
SynxDB Documentation

Release v4.0.0

Synx Data Labs

Jul 15, 2025

Table of Contents

1 Product Overview	1
1.1 Feature Overview	1
Efficient queries in different scenarios	1
Polymorphic data storage	2
Multi-layer data security	3
Data loading	4
Multi-layer fault tolerance	4
Rich data analysis support	5
Flexible workload management	6
Multiple compatibility	7
1.2 Product Architecture	8
Coordinator node	9
Segment node	9
Interconnect	10
MVCC	10
1.3 User Scenarios	11
1.4 Comparison with Greenplum Features	13
General features	13
Performance-related features	14
Security-related features	14
1.5 Releases	15
v4.0.0 Release Notes	15
New features	16
Upgrade path	38
Product change information	38
Bug fixes	45
2 Deployment Guides	59
2.1 Deploy on Physical Machines	59

Deploy Manually	59
Software and Hardware Configuration	59
Prepare to Deploy on Physical Machine	64
Deploy on Multiple Nodes	75
Deploy on Single Node	84
3 Load Data	97
3.1 Data Loading Overview	97
Data loading process	97
Loading tools and scenarios	98
3.2 Load Data from Local Files	99
Load Data Using COPY	99
Load from a file	99
Load from STDIN	99
Load data using \copy in psql	100
Input format	100
Load Data Using gpfdist	101
Step 1. Install gpfdist	101
Step 2. Start and stop gpfdist	102
Step 3. Use gpfdist with external tables to load data	103
About gpfdist	105
Load Data Using the file:// Protocol	107
Usage examples	107
Load Data Using gupload	109
To use gupload	109
3.3 Load External Data Using Foreign Table	111
Use foreign table	111
Create foreign table using the LIKE clause	112
Query a foreign table	113
3.4 Load Data from Web Services	114
Command-based external web tables	114
URL-based external web tables	115
3.5 Load Data from Kafka Using Kafka FDW	116
Basic usage	116
Supported data formats	117
Query	117

Message producer	117
Data import	118
Scheduled import	120
3.6 Load Data from Kafka Using Kafka Connector	121
Data Formats	121
Installation	124
Preparation	124
Installation Steps	125
Parameter Descriptions	127
Troubleshooting	133
References	133
3.7 Load Data from Object Storage and HDFS	134
Install the extension	134
Instructions	134
Load data from object storage	135
Load data from HDFS	137
3.8 Load Data from Hive Data Warehouse	146
Supported Hive file formats	146
Usage limitations	146
Steps	147
Step 1. Create a configuration file on database cluster	147
Step 2. Create foreign data wrapper and Hive Connector plugin	151
Step 3. Create server and user mapping	152
Step 4. Sync Hive objects to the database cluster	153
Examples of syncing tables	156
Sync a Hive text table	156
Sync a Hive ORC table	157
Sync a Hive ORC partitioned table	158
Sync a sample Hive database	160
Sync tables in Iceberg and Hudi formats	160
Data type mapping	163
Known issues	163
Solution	163
3.9 Load Data from MySQL Server Using MySQL_FDW	166
Prerequisites	166
Use MySQL FDW	167

CREATE SERVER options	167
CREATE USER MAPPING options	168
CREATE FOREIGN TABLE options	168
IMPORT FOREIGN SCHEMA options	168
TRUNCATE support	169
Functions	169
Usage Examples	169
Installing the Extension	169
Creating a Foreign Server	170
Granting Permissions to Use Foreign Server	170
Creating a User Mapping	170
Creating a Foreign Table	171
Importing a MySQL Database as a SynxDB Schema	172
3.10 Custom Multi-Character Delimiters for Reading and Writing External Tables	173
Compilation and installation	173
Usage example	173
Read external tables	173
Write to external tables	174
4 Operate with Data	176
4.1 Operate with Database Objects	176
Create and Manage Views	176
Create views	176
Drop views	177
Best practices when creating views	177
Create and Manage Materialized Views	179
Create materialized views	180
Refresh or deactivate materialized views	180
Drop materialized views	181
Create and Manage indexes	182
Index types	183
Manage indexes	184
Index-only scan and covering index	186
Backward index scan	190
Check index usage	191
BRIN Indexes	193

Maintain BRIN indexes	194
Choose Table Storage Model	195
Heap and Append-optimized Table Storage Models	195
PAX Storage Format	199
UnionStore Storage Format	217
Cross-Cluster Federated Query	223
User scenarios	223
Usages	224
Use Tags to Manage Database Objects	232
What is a tag?	232
Features of tags	232
Usage scenarios	232
Use tags	233
System tables related to tags	239
Common errors and tips	241
4.2 SQL Queries	242
Join Queries	242
Join types	243
Join conditions	245
LATERAL	246
Example	246
4.3 Advanced Analytics	250
Use pgvector for Vector Similarity Search	250
Quick start	250
Store data	251
Query data	251
Index data	252
Hybrid search	262
pgvector performance	262
Vectorization Query Computing	264
Enable vectorization	264
Usage	264
Performance evaluation	269
Use PostGIS for Geospatial Data Analysis	273
Create the PostGIS extension	273
Enable GDAL raster drivers	274

Enable out-of-database raster feature	275
Remove PostGIS support	276
Usage examples	277
PostGIS support and limitations	279
Use MADlib for Machine Learning and Deep Learning	281
Install MADlib components	281
Add MADlib functions to the database	282
Uninstall MADlib from the database	282
Usage examples	283
Directory Tables	294
Usage	294
4.4 Use Oracle Compatibility SQL via Orafce	306
Install and remove Orafce	306
SynxDB considerations	307
SynxDB implementation differences	307
Using Orafce	309
5 Optimize performance	310
5.1 Optimize Query Performance	310
Query Processing Overview	310
Query planning and dispatch	310
Query plans	312
Parallel query execution	314
Query Performance Overview	315
Dynamic partition elimination	315
Memory optimization	316
Update Statistics	317
Checking if statistics are up to date	317
Generate statistics selectively	317
Improve statistics quality	318
When to run ANALYZE	318
Configure automatic statistics collection	319
Query Plan Hints	320
Quick example	320
Cardinality hints	322
Table access hints	322

Join type hints	323
Join order hints	323
Supported scope and limitations	324
Best practices for using query plan hints	324
Use GPORCA Optimizer	325
GPORCA overview	325
GPORCA Features and Enhancements	331
GPORCA Optimizer Update Notes	337
Use Automatic Materialized Views for Query Optimization	342
Usage scenarios	342
Implementation	342
Comparison with dynamic tables	343
Restrictions	343
Usage examples	343
Use materialized views to query external tables	346
Aggregate query support	349
5.2 Manage Resources Using Resource Groups	353
Role and component resource groups	353
Resource group attributes and limits	354
Transaction concurrency limit	355
Bypass limits and unassign from resource groups	355
CPU limits	356
Memory limits	360
Disk I/O limits	361
Configure and use resource groups	362
Prerequisites	362
Enable resource groups	367
Create resource groups	368
Configure automatic query termination based on memory usage	370
Assign a resource group to a role	370
Monitor resource group status	371
View resource group limits	371
View resource group query status	371
View resource group memory usage per host	371
View the resource group assigned to a role	372
View resource group disk I/O usage per host	372

View a resource group's running and pending queries	372
Cancel a running or queued transaction in a resource group	373
Move a query to a different resource group	374
Frequently asked questions	375
5.3 Use Dynamic Tables to Speed Up Queries and Auto-Refresh Data	378
Use cases	378
Comparison with materialized views	378
Usage	379
Create a dynamic table	379
Refresh a dynamic table	380
View schedule information	380
Drop a dynamic table	380
View distribution key	381
Examples	382
Example 1: Accelerate external table queries in lake-house architecture	382
Example 2: Create an empty dynamic table	384
Notes	385
6 AI & ML (HashML)	386
6.1 Deploy HashML Platform	386
Steps to deploy HashML Platform	386
Preparation	386
Step 1: Install the RPM packages	387
Step 2: Configure HashML Platform	387
Step 3: Start the HashML Platform service	389
Step 4: Access the HashML Platform console in your browser	389
Manage HashML Platform services	390
Additional notes	391
6.2 Use the HashML Platform	392
Core features of HashML Platform	392
Use cases for HashML Platform	392
Use the HashML Platform console	393
Log into the HashML console	393
Train a model	394
View training tasks	396
Data exploration and insights	397

Model factory	398
Use notebook-based modeling	402
7 Manage System	404
7.1 View Monitoring Data Using the Web Console	404
Install Monitoring Console	405
Prerequisites	405
Step 1: Deploy the Server component	406
Step 2: Deploy the Agent on each node	409
Next step	410
Troubleshooting	410
View Cluster Information	412
Steps	412
Cluster Status and Metrics	414
Access the pages	414
View the overall status and data of the cluster	415
View the status and data of nodes and hosts	417
View Database Object Information	418
View table objects	419
View SQL Monitoring Information	420
Access the page	420
View SQL execution status	421
View session status	425
View Storage Information	426
Steps	426
View and Create Alert Rules	427
Access the alert page	427
View existing alert rules	428
Create an alert rule	428
View and Create Contact Groups	431
Access the contact group page	431
View existing contact groups	432
Create a contact group	432
7.2 Configure Security and Permission	433
Manage Roles and Privileges in SynxDB	433
Create new roles (users)	433

Role membership	435
Manage object privileges	436
Security best practices for roles and privileges	437
Encrypt data	438
7.3 Backup and Restore	439
Backup and Restore Overview	439
Parallel backup with <code>gpbackup</code> and <code>gprestore</code>	439
Non-parallel backup with <code>pg_dump</code>	442
Backup and recovery with CBDR at the WAL level	443
Perform Full Backup and Restore	445
Back up the full database	445
Restore the full database	448
Filter the contents of a backup or restore	450
Check report files	455
Configure email notifications	457
Perform Incremental Backup and Restore	461
About incremental backup sets	461
Use incremental backups	462
Incremental backup notes	467
CBDR	469
Full backup and restore procedure	469
Incremental backup and restore procedure	472
Restore procedure using a restore point	474
Configuration file reference	475
Command usage	476
7.4 Configure Database System	480
Coordinator and local parameters	480
Set configuration parameters	480
Set a local configuration parameter	481
Set a coordinator configuration parameter	481
View server configuration parameter settings	483
View parameters using the <code>pg_settings</code> view	483
8 Reference Guides	486
8.1 System Utilities	486
<code>gpdemo</code>	486

How to use	487
Command-line options	490
8.2 System Catalogs	491
System Views	491
pg_stat_progress_create_index	491
The gp_toolkit Administrative Schema	493
About the extension	494
Upgrade the extension	494
Check for tables that need routine maintenance	495
Check for locks	496
Check append-optimized tables	498
View server log files	506
Check server configuration files	510
Check for failed segments	511
Check resource group activity and status	512
Check resource queue activity and status	516
Check query disk spill space usage	519
View users and groups (roles)	521
Check database object sizes and disk space	522
Check for missing and orphaned data files	527
Move orphaned data files	529
Check for uneven data distribution	530
Maintain partitions	532
8.3 Configuration Parameters	534
autovacuum_freeze_max_age	534
autovacuum_vacuum_cost_delay	534
autovacuum_vacuum_scale_factor	535
autovacuum_vacuum_threshold	535
checkpoint_timeout	535
gp_appendonly_compaction_segfile_limit	536
gp_autostats_lock_wait	536
gp_command_count	536
gp_dynamic_partition_pruning	537
gp_enable_runtime_filter_pushdown	537
gp_enable_statement_trigger	537
gp_max_partition_level	537

gp_resource_manager	538
gp_role	538
gp_session_id	539
krb_server_keyfile	539
log_checkpoints	539
max_connections	540
max_replication_slots	540
optimizer_array_constraints	540
optimizer_array_expansion_threshold	541
optimizer_cost_model	541
optimizer_cost_threshold	542
optimizer_cteInlining_bound	542
optimizer_damping_factor_filter	542
optimizer_damping_factor_groupby	543
optimizer_damping_factor_join	543
optimizer_discard_redistribute_hashjoin	543
optimizer_dpe_stats	544
optimizer_enable_derive_stats_all_groups	544
optimizer_enable_dynamicbitmapscan	544
optimizer_enable_dynamicindexonlyscan	545
optimizer_enable_dynamicindexscan	545
optimizer_enable_foreign_table	545
optimizer_enable_indexonlyscan	546
optimizer_enable_ordereddagg	546
optimizer_enable_push_join_below_union_all	547
optimizer_enable_query_parameter	548
optimizer_enable_right_outer_join	548
optimizer_force_three_stage_scalar_dqa	548
optimizer_nestloop_factor	549
optimizer_penalize_broadcast_threshold	549
optimizer_push_group_by_below_setop_threshold	550
optimizer_replicated_table_insert	550
optimizer_skew_factor	550
optimizer_sort_factor	551
optimizer_trace_fallback	551
optimizer_use_gpdb_allocators	551

optimizer_xform_bind_threshold	552
superuser_reserved_connections	552
track_io_timing	552
wal_compression	553
wal_keep_size	553
work_mem	553
writable_external_table_bufsize	554
9 Developer Guides	555
9.1 Develop Database Extensions Using PGRX	555
Requirements for development environment	555
Basic software environment	556
PostgreSQL dependencies	556
Quick start for PGRX	556
Set up and install PGRX	557
Create an extension	557
Install and use the extension	559
PGRX type mapping	559
Custom type conversions	560
Type mapping details	561
PGRX core features	561
Complete management for development environment	561
Automatic mode generation	561
Security first	562
UDF supports	562
Simple custom types	562
Server programming interface (SPI)	563
Advanced features	563
Considerations and best practices for PGRX	563
Debugging and development tips	564
Learning resources for PGRX	564

Chapter 1

Product Overview

1.1 Feature Overview

SynxDB is one of the most advanced and mature open-source MPP databases available. Built on the latest PostgreSQL 14.4 kernel, it comes with multiple features, including high concurrency and high availability. It can perform quick and efficient computing for complex tasks, meeting the demands of managing and computing vast amounts of data. It is widely applied in multiple fields.

This document gives a general introduction to the features of SynxDB.

Efficient queries in different scenarios

- SynxDB allows you to perform efficient queries in big data analysis environments and distributed environments:
 - **Big data analysis environment:** SynxDB uses the built-in PostgreSQL optimizer, which offers better support for distributed environments. This means that it can generate more efficient query plans when handling big data analysis tasks.
 - **Distributed environment:** Built in with the specially-adapted open-source GPORCA optimizer, SynxDB meets the query optimization needs in distributed environments.
- Multiple technologies are used such as static and dynamic partition pruning, aggregate

push-down, and join filtering to help you get the fastest and most accurate query results possible.

- Both rule-based and cost-based query optimization methods are provided to help you generate more efficient query execution plans.

Polymorphic data storage

For different scenarios, SynxDB supports multiple storage formats, including Heap storage, AO row storage, and AOCS column storage. SynxDB also supports partitioned tables. You can define the partitioning of a table based on certain conditions. When executing a query, it automatically filters out the sub-tables that are not needed for the query to improve query efficiency.

- **Even data distribution:** By using Hash and Random methods for data distribution, SynxDB takes better advantage of disk performance and solves I/O bottleneck issues.
- **Storage types:**
 - Row-based storage: Suitable for scenarios where most fields are frequently queried, and there are many random row accesses.
 - Column-based storage: When you need to query a small number of fields, this method can greatly save I/O operations, making it ideal for scenarios where large amounts of data are accessed frequently.
- **Specialized storage modes:** SynxDB has different storage modes such as Heap storage, AO row storage, AOCS column storage to optimize the performance of different types of applications. At the finest granularity level of partitioning, a table can have multiple storage modes.
- **Support for partitioned tables:** You can define the partitioning of a table based on specific conditions. During querying, the system will automatically filter out the sub-tables that are not needed for the query to improve query efficiency.
- **Efficient data compression function:** SynxDB supports multiple compression algorithms, such as Zlib 1-9 and Zstandard 1~19, to improve data processing performance and maintain a balance between CPU and compression ratio.
- **Optimization for small tables:** You can choose to use the Replication Table and specify a custom Hash algorithm when creating the table, allowing for more flexible control of data

distribution.

Multi-layer data security

SynxDB enhances user data protection by supporting function encryption and transparent data encryption (TDE). TDE means that the SynxDB kernel performs these processes invisibly to users. The data formats subject to TDE include Heap tables, AO row storage, and AOCS column storage. In addition to common encryption algorithms like AES, SynxDB also supports national secret algorithms, allowing seamless integration of your own algorithms into TDE process.

SynxDB focuses on data security and provides security protection measures. These security measures are designed to satisfy different database environment needs and offer multi-layer security protection:

- **Database isolation:** In SynxDB, data is not shared between databases, which achieves isolation in a multi-database environment. If cross-database access is required, you can use the DBLink feature.
- **Internal data organization:** The logical organization of data in the database includes data objects such as tables, views, indexes, and functions. Data access can be performed across schemas.
- **Data storage security:** SynxDB offers different storage modes to support data redundancy. It uses encryption methods including AES 128, AES 192, AES 256, DES, and national secret encryption to secure data storage. It also supports ciphertext authentication, which includes encryption algorithms like SCRAM-SHA-256, MD5, LDAP, RADIUS.
- **User data protection:** SynxDB supports function encryption and decryption, and transparent data encryption and decryption. The process is implemented by the SynxDB kernel without any user interaction. It supports data formats such as Heap tables, AO row storage, and AOCS column storage. In addition to common encryption algorithms like AES, SynxDB also supports national secret algorithms, allowing you to easily add your own algorithms into transparent data encryption.
- **Detailed permission settings:** To satisfy different users and objects (like schemas, tables, rows, columns, views, functions), SynxDB provides a range of permission setting options, including SELECT, UPDATE, execution, and ownership.

Data loading

SynxDB provides a series of efficient and flexible data loading solutions to meet various data processing needs, including parallel and persistent data loading, support for flexible data sources and file formats, integration of multiple ETL tools, and support for stream data loading and high-performance data access.

- **Parallel and persistent data loading:** SynxDB supports massive parallel and persistent data loading through external table technology, and performs automatic conversion between character sets, such as from GBK to UTF-8. This feature makes data entry much smoother.
- **Flexible data source and file format support:** SynxDB supports data sources such as external file servers, Hive, Hbase, HDFS or S3, and supports data formats such as CSV, Text, JSON, ORC, and Parquet. In addition, the database can also load compressed data files such as Zip.
- **Integrate multiple ETL tools:** SynxDB is integrated with ETL tools such as DataStage, Informatica, and Kettle to facilitate data processing.
- **Support stream data loading:** SynxDB can start multiple parallel read tasks for the subscribed Kafka topic, cache the read records, and load the records into the database via gpfdist after a certain time or number of records. This method can ensure the integrity of data without duplication or loss, and is suitable for stream data collection and real-time analysis scenarios. SynxDB supports data loading throughput of tens of millions per minute.
- **High-performance data access:** PXF is a built-in component of SynxDB, which can map external data sources to external tables of SynxDB to achieve parallel and high-speed data access. PXF supports the management and access of hybrid data ecology and helps realize the Data Fabric architecture.

Multi-layer fault tolerance

To ensure data security and service continuity, SynxDB adopts a multi-level fault-tolerant mechanism of data pages, checksum, mirror node configuration, and control node backup.

- **Checksum of data page:** In the underlying storage, SynxDB uses the checksum mechanism to detect bad blocks to ensure data integrity.
- **Mirror node configuration:** By configuring mirror nodes among segments (or data nodes),

SynxDB can achieve high availability and failover of services. Once an unrecoverable failure of the coordinator node is detected, the system will automatically switch to the backup segment to ensure that user queries will not be affected.

- **Backup of control nodes:** Similar to segments, coordinator nodes (or control nodes) can also be configured as backup nodes or standby nodes in case the coordinator node fails. Once the coordinator node fails, the system will automatically switch to the standby node to ensure the continuity of services.

Rich data analysis support

SynxDB provides powerful data analysis features. These features make data processing, query and analysis more efficient, and meets multiple complex data processing, analysis and query requirements.

- **Parallel optimizer and executor:** The SynxDB kernel has a built-in parallel optimizer and executor, which is not only compatible with the PostgreSQL ecosystem, but also supports data partition pruning and multiple indexing technologies (including B-Tree, Bitmap, Hash, Brin, GIN), and JIT (expression just-in-time compilation processing).
- **Machine learning components MADlib:** SynxDB integrates MADlib components, providing users with fully SQL-driven machine learning features, enabling deep integration of algorithms, computing power, and data.
- **Support multiple programming languages:** SynxDB provides developers with rich programming languages, including R, Python, Perl, Java, and PostgreSQL, so that they can easily write custom functions.
- **High-performance parallel computing based on MPP engine:** The MPP engine of SynxDB supports high-performance parallel computing, seamlessly integrated with SQL, and can perform fast computing and analysis on SQL execution results.
- **PostGIS geographic data processing:** SynxDB introduces an upgraded version of PostGIS 2.X, supports its MPP architecture, and further improves the processing capability of geospatial data. Key features include:
 - Support for object storage: supports directly loading large-capacity geospatial data from object storage (OSS) into the database.

- Comprehensive spatial data type support: including geometry, geography, and raster.
- Spatio-temporal index: Provides spatio-temporal index technology, which can effectively accelerate spatial and temporal queries.
- Complex spatial and geographic calculations: including sphere length calculations as well as spatial aggregation functions (such as contain, cover, intersect).
- **[product_name] text component:** This component supports using ElasticSearch to accelerate file retrieval capabilities. Compared with traditional GIN data text query performance, this component has an order of magnitude improvement. It supports multiple word segmentation, natural language processing, and query result rendering.

Flexible workload management

SynxDB provides comprehensive workload management capabilities designed to effectively utilize and optimize database resources to ensure efficient and stable operations. Its workload management includes three levels of control: connection level management, session level management, and SQL level management.

- **Connection pool PGBouncer (connection-level management):** Through the connection pool, SynxDB manages user access in a unified manner, and limits the number of concurrently active users to improve efficiency, and avoid wasting resources caused by frequently creating and destroying service processes. The connection pool has a small memory footprint and can support high concurrent connections, using libevent for Socket communication to improve communication efficiency.
- **Resource Group (session-level management):** Through resource groups, SynxDB can analyze and categorize typical workloads, and quantify the CPU, memory, concurrency and other resources required by each workload. In this way, according to the actual requirements of the workload, you can set a suitable resource group and dynamically adjust the resource usage to ensure the overall operating efficiency. At the same time, you can use rules to clean up idle sessions and release unnecessary resources.
- **Dynamic resource group allocation (SQL-level management):** Through dynamic resource group allocation, SynxDB can flexibly allocate resources before or during the execution of SQL statements, which can give priority to specific queries and shorten the execution time.

Multiple compatibility

The compatibility of SynxDB is reflected in multiple aspects such as SQL syntax, components, tools and programs, hardware platforms and operating systems. This makes the database flexible enough to deal with different tools, platforms and languages.

- **SQL compatibility:** SynxDB is compatible with PostgreSQL and Greenplum syntax, supports SQL-92, SQL-99, and SQL 2003 standards, including SQL 2003 OLAP extensions, such as window functions, `rollup`, and `cube`.
- **Component compatibility:** Based on the PostgreSQL 14.4 kernel, SynxDB is compatible with most of the PostgreSQL components and extensions commonly used.
- **Tool and program compatibility:** Good connectivity with various BI tools, mining forecasting tools, ETL tools, and J2EE/.NET applications.
- **Hardware platform compatibility:** Can run on a variety of hardware architectures, including X86, ARM, Phytium, Kunpeng, and Haiguang.
- **Operating system compatibility:** Compatible with multiple operating system environments, such as CentOS, Ubuntu, Kylin, and BC-Linux.

1.2 Product Architecture

This document introduces the product architecture and the implementation mechanism of the internal modules in SynxDB.

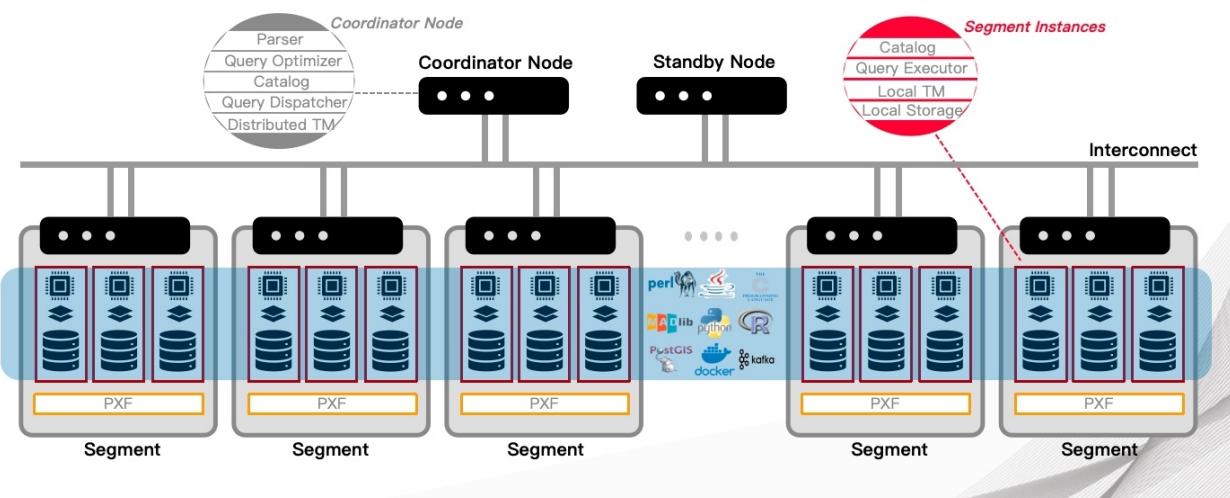
In most cases, SynxDB is similar to PostgreSQL in terms of SQL support, features, configuration options, and user functionalities. Users can interact with SynxDB in a similar way to how they interact with a standalone PostgreSQL system.

SynxDB uses MPP (Massively Parallel Processing) architecture to store and process large volumes of data, by distributing data and computing workloads across multiple servers or hosts.

MPP, known as the shared-nothing architecture, refers to systems with multiple hosts that work together to perform a task. Each host has its own processor, memory, disk, network resources, and operating system. SynxDB uses this high-performance architecture to distribute data loads and can use all system resources in parallel to process queries.

From users' view, SynxDB is a complete relational database management system (RDBMS). In a physical view, it contains multiple PostgreSQL instances. To make these independent PostgreSQL instances work together, SynxDB performs distributed cluster processing at different levels for data storage, computing, communication, and management. SynxDB hides the complex details of the distributed system, giving users a single logical database view. This greatly eases the work of developers and operational staff.

The architecture diagram of SynxDB is as follows:



Coordinator node

Coordinator node (or control node) is the gateway to the SynxDB system, which accepts client connections and SQL queries, and allocates tasks to data node instances. Users interact with SynxDB by connecting to the coordinator node using a client program (such as psql) or an application programming interface (API) (such as JDBC, ODBC, or libpq PostgreSQL C API).

- The coordinator node acts as the global system directory, containing a set of system tables that record the metadata of SynxDB.
- The coordinator node does not store user data. User data is stored only in data node instances.
- The coordinator node performs authentication for client connections, processes SQL commands, distributes workload among segments, coordinates the results returned by each segment, and returns the final results to the client program.
- SynxDB uses Write Ahead Logging (WAL) for coordinator/standby mirroring. In WAL-based logging, all modifications are first written to a log before being written to the disk, which ensures the data integrity of in-process operations.

Segment node

Segment (or data node) instances are individual Postgres processes, each storing a portion of the data and executing the corresponding part of the query. When a user connects to the database through the coordinator node and submits a query request, a process is created on each segment node to handle the query. User-defined tables and their indexes are distributed across the available segments, and each segment node contains distinct portions of the data. The processes of data processing runs in the corresponding segment. Users interact with segments through the coordinator, and the segment operate on servers known as the segment host.

Typically, a segment host runs 2 to 8 data nodes, depending on the processor, memory, storage, network interface, and workload. The configuration of the segment host needs to be balanced, because evenly distributing the data and workload among segments is the key to achieving optimal performance with SynxDB, which allows all segments to start processing a task and finish the work at the same time.

Interconnect

Interconnect is the network layer in the SynxDB system architecture. Interconnect refers to the network infrastructure upon which the communication between the coordinator node and the segments relies, which uses a standard Ethernet switching structure.

For performance reasons, a 10 GB or faster network is recommended. By default, the Interconnect module uses the UDP protocol with flow control (UDPIFC) for communication to send messages through the network. The data packet verification performed by SynxDB exceeds the scope provided by UDP, which means that its reliability is equivalent to using the TCP protocol, and its performance and scalability surpass the TCP protocol. If the Interconnect is changed to the TCP protocol instead, the scalability of SynxDB is limited to 1000 segments. This limit does not apply when UDPIFC is used as the default protocol.

MVCC

SynxDB uses Multiversion Concurrency Control (MVCC) to ensure data consistency. When querying the database, each transaction only sees a snapshot of the data, ensuring that current transactions do not see modifications made by other transactions on the same records. In this way, MVCC provides transaction isolation in the database.

MVCC minimizes lock contention to ensure performance in a multi-user environment. This is done by avoiding explicit locking for database transactions.

In concurrency control, MVCC does not introduce conflicts for query (read) locks and write locks. In addition, read and write operations do not block each other. This is the biggest advantages of MVCC over the lock mechanism.

1.3 User Scenarios

This document introduces the use cases of SynxDB.

Scenario 1: Batch processing data warehouse offline and building data marts

- Builds high-performance SynxDB warehouses and data marts for storing and querying large-scale datasets. This includes Operational Data Store (ODS), Data Warehouse Detail (DWD), and Data Warehouse Summary (DWS). Supports building source model, normalization model, dimension tables, fact tables, and more, with multiple ways to load source data into the data warehouse.
- Supports multiple types of data processing.
- Supports building data warehouse and data marts with high concurrency, high performance, and low maintenance cost.
- Supports complex data analysis and query needs, including data aggregation, multi-dimensional analysis, and correlated queries.

Scenario 2: Building data warehouse in real-time

- Supports building data warehouse in real-time, and supports collecting and processing streaming data to make real-time data analysis possible.

Scenario 3: Building mid-end

- Supports building MPP data platform in the data mid-end. Supports the distributed parallel processing architecture.
- Supports building data warehouse in the data mid-end. Supports docking with mainstream ETL tools.

Scenario 4: Building lake-warehouse integration

- Supports building enterprise-level data lake-warehouse integration. Supports efficient data exchange between data lake and data warehouse.

Scenario 5: Alternative to existing MPP databases

- Supports replacing common databases, such as Oracle, TeraData, Greenplum, and Vertical.

- Supports replacing other types of MPP databases, such as Gbase 8a, and GaussDB.

Scenario 6: Applicable to Geographic Information System (GIS) applications

- Builds Geographic Information System (GIS) applications on SynxDB.
- Stores and queries geographic location data. Supports spatial data analysis, geocoding, and map visualization.
- Can be applied to city planning, geographic analysis, and map navigation.

1.4 Comparison with Greenplum Features

SynxDB is 100% compatible with Greenplum, and provides all the Greenplum features you need.

In addition, SynxDB possesses some features that Greenplum currently lacks or does not support. More details are listed below.

General features

Note

- In the following tables, ✓ means support, and ✗ means no support.
- The feature comparison in the following tables is based on Greenplum 7 Beta.3.

Feature names	SynxDB	Greenplum
EXPLAIN (WAL) support	✓	✗
Multiranges	✓	✗
B-tree bottom-up index deletion	✓	✗
Covering indexes for GiST (INCLUDE)	✓	✓ (Upcoming)
The range_agg range type aggregation function	✓	✗
CREATE ACCESS METHOD	✓	✓ (Upcoming)
LZ4 compression for TOAST tables	✓	✗
JSONB subscripting	✓	✗
Configure the maximum WAL retention for replication slots	✓	✗
Verify backup integrity (pg_verifybackup)	✓	✗
Clients can require SCRAM channel binding	✓	✗
Vacuum “emergency mode”	✓	✗
Certificate authentication with postgres_fdw	✓	✗
UPSERT	✓	✓ (Upcoming)
COPY FROM WHERE	✓	✗
VACUUM / ANALYZE Skip Lock Table	✓	✗
HASH partitioned table	✓	✗
CTE (SEARCH and CYCLE)	✓	✗
Procedure OUT parameters	✓	✗
CHECK constraints for foreign tables	✓	✗
Timeout parameter for pg_terminate_backend	✓	✗
Auto failover for coordinator	✓	✗
Kubernetes deployment support	✓	✗

Performance-related features

Feature names	SynxDB	Greenplum
REINDEX CONCURRENTLY	✓	✗
Aggregation pushdown	✓	✗
CREATE STATISTICS - OR and IN/ANY statistics	✓	✗
Incremental sort	✓	✗
Incremental sort for window functions	✓	✗
Query pipelining	✓	✗
BRIN Index (multi-minmax, bloom)	✓	✗
Query parallelism	✓	✗
Abbreviated keys for sorting	✓	✗
Hash Index WAL support	✓	✗
postgres_fdw aggregation pushdown	✓	✗
No need to rewrite the whole table when adding a column	✓	✗
Runtime Filter for Join	✓	✗
Index Scan for the AppendOnly table	✓	✗

Security-related features

Feature names	SynxDB	Greenplum
Transparent Data Encryption (TDE)	✓	✗
Trusted extensions	✓	✗
SCRAM-SHA-256	✓	✗
Encrypted TCP/IP connection when GSSAPI	✓	✗
Row-level security policy	✓	✗

1.5 Releases

v4.0.0 Release Notes

Release date: July, 2025

Version number: v4.0.0

SynxDB v4.0.0 is a major release that brings significant enhancements to the database kernel and introduces a new monitoring tool, CBCC and a new backup and recovery tool, CBDR. This release represents a substantial leap forward in performance, reliability, and manageability.

- **Enhanced Query Processing:** Major improvements in query optimization and execution, including advanced index-only scans, dynamic partition elimination, and improved join strategies. The ORCA optimizer now supports more complex query patterns and provides better performance for analytical workloads.
- **New Monitoring Solution:** Introduction of CBCC, a modern web-based monitoring and operations dashboard that replaces the legacy Web Platform. CBCC provides comprehensive cluster monitoring, resource tracking, and alert management capabilities.
- **Improved Storage Engine:** Significant enhancements to AO/CO tables, including optimized index creation, better transaction safety, and improved memory management. The storage layer now offers better performance and reliability for large-scale data processing.
- **Enhanced Security:** Added FIPS mode support in pgcrypto, improved permission management, and strengthened security controls across the database system.
- **Resource Management:** Refined resource group management with clearer parameter naming and improved monitoring capabilities. The system now provides better control over CPU, memory, and I/O resources.
- **High Availability and High Reliability:** Introduction of CBDR, a backup and recovery tool built on WAL-G, offering flexible backup strategies and improved disaster recovery capabilities.

This release also includes numerous bug fixes and stability improvements.

New features

- *Database*
- *Interactive manager CBCC*
- *Data intelligence platform*

Database

Category	Sub-category	Feature and documents
Query processing and optimization	Index and scan: introduces multiple enhancements in indexing and scanning, including: <ul style="list-style-type: none"> • Enhanced index-only scan capabilities: supports dynamic index-only scans and index-only scans on AO/PAX tables; expands support to indexes with <code>INCLUDE</code> columns. • Improved index scan performance and flexibility: adds support for backward index scans, bitmap scans with array comparisons, and more accurate cost estimation for covering indexes. • BRIN index enhancements: optimizes internal structure for AO/CO tables and improves summarization control and robustness. 	<i>Index and scan</i>
Query processing and optimization	View and materialized view: improves performance of <code>REFRESH MATERIALIZED VIEW WITH NO DATA</code> by avoiding full query execution.	<i>View and materialized view</i>
Query processing and optimization	Join: supports left join pruning when the inner table is uniquely constrained and unused, enabling more efficient plans. <code>FULL JOIN</code> now uses <code>Hash Full Join</code> by default to avoid sorting and reduce data movement. For symmetric multi-way outer self joins, unnecessary data redistribution is eliminated. Broadcast plans for <code>NOT IN</code> queries are no longer penalized, improving parallelism and reducing memory pressure.	<i>Join</i>
Query processing and optimization	Function and aggregate: adds support for intermediate aggregates to optimize queries with both <code>DISTINCT</code> and regular aggregates, and introduces index-based plans for <code>MIN()</code> and <code>MAX()</code> using <code>LIMIT</code> , even with <code>IS NULL</code> conditions. Additional <code>HashAggregate</code> plan alternatives improve support for <code>DISTINCT</code> aggregates through two-stage planning. Support for <code>GROUP BY CUBE</code> enables multi-dimensional grouping, expanding analytic capabilities.	<i>Function and aggregate</i>

continues on next page

Table 1 – continued from previous page

Category	Sub-category	Feature and documents
Query processing and optimization	Preprocessing: inlines Common Table Expressions (CTEs) with outer references, allowing broader optimization coverage and avoiding fallback to the legacy planner. IN-to-EXISTS rewrites are now skipped when set-returning functions are present in the subquery, ensuring correctness and preventing execution errors.	<i>Preprocessing</i>
Query processing and optimization	Optimization and performance enhancements: <ul style="list-style-type: none"> Plan hint: adds support for scan type hints and join row estimates using pg_hint_plan-style comments, allowing users to influence query planning more precisely. Join order hints are now extended to include left and right outer joins. Additionally, the plan hint field is now mandatory in the ORCA configuration, ensuring consistent parsing and behavior. Enhancements to ORCA: improve compatibility, performance, and usability across a wide range of query scenarios. Support is added for table aliases, query parameters, row-level security, and set-returning functions, reducing fallback to the legacy planner. Predicate pushdown, hashed subplans, and direct dispatch for random tables improve execution efficiency. Planning is further optimized for foreign tables, UNION ALL queries, and composite types, while EXPLAIN and log output are updated for greater clarity. 	<i>Optimization and performance enhancements</i>
Transaction management	Lock management: updates logic to ignore invalidated slots during catalog Xmin computation, performs early serializable isolation checks on AO/CO tables to improve consistency, and enhances index creation by synchronizing lock acquisition across coordinator and segments to prevent deadlocks. These changes improve transaction reliability, concurrency, and system stability.	<i>Lock management</i>
Transaction management	Transaction performance and reliability improvements: avoid replaying DTX information on newly added segments to ensure consistent recovery, enhance observability with gp_stat_progress_dtx_recovery, and improve error reporting for DTX dispatch issues. The update also extends the default retry period for phase 2 transactions to 600 seconds and allows utility mode to skip unnecessary lock upgrades, improving efficiency during maintenance.	<i>Transaction performance and reliability</i>
Storage	AO/CO: optimizes CREATE INDEX operations on AO tables by adding scan progress reporting for better efficiency, and improves transaction safety by marking connection-related variables as volatile, aligning with PostgreSQL best practices for exception handling.	<i>AO/CO table enhancements</i>

continues on next page

Table 1 – continued from previous page

Category	Sub-category	Feature and documents
Storage	Partitioning: extends ORCA's planning to support foreign partitions with full integration of static and dynamic pruning, improves partition analysis efficiency, and enables advanced dynamic partition elimination in complex join scenarios. The update also introduces opfamily-aware pruning, caching for repeated lookups, and better cardinality estimation from leaf partitions. In addition, the new version enhances the DPv2 algorithm with distribution specs and introduces a new Non-Replicated distribution type to reduce unnecessary data motion in join operations.	Partitioning
Storage	Memory management: implements a custom allocator for ORCA to improve heap handling and enable the use of standard C++ containers, optimizes lazy serialization of metadata objects to speed up planning, and ensures proper deallocation of memory to prevent leaks. In addition, MPP support for pg_buffercache improves scalability, and the new pg_buffercache_summary() function provides simplified monitoring of buffer cache activity.	Memory management
Storage	Memory and metadata: adds support for defining lock modes and relocations for storage transitions, introduces origin tracking for partitioned table creation, and improves catalog access with syscache lookups. A new pg_aggregate field supports marking aggregates as replication-safe, optimizing execution on replicated data. In addition, ALTER DATABASE options are now properly dispatched to segments to ensure consistent catalog state.	Memory and metadata
Data loading	External table: improves consistency and usability by disabling SET DISTRIBUTED REPLICATED for ALTER EXTERNAL TABLE and enforcing stricter rules when exchanging or attaching external tables. Writable external tables can no longer be used as partitions, and attaching readable external tables now issues a warning when done without validation.	External table enhancements
Data loading	Foreign data wrapper: improves gpfdist external table performance by enabling TCP keepalive and increasing default write buffer size. ORCA now gracefully falls back to the planner for queries involving foreign partitions with greenplum_fdw, ensuring stability, while maintaining support for non-partitioned foreign tables.	Foreign data wrapper enhancements
High availability and high reliability	Backup and disaster recovery: introduces CBDR, a backup and recovery tool for SynxDB and Apache Cloudberry databases, offering a user-friendly CLI, full and incremental backups, support for multiple compression formats, and backup encryption, enhancing disaster recovery and data safety.	Backup and disaster recovery
High availability and high reliability	WAL: restricts coordinator-specific WAL tracking to simplify primary segment behavior, improves WAL retention logic for reliable incremental recovery using pg_rewind, and switches WAL replication connections to the standard libpq protocol for enhanced compatibility.	WAL

continues on next page

Table 1 – continued from previous page

Category	Sub-category	Feature and documents
Security	Database operations: prevents potential privilege escalation by running all internal operations in the correct security context for REFRESH MATERIALIZED VIEW CONCURRENTLY and aligns tuple freezing behavior for new aoseg and aocsseg tuples with other catalog operations to enhance consistency.	<i>Database operations</i>
Security	System processes: excludes idle sessions from orphaned file checks, adds safety checks in signal handlers, prevents child processes from calling <code>proc_exit()</code> , and removes unnecessary permission checks, enhancing process safety and stability.	<i>System processes</i>
Security	Replication/Mirrorless clusters: improves replication error reporting by setting persistent <code>WalSndError</code> when a replication slot is invalidated, ensuring accurate error visibility in <code>gp_stat_replication</code> .	<i>Replication/Mirrorless clusters</i>
Security	Permission management: strengthens security by rejecting unsafe characters in extension schema or owner substitutions, prevents SQL injection, reserves the <code>system_group</code> resource group for internal use, reverts the superuser requirement for setting <code>gp_resource_group_bypass</code> , and disallows altering the <code>mpp_execute</code> option to ensure plan correctness.	<i>Permission management</i>
Security	pgcrypto: adds FIPS mode support in pgcrypto, enabling SynxDB to operate in FIPS-compliant environments and allowing FIPS mode to be enabled regardless of pre-existing system settings.	<i>pgcrypto</i>
Resource management	Resource group management: renames the <code>memory_limit</code> parameter to <code>memory_quota</code> for clarity, adds a system view to monitor per-segment memory usage, improves logging for memory limit breaches, and removes an unnecessary permission check to enhance compatibility.	<i>Resource group management</i>
Resource management	Logging and monitoring: adds detailed log messages for GDD backends, reduces log noise for connection terminations, increases verbosity in deadlock checks, logs queue and portal IDs, dumps more information on resource lock releases, and enhances error reporting for dispatcher and connection liveness checks.	<i>Logging and monitoring</i>
Resource management	Platform compatibility and build: enhances compatibility and consistency by improving <code>gp_sparse_vector</code> handling on ARM platforms, aligning signal handling on Windows with other platforms, and ensuring consistent ACL mode type usage in ORCA.	<i>Platform compatibility and build</i>
Resource management	System views and statistics: improves join cardinality estimation, increases precision for frequency and NDV values, considers null value skew, supports extended statistics, logs memory contexts across segments, supports statistics derivation for time-related data types, uses SKIP LOCKED for ANALYZE operations, and supports STATS_EXT_NDISTINCT extended statistics.	<i>System views and statistics</i>
Resource management	Network connections: enhances reliability of internal QD-to-entry DB connections.	<i>Network connections</i>

continues on next page

Table 1 – continued from previous page

Category	Sub-category	Feature and documents
Tools and utilities	<p>Several tools are enhanced to improve performance and usage:</p> <ul style="list-style-type: none"> • analyzedb: analyzes materialized views to improve performance immediately after analysis. • gpexpand: performs cluster health checks to ensure all segments are up and in their preferred roles before proceeding, preventing issues during expansion. • gp_toolkit: updates to version 1.6. 	<i>Tools and utilities</i>

Query processing and optimization

Index and scan

- Enhanced index-only scan capabilities

- Supports index-only scans on a broader range of index types when using the GPORCA optimizer, including those with covering indexes using `INCLUDE` columns. This helps improve query performance. See [related document](#).
- Supports dynamic index-only scan when using the GPORCA optimizer to accelerate queries on partitioned tables. This feature combines partition pruning with index-only access to avoid heap lookups, significantly reducing I/O and improving performance. It is ideal for wide tables with narrow covering indexes and can be enabled using `SET optimizer_enable_dynamicindexonlyscan = on`. See [related document](#).
- Supports index-only scans when using the GPORCA optimizer on append-only (AO) tables and PAX tables, enabling faster query execution by avoiding block access when possible. This improves performance in scenarios where traditional index scans on AO and PAX tables were previously inefficient. See [related document](#).

- Improved index scan performance and flexibility

- Supports backward index scans when using the GPORCA optimizer for queries with `ORDER BY ... DESC`, eliminating the need for explicit sorting when a B-tree index exists in the opposite order. This optimization reduces resource usage and improves performance, especially for top-N and pagination queries. See [related document](#).

- The GPORCA optimizer supports triggering Bitmap Index Scans using array comparison predicates like `col IN (...)` or `col = ANY(array)`, including for hash indexes. This improves query performance on large datasets by enabling more efficient multi-value matching. The optimizer automatically chooses the bitmap scan path based on cost estimation. See [related document](#).
 - The GPORCA optimizer now considers the width of `INCLUDE` columns when costing index-only scans, favoring narrower indexes that return fewer unused columns. This improves plan selection for queries where multiple covering indexes are available. The cost model also more accurately estimates I/O by refining how `relallvisible` is used in index-only scan costing. See [related document](#).
- **BRIN index enhancements** ([related document](#))
 - Redesigns BRIN index internals for AO/CO tables to replace the UPPER page structure with a more efficient chaining model. This significantly reduces disk space usage for empty indexes and improves performance by avoiding unnecessary page access. The new design better handles the unique layout of AO/CO tables while maintaining correctness and compatibility.
 - BRIN indexes on AO/CO tables now support summarizing specific logical heap block ranges using `brin_summarize_range()`, enabling more precise control during index maintenance and testing. This enhancement also adds improved coverage for scenarios involving aborted rows, increasing robustness and correctness in edge cases.
 - Supports generating `IndexScan` plans when using the GPORCA optimizer with `ScalarArrayOp` qualifiers (for example, `col = ANY(array)`) for B-tree indexes. This enhancement aligns ORCA with the planner's behavior and allows more efficient execution of array comparison queries, as long as the predicate column is the first key in a multicolumn index.

View and materialized view

- Improves performance of REFRESH MATERIALIZED VIEW WITH NO DATA by avoiding full query execution. The command now behaves like a TRUNCATE, significantly reducing execution time while preserving proper dispatch to segments.

Join

- Supports left join pruning when using the GPORCA optimizer, allowing unnecessary left joins to be eliminated during query optimization. This applies when the query only uses columns from the outer table and the join condition fully covers the inner table's unique or primary keys. This can lead to more efficient query plans.
- Supports FULL JOIN using the Hash Full Join strategy when using the GPORCA optimizer. This approach avoids sorting join keys and reduces data redistribution, making it suitable for large datasets or joins on non-aligned distribution keys. All FULL JOIN queries now use Hash Full Join. See [related document](#).
- The GPORCA optimizer now avoids unnecessary data redistribution for multi-way self joins using left or right outer joins when the join keys are symmetric. This optimization improves performance by recognizing that such joins preserve data colocation, eliminating redundant motion operations. See [related document](#).
- The GPORCA optimizer no longer penalizes broadcast plans for NOT IN queries (Left Anti Semi Join), regardless of the optimizer_penalize_broadcast_threshold setting. This change improves performance and avoids potential OOM issues by enabling parallel execution instead of concentrating large tables on the coordinator node. See [related document](#).

Function & aggregate

- Supports intermediate aggregates when using the GPORCA optimizer, enabling more efficient execution of queries that include both DISTINCT aggregates and regular aggregates. This ensures correct handling of aggregation stages using AGGSPLIT. In addition, ORCA introduces an optimization for MIN() and MAX() functions by using index scans with a limit, instead of full table scans with regular aggregation. This optimization also supports

IS NULL and IS NOT NULL conditions on indexed columns, significantly improving performance for applicable queries.

- Enables more HashAggregate plan alternatives for queries that include DISTINCT aggregates when using the GPORCA optimizer. By generating a two-stage aggregation plan that avoids placing DISTINCT functions in hash-based nodes, ORCA ensures compatibility with the executor and expands the range of supported query plans. This improvement enhances optimization choices for group-by queries.
- Supports queries using GROUP BY CUBE, enabling multi-dimensional grouping sets in query plans. This expands analytic query capabilities. Note that optimization time for CUBE queries may be high due to the large number of generated plan alternatives.

Preprocessing

- Inlines Common Table Expressions (CTEs) that contain outer references, allowing such queries to be planned and explained successfully. Previously, these queries would fall back to the legacy planner due to limitations in handling shared scans with outer references. This change improves compatibility and enables ORCA to optimize a broader range of CTE-based queries.
- No longer rewrites IN queries to EXISTS when the inner subquery contains a set-returning function. This prevents invalid query transformations that could previously result in execution errors. The change ensures correct handling of queries like a IN (SELECT generate_series(1, a)).

Optimization and performance enhancements

- **Plan hint** (*related document*)
 - Supports plan hints for scan types and join row estimates when using the GPORCA optimizer, enabling users to guide query planning using pg_hint_plan-style comments. Supports scan hints include SeqScan, IndexScan, BitmapScan, and their negations, while row hints allow users to specify expected join cardinalities.
 - The plan hint field is now required in the ORCA optimizer configuration. This change simplifies internal parsing logic and ensures consistent handling of optimizer

configuration files.

- Supports join order hints for left and right outer joins when using the GPORCA optimizer, extending the existing hint framework beyond inner joins. This enhancement allows users to guide the optimizer’s join order more precisely in complex queries involving outer joins, improving plan control and potentially execution performance.

- **Enhancements to ORCA** (*related document*)

- Supports table aliases in query plans when using the GPORCA optimizer, making EXPLAIN outputs more descriptive and aligned with user-defined query syntax. In addition, ORCA adds support for query parameters, including those used in functions and prepared statements, enabling better compatibility with parameterized workloads and dynamic SQL execution.
- When using the GPORCA optimizer, supports generating plans for queries on tables with row-level security (RLS) enabled. Security policies are enforced during plan generation, ensuring only permitted rows are visible to each user. ORCA still falls back to the planner for RLS queries with sublinks, foreign tables, or for `INSERT` and `UPDATE` statements.
- The GPORCA optimizer now gracefully falls back to the Postgres planner when a function in the `FROM` clause uses `WITH ORDINALITY`, which is not currently supported. The fallback includes a clear error message indicating the unsupported feature.
- When using the GPORCA optimizer, supports pushing down filters with `BETWEEN` predicates when combined with constant filters, enabling more effective predicate propagation. This enhancement can reduce the number of rows processed during joins, improving query performance in applicable cases.
- When using the GPORCA optimizer, supports hashed subplans when the subquery expression is hashable and contains no outer references. This enhancement can significantly improve query performance by reducing execution time in applicable cases.
- ORCA now supports executing foreign tables with `mpp_execute='ANY'` on either the coordinator or segments, depending on cost. This allows more flexible and efficient execution plans for foreign data sources. A new “Universal” distribution type is introduced to support this behavior, similar to how `generate_series()` is handled.

- ORCA now supports direct dispatch for randomly distributed tables when the query includes a filter on `gp_segment_id`. This enhancement improves query performance by routing execution directly to the relevant segment, reducing unnecessary data processing across the cluster.
- ORCA now supports generating plans with the `ProjectSet` node, enabling correct execution of queries that include set-returning functions (SRFs) in the target list. This enhancement prevents fallback to the legacy planner and ensures compatibility with PostgreSQL 11+ behavior.
- ORCA now supports the `FIELDSELECT` node, which allows it to optimize a broader range of queries involving composite data types. Previously, such queries would fall back to the legacy planner. This enhancement improves compatibility and reduces unnecessary planner fallbacks.
- ORCA now derives statistics only for the columns used in `UNION ALL` queries, instead of all output columns from the input tables. This optimization reduces unnecessary computation and can improve planning performance for large queries.
- Updates naming in logs and `EXPLAIN` output to refer to the optimizers as “GPORCA” and “Postgres based planner” for improved clarity and consistency.
- Optimizes ORCA’s `Union All` performance by deriving statistics only for columns used in the query output. This reduces unnecessary computation and improves planning efficiency for queries with unused columns.

Transaction management

Lock management

- Updates logic to ignore invalidated slots while computing the oldest catalog Xmin, reducing the risk of deadlocks and improving transaction concurrency.
- Performs serializable isolation checks early for AO/CO tables, ensuring stricter consistency guarantees and reducing the likelihood of isolation conflicts.
- Enhances the index creation process to prevent deadlocks by ensuring the coordinator acquires an `AccessShareLock` on `pg_index` before dispatching a synchronization query to segments, thus aligning `indcheckxmin` and avoiding conflicts that GDD cannot resolve.

Transaction performance and reliability

- Avoids replaying DTX information in checkpoints for newly expanded segments, preventing potential inconsistencies during recovery.
- Adds `gp_stat_progress_dtx_recovery` for better observability of distributed transaction recovery progress.
- Improves error reporting for DTX protocol command dispatch errors, making it easier to diagnose and resolve issues.
- Allows utility mode on the coordinator to skip upgrading locks for `SELECT` locking clauses, improving efficiency for maintenance operations.

Storage

AO/CO table enhancements

- Optimizes `CREATE INDEX` operations on AO tables with scan progress reporting, enhancing the efficiency of index creation. See [related document](#).
- Declares the connected variable as “volatile” to ensure proper handling across `PG_TRY` and `PG_CATCH` blocks, mirroring PostgreSQL’s best practices for exception-safe variable usage in transaction control.

Partitioning

Related documents (in part)

- Extends Orca’s planning capabilities to include support for foreign partitions, enabling optimized query execution for tables with a mix of foreign and non-foreign partitions. The implementation introduces new logical and physical operators for foreign partitions, supports static and dynamic partition elimination, and integrates with any foreign data wrapper compatible, enhancing performance and flexibility for external data queries.
- Optimizes the analysis of leaf partitions in multi-level partition tables to avoid unnecessary resampling of intermediate partitions.

