OpenPie | πCloudDB

PieCloudDB Community Edition Documentation

www.openpie.com

Contents

Legal Notice

This manual is only an informational introduction and help manual for the software products involved. This manual cannot be used as a basis for a comprehensive judgment on whether the software product has or does not have certain functions and/or technical parameters, nor can it be used to determine whether the software product meets certain requirements, technical tasks and/or parameters and other third-party specification documents.

Trademark Statement

Disclaimer

In any case, OpenPie shall not be liable for any possible errors and/or omissions in this document and any losses (direct or indirect losses, including unearned profits) caused by the purchaser of the product.

Note

Certain features and functions of the products and accessories described in this manual depend on the design and performance of the local network and the environment in which you install them; the products in this manual have been tested in detail, but cannot be guaranteed to be fully compatible with all software and hardware systems, and cannot be guaranteed to be completely error-free.

This manual is for reference only and does not constitute any form of commitment. Please refer to the actual presentation of the product-related content during use. OpenPie reserves the right to modify any information in this manual at any time without prior notice and without any responsibility.

Preface

This documentation outlines the process for installing, configuring, and using PieCloudDB Community Edition. It is designed to assist database administrators and other IT professionals in understanding PieCloudDB and cloud-native database technology.

The documentation is organized into several sections:

- Introduction to PieCloudDB Community Edition
- Overview of PieCloudDB
- Deploy PieCloudDB Community Edition
- Manage PieCloudDB
- Import and Export Data
- Glossary

Audience

This document serves as a reference guide for administrators and operators of the PieCloudDB database system, aiming to enhance their management and operational capabilities when working with PieCloudDB.

It is presumed that readers possess knowledge in Linux/Unix system administration, data management systems, database administration, and are familiar with the SQL language.

Conventions

The following conventions are used in this document:

Convention	Note
Dangers	Indicates that a serious failure or problem could arise and may result in a total system breakdown or unstable operation.
Warning	Indicates that a significant issues or anomalies may arise, potentially impacting the system's functionality or performance noticeably, but the system should remain operational.
Attention	Indicates that issues or anomalies may arise and are not expected to have a significant impact on normal system operations. However, such alerts require attention from users.
Tip	Provide additional information, guidance, or supplementary explanations as needed.
>	Hierarchical menu path.
Boldface	User Interface (UI) elements like buttons or menus.

Convention	Note
`monospace`	Commands, URLs, code in examples, text output on the screen, or text input.
Italic	Parameters or variables.

1 About PieCloudDB Community Edition

PieCloudDB Community Edition introduces a one-command, containerized deployment solution that is seamless, integrating the robust PieCloudDB core engine with the PieCloudDB Cluster Controller (PDBCC).

This edition is designed to support flexible deployment options, both online and offline, enabling users to quickly utilize the main capabilities of PieCloudDB. These capabilities include high elasticity; multi-tenancy support; multi-cluster management; separation of storage and computing resources; and an extensive array of ecosystem components.

Available as a free download, the Community Edition is an ideal choice for users looking to explore product features, engage in personal learning endeavors, or conduct PoC (Proof of Concept) validations. It provides a platform for the community to immerse themselves in the cutting-edge technology of data warehouse virtualization.

Product Edition Differences

PieCloudDB Community Edition lacks access to some of the advanced features that are standard in the enterprise and Cloud on Cloud (COC) editions, including but not limited to:

- An intuitive graphical interface for managing virtual data warehouses and governing data development;
- Advanced data import and export tools, such as Dataflow and Flink CDC;
- A suite of operational tools for monitoring and alerts, usage and billing, etc.;
- PieProxy, a visualization tool for external access;
- Backup and restore solutions for both system and user data;
- High availability and disaster recovery protocols for all components;
- Detailed resource allocation and isolation for CPU, memory, and IO at the node level of the virtual data warehouse;
- The ability to deploy across multiple physical machines;

If you're looking for a more immersive experience with these features, welcome to have a free trial of PieCloudDB's Cloud on Cloud Edition, please visit https://app.piecloudb.com, or get in touch with us to explore the advanced capabilities of PieCloudDB's Enterprise Edition.

Related Documentation

For an overview of PieCloudDB, please refer to the following documents:

- Overview of PieCloudDB

- System Architecture

For deployment and management features of the PieCloudDB Community Edition, please visit the following documents:

- Deploy PieCloudDB Community Edition
  - Deploy PieCloudDB Using the Online Installer
  - Deploy PieCloudDB Using the Offline Installer
- Manage PieCloudDB
  - Add a New Virtual Data Warehouse
  - Start a Virtual Data Warehouse
  - Stop a Virtual Data Warehouse
  - Resize a Virtual Data Warehouse
  - Configure Virtual Data Warehouse GUC
  - View Virtual Data Warehouse Details
  - Connect to a Virtual Data Warehouse Using a Client
- Import and Export Data

2 Introduction to PieCloudDB

2.1 Overview of PieCloudDB

PieCloudDB, the first data computing engine of OpenPie's Data Computing System, known as the PieDataComputing System (PieDataCS), is a cloud-native database system leveraging cloud computing and containerization technologies.

With its new elastic Massively Parallel Processing (eMPP) architecture, which separates storage and computation resources, PieCloudDB provides extreme flexibility, high scalability, and unbreakable security, achieving optimal cloud resource allocation.

Currently, PieCloudDB offers Cloud on Cloud (Serverless) Edition, Community Edition, and Enterprise Edition. For detailed product edition information and differences, please refer to the OpenPie Website.

Core Product Features

- Multi-dimensional Scalability and Elasticity

  PieCloudDB adopts a new eMPP (elastic Massive Parallel Processing) architecture that separates storage and computation, allowing for elastic and large-scale, parallel processing.

  For storage, it supports a standard object storage system, which can fully leverage the advantages of cloud computing platforms, offering virtually limitless capacity.

  For computation, the stateless design enables compute nodes to tap into the vast pool of cloud-based compute nodes, expanding and contracting as needed. This ensures that the system can scale horizontally, vertically, and at the cluster level, according to changes in user business requirements and data volumes.
- High Availability Capability

  PieCloudDB is built on a highly available architecture that separates metadata, computation, and data into three distinct layers. This separation allows for independent management of cloud-based storage and computational resources, including the dynamic allocation of compute resources.

  In the event of a compute node failure, the system automatically detects the issue and swiftly reallocates a new node to replace the faulty one. Each piece of metadata is distributed across multiple service nodes in the form of multiple replicas and is regularly backed up to prevent data loss.

  The integrity, consistency, and reliability of stored data are safeguarded by the distributed storage's multiple replicas, erasure coding (EC), and disaster

recovery capabilities, which reduce the likelihood of data loss due to human error or natural causes. This ensures that the failure of a single hardware component does not impact business operations.

- Lakehouse Integration Analysis

  PieCloudDB integrates a lakehouse architecture through its distributed computing engine and data interfaces, bridging the gap between data lakes and data warehouses. Its built-in Foreign-Data Wrapper (FDW) module allows users to access external data from sources such as HDFS, MySQL, and Oracle.

  PieCloudDB supports the development of custom analysis modules using procedural language (PL) and is compatible with mainstream big data processing frameworks like Spark and Flink.

- Comprehensive SQL Compatibility

  PieCloudDB offers high compatibility with the SQL:2016 standard, full support for the SQL:1992 standard, and broad support for the SQL:1999 standard. It also provides partial support for the SQL:2003 standard, with an emphasis on OLAP features.

  Additionally, PieCloudDB is compatible with the PostgreSQL protocol and supports standard database interfaces such as ODBC and JDBC.

Application Scenarios

- Diverse Data Processing

  The flexible and scalable data engine of PieCloudDB integrates a variety of databases and data processing methods. These include relational database SQL, stream and batch processing using Spark/Flink, vector databases with Large Language Models (LLMs), and GIS geographical databases. Users can choose the most suitable data storage and processing methods based on their specific needs.

- Ad-hoc Queries and Real-time Analysis

  PieCloudDB supports ad-hoc queries and real-time analysis techniques; it allows for flexible data exploration and analysis according to user requirements, making it suitable for scenarios that require real-time monitoring and decision support.

- Advanced Analytics and Machine Learning

  PieCloudDB offers a suite of advanced analytical functions, such as window functions and complex aggregation functions, to support an intricate data analysis and calculations. Users can leverage PieCloudDB for data exploration, feature engineering, and model training to facilitate machine learning and data-driven decision-making.

- Data Sharing and Analysis

  PieCloudDB provides a unified environment for data management and analysis, supporting multi-model data processing-including structured, semi-

structured, and unstructured data-data sharing, and analysis. This enables businesses and organizations to better utilize various types of data to support decision-making and innovation.

