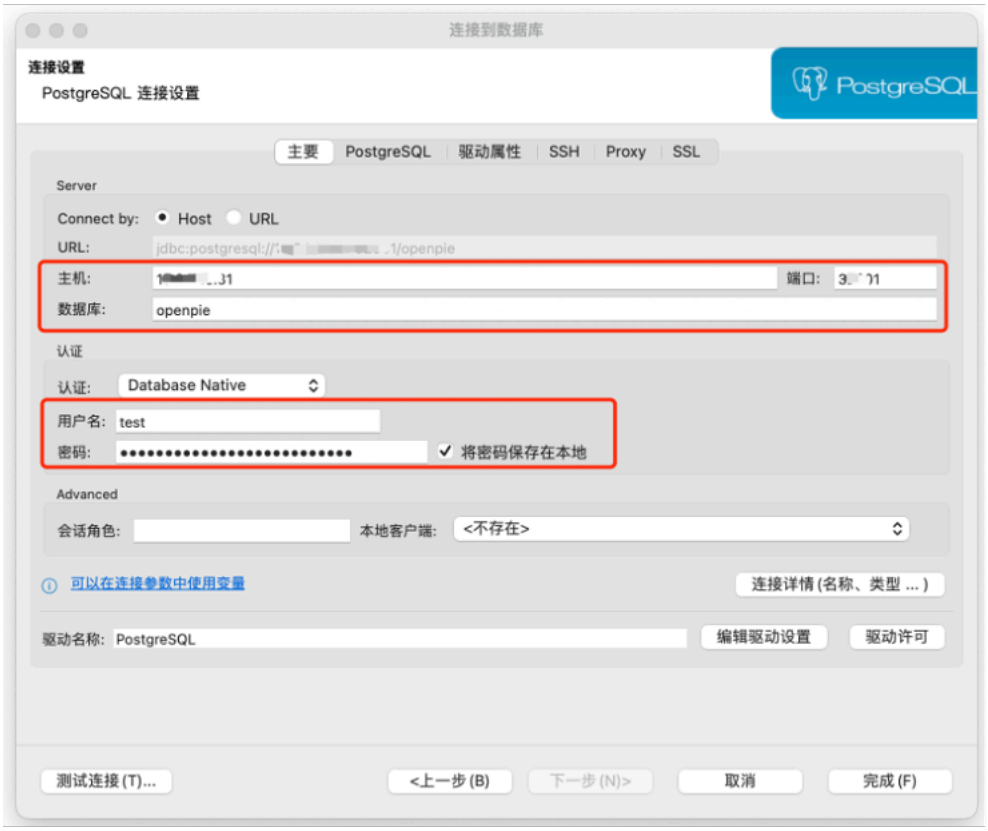
示例如下：

```
psql -h 192.x.x.x -U david -p 5432 -d openpie
```

通过 DBeaver 连接

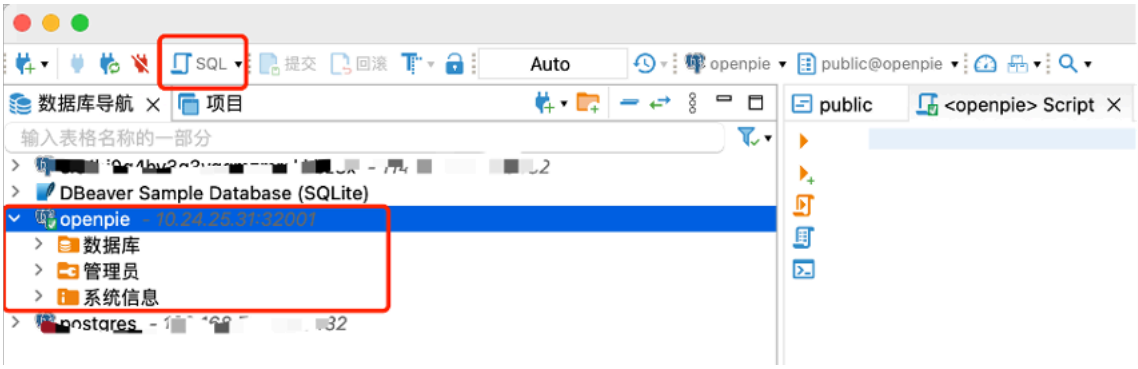
通过 DBeaver 连接虚拟数仓的步骤如下：

1. 进入 DBeaver 软件操作界面后，点击 **数据库** > **新建数据库连接**，选择 PostgreSQL，并设置主机 Host、数据库 Database、用户名 Username 和密码 Password。



2. 点击 **测试连接**，验证是否成功连接数据库。下图为连接成功示例。

3. 点击 **完成**，完成连接设置。
4. 点击 **SQL** 选择所设置的数据库连接，添加一个 SQL 文件。



5. 在所连接的“openpie”数据库中新建一张表 test_1，插入数据并查询验证。

```
CREATE TABLE test_1(id int);
INSERT INTO test_1 VALUES (1),(2),(3);

SELECT * FROM test_1;
```

运行结果如下图所示。

通过 Python psycopg2 连接

通过 Python psycopg2 连接的前提条件是当前 Python 环境已安装 psycopg2。连接示例如下：

```
import psycopg2
conn=psycopg2.connect(host="1**.***.***",
                      port="5432",
                      dbname="openpie",
                      user="openpie",
                      password="****",
                      sslmode="disable")
cursor=conn.cursor()
cursor.execute("select * from pg_roles limit 5")
result=cursor.fetchall()
```

5 管理数据库对象

5.1 管理数据库对象概述

用户在成功连接到 PieCloudDB 数据库后，可以利用 SQL 命令来执行各种查询任务。PieCloudDB 通常采用 PostgreSQL 的交互式终端“psql”来执行 SQL 命令。

本节内容主要基于“psql“命令行界面，介绍如何使用 SQL 语句来管理数据库对象。主要包括如下内容：

- 数据库对象访问控制
- 管理数据库
- 管理 Schema
- 管理表
- 管理分区表
- 管理序列
- 管理视图

5.2 数据库对象访问控制

数据库（Database）是在物理层面组织数据的集合。当需要使用不同的文字编码或排序规则来管理数据对象时，通常需要为每种不同的文字编码规则或排序规则创建一个独立的数据库。

角色属性

PieCloudDB 通过角色（Role）来管理对数据库的访问权限。角色可以代表单个数据库用户或用户组。角色不仅可以拥有对数据库对象（例如表和函数）的访问权限，而且还可以将其拥有的权限赋予给其他角色。

下表列出了数据库角色的属性，这些属性用于定义角色的权限以及与客户端进行身份验证。

属性	说明	SQL 语句
LOGIN	具有 LOGIN 属性的角色等同于数据库用户，它们用作连接数据库的初始用户。	*CREATE ROLE name LOGIN;* 或者 *CREATE USER name;*。*CREATE USER* 默认包含 *LOGIN*，等同于 *CREATE ROLE*。
SUPERUSER	可以将超级用户权限授予特定角色，使其能够绕过除登录权限之外的所有权限检查。由于超级用户具有极高的权限，应谨慎授予此权限。	*CREATE ROLE name SUPERUSER;*
CREATEDB	授予角色创建数据库的权限。	*CREATE ROLE name CREATEDB;*

属性	说明	SQL 语句
CREATEROLE	可以授予角色创建更多角色的权限。具有 CREATEROLE 属性的角色能够创建新角色，并可以更改和删除通过使用 ADMIN 选项并授予 CREATEROLE 属性的其他用户所创建的角色。当非超级用户且具有 CREATEROLE 属性的用户创建新角色时，新角色会自动被授予 CREATEROLE 属性，因此默认情况下，这些用户可以更改和删除他们所创建的角色。然而，具有 CREATEROLE 属性的角色无法创建具有 SUPERUSER 属性的角色。此外，CREATEROLE 属性不赋予创建具有 REPLICATION 属性的用户的能力，也不授予赋予或撤销 REPLICATION 权限的能力，以及修改此类用户角色属性的能力。尽管如此，CREATEROLE 属性允许用户对 REPLICATION 角色执行 *ALTER ROLE ... SET*、*ALTER ROLE ... RENAME* 操作，以及使用 *COMMENT ON ROLE*、*SECURITY LABEL ON ROLE* 和 *DROP ROLE* 命令。此外，CREATEROLE 属性也不包括授予或撤销 BYPASSRLS 权限的能力。	*CREATE ROLE name CREATEROLE;*
REPLICATION LOGIN	可以授予角色启动物理复制流程的权限。此外，该角色还需要具备 LOGIN 权限。	*CREATE ROLE name REPLICATION LOGIN;*
PASSWORD	在连接数据库时，只有当客户端身份验证要求用户提供密码的情况下，才需要为角色设置密码。	*CREATE ROLE name PASSWORD 'string';*
NOINHERIT	默认情况下，角色会继承其成员的权限。可以通过在创建角色时指定 NOINHERIT 属性来创建一个默认情况下不继承权限的角色。此外，可以使用 *WITH INHERIT TRUE* 或 *WITH INHERIT FALSE* 选项来覆盖对个别权限的继承设置。	*CREATE ROLE name NOINHERIT;*
BYPASSRLS	可以向角色授予权限，使其能够绕过行级安全（Row-Level Security, RLS）策略的限制，但这一权限不适用于超级用户。	*CREATE ROLE name BYPASSRLS;*
CONNECTION LIMIT	可以为角色指定允许建立的并发连接数。默认值“-1”表示没有并发连接的限制。	*CREATE ROLE name CONNECTION LIMIT 'integer';*

PieCloudDB 支持使用 *ALTER ROLE* 命令来更改角色的密码等属性。此外，还支持使用 *COMMENT* 和 *SECURITY LABEL* 命令对角色进行进一步的修改。如果需要删除针对特定角色的默认设置，可以使用 *ALTER ROLE role_name RESET var_name* 命令。

权限控制

数据库由多个 Schema 组成，而每个 Schema 又包含了表、函数等 SQL 对象。当数据库对象被创建时，它们会被分配一个所有者，这个所有者通常是执行创建操作的 SQL 命令的角色。在初始状态下，只有所有者或者超级用户能够对这些对象执行操作。其他角色在被分配了相应的权限之后，才能对这些 SQL 对象执行操作。

可以使用 *ALTER* 命令来更改数据库对象的所有者。通常，只有对象的所有者或者超级用户才有权限修改或销毁该对象。

如下语法示例为，使用 *ALTER* 命令将对象分配给新所有者：

```
ALTER TABLE table_name OWNER TO new_owner;
```

PieCloudDB 支持使用 *GRANT* 命令分配权限。如下示例为，授予角色 Alex 对表 sales 的更新权限。

```
GRANT UPDATE ON sales TO Alex;
```

可以使用 *REVOKE* 命令来撤销之前授予的权限。如下示例为，从“PUBLIC”角色或所有者中撤销对表 sales 的所有权限。

```
REVOKE ALL ON sales FROM PUBLIC;
```

通常情况下，只有数据库对象的所有者或超级用户才有权授予或撤销该对象的权限。在授予权限时，可以使用 *WITH GRANT OPTION* 子句来允许被授予权限的角色进一步将这些权限转授给其他角色。如果随后撤销了这个授予选项，则所有直接或间接从该角色获得的权限也将被一并撤销。

PieCloudDB 的访问控制列表（ACL）如下表所示。

权限	说明	缩写	适用对象
SELECT	• 允许从表、视图等对象的任意列或特定列执行 *SELECT*。 • 允许在 *UPDATE*、*DELETE* 或 *MERGE* 中引用现有列值。 • 允许序列使用 *currval()* 函数。 • 允许读取大对象类型数据。	r (read)	• Large Object • Sequence • Table • Table Column
INSERT	允许将新行插入到表、视图等。在特定列上授予权限时，则只有这些列可以通过 *INSERT* 命令分配新值（其他列将得到默认值）。	a (append)	• Table • Table Column

权限	说明	缩写	适用对象
UPDATE	• 允许更新表、视图等的所有列或特定列（*UPDATE* 命令通常也需要 *SELECT* 权限来确定要更新哪些行，或计算列的新值）。此外，*SELECT ... FOR UPDATE* 和 *SELECT ... FOR SHARE* 需要至少一列具有 *UPDATE* 和 *SELECT* 权限。 • 允许序列使用 *nextval()* 和 *setval()* 函数。 • 允许写入或截断大对象类型的数据。	w (write)	• Large Object • Sequence • Table • Table Column
DELETE	允许从表、视图等中删除行（*DELETE* 命令通常也需要 *SELECT* 权限来确定要删除哪些行）。	d	Table
TRUNCATE	允许对表执行 *TRUNCATE*。	D	Table
REFERENCES	允许创建表或表中特定列的外键约束。	x	• Table • Table Column
TRIGGER	允许在表、视图（等）上创建触发器。	t	Table
CREATE	• 允许在数据库中创建新 Schema 和 Publication，并允许在数据库中安装受信任的扩展。 • 允许在 Schema 内创建新对象。要重命名现有对象，必须拥有对该对象及包含该对象 Schema 的 *CREATE* 权限。 • 允许在表空间内创建表、索引和临时文件，并允许创建将该表空间作为默认表空间的数据库。请注意，撤销此权限不会改变现有对象的位置。	C	• Database • Schema • Tablespace
CONNECT	允许连接到数据库。在连接启动时会检查此权限（同时也会检查 *pg_hba.conf* 配置文件中的相关限制）。	c	Database
TEMPORARY	允许在使用数据库时创建临时表。	T	Database
EXECUTE	允许调用函数或过程。这是唯一适用于函数和过程的权限类型。	X	• Function • Procedure

权限	说明	缩写	适用对象
USAGE	• 允许创建 PL 函数。这是唯一适用于 PL 的权限类型。 • 允许访问 Schema 中的对象。如果没有此权限，仍然可以通过查询系统目录等方式查看对象名称。此外，撤销此权限后，现有会话可能具有先前执行过此查找的语句，此方法不能完全阻止对象访问。 • 允许序列使用 *currval()* 和 *nextval()* 函数。 • 允许在创建表、函数和其他 Schema 对象时使用 *TYPE* 或 *DOMAIN*。此权限的主要目的是控制用户可以创建对类型的依赖关系，以阻止所有者更改类型。 • 允许使用 FDW 创建新服务器。 • 允许使用外部服务器创建外部表。受权人还可以创建、更改或删除自己与该服务器关联的用户映射。	U	• Domain • Foreign Data Wrapper • Foreign Server • Language • Schema • Sequence • Type
SET	允许在当前会话中将服务器配置参数设置为新值。一般用于需要超级用户权限才能设置的参数。	s	Parameter
ALTER SYSTEM	允许使用 *ALTER SYSTEM* 命令将服务器配置参数设置为新值。	A	Parameter

默认情况下，在创建对象时会将某些类型的对象的权限授予“PUBLIC”，但不会向“PUBLIC”授予对表、表的列、序列、FDW、外部服务器、大对象、Schema、表空间或配置参数的任何权限。对于其他类型的对象，默认授予“PUBLIC”的权限如下：

- 数据库的 CONNECT 和 TEMPORARY（创建临时表）权限。
- 函数和过程的 EXECUTE 权限。
- 语言和数据类型（包括 DOMAIN）的 USAGE 权限。

对象的所有者通常可以撤销默认授予的权限以及明确授予的权限。建议在创建对象的同一事务中执行 *REVOKE* 命令。此外，可以使用 *ALTER DEFAULT PRIVILEGES* 命令来修改默认的权限设置。

下表总结了 SQL 对象的可用权限以及可用于检查权限设置的 psql 命令。

对象类型	所有权限	PUBLIC 默认权限	psql 命令
Database	CTc	Tc	\l
Domain	U	U	\dD+

对象类型	所有权限	PUBLIC 默认权限	psql 命令
Function 和 Procedure	X	X	\df+
Foreign Data Wrapper	U	无	\dew+
Foreign Server	U	无	\des+
Language	U	U	\dL+
Large Object	rw	无	\dl+
Parameter	sA	无	\dconfig+
Schema	UC	无	\dn+
Sequence	rwU	无	\dp
Table	arwdDxt	无	\dp
Table Column	arwx	无	\dp
Tablespace	C	无	\db+
Type	U	U	\dT+

5.3 管理数据库

PieCloudDB 提供了两个特殊的数据库：一个是模板数据库“template1”，另一个是默认数据库“openpie”。默认情况下，用户创建的每个新数据库都是以模板数据库为基础的。PieCloudDB 通常使用“template1”作为创建新数据库的模板，除非在创建时指定了其他模板。

用户首次连接到 PieCloudDB 时可以使用“openpie”数据库。PieCloudDB 将“openpie”用作管理连接的默认数据库。例如，启动进程、全局死锁检测器进程以及用于目录访问的 FTS（容错服务器）进程都默认使用“openpie”数据库。

创建数据库

CREATE DATABASE 命令用于创建数据库。

只有具有 CREATEDB 权限的角色或超级用户（默认为“openpie”）才能创建数据库。

创建一个新的数据库的命令如下：

```
CREATE DATABASE new_dbname [WITH OWNER[=]new_user ];
```

在创建数据库时，也可以指定数据库所有者。

如下示例为，在创建数据库 mydatabase01 时指定数据库所有者为 testuser01。

```
CREATE DATABASE mydatabase01 OWNER testuser01;
```

用户也可以使用客户端程序执行 *createdb* 创建数据库。如下示例为，在命令行终端运行以下命令，使用提供的主机名 coordinator_host 和端口 5432 连接到 PieCloudDB，并创建名为“mydatabase”的数据库：

```
$ createdb -h coordinator_host -p 5432 mydatabase
```

注意：

主机名和端口必须与已安装的 PieCloudDB 系统的主机名和端口匹配。

查看数据库列表

SELECT 命令用于从 *pg_database* 系统目录表中查询数据库列表。参考命令如下：

```
=> SELECT datname from pg_database;

datname
-----------
template1
openpie
(2 rows)
```

用户在使用 psql 客户端程序连接到 PieCloudDB 时，可以通过执行 *\l* 命令来显示系统中所有可用的数据库及其模板。这个命令提供了一个快速查看数据库列表的方法，包括数据库名称、所有者、编码方式和其他相关信息。

```
# \l

List of databases
Name | Owner | Encoding | Collate | Ctype | Access privileges
-----------+---------+----------+-------------+-------------+-------------------
openpie | openpie | UTF8 | en_HK.UTF-8 | en_HK.UTF-8 |
template1 | openpie | UTF8 | en_HK.UTF-8 | en_HK.UTF-8 |
(2 rows)
```

修改数据库

ALTER DATABASE 命令用于修改数据库。只有数据库所有者或超级用户才有权限修改数据库。

ALTER DATABASE 命令可以更改数据库的所有属性，包括所有者、名称或默认配置属性。相关命令如下：

```
ALTER DATABASE name RENAME TO new_name;

ALTER DATABASE name OWNER TO { new_owner | CURRENT_USER |SESSION_USER }

ALTER DATABASE name SET configuration_parameter { TO | = } { value| DEFAULT }
```

如下示例为，将数据库名称由 mydatabase 修改为 yourdatabase。

```
ALTER DATABASE mydatabase RENAME TO yourdatabase;
```

如下示例为，通过设置 *search_path* 配置参数来更改数据库的默认 Schema 搜索路径。

```
ALTER DATABASE mydatabase SET search_path TO myschema,public, pg_catalog;
```

删除数据库

DROP DATABASE 命令用于删除数据库。只有数据库所有者，或者具有 CREATEDB 权限的角色或超级用户才有权限删除数据库。

删除数据库的命令如下：

```
DROP DATABASE [IF EXISTS] db_name;
```

DROP DATABASE 命令会删除数据库的系统目录项，并删除包含数据的磁盘上的数据库目录，所以处于使用中数据库不能被删除。在删除当前数据库之前，请先切换并连接到其他数据库。

如果删除当前使用中的数据库，则发生报错，示例如下：

```
=> DROP DATABASE mydatabase;
ERROR: cannot drop the currently open database
```

用户还可以使用客户端程序执行 *dropdb* 命令删除数据库。

如下示例为，使用提供的主机名 coordinator_host 和端口 5432 连接到 PieCloudDB，并删除数据库 mydatabase。

```
$ dropdb -h coordinator_host -p 5432 mydatabase
```

注意：

删除数据库的操作是不可逆的。

5.4 管理 Schema

在数据库中，Schema 用于在逻辑上组织对象和数据。如果数据库对象分布在不同的 Schema 中，那么即使这些对象的名称相同，也不会在数据库中发生冲突。

每个数据库都包含一个名称为“public”的默认 Schema。如果用户没有显式创建 Schema，那么新的对象将默认创建在 public Schema 中。所有数据库角色（用户）默认拥有 public Schema 的 CREATE 和 USAGE 权限。当创建新的 Schema 时，需要授予用户相应的权限，以便他们能够访问该 Schema。

每个数据库都提供如下系统级别的 Schema：

- *pg_catalog*：包含了系统目录表、内置数据类型、函数和运算符。它是 Schema 搜索路径的一部分，即使在搜索路径中没有显式包含它。
- *information_schema*：由一组标准化的视图组成，这些视图提供了关于数据库中对象的信息。这些视图以标准化的方式从系统目录表中获取系统信息。

创建 Schema

CREATE SCHEMA 命令用于创建新 Schema。只有具有当前数据库的 CREATE 权限或超级用户（默认为“openpie”）才能创建 Schema。

创建一个新 Schema 的命令如下：

```
CREATE SCHEMA schema_name [AUTHORIZATION user_name];
```

创建 Schema 时可以指定归属者。如下示例为，在创建 Schema myschema01 时指定 Schema 的所有者为 testuser01。

```
CREATE SCHEMA myschema01 AUTHORIZATION testuser01;
```

使用 Schema 限定名称可以指定数据库中对象的位置，示例如下：

```
SELECT * FROM myschema01.mytable;
```

此外，*search_path* 配置参数可以指定可用 Schema 的搜索顺序。在搜索路径中指定的 Schema 为默认值。如果未指定，则所创建的对象将归属在默认 Schema 中。

ALTER DATABASE 命令可以用于设置搜索路径，示例如下：

```
ALTER DATABASE mydatabase SET search_path TO myschema, public, pg_catalog;
```

查看 Schema

使用 current_schema() 函数可以查看当前 Schema。示例如下：

```
=> SELECT current_schema();

current_schema
----------------
public
(1 row)
```

使用 *SHOW* 命令可以查看当前搜索路径。示例如下：

```
=> SHOW search_path;

search_path
----------------
"$user", public
(1 row)
```

删除 Schema

DROP SCHEMA 命令用于删除 Schema，此命令将删除 Schema 中的所有对象，包括表、视图、函数等，这是一个不可逆操作。只有 Schema 的所有者，或者具有 CREATEDB 权限的角色或超级用户才有权限删除 Schema。

删除一个 Schema 的示例如下：

```
DROP SCHEMA myschema;
```

默认情况下，所删除的 Schema 必须为空。但要删除一个 Schema 及其所有对象（表、数据、函数等）可以使用如下命令：

```
DROP SCHEMA myschema CASCADE;
```

5.5 管理表

创建表时需要考虑的参数包括列的数据类型、表和列的约束条件以及表的数据压缩方式。

创建表

CREATE TABLE 命令用于创建新表并定义其结构。

创建表需要用户具有 CREATE 权限以及所有列类型或 OF 子句中类型的 USAGE 权限。

如下示例为，创建表 distributors 并为不同的列指定了数据类型。

```
CREATE TABLE distributors (
         id INT,
         name VARCHAR(50) NOT NULL,
         address TEXT,
         phone VARCHAR(20)
        );
```

上述示例中表 distributors 定义了如下列并指定了数据类型：

- id：供应商的 ID，数据类型为整型。
- name：供应商名称，数据类型为字符串类型且不允许为 Null。
- address：供应商地址，数据类型为文本类型。

- phone：联系电话，数据类型为字符串类型。

列的数据类型决定了列可以包含的数据值的类型。选择合适的数据类型不仅可以有效地利用存储空间，还能更好地约束数据。例如，对于字符串类型的数据，通常会使用字符型数据类型；对于包含文本数据的列，常用的数据类型有 VARCHAR 或 TEXT。

在选择数值型数据类型时，建议选择能够容纳当前数据且为将来数据扩展留有余地的类型。例如，如果数据适合使用 INT 或 SMALLINT，那么使用 BIGINT 可能会导致存储空间的浪费。如果数据值预计会随时间增长，那么在加载大量数据后，再从较小的数据类型转换为较大的数据类型可能会付出较高的成本。例如，如果当前数据适合 SMALLINT，但预计未来可能会扩展，则选择 INT 更为合适。

此外，在跨表连接时，通常会在一张表中使用主键，在另一张表中使用外键。如果相关列的数据类型不一致，数据库在连接时必须进行数据类型转换，这会增加额外的性能开销。因此，建议在跨表连接中使用的列具有相同的数据类型。

设置表和列约束

PieCloudDB 支持 CHECK 约束、NOT NULL 约束和 EXCLUDE 约束，但不支持 UNIQUE、PRIMARY KEY 和 FOREIGN KEY 约束。

CHECK 约束用于指定某列中的值必须满足布尔（真值）表达式。如下示例中 *CHECK (price > 0)* 约束产品价格 price 为正数。

```
CREATE TABLE products
        ( productno INTEGER,
          name TEXT,
          price NUMERIC CHECK (price > 0)
         );
```

NOT NULL 约束用于指定列不能为 Null，且只能作为列约束。如下示例中列 productno 和 name 的值被约束为不能为 Null。

```
CREATE TABLE products
        ( productno INTEGER NOT NULL,
          name TEXT NOT NULL,
          price NUMERIC CHECK (price > 0)
        );
```

EXCLUDE 约束用于确保表中的数据满足指定的排除条件，通过比较表达式中的指定列或运算符并返回布尔值，来排除重复或冲突的数据。

如下示例中使用“gist”方法，并使用“&&”运算符来排除重叠数据。

```
CREATE TABLE circles
        (c circle,
         EXCLUDE USING gist (c WITH &&)
        ) DISTRIBUTED REPLICATED;
```