- Supports dynamic partition elimination (DPE) when using the GPORCA optimizer for plans involving duplicate-sensitive random motions. This allows partition selectors to pass through segment filters, enabling more efficient query plans and reducing the number of scanned partitions.
- Adds Dynamic Partition Elimination for Hash Right Joins, which enhances the efficiency of join operations on partitioned tables.
- Supports boolean static partition pruning in ORCA, enhancing the efficiency of partition pruning during query optimization.
- Enhances ORCA's query planning by incorporating partition key opfamily checks during partition pruning to optimize data distribution and partition scanning, ensuring correct motion triggering and partition scanning by aligning predicate operators with the distribution or partition key's opfamily, addressing issues with missing motion, incorrect direct dispatch, and ineffective partition pruning.
- Caches the last found partition in `ExecFindPartition` to improve performance for repeated partition lookups.
- Enables ORCA to derive dynamic table scan cardinality from leaf partitions, addressing limitations in handling date and time-related data types by changing their internal representation to doubles.
- Enhances the DPv2 algorithm to include distribution spec information with partition selectors, improving the efficiency of distributed query execution.
- Introduces a new Non-Replicated distribution specification to optimize join operations in database processing. By relaxing the enforcement of singleton distribution for outer tables when the inner table is universally distributed, it aims to reduce unnecessary data gathering and duplicate-sensitive motions, thereby generating more efficient execution plans.

Memory management

- Implements a custom allocator to enable ORCA to use standard C++ containers, addressing heap allocation management.
- Refactors ORCA's memory pool by making several methods static and adds assertions to ensure pointer safety.

- Optimizes serialization of IMDId objects in ORCA to be lazy, improving performance by deferring serialization until necessary. Improves optimization time when loading objects into the relcache and when involving large and wide partition tables.
- Ensures that strings returned by GetDatabasePath are always freed using pfree, preventing memory leaks.
- Enables MPP (Massively Parallel Processing) support for pg_buffercache and builds it by default, making buffer cache management more scalable and efficient in distributed environments.
- Introduces pg_buffercache_summary() to offer a high-level overview of buffer cache activity.

Metadata and access methods

- Allows the definition of lock modes for custom reloptions, providing more control over table and index access.
- Supports specification of reloptions when switching storage models, allowing seamless transitions between different storage formats.
- Introduces a new struct member in CreateStmt to indicate the origin of the statement, specifying if it was generated from GP style classic partitioning syntax.
- Adds syscache lookup for pg_attribute_encoding and pg_appendonly, improving performance and efficiency in metadata access.
- Introduces a new catalog entry in pg_aggregate to store replication safety information for aggregates, allowing users to mark specific aggregates as safe for execution on replicated slices via an optional repsafe parameter during the CREATE AGGREGATE command. This helps optimize performance by avoiding unnecessary broadcasts on large replicated datasets.
- Enhances the dispatch of ALTER DATABASE commands by allowing options like ALLOW_CONNECTIONS and IS_TEMPLATE to be dispatched to segments, ensuring catalog changes are reflected everywhere.

Data loading and external tables

External table enhancements

- Adds clearer restrictions and warnings when exchanging or attaching external tables. Writable external tables can no longer be used as partitions, and attaching readable external tables without validation now triggers a warning instead of requiring a no-op clause.
- Disables `SET DISTRIBUTED REPLICATED` for `ALTER EXTERNAL TABLE` to prevent misuse and ensure consistency.

Foreign data wrapper

- Improves performance and stability for `gpfdist` external tables. Adds TCP keepalive support for more reliable reads, and increases the default buffer size to enhance write throughput for writable external tables.
- ORCA now falls back to the planner for queries involving foreign partitions using `greenplum_fdw`, preventing crashes caused by incompatible execution behavior. Queries on non-partitioned foreign tables using `greenplum_fdw` remain supported by ORCA.

High availability and high reliability

Backup and disaster recovery

- CBDR

Introduces CBDR, a backup and recovery tool designed for SynxDB and Apache Cloudberry databases. Built on WAL-G, CBDR offers a user-friendly command-line interface that simplifies disaster recovery and ensures data safety.

CBDR supports both full and incremental backups, making it efficient for large-scale clusters. Users can list available backups, restore from any selected point, and store backups in S3-compatible object storage. Compared to existing tools like `gpbackup` and `gprestore`, CBDR provides enhanced flexibility with features such as support for multiple compression formats (lz4, lzma, zstd, brotli) and backup encryption, making it a comprehensive solution

for enterprise-grade backup strategies.

See [related document](#).

- DB recovery and synchronization
 - Improves archiver performance when handling many `.ready` files by reducing redundant directory scans. This change speeds up WAL archiving, especially when `archive_command` has been failing and many files have accumulated.
 - `gp_create_restore_point()` can only be executed on the Coordinator node. Calling this function on a segment node will result in an error. The function returns a structured record value, including the restore point name and LSN, which you can view directly by running `SELECT * FROM gp_create_restore_point()`.

WAL

- Improves WAL replication management by restricting a coordinator-specific tracking mechanism to the coordinator only. This change simplifies primary segment behavior and aligns replication practices more closely across segments. No functional change for users, but helps reduce unnecessary complexity in WAL retention logic.
- Enhances WAL retention logic to improve reliability of incremental recovery using `pg_rewind`. Physical replication slots now retain WAL files up to the last common checkpoint, reducing risk of missing WAL during recovery. This change also simplifies the underlying logic and adds test coverage for WAL recycling.
- Switches WAL replication connections to use the standard libpq protocol instead of a legacy internal one. This improves compatibility and reliability of replication behavior. Also fixed test failures and improved error handling for replication connections.

Security

DB Operations

- `REFRESH MATERIALIZED VIEW CONCURRENTLY` runs all internal operations in the correct security context to prevent potential privilege escalation. This change ensures safer execution by restricting operations to the appropriate permission level.

- Improves internal handling of new aoseg and aocsseg tuples by aligning tuple freezing behavior with other catalog operations. This change enhances consistency with upstream PostgreSQL practices and removes the need for CatalogTupleFrozenInsert.

System processes

- Orphaned file checks now exclude idle sessions during safety validation. This prevents unnecessary errors when persistent connections from services are active, allowing the detection process to complete successfully.
- Adds a safety check in backend signal handlers to ensure signals are handled by the correct process. This prevents unintended shared memory access by child processes and improves overall process isolation and stability.
- Improves process safety by preventing child processes spawned via `system()` from calling `proc_exit()`. This avoids potential corruption of shared memory structures and ensures only the parent process performs cleanup operations.
- Removes the permission check for `cpu.pressure` when using `gp_resource_manager='group-v2'`. This prevents startup failures on systems where PSI is disabled, without affecting resource management functionality.

Replication/Mirrorless clusters

- Improves replication error reporting by setting persistent `WalSndError` when a replication slot is invalidated. This ensures accurate error visibility in `gp_stat_replication`.

Permission management

- Strengthens security by rejecting extension schema or owner substitutions containing potentially unsafe characters like \$, ', or \. This prevents SQL injection in extension scripts and protects against privilege escalation in certain non-bundled extensions.
- Creating or assigning roles to the `system_group` resource group now results in an error, as this group is reserved for internal system processes only.
- Reverts the restriction requiring superuser privileges to set the

`gp_resource_group_bypass` GUC. This allows applications like GPCC to function more easily while still limiting resource impact.

- Altering the `mpp_execute` option of a foreign server or wrapper is now disallowed to prevent inconsistencies in foreign table distribution policies. Changing these options previously could result in outdated cached plans and incorrect query execution. This update ensures plan correctness by enforcing cache invalidation only when appropriate.

pgcrypto

- Adds support for FIPS mode in `pgcrypto`, controlled by a GUC. This allows SynxDB to operate in FIPS-compliant environments when linked with a supported FIPS-enabled OpenSSL version. Certain ciphers are disabled in this mode to comply with FIPS requirements.
- `pgcrypto` now allows enabling FIPS mode even on systems where FIPS is not pre-enabled by the OS or environment. This change removes the dependency on `FIPS_mode()` checks, offering more flexibility in managing FIPS compliance through the database.

Resource management

Resource group management

- Renames the `memory_limit` parameter to `memory_quota` in `CREATE/ALTER RESOURCE GROUP` to clarify its meaning and unit. See [related document](#).
- Adds a new system view `gp_toolkit.gp_resgroup_status_per_segment` to monitor memory usage per resource group on each segment. This view helps database administrators track real-time vmem consumption (in MB) when resource group-based resource management is enabled. See [related document](#).
- Improves logging behavior when memory usage reaches Vmem or resource group limits. The system now prints log messages directly to stderr to avoid stack overflow errors during allocation failures.
- Removes unnecessary permission check for `cpu.pressure` when using the `group-v2` resource manager. This prevents startup failures on systems where PSI is not enabled,

improving compatibility across Linux distributions. See [related document](#).

Logging and monitoring

- Adds additional log messages for GDD backends to help investigate memory-related issues. These logs provide better visibility into backend behavior during high memory usage scenarios.
- Adds a log ignore rule for “terminating connection” messages to reduce noise in test outputs. This helps avoid unnecessary diffs in CI for tests that involve connection termination.
- Adds more verbose logging to `ResCheckSelfDeadlock()`.
- Logs queue IDs and portal IDs in resource queue logs.
- Dumps more information when releasing resource queue locks to aid in troubleshooting and monitoring.
- Uses `ERROR` for dispatcher liveness checks.
- Enhances logging for dispatch connection liveness checks to improve clarity during connection failures. Logs now include more accurate error messages based on socket state and system errors.

Platform compatibility and build

- Improves `gp_sparse_vector` compatibility with ARM platforms by fixing type handling in serialization logic. This ensures consistent behavior across different architectures.
- Adds support for `sigaction()` on Windows to align signal handling behavior with other platforms. This reduces platform-specific differences and improves code consistency.
- Updates ACL mode type in ORCA to match the parser’s definition, ensuring consistent type usage.

System views and statistics

- Improves join cardinality estimation for projected columns that preserve the number of distinct values (NDVs), such as additions or subtractions with constants. This allows the optimizer to use underlying column histograms for more accurate estimates, improving plan quality for queries with scalar projections in join conditions.
- Increases precision for frequency and NDV values in ORCA when processing metadata population scripts (MDPs). This change ensures consistent behavior between MDPs and live database queries, reducing discrepancies caused by rounding small values.
- ORCA now considers null value skew when costing redistribute motions, improving plan accuracy for queries involving columns with many nulls. This helps avoid performance issues caused by data being unevenly distributed across segments.
- ORCA now supports extended statistics to improve cardinality estimation for queries with correlated columns. This allows the optimizer to use real data-driven correlation factors instead of relying on arbitrary GUC settings, leading to more accurate query plans.
- Introduces `gp_log_backend_memory_contexts` to log memory contexts across segments, with optional targeting by content ID. This enhances observability and helps diagnose memory issues in distributed queries.
- ORCA now supports statistics derivation for predicates involving different time-related data types, such as date and timestamp. This improves plan accuracy and performance for queries comparing mixed temporal types.
- Autostats now uses `SKIP LOCKED` for `ANALYZE` operations to avoid blocking on locks, reducing the risk of deadlocks and improving predictability. This behavior is enabled by default and can be controlled using the `gp_autostats_lock_wait` GUC.
- ORCA now supports `STATS_EXT_NDISTINCT` extended statistics for estimating cardinality on correlated columns. This improves accuracy for queries using `GROUP BY` or `DISTINCT` on such columns.

Network connections

- Marks `gp_reject_internal_tcp_connection` as defunct to improve reliability of internal QD-to-entry DB connections. These connections over TCP/IP are now treated as authenticated by default, preventing authentication errors caused by `pg_hba.conf` settings.

Tools and utilities

- **analyzedb**
 - `analyzedb` now includes materialized views in its list of tables to analyze. This improves the performance immediately after analysis.
- **gpexpand**
 - `gpexpand` now includes a cluster health check to ensure all segments are up and in their preferred roles before proceeding. This prevents incorrect port assignments and avoids potential issues during expansion when nodes are not in a stable state.
- **gp_toolkit**
 - Added an update path for the `gp_toolkit` extension to version 1.6. This update renames the column `memory_limit` to `memory_quota` in the `gp_resgroup_config` view for improved clarity. Users can apply the update using `ALTER EXTENSION gp_toolkit UPDATE TO '1.6'.`

Interactive manager CBCC

CBCC is a monitoring and operations dashboard component designed for SynxDB. It provides a unified web console that displays real-time cluster status, resource usage, database object details, SQL execution metrics, and more. It supports automated operations and alert management across the entire cluster. CBCC must be installed separately.

CBCC provides the following key features:

- **Cluster overview:** Displays core metrics such as database version, uptime, number of connections, total tables, and disk usage. Helps users quickly understand the health of the cluster.

- **Resource monitoring:** Shows near real-time usage of CPU, memory, disk I/O, and network I/O. Supports viewing the status and metrics of individual nodes and hosts.
- **Storage analysis:** Displays disk usage by node to support capacity planning and storage optimization.
- **Database object view:** Provides detailed table information by schema, user, and partitioning. Includes table size and estimated row count.
- **SQL monitoring:** Tracks the execution status of active and historical SQL statements to support query performance optimization.
- **Alerting:** Supports creating alert rules to monitor abnormal resource usage or system events and trigger notifications.
- **Notification delivery:** Allows configuration of contact groups to ensure the right people are notified when alerts are triggered.

Category	Features	User document
Deploy	Deploy CBCC	<i>Install Monitoring Console</i>
Cluster	View cluster overview	<i>View Cluster Information</i>
Cluster	View cluster runtime details	<i>View the overall status and data of the cluster</i>
Cluster	View storage overview	<i>View Storage Information</i>
Database	View database object information	<i>View Database Object Information</i>
Database	View SQL monitoring information	<i>View SQL Monitoring Information</i>
Alert	Configure alert rules	<i>View and Create Alert Rules</i>
Alert	View and create contact groups	<i>View and Create Contact Groups</i>

Data intelligence platform (HashML)

HashML Platform is a high-performance distributed computing platform designed for enterprise-grade AI applications. It integrates the strengths of open-source technologies and enhances them with enterprise-grade features. The platform offers a complete end-to-end solution covering data preparation, feature engineering, model training, and production deployment. It supports unified scheduling and management of heterogeneous computing resources, including CPU, GPU, and NPU. HashML is compatible with mainstream deep learning frameworks such as PyTorch and TensorFlow, and provides an interactive visual workspace that enables data scientists and algorithm engineers to efficiently manage the full lifecycle of data mining, model development, deployment, and operations.

HashML Platform offers the following features and advantages:

- **Deep integration with enterprise-grade database capabilities:** HashML shares metadata management and storage engines with SynxDB, enabling unified cataloging, access control, and cross-layer scheduling for both structured and unstructured data. It supports full-text search, vector search, and hybrid retrieval, providing native capabilities for building knowledge bases and enhancing large model retrieval (RAG).
- **Elastic resource scheduling and heterogeneous computing support:** Built on deep integration with KubeRay and Kubernetes, the platform dynamically allocates CPU and GPU resource pools and supports on-demand scaling of training tasks. It is compatible with physical machines, virtual machines, and containerized environments, enabling unified management and efficient utilization of heterogeneous compute resources.
- **Full-stack algorithm support:** The platform includes a comprehensive library of AI algorithms, covering traditional machine learning (such as XGBoost and LightGBM), deep learning (such as ResNet and BERT), and large language model fine-tuning technologies (such as QLoRA and Unslot), meeting diverse modeling needs across scenarios.
- **Enterprise-grade security and compliance:** HashML provides built-in support for data encryption, model-level access isolation, and audit logging to ensure full-process compliance for sensitive data and AI models. It also supports tiered storage strategies and access control to meet the strict security requirements of industries such as finance and government.
- **Low-code AI application development:** The platform simplifies model training and deployment with visual modeling and one-click publishing. It exposes core capabilities—such as knowledge bases, intelligent Q&A, and smart diagnostics—via Rest APIs, enabling rapid integration with third-party systems and supporting secondary development.

Category	Feature	User document
Deploy	Deploy the HashML Platform console	<i>Deploy HashML Platform</i>
Platform	Data management, model development, training deployment	<i>Use the HashML Platform</i>

Upgrade path

To upgrade to SynxDB v4.0.0 from an earlier version, use the `cbccopy` or `hashcopy` utility to export data from the existing cluster and import it into a newly deployed v4.0.0 cluster. This export-import approach ensures data consistency and supports a clean migration path to the latest version.

Product change information

SQL syntax

- The `memory_limit` parameter has been **renamed to** `memory_quota` in both `CREATE RESOURCE GROUP` and `ALTER RESOURCE GROUP` statements.

Before:

```
CREATE RESOURCE GROUP rg_memory_test WITH (memory_limit = -1, cpu_max_percent = 20, concurrency = 2);
ALTER RESOURCE GROUP rg_memory_test SET memory_limit = 1000;
```

After:

```
CREATE RESOURCE GROUP rg_memory_test WITH (memory_quota = -1, cpu_max_percent = 20, concurrency = 2);
ALTER RESOURCE GROUP rg_memory_test SET memory_quota = 1000;
```

- The `cpu_hard_quota_limit` parameter has been **replaced by** `cpu_max_percent`.

Before:

```
CREATE RESOURCE GROUP rg_test_group WITH (cpu_hard_quota_limit = 10);
ALTER RESOURCE GROUP rg_test_group SET cpu_hard_quota_limit = 100;
```

After:

```
CREATE RESOURCE GROUP rg_test_group WITH (cpu_max_percent = 10);
ALTER RESOURCE GROUP rg_test_group SET cpu_max_percent = 100;
```

- The `CREATE RESOURCE GROUP` statement now supports a new parameter: `io_limit`.

Example:

```
CREATE RESOURCE GROUP rg_test_group1 WITH (
  concurrency = 10,
  cpu_max_percent = 10,
  io_limit = 'pg_default:rbps=1000,wbps=1000,riops=1000,wiops=1000'
);
```

- Another new parameter cpuset has also been introduced to control CPU binding.

Example:

```
CREATE RESOURCE GROUP rg_test_group WITH (cpuset = '1;4-5');
```

A new clause ALTER TABLE ... SET ACCESS METHOD has been added to support changing the access method of a table.

```
ALTER TABLE heaptable SET ACCESS METHOD heap2;
```

Configuration parameters (GUCs)

For details of GUCs, see the *GUC document*.

The following configuration parameters are added in v4.0.0:

- allow_dml_directory_table: The default value is off.
- autovacuum_freeze_max_age: The age at which to autovacuum a table to prevent transaction ID wraparound.
- autovacuum_vacuum_cost_delay: The vacuum cost delay in milliseconds, for autovacuum.
- autovacuum_vacuum_scale_factor: The number of tuple updates or deletes prior to vacuum as a fraction of reltuples.
- autovacuum_vacuum_threshold: The minimum number of tuple updates or deletes prior to vacuum.
- gp_appendonly_compaction_segfile_limit: Sets the minimum number of segfiles that must always be available for inserts.

- `gp_autostats_lock_wait`: The autostats generated ANALYZE statements to wait for lock acquisition.
- `gp_enable_runtime_filter_pushdown`: Tries to push the hash table of hash join to the seqscan or AM as bloom filter.
- `gp_enable_statement_trigger`: Enables statement triggers to be created instead of erroring out.
- `gp_max_partition_level`: Sets the maximum number of levels allowed when creating a partitioned table using Greenplum classic syntax.
- `optimizer_array_constraints`: Allows the optimizer's constraint framework to derive array constraints. The default value is `on`.
- `optimizer_array_expansion_threshold`: The item limit for expansion of arrays in WHERE clause for constraint derivation.
- `optimizer_cost_model`: Sets the optimizer cost model.
- `optimizer_cost_threshold`: Sets the threshold for plan sampling relative to the cost of best plan. `0.0` means unbounded.
- `optimizer_cte_inlining_bound`: Sets the CTE inlining cutoff.
- `optimizer_damping_factor_filter`: Selects predicate damping factor in optimizer. `1.0` means no damping.
- `optimizer_damping_factor_groupby`: The group by operator damping factor in optimizer. `1.0` means no damping.
- `optimizer_damping_factor_join`: The join predicate damping factor in optimizer. `1.0` means no damping; `0.0` means square root method.
- `optimizer_discard_redistribute_hashjoin`: Discards hash join with redistribute motion in the optimizer.
- `optimizer_dpe_stats`: Enables statistics derivation for partitioned tables with dynamic partition elimination.
- `optimizer_enable_derive_stats_all_groups`: Enables stats derivation for all groups after exploration.

- `optimizer_enable_dynamicbitmapscan`: Enables the optimizer' s use of plans with dynamic bitmap scan.
- `optimizer_enable_dynamicindexonlyscan`: Enables the optimizer' s use of plans with dynamic index only scan.
- `optimizer_enable_dynamicindexscan`: Enables the optimizer' s use of plans with dynamic index scan.
- `optimizer_enable_foreign_table`: Enables foreign tables in GPORCA.
- `optimizer_enable_orderedagg`: Enables ordered aggregate plans.
- `optimizer_enable_push_join_below_union_all`: Enables transform of join of union all to union all of joins, which might improve the join performance.
- `optimizer_enable_query_parameter`: Enables query parameters in GPORCA.
The default value is `on`.
- `optimizer_enable_right_outer_join`: Enables GPORCA to generate plans containing right outer joins.
- `optimizer_force_three_stage_scalar_dqa`: Forces the optimizer to always pick 3-stage aggregate plan for scalar distinct qualified aggregate.
- `optimizer_nestloop_factor`: Sets the nestloop join cost factor in the optimizer.
- `optimizer_penalize_broadcast_threshold`: The maximum number of rows of a relation that can be broadcasted without penalty. A value of 0 means disabled.
- `optimizer_push_group_by_below_setop_threshold`: The maximum number of children that setops have to consider pushing group by s below it.
- `optimizer_replicated_table_insert`: Omits broadcast motion when inserting into replicated table.
- `optimizer_skew_factor`: The coefficient of skew ratio computed from sample stastics.
- `optimizer_sort_factor`: Sets the sort cost factor in the optimizer. The value of `1.0` means the same as the default. If the value is greater than `> 1.0`, the optimizer is more costly than the default. If the value is less than `1.0`, the optimizer is less costly than the default.

- `optimizer_trace_fallback`: Prints a message at `INFO` level, whenever GPORCA falls back.
- `optimizer_use_gpdb_allocators`: Enables the GPORCA to use Memory Contexts.
- `optimizer_xform_bind_threshold`: The maximum number bindings per xform per group expression. The value 0 means disabled.

The following configuration parameters are removed in v4.0.0:

- `gp_enable_sort_distinct`
- `gp_safefswritesize`
- `vector.enable_plan_merge`
- The default value of `gp_command_count` in v4.0.0 has been changed from 2 to 0, which means that the system will not show any command received from the client in a session by default.
- The default value of `krb_server_keyfile` in v4.0.0 has been changed from `FILE:/usr/local/cloudberry-db-devel/etc/postgresql krb5.keytab` to `FILE:/workspace/dist/database/etc/postgresql krb5.keytab`, which means that the default location of the Kerberos server key files has been changed to `FILE:/workspace/dist/database/etc/postgresql krb5.keytab`.
- The category of `work_mem` in v4.0.0 has been changed from `Deprecated Options` to `Resource Usage / Memory`.
- The minimum value of `max_replication_slots` in v4.0.0 has been changed from 1 to 0, which means that the maximum number of simultaneously defined replication slots has been changed to 0 by default.
- The default value of `writable_external_table_bufsize` in v4.0.0 has been changed from 64 to 1024, which means that the default buffer size for the writable external table before writing data to gpfdist has been changed to 1024 KB.
- The default value of `wal_compression` in v4.0.0 has been changed from `off` to `on`, which means that the system will compress full-page writes written in WAL files by default.
- The default value of `log_checkpoints` in v4.0.0 has been changed from `off` to `on`, which means that the system will log each checkpoint by default.

- The default value, minimum value, and maximum value of `gp_server_version_num` in v4.0.0 have been changed from 10000 to 20000.

Extensions and plugins

The following extensions and plugins are **bundled with the |product_name| v4.0.0 RPM installation package**. These components are preinstalled and ready for use in supported environments.

- `kafka_fdw`
- `oracle_fdw`
- `mysql_fdw`
- `hive_connector`
- `datalake_fdw`
- `PL/R`
- `PL/Perl`
- `PL/Java`
- `pgvector`
- `zombodb`
- `postgis`
- `pg_jieba`
- `roaringbitmap`
- `vectorization`
- `madlib`
- `perfmon`
- `pg_pool`
- `pgaudit`

- gophermeta
- unionstore_ext
- tablespace (with enable_dfs_tablespace support)
- gp_exttable_delimiter
- gpbackup
- filedump
- cbcopy
- cbdr
- anon
- pax

These tools are **not included** in the main RPM and must be installed separately.

- PXF (platform extension framework for accessing external data sources)
- CBCC (for web-based cluster monitoring and alerting)

Removed features

In v4.0.0, zhparser has been removed and is not available.

Starting from v4.0.0, **Web Platform** has been replaced by **CBCC** as the default monitoring and management console.

CBCC provides the same core monitoring capabilities as Web Platform, including:

- Cluster overview: View key metrics such as database version, uptime, number of connections, total tables, and disk usage.
- Resource monitoring: Monitor near real-time CPU, memory, disk I/O, and network I/O usage at both the cluster and host levels.
- Storage analysis: Inspect disk usage by node to assist with capacity planning.
- Database object view: View detailed table information by schema, user, and partition,

including estimated row counts and table sizes.

- SQL monitoring: Track active and historical SQL execution status to support performance tuning.

CBCC also introduces new features that were not available in Web Platform:

- Alerting system: Define alert rules to monitor abnormal resource usage or system events and trigger notifications.
- Notification delivery: Configure contact groups to ensure alerts are delivered to the appropriate recipients.
- Standalone deployment: CBCC is not bundled with the database and must be installed separately.

As of v4.0.0, **Web Platform is no longer available**. This includes all Web Platform-specific features such as:

- Built-in deployment interface for provisioning SynxDB clusters.
- Built-in SQL editor and worksheet workspace.
- Native bundling with the database installation.

Users upgrading to v4.0.0 are advised to install CBCC separately to continue using web-based monitoring and operational tools.

Bug fixes

- Fixed data loss caused by incorrect shared snapshot handling.
- Fixed memory corruption during AOCO ADD COLUMN abort.
- Fixed checkpoint WAL replay failure.
- Fixed incorrect results when using UNION for RECURSIVE_CTE.
- Fixed incorrect results from hash joins on char columns.
- Fixed incorrect results produced by WITH RECURSIVE queries.
- Fixed incorrect results when a REPLICATED table is unioned with a DISTRIBUTED table.

- Fixed incorrect results when the outer query had ORDER BY after a LATERAL subquery.
- Fixed incorrect behavior of DELETE with split update.
- Fixed incorrect results when using direct dispatch.
- Fixed memory leaks in ORCA and various components.
- Fixed long-running execution with bitmap indexes.
- Fixed redundant SORT enforcement on group aggregates.
- Fixed incorrect index position in target list in ExecTupleSplit.
- Fixed incorrect value in the cpu_usage column returned by pg_resgroup_get_status().
- Fixed incorrect behavior of gp_toolkit.gp_move_orphaned_files.
- Fixed incorrect results in multi-stage aggregate queries.
- Fixed incorrect plan and output in multi-stage aggregate queries.
- Fixed incorrect reltuples value after VACUUM.
- Fixed incorrect index->reltuples value after VACUUM.
- Fixed a vulnerability where LDAP leaked user information.
- Fixed incorrect permissions warning on the pgpass file.
- Fixed incorrect handling of ONLY keyword for multiple tables in GRANT/REVOKE statements.
- Fixed incorrect permissions in resource management DDL.
- Fixed incorrect security context in REFRESH MATERIALIZED VIEW CONCURRENTLY.
- Fixed deadlock between coordinator and segments.
- Fixed race condition in CTE reader-writer communication.
- Fixed race condition when invalidating obsolete replication slots.
- Fixed deadlock by allowing concurrent creation of non-first indexes on AO tables.
- Fixed locking issue when opening range tables inside ExecInitModifyTable().

- Fixed incorrect unlock mode in DefineRelation.
- Fixed incorrect locking in partition distribution policies.
- Fixed issues with rle_type when converting a table from AO to AOCO.
- Fixed incorrect handling of empty ranges and NULL values in BRIN indexes.
- Fixed incorrect handling of NULL values when merging BRIN summaries.
- Fixed incorrect TIDs order when building bitmap indexes.
- Fixed possible inconsistency between bitmap LOV table and its index.
- Fixed incorrect behavior of VACUUM in AO tables with indexes.
- Fixed incorrect handling of TOAST values for invisible AppendOptimized tuples during VACUUM.
- Fixed ORCA's invalid processing of nested SubLinks under aggregates.
- Fixed ORCA's invalid processing of nested SubLinks referenced in GROUP BY clauses.
- Fixed ORCA's invalid processing of nested SubLinks with GROUP BY attributes.
- Fixed incorrect predicate pushdown when using casted columns.
- Fixed incorrect join condition loss after pulling up sublinks to join nodes.
- Fixed incorrect hash-key generation for Redistribute Motion in multi-DQA expressions.
- Fixed incorrect plan generation for SEMI JOIN with RANDOM distributed tables.
- Fixed incorrect behavior of gp_stat_bgwriter.
- Fixed incorrect monitoring in pg_stat_slru.
- Fixed incorrect monitoring in gp_stat_progress_dtx_recovery.
- Fixed incorrect monitoring in pg_resgroup_get_status().
- Fixed incorrect monitoring in gp_toolkit.gp_resgroup_config.
- Fixed compilation issues on various platforms.
- Fixed documentation and comment typos.

- Fixed build system and Makefile issues.
- Fixed various memory leaks and resource management issues.
- Fixed various error handling and logging improvements.
- Fixed mismatched types.
- Fixed the ORCA preprocess step for queries with the Select-Project-NaryJoin pattern.
- Fixed the missing discard_output variable in shared scan node functions.
- Fixed the crash caused by running VACUUM AO_AUX_ONLY on an AO-partitioned table.
- Fixed an obvious memory leak in _bitmap_xlog_insert_bitmapwords().
- Fixed a memory leak in the merge join implementation.
- Fixed the issue where the token for user ID xxx did not exist.
- Fixed the issue where plan hints could not derive table descriptors.
- Fixed the issue where inject_fault suspend could not be canceled.
- Fixed fallback in debug builds due to scalars with invalid return types.
- Fixed relptr encoding of the base address.
- Fixed visimap consults for unique checks during UPDATE operations.
- Fixed the issue where external table location URIs containing | caused errors.
- Fixed handling of the time command output containing commas.
- Fixed a small overestimation of the output length of base64 encoding.
- Fixed gp_toolkit.__gp_aocsseg_history crash on non-AO columnar tables.
- Fixed a race condition between termination and resqueue wakeup.
- Fixed a statement leak involving self-deadlocks.
- Fixed the detection of child output columns when the parent is a UNION during join pruning.
- Fixed a query crash when using a negative memory_limit value in resource groups.
- Fixed issues in pgarch new directory-scanning logic.

- Fixed a memory leak in the FTS PROBE process.
- Fixed check_multi_column_list_partition_keys.
- Fixed a memory leak caught via ICW with memory check enabled.
- Fixed query hang and fallback issues involving CTEs on replicated tables.
- Fixed the unrecognized join type error with LASJ Not-In and network types.
- Fixed issues in upgrade_adapt.sql related to queries using WITH OIDS.
- Fixed the double declaration of check_ok() in pg_upgrade.h.
- Fixed logic error with subdirectories generated by pg_upgrade for internal files.
- Fixed a typo in the pg_upgrade file header.
- Fixed the bug where PL/Python functions caused the master process to reset.
- Fixed the Shared Scan hang issue involving initplans.
- Fixed motion toast error.
- Fixed a memory leak related to fsync in AO tables.
- Fixed CDatumSortedSet handling of empty arrays that caused errors in ORCA.
- Fixed ORCA returning incorrect column type modifier information.
- Fixed DbgStr output when printing DP structs in ORCA.
- Fixed the comment on performDtxProtocolPrepare.
- Fixed a memory leak in Dynamic Index, IndexOnly, and BitmapIndex scans during execution.
- Fixed the memory accounting bug when moving MemoryContext under another accounting node.
- Fixed the ALTER TABLE ALTER COLUMN TYPE issue that reuses an incorrect index.
- Fixed query fallback when a subquery is present within LEAST() or GREATEST().
- Fixed the typo in timestamp.
- Fixed unexpected warnings related to pg_stat_statements node types.

- Fixed the crash involving initplan in MPP.
- Fixed LeftJoinPruning pruning essential LEFT JOINs.
- Fixed the SET command that incorrectly sends DTX protocol commands.
- Fixed the segmentation fault in addOneOption().
- Fixed parallel_retrieve_cursor diffs.
- Fixed gpdifff.pl to ignore information when EXPLAIN ignores costs.
- Fixed the uninitialized-use warning in CTranslatorDXLToPISmt.cpp.
- Fixed the bug where the LOCALE flag cannot be used with a string pattern.
- Fixed a typo in cdbmutate.c.
- Fixed CColRefSet debug printing.
- Fixed ORCA producing incorrect plans when handling SEMI JOIN with RANDOM distributed tables.
- Fixed orphaned temp tables on the coordinator.
- Fixed the segmentation fault caused by concurrent INSERT ON CONFLICT and DROP TABLE.
- Fixed redundant columns in a multi-stage aggregate plan.
- Fixed the import of ICU collations in pg_import_system_collations().
- Fixed the error: “Cannot add cell to table content: total cell count of XXX exceeded.”
- Fixed orphaned temporary namespace catalog entries left on the coordinator.
- Fixed REFRESH MATERIALIZED VIEW on AO tables with indexes.
- Fixed the use of PORTNAME in the gp_toolkit Makefile.
- Fixed pg_stat_activity display for bypassed and unassigned queries.
- Fixed the recursive CTE MergeJoin that involved a motion on WTS.
- Fixed the column width display for partitioned tables.

- Fixed the LDAP crash when ldaptls=1 and ldapscheme is not set.
- Fixed the gpstop pipeline flakiness after the referenced change.
- Fixed the ANALYZE bug in expand_vacuum_rels.
- Fixed the compilation error.
- Fixed the ORCA crash due to improper colref mapping with CTEs.
- Fixed the bug where gupload insert mode was not included in a transaction.
- Fixed the bug where resgroup total wait time was always zero.
- Fixed the gpcheckcat error against pg_description.
- Fixed flakiness caused by waiting for a different number of fault triggers.
- Fixed the bug involving RelabelType in the GROUP BY clause.
- Fixed the planner error with multiple copies of an AlternativeSubPlan.
- Fixed the issue with bitmap indexes.
- Fixed the bug in HashAgg related to selective-column-spilling logic.
- Fixed the bug in disk-based hash aggregation.
- Fixed the pipeline stall issue in LookupTupleHashEntryHash().
- Fixed the use of version in ArgumentParser, which is deprecated.
- Fixed the use of BaseException.message, which has been deprecated since Python 2.6.
- Fixed the case pg_rewind_fail_missing_xlog.
- Fixed the compiler warning for gcc-12.
- Fixed support for the DEFERRABLE keyword on primary and unique keys.
- Fixed the unlocking of pruned partitions in partitioned tables.
- Fixed the crash in ORCA involving skip-level correlated queries.
- Fixed the removal of Assert statements in release builds.
- Fixed the typo in comments: JOIN_SEMI_DEDUP/JOIN_SEMI_DEDUP_REVERSE.

- Fixed the issue where REORGANIZE=TRUE did not redistribute randomly-distributed tables.
- Fixed the core dump caused by concurrent updates on partition tables in DynamicScan.
- Fixed the typo: ANALZE to ANALYZE.
- Fixed the issue where cgroup v1 cpu_quota_us cannot be larger than its parent's value.
- Fixed indentation and trailing whitespace in UDFs in resgroup/resgroup_auxiliary_tools_v1.
- Fixed the name of cpu_hard_quota_limit in resgroup_syntax.sql.
- Fixed multi-row DEFAULT handling in INSERT … SELECT rules.
- Fixed invalid function references in several comments.
- Fixed the bug where COPY FORM does not throw ERROR: extra data after last expected column.
- Fixed the issue where file .204800 was not being checked in ao_foreach_extent_file.
- Fixed the issue of incorrectly incrementing the command counter.
- Fixed the coordinator crash in MPPnoticeReceiver.
- Fixed the dangling pointer in ExecDynamicIndexScan().
- Fixed the ORCA bug that incorrectly removed required redistribution motion when using GROUP BY over gp_segment_id.
- Fixed header handling in url_curl.c.
- Fixed ao_filehandler to support new attnum to filenum mapping changes.
- Fixed pg_aocsseg to work with attnum to filenum mapping.
- Fixed a comment in pg_dump.
- Fixed the ORCA build break.
- Fixed the gpconfig SSH retry undefined parameter issue.
- Fixed the stale gp_default_storage_options comment.
- Fixed the bug: unrecognized node type: 147.

- Fixed spelling errors identified by lintian.
- Fixed the bypass catalog unit test.
- Fixed erroneous Valgrind markings in AllocSetRealloc.
- Fixed the legacy bug in the DatabaseFrozenIds lock.
- Fixed the mirror checkpointer error on the ALTER DATABASE query.
- Fixed the bug: get_ao_compression_ratio() failed on root partitioned tables with AO children.
- Fixed the issue where InterruptHoldoffCount was not being reset.
- Fixed gpexpand failure caused by an event trigger.
- Fixed missing redistribute for CTAS or INSERT INTO on randomly distributed tables when using ORCA.
- Fixed the double free of remapper->typmodmap in TeardownUDPIFCInterconnect().
- Fixed the bug in the upstream-merged COMMIT AND CHAIN feature.
- Fixed inconsistency between gp_fastsequence row and index after a crash.
- Fixed the typo allocatd to allocated.
- Fixed the error: unrecognized node type: 145 in transformExpr.
- Fixed build error caused by unused variable.
- Fixed the issue where the distribution key was missing when creating a stage table.
- Fixed the regex for etc/environment.d.
- Fixed the string comparison warning.
- Fixed obsolete references to SnapshotNow in comments.
- Fixed pull-up error when the target list contains a RelabelType node.
- Fixed the issue where index DDL operations were recorded in QEs' pg_last_stat_operation.
- Fixed two compiler warnings.
- Fixed the wrong value of maxAttrNum in TupleSplitState.

- Fixed the bug of incorrect index position in target list in ExecTupleSplit.
- Fixed the format error of the library name on Mac M1.
- Fixed the pg_resgroup_get_status_kv() function.
- Fixed interconnect bugs in ic_proxy_ibuf_push().
- Fixed memory leaks in auto_explain.
- Fixed ic_proxy compilation when HOST_NAME_MAX is unavailable.
- Fixed duplicate filters caused by reversed operator argument order.
- Fixed pg_rewind when the log file is a symbolic link.
- Fixed and enabled 64-bit bitmapset and updated visimap.
- Fixed the hang caused by multi-DQA with filters in the planner.
- Fixed the bogus ORCA plan that incorrectly joins a CTE and a REPLICATED table.
- Fixed the error in ATSETAM when applied to ao_column with a dropped column.
- Fixed the LWLockHeldByMe assert failure in SharedSnapshotDump.
- Fixed the KeepLogSeg() unit test.
- Fixed the race condition when invalidating obsolete replication slots.
- Fixed the uninitialized value in segno calculation.
- Fixed issues in the invalidation logic for obsolete replication slots.
- Fixed checkpoint signalling.
- Fixed memory overrun when querying pg_stat_slru.
- Fixed the bug where ORCA fails to decorrelate subqueries ordered by outer references.
- Fixed unused variable compile warnings.
- Fixed the bug where NestLoop join fails to materialize the inner child in some cases.
- Fixed COPY execution via FDW on coordinator as executor.
- Fixed inFunction usage for auto_stats in CTAS.

- Fixed a compiler warning.
- Fixed the syntax error with CREATE MATERIALIZED VIEW.
- Fixed the issue preventing temporary table creation LIKE existing tables with comments.
- Fixed and rewrote IndexOpProperties API.
- Removed redundant Get/SetStaticPruneResult usage.
- Fixed EPQ handling for DML operations.
- Fixed gpcheckperf failure when using -V with -f option.
- Fixed possible mirror startup failure triggered by FTS promotion.
- Fixed the parallel retrieve cursor issue when selecting transient record types.
- Fixed the resource management DDL warning: unrecognized node type when log_statement=' ddl'
- Fixed the resgroup init error when many cores are present in cpuset.cpus.
- Fixed resqueue malfunction when using JDBC extended protocol.
- Fixed the missing LOCKING CLAUSE on foreign tables when ORCA is enabled.
- Fixed the test_consume_xids behavior where it consumes one more transaction ID than expected.
- Fixed the ONLY keyword handling for multiple tables in GRANT/REVOKE statements.
- Fixed the regression test to ignore memory usage values in JSON format EXPLAIN output.
- Fixed relcache lookup in ORCA when selecting from sequences.
- Fixed missing WAL files required by pg_rewind.
- Fixed the gp_dqa test to explicitly ANALYZE tables.
- Fixed the crash of AggNode in the executor caused by an ORCA plan.
- Fixed the resource group cpuset test case.
- Fixed the compiler warning caused by gpfldist with compressed external tables.

- Fixed link issues on macOS and Windows.
- Fixed failure when DynamicSeqScan contains a SubPlan.
- Fixed the error: cache lookup failed for type 0.
- Fixed the multi-level correlated subquery bug.
- Fixed checkpoint WAL replay failure.
- Fixed the check for BufFileRead() in ExecHashJoinGetSavedTuple().
- Fixed the test extension to allow executing SQL code inside a Portal.
- Fixed resgroup view test cases.
- Fixed incorrect DISTKEY assignment when copying partitions on segments.
- Fixed ic-proxy mis-disconnecting addresses after reloading the config file.
- Fixed the gpcheckcat check on partition distribution policies.
- Fixed colid remapping in disjunctive constraints.
- Fixed the Makefile by removing the tablespace-step target from all.
- Fixed CBitSet intersection logic in ORCA.
- Fixed the query preprocessor for nested Select-Project-NaryJoin patterns.
- Fixed incorrect unlock mode in DefineRelation.
- Fixed the upgrade process for external tables with dropped columns.
- Fixed the formatting issue in SECURITY.md.
- Fixed gp_gettmid to return the correct startup timestamp.
- Fixed the gload regression test failure when the OS user is not gpadmin.
- Fixed the compiler warning in appendonlyblockdirectory.c.
- Fixed missing reoptions in partition roots created using SynxDB syntax.
- Fixed the crash when calling get_ao_compression_ratio on HEAP tables.
- Fixed incorrect sortOp and eqOp values generated by IsCorrelatedEqualityOpExpr.

- Fixed the dependency bug involving minirepro and materialized views.
- Fixed recursion handling in ALTER TABLE ...ENABLE/DISABLE TRIGGER.
- Fixed SPE plans to display Partitions selected: 1 (out of 5).
- Fixed incorrect hash-key generation for Redistribute Motion when creating paths for multi-DQA expressions.
- Removed gp_enable_sort_distinct and noduplicates optimizations.
- Fixed gpinit system Behave tests that use environment variables.
- Fixed false alarms in gpcheckcat for pg_default_acl.
- Fixed gpinit system failure with custom locale settings.
- Fixed a panic in the greenplum_fdw test.
- Fixed the failure in bitmap index null-array condition.
- Fixed the compilation warning in gram.y.
- Fixed multiple issues related to DistributedTransaction handling.
- Fixed compile-time warnings in pg_basebackup code.
- Fixed gplogfilter to correctly generate CSV output.
- Fixed the assert in the OpExecutor node.
- Fixed improper copying of group statistics in ORCA.
- Fixed error reporting after ioctl() call in pg_upgrade –clone mode.
- Fixed replay of CREATE DATABASE records on standby.
- Fixed a minor memory leak in pg_dump.
- Fixed parallel restore of foreign keys to partitioned tables.
- Fixed the issue where the pg_appendonly entry was not removed during AO-to-HEAP table conversion.
- Fixed assertion failure and segmentation fault in the backup code.

- Fixed fallback behavior for non-default collations.
- Fixed the subtransaction test for Python 3.10.
- Fixed Windows client compilation of libpgcommon.
- Fixed compiler warnings introduced by the Dynamic Scan commit.
- Fixed the issue where CREATE OR REPLACE TRANSFORM failed.
- Fixed compiler warnings for non-assert builds.
- Fixed lock assertions in dhash.c.
- Fixed watch interaction with libedit on C.

Chapter 2

Deployment Guides

2.1 Deploy on Physical Machines

Deploy Manually

Software and Hardware Configuration

This document introduces the software and hardware configuration required for SynxDB.

Hardware requirements

Physical machine

The following section describes the recommended physical machine configuration for SynxDB in test and production environments.

For development or test environments

Component	CPU	Memory	Disk type	Network	Number of instances
Coordinator	4 cores	8 GB	SSD	10 Gbps NIC (2 preferred)	1+
Segment	4 cores	8 GB	SSD	10 Gbps NIC (2 preferred)	1+
ETCD	2 cores	4 GB	SSD	10 Gbps NIC (2 preferred)	2+
FTS	2 cores	4 GB	SSD	10 Gbps NIC (2 preferred)	1+

For production environments

Component	CPU	Memory	Disk type	Network	Instance count
Coordinator	16+ cores	32+ GB	SSD	10 Gbps NIC (2 preferred)	2+
Segment	8+ cores	32+ GB	SSD	10 Gbps NIC (2 preferred)	2+
ETCD	16+ cores	64+ GB	SSD	10 Gbps NIC (2 preferred)	3+
FTS	4+ cores	8+ GB	SSD	10 Gbps NIC (2 preferred)	3+

Storage

- To prevent a high data disk load from affecting the operating system's normal I/O response, mount the operating system and the data disk on separate disks.
- If the host configuration allows, it is recommended to use 2 independent SAS disks as the system disk (RAID1), and another 10 SAS disks as the data disk (RAID5).
- It is recommended to use LVM logical volumes to manage disks for more flexible disk configuration.

For the system disk: The system disk should use an independent disk to avoid impact on the operating system when data disks are heavily loaded. It is recommended that the system disk be configured in dual-disk RAID 1 and the operating system of the system disk be XFS.

For data disks: It is recommended to use LVM to manage data disks. According to test statistics, creating an independent logical volume for each physical volume can achieve the best disk performance. For example:

```
pvcreate /dev/vdb
pvcreate /dev/vdc
```

(continues on next page)

(continued from previous page)

```
pvcreate /dev/vdd
vgcreate data /dev/vdb /dev/vdc /dev/vdd
lvcreate --extents 100%pvfs -n data0 data /dev/vdb
lvcreate --extents 100%pvfs -n data1 data /dev/vdc
lvcreate --extents 100%pvfs -n data2 data /dev/vdd
```

The names of mount points must be consecutive, and the mount points of data disks should be /data0, /data1, …, /dataN. Data disks should use the XFS file format. For example:

```
mkdir -p /data0 /data1 /data2
mkfs.xfs /dev/data/data0
mkfs.xfs /dev/data/data1
mkfs.xfs /dev/data/data2
mount /dev/data/data0 /data0/
mount /dev/data/data1 /data1/
mount /dev/data/data2 /data2/
```

Data exchange network

- **Network card configuration**

The data exchange network is used for transmitting business data, which has high requirements on network performance and throughput. In a production environment, two 10 Gbps NICs are generally required, and they will be used after bonding. The recommended bond 4 parameter are as follows:

```
BONDING_OPTS='mode=4 miimon=100 xmit_hash_policy=layer3+4'
```

- **Connectivity requirements**

- Connect the management console and the database host in the data exchange network. If there is a firewall device between the management console and the database host, ensure that the TCP idle connection can be kept for more than 12 hours.
- Connect database hosts and management console hosts in the data exchange network, and do not limit the TCP idle connection time.
- Connect database clients and application programs that access the database with the

database coordinator node in the data exchange network.

- Ensure that the TCP idle connection can be kept for more than 12 hours.

- **Default gateway**

If the host is configured with a management network, the network card (bond0) of the data exchange network should be used as the default gateway device; otherwise, it might cause abnormal traffic monitoring of the host network, deployment failure, and performance problems. The following is an example of viewing the default gateway.

```
netstat -rn | grep ^0.0.0.0
```

- **Switch**

- Make sure that the egress bandwidth of the data network switch from layer 1 to layer 2 is no lower than the maximum disk I/O throughput capacity of a single cabinet (calculated with a single RAID card of 500 MBps).
- A switch convergence ratio of 4:1 is recommended. When the convergence ratio reaches 6:1, most links will be saturated. Significant packet loss occurs when the convergence ratio reaches 8:1.

Software requirements

Supported OS

SynxDB supports the following operating systems:

- RHEL/CentOS 7.6+ (x86_64/aarch64)
- Kylin v10 (x86_64/aarch64)
- BCLinux 8 (x86_64)
- BCLinux-for-euler-2210 (x86_64/aarch64)
- OpenEuler 2203 SP2 (x86_64/aarch64)
- UOS1060e (x86_64/aarch64)

- NeoKylin V7update6

SSH configurations

- The recommended configuration for the SSH server side (`/etc/ssh/sshd_config`) is as follows. After the configuration is complete, run `systemctl restart sshd.service` to make it effective.

Parameter	Value	Description
Port	22	Listening port.
PasswordAuthentication	yes	Allows password login, which can be changed after cluster initialization.
PermitEmptyPasswords	no	Empty password is not allowed for login.
UseDNS	no	DNS is not used.

Configure SSH password-free login for all nodes. For example:

```
ssh-keygen -t rsa
ssh-copy-id root@192.168.66.154
```

Prepare to Deploy on Physical Machine

Before deploying SynxDB on physical machines, you need to do some preparations. Read this document and [Software and Hardware Configuration](#) before you start to deploy SynxDB.

Plan the deployment architecture

Plan your deployment architecture based on the [Product Architecture](#) and [Software and Hardware Configuration](#) and determine the number of servers needed. Ensure that all servers are within a single security group and have mutual trust configured.

The deployment plan for the example of this document includes 1 coordinator + 1 standby + 3 segments (primary + mirror), totaling 5 servers.

Modify server settings

Log into each host as the root user, and modify the settings of each node server in the order of the following sections.

Change hostname

Use the `hostnamectl set-hostname` command to modify the hostname of each server respectively, following these naming conventions:

- Only include letters, numbers, and the hyphen -. Note: The underscore _ is not a valid character.
- Case-insensitive, but it is recommended to use all lowercase letters. Using uppercase letters for the hostname might cause Kerberos authentication to fail.
- Each hostname must be globally unique across all hosts.

Example:

```
hostnamectl set-hostname cbdb-coordinator
hostnamectl set-hostname cbdb-standbycoordinator
hostnamectl set-hostname cbdb-datanode01
```

(continues on next page)

(continued from previous page)

```
hostnamectl set-hostname cbdb-datanode02  
hostnamectl set-hostname cbdb-datanode03
```

Add gpadmin admin user

Follow the example below to create a user group and username gpadmin. Set the user group and username identifier to 520. Create and specify the gpadmin home directory /home/gpadmin.

```
groupadd -g 520 gpadmin # Adds user group gpadmin.  
useradd -g 520 -u 520 -m -d /home/gpadmin/ -s /bin/bash gpadmin # Adds  
username gpadmin and creates the home directory of gpadmin.  
passwd gpadmin # Sets a password for gpadmin; after executing, follow the  
prompts to input the password.
```

Disable SELinux and firewall software

Run `systemctl status firewalld` to view the firewall status. If the firewall is on, you need to turn it off by setting the `SELINUX` parameter to disabled in the `/etc/selinux/config` file.

1. Set the `SELINUX` parameter value to `disabled` in the `/etc/selinux/config` file.

```
SELINUX=disabled
```

2. Then run the following commands to completely disable the firewall.

```
systemctl stop firewalld.service  
systemctl disable firewalld.service
```

Modify network mapping

Check the `/etc/hosts` file to make sure that it contains mappings of all host aliases to their network IP addresses. Examples are as follows:

```
192.168.1.101 cbdb-coordinator
192.168.1.101 cbdb-datanode01
192.168.1.101 cbdb-datanode02
```

Set system parameters

Add relevant system parameters in the `/etc/sysctl.conf` configuration file, and run the `sysctl -p` command to make the configuration file effective.

When setting the configuration parameters, you can take the following example as a reference and set them according to your needs. Details of some of these parameters and recommended settings are provided below.

```
# kernel.shmall = _PHYS_PAGES / 2
kernel.shmall = 197951838
# kernel.shmmax = kernel.shmall * PAGE_SIZE
kernel.shmmax = 810810728448
kernel.shmmni = 4096
vm.overcommit_memory = 2
vm.overcommit_ratio = 95
net.ipv4.ip_local_port_range = 10000 65535
kernel.sem = 250 2048000 200 8192
kernel.sysrq = 1
kernel.core_uses_pid = 1
kernel.msgmnb = 65536
kernel.msgmax = 65536
kernel.msgmni = 2048
net.ipv4.tcp_syncookies = 1
net.ipv4.conf.default.accept_source_route = 0
net.ipv4.tcp_max_syn_backlog = 4096
net.ipv4.conf.all.arp_filter = 1
net.ipv4.ipfrag_high_thresh = 41943040
net.ipv4.ipfrag_low_thresh = 31457280
net.ipv4.ipfrag_time = 60
net.core.netdev_max_backlog = 10000
net.core.rmem_max = 2097152
net.core.wmem_max = 2097152
vm.swappiness = 10
vm.zone_reclaim_mode = 0
```

(continues on next page)

(continued from previous page)

```
vm.dirty_expire_centisecs = 500
vm.dirty_writeback_centisecs = 100
vm.dirty_background_ratio = 0
vm.dirty_ratio = 0
vm.dirty_background_bytes = 1610612736
vm.dirty_bytes = 4294967296
```

Shared memory

In the `/etc/sysctl.conf` configuration file, `kernel.shmall` represents the total amount of available shared memory, in pages. `kernel.shmmax` represents the maximum size of a single shared memory segment, in bytes.

You can define these 2 values using the operating system's `_PHYS_PAGES` and `PAGE_SIZE` parameters:

```
kernel.shmall = (_PHYS_PAGES / 2)
kernel.shmmax = (_PHYS_PAGES / 2) * PAGE_SIZE
```

To get the values of these 2 operating system parameters, you can use `getconf`, for example:

```
## echo $$expr $(getconf _PHYS_PAGES) / 2
## echo $$expr $(getconf _PHYS_PAGES) / 2 \* $(getconf PAGE_SIZE)
```

Segment memory

In the `/etc/sysctl.conf` configuration file:

- `vm.overcommit_memory` indicates the overcommit handling modes for memory.