2.2 System Architecture

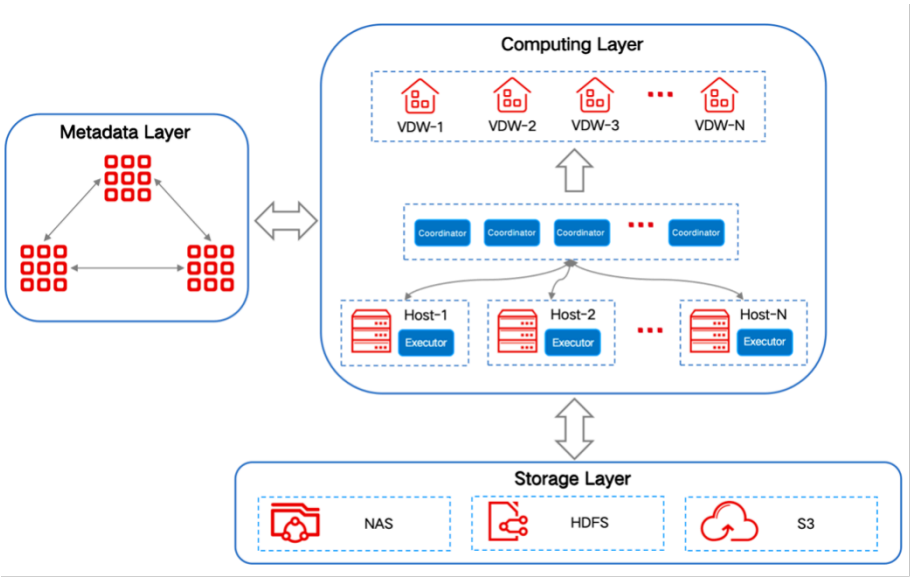
2.2.1 Overview of System Architecture

PieCloudDB utilizes a new eMPP (elastic Massive Parallel Processing) architecture, which adds elastic—a new dimension—to the traditional MPP (Massive Parallel Processing). With this architecture, PieCloudDB is capable of concurrent task execution across multiple clusters, offering the flexibility to scale the virtual data warehouse up or down. It efficiently adapts to load variations, allowing the scale of computation to dynamically match the scale of data, easily handling massive datasets at the petabyte (PB) level.

Compared to traditional OLAP databases, the cloud-native, distributed database PieCloudDB offers the following advantages:

- Storage-computing separation architecture that is highly scalable and supports rapid, elastic scaling.
- Shared storage architecture, compatible with Amazon S3, HDFS, NAS, and other protocols, ensuring high availability of data storage.
- Intelligently and efficiently generates statistical information and query plans, supporting advanced features like Aggregation Pushdown, Pre-Computation and Data Skipping, to fully meet the demands of complex analytical queries.
- Utilizing an efficient hybrid row-column storage format and a vectorized engine greatly enhances computational speed.
- Compatible with PostgreSQL's protocol, syntax, and ecosystem.

The overall architecture of PieCloudDB is as follows:



PieCloudDB features a three-layer architecture that separates metadata, computing, and user data, achieving independent management of storage and computing resources in the cloud. Cloud computing resources can be dynamically allocated and initiated on-demand based on computing tasks, with costs calculated based on usage time and scale.

2.2.2 Computing Layer

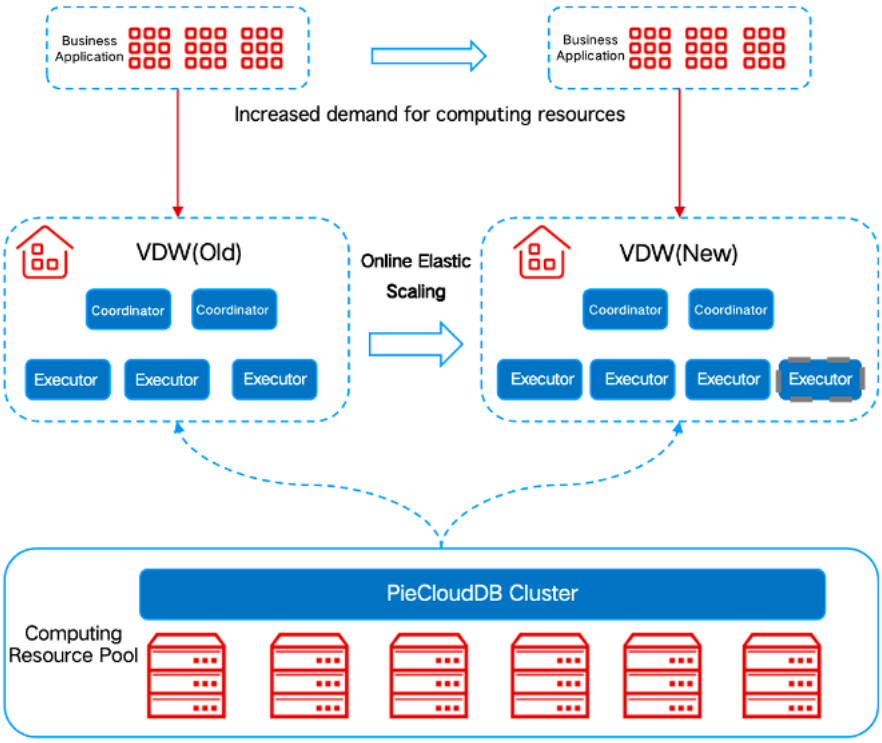
PieCloudDB's computing layer leverages eMPP technology to form a large pool of computing resources from all compute nodes. Users can create cloud-native virtual data warehouses (VDWs) on this resource pool, enabling multiple VDWs to operate simultaneously and perform concurrent data processing. Additionally, PieCloudDB can construct multiple VDWs to accommodate various business needs based on workload demands.

A virtual data warehouse comprises executors and coordinators, each operating their own independent computing clusters in a shared-nothing architecture.

During the execution of specific application requests, the coordinator oversees the entire computing resource pool, dispatching optimized SQL queries to the executors. The executors are tasked with executing these SQL queries, with multiple executors conducting parallel computations and aggregating results to return to the coordinator. The coordinator then compiles this information to provide the final output back to the application.

As computational tasks for applications increase or the volume of data to be processed grows, the demand for corresponding computing resources, primarily CPU and memory (MEM), also rises. Virtual data warehouses can share computing resources and achieve online elastic scaling based on the required computational demands.

As illustrated in the figure, the number of executors can be scaled up from three to four, and the process of scaling down adheres to the same principle.



PieCloudDB's executors are stateless. In the event of parallel data processing by multiple executors, if one fails abruptly, PieCloudDB employs a method of 'quick-fault detection --> quick-executor reconstruction' for fault recovery. This process is concise, and interactions are expedited, minimizing the impact on SQL execution and business operations, and ensuring the high availability of PieCloudDB services.

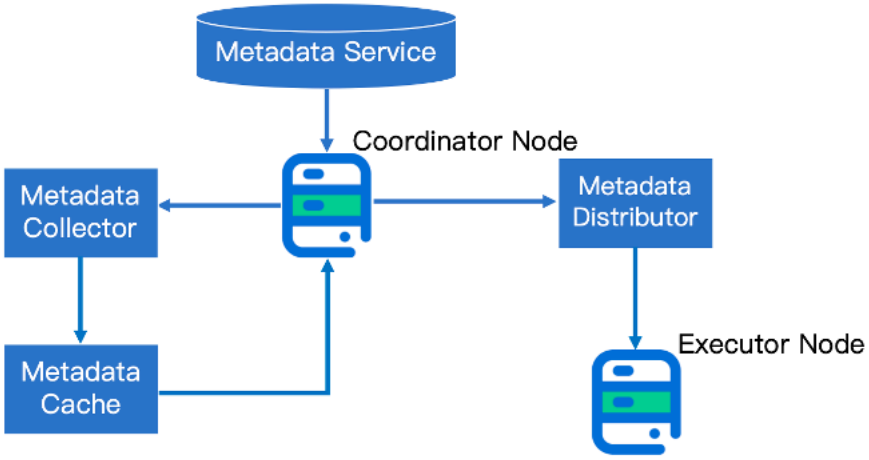
Additionally, each compute node within the computing layer is equipped with a multi-layer caching structure for both metadata and user data, which reduces network latency and minimizes data movement. This design not only enhances computational efficiency but also ensures that real-time requirements are met for users.

2.2.3 Metadata Layer

PieCloudDB's metadata primarily serves to describe, manage, and control the critical functions of the database and user data, which are managed using a distributed KV storage system. PieCloudDB's independent metadata service, known as the Catalog Service, ensures the efficiency and high availability of metadata access.