设置表数据压缩方式

PieCloudDB 使用 JANM 引擎存储用户数据，JANM 将数据按列进行压缩和存储，并且表的所有列采用同样的压缩方式。目前支持的表级压缩方式有 none、pglz 和 zstd，默认为 zstd。虽然数据压缩或解压会消耗 CPU 资源，但可以减少文件数量和磁盘占用。

CREATE TABLE 命令的 WITH 子句可以配置表的压缩方式。

如下示例为配置表的压缩方式为“pglz”。

```
CREATE TABLE foo (a INT, b TEXT) WITH (compresstype=pglz);
```

修改表定义

ALTER TABLE 命令用于修改表结构或定义，支持添加/删除列、添加/删除约束、重命名表和列等操作。只有表的所有者和超级用户才有权限修改表。

如下为修改表定义的示例：

- 示例 1：为表 distributors 添加列“address”。

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

- 示例 2：删除表 distributors 的列 address。

```
ALTER TABLE distributors DROP COLUMN address;
```

- 示例 3：重命名表 distributors 的列 address 为“city”。

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

- 示例 4：重命名表 distributors 为“suppliers”。

```
ALTER TABLE distributors RENAME TO suppliers;
```

- 示例 5：为表 distributors 的列 id 添加 NOT NULL 约束。

```
ALTER TABLE distributors ALTER COLUMN id SET NOT NULL;
```

删除表

DROP TABLE 命令用于删除表。只有表的所有者、表所在 Schema 的所有者和超级用户才有权限删除表。如果只清空表中的数据而不删除表，请使用 DELETE 或 TRUNCATE 命令。

DROP TABLE 会删除表中存在的所有规则、触发器和约束，如果要删除被视图引用的表，必须指定 *CASCADE*，相关的视图才会被完全删除。

如下示例为删除表 mytable。

```
DROP TABLE mytable;
```

5.6 管理分区表

PieCloudDB 通过分区机制将大型表按照特定标准划分为多个部分，这有助于实现大规模并行处理并提升查询性能。

分区操作会创建一个顶层的父表，并在它下面包含一层或多层子表。在数据库内部，PieCloudDB 会在顶层父表和其底层分区之间建立继承关系。系统目录会记录分区层级的信息，确保插入到顶层父表的数据能够被正确地分配到相应的子表分区中。

关于分区方式

PieCloudDB 目前支持如下分区方式：

- RANGE 分区：基于数值范围（例如日期或价格）对数据进行划分。这种分区方式适用于当数据的键值可以表示为一个连续范围时。
- LIST 分区：基于值列表（例如销售领域或产品线）对数据进行划分。这种分区方式适用于当数据的键值可以明确列出时。
- RANGE+LIST 分区：结合了 RANGE 分区和 LIST 分区的特点，允许更复杂的分区策略。

不是所有的表都适合进行分区。在考虑创建分区时，建议考虑以下因素：

- 表的大小

  如果表中包含数百万或数十亿条记录，将数据逻辑上划分为较小的块，分区可以提升性能。然而，对于只有几千行甚至更少数据的小表，维护分区的管理开销可能会超过潜在的性能提升。
- 查询性能

  如果针对单个表的查询响应时间较慢，考虑对该表进行分区。
- 数据的均匀划分

  如果分区能够包含大致相等数量的记录，查询性能将根据分区的数量得到提升。例如，将一个大型表划分为 10 个分区，理论上查询速度可以在满足特定查询条件下比未分区表快 10 倍。
- 查询谓词提供的信息

  分析用户查询中使用的 WHERE 子句，确定哪些列常用于访问数据。例如，如果大多数查询都基于日期进行记录查找，那么按月或周进行日期分区可能更合适。同样，如果用户经常根据地区访问记录，可以考虑使用 LIST 分区。
- 基于查询谓词实现分区排除

  查询优化器可以根据查询谓词排除某些分区，从而提高查询性能。如果查询无法实现分区排除，那么分区可能不会改善查询性能，甚至可能因为扫描每个分区而导致查询速度比未分区时更慢。使用查询 EXPLAIN 计划可以帮助确认分区是否被有效排除。
- 维护历史数据的需求

  例如，如果数据仓库需要保留过去十二个月的数据，按月分区可以方便地从仓库中删除最旧的月份分区，并将当前数据加载到最近的月份分区中。

注意:

分区数量不应超过实际需求。创建过多分区可能会增加管理和维护工作，例如清理、扩展集群、检查磁盘用量等。对于多级分区也要格外谨慎，因为如果分区文件数量迅速增长，就会增加系统的维护成本。

创建分区表

在使用 *CREATE TABLE* 命令创建表时，指定 *PARTITION BY* 以及可选的 *SUBPARTITION BY* 子句对表进行分区。本文将通过具体示例介绍如何创建 RANGE、LIST 和多级分区。

创建 RANGE 分区表示例

RANG 分区表一般使用单个 DATE 或者 TIMESTAMP 列作为分区键列，也可以使用同一个分区键列来创建子分区，例如按“月”分区然后按“日”创建子分区。

如下示例为通过指定 START 值、END 值以及 EVERY 子句，每隔 1 个月自动创建一个分区，一共创建了 12 个分区。默认情况下，START 值总是被 INCLUSIVE，而 END 值总是被 EXCLUSIVE。

```
CREATE TABLE sales01 (id INT, date DATE, amt DECIMAL(10,2))
           DISTRIBUTED BY (id)
           PARTITION BY RANGE (date)
               ( START (date '2022-01-01') INCLUSIVE
                 END (date '2023-01-01') EXCLUSIVE
                 EVERY (INTERVAL '1 month')
                );
```

如下示例为逐个声明和命名每一个分区。不需要为每一个分区声明一个 END 子句，只需要为最后一个分区声明即可。

```
CREATE TABLE sales02 (id INT, date DATE, amt DECIMAL(10,2))
           DISTRIBUTED BY (id)
           PARTITION BY RANGE (date)
               ( PARTITION Jan01 START (date '2022-01-01') INCLUSIVE ,
                 PARTITION Feb02 START (date '2022-02-01') INCLUSIVE ,
                 PARTITION Mar03 START (date '2022-03-01') INCLUSIVE ,
                 PARTITION Apr04 START (date '2022-04-01') INCLUSIVE ,
                 PARTITION May05 START (date '2022-05-01') INCLUSIVE ,
                 PARTITION Jun06 START (date '2022-06-01') INCLUSIVE ,
                 PARTITION Jul07 START (date '2022-07-01') INCLUSIVE ,
                 PARTITION Aug08 START (date '2022-08-01') INCLUSIVE ,
                 PARTITION Sep09 START (date '2022-09-01') INCLUSIVE ,
                 PARTITION Oct10 START (date '2022-10-01') INCLUSIVE ,
                 PARTITION Nov11 START (date '2022-11-01') INCLUSIVE ,
                 PARTITION Dec12 START (date '2022-12-01') INCLUSIVE
                 END (date '2023-01-01') EXCLUSIVE
                );
```

RANG 分区表也可以使用单个数字列作为分区键列，示例如下。

```
CREATE TABLE rank (id INT, rank INT, year INT, gender CHAR(1), count INT)
            DISTRIBUTED BY (id)
            PARTITION BY RANGE (year)
                ( START (2010) END (2023) EVERY (1),
                  DEFAULT PARTITION extra );
```

创建 LIST 分区表示例

在数据库系统中实现 LIST 分区时，可以选择任意支持等值比较的数据类型的列作为分区键列。这包括但不限于整数、字符串、日期等数据类型。

对于 LIST 分区，用户必须逐个声明和命名每一个分区，示例如下。

```
CREATE TABLE student (id INT, name TEXT NOT NULL, year INT, gender CHAR(1), count INT)
            DISTRIBUTED BY (id)
            PARTITION BY LIST (gender)
              ( PARTITION girls VALUES ('F'),
                PARTITION boys VALUES ('M'),
                DEFAULT PARTITION other );
```

创建多级分区表示例

多级分区表一般包含多个子分区，对于 RANGE 多级分区，为避免子分区冗余，建议增加分区的范围区间或者结合其他分区策略。子分区模板可以确保每一个分区都有相同的子分区定义。

如下示例创建了二级分区，其中表 sales03 被按照列 date 进行 RANGE 分区，之后又按照列 region 进行 LIST 分区。

```
CREATE TABLE sales03 (order_id INT, date DATE, amount DECIMAL(9,2), region TEXT)
            DISTRIBUTED BY (order_id)
            PARTITION BY RANGE (date)
                SUBPARTITION BY LIST (region)
                SUBPARTITION TEMPLATE
                    (  SUBPARTITION usa VALUES ('usa'),
                       SUBPARTITION asia VALUES ('asia'),
                       SUBPARTITION europe VALUES ('europe'),
                       DEFAULT SUBPARTITION other_regions)
                          (START (date '2012-01-01') INCLUSIVE
                           END (date '2023-01-01') EXCLUSIVE
                           EVERY (INTERVAL '1 month'),
                           DEFAULT PARTITION outlying_dates
                          );
```

如下示例创建了三级分区，其中表 sales04 先后按照列 year 和列 month 进行 RANGE 二级分区，最后又按照列 region 进行 LIST 分区，并且每一级分区上都声明了一个 DEFAULT 分区。

```
CREATE TABLE sales04 (order_id INT, year INT, month INT, day INT, amount DECIMAL(9,2),
 region TEXT)
          DISTRIBUTED BY (order_id)
          PARTITION BY RANGE (year)
             SUBPARTITION BY RANGE (month)
             SUBPARTITION TEMPLATE (
                 START (1) END (13) EVERY (1),
                 DEFAULT SUBPARTITION other_months )
                    SUBPARTITION BY LIST (region)
                        SUBPARTITION TEMPLATE (
                            SUBPARTITION usa VALUES ('usa'),
                            SUBPARTITION europe VALUES ('europe'),
                            SUBPARTITION asia VALUES ('asia'),
                            DEFAULT SUBPARTITION other_regions )
           ( START (2012) END (2023) EVERY (1),
             DEFAULT PARTITION outlying_years );
```

对已有表进行分区

CREATE TABLE...PARTITION OF 命令可以用于对已有的分区表进行分区。

如下示例为，先创建表 orders 并进行 RANGE 分区，然后创建表 orders_201801 作为表 orders 的分区，并将 2018 年一月份的数据存储到该分区中。

```
CREATE TABLE orders (
              order_date DATE NOT NULL,
              geo VARCHAR(2) NOT NULL,
              vender VARCHAR(5)  DEFAULT 0 NOT NULL,
              sales INTEGER DEFAULT 0 NOT NULL
              )PARTITION BY RANGE (order_date);

CREATE TABLE orders_201801
          PARTITION OF orders
          FOR VALUES FROM ('2018-01-01') TO ('2018-01-31');
```

对表 orders_201801 也可以再进行 LIST 分区，并根据不同的地理位置创建 LIST 分区，示例如下。

```
CREATE TABLE orders_201801
          PARTITION OF orders
          FOR VALUES FROM ('2018-01-01') TO ('2018-01-31')
          PARTITION BY LIST (geo);

CREATE TABLE orders_UK_201801 PARTITION OF orders_201801 FOR VALUES IN ('UK');

CREATE TABLE orders_US_201801 PARTITION OF orders_201801 FOR VALUES IN ('US');

CREATE TABLE orders_AU_201801 PARTITION OF orders_201801 FOR VALUES IN ('AU');
```

如果对已有的非分区表进行分区，可以首先创建一个新分区表，然后把原始表的数据载入到新表，再删除原始表并且把新分区表重命名为原始表。需要注意的是，由于原始表被删除，需要重新授予表对象的相关权限。

如下示例中，表 sales 为非分区表，需要通过上述方式将其进行分区并重新授予权限。

```
CREATE TABLE sales_temp (LIKE sales)
          PARTITION BY RANGE (date)
             ( START (date '2022-01-01') INCLUSIVE
               END (date '2023-01-01') EXCLUSIVE
               EVERY (INTERVAL '1 month')
             );

INSERT INTO sales_temp SELECT * FROM sales;

DROP TABLE sales;

ALTER TABLE sales_temp RENAME TO sales;

GRANT ALL PRIVILEGES ON sales TO admin;

GRANT SELECT ON sales TO guest;
```

此外，使用 *ALTER TABLE...ADD PARTITION* 命令可以为一个一级分区表增加一个分区。示例如下：

```
ALTER TABLE sales01 ADD PARTITION
        START (date '2023-02-01') INCLUSIVE
        END (date '2023-03-01') EXCLUSIVE;
```

使用 *ALTER TABLE...ADD DEFAULT PARTITION* 命令也可以为一级分区表增加一个默认分区。示例如下：

```
ALTER TABLE sales02 ADD DEFAULT PARTITION other;
```

默认分区可以确保将不匹配的数据插入到默认分区中。如果没有默认分区，当插入的数据不匹配一个分区的 CHECK 约束时，则会被拒绝插入该数据。

加载分区表

向一个已分区的表中插入数据时，用户需要指定根分区表，即使用 *CREATE TABLE* 命令创建的表。

在一个多级分区中，只有层级底部的子分区能够包含数据。在用户创建了分区表之后，顶层父表为空，当使用 *COPY* 或者 *INSERT* 命令加载数据到父表时，数据会被路由到底层的子表分区中。

建议为分区层次定义一个 DEFAULT 分区，这样可以使任何不匹配某个分区 CHECK 约束的数据加载到 DEFAULT 分区，而不会发生加载失败。需要注意的是，查询优化器总会扫描 DEFAULT 分区，所以包含数据的 DEFAULT 分区会拖慢总体扫描时间。

向分区表中加载数据的最佳方法是创建一个中间状态表，加载数据后，再把这个中间状态表交换到用户的分区中。详细信息请参见 **交换分区**。

验证分区策略

PieCloudDB 使用创建表时定义的分区标准来创建每一个分区，并通过 CHECK 约束限制该表的分区允许包含的数据，查询优化器也会使用 CHECK 约束来决定要扫描的表分区。

当分区表使用查询谓词作为分区键列时，用户可以使用 EXPLAIN 命令来验证查询优化器是否只扫描相关的数据并满足给定的查询谓词。

如下示例为，先将表 sales03 按日期分区，再按地区进行子分区，执行如下查询计划：

```
EXPLAIN SELECT * FROM sales03 WHERE date='01-07-23' AND region=' asia ';
```

对于此查询，查询计划只展示涉及到的扫描为“返回若干行的 ASIA 地区子分区（sales03_1_prt_outlying_dates_2_prt_asia）”。

```
=> EXPLAIN SELECT * FROM sales03 WHERE date='01-07-23' AND region=' asia ';

                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..168.02 rows=1 width=54)
   ->  Seq Scan on sales03_1_prt_outlying_dates_2_prt_other_regions  (cost=0.00..168.00
 rows=1 width=54)
         Filter: ((date = '2023-01-07'::date) AND (region = ' asia '::text))
 Optimizer: Postgres query optimizer
(4 rows)
```

查看分区设计

pg_inherits 视图用于查看相关分区设计的信息。如下示例为，查看表 sales01 的分区设计。

```
=> SELECT inhrelid::regclass AS child, inhparent::regclass AS parent
     FROM pg_inherits WHERE inhparent = 'sales01'::regclass::oid;

     child     | parent
-------------------+--------
 sales01_1_prt_1  | sales01
 sales01_1_prt_2  | sales01
 sales01_1_prt_3  | sales01
 sales01_1_prt_4  | sales01
 sales01_1_prt_5  | sales01
 sales01_1_prt_6  | sales01
 sales01_1_prt_7  | sales01
 sales01_1_prt_8  | sales01
 sales01_1_prt_9  | sales01
 sales01_1_prt_10 | sales01
 sales01_1_prt_11 | sales01
 sales01_1_prt_12 | sales01
(12 rows)
```

此外，如下视图也可以展示分区表的相关信息。

- *gp_partition_template*：记录分区表的分区格式定义信息。
- *pg_partitioned_table*：记录分区表的分区键列等元数据信息。

修改分区表

要更改分区设计或者表结构，可对顶层父表使用 *ALTER TABLE* 命令并指定 PARTITION 子句（*ALTER TABLE...PARTITION*）。分区并不要求有名称，如果一个分区没有名称则可以使用 *PARTITION FOR(value)* 表达式来指定。

常见的分区修改操作包括：重命名分区、增加默认分区、删除分区、交换分区、分裂分区、修改子分区模板和用外部表交换叶子子分区等。本文将通过具体示例介绍如何修改分区。

重命名分区

分区命名的语法格式如下：

```
<parentname>_<level>_prt_<partition_name>
```

如下示例为，表 sales02 的一级分区名称为 sales02_1_prt_jan01。

```
sales02_1_prt_jan01
```

如果是自动生成的 RANGE 分区，如果没有指定分区名称，则系统会自动分配数字用于命名分区。如下示例中，为表 sales01 的所有一级分区自动分配了数字作为分区名称。

```
sales01_1_prt_1 FOR VALUES FROM ('2022-01-01') TO ('2022-02-01'),
sales01_1_prt_10 FOR VALUES FROM ('2022-10-01') TO ('2022-11-01'),
sales01_1_prt_11 FOR VALUES FROM ('2022-11-01') TO ('2022-12-01'),
sales01_1_prt_12 FOR VALUES FROM ('2022-12-01') TO ('2023-01-01'),
sales01_1_prt_2 FOR VALUES FROM ('2022-02-01') TO ('2022-03-01'),
sales01_1_prt_3 FOR VALUES FROM ('2022-03-01') TO ('2022-04-01'),
sales01_1_prt_4 FOR VALUES FROM ('2022-04-01') TO ('2022-05-01'),
sales01_1_prt_5 FOR VALUES FROM ('2022-05-01') TO ('2022-06-01'),
sales01_1_prt_6 FOR VALUES FROM ('2022-06-01') TO ('2022-07-01'),
sales01_1_prt_7 FOR VALUES FROM ('2022-07-01') TO ('2022-08-01'),
sales01_1_prt_8 FOR VALUES FROM ('2022-08-01') TO ('2022-09-01'),
sales01_1_prt_9 FOR VALUES FROM ('2022-09-01') TO ('2022-10-01')
```

要重命名一个已分区的子表，应重命名顶层父表，在所有相关的子表分区的表名中的 *<parentname>* 字段都会改变。如果执行如下命令，将表的名称由 sales01 改为 globalsales，则相应的分区名称也会改变。

```
=> ALTER TABLE sales01 RENAME TO globalsales;

=>\d+ globalsales;

                    Partitioned table "public.globalsales"
 Column |    Type     | Collation | Nullable | Default | Storage | Stats target | Description
--------+-------------+-----------+----------+---------+---------+--------------+-------
 id     | integer     |           |          |         | plain   |              |
 date   | date        |           |          |         | plain   |              |
 amt    | numeric(10,2) |         |          |         | main    |              |
Partition key: RANGE (date)
Partitions: globalsales_1_prt_1 FOR VALUES FROM ('2022-01-01') TO ('2022-02-01'),
            globalsales_1_prt_10 FOR VALUES FROM ('2022-10-01') TO ('2022-11-01'),
            globalsales_1_prt_11 FOR VALUES FROM ('2022-11-01') TO ('2022-12-01'),
            globalsales_1_prt_12 FOR VALUES FROM ('2022-12-01') TO ('2023-01-01'),
            globalsales_1_prt_2 FOR VALUES FROM ('2022-02-01') TO ('2022-03-01'),
            globalsales_1_prt_3 FOR VALUES FROM ('2022-03-01') TO ('2022-04-01'),
            globalsales_1_prt_4 FOR VALUES FROM ('2022-04-01') TO ('2022-05-01'),
            globalsales_1_prt_5 FOR VALUES FROM ('2022-05-01') TO ('2022-06-01'),
            globalsales_1_prt_6 FOR VALUES FROM ('2022-06-01') TO ('2022-07-01'),
            globalsales_1_prt_7 FOR VALUES FROM ('2022-07-01') TO ('2022-08-01'),
            globalsales_1_prt_8 FOR VALUES FROM ('2022-08-01') TO ('2022-09-01'),
            globalsales_1_prt_9 FOR VALUES FROM ('2022-09-01') TO ('2022-10-01')
Distributed randomly
```

使用 *ALTER TABLE...RENAME PARTITION FOR* 命令可以更改一个分区的名称。例如，将分区 globalsales_1_prt_1 的名称更改为 globalsales_1_prt_jan01。

```
ALTER TABLE globalsales RENAME PARTITION FOR ('2022-01-01') TO jan01;
```

通过上述示例可以得出，在使用 *ALTER TABLE* 命令修改分区表时，引用表时只需要使用分区名（例如 jan01）而不必引用完整的表名（例如 sales_1_prt_jan01）。

删除分区

ALTER TABLE...DROP PARTITION 命令用于删除一个分区表的分区。

当用户删除一个具有子分区的分区时，子分区（以及其中的所有数据）也会被自动删除。对于 RANGE 分区，删除较旧分区的操作很常见。示例如下：

```
ALTER TABLE globalsales DROP PARTITION jan01;
```

交换分区

ALTER TABLE...EXCHANGE PARTITION 命令用于交换一个分区，即用一个表换掉一个现有的分区。只有包含数据的分区才可以被交换，不可以交换包含复制表的分区，也不支持使用分区表或分区表的子分区交换分区。

分区交换可用于数据重载。如下示例为，加载一个分区表 sales 并且把加载好的表交换到可用分区中。

```
=>\d+ sales;

                        Partitioned table "public.sales"
 Column |  Type        | Collation | Nullable | Default | Storage | Stats target | Description
--------+--------------+-----------+----------+---------+---------+--------------+-------------
 id     | integer      |           |          |         | plain   |              |
 date   | date         |           |          |         | plain   |              |
 amt    | numeric(10,2)|           |          |         | main    |              |
Partition key: RANGE (date)
Partitions: sales_1_prt_1 FOR VALUES FROM ('2022-01-01') TO ('2022-02-01'),
            sales_1_prt_10 FOR VALUES FROM ('2022-10-01') TO ('2022-11-01'),
            sales_1_prt_11 FOR VALUES FROM ('2022-11-01') TO ('2022-12-01'),
            sales_1_prt_12 FOR VALUES FROM ('2022-12-01') TO ('2023-01-01'),
            sales_1_prt_13 FOR VALUES FROM ('2023-02-01') TO ('2023-03-01'),
            sales_1_prt_2 FOR VALUES FROM ('2022-02-01') TO ('2022-03-01'),
            sales_1_prt_3 FOR VALUES FROM ('2022-03-01') TO ('2022-04-01'),
            sales_1_prt_4 FOR VALUES FROM ('2022-04-01') TO ('2022-05-01'),
            sales_1_prt_5 FOR VALUES FROM ('2022-05-01') TO ('2022-06-01'),
            sales_1_prt_6 FOR VALUES FROM ('2022-06-01') TO ('2022-07-01'),
            sales_1_prt_7 FOR VALUES FROM ('2022-07-01') TO ('2022-08-01'),
            sales_1_prt_8 FOR VALUES FROM ('2022-08-01') TO ('2022-09-01'),
            sales_1_prt_9 FOR VALUES FROM ('2022-09-01') TO ('2022-10-01')
Distributed randomly

=> CREATE TABLE jan01 (LIKE sales);
=> INSERT INTO sales values(1,'01-02-22',10000),(2,'02-03-22',20000),(3,'03-03-22',30000);
=> INSERT INTO jan01 SELECT * FROM sales_1_prt_1 ;
=> ALTER TABLE sales EXCHANGE PARTITION FOR (DATE '2022-01-01')  WITH TABLE jan01;
```

如果指定了 *WITHOUT VALIDATION* 子句，用户必须确保用于交换的表中的数据对于该分区上的约束是合法的，否则将返回错误。

分裂分区

ALTER TABLE...SPLIT PARTITION 命令用于分裂分区，即把一个分区划分成两个分区，用户指定的分裂值会被分配在后面的分区中。只能在最低层级的分区做分裂（包含数据的分区）。对于多级分区，只能拆分 RANGE 分区，而不能拆分 LIST 分区。

如下示例为，把表 sales 一个月度分区分裂成两个，第一个分区包含日期“January 1-15”而第二个分区包含日期“January 16-31”。

```
ALTER TABLE sales SPLIT PARTITION FOR ('2022-01-01') AT ('2022-01-16') INTO (PARTITION
 jan171to15, PARTITION jan1716to31);
```