Available options are:

0: Heuristic overcommit handling 1: Always overcommit 2: Don't overcommit

Set the value of this parameter to 2 to refuse overcommit.

- `vm.overcommit_ratio` is a kernel parameter and is the percentage of RAM occupied by the application process. The default value on CentOS is 50. `vm.overcommit_ratio` is

calculated as follows:

```
vm.overcommit_ratio = (RAM - 0.026 * gp_vmem) / RAM
```

The calculation method of `gp_vmem` is as follows:

```
# If the system memory is less than 256 GB, use the following
formula to calculate:
gp_vmem = ((SWAP + RAM) - (7.5GB + 0.05 * RAM)) / 1.7

# If the system memory is greater than or equal to 256 GB, use the
following formula to calculate:
gp_vmem = ((SWAP + RAM) - (7.5GB + 0.05 * RAM)) / 1.17

# In the above formulas, SWAP is the swap space on the host, in
GB.
# RAM is the size of the memory installed on the host, in GB.
```

Port

In the `/etc/sysctl.conf` configuration file, `net.ipv4.ip_local_port_range` is used to specify the port range. To avoid port conflicts between SynxDB and other applications, you need to specify the port range via operating system parameters. When you later set SynxDB initialization parameters, avoid setting SynxDB related ports in this range.

For example, for `net.ipv4.ip_local_port_range = 10000 65535`, you need to avoid setting the SynxDB related ports in the interval `[10000, 65535]`. You can set them to 6000 and 7000:

```
PORT_BASE = 6000
MIRROR_PORT_BASE = 7000
```

IP segmentation

When the SynxDB uses the UDP protocol for internal connection, the network card controls the fragmentation and reassembly of IP packets. If the size of a UDP message is larger than the maximum size of network transmission unit (MTU), the IP layer fragments the message.

- `net.ipv4.ipfrag_high_thresh`: When the total size of IP fragments exceeds this threshold, the kernel will attempt to reorganize IP fragments. If the fragments exceed this threshold but all fragments have not arrived within the specified time, the kernel will not reorganize the fragments. This threshold is typically used to control whether larger shards are reorganized. The default value is 4194304 bytes (4 MB).
- `net.ipv4.ipfrag_low_thresh`: Indicates that when the total size of IP fragments is below this threshold, the kernel will wait as long as possible for more fragments to arrive, to allow for larger reorganizations. This threshold is used to minimize unfinished reorganization operations and improve system performance. The default value is 3145728 bytes (3 MB).
- `net.ipv4.ipfrag_time` is a kernel parameter that controls the IP fragment reassembly timeout. The default value is 30.

It is recommended to set the above parameters to the following values:

```
net.ipv4.ipfrag_high_thresh = 4194304
net.ipv4.ipfrag_low_thresh = 31457280
net.ipv4.ipfrag_time = 60
```

System memory

- If the server memory exceeds 64 GB, it is recommended to set the following parameters in the `/etc/sysctl.conf` configuration file:

```
vm.dirty_background_ratio = 0
vm.dirty_ratio = 0
vm.dirty_background_bytes = 1610612736 # 1.5GB
vm.dirty_bytes = 4294967296 # 4GB
```

- If the server memory is less than 64 GB, do not set `vm.dirty_background_bytes` and `vm.dirty_bytes`, it is recommended to set the following parameters in the

/etc/sysctl.conf configuration file:

```
vm.dirty_background_ratio = 3
vm.dirty_ratio = 10
```

- To deal with emergencies when the system encounters memory pressure, it is recommended to add the `vm.min_free_kbytes` parameter in the `/etc/sysctl.conf` configuration file to specify the amount of available memory reserved by the system. It is recommended to set `vm.min_free_kbytes` to 3% of the system's physical memory. The command is as follows:

```
awk 'BEGIN {OFMT = "%.0f";} /MemTotal/ {print "vm.min_free_kbytes =", $2 * .03;}' /proc/meminfo >> /etc/sysctl.conf
```

- It is not recommended that the setting of `vm.min_free_kbytes` exceed 5% of the system's physical memory.

Resource limit

Edit the `/etc/security/limits.conf` file and add the following content, which limits the usage of software and hardware resources.

```
* soft nofile 524288
* hard nofile 524288
* soft nproc 131072
* hard nproc 131072
* soft core unlimited
```

CORE DUMP

- Add the following parameter to the `/etc/sysctl.conf` configuration file:

```
kernel.core_pattern=/var/core/core.%h.%t
```

- Run the following command to make the configuration effective:

```
sysctl -p
```

Set mount options for the XFS file system

XFS is the file system for the data directory of SynxDB. XFS has the following mount options:

```
rw, nodev, noatime, inode64
```

You can set up XFS file mounting in the `/etc/fstab` file. See the following commands. You need to choose the file path according to the actual situation:

```
mkdir -p /data0/  
mkfs.xfs -f /dev/vdc  
echo "/dev/vdc /data0 xfs rw,nodev,noatime,nobarrier,inode64 0 0" >> /etc/  
fstab  
mount /data0  
chown -R gpadmin:gpadmin /data0/
```

Run the following command to check whether the mounting is successful:

```
df -h
```

Blockdev value

The blockdev value for each disk file should be 16384. To verify the blockdev value of a disk device, use the following command:

```
sudo /sbin/blockdev --getra <devname>
```

For example, to verify the blockdev value of the example server disk:

```
sudo /sbin/blockdev --getra /dev/vdc
```

To modify the blockdev value of a device file, use the following command:

```
sudo /sbin/blockdev --setra <bytes> <devname>
```

For example, to modify the file blockdev value of the hard disk of the example server:

```
sudo /sbin/blockdev --setra 16384 /dev/vdc
```

I/O scheduling policy settings for disks

The disk type, operating system and scheduling policies of SynxDB are as follows:

Storage device type	OS	Recommended scheduling policy
NVMe	RHEL 7	none
	RHEL 8	none
	Ubuntu	none
SSD	RHEL 7	noop
	RHEL 8	none
	Ubuntu	none
Other	RHEL 7	deadline
	RHEL 8	mq-deadline
	Ubuntu	mq-deadline

Refer to the following command to modify the scheduling policy. Note that this command is only a temporary modification, and the modification becomes invalid after the server is restarted.

```
echo schedulername > /sys/block/<devname>/queue/scheduler
```

For example, temporarily modify the disk I/O scheduling policy of the example server:

```
echo deadline > /sys/block/vdc/queue/scheduler
```

To permanently modify the scheduling policy, use the system utility grubby. After using grubby, the modification takes effect immediately after you restart the server. The sample command is as follows:

```
grubby --update-kernel=ALL --args="elevator=deadline"
```

To view the kernel parameter settings, use the following command:

```
grubby --info=ALL
```

Disable Transparent Huge Pages (THP)

You need to disable Transparent Huge Pages (THP), because it reduces database performance. The command is as follows:

```
grubby --update-kernel=ALL --args="transparent_hugepage=never"
```

Check the status of THP:

```
cat /sys/kernel/mm/*transparent_hugepage/enabled
```

Disable IPC object deletion

Disable IPC object deletion by setting the value of RemoveIPC to no. You can set this parameter in the `/etc/systemd/logind.conf` file of SynxDB.

```
RemoveIPC=no
```

After disabling it, run the following command to restart the server to make the disabling setting effective:

```
service systemd-logind restart
```

SSH connection threshold

To set the SSH connection threshold, you need to modify the `/etc/ssh/sshd_config` configuration file's MaxStartups and MaxSessions parameters.

```
MaxStartups 200  
MaxSessions 200
```

Run the following command to restart the server to make the setting take effect:

```
service sshd restart
```

Clock synchronization

SynxDB requires the clock synchronization to be configured for all hosts, and the clock synchronization service should be started when the host starts. You can choose one of the following synchronization methods:

- Use the coordinator node's time as the source, and other hosts synchronize the clock of the coordinator node host.
- Synchronize clocks using an external clock source.

The example in this document uses an external clock source for synchronization, that is, adding the following configuration to the `/etc/chrony.conf` configuration file:

```
# Use public servers from the pool.ntp.org project.  
# Please consider joining the pool (http://www.pool.ntp.org/join.html).  
server 0.centos.pool.ntp.org iburst
```

After setting, you can run the following command to check the clock synchronization status:

```
systemctl status chronyd
```

Deploy on Multiple Nodes

This document introduces how to manually deploy SynxDB on physical machines using RPM package on multiple nodes. Before reading this document, it is recommended to first read [Software and Hardware Configuration](#).

The deployment method in this document is for production environments.

The example in this document uses CentOS 7.6 and deploys SynxDB. The main steps are as follows:

1. Prepare node servers.
2. Install the RPM package.
3. Configure mutual trust between nodes.
4. Initialize the database.
5. Log into the database.

Step 1: Prepare server nodes

Read the [Prepare to Deploy on Physical Machine](#) document to prepare the server nodes.

Step 2. Install the RPM package

After the preparation, it is time to install SynxDB. You can get the corresponding RPM package from the support personnel, and then install the database on each node using the installation package.

1. Get the RPM package and copy it to the home directory (/home/gpadmin/) of gpadmin:
2. Install the RPM package in the /home/gpadmin directory.

When running the following command, you need to replace <RPM package path> with the actual RPM package path, as the root user. During the installation, the directory /usr/local/synxdb/ is automatically created.

```
cd /home/gpadmin
sudo yum install <RPM package path>
```

- Grant the gpadmin user the permission to access the /usr/local/synxdb/ directory.

```
chown -R gpadmin:gpadmin /usr/local
chown -R gpadmin:gpadmin /usr/local/synxdb*
```

Step 3. Configure mutual trust between nodes

- Switch to the gpadmin user, and use the gpadmin user for subsequent operations.

```
su - gpadmin
```

- Create a configuration file for node information.

Create the node configuration file in the /home/gpadmin/ directory, including the all_hosts and seg_hosts files, which store the host information of all nodes and data nodes respectively. The example node information is as follows:

```
[gpadmin@cbdb-coordinator gpadmin]$ cat all_hosts
cbdb-coordinator
cbdb-standbycoordinator
cbdb-datanode01
cbdb-datanode02
cbdb-datanode03

[gpadmin@cbdb-coordinator gpadmin]$ cat seg_hosts
cbdb-datanode01
cbdb-datanode02
cbdb-datanode03
```

- Configure SSH trust between hosts.

- Run ssh-keygen on each host to generate SSH key. For example:

```
[gpadmin@cbbd-coordinator synxdb-4.0.0]$ ssh-keygen

Generating public/private rsa key pair.
Enter file in which to save the key (/usr/local/synxdb/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /usr/local/synxdb/.ssh/id_rsa.
Your public key has been saved in /usr/local/synxdb/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:cvcYS87egYCyh/v6UtdqrejVU5qqF7OvpcHg/T9lRrg gpadmin@cbbd-
coordinator
The key's randomart image is:
+---[RSA 2048]---+
| |
| |
| +   |
| +     O   |
| o ... S   |
| . +o= B C   |
| o B=00 D   |
| .o=o0o.. =   |
| O=++*+o+..   |
+---[SHA256]---
```

- Run `ssh-copy-id` on each host to configure password-free login. The example is as follows:

```
ssh-copy-id cbdb-coordinator
ssh-copy-id cbdb-standbycoordinator
ssh-copy-id cbdb-datanode01
ssh-copy-id cbdb-datanode02
ssh-copy-id cbdb-datanode03
```

- Verify that SSH between nodes is all connected, that is, the password-free login between servers is successful. The example is as follows:

```
[gpadmin@cbdb-coordinator ~]$ source /usr/local/synxdb/greenplum_path.
sh
[gpadmin@cbdb-coordinator ~]$ gpssh -f all_hosts
=> pwd
```

(continues on next page)

(continued from previous page)

```
[ cbdb-datanode03] b'/usr/local/synxdb\r'
[ cbdb-coordinator] b'usr/local/synxdb\r'
[ cbdb-datanode02] b'usr/local/synxdb\r'
[cbdb-standbycoordinator] b'usr/local/synxdb\r'
[ cbdb-datanode01] b'usr/local/synxdb\r'
=>
```

Step 4. Initialize SynxDB

Before performing the following operations, run `su - gpadmin` to switch to the `gpadmin` user.

1. Add a new line of `source` command to the `~/.bashrc` files of all nodes (coordinator/standby coordinator/segment). The example is as follows:

```
source /usr/local/synxdb/greenplum_path.sh
```

2. Run the `source` command to make the newly added content effective:

```
source ~/.bashrc
```

3. Use the `gpssh` command on the coordinator node to create data directories and mirror directories for segment nodes. In this document, the 2 directories are `/data0/primary/` and `/data0/mirror/`, respectively. The example is as follows:

```
gpssh -f seg_hosts
mkdir -p /data0/primary/
mkdir -p /data0/mirror/
```

4. Create data directory on the coordinator node. In this document, the directory is `/data0/coordinator/`.

```
mkdir -p /data0/coordinator/
```

5. Use the `gpssh` command on the coordinator node to create data directory for the standby node. In this document, the directory is `/data0/coordinator/`.

```
gpssh -h cbdb-standbycoordinator -e 'mkdir -p /data0/coordinator/'
```

6. On the hosts of the coordinator and standby nodes, add a line to the `~/.bashrc` file to declare the path of `COORDINATOR_DATA_DIRECTORY`, which is {the path step 5} + `gpseg-1`. For example:

```
export COORDINATOR_DATA_DIRECTORY=/data0/coordinator/gpseg-1
```

7. Run the following command on the hosts of the coordinator and standby nodes to make the declaration of `COORDINATOR_DATA_DIRECTORY` in the previous step effective.

```
source ~/.bashrc
```

8. Configure the `gpinitSystem_config` initialization script:

1. On the host where the coordinator node is located, copy the template configuration file to the current directory:

```
cp $GPHOME/docs/cli_help/gpconfigs/gpinitSystem_config .
```

2. Modify the `gpinitSystem_config` file as follows:

- Pay attention to the port, coordinator node, segment node, and mirror node.
- Modify `DATA_DIRECTORY` to the data directory of the segment node, for example, `/data0/primary`.
- Modify `COORDINATOR_HOSTNAME` to the hostname of the coordinator node, for example, `cbdb-coordinator`.
- Modify `COORDINATOR_DIRECTORY` to the data directory of the coordinator node, for example, `/data0/coordinator`.
- Modify `MIRROR_DATA_DIRECTORY` to the data directory of the mirror node, for example, `/data0/mirror`.

```
[gpadmin@cbdb-coordinator ~]$ cat gpinitSystem_config
# FILE NAME: gpinitSystem_config

# Configuration file needed by the gpinitSystem
```

(continues on next page)

(continued from previous page)

```
#####
##### REQUIRED PARAMETERS
#####

##### Naming convention for utility-generated data directories.
SEG_PREFIX=gpseg

##### Base number by which primary segment port numbers
##### are calculated.
PORT_BASE=6000

##### File system location(s) where primary segment data directories
##### will be created. The number of locations in the list dictate
##### the number of primary segments that will get created per
##### physical host (if multiple addresses for a host are listed in
##### the hostfile, the number of segments will be spread evenly
##### across
##### the specified interface addresses).
declare -a DATA_DIRECTORY=(/data0/primary)

##### OS-configured hostname or IP address of the coordinator host.
COORDINATOR_HOSTNAME=cbdb-coordinator

##### File system location where the coordinator data directory
##### will be created.
COORDINATOR_DIRECTORY=/data0/coordinator

##### Port number for the coordinator instance.
COORDINATOR_PORT=5432

##### Shell utility used to connect to remote hosts.
TRUSTED_SHELL=ssh

##### Default server-side character set encoding.
ENCODING=UNICODE

#####
##### OPTIONAL MIRROR PARAMETERS
#####
```

(continues on next page)

(continued from previous page)

```

##### Base number by which mirror segment port numbers
##### are calculated.
MIRROR_PORT_BASE=7000

##### File system location(s) where mirror segment data directories
##### will be created. The number of mirror locations must equal the
##### number of primary locations as specified in the
##### DATA_DIRECTORY parameter.
declare -a MIRROR_DATA_DIRECTORY=(/data0/mirror)

```

- To create a default database during initialization, you need to fill in the database name. In this example, the `warehouse` database is created during initialization.

```

#####
##### OTHER OPTIONAL PARAMETERS
#####

#####
##### Create a database of this name after initialization.
DATABASE_NAME=warehouse

```

9. Use `gpinitsystem` to initialize SynxDB. For example:

```
gpinitsystem -c gpinitsystem_config -h /home/gpadmin/seg_hosts
```

In the command above, `-c` specifies the configuration file and `-h` specifies the computing node list.

If you need to initialize the standby coordinator node, refer to the following command:

```
gpinitstandby -s cbdb-standbycoordinator
```

Step 5. Log into the Database

Now you have successfully deployed SynxDB. To log into the database, refer to the following command:

```
psql -h <hostname> -p <port> -U <username> -d <database>
```

In the command above:

- <hostname> is the IP address of the coordinator node of the SynxDB server.
- <port> is the default port number of SynxDB, which is 5432 by default.
- <username> is the user name of the database.
- <database> is the name of the database to connect.

After you run the psql command, the system will prompt you to enter the database password.

After you enter the correct password, you will successfully log into the SynxDB and can perform SQL queries and operations. Make sure that you have the correct permissions to access the target database.

```
[gpadmin@cdedb-coordinator ~]${ psql warehouse
psql (14.4, server 14.4)
Type "help" for help.

warehouse=# SELECT * FROM gp_segment_configuration;
dbid | content | role | preferred_role | mode | status | port | hostname
      | address |          | datadir
-----
1    | -1     | p     | p           | n     | u     | 5432 | cdedb-
coordinator          | cdedb-coordinator          | /data0/coordinator/gpseg-1
8    | -1     | m     | m           | s     | u     | 5432 | cdedb-
standbycoordinator | cdedb-standbycoordinator | /data0/coordinator/gpseg-1
2    | 0      | p     | p           | s     | u     | 6000 | cdedb-
datanode01          | cdedb-datanode01          | /data0/primary/gpseg0
5    | 0      | m     | m           | s     | u     | 7000 | cdedb-
datanode02          | cdedb-datanode02          | /data0/mirror/gpseg0
3    | 1      | p     | p           | s     | u     | 6000 | cdedb-
datanode02          | cdedb-datanode02          | /data0/primary/gpseg1
```

(continues on next page)

(continued from previous page)

6	1	m	m	s	u	7000	cddb-
			cddb-datanode03			/data0/mirror/gpseg1	
4	2	p	p	s	u	6000	cddb-
			cddb-datanode03			/data0/primary/gpseg2	
7	2	m	m	s	u	7000	cddb-
			cddb-datanode01			/data0/mirror/gpseg2	
(8 rows)							

Deploy on Single Node

SynxDB is not fully compatible with PostgreSQL, and some features and syntax are SynxDB-specific. If your business already relies on SynxDB and you want to use the SynxDB-specific syntax and features on a single node to avoid compatibility issues with PostgreSQL, you can consider deploying SynxDB free of segments.

SynxDB provides the single-computing-node deployment mode. This mode runs under the utility `gp_role`, with only one coordinator (QD) node and one coordinator standby node, without a segment node or data distribution. You can directly connect to the coordinator and run queries as if you were connecting to a regular multi-node cluster. Note that some SQL statements are not effective in this mode because data distribution does not exist, and some SQL statements are not supported. See *User-behavior changes* for details.

How to deploy

Step 1. Prepare to deploy

Log into each host as the root user, and modify the settings of each node host in the order of the following sections.

Add `gpadmin` admin user

Follow the example below to create a user group and username `gpadmin`, set the user group and username identifier to 520, and create and specify the home directory `/home/gpadmin/`.

```
groupadd -g 520 gpadmin # Adds user group gpadmin.  
useradd -g 520 -u 520 -m -d /home/gpadmin/ -s /bin/bash gpadmin # Adds  
username gpadmin and creates the home directory.  
passwd gpadmin # Sets a password for gpadmin. Follow the prompts to input  
the password after executing._
```

Disable SELinux and firewall software

Run `systemctl status firewalld` to view the firewall status. If the firewall is on, you need to turn it off by setting the `SELINUX` parameter to `disabled` in the `/etc/selinux/config` file.

```
SELINUX=disabled
```

You can also disable the firewall using the following commands:

```
systemctl stop firewalld.service
systemctl disable firewalld.service
```

Set system parameters

Edit the `/etc/sysctl.conf` configuration file, add the relevant system parameters, and run the `sysctl -p` command to make the configuration effective.

The following configuration parameters are for reference. Please adjust according to your actual needs. Detailed explanations and recommended settings for some parameters are provided below.

```
kernel.shmall = _PHYS_PAGES / 2
kernel.shmall = 197951838
kernel.shmmax = kernel.shmall * PAGE_SIZE
kernel.shmmax = 810810728448
kernel.shmmni = 4096
vm.overcommit_memory = 2
vm.overcommit_ratio = 95
net.ipv4.ip_local_port_range = 10000 65535
kernel.sem = 250 2048000 200 8192
kernel.sysrq = 1
kernel.core_uses_pid = 1
kernel.msgmnb = 65536
kernel.msgmax = 65536
kernel.msgmni = 2048
net.ipv4.tcp_syncookies = 1
net.ipv4.conf.default.accept_source_route = 0
net.ipv4.tcp_max_syn_backlog = 4096
```

(continues on next page)

(continued from previous page)

```

net.ipv4.conf.all.arp_filter = 1
net.ipv4.ipfrag_high_thresh = 41943040
net.ipv4.ipfrag_low_thresh = 31457280
net.ipv4.ipfrag_time = 60
net.core.netdev_max_backlog = 10000
net.core.rmem_max = 2097152
net.core.wmem_max = 2097152
vm.swappiness = 10
vm.zone_reclaim_mode = 0
vm.dirty_expire_centisecs = 500
vm.dirty_writeback_centisecs = 100
vm.dirty_background_ratio = 0
vm.dirty_ratio = 0
vm.dirty_background_bytes = 1610612736
vm.dirty_bytes = 4294967296

```

Shared memory settings

In the `/etc/sysctl.conf` configuration file:

- `kernel.shmall` represents the total amount of available shared memory, in pages. `kernel.shmmax` represents the maximum size of a single shared memory segment, in bytes. You can define these values using the operating system's `_PHYS_PAGES` and `PAGE_SIZE` parameters:

```

kernel.shmall = ( _PHYS_PAGES / 2)
kernel.shmmax = ( _PHYS_PAGES / 2) * PAGE_SIZE

```

To get the values of these two operating system parameters, you can use `getconf`, as shown below:

```

$ echo $(expr $(getconf _PHYS_PAGES) / 2)
$ echo $(expr $(getconf _PHYS_PAGES) / 2 \* $(getconf PAGE_SIZE))

```

- `vm.overcommit_memory` is a Linux kernel parameter that controls the system's memory overcommit handling. Setting `vm.overcommit_memory` to 2 means the system will refuse memory overcommit when the allocation exceeds 2 GB.

- `vm.overcommit_ratio` is a kernel parameter that represents the percentage of RAM allocated to processes. The default value on CentOS is 50. The formula to calculate `vm.overcommit_ratio` is as follows:

```
vm.overcommit_ratio = (RAM - 0.026 * gp_vmem) / RAM
```

The method to calculate `gp_vmem` is as follows:

```
# If the system memory is less than 256 GB, use the following
formula:
gp_vmem = ((SWAP + RAM) - (7.5GB + 0.05 * RAM)) / 1.7

# If the system memory is greater than or equal to 256 GB, use the
following formula:
gp_vmem = ((SWAP + RAM) - (7.5GB + 0.05 * RAM)) / 1.17

# In the formulas above, SWAP represents the swap space on the
host, in GB.
# RAM represents the installed memory on the host, in GB.
```

IP segmentation settings

When the SynxDB uses the UDP protocol for internal connection, the network card controls the fragmentation and reassembly of IP packets. If the size of a UDP message is larger than the maximum size of network transmission unit (MTU), the IP layer fragments the message.

- `net.ipv4.ipfrag_high_thresh`: When the total size of IP fragments exceeds this threshold, the kernel will attempt to reorganize IP fragments. If the fragments exceed this threshold but all fragments have not arrived within the specified time, the kernel will not reorganize the fragments. This threshold is typically used to control whether larger fragments are reorganized. The default value is 4194304 bytes (4 MB). Set mount options for the XFS file system.
- `net.ipv4.ipfrag_low_thresh`: Indicates that when the total size of IP fragments is below this threshold, the kernel will wait as long as possible for more fragments to arrive, to allow for larger reorganizations. This threshold is used to minimize unfinished reorganization operations and improve system performance. The default value is 3145728 bytes (3 MB).
- `net.ipv4.ipfrag_time` is a kernel parameter that controls the IP fragment reassembly

timeout. The default value is 30.

It is recommended to set the above parameters to the following values:

```
net.ipv4.ipfrag_high_thresh = 41943040
net.ipv4.ipfrag_low_thresh = 31457280
net.ipv4.ipfrag_time = 60
```

System memory

- If the server memory exceeds 64 GB, the following parameters are recommended in the `/etc/sysctl.conf` configuration file:

```
vm.dirty_background_ratio = 0
vm.dirty_ratio = 0
vm.dirty_background_bytes = 1610612736 # 1.5 GB
vm.dirty_bytes = 4294967296 # 4 GB
```

- If the server memory is less than 64 GB, you do not need to set `vm.dirty_background_bytes` or `vm.dirty_bytes`. It is recommended to set the following parameters in the `/etc/sysctl.conf` configuration file:

```
vm.dirty_background_ratio = 3
vm.dirty_ratio = 10
```

- To deal with emergency situations when the system is under memory pressure, it is recommended to add the `vm.min_free_kbytes` parameter to the `/etc/sysctl.conf` configuration file to control the amount of available memory reserved by the system. It is recommended to set `vm.min_free_kbytes` to 3% of the system's physical memory, with the following command:

```
awk 'BEGIN {OFMT = "%.0f";} /MemTotal/ {print "vm.min_free_kbytes =", $2 * .03;}' /proc/meminfo /etc/sysctl.conf
```

- The setting of `vm.min_free_kbytes` is not recommended to exceed 5% of the system's physical memory.

Resource limit settings

Edit the `/etc/security/limits.conf` file and add the following content, which will limit the amount of hardware and software resources.

```
*soft nofile 524288
*hard nofile 524288
*soft nproc 131072
*hard nproc 131072
```

CORE DUMP settings

1. Add the following parameter to the `/etc/sysctl.conf` configuration file:

```
kernel.core_pattern=/var/core/core.%h.%t
```

2. Run the following command to make the configuration effective:

```
sysctl -p
```

3. Add the following parameter to `/etc/security/limits.conf`:

```
soft core unlimited
```

Set mount options for the XFS file system

XFS is the file system for the data directory of SynxDB. XFS has the following mount options:

```
rw, nodev, noatime, inode64
```

You can set up XFS file mounting in the `/etc/fstab` file. See the following commands. You need to choose the file path according to the actual situation:

```
mkdir -p /data0/
mkfs.xfs -f /dev/vdc
echo "/dev/vdc /data0 xfs rw,nodev,noatime,nobarrier,inode64 0 0" /etc/fstab
```

(continues on next page)

(continued from previous page)

```
mount /data0
chown -R gpadmin:gpadmin /data0/
```

Run the following command to check whether the mounting is successful:

```
df -h
```

Blockdev value

The blockdev value for each disk device file should be 16384. To verify the blockdev value of a disk device, you can use the following command:

```
sudo /sbin/blockdev --getra <devname>
```

For example, to verify the blockdev value of the hard disk of the example server:

```
sudo /sbin/blockdev --getra /dev/vdc
```

To modify the blockdev value of a device file, you can use the following command:

```
sudo /sbin/blockdev --setra <bytes> <devname>
```

For example, to modify the blockdev value of the hard disk of the example server:

```
sudo /sbin/blockdev --setra 16384 /dev/vdc
```

I/O scheduling policy settings for disks

The disk type, operating system, and scheduling policy of SynxDB are as follows:

Refer to the following command to modify the scheduling policy. Note that this command is only a temporary modification, and the modification will become invalid after the server is restarted.

```
echo schedulername /sys/block/<devname>/queue/scheduler
```

For example, to temporarily modify the disk I/O scheduling policy of the example server:

Storage device type	OS	Recommended scheduling policy
NVMe	RHEL 7	none
	RHEL 8	none
	Ubuntu	none
SSD	RHEL 7	noop
	RHEL 8	none
	Ubuntu	none
Other	RHEL 7	deadline
	RHEL 8	mq-deadline
	Ubuntu	mq-deadline

```
echo deadline /sys/block/vdc/queue/scheduler
```

To permanently modify the scheduling policy, use the system utility grubby. After using grubby, the modification takes effect immediately after you restart the server. The sample command is as follows:

```
grubby --update-kernel=ALL --args="elevator=deadline"
```

You can view the kernel parameter settings by using the following command:

```
grubby --info=ALL
```

Disable Transparent Huge Pages (THP)

You need to disable Transparent Huge Pages (THP), because it reduces SynxDB performance. The command is as follows:

```
grubby --update-kernel=ALL --args="transparent_hugepage=never"
```

Check the status of THP:

```
cat /sys/kernel/mm/*transparent_hugepage/enabled
```

Disable IPC object deletion

Disable IPC object deletion by setting the value of RemoveIPC to no. You can set this parameter in SynxDB's /etc/systemd/logind.conf file.

```
RemoveIPC=no
```

After disabling it, run the following command to restart the server to make the disabling setting effective:

```
service systemd-logind restart
```

SSH connection threshold

To set the SSH connection threshold, you need to modify the /etc/ssh/sshd_config configuration file's MaxStartups and MaxSessions parameters. Both of the following writing methods are acceptable.

```
MaxStartups 200  
MaxSessions 200
```

```
MaxStartups 10:30:200  
MaxSessions 200
```

Run the following command to restart the server to make the setting take effect:

```
service sshd restart
```

Clock synchronization

SynxDB requires the clock synchronization to be configured for all hosts, and the clock synchronization service should be started when the host starts. You can choose one of the following synchronization methods:

- Use the coordinator node's time as the source, and other hosts synchronize the clock of the coordinator node host.

- Synchronize clocks using an external clock source.

The example in this document uses an external clock source for synchronization, that is, adding the following configuration to the `/etc/chrony.conf` configuration file:

```
# Use public servers from the pool.ntp.org project
# Please consider joining the pool (http://www.pool.ntp.org/join.html)
server 0.centos.pool.ntp.org iburst
```

After setting, you can run the following command to check the clock synchronization status:

```
systemctl status chronyd
```

Step 2: Install SynxDB via RPM package

1. Download the SynxDB RPM package to the `gpadmin` home directory `/home/gpadmin/`:

```
wget -P /home/gpadmin <download address>
```

2. Install the RPM package in the `/home/gpadmin` directory.

When running the following command, you need to replace `<RPM package path>` with the actual RPM package path, and execute it as the `root` user. During installation, the default installation directory `/usr/local/synxdb/` will be automatically created.

```
cd /home/gpadmin
yum install <RPM package path>
```

3. Grant the `gpadmin` user permission for the installation directory:

```
chown -R gpadmin:gpadmin /usr/local
chown -R gpadmin:gpadmin /usr/local/synxdb*
```

4. Configure local SSH login for the node. As the `gpadmin` user:

```
ssh-keygen
ssh-copy-id localhost
ssh `hostname` # Ensure the local SSH login works properly
```

Step 3: Deploy SynxDB with a single computing node

Use the scripting tool `gpdemo` to quickly deploy SynxDB. `gpdemo` is included in the RPM package and will be installed in the `GPHOME/bin` directory along with the configuration scripts (`gpinitSystem`, `gpstart`, `gpstop`, etc.), and it supports quickly deploying SynxDB with a single computing node. For more details about this tool, refer to [gpdemo](#).

In the above *Set mount options for the XFS file system*, the XFS file system's data directory is mounted on `/data0`. The following commands deploy a single-computing-node cluster in this data directory:

```
cd /data0
NUM_PRIMARY_MIRROR_PAIRS=0 gpdemo # Uses the gpdemo tool
```

When `gpdemo` is running, a new warning will be output: [WARNING] :–SinglenodeMode has been enabled, no segment will be created., which indicates that SynxDB is currently being deployed in the single-computing-node mode.

Common issues

How to check the deployment mode of a cluster

Perform the following steps to confirm the deployment mode of the current SynxDB cluster:

1. Connect to the coordinator node.
2. Execute `SHOW gp_role;` to view the operating mode of the cluster.
 - If the result returns `utility`, it indicates that the cluster is in Utility mode, which is the maintenance mode where only the coordinator node is available.

At this point, continue to run `SHOW gp_internal_is_singlenode;` to see whether the cluster is in the single-computing-node mode.

- If the result returns `on`, it indicates that the current cluster is in the single-computing-node mode.
- If the result returns `off`, it indicates that the current cluster is in regular utility maintenance mode.

- If the result returns `dispatch`, it indicates that the current cluster is a regular cluster containing segment nodes. You can further confirm the number of segments, their status, ports, data directories, and other information by running `SELECT * FROM gp_segment_configuration;`.

Where is the data directory

`gpdemo` automatically creates a data directory in the current path (`$PWD`). For the single-computing-node deployment:

- The default directory of the coordinator is `./datadirs/singlenodedir`.
- The default directory of the coordinator standby node is `./datadirs/standby`.

How it works

When you are deploying SynxDB in the single-computing-node mode, the deployment script `gpdemo` writes `gp_internal_is_singlenode = true` to the configuration file `postgresql.conf` and starts a coordinator and a coordinator standby node with the `gp_role = utility` parameter setting. All data is written locally without a segment or data distribution.

User-behavior changes

In the single-computing-node mode, the product behavior of SynxDB has the following changes. You should pay attention to these changes before performing related operations:

- When you execute `CREATE TABLE` to create a table, the `DISTRIBUTED BY` clause no longer takes effect. A warning is output: `WARNING: DISTRIBUTED BY clause has no effect in singlenode mode.`
- The `SCATTER BY` clause of the `SELECT` statement is no longer effective. A warning is output: `WARNING: SCATTER BY clause has no effect in singlenode mode.`
- Other statements that are not supported (for example, `ALTER TABLE SET DISTRIBUTED BY`) are declined with an error.

- The lock level of UPDATE and DELETE statements will be reduced from ExclusiveLock to RowExclusiveLock to provide better concurrency performance, because there is only a single node without global transactions or global deadlocks. This behavior is consistent with PostgreSQL.

Chapter 3

Load Data

3.1 Data Loading Overview

SynxDB loads external data by converting it into external tables via loading tools. Then, it reads from or writes to these external tables to complete the external data loading process.

Data loading process

The general process of loading data into SynxDB is as follows:

1. Evaluate the data loading scenario (such as data source location, data types, and data volume) and choose the appropriate loading tool.
2. Configure and activate the loading tool.
3. Create an external table, specifying the loading tool protocol, data source address, and data format in the `CREATE EXTERNAL TABLE` statement. For details, see [*Load External Data Using Foreign Table*](#).
4. Once the external table is created, you can directly query the data using the `SELECT` statement or load the data into a table using the `INSERT INTO SELECT` statement.

Loading tools and scenarios

SynxDB provides multiple data loading solutions, allowing you to choose different methods based on the data source.

Loading method/tool	Data source	Data format	Parallel loading
<i>Load Data Using COPY</i>			
	Local file system	<ul style="list-style-type: none"> • Coordinator node (for single files) • Segment nodes (for multiple files) 	No
<i>Load Data Using the file:// Protocol</i>	Local file system (local Segment nodes, only accessible by superusers)	<ul style="list-style-type: none"> • TXT • CSV 	Yes
<i>gpfdist</i>	Local host files or files accessible over the intranet	<ul style="list-style-type: none"> • TXT • CSV • Any delimited text format supported by the FORMAT clause • XML and JSON (converted to text via YAML configuration) 	Yes
Bulk loading using <i>gupload</i> (using <i>gpfdist</i> as the underlying component)	Local host files or files accessible over the intranet	<ul style="list-style-type: none"> • TXT • CSV • Any delimited text format supported by the FORMAT clause • XML and JSON (converted to text via YAML configuration) 	Yes
<i>Create external Web tables</i>	Data fetched from web services or any source accessible via command line	<ul style="list-style-type: none"> • TXT • CSV 	Yes
<i>Kafka FDW and Kafka Connector</i> <i><load-data/load-data-using-kafka-connecto</i>	Kafka	<ul style="list-style-type: none"> • CSV • JSON 	Yes
<i>Data Lake Connector</i>	<ul style="list-style-type: none"> • Public cloud object storage (for example, Amazon S3, QingCloud, Alibaba Cloud, Huawei Cloud, Tencent Cloud) • Hadoop storage 	<ul style="list-style-type: none"> • CSV • TEXT • ORC • PARQUET 	Yes
<i>Hive Connector</i> with <code>data_lake_fdw</code>	Hive data warehouse	<ul style="list-style-type: none"> • CSV • TEXT • ORC • PARQUET • Iceberg • Hudi 	Yes
<i>mysql_fdw</i>	MySQL	MySQL table data	Yes

3.2 Load Data from Local Files

Load Data Using COPY

COPY FROM copies data from a file or standard input in a local file system into a table and appends the data to the table contents. COPY is non-parallel: data is loaded in a single process using the SynxDB coordinator instance. Using COPY is only recommended for very small data files.

The COPY source file must be accessible to the postgres process on the coordinator host. Specify the COPY source file name relative to the data directory on the coordinator host, or specify an absolute path.

SynxDB copies data from STDIN or STDOUT using the connection between the client and the coordinator server.

Load from a file

The COPY command requests the postgres backend to open the specified file, to read it and append it to the target table. To be able to read the file, the backend needs to have read permissions on the file, and you need to specify the file name using an absolute path on the coordinator host, or using a relative path to the coordinator data directory.

```
COPY <table_name> FROM </path/to/filename>;
```

Load from STDIN

To avoid the problem of copying the data file to the coordinator host before loading the data, COPY FROM STDIN uses the Standard Input channel and feeds data directly into the postgres backend. After the COPY FROM STDIN command starts, the backend will accept lines of data until a single line only contains a backslash-period (\ .).

```
COPY <table_name> FROM <STDIN>;
```

Load data using \copy in psql

Do not confuse the psql \copy command with the COPY SQL command. The \copy invokes a regular COPY FROM STDIN and sends the data from the psql client to the backend. Therefore, any file must locate on the host where the psql client runs, and must be accessible to the user which runs the client.

To avoid the problem of copying the data file to the coordinator host before loading the data, COPY FROM STDIN uses the Standard Input channel and feeds data directly into the postgres backend. After the COPY FROM STDIN command started, the backend will accept lines of data until a single line only contains a backslash-period (\ .). psql is wrapping all of this into the handy \copy command.

```
\copy <table_name> FROM <filename>;
```

Input format

COPY FROM accepts a FORMAT parameter, which specifies the format of the input data. The possible values are TEXT, CSV (Comma Separated Values), and BINARY.

```
COPY <table_name> FROM </path/to/filename> WITH (FORMAT csv);
```

The FORMAT csv will read comma-separated values. The FORMAT text by default uses tabulators to separate the values, the DELIMITER option specifies a different character as value delimiter.

```
COPY <table_name> FROM </path/to/filename> WITH (FORMAT text, DELIMITER '|');
```

By default, the default client encoding is used, and you can change this using the ENCODING option. This is useful if data is coming from another operating system.

```
COPY <table_name> FROM </path/to/filename> WITH (ENCODING 'latin1');
```

Load Data Using gpfdist

To load data from local host files or files accessible via internal network, you can use the `gpfdist` protocol in the `CREATE EXTERNAL TABLE` statement. `gpfdist` is a file server utility that runs on a host other than the SynxDB coordinator or standby coordinator. `gpfdist` serves files from a directory on the host to SynxDB segments.

When external data is served by `gpfdist`, all segments in the SynxDB system can read or write external table data in parallel.

The supported data formats are:

- CSV and TXT
- Any delimited text format supported by the `FORMAT` clause

The general procedure for loading data using `gpfdist` is as follows:

1. Install `gpfdist` on a host other than the SynxDB coordinator or standby coordinator. See [*Step 1. Install gpfdist*](#).
2. Start `gpfdist` on the host. See [*Step 2. Start and stop gpfdist*](#).
3. Create an external table using the `gpfdist` protocol. See [*Step 3. Use gpfdist with external tables to load data*](#).

Step 1. Install gpfdist

`gpfdist` is installed in `$GPHOME/bin` of your SynxDB coordinator host installation. Run `gpfdist` on a machine other than the SynxDB coordinator or standby coordinator, such as on a machine devoted to ETL processing. Running `gpfdist` on the coordinator or standby coordinator can have a performance impact on query execution.

Step 2. Start and stop gpfdist

You can start `gpfdist` in your current directory location or in any directory that you specify. The default port is 8080.

From your current directory, type:

```
gpfdist &
```

From a different directory, specify the directory from which to serve files, and optionally, the HTTP port to run on.

To start `gpfdist` in the background and log output messages and errors to a log file:

```
$ gpfdist -d /var/load_files -p 8081 -l /home/`gpadmin`/log &
```

For multiple `gpfdist` instances on the same ETL host, use a different base directory and port for each instance. For example:

```
$ gpfdist -d /var/load_files1 -p 8081 -l /home/`gpadmin`/log1 &
$ gpfdist -d /var/load_files2 -p 8082 -l /home/`gpadmin`/log2 &
```

The logs are saved in `/home/gpadmin/log`.

Tip

To stop `gpfdist` when it is running in the background:

1. First find its process id:

```
$ ps -ef | grep gpfdist
```

2. Then stop the process, for example (where 3457 is the process ID in this example):

```
$ ps -ef | grep gpfdist
```

Step 3. Use gpfdist with external tables to load data

The following examples show how to use gpfdist when creating an external table to load data into SynxDB.

 **Note**

When using IPv6, always enclose the numeric IP addresses in square brackets.

Example 1 - Run single gpfdist instance on a single-NIC machine

Creates a readable external table, ext_expenses, using the gpfdis``t protocol. The files are formatted with a pipe (` `|) as the column delimiter.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
    date date, amount float4, category text, desc1 text )
LOCATION ('gpfdist://etlhost-1:8081/*.txt',
    'gpfdist://etlhost-2:8081/*.txt')
FORMAT 'TEXT' ( DELIMITER '|'|NULL ' ' );
```

Example 2 —Run multiple gpfdist instances

Creates a readable external table, ext_expenses, using the gpfdist protocol from all files with the txt extension. The column delimiter is a pipe (|) and NULL (' ') is a space.

```
# CREATE EXTERNAL TABLE ext_expenses ( name text,
    date date, amount float4, category text, desc1 text )
LOCATION ('gpfdist://etlhost-1:8081/*.txt',
    'gpfdist://etlhost-2:8081/*.txt')
FORMAT 'TEXT' ( DELIMITER '|'|NULL ' ' ) ;
```

Example 3 —Single gpfdist instance with error logging

Uses the gpfdist protocol to create a readable external table, ext_expenses, from all files with the txt extension. The column delimiter is a pipe (|) and NULL (' ') is a space.

Access to the external table is single row error isolation mode. Input data formatting errors are captured internally in SynxDB with a description of the error. You can view the errors, fix the issues, and then reload the rejected data. If the error count on a segment is greater than 5 (the SEGMENT REJECT LIMIT value), the entire external table operation fails and no rows are processed.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
    date date, amount float4, category text, desc1 text )
  LOCATION ('gpfdist://etlhost-1:8081/*.txt',
             'gpfdist://etlhost-2:8082/*.txt')
  FORMAT 'TEXT' ( DELIMITER '|' NULL ' ')
  LOG ERRORS SEGMENT REJECT LIMIT 5;
```

To create the readable ext_expenses table from CSV-formatted text files:

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
    date date, amount float4, category text, desc1 text )
  LOCATION ('gpfdist://etlhost-1:8081/*.txt',
             'gpfdist://etlhost-2:8082/*.txt')
  FORMAT 'CSV' ( DELIMITER ',' )
  LOG ERRORS SEGMENT REJECT LIMIT 5;
```

Example 4 - Create a writable external table with gpfdist

Creates a writable external table, . sales_out, that uses gpfdist to write output data to the file sales.out. The column delimiter is a pipe (|) and NULL (' ') is a space. The file will be created in the directory specified when you started the gpfdist file server.

```
=# CREATE WRITABLE EXTERNAL TABLE sales_out (LIKE sales)
  LOCATION ('gpfdist://etl1:8081/sales.out')
  FORMAT 'TEXT' ( DELIMITER '|' NULL ' ')
  DISTRIBUTED BY (txn_id);
```

About gpfdist

Before using gpfdist, you might need to know how it works. This section provides an overview of gpfdist and how to use it with external tables.

About gpfdist and external tables

The gpfdist file server utility is located in the \$GPHOME/bin directory on your SynxDB coordinator host and on each segment host. When you start a gpfdist instance you specify a listen port and the path to a directory containing files to read or where files are to be written. For example, this command runs gpfdist in the background, listening on port 8801, and serving files in the /home/gpadmin/external_files directory:

```
$ gpfdist -p 8801 -d /home/gpadmin/external_files &
```

The CREATE EXTERNAL TABLE command LOCATION clause connects an external table definition to one or more gpfdist instances. If the external table is readable, the gpfdist server reads data records from files from in specified directory, packs them into a block, and sends the block in a response to a SynxDB segment's request. The segments unpack rows that they receive and distribute the rows according to the external table's distribution policy. If the external table is a writable table, segments send blocks of rows in a request to gpfdist and gpfdist writes them to the external file.

External data files can contain rows in CSV format or any delimited text format supported by the FORMAT clause of the CREATE EXTERNAL TABLE command.

For readable external tables, gpfdist uncompresses gzip (.gz) and bzip2 (.bz2) files automatically. You can use the wildcard character (*) or other C-style pattern matching to denote multiple files to read. External files are assumed to be relative to the directory specified when you started the gpfdist instance.

About gpfdist setup and performance

You can run `gpfdist` instances on multiple hosts and you can run multiple `gpfdist` instances on each host. This allows you to deploy `gpfdist` servers strategically so that you can attain fast data load and unload rates by utilizing all of the available network bandwidth and SynxDB's parallelism.

- Allow network traffic to use all ETL host network interfaces simultaneously. Run one instance of `gpfdist` for each interface on the ETL host, then declare the host name of each NIC in the `LOCATION` clause of your external table definition (see [*Example 1 - Run single gpfdist instance on a single-NIC machine*](#)).
- Divide external table data equally among multiple `gpfdist` instances on the ETL host. For example, on an ETL system with two NICs, run two `gpfdist` instances (one on each NIC) to optimize data load performance and divide the external table data files evenly between the two `gpfdist` servers.

Load Data Using the file:// Protocol

The `file://` protocol is a SynxDB protocol that allows you to load data from a local segment host server file into SynxDB.

The `file://` protocol is used in a URI that specifies the location of an operating system file. External tables that you create that specify the `file://` protocol are read-only tables.

The URI includes the host name, port, and path to the file. Each file must locate on a segment host in a location accessible by the SynxDB superuser (`gpadmin`). The host name used in the URI must match a segment host name registered in the `gp_segment_configuration` system catalog table.

The `LOCATION` clause can have multiple URIs, as shown in [Usage examples](#).

The number of URIs you specify in the `LOCATION` clause is the number of segment instances that will work in parallel to access the external table. For each URI, SynxDB assigns a primary segment on the specified host to the file. For maximum parallelism when loading data, divide the data into as many equally sized files as you have primary segments. This ensures that all segments participate in the load. The number of external files per segment host cannot exceed the number of primary segment instances on that host. For example, if your array has 4 primary segment instances per segment host, you can place 4 external files on each segment host. Tables based on the `file://` protocol can only be readable tables.

The system view `pg_max_external_files` shows how many external table files are permitted per external table. This view lists the available file slots per segment host when using the `file://` protocol. The view is only applicable for the `file://` protocol. For example:

```
SELECT * FROM pg_max_external_files;
```

Usage examples

Load multiple files in CSV format with header rows:

Creates a readable external table, `ext_expenses`, using the `file` protocol. The files are CSV format and have a header row.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
    date date, amount float4, category text, desc1 text )
LOCATION ('file://filehost/data/international/*',
    'file://filehost/data/regional/*',
    'file://filehost/data/supplement/*.csv')
FORMAT 'CSV' (HEADER);
```

```
=# CREATE EXTERNAL TABLE ext_expenses (
    name text, date date, amount float4, category text, desc1 text )
LOCATION ('file://host1:5432/data/expense/*.csv',
    'file://host2:5432/data/expense/*.csv',
    'file://host3:5432/data/expense/*.csv')
FORMAT 'CSV' (HEADER);
```

Load Data Using gupload

The `gupload` utility of SynxDB loads data using readable external tables and the SynxDB parallel file server (`gpfdist`). It handles parallel file-based external table setup and allows users to configure their data format, external table definition, and `gpfdist` setup in a single configuration file.

i Note

In `gupload`, MERGE and UPDATE operations are not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes " " to identify the column.

To use gupload

1. Ensure that your environment is set up to run `gupload`. Some dependent files from your SynxDB installation are required, such as `gpfdist` and Python 3, as well as network access to the SynxDB segment hosts. `gupload` also requires that you install the following packages:

```
pip install psycopg2 pyyaml
```

2. Create your load control file. This is a YAML-formatted file that specifies the SynxDB connection information, `gpfdist` configuration information, external table options, and data format.

For example:

```
---
VERSION: 1.0.0.1
DATABASE: ops
USER: gpadmin
HOST: cdw-1
PORT: 5432
GUPLOAD:
  INPUT:
    - SOURCE:
      LOCAL_HOSTNAME:
```

(continues on next page)

(continued from previous page)

```
- etl1-1
- etl1-2
- etl1-3
- etl1-4
PORT: 8081
FILE:
- /var/load/data/*
- COLUMNS:
- name: text
- amount: float4
- category: text
- descr: text
- date: date
- FORMAT: text
- DELIMITER: '|'
- ERROR_LIMIT: 25
- LOG_ERRORS: true
OUTPUT:
- TABLE: payables.expenses
- MODE: INSERT
PRELOAD:
- REUSE_TABLES: true
# SQL:
# - BEFORE: "INSERT INTO audit VALUES('start', current_timestamp)"
# - AFTER: "INSERT INTO audit VALUES('end', current_timestamp)"
```

3. Run gupload, passing in the load control file. For example:

```
gupload -f my_load.yml
```

3.3 Load External Data Using Foreign Table

SynxDB allows you to access data stored in remote data sources using foreign tables. You can use foreign tables to connect to other databases (such as Oracle and MySQL) or external data sources (such as CSV files) through a foreign data wrapper (FDW).

Use foreign table

You can create a foreign table using the following command:

 **Note**

Before creating a foreign table, you need to create a foreign server first.

```
CREATE FOREIGN TABLE <external_table_name> (
    col_1 data_type,
    col_2 data_type,
    ...
) SERVER <server_name>
OPTIONS (<option_name> '<option_value>');
```

For example:

```
CREATE FOREIGN TABLE my_foreign_table (
    id INTEGER,
    name TEXT
) SERVER remote_data
OPTIONS (table_name 'external_table');
```

In this example, the table `my_foreign_table` is the foreign table created in the local database, while the actual data is stored in a remote table called `external_table`.

Create foreign table using the LIKE clause

You can use the `LIKE` clause to quickly create a foreign table based on the structure of an existing table. By using this clause, you can define a foreign table without explicitly listing the table's structure.

Note

Foreign tables created using `LIKE` do not inherit the distribution settings of the existing table. These settings should be defined when you create the new foreign table.

The following example shows how to use the `LIKE` clause to create a foreign table:

```

CREATE FOREIGN DATA WRAPPER dummy; -- Creates a foreign data wrapper.
CREATE SERVER s0 FOREIGN DATA WRAPPER dummy; -- Creates a foreign server.
CREATE TABLE ft_source_table(a INT, b INT, c INT) DISTRIBUTED BY (b); -- 
Creates a table 'ft_source_table'.
CREATE FOREIGN TABLE my_foreign_table (LIKE ft_source_table) SERVER s0; -- 
Creates a Foreign Table based on 'ft_source_table'.
\dd+ ft_like
                                         Foreign table "public.ft_like"
Column |  Type   | Collation | Nullable | Default | FDW options | Storage |
Stats target | Description
-----+-----+-----+-----+-----+-----+-----+
-----+-----+
a    | integer |          |          |          |          | plain   |
      |
b    | integer |          |          |          |          | plain   |
      |
c    | integer |          |          |          |          | plain   |
      |
Server: s0

```

Query a foreign table

After creating a foreign table, you can query it just like a local table:

```
SELECT * FROM my_foreign_table;
```

This query will retrieve remote data in real-time over the network.

3.4 Load Data from Web Services

In SynxDB, to load data from web services or from any source accessible by command lines, you can create external web tables. The supported data formats are TEXT and CSV.

External web tables allow SynxDB to treat dynamic data sources like regular database tables. Because web table data can change as a query runs, the data is not rescannable.

`CREATE EXTERNAL WEB TABLE` creates a web table definition. You can define command-based or URL-based external web tables. The definition forms are different. Do not mix command-based and URL-based definitions.

Command-based external web tables

In SynxDB, the output of a shell command or script defines command-based web table data. Specify the command in the `EXECUTE` clause of `CREATE EXTERNAL WEB TABLE`. The data is current as of the time the command runs. The `EXECUTE` clause runs the shell command or script on the specified coordinator, and/or segment host or hosts. The command or script must reside on the hosts corresponding to the host(s) defined in the `EXECUTE` clause.

By default, the command is run on segment hosts when active segments have output rows to process. For example, if each segment host runs 4 primary segment instances that have output rows to process, the command runs 4 times per segment host. You can optionally limit the number of segment instances that run the web table command. All segments included in the web table definition in the `ON` clause run the command in parallel.

The command that you specify in the external table definition is run from the database and cannot access environment variables from `.bashrc` or `.profile`. Set environment variables in the `EXECUTE` clause. For example:

```
=# CREATE EXTERNAL WEB TABLE output (output **text**)
  EXECUTE 'PATH=/home/gpadmin/programs; export PATH; myprogram.sh'
  FORMAT 'TEXT';
```

Scripts must be executable by the `gpadmin` user and reside in the same location on the coordinator or segment hosts.

The following command defines a web table that runs a script. The script runs on each segment

host where a segment has output rows to process.

```
=# CREATE EXTERNAL WEB TABLE log_output
  (linenum **int**, message **text**)
  EXECUTE '/var/load_scripts/get_log_data.sh' ON HOST
  FORMAT 'TEXT' (DELIMITER '|');
```

URL-based external web tables

In SynxDB, a URL-based web table accesses data from a web server using the HTTP protocol. Web table data is dynamic; the data is not rescannable.