The metadata service in PieCloudDB relies mainly on three components: the metadata collector within the coordinator, the metadata cache within the executor, and the metadata distributor, also within the coordinator. During data distribution, the coordinator is responsible for analyzing user queries and collecting and returning results; the executor is responsible for processing user

queries and generating computational results. Both the coordinator and the executor require metadata to accurately analyze and process user requests.



The general workflow of PieCloudDB's metadata service can be described as follows: The metadata collector at the coordinator is responsible for collecting data from the executors; the cache is tasked with storing the collected metadata and analyzing the metadata necessary for executing current user queries; the distributor is in charge of loading the updated metadata into the executors for computation.

The metadata cache selects the appropriate caching mode based on the nature of the query request. For specific metadata requests, it transitions intelligently to a pre-computed mode, proactively calculating and caching results to reduce the data movement overhead to the metadata distributor. For other types of requests, it reverts to a standard data caching mode. Furthermore, the cache categorizes metadata into 'hot' and 'cold' tiers based on frequency of access, retaining frequently accessed 'hot' data in memory to improve cache hit rates.

The metadata collector and cache are linked to the coordinator, with each coordinator's session launching an instance of both. Similarly, the distributor is connected to the executor, and each coordinator's session also launches a distributor instance.

As a key component of PieCloudDB's architecture, the metadata layer plays a crucial role in supporting distributed and highly available services. It enables cloud data warehouses to manage various inherent challenges in distributed environments effortlessly. The layer provides robust support for data sharing, user privileges, multi-virtual data warehouse concurrency, distributed locks, and more.

2.2.4 Storage Layer

PieCloudDB's architecture incorporates a storage-computing separation model, which allows the storage layer to operate relatively independently. This design

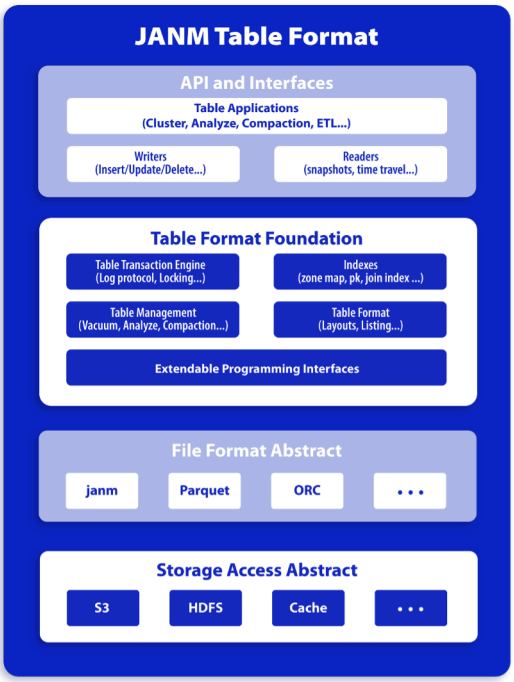
enables the storage capacity to scale independently from the computing layer. PieCloudDB supports various distributed storage solutions, such as object storage, HDFS, and NAS, providing users with scalable and shared cloud storage services for both user data and metadata.

JANM, which serves as the storage foundation for PieCloudDB's computing engine, resides above the underlying data storage layer. Designed to be compatible with various cloud environments, JANM is primarily intended for object storage, utilizing this technology as the persistent storage layer.

In response to the distributed nature and elasticity of data within PieCloudDB's eMPP architecture, JANM uses Consistent Hashing to ensure consistent data access across each computing node in the distributed system. This approach minimizes data cache migration, especially during scaling events. For data security, JANM utilizes Transparent Data Encryption (TDE) technology, employing a three-tier key system to ensure absolute data security.

The JANM storage system aims to provide a robust foundation for data management and storage in high-performance computing systems across various cloud scenarios. Its development is strategically planned in three phases: first, to become the next-generation cloud-native storage engine; second, to evolve into the cloud storage foundation for big data computing systems; and third, to transform into a unified storage engine for data computing systems. To date, JANM has successfully met the milestones of the first two stages in its evolutionary journey.

The first stage of JANM's evolution is aimed at establishing it as the cloud-native storage solution for PieCloudDB, a next-generation cloud-native virtual data warehouse. Central to this is the self-developed janm file format, which employs a hybrid row-column storage structure. This structure combines the high performance of row storage with the compression efficiency and cache line optimization of columnar storage. Furthermore, when reorganizing data, JANM redefines the data format for both disks and memory, ensuring that there is no additional overhead for data conversion between these storage media. The janm storage format is illustrated in the figure below.



The janm file format in JANM not only collects file statistics to accelerate queries but also supports a variety of performance optimization features. These include vectorized (SIMD) computation, parallel processing, and pre-computation. To further enhance I/O performance, the janm file format integrates a variety of compression algorithms, such as zstd and lz4. Additionally, JANM has been extensively optimized for data reading and querying. It implements features like data skipping, pre-computation-accelerated aggregation queries, Smart Analyze, and TOAST, all of which significantly enhance the efficiency of data loading and querying.

The second stage of JANM's evolution is aimed at establishing it as the cloud storage foundation for data computing systems. Central to this system is the JANM Table Format, which is structured into four distinct layers: Storage Access Abstraction, File Format Abstraction, Table Format Foundation, and APIs and Interfaces. Each layer depends on and extends the functionality of the layer beneath it.

The hierarchical structure of the JANM Table Format is illustrated in the figure below.

The JANM Table Format is based on the Storage Access Abstraction layer, which employs abstract APIs to interact with various types of storage, such as cloud object storage (e.g., S3) and HDFS. This approach ensures broad compatibility across all storage engines by leveraging its robust storage adapter interface capabilities. Furthermore, JANM enhances storage functionality with features like file system monitoring and a variety of read-write strategies.

The File Format Abstraction layer within the JANM Table Format supports a multitude of file formats. This capability enables JANM to adapt to various file formats while providing a unified access interface, simplifying data access and allowing users to freely choose different file formats for storage within JANM. Additionally, JANM's unique file layout design tracks all changes made to each file and maintains a separate redo log. These features enable the implementation of a broader range of functionalities.

The Table Format Foundation layer within the JANM Table Format is designed to encapsulate and implement a variety of key features. It primarily encompasses the transaction engine for the table, indexing, adaptive management of table data, and the encapsulation of operations and controls related to the table format.

The primary functions of the Table Format Foundation layer include the following:

- The transaction engine within the table employs Multi-Version Concurrency Control (MVCC) at the file level, supporting database visibility assessments based on isolation levels and ensuring effective concurrency control.
- Indexing enhances the database's ability to plan queries more efficiently, thereby reducing overall I/O and providing faster responses. Currently, JANM supports the necessary indexing for Data Skipping.

In OLAP database scenarios, indexing information about file lists and columns enables the OLAP engine to quickly generate efficient query plans.

- Adaptive data management primarily provides functions such as data cleaning (Vacuum), data distribution information sampling (Smart Analyze), and small file merging (Compaction), which can significantly enhance I/O efficiency.
- Encapsulating control and functionality related to table composition and layout facilitates rapid file traversal and enables efficient data size statistics. It also minimizes the overhead associated with list operations on files in object storage.

The JANM Table Format also supports extensible programming interfaces, offering a unified API for interacting with external services and accessing data. This facilitates the integration of third-party applications with JANM. For table application services, JANM provides stateless data management applications that can be registered with any service, thereby enabling adaptive data management across various platforms.

2.3 New Features

2.3.1 Flexible Gang

PieCloudDB's flexible gang feature enhances the ability of individual queries to utilize underlying resources in parallel, thereby simultaneously speeding up query response times.

> **Tip:**
>
> This feature is available in PieCloudDB version 2.14.0 and later versions.

Application Scenarios

The flexible gang feature is primarily applicable to scenarios that involve individual SELECT queries.

Feature Details

Use the *pdb_parallel_factor* parameter to configure the number of gangs, which determines the multiplier for the number of executor backends. The total number of computing processes is calculated as 'number of executors * pdb_parallel_factor', thereby enabling scalable parallelized queries.

The example below illustrates the working principle of this feature. First, execute an *EXPLAIN* for a *SELECT* query involving three tables: t1, t2, and t3.

```
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)
```

By setting *pdb_parallel_factor* to 3, the concurrency of query execution is increased, followed by the execution of an EXPLAIN analysis.

```
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)
```

Upon examining the EXPLAIN output from the example, the parallelization of queries has led to the following enhancements:

- The Gather Motion ratio has increased from 2:1 to 6:1, indicating that three processes are now operating in parallel on each cluster executor.
- The Broadcast Motion ratio has improved from 2:2 to 6:6, demonstrating that the broadcasting of each data set now utilizes six executors.