如果分区中包含一个默认分区，则必须通过分裂该默认分区来增加分区。在使用 INTO 子句时，需要指定当前的默认分区作为第二个分区名。

如下示例中，首先查看表 sales02 的当前分区信息，之后通过分裂表 sales02 的默认 RANGE 分区来为“January 2023”增加一个新的月度分区，新增的分区名称为“sales02_1_prt_jan23 FOR VALUES FROM ('2023-01-01') TO ('2023-02-01')”。

```
=> \d+ sales02

                        Partitioned table "public.sales02"
 Column |     Type      | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------------+-----------+----------+---------+---------+--------------+------
 id     | integer       |           |          |         | plain   |              |
 date   | date          |           |          |         | plain   |              |
 amt    | numeric(10,2) |           |          |         | main    |              |
Partition key: RANGE (date)
Partitions: sales02_1_prt_apr04 FOR VALUES FROM ('2022-04-01') TO ('2022-05-01'),
            sales02_1_prt_aug08 FOR VALUES FROM ('2022-08-01') TO ('2022-09-01'),
            sales02_1_prt_dec12 FOR VALUES FROM ('2022-12-01') TO ('2023-01-01'),
            sales02_1_prt_feb02 FOR VALUES FROM ('2022-02-01') TO ('2022-03-01'),
            sales02_1_prt_jan01 FOR VALUES FROM ('2022-01-01') TO ('2022-02-01'),
            sales02_1_prt_jul07 FOR VALUES FROM ('2022-07-01') TO ('2022-08-01'),
            sales02_1_prt_jun06 FOR VALUES FROM ('2022-06-01') TO ('2022-07-01'),
            sales02_1_prt_mar03 FOR VALUES FROM ('2022-03-01') TO ('2022-04-01'),
            sales02_1_prt_may05 FOR VALUES FROM ('2022-05-01') TO ('2022-06-01'),
            sales02_1_prt_nov11 FOR VALUES FROM ('2022-11-01') TO ('2022-12-01'),
            sales02_1_prt_oct10 FOR VALUES FROM ('2022-10-01') TO ('2022-11-01'),
            sales02_1_prt_sep09 FOR VALUES FROM ('2022-09-01') TO ('2022-10-01'),
            sales02_1_prt_other DEFAULT
Distributed randomly

=> ALTER TABLE sales02 SPLIT DEFAULT PARTITION START ('2023-01-01') INCLUSIVE END
 ('2023-02-01') EXCLUSIVE INTO (PARTITION jan23, default partition);

=> \dt sales02*

                 List of relations
 Schema |       Name        |       Type        |  Owner  | Storage
--------+-------------------+-------------------+---------+---------
 public | sales02           | partitioned table | openpie |
 public | sales02_1_prt_apr04 | table           | openpie |
 public | sales02_1_prt_aug08 | table           | openpie |
 public | sales02_1_prt_dec12 | table           | openpie |
 public | sales02_1_prt_feb02 | table           | openpie |
 public | sales02_1_prt_jan01 | table           | openpie |
 public | sales02_1_prt_jan23 | table           | openpie |
 public | sales02_1_prt_jul07 | table           | openpie |
 public | sales02_1_prt_jun06 | table           | openpie |
 public | sales02_1_prt_mar03 | table           | openpie |
 public | sales02_1_prt_may05 | table           | openpie |
 public | sales02_1_prt_nov11 | table           | openpie |
 public | sales02_1_prt_oct10 | table           | openpie |
 public | sales02_1_prt_other | table           | openpie |
 public | sales02_1_prt_sep09 | table           | openpie |
(15 rows)
```

修改子分区模版

ALTER TABLE SET SUBPARTITION TEMPLATE 命令用于修改一个分区表的子分区模板。在设置了新子分区模板之后，新增分区会依照新的分区设计，但现有的分区不会被改变。

如下示例为，创建表 sales04 并修改其子分区模板。

```
=> CREATE TABLE sales04 (order_id INT, date DATE, amount DECIMAL(9,2), region TEXT)
          DISTRIBUTED BY (order_id)
          PARTITION BY RANGE (date)
             SUBPARTITION BY LIST (region)
             SUBPARTITION TEMPLATE
                   ( SUBPARTITION usa VALUES ('usa'),
                     SUBPARTITION asia VALUES ('asia'),
                     SUBPARTITION europe VALUES ('europe'),
                     DEFAULT SUBPARTITION other_regions )
                    ( START (date '2022-01-01') INCLUSIVE
                     END (date '2022-04-01') EXCLUSIVE
                     EVERY (INTERVAL '1 month') );

=> ALTER TABLE sales04 SET SUBPARTITION TEMPLATE
         ( SUBPARTITION usa VALUES ('usa'),
           SUBPARTITION asia VALUES ('asia'),
           SUBPARTITION europe VALUES ('europe'),
           SUBPARTITION africa VALUES ('africa'),
           DEFAULT SUBPARTITION regions );
```

当表 sales 增加一个日期 RANGE 分区时，则会包括非洲的新地区列表子分区。如下示例为，创建子分区“usa”、“asia”、“europe”、“africa”以及一个名为“other”的默认分区。

```
ALTER TABLE sales04 ADD PARTITION "4" START ('2022-04-01') INCLUSIVE END ('2022-05-01')
 EXCLUSIVE ;
```

可以使用命令 `\dt sales04*` 查看为表 sales04 新创建的分区表的信息。

```
=> \dt sales04*

                      List of relations
 Schema |           Name            |     Type      |  Owner  | Storage
--------+---------------------------------------+-------------------+---------+---------
 public | sales04                   | partitioned table | openpie |
 public | sales04_1_prt_1           | partitioned table | openpie |
 public | sales04_1_prt_1_2_prt_asia        | table         | openpie |
 public | sales04_1_prt_1_2_prt_europe      | table         | openpie |
 public | sales04_1_prt_1_2_prt_other_regions | table         | openpie |
 public | sales04_1_prt_1_2_prt_usa         | table         | openpie |
 public | sales04_1_prt_2           | partitioned table | openpie |
 public | sales04_1_prt_2_2_prt_asia        | table         | openpie |
 public | sales04_1_prt_2_2_prt_europe      | table         | openpie |
 public | sales04_1_prt_2_2_prt_other_regions | table         | openpie |
 public | sales04_1_prt_2_2_prt_usa         | table         | openpie |
 public | sales04_1_prt_3           | partitioned table | openpie |
 public | sales04_1_prt_3_2_prt_asia        | table         | openpie |
 public | sales04_1_prt_3_2_prt_europe      | table         | openpie |
 public | sales04_1_prt_3_2_prt_other_regions | table         | openpie |
 public | sales04_1_prt_3_2_prt_usa         | table         | openpie |
 public | sales04_1_prt_4           | partitioned table | openpie |
 public | sales04_1_prt_4_2_prt_africa      | table         | openpie |
 public | sales04_1_prt_4_2_prt_asia        | table         | openpie |
 public | sales04_1_prt_4_2_prt_europe      | table         | openpie |
 public | sales04_1_prt_4_2_prt_regions     | table         | openpie |
 public | sales04_1_prt_4_2_prt_usa         | table         | openpie |
(22 rows)
```

使用命令 *SET SUBPARTITION TEMPLATE()* 可以移除一个子分区模板。如下示例为，清除表 sales04 的子分区模板：

```
ALTER TABLE sales04 SET SUBPARTITION TEMPLATE ();
```

用外部表交换叶子子分区

一个可读的外部表可以用来交换分区表中的一个叶子子分区。外部表数据可以位于主机文件系统、NFS 挂载或者 Hadoop 文件系统（HDFS）上。

例如，如果用户创建了一个按月分区的分区表，并且对该表的大多数查询是只访问最新的数据，由此用户可以把较旧的、较少访问的数据拷贝到外部表，并且把较旧的分区与这些外部表进行交换。如果分区表包含具有 CHECK 约束或 NOT NULL 约束的列，则不支持将叶子子分区与外部表进行交换。

如下示例为，分区表 sales05 包含 2015 至 2022 年份的数据，将分区表 sales05 的一个叶子子分区交换为一个外部表。

```
=> CREATE TABLE sales05 (id INT, year INT, qtr INT, day INT, region TEXT)
          DISTRIBUTED BY (id)
          PARTITION BY RANGE (year)
          ( PARTITION yy START (2015) END (2022) EVERY (1) );

=> \dt sales05*

                  List of relations
 Schema |      Name      |      Type      | Owner | Storage
--------+----------------+----------------+---------+---------
 public | sales05        | partitioned table | openpie |
 public | sales05_1_prt_yy_1 | table          | openpie |
 public | sales05_1_prt_yy_2 | table          | openpie |
 public | sales05_1_prt_yy_3 | table          | openpie |
 public | sales05_1_prt_yy_4 | table          | openpie |
 public | sales05_1_prt_yy_5 | table          | openpie |
 public | sales05_1_prt_yy_6 | table          | openpie |
 public | sales05_1_prt_yy_7 | table          | openpie |
(8 rows)
```

该分区表有 7 个叶子子分区，每一个叶子子分区含有单一年份的数据，例如，叶子子分区表 sales05_1_prt_yy_1 包含 2015 年的数据。参考如下步骤将表 sales05_1_prt_yy_1 交换为一个外部表：

1. 启用外部表协议。

   ```
   $ gpfdist
   ```

2. 创建可写的外部表，使用和分区表相同的列创建一个可写的外部表。

   CREATE WRITABLE EXTERNAL TABLE 命令用于创建外部表。

   ```
   CREATE WRITABLE EXTERNAL TABLE my_sales05_ext ( LIKE sales05_1_prt_yy_1 )
          LOCATION ( 'gpfdist://hostname/sales_2015' )
          FORMAT 'csv'
          DISTRIBUTED BY (id) ;
   ```

3. 创建一个可读的外部表，从步骤 2 所创建的可写外部表的目的地址读取数据。

使用 *CREATE EXTERNAL TABLE* 命令创建一个可读外部表，并使用和可写外部数据相同的外部数据。

```
CREATE EXTERNAL TABLE sales05_2015_ext ( LIKE sales05_1_prt_yy_1)
    LOCATION ( 'gpfdist://hostname/sales_2015' )
    FORMAT 'csv';
```

4. 从叶子子分区中拷贝数据到可写外部表 my_sales05_ext。

```
INSERT INTO my_sales05_ext SELECT * FROM sales05_1_prt_yy_1;
```

5. 使用外部表 my_sales05_ext 交换现有的叶子子分区。

   ALTER TABLE 命令指定了 *EXCHANGE PARTITION* 子句来切换可读外部表和叶子子分区，将表 sales05_1_prt_yy_1 换为叶子子分区的外部表，并且旧的叶子子分区换为表 sales05_2015_ext。

```
ALTER TABLE sales05 EXCHANGE PARTITION yy_1 WITH TABLE sales05_2015_ext;
```

   注意：

   为了确保针对分区表的查询返回正确的结果，外部表数据必须针对叶子子分区上的 CHECK 约束有效。在这种情况下，会从其上定义有 CHECK 约束的叶子子分区表中读取数据。

6. 删除弃用的分区表。

```
DROP TABLE sales05_2015_ext;
```

用户可以重命名该叶子子分区的名称来标识一个外部表。如下示例中，将分区 yy_1 重命名为 yy_1_ext，即把叶子子分区表的名称改为 sales05_1_prt_yy_1_ext。

```
ALTER TABLE sales05 RENAME PARTITION yy_1 TO yy_1_ext;
```

5.7 管理序列

序列是一种特殊的单行对象，用于为添加到表中的行生成唯一的整数标识符。

序列对象的属性包括序列名称、增量值以及序列计数器的最后值、最小值和最大值。PieCloudDB 提供了内置函数 nextval() 来获取序列的下一个值，和 setval() 来设置序列的特定起始值。

创建序列

CREATE SEQUENCE 命令用于创建并初始化序列。创建序列时需要指定的序列名称，序列名称必须不同于同一 Schema 中任何其他序列、表、索引或视图的名称。示例如下：

```
CREATE SEQUENCE myserial START 10;
```

创建新序列时，*START* 用于指定起始值，该关键字是可选的。使用 *SELECT* 命令可以检查序列的当前属性，示例如下：

```
=> SELECT * FROM myserial;

last_value | log_cnt | is_called
------------+---------+-----------
10 | 0 | f
(1 row)
```

序列还具有一个特殊的布尔属性 *is_called*，它管理着序列计数器的自动递增行为：

- 当序列的 *is_called* 属性设置为 TRUE 时，调用 nextval() 会在返回值之前递增序列计数器。
- 当序列的 *is_called* 属性设置为 FALSE 时，调用 nextval() 会在返回值之前不会递增计数器。

默认情况下，序列的 *is_called* 属性被设置为 FALSE。这意味着在新创建的序列上调用 nextval() 时，不会递增序列，但会返回序列的起始值，并将 *is_called* 设置为 TRUE。之后再次调用 nextval() 时，将返回递增后的值。示例如下：

```
=> SELECT nextval(' myserial ');

nextval
---------
10
(1 row)

=> SELECT nextval(' myserial ');

nextval
---------
11
(1 row)
```

修改序列

ALTER SEQUENCE 命令用于更改现有序列的属性，包括序列的初始值、最小值、最大值和增量值。用户也可以在起始值或指定值处重新启动序列。

使用 *ALTER SEQUENCE...START WITH start_value* 命令可以将序列的初始值 *start_value* 属性的当前值设置为新的起始值，但是对 *last_value* 属性或 nextval() 函数的返回值没有影响。

使用 *ALTER SEQUENCE...RESTART* 命令可以将序列的 *last_value* 属性重置为 *start_value* 属性的当前值，并将 *is_called* 属性重置为“FALSE”，在下一次调用 nextval() 函数时返回 *start_value*。

使用 *ALTER SEQUENCE...RESTART WITH restart_value* 命令可以将序列的 *last_value* 属性设置为一个新值，并将 *is_called* 属性设置为“FALSE”，在下一次调用 nextval() 时会返回 *restart_value*。这等效于调用 *setval(sequence, restart_value, false)*。

如下示例为，在序列值 15 处重新启动名称为 myserial 的序列。

```
ALTER SEQUENCE myserial RESTART WITH 15;
```

内置函数 setval() 可以用来设置序列的计数值，它包含 *setval(sequence, start_val)* 和 *setval(sequence, start_val, is_called)* 两种格式。*setval(sequence, start_val)* 的默认行为会设置序列*is_called*属性值为 TRUE。

如下示例为将序列 myserial 的计数值设置为 2001。

```
=> SELECT setval('myserial', 2001);

setval
--------
2001
(1 row)
```

如果不希望序列在下一次调用 nextval() 时发生递增，可以使用 setval(sequence, start_val, is_called) 函数，将 *is_called* 属性设置为“FALSE”。

如下示例为，使用 setval(sequence, start_val, is_called) 函数 将 *is_called* 属性设置为 FALSE。如果当前序列值为 2001，之后调用 nextval() 函数时，序列值不会递增，仍然为 2001。

```
=> SELECT setval('myserial', 2001, false);

setval
--------
2001
(1 row)

=> SELECT nextval(' myserial ');

nextval
---------
2001
(1 row)
```

默认情况下，序列的数字会一直增长，当一个序列到达了最大值时（SMALLSERIAL 为 +32767，SERIAL 为 +2147483647，BIGSERIAL 为 +9223372036854775807），再次调用 nextval() 会失败。

指定 *CYCLE* 选项后，序列在达到最大值后会循环回到最小值重新开始计数。使用 *ALTER SEQUENCE* 命令可以修改一个序列，使其从 1 开始循环，示例如下：

```
ALTER SEQUENCE myserial CYCLE;
```

使用 *CREATE SEQUENCE* 命令创建序列时也可以指定 CYCLE 行为。当序列达到其最大值或最小值时，如果设置了 CYCLE，则序列会重新开始从最小值或最大值计数，形成一个循环。示例如下：

```
CREATE SEQUENCE myserial CYCLE;
```

引用序列值

在 *CREATE TABLE* 命令中可以指定列直接引用序列并将其作为默认值，该列的数据类型可以不是 SERIAL 或 BIGSERIAL 类型。示例如下：

```
CREATE TABLE ser_table ( id INT4 DEFAULT nextval('myserial'), name TEXT );
```

ALTER TABLE 命令也可以用于更改指定列以将其默认值设置为序列计数器，示例如下：

```
ALTER TABLE ser_table ALTER COLUMN id SET DEFAULT nextval('myserial');
```

INSERT INTO 命令可以通过调用 nextval() 函数来返回并引用序列中的值。

如下示例为，将序列 myserial 的下一个值插入表 ser_table 的第一列：

```
INSERT INTO ser_table VALUES (nextval('myserial'), 'amy');
```

删除序列

DROP SEQUENCE 命令用于删除一个序列。

如下示例为，删除序列 myserial：

```
DROP SEQUENCE myserial;
```

5.8 管理视图

视图通常用于保存常用或复杂的查询。使用视图查询数据与使用普通表相似，但视图本身并不存储数据，而是动态地从其定义中指定的一个或多个表中获取数据。

数据库对象，包括表和视图，其元数据都存储在系统目录表 *pg_class* 中。视图具有一个名为 *_RETURN* 的 ON SELECT 查询重写规则，该规则包含了视图的定义，并存储在 *pg_rewrite* 目录表的 *ev_action* 字段中。

视图的定义不是以字符串形式存储的，而是以查询解析树的形式存储。创建视图时会进行对象名称解析，因此当前的 *search_path* 设置会影响视图的定义。视图引用的是对象的内部 ID 而不是其名称，这意味着可以重命名视图定义中引用的对象或列，而无需删除视图。

在 PieCloudDB 的数据库系统目录中，物化视图的信息与表或视图完全相同。当在查询中引用物化视图时，数据会直接从物化视图返回，但物化视图的数据更新是周期性的。物化视图的一个用途是允许更快地访问从外部数据源带入的数据。此外，可以在物化视图上定义索引，但需要注意的是，外部数据包装器不支持索引。

当用户使用 *SELECT* 访问视图时，该查询会作为一个子查询运行。如果一个子查询与一个单一查询相关联，建议使用 *SELECT* 命令与 *WITH* 子句而不是创建一个使用频率较低的（物化）视图。

创建视图

CREATE VIEW 命令用于创建一个视图。

如下示例为，创建一个视图 order_view 用于查看所有订单信息。

```
CREATE VIEW order_view
   AS SELECT order_id, vender, order_date, amt
     FROM orders;
```

删除视图

DROP VIEW 命令用于删除一个视图。

如下示例为，删除视图 order_view。

```
DROP VIEW order_view ;
```

如果某个表与要删除的视图存在依赖关系，则必须指定 CASCADE 关键字，即 *DROP VIEW...CASCADE*。此外，如果目标表中存在视图依赖关系，也无法对该表进行修改。

以下示例为，在删除存在视图依赖关系的一张表 tbl1 时发生报错。

```
=> CREATE TABLE tbl1 (id INTEGER);

=> CREATE VIEW v1 AS SELECT * FROM tbl1;

=> DROP TABLE tbl1;
ERROR: cannot drop table tbl1 because other objects depend on it
DETAIL: view v1 depends on table tbl1
HINT: Use DROP ... CASCADE to drop the dependent objects too.

=> ALTER TABLE tbl1 DROP id;
ERROR: cannot drop column id of table tbl1 because other objects depend on it
DETAIL: view v1 depends on column id of table tbl1
HINT: Use DROP ... CASCADE to drop the dependent objects too.
```

在更改表的列属性时，需要考虑视图与表之间的依赖关系。例如，如果要将列的数据类型从 INTEGER 更改为 BIGINT，而该列被一个或多个视图所引用，则无法直接对表进行更改。正确的做法是，首先删除这些视图，然后更改列的数据类型，最后使用 *CREATE VIEW* 语句重新创建视图。

如下系统目录表包含用于确定视图依赖的表的信息：

- *pg_class*：记录包括表和视图在内的对象信息。其中，*relkind* 字段描述了对象的类型，视图的 *relkind* 值为“v”。
- *pg_depend*：记录面向数据库特定（非共享）对象的对象依赖信息。
- *pg_rewrite*：记录表和视图的重写规则。
- *pg_attribute*：记录关于对象属性或表列的信息。
- *pg_namespace*：记录关于命名空间的信息。需要注意的是，视图对使用命名空间的对象不存在直接依赖关系，依赖对象实际上是视图的重写规则，这使视图与命名空间的依赖信息具有间接性。

创建物化视图

物化视图与视图类似，物化视图能够保存频繁使用的查询或复杂查询。用户可以使用 *SELECT* 语句访问物化视图所保存的查询结果。

使用 *CREATE MATERIALIZED VIEW* 命令可以基于一个查询定义一个物化视图。示例如下：

```
CREATE MATERIALIZED VIEW china_users
   AS SELECT c.id, c.name, c.zone
     FROM users c, address a
       WHERE a.country = 'China';
```

如果一个查询中包含 ORDER BY 或 SORT 子句，则在执行物化视图的查询时，该子句会被忽略。

更新或停用物化视图

REFRESH MATERIALIZED VIEW 命令用于更新物化视图的数据。示例如下：

```
REFRESH MATERIALIZED VIEW china_users;
```

使用 *WITH NO DATA* 子句可以移除当前数据，不生成新数据，并将物化视图保持在不可扫描状态中。示例如下：

```
REFRESH MATERIALIZED VIEW china_users WITH NO DATA;
```

如果查询尝试访问一个不可扫描的物化视图，会返回错误。

删除物化视图

DROP MATERIALIZED VIEW 命令用于移除一个物化视图定义和数据。

如下示例为，删除一个物化视图 china_users。

```
DROP MATERIALIZED VIEW china_users;
```

由于 *DROP MATERIALIZED VIEW...CASCADE* 命令中指定了 *CASCADE*，所以该命令会移除所有视图相关的依赖项。也就是说，如果一个物化视图依赖于一个即将被删除的物化视图，则另一个物化视图也将被删除。此种场景下，如果不使用 *CASCADE* 选项，则导致删除失败。

6 管理数据

6.1 操纵数据

表是数据库中最基本的操作对象，用于存储用户数据。在创建表并定义其架构之后，就可以开始向表中填充数据了。

本章节主要介绍与数据的增加、删除、修改操作相关的 SQL 语句。

插入数据

当一个表被创建后，它初始是空的，即不包含任何数据。使用 *INSERT* 命令可以向表中添加数据。在执行该命令时，需要指定目标表的名称以及要插入到表中每一列的数据值。用户可以按照任意顺序指定列名和对应的值。如果不指定列名，则必须按照表中列的顺序提供数据值，并且各个值之间用逗号（,）分隔。此外，*INSERT* 命令还支持一次性插入多行数据。

如下示例为，向表 toys 中插入一行数据并指定列名和值。

```
INSERT INTO toys (name, price, toy_no) VALUES ('doll', 9.99, 1);
```

如下示例为，向表 toys 中插入一行数据，但只指定要插入的值。

```
INSERT INTO toys VALUES (1,'doll', 9.99);
```

通常，插入表中的数据值都是常量，但也可以使用标量表达式。如下示例使用了标量表达式 *SELECT * FROM tmp_toys WHERE price< 1000* 向表 toys 中插入数据。

```
INSERT INTO toys SELECT * FROM tmp_toys WHERE price < 1000;
```

PieCloudDB 也支持在单个命令中同时插入多行。示例如下：

```
INSERT INTO toys (toy_no,name, price) VALUES (1,'doll', 9.99), (2,'car', 19.99), (3,'ball', 29.99);
```

要向分区表中插入数据，需要指定根分区表，即使用 *CREATE TABLE* 命令创建的表，还可以在 *INSERT* 命令中指定分区表的叶子子分区表。如果数据不符合指定的叶子子表，会返回错误。需要注意的是，PieCloudDB 不支持在 *INSERT* 命令中指定非叶子子表。

对于大量数据的插入，建议使用外部表或 *COPY* 命令，以避免在存储系统中产生大量小数据块而影响查询性能。

更新数据

UPDATE 命令用于更新表中的数据，可以更新所有行、所有行的子集或表中的单个行。也可以单独更新每个列而不会影响其他列。更新数据时需要指定要更新的表和列的名称、列的新值，以及要指定更新的行的一个或者多个条件。

如下示例为，将所有价格为 9.99 的产品的价格更新为 10。

```
UPDATE toys SET price = 10 WHERE price = 9.99;
```

需要注意的是，*UPDATE* 不能更新分区表的键列，如果执行该操作就会发生报错。如下例中，表 sales 的分区键是“date”，在更新该列时发生报错。

```
=> UPDATE sales SET date = '2023-01-02' WHERE amt = 10000;
ERROR: no partition of relation "sales" found for row (seg0 1.2.1.6:6000 pid=58631)
DETAIL: Partition key of the failing row contains (date) = (2023-01-02).
```

删除数据

DELETE 命令用于从表中删除数据。在该命令中指定一个 WHERE 子句可以删除与某些条件匹配的行；如果不指定 WHERE 子句，则会删除表中的所有行。

如下示例为，从表 toys 中删除价格为 10 的所有行。

```
DELETE FROM toys WHERE price = 10;
```

如下示例为，删除表 toys 中的所有行。

```
DELETE FROM toys;
```

清空数据

TRUNCATE 命令用于快速删除表中的所有行。示例如下：

```
TRUNCATE mytable;
```

注意：

执行 TRUNCATE 会一次性清空表及其子表中的所有行，默认情况下子表的数据会被包括在对父表的扫描中。

返回修改后数据

INSERT、*UPDATE* 和 *DELETE* 命令都支持可选的 RETURNING 子句，该子句用于返回操作影响的行。RETURNING 子句的一个常见用法是简写形式 *RETURNING **，它按照表中列的顺序返回所有列的值，这与 *SELECT* 命令的输出结果相同。

对于 INSERT 操作，RETURNING 子句尤其有用。例如，对于使用序列列来提供唯一标识符的场景，RETURNING 可以返回分配给新行的 ID。

```
=> CREATE TABLE visitors (firstname TEXT, lastname TEXT, id SERIAL);
=> INSERT INTO visitors (firstname, lastname) VALUES ('Joe', 'Cool') RETURNING id;

 id
----
 1
(1 row)
INSERT 0 1
```

此外，RETURNING 子句也常用于 *INSERT...SELECT* 命令。

对于 *UPDATE* 命令，RETURNING 子句可以返回新修改的内容。示例如下：

```
=> UPDATE toys SET price = price * 1.10 WHERE price <= 99.99 RETURNING name, price AS
 new_price;

name | new_price
------+-----------
car | 21.99
ball | 32.99
doll | 10.99
(3 rows)
```

对于 *DELETE* 命令，RETURNING 子句可以返回被删除的内容。示例如下：

```
=> DELETE FROM toys WHERE price = 32.99 RETURNING *;

toy_no | name | price
--------+------+-------
3 | ball | 32.99
(1 row)
```

6.2 并发控制

PieCloudDB 采用多版本并发控制（Multi-Version Concurrency Control，MVCC）。MVCC 为每个数据库会话提供了事务隔离，确保每个查询事务都能看到一个数据的一致性视图，即数据快照，而不受其他并发事务的影响。这种机制确保了在并发环境下，查询（读取）操作获取的锁不会与写入操作获取的锁发生冲突，从而保证了数据库在多用户环境下的高性能。

锁模式

PieCloudDB 提供多种锁模式来控制对表中数据的并发访问。大多数 SQL 命令会自动获取适当的锁，以确保在命令执行期间不会删除或修改被引用的表。此外，用户也可以使用 *LOCK* 命令显式获取锁。