Specify the LOCATION of files on a web server using `http://`. The web data file(s) must reside on a web server that SynxDB segment hosts can access. The number of URLs specified corresponds to the number of segment instances that work in parallel to access the web table. For example, if you specify 2 external files to a SynxDB system with 8 primary segments, 2 of the 8 segments access the web table in parallel at query runtime.

The following sample command defines a web table that gets data from several URLs.

```
=# CREATE EXTERNAL WEB TABLE ext_expenses (name **text**,
  **date** **date**, amount float4, category **text**, description **text**)
  LOCATION (
    'http://intranet.company.com/expenses/sales/file.csv',
    'http://intranet.company.com/expenses/exec/file.csv',
    'http://intranet.company.com/expenses/finance/file.csv',
    'http://intranet.company.com/expenses/ops/file.csv',
    'http://intranet.company.com/expenses/marketing/file.csv',
    'http://intranet.company.com/expenses/eng/file.csv'
  )
  FORMAT 'CSV' ( HEADER );
```

3.5 Load Data from Kafka Using Kafka FDW

Kafka Foreign Data Wrapper (FDW) enables SynxDB to connect directly to Apache Kafka, allowing it to read and operate on Kafka data as external tables. This integration allows SynxDB users to process real-time Kafka data more efficiently, flexibly, and reliably, greatly boosting data processing and business operations.

SynxDB supports using Kafka FDW to create external tables and import data.

Basic usage

- Create the kafka_fdw extension:

```
postgres=# CREATE EXTENSION kafka_fdw;
CREATE EXTENSION
```

- Create an external server and specify Kafka's cluster address:

```
CREATE SERVER kafka_server
FOREIGN DATA WRAPPER kafka_fdw
OPTIONS (mpp_execute 'all segments', brokers 'localhost:9092');
```

- Create user mapping:

```
CREATE USER MAPPING FOR PUBLIC SERVER kafka_server;
```

- Create an external table:

When creating an external table, you need to specify two metadata columns: `partition` and `offset`, which identify the partition and offset of messages in a Kafka topic. Here is an example:

```
CREATE FOREIGN TABLE kafka_test (
    part int OPTIONS (partition 'true'),
    offs bigint OPTIONS (offset 'true'),
    some_int int,
    some_text text,
    some_date date,
```

(continues on next page)

(continued from previous page)

```

    some_time timestamp
)
SERVER kafka_server OPTIONS
  (format 'csv', topic 'contrib_regress_csv', batch_size '1000', buffer_
delay '1000');

```

Parameter description:

- batch_size: The size of data read from Kafka at once.
- buffer_delay: The timeout for getting data from Kafka.

Supported data formats

Currently, CSV and JSON data formats are supported.

Query

You can specify the message partition and offset in your query by using the `partition` or `offset` column condition:

```
SELECT * FROM kafka_test WHERE part = 0 AND offs > 1000 LIMIT 60;
```

You can also specify multiple conditions:

```
SELECT * FROM kafka_test WHERE (part = 0 AND offs > 100) OR (part = 1 AND offs
> 300) OR (part = 3 AND offs > 700);
```

Message producer

Currently, Kafka FDW supports inserting data into external tables, which acts as a message producer for Kafka. You only need to use the `INSERT` statement.

```

INSERT INTO kafka_test(part, some_int, some_text)
VALUES
  (0, 5464565, 'some text goes into partition 0'),
  (1, 5464565, 'some text goes into partition 1'),

```

(continues on next page)

(continued from previous page)

```
(0, 5464565, 'some text goes into partition 0'),
(3, 5464565, 'some text goes into partition 3'),
(NULL, 5464565, 'some text goes into partition selected by kafka');
```

When inserting data, you can specify `partition` to specify which partition to insert into.

Data import

To use Kafka FDW for data import, you can create custom functions, such as the `INSERT INTO SELECT` statement. The basic principle is to fetch all data from the external table and insert it into the target table sequentially.

Here is a simple example, which you can modify according to your needs:

```
CREATE OR REPLACE FUNCTION import_kafka_data(
    src_table_name text,
    dest_table_name text,
    dest_table_columns text []
) RETURNS void AS $$

DECLARE
    current_row RECORD;
    current_part integer;
    current_offs bigint;
    max_off bigint;
    import_progress_table_name text;
    max_off_result bigint;

BEGIN

    import_progress_table_name := src_table_name || '_import_progress';

    -- Creates progress record table.
    EXECUTE FORMAT('CREATE TABLE IF NOT EXISTS %I (part integer PRIMARY KEY,
offs bigint NOT NULL)', import_progress_table_name);

    -- The number of partitions in the topic table might change, so
    -- reinitialize before each import.
    EXECUTE FORMAT('INSERT INTO %I SELECT DISTINCT part, 0 FROM %I ON CONFLICT
(part) DO NOTHING', import_progress_table_name, src_table_name);
```

(continues on next page)

(continued from previous page)

```

-- Imports data partition by partition.
FOR current_row IN
    EXECUTE FORMAT('SELECT part, offs FROM %I', import_progress_table_
name)
LOOP
    current_part := current_row.part;
    current_offs := current_row.offsets;

    -- Gets the maximum offset for the current partition.
    EXECUTE FORMAT('SELECT MAX(offs) FROM %I WHERE part = %s', src_table_-
name, current_part) INTO max_off_result;
    max_off := max_off_result;

    -- Skips if there is no new data.
    IF max_off+1 = current_offs THEN
        CONTINUE;
    END IF;

    -- Imports data.
    EXECUTE FORMAT('
        INSERT INTO %I (%s)
        SELECT %s
        FROM %I
        WHERE part = %s AND offs >= %s AND offs <= %s',
        dest_table_name,
        array_to_string(dest_table_columns, ', '),
        array_to_string(dest_table_columns, ', '),
        src_table_name,
        current_part,
        current_offs,
        max_off
    );
    -- Updates import progress.
    EXECUTE FORMAT('UPDATE %I SET offs = %s WHERE part = %s', import_-
progress_table_name, max_off + 1, current_part);
END LOOP;

RETURN;

```

(continues on next page)

(continued from previous page)

```
END;  
$$ LANGUAGE plpgsql;
```

When executing the query, call this function, passing in the external table name, target table name, and the fields to be imported. Here is an example:

```
SELECT import_kafka_data('kafka_test', 'dest_table_fdw', ARRAY['some_int',  
'some_text', 'some_date', 'some_time']);
```

Scheduled import

To create a scheduled task to import data in the background, you can use the Task feature in SynxDB to execute the import function periodically.

```
CREATE TASK import_kafka_data schedule '1 seconds' AS $$SELECT import_kafka_  
data('kafka_test', 'dest_table_fdw', ARRAY['some_int', 'some_text', 'some_date'  
, 'some_time']);$$;
```

In the example above, the function to import data is called every second. This setup allows for the continuous use of Kafka FDW to import data from the source external table into the target table.

3.6 Load Data from Kafka Using Kafka Connector

SynxDB Kafka Connector is a plugin based on the [Apache Kafka Connect](#) framework. It consumes data from Kafka topics and loads it into SynxDB tables, enabling scenarios such as data loading and real-time data warehousing.

Data Formats

The Kafka message data format supports the following three formats:

- **Delimited Plain Text**

```
678,mike,abc
456,jack,zzz
```

- **JSON Format**

```
{"id": "100", "name": "zhangsan", "address": "beijing"}
{"id": "200", "name": "lisi", "address": "shanghai"}
```

- **JSON Format for Change Data Capture (CDC)**

Supports both DML (Data Manipulation Language) and DDL (Data Definition Language) operations, such as those in [debezium](#) format.

```
// insert
{
  "payload": {
    "before": null,
    "after": {
      "c1": 10,
      "c2": "abcd"
    },
    "source": {
      "version": "2.1.1.Final",
      "connector": "postgresql",
      "name": "debezium",
      "ts_ms": 1697280773034,
      "snapshot": "false",
    }
  }
}
```

(continues on next page)

(continued from previous page)

```
        "db": "postgres",
        "sequence": [
            "23413560",
            "23413848"
        ],
        "schema": "s2",
        "table": "t2",
        "txId": 771,
        "lsn": 23413848,
        "xmin": null
    },
    "op": "c",
    "ts_ms": 1697280773249,
    "transaction": null
}

}

// delete
{
    "payload": {
        "before": {
            "c1": 11,
            "c2": null
        },
        "after": null,
        "source": {
            "version": "2.1.1.Final",
            "connector": "postgresql",
            "name": "debezium",
            "ts_ms": 1697280877563,
            "snapshot": "false",
            "db": "postgres",
            "sequence": [
                "23415392",
                "23415448"
            ],
            "schema": "s2",
            "table": "t2",
            "txId": 774,
            "lsn": 23415448,
            "xmin": null
        }
    }
}
```

(continues on next page)

(continued from previous page)

```
        "xmin": null
    },
    "op": "d",
    "ts_ms": 1697280878032,
    "transaction": null
}
}
```

When the Kafka message is in JSON CDC format, it supports DML (insert/update/delete) and DDL operations. For DML operations (insert/update/delete), the connector performs the following merge actions:

- Converts an update operation into a delete+insert operation.
- Operations on the same row (with the same primary key) will be merged. For example, if a row with the same primary key is updated 10,000 times, only the final value will be sent.
- During data loading, SynxDB uses its unique `gpfdist` external table method to load the merged result data to the target with maximum throughput:
 - * For inserts, it loads the data directly into the user table via `gpfdist` external tables.
 - * For deletes, it creates a temporary table, loads the delete data into this table via `gpfdist`, and then performs a join delete operation (i.e., `DELETE FROM user_table WHERE user_table.pk = temp_table.pk`) to update the user table.
- **|product_name| Installation Directory:** SynxDB Kafka Connector utilizes SynxDB's unique `gpfdist` external table method to load data with maximum throughput. Therefore, the machine must have the SynxDB installation directory.

Installation

Preparation

Before using SynxDB Kafka Connector, make sure to prepare the following components:

- **Java JDK:** SynxDB Kafka Connector is released as a Java JAR file. Since Java is a cross-platform product, you only need to have the appropriate JDK installed on your system to run the connector. [[Download JDK here](#)].
- **Kafka:** SynxDB Kafka Connector is based on the Kafka Connector framework, so it must run within the Kafka installation directory. [[Download Kafka here](#)].
- **|product_name| Kafka Connector JAR File:** Check the available versions to download the appropriate JAR file from SynxDB.
- **Configuration File:**

```
# entry class
connector.class=cn.synxdb.kafka.connector.SynxDBSinkConnector

# task max number
tasks.max=2

# connector name
name=z1connector

# kafka topic name
topics=delimited1topic

# kafka topic message format
synxdb.topic.format=delimited_file
synxdb.format.delimiter=|


# target table
synxdb.topic.table=public.test1table

# database connection info
synxdb.url.name=jdbc:postgresql://192.168.176.110:5432/db1

synxdb.user.name=gpadmin
```

(continues on next page)

(continued from previous page)

```

synxdb.user.password=gpadmin
synxdb.database.name=db1

# directory to put gpfdist data and kafka offset
synxdb.data.dir=/xxx/synxdb_kafka_connector/data

# data flush condition, e.g. here flush after consume 10k records or 5M
bytes or 1 second
buffer.count.records=10000
buffer.size.bytes=5000000
buffer.flush.time=1

# converter
key.converter=org.apache.kafka.connect.storage.StringConverter
value.converter=org.apache.kafka.connect.storage.StringConverter

```

Installation Steps

1. Extract Kafka.

```
tar xvfz kafka_2.13-3.1.0.tgz
```

2. Edit the configuration file kafka_2.13-3.1.0/config/connect-standalone.properties.

```

# Modify the following two settings

# 1.Kafka server address
bootstrap.servers=localhost:9092

# 2.Location of the Kafka Connector JAR file
plugin.path=/xxx/synxdb_kafka_connector/synxdb-kafka-connector-0.0.12.jar

```

3. Edit synxdb-kafka-connector-standalone.properties. For example:

```

# Custom connector name
name=zyzxconnector

# Kafka topic to consume

```

(continues on next page)

(continued from previous page)

```
topics=zyzx1topic

# Data format in the Kafka topic
synxdb.topic.format=delimited_file
synxdb.format.delimiter=|


# Database connection details
synxdb.url.name=jdbc:postgresql://192.168.176.110:5432/db1

synxdb.user.name=gpadmin
synxdb.user.password=gpadmin
synxdb.database.name=db1

# Directory for intermediate data: create this directory in advance with
mkdir -p /xxx/synxdb_kafka_connector/data
synxdb.data.dir=/xxx/synxdb_kafka_connector/data
```

💡 Tip

Explanation of synxdb.data.dir

This directory will store a metadata file that tracks the Kafka message offset that has been replicated. You can manually edit this file to skip unwanted data before restarting the connector.

4. Set the |product_name| environment variables.

```
source /<installation directory>/greenplum_path.sh
```

5. Start |product_name| Kafka Connector.

```
kafka_2.13-3.1.0/bin/connect-standalone.sh -daemon kafka_2.13-3.1.0/
config/connect-standalone.properties synxdb-kafka-connector-standalone.
properties
```

Parameter Descriptions

The following section explains the parameters in the `synxdb-kafka-connector-standalone.properties` file.

- Kafka Connect Framework Parameters

Parameter	Description	Example
<code>name</code>	Connector name	<code>name=zyzxconnector</code>
<code>topics</code>	Kafka topic name	<code>topics=zyzx1topic</code>
<code>tasks.max</code>	Number of tasks to run in the Kafka Connect framework, generally not exceeding the number of partitions in the topic.	<code>tasks.max=2</code>
<code>connector.class</code>	Fixed value	<code>connector.class=cn.synxdb.kafka.connector.SynxDBSinkConnector</code>
<code>key.converter</code>	Fixed value	<code>org.apache.kafka.connect.storage.StringConverter</code>
<code>value.converter</code>	Fixed value	<code>org.apache.kafka.connect.storage.StringConverter</code>

- SynxDB Kafka Connector Parameters

- Kafka Data Format Parameters

Parameter	Description	Example
<code>synxdb.topic.format</code>	Format of the content in Kafka topics.	<code>synxdb.topic.format=delimited_file</code>
<code>synxdb.format.delimiter</code>	Delimiter for plain text data in Kafka (non-JSON).	<code>synxdb.format.delimiter=</code>
<code>synxdb.topic.table</code>	Target table name	<code>synxdb.topic.table=schema1.table1</code>

- Database Connection Parameters

Parameter	Description	Example
<code>synxdb.url.name</code>	Database JDBC connection string	<code>synxdb.url.name=jdbc:postgresql://192.168.176.110:5432/db1</code>
<code>synxdb.user.name</code>	Database username	<code>synxdb.user.name=gpadmin</code>
<code>synxdb.user.password</code>	Database user password	<code>synxdb.user.password=gpadmin</code>
<code>synxdb.database.name</code>	Database name	<code>synxdb.database.name=db1</code>

- Data Loading Real-time Parameters

Parameter	Description	Example
<code>buffer.count.records</code>	Number of records to consume from Kafka before loading data into the database.	<code>buffer.count.records=10000</code>
<code>buffer.size.bytes</code>	Size of consumed records from Kafka before loading data into the database.	<code>buffer.size.bytes=5000000</code>
<code>buffer.flush.time</code>	Time to wait before loading data into the database.	<code>buffer.flush.time=1</code>

- o Working Directory Configuration

Parameter	Description	Example
<code>synxdb.dir</code>	<p>Working directory used during operation. This directory will store:</p> <ul style="list-style-type: none"> * Kafka topic offsets (important, do not delete). * Other temporary files (will be deleted after operation). 	<code>synxdb.data.dir=/xxx/synxdb_kafka_connector/data</code>
<code>synxdb.ext.table.log.errors</code>	Whether to include a LOG ERRORS clause when creating the <code>gpfdist</code> external table. Useful for skipping invalid data during import into SynxDB.	<code>synxdb.ext.table.log.errors=LOG ERRORS SEGMENT REJECT LIMIT 5</code>
<code>synxdb.table.upsert</code>	Specify if the <code>INSERT ... SELECT ... FROM <gpfdist external table></code> statement should include an upsert clause. If present, it updates existing rows; otherwise, it performs a direct insert.	<p>Example without upsert: <code>insert into public.t1 select * from public_t1_0_ins_ext.</code></p> <p>Example with upsert: <code>insert into public.t1 select * from public_t1_0_ins_ext ON CONFLICT (c1) DO UPDATE SET c2 = EXCLUDED.c2, c3 = EXCLUDED.c3.</code></p>

Parameter Descriptions for `synxdb.topic.format`

The `synxdb.topic.format` parameter defines the format of data in the Kafka topic. The valid values and their descriptions are:

- **delimited_file:** Data with delimiters.

- o **Delimiter Example:** When `synxdb.format.delimiter=|`, the Kafka topic messages appear as:

```
678 | mike | abc
456 | jack | zzz
```

- o **Alternate Delimiter:** When `synxdb.format.delimiter=@`, the Kafka topic messages appear as:

```
678@mike@abc
456@jack@zzz
```

- **debeziumjson:** Data in debezium format.

```
// insert
{
  "payload": {
    "before": null,
    "after": {
      "c1": 10,
      "c2": "abcd"
    },
    "source": {
      "version": "2.1.1.Final",
      "connector": "postgresql",
      "name": "debezium",
      "ts_ms": 1697280773034,
      "snapshot": "false",
      "db": "postgres",
      "sequence": [
        "23413560",
        "23413848"
      ],
      "schema": "s2",
      "table": "t2",
      "txId": 771,
      "lsn": 23413848,
      "xmin": null
    },
    "op": "c",
    "ts_ms": 1697280773249,
    "transaction": null
  }
}

// delete
{
  "payload": {
    "before": {

```

(continues on next page)

(continued from previous page)

```

        "c1": 11,
        "c2": null
    },
    "after": null,
    "source": {
        "version": "2.1.1.Final",
        "connector": "postgresql",
        "name": "debezium",
        "ts_ms": 1697280877563,
        "snapshot": "false",
        "db": "postgres",
        "sequence": [
            "23415392",
            "23415448"
        ],
        "schema": "s2",
        "table": "t2",
        "txId": 774,
        "lsn": 23415448,
        "xmin": null
    },
    "op": "d",
    "ts_ms": 1697280878032,
    "transaction": null
}
}

// update
{
    "payload": {
        "before": {
            "c1": 10,
            "c2": "abcd"
        },
        "after": {
            "c1": 10,
            "c2": "upd-c2-kkk"
        },
        "source": {
            "version": "2.1.1.Final",

```

(continues on next page)

(continued from previous page)

```

    "connector": "postgresql",
    "name": "debezium",
    "ts_ms": 1697285262028,
    "snapshot": "false",
    "db": "postgres",
    "sequence": [
        "23508840",
        "23508896"
    ],
    "schema": "s2",
    "table": "t3",
    "txId": 784,
    "lsn": 23508896,
    "xmin": null
},
"op": "u",
"ts_ms": 1697285262211,
"transaction": null
}
}
}

```

Note: When using `debeziumjson`, the following parameters are ignored:

- `synxdb.format.delimiter`
- `synxdb.topic.table`
- **zyzxjson/csgjson/ttjson:** Custom JSON data formats for specific clients.

```

// DML

{
    "_source_schema": "PUBLIC",
    "_source_table": "PERSON",
    "_committime": "2023-03-14 14:57:35.863",
    "_optype": "INSERT",
    "_seqno": "2261",
    "record": {
        "PKID": "825",
        "ID": "20211128",

```

(continues on next page)

(continued from previous page)

```
"NAME": "zhangsan",
"LOADING_DATE": "2023-03-14 00:00:00.0",
"DELETE_FLAG": "1",
"MOD_USER": "annoy",
"MOD_USER_ID": "75589"
}
}

{
  "_source_schema": "PUBLIC",
  "_source_table": "PERSON",
  "_committime": "2023-03-14 18:13:43.622",
  "_optype": "UPDATE",
  "_seqno": "2264",
  "record": {
    "PKID": "279",
    "ID": "20210582",
    "NAME": "zhangsan",
    "LOADING_DATE": "2023-03-14 00:00:00.0",
    "DELETE_FLAG": "1",
    "MOD_USER": "admin",
    "MOD_USER_ID": "94950"
  },
  "key": {
    "PKID": "279"
  }
}

{
  "_source_schema": "PUBLIC",
  "_source_table": "PERSON",
  "_committime": "2023-03-17 15:02:05.19",
  "_optype": "DELETE",
  "_seqno": "2267",
  "record": {
    "PKID": "5"
  },
  "key": {
    "PKID": "5"
  }
}
```

(continues on next page)

(continued from previous page)

```
}

// DDL

{
    "_source_schema": "PUBLIC",
    "_committime": "2023-03-17 15:06:05.249",
    "_optype": "DDL",
    "_seqno": "2268",
    "record": "alter table PUBLIC.PERSON add add_column integer"
}
```

Troubleshooting

SynxDB Kafka Connector's log files are located in the `kafka_2.13-3.1.0/logs` directory. Check these files to troubleshoot issues.

To print more detailed information, you can adjust the logging level by editing the `kafka_2.13-3.1.0/config/connect-log4j.properties` file. Change the log level for the Kafka Connector from `INFO` to `DEBUG` by modifying the following line:

```
log4j.logger.cn.synxdb=DEBUG
```

This will enable more detailed logging for troubleshooting.

References

- [APACHE KAFKA QUICKSTART](#)
- [Kafka Connect](#)

3.7 Load Data from Object Storage and HDFS

You can use the `datalake_fdw` extension to load data from an object storage (such as Amazon S3, QingCloud, Alibaba Cloud, Huawei Cloud, and Tencent Cloud), HDFS, and ORC tables in Hive into SynxDB for data query and access.

Currently, supported data formats are CSV, TEXT, ORC, and PARQUET.

 **Note**

`datalake_fdw` does not support loading data in parallel.

For information on how to load tables from Hive into SynxDB, see [Load Data from Hive Data Warehouse](#).

Install the extension

To install the `datalake_fdw` extension to the database, execute the SQL statement `CREATE EXTENSION data_fdw;`.

```
CREATE EXTENSION datalake_fdw;
```

Instructions

This section explains how to use `datalake_fdw` to load data from object storage and HDFS into SynxDB.

To load data using `datalake_fdw`, you need to create a foreign data wrapper (FDW) first. This includes creating an FDW server and user mapping.

Load data from object storage

You can load data from Amazon S3, QingCloud, Alibaba Cloud, Tencent Cloud, and other object storages into SynxDB. Follow these steps:

1. Create a foreign table wrapper FOREIGN DATA WRAPPER. Note that there are no options in the SQL statement below, and you need to execute it exactly as provided.

```
CREATE FOREIGN DATA WRAPPER datalake_fdw
HANDLER datalake_fdw_handler
VALIDATOR datalake_fdw_validator
OPTIONS ( mpp_execute 'all segments' );
```

2. Create an external server foreign_server.

```
CREATE SERVER foreign_server
FOREIGN DATA WRAPPER datalake_fdw
OPTIONS (host 'xxx', protocol 's3b', isvirtual 'false', ishttps 'false');
```

The options in the above SQL statement are explained as follows:

Option name	Description	Details
host	Sets the host information for accessing the object storage.	Required: Must be set Example: <ul style="list-style-type: none">• Host for QingCloud public cloud: pek3b.qingstor.com• Host for private cloud: 192.168.1.1:9000
protocol	Specifies the cloud platform for the object storage.	Required: Must be set Options: <ul style="list-style-type: none">• s3b: Amazon Cloud (uses v2 signature)• s3: Amazon Cloud (uses v4 signature)• ali: Alibaba Cloud object storage• qs: QingCloud object storage• cos: Tencent object storage• huawei: Huawei object storage• ks3: Kingstor object storage
isvirtual	Use virtual-host-style or path-host-style to parse the host of the object storage.	Required: Optional Options: <ul style="list-style-type: none">• true: Uses virtual-host-style.• false: Uses path-host-style. Default value: false
ishttps	Whether to use HTTPS to access the object storage.	Required: Optional Options: <ul style="list-style-type: none">• true: Uses HTTPS.• false: Does not use HTTPS. Default value: false

3. Create a user mapping.

```
CREATE USER MAPPING FOR gpadmin SERVER foreign_server
OPTIONS (user 'gpadmin', accesskey 'xxx', secretkey 'xxx');
```

The options in the above SQL statement are explained as follows:

Option name	Description	Required
user	Creates the specific user specified by <code>foreign_server</code> .	Yes
accesskey	The key needed to access the object storage.	Yes
secretkey	The secret key needed to access the object storage.	Yes

4. Create a foreign table example. After creating it, the data on the object storage is loaded into SynxDB, and you can query this table.

```
CREATE FOREIGN TABLE example(
    a text,
    b text
)
SERVER foreign_server
OPTIONS (filePath '/test/parquet/', compression 'none', enableCache 'false',
    format 'parquet');
```

The options in the SQL statement above are explained as follows:

Option name	Description	Details
<code>filePath</code>	Sets the specific path for the target foreign table.	<ul style="list-style-type: none"> Required: Must be set Path format should be <code>/bucket/prefix</code>. Example: <ul style="list-style-type: none"> If the bucket name is <code>test-bucket</code> and the path is <code>bucket/test/orc_file_folder/</code>, and there are files like <code>0000_0, 0001_1, 0002_2</code>, then to access file <code>0000_0</code>, set <code>filePath</code> to <code>filePath '/test-bucket/test/orc_file_folder/0000_0'</code>. To access all files in <code>test/orc_file_folder/</code>, set <code>filePath</code> to <code>filePath '/test-bucket/test/orc_file_folder/'</code>. Note: <code>filePath</code> is parsed in the format <code>/bucket/prefix/</code>. Incorrect formats might lead to errors, such as: <ul style="list-style-type: none"> <code>filePath 'test-bucket/test/orc_file_folder/'</code> <code>filePath '/test-bucket/test/orc_file_folder/0000_0'</code>

continues on next page

Table 4 – continued from previous page

Option name	Description	Details
compression	Sets the write compression format. Currently supports gzip, zstd, lz4.	<ul style="list-style-type: none"> Required: Optional Options: <ul style="list-style-type: none"> none: Supports CSV, ORC, TEXT, PARQUET. gzip: Supports CSV, TEXT, PARQUET. zstd: Supports PARQUET. lz4: Supports PARQUET. Default value: none, which means no compression. Not setting this value means no compression.
enableCache	Specifies whether to use Gopher caching.	<ul style="list-style-type: none"> Required: Optional Options: <ul style="list-style-type: none"> true: Enables Gopher caching. false: Disables Gopher caching. Default value: false Deleting the foreign table does not automatically clear its cache. To clear the cache, you need to manually run a specific SQL function, such as: <code>select gp_toolkit._gopher_cache_free_relation_name(text);</code>
format	The file format supported by FDW.	<ul style="list-style-type: none"> Required: Must be set Options: <ul style="list-style-type: none"> csv: Read, Write text: Read, Write orc: Read, Write parquet: Read, Write

Load data from HDFS

You can load data from HDFS into SynxDB. The following sections explain how to load data from an HDFS cluster without authentication and how to load data from an HDFS cluster with Kerberos authentication. SynxDB also supports loading data from an HDFS HA (High Availability) cluster, which is also explained below.

Load HDFS data without authentication

Load data from HDFS in the simple mode, which is the basic HDFS mode without using complex security authentication. For details, see the Hadoop documentation: [Hadoop in Secure Mode](#). The steps are as follows:

1. Create an external table wrapper FOREIGN DATA WRAPPER. Note that there are no options in the SQL statement below, and you need to execute the statement exactly as provided.

```
CREATE FOREIGN DATA WRAPPER datalake_fdw
HANDLER datalake_fdw_handler
VALIDATOR datalake_fdw_validator
OPTIONS ( mpp_execute 'all segments' );
```

2. Create an external server. In this step, you can create an external server for a single-node HDFS or for HA (High Availability) HDFS.

- Create an external server `foreign_server` for a single-node HDFS:

```
CREATE SERVER foreign_server FOREIGN DATA WRAPPER datalake_fdw
OPTIONS (
    protocol 'hdfs',
    hdfs_namenodes '[192.168.178.95] (http://192.168.178.95)',
    hdfs_port '9000',
    hdfs_auth_method 'simple',
    hadoop_rpc_protection 'authentication');
```

The options in the above SQL statement are explained as follows:

Option name	Description	Details
protocol	Specifies the Hadoop platform.	<ul style="list-style-type: none"> ○ Required: Must be set ○ Setting: Fixed as <code>hdfs</code>, which means Hadoop platform, cannot be changed. ○ Default value: <code>hdfs</code>
hdfs_namenodes	Specifies the namenode host for accessing HDFS.	<ul style="list-style-type: none"> ○ Required: Must be set ○ Example: For example, <code>hdfs_namenodes '192.168.178.95:9000'</code>
hdfs_auth_method	Specifies the authentication mode for accessing HDFS.	<ul style="list-style-type: none"> ○ Required: Must be set ○ Options: <ul style="list-style-type: none"> * <code>simple</code>: Uses Simple authentication to access HDFS. * <code>kerberos</code>: Uses Kerberos authentication to access HDFS. ○ Note: To access in Simple mode, set the value to <code>simple</code>, for example, <code>hdfs_auth_method 'simple'</code>.
hadoop_rpc_protection	Configures the authentication mechanism for setting up a SASL connection.	<ul style="list-style-type: none"> ○ Required: Must be set ○ Options: Three values are available: <code>authentication</code>, <code>integrity</code>, and <code>privacy</code>. ○ Note: This option must match the <code>hadoop.rpc.protection</code> setting in the HDFS configuration file <code>core-site.xml</code>. For more details, see the Hadoop documentation Explanation of core-site.xml.

- Create an external server for a multi-node HA cluster. The HA cluster supports

node failover. For more information about HDFS high availability, see the Hadoop documentation [HDFS High Availability Using the Quorum Journal Manager](#).

To load an HDFS HA cluster, you can create an external server using the following template:

```
CREATE SERVER foreign_server
  FOREIGN DATA WRAPPER datalake_fdw
  OPTIONS (
    protocol 'hdfs',
    hdfs_namenodes 'mycluster',
    hdfs_auth_method 'simple',
    hadoop_rpc_protection 'authentication',
    is_ha_supported 'true',
    dfs_nameservices 'mycluster',
    dfs_ha_namenodes 'nn1,nn2,nn3',
    dfs_namenode_rpc_address '192.168.178.95:9000,192.168.178.160:9000,192.168.186:9000',
    dfs_client_failover_proxy_provider 'org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider');
```

In the above SQL statement, `protocol`, `hdfs_namenodes`, `hdfs_auth_method`, and `hadoop_rpc_protection` are the same as in the single-node example. The HA-specific options are explained as follows:

Option name	Description	Details
<code>is_ha_supported</code>	Specifies whether to access the HDFS HA service (high availability).	<ul style="list-style-type: none"> ○ Required: Must be set. ○ Setting: Set to <code>true</code>. ○ Default value: /
<code>dfs_nameservices</code>	When <code>is_ha_supported</code> is <code>true</code> , specify the name of the HDFS HA service to access.	<ul style="list-style-type: none"> ○ Required: If using an HDFS HA cluster, must be set. ○ Matches the <code>dfs.ha.namenodes.mycluster</code> item in the HDFS config file <code>hdfs-site.xml</code>. ○ Note: For example, if <code>dfs.ha.namenodes.mycluster</code> is <code>cluster</code>, set this option as <code>dfs_nameservices 'mycluster'</code>.
<code>dfs_ha_namenodes</code>	When <code>is_ha_supported</code> is <code>true</code> , specify the accessible nodes for HDFS HA.	<ul style="list-style-type: none"> ○ Required: If using an HDFS HA cluster, must be set. ○ Setting: Matches the value of the <code>dfs.ha.namenodes.mycluster</code> item in the HDFS config file <code>hdfs-site.xml</code>. ○ Note: For example, <code>dfs_ha_namenodes 'nn1,nn2,nn3'</code>.

continues on next page

Table 6 – continued from previous page

Option name	Description	Details
dfs_namenode_rpc_address	When <code>is_ha_supported</code> is <code>true</code> , specifies the IP addresses of the high availability nodes in HDFS HA.	<ul style="list-style-type: none"> ○ Required: If using an HDFS HA cluster, must be set. ○ Setting: Refer to the <code>dfs.ha_namenodes</code> configuration in the HDFS <code>hdfs-site.xml</code> file. The node address matches the <code>namenode</code> address in the configuration. ○ Note: For example, if <code>dfs.ha.namenodes.mycluster</code> has three namenodes named <code>nn1, nn2, nn3</code>, find their addresses from the HDFS configuration file and enter them into this field. <pre>dfs_namenode_rpc_address '192.168.178.95:9000,192. 168.178.160:9000,192.168. 178.186:9000'</pre>
dfs_client_failover_proxy	Specifies whether HDFS HA has failover enabled.	<ul style="list-style-type: none"> ○ Required: If using an HDFS HA cluster, must be set. ○ Setting: Set to the default value: <pre>dfs_client_failover_proxy_provider 'org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider'</pre> <ul style="list-style-type: none"> ○ Default value: /

3. Create a user mapping.

```
CREATE USER MAPPING FOR gpadmin SERVER foreign_server
OPTIONS (user 'gpadmin');
```

In the above statement, the `user` option specifies the specific user for `foreign_server` and must be set.

4. Create the foreign table `example`. After creating it, the data from object storage is already loaded into SynxDB, and you can query this table.

```
CREATE FOREIGN TABLE example(
a text,
b text
)
SERVER foreign_server
OPTIONS (filePath '/test/parquet/', compression 'none', enableCache 'false',
format 'parquet');
```

The options in the above SQL statement are explained as follows:

Option name	Description	Details
filePath	Sets the specific path of the target foreign table.	<ul style="list-style-type: none"> Required: Must be set Setting: The path format should be `/hdfs_prefix`. Example: If the hdfs_prefix to be accessed is named test-example, the access path is /test-example/. If there are multiple files in the path (for example, 0000_0, 0001_1, 0002_2), you can set the filePath of 0000_0 file as filePath '/test-example/0000_0'. To access all the files in /test-example/, you can set the filePath as filePath '/test-example/'. Note: filePath should follow the /hdfs_prefix/ format. Incorrect formats might lead to errors, such as: <ul style="list-style-type: none"> o filePath 'test/test/orc_file_folder/' o filePath '/test/test/orc_file_folder/0000_0'
compression	Sets the compression format for writing. Currently supports gzip, zstd, lz4 formats.	<ul style="list-style-type: none"> Required: Optional Setting: <ul style="list-style-type: none"> o none: Supports CSV, ORC, TEXT, PARQUET formats. o gzip: Supports CSV, TEXT, PARQUET formats. o zstd: Supports PARQUET format. o lz4: Supports PARQUET format. Default value: none, which means no compression. Not setting this value also means no compression.
enableCache	Specifies whether to use the Gopher cache.	<ul style="list-style-type: none"> Required: Optional Setting: <ul style="list-style-type: none"> o true: Enables Gopher cache. o false: Disables Gopher cache. Default: false Note: Deleting a foreign table does not automatically clear the cache. To clear the cache for this table, you need to manually run a specific SQL function, for example: <pre>select gp_toolkit._gopher_cache_free_ relation_name(text);</pre>
format	The file format supported by FDW.	<ul style="list-style-type: none"> Required: Must be set Setting: <ul style="list-style-type: none"> o csv: Readable, writable o text: Readable, writable o orc: Readable, writable o parquet: Readable, writable

Load HDFS data with Kerberos authentication

If the target HDFS cluster uses Kerberos for authentication, you can follow these steps to load data from HDFS to SynxDB.

1. Create a foreign data wrapper FOREIGN DATA WRAPPER. Note that there are no options in the following SQL statement, and you need to execute it exactly as provided.

```
CREATE FOREIGN DATA WRAPPER datalake_fdw
HANDLER datalake_fdw_handler
```

(continues on next page)

(continued from previous page)

```
VALIDATOR datalake_fdw_validator
OPTIONS ( mpp_execute 'all segments' );
```

2. Create an external server. In this step, you can create an external server for a single-node HDFS or for an HA (High Availability) HDFS.

- Create an external server `foreign_server` for a single-node HDFS:

```
DROP SERVER foreign_server;

CREATE SERVER foreign_server
    FOREIGN DATA WRAPPER datalake_fdw
    OPTIONS (hdfs_namenodes '192.168.3.32',
             hdfs_port '9000',
             protocol 'hdfs',
             hdfs_auth_method 'kerberos',
             krb_principal 'gpadmin/hdw-68212a9a-master0@GPADMINCLUSTER2.
COM',
             krb_principal_keytab '/home/gpadmin/hadoop.keytab',
             hadoop_rpc_protection 'privacy'
    );
```

The options in the above SQL statement are explained as follows:

Option name	Description	Details
hdfs_namenodes	Specifies the namenode host for accessing HDFS.	<ul style="list-style-type: none"> ○ Required: Must be set ○ Setting: / ○ Example: For example, <code>hdfs_namenodes '192.168.178.95:9000'</code>
protocol	Specifies the Hadoop platform.	<ul style="list-style-type: none"> ○ Required: Must be set ○ Setting: Fixed to <code>hdfs</code>, meaning the Hadoop platform. Cannot be changed. ○ Default: <code>hdfs</code>
hdfs_auth_method	Specifies the authentication method for accessing HDFS, which is Kerberos.	<ul style="list-style-type: none"> ○ Required: Must be set ○ Setting: <code>kerberos</code>, to access HDFS using Kerberos authentication.
krb_principal	Specifies the primary user set in the HDFS keytab.	<ul style="list-style-type: none"> ○ Required: Must be set ○ Setting: Must match the user information in the keytab. You need to check the relevant user information and set this value accordingly.
krb_principal_keytab	Specifies the path to the HDFS keytab.	<ul style="list-style-type: none"> ○ Required: Must be set ○ Setting: The value must match the actual path of the keytab in HDFS.

continues on next page

Table 8 – continued from previous page

Option name	Description	Details
hadoop_rpc_protection	Configures the authentication mechanism for establishing a SASL connection.	<ul style="list-style-type: none"> ○ Required: Must be set ○ Setting: Three options are available: authentication, integrity, and privacy. ○ Note: This option must match the hadoop.rpc.protection setting in the HDFS configuration file core-site.xml. For more details, see the Hadoop core-site.xml documentation.

- Create an external server for a multi-node HA cluster. The HA cluster supports failover. For more information about HDFS high availability, see the Hadoop documentation [HDFS High Availability Using the Quorum Journal Manager](#).

To load an HDFS HA cluster, you can use the following template to create an external server:

```
CREATE SERVER foreign_server
  FOREIGN DATA WRAPPER datalake_fdw
  OPTIONS (hdfs_namenodes 'mycluster',
  protocol 'hdfs',
  hdfs_auth_method 'kerberos',
  krb_principal 'gpadmin/hdw-68212a9a-master0@GPADMINCLUSTER2.
COM',
  krb_principal_keytab '/home/gpadmin/hadoop.keytab',
  hadoop_rpc_protection 'privacy',
  is_ha_supported 'true',
  dfs_nameservices 'mycluster',
  dfs_ha_namenodes 'nn1,nn2,nn3',
  dfs_namenode_rpc_address '192.168.178.95:9000,192.168.178.
160:9000,192.168.178.186:9000',
  dfs_client_failover_proxy_provider 'org.apache.hadoop.hdfs.
server.namenode.ha.ConfiguredFailoverProxyProvider'
);
```

In the above SQL statement, the explanations for hdfs_namenodes, protocol, hdfs_auth_method, krb_principal, krb_principal_keytab, and hadoop_rpc_protection are the same as for the single-node example. The HA-specific options are explained as follows:

Option name	Description	Details
is_ha_supported	Specifies whether to access the HDFS HA service, which means high availability.	<ul style="list-style-type: none"> o Required: Must be set o Setting: Set to <code>true</code>.
dfs_nameservices	When <code>is_ha_supported</code> is <code>true</code> , specifies the name of the HDFS HA service to access.	<ul style="list-style-type: none"> o Required: Must be set if using an HDFS HA cluster. o Setting: Should match the <code>dfs.ha.namenodes.mycluster</code> item in the HDFS configuration file <code>hdfs-site.xml</code>. o Note: For example, if <code>dfs.ha.namenodes.mycluster</code> is <code>cluster</code>, set this parameter to <code>dfs_nameservices 'mycluster'</code>.
dfs_ha_namenodes	When <code>is_ha_supported</code> is <code>true</code> , specifies the accessible nodes for HDFS HA.	<ul style="list-style-type: none"> o Required: Must be set if using an HDFS HA cluster. o Setting: Should match the value of the <code>dfs.ha.namenodes.mycluster</code> item in the HDFS configuration file <code>hdfs-site.xml</code>. o Note: For example, <code>dfs_ha_namenodes 'nn1,nn2,nn3'</code>
dfs_namenode_rpc_address	When <code>is_ha_supported</code> is <code>true</code> , specifies the IP addresses of the high availability nodes in HDFS HA.	<ul style="list-style-type: none"> o Required: Must be set if using an HDFS HA cluster. o Setting: Refer to the <code>dfs.ha_namenodes</code> configuration in the HDFS <code>hdfs-site.xml</code> file. The node address should match the namenode address in the configuration. o Example: If <code>dfs.ha.namenodes.mycluster</code> has three namenodes <code>nn1, nn2, nn3</code>, you can find the addresses like <code>dfs.namenode.rpc-address.mycluster.nn1</code> in the HDFS config and fill them in. <pre>dfs_namenode_rpc_address '192.168.95:9000,192.168.178.160:9000,192.168.178.186:9000'</pre>
dfs_client_failover_proxy	Specifies whether HDFS HA has failover enabled.	<ul style="list-style-type: none"> o Required: Must be set if using an HDFS HA cluster. o Setting: Set to the default value <code>dfs_client_failover_proxy_provider 'org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverPr...</code>

3. Create a user mapping.

```
CREATE USER MAPPING FOR gpadmin SERVER foreign_server
OPTIONS (user 'gpadmin');
```

In the above statement, the `user` option specifies the specific user for `foreign_server` and must be set.

4. Create the foreign table example. After creating it, the data from object storage is already loaded into SynxDB, and you can query this table.

```
CREATE FOREIGN TABLE example(
    a text,
```

(continues on next page)

(continued from previous page)

```
b text
)
SERVER foreign_server
OPTIONS (filePath '/test/parquet/', compression 'none', enableCache 'false',
', format 'parquet');
```

The options in the above SQL statement are explained as follows:

Option name	Description	Details
filePath	Sets the specific path of the target foreign table.	<ul style="list-style-type: none"> Required: Must be set Setting: The path format should be `/hdfs_prefix`. Example: If the <code>hdfs_prefix</code> to be accessed is named <code>test-example</code>, the access path is <code>/test-example/</code>. If there are multiple files in the path (for example, <code>0000_0</code>, <code>0001_1</code>, <code>0002_2</code>), you can set the <code>filePath</code> of <code>0000_0</code> file as <code>filePath '/test-example/0000_0'</code>. To access all the files in <code>/test-example/</code>, you can set the <code>filePath</code> as <code>filePath '/test-example/'</code>. Note: <code>filePath</code> should follow the <code>/hdfs_prefix/</code> format. Incorrect formats might lead to errors, such as: <ul style="list-style-type: none"> <code>filePath 'test/test/orc_file_folder/'</code> <code>filePath '/test/test/orc_file_folder/0000_0'</code>
compression	Sets the compression format for writing, supports gzip, zstd, lz4 formats.	<ul style="list-style-type: none"> Required: Optional Setting: <ul style="list-style-type: none"> <code>none</code>: Supports CSV, ORC, TEXT, PARQUET formats. <code>gzip</code>: Supports CSV, TEXT, PARQUET formats. <code>zstd</code>: Supports PARQUET format. <code>lz4</code>: Supports PARQUET format. Default: <code>none</code>, which means no compression. Not setting this value also means no compression.
enableCache	Specifies whether to use the Gopher cache.	<ul style="list-style-type: none"> Required: Optional Setting: <ul style="list-style-type: none"> <code>true</code>: Enables Gopher cache. <code>false</code>: Disables Gopher cache. Default: <code>false</code> Note: Deleting a foreign table does not automatically clear the cache. To clear the cache, you need to manually run an SQL function, for example: <pre>select gp_toolkit.__gopher_cache_free_ relation_name(text);</pre>
format	The file format currently supported by FDW.	<ul style="list-style-type: none"> Required: Must be set Setting: <ul style="list-style-type: none"> <code>csv</code>: Readable, writable <code>text</code>: Readable, writable <code>orc</code>: Readable, writable <code>parquet</code>: Readable, writable

3.8 Load Data from Hive Data Warehouse

Hive data warehouse is built on the HDFS of the Hadoop cluster, so the data in the Hive data warehouse is also stored in HDFS. SynxDB supports loading tables from a Hive cluster into SynxDB using the extended Hive Connector and [Load Data from Object Storage and HDFS](#).

The Hive Connector loads tables from the Hive cluster as foreign tables in SynxDB, which store the paths to the data in HDFS. `datalake_fdw` reads data from these external tables, thus loading data from Hive into SynxDB.

This document explains how to use the Hive Connector and `datalake_fdw` to load tables from a Hive cluster into SynxDB.

Supported Hive file formats

You can load files in TEXT, CSV, ORC, PARQUET, Iceberg, or Hudi formats from Hive into SynxDB.

Usage limitations

- Synchronizing Hive external tables is not supported.
- Synchronizing Hive table statistics is not supported.
- SynxDB can read data from HDFS and write data to HDFS, but the written data cannot be read by Hive.

Note

Q: How is write and update on HDFS synchronized to SynxDB? Are there any limitations?

A: The data is still stored in HDFS, and the Foreign Data Wrapper only reads the data from HDFS.

Steps

The general steps to use the Hive Connector are as follows:

1. Create a configuration file on the SynxDB node, specifying the target Hive cluster and HDFS information in the file. See [Step 1. Create a configuration file on database cluster](#).
2. Create the foreign data wrapper and Hive Connector extension.
3. Create the server and user mapping.
4. Load Hive objects into SynxDB. You can load a single table from Hive or load an entire database.

Step 1. Create a configuration file on database cluster

Create a configuration file on the node of SynxDB, specifying the target Hive cluster and HDFS information in the file.

Configure Hive cluster information

The Hive Connector supports Hive v2.x and v3.x. You need to create a configuration file named `gphive.conf` on the coordinator and standby nodes of the SynxDB data warehouse.

Configuration items

Item name	Description	Default value
uris	Address of the Hive Metastore Service (HMS hostname)	/
auth_method	Authentication method for Hive Metastore Service: simple or kerberos	simple
krb_service_principal	The service principal needed for Kerberos authentication of the Hive Metastore Service. If using HMS HA, set the instance in the principal to _HOST, for example, <code>hive/_HOST@SYNDB</code> .	/
krb_client_principal	The client principal needed for Kerberos authentication of the Hive Metastore Service.	/
krb_client_keytab	The keytab file corresponding to the client principal for Kerberos authentication of the Hive Metastore Service.	/
debug	Hive Connector debug flag: true or false	false

gphive.conf configuration example

Create the gphive.conf file on the coordinator and standby nodes of the SynxDB data warehouse using the following content. Replace example.net:8083 with the address of your Hive Metastore Service.

```
hive-cluster-1: # connector name
  uris: thrift://example.net:8083
  auth_method: simple
```

Configure multiple Hive clusters

To configure multiple Hive clusters, add more entries in gphive.conf. The following example adds a new Hive cluster named `hive-cluster-2`, which requires Kerberos authentication, and another Hive HA cluster named `hive-cluster-3`, which also requires Kerberos authentication.

```
hive-cluster-1: #simple auth
  uris: thrift://example1.net:9083
  auth_method: simple

hive-cluster-2: #kerberos auth
  uris: thrift://example2.net:9083
  auth_method: kerberos
  krb_service_principal: hive/synxdb@SYNDB.CN
  krb_client_principal: user/synxdb@SYNDB.CN
  krb_client_keytab: /home/gpadmin/user.keytab

hive-cluster-3: #kerberos auth (HMS HA)
  uris: thrift://hms-primary.example2.net:9083,thrift://hms-standby.
example2.net:9083
  auth_method: kerberos
  krb_service_principal: hive/_HOST@SYNDB.CN
  krb_client_principal: user/synxdb@SYNDB.CN
  krb_client_keytab: /home/gpadmin/user.keytab
```

Configure HDFS cluster information

The Hive Connector needs information about the HDFS cluster where the Hive cluster is located to create external tables and read them using the `datalake_fdw` plugin. Therefore, you need to provide a configuration file named `gphdfs.conf` on the Coordinator and Standby nodes of SynxDB.

Configuration Options

Option Name	Description	Default Value
<code>hdfs_namenode_host</code>	Configure the HDFS host information, e.g., "hdfs://mycluster", where <code>hdfs://</code> can be omitted.	/
<code>hdfs_namenode_port</code>	Configure the HDFS port. If not set, it defaults to 9000.	9000
<code>hdfs_auth_method</code>	Configure the HDFS authentication method. Use <code>simple</code> for regular HDFS, or <code>kerberos</code> for Kerberos.	/
<code>krb_principal</code>	Kerberos principal. Set this when <code>hdfs_auth_method</code> is Kerberos.	/
<code>krb_principal_keytab</code>	Location of the keytab generated by the user.	/
<code>hadoop_rpc_protection</code>	Should match the configuration in <code>hdfs-site.xml</code> of the HDFS cluster.	/
<code>data_transfer_protocol</code>	When Kerberos is configured for the HDFS cluster, there are two methods: 1. privileged resources 2. SASL RPC data transfer protection and SSL for HTTP. If the second method ("SASL") is used, set <code>data_transfer_protocol</code> to <code>true</code> .	/
<code>is_ha_supported</code>	Indicates whether to use <code>hdfs-ha</code> . Set to <code>true</code> to use it, otherwise set to <code>false</code> . Default is <code>false</code> .	false

hdfs-ha Configuration Instructions

The program reads the HA configuration only when `is_ha_supported` is set to `true`. Users should provide the `hdfs-ha` configuration information in key-value format in the configuration file. The program will read all HA configurations in sequence, and each HA configuration must match the corresponding configuration in the HDFS cluster. The value of each configuration item must be in lowercase; if it is in uppercase, it must be converted to lowercase before configuration. The configuration is as shown in the table below:

Option Name	Description	Default Value
dfs.nameservices	The NameServices name of the HDFS cluster, represented as \${service} in the following configuration.	/
dfs.ha.namenodes.\${s}	The list of NameNodes in the cluster where the NameService is \${service}, separated by commas. One NameNode is represented as \${node} in the following configuration.	/
dfs.namenode.rpc-add	The RPC address of the NameNode named \${node} in the \${service} cluster.	/
dfs.namenode.http-add	The HTTP address of the NameNode named \${node} in the \${service} cluster.	/
dfs.client.failover.	The Java class used to communicate with the Active NameNode in the \${service} cluster.	/

HDFS cluster configuration example

The following configuration file contains the configurations for three HDFS clusters: paa_cluster, pab_cluster, and pac_cluster. Among them, paa_cluster does not use Kerberos authentication and does not use hdfs-ha. pab_cluster uses Kerberos authentication but does not use hdfs-ha. pac_cluster uses Kerberos authentication and is configured with a two-node hdfs-ha cluster.

```
paa_cluster:
# namenode host
hdfs_namenode_host: paa_cluster_master
# name port
hdfs_namenode_port: 9000
# authentication method
hdfs_auth_method: simple
# rpc protection
hadoop_rpc_protection: privacy
data_transfer_protocol: true


pab_cluster:
hdfs_namenode_host: pab_cluster_master
hdfs_namenode_port: 9000
hdfs_auth_method: kerberos
krb_principal: gpadmin/hdw-68212b9b-master0@GPADMINCLUSTER2.COM
krb_principal_keytab: /home/gpadmin/hadoop.keytab
hadoop_rpc_protection: privacy
```

(continues on next page)

(continued from previous page)

```

data_transfer_protocol: true

pac_cluster:
  hdfs_namenode_host: pac_cluster_master
  hdfs_namenode_port: 9000
  hdfs_auth_method: kerberos
  krb_principal: gpadmin/hdw-68212b9b-master0@GPADMINCLUSTER2.COM
  krb_principal_keytab: /home/gpadmin/hadoop.keytab
  hadoop_rpc_protection: privacy
  is_ha_supported: true
  dfs.nameservices: mycluster
  dfs.ha.namenodes.mycluster: nn1,nn2
  dfs.namenode.rpc-address.mycluster.nn1: 192.168.111.70:8020
  dfs.namenode.rpc-address.mycluster.nn2: 192.168.111.71:8020
  dfs.namenode.http-address.mycluster.nn1: 192.168.111.70:50070
  dfs.namenode.http-address.mycluster.nn2: 192.168.111.71:50070
  dfs.client.failover.proxy.provider.mycluster: org.apache.hadoop.hdfs.server.
namenode.ha.ConfiguredFailover..

```

Step 2. Create foreign data wrapper and Hive Connector plugin

Before synchronization, load the `datalake_fdw` plugin used for reading HDFS, and create the foreign data wrapper for reading external tables.

1. Create the foreign data wrapper.

```

CREATE EXTENSION datalake_fdw;

CREATE FOREIGN DATA WRAPPER datalake_fdw
  HANDLER datalake_fdw_handler
  VALIDATOR datalake_fdw_validator
  OPTIONS (mpp_execute 'all segments');

```

2. Before calling the function, you need to load the Hive Connector plugin.

```
CREATE EXTENSION hive_connector;
```

Step 3. Create server and user mapping

After creating the foreign data wrapper and Hive Connector, you need to create the server and user mapping, as shown in the following example:

```
SELECT public.create_foreign_server('sync_server', 'gpadmin', 'datalake_fdw',
'hdःfs-cluster-1');
```

In the above example, the `create_foreign_server` function takes the form as follows:

```
create_foreign_server(serverName,
                      userMapName,
                      dataWrapName,
                      hdःfsClusterName);
```

This function creates a server and user mapping pointing to an HDFS cluster, which can be used by the Hive Connector to create foreign tables. The `datalake_fdw` uses the server configuration to read data from the corresponding HDFS cluster when accessing external tables.

The parameters in the function are explained as follows:

- `serverName`: The name of the server to be created.
- `userMapName`: The name of the user to be created on the server.
- `dataWrapName`: The name of the data wrapper used for reading HDFS data.
- `hdःfsClusterName`: The name of the HDFS cluster where the Hive cluster is located, as specified in the configuration file.

Executing this function is equivalent to executing:

```
CREATE SERVER serverName FOREIGN DATA WRAPPER dataWrapName OPTIONS (.....);
CREATE USER MAPPING FOR userMapName SERVER serverName OPTIONS (user
'userMapName');
```

Where the content of `OPTIONS (.....)` is be read from the configuration specified by `hdःfsClusterName` in the configuration file.