When the *pdb_parallel_factor* parameter is adjusted to values of 1, 2, 3, and 4, the execution time for the identical *SELECT* query is reduced progressively, indicating a linear enhancement in query performance, as shown below.

```
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)
```

The *pdb_parallel_factor* parameter takes effect dynamically and its value must be an integer. The value range is limited such that the product of 'the number of executors * pdb_parallel_factor' must not exceed the number of CPU cores (not the number of threads). The reference command is as follows:

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

2.3.2 Multi-process ic-proxy

PieCloudDB supports the multi-process operation mode of ic-proxy, which is used to enhance the load balancing capabilities of distributed systems, increase the data transfer capacity for Motion operations, and improve the parallel query performance of the database.

> **Tip:**
>
> This feature is available in PieCloudDB version 2.14.0 and later versions.

Application Scenarios

In parallel execution, if the volume of Motion data is substantial, a single-process ic-proxy can quickly reach a CPU usage rate of 100%. This can result in an overloaded system and consequently degrade query performance.

In contrast, a multi-process ic-proxy effectively enhances load distribution and optimizes network performance. This is instrumental in mitigating the

performance bottlenecks typically associated with single-process parallel queries when handling large datasets.

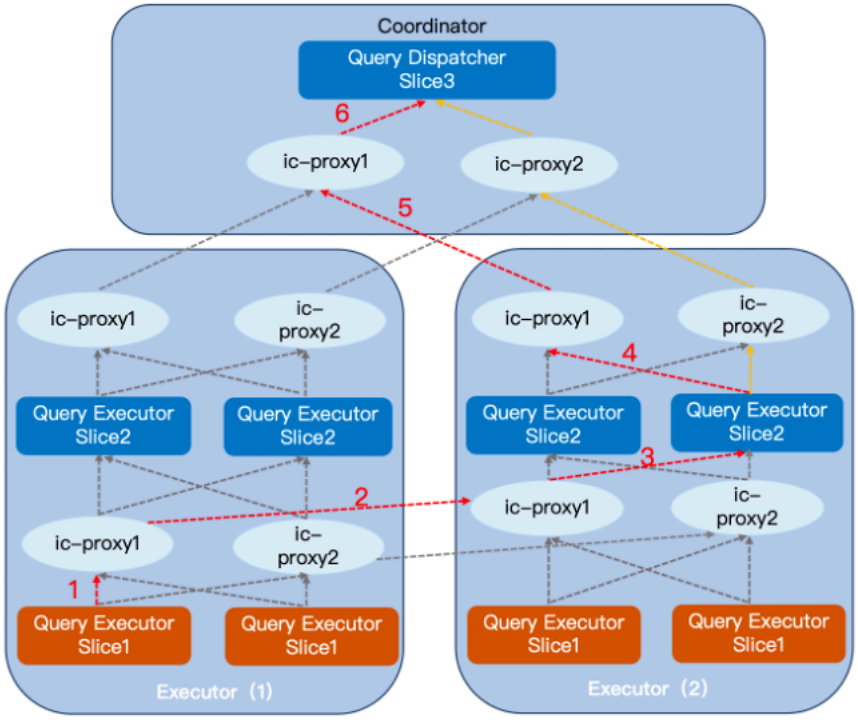
Feature Details

PieCloudDB's parallel computing is accomplished through a coordinator and one or more executors. The coordinator is responsible for receiving, parsing, and optimizing queries, while the executors carry out the tasks assigned by the coordinator. Multiple executors are capable of executing a single SQL query task in parallel.

During query processing, PieCloudDB creates several processes to handle parallel query tasks. If Motion operations are involved, data must be transferred between executors through the Interconnect's ic-proxy. When a query plan includes a Motion operation, the plan is divided and distributed across the endpoints involved in the data transfer.

When executing parallel queries, each executor runs several processes in parallel, with each slice of the query plan assigned to at least one worker process, which operates independently on its portion of the query plan.

The concept of PieCloudDB's ic-proxy is illustrated in the figure below.



In scenarios involving parallel queries, the example uses the data transfer process from executor (1) of Slice1 to executor (2) of Slice2 to explain how multi-process ic-proxy transfers Motion data.

1. Executor (1) in Slice1 sends data to its corresponding ic-proxy1.
2. Upon receiving the data, ic-proxy1 of executor (1) routes it to ic-proxy1 on executor (2) within Slice1.

3. The ic-proxy1 on executor (2) within Slice1 then routes the data to the execution process corresponding to Query Executor (QE) in Slice2, based on the ic-proxy Key.
4. The QE in Slice2 then sends the data to its corresponding ic-proxy (either ic-proxy1 or ic-proxy2).
5. The ic-proxy (either ic-proxy1 or ic-proxy2) on executor (2) in Slice2 sends the data to the coordinator's ic-proxy (either ic-proxy1 or ic-proxy2).
6. Finally, the results are returned to the Query Dispatcher (QD) of Slice3 through the coordinator's ic-proxy (either ic-proxy1 or ic-proxy2).

Note that the receiving and sending ends of ic-proxy_i between different executors or between an executor and the coordinator are uniquely corresponding; for example, ic-proxy_1 can only accept data from or send data to ic-proxy_1.

The coordinator and executor communicate with ic-proxy through Domain Sockets. For instance, when the coordinator or an executor sends data to an ic-proxy, the ic-proxy process, acting as the server, listens for this connection upon startup, waiting for the coordinator or executor to establish a connection and send data. Since each ic-proxy process does not reside on the same executor (physical machine), inter-process data transfer is conducted over network Sockets.

Use the parameter `pdb_ic_proxy_num_worker` to set the number of ic-proxy processes launched on the coordinator and executors. Both the coordinator and executors should start the same number of ic-proxy processes. The value range for this parameter depends on the network bandwidth of the physical machine, with a minimum value of 1. This is also the default value, indicating that multi-process ic-proxy is not enabled.

The `pdb_ic_proxy_num_worker` parameter is typically configured during the initial deployment. It is important to note that changes to this parameter's value require a restart of the virtual data warehouse to take effect.

2.3.3 LocalCache

LocalCache in PieCloudDB manages local file caches and offers capabilities for file uploads, downloads, error processing, cache querying, and cache clearing. It efficiently minimizes network communication costs associated with fetching files from cloud storage services like S3 and helps reduce latency during queries.

> **Tip:**
>
> This feature is available in PieCloudDB version 2.13.0 and later versions.

Application Scenarios

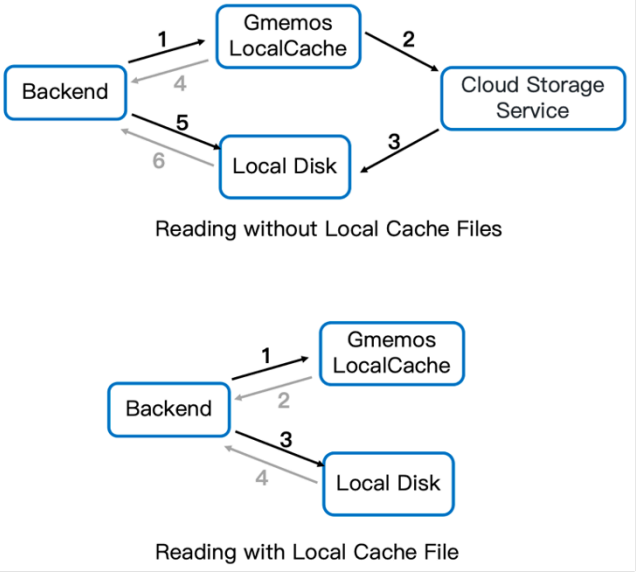
LocalCache retains historical query records for files, making it ideal for situations where the same or a few tables are accessed repeatedly. However, in scenarios with numerous queries across different tables, the historical cache may be

replaced and purged, which could potentially not provide a significant boost to query performance.

Feature Details

PieCloudDB has separated file uploading and downloading from the Backend, assigning these tasks to the LocalCache module. As a result, the Backend does not need to interact directly with cloud storage services such as S3. Instead, it sends query requests to LocalCache, which then carries out the corresponding file operations based on the requests from the Backend.

The diagrams below depict the file reading process in two scenarios: 'without local cached files' and 'with local cached files'.



If LocalCache is enabled, the existing local cache files for the database will persist after a restart and can continue to be utilized.

To utilize the LocalCache feature, you need to configure the following parameter:

- *pdb_janm_enable_disk_cache* setting determines whether the LocalCache feature is activated. 'ON' enables the feature, while 'OFF' disables it. If LocalCache is disabled, files will be directly loaded into memory without being cached on the local disk. The following example demonstrates how to enable LocalCache.