PieCloudDB 支持的表级锁模式如下表所示。

锁模式	相应的 SQL 命令
访问共享锁 ACCESS SHARE	SELECT
行排它锁 ROW EXCLUSIVE	INSERT
共享更新排它锁 SHARE UPDATE EXCLUSIVE	• VACUUM（不支持 FULL） • ANALYZE
共享行排它锁 SHARE ROW EXCLUSIVE	CREATE TRIGGER
排它锁 EXCLUSIVE	• SELECT FOR UPDATE • SELECT FOR SHARE • UPDATE • DELETE
访问排它锁ACCESS EXCLUSIVE	• ALTER TABLE • DROP TABLE • TRUNCATE • CLUSTER

需要注意的是，一旦事务获取了锁，通常在事务结束时或 Savepoint 回滚的时候被释放。

不同锁之间的冲突关系如下表所示。

请求的锁模式	ACCESS SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE						X
ROW EXCLUSIVE				X	X	X
SHARE UPDATE EXCLUSIVE			X	X	X	X
SHARE ROW EXCLUSIVE		X	X	X	X	X
EXCLUSIVE		X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X

死锁检测

PieCloudDB 的死锁检测器后台进程会收集所有锁的信息，并使用定向算法检测是否存在死锁。这一算法放宽了对并发更新和删除操作的限制。

PieCloudDB 通过将 *deadlock_timeout* 参数设置为非零值来启用死锁检测功能。该参数的默认值为 1 秒。如果死锁检测器确定存在死锁，它会通过取消与最新事务相关的一个或多个后端进程来打破死锁。

在同一并发事务中（例如，第一个事务执行更新操作，接下来的事务执行更新或删除，并且查询计划包含一个动作操作符），如果死锁检测器确定此类事务存在死锁，则只有其中一个事务会成功执行，并提示不允许对同一行进行并发更新。

PieCloudDB 使用 *deadlock_timeout* 服务器配置参数来设置死锁检测的时间间隔。如果启用了 *lock_timeout* 服务器配置参数，并将其设置为小于 *deadlock_timeout*，则在会话进行死锁检查之前，查询将被终止。

pg_stat_activity 视图用于查看阻塞或等待的事务，结合 *pg_locks* 视图，可以监控锁竞争和死锁情况，帮助排查锁冲突和死锁问题，从而优化并发操作。如下命令为，结合 *pg_stat_activity* 与 *pg_lock* 视图一起查看锁等待的详细信息。

```
SELECT * FROM pg_locks pl
   JOIN pg_stat_activity ps ON pl.pid = ps.pid
      WHERE NOT granted;
```

6.3 事务管理

事务隔离

事务允许将多个 SQL 语句捆绑在同一个操作中。事务中的所有 SQL 要么全部执行，要么全部不执行。 事务允许将多个 SQL 语句捆绑在同一个操作中，确保这些操作要么全部成功执行，要么全部不执行，从而维护数据的完整性。

PieCloudDB 事务支持的 SQL 命令如下：

- BEGIN 或 START TRANSACTION：开始一个事务块。
- END 或 COMMIT：提交事务的结果。
- ROLLBACK：放弃一个事务，不做任何更改。
- SAVEPOINT：标记了事务中的某一个位置并启用了部分回滚。
- ROLLBACK TO SAVEPOINT：将事务回滚到一个 Savepoint。
- RELEASE SAVEPOINT：释放事务中的一个 Savepoint。

PieCloudDB 支持如下事务隔离级别：

- READ UNCOMMITTED：对此隔离级别只提供语法支持，语义上等同于 READ COMMITTED。
- READ COMMITTED：只可以读取已经提交的变更，可以避免脏读但可能出现不可重复读问题。这是默认的隔离级别。
- REPEATABLE READ：每个事务都有一个独立的快照，使得在同一事务中对同一条记录多次读取的结果一致，可以避免脏读和不可重复读问题，但可能会出现幻读。
- SERIALIZABLE：事务串行执行，完全排除所有可能的并发冲突，它提供最高级别的隔离但是可能会导致更多的锁冲突，也可能性能较差。

使用 *SET TRANSACTION* 命令可以设置一个事务的事务隔离级别，也可以使用 *BEGIN* 或者 *START TRANSACTION* 指定事务隔离级别。

如果执行 *SET TRANSACTION* 之前没有执行过 *START TRANSACTION* 或者 *BEGIN*，则会收到警告并且该命令不会生效。示例如下：

```
=> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
WARNING: SET TRANSACTION can only be used in transaction blocks
SET

=> BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN

=> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
```

READ UNCOMMITTED 和 READ COMMITTED

READ COMMITTED 提供了一种快速且简单的事务隔离级别。在 PieCloudDB 中，不允许任何命令看到另一个并发事务中的未提交更新，因此 READ UNCOMMITTED 的行为实际上与 READ COMMITTED 相同。

在一个事务中，SELECT、UPDATE 和 DELETE 操作基于查询开始时的数据库快照进行。SELECT 查询可以看到查询开始之前已经提交的数据，而 UPDATE 和 DELETE 命令只能操作在命令开始之前已经提交的数据。

对于 SELECT 查询，其特性如下：

- 可以看到查询开始之前其他事务提交的数据。
- 可以看到同一事务中其他操作执行的更新。
- 不会看到同一事务中尚未提交的数据。
- 如果其他并发事务在查询事务的最初读操作之前提交，那么该查询可能看到这些并发事务的更改。
- 在同一事务中连续执行 SELECT 查询时，如果其他并发事务在此期间提交了更改，那么后续的查询可能会看到不同的数据。

READ COMMITTED 隔离级别允许并发事务在 UPDATE 或 DELETE 命令查找行之前修改或锁定这些行。对于执行复杂查询和需要数据库一致视图的应用程序，READ COMMITTED 可能不足以满足需求。

PieCloudDB 的默认事务隔离级别为 READ COMMITTED。如果要更改事务的隔离级别，可以在事务开始时声明隔离级别，或者在事务开始后使用 *SET TRANSACTION* 命令进行更改。

REPEATABLE READ 和 SERIALIZABLE

SERIALIZABLE 事务隔离级别确保并发运行的事务产生的结果与它们串行运行的结果相同，它提供了最严格的事务隔离级别。在 PieCloudDB 中，如果指定了 SERIALIZABLE，系统实际上会将其降级到 REPEATABLE READ。

REPEATABLE READ 事务隔离级别可以防止脏读、不可重复读和幻读，但 PieCloudDB 不会检测并发事务执行期间可能发生的所有序列化交互。例如，当两个事务并发更新同一数据集（例如同一行）时，可能会产生冲突，这时可以通过显式表锁或要求代表冲突事务的虚拟行来防止这些交互。

对于 REPEATABLE READ 事务，*UPDATE*、*DELETE*、*SELECT FOR UPDATE* 和 *SELECT FOR SHARE* 命令只能查找到在命令开始前已经提交的行。对于 SELECT 查询，其特性如下：

- 只能看到事务开始时的数据快照，而不是在事务内的当前查询开始时的数据。
- 只能看到查询开始前已经提交的数据。
- 可以看到同一事务内的其他更新。
- 不能看到同一事务外的未提交数据。
- 不能看到其他并发事务所做的更改。
- 同一事务内连续执行的 SELECT 命令总是查询到相同的数据。

如果一个并发事务已经更新、删除或锁定了目标行，则 REPEATABLE READ 事务会等待该并发事务提交或回滚更改。如果并发事务提交了更改，则 REPEATABLE READ 事务可能会回滚；如果并发事务回滚了更改，则 REPEATABLE READ 事务可以提交其更改。

事务 ID 管理

在 PieCloudDB 中，事务的可见性是通过比较事务 ID（XID）来确定的。通常，只有涉及 DDL 或 DML 操作的事务会被分配 XID，因为这些事务通常需要 XID 来确保事务的隔离性。

PieCloudDB 会监视事务 ID，并在大量事务 ID 不再可用且在事务 ID 循环发生之前发出警告，告警信息一般为“WARNING: database "database_name" must be vacuumed within *number_of_transactions *transactions”。这时，用户需要执行 VACUUM 操作来清理不再需要的事务 ID，以避免潜在的数据丢失问题。

如果未执行 VACUUM 操作，PieCloudDB 会在到达事务 ID 限制并触发循环之前停止创建新事务，并发出错误信息以防止数据丢失，错误信息一般为“FATAL: database is not accepting commands to avoid wraparound data loss in database 'database_name'”。为了避免这种情况，可以通过设置 *deadlock_timeout* 参数为非零值来启用死锁检测功能。死锁检测器会通过终止与最新事务所关联的一个或多个后端进程来解决死锁问题。

此外，*xid_warn_limit* 参数用于控制显示警告的时间，而 *xid_stop_limit* 参数用于控制停止创建新事务的时间。这些参数有助于管理事务 ID，以确保数据库的稳定性和性能。如下示例为，查看上述参数的当前值。

```
=> SELECT current_setting('xid_warn_limit');

current_setting
-----------------
500000000
(1 row)

=> SELECT current_setting('xid_stop_limit');

current_setting
-----------------
100000000
(1 row)
```

如果因为不经常执行 *VACUUM* 命令而达到 *xid_stop_limit* 参数的阈值，导致 PieCloudDB 无法创建新事务。此时，管理员可以按照以下步骤进行恢复：

1. 关闭 PieCloudDB。
2. 临时将 *xid_stop_limit* 参数值降低至 10,000,000。
3. 重新启动 PieCloudDB。
4. 对所有受影响的数据库执行 *VACUUM FREEZE*。
5. 将 *xid_stop_limit* 参数重置为其原始值。
6. 再次重新启动 PieCloudDB。

6.4 回收空间

在 PieCloudDB 中使用的 MVCC 事务并发模型中，已删除或更新的数据虽然对新事务不可见，但仍然占据物理磁盘空间。如果数据库中存在大量的更新和删除操作，就会积累许多过期数据。因此，建议定期使用 *VACUUM* 命令来回收这些数据所占用的空间。

此外，*VACUUM* 命令还会收集表级别的统计信息，例如行数和页数。因此，即使没有更新或删除的行，仍然可以执行 VACUUM 操作以回收空间并维护数据库的性能。

定期运行 *VACUUM* 命令可以删除过期数据。示例如下：

```
VACUUM mytable;
```

在 PieCloudDB 中，*VACUUM*、*VACUUM ANALYZE* 和 *VACUUM COMPACT* 命令是维护数据的重要工具，尤其适用于频繁进行数据更新和删除操作的场景。

- *VACUUM*：此命令用于清除表中的已删除行，重新排序剩余的行，并更新索引。它有助于维护表的物理顺序和索引的准确性，但不会回收完全释放的磁盘空间。
- *VACUUM ANALYZE*：此命令首先执行 *VACUUM* 操作以清理已删除的行，然后更新表和索引的统计信息。更新统计信息后，查询优化器可以获取最新的数据分布情况，从而生成更有效的查询执行计划。
- *VACUUM COMPACT*：此命令将存储系统中的小文件块合并为默认大小的文件块。合并后的小文件块可以通过 *VACUUM* 命令进行清理，从而提高存储效率。

7 查询数据

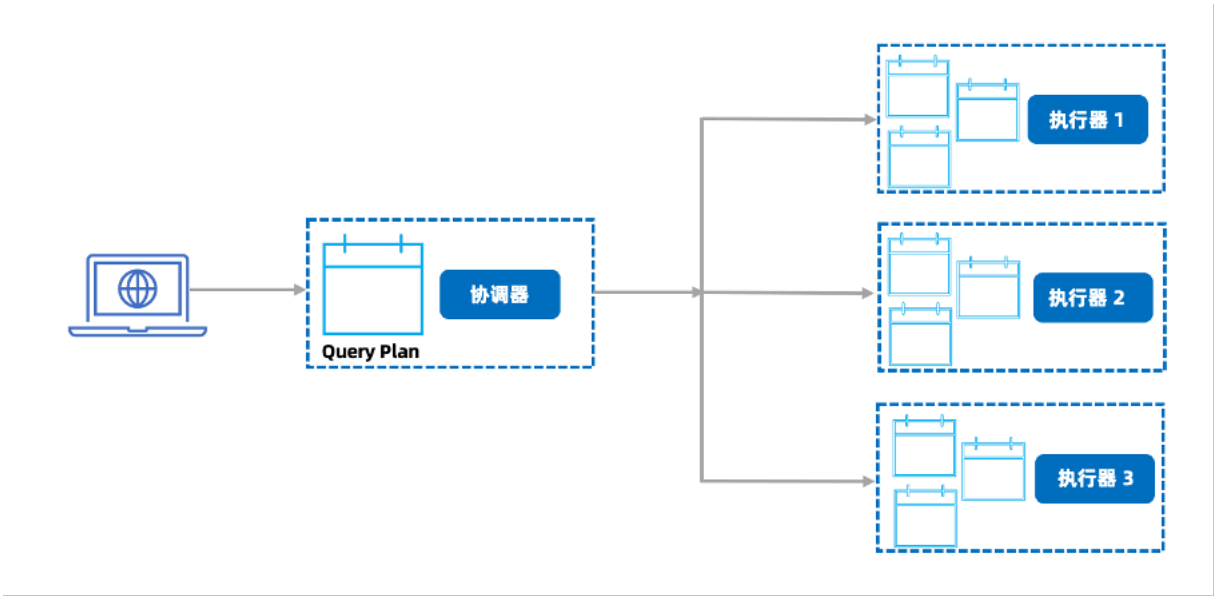
7.1 理解查询处理

用户可以使用 psql 等客户端应用程序连接到 PieCloudDB 数据库实例所在的 Master 主机，提交并执行 SQL 语句以完成查询任务。

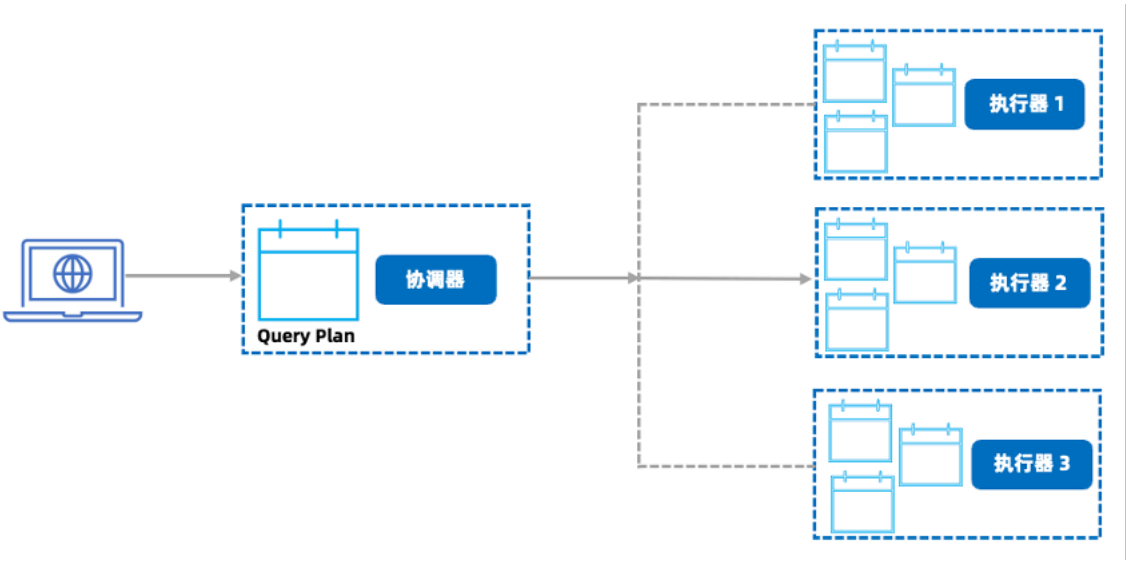
查询执行与分发

PieCloudDB 的协调器负责接收、解析并优化查询，而执行器则用于执行协调器分发的任务。每个集群包含一个或多个执行器，这些执行器能够并行处理同一个 SQL 查询任务。

查询计划可以是并行的，也可以是串行的。当协调器从客户端接收到查询请求后，它会将并行查询计划分发到所有可用的执行器。大多数的数据库操作，如表扫描、连接、聚集（Aggregation）和排序（Sorting），都会在所有执行器上以并行方式执行，每个执行器上的操作都是独立进行的，如下图所示。



对于定向查询计划，协调器会将这类查询计划分发到单个执行器进行处理。这类查询通常包括单行的 INSERT、UPDATE、DELETE 操作，或者是基于过滤表分布键列的 SELECT 查询。如下图所示。



执行计划

查询计划是一系列将要执行的操作的集合，这些操作包括但不限于表扫描、连接、聚集、排序等，它们共同协作以产生查询结果。

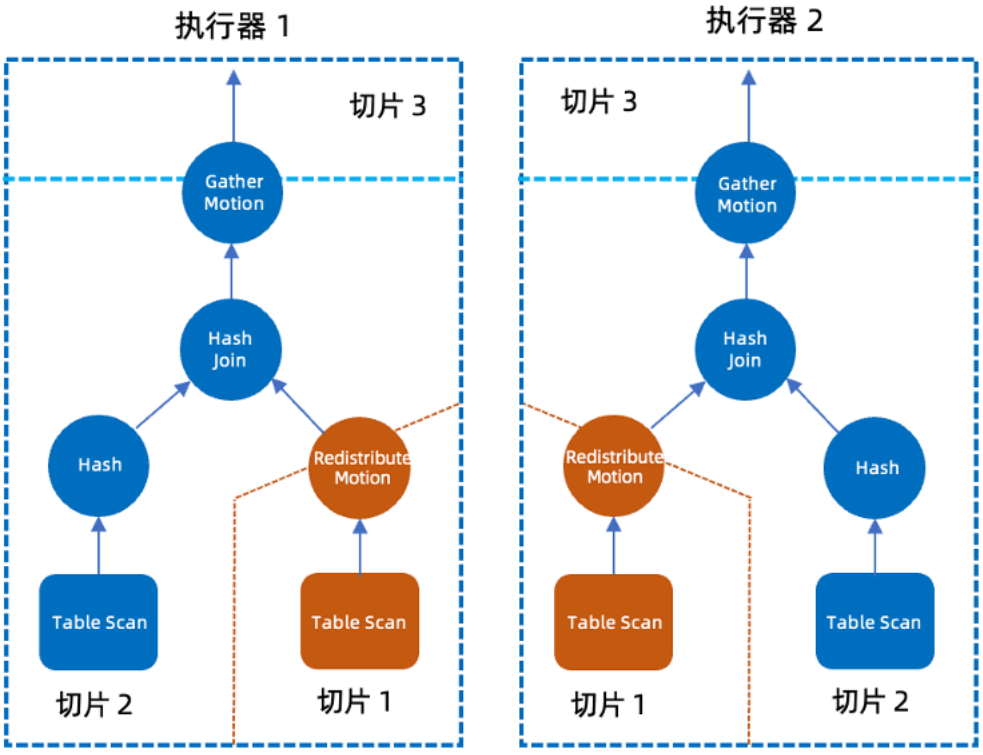
除了常见的表扫描和连接等数据库操作之外，PieCloudDB 还支持移动操作（Motion Operation）。在查询处理过程中，移动操作可能涉及到在执行器之间通过 Interconnect 传输元组（Tuples）。然而，并非所有查询都需要进行数据移动；例如，定向查询计划就不需要通过 Interconnect 传输数据。

为了在查询执行期间实现最大的并行度，PieCloudDB 将查询计划分解为多个切片（Slices）。每个切片都是执行器能够独立工作的计划片段。如果查询计划中包含移动操作，那么该计划就会被划分为多个切片，并在移动操作的两端各分布一个切片。

如下示例展示了一个涉及连接两个表 sales 和 customer 的简单查询。在处理此类查询时，每个执行器会接收到一份查询计划的副本，并根据该计划并行执行任务。

```
SELECT cust_name, amount FROM sales JOIN customer USING (cust_id)
   WHERE order_date = '09-30-2022';
```

在本示例的查询计划中，包含一个必要的重分布移动操作（Redistribute Motion），该操作采用默认的是随机分布（Random Distribution）方法，在执行器之间传输元组（Tuples）以完成表之间的连接操作。为了执行上述的连接，表 sales 中的元组必须根据列 cust_id 的值进行重新分布。查询计划在重分布移动操作的两侧被分割成两个切片，即切片 1 和切片 2，如下面的图所示：



在本示例的查询计划中，除了重分布移动操作外，还涉及到另一种类型的移动操作——收集移动（Gather Motion）。收集移动负责记录执行器何时将处理结果发回给协调器，随后协调器将这些结果呈现给客户端。由于查询计划中存在移动操作，它会被进一步被切片处理。因此，此计划在最顶层隐式地包含了一个切片 3。

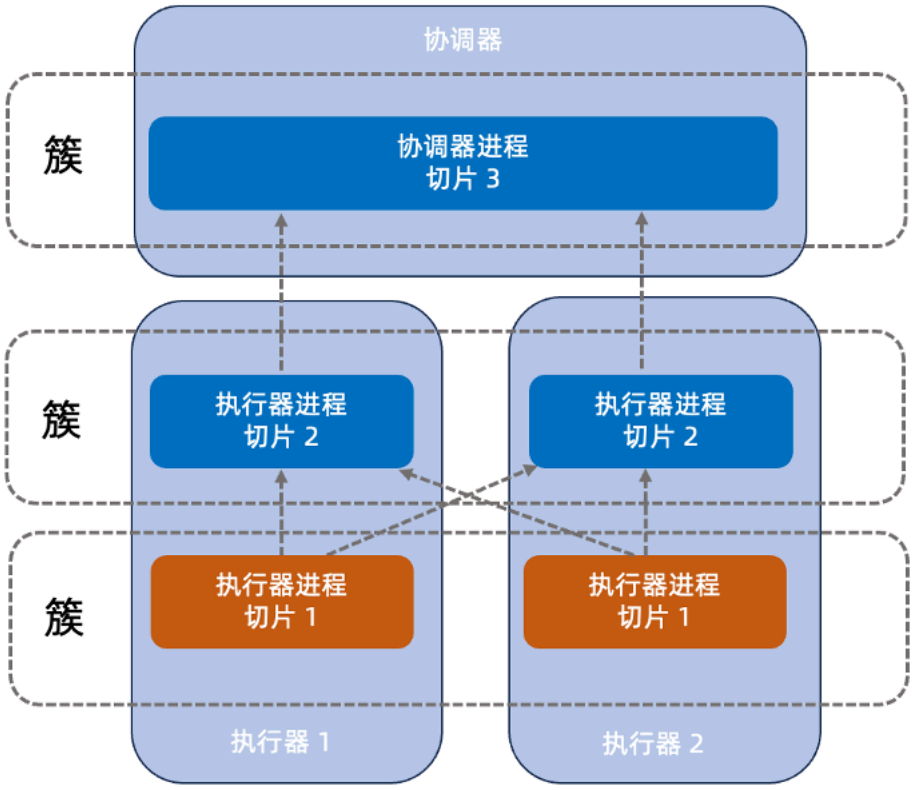
并非所有查询计划都包含收集移动。例如，`CREATE TABLE...AS SELECT...` 语句就不会涉及收集移动，因为在这种情况下，元组直接被发送到新创建的表中，而不是返回给协调器。

并行查询

PieCloudDB 创建多个进程来处理并行查询任务。在协调器上，有一组工作进程负责创建查询计划，并将这些计划分发给执行器；同时，它们也负责收集各执行器的结果，并将最终结果发送给客户端。在执行器上，另一组工作进程负责执行分发下来的查询计划，并在需要时与其他执行器上的工作进程交换中间结果。

每个查询计划的切片都至少分配给一个工作进程，这些工作进程独立地执行它们所分配到的查询计划部分。在查询执行期间，每个执行器会并行运行多个进程。如果为同一个查询计划切片工作但位于不同执行器上的进程，它们会组成一个“簇”。随着查询的执行，元组会在进程簇之间流动，直至达到查询计划中的下一个簇。

这种执行器之间的进程间通信是通过 Interconnect 组件实现的，确保了数据在执行器之间的高效传输。



7.2 查询定义

SQL（Structured Query Language）是一种用于访问数据库的标准编程语言，它允许用户通过 SQL 命令来构造数据库引擎能够理解的查询和命令。PieCloudDB 的 SQL 命令是基于 PostgreSQL 的 SQL 标准实现的。

SQL 词法和 SQL 值表达式是构成 SQL 命令的关键部分。SQL 词法由正确语法顺序排列的合法符号组成，并且通常以分号（;）结束。SQL 值表达式由一个或多个值、符号、运算符、SQL 函数及数据构成。SQL 值表达式通过比较数据或执行计算（包括逻辑、数学和集合操作）来返回一个值作为结果。一个 SQL 值表达式的结果有时被称为标量，因此也被称为标量表达式，或者简称为表达式。

SQL 值表达式支持如下形式：

- 常量或字面量，例如数字、字符串、布尔值等。
- 列引用，即数据库表中的列名。
- 在函数定义或 Prepared 语句中引用的位置参数。
- 数组下标表达式，即用于访问数组元素的表达式。
- 字段选择表达式，即用于从复合类型（如结构体）中选择特定字段的表达式。
- 运算符调用，即使用运算符（例如加号 +、减号 - 等）进行计算的表达式。
- 函数调用，即调用 SQL 函数的表达式。
- 聚集表达式，即使用聚集函数（如SUM、COUNT等）的表达式。
- 窗口函数调用，即调用窗口函数（如ROW_NUMBER、RANK等）的表达式。

- 类型转换，即将值转换为另一种数据类型的表达式。
- 排序规则表达式，定义排序规则的表达式，例如 COLLATE 关键字。
- 标量子查询，即返回单个值的子查询。
- 数组构造器，即用于创建数组的表达式。
- 行构造器，即用于创建行的表达式。
- 圆括号中的值表达式，即用于子表达式分组以及优先级重载的表达式。