Step 4. Sync Hive objects to the database cluster

Syncing a Hive table

To sync a table from Hive to SynxDB, see the following example:

```
-- Syncs Hive tables in postgresql.

gpadmin=# select public.sync_hive_table('hive-cluster-1', 'mytestdb', 'weblogs',
  'hdfs-cluster-1', 'mytestdb.weblogs', 'sync_server');
sync_hive_table
-----
t
(1 row)
```

The above example uses the `sync_hive_table` function to perform the synchronization. The general form of the function is as follows:

```
sync_hive_table(hiveClusterName,
                hiveDatabaseName,
                hiveTableName,
                hdfsClusterName,
                destTableName,
                serverName);

sync_hive_table(hiveClusterName,
                hiveDatabaseName,
                hiveTableName,
                hdfsClusterName,
                destTableName,
                serverName,
                forceSync);
```

This function syncs a table to SynxDB, with both non-forced and forced modes available. When `forceSync` is set to `true`, the sync is forced, which means that if a table with the same name already exists in SynxDB, the existing table is dropped before syncing. If the `forceSync` parameter is not provided or is set to `false`, the sync is forced, and an error will occur if a table with the same name exists.

The parameters are explained as follows:

- **hiveClusterName:** The name of the Hive cluster where the table to be synced is located, as specified in the configuration file.
- **hiveDatabaseName:** The name of the database in Hive where the table to be synced belongs.
- **hiveTableName:** The name of the table to be synced.
- **hdfsClusterName:** The name of the HDFS cluster where the Hive cluster is located, as specified in the configuration file.
- **destTableName:** The name of the table in SynxDB where the data will be synced.
- **serverName:** The name of the server to be used when creating the foreign table with the `datalake_fdw` plugin.
- **forceSync:** Indicates whether the sync should be forced. If yes, set to `true`; otherwise, set to `false`.

Sync a Hive database

The following example shows how to sync a Hive database to SynxDB:

```
gpadmin=# select public.sync_hive_database('hive-cluster-1', 'default', 'hdfs-cluster-1', 'mytestdb', 'sync_server');
sync_hive_database
**-----
** t
(1 row)
```

The above example uses the `sync_hive_database` function to perform the synchronization.

The general form of the function is as follows:

```
sync_hive_database(hiveClusterName,
                   hiveDatabaseName,
                   hdfsClusterName,
                   destSchemaName,
                   serverName);

sync_hive_database(hiveClusterName,
```

(continues on next page)

(continued from previous page)

```
hiveDatabaseName,  
hdfsClusterName,  
destSchemaName,  
serverName,  
forceSync);
```

This function syncs a Hive database into a schema in SynxDB, similar to syncing a single table. It supports both non-forced and forced modes. When `forceSync` is set to `true`, the sync is forced, which means that if tables with the same name already exist in SynxDB, the existing tables are dropped before syncing. If the `forceSync` parameter is not provided or is set to `false`, the sync is forced, and an error will occur if tables with the same name exist.

The parameters are explained as follows:

- `hiveClusterName`: The name of the Hive cluster as specified in the configuration file.
- `hiveDatabaseName`: The name of the database to be synced.
- `hdfsClusterName`: The name of the HDFS cluster where the Hive cluster is located, as specified in the configuration file.
- `destSchemaName`: The name of the schema in SynxDB where the database is synced.
- `serverName`: The name of the server to be used when creating the foreign table with the `datalake_fdw` plugin.

Note

The interfaces used by the above functions are as follows:

- `sync_hive_table` calls the HMS `thrift getTable` interface.
- `sync_hive_database` calls the HMS `thrift getTables` and `getTable` interfaces.

Examples of syncing tables

The following examples show only the commands for creating a table in Hive and syncing it to SynxDB, specifically the commands in *Step 4. Sync Hive objects to the database cluster*. The complete synchronization process should also include the steps before this Step 4.

Sync a Hive text table

1. Create the following text table in Hive.

```
-- Creates the Hive table in Beeline.

CREATE TABLE weblogs
(
    client_ip          STRING,
    full_request_date STRING,
    day                STRING,
    month              STRING,
    month_num          INT,
    year               STRING,
    referrer           STRING,
    user_agent          STRING
) STORED AS TEXTFILE;
```

2. Sync the text table to SynxDB.

```
-- Syncs the Hive table in psql.

gpadmin=# select public.sync_hive_table('hive-cluster-1', 'mytestdb',
'weblogs', 'hdfs-cluster-1', 'mytestdb.weblogs', 'sync_server');
sync_hive_table
-----
t
(1 row)
```

Sync a Hive ORC table

1. Create an ORC table in Hive.

```
-- Creates the Hive table in Beeline.  
CREATE TABLE test_all_type  
(  
    column_a tinyint,  
    column_b smallint,  
    column_c int,  
    column_d bigint,  
    column_e float,  
    column_f double,  
    column_g string,  
    column_h timestamp,  
    column_i date,  
    column_j char(20),  
    column_k varchar(20),  
    column_l decimal(20, 10)  
) STORED AS ORC;
```

2. Sync the ORC table to SynxDB:

```
-- Syncs the Hive table in pgsql.  
  
gpadmin=# select public.sync_hive_table('hive-cluster-1', 'mytestdb',  
'test_all_type', 'hdfs-cluster-1', 'mytestdb.test_all_type', 'sync_server  
');  
sync_hive_table  
-----  
t  
(1 row)
```

Sync a Hive ORC partitioned table

1. Create an ORC partitioned table in Hive.

```
-- Creates the Hive table in Beeline.

CREATE TABLE test_partition_1_int
(
    a tinyint,
    b smallint,
    c int,
    d bigint,
    e float,
    f double,
    g string,
    h timestamp,
    i date,
    j char(20),
    k varchar(20),
    l decimal(20, 10)
)
PARTITIONED BY
(
    m int
)
STORED AS ORC;

INSERT INTO test_partition_1_int VALUES (1, 1, 1, 1, 1, 1, 1, '1', '2020-01-01 01:01:01', '2020-01-01', '1', '1', 10.01, 1);
INSERT INTO test_partition_1_int VALUES (2, 2, 2, 2, 2, 2, 2, '2', '2020-02-02 02:02:02', '2020-02-01', '2', '2', 11.01, 2);
INSERT INTO test_partition_1_int VALUES (3, 3, 3, 3, 3, 3, 3, '3', '2020-03-03 03:03:03', '2020-03-01', '3', '3', 12.01, 3);
INSERT INTO test_partition_1_int VALUES (4, 4, 4, 4, 4, 4, 4, '4', '2020-04-04 04:04:04', '2020-04-01', '4', '4', 13.01, 4);
INSERT INTO test_partition_1_int VALUES (5, 5, 5, 5, 5, 5, 5, '5', '2020-05-05 05:05:05', '2020-05-01', '5', '5', 14.01, 5);
```

2. Sync the ORC partitioned table to SynxDB.

```
-- psql syncs the Hive partitioned tables as one foreign table.
```

(continues on next page)

(continued from previous page)

```
gpadmin=# select public.sync_hive_table('hive-cluster-1', 'mytestdb',
'test_partition_1_int', 'hdfs-cluster-1', 'mytestdb.test_partition_1_int',
'sync_server');
sync_hive_table
-----
t
(1 row)
```

3. View the sync result.

```
gpadmin=# \d mytestdb.test_partition_1_int;
          Foreign table "mytestdb.test_partition_1_int"
Column | Type | Collation | Nullable | Default | FDW options
-----+-----+-----+-----+-----+
-----+
a | smallint | | | |
b | smallint | | | |
c | integer | | | |
d | bigint | | | |
e | double precision | | | |
f | double precision | | | |
g | text | | | |
h | timestamp without time zone | | | |
i | date | | | |
j | character(20) | | | |
k | character varying(20) | | | |
l | numeric(20,10) | | | |
m | integer | | | |
Server: sync_server
FDW options: (filepath '/opt/hadoop/apache-hive-3.1.0-bin/user/hive/
warehouse/mytestdb.db/test_partition_1_int', hive_cluster_name 'hive-
cluster-1', datasource 'mytestdb.test_partition_1_int', hdfs_cluster_name
'hdfs-cluster-1', enablecache 'true', transactional 'false', partitionkeys
'm', format 'orc')
```

Sync a sample Hive database

1. Sync the Hive database to SynxDB.

```
gpadmin=# select public.sync_hive_database('hive-cluster-1', 'default',
'hdfs-cluster-1', 'mytestdb', 'sync_server');
sync_hive_database
** -----
** t
(1 row)
```

2. View the result.

```
gpadmin=# \d mytestdb.*
                                         List of relations
 Schema |           Name           |   Type   | Owner |
Storage
-----+-----+-----+-----+
-----+
mytestdb | test_all_type      | foreign table | gpadmin |
mytestdb | weblogs            | foreign table | gpadmin |
mytestdb | test_csv_default_option | foreign table | gpadmin |
mytestdb | test_partition_1_int | foreign table | gpadmin |
(4 rows)
```

Sync tables in Iceberg and Hudi formats

Apache Iceberg (hereafter referred to as Iceberg) is an open-source table format designed to improve the storage, access, and processing of big data. It is tailored for large-scale data warehouse scenarios, providing efficient data storage and query optimization. Apache Hudi (hereafter referred to as Hudi) is a library that offers efficient storage management for data lakes, aiming to simplify incremental data processing and stream processing.

When Hive was initially designed, it did not account for some of the modern data lake requirements, such as real-time data processing and more granular control over updates. However, Iceberg and Hudi offer Hive-compatible interfaces. Iceberg and Hudi provide efficient and flexible data management capabilities for modern big data platforms. Compared to traditional Hive data warehouses, they deliver higher performance and advanced data management features when handling large datasets. By integrating with Hive, they offer a smooth upgrade path, enabling

you to transition from traditional data warehouse architectures to more modern and efficient data platform solutions.

The Hive Connector and `datalake_fdw` support loading tables in Iceberg and Hudi formats into SynxDB.

Load Iceberg tables

1. Create a table in Iceberg format on Hive (using Hive v2.3.2 as an example).

```
CREATE DATABASE icebergdb;
USE icebergdb;

CREATE TABLE iceberg_table1 (
    id int,
    name string,
    age int,
    address string
) STORED BY 'org.apache.iceberg_mr.hive.HiveIcebergStorageHandler';
```

2. Create the corresponding foreign table in SynxDB and import the data.

```
CREATE FOREIGN TABLE iceberg_table1 (
    id int,
    name text,
    age int,
    address text
)
server sync_server
OPTIONS (filePath 'icebergdb.iceberg_table1', catalog_type 'hive', server_
name 'hive-cluster-1', hdfs_cluster_name 'hdfs-cluster-1', table_
identifier 'icebergdb.iceberg_table1', format 'iceberg');
```

The table creation parameters are as follows:

- `catalog_type`: Specifies either `hive` or `hadoop`.
- `filePath`
 - If the `catalog_type` value is `hive`, specify
`<database_name>.<table_name>`.

- If the catalog_type value is hadoop, specify the path of the table in HDFS, such as /user/hadoop/hudidata/.
- table_identifier: Specifies <database_name>.<table_name>.
- format: Specifies iceberg.

Load Hudi tables

1. Create a table in Hudi format on Hive (using Spark v2.4.4 as an example).

```
CREATE DATABASE hudidb;
USE hudidb;

----- hudi_table1 -----
CREATE TABLE hudi_table1 (
    id int,
    name string,
    age int,
    address string
) using hudi;
```

2. Create the corresponding foreign table in SynxDB and import the data.

```
CREATE FOREIGN TABLE hudi_table1 (
    id int,
    name text,
    age int,
    address text
)
server sync_server
OPTIONS (filePath 'hudidb.hudi_table1', catalog_type 'hive', server_name
'hive-cluster-1', hdfs_cluster_name 'hdfs-cluster-1', table_identifier
'hudidb.hudi_table1', format 'hudi');
```

Data type mapping

The following table shows the one-to-one mapping between table data types on a Hive cluster and table data types in SynxDB.

Hive	SynxDB
binary	bytea
tinyint	smallint
smallint	smallint
int	int
bigint	bigint
float	float4
double	double precision
string	text
timestamp	timestamp
date	date
char	char
varchar	varchar
decimal	decimal

Known issues

When the SynxDB coordinator and standby nodes are on the same machine, there might be port conflicts because of shared configurations. This might cause the `dagent` process to continuously restart, leading to high CPU usage.

Solution

1. Create a `config` folder in the standby node's working directory
`(/home/gpadmin/workspace/cbdb_dev/gpAux/gpdemo/datadirs/standby/).`
2. In the `config` directory, create a configuration file `application.properties`.
 Modify the port using `server.port`, change the log file name with `logging.file.name`, and update the log file path with `logging.file.path`.

The `application.properties` file is as follows:

```
# Expose health, info, shutdown, metrics, and prometheus endpoints by default
```

(continues on next page)

(continued from previous page)

```

# 1. health: returns the status of the application {"status":"UP"}
# 2. info: returns information about the build {"build": {"version": "X.X.X",
#, "artifact": "dagent", "name": "dagent", "time": "timestamp" } }
# 3. shutdown: allows shutting down the application
# 4. metrics: shows 'metrics' information for the application
# 5. prometheus: exposes metrics in a format that can be scraped by a
Prometheus server
management.endpoints.web.exposure.include=health,info,shutdown,metrics,
prometheus
management.endpoint.shutdown.enabled=true
management.endpoint.health.probes.enabled=true
# common tags applied to all metrics
management.metrics.tags.application=dagent
# dagent-specific metrics
dagent.metrics.partition.enabled=true
dagent.metrics.report-frequency=1000
spring.profiles.active=default
server.port=5888
# Whitelabel error options
server.error.include-message=always
server.error.include-stacktrace=on_param
server.error.include-exception=false
server.server-header=DlAgent Server
server.max-http-header-size=1048576
# tomcat specific
server.tomcat.threads.max=200
server.tomcat.accept-count=100
server.tomcat.connection-timeout=5m
server.tomcat.mbeanregistry.enabled=true
dagent.tomcat.max-header-count=30000
dagent.tomcat.disable-upload-timeout=false
dagent.tomcat.connection-upload-timeout=5m
# timeout (ms) for the request - 1 day
spring.mvc.async.request-timeout=86400000
dagent.task.thread-name-prefix=dagent-response-
dagent.task.pool.allow-core-thread-timeout=false
dagent.task.pool.core-size=8
dagent.task.pool.max-size=200
dagent.task.pool.queue-capacity=0
# logging

```

(continues on next page)

(continued from previous page)

```
dlagent.log.level=info
logging.config=classpath:log4j2-dlagent.xml
logging.file.name=${MASTER_DATA_DIRECTORY:/home/gpadmin/workspace/cbdb_
dev/gpAux/gpdemo/datadirs/standby/demoDataDir-1}/pg_log/dlagent.log
logging.file.path=${MASTER_DATA_DIRECTORY:/home/gpadmin/workspace/cbdb_
dev/gpAux/gpdemo/datadirs/standby/demoDataDir-1}/pg_log
```

3.9 Load Data from MySQL Server Using MySQL_FDW

MySQL FDW is a Foreign Data Wrapper (FDW). You can use it to connect SynxDB to a MySQL database. MySQL FDW is developed based on Linux and can run on any POSIX-compliant system.

Prerequisites

Before enabling MySQL FDW, ensure the following steps have been completed:

1. MySQL Server has been correctly installed and is accessible.
2. Install the MySQL Client on the Coordinator node of SynxDB. You can download and install client from the [MySQL official website](#).

Note

Ensure that `libmysqlclient.so` is installed. You can use `find / -name libmysqlclient.so` to check its path. If no path is returned, you can install the MySQL Client development package using the following command:

```
sudo yum install mysql-community-devel
```

3. On the Coordinator node, as the `gpadmin` user, add the directory path of `libmysqlclient.so` to the `LD_LIBRARY_PATH` environment variable: `export LD_LIBRARY_PATH=<path_to_directory_of_libmysqlclient.so>:$LD_LIBRARY_PATH`. You can find the path using the command from step 2.

Use MySQL FDW

CREATE SERVER options

MySQL FDW accepts the following options via the CREATE SERVER command:

Option	Description
host	String, optional, default value <code>127.0.0.1</code> . Address or hostname of the MySQL server.
port	Integer, optional, default value <code>3306</code> . Port number of the MySQL server.
secure_auth	Boolean, optional, default value <code>true</code> . Enable or disable secure authentication.
init_command	String, optional, no default value. SQL command to execute upon connection to the MySQL server.
use_remote_estimate	Boolean, optional, default value <code>false</code> . Controls whether MySQL FDW issues a remote EXPLAIN command for cost estimation.
reconnect	Boolean, optional, default value <code>false</code> . Enable or disable automatic reconnection to the MySQL server when an existing connection is lost.
sql_mode	String, optional, default value <code>ANSI_QUOTES</code> . Sets MySQL's sql_mode for the established connection.
ssl_key	String, optional, no default value. Path to the client private key file.
ssl_cert	String, optional, no default value. Path to the client public key certificate file.
ssl_ca	String, optional, no default value. Path to the Certificate Authority (CA) certificate file. If this option is used, the same certificate as the server must be specified.
ssl_capath	String, optional, no default value. Directory path containing trusted SSL CA certificate files.
ssl_cipher	String, optional, no default value. Allowed cipher list for SSL encryption.
fetch_size	Integer, optional, default value <code>100</code> . Specifies the number of rows that MySQL FDW should fetch per fetch operation. Can be specified for a foreign table or a foreign server. The table option overrides the server option.
character_set	String, optional, default value <code>auto</code> . The character set used for the MySQL connection. The default value is auto, which automatically detects based on the OS settings. Before the introduction of the character_set option, the character set setting was similar to the SynxDB database encoding. To get this old behavior, set character_set to the special value PGDatabaseEncoding.
mysql_default_file	String, optional, no default value. Specifies the path to the MySQL default file if connection details (like username, password, etc.) are to be retrieved from it.
truncatable	Boolean, optional, default value <code>true</code> . Controls whether MySQL FDW allows the TRUNCATE command on foreign tables. Can be specified for foreign tables or foreign servers. Table-level options override server-level options.

CREATE USER MAPPING options

MySQL FDW accepts the following options via the CREATE USER MAPPING command:

Option	Description
username	String, no default value. Username used to connect to MySQL.
password	String, no default value. Password used for authenticating the MySQL server.

CREATE FOREIGN TABLE options

MySQL FDW accepts the following table-level options via the CREATE FOREIGN TABLE command:

Option	Description
dbname	String, required. The name of the MySQL database to query.
table_name	String, optional, default is the name of the foreign table. Name of the MySQL table.
fetch_size	Integer, optional. Same as the fetch_size parameter for the foreign server.
max_blob_size	Integer, optional. The maximum blob size to read without truncation.
truncatable	Boolean, optional, default value <code>true</code> . Same as the foreign server option.

IMPORT FOREIGN SCHEMA options

MySQL FDW supports the IMPORT FOREIGN SCHEMA command and accepts the following custom options:

Option	Description
import_default	Boolean type, optional, default value is <code>false</code> . This option controls whether the column DEFAULT expressions are included when importing foreign table definitions from a foreign server.
import_not_null	Boolean type, optional, default value is <code>true</code> . This option controls whether the column NOT NULL constraints are included when importing foreign table definitions from a foreign server.
import_enum_as_text	Boolean type, optional, default value is <code>false</code> . This option allows mapping the MySQL ENUM type to the TEXT type in foreign table definitions. Otherwise, a warning is issued that the creation of a custom type is required.
import_generated	Boolean type, optional, default value is <code>true</code> . This option controls whether GENERATED column expressions are included when importing foreign table definitions from a foreign server. If the imported generated expression uses functions or operators not available in SynxDB, the import will completely fail.

TRUNCATE support

MySQL FDW implements the TRUNCATE API, which is available for PostgreSQL kernel versions 14 and above. MySQL provides a TRUNCATE command; see the [TRUNCATE TABLE Statement](#).

Note the following usage limitations:

- `TRUNCATE ... CASCADE` is not supported.
- `TRUNCATE ... RESTART IDENTITY` is not supported and is ignored.
- `TRUNCATE ... CONTINUE IDENTITY` is not supported and is ignored.
- MySQL tables with foreign key references cannot be truncated.

Functions

In addition to the standard `mysql_fdw_handler()` and `mysql_fdw_validator()` functions, MySQL FDW also provides the following utility functions for user invocation:

- `mysql_fdw_version()`: Returns the version number as an integer.
- `mysql_fdw_display_pushdown_list()`: Displays the contents of the `mysql_fdw_pushdown.config` file.

Note

While MySQL FDW may insert or update values for generated columns in MySQL, it cannot prevent these values from being modified in MySQL. Therefore, there is no guarantee that a subsequent SELECT operation will retrieve the expected generated value for the column.

Usage Examples

Installing the Extension

In a SynxDB database, enable the MySQL FDW extension as a superuser `gpadmin`:

```
CREATE EXTENSION mysql_fdw;
```

Creating a Foreign Server

Create a foreign server object and specify the connection parameters for the MySQL database:

```
CREATE SERVER <mysql_server>
FOREIGN DATA WRAPPER mysql_fdw
OPTIONS (host '<localhost>', port '<3306>');
```

Note

If MySQL Server and SynxDB Coordinator are installed on the same node, the value for `host` must be `localhost`.

Granting Permissions to Use Foreign Server

Grant permission for a regular user to use the foreign server:

```
GRANT USAGE ON FOREIGN SERVER <mysql_server> TO <gpadmin>;
```

Creating a User Mapping

Create a user mapping for the MySQL server, specifying the username and password. The following example code creates a MySQL user mapping for the current user:

```
CREATE USER MAPPING for CURRENT_USER
SERVER <mysql_server>
OPTIONS (username '<mysql_username>', password '<mysql_pwd>');
```

Creating a Foreign Table

Create a foreign table, ensuring that the user has been granted permission to use the foreign server.

If the MySQL table name differs from the foreign table name, use the `table_name` option to specify the MySQL table name.

Note

The corresponding MySQL table must exist for the foreign table. To perform insert, delete, or update operations, the first column of the MySQL table must be unique. You can either set it as the primary key or add a `UNIQUE` constraint.

The following example demonstrates how to create a foreign table. Replace the values in the command before running it:

```
CREATE FOREIGN TABLE external_warehouse (
warehouse_id int,
warehouse_name text,
warehouse_created timestamp
)
SERVER mysql_server
OPTIONS (dbname 'mysql_db', table_name 'mysql_warehouse');
```

You can perform insert, select, delete, and update operations in the same way as on a local table.

```
-- Insert new rows into the table
INSERT INTO external_warehouse values (1, 'UPS', current_date);
INSERT INTO external_warehouse values (2, 'TV', current_date);
INSERT INTO external_warehouse values (3, 'Table', current_date);

-- Query data from the table
SELECT * FROM external_warehouse ORDER BY 1;

warehouse_id | warehouse_name | warehouse_created
-----+-----+-----
 1 | UPS          | 10-JUL-20 00:00:00
 2 | TV           | 10-JUL-20 00:00:00
 3 | Table        | 10-JUL-20 00:00:00
```

(continues on next page)

(continued from previous page)

```
-- Delete rows from the table
DELETE FROM external_warehouse WHERE warehouse_id = 3;

-- Update rows in the table
UPDATE warehouse SET warehouse_name = 'UPS_NEW' WHERE warehouse_id = 1;

-- Use the VERBOSE option to EXPLAIN the table
EXPLAIN VERBOSE SELECT warehouse_id, warehouse_name FROM external_warehouse
WHERE warehouse_name LIKE 'TV' limit 1;
```

QUERY PLAN

```
Limit  (cost=10.00..11.00 rows=1 width=36)
  Output: warehouse_id, warehouse_name
    -> Foreign Scan on public.warehouse  (cost=10.00..1010.00 rows=1000
width=36)
      Output: warehouse_id, warehouse_name
      Local server startup cost: 10
      Remote query: SELECT `warehouse_id`, `warehouse_name` FROM
`db`.`warehouse` WHERE ((`warehouse_name` LIKE BINARY 'TV'))
```

Importing a MySQL Database as a SynxDB Schema

Use the `IMPORT FOREIGN SCHEMA` command to import a schema from a MySQL database into SynxDB:

```
IMPORT FOREIGN SCHEMA someschema
FROM SERVER <mysql_server>
INTO public;
```

3.10 Custom Multi-Character Delimiters for Reading and Writing External Tables

When reading and writing external table data, SynxDB supports custom delimiters by extending `gp_exttable_delimiter`, allowing for more flexible data formats.

Compilation and installation

On the server where SynxDB is installed, download the extension and install it for usage.

```
cd gp_exttable_delimiter
```

```
make install
```

Usage example

Read external tables

1. Prepare a sample file `1.txt` and start `gpfdist` to provide network distribution capabilities.

```
touch 1.txt
echo 'aa|@|bb' > 1.txt
gpfdist -p 8088 -d .
```

2. Load the `gp_exttable_delimiter` extension.

```
CREATE EXTENSION IF NOT EXISTS gp_exttable_delimiter;
CREATE TABLE t1 (c1 text, c2 text);
```

3. Create an external table.

```
CREATE EXTERNAL TABLE t1_ext (LIKE t1) LOCATION ('gpfdist://localhost:8088/
1.txt') FORMAT 'CUSTOM' (formatter=delimiter_in, entry_delim='|@|', line_
delim=E'\n');
```

In the external table creation statement above:

- `FORMAT 'CUSTOM'` specifies the use of a custom delimiter.

- `formatter=delimiter_in` is a string parsing function provided by `gp_exttable_delimiter`, indicating that the file will be read.
- `entry_delim='|@|'` specifies that columns are separated by the `|@|` string.
- `line_delim` specifies that each row is separated by the newline character `\n`.

4. Query the external table to read the related content.

```
SELECT * FROM t1_ext;
```

Write to external tables

1. Prepare a sample file `2.txt` for writing.

```
touch 2.txt
```

2. Load the `gp_exttable_delimiter` extension.

```
CREATE EXTENSION IF NOT EXISTS gp_exttable_delimiter;
CREATE TABLE t2 (a int,b int);
```

3. Create an external table and write data into `2.txt`.

```
CREATE WRITABLE EXTERNAL TABLE t2_ext(LIKE t2) LOCATION ('gpfdist://
localhost:8088/2.txt') FORMAT 'CUSTOM' (FORMATTER=delimiter_ou_any,entry_
delim='|@|',line_delim=E'\n',null='');

INSERT INTO t2_ext values(1,2);
```

In the external table creation statement above:

- `FORMAT 'CUSTOM'` specifies the use of a custom delimiter.
- `FORMATTER=delimiter_ou_any` is a string parsing function provided by `gp_exttable_delimiter`, indicating that data will be written to the file.
- `entry_delim='|@|'` specifies that columns are separated by the `|@|` string.
- `line_delim=E'\n'` specifies that each row is separated by the newline character `\n`.

4. You can see the written data in the file.

```
cat 2.txt
```

Chapter 4

Operate with Data

4.1 Operate with Database Objects

Create and Manage Views

In SynxDB, views enable you to save frequently used or complex queries, then access them in a SELECT statement as if they were a table. A view is not physically materialized on disk: the query runs as a subquery when you access the view.

Create views

The CREATE VIEW command defines a view of a query. For example:

```
CREATE VIEW comedies AS SELECT * FROM films WHERE kind = 'comedy';
```

Drop views

The `DROP VIEW` command removes a view. For example:

```
DROP VIEW topten;
```

The `DROP VIEW ... CASCADE` command also removes all dependent objects. As an example, if another view depends on the view which is about to be dropped, the other view will be dropped as well. Without the `CASCADE` option, the `DROP VIEW` command will fail.

Best practices when creating views

When defining and using a view, remember that a view is just a SQL statement and is replaced by its definition when the query is run.

These are some common uses of views:

- They allow you to have a recurring SQL query or expression in one place for easy reuse.
- They can be used as an interface to abstract from the actual table definitions, so that you can reorganize the tables without having to modify the interface.
- If a subquery is associated with a single query, consider using the `WITH` clause of the `SELECT` command instead of creating a seldom-used view.

In general, these uses do not require nesting views, which means defining views based on other views.

These are two patterns of creating views that tend to be problematic because the view's SQL is used during query execution:

- Defining many layers of views so that your final queries look deceptively simple.

Problems arise when you try to enhance or troubleshoot queries that use the views, for example, by examining the execution plan. The query's execution plan tends to be complicated and it is difficult to understand and improve.

- Defining a denormalized “world” view.

A view that joins a large number of database tables and is used for a wide variety of queries. Performance issues can occur for some queries that use the view for certain `WHERE`

conditions, while other WHERE conditions work well.

Create and Manage Materialized Views

In SynxDB, materialized views are similar to views. A materialized view enables you to save a frequently used or complex query and then access the query results in a SELECT statement as if they were a table. Materialized views persist the query results in a table-like form.

Although accessing the data stored in a materialized view can be much faster than accessing the underlying tables directly or through a regular view, the data is not always current. The materialized view data cannot be directly updated. To refresh the materialized view data, use the REFRESH MATERIALIZED VIEW command.

The query used to create the materialized view is stored in exactly the same way that a view's query is stored. For example, you can create a materialized view that quickly displays a summary of historical sales data for situations where having incomplete data for the current date is acceptable.

```
CREATE MATERIALIZED VIEW sales_summary AS
  SELECT seller_no, invoice_date, sum(invoice_amt) ::numeric(13,2) as sales_amt
    FROM invoice
   WHERE invoice_date < CURRENT_DATE
  GROUP BY seller_no, invoice_date;

CREATE UNIQUE INDEX sales_summary_seller
  ON sales_summary (seller_no, invoice_date);
```

The materialized view might be useful for displaying a graph in the dashboard created for salespeople. You can schedule a job to update the summary information each night using the following command:

```
REFRESH MATERIALIZED VIEW sales_summary;
```

The information about a materialized view in the SynxDB system catalogs is exactly the same as it is for a table or view. A materialized view is a relation, just like a table or a view. When a materialized view is referenced in a query, the data is returned directly from the materialized view, just like from a table. The query in the materialized view definition is only used for populating the materialized view.

If you can tolerate periodically updating the materialized view data, you can get great performance benefits from the view.

One use of a materialized view is to allow faster access to data brought in from an external data source, such as an external table or a foreign data wrapper. Also, you can define indexes on a materialized view, whereas foreign data wrappers do not support indexes. This advantage might not apply to other types of external data access.

If a subquery is associated with a single query, consider using the WITH clause of the SELECT command instead of creating a seldom-used materialized view.

Create materialized views

The CREATE MATERIALIZED VIEW command defines a materialized view based on a query.

```
CREATE MATERIALIZED VIEW us_users AS
SELECT u.id, u.name, a.zone
FROM users u, address a
WHERE a.country = 'USA';
```

Tip

When a materialized view is created with an ORDER BY or SORT clause, this sorting is applied only at the time of the view's initial creation. Subsequent refreshes of the materialized view do not maintain this order, because the view is essentially a static snapshot of data and does not dynamically update or preserve the sorting with new data insertions.

Refresh or deactivate materialized views

The REFRESH MATERIALIZED VIEW command updates the materialized view data.

```
REFRESH MATERIALIZED VIEW us_users;
```

With the WITH NO DATA clause, the current data is removed, no new data is generated, and the materialized view is left in an unscannable state. An error is returned if a query attempts to access an unscannable materialized view.

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

Drop materialized views

The `DROP MATERIALIZED VIEW` command removes a materialized view definition and data. For example:

```
DROP MATERIALIZED VIEW us_users;
```

The `DROP MATERIALIZED VIEW ... CASCADE` command also removes all dependent objects. For example, if another materialized view depends on the materialized view which is about to be dropped, the other materialized view will be dropped as well. Without the `CASCADE` option, the `DROP MATERIALIZED VIEW` command fails.

Create and Manage indexes

In traditional databases, indexes are commonly used to significantly improve data access performance. However, in distributed systems like SynxDB, indexes must be used more cautiously. SynxDB provides high-speed sequential scans, while index lookups rely on random disk access. Because data is distributed across multiple segments, each segment only needs to scan its local subset of data to return results. Combined with table partitioning, the amount of data scanned during a query can be further reduced. As a result, in business intelligence (BI) workloads—which typically return large result sets—indexes might introduce performance overhead instead of benefits.

The recommended approach is to run queries without indexes first. If necessary, add indexes based on actual performance observations. Indexes are usually more suitable for OLTP scenarios, where queries often retrieve a single row or a small data range. For queries that return specific target rows, indexes can also provide performance gains on AO tables. In suitable cases, the planner might choose to use the index instead of scanning the entire table. For compressed data, index access allows the system to decompress only the required rows, improving overall efficiency.

When a table defines a primary key, SynxDB automatically creates a `PRIMARY KEY` constraint. For partitioned tables, indexes must be created on the root table; the system will automatically propagate the index to all child tables. Note that you cannot manually add indexes to system-created child tables.

In addition, a `UNIQUE CONSTRAINT` (such as a `PRIMARY KEY CONSTRAINT`) implicitly creates a `UNIQUE INDEX`. This index must include all columns in both the distribution key and partition key, and it must enforce uniqueness across the entire table, including all partitions.

Indexes introduce additional overhead: they consume storage and require maintenance during data updates. Therefore, always confirm that your query workload actually uses the indexes you create, and verify that they provide measurable performance improvements. You can check this by examining the query execution plan using `EXPLAIN`.

Index types

SynxDB supports multiple PostgreSQL index types, including B-tree, Hash, GiST, SP-GiST, GIN, and *BRIN indexes*. Each type is suited to different query patterns. B-tree is the default and most widely applicable index type. For detailed explanations of each index type, refer to the PostgreSQL documentation on [index types](#).

Note

In SynxDB, the indexed columns of a unique index must either match the distribution key or include all its columns. On partitioned tables, unique indexes enforce uniqueness only within individual child tables, not across all partitions.

Bitmap indexes

Bitmap indexes are especially suitable for typical data warehouse scenarios with large data volumes, frequent queries, and infrequent updates. Compared to regular indexes, bitmap indexes save storage space while efficiently handling multi-condition queries.

A bitmap index maintains a bitmap for each key value. Each bit in the bitmap indicates whether the corresponding row contains that value. Bitmaps can be combined using Boolean operations (such as AND and OR), allowing efficient filtering across multiple conditions. Before accessing data, bitmap operations can eliminate large numbers of irrelevant rows, significantly improving query performance.

Recommended usage:

- Ideal for analytical workloads in data warehouses.
- Best suited for columns with medium cardinality (around 100 to 100,000 unique values).
- Especially effective for queries with multiple AND or OR conditions in the WHERE clause.
- Starting from v4.0.0, bitmap index scans can be triggered by array predicates (such as `col IN (...)` or `col = ANY(array)`):
 - Works with both B-tree and hash indexes.
 - Hash indexes, which previously only supported equality matches, now support array

comparison predicates.

- The planner uses a cost-based model to decide whether to use a bitmap path.

Example: The following query triggers a bitmap index scan, allowing the hash index to efficiently handle multi-value conditions in large datasets.

```
CREATE TABLE users(id int, name text) DISTRIBUTED BY (id);
CREATE INDEX ON users USING hash (name);

SELECT * FROM users WHERE name IN ('alice', 'bob', 'carol');
```

Limitations:

- Not suitable for unique or high-cardinality columns (such as user IDs or phone numbers).
- Not recommended for OLTP workloads with frequent updates.
- Always validate performance benefits through testing before adding bitmap indexes.

Manage indexes

Cluster a table by index

You can use the CLUSTER command to physically reorder table data based on an index. However, this operation can be time-consuming for very large tables. A more efficient alternative is to manually reorder the data by creating an intermediate table and inserting data in the desired order. For example:

```
CREATE TABLE new_table (LIKE old_table)
  AS SELECT * FROM old_table ORDER BY myixcolumn;
DROP old_table;
ALTER TABLE new_table RENAME TO old_table;
CREATE INDEX myixcolumn_ix ON old_table;
VACUUM ANALYZE old_table;
```

Create indexes

Use the `CREATE INDEX` command to create indexes on a table. The default index type is B-tree. For example, to create a B-tree index on the `gender` column of the `employee` table:

```
CREATE INDEX gender_idx ON employee (gender);
```

To create a bitmap index on the `title` column of the `films` table:

```
CREATE INDEX title_bmp_idx ON films USING bitmap (title);
```

Rebuild all indexes on a table

You can use the `REINDEX` command to rebuild all indexes on a table or a specific index:

```
REINDEX my_table;
```

Rebuild a single index on a table:

```
REINDEX my_index;
```

Drop indexes

Use the `DROP INDEX` command to remove an index. For example:

```
DROP INDEX title_idx;
```

During data loading, dropping all indexes beforehand and recreating them after the load is complete can often improve overall performance.

Index-only scan and covering index

Note

SynxDB supports index-only scans and covering indexes only on newly created tables.

What is an index-only scan

In SynxDB, all indexes are secondary and stored separately from the main data (heap).

In a typical index scan, the planner uses the index to locate matching tuple positions, then accesses the heap via tuple pointers to fetch the actual data. Because data in the heap is usually not stored contiguously, this approach can lead to significant random I/O, especially on traditional spinning disks. Although bitmap scans can reduce this overhead to some extent, heap access is still required.

An index-only scan is a scan method that can return results entirely from the index, without accessing the heap, which significantly improves query performance.

Note

Starting from v4.0.0, SynxDB (using the ORCA optimizer) supports index-only scans on append-optimized (AO) and PAX tables, improving performance for repeated-read query workloads on these table types.

Requirements

Two conditions must be met to enable index-only scans:

- The index type must support index-only scans:
 - B-tree indexes always support it.
 - GiST and SP-GiST indexes support it with certain operator classes.
 - GIN indexes do not support it (they only store partial field data).
 - Other index types generally do not support it.

- All columns referenced in the query must be included in the index:

Example queries that can use index-only scans:

```
SELECT x, y FROM tab WHERE x = 'key';
SELECT x FROM tab WHERE x = 'key' AND y < 42;
```

Example queries that cannot use index-only scans:

```
SELECT x, z FROM tab WHERE x = 'key';
SELECT x FROM tab WHERE x = 'key' AND z < 42;
```

Additional note: MVCC visibility checks

Even if both conditions above are met, the system must still verify whether each record is visible to the current transaction, according to multi-version concurrency control (MVCC) rules. Because visibility information is not stored in indexes, the system usually needs to access the heap for confirmation.

To reduce the performance overhead of heap access, SynxDB uses a visibility map mechanism:

- If all tuples in a heap page are visible to all transactions, the page is marked as “all-visible” .
- During query execution, the system checks the visibility map first.
- If the page is “all-visible” , it skips heap access and returns results directly.
- Otherwise, the heap must still be accessed for visibility checks.

Visibility maps are lightweight and usually cached entirely in memory, significantly reducing the cost of random I/O during queries. As a result, index-only scans offer real performance benefits only when most heap pages are marked as “all-visible” .

Purpose of covering indexes

To better support index-only scans, you can explicitly create covering indexes that include all columns referenced in the query. SynxDB supports the `INCLUDE` clause, which allows you to add non-filter columns to the index:

Example:

```
-- Traditional index: accelerates only the WHERE clause, does not support
-- index-only scan
CREATE INDEX tab_x ON tab(x);

-- Covering index: includes column y in the index, supports index-only scan
CREATE INDEX tab_x_y ON tab(x) INCLUDE (y);

-- Example query
SELECT y FROM tab WHERE x = 'key';
```

Columns added using `INCLUDE` are not used for index matching. They are included only to cover the columns returned by the query. Therefore:

- They do not need to support indexable operations.
- In a unique index, they are not considered in uniqueness enforcement.

Example:

```
CREATE UNIQUE INDEX tab_x_y ON tab(x) INCLUDE (y);
-- Uniqueness is enforced on column x only, not on y
```

Limitations and considerations:

- Index tuple size is limited. Including wide columns might lead to large index size and insertion failures.
- Adding non-key columns increases index size and might affect read or update performance.
- Covering indexes only help trigger index-only scans if the table is rarely updated (i.e., most pages are marked as all-visible).
- Expressions are not currently supported in the `INCLUDE` clause.

- Currently, only B-tree and GiST index types support `INCLUDE`.

i Note

Starting from v4.0.0, SynxDB (using the ORCA optimizer) supports PostgreSQL-style `INCLUDE` clauses on AO and PAX tables, enabling the creation of covering indexes.

From v4.0.0 onward, the GPORCA optimizer considers both the width and the number of included columns when selecting an index-only scan path. If multiple indexes satisfy the query predicates and output requirements, the optimizer prefers narrower indexes with fewer fields to reduce I/O costs.

Example:

```
CREATE TABLE t1 (c1 int, c2 int, c3 int, c4 int, c5 int);

CREATE INDEX idx_large ON t1 USING btree(c1) INCLUDE (c2, c3, c4, c5);
CREATE INDEX idx_c1 ON t1 USING btree(c1);

EXPLAIN ANALYZE SELECT c1 FROM
```

Dynamic index-only scan

Dynamic index-only scan is an efficient query strategy used by SynxDB (with the ORCA optimizer) when querying partitioned tables. It combines two key techniques:

- **Index-only scan:** The query accesses only the index and avoids heap access, provided all referenced columns are covered by the index and the corresponding pages are marked as “all-visible” (e.g., after a `VACUUM` operation).
- **Dynamic scan:** During execution, only the relevant partitions are selected based on query conditions, avoiding unnecessary partition access (i.e., partition pruning).

The core idea of dynamic index-only scan is to combine index-only access with partition pruning. This allows the system to scan only the relevant partitions and skip heap access, greatly improving query efficiency.

It applies in the following scenarios:

- The target is a partitioned table.
- The query references only columns included in the index (i.e., using a covering index).
- The table has been vacuumed, and related pages are marked as all-visible.
- The table is wide, but the index is compact and includes only necessary columns.

Dynamic index-only scan significantly improves performance, reduces I/O, adapts automatically to partition structure, and is completely transparent to users.

This feature is enabled by default. If it is disabled, run the following command to enable it:

```
SET optimizer_enable_dynamicindexonlyscan = on;
```

Example:

```
CREATE TABLE pt(a int, b text, c text)
PARTITION BY RANGE(a) (START (0) END (100) EVERY (20));

CREATE INDEX idx ON pt(a); -- Covering index, includes only column a

-- Insert a large volume of data and clean up
INSERT INTO pt ...
VACUUM pt;

-- Query involves only column a and supports partition pruning
SELECT a FROM pt WHERE a < 42;
```

Backward index scan

For queries that include ORDER BY ... DESC, the GPORCA optimizer might choose a backward index scan path using a B-tree index to avoid additional sorting operations.

Example:

```
CREATE TABLE foo (a int, b int, c int) DISTRIBUTED BY (a);
CREATE INDEX foo_b ON foo(b);
INSERT INTO foo SELECT i, i, i FROM generate_series(1,10000) i;
ANALYZE foo;
```

(continues on next page)

(continued from previous page)

```
EXPLAIN SELECT * FROM foo ORDER BY b DESC LIMIT 1;
```

In this query, although the `foo_b` index is ordered in ascending order by default, the query requires descending order. The optimizer automatically selects a Backward IndexScan to scan the index in descending order, eliminating the need for a `Sort` node:

```
Limit
  -> Gather Motion 3:1
    Merge Key: b
      -> Limit
        -> Index Scan Backward using foo_b on foo
```

This optimization applies to both regular index scans (`IndexScan`) and index-only scans (`IndexOnlyScan`).

The following conditions must be met for the optimization to take effect:

- A B-tree index is used.
- The query includes an `ORDER BY` clause that matches the index column.
- The sort direction in the query is opposite to the index's default order.
- If `NULLS FIRST` or `NULLS LAST` is specified, the sort behavior must also match.

Backward IndexScan can significantly reduce the overhead of sorting operations, making it particularly effective for pagination or top-N queries.

Check index usage

Indexes in SynxDB require no manual tuning or maintenance. However, you can evaluate their effectiveness by inspecting the actual query execution plan. Use the `EXPLAIN` command to see whether a query uses an index.

The query plan reveals how the database executes a query (known as *plan nodes*) and provides cost estimates for each step. To check if an index is used, look for the following nodes in the `EXPLAIN` output:

- **Index Scan:** retrieves data directly using the index.

- **Bitmap Index Scan:** builds a bitmap based on query conditions and performs bitmap scan.
- **Bitmap Heap Scan:** fetches rows from the heap using the bitmap.
- **BitmapAnd / BitmapOr:** merges multiple bitmaps for compound conditions.

Here are some recommendations for evaluating and optimizing index usage:

- Always run `ANALYZE` after creating or modifying an index. This updates table statistics, which the optimizer uses to estimate row counts and choose the best plan.
- Test with real data. Test data reflects the test environment only and does not represent actual production data distribution or behavior.
- Avoid using small datasets for testing. Their behavior often differs significantly from large-scale data, and the results might not be meaningful.
- Pay attention to data distribution when generating test data. Random and skewed values can all affect the accuracy of statistics.
- You can adjust GUC parameters to force the optimizer to use specific plans and evaluate index behavior. For example:

```
SET enable_seqscan = off;
SET enable_nestloop = off;
```

Disabling sequential scan and nested loop join encourages the system to choose an index-based path. Use `EXPLAIN ANALYZE` on both indexed and non-indexed queries, compare the execution times and plans, and determine whether the index improves performance.

BRIN Indexes

When a table is very large and the values of certain columns are naturally correlated with their physical storage locations, you can use block range indexes (BRIN). A BRIN index summarizes values for a range of blocks (or pages), where each range includes a group of physically adjacent pages in the table. For example, in a table that stores order data, earlier orders tend to appear at the beginning of the table. Similarly, in a table of postal codes, codes from the same city may be physically clustered.

BRIN indexes participate in queries using bitmap index scans. When the summary information in the index matches the query condition, the query accesses all pages within that block range and returns all tuples found. The executor then verifies these tuples one by one and filters out those that do not satisfy the condition. For this reason, BRIN is a “lossy” index. Since BRIN indexes are typically very small, they introduce minimal overhead compared to sequential scans, while enabling the system to skip many unrelated blocks.

The type of data a BRIN index stores and the queries it can accelerate depend on the operator class selected for each column. For linearly ordered data types, the index records the minimum and maximum values within each range. For geometric types, it stores the bounding box that encloses all objects in the range.

The size of each block range is determined by the `pages_per_range` parameter at index creation. The number of index entries equals the total number of pages in the table divided by `pages_per_range`. A smaller value leads to a larger index (more entries), but the summaries become more fine-grained, allowing the scan to skip more unrelated data blocks. The default `pages_per_range` value is 32 for heap tables and 1 for append-optimized tables.

Note

Starting from v4.0.0, BRIN indexes are supported on AO and CO tables. A chained revmap structure is used to optimize storage and reduce space usage. This optimization improves both storage layout and summary maintenance, and allows summaries to be generated manually using `brin_summarize_range()`.

Although BRIN indexes can be created and maintained on AO/CO tables, whether the query planner chooses to use them depends on query predicates, data distribution, page count, and the heuristic cost model. As such, not all BRIN use cases lead to automatic performance gains.

BRIN is best suited for large datasets with strong data clustering.

Maintain BRIN indexes

When a BRIN index is created, SynxDB scans existing heap pages and generates summary tuples for each block range, including the possibly incomplete tail range. As new data fills additional pages, the system attempts to update the summary information for ranges that have already been summarized. However, for ranges that have not yet been summarized, the system does not automatically create summary tuples. These ranges remain in an “unsummarized” state and require manual intervention to generate summaries.

You can generate initial summaries using the following methods:

- During a VACUUM operation, the system automatically summarizes all unprocessed block ranges.

Alternatively, you can manually control summarization using the following functions:

- `brin_summarize_new_values (regclass)`: summarizes all unprocessed block ranges.
- `brin_summarize_range (regclass, bigint)`: summarizes only the range containing the specified page, if it has not been summarized.

To remove summary information instead, you can use the `brin_desummarize_range (regclass, bigint)` function to clear summaries for a specific range. This is especially useful when data changes make existing summaries inaccurate.

The table below lists the functions available for BRIN index maintenance. These functions cannot be executed in recovery mode and are restricted to superusers or the index owner.

Name	Return type	Description
<code>brin_summarize_new_values (index regclass)</code>	integer	Summarizes all unprocessed block ranges.
<code>brin_summarize_range (index regclass, blockNumber bigint)</code>	integer	Summarizes the range containing the specified block, if not already summarized.
<code>brin_desummarize_range (index regclass, blockNumber bigint)</code>	integer	Removes the summary for the range containing the specified block, if already summarized.

The `brin_summarize_new_values` function accepts an index name or OID and automatically finds unsummarized ranges. It scans the corresponding table pages and generates new summary index tuples. It returns the number of summary entries created. The `brin_summarize_range` function processes only the range that contains the specified block number.

Note

- Normally, after running VACUUM, the `brin_summarize_new_values()` function does not generate new summaries because the default behavior includes summarizing all newly added block ranges during cleanup. This function is mainly used for manual summary refreshes, which is useful when auto-vacuum is disabled or the table is frequently updated.
- If the table has few pages or the query predicate is not selective enough, the planner might still choose not to use a BRIN index path.
- The `brin_summarize_range` and `brin_desummarize_range` functions only affect ranges that actually exist. If a non-existent block number is provided, the function returns 0. Due to uneven data distribution, some ranges may exist only on a subset of segments. In that case, the function returns the number of segments that successfully processed the range.

Choose Table Storage Model

Heap and Append-optimized Table Storage Models

SynxDB provides heap and append-optimized storage models. The storage model you choose depends on the type of data you are storing and the type of queries you are running. This section provides an overview of the two storage models and guidelines for choosing the optimal storage model for your data.

Heap storage

By default, SynxDB uses the same heap storage model as PostgreSQL. Heap table storage works best with OLTP-type workloads where the data is often modified after it is initially loaded. UPDATE and DELETE operations require storing row-level versioning information to ensure reliable database transaction processing. Heap tables are best suited for smaller tables, such as dimension tables, that are often updated after they are initially loaded.

Append-optimized storage

Append-optimized table storage works best with denormalized fact tables in a data warehouse environment. Denormalized fact tables are typically the largest tables in the system. Fact tables are usually loaded in batches and accessed by read-only queries. Moving large fact tables to an append-optimized storage model eliminates the storage overhead of the per-row update visibility information. This allows for a leaner and easier-to-optimize page structure. The storage model of append-optimized tables is optimized for bulk data loading. Single row INSERT statements are not recommended.

To create a table with specified storage options

Row-oriented heap tables are the default storage type.

```
CREATE TABLE foo (a int, b text) DISTRIBUTED BY (a);
```

Use the WITH clause of the CREATE TABLE command to declare the table storage options. The default is to create the table as a regular row-oriented heap-storage table. For example, to create an append-optimized table with no compression:

```
CREATE TABLE bar (a int, b text)
WITH (appendoptimized=true)
DISTRIBUTED BY (a);
```

Note

You use the appendoptimized=value syntax to specify the append-optimized table

storage type. `appendoptimized` is a thin alias for the `appendonly` legacy storage option. SynxDB stores `appendonly` in the catalog, and displays the same when listing storage options for append-optimized tables.

`UPDATE` and `DELETE` are not allowed on append-optimized tables in a repeatable read or serializable transaction and will cause the transaction to end prematurely.

Choose row or column-oriented storage

SynxDB provides a choice of storage orientation models: row, column, or a combination of both. This section provides general guidelines for choosing the optimal storage orientation for a table. Evaluate performance using your own data and query workloads.

- **Row-oriented storage:** good for OLTP workloads with many iterative transactions and where many columns of a single row are needed at once, making retrieval efficient.
- **Column-oriented storage:** good for data warehouse workloads with aggregations of data computed over a small number of columns or for single columns that require regular updates without modifying other column data.

For most general-purpose or mixed workloads, row-oriented storage offers the best combination of flexibility and performance. However, there are use cases where a column-oriented storage model provides more efficient I/O and storage. Consider the following requirements when deciding on the storage orientation model for a table:

- **Updates of table data:** If you load and update table data frequently, choose a row-oriented heap table. Column-oriented table storage is only available on append-optimized tables.
- **Frequent INSERTs:** If rows are frequently inserted into the table, consider a row-oriented model. Column-oriented tables are not optimized for write operations because column values for a row must be written to different places on disk.
- **Number of columns requested in queries:** If you typically request all or the majority of columns in the `SELECT` list or `WHERE` clause of your queries, consider a row-oriented model. Column-oriented tables are best suited to queries that aggregate many values of a single column where the `WHERE` or `HAVING` predicate is also on the aggregate column. For example:

```
SELECT SUM(salary) ...
```

```
SELECT AVG(salary) ... WHERE salary > 10000
```

Or where the WHERE predicate is on a single column and returns a relatively small number of rows. For example:

```
SELECT salary, dept ... WHERE state='CA'
```

- **Number of columns in the table:** Row-oriented storage is more efficient when many columns are required at the same time, or when the row size of a table is relatively small. Column-oriented tables can offer better query performance on tables with many columns where you access a small subset of columns in your queries.
- **Compression:** Column data has the same data type, so storage size optimizations are available in column-oriented data that are not available in row-oriented data. For example, many compression schemes use the similarity of adjacent data to compress. However, the greater adjacent compression achieved, the more difficult random access can become, as data must be uncompressed to be read.

To create a column-oriented table

The WITH clause of the CREATE TABLE command specifies the table's storage options. The default is a row-oriented heap table. Tables that use column-oriented storage must be append-optimized tables. For example, to create a column-oriented table:

```
CREATE TABLE bar (a int, b text)
  WITH (appendoptimized=true, orientation=column)
  DISTRIBUTED BY (a);
```

PAX Storage Format

SynxDB supports the PAX (Partition Attributes Across) storage format.

PAX is a database storage format that combines the benefits of row-based storage (NSM, N-ary Storage Model) and column-based storage (DSM, Decomposition Storage Model). It is designed to improve query performance, particularly in terms of cache efficiency. In OLAP scenarios, PAX offers batch write performance similar to row-based storage and read performance like column-based storage. PAX can adapt to both cloud environments with object storage models and traditional offline physical file-based storage methods.

Compared to traditional storage formats, PAX has the following features:

- Data updates and deletions: PAX uses a mark-and-delete approach for data updates and deletions. This effectively manages changes in physical files without immediately rewriting the entire data file.
- Concurrency control and read-write isolation: PAX uses Multi-Version Concurrency Control (MVCC) to achieve efficient concurrency control and read-write isolation. The control granularity reaches the level of individual data files, enhancing operation safety and efficiency.
- Index support: PAX supports B-tree indexes, which help speed up query operations. This is particularly useful for improving data retrieval speed when dealing with large amounts of data.
- Data encoding and compression: PAX offers multiple data encoding methods (such as run-length encoding and delta encoding) and compression options (such as zstd and zlib), with various compression levels. These features help reduce storage space requirements while optimizing read performance.
- Statistics: Data files contain detailed statistics that are used for quick filtering and query optimization, reducing unnecessary data scanning and speeding up query processing.
- Vectorized engine: PAX also supports a vectorized engine. It aims to further enhance data processing capabilities and query performance, especially in applications like data analysis and report generation.