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

- *gmemos.local_cache_usage* configures the cache space size for LocalCache, with a default value of 10240. It is recommended that the minimum cache size be set to 'BlockSize * concurrency level'. Values below this threshold may cause some threads to enter a waiting state, thereby reducing query performance.

  The larger the cache space, the more files can be cached, and the higher the cache hit rate, resulting in better performance. If there is insufficient disk space available for local caching due to low local disk capacity or other reasons, it

is advised to disable the LocalCache feature. If the current cache space is full, new query requests will trigger a cleanup operation.

- *gmemos.single_table_constrict_percentage* sets the maximum percentage of cache space that can be occupied by single-table caches in LocalCache. To prevent excessive consumption of cache by large table data during reads, ensure that the value of 'single_table_constrict_percentage * local_cache_usage' is greater than the BlockSize.

Additionally, the function *pdb_get_local_cache_hit_rate()* can be used to query the cache hit rate and other information for the current Segments. The reference command is as follows:

```
=> 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)
```

Details for the *pdb_get_local_cache_hit_rate()* function are as follows:

Fields	Data Types	Details
seg_id	smallint	Segment ID
cache_size	bigint	Existing Cache Size
cache_file_count	bigint	Numbers of Cache Files
hit_query_count	bigint	Cache Hits
total_query_count	bigint	Cache Requests
hit_rate	text	Hit Rate

3 Deploy PieCloudDB Community Edition

3.1 Overview of PieCloudDB Community Edition Deployment

A typical PieCloudDB cluster comprises computing services (including a coordinator and multiple executors), metadata services, and object storage services. The PieCloudDB Community Edition offers a deployment solution with an all-inclusive image that includes all the essential services mentioned. It facilitates a One-Command deployment process for the PieCloudDB cluster.

The PieCloudDB Community Edition deployment image encompasses computing services, including a coordinator and three executors; FoundationDB services; MinIO object storage services; and metadata caching services. Additionally, it includes the PieCloudDB Cluster Controller (PDBCC) and two PieCloudDB Cluster Management Agents (PDBCAs).

Deployment Methods

The PieCloudDB Database Community Edition supports the following two containerized deployment methods:

- Deploy PieCloudDB Using the Online Installer
- Deploy PieCloudDB Using the Offline Installer

Environment Preparation

Ensure that the system meets the following prerequisites prior to deployment:

- Ensure that you have a Linux system with an x86 architecture and that Docker is installed.
- It is recommended to allocate at least 8 GB of RAM and 10 GB of disk storage for Docker.
- Install the PostgreSQL client, psql.To do so, follow these commands:
  - Ubuntu Operating System:

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

  - Centos Operating System

    ```
    sudo yum install postgresql
    ```

  Execute the command `psql --help`. If the help information for `psql` is displayed, it confirms that the installation has been successful.

3.2 Deploy PieCloudDB Using the Online Installer

This chapter describes how to deploy PieCloudDB Community Edition using the online installer. The online installer downloads the PieCloudDB images from Docker Hub.

Log into the deployment environment with a user account that has sudo privileges, and then follow the steps below to deploy the PieCloudDB Community Edition:

1. Launch the PieCloudDB Community Edition image using the following Docker command:

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

2. Examine the logs to verify that the PieCloudDB deployment has been successful.

```
sudo docker logs -f piecloudb
```

Here is an example of the output:

```
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"
```

After deployment, connect to PieCloudDB using the psql command and perform a test to verify the setup, see Connect to PieCloudDB Using the psql Command for Testing.

3.3 Deploy PieCloudDB Using the Offline Installer

This chapter describes how to deploy the PieCloudDB Community Edition with the offline installer. Use the offline installer if the host to which you are deploying PieCloudDB does not have an Internet connection.

Log into the deployment environment with a user account that has sudo privileges, and then follow the steps below to deploy the PieCloudDB Community Edition:

1. Download the offline installer package for PieCloudDB Community Edition.

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

2. Extract the downloaded offline installer package for PieCloudDB Community Edition to the desired directory.

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

   Here is an example of the output:

```
./piecloudb-ce-allin1/
./piecloudb-ce-allin1/start-piecloudb.sh
./piecloudb-ce-allin1/piecloudb-ce-allin1.tar
```

3. Navigate to the installer directory and execute the script *start-piecloudb.sh* to initialize PieCloudDB.

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

   Here is an example of the output:

```
Loaded image: reg.dev.openpie.com/release/piecloudb-ce-allin1:latest
piecloudb is starting
print log: docker logs -f piecloudb
```

4. Check the logs to verify that the deployment of PieCloudDB has been successful.

```
sudo docker logs -f piecloudb
```

   Here is an example of the output:

```
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"
```

After deployment, connect to PieCloudDB using the psql command and perform a test to verify the setup, see Connect to PieCloudDB Using the psql Command for Testing.

3.4 Connect to PieCloudDB Using the psql Command for Testing

After deployment, connect to PieCloudDB using the psql command and perform a test to verify the setup.

The following connection example utilizes the default user 'openpie' and the default database 'openpie'.

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

Execute the following simple SQL statements to test the functionality of your PieCloudDB installation:

```
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 Manage PieCloudDB

4.1 Overview of PieCloudDB Management

The PieCloudDB Community Edition includes the pdbcli command-line utility for managing virtual data warehouses. This utility is pre-packaged within the installation package (allInOne.tar.gz), eliminating the need for users to install it separately.

> **Attention:**
>
> sudo privileges are required to execute the pdbcli command within a Docker container.

4.2 Add a New Virtual Data Warehouse

The `pdbcli cluster create` command is used to create and add a new virtual data warehouse. The `--cluster-size` option specifies the number of executors within the virtual data warehouse, and the `--tenant` option specifies the user.

Note that the ports for the new virtual data warehouse are incremented based on the initial port number. The default initial port for the coordinator is 5433, and the default initial ports for the three executors are 6003, 6004, and 6005.

The following example shows how to create a virtual data warehouse with three executors for the default user 'openpie'.

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

Here is an example of the output:

```
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"
```

The output information indicates that the newly created virtual data warehouse has an ID of 2. The coordinator, identified by a content ID of -1, runs on port 5433, and the three executors, identified by content IDs of 0, 1, and 2, operate on ports 6003, 6004, and 6005, respectively.

4.3 Start a Virtual Data Warehouse

The `pdbcli cluster start` command is used to initiate the virtual data warehouse online. The virtual data warehouse created and added via the `pdbcli cluster create` command only completes the necessary initialization tasks, such as setting up directories. The `--cluster` option is used to specify the ID of the virtual data warehouse.

The following example demonstrates how to start a virtual data warehouse with an ID of 2.

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

Here is an example of the output:

```
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 Stop a Virtual Data Warehouse

The `pdbcli cluster stop` command is used to stop a virtual data warehouse, thereby releasing CPU and memory resources.

The following example demonstrates how to stop a virtual data warehouse with an ID of 2.

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

Here is an example of the output:

```
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 Resize a Virtual Data Warehouse

A virtual data warehouse can be scaled out or in by increasing or decreasing the number of executors as needed, with the coordinator remaining static as a single instance.

The pdbcli cluster resize command is used to adjust the number of executors in a virtual data warehouse. The `--cluster` option specifies ID of virtual data warehouse's ID, and the `--cluster-size` option specifies the desired number of executors. It is important to ensure that the target virtual data warehouse is stopped before performing the resize operation.

The following example demonstrates how to set the number of executors to 4 for a virtual data warehouse with an ID of 2.

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

Here is an example of the output:

```
2024/08/08 09:40:25 PrepareSegment cluster 2 content 3 on segment
```

Note that once the number of executors in a virtual data warehouse has been modified, it is necessary to restart the virtual data warehouse for the changes to take effect. Here is an example:

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

Here is an example of the output:

```
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
```

The output information above indicates that the virtual data warehouse has one coordinator, identified by a content ID of -1, and four executors, each with content IDs of 0, 1, 2, and 3.

4.6 Configure Virtual Data Warehouse GUC

The `pdbcli cluster config` command is used to reset the configuration settings of a virtual data warehouse. The `--cluster` option specifies the ID of the virtual data warehouse, the `--name` option specifies the parameter to be modified, and the `--value` option specifies the new value for that parameter. Please be aware that you must restart the virtual data warehouse for the changes to take effect.

The following example demonstrates how to set the value of `pdb_parallel_factor` to 3 for a virtual data warehouse with an ID of 2.

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

If the command executes successfully, it will display the following message:

```
Config success
```

Then, to ensure the changes take effect, restart the virtual data warehouse by running the following command:

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

Connect to the database and use the `SHOW` statement to verify if the changes have taken effect:

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

4.7 View Virtual Data Warehouse Details

The `pdbcli cluster list` command retrieves information about the virtual data warehouses that currently exist. The `--tenant` option specifies the user.

The following example demonstrates how to view details of all virtual data warehouses for the default user 'openpie'.

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

Here is an example of the output:

```
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"
```

Use the `--cluster` option in the command to specify the ID and view detailed information for a particular virtual data warehouse. The following example demonstrates how to check the details for a virtual data warehouse with an ID of 2.

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

Here is an example of the output:

```
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 Connect to a Virtual Data Warehouse Using a Client

The PieCloudDB Community Edition currently supports client connections, including those made via psql and DBeaver, to virtual data warehouses.