列引用

SQL 值表达式可以通过以下语法引用某个列：

```
correlation.column_name
```

其中，*correlation* 可以是：

- 表名称：可以带有 Schema 名称限定。
- 表别名：在 FROM 子句中定义的别名。
- 特殊关键词：NEW 或 OLD。NEW 和 OLD 用于引用触发器操作中新增或修改前的行，它们只能出现在触发器的重写规则中。

如果列名在查询涉及的所有表中都是唯一的，可以省略 correlation 部分，直接使用 column_name。

位置参数

SQL 位置参数是 SQL 语句或函数的参数，它们通过在参数序列中的位置来引用。位置参数的值通常在 SQL 语句执行时由外部设置，或者在 SQL 函数被调用时提供。某些客户端允许在 SQL 命令之外单独指定数据值，在这种情况下，参数引用的是这些外部数据值。

SQL 值表达式引用一个位置参数的语法如下：

```
$number
```

例如，*$1* 表示引用第一个参数，*$2* 表示引用第二个参数，以此类推。如下示例中，*$1* 表示引用 dept() 函数被调用时的第一个参数的值。这种引用方式通常用于函数定义内部，以访问传递给函数的参数。

```
CREATE FUNCTION dept(text) RETURNS dept AS $$
   SELECT * FROM dept
      WHERE dname = $1 $$ LANGUAGE SQL;
```

数组下标

如果一个 SQL 值表达式引用的是一个数组类型的值，可以使用以下语法来抽取该数组值中的指定元素：

```
expression[subscript]
```

如果一个 SQL 值表达式引用的是一个数组类型的值，可以使用以下语法来抽取该数组中的一个切片，即多个相邻的元素。每个下标都是一个表达式，且对应一个整数值：

```
expression[lower_subscript:upper_subscript]
```

在 SQL 中，数组表达式通常需要放在圆括号内。但是，当指定下标的表达式是一个列引用或者位置参数时，可以省略这些圆括号。此外，如果原始数组是多维数组，可以通过串联多个下标来访问特定的元素。示例如下：

```
mytable.arraycolumn[4]mytable.two_d_column[27][44]$1[20:45](arrayfunction(a,b))[45]
```

字段选择

如果一个表达式返回一个复合类型（行类型）的值，可以使用如下语法来抽取该行的指定字段：

```
expression.fieldname
```

如果一个表达式返回一个复合类型（行类型）的值，通常需要将该表达式放在圆括号中，然后使用点号（.）来访问特定的字段。然而，如果表达式是一个表引用或者位置参数，可以省略圆括号。示例如下：

```
mytable.column1

$1.somecolumn
```

如下示例中，被限定的列引用是字段选择语法的一种特殊情况。

```
(mytable.compositecol).somefield
```

运算符调用

运算符调用是指使用运算符来对一个或多个表达式求值，有下列可能的语法：

```
表达式 运算符 表达式 (二元运算符)
运算符 表达式 (一元运算符)
表达式 运算符 (一元运算符)
```

其中，运算符是一个运算符号（+ - * / < > = ~ ! @ # % ^ & | ` ?），或者关键词 AND、OR 或 NOT 之一，还可以是限定运算符名称，即显式调用数据库中已定义的运算符，示例如下：

```
OPERATOR(schema.operator_name)
```

二元运算符需要两个操作数，例如“a > b”；一元运算符只对一个操作数进行操作，例如“-a”。可用的一元或二元运算符可以由系统或者用户定义。

函数调用

函数调用的语法是由一个函数名（可能由一个 Schema 名称限定）后面跟着圆括号中的参数，如下所示：

```
function_name ( [expression [, expression ...] ] )
```

如下示例为，调用函数 sqrt() 计算 2 的平方根。

```
sqrt(2)
```

聚集表达式

聚集表达式通常用于查询中，以在所选择的行上应用聚集函数。聚集函数对一组值执行单一计算并返回一个值，例如计算一组值的总和或平均值。聚集表达式支持如下多种语法：

```
-- 在所有输入行上操作，每一个输入行调用一次聚合，预期的结果值非空。默认是 ALL。
aggregate_name (expression [, ...] [order_by_clause] ) [FILTER ( WHERE filter_clause )]

aggregate_name ( [expression [ , ...] ] ) WITHIN GROUP ( order_by_clause ) [FILTER ( WHERE
 filter_clause )]

-- 和第一种语法形式相同，默认都是 ALL。
aggregate_name (ALL expression [, ...] [order_by_clause] ) [FILTER ( WHERE filter_clause )]

-- 在输入行的所有可区分的非空值上调用一次聚集
aggregate_name (DISTINCT expression [, ...] [order_by_clause] ) [FILTER ( WHERE filter_clause )]

-- 在所有值为空以及非空的行上调用一次聚集。通常这种形式最适用于 count(*) 聚集函数。
aggregate_name ( * ) [FILTER ( WHERE filter_clause )]
```

其中，*aggregate_name* 是一个之前定义的聚集（可能由 Schema 限定），而 *expression* 是任何不包含聚集表达式的值表达式或一个窗口函数调用。*order_by_clause* 和 *filter_clause* 子句是可选的。

例如，*count(*)* 会得到输入行的总数，*count(column1)* 得到输入行中 *column1* 为非空的数量，而 *count(distinct column1)* 会得到列 *column1* 中可区分非空值的数量。

一般地，传递给聚集函数的输入行是未排序的。对于某些聚合函数（例如 array_agg() 和 string_agg()）需要依据输入行的排序产生结果，这时可以选择使用 *order_by_clause* 指定顺序。示例如下：

```
SELECT array_agg(a ORDER BY b DESC) FROM tablename;
```

如果指定了 FILTER，那么只有对 *filter_clause* 计算为“TURE”的输入行会被传递该聚集函数，其他行会被丢弃。示例如下：

```
=> SELECT count(*) AS unfiltered, count(*) FILTER (WHERE i < 50) AS filtered
     FROM generate_series(1,100) AS s(i);

unfiltered | filtered
-----------+----------
100 | 49
(1 row)
```

PieCloudDB 支持 MEDIAN() 聚集函数，用于计算一组值的中位数，MEDIAN() 函数是通过调用的 PERCENTILE_CONT() 函数来实现的。当前 PieCloudDB 只支持如下两个表达式与关键词 WITHIN GROUP 一起使用：

```
PERCENTILE_CONT(_percentage_) WITHIN GROUP (ORDER BY _expression_)
PERCENTILE_DISC(_percentage_) WITHIN GROUP (ORDER BY _expression_)
```

聚集表达式通常只能出现在 SELECT 语句的结果列表或者 HAVING 子句中。它们不允许在其他子句（例如 WHERE）中使用，这是由于逻辑层面上的聚集表达式的计算应该在聚集结果形成之后进行。

当聚集表达式出现在子查询中时，它通常对子查询的行进行聚集计算。如果聚集表达式的参数只包含来自外层查询的变量，那么这个聚集表达式属于最近的外层查询，并在该外层查询的行上进行计算。这类查询表达式总体上是其所在子查询的外部引用，并且该聚集表达式在该子查询的任何一次计算中都被视为一个常量。

需要注意的是，对于某些数据库系统，例如 PieCloudDB，不支持带有多个输入表达式的 DISTINCT 语法。

窗口函数调用

窗口函数通过使用标准 SQL 命令，提供了构建复杂的在线分析处理（OLAP）查询的功能。

窗口分区是指将查询结果集中的行划分为多个组，这些组内的行将一起用于计算窗口函数。与聚集函数不同，窗口函数为每一行返回一个结果值，但这个值是基于该行所属的特定窗口分区中的行计算得出的。如果未指定分区，则窗口函数将在完整的结果集上对每一行进行计算。

窗口函数调用的语法如下：

```
window_function ([expression [, expression ...]]) [FILTER ( WHERE filter_clause )] OVER
 window_name

window_function ([expression [, expression ...]]) [FILTER ( WHERE filter_clause )] OVER
 ( window_definition )

window_function ( * ) [FILTER ( WHERE filter_clause )] OVER window_name

window_function ( * ) [FILTER ( WHERE filter_clause )] OVER ( window_definition )
```

其中，*window_definition* 的语法如下：

```
[existing_window_name]
[PARTITION BY expression [, ...] ]
[ORDER BY expression [ ASC | DESC | USING operator] [NULLS { FIRST | LAST }] [, ...] ]
[frame_clause]
```

对于 *window_definition* 中的 *frame_clause* 语法是可选的，支持如下两种语法格式：

```
{ RANGE | ROWS | GROUPS } frame_start [frame_exclusion]
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end [frame_exclusion]
```

对于 *frame_clause* 中的 *frame_start* 和 *frame_end* 语法的可选格式如下：

```
UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING
```

对于 *frame_clause* 中的 *frame_exclusion* 语法的可选格式如下：

```
EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

其中，*expression* 是自身不包含窗口表达式的值表达式，*window_name* 是对在 WINDOW 子句中定义的一个已命名窗口声明的引用。

PARTITION BY 子句将查询的行分组为多个分区。如果省略此子句，整个结果集被视为一个单一的分区。ORDER BY 子句指定了在窗口函数处理时，一个分区内行的排序方式。对于 OLAP 聚集操作，如果要使用窗口帧（ROWS 或 RANGE 模式），则必须包含 ORDER BY 子句。

frame_clause 为聚集（非排名）窗口函数定义窗口帧。一个窗口帧在一个窗口分区内定义一个行集合。当一个窗口帧被定义时，窗口函数会在这个帧的可移动内容上进行计算，而不是在整个窗口分区的固定内容上计算。窗口帧可以是基于行的（ROWS）或者基于值的（RANGE）。如果 *frame_end* 被忽略，则默认运行到 CURRENT ROW。

需要注意的是，只有在 SELECT 查询和 ORDER BY 子句中才允许窗口函数调用。

如下示例为，通过调用窗口函数比较员工工资和部门平均工资，并输出部门名称、员工编号、员工工资和部门平均工资。

```
=> SELECT deptname, empno, salary, avg(salary) OVER(PARTITION BY deptname) FROM
 emp_salary;

deptname | empno | salary | avg
-----------+-------+--------+----------------------
develop | 19 | 45000 | 50200.00
develop | 110 | 52000 | 50200.00
develop | 111 | 52000 | 50200.00
develop | 17 | 42000 | 50200.00
develop | 18 | 60000 | 50200.00
personnel | 15 | 35000 | 37000.00
personnel | 12 | 39000 | 37000.00
sales | 11 | 50000 | 48660.67
sales | 13 | 48000 | 48660.67
sales | 14 | 48000 | 48660.67
(10 rows)
```

类型转换

类型转换是指从一种数据类型转换到另一种数据类型。PieCloudDB 支持如下两种等效的类型转换语法：

```
CAST ( expression AS type )

expression::type
```

类型转换允许 PieCloudDB 跨不同类型的数据进行查询。显式的数据类型转换可以通过 CAST 函数或使用双冒号“::”操作符来完成。同时，数据库系统也会自动执行隐式的数据类型转换。

CAST 函数的相关信息和类型转换规则存储在系统目录表 *pg_cast* 中。通过在 psql 命令行工具中使用 *\dC* 命令，可以显示类型转换的相关信息。

排序规则表达式

COLLATE 子句可以重载一个表达式的排序规则，语法如下：

```
expr COLLATE collation
```

collation 可能是一个由 Schema 限定的标识符。如果没有显式指定排序规则，数据库系统会从排序规则表达式所涉及的列中获取排序规则；如果该表达式没有涉及列，则会默认采用数据库的默认排序规则。

COLLATE 子句常用于重载 ORDER BY 子句中的排序顺序，示例如下：

```
SELECT a, b, c FROM tablename WHERE(b > 'foo') ORDER BY b COLLATE "C";
```

COLLATE 还用于覆盖函数或运算符调用的排序规则，示例如下：

```
SELECT * FROM tablename WHERE b > 'foo' COLLATE "C";
```

标量子查询

标量子查询是包含在一个圆括号内的 SELECT 查询，返回值为一行一列。需要注意的是，不要将返回多行或者多列的 SELECT 查询用作一个标量子查询。示例如下：

```
SELECT * FROM toys WHERE price > (SELECT AVG(price) FROM toys);
```

数组构造器

数组构造器是一种表达式，用于基于其成员元素的值来构造一个数组。一个简单的数组构造器由关键词 ARRAY、一个左方括号“[”、一个或多个由逗号分隔的表达式（用于表示数组元素的值）以及一个右方括号“]”组成。示例如下：

```
=> SELECT ARRAY[1+1,2,6-3];
array
---------
{2,2,3}
(1 row)
```

数组元素类型通常是从数组中包含的成员表达式的公共类型推断出来的。然而，也可以通过显式指定来定义数组元素的类型。这种显式指定可以在创建数组时进行，以确保数组元素遵循特定的数据类型。如下为显式指定数组元素为整型的示例。

```
=> SELECT ARRAY[1,2,3.4]::integer[];
array
---------
{1,2,3}
(1 row)
```

在 SQL 中，多维数组必须是矩形的，这意味着每个维度的大小必须是一致的。可以通过嵌套使用数组构造器来构建多维数组。当构造多维数组时，每个处于同一层次的数组构造器必须生成相同数量的子数组，以保持维度一致。在构造多维数组时，可以省略关键词 ARRAY，直接使用数组构造器语法。如下两个 SELECT 语句产生了相同的结果：

```
=> SELECT ARRAY[ARRAY[1,2], ARRAY[3,4] , ARRAY[5,6]];
array
---------------------
{{1,2},{3,4},{5,6}}
(1 row)

=> SELECT ARRAY[[1,2],[3,4],[5,6]];
array
---------------------
{{1,2},{3,4},{5,6}}
(1 row)
```

使用 ARRAY 构建的数组值的下标总是从 1 开始。子查询的结果也可以用于构造数组，即 ARRAY 关键词后面跟随包含在圆括号内的子查询。示例如下：

```
=> SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
array
------------------------------------------------------------------------------------------
{3331,3544,3543,2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412,2413}
(1 row)
```

子查询必须返回单列值。返回结果中一维数组的每一个元素对应于子查询结果中的每一行，元素类型匹配子查询的输出列。

行构造器

行构造器是一种表达式，用于基于其成员字段的值来构建一个行值（也称为组合值或复合值）。一个行构造器由关键词 ROW、一个左圆括号“(”、表示行字段值的表达式列表（多个表达式用逗号分隔）以及一个右圆括号“)”组成。示例如下：

```
=> SELECT ROW(1,2.6,'this is a row test');
row
------------------------------
(1,2.6,"this is a row test")
(1 row)
```

行构造器可以使用求值语法 *rowvalue.**，如下示例中，如果表 test 包含列 c1 和 c2，下列查询是相同的：

```
=> SELECT ROW(test.*, 41) FROM test;
=> SELECT ROW(test.c1, test.c2, 41) FROM test;
```

默认情况下，由 ROW 表达式创建的值属于一种匿名记录类型。这意味着它们没有预定义的类型名称，但可以包含多个命名的字段。如果有必要，可以将这些匿名记录类型值转换成一种已命名的复合类型。这种转换可能是针对数据库中已存在的表的行类型，或者是通过 CREATE TYPE AS 语句创建的复合类型。

```
=> CREATE TABLE testtable(c1 int, c2 float, c3 text);
=> CREATE FUNCTION getcol1(testtable) RETURNS int AS 'SELECT $1.c1' LANGUAGE SQL;
=> SELECT getcol1(ROW(1,2.6,'this is a row test'));
getcol1
---------
1
(1 row)
```

如下示例中，查询需要执行类型转换来指示调用的函数。

```
=> CREATE TYPE rowtype AS (c1 int, c2 text, c3 numeric);
=> CREATE FUNCTION getcol1(rowtype) RETURNS int AS 'SELECT $1.c1' LANGUAGE SQL;
=> SELECT getcol1(ROW(1,2.6,'this is a test'));
ERROR: function getcol1(record) is not unique

-- 进行造型
=> SELECT getcol1(ROW(1,2.6,'this is a test')::testtable);
getcol1
---------
1
(1 row)

=> SELECT getcol1(CAST(ROW(111,'this is a test',2.6) AS rowtype));
getcol1
---------
111
(1 row)
```

表达式计算规则

在 PieCloudDB 中，并没有定义子表达式的计算顺序。这意味着表达式中的操作符或函数的计算顺序并不是固定的，即不必按照从左向右或者其他固定顺序执行计算。

如果表达式的计算结果可以通过仅计算部分子表达式来确定，则其他子表达式可能不会被计算。如下示例中，testfunc() 可能根本不会被调用：

```
SELECT true OR testfunc();
SELECT testfunc() OR true;
```

要强制指定计算顺序，可使用 CASE 结构。例如，在 WHERE 子句中避免除数为零，下面的方法不是很可靠：

```
SELECT ... WHERE x <> 0 AND y/x > 2.5;
```

针对上述示例，建议使用如下的方法指定计算顺序：

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 2.5 ELSE false END;
```

需要注意的是，这种 CASE 结构的用法会使优化器无法进行优化，因此如非必要不建议使用它。

7.3 使用 WITH 语句

在一个大型的 SELECT 查询中，WITH 子句可以提供辅助语句进行子查询或执行数据修改操作，辅助语句可以是 SELECT、INSERT、UPDATE 或 DELETE。

在 WITH 子句中，可以包含多个 SELECT 语句，但只能包含一个数据修改命令（INSERT、UPDATE 或 DELETE），并且这个数据修改命令必须是 CTE 中的最后一个语句。

默认情况下，WITH 子句启用 RECURSIVE 关键字，即允许 CTE（Common Table Expression）定义递归查询。通过将服务器配置参数 *gp_recursive_cte* 设置为“FALSE”，可以禁用 WITH 子句的 RECURSIVE 关键字。如下示例说明数据库当前已启用 RECURSIVE 关键字。

```
=> SHOW gp_recursive_cte;
gp_recursive_cte
------------------
on
(1 row)
```

执行如下命令可以禁用 RECURSIVE 关键字。

```
SET gp_recursive_cte TO 'off';
```

在 WITH 子句中使用 SELECT

WITH 子句中的 SELECT 命令在每次执行父查询时只计算一次，即使在父查询或 WITH 子句的同级中多次引用了该命令。这意味着数据库引擎会优化执行计划，避免重复计算。通过将复杂的查询分解成简单的部分，WITH 子句可以提高查询的可读性和维护性。

如下示例为，查询仅在顶部排名的销售区域中显示每个产品的销售总额，WITH 子句定义了两个辅助语句 reg_sales 和 top_regions；如果不使用 WITH 子句编写查询，则需要使用两级嵌套 SELECT，相比较来说，WITH 子句会比较简单。

```
WITH reg_sales AS (
  SELECT region, SUM(amount) AS total_sales
     FROM neworders
     GROUP BY region
   ), top_regions AS (
     SELECT region
       FROM reg_sales
         WHERE total_sales > (SELECT SUM(total_sales)/10 FROM reg_sales)
      )SELECT region, product, SUM(quantity) AS pro_units, SUM(amount) AS pro_sales
        FROM neworders
          WHERE region IN (SELECT region FROM top_regions)
            GROUP BY region, product;
```

递归查询

当 WITH 子句启用可选 RECURSIVE 关键字时，WITH 子句中的查询可以引用自身的输出。WITH 递归查询的语法如下：

```
non_recursive_term UNION [ALL] recursive_term
```

如下示例为，使用递归查询计算从 1 到 1000 的整数之和。

```
=> WITH RECURSIVE tt(n) AS (VALUES (1)
          UNION ALL
          SELECT n+1 FROM tt WHERE n < 1000 )SELECT sum(n) FROM tt;

sum
--------
 500500
(1 row)
```

WITH 包含 *UNION [ALL]* 的递归查询的执行步骤如下：

1. 计算非递归项。对于 UNION（而不是 UNION ALL），会丢弃重复的行，包括递归查询结果中的所有剩余行，并将它们放在临时工作表中。
2. 只要工作表非空，重复执行以下步骤：
    1. 计算递归项，将工作表的当前内容替换为递归自引用。对于 UNION（而不是 UNION ALL），放弃重复的行以及复制任何先前结果的行。在递归查询的结果中包含所有剩余的行，并将它们放在临时"中间表"中。
    2. 将当前工作表的内容替换为中间表的内容，然后清空中间表。

严格来说，递归过程其实是迭代计算的过程。在上述示例中，每个步骤的工作表只有一行，并且在后续步骤中采用从 1 到 1000 的值。在第 1000 步中，由于 WHERE 子句没有值输出，因此查询终止。

WITH 递归查询通常用于处理具有层次结构或树状结构的数据。如下示例为，通过给出的表 product_parts（唯一包含数据的表），查询产品相关的所有直接和间接子结果。

```
WITH RECURSIVE temp_parts(sub_part, pro_part, quantity) AS (
    SELECT sub_part, pro_part, quantity FROM product_parts WHERE pro_part = 'our_product'
      UNION ALL
       SELECT p.sub_part, p. pro_part, p.quantity
         FROM temp_parts pr, parts p
           WHERE p. pro_part = pr.sub_part
      )SELECT sub_part, SUM(quantity) as total_quantity
        FROM temp_parts GROUP BY sub_part ;
```

使用递归查询计算树形结构的遍历时，用户可能希望按深度优先或宽度优先的顺序对结果进行排序。为了创建深度优先顺序，可以为每个结果行计算迄今为止访问过的行的数组。如下示例为，使用 link 字段搜索表 tree 的查询。

```
WITH RECURSIVE query_tree(id, link, data) AS (
   SELECT t.id, t.link, t.data
   FROM tree t
 UNION ALL
   SELECT t.id, t.link, t.data
   FROM tree t, query_tree qt
   WHERE t.id = qt.link
)
SELECT * FROM query_tree;
```

要添加深度优先排序信息，可以按照如下示例改写：

```
WITH RECURSIVE query_tree(id, link, data, path) AS (
   SELECT t.id, t.link, t.data, ARRAY[t.id]
   FROM tree t
 UNION ALL
   SELECT t.id, t.link, t.data, path || t.id
   FROM tree t, query_tree qt
   WHERE t.id = qt.link
)
SELECT * FROM query_tree ORDER BY path;
```

要创建宽度优先顺序，可以添加跟踪搜索深度的列，示例如下：

```
WITH RECURSIVE query_tree(id, link, data, depth) AS (
   SELECT t.id, t.link, t.data, 0
   FROM tree t
 UNION ALL
   SELECT t.id, t.link, t.data, depth + 1
   FROM tree t, query_tree qt
   WHERE t.id = qt.link
)
SELECT * FROM query_tree ORDER BY depth;
```

PieCloudDB 支持内置语法计算深度优先或宽度优先的排序列。示例如下：

```
WITH RECURSIVE query_tree(id, link, data) AS (
   SELECT t.id, t.link, t.data
   FROM tree t
 UNION ALL
   SELECT t.id, t.link, t.data
   FROM tree t, query_tree qt
   WHERE t.id = qt.link
) SEARCH DEPTH FIRST BY id SET ordercol
SELECT * FROM query_tree ORDER BY ordercol;

WITH RECURSIVE query_tree(id, link, data) AS (
   SELECT t.id, t.link, t.data
   FROM tree t
 UNION ALL
   SELECT t.id, t.link, t.data
   FROM tree t, query_tree qt
   WHERE t.id = qt.link
) SEARCH BREADTH FIRST BY id SET ordercol
SELECT * FROM query_tree ORDER BY ordercol;
```

上述示例中 SEARCH 子句指定是否需要深度优先搜索或宽度优先搜索、要跟踪排序的列以及包含用于排序的数据的列名称，该列将隐式添加到 CTE 的输出行。

使用 WITH 递归时，必须确保查询的递归部分最终不会返回元组，否则查询将无限循环。对于某些查询，使用 UNION（而不是 UNION ALL）可以通过丢弃先前重复的输出行来确保查询的递归部分最终不返回元组。

PieCloudDB 支持内置语法 CYCLE 子句来简化循环检测。CYCLE 子句首先指定要跟踪循环检测的列，然后列名称显示指定是否已检测到循环，最后另一列的名称将跟踪路径。循环和路径列将隐式添加到 CTE 的输出行。示例如下：

```
WITH RECURSIVE search_image(id, link, data, depth) AS (
    SELECT i.id, i.link, i.data, 1
    FROM image i
  UNION ALL
    SELECT i.id, i.link, i.data, i.depth + 1
    FROM image i, search_ image si
    WHERE i.id = si.link
) CYCLE id SET is_cycle USING path
SELECT * FROM search_ image;
```

测试循环查询的一个有用技巧是在父查询中加入 LIMIT。如下示例中，如果没有 LIMIT，此查询将永远循环，但不建议在实际业务中使用此技巧。

```
WITH RECURSIVE tt(n) AS (
    SELECT 1
  UNION ALL
    SELECT n+1 FROM tt
)
SELECT n FROM tt LIMIT 1000;
```

WITH 子句中的数据修改语句

对于 SELECT 命令，可以在 WITH 子句中使用数据修改命令 INSERT、UPDATE 或 DELETE，以实现在同一查询中执行不同的操作。

WITH 子句中的数据修改语句只执行一次，并且始终执行到结束，这与主查询是否读取了所有（或任何）输出无关，而在 WITH 子句中的 SELECT 只有在主查询要求输出时才会继续执行。

如下示例为在 WITH 子句中使用 DELETE 从表 products 中删除行，并通过 RETURNING 子句返回其内容。

```
WITH del_rows AS (
    DELETE FROM products
    WHERE "date" >= '2020-10-01' AND "date" < '2023-11-01'
    RETURNING *
)
SELECT * FROM del_rows;
```

WITH 子句中的数据修改语句必须包含 RETURNING 子句，且为 RETURNING 子句提供输出，形成可供后续查询引用的临时表。如果 WITH中的数据修改语句缺少 RETURNING 子句，则会返回错误。