Applicable scenarios

The hybrid storage capability of PAX makes it suitable for complex OLAP applications that need to handle large amounts of data writes and frequent queries. Whether you are looking for a high-performance data analysis solution in a cloud infrastructure or dealing with large datasets in a traditional data center environment, PAX can provide strong support.

Usage

Create a PAX table

To create a table in PAX format, you need to set the table access method to PAX. You can do this in one of the following ways:

- Use the `USING PAX` clause explicitly when creating the table, for example:

```
CREATE TABLE t1(a int, b int, c text) USING PAX;
-- t1 is a PAX table and can be used like a normal heap table.
```

- Set the default table access method to PAX and then create the table:

```
-- Set the default table access method. From now on, newly created tables
will use PAX format.
SET default_table_access_method = pax;

-- Implicitly use the default access method, which is PAX.
CREATE TABLE t1(a int, b int, c text);
```

When creating a table, you can also specify minimum and maximum value information for certain columns to speed up queries:

```
-- Use WITH(minmax_columns='b,c') to specify that columns b and c
-- should record min and max statistics.
-- This helps optimize queries involving these two columns,
-- because the system can quickly determine which data blocks might contain
matching data.
CREATE TABLE p2(a INT, b INT, c INT) USING pax WITH(minmax_columns='b,c');
```

(continues on next page)

(continued from previous page)

```
INSERT INTO p2 SELECT i, i * 2, i * 3 FROM generate_series(1,10)i;

-- because column b has minmax statistics,
-- the system can quickly locate the data blocks that might contain the value,
speeding up the query.

SELECT * FROM p2 WHERE b = 4;

-- Similarly, because of the minmax information on column b, the system can
quickly determine that no data blocks can meet this condition
-- (if all generated values are positive), possibly returning no data
immediately and avoiding unnecessary data scans.

SELECT * FROM p2 WHERE b < 0;

-- Modify the minmax statistics settings for table p2 to apply only to column
b. For data inserted later,
-- only column b will maintain this statistical information, and it won't
affect existing data or trigger any rewrites or adjustments.

ALTER TABLE p2 SET(minmax_columns='b');
```

View the format of an existing table

To check whether a table is in PAX format, you can use one of the following methods:

- Use the `psql` command `\d+`:

```
gpadmin=# \d+ t1
                                         Table "public.t1"
   Column |  Type   | Collation | Nullable | Default | Storage | Compression
   | Stats target | Description
-----+-----+-----+-----+-----+-----+-----+
a      | integer |           |          |          | plain   |
|           |
b      | integer |           |          |          | plain   |
|           |
c      | text    |           |          |          | extended|
|           |
Distributed by: (a)
Access method: pax
```

- Query the system catalog tables pg_class and pg_am:

```
SELECT relname, amname FROM pg_class, pg_am WHERE relam = pg_am.oid AND
relname = 't1';

relname | amname
-----+-----
t1      | pax
(1 row)
```

Support for TOAST

If some columns in a PAX table contain large values, you can enable TOAST storage to store these large values in a separate TOAST file. This helps to make the data in the main data file more compact, allowing you to scan more tuples within the same data size.

By default, the TOAST storage is enabled for PAX tables. Unlike PostgreSQL, the TOAST storage supported by PAX does not rely on Page management, which allows PAX to store data larger than 2 MiB.

You can configure TOAST-related thresholds using the following parameters. For more details, refer to [PAX-related system parameters](#).

- pax_enable_toast
- pax_min_size_of_compress_toast
- pax_min_size_of_external_toast

Support for clustering

SynxDB has introduced z-ordering support for PAX tables, which greatly improves performance when managing and querying multi-dimensional data. Using the clustering feature, you can organize data in PAX tables using two different sorting strategies: index-based clustering and reoptions-based clustering. Note that the two strategies are mutually exclusive, and you can only choose one.

Clustering strategies

You can use the clustering feature of SynxDB to physically sort and optimize table data. The goal is to improve query performance by reorganizing the physical storage order of the data. In PAX tables, the clustering feature provides two different strategies for sorting data:

- Index-based clustering. This strategy is based on B-tree indexes and works similarly to the native clustering method in PostgreSQL. It is suitable when an index has already been created for the table.

Example:

```
CREATE TABLE t2(c1 int, c2 int) USING PAX;
CREATE INDEX c1_idx ON t2(c1);
CLUSTER t2 USING c1_idx;
DROP TABLE t2;
```

- `reloptions`-based clustering. PAX tables support clustering operations based on `reloptions` where `cluster_columns` are specified, and it provides the following sorting methods:
 - Z-order clustering: This method encodes the values of multiple columns into a byte array and sorts by this byte array. It is ideal for cases where multiple columns are used as query conditions (with no fixed order for the columns in the query). This method greatly enhances performance for multi-dimensional data queries. However, it is not suitable for string columns with the same prefix. When `cluster_type` is not specified, PAX uses z-order sorting as the default clustering strategy:

```
CREATE TABLE t2(c1 int, c2 float, c3 text) USING PAX WITH (cluster_
columns='c1,c2');
INSERT INTO t2 SELECT i, i, i::text FROM generate_series(1, 100000) i;
CLUSTER t2;
DROP TABLE t2;
```

In this example, the data in table `t2` is sorted using z-order by the `c1` and `c2` columns. This sorting method provides great performance improvement in multi-dimensional query scenarios.

- Lexical clustering: This method sorts by the values and order of the columns, and it is

primarily used for sorting and optimizing queries on string columns with the same prefix.
For example:

```
CREATE TABLE t2(c1 int, c2 float, c3 text) USING PAX WITH (cluster_
columns='c1,c2', cluster_type='lexical');
INSERT INTO t2 SELECT i, i, i::text FROM generate_series(1, 100000) i;
CLUSTER t2;
DROP TABLE t2;
```

In this example, the data in table t2 is sorted lexically by the c1 and c2 columns. This sorting method is well-suited for columns with the same prefix.

Conflicts between clustering types

Note that index-based clustering and cluster_columns-based clustering cannot be used at the same time. If a table has already been clustered based on an index (or cluster_columns), you cannot specify cluster_columns (or an index) for clustering.

An example of handling cluster conflicts:

```
-- Creates a table and uses index-based clustering
CREATE TABLE t2(c1 int, c2 float, c3 text) USING PAX;
CREATE INDEX t2_idx ON t2(c1);
CLUSTER t2 USING t2_idx;

-- Tries to use cluster columns (will fail)
ALTER TABLE t2 SET (cluster_columns='c1,c2', cluster_type='zorder');
-- Tries to use lexical clustering (will fail)
ALTER TABLE t2 SET (cluster_columns='c1,c2', cluster_type='lexical');

-- Drops the index and successfully use lexical clustering
DROP INDEX t2_idx;
ALTER TABLE t2 SET (cluster_columns='c1,c2', cluster_type='lexical');
```

In the above example, after table t2 is clustered using an index, attempts to set z-order or lexical clustering using cluster_columns will fail, until the index is dropped.

Bloom filter support

SynxDB introduces support for bloom filters, allowing you to generate and maintain bloom filter information at the column level. This feature helps quickly filter data blocks and improves query performance, especially when using `IN` conditions with multiple values, greatly reducing unnecessary data scans.

Bloom filter options

You can specify the columns for which you want to record bloom filter information by setting the `bloomfilter_columns` option. For example:

```
CREATE TABLE p1 (
    a int,
    b int,
    c text
) USING PAX WITH (bloomfilter_columns='b,c,a');
```

In this example, bloom filters will be generated for columns `b`, `c`, and `a` in the table `p1`.

The size of the bloom filter is controlled by the following GUC (global user configuration) parameters:

- `pax_max_tuples_per_file`: Controls the maximum number of tuples stored in each data file.
- `pax_bloom_filter_work_memory_bytes`: Controls the maximum memory allowed for the bloom filter.

For example:

```
-- Sets the maximum number of tuples per file.
SET pax_max_tuples_per_file = 131073;

-- Sets the maximum bloom filter memory.
SET pax_bloom_filter_work_memory_bytes = 1048576; -- 1MB

-- Creates a table and specifies bloom filter options for columns.
CREATE TABLE p1 (
```

(continues on next page)

(continued from previous page)

```
a int,
b int,
c text
) USING PAX WITH (bloomfilter_columns='b,c,a');
```

The size of the generated bloom filter is calculated as follows:

```
ceil(min(pax_max_tuples_per_file * 2, pax_bloom_filter_work_memory_bytes))
```

`ceil` is a rounding function, which ensures that the size of the bloom filter is always a power of 2.

Sparse filtering

Sparse filtering is a data scanning optimization provided by the PAX storage format. It improves query performance by skipping file scans that do not meet conditions and reducing the amount of data block to be scanned within files.

To use sparse filtering, make sure that the following conditions are met:

- The system parameter `pax_enable_sparse_filter` is set to ON (default value).
- Statistics have been configured for the relevant columns (`min/max` or bloom filter).
- The query contains filter conditions (WHERE clause) for these columns

Expression support examples

PAX supports basic conditional expressions. For example:

```
-- Creates test table
CREATE TABLE a(v1 int, v2 int, v3 int) USING PAX WITH (minmax_columns='v1, v2, v3');

-- Basic conditions
SELECT * FROM a WHERE v1 <= 3;                                         -- Full
condition support. root is OpExpr.
SELECT * FROM a WHERE v1 != NULL;                                         -- Full
condition support. root is NULLTest.
```

(continues on next page)

(continued from previous page)

```
-- Multiple conditions
SELECT * FROM a WHERE v1 <= 3 AND v2 >= 3; -- Full
condition support. Tree structure has only one level.
SELECT * FROM a WHERE v1 <= 3 AND v2 >= 3 AND v3 >= 3; -- Full
condition support.
SELECT * FROM a WHERE v1 <= 3 AND v2 >= 3 AND v3 != NULL; -- Full
condition support.
```

PAX supports nested expression structures and certain operators. For example:

```
-- Creates test table.
CREATE TABLE a(v1 int, v2 int, v3 int) USING PAX WITH(minmax_columns='v1, v2,
v3');

-- Nested expressions.
SELECT * FROM a WHERE (v1 <= 3 OR v2 > 3) AND v3 >= 10; -- Full
support for all conditions.
SELECT * FROM a WHERE (v1 <= 3 AND v2 > 3) OR v3 >= 10; -- Supports
complete nested expressions.

-- Operators.
SELECT * FROM a WHERE v1 + v2 <= 3;
SELECT * FROM a WHERE v1 + 10 <= 3;
```

PAX can handle complete expression trees, including nested AND/OR conditions. This means that all the above queries can be optimized, and SynxDB will use all available filter conditions to improve query efficiency.

Supported expression types

PAX sparse filtering supports the following expression types:

- Arithmetic operators (*OpExpr*)
 - Supports addition (+), subtraction (-), and multiplication (*)
 - Division operations are not supported (due to difficulty in estimating negative number ranges)

- Examples: `a + 1, 1 - a, a * b`
- Comparison operators (*OpExpr*)
 - Supports `<, <=, =, >=, >`
 - Examples: `a < 1, a + 1 < 10, a = b`
- Logical operators (*BoolExpr*)
 - Supports AND, OR, NOT
 - Example: `a < 10 AND b > 10`
- NULL value tests (*NullTest*)
 - Supports `IS NULL, IS NOT NULL`
 - Examples: `a IS NULL, a IS NOT NULL`
- Type casting (*FuncExpr*)
 - Only supports basic type casting
 - Example: `a::float8 < 1.1`
 - Custom functions are not supported, such as `func(a) < 10`
- IN operator (*ScalarArrayOpExpr*)
 - Supports IN expressions
 - Example: `a IN (1, 2, 3)`

Partial condition support

When a query contains unsupported expressions (such as custom functions), PAX adopts a partial condition support strategy:

- Identifies and extracts supported conditions.
- Uses supported conditions for sparse filtering.

For example:

```
-- Creates custom function v2
CREATE OR REPLACE FUNCTION func(v2 double precision)
RETURNS double precision AS $$
BEGIN
    RETURN v2 * 2;
END;
$$ LANGUAGE plpgsql;

-- Uses custom function in query
SELECT * FROM a WHERE v1 < 3 AND func(v2) < 10; -- PAX will use v1 < 3 for filtering
```

In this example, although `func(v2) < 10` cannot be used for sparse filtering, SynxDB still uses `v1 < 3` to optimize query performance. This approach ensures that partial performance optimization is achieved even in complex queries.

Note

- The effectiveness of sparse filtering depends on data distribution and query conditions.
- It is recommended to enable statistics on columns that are frequently used as filter conditions.
- Certain conditions (such as custom functions) are ignored.

View data change records on PAX tables in WAL logs

Data operations related to PAX tables are recorded in the Write-Ahead Logging (WAL) system. This ensures reliable backups between primary and mirror nodes, which assists in failover processes.

You can use the `pg_waldump` tool to view the WAL logs for PAX tables. The logs are stored in the `$COORDINATOR_DATA_DIRECTORY/pg_wal` directory or the `pg_wal` directory within the Segment node's data directory.

The PAX table WAL logs can be used for these purposes:

- **Data recovery:** Helps restore data during system failures or data inconsistencies.

- **Failover support:** Ensures data synchronization between primary and mirror nodes.
- **Debugging and analysis:** Identifies and analyzes specific operations on PAX tables.

Example: operate PAX tables and view WAL logs

1. Create a PAX table.

```
CREATE TABLE a (
    v1 int,
    v2 int,
    v3 int
) USING PAX;
```

This creates a PAX table `a` with columns `v1`, `v2`, and `v3`.

2. Insert data into the table:

```
INSERT INTO a VALUES (1, 2, 3);
```

3. Locate the node where the PAX table resides:

```
-- Views the IDs and data directory locations of all nodes in the cluster
SELECT * FROM gp_segment_configuration;

-- Identifies the Segment ID of table a based on the results of the above
query
-- This helps determine the segment and data directory location of table a
SELECT gp_segment_id FROM gp_distribution_policy WHERE localoid = 'a
'::regclass;
```

4. Find the corresponding WAL log in the `pg_wal` directory of the identified node. For example:

```
ls $COORDINATOR_DATA_DIRECTORY/pg_wal
```

Locate the most recent WAL file to analyze the relevant operations. In this example, assume the WAL log file is named `00000001000000000000000A`.

5. Query the `realfilenode` of the PAX table to associate the WAL log with the PAX table:

```
SELECT relfilenode FROM pg_class WHERE relname = 'a';
```

The `relfilenode` is the unique identifier of the table in the WAL log.

6. Use the `pg_waldump` tool to parse the WAL log and identify operations related to the PAX table:

```
pg_waldump -f $COORDINATOR_DATA_DIRECTORY/pg_wal/00000001000000000000A
```

In the output, search for the previously obtained `relfilenode` to find operations associated with the PAX table.

Example output of `pg_waldump`:

```
rmgr: PAX      len (rec/tot):     68/    104, tx:      593, lsn: 0/016E1D98,
prev 0/016E1D60, desc: INSERT off 3, blkref #0: rel 1663/16384/19780 blk 0
```

In this example:

- `rmgr: PAX` indicates the record type.
- `rel 1663/16384/19780` corresponds to the `relfilenode` of the PAX table.
- `INSERT` indicates that an insert operation was recorded.

By following the steps above, you can track and analyze WAL logs related to PAX tables in SynxDB for debugging, recovery, or performance optimization purposes.

Limitations for PAX tables

- For index support, the PAX storage format currently only supports B-tree (`btree`) indexes. Bugs might occur when using GiST, SP-GiST (`gist/spgist`) or Brin indexes. Supports for other index types are still experimental and might be unavailable.
- Currently, if a field is too long, it will be stored in a TOAST file. This TOAST is different from PostgreSQL's TOAST tables, and they only share the same name.
- Unlike traditional heap tables, PAX format does not support TOAST fields. Currently, all column data is stored in the same data file.

- The PAX format does not support data backup and restore using `pg_dump` or `pg_restore`. PAX tables are ignored during these operations.
- PAX format does not support Write-Ahead Logging (WAL), so there is no data backup between the primary and mirror servers.

PAX-related SQL options

PAX supports SQL options to control its behavior. You can use these options in the `WITH()` clause, for example, `WITH(minmax_columns='b,c', storage_format=porc')`.

Name	Type	Valid values	Description
<code>storage_format</code>	<code>string</code>	<ul style="list-style-type: none"> <code>porc</code> <code>porc_vec</code> 	Controls the internal storage format. <code>porc</code> is the default value, which stores data in a regular format and does not preserve null values. <code>porc_vec</code> stores data in a vector format and always preserves null values for fixed-length fields, regardless of whether the column value is null or not.
<code>compressstype</code>	<code>string</code>	<ul style="list-style-type: none"> <code>none</code> <code>rle</code> <code>delta</code> <code>zstd</code> <code>zlib</code> <code>dict</code> 	Specifies the compression method for column values. You can only choose one method. The default value is <code>none</code>. <ul style="list-style-type: none"> <code>none</code> means no compression is applied. <code>rle</code> uses run-length encoding to compress repeated consecutive data. <code>delta</code> is used for numeric columns, storing the difference between adjacent values. <code>zstd</code> uses the zstd algorithm, which offers high compression ratio and fast compression speed. <code>zlib</code> is a general-purpose compression algorithm, suitable for compressing general data. <code>dict</code> uses dictionary encoding to speed up the processing of many repeated strings. Currently, it is an experimental feature and is not recommended for production use.
<code>compresslevel</code>	<code>int</code>	Range [0, 19]	Indicates the compression level. The default value is 0. A smaller value means faster compression, while a larger value provides higher compression. This option is only effective when used with <code>compressstype</code> .
<code>minmax_columns</code>	<code>string</code>	Valid column names in the table, separated by commas	Records minmax statistics for the defined columns to speed up data queries. After renaming a column, the statistics will no longer be recorded for that column. If you modify <code>minmax_columns</code> using <code>ALTER TABLE</code> , the change only applies to data files written afterward and does not affect existing data files.
<code>cluster_columns</code>	<code>string</code>	A comma-separated list of valid column names in the table.	Indicates that the PAX table stores the internal data in a clustered way. When using the <code>cluster</code> command, the data can be sorted by these columns, and the sorting method is controlled by the <code>cluster_type</code> parameter.

continues on next page

Table 1 – continued from previous page

Name	Type	Valid values	Description
bloomfilter_columns	string	A comma-separated list of valid column names in the table.	Calculates bloom filters for the data in the specified columns to help data filtering.
parallel_workers	int	[0, 64]	A PostgreSQL option that sets the number of parallel processes for concurrent scanning.
cluster_type	string	lexical and zorder	Specifies the cluster type when clustering is based on custom columns defined in <code>reloptions</code> rather than an index. Valid values are <code>lexical</code> and <code>zorder</code> . <code>lexical</code> sorts by the values and order of the columns, while <code>zorder</code> encodes the values of multiple columns into a byte array and sorts by that array.

The values of these options only affect newly inserted and updated data and do not change the existing data.

PAX-related system parameters

The following system parameters (GUC) are used to set the behavior of PAX tables in the current session. Use the command `SET <parameter>=<value>` to configure them, for example, `SET pax_enable_sparse_filter=on`.

Parameter name	Value type	Valid values	Description
pax_enable_sparse_filter	bool	on and off	Specifies whether to enable sparse filtering based on statistics. The default value is on, which means that sparse filtering is enabled by default.
pax_enable_row_filter	bool	on and off	Specifies whether to enable row filtering. The default value is off, which means that row filtering is disabled. It is not recommended to enable this parameter.
pax_scan_reuse_buffer_size	int	Range [1048576, 33554432] (1MiB to 32MiB)	The buffer block size used during scanning. The default value is 8388608 (8MiB).
pax_max_tuples_per_group	int	Range [5, 524288]	Specifies the maximum number of tuples allowed in each group. The default value is 131072.
pax_max_tuples_per_file	int	Range [131072, 8388608]	Specifies the maximum number of tuples allowed in each data file. The maximum value is 8388608. The default value is 131072.
pax_max_size_per_file	int	Range [8388608, 335544320] (8MiB to 320MiB)	The maximum physical size allowed for each data file. The default value is 67108864 (64MiB). The actual file size might be slightly larger than the set size. Very large or small values might negatively impact performance.
pax_enable_toast	bool	on and off. The default value is on.	Specifies whether to enable TOAST support.
pax_min_size_of_compress_toa	int	Range [524288, 1073741824] (512KiB to 1GiB). The default value is 524288 (512KiB).	Specifies the threshold for creating compressed TOAST tables. If the character length exceeds this threshold, SynxDB creates compressed TOAST tables for storage.
pax_min_size_of_external_toa	int	Range [10485760, 2147483647] (10MiB to 2GiB). The default value is 10485760 (10MiB).	Specifies the threshold for creating external TOAST tables. If the character length exceeds this threshold, SynxDB creates external TOAST tables for storage.
pax_default_storage_format	string	porc (the default value)	Controls the default storage format.
pax_bloom_filter_work_memory	int	Range [1024, 2147483647] (1KiB to 2GiB). The default value is 10240 (10KiB).	Controls the maximum memory allowed for bloom filter usage.

Best practices

- Use partitioning options:
 - It is recommended to use partitioning options when data needs to be imported based on an integer column and meets the following conditions:
 - * The data is evenly distributed across this column, with a wide range and no extreme concentration.
 - * This column is often used as a filter condition in queries or as a key for joins.
 - Keep in mind that the PAX partition key is only effective during a single batch data import. It cannot be adjusted between multiple data writes. The partition key settings only apply to future inserts or updates, so after changing the partition key, newly imported data will follow the new partition key.

- Use minmax statistics:
 - For columns with a wide data range that are often used in query filters, setting minmax values for these columns can greatly speed up the query process.
 - By using minmax statistics, if a column in a data file does not match the minmax values or null tests, the entire file can be skipped quickly, avoiding unnecessary data scans.
 - Important note: The effectiveness of minmax depends on how data is inserted. If the data in a PAX table is inserted in batches (such as with batch insert or copy) and the data range within each batch is continuous, then minmax will be very effective. However, if the data is inserted randomly, the effectiveness of minmax filtering may be reduced.
- PAX has been well-integrated with the vector execution engine. You can control whether to generate vector query plans by setting parameters in the query plan, which helps to improve query performance.

Vector plans take advantage of the columnar storage format of PAX tables, efficiently using memory and speeding up query execution. By processing data in batches, PAX tables can better support high-throughput queries. For more details, see *Vectorization Query Computing*.

For example:

```
SET vector.enable_vectorization = on; -- Enables vector query plans.
SET vector.max_batch_size = 16384; -- Sets the maximum number of rows per batch.
```

- If a PAX table has multiple columns used in query filters, you can use `with(cluster_columns='b,c,d', cluster_type='zorder')`. After sorting the data with zorder encoding, any of the cluster_columns can benefit from filtering. In comparison, for single-column-based cluster sorting, filtering is only effective on the sorted key, with minimal impact on other columns. Clustering sorting influences minmax-based filtering but does not affect bloom filter-based filtering.
- If your query uses filters like `column1 in (12, 11, 13)` or `column1 = 'X'`, you might consider using a bloom filter (`with(bloomfilter_columns='b,c')`). In internal implementation, PAX calculates bloom filters for certain data. If the bloom filter can confirm that the data is not in the block, PAX can skip the current block. Note that the effectiveness of bloom filter depends on several factors:

- The amount of data within a group. If the data in a group is too small, it might reduce the efficiency of vector execution.
 - The space used by the bloom filter.
- SynxDB PAX storage supports vector thread-level scanning, but this is limited to parallel scanning of local files only. For specific settings, see *Threaded execution on single node*.

UnionStore Storage Format

This document describes the use cases, methods, limitations, and frequently asked questions for UnionStore.

UnionStore is a storage engine for heap tables and their indexes, which, combined with SynxDB, forms a decoupled compute and storage architecture. The core idea of UnionStore is “Log is database,” where data is constructed by persisting and replaying logs from the compute layer, making it available for queries.

By decoupling compute and storage, UnionStore allows for adjusting compute resources based on the actual workload, improving cost-effectiveness. UnionStore supports multi-tenancy and multi-instance read/write operations for a single tenant, enabling efficient resource utilization and data sharing across multiple clusters.

Use cases

Building a UnionStore cluster to store business/user data is suitable for the following scenarios:

- **Write-intensive, read-light workloads:** For scenarios with high write volumes, low query volumes, and large datasets, UnionStore enables storage expansion and stores data on more cost-effective, reliable storage, improving storage cost-effectiveness.
- **Read-intensive, write-light workloads:** For scenarios with high query volumes and relatively low write volumes, which are heavily dependent on CPU and memory resources. You can store data in UnionStore and configure compute nodes with more compute resources and less local storage to enhance query performance.
- **Multi-tenancy:** UnionStore supports multi-tenancy, allowing data from multiple compute clusters to be stored under different tenants within the same UnionStore, achieving efficient resource utilization.

Prerequisites

To use the UnionStore feature on SynxDB, you must first have a running SynxDB cluster.

Install UnionStore

Before using UnionStore, you need to install it. Follow these steps for installation:

Caution

The following installation method is only for local deployment of SynxDB in a test environment and must not be used in a production environment.

1. In the cluster node directory, find the UnionStore installation package `unionstore.tar.gz` and the installation script `unionstore_deploy.sh`.
2. Open the `unionstore_deploy.sh` script file with a text editor like Vim. Fill in the required parameters as described in the script's comments, then save and close the file.

```

1 #!/usr/bin/env bash
2
3 # binary directories, put 'target' and 'pg_install' into this directory.
4 # this should be the same on all pageserver and safekeeper hosts.
5 unionstore_bin_dir=""          Path to the unionstore binary
6
7 # data directories, please make sure those directories are already exist and empty.
8 # page server data directory
9 page_server_data_dir=""
10 # safekeeper data directory for wal persistence
11 safekeeper1_data_dir=""
12 safekeeper2_data_dir=""          Create these local directories
13 safekeeper3_data_dir=""
14 # data directory for wal backup(optional),
15 # this dir is located with safekeeper_data_dir together.
16 # please make sure this directory is exist on all safekeeper hosts.
17 wal_backup_dir=""
18
19
20 # page server host

```

3. Run the `unionstore_deploy.sh` script. UnionStore will be deployed automatically.

Usage

Step 1: Create a UnionStore tenant

To create a UnionStore tenant, you need to use the `neon_local` tool included in the UnionStore installation package.

1. Set the `NEON_REPO_DIR` environment variable to the directory where the page server is located. For example:

```
export NEON_REPO_DIR=/home/gpadmin/pageserver
```

2. On the machine where the page server is running, run the following command to create a tenant:

```
./target/release/neon_local tenant create
```

The result will be similar to the following:

```
tenant 176349c483c0578faca41101fa70e19f successfully created on the
pageserver

Created an initial timeline '30cf96abf49fbb6f6c9712fc71c83d40' at Lsn 0/
4AABC88 for tenant: 176349c483c0578faca41101fa70e19f
```

In the returned result, "`176349c483c0578faca41101fa70e19f`" is the newly created tenant ID, which is unique within a UnionStore cluster. "`30cf96abf49fbb6f6c9712fc71c83d40`" is the timeline ID, which is similar to a Git branch. For this purpose, using a single timeline is sufficient.

Step 2: Configure SynxDB parameters

After creating the UnionStore tenant, you need to add the tenant information to the `postgresql.conf` configuration file for SynxDB. You must add these settings to the configuration file on each node:

Parameter name	Description	Default value	Required	Example	Note
shared_preload_libraries	The name of the plugin's dynamic library to be loaded when the SynxDB database starts.	Empty	Yes	shared_preload_libraries = unionstore	
unionstore.tenant_id	The tenant ID for UnionStore.	Empty	Yes	unionstore.tenant_id='176349c483c0578faca41101fa70e19f'	'a70e19f'
unionstore.timeline_id	The timeline ID for the UnionStore tenant.	Empty	Yes	unionstore.timeline_id='30cf96abf49fbb6f6c9712fc71c83d40'	'c71c83d40'
unionstore.safekeepers	The IP addresses and ports of the safekeeper components, which default to a three-replica setup. Used to establish connections with the log service and persist logs. This must match the values you provided during the UnionStore installation.	Empty	Yes	unionstore.safekeepers='127.0.0.1:5454,127.0.0.1:5455,127.0.0.1:5457'	1:5455,127.0.0.1:5457'
unionstore.pageserver_connstr	The IP address and port of the UnionStore PageServer component. Used to establish a connection with the PageServer to read pages and other data. This must match the values you provided during the UnionStore installation.	Empty	Yes	unionstore.pageserver_connstring='pgsql://no_user:@127.0.0.1:64000'	'no_user:@127.0.0.1:64000'

Below is a sample configuration. You need to replace the values with your actual parameters:

```
shared_preload_libraries=unionstore
unionstore.tenant_id='176349c483c0578faca41101fa70e19f'
unionstore.timeline_id='30cf96abf49fbb6f6c9712fc71c83d40'
unionstore.safekeepers='127.0.0.1:5454,127.0.0.1:5455,127.0.0.1:5457'
unionstore.pageserver_connstring='pgsql://no_user:@127.0.0.1:64000'
```

Step 3: Install the SynxDB extension

SynxDB uses an extension to interact with UnionStore for operations like writing logs and reading data. After completing the configuration, you need to install the extension in the database where you plan to use UnionStore:

```
CREATE EXTENSION unionstore;
```

After the extension is installed, SynxDB creates a new access method. You can view it with the following SQL query:

```
unionstore=# SELECT * FROM pg_am;
```

oid	amname	amhandler	amttype
2	heap	heap_tableam_handler	t
403	btree	bthandler	i
405	hash	hashhandler	i
783	gist	gisthandler	i

(continues on next page)

(continued from previous page)

2742	gin	ginhandler	i
4000	spgist	spghandler	i
3580	brin	brinhandler	i
7024	ao_row	ao_row_tableam_handler	t
7166	ao_column	ao_column_tableam_handler	t
7013	bitmap	bmhandler	i
16403	union_store	heap_tableam_handler	t
(11 rows)			

In the results above, `union_store` is the new access method created for using UnionStore.

Step 4: Create and use tables and indexes in UnionStore

After installing the SynxDB extension and creating the `union_store` access method, you can start creating UnionStore tables and indexes.

The syntax for creating a UnionStore table is as follows:

```
CREATE TABLE <table_name> (xxx) USING union_store;
```

The syntax for creating a UnionStore B-tree index (similar for other index types) is as follows:

```
CREATE INDEX ON <table_name> USING BTREE (<column_name>);
```

Example:

```
--- Creates a table.
CREATE TABLE unionstore_table (c1 INT, c2 VARCHAR, c3 TIMESTAMP) USING union_
store;

unionstore=# \dt+ unionstore_table
                                         List of relations
 Schema |           Name            |   Type   |  Owner   | Storage | Persistence | Access
method |      Size      | Description
-----+-----+-----+-----+-----+-----+-----+
 public | unionstore_table | table | gpadmin | permanent | union_
store | 128 kB |
```

(continues on next page)

(continued from previous page)

```
--- Creates an index.  
CREATE INDEX ON unionstore_table USING btree (c1);  
  
--- Inserts data.  
INSERT INTO unionstore_table SELECT t,t,now() FROM generate_series(1,100) t;  
  
--- Queries data.  
SELECT * FROM unionstore_table WHERE c1 = 55;  
  
c1 | c2 | c3  
----+----+-----  
55 | 55 | 2023-07-04 16:47:49.373224  
(1 row)
```

Limitations

- UnionStore does not support storing AO or AOCS tables.
- UnionStore does not support temp and unlogged tables or their indexes.

The core idea of UnionStore is “Log is database.” However, since temp and unlogged tables do not generate logs, their data cannot be persisted to UnionStore, and thus they are not supported.

- UnionStore does not support tablespaces.

The underlying implementation of UnionStore currently only supports the default tablespace. Therefore, you cannot create new tablespaces or modify the tablespace for a database, table, or index when using UnionStore.

Cross-Cluster Federated Query

SynxDB has supported cross-cluster federated queries, allowing you to query data across multiple PostgreSQL database clusters. This feature enables cross-cluster data pushdown, allowing queries to be executed locally in multiple clusters, and then the results are aggregated. The pushdown capability effectively reduces data transfer and improves query performance and processing efficiency.

This feature is implemented through the MPP foreign data wrapper `postgres_fdw`, which allows remote PostgreSQL databases to be treated as shards in SynxDB, enabling data sharing and federated queries across homogenous systems. Each remote PostgreSQL table can be viewed as an external table, and multiple remote tables can be added to SynxDB for operations. In this way, you can access and query data from other clusters within one cluster without complex data import or migration operations. This provides great convenience for scenarios that involve handling similar data across multiple clusters.

User scenarios

- **Data integration and analysis:** In some companies, data is often spread across multiple database clusters of the same type. This feature helps companies integrate data across clusters for unified analysis and processing. The ability to perform cross-cluster federated queries allows businesses to quickly merge data from different units and generate unified reports and analysis results. This integration reduces the complexity of manual data importing and processing, while improving the real-time aspect of data analysis.
- **Simplified data management:** The cross-cluster query feature reduces the need for duplicating data across databases, enabling data in each cluster to be directly involved in queries, avoiding the maintenance of redundant data. By treating remote databases as external shards, you can easily manage and access these shards without worrying about data synchronization. At the same time, this also reduces the potential risks and maintenance costs associated with cross-cluster data synchronization.
- **Efficient distributed querying:** By pushing down queries to reduce network traffic and bottlenecks caused by centralized processing, this feature is especially suited for large-scale distributed data analyses. When handling big data, distributed querying can significantly reduce the computing and communication load, enhancing the system's overall processing capacity. Especially when performing complex analytical calculations across multiple clusters,

pushdown queries effectively utilize the computing resources of each cluster, improving processing efficiency.

Usages

This section explains how to perform cross-cluster federated queries using the `postgres_fdw` extension.

Prerequisites

- Ensure that all clusters involved in the federated query can communicate with each other. This includes network connectivity and secure access configurations to make sure data can be transmitted smoothly between the clusters without restrictions.

Step 1. Create a foreign data wrapper

1. Load the `postgres_fdw` foreign data wrapper in SynxDB.

```
CREATE EXTENSION postgres_fdw;
```

2. Create a remote server object, for example, `testserver`.

```
CREATE SERVER testserver FOREIGN DATA WRAPPER postgres_fdw (host '<remote_server_IP>', dbname '<remote_database_name>', port '<remote_port_number>');
');
```

You can create multiple server objects as needed, for example, creating `mpps1`, `mpps2`, and `mpps3` for the remote databases `fdw1`, `fdw2`, and `fdw3`:

```
CREATE SERVER mpps1 FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'xxx.xxx.xxx.xxx', dbname 'fdw1', port '7000');
CREATE SERVER mpps2 FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'xxx.xxx.xxx.xxx', dbname 'fdw2', port '7000');
CREATE SERVER mpps3 FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'xxx.xxx.xxx.xxx', dbname 'fdw3', port '7000');
```

These commands allow you to define different server objects for different remote databases,

making it easier to create external tables and perform queries later.

Step 2. Create a user mapping

Create a user mapping for each server. You need to fill in the actual database username and password in the OPTIONS section.

```
CREATE USER MAPPING FOR CURRENT_USER SERVER mpps1 OPTIONS (user 'postgres',
password 'xxx');
CREATE USER MAPPING FOR CURRENT_USER SERVER mpps2 OPTIONS (user 'postgres',
password 'xxx');
CREATE USER MAPPING FOR CURRENT_USER SERVER mpps3 OPTIONS (user 'postgres',
password 'xxx');
```

User mapping links the current database user with the remote database user, allowing access to remote data when running queries. Depending on your needs, you can also specify different access permissions for each server.

Step 3. Create a foreign table and add shards

Create a foreign table `fs1`, and add the table `t1` from the remote server `mpps1` as a foreign shard:

```
CREATE FOREIGN TABLE fs1 (
    a int,
    b text
) SERVER mpps1 OPTIONS (schema_name 'public', table_name 't1', mpp_execute
'all segments');

ADD FOREIGN SEGMENT FROM SERVER mpps1 INTO fs1;
```

With these commands, you can add one or more tables from remote databases as foreign shards to an external table. The data from these shards automatically participates in queries. This allows you to access and manage remote shard data just like you would with local tables.

Step 4. Perform Cross-Cluster Federated Query

Single table query

For single table queries, you can directly query the foreign table `fs1`. The database system distributes the query to the remote cluster for execution.

1. Check the query plan.

```
EXPLAIN (COSTS OFF) SELECT * FROM fs1;
```

The execution plan shows Foreign Scan on `fs1`, which indicates that the query operation has been distributed to the remote server for execution.

```
QUERY PLAN
---
Gather Motion 1:1 (slice1; segments: 1)
  -> Foreign Scan on fs1
Optimizer: Postgres query optimizer
(3 rows)
```

2. Run the query.

```
SELECT * FROM fs1;
```

The returned data is from the remote table `t1`.

```
a | b
---+-----
1 | fdw1
(1 row)
```

From the output, you can see that the data is only from the `mpps1` server.

3. Add other remote servers as shards.

```
ADD FOREIGN SEGMENT FROM SERVER mpps2 INTO fs1;
ADD FOREIGN SEGMENT FROM SERVER mpps3 INTO fs1;
```

4. Check the query plan again.

```
EXPLAIN (COSTS OFF) SELECT * FROM fs1;
```

```
QUERY PLAN
-----
Gather Motion 3:1 (slice1; segments: 3)
  -> Foreign Scan on fs1
Optimizer: Postgres query optimizer
(3 rows)
```

- Run the query again.

```
SELECT * FROM fs1;
```

Returned result:

a	b
1	fdw2
1	fdw1
1	fdw3
(3 rows)	

The result shows that the newly added shards mpps2 and mpps3 have successfully participated in the query.

Multi-table join

For multi-table join operations, the optimizer automatically generates an appropriate execution plan based on the number of segments in the tables.

- Create the foreign table fs2.

```
CREATE FOREIGN TABLE fs2 (
    a int,
    b text
)
SERVER mpps1
OPTIONS (schema_name 'public', table_name 't2', mpp_execute 'all segments
');
```

2. Add segments.

```
ADD FOREIGN SEGMENT FROM SERVER mpps1 INTO fs2;
ADD FOREIGN SEGMENT FROM SERVER mpps2 INTO fs2;
ADD FOREIGN SEGMENT FROM SERVER mpps3 INTO fs2;
```

3. Execute the join query between the two tables.

```
EXPLAIN (COSTS OFF) SELECT * FROM fs1, fs2 WHERE fs1.a = fs2.a;
```

The query plan shows that the optimizer has chosen Hash Join, and the data from both tables is redistributed based on key values, allowing the join operation to be performed across different nodes.

```
QUERY PLAN
---
Gather Motion 3:1 (slice1; segments: 3)
    -> Hash Join
        Hash Cond: (fs1.a = fs2.a)
        -> Redistribute Motion 3:3 (slice2; segments: 3)
            Hash Key: fs1.a
            -> Foreign Scan on fs1
        -> Hash
            -> Redistribute Motion 3:3 (slice3; segments: 3)
            Hash Key: fs2.a
            -> Foreign Scan on fs2
Optimizer: Postgres query optimizer
(11 rows)
```

4. Run the actual join query.

```
SELECT * FROM fs1, fs2 WHERE fs1.a = fs2.a;
```

a	b	a	b
1	fdw1	1	fdw2
1	fdw1	1	fdw1
1	fdw1	1	fdw3
1	fdw2	1	fdw2
1	fdw2	1	fdw1

(continues on next page)

(continued from previous page)

```

1 | fdw2 | 1 | fdw3
1 | fdw3 | 1 | fdw2
1 | fdw3 | 1 | fdw1
1 | fdw3 | 1 | fdw3
(9 rows)

```

The returned results show that all matching rows between `fs1` and `fs2` are successfully joined.

Join pushdown with `gp_foreign_server` condition

To further optimize query performance, you can add the `gp_foreign_server` condition in the join to enable join pushdown.

1. Check the query plan.

```
EXPLAIN (COSTS OFF) SELECT * FROM fs1, fs2 WHERE fs1.a = fs2.a AND fs1(gp_
foreign_server = fs2_gp_foreign_server;
```

The query plan shows that the Foreign Scan operation has pushed the join down to the remote server, reducing the burden on local computing.

```

QUERY PLAN
---
Gather Motion 3:1 (slice1; segments: 3)
    -> Foreign Scan
        Relations: (fs1) INNER JOIN (fs2)
Optimizer: Postgres query optimizer
(4 rows)

```

2. Run the query.

```
SELECT * FROM fs1, fs2 WHERE fs1.a = fs2.a AND fs1_gp_foreign_server =
fs2_gp_foreign_server;
```

The returned result:

```
a | b | a | b
---+-----+---+-----
1 | fdw3 | 1 | fdw3
1 | fdw1 | 1 | fdw1
1 | fdw2 | 1 | fdw2
(3 rows)
```

By adding the `gp_foreign_server` condition, the query is pushed down to the remote server, making fewer matching rows and greatly improving query efficiency.

Aggregate pushdown

Aggregate queries can be pushed down to the remote server to greatly reduce data transfer, improving query efficiency.

1. Run an aggregate query.

```
SELECT count(*) FROM fs1, fs2 WHERE fs1.a = fs2.a;
```

The returned result:

```
count
---
9
(1 row)
```

2. Check the query plan.

```
EXPLAIN (COSTS OFF) SELECT count(*) FROM fs1, fs2 WHERE fs1.a = fs2.a AND
fs1(gp_foreign_server = fs2(gp_foreign_server);
```

The query plan shows that the aggregate operation has been pushed down to the remote server, reducing the aggregation load on the local node.

```
QUERY PLAN
---
Finalize Aggregate
-> Gather Motion 3:1 (slice1; segments: 3)
```

(continues on next page)

(continued from previous page)

```
-> Foreign Scan
    Relations: Aggregate on ((fs1) INNER JOIN (fs2))
Optimizer: Postgres query optimizer

(5 rows)
```

By pushing the aggregation operation to the remote server, the database system can leverage the remote server's computing power to perform part of the aggregation work, thus improving overall query performance.

Use Tags to Manage Database Objects

What is a tag?

A tag is a database-level object that helps users label, classify, and manage other objects in the database. Using tags, you can add custom labels to database objects such as databases, users, and tables, making it easier to manage and monitor different objects in the database.

You can use tags in the database mainly used for the following purposes:

- Compliance monitoring for sensitive data: Tags can mark sensitive data to help administrators track and protect data, thus ensuring compliance.
- Data governance and resource usage management: Tags can classify and label different types of database objects, which helps discover data, protect it, and monitor resource usage.

Tags are stored as key-value pairs, with a tag name and its value making up the pair. You can assign the same tag to multiple database objects with different values for flexible management.

SynxDB supports and enables tags by default, with no extra setup needed.

Features of tags

- You can assign tags to any object in the database, such as databases, users, or tables.
- You can assign tags when creating a database object with the `CREATE <object>` statement, or add or change tags later using the `ALTER <object>` statement.
- You can assign the same tag to different types of objects (like databases and tables) at the same time, and the tag value can be the same or different each time.

Usage scenarios

The tag is suitable for these situations:

- Object classification: Tags can be used to classify database objects. For example, you can use tags to tell the difference between objects in a development environment and a production environment. This makes it easier to quickly find and manage database objects in different environments.

- Permission management: By giving objects-specific tags, you can mark objects with special permissions or sensitive data. This helps administrators be more accurate when checking data, giving permissions, and managing compliance.
- Version control and status tracking: Tags can also be used to mark the version or status of database objects. This helps teams track the history of changes or see the database setup at a specific point in time. For example, a tag can show which project stage or version an object belongs to.
- Resource usage monitoring: Giving tags to resources (such as warehouses or tables) can help you accurately monitor resource usage. For example, tags can be used to group data warehouses by cost centers or business units, making it easier to analyze how resources are used and their efficiency.

Use tags

Query existing tag information

You can use the following commands to check the structure and content of the `pg_tag` and `pg_tag_description` tables:

```
\d+ pg_tag; -- Views the detailed structure of the pg_tag table.
\d+ pg_tag_description; -- Views the detailed structure of the pg_tag_
description table.
SELECT * FROM pg_tag; -- Gets all tag information.
SELECT * FROM pg_tag_description; -- Gets all tag description information.
```

You can use the following commands to query the created tags and descriptions:

```
SELECT tagname, tagowner, allowed_values FROM pg_tag ORDER BY 1; -- Gets all
tag names, owners, and allowed values.
SELECT count(*) FROM pg_tag_description; -- Gets the total count of tag
descriptions.
```

You can also use the following views to quickly query tag information in the database system:

- `database_tag_descriptions`: Queries tag information for database objects.
- `user_tag_descriptions`: Queries tag information for user (role) objects.

- `tablespace_tag_descriptions`: Queries tag information for tablespaces.
- `schema_tag_descriptions`: Queries tag information for schemas.
- `relation_tag_descriptions`: Queries tag information for relation objects such as tables, views, and indexes.

Example queries:

```
SELECT * FROM database_tag_descriptions;
SELECT * FROM user_tag_descriptions;
SELECT * FROM tablespace_tag_descriptions;
SELECT * FROM schema_tag_descriptions;
SELECT * FROM relation_tag_descriptions;
```

Create tags

You can create a tag using the `CREATE TAG` statement and specify allowed values. The syntax is:

```
CREATE TAG [ IF NOT EXISTS ] <tag_name>
CREATE TAG [ IF NOT EXISTS ] <tag_name> [ ALLOWED_VALUES '<val_1>' [ , '<val_2>' [ , ... ] ] ]
```

In the above statements, `ALLOWED_VALUES` can define up to 300 allowed values. By default, any string (including empty strings) is allowed, and each string can be up to 256 characters long.

Examples:

```
CREATE TAG tag_test_a; -- Creates a tag named tag_test_a.
CREATE TAG IF NOT EXISTS tag_test_b; -- Creates tag_test_b if it does not exist.
CREATE TAG tag_test_c ALLOWED_VALUES '123'; -- Creates a tag_test_c tag with the allowed value '123'.
CREATE TAG tag_test_d ALLOWED_VALUES '123', '456', ''; -- Creates a tag_test_d tag with multiple allowed values.
CREATE TAG IF NOT EXISTS tag_test_e ALLOWED_VALUES '123', 'val1'; -- Creates a tag_test_e tag if it does not exist.
```

Delete tags

Use the `DROP TAG` statement to delete a tag. The syntax is:

```
DROP TAG [ IF EXISTS ] <tag_name>;
```

Before deleting a tag, the system checks whether the tag is referenced by any database object. If it is, an error is thrown. The user who executes the deletion must be a superuser or the owner of the tag.

Examples:

```
DROP TAG tag_test_a; -- Deletes the tag_test_a tag.  
DROP TAG IF EXISTS tag_test_b; -- Deletes the tag_test_b tag if it exists.
```

Modify tags

Use the `ALTER TAG` statement to change the tag name or allowed values. The syntax is:

```
ALTER TAG [ IF EXISTS ] <tag_name> RENAME TO <new_name>;  
ALTER TAG [ IF EXISTS ] <tag_name> { ADD | DROP } ALLOWED_VALUES '<val_1>' [,  
'<val_2>', ...];  
ALTER TAG <tag_name> UNSET ALLOWED_VALUES;
```

- `RENAME TO`: Renames the existing tag to a new name.
- `ADD` or `DROP`: Adds or removes allowed values for the tag.
- `UNSET ALLOWED_VALUES`: Resets the allowed values for the tag.

Rename tags

```
ALTER TAG tag_test_a RENAME TO tag_test_a_new; -- Renames tag_test_a to tag_test_a_new.  
ALTER TAG IF EXISTS tag_test_b RENAME TO tag_test_b_new; -- If tag_test_b exists, renames it to tag_test_b_new.  
ALTER TAG tag_test_c_new RENAME TO tag_test_c; -- Renames tag_test_c_new back to tag_test_c.
```

(continues on next page)

(continued from previous page)

```
ALTER TAG tag_test_d_new RENAME TO tag_test_d; -- Renames tag_test_d_new to tag_test_d.
```

Modify allowed values

Example of removing allowed values:

```
ALTER TAG tag_test_a UNSET ALLOWED_VALUES; -- Removes all allowed values from tag_test_a.  
ALTER TAG tag_test_b UNSET ALLOWED_VALUES; -- Removes all allowed values from tag_test_b.  
ALTER TAG tag_test_c UNSET ALLOWED_VALUES; -- Removes all allowed values from tag_test_c.
```

Example of adding allowed values:

```
ALTER TAG tag_test_a ADD ALLOWED_VALUES 'val1'; -- Adds 'val1' as an allowed value to tag_test_a.  
ALTER TAG tag_test_b ADD ALLOWED_VALUES 'val2', 'val3'; -- Adds multiple allowed values to tag_test_b.  
ALTER TAG IF EXISTS tag_test_c ADD ALLOWED_VALUES ' '; -- Adds an empty string as an allowed value to tag_test_c.
```

Assign tags to objects

You can use the TAG keyword to assign tags when creating or modifying database objects.

Assign tags to a database

The syntax is:

```
CREATE [ OR REPLACE ] DATABASE <database_name> ... [ TAG ( <tag_name> = '<tag_value>' [, ...] ) ];  
ALTER DATABASE [ IF EXISTS ] <database_name> TAG ( <tag_name> = '<tag_value>' [, ...] );
```

(continues on next page)

(continued from previous page)

```
ALTER DATABASE [ IF EXISTS ] <database_name> UNSET TAG ( <tag_name> [, <tag_name>, ...] );
```

You can assign or remove tags for a database object using the TAG keyword in the CREATE DATABASE and ALTER DATABASE statements.

Examples:

```
CREATE DATABASE sales_db TAG ( environment = 'production' );

ALTER DATABASE sales_db TAG ( environment = 'staging' ); -- Updates the environment tag of the sales_db database to staging.
ALTER DATABASE sales_db UNSET TAG ( environment ); -- Removes the environment tag from the sales_db database.
```

Assign tags to tables

The syntax is:

```
CREATE [ OR REPLACE ] TABLE <table_name> ... [ TAG ( <tag_name> = '<tag_value>' [, ...] ) ];
ALTER TABLE [ IF EXISTS ] <table_name> TAG ( <tag_name> = '<tag_value>' [, ...] );
ALTER TABLE [ IF EXISTS ] <table_name> UNSET TAG ( <tag_name> [, <tag_name>, ...] );
```

- You can use TAG in the CREATE TABLE and ALTER TABLE statements to assign or remove tags from table objects.
- A database object can have up to 50 tags, and each tag must be unique.

Example:

```
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    order_date DATE
) TAG ( priority = 'high' );
```

(continues on next page)

(continued from previous page)

```
ALTER TABLE orders TAG ( priority = 'urgent' );
ALTER TABLE orders UNSET TAG ( priority );
```

Assign tags to users

The syntax is:

```
CREATE [ OR REPLACE ] USER <user_name> ... [ TAG ( <tag_name> = '<tag_value>' [, ...] ) ];
ALTER USER [ IF EXISTS ] <user_name> TAG ( <tag_name> = '<tag_value>' [, ...] );
ALTER USER [ IF EXISTS ] <user_name> UNSET TAG ( <tag_name> [, <tag_name>, ...] );
```

You can use TAG in the CREATE USER and ALTER USER statements to assign or remove tags from user objects.

Example:

```
CREATE USER john_doe TAG ( role = 'admin' );

ALTER USER john_doe TAG ( role = 'super_admin' );
ALTER USER john_doe UNSET TAG ( role );
```

Tag comments

Use COMMENT ON TAG to add a comment to a tag. The syntax is:

```
COMMENT ON TAG <tag_name> IS '<comment>';
```

Example:

```
COMMENT ON TAG priority IS 'This tag indicates the urgency level of orders.';
```

System tables related to tags

SynxDB uses two metadata tables to store tag information and the relationship between database objects and tags: `pg_tag` and `pg_tag_description`.

- `pg_tag`: Stores information about all tags, including the tag's OID, name, owner, and allowed values.
- `pg_tag_description`: Records the relationship between database objects and tags, including the object's database ID, class ID, object ID, sub-object number, tag OID, and tag value.

pg_tag table

The `pg_tag` table is used to store information about all tags, including the tag identifier, name, owner, and allowed values. The table structure is as follows:

Column name	Data type	Description
OID	OID	Unique identifier for the tag.
TAGNAME	NAMEDATA	Name of the tag.
TAGOWNER	OID	The OID of the tag owner (user).
ALLOWED_VALUES	ARRAY [TEXT]	Allowed values for this tag (if any).

Indexes:

- `pg_tag_name_index`: Primary key index, a `btree` index based on the tag name (`tagname`), ensuring that tag names are unique.
- `pg_tag_oid_index`: Unique index, based on the tag's OID, ensuring that each tag has a unique identifier.

With the `pg_tag` table, the database system can efficiently manage and store all defined tag information, and each tag can have a set of allowed values, making it flexible to use.

pg_tag_description table

The pg_tag_description table records the relationships between database objects and tags. Each record represents a tag and its value associated with a specific database object. The structure of the table is as follows:

Column name	Data type	Description
oid	OID	The unique identifier for the table.
tddatabaseid	OID	The database ID associated with the tag (set to 0 for globally shared objects).
tdclassid	OID	The class ID of the database object.
tdobjid	OID	The OID of the database object.
tagid	OID	The OID of the tag, indicating the tag associated with the object.
tagvalue	TEXT	The specific value of the tag for the object.

Indexes:

- pg_tag_description_d_c_o_t_index: Primary key index, a `btree` index based on database ID, class ID, object ID, and tag ID, used to quickly locate the tag of a specific database object.
- pg_tag_description_oid_index: Unique constraint, a `btree` index based on `oid`, ensuring the uniqueness of each record.
- pg_tag_description_tagidvalue_index: Regular index, a `btree` index based on tag `OID` (`tagid`) and tag value (`tagvalue`), used to quickly find all objects associated with a specific tag and its value.

The pg_tag_description table stores the tag information for each database object along with its values, supporting many-to-many relationships between tags and objects.

Handle global and non-global objects

For globally shared objects (such as users and repositories), the DBID field value is 0. This means that these objects are not part of a specific database and have global properties. For non-global shared objects (such as a specific database object), the DBID field stores the OID of the database where the object belongs.

Sub-object number (OBJSUBID): Currently, the database system does not support adding labels to sub-objects (like columns), so OBJSUBID is always 0.