Connect via Postgres Client PSQL

The format for connecting to a virtual data warehouse using the PostgreSQL client (psql) is as follows:

```
psql -h <IP address> -U <username> -p <port number> -d <database name>
```

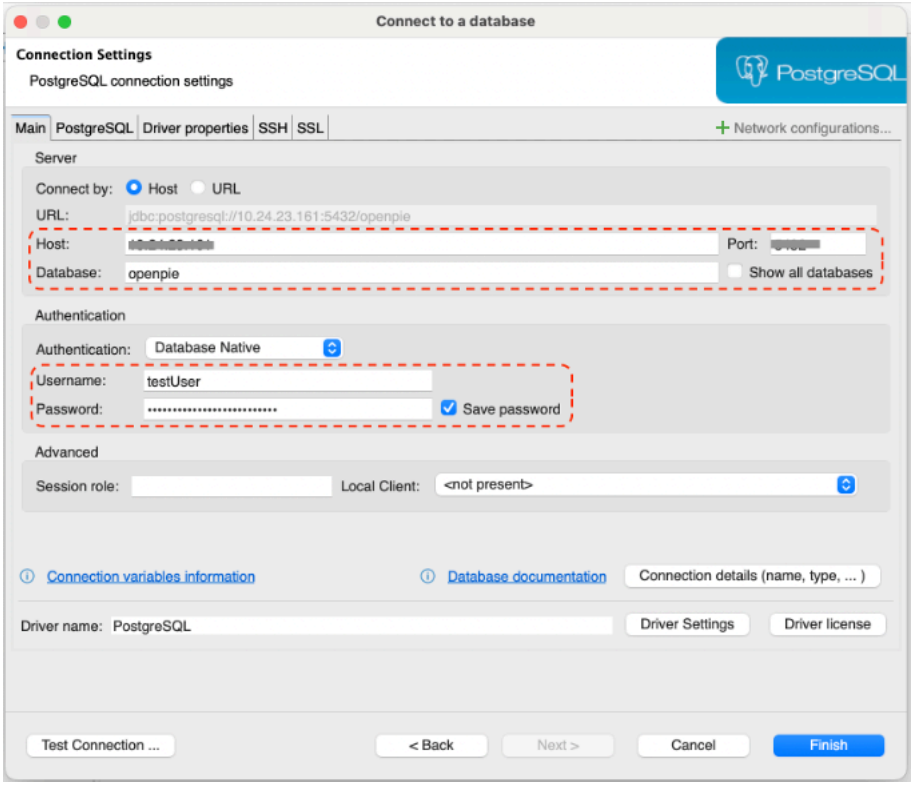
Here is an example:

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

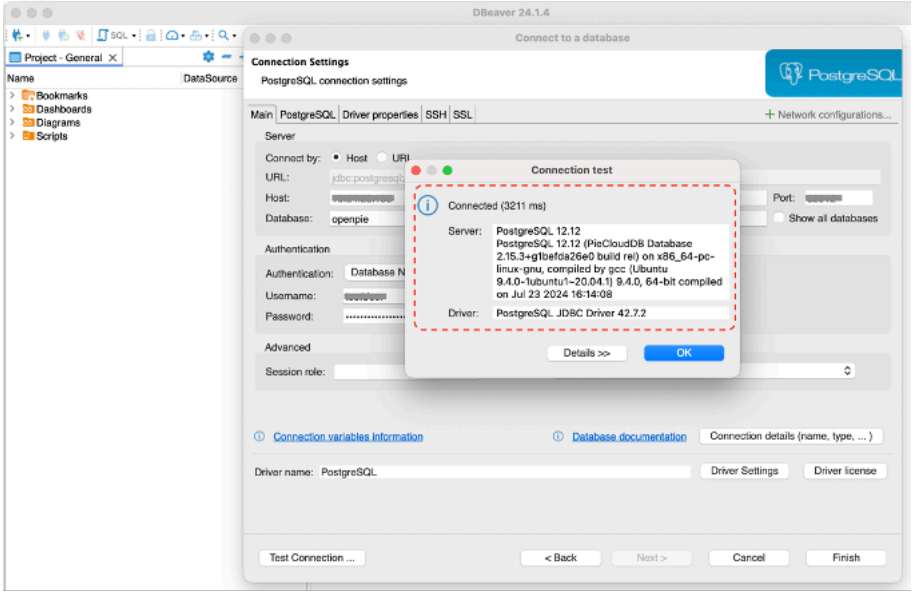
Connect via DBeaver

The steps to connect to a virtual data warehouse using DBeaver are outlined below:

1. Log in to DBeaver, go to **Database** > **New Database** Connection, select PostgreSQL, and configure the host, database, username, and password.

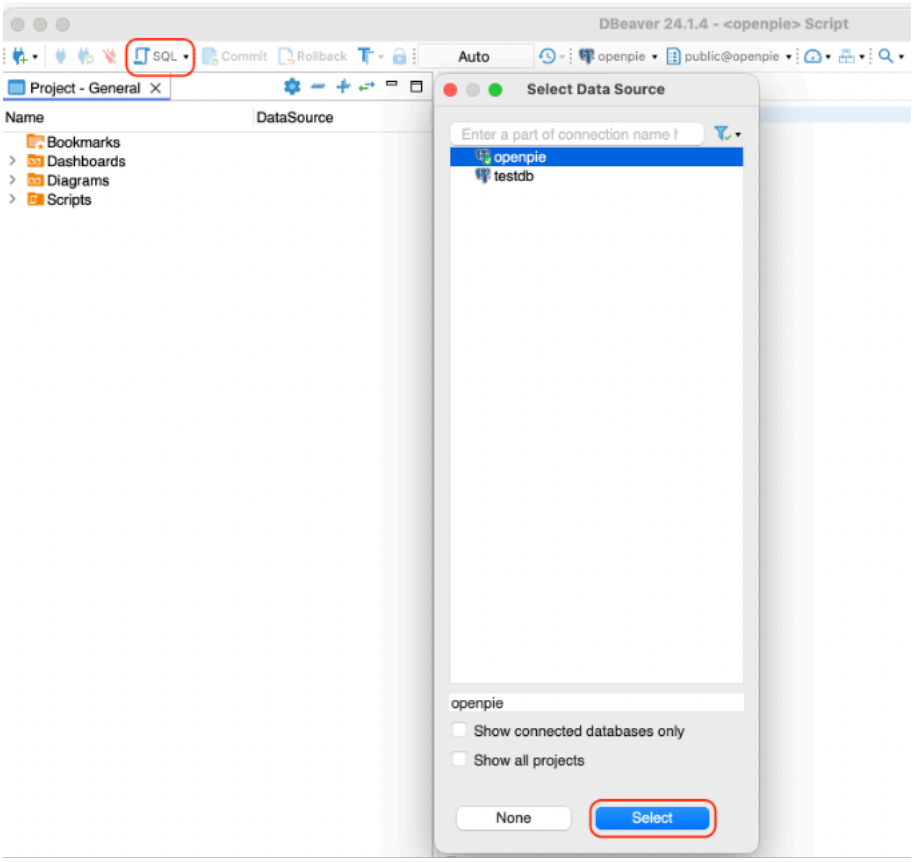


2. Click **Test Connection** to verify the connection to the database. The following figure illustrates an example of a successful connection.



Then click **OK** to close the window.

3. Click **Finish** to complete setup for the database connection.
4. Click **SQL** to select the configured database connection, and then click **Select** to add an SQL file.