如果启用可选的 RECURSIVE 关键字，则不允许在数据修改语句中进行递归自引用。在某些情况下，可以通过引用递归的输出来绕过这个限制。如下示例中的查询将删除产品相关的所有直接和间接子部分。

```
WITH RECURSIVE temp_parts(sub_part, pro_ part) AS (
   SELECT sub_part, part FROM parts WHERE pro_ part = 'our_product'
  UNION ALL
   SELECT p.sub_part, p. pro_ part
   FROM temp_parts pr, parts p
   WHERE p. pro_ part = pr.sub_part
)
DELETE FROM parts
    WHERE pro_ part IN (SELECT pro_ part FROM temp_parts);
```

WITH 子句中的子语句与主查询同时执行，因此将在快照中执行在 WITH 中的数据修改语句，但语句的效果在目标表上不可见。RETURNING 返回的数据是在不同子语句和主查询之间传递更改的唯一方法。如下示例中，外部 SELECT 返回 WITH 子句中 UPDATE 操作之前的原始价格。

```
WITH tt AS (
   UPDATE products SET price = price * 1.25
   RETURNING *
   )SELECT * FROM products;
```

而在如下示例中，外部 SELECT 则返回更新后的数据。

```
WITH tt AS (
   UPDATE products SET price = price * 1.25
   RETURNING *
   )SELECT * FROM tt;
```

需要注意的是，不支持在单个语句中对同一行执行两次 UPDATE。这种情况下，只有一个修改发生了，但是不容易（有时也不可能）预测是哪一个 UPDATE 发生了修改。

此外，在 WITH 子句中数据修改语句所操作的目标表都不能有条件规则，包括 ALSO 和 INSTEAD 等。

7.4 使用函数和操作符

PieCloudDB 支持使用内建函数、窗口函数、高级聚集函数、用户定义函数和运算符。在 psql 命令行工具中，可以使用以下命令来列出自定义函数和运算符：

- \df：用于列出当前数据库中所有可用的用户定义函数。
- \do：用于列出当前数据库中所有可用的用户定义运算符。

PieCloudDB 将函数的执行结果与环境因素的关系分为以下几类：

- VOLATILE：表示函数的结果可能会因每次调用而变化，即使输入参数未变。
- STABLE：表示函数的结果在同一个事务中是一致的，但在不同事务中可能会变化。
- IMMUTABLE：表示函数的结果只依赖于输入参数，与外部环境无关。

在 PieCloudDB 中，如果在 *CREATE FUNCTION* 命令中未指定稳定性类别，则默认为 VOLATILE。

内置函数和运算符

PieCloudDB 支持的内建函数和运算符的类别如下表所示。

运算符/函数分类	说明
逻辑运算符	常用的逻辑运算符有 AND、OR 和 NOT，返回值为 TRUE、FALSE 和 NULL（代表“未知”）。
比较运算符	所有的比较运算符都是二元运算符，返回布尔类型的值。
数学函数和运算符	数学函数包括定值函数、三角函数、反三角函数、指数函数、对数函数和随机函数等。VOLATILE 函数包括 *random()* 和 *setseed()*。
字符串函数和运算符	此上下文中的字符串包括字符、字符串数据类型（VARCHAR）转换和文本类型的值。其中，VOLATILE 函数包括所有内建转换函数；STABLE 函数包括 *convert()* 和*pg_client_encoding()*。
二进制串函数和运算符	用于处理和操作 bytea 类型的值。
位字符串函数和运算符	用于检验和操作位字符串（即位类型和位数据类型转换的值）的函数和运算符。
模式匹配	PieCloudDB 提供了三种不同的模式匹配方法，分别是传统的 SQL LIKE 运算符、新增的 SIMILAR TO 运算符（在 SQL:1999 中添加）和 POSIX 样式的正则表达式。除了判断字符串与模式是否匹配的基本功能，运算符和函数还可用于提取或替换匹配的子字符串以及在匹配位置拆分字符串。
数据类型格式转换函数	格式转换函数用于将各种数据类型（日期/时间、整数、浮点、数字）转换为格式化字符串或者从格式化字符串转换为指定数据类型。其中，STABLE 函数包括*to_char()* 和 *to_timestamp()*。
日期/时间函数和运算符	日期/时间函数和运算符采用的时间或时间戳输入类型有两种，一种是带时区的时间或带时区的时间戳，另一种是不带时区的时间或不带时区的时间戳。其中，VOLATILE 函数包括 *timeofday()*；STABLE 函数包括： • *agecurrent_date()* • *current_time()* • *current_timestamp()* • *localtime()* • *localtimestamp()* • *now()*
枚举函数	枚举 (enum) 类型是包含静态、有序值集的数据类型。枚举函数无需对枚举类型的特定值进行硬编码。
几何函数和运算符	几何类型的点（point）、框（box）、线段（lseg）、线（line）、路径（path）、多边形（polygon）和圆（circle）都支持使用函数和运算符。
网络地址函数和运算符	用于处理和操作网络地址相关数据类型，例如 IPv4、IPv6、MAC 地址和网络掩码等。
全文搜索函数和运算符	用于全文搜索操作。

运算符/函数分类	说明
序列操作函数	用于操作序列对象，也称为序列生成器（或简称序列）。序列对象是使用 `CREATE SEQUENCE` 创建的，通常用于为表的行生成唯一标识符。其中，VOLATILE 函数包括 `currval()`、`lastval()`、`nextval()` 和 `setval()`。
条件表达式	仅涉及可用的符合 SQL 的条件表达式。
数组函数和运算符	用于操作和处理数组类型的数据。数组是一种特殊的聚合数据类型，通过将多个元素按照某种顺序排列在一起，然后以单个数据类型作为输出。所有数组函数都是 STABLE 的。
范围类型函数和运算符	用于操作和处理范围（Range）类型的数据。范围类型是指一组具有顺序的元素，可以用来表示各种域，例如数字区间、日期范围等。
聚集函数	根据一组输入值计算单个结果。
窗口函数	必须使用 OVER 子句调用内置窗口函数，对“窗口”范围内的当前记录及其相关记录执行计算。仅当聚集函数后面有 OVER 子句时，才能充当窗口函数。
子查询表达式	可用的符合 SQL 的子查询表达式，所有表达式形式都返回布尔（True/False）结果。
行及数组比较	用于在值组之间进行多重比较，所有表达式形式都返回布尔（True/False）结果。
集合返回函数	返回值为多行的函数。其中，VOLATILE 函数包括 `generate_series()`。
系统信息函数	用于提取会话和系统信息。其中，STABLE 函数包括： • 所有会话信息函数 • 所有访问权限查询函数 • 所有 Schema 可见性查询函数 • 所有系统目录信息函数 • 所有注释信息函数 • 所有事务 ID 和快照查询函数
系统管理函数	用于控制和监视 PieCloudDB 的安装。其中，VOLATILE 函数包括： • `set_config()` • `pg_cancel_backend()` • `pg_reload_conf()` • `pg_rotate_logfile()` • `pg_start_backup()` • `pg_stop_backup()` • `pg_size_pretty()` • `pg_ls_dir()` • `pg_read_file()` • `pg_stat_file()` STABLE 函数包括 `current_setting()`。
触发器函数	PieCloudDB 支持内置触发器函数 `suppress_redundant_updates_trigger()`，它用于生成一个触发器，来阻止在触发行为中导致重复更新的操作。
事件触发器函数	用于从事件触发器中检索信息。

运算符/函数分类	说明
统计信息函数	用于检验 *CREATE STATISTICS* 命令所定义的复杂统计信息。
JSON 函数	用于处理和创建 JSON 数据。
XML 函数	用于处理 XML 类型的值。

窗口函数

PieCloudDB 所支持的窗口函数如下表所示。所有的窗口函数都是 IMMUTABLE。

函数	完整语法	返回值类型	描述
cume_dist()	*CUME_DIST() OVER ([PARTITION BY partition_expr] ORDER BY order_expr)*	DOUBLE PRECISION	计算一组值中一个值的累积分布。具有相等值的行总是具有相同的累积分布值。
dense_rank()	*DENSE_RANK() OVER ([PARTITION BY partition_expr] ORDER BY order_expr)*	BIGINT	计算一个有序行组中一行的无跳跃排名。具有相等值的行会得到相同的排名值。
first_value()	*FIRST_VALUE(expr) OVER ([PARTITION BY partition_expr] ORDER BY order_expr [ROWS | RANGE frame_expr])*	和输入 *expr* 的类型相同	返回一个有序值集合中的第一个值。
lag()	*LAG(expr [, offset] [, default]) OVER ([PARTITION BY partition_expr] ORDER BY order_expr)*	和输入 *expr* 的类型相同	在不做自连接的情况下，提供对于同一个表中前一行的访问。默认的 *offset* 为 1。*default* 用于设置当偏移量超出窗口范围之外时要返回的值。如果没有指定 *default*，默认是空值。
last_value()	*LAST_VALUE(expr) OVER ([PARTITION BY partition_expr] ORDER BY order_expr [ROWS | RANGE frame_expr])*	和输入 *expr* 的类型相同	返回一个有序值集合中的最后一个值。
lead()	*LEAD (expr [,offset] [,expr default]) OVER ([PARTITION BY partition_expr] ORDER BY order_expr)*	和输入 *expr* 的类型相同	在不做自连接的情况下，提供对于同一个表中前一行的访问。如果没有指定 *offset*，默认偏移量是 1。*default* 设置当偏移量超出窗口范围之外时要返回的值。如果没有指定 *default*，默认值是空值。

函数	完整语法	返回值类型	描述
ntile()	`NTILE(expr) OVER ( [PARTITION BY partition_expr] ORDER BY order_expr )`	BIGINT	把一个有序数据集划分成一些桶（Buckets）并且为每一行分配一个桶号。
nth_value()	`NTH_VALUE (expr, n) OVER (PARTITION BY partition_expr ORDER BY order_expr)`	和输入 `expr` 的类型相同	用于在一组行中获取表达式所指定行的值，如果没有则为 Null。
percent_rank()	`PERCENT_RANK () OVER ( [PARTITION BY partition_expr] ORDER BY order_expr )`	DOUBLE PRECISION	用于计算某个值在排序结果中的百分位排名。
rank()	`RANK() OVER ( [PARTITION BY partition_expr] ORDER BY order_expr )`	BIGINT	计算一行在一个有序值组中的排名。根据排名标准有相等值的行得到相同的排名。被占用的行数被加到排名数上来计算下一个排名值。在这种情况下，排名可能不是连续的数字。
row_number()	`ROW_NUMBER () OVER ( [PARTITION BY partition_expr] ORDER BY order_expr )`	BIGINT	为窗口分区中的每一行或者查询中的每一行分配一个唯一的编号。

高级聚集函数

下表中 PieCloudDB 所支持的高级聚集函数是对 PostgreSQL 数据库的扩展，如下表所示。所有的高级聚集函数都是 IMMUTABLE。

函数	完整语法	返回值类型	描述
median()	`MEDIAN(expr)` 例如：`SELECT dept_id, MEDIAN(salary) FROM employees GROUP BY dept_id;`	• DOUBLE PRECISION • INTERVAL • TIMESTAMP WITHOUT TIME ZONE • TIMESTAMP WITH TIME ZONE	用于计算指定一组输入值的中位数。
pivot_sum()	`pivot_sum (array ['A1', 'A2'], attr, value)`	• INT[] • BIGINT[] • DOUBLE PRECISION[]	使用 SUM 来解决重复项的 Pivot 聚集。

自定义函数

PieCloudDB 支持的用户定义的函数一般分为如下四种：

- 查询语言（SQL）函数。

SQL 语言中的命令集合可以打包在一起并定义为函数。除了 SELECT 查询之外，还可以包括数据修改查询（INSERT、UPDATE 和 DELETE）命令等。

- 过程语言（PL）函数。

  PieCloudDB 允许用 SQL 和 C 之外的其他语言编写用户定义函数，通常称为过程语言(PL)，例如 PL/pgSQL、PL/Python、PL/Perl、PL/Tcl 等。
- 内部函数。

  内部函数是用 C 编写的函数，并静态链接到数据库服务器。
- C 语言函数。

  C 语言函数是用 C 或与 C 兼容的语言（例如 C++）编写的函数。C 语言函数被编译成动态可加载对象（也称为共享库）并由服务器按需加载。动态加载功能是“C 语言”函数与“内部”函数的主要区别。

本节主要介绍最简单的用户自定义 SQL 函数。PieCloudDB 支持使用 *CREATE FUNCTION* 语句创建用户自定义的 SQL 函数。用户定义的函数默认被声明为 VOLATILE，这意味着函数的返回值在每次调用时都可能变化。如果用户自定义的函数是 IMMUTABLE 或者 STABLE，必须在创建该函数时明确指定。

SQL 函数的主体必须是由分号（;）分隔的 SQL 语句列表。最后一条语句后面的分号是可选的。SQL 函数会执行 SQL 语句列表，并返回列表中最后一个查询的结果。除非函数声明为返回“void”，否则最后一个语句必须是 SELECT，或者具有 RETURNING 子句的 INSERT、UPDATE 或 DELETE。

7.5 查询解析

查询计划（Query Plan）是数据库根据查询语句和表结构生成的执行策略，它详细描述了数据库如何执行一个特定的查询。这包括数据的检索方式、表之间的连接顺序和方法、索引的选择使用等关键信息。查询计划的目标是找到执行查询的最高效路径。

查看查询计划是性能调优的重要途径。PieCloudDB 支持使用 *EXPLAIN* 命令查看指定查询的计划，并展示优化器对该查询计划估计的代价；同时，还可以使用 *EXPLAIN ANALYZE* 命令在运行该查询语句的同时显示其查询计划。

查看 EXPLAIN 输出

查询计划通常被表示为一棵树，其中包含多个节点，这些节点以父子关系的方式组织，显示了查询操作之间的层次结构。在文本格式的查询计划中，缩进用来表示操作符之间的父子关系，从而展示查询的执行顺序和操作方式.

在数据库的查询计划树中，每个节点都代表了一个基本操作，例如数据检索、数据转换或数据的聚合和排序等。这些节点按照从底层到顶层的顺序组织，数据行从底层节点开始，逐层向上流动，直至查询结果产生。常见的扫描节点类型有顺序扫描、索引扫描、位图索引扫描，除扫描节点外还有哈希连接节点、嵌套循环连接节点、合并连接节点、排序节点、聚合节点和过滤节点等。

EXPLAIN 命令在数据库系统中用来显示 SQL 查询的执行计划。执行计划由一系列的节点组成，每个节点描述了查询操作的一个阶段。数据库的查询优化器为每个节点计算了成本，这通常包括时间成本和 I/O 成本。成本估计反映了执行该节点所需的资源。

本文通过具体示例介绍如何查看 EXPLAIN 的输出信息。根据如下 SQL 语句创建示例表 t_exp，插入数据后执行 EXPLAIN。

```
=> CREATE TABLE t_exp (x INT, y INT);
=> INSERT INTO t_exp VALUES (1,2) ,(3,4) ,(5,6) ,(7,8) ,(9,10) ,(11,12),(13,14),(15,16),(17,18),
(19,20),(21,22),(23,24),(25,26),(27,28),(29,30),(31,32),(33,34),(35,36),(37,38),(39,40);

=> EXPLAIN SELECT * FROM t_exp;

                          QUERY PLAN
------------------------------------------------------------------------------
 Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..1.33 rows=20 width=8)
   ->  Seq Scan on t_exp  (cost=0.00..1.07 rows=7 width=8)
 Optimizer: Postgres query optimizer
(3 rows)
```

上述示例的查询中不包含 WHERE 子句，所以会进行全表扫描。查询计划输出结果中主要字段的含义如下：

- Gather Motion：表示并行查询计划节点。本示例中，查询被分成了 3 个并行执行的 Segment。
- Seq Scan：表示顺序扫描节点。本示例中，表 t_exp 使用了简单的顺序扫描计划 Seq Scan。
- cost：表示执行计划的开销，一般以磁盘页面读取为衡量单位。本示例中，顺序扫描节点的成本为 0.00 到 1.07 单位，整个查询的成本为 0.00 到 1.33 单位。
- rows：表示查询的估计结果行数。本示例中，顺序扫描节点估计返回 7 行，整个查询估计返回 20 行。
- width：表示计划节点的每行宽度，单位为字节。本示例中，计划节点的每行宽度为 8 字节。

如下示例的 EXPLAIN 中增加了 WHERE 条件，输出结果如下：

```
=> EXPLAIN SELECT * FROM t_exp WHERE x<20;

                          QUERY PLAN
------------------------------------------------------------------------------
 Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..1.26 rows=13 width=8)
   ->  Seq Scan on t_exp  (cost=0.00..1.08 rows=4 width=8)
         Filter: (x < 20)
 Optimizer: Postgres query optimizer
(4 rows)
```

上述查询计划输出结果中，WHERE 子句被附加到 Seq Scan 计划节点的“过滤”条件中，即计划节点会检查其扫描的每一行是否满足该条件，并仅输出满足该条件的行，由此输出行的估计值已减少为 13 行。

如下示例的 EXPLAIN 中增加 LIMIT 条件，输出结果如下：

```
=> EXPLAIN SELECT * FROM t_exp WHERE x<20 LIMIT 2;

                        QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.24..0.43 rows=2 width=8)
   ->  Gather Motion 3:1  (slice1; segments: 3)  (cost=0.24..0.81 rows=6 width=8)
         ->  Limit  (cost=0.24..0.73 rows=2 width=8)
               ->  Seq Scan on t_exp  (cost=0.00..1.08 rows=4 width=8)
                     Filter: (x < 20)
 Optimizer: Postgres query optimizer
(6 rows)
```

由于上述示例添加了 LIMIT 条件，输出结果中的 *Limit* 表示限制节点，限制最多返回 2 行查询结果。需要注意的是，实际查询检索的行数要少于 Index Scan 节点检索的行数，Index Scan 节点的总成本（0.00 到 1.08 单位）和行数（4 行）显示结果并不是实际估计成本。

查看 EXPLAIN ANALYZE 输出

EXPLAIN ANALYZE 在执行查询的同时获取查询计划信息。除了 EXPLAIN 展示的信息外，EXPLAIN ANALYZE 还会展示如下附加信息：

- Planning Time：表示查询优化器生成查询计划所花费的时间（单位为毫秒）。
- Executor memory：表示查询计划中每个 Slice 使用的内存。
- Memory used：表示查询执行过程中使用的内存总量。
- Execution Time：表示查询的实际执行时间（单位为毫秒）。

如下示例为执行 EXPLAIN ANALYZE 的查询计划输出信息。

```
=> EXPLAIN ANALYZE SELECT * FROM t_exp WHERE x<20;

                    QUERY PLAN
-------------------------------------------------------------------------------
 Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..1.26 rows=13 width=8) (actual
 time=7.370..9.580 rows=10 loops=1)
   ->  Seq Scan on t_exp  (cost=0.00..1.08 rows=4 width=8) (actual time=1.546..1.754 rows=5
 loops=1)
         Filter: (x < 20)
         Rows Removed by Filter: 2
 Planning Time: 6.529 ms
   (slice0)    Executor memory: 50K bytes.
   (slice1)    Executor memory: 50K bytes avg x 3 workers, 50K bytes max (seg0).
 Memory used:  276480kB
 Optimizer: Postgres query optimizer
 Execution Time: 44.737 ms
(10 rows)
```

根据上述查询计划的输出结果可以得出，运行该查询花掉的总时间是 44.737 ms，生成执行计划所花费的时间为 6.529 ms，查询执行过程中使用的内存总量为 276480 kB。“Rows Removed by Filter: 2”说明在执行查询时有两行数据不满足查询条件被过滤掉，“(slice0) Executor memory”表示单个 slice0 使用的内存量为50K

bytes，“(slice1) Executor memory”表示 slice1 平均每个 Worker 使用的内存量为 50K bytes，且最大使用量存在于 seg0。

查询优化建议

如果一个查询的执行效果不佳，可以使用 *EXPLAIN* 或 *EXPLAIN ANALYZE* 命令来获取查询计划和执行统计信息，并决定是否需要干预优化器的选择。在 PieCloudDB 中，优化前应考虑如下因素：

- 查询计划的执行时间是否过长

  查询计划的执行时间如果过长，可能会影响到应用程序的响应速度和用户体验。因此，分析查询计划并识别出消耗时间最多的操作是性能优化的关键步骤。例如，一个索引扫描花费的时间比预期长，说明该索引可能过期，可以通过重建索引的方式进行优化。
- 优化器的估计行数是否有偏差

  数据库查询优化器使用统计信息来估计查询操作影响的行数。如果这些估计与实际返回的行数有较大差异，可能会导致优化器选择不是最优的查询计划。使用 *EXPLAIN ANALYZE* 命令来查看查询的执行计划，并比较估计的行数与实际返回的行数。
- 查询计划是否很早就应用了选择性谓词

  如果查询优化器没有正确估计查询谓词的选择度，可能会导致不必要的数据被扫描，从而影响查询性能。在相关列上收集更多的统计信息，以帮助优化器更准确地估计谓词的选择性。用户也可以尝试重新调整 SQL 语句中 WHERE 子句的排序，以优化谓词的评估顺序。
- 优化器是否选择了最佳连接顺序

  在多表连接查询中，连接顺序对性能有重要影响。优化器需要选择最佳的连接顺序，以减少处理的数据量。确保优化器选择了最佳的连接顺序，例如，尽早进行可以消除最多行的连接。在连接列上收集更多的统计信息，以帮助优化器做出更好的决策。
- 优化器是否有选择地扫描分区表

  如果查询涉及分区表，优化器应该只扫描满足查询谓词的子分区，以提高查询效率。检查查询计划，确保优化器选择了正确的分区进行扫描。如果父表的扫描结果返回了非零行，可能表明扫描异常。
- 优化器是否选择了 Hash 聚集和 Hash 连接操作

  Hash 操作（如 Hash Join 或 Hash Aggregate）通常比传统的连接或聚集操作更快，但需要足够的内存来存储中间结果。确保系统有足够的内存来支持 Hash 操作。使用 *EXPLAIN ANALYZE* 可以查看查询执行过程中的内存使用情况，以判断优化器是否选择了 Hash 操作。

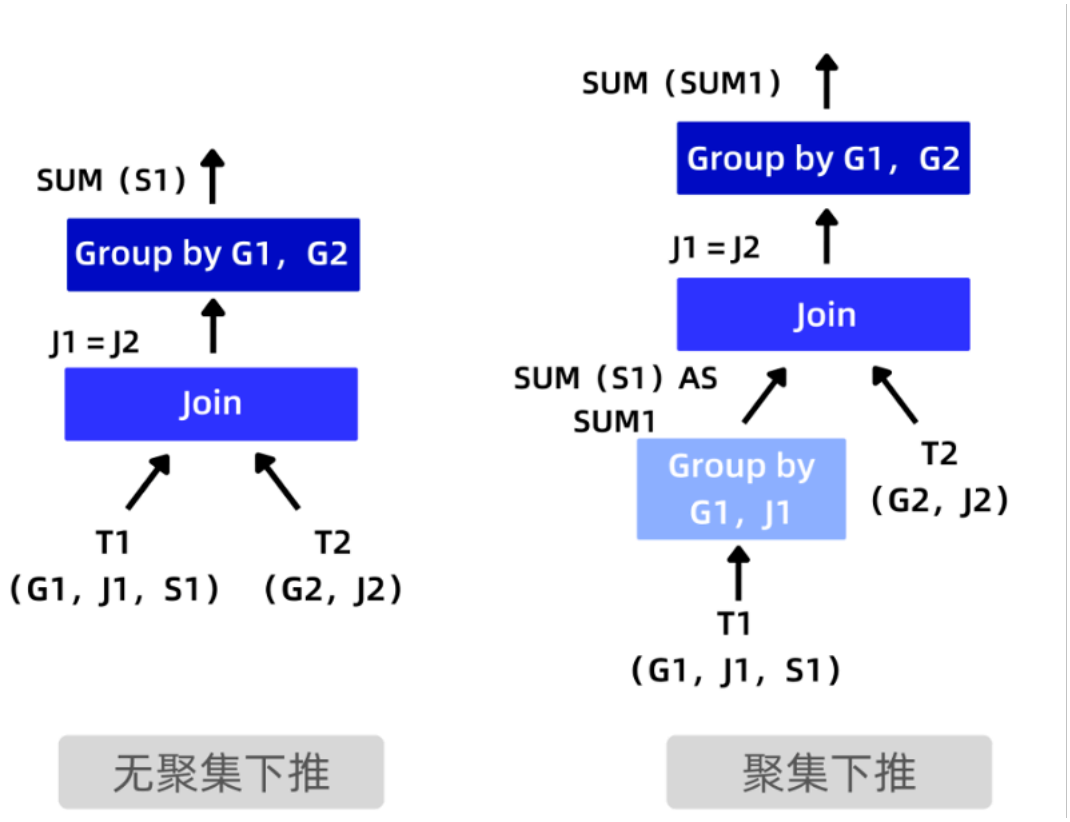
8 查询性能优化

8.1 聚集下推

聚集下推（Pushdown Aggregation）是一种数据库查询优化技术，它通过将聚合操作下推至数据源来减少传输和处理数据的开销，从而提高查询性能和效率。

聚集下推的实现原理在于在查询执行过程中尽早执行聚合操作，并将这些操作下推至数据源进行处理。具体来说，当生成查询计划时，优化器会识别并下推聚合操作至数据源，使得聚合计算能在数据源处完成。这种方法减少了数据传输量，减轻了查询引擎的负担，并有效利用了数据源的计算资源进行局部聚合。

下图展示了无聚集下推与有聚集下推的执行流程对比，展示了聚集下推的主要实现原理和优势。



在未采用聚集下推的场景中，表 t1 和 t2 会首先执行连接操作，随后在连接结果的数据集上进行聚集操作。相对地，在采用聚集下推的场景中，会在对表 t1 和 t2 执行连接操作之前，先对表 t1 执行一次聚集操作。这种做法可以在某些情况下显著减少表 t1 在连接操作中涉及的数据量，进而提升查询性能。