Common errors and tips

- **Duplicate labels:** If you assign duplicate labels to the same object, the database system returns an error message.
- **Permission denied:** If a non-superuser or non-label owner tries to delete or modify a label, the database system returns a permission error.
- **Exceeded label limit:** An object can have a maximum of 50 labels. Adding more than 50 will cause an error.

Using this feature, database administrators can organize and manage database objects more effectively, making the system more flexible and easier to operate.

4.2 SQL Queries

Join Queries

In SynxDB, the JOIN clause combines rows from two or more tables based on related column values. The JOIN clause is part of the FROM clause in a SELECT statement.

The syntax for the JOIN clause is as follows:

```
table1_name join_type table2_name [join_condition]
```

Where:

- `table1_name, table2_name`: Names of the tables to be joined.
- `join_type`: The type of join, which can be one of the following:
 - [INNER] JOIN
 - LEFT [OUTER] JOIN
 - RIGHT [OUTER] JOIN
 - FULL [OUTER] JOIN
 - NATURAL JOIN
- `join_condition`: An optional condition that specifies how to match rows from the two tables. It can take one of the following forms:
 - ON <join_condition>
 - USING (<join_column> [, ...])
 - LATERAL

Note

For FULL JOIN queries, the ORCA optimizer automatically chooses between Merge Join and Hash Join based on cost estimates, so users do not need to manually specify the join method.

Join types

SynxDB supports the following types of joins:

INNER JOIN returns the intersection of rows from both tables that satisfy the join condition.

In other words, it returns only the rows with matching values in both tables. If INNER is omitted before JOIN, it defaults to INNER JOIN.

```
SELECT *
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

LEFT OUTER JOIN

LEFT OUTER JOIN (or simply LEFT JOIN) returns all rows from the left table and the matching rows from the right table. If there is no match in the right table, the result includes NULL values for columns from the right table.

```
SELECT *
FROM table1
LEFT OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

RIGHT OUTER JOIN

RIGHT OUTER JOIN (or simply RIGHT JOIN) is the opposite of LEFT OUTER JOIN. It returns all rows from the right table and the matching rows from the left table. If there is no match in the left table, the result includes NULL values for columns from the left table.

```
SELECT *
FROM table1
RIGHT OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

Note

Starting from v4.0.0, Hash Right Join queries can also trigger dynamic partition elimination (DPE) when partition pruning conditions are met, reducing partition scans and improving performance.

FULL OUTER JOIN

FULL OUTER JOIN (or simply FULL JOIN) returns all rows from both the left and right tables. For unmatched rows in the left table, the right table columns are filled with NULL. For unmatched rows in the right table, the left table columns are filled with NULL.

```
SELECT *
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

CROSS JOIN

CROSS JOIN returns the Cartesian product of two tables. It combines every row from the left table with every row from the right table. If there is no WHERE clause to filter the results, the output will contain the number of rows in the left table multiplied by the number of rows in the right table.

```
SELECT *
FROM table1
CROSS JOIN table2;
```

NATURAL JOIN

The NATURAL clause is a further shorthand for the USING clause. It is used when all columns with the same names in both tables should be used for the join. If there are no columns with the same names, NATURAL JOIN behaves like a CROSS JOIN. Use NATURAL JOIN with caution, as it relies on column names and may produce unexpected results.

```
SELECT *
FROM table1
NATURAL JOIN table2;
```

Join conditions

Join conditions define how to match rows between two tables. There are three types of join conditions:

ON clause

The ON clause specifies a Boolean expression that determines which rows from the two tables should be considered a match. It works similarly to a WHERE clause but is applied specifically to the JOIN.

```
SELECT *
FROM table1
JOIN table2
ON table1.column_name = table2.column_name;
```

USING clause

The USING clause is a shorthand for the ON clause when both tables have one or more columns with the same name. It specifies the common column(s) to be used for the join. The result includes only one instance of each matched column, instead of both.

```
SELECT *
FROM table1
JOIN table2
USING (column_name);
```

LATERAL

The `LATERAL` keyword can be placed before a subquery in the `FROM` clause. It allows the subquery to reference columns from preceding items in the `FROM` list. Without `LATERAL`, SynxDB evaluates subqueries independently and does not allow references to other `FROM` items.

Example

Assume there are two tables: `customers` and `orders`.

`customers` table:

	customer_id	customer_name
1		John Doe
2		Jane Smith
3		Bob Johnson

`orders` table:

	order_id	customer_id	order_date
1		1	2023-01-15
2		2	2023-02-20
3		1	2023-03-10
4		4	2024-05-01

Here are some examples using different `JOIN` types:

INNER JOIN example

This query returns all customers and their orders, including only rows where the customer ID matches in both tables.

```
SELECT customers.customer_name, orders.order_id
FROM customers
INNER JOIN orders
ON customers.customer_id = orders.customer_id;
```

Result:

customer_name		order_id
John Doe		1
Jane Smith		2
John Doe		3

LEFT OUTER JOIN example

This query returns all customers along with their orders. Even if a customer has no orders, the customer information will still be included.

```
SELECT customers.customer_name, orders.order_id
FROM customers
LEFT OUTER JOIN orders
ON customers.customer_id = orders.customer_id;
```

Result:

customer_name		order_id
John Doe		1
Jane Smith		2
Bob Johnson		NULL
John Doe		3

RIGHT OUTER JOIN example

This query returns all orders along with the customers who placed them. Even if an order has no associated customer, the order will still be included. In this example, the order with `order_id` 4 has no matching customer.

```
SELECT customers.customer_name, orders.order_id
FROM customers
RIGHT OUTER JOIN orders
ON customers.customer_id = orders.customer_id;
```

Result:

customer_name		order_id
John Doe		1
Jane Smith		2
John Doe		3
NULL		4

FULL OUTER JOIN example

This query returns all customers and all orders. If a customer has no orders or an order has no customer, the result will still include that customer or order.

```
SELECT customers.customer_name, orders.order_id
FROM customers
FULL OUTER JOIN orders
ON customers.customer_id = orders.customer_id;
```

Result:

customer_name		order_id
John Doe		1
Jane Smith		2
Bob Johnson		NULL
John Doe		3
NULL		4

CROSS JOIN example

This query returns the Cartesian product of the customers and orders tables.

```
SELECT customers.customer_name, orders.order_id
FROM customers
CROSS JOIN orders;
```

Result (partial output; a total of $3 \times 4 = 12$ rows):

```
customer_name | order_id
-----+-----
John Doe      | 1
John Doe      | 2
John Doe      | 3
John Doe      | 4
Jane Smith    | 1
Jane Smith    | 2
...
```

4.3 Advanced Analytics

Use pgvector for Vector Similarity Search

pgvector is an open-source plugin for vector similarity search. It supports both exact and approximate nearest neighbor searches, as well as L2 distance, inner product, and cosine distance. For more details, see [pgvector/pgvector: Open-source vector similarity search for Postgres](#). SynxDB allows you to use pgvector for data storage, querying, indexing, hybrid searches, and more through SQL statements.

This document explains how to use pgvector in SynxDB.

Quick start

Enable the extension (do this once in each database where you want to use it):

```
CREATE EXTENSION vector;
```

Create a vector column with 3 dimensions:

```
CREATE TABLE items (id bigserial PRIMARY KEY, embedding vector(3));
```

Insert vector data:

```
INSERT INTO items (embedding) VALUES ('[1,2,3]'), ('[4,5,6]');
```

Get the nearest neighbors by L2 distance:

```
SELECT * FROM items ORDER BY embedding <-> '[3,1,2]' LIMIT 5;
```

Note: Use `<#>` for inner product and `<=>` for cosine distance. Use `<+>` for L1 distance.

Store data

Create a table with a vector column:

```
CREATE TABLE items (id bigserial PRIMARY KEY, embedding vector(3));
```

Or add a vector column to an existing table:

```
ALTER TABLE items ADD COLUMN embedding vector(3);
```

Insert vectors:

```
INSERT INTO items (embedding) VALUES ('[1,2,3]', '[4,5,6]');
```

Insert and update vectors:

```
INSERT INTO items (id, embedding) VALUES (1, '[1,2,3]'), (2, '[4,5,6]')  
ON CONFLICT (id) DO UPDATE SET embedding = EXCLUDED.embedding;
```

Update vectors:

```
UPDATE items SET embedding = '[1,2,3]' WHERE id = 1;
```

Delete vectors:

```
DELETE FROM items WHERE id = 1;
```

Query data

Get the nearest neighbors to a vector:

```
SELECT * FROM items ORDER BY embedding <-> '[3,1,2]' LIMIT 5;
```

The supported distance functions are:

- <->: L2 distance
- <#>: negative inner product
- <=>: cosine distance

- <+>: L1 distance

Get the nearest neighbors of a row:

```
SELECT * FROM items WHERE id != 1 ORDER BY embedding <-> (SELECT embedding
FROM items WHERE id = 1) LIMIT 5;
```

Get rows within a specific distance range:

```
SELECT * FROM items WHERE embedding <-> '[3,1,2]' < 5;
```

Note: Using ORDER BY and LIMIT together can take advantage of indexing.

Get the distance:

```
SELECT embedding <-> '[3,1,2]' AS distance FROM items;
```

For inner product, multiply by -1 (because <#> returns the negative inner product).

```
SELECT (embedding <#> '[3,1,2]') * -1 AS inner_product FROM items;
```

For cosine similarity, use 1 minus the cosine distance.

```
SELECT 1 - (embedding <=> '[3,1,2]') AS cosine_similarity FROM items;
```

Calculate the average of vectors:

```
SELECT AVG(embedding) FROM items;
```

Calculate the average of a group of vectors:

```
SELECT category_id, AVG(embedding) FROM items GROUP BY category_id;
```

Index data

By default, pgvector performs exact nearest neighbor searches, which provides a high recall rate.

If you need a higher recall rate, you can use approximate nearest neighbor search by adding an index, although this might reduce performance. Unlike adding a regular index, after adding an approximate index, **queries will return different results**.

pgvector supports the following index types:

- HNSW
- IVFFlat

HNSW index

About HNSW

HNSW (Hierarchical Navigable Small World) is an efficient algorithm for approximate nearest neighbor search, designed for processing large-scale and high-dimensional datasets.

The basic principles of HNSW are as follows:

- Multi-level graph structure: HNSW organizes data by building a multi-level graph. In this graph, each node represents a data point (or vector), and the edges between nodes reflect their relative proximity in space.
- Search optimization: This multi-level structure allows the search process to skip over many irrelevant data points quickly, narrowing down the neighboring area of the query vector. This greatly improves query efficiency.

HNSW is particularly useful in the following scenarios:

- High-dimensional data: HNSW index is very effective for high-dimensional datasets, because it is good at handling complex proximity relationships in high-dimensional space.
- Large datasets: HNSW index is suitable for large datasets because it balances query speed and recall rate better than many other types of indexes.

Creating an HNSW index takes longer and uses more memory, but it offers better query performance (speed-recall tradeoff). Unlike IVFFlat, HNSW does not require a training step, so you can create the index even when the table has no data.

Add an index for each distance function you plan to use.

Create an HNSW index

Each distance metric has its specific use cases. The choice of which method to use for creating an index depends on the type of search you want to optimize. For example, if your application focuses on getting vectors with similar direction but possibly different magnitudes, an index created with cosine distance might be more suitable. On the other hand, if you are concerned with the straight-line distance between vectors, you should choose an index based on L2 distance.

L2 distance:

```
CREATE INDEX ON items USING hnsw (embedding vector_l2_ops);
```

Inner product:

```
CREATE INDEX ON items USING hnsw (embedding vector_ip_ops);
```

Cosine distance:

```
CREATE INDEX ON items USING hnsw (embedding vector_cosine_ops);
```

The maximum dimension for indexable vectors is 2000.

Hamming distance:

```
CREATE INDEX ON items USING hnsw (embedding bit_hamming_ops);
```

Jaccard distance:

```
CREATE INDEX ON items USING hnsw (embedding bit_jaccard_ops);
```

Supported types are:

- `vector`: up to 2000 dimensions
- `halfvec`: up to 4000 dimensions
- `bit`: up to 64000 dimensions
- `sparsevec`: up to 1000 non-zero elements

HNSW index options

- m: The maximum number of connections per layer (16 by default).
- ef_construction: The size of the dynamic candidate list used to build the graph (64 by default).

```
CREATE INDEX ON items USING hnsw (embedding vector_12_ops) WITH (m = 16, ef_
construction = 64);
```

A larger ef_construction value provides higher recall but at the cost of index build time and insert speed.

HNSW index query options

Specify the size of the dynamic candidate list for searches (40 by default). A larger value improves recall but reduces speed.

```
SET hnsw.ef_search = 100;
```

Use SET LOCAL within a transaction to apply it to a single query:

```
BEGIN;
SET LOCAL hnsw.ef_search = 100;
SELECT ...
COMMIT;
```

HNSW index build time

Index build speed increases greatly when the internal graph structure of the HNSW index fits into maintenance_work_mem.

```
SET maintenance_work_mem = '8GB';
```

If the graph no longer fits, you will receive a notification:

```

NOTICE: hnsw graph no longer fits into maintenance_work_mem after 100000
tuples
DETAIL: Building will take significantly more time.
HINT: Increase maintenance_work_mem to speed up builds.

```

Note: Do not set `maintenance_work_mem` too high, because this exhausts the server's memory.

Like other index types, it is faster to create the index after loading the initial data.

You can also speed up index creation by increasing the number of parallel workers (2 by default).

```
SET max_parallel_maintenance_workers = 7; -- Including the leader thread
```

For more workers, you might also need to increase `max_parallel_workers` (8 by default).

View HNSW index progress

You can check the progress when creating an HNSW index:

```
SELECT phase, round(100.0 * blocks_done / nullif(blocks_total, 0), 1) AS "%"
FROM pg_stat_progress_create_index;
```

The HNSW index build process includes the following phases:

1. **initializing**: The starting phase of index creation. In this phase, the system prepares all necessary resources and configurations to begin building the index.
2. **loading tuples**: Data points (or vectors) are added to the multi-level graph, and the corresponding connections are established.

IVFFlat index

About IVFFlat

The IVFFlat index is a method for efficient vector search in large datasets, particularly useful for the Approximate Nearest Neighbor (ANN) search.

The basic principles of IVFFlat index are as follows:

- Partitioned search space: IVFFlat works by dividing the data into multiple “lists”. These lists are created by clustering the dataset (for example, using the K-means algorithm), with each list representing a cluster in the data space.
- Reduced search complexity: Instead of searching through the entire dataset, the search first identifies which lists (or clusters) the search vector is likely to belong to, then only searches within those lists, reducing computing time.

IVFFlat is particularly useful in the following scenarios:

- Large datasets: For datasets containing many vectors, a full search (checking every vector) can be very time-consuming. IVFFlat optimizes the search process through clustering and partitioning.
- Approximate search: IVFFlat is an approximate nearest neighbor search method, suitable for scenarios where quick response times are needed and some inaccuracy in search results is acceptable.

To achieve good recall with IVFFlat, follow these best practices:

- Create the index after some data has been inserted into the table.
- Choose an appropriate number of lists. For tables with up to 1 million rows, it is recommended to use the number of rows divided by 1000 as the number of lists. For tables with more than 1 million rows, use the square root of the number of rows as the number of lists.
- Specify an appropriate number of probes during queries (the higher the number of probes, the higher the recall, but the slower the query). It is recommended to start by trying the square root of the number of lists. Add an index for each distance function you plan to use.

Create an IVFFlat index

Each distance metric has its specific use cases. The choice of which method to use for creating an index depends on the type of search you want to optimize. For example, if your application focuses on finding vectors with similar direction but possibly different magnitudes, an index created with cosine distance might be more suitable. On the other hand, if you are concerned with the straight-line distance between vectors, you should choose an index based on L2 distance.

The `lists` parameter specifies the number of partitions (lists).

L2 distance:

```
CREATE INDEX ON items USING ivfflat (embedding vector_l2_ops) WITH (lists = 100);
```

Inner product:

```
CREATE INDEX ON items USING ivfflat (embedding vector_ip_ops) WITH (lists = 100);
```

Cosine distance:

```
CREATE INDEX ON items USING ivfflat (embedding vector_cosine_ops) WITH (lists = 100);
```

⚠️ Attention

SynxDB currently supports indexing vectors with up to 2000 dimensions.

Hamming distance:

```
CREATE INDEX ON items USING ivfflat (embedding bit_hamming_ops) WITH (lists = 100);
```

Supported types:

- `vector`: up to 2000 dimensions.
- `halfvec`: up to 4000 dimensions

- bit: up to 64,000 dimensions

Specify the number of Probes

⚠️ Attention

The number of probes means how many “lists” the system checks during an approximate nearest neighbor search. These lists are subsets of the dataset created through clustering algorithms like K-means. Increasing the number of probes means the system checks more lists to get the nearest neighbors, thus improving accuracy.

A higher number of probes increases search accuracy but also adds computing overhead, slowing down the search. Therefore, the number of probes is a parameter that should be balanced based on specific use cases.

Specify the number of probes (1 by default):

```
SET ivfflat.probes = 10;
```

If you choose a large number of probes, there might be some performance loss that impacts speed, but you will achieve higher recall. You can also set it to match the number of lists for an exact nearest neighbor search (in this case, the optimizer does use the index).

Use `SET LOCAL` within a transaction to set the number of probes for a single query:

```
BEGIN;  
SET LOCAL ivfflat.probes = 10;  
SELECT ...  
COMMIT;
```

View IVFFlat index progress

You can check the index progress during its creation:

```
SELECT phase, tuples_done, tuples_total FROM pg_stat_progress_create_index;
```

The progress phases include:

1. initializing: The starting phase of index creation. The system prepares all necessary resources and configurations during this phase.
2. performing k-means: The k-means algorithm is used to divide the vector dataset into multiple lists (or clusters).
3. sorting tuples: Sorting the data (tuples). This is done based on vector values or the lists they belong to, optimizing the index structure and improving search efficiency.
4. loading tuples: Data is actually loaded into the index structure, which means that tuple data is written to the index to ensure it meets indexing requirements.

⚠️ Attention

tuples_done and tuples_total are only populated during the loading tuples phase.

Use filters in indexes

When creating an index, you can use a WHERE clause to limit the scope of the index. This method allows vector searches to only consider rows that meet specific conditions, which improves search efficiency and accuracy.

```
SELECT * FROM items WHERE category_id = 123 ORDER BY embedding <-> '[3,1,2]'  
LIMIT 5;
```

Create an index on one or more WHERE columns for exact searches:

```
CREATE INDEX ON items (category_id);
```

Create a partial index on a vector column for approximate searches:

```
CREATE INDEX ON items USING ivfflat (embedding vector_l2_ops) WITH (lists = 100) WHERE (category_id = 123);
```

For approximate searches with different values in multiple WHERE columns, use partitioning:

```
CREATE TABLE items (embedding vector(3), category_id int) PARTITION BY LIST(category_id);
```

Half-precision vectors

Use the `halfvec` type to store half-precision vectors:

```
CREATE TABLE items (id bigserial PRIMARY KEY, embedding halfvec(3));
```

Half-precision index

Use half-precision vector index to reduce index size:

```
CREATE INDEX ON items USING hnsw ((embedding::halfvec(3)) halfvec_l2_ops);
```

Get the nearest neighbors:

```
SELECT * FROM items ORDER BY embedding::halfvec(3) <-> '[1,2,3]' LIMIT 5;
```

Binary vectors

Use the `bit` type to store binary vectors:

```
CREATE TABLE items (id bigserial PRIMARY KEY, embedding bit(3));
INSERT INTO items (embedding) VALUES ('000'), ('111');
```

Get the nearest neighbors using the Hamming distance:

```
SELECT * FROM items ORDER BY bit_count(embedding # '101') LIMIT 5;
```

Hybrid search

Perform hybrid search using SynxDB full-text search:

```
SELECT id, content FROM items, plainto_tsquery('hello search') query
  WHERE textsearch @@ query ORDER BY ts_rank_cd(textsearch, query) DESC
LIMIT 5;
```

pgvector performance

Use EXPLAIN ANALYZE for performance debugging:

```
EXPLAIN ANALYZE SELECT * FROM items ORDER BY embedding <-> '[3,1,2]' LIMIT 5;
```

Exact search

To speed up queries without an index, you can increase the value of the max_parallel_workers_per_gather parameter.

```
SET max_parallel_workers_per_gather = 4;
```

If vectors are already normalized to a length of 1 (for example, the [OpenAI embeddings](#)), using inner product can provide the best performance.

```
SELECT * FROM items ORDER BY embedding <#> '[3,1,2]' LIMIT 5;
```

Approximate search

To speed up queries with an index, you can increase the number of inverted lists (at the cost of some recall).

```
CREATE INDEX ON items USING ivfflat (embedding vector_12_ops) WITH (lists =
1000);
```

These are some guidelines for nearest neighbor search and performance optimization in pgvector. Depending on your needs and data structure, you can adjust and optimize based on these

recommendations.

Vectorization Query Computing

When handling large datasets, the vectorization execution engine can greatly improve computing efficiency. By vectorizing data, multiple data elements can be processed at the same time, using parallel computing and SIMD instruction sets to speed up the process. SynxDB Vectorization (referred to as Vectorization) is a vectorization plugin based on the SynxDB kernel, designed to optimize query performance.

Enable vectorization

1. Before enabling SynxDB Vectorization, you need to install and deploy SynxDB.
2. Connect to the target database. Replace `your_database_name` with the actual name of your database.

```
$ psql your_database_name;
```

3. Enable vectorization in the database.

```
SET vector.enable_vectorization = ON;
```

Usage

Vectorization provides two parameters for users: `vector.enable_vectorization` and `vector.max_batch_size`.

Parameter name	Description
<code>vector.enable_vectorization</code>	Controls whether vectorized queries are enabled. It is disabled by default.
<code>vector.max_batch_size</code>	The size of the vectorized batch, which controls how many rows the executor processes in one cycle. The range is [0, 600000], and the default value is 16384.

Enable or disable vectorization queries

You can temporarily enable or disable the vectorization feature in a connection by setting the `vector.enable_vectorization` variable. This setting is effective only for the current connection, and it will reset to the default value after disconnecting.

After uninstalling vectorization, setting the `vector.enable_vectorization` variable will have no effect. When you reinstall it, the `vector.enable_vectorization` will be restored to the default value `on`.

```
SET vector.enable_vectorization TO [on | off];
```

Set vectorization batch size

The batch size of vectorization greatly affects performance. If the value is too small, queries might slow down. If the value is too large, it might increase memory usage without improving performance.

```
SET vector.max_batch_size TO <batch_number>;
```

Verify whether a query is vectorized

You can use `EXPLAIN` to check whether a query is a vectorization query.

- If the first line of the `QUERY PLAN` has a “Vec” label, it means the query uses vectorization.

```
gpadmin=# EXPLAIN SELECT * FROM int8_tbl;
          QUERY PLAN
-----
-----  

Vec Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..431.00
rows=1 width=16)
->  Vec Seq Scan on int8_tbl  (cost=0.00..431.00 rows=1 width=16)
Optimizer: Pivotal Optimizer (GPORCA)
(3 rows)
```

- If the first line of the `QUERY PLAN` does not have a “Vec” label, it means the query does

not use vectorization.

```
gpadmin=# EXPLAIN SELECT * FROM int8_tbl;
          QUERY PLAN
-----
-----
Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..431.00
  rows=1 width=16)
->  Seq Scan on int8_tbl  (cost=0.00..431.00 rows=1 width=16)
Optimizer: Pivotal Optimizer (GPORCA)
(3 rows)
```

Features supported by vectorization

Feature	Supported or not	Description
Storage format	Supported	AOCS
Storage format	Not supported	HEAP
Data types	Supported	int2, int4, int8, float8, bool, char, tid, date, time, timestamp, timestamptz, varchar, text, numeric
Data types	Not supported	Custom type
Scan operator	Supported	Scanning AOCS tables, complex filter conditions
Scan operator	Not supported	Non-AOCS tables
Agg operator	Supported	Aggregate functions: min, max, count, sum, avg Aggregation plans: PlanAggregate (simple aggregate), GroupAggregate (sorted aggregate), HashAggregate (hash aggregate)
Agg operator	Not Supported	Aggregate functions: sum(int8), sum(float8), stddev (standard deviation), variance Aggregation plans: MixedAggregate (mixed aggregate)
Limit operator	Supported	All
ForeignScan operator	Supported	All
Result operator	Supported	All
Append operator	Supported	All
Subquery operator	Supported	All
Sequence operator	Supported	All
NestedLoopJoin operator	Supported	Join types: inner join, left join, semi join, anti join
NestedLoopJoin operator	Not supported	Join types: right join, full join, semi-anti join Join conditions: different data types, complex inequality conditions
Material operator	Supported	All
ShareInputScan operator	Supported	All
ForeignScan operator	Supported	All
HashJoin operator	Supported	Join types: inner join, left join, right join, full join, semi join, anti join, semi-anti join
HashJoin operator	Not Supported	Join conditions: different data types, complex inequality conditions
Sort operator	Supported	Sorting order: ascending, descending algorithms: order by, order by limit
Motion operator	Supported	GATHER (sending tuples from multiple senders to one receiver), GATHER_SINGLE (single-node gathering), HASH (simple hash conditions), BROADCAST (broadcast gathering), EXPLICIT (explicit gathering)
Motion operator	Not supported	Hash gathering (complex hash conditions)
Expressions	Supported	case when, is distinct, is not distinct, grouping, groupid, stddev_sample, abs, round, upper, textcat, date_pli, coalesce, substr
Bench	Supported	ClickHouse, TPC-H, TPC-DS, ICW, ICW-ORCA

Threaded execution on single node

By performing threaded parallel computation within the execution node, better utilization of multi-core machine resources can be achieved, reducing query time and improving query performance. The following vectorized operators support threaded acceleration:

- Scan (Filter)
- Join
- Agg
- Sort

Currently, the vectorized query feature supports enabling threaded queries in a single-node deployment. To enable it, the following GUC values need to be configured:

```
set vector.enable_vectorization=on;    -- Enable vectorized query
set vector.enable_plan_merge=on;    -- Enable Pipeline scheduling execution
set vector.pool_threads=8;    -- Configure the number of execution threads
```

Vectorized threaded execution relies on the underlying PAX storage file format. Take the tpccds PAX table store_sales as an example. When threaded execution is not enabled:

```
gpadmin=# set vector.pool_threads=0;
SET
gpadmin=# explain analyse select ss_quantity from store_sales
where ss_quantity between 6 and 10 and ss_list_price between 5 and 5+10;
                                         QUERY PLAN
-----
-----
-----
Vec Seq Scan on store_sales  (cost=0.00..661957.82 rows=28801 width=4) (actual
time=8804.432..8805.236 rows
=94117 loops=1)
  Filter: ((ss_quantity >= 6) AND (ss_quantity <= 10) AND (ss_list_price >=
'5'::numeric(15,2)) AND (ss_li
t_price <= '15'::numeric(15,2)))
Planning Time: 0.475 ms
(slice0)      Executor memory: 114K bytes.
```

(continues on next page)

(continued from previous page)

```
Memory used: 128000kB
Optimizer: Postgres query optimizer
Execution Time: 8809.841 ms
(7 rows)
```

When threaded execution is enabled and the pool_threads parameter is set to 8:

```
gpadmin=# set vector.pool_threads=8;
SET
gpadmin=# explain analyse select ss_quantity from store_sales
where ss_quantity between 6 and 10 and ss_list_price between 5 and 5+10;
                                         QUERY PLAN
-----
-----
```

After enabling threaded queries, the query time is significantly reduced.

Performance evaluation

TPC-H

There are 22 query SQL statements in TPC-H. Compared to non-vectorized execution, vectorization improves the overall performance by 2 times or more. For pure aggregation SQL statements, vectorization can speed up the process by 3 times or more compared to non-vectorized execution.

Table 3: Performance comparison of vectorized queries

Statement	Query time without vec (s)	Query time with vec (s)	Time diff (s) = non-vec - vec	Improvement (times)
1	18	54	36	3
2	3	9	6	3
3	14	23	9	1.64
4	22	44	22	2
5	11	28	17	2.54
6	7	10	3	1.43
7	13	22	9	1.69
8	11	28	17	2.55
9	21	62	41	2.95
10	12	22	10	1.83
11	5	5	0	
12	11	15	4	1.36
13	13	28	15	2.15
14	7	10	3	1.43
15	7	12	5	1.71
16	4	7	3	1.75
17	20	92	72	4.6
18	20	79	59	3.95
19	13	13	0	
20	13	23	10	1.77
21	61	72	11	1.15
22	18	18	0	
Total	324	676	352	2.086419753

TPC-DS

TPC-DS has 99 query SQL statements, tested in a 1T data environment. Compared to non-vectorized execution, vectorization improves the overall performance by 2 times or more.

Table 4: Performance comparison of vectorized queries

Statement	Query time without vec (s)	Query time with vec (s)	Time diff (s) = non-vec - vec	Improvement (times)
1	5	2	3	2.50
2	10	4	6	2.50
3	5	2	3	2.50
4	41	19	22	2.16
5	11	11	0	1.00
6	10	4	6	2.50
7	12	4	8	3.00
8	13	8	5	1.63

continues on next page

Table 4 – continued from previous page

Statement	Query time without vec (s)	Query time with vec (s)	Time diff (s) = non-vec - vec	Improvement (times)
9	11	7	4	1.57
10	12	5	7	2.40
11	30	14	16	2.14
12	3	2	1	1.50
13	10	6	4	1.67
14	46	37	9	1.24
15	10	2	8	5.00
16	13	7	6	1.86
17	13	5	8	2.60
18	15	5	10	3.00
19	12	4	8	3.00
20	3	2	1	1.50
21	6	1	5	6.00
22	14	5	9	2.80
23	125	43	82	2.91
24	147	60	87	2.45
25	13	4	9	3.25
26	7	2	5	3.50
27	11	5	6	2.20
28	7	6	1	1.17
29	12	4	8	3.00
30	11	3	8	3.67
31	13	6	7	2.17
32	7	2	5	3.50
33	10	5	5	2.00
34	11	4	7	2.75
35	16	5	11	3.20
36	10	4	6	2.50
37	6	3	3	2.00
38	23	7	16	3.29
39	25	22	3	1.14
40	5	2	3	2.50
41	1	1	0	1.00
42	5	2	3	2.50
43	7	3	4	2.33
44	4	4	0	1.00
45	12	3	9	4.00
46	17	6	11	2.83
47	13	7	6	1.86
48	8	5	3	1.60
49	7	6	1	1.17
50	11	3	8	3.67
51	11	18	-7	0.61
52	6	2	4	3.00
53	7	3	4	2.33
54	20	16	4	1.25

continues on next page

Table 4 – continued from previous page

Statement	Query time without vec (s)	Query time with vec (s)	Time diff (s) = non-vec - vec	Improvement (times)
55	6	2	4	3.00
56	9	7	2	1.29
57	8	4	4	2.00
58	7	4	3	1.75
59	19	9	10	2.11
60	9	6	3	1.50
61	13	5	8	2.60
62	4	2	2	2.00
63	7	3	4	2.33
64	28	13	15	2.15
65	18	7	11	2.57
66	5	3	2	1.67
67	489	205	284	2.39
68	16	8	8	2.00
69	10	4	6	2.50
70	17	16	1	1.06
71	9	4	5	2.25
72	80	47	33	1.70
73	11	4	7	2.75
74	25	9	16	2.78
75	20	13	7	1.54
76	4	4	0	1.00
77	9	5	4	1.80
78	29	12	17	2.42
79	16	6	10	2.67
80	11	7	4	1.57
81	10	3	7	3.33
82	10	4	6	2.50
83	3	3	0	1.00
84	8	2	6	4.00
85	6	3	3	2.00
86	4	2	2	2.00
87	24	7	17	3.43
88	21	8	13	2.63
89	7	4	3	1.75
90	4	1	3	4.00
91	6	2	4	3.00
92	6	1	5	6.00
93	11	3	8	3.67
94	8	5	3	1.60
95	66	24	42	2.75
96	7	2	5	3.50
97	14	5	9	2.80
98	6	3	3	2.00
99	7	3	4	2.33
Total	000	916	/	2.18

Use PostGIS for Geospatial Data Analysis

PostGIS is a spatial extension for PostgreSQL, enabling the database to store geographic information system (GIS) objects. SynxDB supports PostGIS, which also includes optional features, such as GiST-based R-tree spatial indexes and functions for analyzing and processing GIS objects. In addition, PostGIS as a SynxDB extension supports the PostGIS Raster data type. PostGIS Raster, together with PostGIS' s geometry data types, provides a set of SQL functions (for example, ST_Intersects) that seamlessly integrate vector and raster geospatial data processing. PostGIS Raster uses GDAL (geospatial data abstraction library) as its conversion library for raster geospatial data formats, giving applications a unified raster data model.

For details about SynxDB' s support and limitations for the PostGIS extension, see *PostGIS support and limitations*.

For more information about PostGIS, visit <https://postgis.net/>.

Create the PostGIS extension

The following steps enable the PostGIS extension and other extensions used with PostGIS.

1. To enable PostGIS and PostGIS Raster in a database, run the following command after logging into the database.

```
CREATE EXTENSION postgis;
```

To enable PostGIS and PostGIS Raster in a specific schema, create the schema first, set the search_path to the PostGIS schema, and then enable the postgis extension using the WITH SCHEMA clause.

```
SHOW search_path; -- Views the current search_path
CREATE SCHEMA <schema_name>;
SET search_path TO <schema_name>;
CREATE EXTENSION postgis WITH SCHEMA <schema_name>;
```

After enabling the extension, reset the search_path. You can include the PostGIS schema in the search_path if needed.

2. If necessary, you can enable the PostGIS TIGER geocoder after enabling the postgis extension.

To enable the PostGIS TIGER geocoder, you need to enable the `fuzzystrmatch` extension before enabling `postgis_tiger_geocoder`. Use the following commands to enable these extensions.

```
CREATE EXTENSION fuzzystrmatch;
CREATE EXTENSION postgis_tiger_geocoder;
```

3. In addition, if needed, you can enable the rule-based address standardizer and add rule tables for the standardizer.

```
CREATE EXTENSION address_standardizer;
CREATE EXTENSION address_standardizer_data_us;
```

Enable GDAL raster drivers

To process raster data in PostGIS, such as using functions like `ST_AsJPEG()`, GDAL raster drivers are required. By default, all raster drivers are disabled. To enable raster drivers, set the `POSTGIS_GDAL_ENABLED_DRIVERS` environment variable in the `greenplum_path.sh` file on each host in the SynxDB cluster.

Alternatively, you can set drivers at the session level by modifying the `postgis.gdal_enabled_drivers` parameter directly. For example, to enable 3 GDAL raster drivers in a session in SynxDB, use the following SET command:

```
SET postgis.gdal_enabled_drivers TO 'GTiff PNG JPEG';
```

To reset the enabled drivers in the session to the default, use the following SET command:

```
SET postgis.gdal_enabled_drivers = default;
```

To view the list of GDAL raster drivers supported by SynxDB, run the `raster2pgsql` tool with the `-G` option on the SynxDB coordinator.

```
raster2pgsql -G
```

The command lists the driver long format name. The *GDAL Raster Formats* table at http://www.gdal.org/formats_list.html lists the long format names and the corresponding codes that you specify as the value of the environment variable. For example, the code for the long name Portable

Network Graphics is PNG. The following example export line enables four GDAL raster drivers.

```
export POSTGIS_GDAL_ENABLED_DRIVERS="GTiff PNG JPEG GIF"
```

Use the `gpstop -r` command to restart the SynxDB system so that the updated `greenplum_path.sh` settings take effect.

After updating the `greenplum_path.sh` file on all hosts and restarting the SynxDB system, you can display the enabled raster drivers using the `ST_GDALDrivers()` function. The following SELECT command lists all enabled raster drivers.

```
SELECT short_name, long_name FROM ST_GDALDrivers();
```

Enable out-of-database raster feature

After installing PostGIS, support for out-of-database rasters is disabled by default. This setting is controlled by the `POSTGIS_ENABLE_OUTDB_RASTERS` variable in the `greenplum_path.sh` file, which is set to 0 by default. To enable this feature, change the value to `true` (any non-zero value) and make the same update on all hosts. Then, restart the SynxDB system.

You can also enable or disable this feature only for the current SynxDB session. For example, the following SET command enables the feature for the current session only.

```
SET postgis.enable_outdb_rasters = true;
```

Note

When out-of-database raster support is enabled, you can specify which raster formats to use with the server configuration parameter `postgis.gdal_enabled_drivers`.

Remove PostGIS support

To remove the PostGIS extension and its related extensions, use the `DROP EXTENSION` command.

Removing PostGIS support from the database does not delete these PostGIS raster environment variables from the `greenplum_path.sh` file: `GDAL_DATA`, `POSTGIS_ENABLE_OUTDB_RASTERS`, and `POSTGIS_GDAL_ENABLED_DRIVERS`.

Note

Removing PostGIS support from the database will delete all PostGIS data objects without warning. If you are accessing PostGIS objects, the removal process might be interrupted.

Use the `DROP EXTENSION` command

Depending on the extensions you have enabled for PostGIS, you can remove their support from the database.

- If you have enabled the address standardizer and example rule table, use the following commands to remove these extensions from the current database.

```
DROP EXTENSION IF EXISTS address_standardizer_data_us;
DROP EXTENSION IF EXISTS address_standardizer;
```

- If you have enabled the TIGER geocoder and the `fuzzystrmatch` extension, use the following commands to remove these extensions from the current database.

```
DROP EXTENSION IF EXISTS postgis_tiger_geocoder;
DROP EXTENSION IF EXISTS fuzzystrmatch;
```

- To remove support for PostGIS and PostGIS Raster, use the following command to remove the extension from the current database.

```
DROP EXTENSION IF EXISTS postgis;
```

- If you have enabled PostGIS in a specific schema using the `CREATE EXTENSION` command, you might need to update the `search_path` and remove the PostGIS schema

as needed.

Usage examples

Scenario 1: Use PostGIS to create a non-openGIS table, insert and query geometric objects

```
-- Creates a table named geom_test.

CREATE TABLE geom_test ( gid int4, geom geometry,
    name varchar(25) );

-- Inserts a row with gid 1, a 3D polygon object (a 3D square) in WKT format
-- for the geometry field, and name as '3D Square'.
INSERT INTO geom_test ( gid, geom, name )
VALUES ( 1, 'POLYGON((0 0 0,0 5 0,5 5 0,5 0 0,0 0 0))', '3D Square' );

-- Inserts a second row with gid 2, a 3D linestring geometry, and name as '3D
Line'.
INSERT INTO geom_test ( gid, geom, name )
VALUES ( 2, 'LINESTRING(1 1 1,5 5 5,7 7 5)', '3D Line' );

-- Inserts a third row with gid 3, a 2D multipoint object, and name as '2D
Aggregate Point'.
INSERT INTO geom_test ( gid, geom, name )
VALUES ( 3, 'MULTIPOINT(3 4,8 9)', '2D Aggregate Point' );

-- Uses ST_GeomFromEWKT to create a 3D linestring object from EWKT,
-- then uses Box3D to get the 3D bounding box for this object.
-- Uses the && operator to query all rows in the geom_test table
-- where the geom field intersects this bounding box.
SELECT * from geom_test WHERE geom &&
Box3D(ST_GeomFromEWKT('LINESTRING(2 2 0, 3 3 0)'));
```

Scenario 2: Use PostGIS to create a georeferenced table, insert geocoded point data, and output points in standard text format

```
-- Creates a table named geotest.

CREATE TABLE geotest (id INT4, name VARCHAR(32) );

-- Adds a column named geopoint to the geotest table, defines it as POINT
-- type,
-- sets it to 2D coordinates, and specifies the spatial reference system
-- (SRID) as 4326 (WGS84).
SELECT AddGeometryColumn('geotest','geopoint', 4326,'POINT',2);

-- Inserts a row with id 1, name 'Olympia', and geopoint as a point created
-- with ST_GeometryFromText
-- from WKT text, with coordinates (-122.90, 46.97) and SRID 4326.
INSERT INTO geotest (id, name, geopoint)
VALUES (1, 'Olympia', ST_GeometryFromText('POINT(-122.90 46.97)', 4326));

-- Inserts a second row with id 2, name 'Renton',
-- with point coordinates (-122.22, 47.50) and SRID 4326.
INSERT INTO geotest (id, name, geopoint)
VALUES (2, 'Renton', ST_GeometryFromText('POINT(-122.22 47.50)', 4326));

-- Selects the name and geopoint fields from the geotest table,
-- but converts the geopoint field to standard text (WKT) format using ST_
-- AsText.
SELECT name, ST_AsText(geopoint) FROM geotest;
```

Scenario 3: Support spatial indexes

```
-- Creates a table
CREATE TABLE spatial_data (
    id SERIAL PRIMARY KEY,
    geom geometry
);

-- Inserts data
INSERT INTO spatial_data (geom) VALUES
```

(continues on next page)

(continued from previous page)

```
(ST_GeomFromText('POINT(0 0)'),  

(ST_GeomFromText('POINT(1 1)'),  

(ST_GeomFromText('POLYGON((0 0, 4 0, 4 4, 0 4, 0 0))'));  

-- Creates a spatial index  

CREATE INDEX spatial_data_gist_idx  

ON spatial_data  

USING GIST (geom);
```

PostGIS support and limitations

This section describes the features supported by the SynxDB PostGIS extension and its limitations. In general, the SynxDB PostGIS extension does not support the following features:

- The PostGIS topology extension (`postgis_topology`)
- The PostGIS 3D and geoprocessing extension (`postgis_sfrcgal`)
- Certain user-defined functions and aggregate operations
- Long-duration transactions in PostGIS

Supported PostGIS data types

The SynxDB PostGIS extension supports the following PostGIS data types:

- `box2d`
- `box3d`
- `geometry`
- `geography`

For a complete list of PostGIS data types, operators, and functions, see the [PostGIS Reference Documentation](#).

Supported PostGIS indexes

The SynxDB PostGIS extension supports GiST (generalized search tree) indexes.

Limitations of the PostGIS extension

This section introduces the limitations of the SynxDB PostGIS extension about user-defined functions (UDFs), data types, and aggregations.

- SynxDB does not support data types and functions related to PostGIS topology features, such as `TopoGeometry`.
- SynxDB does not support the following PostGIS aggregates:
 - `ST_Collect`
 - `ST_MakeLine`

In a SynxDB cluster with multiple segments, calling the same aggregate function repeatedly might yield different results.

- SynxDB does not support long-duration transactions in PostGIS.

PostGIS relies on triggers and the `public.authorization_table` in PostGIS tables to support long-duration transactions. When PostGIS tries to lock a long-duration transaction, SynxDB returns an error, indicating that the function cannot access a table named `authorization_table`.

- SynxDB does not support the `_postgis_index_extent` function.
- The `<->` operator (`geometry <-> geometry`) returns the distance between the centroids of two geometries.
- SynxDB supports the TIGER geocoder extension but does not support upgrading the TIGER geocoder extension.
- The `standardize_address()` function accepts tables like `lex`, `gaz`, or `rules` as parameters. If you use tables other than `us_lex`, `us_gaz`, or `us_rules`, set them to a `DISTRIBUTED REPLICATED` distribution policy to ensure they work correctly in SynxDB.

Use MADlib for Machine Learning and Deep Learning

MADlib is an open-source library that provides scalable, in-database analytics functionalities. It implements data-parallel mathematical, statistical, and machine learning algorithms for both structured and unstructured data.

In SynxDB, you can use MADlib by installing the MADlib extension. MADlib offers a set of SQL-based machine learning, data mining, and statistical algorithms, which run at scale within the database engine. This spares you from transferring data between the database and other tools.

Using MADlib leverages the scalability and performance of the database, allowing you to perform analyses within a familiar SQL environment, which improves efficiency. It overcomes memory and CPU limitations often encountered with external database tools.

Install MADlib components

To install the MADlib components, you first need to install a compatible SynxDB MADlib package, and then add the MADlib functions to the database.

The `gppkg` tool installs database extensions and their dependencies across all hosts in a SynxDB cluster. `gppkg` also automatically installs extensions on any new hosts added during system expansion or segment recovery.

Install the SynxDB MADlib package

Before installing the MADlib package, make sure that SynxDB is running, that `greenplum_path.sh` is configured, and that the `$MASTER_DATA_DIRECTORY` and `$GPHOME` variables are set.

1. Obtain the MADlib extension package.
2. Copy the MADlib package to the coordinator node host of SynxDB.
3. Extract the MADlib extension package, as shown in the following command:

```
$ tar xzvf madlib-1.21.0+1-gp5-rhel7-x86_64.tar.gz
```

4. Use the `gppkg` tool to install the package, as shown in the following command:

```
$ gppkg -i ./madlib-1.21.0+1-gp5-rhel7-x86_64/madlib-1.21.0+1-gp5-rhel7-x86_64.gppkg
```

Add MADlib functions to the database

After installing the MADlib package, run the `madpack` command to add MADlib functions to SynxDB. The `madpack` tool is located in the `$GPHOME/madlib/bin` directory.

```
$ $GPHOME/madlib/bin/madpack install [-s <schema_name>] -p cloudberry -c <user>@<host>:<port>/<database>
```

For example, to create MADlib functions in the `testdb` database on the server `mdw` with port 5432, specify the `gpadmin` user in the `madpack` command, which will prompt for a password. The target schema is `madlib`.

```
$ $GPHOME/madlib/bin/madpack install -s madlib -p cloudberry -c gpadmin@mdw:5432/testdb
```

After installing the functions, the superuser role `gpadmin` in SynxDB grants all permissions on the target schema (`madlib` in this example) to users who need access to MADlib functions. Without access permissions, users will encounter the error `ERROR: permission denied for schema MADlib` when trying to access the target schema.

Uninstall MADlib from the database

To uninstall the MADlib components from the SynxDB cluster, use the `madpack uninstall` command. The following example removes MADlib objects from the `testdb` database. Any schema or other database objects that rely on MADlib will also be removed.

```
$ $GPHOME/madlib/bin/madpack uninstall -s madlib -p cloudberry -c gpadmin@mdw:5432/testdb
```

Usage examples

Check the MADlib version

```
SELECT version FROM madlib.migrationhistory ORDER BY applied DESC LIMIT 1;
```

Scenario 1: Perform linear regression on a database table

The following example runs a linear regression on the `regr_example` table. The dependent variable data is in the `y` column, and the independent variable data is in the `x1` and `x2` columns.

The statements below create the `regr_example` table and load some sample data:

```
DROP TABLE IF EXISTS regr_example;
CREATE TABLE regr_example (
    id int,
    y int,
    x1 int,
    x2 int
);
INSERT INTO regr_example VALUES
    (1, 5, 2, 3),
    (2, 10, 7, 2),
    (3, 6, 4, 1),
    (4, 8, 3, 4);
```

MADlib's `linregr_train()` function generates a regression model from the input table containing training data. The following SELECT statement performs a simple multiple regression on the `regr_example` table and saves the model in the `reg_example_model` table.

```
SELECT madlib.linregr_train (
    'regr_example',           -- Source table
    'regr_example_model',     -- Output model table
    'y',                      -- Dependent variable
    'ARRAY[x1, x2]'          -- Independent variables
);
```

The `madlib.linregr_train()` function can accept additional parameters to set grouping

columns and calculate heteroskedasticity of the model.

⚠️ Attention

The intercept is calculated by setting one of the independent variables to the constant 1, as shown in the example above.

Running this query against the `regr_example` table creates the `regr_example_model` table containing one row of data:

(continues on next page)

(continued from previous page)

```
09,0.20873079069527753} | 22.650203241881005 | 4 |
    0 | {{2.2376543209859783,-0.2572016460903422,-0.4372427983535821},
{-0
.2572016460903422,0.042866941015057024,0.034293552812045644},{-0.
437242798353582
1,0.03429355281204565,0.12002743484215979}}}
(1 row)
```

The model saved in the `regr_example_model` table can be used with the MADlib linear regression prediction function, `madlib.linregr_predict()`, to view the residuals:

```
SELECT regr_example.*,
        madlib.linregr_predict ( ARRAY[1, x1, x2], m.coef ) as predict,
        y - madlib.linregr_predict ( ARRAY[1, x1, x2], m.coef ) as residual
FROM regr_example, regr_example_model m;
```

id	y	x1	x2	predict	residual
4	8	3	4	7.629629629629636	0.370370370370364
1	5	2	3	5.462962962962971	-0.4629629629629708
2	10	7	2	10.185185185185201	-0.1851851851852011
3	6	4	1	5.72222222222238	0.2777777777777617

(4 rows)

Scenario 2: Use association rules

The following example shows the association rules data mining technique on a transactional data set. Association rule mining is a technique for discovering relationships between variables in a large data set. This example considers items in a store that are commonly purchased together. In addition to market basket analysis, association rules are also used in bioinformatics, web analytics, and other fields.

The example analyzes purchase information for seven transactions that are stored in a table with the MADlib function `MADlib.assoc_rules`. The function assumes that the data is stored in two columns with a single item and transaction ID per row. Transactions with multiple items consist of multiple rows with one row per item.

1. Create the table.

```
DROP TABLE IF EXISTS test_data;
CREATE TABLE test_data (
    trans_id INT,
    product text
);
```

2. Add the data to the table.

```
INSERT INTO test_data VALUES
(1, 'beer'),
(1, 'diapers'),
(1, 'chips'),
(2, 'beer'),
(2, 'diapers'),
(3, 'beer'),
(3, 'diapers'),
(4, 'beer'),
(4, 'chips'),
(5, 'beer'),
(6, 'beer'),
(6, 'diapers'),
(6, 'chips'),
(7, 'beer'),
(7, 'diapers');
```

The MADlib function `madlib.assoc_rules()` analyzes the data and determines association rules with the following characteristics.

- A support value of at least .40. Support is the ratio of transactions that contain X to all transactions.
- A confidence value of at least .75. Confidence is the ratio of transactions that contain X to transactions that contain Y. You can view this metric as the conditional probability of X given Y.

This SELECT command determines association rules, creates the table `assoc_rules`, and adds the statistics to the table.

```
SELECT * FROM madlib.assoc_rules (
    .40,          -- Support
```

(continues on next page)

(continued from previous page)

```
.75,          -- Confidence
'trans_id',   -- Transaction column
'product',    -- Product purchased column
'test_data',  -- Table name
'public',     -- Schema name
>false);      -- Displays processing details
```

This is the output of the SELECT command. There are two rules that fit the characteristics.

output_schema output_table total_rules total_time
public assoc_rules 2 00:00:04.340151
(1 row)

To view the association rules, run this SELECT command.

```
SELECT pre, post, support FROM assoc_rules
ORDER BY support DESC;
```

This is the output. The pre and post columns are the itemsets of left and right hand sides of the association rule respectively.

pre post support
{diapers} {beer} 0.7142857142857143
{chips} {beer} 0.42857142857142855
(2 rows)

Scenario 3: Perform Naive Bayes classification

Naive Bayes analysis predicts the likelihood of an outcome of a class variable, or category, based on one or more independent variables, or attributes. The class variable is a non-numeric categorial variable, a variable that can have one of a limited number of values or categories. The class variable is represented with integers, each integer representing a category. For example, if the category can be one of “true”, “false”, or “unknown,” the values can be represented with the integers 1, 2, or 3.

The attributes can be of numeric types and non-numeric, categorical, types. The training function

has two signatures –one for the case where all attributes are numeric and another for mixed numeric and categorical types. Additional arguments for the latter identify the attributes that should be handled as numeric values. The attributes are submitted to the training function in an array.

The MADlib Naive Bayes training functions produce a features probabilities table and a class priors table, which can be used with the prediction function to provide the probability of a class for the set of attributes.

Naive Bayes example 1 - Simple all-numeric attributes

In the first example, the `class` variable is either 1 or 2 and there are three integer attributes.

1. Create the input table and load sample data.

```
DROP TABLE IF EXISTS class_example CASCADE;
CREATE TABLE class_example (
    id int, class int, attributes int[]);
INSERT INTO class_example VALUES
(1, 1, '{1, 2, 3}'),
(2, 1, '{1, 4, 3}'),
(3, 2, '{0, 2, 2}'),
(4, 1, '{1, 2, 1}'),
(5, 2, '{1, 2, 2}'),
(6, 2, '{0, 1, 3}');
```

Actual data in production scenarios is more extensive than this example data and yields better results. Accuracy of classification improves significantly with larger training data sets.

2. Train the model with the `create_nb_prepared_data_tables()` function.

```
SELECT * FROM madlib.create_nb_prepared_data_tables (
    'class_example',           -- Name of the training table
    'class',                  -- Name of the class (dependent) column
    'attributes',             -- Name of the attributes column
    3,                        -- The number of attributes
    'example_feature_probs', -- Name for the feature probabilities output
    table
    'example_priors'         -- Name for the class priors output table
);
```

3. Create a table with data to classify using the model.

```
DROP TABLE IF EXISTS class_example_topredict;

CREATE TABLE class_example_topredict (
    id int, attributes int[]);

INSERT INTO class_example_topredict VALUES
(1, '{1, 3, 2}'),
(2, '{4, 2, 2}'),
(3, '{2, 1, 1});
```

4. Create a classification view using the feature probabilities, class priors, and class_example_topredict tables.

```
SELECT madlib.create_nb_probs_view (
    'example_feature_probs',      -- Feature probabilities output table
    'example_priors',            -- Class priors output table
    'class_example_topredict',   -- Table with data to classify
    'id',                        -- Name of the key column
    'attributes',                -- Name of the attributes column
    3,                           -- Number of attributes
    'example_classified'        -- Name of the view to create
);
```

5. Display the classification results.

```
SELECT * FROM example_classified;
```

key	class	nb_prob
1	1	0.4
1	2	0.5999999999999999
2	1	0.2499999999999992
2	2	0.75
3	1	0.5
3	2	0.5

(6 rows)

Naive Bayes example 2 –Weather and outdoor sports

This example calculates the probability that the user will play an outdoor sport, such as golf or tennis, based on weather conditions.

The table `weather_example` contains the example values.

The identification column for the table is `day`, an integer type.

The `play` column holds the dependent variable and has two classifications:

- 0 - No
- 1 - Yes

There are four attributes: outlook, temperature, humidity, and wind. These are categorical variables. The MADlib function `create_nb_classify_view()` expects the attributes to be provided as an array of `INTEGER`, `NUMERIC`, or `FLOAT8` values, so the attributes for this example are encoded with integers as follows:

- outlook might be sunny (1), overcast (2), or rain (3).
- temperature might be hot (1), mild (2), or cool (3).
- humidity might be high (1), or normal (2).
- wind might be strong (1) or weak (2).

The following table shows the training data, before encoding the variables.