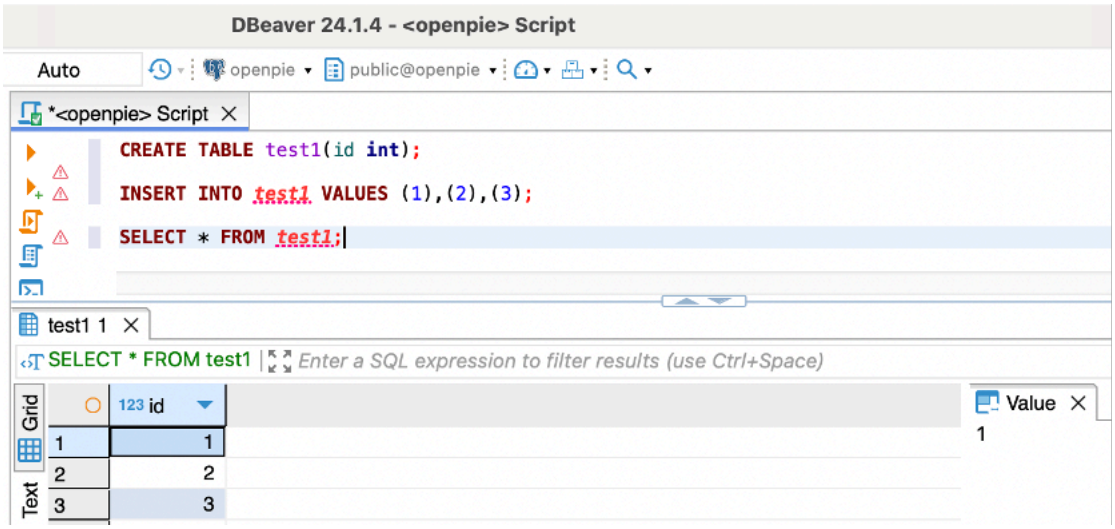


5. Create a new table named test1 in the 'openpie' database, insert sample data, and then run an SQL query to verify the results.

```
CREATE TABLE test1(id int);
INSERT INTO test1 VALUES (1),(2),(3);

SELECT * FROM test1;
```

The results are shown in the figure below.



Connect via Python psycopg2

To connect to a PostgreSQL database using the psycopg2 library in Python, ensure that psycopg2 is installed in your Python environment. Here is an example of how to establish a database connection:

```
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 Import and Export Data

5.1 Import Data

PSQL is supported in PieCloudDB for executing the `COPY` command, which facilitates the import of data from files or standard input into tables.

The `COPY` command operates in parallel, meaning that the Query Dispatcher (QD) divides the data file into slices and simultaneously distributes these slices to different executors for parsing and loading.

Both PieCloudDB and the client can utilize STDIN and STDOUT to transfer data within the command-line environment. When connected via `psql`, the `\copy` command can be used to specify the file path on the client host.

Data Format

The input format for the `COPY FROM` command is determined by the `FORMAT` parameter, which accepts 'text', 'csv' (for comma-separated values), and 'binary' as valid values, with 'text' being the default. Additionally, the `DELIMITER` option allows you to specify different characters as value delimiters.

The syntax for `COPY FROM` command is as follows:

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

CSV files use commas to separate values. By default, text files use tabs as delimiters, but you can specify different characters as value delimiters using the `DELIMITER` option.

The following example demonstrates the use of the vertical bar (|) character as the value delimiter for a text file.

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

Input data is parsed based on the `ENCODING` option or the current client encoding, and output data is encoded using either the `ENCODING` setting or the client's current encoding.

By default, PieCloudDB uses the client's default encoding. You can change this setting using the `ENCODING` option.

The following example demonstrates changing the file encoding to 'latin1'.

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

Load Data From Host Files

When using the `COPY` command to import data from a host file, the PieCloudDB server needs to open, read, and load the file's content into the target table. Users must have `INSERT` privileges for the target table as well as the necessary file read privileges.

The `COPY FROM` command requires the file location to be specified using either an absolute path on the host or a relative path to the data directory. The syntax is as follows:

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

The following example demonstrates how to copy data from a host file into the table 'country'.

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

Load Data from STDIN

The `STDIN` channel utilizes the standard input to provide data directly to the server process. Once the `COPY FROM STDIN` command is executed, the server backend begins accepting data, which continues until a line containing only a dot (.) is encountered.

The syntax for the `COPY FROM STDIN` command is as follows:

```
COPY table_name FROM STDIN;
```

The following example imports data into the table t via `STDIN`. Note that the column delimiter for each line is a tab.

```
=> 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
```

Load Data From Client Files

Unlike the `COPY` SQL command, the `\copy` command in psql loads data by invoking `COPY FROM STDIN` to send data from the psql client to the server's background process. All files must reside on the host where the psql client is running and must be accessible by the user executing the client.

Once the `COPY FROM STDIN` command is initiated, the server backend begins to accept data, continuing until a line containing only a dot (.) is encountered. Psql encapsulates these capabilities into the `\copy` command, with the syntax as follows:

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

The example below demonstrates the use of the command to copy data from the client file 'country_data.text' into the table 'country'.

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

5.2 Export Data Using COPY TO

PieCloudDB supports the use of PSQL to execute the `COPY` command for exporting data from tables to files or standard input.

You can use the `COPY TO` command to copy the contents of a table to a file, and using this command requires that the user has `SELECT` privileges on the table. The syntax is as follows:

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

The `COPY TO` command can be used to copy the results of a `SELECT` query to a file. If specific columns are specified, the `COPY TO` command will copy only the data from those columns. This command is applicable only to regular tables and cannot be used for views. It does not copy rows from subtables or subpartitions

The `COPY TO` command supports the specification of file formats, field delimiters, representations of null values, and other options. These parameters can be defined using the `WIT` clause, followed by the appropriate options.

The following example illustrates how to export data from the table 't' in text format, with fields delimited by '|', and null values represented as 'N/A'.

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

The `COPY TO STDOUT` command provides a standard output channel for data in a table. The syntax for this command is as follows:

```
COPY table_name TO STDOUT;
```

The example below demonstrates how to export data from the table t via `STDOUT`, with the data displayed in text format on the terminal.

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

If you need to export data from a table to the local client, use the `\copy` command, which invokes the `COPY TO STDIN` command. The syntax is as follows:

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

The following example demonstrates how to export the column 'name' from the table 't' to a client-side file named 'data'.

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

5.3 Format Data Files

When importing and exporting data, the `COPY` command supports specifying the text or CSV (Comma-Separated Values) data format. Therefore, external data must be properly formatted to be accurately read by PieCloudDB.

Format Rows

Text files use the Line Feed (LF) character to separate different rows, ensuring that the text is displayed in the correct format on the screen or in print output.

Line Feeds vary depending on the operating system and application. PieCloudDB expects rows of data to be separated by the character LF (Line Feed, \n or 0x0A), CR (Carriage Return, \r or 0x0D), or CRLF (Carriage Return + Line Feed, \r\n or 0x0D 0x0A). On UNIX or UNIX-like operating systems, LF is the standard representation for a new row. Operating systems like Windows use CR or CRLF. PieCloudDB supports all these line representations as row delimiters.

Format Columns

The tab character is often used to create tables, align text and columns in code, or create indentations in documents. The default column delimiter for text files is the horizontal tab character (TAB, 0x09), and for CSV files, it is the comma character (0x2C).

When defining a data format, use the `COPY` or `CREATE EXTERNAL TABLE` command with the `DELIMITER` option to designate a single-character column

delimiter. The delimiter character must appear between any two data value fields and should not be placed at the beginning or end of a row.

The following example shows that the vertical bar character (|) is used as the delimiter.

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

The following example demonstrates the use of the vertical bar character (|) as a column delimiter in the *CREATE EXTERNAL TABLE* command.

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

Identify NULL Values

Identifying NULL values is crucial for handling unknown data or blank values in columns or fields. Users may designate a string to represent null values in data files.

The default NULL string representation is *\N*(a backslash followed by the letter 'N') for text format, or an unquoted empty string for CSV format.

When defining a data format, the *COPY* command uses the *NULL* option to designate a string that identifies null values. The following example demonstrates using 'N/A' to indicate null values.

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

Escaping

PieCloudDB reserves the following two types of characters for special purposes:

- Delimiter characters used in data files to separate columns or fields, such as the vertical bar (|) and the comma (,).
- Line feed characters used in data files to denote a new row, such as '\n'.

These characters, when present in the data, must be escaped so that PieCloudDB recognizes them as data rather than as delimiters or line breaks.

By default, the escape character for text format files is a backslash (\), while for CSV format files, it is a double quotation mark (").

Escaping in Text Formatted Files

Execute the *COPY* command with the *ESCAPE* option to designate an alternative escape character. If the escape character appears in the data, it must escape itself.

The following example demonstrates importing the following data into a table with three columns:

- A backslash = \
- A vertical bar = |
- An exclamation point = !

If a user designates the delimiter character as the vertical bar (|) and the escape character as the backslash (\), the formatted row in the data file would appear as follows:

```
A backslash = \\ | A vertical bar = \| | An exclamation point = !
```

Users can utilize the escape character to escape decimal and hexadecimal sequences. The escaped values are transformed into their equivalent characters when loaded into PieCloudDB. The following example demonstrates using the escape character to represent its equivalent hexadecimal value (\0x26) or octal value (\046) to escape the ampersand character (&).

The *ESCAPE* option in the *COPY* command allows users to deactivate escaping in text-formatted files as follows.