为了精确提升查询效率，PieCloudDB 会同时生成上述两种执行计划，并依据代价模型的估算结果选择成本较低的方案。当涉及更多表的连接操作时，优化器会尝试将聚集操作下推至不同的连接层级，并通过比较各方案的成本来确定最优的下推位置。

本文将通过具体的查询示例来展示聚集下推的效果。首先，根据以下 SQL 语句创建示例表 t_agg 并插入数据，然后执行 ANALYZE 以收集统计信息。

```
CREATE TABLE t_agg (x INT, y INT);

INSERT INTO t_agg SELECT i % 30, i % 30 FROM generate_series(1, 10240) i;

ANALYZE t_agg;
```

对示例表 t_agg 执行如下查询语句：

```
SELECT t1.x, sum(t2.y + t3.y), count(*) FROM t_agg t1
   JOIN t_agg t2 ON t1.x = t2.x JOIN t t3 ON t2.x = t3.x
      GROUP BY t1.x;
```

对于没有聚集下推的场景，执行查询的计划信息如下：

```
=> EXPLAIN (ANALYZE, COSTS OFF) SELECT t1.x, sum(t2.y + t3.y), count(*) FROM t_agg t1 JOIN
 t_agg t2 ON t1.x = t2.x JOIN t_agg t3 ON t2.x = t3.x GROUP BY t1.x;

                QUERY PLAN
-----------------------------------------------------------------------------------
Gather Motion 3:1 (slice1; segments: 3) (actual time=153884.859..274102.066 rows=30
 loops=1)
 -> HashAggregate (actual time=274100.004..274100.011 rows=12 loops=1)
    Group Key: t1.x
    Peak Memory Usage: 0 kB
  -> Hash Join (actual time=38.717..100579.782 rows=477571187 loops=1)
     Hash Cond: (t1.x = t3.x)
     Extra Text: (seg0) Hash chain length 341.4 avg, 342 max, using 12 of 131072 buckets.
   -> Hash Join (actual time=2.088..429.203 rows=1398787 loops=1)
      Hash Cond: (t1.x = t2.x)
      Extra Text: (seg0) Hash chain length 341.4 avg, 342 max, using 12 of 131072 buckets.
     -> Redistribute Motion 3:3 (slice2; segments: 3) (actual time=0.044..4.590 rows=4097
 loops=1)
        Hash Key: t1.x
      -> Seq Scan on t_agg t1 (actual time=1.382..32.683 rows=3496 loops=1)
      -> Hash (actual time=1.760..1.761 rows=4097 loops=1)
         Buckets: 131072 Batches: 1 Memory Usage: 1185kB
        -> Redistribute Motion 3:3 (slice3; segments: 3) (actual time=0.049..0.922 rows=4097
 loops=1)
           Hash Key: t2.x
          -> Seq Scan on t_agg t2 (actual time=1.628..32.837 rows=3496 loops=1)
           -> Hash (actual time=36.153..36.153 rows=4097 loops=1)
              Buckets: 131072 Batches: 1 Memory Usage: 1185kB
             -> Redistribute Motion 3:3 (slice4; segments: 3) (actual time=3.918..35.169 rows=4097
 loops=1)
                Hash Key: t3.x
               -> Seq Scan on t_agg t3 (actual time=1.380..30.316 rows=3496 loops=1)

Planning Time: 8.810 ms
(slice0) Executor memory: 257K bytes.
(slice1) Executor memory: 2484K bytes avg x 3 workers, 2570K bytes max (seg0). Work_mem:
 1185K bytes max.
(slice2) Executor memory: 32840K bytes avg x 3 workers, 32841K bytes max (seg0).
(slice3) Executor memory: 32860K bytes avg x 3 workers, 32860K bytes max (seg0).
(slice4) Executor memory: 32860K bytes avg x 3 workers, 32860K bytes max (seg0).
Memory used: 128000kB
Optimizer: Postgres query optimizer
Execution Time: 274130.589 ms
(32 rows)
```

对于有聚集下推的场景，执行查询的计划信息如下：

```
=> EXPLAIN (ANALYZE, COSTS OFF) SELECT t1.x, sum(t2.y + t3.y), count(*) FROM t_agg t1 JOIN
 t_agg t2 ON t1.x = t2.x JOIN t_agg t3 ON t2.x = t3.x GROUP BY t1.x;

                QUERY PLAN
------------------------------------------------------------------------------
Gather Motion 3:1 (slice1; segments: 3) (actual time=835.755..836.406 rows=30 loops=1)
 -> Finalize GroupAggregate (actual time=834.227..835.432 rows=12 loops=1)
   Group Key: t1.x
  -> Sort (actual time=834.031..834.441 rows=4097 loops=1)
    Sort Key: t1.x
    Sort Method: quicksort Memory: 1266kB
   -> Redistribute Motion 3:3 (slice2; segments: 3) (actual time=812.139..830.706 rows=4097
 loops=1)
     Hash Key: t1.x
    -> Hash Join (actual time=810.536..828.097 rows=3496 loops=1)
      Hash Cond: (t1.x = t2.x)
      Extra Text: (seg0) Hash chain length 1.0 avg, 1 max, using 30 of 131072 buckets.
     -> Seq Scan on t_agg t1 (actual time=1.689..16.674 rows=3496 loops=1)
     -> Hash (actual time=808.497..808.498 rows=30 loops=1)
       Buckets: 131072 Batches: 1 Memory Usage: 1026kB
      -> Broadcast Motion 3:3 (slice3; segments: 3) (actual time=461.065..808.466 rows=30
 loops=1)
       -> Partial HashAggregate (actual time=810.026..810.033 rows=12 loops=1)
         Group Key: t2.x
         Peak Memory Usage: 0 kB
        -> Hash Join (actual time=28.070..331.181 rows=1398787 loops=1)
          Hash Cond: (t2.x = t3.x)
          Extra Text: (seg0) Hash chain length 341.4 avg, 342 max, using 12 of 262144 buckets.
         -> Redistribute Motion 3:3 (slice4; segments: 3) (actual time=0.040..1.270 rows=4097
 loops=1)
           Hash Key: t2.x
          -> Seq Scan on t_agg t2 (actual time=1.449..19.963 rows=3496 loops=1)
          -> Hash (actual time=27.834..27.835 rows=4097 loops=1)
            Buckets: 262144 Batches: 1 Memory Usage: 2209kB
           -> Redistribute Motion 3:3 (slice5; segments: 3) (actual time=3.836..27.025 rows=4097
 loops=1)
             Hash Key: t3.x
            -> Seq Scan on t_agg t3 (actual time=1.537..23.654 rows=3496 loops=1)

Planning Time: 14.425 ms
(slice0) Executor memory: 328K bytes.
(slice1) Executor memory: 408K bytes avg x 3 workers, 514K bytes max (seg0). Work_mem:
 450K bytes max.
(slice2) Executor memory: 33951K bytes avg x 3 workers, 33952K bytes max (seg0).
 Work_mem: 1026K bytes max.
(slice3) Executor memory: 2298K bytes avg x 3 workers, 2341K bytes max (seg0). Work_mem:
 2209K bytes max.
(slice4) Executor memory: 32860K bytes avg x 3 workers, 32860K bytes max (seg0).
(slice5) Executor memory: 32860K bytes avg x 3 workers, 32860K bytes max (seg0).
Memory used: 128000kB
Optimizer: Postgres query optimizer
Execution Time: 865.305 ms
(39 rows)
```

对比上述两个查询计划的结果表明：在不采用聚集下推的情况下，必须先完成所有连接操作，然后才能在连接后的数据集上执行聚集操作。相反，聚集下推能够将聚集操作下推至最优的连接层级，有效减少后续连接操作所需处理的数据量，从而显著提升性能。

在采用聚集下推的查询示例中，PieCloudDB 将聚集操作下推至表 t2 与 t3 连接完成之后、与 t1 连接之前的位置。与未采用聚集下推的示例相比，执行时间从 274130.589 毫秒减少至 865.305 毫秒，查询性能提升了三百多倍。这一结果表明，随着数据量的增加或涉及的表数量的增多，聚集下推带来的性能提升将更加显著。

8.2 数据裁减

在 OLAP 场景中，数据库通常需要处理大量数据的聚合、过滤和分析查询。因此，PieCloudDB 采用了数据裁减（Data Skipping）技术来加速对大规模数据集的查询。Data Skipping 的核心原理是在查询执行过程中跳过那些不符合查询条件的数据块或分区的扫描，有效减少 I/O 操作，从而加快查询执行速度。

在 OLAP 应用中，数据裁减技术的常见实现包括 Zone Map 索引和 BRIN（Block Range INdex）索引。这两种技术的工作原理相似，都是利用预先计算的统计信息来跳过不满足查询条件的数据块，以此加速查询执行。Zone Map 索引会存储每个数据块中选定列的统计信息（例如最小值和最大值）及其他聚合信息。在查询期间，数据库可以利用这些统计信息来裁减不必要访问的数据块，减少不必要的 I/O 操作，提升查询性能。

PieCloudDB 采用混合存储模型，即水平地对数据进行分组，并将它们的属性垂直划分为列。这种模型既保留了行存储的空间局部性优势，又能获得列式存储的高效处理和压缩性能，降低了数据重组的开销。

Data Skipping 设计基于混合存储模型，将每个数据块视为一组记录的列式存储数据。在数据导入时，系统会为每个文件统计对应数据块的所需列的统计信息。当用户执行查询时，系统会根据查询条件对应列的统计信息判断每个数据块是否满足条件，满足则访问，否则跳过。

PieCloudDB 的 Data Skipping 功能默认是开启的，由配置参数 *enable_bitmapscan* 控制。本文将通过具体示例展示 Data Skipping 功能的效果。

根据如下 SQL 创建两张示例表 dataskip 和 j_tbl，并导入测试数据。

```
CREATE TABLE dataskip (a INT, b INT);
INSERT INTO dataskip SELECT i, i*2 FROM generate_series(1, 1000)i;
INSERT INTO dataskip SELECT i, i*2 FROM generate_series(1001, 2000)i;
INSERT INTO dataskip SELECT i, i*2 FROM generate_series(2001, 3000)i;
INSERT INTO dataskip SELECT i, i*2 FROM generate_series(3001, 4000)i;
INSERT INTO dataskip SELECT i, i*2 FROM generate_series(4001, 5000)i;
INSERT INTO dataskip SELECT i, i*2 FROM generate_series(5001, 6000)i;
INSERT INTO dataskip SELECT i, i*2 FROM generate_series(6001, 7000)i;
INSERT INTO dataskip SELECT i, i*2 FROM generate_series(7001, 8000)i;
INSERT INTO dataskip SELECT i, i*2 FROM generate_series(8001, 9000)i;
INSERT INTO dataskip SELECT i, i*2 FROM generate_series(9001, 10000)i;

CREATE TABLE j_tbl (a INT, b INT);
INSERT INTO j_tbl SELECT i, i * random() FROM generate_series(1, 1000) i;
```

当开启 Data Skipping 功能时，执行查询的计划信息如下：

```
=> EXPLAIN ANALYZE SELECT * FROM dataskip WHERE a < 10;

                QUERY PLAN
-------------------------------------------------------------------------------------------
Gather Motion 3:1 (slice1; segments: 3) (cost=2.00..10.21 rows=3 width=8) (actual
 time=34.361..36.928 rows=9 loops=1)
 -> Bitmap Heap Scan on dataskip (cost=2.00..10.17 rows=1 width=8) (actual
 time=16.189..31.790 rows=5 loops=1)
    Recheck Cond: (a < 10)
    Rows Removed by Index Recheck: 316
  -> Bitmap Index Scan on dataskip (cost=0.00..2.00 rows=333 width=0) (actual
 time=2.908..2.910 rows=1 loops=1)
     Index Cond: (a < 10)
Planning Time: 4.259 ms
(slice0) Executor memory: 159K bytes.
(slice1) Executor memory: 32972K bytes avg x 3 workers, 32972K bytes max (seg0).
Memory used: 128000kB
Optimizer: Postgres query optimizer
Execution Time: 55.895 ms
(12 rows)
```

如果关闭 Data Skipping 功能（即参数 enable_bitmapscan 的值设置为“OFF”），执行查询的计划信息如下：

```
=> SET enable_bitmapscan = OFF;
=> EXPLAIN ANALYZE SELECT * FROM dataskip WHERE a < 10;

                QUERY PLAN
-------------------------------------------------------------------------------------------
Gather Motion 3:1 (slice1; segments: 3) (cost=0.00..51.71 rows=3 width=8) (actual
 time=129.916..140.925 rows=9 loops=1)
 -> Seq Scan on dataskip (cost=0.00..51.67 rows=1 width=8) (actual time=2.939..132.546
 rows=5 loops=1)
    Filter: (a < 10)
    Rows Removed by Filter: 3292
Planning Time: 0.099 ms
(slice0) Executor memory: 123K bytes.
(slice1) Executor memory: 32825K bytes avg x 3 workers, 32825K bytes max (seg0).
Memory used: 128000kB
Optimizer: Postgres query optimizer
Execution Time: 154.416 ms
(10 rows)
```

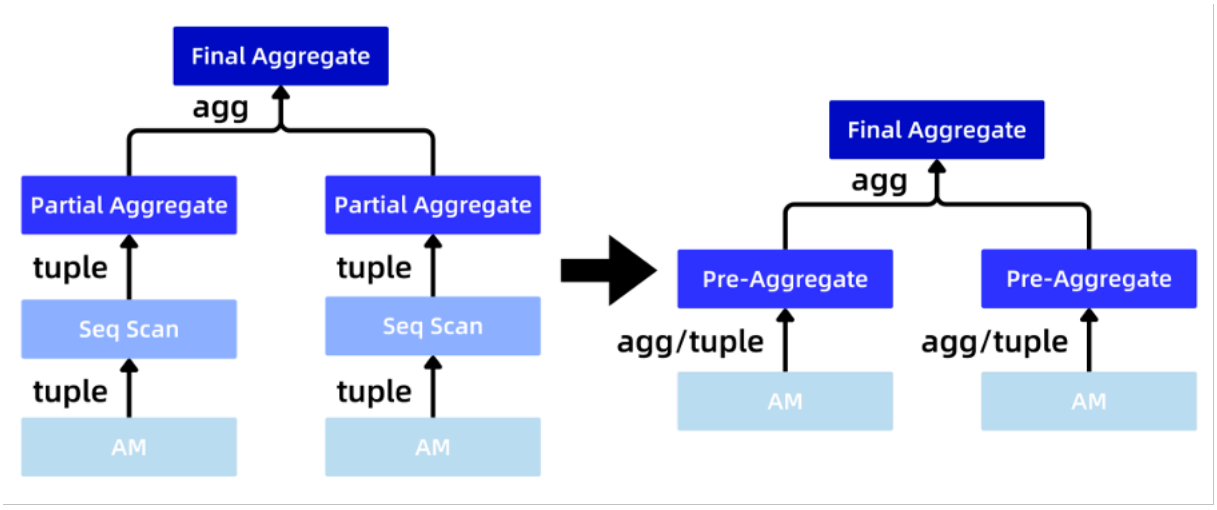
对比上述两个查询计划的结果表明：关闭 Data Skipping 功能时的执行时间大约是开启该功能时的三倍。

8.3 预聚集

预聚集（Pre-Aggregate）功能，依托于全新的存储引擎简墨（JANM），实现了一种高效的数据处理方式。该技术的主要工作原理是在数据插入的同时计算数据列的聚合（Aggregate）信息，并将其预先保存，以供后续查询使用。由于聚合数据直接保存在文件中，可以实现快速访问并直接应用于查询，从而摒弃了传统方式中在查询时所需的复杂计算，显著提升了查询速度。

PieCloudDB 能够为用户查询自动生成包含预聚集的执行计划。当聚合数据需要时，系统会检查预存储的聚合值，并直接读取符合条件的聚合数据，避免了扫描整个数据集的过程，大幅提升了查询速度。对于只满足部分条件的数据块，PieCloudDB 将采用传统方法计算聚合值，这样既利用了预聚合的数据，又只需计算缺失的部分，降低了计算成本并提高了运算效率。

PieCloudDB 存储引擎在数据插入时会即时更新聚合信息。为了能够正确获取下推的聚合数据，优化器与执行器增加了两个新的预聚集计算节点。下图展示了优化前后的计划树（Plan Tree）结构变化。



经过优化后，上图右侧的 Pre-Aggregate 计算节点负责从访问方法（AM）中提取预先计算的聚合（Aggregate）数据。如果未找到合适的聚合数据，Pre-Aggregate 计算节点将从 AM 中检索满足条件的元组（Tuple）并计算相应的聚合数据，以便将其传递给上层计算节点使用。

总体而言，Pre-Aggregate 是 OLAP 优化技术中 Zone Maps 的具体实现，它涉及预先计算并保存一批元组属性值的聚合信息。PieCloudDB 会检查这些预计算的聚合信息，以决定是否需要访问特定的数据块（Block）。如果找到可用的聚合数据，则直接返回这些数据；如果没有找到，则继续访问数据块以检索具体的元组。

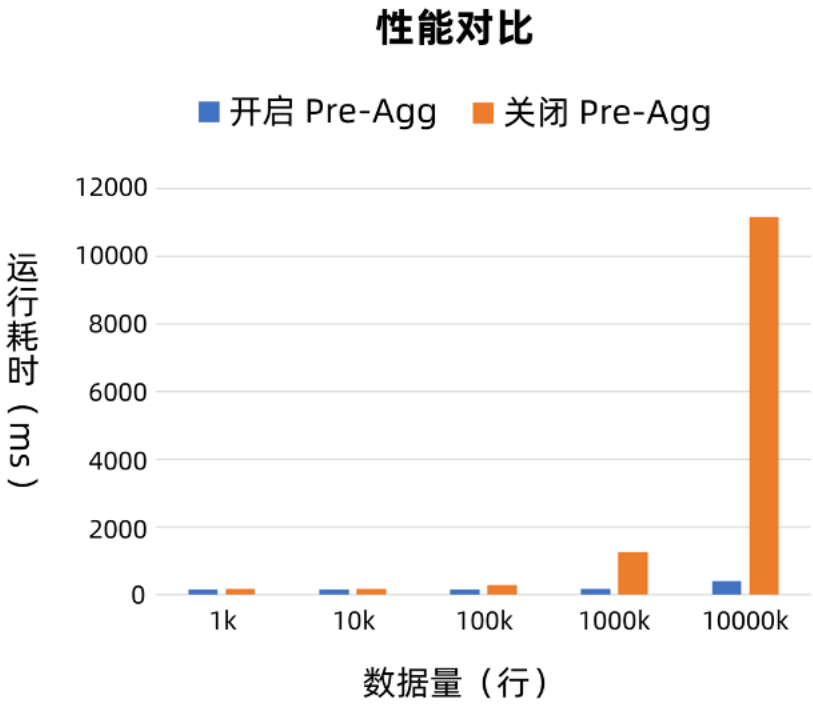
对于有条件的预聚集操作，其效果依赖于预先计算所涵盖的数据范围。PieCloudDB 将预聚集的范围限定在块文件级别，为每个块文件独立进行预计算和存储，确保了有条件的预聚集查询的效率。

本文通过创建测试表 preaggdata 并分别插入不同量级（1k、10k、100k、1000k 和 10000k）的数据，来展示预聚集功能的性能表现。

```
-- 测试表
CREATE TABLE preaggdata (a INT, b INT);

--测试语句
EXPLAIN ANALYZE SELECT SUM(a), AVG(a), COUNT(*) ， MAX(b) FROM preaggdata;
```

由测试结果得到的耗时对比图如下。



测试结果表明：在未启用预聚集功能的情况下，随着数据量的增加，查询耗时持续增长，且增长速度逐渐加快。相反，当启用预聚集功能时，耗时呈现出稳定的增长趋势，且增长速度较慢。特别是在数据量达到10000K时，预聚集功能实现了近 28 倍的速度提升。

PieCloudDB 的预聚集功能包括两种方式：Preagg Block Scan 和支持 Block Skipping 的 Preagg Bitmap Block Scan。在条件过滤较多或自定义类型较多的情况下，建议使用 Preagg Bitmap Block Scan。PieCloudDB 的优化器会根据查询场景自动选择最合适的方式。同时，用户也可以手动启用并使用任意一种预聚集方式，具体方法如下：

- Preagg Block Scan 的启用方式：

  Preagg Block Scan 是默认启用的，由配置参数 *pdb_enable_preagg* 控制。手动启用的命令如下：

  ```
  SET pdb_enable_preagg = ON;
  ```

  关闭 Preagg Block Scan 的命令为 *SET pdb_enable_preagg = OFF;*
- Preagg Bitmap Block Scan 的启用方式：

  1. 启用预聚集功能。

     ```
     SET pdb_enable_preagg = ON;
     ```

  2. 启用 Preagg Bitmap Block Scan 功能，参考命令如下。

     ```
     SET enable_seqscan = OFF;
     SET enable_bitmapscan = ON;
     SET enable_indexscan = ON;
     ```

9 导入与导出数据

9.1 导入数据

PieCloudDB 支持使用 PostgreSQL 的交互式终端“psql”执行 *COPY* 命令将文件或标准输入中的数据导入到表中。

COPY 命令是并行执行的，即协调器进程对文本文件进行切片，同时分发到不同的执行器上完成数据的解析和加载。

PieCloudDB 与客户端可以在命令行环境中使用 STDIN 和 STDOUT 传递数据。通过“psql”连接时，可以使用 *\copy* 来指定客户端主机上的文件路径。

数据格式

COPY FROM 的输入格式取决于 *FORMAT* 参数，*FORMAT* 参数的有效值为 text、csv（使用逗号分隔值）和 binary，默认为 text。此外，*DELIMITER* 选项将不同的字符指定为值分隔符。语法如下：

```
COPY table_name FROM 'path_to_filename' WITH (FORMAT format_name [, DELIMITER
 'delimiter_character'] );
```

csv 格式的文件使用逗号分隔值。默认情况下，text 格式使用制表符作为分隔值，也可以设置 DELIMITER 选项将不同的字符指定为值分隔符。

如下示例为，将“|”作为 text 格式文件的值分隔符。

```
COPY table_name FROM 'path_to_filename' WITH (FORMAT text, DELIMITER '|');
```

输入数据根据 *ENCODING* 选项或当前客户端编码进行解析，输出数据以 *ENCODING* 或当前客户端编码进行编码。

默认情况下，PieCloudDB 使用默认客户端编码，用户可以使用 *ENCODING* 选项更改此编码。

如下示例为，更改文件的编码方式为 latin1。

```
COPY table_name FROM 'path_to_filename' WITH (ENCODING 'latin1');
```

从主机文件加载数据

使用 COPY 命令从主机上的文件导入数据时，PieCloudDB 服务器进程需要打开指定文件后再读取文件，并将内容加载到目标表中，由此用户需要具备目标表的 INSERT 权限和读取文件的权限。

COPY FROM 命令必须使用主机上的绝对路径或数据目录的相对路径来指定文件位置。语法如下：

```
COPY table_name FROM 'path_to_filename';
```

如下示例为，将主机文件中的数据复制到表 country 中。

```
COPY country FROM '/usr1/sql/country_data';
```

从 STDIN 加载数据

STDIN 使用标准输入通道将数据直接提供给服务器进程。当执行 *COPY FROM STDIN* 命令时，数据库服务器将开始从标准输入接收数据。数据的传输将持续进行，直到服务器读取到一个特殊的结束行，该行仅由反斜杠（\）和句点（.）组成。

COPY FROM STDIN 命令的语法如下：

```
COPY table_name FROM STDIN;
```

如下示例为，通过 STDIN 将数据导入到表 t 中。需要注意的是，每行上的列的分隔符是制表符。

```
=> CREATE TABLE t (id INT, name TEXT);
=> COPY t from STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself, or an EOF signal.
>> 1 John
>> 2  Jane
>> 3  Alice
>> \.
COPY 3
```

从客户端文件加载数据

与作为 SQL 命令的 COPY 不同，*\copy* 加载数据会调用 *COPY FROM STDIN* 将数据从 psql 客户端发送到服务器后台进程。因此，所有文件都必须驻留在运行 psql 客户端的主机上，并且允许运行客户端的用户访问。

当执行 *COPY FROM STDIN* 命令时，数据库服务器将开始从标准输入接收数据。数据的传输将持续进行，直到服务器读取到一个特殊的结束行，该行仅由反斜杠（\）和句点（.）组成。psql 将这些能力包裹到 *\copy* 命令中，语法如下：

```
\copy table_name FROM 'path_to_filename';
```

如下示例为，使用命令将客户端文件 country_data.text 中的数据复制到表 country 中。

```
\copy country FROM '/workspace/country_data';
```

9.2 使用 COPY TO 导出数据

PieCloudDB 主要使用 PostgreSQL 的交互式终端“psql”执行 COPY 命令将数据导出到文件。

`COPY TO` 命令可以将表的内容复制到文件，使用该命令需要用户具备表的 SELECT 权限。语法如下：

```
COPY table_name TO 'path_to_filename';
```

`COPY TO` 命令不仅可以复制表中的数据，还可以复制 SELECT 查询的结果。如果指定了列，则 `COPY TO` 仅将这些指定列的数据复制到文件中。但是，`COPY TO` 命令只能用于普通表，不能用于视图，并且不会从子表或子分区复制行。

此外，`COPY TO` 命令还支持指定文件格式、字段分隔符、空值表示方式等选项。用户可以通过在命令后使用 WITH 子句并附上相关选项来指定这些参数。

如下示例为，将表 t 中的数据以文本 text 格式导出，字段之间使用“|”作为分隔符，空值将表示为“N/A”。

```
COPY t TO '/path/to/output.txt' WITH (FORMAT text, DELIMITER '|', NULL 'N/A');
```

`COPY TO STDOUT` 命令为表中的数据提供了标准输出通道。语法如下：

```
COPY table_name TO STDOUT;
```

如下示例为，通过标准输入（STDIN）导出表 t 中的数据，并将数据以文本格式显示在终端上。

```
=> COPY t TO STDOUT;
1 John
2  Jane
3  Alice
COPY 3
```

如果需要将表中的数据导出到客户端本地，应使用 `\copy` 命令。`\copy` 会调用 `COPY TO STDIN`，语法如下：

```
\copy table_name TO 'path_to_filename';
```

如下示例为，使用 `\copy` 命令将表 t 中的列 name 的数据导出到客户端文件 data 中。

```
\copy (SELECT name FROM t) TO '/workspace/data' WITH (FORMAT csv);
```

9.3 格式化数据文件

在导入和导出数据时，用户必须指定数据的格式。COPY 命令允许用户指定其数据格式可以是文本（text）格式或者逗号分隔值（CSV）格式。为了确保 PieCloudDB 能够正确读取，外部数据必须经过适当的格式化。

格式化行

在文本文件中，换行符（Line Feed，通常表示为 LF）用于分隔不同的行，确保文本能够以适当的格式显示在屏幕上或打印输出中。

换行符的使用方式因操作系统和应用程序的不同而有所差异。PieCloudDB 能够识别数据中的行分隔符，这些分隔符可以是 LF（换行，\n 或 0x0A）、CR（回车，\r 或 0x0D）或者 CRLF（回车加换行，\r\n 或 0x0D 0x0A）。在 UNIX 或类 UNIX 操作系统中，LF 是新行的标准表示方式。而 Windows 等操作系统则通常使用 CR 或者 CR +LF。PieCloudDB 支持所有这些新行表示形式作为行定界符。

格式化列

制表符通常用于创建表格、对齐文本和代码中的列，或在文档中创建缩进。在文本（text）文件中，默认的列定界符是水平制表符（TAB，0x09），而在 CSV 文件中，默认的列定界符是逗号（0x2C）。

在定义数据格式时，可以使用 *COPY* 或 *CREATE EXTERNAL TABLE* 命令的 *DELIMITER* 选项来设置一个单字符的列定界符。定界符字符必须位于任意两个数据值之间，且不应放置在行的首尾。

如下示例使用竖线字符（|）作为定界符。

```
data value 1|data value 2|data value 3
```

如下示例为，*CREATE EXTERNAL TABLE* 命令使用竖线字符（|）作为列定界符。

```
CREATE EXTERNAL TABLE ext_table (name text, date date)
   LOCATION ('gpfdist://<hostname>/filename.txt)
     FORMAT 'TEXT' (DELIMITER '|');
```

标识 NULL 值

NULL 用于标识列中或域中的未知数据，即空值。在数据文件中，用户可以指定一个字符串来表示空值。对于文本（text）文件，默认的空值表示字符串是“\N”（反斜线后跟字母 N），而在 CSV 文件中，空值通常表示为一个不带引号的空字符串。

在定义数据格式时，*COPY* 命令允许使用 *NULL* 选项来设置一个字符串以表示空值。如下示例展示了如何使用 *COPY* 命令将“N/A”指定为表示空值的字符串。

```
COPY t TO '/path/to/output.txt' WITH (FORMAT text, DELIMITER '|', NULL 'N/A');
```

字符转义

PieCloudDB 保留如下两类特殊字符：

- 列分隔符：在数据文件中用于分隔列或域的字符，例如竖线（|）和逗号（,）。
- 行分隔符：在数据文件中用于标识新行的换行符，例如“\n”。

如果用户的数据中包含这些特殊字符，必须对它们进行转义处理，以便 PieCloudDB 将它们识别为数据内容，而非分隔符或换行符。

在默认设置中，text 格式文件使用反斜线（\）作为转义字符，而 csv 格式文件使用双引号（"）作为转义字符。

在文本格式文件中转义

用户可以在 *COPY* 命令中使用 *ESCAPE* 选项来指定一个不同的转义字符。这样，如果转义字符本身出现在用户的数据中，它将被自身转义，从而允许 PieCloudDB 正确识别和处理数据。

例如，用户想将如下数据导入到有一个具有三列的表中：

- A backslash = \
- A vertical bar = |
- An exclamation point = !

如果用户选择竖线字符（|）作为分隔符，并将反斜线（\）设置为转义字符，那么用户的数据文件中，已格式化的行应该如下所示：

```
A backslash = \\ | A vertical bar = \| | An exclamation point = !
```

用户可以使用转义字符来转义十进制和十六进制序列。在将数据导入 PieCloudDB 时，这些被转义的值会被转换成相应的字符。例如，如果要表示花号字符（&），用户可以使用转义字符将其表示为等效的十六进制形式（\0x26）或十进制形式（\046）。

此外，用户还可以通过 *COPY* 命令的 *ESCAPE* 选项来禁用文本文件中的转义功能。这种设置通常用于处理包含大量反斜线字符的输入数据，例如 Web 日志数据。

```
ESCAPE 'OFF'
```

在 CSV 格式的文件中转义

默认情况下，CSV 格式的文件使用双引号（"）作为转义字符。用户可以通过 *COPY* 命令的 *ESCAPE* 选项来指定一个不同的转义字符。如果转义字符本身出现在用户的数据中，它将使用自身来进行转义。

例如，用户想将如下数据导入到有一个具有三列的表中：

- Free trip to A,C
- 3.69
- Special rate "2.75"

如果用户选择逗号（,）作为分隔字符，并将双引号（"）设置为转义字符，那么用户的数据文件中，已格式化的行应该如下所示：

```
"Free trip to A,C","3.69","Special rate ""2.75"""
```

用户也可以通过在一组双引号内嵌入整个字段，以确保字段前后的空白字符被保留。示例如下：

```
" Free trip to A,B ","5.89 ","Special rate ""1.79"" "
```

注意：

在 CSV 格式文件中，所有字符都具有特定含义。如果用户需要导入的数据来源于一个使用固定宽度填充空白字符的系统，那么在导入之前，用户应预处理 CSV 文件，即移除每行末尾的空白字符。

9.4 字符编码

PieCloudDB 支持多种字符集，包括 ISO 8859 系列等单字节字符集以及 EUC、UTF-8、Mule 内部编码等多字节字符集。

服务器端字符集是在数据库初始化时定义的，其中 UTF-8 是默认字符集，但可以根据需要进行更改。客户端可以透明地使用所有支持的字符集，但有些字符集不支持作为服务器端的编码。

在数据导入过程中，PieCloudDB 会将数据从客户端指定的编码转换为服务器端的编码。同样，在数据导出到客户端时，PieCloudDB 也会将数据从服务器端的字符编码转换为客户端指定的编码。

数据文件必须使用 PieCloudDB 能够识别的字符编码。如果数据文件包含非法或不支持的编码序列，加载时可能会出现错误。

特别需要注意的是，对于在微软 Windows 操作系统下生成的数据文件，在导入到 PieCloudDB 之前，通常需要运行 dos2unix 命令来移除 Windows 特有的字符。

客户端的字符编码可以通过服务器配置参数 *client_encoding* 进行设置。示例如下：

```
SET client_encoding TO 'latin1';
```

RESET 命令用于将客户端字符编码更改回默认值，参考命令如下：

```
RESET client_encoding;
```

SHOW 命令用于显示当前客户端字符编码设置，参考命令如下：

```
SHOW client_encoding;
```

10 PieCloudDB 内核版本发布记录

10.1 PieCloudDB 内核 V2.17.x

PieCloudDB 内核 V2.17.2

版本信息

- 发布时间：2024 年 09 月 27 日
- 版本号：V2.17.2

功能增强

PieCloudDB 内核 V2.17.2 支持 oracle_fdw 插件。

升级说明

支持 PieCloudDB 内核 V2.x.x 小版本升级到 V2.17.2。

PieCloudDB 内核 V2.17.1

版本信息

- 发布时间：2024 年 09 月 24 日
- 版本号：V2.17.1

缺陷修复

- 修复可能导致 postmaster 进程启动失败的未初始化内存问题。
- 修复在 gmemos 未正确安装时可能导致后台进程崩溃的问题。

升级说明

支持 PieCloudDB 内核 V2.x.x 小版本升级到 V2.17.1。

PieCloudDB 内核 V2.17.0

版本信息

- 发布时间：2024 年 09 月 09 日
- 版本号：V2.17.0

缺陷修复

修复多个 Golang 第三方库的安全漏洞。

升级说明

支持 PieCloudDB 内核 V2.x.x 小版本升级到 V2.17.0。

10.2 PieCloudDB 内核 V2.16.x

PieCloudDB 内核 V2.16.1

版本信息

- 发布时间：2024 年 08 月 28 日
- 版本号：V2.16.1

版本定位

修复脱敏插件可能导致建表失败的问题。

升级说明

支持 PieCloudDB 内核 V2.x 小版本升级到 V2.16.1。

PieCloudDB 内核 V2.16.0

版本信息

- 发布时间：2024 年 08 月 12 日
- 版本号：V2.16.0

版本定位

PieCloudDB 内核 V2.16.0 主要适配 PieCloudDB 社区版的相关功能并支持社区版的发布。关于社区版的文档内容，请参见 PieCloudDB 社区版管理员指南。

缺陷修复

- 修复偶发的元数据访问内存泄露问题
- 修复 COPY 未触发执行器节点 constraint 检查的问题
- 修复偶发的 Socket 文件描述符（fd）泄露问题
- 修复通过 WebSQL 使用 PG FDW 时报错“portal not found”

升级说明

支持 PieCloudDB 内核 V2.x 小版本升级到 V2.16.0。

10.3 PieCloudDB 内核 V2.15.x

PieCloudDB 内核 V2.15.3

版本信息

- 发布时间：2024 年 07 月 23 日
- 版本号：V2.15.3

版本定位

PieCloudDB 内核 V2.15.3 主要开放了用户对 pg_cancel_backend 函数的可执行权限。

升级说明

支持 PieCloudDB 内核 V2.x 小版本升级到 V2.15.3。

PieCloudDB 内核 V2.15.2

版本信息

- 发布时间：2024 年 07 月 23 日
- 版本号：V2.15.2

版本定位

PieCloudDB 内核 V2.15.2 支持 mysql_fdw 和 postgresql_fdw 插件。

升级说明

支持 PieCloudDB 内核 V2.x 小版本升级到 V2.15.2。

PieCloudDB 内核 V2.15.0

版本信息

- 发布时间：2024 年 07 月 15 日
- 版本号：V2.15.0

版本定位

PieCloudDB 内核 V2.15.0 主要开放了 pg_terminate_backend() 函数和 PUBLIC Schema 的相关操作权限。

功能增强

- 开放普通用户对 pg_terminate_backend() 函数具有默认的可执行权限，用户可以 kill 其具有权限的进程。
- 允许对象拥有者和超级用户 SUPERUSER 对 PUBLIC Schema 执行 REVOKE 和 GRANT 操作。

缺陷修复

- 修复一些 CVE 漏洞以增强系统和数据的安全性。
- 修复偶发的初始化对象存储时崩溃的问题。

升级说明

支持 PieCloudDB 内核 V2.x 小版本升级到 V2.15.0。

10.4 PieCloudDB 内核 V2.14.0

版本信息

- 发布时间：2024 年 06 月 06 日
- 版本号：V2.14.0

版本定位

PieCloudDB 内核 V2.14.0 支持了动态配置查询簇的功能。

功能增强

PieCloudDB 支持通过 pdb_parallel_factor 参数动态配置查询簇的大小，从而提升单个查询并行使用底层资源的能力，并加速查询响应时间。详细信息请参见 动态配置查询簇。

缺陷修复

- 修复在极端条件下文件句柄可能泄露的问题。

升级说明

支持 PieCloudDB 内核 V2.x 小版本升级到 V2.14.0。

10.5 PieCloudDB 内核 V2.13.0

版本信息

- 发布时间：2024 年 03 月 29 日
- 版本号：V2.13.0

版本定位

PieCloudDB 内核 V2.13.0 支持了本地缓存重用的功能，并实现了 Interconnect proxy 多进程执行的能力。

功能增强

- 本地缓存重用功能增强了本地缓存文件的生命周期管理，执行器节点重启之后可以继续使用之前缓存在本地的数据文件，节省了重新从远端下载数据文件的带宽资源，提升查询的首次响应时间。详细信息请参见 LocalCache。
- PieCloudDB 可以为执行器节点配置多个 Interconnect proxy 代理进程。多个代理进程可以提升 Interconnect 处理跨节点之间的网络处理能力，解决部分查询带来的 Interconnect 流量过大导致 proxy 进程成为瓶颈的问题。详细信息请参见 多进程 ic-proxy。

缺陷修复

- 修复多个稳定性问题。

升级说明

支持 PieCloudDB 内核 V2.x 小版本升级到 V2.13.0。

10.6 PieCloudDB 内核 V2.12.0

版本信息

- 发布时间：2024 年 02 月 05 日
- 版本号：V2.12.0

版本定位

PieCloudDB 内核 V2.12.0 实现了元数据 MetaCache 缓存服务和本地缓存 LocalCache 的阶段性功能。

功能增强

- 元数据缓存服务支持缓存进程信息，大幅度提升了获取快照操作的性能，从而加快查询响应速度。
- 本地缓存清理策略的粒度更加精细化，新增缓存命中率展示函数 pdb_get_local_cache_hit_rate()，参见 LocalCache。

缺陷修复

- 修复并发 Compaction 可能导致异常报错。

升级说明

支持 PieCloudDB 内核 V2.x 小版本升级到 V2.12.0。

10.7 PieCloudDB 内核 V2.11.0

版本信息

- 发布时间：2024 年 01 月 12 日
- 版本号：V2.11.0

版本定位

PieCloudDB 内核 V2.11.0 主要修复了一些功能缺陷，无新增功能。

缺陷修复

- 修复删除大量对象时可能出现的内存泄露问题。
- 修复 autovacuum 工作进程可能导致创建 Database 失败的问题。

升级说明

支持 PieCloudDB 内核 V2.x 小版本升级到 V2.11.0。

10.8 PieCloudDB 内核 V2.10.0

版本信息

- 发布时间：2023 年 12 月 01 日

- 版本号：V2.10.0

版本定位

PieCloudDB 内核 V2.10.0 主要新增本地块存储缓存功能开关、支持元数据缓存数据的远端存储和访问外部 Iceburg 数据等功能。

功能增强

- 增加配置参数 pdb_janm_enable_disk_cache，用于控制本地块存储缓存功能的开启（ON）和关闭（OFF）。
- 支持元数据的缓存数据上传远端存储的功能。
- 支持通过 mpp_sql FDW 访问 Iceburg 数据。

缺陷修复

- 修复偶然发生的锁信息残留问题，此问题可能导致用户查询死锁或后台 autovacuum 进程挂起。

升级说明

支持 PieCloudDB 内核 V2.x 小版本升级到 V2.10.0。

10.9 PieCloudDB 内核 V2.9.0

版本信息

- 发布时间：2023 年 11 月 10 日
- 版本号：V2.9.0

版本定位

PieCloudDB 内核 V2.9.0 版本增强了元数据的跟踪日志和元数据缓存功能、支持 Preagg Block Skipping 预聚集特性和 *VACUUM COMPACT* 命令等。

功能增强

- 增加新的元数据缓存服务，可以降低元数据存储节点负载，提高查询响应速度。
- 增加更多的元数据跟踪日志（Trace Point）。
- 支持 Preagg Block Skipping，可以通过 JAMN 虚拟索引直接获取指定文件块的预聚集计算结果，加速聚集计算响应时间。
- 优化全局缓存系统的本地缓存上限控制能力。
- 支持 *VACUUM COMPACT* 命令将指定表的小数据文件合并为指定大小的数据文件，以降低小文件数量，从而有助于提升查询响应时间和降低远端存储使用开销。
- Interconnect 协议缺省使用 proxy（之前版本缺省为 udpifc），提升性能和稳定性。

升级说明

对于使用 PieCloudDB 内核 V2.9.0 之前版本进行初始化的租户，需要参照文档进行升级（https://www.tapd.cn/47647475/prong/stories/view/1147647475001002741）。

10.10 PieCloudDB 内核 V2.8.0 及以下版本

PieCloudDB 内核 V2.8.0 及以下版本所发布的主要功能汇总如下：

- 回收站功能

  实现回收站来跟踪逻辑上被删除（TRUNCATE/DROP/ALTER 等）的表，这些表对应的 S3 Block 文件在超过保留期限后会被 autovacuum 删除。
- PieCloudDB 全局缓存系统
  - 支持在计算节点为一个查询缓存所需的目录（catalog）信息，在每个虚拟数仓中，只有一个目录信息副本从协调节点分发，并共享给所有 Slices/Backends。
  - 支持在 PieCloudDB 全局缓存系统中添加本地缓存。
- HLL（HyperLogLog） 压缩

  当 HLL 比较稀疏时支持使用游程编码，可以节省 60%~95% 的存储空间。
- 优化 Block Skipping

  实现 JANM 的虚拟索引。对于某些特殊的访问方法，表的数据文件/块已经包含可以用作索引的信息，通过虚拟索引不仅可以利用这些信息来提高性能，而且能减少维护开销。
- 简墨 JAMN 相关优化
  - 通过 Delte Encoding 优化 JAMN 文件存储空间。
  - 支持指定 JAMN 文件数据压缩方法，包括 None、PGLZ 或者 ZSTD。使用ZSTD压缩方法可以大幅度提高数据文件压缩率，降低数据文件存储成本。
  - 支持预聚集块扫描节点，对 JAMN 文件块中的数据进行预聚集计算。
  - 增强 JANM Data Skipping 对 IN 条件的处理能力。
  - 新增 GUC 参数 pdb_enable_janm_toast，控制 JANM 访问方法中的 Toast。
  - 新增建表选项 *page_size*，设置生成新 Page 的大小限制，以兆字节为单位。
  - 新增建表选项 *block_file_size*，设置生成新文件的 Block 文件大小限制，以兆字节为单位。
  - 新增建表选项 *num_stat_col*，控制文件内统计列数，为每个 JANM 文件节省 CPU 和元数据大小。默认值为 32，与 INDEX_MAX_KEYS 相同。
- 外表 FDW 相关
  - 新增 raw_fdw 外表接口，支持在协调节点和工作节点上执行，以原始字节（bytea）读取文件。
  - 为外表启用部分聚合能力，支持并行聚集下推。
  - 新增 fdb_max_retry（替代原 MaxRetry 功能），默认值设置为 50。
- 临时表相关
  - 新增 GUC 参数 pdb_temptable_local_limit_filesize 和 pdb_temptable_local_limit_filenum，用于控制临时表的数据文件溢出到远端存储。
  - 支持 autovacuum 清理孤立的临时表。
- pgaudit 插件

支持使用 pgaudit 插件生成审计日志。

- Kafka 接口
  - 支持获取 Kafka FDW 统计数据。
  - Kafka FDW 支持 JSON 格式。
- 执行计划信息

  支持在 EXPLAIN ANALYZE VERBOSE 执行结果中显示调度相关状态信息。
- 用户定义函数 UDF
  - 新增 janm_page_size_read()，用以获取 Page 大小。
  - 新增 janm_block_file_size_read()，用以获取 Block 文件大小。
  - 新增 pdb_database_sizes()，在 pdb_database_tools() 扩展中用于收集所有数据库大小。

11 术语表

Analyze

查询优化器收集数据仓库中表的内容统计信息，并将这些信息存储在系统表中，以便使用这些统计信息来确定查询的最有效执行计划。

存储引擎 | Storage Engine

在数据库管理系统（DBMS）中，负责处理数据存储和检索的组件或模块定义了数据在物理存储介质上的组织方式、读取方法和写入过程。PieCloudDB 计算引擎的数据存储底座称为简墨（JANM），更详细信息请参见相关词条 **简墨**。

查询规划器 | Query Planner

查询规划器是数据库系统的重要组成部分。在接收到查询请求后，它会考虑数据库结构、索引以及当前系统负载等因素，通过以下四个步骤来生成最优的查询策略：解析查询、分析优化、生成执行计划、以及优化执行计划，从而提高查询性能并高效地返回结果。

查询优化器 | Query Optimizer

查询优化器有多种含义。在某些情况下，它指的是对已有查询计划进行进一步优化的过程。PieCloudDB 达奇查询优化器（Query Optimizer），也被称作查询规划器（Query Planner），负责从接收查询任务开始，到优化并执行查询计划的整个流程。

大规模并行计算 | MPP

大规模并行处理（MPP，Massive Parallel Processing）是一种计算框架，它通过多个处理器协调处理一个程序来实现。这是实现数据库架构的多种结构之一。在 MPP 框架中，每个计算节点都拥有自己的操作系统、处理器和内存，并分别处理程序的不同部分。这样的框架需要考虑如何在各个节点之间划分共享的数据库，以及如何在节点间合理分配计算任务。

ETL

Extract-Transform-Load，简称 ETL，描述了一套数据处理过程：数据从源端经过抽取（Extract）、转换（Transform）、加载（Load）至目标端。

服务无感知 | Serverless

数据仓库的“Serverless”化允许云虚拟数仓的后台自动化地处理所有底层资源的调度，使用户在使用数仓服务时无需关注服务器架构或技术复杂性，从而提供了无缝且流畅的用户体验。

高可用 | High Availability

高可用（HA）指的是系统或服务具备高度可靠的持续运行能力，即使在服务故障发生时也能保持可用性，从而确保业务的正常运行。

简墨 | JANM

PieCloudDB 拥有高效的数据缓存结构和全新的存储引擎“简墨”。“简墨”主要为对象存储而设计，采用对象存储作为持久化存储层。针对 eMPP 架构下数据的分布特性和

弹性，简墨利用一致性 Hash 算法确保分布式系统中每个计算节点都能访问到一致的数据。此外，即使在进行扩缩容操作时，简墨也能最大限度地减少数据缓存的迁移。

全球数据圈 | Global Datasphere

数据圈（Datasphere）是国际数据公司（IDC, International Data Corporation）提出的概念，指的是每年被创建、采集或复制的数据集合。从另一个角度来看，它也代表了所有被存储在硬盘、SSD、光盘、磁带等存储介质上的数据的总和。

Schema

Schema 是构成 PieCloudDB 数据结构的逻辑组成部分，它包括一组表以及基于这些表生成的函数和视图的集合。

数据库 | Database

数据库是数据的逻辑组，通过 Schema 与表进行组织。它仅包含数据库对象，而不涉及任何计算资源。在 PieCloudDB 中，所有数据都保存并维护在数据库中。每个用户都可以创建多个数据库，且不同数据库之间相互隔离，以确保数据安全。

数据库对象 | Database Object

数据库对象是用于存储或引用数据的数据结构。表是其中一种常用的数据库对象，而其他数据库对象包括索引、存储过程、函数、序列、视图等。

透明数据加密 | TDE

透明数据加密（TDE, Transparent Data Encryption）利用加密密钥（DEK）对输入/输出数据进行实时加密和解密，整个过程对应用程序完全透明，应用程序无需感知。

弹性大规模并行计算 | eMPP

弹性大规模并行处理（eMPP, elastic Massive Parallel Processing）与 MPP 类似，它是一种由多个处理器协调处理单一程序的计算框架，也是实现数据库架构的组成结构之一。eMPP 利用云计算系统的特性，将数据库中的元数据、存储和计算进行分离；其中，元数据作为一个独立的系统，单独存储用户、数据库对象以及计算节点的信息。

在查询数据时，各个计算任务会调取同一份元数据，并以此为基础执行计算任务。当 eMPP 系统接收到计算任务时，协调器会通过查询规划器与优化器先对计算任务（例如某个查询任务）进行解析，并将其转换为一个由多个子查询组成的查询计划，然后通过协调器将任务分配给执行器。计算部分的节点组合可以根据查询的具体情况动态调整，以提高计算效率。

协调器

协调器在虚拟数仓中负责协调和管理，其主要任务包括响应客户端的 SQL 请求、生成查询计划、分发执行信息和元数据，以及汇总最终的执行结果。

虚拟数仓 | Virtual Warehouse

虚拟数仓代表纯粹的计算资源，它是由云上的虚拟机提供的包括 CPU、内存和临时存储在内的动态计算资源集群。虚拟数仓本身不存储真实数据，仅提供 SQL 执行服务。

在虚拟数仓中，至少包含一个协调器和一个执行器，每个虚拟数仓都是由 CPU、内存和磁盘资源组成的虚拟资源集合。用户无法直接获取这些虚拟数仓的信息，但有权限的用户可以观察到虚拟数仓的启动、停止和扩缩容过程。

云原生 | Cloud Native

云原生是指在设计、构建和运行时完全依托于云平台，并充分利用云计算模型优势的系统或应用。

元数据 | Metadata

PieCloudDB 的元数据独立于用户数据存在。它不仅包含用户数据的传统元数据（例如名称、大小、字段信息等），还包含用户数据统计信息、用户信息、虚拟数仓信息等。在 PieCloudDB 中，元数据常用于数据功能开发和性能优化，例如，查询优化器“达奇”便是利用统计信息等元数据来优化查询计划的。

元数据服务 | Catalog Service

PieCloudDB 提供管理和存储服务，用于收集、组织、存储和管理数据资产的元数据，帮助组织更好地理解、管理和利用其数据资产。在元数据层，PieCloudDB 提供独立的元数据服务，以确保元数据存取的效率和高可用性。

元组 | Tuple

在 PieCloudDB 中，一行或一条记录可以表示为一组有序的值，这组值称为元组。每个元组由多个字段或属性组成，每个字段都具有指定的数据类型。

执行器

每个虚拟数仓包含一个或多个执行器，负责执行协调器下发的任务，这些执行器能够并行处理同一个 SQL 查询任务。

执行计划 | Execution Plan

执行计划是数据库管理系统（DBMS）根据查询语句、表结构、索引统计信息等，在执行查询或操作时生成的。它利用优化器的算法和规则，为查询提供最高效的执行路径。


OpenPie | PieCloudDB
欢迎加入技术讨论群
扫码添加 PieCloudDB 小助手
联系我们：community@openpie.com