```
ESCAPE 'OFF'
```

This feature is particularly useful when the input data contains a high frequency of backslashes, such as in web server log files.

Escaping in CSV-Formatted Files

By default, the escape character for CSV-formatted files is a double quotation mark ("). Users can designate a different escape character using the *ESCAPE* option in the *COPY* command. If the escape character appears in the data, it must escape itself.

The following example demonstrates how to import the data below into a table with three columns:

- Free trip to A,C
- 3.69
- Special rate "2.75"

If the user designates the delimiter character as a comma (,) and the escape character as a double quotation mark ("), the formatted row in the data file will appear as follows:

```
"Free trip to A,C","3.69","Special rate ""2.75"""
```

Users can also enclose entire fields within double quotes to ensure that any leading or trailing whitespace is retained. The following example demonstrates how spaces at the start and end of the text within each field are preserved:

```
" Free trip to A,B ","5.89 ","Special rate ""1.79"" "
```

Attention:

In CSV-formatted files, all characters are significant. If the data being imported comes from a system that uses padding with spaces to achieve fixed width, it is necessary to preprocess the CSV file by removing trailing whitespace before importing.

5.4 Character Set

PieCloudDB supports a variety of character sets, including single-byte character sets such as the ISO 8859 series, and multi-byte character sets like EUC, UTF-8, and Mule internal encoding.

The server-side character set is defined during the database initialization process, with UTF-8 being the default character set, although it can be changed. Clients can transparently use all supported character sets; however, some character sets are not supported as server encodings.

When data is imported, PieCloudDB converts the data from the client-specified encoding to the server encoding. Conversely, when data is exported to the client, PieCloudDB transforms the data from the server's character encoding to the client-specified encoding.

Data files must use a character encoding that PieCloudDB can recognize. If they contain illegal or unsupported encoding sequences, the data files will generate an error upon loading.

Noted that data files generated on the Microsoft Windows operating system need to be processed with the dos2unix system command to remove Windows-specific characters before being loaded into PieCloudDB.

The client's character encoding can be set through the server configuration parameter *client_encoding*. An example is as follows:

```
SET client_encoding TO 'latin1';
```

The *RESET* command is used to revert the client's character encoding to its default value. The syntax of the command is as follows:

```
RESET client_encoding;
```

The *SHOW* command is used to display the current client-side character encoding settings. The syntax of the command is as follows:

```
SHOW client_encoding;
```

6 Glossary

A

Analyze

PieCloudDB collects statistical metadata from the tables within a data warehouse and stores the results in system tables. The query optimizer utilizes this data to devise the most effective execution plan for queries.

C

Catalog Service

PieCloudDB provides management and storage services for metadata, which is essential for the collection, organization, storage, and management of data assets.

The Catalog Service helps organizations better understand, manage, and leverage their data assets. At the metadata layer, PieCloudDB offers a dedicated service to ensure efficient access and high availability of metadata.

Cloud Native

Cloud-native refers to systems or applications that are designed, developed, and operated entirely on cloud platforms, fully leveraging the advantages of cloud computing models.

Coordinator

In a PieCloudDB virtual data warehouse, the coordinator node is responsible for coordination and management. It primarily handles SQL requests from clients, generates query plans, distributes execution information, manages metadata, and compiles final results.

D

Database

A database is composed of schemas and tables, which includes only database objects and excludes computational resources. In PieCloudDB, all data is preserved and managed within databases. Users can create multiple databases to ensure data integrity and security through segregation.

Database Object

A database object is a data structure designed to store or reference data. Tables are a common type of database object, and other objects include indexes, stored procedures, functions, sequences, views, etc. You can create database objects and become the owners of these objects. You can also manage your own database permissions for other roles, such as granting read and write access.

E

elastic Massive Parallel Processing(eMPP)

elastic Massive Parallel Processing (eMPP), similar to traditional MPP, is a computational framework that employs multiple processors to concurrently manage a single program, forming a key component of database architecture.

eMPP capitalizes on cloud computing capabilities to separate database metadata, storage, and computation, with metadata operating as a distinct system that stores data on users, database objects, and computational nodes. When performing data queries, all computational tasks retrieve from the same metadata set to execute their calculations.

Once the eMPP system receives a computational task, the coordinator parses the task (e.g., a query) using a query planner and optimizer, transforming it into a plan that includes multiple sub-queries. The coordinator then assigns these tasks to the executors. The computational nodes dynamically adjust based on the specifics of the query to optimize computational efficiency.

ETL

ETL, an acronym for Extract, Transform, and Load, refers to the comprehensive data processing workflow that involves extracting data from a source, transforming it to meet the specifications of the target system, and then loading it into the destination.

Executor

Executors are responsible for executing tasks that are delegated by the coordinator within a virtual data warehouse. A virtual data warehouse contains one or multiple executors to execute the same SQL query task across multiple executors in parallel.

Execution Plan

The Database Management System (DBMS) constructs an execution plan based on the query statement, table structure, index statistics, and additional information when executing a query or operation. It provides the most efficient execution path for the query based on the optimizer's algorithms and rules.

G

Global Datasphere

Datasphere, a concept introduced by IDC (International Data Corporation), refers to the total volume of data generated, collected, or replicated on an annual basis. It can also refer to all the data stored across various storage media, including hard drives, SSDs, optical discs, magnetic tapes, and others.

H

High Availability (HA)

High Availability (HA) refers to the characteristic of a system or service that ensures highly reliable and continuous operation, maintaining business in good operating status even in the event of service failures.

J

JANM

PieCloudDB features an efficient data caching structure and a new storage engine named 'JANM', which is primarily designed for object storage and utilized as the persistent storage layer. Under the eMPP architecture, data is distributed and elastic.

JANM leverages consistent hashing to ensure data access consistency across each computational node within the distributed system. This approach minimizes the data caching volume even during scaling operations.

M

Massive Parallel Processing (MPP)

Massive Parallel Processing (MPP) is a high-performance computing architecture that harnesses multiple processors to execute a single program concurrently.

Commonly used in advanced database systems, MPP utilizes a cluster where each node has its own operating system, CPU, and memory to manage separate portions of the workload. Optimizing data distribution across nodes and effectively balancing the computational load are critical in designing an MPP framework.

Metadata

PieCloudDB's metadata is stored separately from user data. It includes not only the traditional metadata associated with user data, such as names, sizes, and

field information, but also extends to include statistical data about users, user profiles, and virtual data warehouse details.

Metadata is frequently utilized in PieCloudDB for data functionality development and performance optimization. For instance, the query optimizer 'Dachi' leverages statistical metadata to refine query plans.

Q

Query Planner

The Query Planner, a critical component of a database system, receives query requests and integrates various factors—including the database schema, indexes, and current system load—to formulate an effective execution strategy. It undergoes a series of processes, including parsing, analyzing, generating, and optimizing an execution plan to enhance performance and efficiently deliver the results.

Query Optimizer

In certain scenarios, the Query Optimizer can further optimize an existing query plan. In PieCloudDB, the 'Dachi' Query Optimizer manages the entire process, from receiving a query request to optimizing and executing the query plan.

S

Serverless

The 'Serverless' approach to data warehousing automates all underlying resource management through the cloud-based virtual data warehouse backend. This allows users to utilize warehousing services without worrying about server architecture or technical complexities, thereby providing a seamless and smooth user experience.

Schema

A schema constitutes the data structure of PieCloudDB, representing a collection of tables, as well as the functions and views derived from them.

Storage Engine

A component or module within a Database Management System (DBMS) handles data storage and retrieval, defines how data is organized, and determines how it is read from or written to physical storage media. JANM is the data storage foundation of PieCloudDB's computing engine. For more details, see the entry for **JANM**.

T

Transparent Data Encryption(TDE)

Transparent Data Encryption (TDE) uses an encryption key, known as the Data Encryption Key (DEK), to dynamically encrypt and decrypt data in transit between storage and the application. This process ensures seamless and transparent data protection for the application.

Tuple

A tuple refers to a set of ordered values that constitute a row or record within PieCloudDB. A tuple consists of multiple fields or attributes, each with a defined data type.

V

Virtual Data Warehouse

A virtual data warehouse represents a dynamic cluster of pure computational resources, such as CPU, memory, and temporary storage, provided by cloud-based virtual machines. It does not contain actual data but only offers SQL execution services.

A virtual data warehouse comprises at least one coordinator and multiple executors, with each node being a virtual resource group that includes CPU, memory, and disk resources. At the user level, there is no direct access to information about these nodes; however, users with the appropriate permissions can see the processes for starting, stopping, and scaling the virtual data warehouse.

Contact us

community@openpie.com