<code>day</code>	<code>play</code>	<code>outlook</code>	<code>temperature</code>	<code>humidity</code>	<code>wind</code>
2	No	Sunny	Hot	High	Strong
4	Yes	Rain	Mild	High	Weak
6	No	Rain	Cool	Normal	Strong
8	No	Sunny	Mild	High	Weak
10	Yes	Rain	Mild	Normal	Weak
12	Yes	Overcast	Mild	High	Strong
14	No	Rain	Mild	High	Strong
1	No	Sunny	Hot	High	Weak
3	Yes	Overcast	Hot	High	Weak
5	Yes	Rain	Cool	Normal	Weak

(continues on next page)

(continued from previous page)

7	Yes	Overcast	Cool	Normal	Strong
9	Yes	Sunny	Cool	Normal	Weak
11	Yes	Sunny	Mild	Normal	Strong
13	Yes	Overcast	Hot	Normal	Weak

(14 rows)

1. Create the training table.

```
DROP TABLE IF EXISTS weather_example;
CREATE TABLE weather_example (
    day int,
    play int,
    attrs int[]
);
INSERT INTO weather_example VALUES
( 2, 0, '{1,1,1,1}' ), -- sunny, hot, high, strong
( 4, 1, '{3,2,1,2}' ), -- rain, mild, high, weak
( 6, 0, '{3,3,2,1}' ), -- rain, cool, normal, strong
( 8, 0, '{1,2,1,2}' ), -- sunny, mild, high, weak
(10, 1, '{3,2,2,2}' ), -- rain, mild, normal, weak
(12, 1, '{2,2,1,1}' ), -- etc.
(14, 0, '{3,2,1,1}' ),
( 1, 0, '{1,1,1,2}' ),
( 3, 1, '{2,1,1,2}' ),
( 5, 1, '{3,3,2,2}' ),
( 7, 1, '{2,3,2,1}' ),
( 9, 1, '{1,3,2,2}' ),
(11, 1, '{1,2,2,1}' ),
(13, 1, '{2,1,2,2}' );
```

2. Create the model from the training table.

```
SELECT madlib.create_nb_prepared_data_tables (
    'weather_example', -- Training source table
    'play',           -- Dependent class column
    'attrs',          -- Attributes column
    4,                -- Number of attributes
    'weather_probs', -- Feature probabilities output table
    'weather_priors' -- Class priors
);
```

- View the feature probabilities:

```
SELECT * FROM weather_probs;

class | attr | value | cnt | attr_cnt
-----+-----+-----+-----+
 0   |   3 |     1 |   4 |     2
 1   |   2 |     3 |   3 |     3
 0   |   2 |     3 |   1 |     3
 1   |   1 |     1 |   2 |     3
 1   |   2 |     1 |   2 |     3
 1   |   2 |     2 |   4 |     3
 1   |   4 |     1 |   3 |     2
 0   |   2 |     1 |   2 |     3
 0   |   1 |     1 |   3 |     3
 0   |   2 |     2 |   2 |     3
 0   |   4 |     1 |   3 |     2
 1   |   3 |     2 |   6 |     2
 0   |   3 |     2 |   1 |     2
 0   |   1 |     2 |   0 |     3
 1   |   1 |     3 |   3 |     3
 1   |   4 |     2 |   6 |     2
 0   |   1 |     3 |   2 |     3
 1   |   1 |     2 |   4 |     3
 1   |   3 |     1 |   3 |     2
 0   |   4 |     2 |   2 |     2
```

(20 rows)

- To classify a group of records with a model, first load the data into a table. In this example, the table t1 has four rows to classify.

```
DROP TABLE IF EXISTS t1;
CREATE TABLE t1 (
    id integer,
    attributes integer[]);

insert into t1 values
    (1, '{1, 2, 1, 1}),
    (2, '{3, 3, 2, 1}),
    (3, '{2, 1, 2, 2}),
    (4, '{3, 1, 1, 2});
```

5. Use the MADlib function `create_nb_classify_view()` to classify the rows in the table.

```
SELECT madlib.create_nb_classify_view ( -- Feature probabilities table  
    'weather_probs', -- ClassPriorsName  
    'weather_priors', -- Table containing values to classify  
    't1', -- Key column  
    'id', -- Attributes column  
    4, -- Number of attributes  
    't1_out' -- Output table name  
) ;
```

The result is four rows, one for each record in the `t1` table.

```
SELECT * FROM t1_out ORDER BY key;  
key | nb_classification  
----+-----  
1 | {0}  
2 | {1}  
3 | {1}  
4 | {0}  
(4 rows)
```

Directory Tables

SynxDB has introduced directory tables for unified management of unstructured data on local or object storages.

Large-scale AI applications need to handle unstructured, multi-modal datasets. Therefore, AI application developers must continually prepare a large amount high-quality unstructured data, train large models through repeated iterations, and build rich knowledge bases. This creates technical challenges in managing and processing unstructured data.

To address these challenges, SynxDB introduces directory tables for managing multiple types of unstructured data. Developers can use simple SQL statements to leverage multiple computing engines for unified data processing and application development.

Directory tables store, manage, and analyze unstructured data objects within tablespaces. When unstructured data files are imported, a directory table record (file metadata) is created, and the file itself is loaded into object storage. The table metadata remains linked to the corresponding object storage file.

Usage

Create a directory table

You can create a directory table in a local tablespace or in a tablespace of an external storage (such as object storage services or distributed file systems like HDFS).

Create in local storage

To create a directory table in local storage, follow the SQL syntax below. You need to replace <table_name> and <tablespace_name> with the actual table name and tablespace name.

```
-- Method 1: To create a directory table in a tablespace other than the
default one,
-- create the tablespace first and then create the directory table within that
tablespace.
CREATE DIRECTORY TABLE <table_name>;
```

(continues on next page)

(continued from previous page)

```
-- Method 2: To create a directory table in the default tablespace,
-- simply omit the TABLESPACE clause.

CREATE TABLESPACE <tablespace_name>
    LOCATION '<tablespace_path>';

CREATE DIRECTORY TABLE <table_name>
    TABLESPACE <tablespace_name>;
```

Create in external storage

To create a directory table in an external storage, you first need to create a tablespace in that storage. You'll need to provide connection information of the external storage server, such as server IP address, protocol, and access credentials. The following examples show how to create directory tables on QingCloud Object Storage and HDFS.

1. Create server objects and define connection methods for external data sources. SynxDB supports protocols for multiple storage options, including S3 object storage and HDFS. The following examples create server objects named `oss_server` and `hdfs_server` on QingCloud and HDFS, respectively.

- For QingCloud:

```
CREATE STORAGE SERVER oss_server OPTIONS(protocol 'qingstor', prefix '<path_prefix>', endpoint '<endpoint_address>', https 'true', virtual_host 'false');
```

- For HDFS:

```
CREATE STORAGE SERVER hdfs_server OPTIONS(port '<port_number>',
    protocol 'hdfs', namenode '<HDFS_node_IP:port_number>', https 'false');
```

The parameters in the above commands are described as follows:

- `protocol`: The protocol used to connect to the external data source. In the examples above, '`qingstor`' indicates using the QingCloud object storage service protocol, and '`hdfs`' indicates using the HDFS storage service protocol.

- **prefix:** Sets the path prefix when accessing object storage. If this prefix is set, all operations will be limited to this specific path, such as prefix '/rose-oss-test4/usf1'. This is typically used to organize and isolate data stored in the same bucket.
- **endpoint:** Specifies the network address of the external object storage service. For example, 'pek3b.qingstor.com' is a specific regional node of the QingCloud service. Through this endpoint, SynxDB can access external data.
- **https:** Specifies whether to connect to the object storage service using the HTTPS protocol. In this command, 'false' indicates using an unencrypted HTTP connection. This setting might be influenced by data transmission security requirements, and it is generally recommended to use HTTPS to ensure data security.
- **virtual_host:** Determines whether to access the bucket using virtual hosting. 'false' means that bucket access is not done in virtual host style (which means that the bucket name is not included in the URL). This option is typically dependent on the URL format support provided by the storage service provider.
- **namenode:** Represents the IP of the HDFS node. You need to replace <HDFS_node_IP:port> with the actual IP address and port number, such as '192.168.51.106:8020'.
- **port:** Specifies the port number of the HDFS node. You need to replace <port_number> with the real port number, for example, 8020.

2. Create user mappings to provide the current user with the authentication information required to access these external servers.

- For QingCloud:

```
CREATE STORAGE USER MAPPING FOR CURRENT_USER STORAGE SERVER oss_server
OPTIONS (accesskey '<QingCloud access key>', secretkey '<QingCloud
secret key>');
```

- HDFS:

```
CREATE STORAGE USER MAPPING FOR CURRENT_USER STORAGE SERVER hdfs_
server OPTIONS (auth_method 'simple');
```

The parameters in the above commands are described as follows:

- accesskey and secretkey: These parameters provide the necessary authentication information. 'accesskey' and 'secretkey' are similar to username and password, and are used to access the object storage service.
- auth_method: Indicates the authentication method for accessing HDFS. simple indicates simple authentication mode, and kerberos indicates using Kerberos authentication mode.

3. Create tablespaces on the external servers. These tablespaces are specifically associated with the previously defined external servers, and the location option of the tablespace specifies a specific path on the external storage. The following examples create tablespaces `dir_oss` and `dir_hdfs` on QingCloud and HDFS, respectively.

- For QingCloud:

```
CREATE TABLESPACE dir_oss location '<object_storage_path>' SERVER oss_
server HANDLER '$libdir/dfs_tablespace, remote_file_handler';

-- You need to replace <object_storage_path> with the actual path on
the object storage,
-- such as /tbs-49560-0-mqq-multi/oss-server-01-17.
```

- HDFS:

```
CREATE TABLESPACE dir_hdfs location '<object_storage_path>' SERVER
hdfs_server HANDLER '$libdir/dfs_tablespace, remote_file_handler';

-- You need to replace <object_storage_path> with the actual path on
the object storage,
-- such as /tbs-49560-0-mqq-multi/oss-server-01-17.
```

4. Create directory tables in the tablespaces. The following statements create directory tables `dir_table_oss` and `dir_table_hdfs` in tablespaces `dir_oss` and `dir_hdfs`, respectively.

```
CREATE DIRECTORY TABLE dir_table_oss TABLESPACE dir_oss;
CREATE DIRECTORY TABLE dir_table_hdfs TABLESPACE dir_hdfs;
```

 **Tip**

If you encounter the error message `directory ... does not exist` while creating the tablespace, you need to configure `shared_preload_libraries` for the cluster and then import the library to the object storage. To do this, follow these steps:

1. Run the command `gpconfig -c shared_preload_libraries -v 'dfs_tablespace'` to configure the cluster.
2. Restart the cluster using the command `gpstop -ra`.

Create index on a directory table

Directory tables support index to accelerate queries. The syntax to create index is as follows:

```
CREATE INDEX index_name ON <directory_table_name> (<column_name>);
```

View the field information of directory table

```
\dY -- Lists all the directory tables.  
\d <directory_table> -- Shows the field information of a directory table.
```

In general, the fields of a directory table are as follows:

Field name	Data type	Note
RELATIVE_PATH	TEXT	
SIZE	NUMBER	
LAST_MODIFIED	TIMESTAMP_LTZ	
MD5	HEX	
TAG	TEXT	User-defined tags. Can be used to mark data lineage, file upload department/team, classification. “k1=v1, k2=v2”

Upload file into directory table

After uploading a file to a directory table, SynxDB manages the file's upload to local storage or object storage and stores the file's metadata in the directory table. Currently, users cannot directly manage object storage directory files.

Upload files from local storage to database object storage:

```
\COPY BINARY '<directory_table_name>' FROM '<local_path_to_file>' '<relative_
path>';
COPY BINARY '<directory_table_name>' FROM '<local_path_to_file>' '<relative_
path>'; -- the starting slash \ can be omitted.

-- <directory_table_name> is the directory table name.
-- <local_path_to_file> is the local path to the file to be uploaded.
-- <relative_path> is the target path to the local or object storage,
-- and the file will be uploaded to this path.
```

Tip

It is recommended to use the subdirectory capability of <path> to ensure that the directory path after uploading is consistent with the local one, which simplifies file management.

For enhanced file management and data flow tracking, you can add tags to the upload command. These tags provide additional information or markings for your files.

```
\COPY BINARY '<directory_table_name>' FROM '<local_path_to_file>' '<relative_
path>' WITH tag '<tag_name>';
```

Examples:

```
-- Uploads the file to the root path.
\COPY BINARY dir_table_oss FROM '/data/country.data' 'country.data';

-- Uploads the file to a specified path top_level/second_level
\COPY BINARY dir_table_oss FROM '/data/region.tbl' 'top_level/second_level/
region.tbl';
```

(continues on next page)

(continued from previous page)

```
-- Uploads the file to the root path with a tag
\COPY BINARY dir_table_oss FROM '/data/country1.data' 'country1.data' with tag
'country';

-- Uploads the file to a specified path top_level/second_level with a tag
\COPY BINARY dir_table_oss FROM '/data/region1.tbl' 'top_level/second_level/
region1.tbl' with tag 'region';
```

You can also use the command-line tool `gpdirtableload` to upload files in bulk to object storage. Use the syntax `gpdirtableload --inputfile <directory>` to upload files in a directory to object storage. The command-line flags for `gpdirtableload` are as follows:

Listing 1: `gpdirtableload` Usage

```
Usage:
gpdirtableload [flags]

Flags:
  --database string          Database to connect to (default gpadmin)
  --dest-path string         Target path to the table root directory
                            (default: root directory of the table)
  --force-password-auth     Force a password prompt (default false)
  --help                     Print help info and exit
  --mode                     Upload or download data
  --host string              Host to connect to (default localhost)
  --input-file strings       Relative path to the directory of input files.
                            Uploading or downloading a single file is supported.
  --logfile string           Log output to logfile (default none)
  --port int                 Port to connect to (default 5432)
  --stop-on-error           Stop loading files when an error occurs (default
false)
  --table string             Table to load to
  --tag string               File tag
  --tasks int                The maximum number of files that concurrently
loads (default 1)
  --user string              User to connect as (default gpadmin)
  --verbose                  Indicates that the tool should generate verbose
output (default false)
  --version                  Print version info and exit
```

Export files to local

You can export data from a directory table to your local file system for backup:

```
-- Exports data to the machine where psql client is running.
\COPY BINARY DIRECTORY TABLE '<directory_table_name>' '<relative_path>' TO '<target_path>';

-- Exports to the machine where the coordinator is running.
COPY BINARY DIRECTORY TABLE '<directory_table_name>' '<relative_path>' TO '<target_path>';

-- <directory_table_name> is the table name.
-- <relative_path> is the relative path of the directory table.
-- <target_path> is the destination, including the path to your local file
system and the target path.
```

For example:

```
-- Exports file to your local root directory.
\COPY BINARY DIRECTORY TABLE dir_table_oss 'country.data' TO '/data/country.
CSV';
```

You can also use the command-line tool `gpdirtableload` to download files to your local system in a batch. For the detailed command-line flags, see the [gpdirtableload Usage](#).

Query and use the files managed by directory table

Query the metadata of a file in directory table:

```
-- Uses the table function directory_table() to read the file metadata and
content.
SELECT relative_path,
       size,
       last_modified,
       md5,
       tag,
       content
  FROM directory_table('<directory_table>');
```

(continues on next page)

(continued from previous page)

```
-- You can use one of the following statements to query the data of a
directory table.
```

```
SELECT * FROM <directory_table>;
SELECT * FROM DIRECTORY_TABLE('<directory_table>');
```

Add transaction lock to directory table

You can use the lock statement to control access to the directory table. This helps keep data consistent and safe. By using different lock types, you can limit how other transactions access the table to avoid problems.

Lock statement syntax:

```
LOCK TABLE <table_name> IN <lock_mode>;
```

Where `<lock_mode>` can be one of the following:

- ACCESS SHARE MODE: This lock lets other transactions read the table, but not change it. It is for read-only queries.
- ROW SHARE MODE: This lock lets other transactions read the rows in the table, but not change them. It works with `SELECT FOR UPDATE` or `FOR SHARE`.
- ROW EXCLUSIVE MODE: This lock lets the transaction change the table. Other transactions can read but not change it.
- SHARE UPDATE EXCLUSIVE MODE: This lock allows some maintenance tasks. Other transactions can read but not change the table.
- SHARE MODE: This lock lets other transactions read the table, but not change it.
- SHARE ROW EXCLUSIVE MODE: This lock allows some actions, like creating triggers. Other transactions can read but not change the table.
- EXCLUSIVE: This lock allows only the Access Share lock at the same time. It only lets the table be read. It is usually acquired by `REFRESH MATERIALIZED VIEW CONCURRENTLY`.

- ACCESS EXCLUSIVE MODE: This is the highest level of lock. It makes sure the holder is the only one accessing the table. It is acquired by commands like DROP TABLE, TRUNCATE, and VACUUM FULL.

Here is how to use the lock statement with ACCESS SHARE MODE:

```
BEGIN;
LOCK TABLE dir_table1 IN ACCESS SHARE MODE; -- Others can still read dir_table1
SELECT * FROM dir_table1; -- Read dir_table1
LOCK TABLE dir_table1 IN ACCESS EXCLUSIVE MODE; -- Request exclusive lock;
others cannot access
COMMIT;
```

Here is how to use the lock statement with ACCESS EXCLUSIVE MODE:

```
BEGIN;
LOCK TABLE dir_table1 IN ACCESS EXCLUSIVE MODE; -- Others cannot read or
change dir_table1

-- Do some actions...
ROLLBACK; -- Undo actions and release the lock
```

Delete the file managed by directory table

To delete a file managed by directory table, you need the admin privilege:

```
SELECT remove_file('dir_table_oss', 'country.data');

-- This command deletes the file country.data managed in the table dir_table_oss.
```

Delete a directory table

Delete a directory table. After the deletion, all the file managed by the table will be deleted as well. To delete a directory table, you need the admin privilege.

```
DROP DIRECTORY TABLE <table_name>;
```

Back up and restore directory tables

You can use the command-line tools `gpbackup` and `gprestore` to back up and restore data stored in directory tables.

For more details about `gpbackup` and `gprestore`, see [Backup and Restore Overview](#).

Implementation

The backup and restore process for directory tables is similar to that of regular tables, using the `copy to` and `copy from` commands. However, because the `copy` command for directory tables is not supported on segment nodes, directory tables are not implemented on segment nodes.

Back up directory tables using `gpbackup`

See the following example command to back up a local directory table:

```
gpbackup --include-table public.dir_table --dbname postgres --backup-dir /home/tmp/backup/
```

- `--include-table`: Backs up a single table, used the same way as for regular tables.
- `--include-table-file`: Backs up multiple tables, can back up directory tables along with other types of tables.
- Full backups are the same as full backups for regular tables, with no additional parameters required.
- Incremental backups are the same as incremental backups for regular tables, with no additional parameters required.

Restore directory tables using gprestore

See the following example command to restore a local directory table:

```
gprestore --timestamp 20240909152156 --backup-dir <local-directory> --on-error-continue
```

--backup-dir <local-directory>: This parameter specifies the local directory where the directory table backup is stored. For example, --backup-dir /home/tmp/backup/.

See the following command to restore a remote directory table:

```
gprestore --timestamp 20240923172522 --plugin-config <config-of-remote-directory>
```

--plugin-config <config-of-remote-directory>: This parameter points to the configuration file with remote directory information. For example, --plugin-config /home/gpadmin/s3.yaml. The file content is as follows:

```
executablename: /usr/local/cloudberry-db-1.6.0+dev.58.geb9f37ef/bin/gpbackup_s3_plugin
options:
endpoint: https://obs.***.myhuaweicloud.com
aws_access_key_id: ***
aws_secret_access_key: ***
bucket: ***
folder: backup_folder
```

4.4 Use Oracle Compatibility SQL via Orafce

SynxDB supports Orafce version 4.9.

The `orafce` module provides Oracle Compatibility SQL functions in SynxDB. These functions target PostgreSQL but can also be used in SynxDB.

The SynxDB database `orafce` module is a modified version of the [open source Orafce](#) PostgreSQL module extension.

Install and remove Orafce

The `orafce` module is installed when you install SynxDB. Before you can use any of the functions defined in the module, you must register the `orafce` extension in each database in which you want to use the functions.

Note

Always use the Oracle Compatibility Functions module included with your SynxDB version. Before upgrading to a new SynxDB version, uninstall the compatibility functions from each of your databases, and then, when the upgrade is complete, reinstall the compatibility functions from the new SynxDB release.

To register the module in the database named `testdb`, use the following command:

```
psql -d testdb -c 'CREATE EXTENSION orafce;' -- Install
```

You can remove it from a database using the following command:

```
psql -d testdb -c 'DROP EXTENSION orafce;' -- Remove
```

SynxDB considerations

There are some restrictions and limitations when you use the module in SynxDB.

The following functions are available by default in SynxDB and do not require installing the Oracle Compatibility Functions:

- `sinh()`
- `tanh()`
- `cosh()`
- `decode()`

SynxDB implementation differences

There are differences in the implementation of the compatibility functions in SynxDB from the original PostgreSQL `orafce` module extension implementation. Some of the differences are as follows:

- The original `orafce` module implementation performs a decimal round off, the SynxDB implementation does not:
 - 2.00 becomes 2 in the original module implementation
 - 2.00 remains 2.00 in the SynxDB implementation
- The provided Oracle compatibility functions handle implicit type conversions differently. For example, using the `decode` function:

```
decode(<expression>, <value>, <return> [,<value>, <return>] ...
      [, default])
```

The original `orafce` module implementation automatically converts expression and each value to the data type of the first value before comparing. It automatically converts return to the same data type as the first result.

The SynxDB implementation restricts `return` and `default` to be of the same data type. The expression and value can be different types if the data type of value can be converted into the data type of the expression. This is done implicitly. Otherwise,

decode fails with an invalid input syntax error. For example:

```
SELECT decode('a','M',true,false);

CASE
-----
f
(1 row)

SELECT decode(1,'M',true,false);

ERROR: Invalid input syntax for integer: *"M"
*LINE 1: SELECT decode(1,'M',true,false);
```

- Numbers in bigint format are displayed in scientific notation in the original orafce module implementation but not in the SynxDB implementation:
 - 9223372036854775 displays as 9.2234E+15 in the original implementation
 - 9223372036854775 remains 9223372036854775 in the SynxDB implementation
- The default date and timestamp format in the original orafce module implementation is different than the default format in the SynxDB implementation. If the following code is run:

```
CREATE TABLE TEST(date1 date, time1 timestamp, time2
                  timestamp with time zone);

INSERT INTO TEST VALUES ('2001-11-11','2001-12-13
                           01:51:15','2001-12-13 01:51:15 -08:00');

SELECT DECODE(date1, '2001-11-11', '2001-01-01') FROM TEST;
```

The SynxDB implementation returns the row, but the original implementation returns no rows.

i Note

The correct syntax when using the original *orafce* implementation to return the row is:

```
SELECT DECODE(to_char(date1, 'YYYY-MM-DD'), '2001-11-11', '2001-
01-01') FROM TEST
```

- The functions in the Oracle Compatibility Functions `dbms_alert` package are not implemented for SynxDB.
- The `decode()` function is removed from the SynxDB Oracle Compatibility Functions. The SynxDB parser internally converts a `decode()` function call to a CASE statement.

Using Orafce

Some Oracle Compatibility Functions reside in the `oracle` schema. To access them, set the search path for the database to include the `oracle` schema name. For example, this command sets the default search path for a database to include the `oracle` schema:

```
ALTER DATABASE <db_name> SET <search_path> = "$user", public, oracle;
```

Note the following differences when using the Oracle Compatibility Functions with PostgreSQL vs. using them with SynxDB:

- If you use validation scripts, the output may not be exactly the same as with the original `orafce` module implementation.
- The functions in the Oracle Compatibility Functions `dbms_pipe` package run only on the SynxDB coordinator host.
- The upgrade scripts in the Orafce project do not work with SynxDB.

Chapter 5

Optimize performance

5.1 Optimize Query Performance

Query Processing Overview

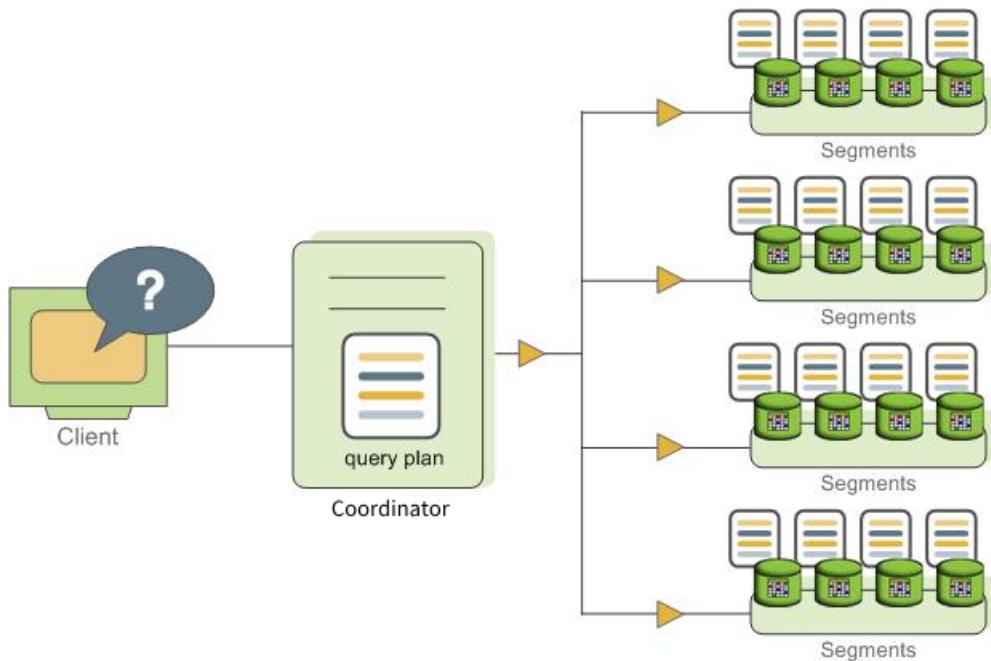
This topic provides an overview of how SynxDB processes queries. Understanding this process can be useful when writing and tuning queries.

Users issue queries to SynxDB as they would to any database management system. They connect to the database instance on the SynxDB coordinator host using a client application such as `psql` and submit SQL statements.

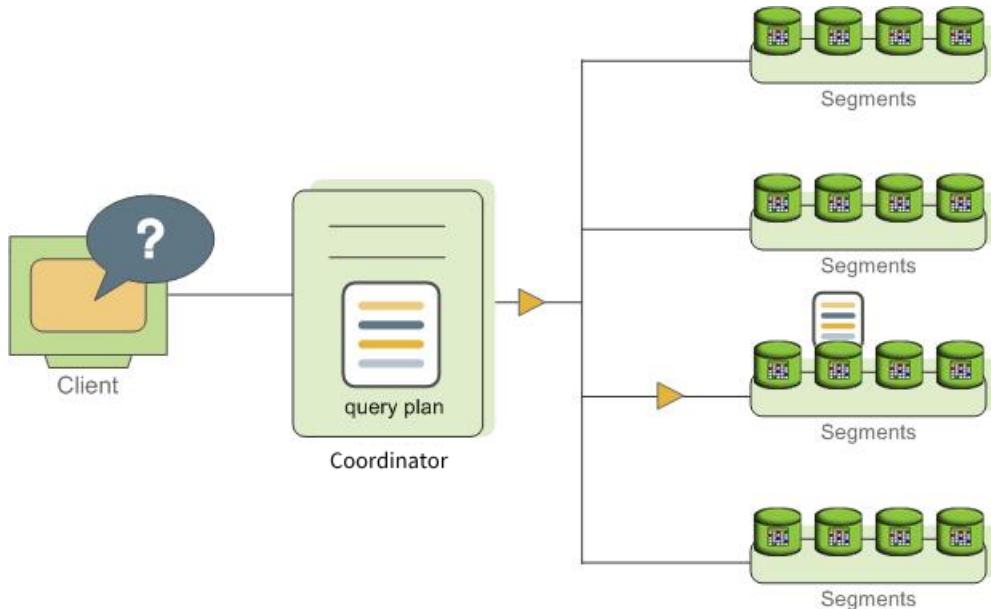
Query planning and dispatch

When a query is submitted, the SynxDB query processing system performs the following steps:

The coordinator receives, parses, and optimizes the query. The resulting query plan is either parallel or targeted. The coordinator dispatches parallel query plans to all segments.



The coordinator dispatches targeted query plans to a single segment. Each segment is responsible for running local database operations on its own set of data. Most database operations—such as table scans, joins, aggregations, and sorts—run across all segments in parallel. Each operation is performed on a segment database independent of the data stored in the other segment databases.



Certain queries may access only data on a single segment, such as single-row INSERT, UPDATE, DELETE, or SELECT operations or queries that filter on the table distribution key column(s). In queries such as these, the query plan is not dispatched to all segments, but is targeted at the segment

that contains the affected or relevant row(s).

Query plans

A query plan is the set of operations SynxDB will perform to produce the answer to a query. Each *node* or step in the plan represents a database operation such as a table scan, join, aggregation, or sort. Plans are read and run from bottom to top.

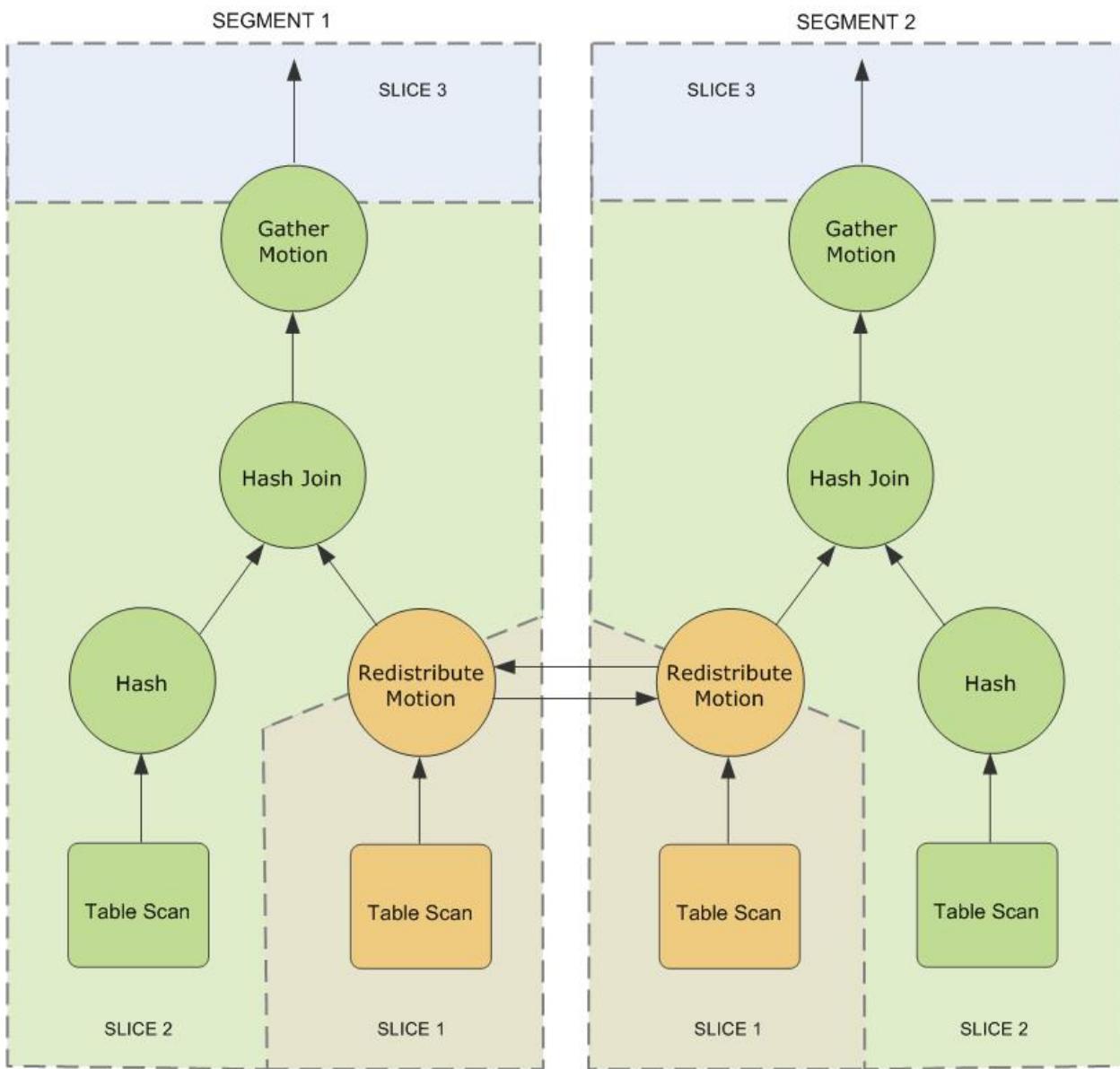
In addition to common database operations such as table scans, joins, and so on, SynxDB has an additional operation type called *motion*. A motion operation involves moving tuples between the segments during query processing. Note that not every query requires a motion. For example, a targeted query plan does not require data to move across the interconnect.

To achieve maximum parallelism during query runtime, SynxDB divides the work of the query plan into *slices*. A slice is a portion of the plan that segments can work on independently. A query plan is sliced wherever a *motion* operation occurs in the plan, with one slice on each side of the motion.

For example, consider the following simple query involving a join between two tables:

```
SELECT customer, amount
FROM sales JOIN customer USING (cust_id)
WHERE dateCol = '04-30-2016';
```

The following figure shows the query plan. Each segment receives a copy of the query plan and works on it in parallel.



The query plan for this example has a *redistribute motion* that moves tuples between the segments to complete the join. The redistribute motion is necessary because the customer table is distributed across the segments by `cust_id`, but the sales table is distributed across the segments by `sale_id`. To perform the join, the `sales` tuples must be redistributed by `cust_id`. The plan is sliced on either side of the redistribute motion, creating *slice 1* and *slice 2*.

This query plan has another type of motion operation called a *gather motion*. A gather motion is when the segments send results back up to the coordinator for presentation to the client. Because a query plan is always sliced wherever a motion occurs, this plan also has an implicit slice at the very top of the plan (*slice 3*). Not all query plans involve a gather motion. For example, a `CREATE TABLE x AS SELECT...` statement would not have a gather motion because tuples are sent to

the newly created table, not to the coordinator.

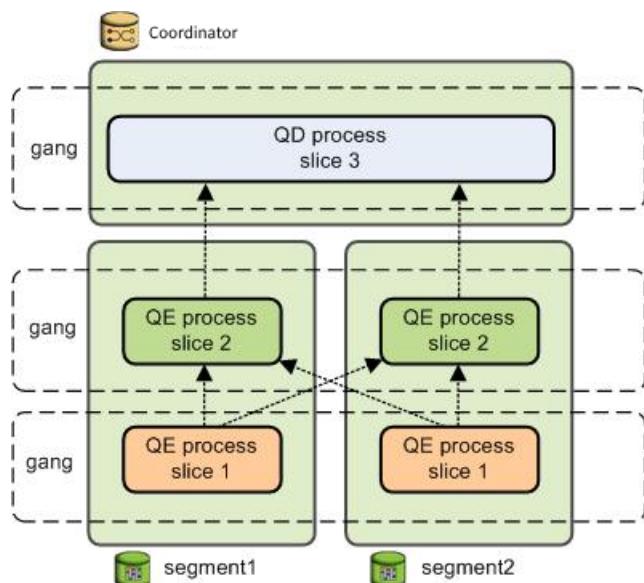
Parallel query execution

SynxDB creates a number of database processes to handle the work of a query. On the coordinator, the query worker process is called the *query dispatcher* (QD). The QD is responsible for dispatching the query plan. It also accumulates and presents the final results. On the segments, a query worker process is called a *query executor* (QE). A QE is responsible for completing its portion of work and communicating its intermediate results to the other worker processes.

There is at least one worker process assigned to each *slice* of the query plan. A worker process works on its assigned portion of the query plan independently. During query runtime, each segment will have a number of processes working on the query in parallel.

Related processes that are working on the same slice of the query plan but on different segments are called *gangs*. As a portion of work is completed, tuples flow up the query plan from one gang of processes to the next. This inter-process communication between the segments is referred to as the *interconnect* component of SynxDB.

The following figure shows the query worker processes on the coordinator and two segment instances for previous query plan.



Query Performance Overview

SynxDB improves query performance through dynamic partition elimination and adaptive memory allocation. These mechanisms help reduce the amount of data scanned, speed up query execution, and enhance overall concurrency.

Tip

SynxDB uses the GPORCA optimizer by default, which extends the native Postgres planner with more advanced optimization capabilities.

Dynamic partition elimination

SynxDB supports dynamic partition elimination (DPE), a feature that prunes partitions at query execution time based on runtime values. This reduces the data scanned and improves query efficiency.

DPE is supported for the following join types:

- Hash Inner Join
- Hash Left Join
- Hash Right Join (since v4.0.0)

DPE is enabled when the following conditions are met:

- The partitioned table is on the outer side of the join.
- The join condition is an equality predicate on the partition key.
- Statistics are collected on the partitioned tables. For example:

```
ANALYZE <root partition>;
```

The `gp_dynamic_partition_pruning` parameter controls whether DPE is enabled. It is ON by default but only applies to the Postgres optimizer. You can verify if DPE is in effect by checking the `EXPLAIN` plan for the presence of a `Partition Selector` node.

Memory optimization

SynxDB dynamically allocates memory based on the characteristics of query operators and proactively releases or reallocates memory during different query phases. This leads to more efficient memory usage and faster query execution.

Update Statistics

Accurate statistics are essential for good query performance. By running the `ANALYZE` command, you update table statistics, enabling the query optimizer to generate optimal execution plans. When SynxDB analyzes a table, it stores relevant statistics in system catalog tables. If these statistics become outdated, the optimizer may produce inefficient plans.

Checking if statistics are up to date

To check whether a table's statistics are current, use the `pg_stat_all_tables` system view. The `last_analyze` column shows the last time the table was manually analyzed, while `last_autoanalyze` shows the last automatic analyze time. Running an `ANALYZE` command updates both timestamps.

For example, to check the statistics status of the `test_analyze` table, run:

```
SELECT schemaname, relname, last_analyze, last_autoanalyze
FROM pg_stat_all_tables
WHERE relname = 'test_analyze';
```

Generate statistics selectively

Running `ANALYZE` without parameters updates statistics for all tables in the database, which can be very time-consuming and is generally not recommended. It's better to selectively analyze tables that have changed or to use the `analyzedb` utility.

For large tables, `ANALYZE` can take a long time. If you cannot analyze all columns, you can generate statistics for specific columns using `ANALYZE table(column, ...)`. Be sure to include columns used in joins, `WHERE` clauses, `SORT`, `GROUP BY`, or `HAVING` clauses.

For partitioned tables, it is sufficient to analyze only the partitions that have changed—for example, after adding a new partition. Note that `ANALYZE` can be run on either the root partitioned table or on individual leaf partitions (which store actual data and statistics). In SynxDB, analyzing a single partition also updates the root table's statistics, meaning it may influence the optimizer's plan for the entire partitioned table. You can use the `pg_partition_tree()` function to list the names of all leaf partitions:

```
SELECT * FROM pg_partition_tree('parent_table');
```

Improve statistics quality

There is a trade-off between the time spent generating statistics and the quality or accuracy of those statistics. You need to find a balance that fits your workload.

To analyze large tables within a reasonable time, ANALYZE performs random sampling instead of scanning every row. You can increase the number of sample values collected for all columns in a table by adjusting the `default_statistics_target` configuration parameter. The valid range for this parameter is 1 to 10000, and the default is 100.

By default, `default_statistics_target` applies to all columns and determines how many values are stored in the most-common-values list. A higher value may improve the optimizer's estimation accuracy, especially for columns with skewed data distributions.

You can set `default_statistics_target` at the session level using the `SET` command. To apply it globally, set it in the `postgresql.conf` file and reload the configuration.

When to run ANALYZE

It is recommended to run ANALYZE in the following situations:

- After loading data
- After executing a `CREATE INDEX` command
- After `INSERT`, `UPDATE`, or `DELETE` operations that significantly change the data

ANALYZE only requires a read lock on the table, so it can run in parallel with other database operations. However, for performance reasons, it is not recommended to run ANALYZE at the same time as `INSERT`, `UPDATE`, `DELETE`, `CREATE INDEX`, or data loading operations.

Note

SynxDB improves the behavior of ANALYZE on partitioned tables. When you explicitly run statistics collection on a leaf partition (e.g., `ANALYZE sales_1_prt_p2023`), the system no longer updates statistics for the root or other partitions. Only when ANALYZE is run on

the root table (e.g., `ANALYZE sales`), will statistics for the entire table, including all child partitions, be refreshed.

This change gives you finer control over statistics maintenance and avoids unnecessary updates. In practice, it's recommended to selectively analyze specific partitions or the entire table based on data change patterns.

Configure automatic statistics collection

The configuration parameters `gp_autostats_mode` and `gp_autostats_on_change_threshold` determine when automatic statistics collection is triggered. When triggered, the optimizer will include an `ANALYZE` step during query execution.

By default, `gp_autostats_mode` is set to `none`. If you set it to `on_no_stats`, statistics will be automatically collected when the table owner performs `CREATE TABLE AS SELECT`, `INSERT`, or `COPY` operations on a table that currently has no statistics.

If `gp_autostats_mode` is set to `on_change`, statistics will only be collected when the number of affected rows exceeds the threshold defined by `gp_autostats_on_change_threshold`. The default threshold is 2147483647. In this mode, when the table owner performs `CREATE TABLE AS SELECT`, `UPDATE`, `DELETE`, `INSERT`, or `COPY`, and the affected row count exceeds the threshold, automatic statistics collection will be triggered.

Additionally, if the server parameter `gp_autostats_allow_nonowner` is set to `true`, SynxDB will collect statistics even when a non-owner user is the first to perform an `INSERT` or `COPY` operation, provided that:

- `gp_autostats_mode` is set to `on_no_stats`.

Setting `gp_autostats_mode` to `none` disables automatic statistics collection entirely.

For partitioned tables, inserting data into the top-level parent table does not trigger automatic statistics collection. However, inserting directly into a leaf partition (which physically stores the data) does trigger automatic statistics collection.

Query Plan Hints

SynxDB uses two types of query optimizers: the Postgres-based optimizer and GPORCA. Each optimizer is tuned for specific types of workloads:

- Postgres-based optimizer: suitable for transactional workloads.
- GPORCA: suitable for analytical and hybrid transactional-analytical workloads.

When processing a query, the optimizer explores a large search space of equivalent execution plans. It estimates the number of rows for each operation using table statistics and a cardinality estimation model. Based on this, the optimizer assigns a cost to each plan and selects the one with the lowest cost as the final execution plan.

Query plan hints or optimizer hints are directives that users provide to influence the optimizer's execution strategy. These hints allow users to override default optimizer behavior to address issues such as inaccurate row estimates, suboptimal scan methods, inappropriate join types, or inefficient join orders. This document describes the different types of hints and their applicable scenarios.

i Note

SynxDB currently does not support hints for controlling Motion operators.

Quick example

i Note

To enable query plan hints, you must first load the relevant module in the psql session:

```
LOAD 'pg_hint_plan';
```

You can also configure the database or user to load the module automatically:

```
ALTER DATABASE a_database SET session_preload_libraries='pg_hint_plan';
ALTER USER a_user SET session_preload_libraries='pg_hint_plan';

CREATE TABLE foo(a int);
CREATE INDEX ON foo(a);
INSERT INTO foo SELECT i FROM generate_series(1, 100000)i;
```

```

LOAD 'pg_hint_plan';
SHOW pg_hint_plan.enable_hint;
pg_hint_plan.enable_hint
-----
on
(1 row)

EXPLAIN SELECT count(*) FROM foo WHERE a > 6;
          QUERY PLAN
-----
Finalize Aggregate (cost=537.05..537.06 rows=1 width=8)
  -> Gather Motion 3:1 (slice1; segments: 3) (cost=536.99..537.04
rows=3 width=8)
    -> Partial Aggregate (cost=536.99..537.00 rows=1 width=8)
      -> Seq Scan on foo (cost=0.00..453.67 rows=33330 width=0)
          Filter: (a > 6)
Optimizer: Postgres-based planner
(6 rows)

/*+ IndexScan(foo foo_a_idx) */
EXPLAIN SELECT count(*) FROM foo WHERE a > 6;
          QUERY PLAN
-----
Finalize Aggregate (cost=809.00..809.01 rows=1 width=8)
  -> Gather Motion 3:1 (slice1; segments: 3) (cost=808.94..808.99
rows=3 width=8)
    -> Partial Aggregate (cost=808.94..808.95 rows=1 width=8)
      -> Index Scan using foo_a_idx on foo (cost=0.17..725.61
rows=33330 width=0)
          Index Cond: (a > 6)
Optimizer: Postgres-based planner
(6 rows)

```

You can also specify multiple hints at the same time, such as controlling scan methods and row estimates:

```
/** IndexScan(t1 my_index) Rows(t1 t2 #1000) */
SELECT * FROM t1 JOIN t2 ON t1.a = t2.a WHERE t1.a < 100;
```

Cardinality hints

When the optimizer inaccurately estimates the number of rows for join operations, it may choose inefficient plans—such as using Broadcast instead of Redistribute, or preferring Merge Join over Hash Join incorrectly. Cardinality hints allow you to adjust the estimated number of rows for specific operations. This is especially useful when statistics are missing or outdated.

Examples:

```
/** Rows(t1 t2 t3 #42) */ SELECT * FROM t1, t2, t3; -- set row estimate to 42
/** Rows(t1 t2 t3 +42) */ SELECT * FROM t1, t2, t3; -- increase original
estimate by 42
/** Rows(t1 t2 t3 -42) */ SELECT * FROM t1, t2, t3; -- decrease original
estimate by 42
/** Rows(t1 t2 t3 *42) */ SELECT * FROM t1, t2, t3; -- multiply original
estimate by 42
```

Note

Cardinality hints are currently only effective with the ORCA optimizer. The Postgres optimizer does not recognize them.

Table access hints

Due to inaccurate statistics or biased cost estimation, the optimizer might choose suboptimal scan strategies. Compared to global configuration parameters (GUCs), table access hints offer finer-grained control over how each table is scanned in a query. You can choose whether to use an index or force a specific index.

Examples:

```
/** SeqScan(t1) */ SELECT * FROM t1 WHERE t1.a > 42; -- force sequential scan
/** IndexScan(t1 my_index) */ SELECT * FROM t1 WHERE t1.a > 42; -- force
```

(continues on next page)

(continued from previous page)

```
index scan
/*+ IndexOnlyScan(t1) */ SELECT * FROM t1 WHERE t1.a > 42; -- force index-
only scan
/*+ BitmapScan(t1 my_bitmap_index) */ SELECT * FROM t1 WHERE t1.a > 42; -- 
force bitmap index scan
```

Note

Starting from v4.0.0, the ORCA optimizer supports scan method hints such as `IndexScan` and `SeqScan`, and generates plans accordingly. This feature depends on both the ORCA optimizer and the `pg_hint_plan` extension.

Join type hints

When using a Hash Join, some intermediate results may be spilled to disk, which can affect performance. If a user knows that a specific query would benefit from a Nested Loop Join, they can explicitly specify the join type and the order of inner and outer tables using hints.

Examples:

```
/*+ HashJoin(t1 t2) */ SELECT * FROM t1, t2;
/*+ NestLoop(t1 t2) */ SELECT * FROM t1, t2;
/*+ MergeJoin(t1 t2) */ SELECT * FROM t1 FULL JOIN t2 ON t1.a = t2.a;
```

Join order hints

When the optimizer chooses a suboptimal join order due to missing statistics or inaccurate estimates, you can use the `Leading(...)` hint to specify the join order between tables.

Examples:

```
/*+ Leading(t1 t2 t3) */ SELECT * FROM t1, t2, t3;
/*+ Leading(t1 (t3 t2)) */ SELECT * FROM t1, t2, t3;
```

You can also use `Leading(...)` to control join order in queries involving LEFT OUTER JOIN or RIGHT OUTER JOIN. The following constraints apply:

- The join order in the hint must match the join structure in the original SQL. For example:

```
SELECT * FROM t1 LEFT JOIN t2 ON t1.a = t2.a;
```

The hint `/*+ Leading((t1 t2)) */` preserves the left join. The hint `/*+ Leading((t2 t1)) */` transforms it into a right join (semantically equivalent but generates a different plan).

- For nested outer joins, you must specify the hint in the same semantic nesting order;
- Changing join direction is not supported for non-equality join conditions (e.g. `t1.a > t2.a`), as it would break the query semantics.

Example: The following hint instructs the optimizer to first join `t3` with the result of `t2` joined to `t1`:

```
/*+ Leading((t3 (t2 t1))) */
SELECT * FROM t1 LEFT JOIN t2 ON t1.a = t2.a LEFT JOIN t3 ON t2.b = t3.b;
```

Supported scope and limitations

- Query plan hints rely on the `pg_hint_plan` extension, which must be explicitly loaded.
- Hints for controlling data redistribution strategies are not yet supported.

Best practices for using query plan hints

When using hints, follow these best practices:

- Focus on solving specific issues, such as inaccurate row estimates, suboptimal scan methods, or inefficient join types or join orders.
- Test thoroughly before deployment to ensure that the hint improves performance and reduces resource usage.
- Use hints as a temporary measure. They are intended for short-term tuning and should be reviewed and adjusted as data changes.
- Avoid conflicts with GUC settings. If a GUC disables a feature (such as `IndexScan`) that a hint tries to enable, the hint will be ignored. Make sure global settings align with your hints.

Note

pg_hint_plan allows you to include GUC settings directly in hints. For example:

```
/*+ SeqScan(mytable) optimizer_enable_seqscan=on */ /*Set(enable_indexscan
off) */
EXPLAIN (COSTS false) SELECT * FROM t1, t2 WHERE t1.id = t2.id;
```

Use GPORCA Optimizer

GPORCA overview

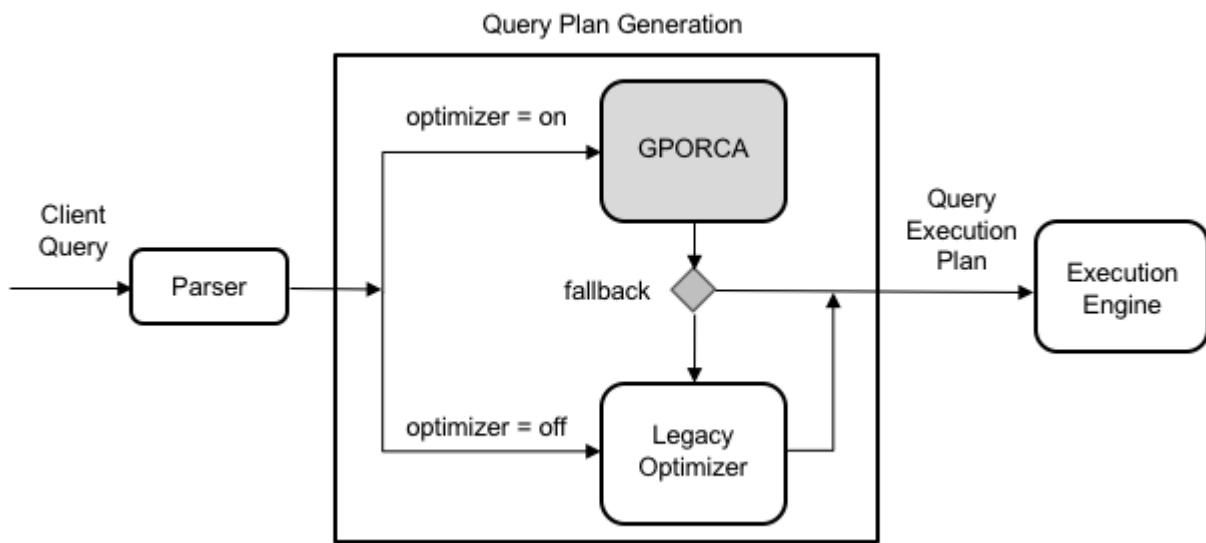
GPORCA is an enhanced query optimizer built on top of the PostgreSQL planner, designed to improve query planning and optimization. It offers high scalability and delivers more efficient optimization, especially on multi-core architectures. SynxDB uses GPORCA by default to generate query execution plans when supported.

GPORCA significantly enhances query performance in the following areas:

- Queries on partitioned tables;
- Queries with common table expressions (CTEs);
- Queries with subqueries.

In SynxDB, GPORCA and the PostgreSQL-based planner coexist. By default, the system first attempts to use GPORCA. If GPORCA does not support a particular query, the system automatically falls back to the PostgreSQL optimizer.

The following diagram illustrates GPORCA's role in the overall query planning architecture:



Note that all server parameters used to configure PostgreSQL planner behavior are ignored when GPORCA is enabled. These parameters only take effect if the system falls back to using the PostgreSQL optimizer.

Enable or disable GPORCA

You can enable or disable GPORCA using the `optimizer` server configuration parameter.

Although GPORCA is enabled by default, you can configure the `optimizer` parameter at the system, database, session, or query level to control whether GPORCA is used.

i Note

- The `optimizer` parameter enables or disables GPORCA.
- The `optimizer_control` parameter determines whether changes to `optimizer` are allowed. If `optimizer_control` is set to `off`, any attempt to modify `optimizer` will result in an error. To allow changes, set `optimizer_control` to `on`.

Enable GPORCA system-wide

You can enable GPORCA across the entire system by setting the `optimizer` server parameter.

1. Log in to the primary node as the SynxDB administrator user `gpadmin`.
2. Run the following `gpconfig` command to set the parameter to `on`:

```
gpconfig -c optimizer -v on --coordinatoronly
```

3. Reload the configuration to apply changes without restarting the system:

```
gpstop -u
```

Enable GPORCA for a database

You can enable GPORCA for a specific database using the `ALTER DATABASE` command. The following example enables GPORCA for the `test_db` database:

```
ALTER DATABASE test_db SET optimizer = on;
```

Enable GPORCA for a session or query

You can enable GPORCA in the current session using the `SET` command. For example, after connecting to SynxDB via `psql`, run:

```
SET optimizer = on;
```

To enable GPORCA for a single query only, run the above `SET` command just before executing the query.

Determine which query optimizer is used

When GPORCA is enabled (which is the default), you can determine whether SynxDB is using GPORCA or has fallen back to the PostgreSQL planner by inspecting the EXPLAIN output.

The most straightforward method is to look at the end of the query plan:

- The optimizer used is indicated at the end of the plan. For example:

- If the plan is generated by GPORCA, it shows:

```
Optimizer: GPORCA
```

- If the plan is generated by the PostgreSQL planner, it shows:

```
Optimizer: Postgres-based planner
```

- If the plan includes nodes such as `Dynamic <any> Scan` (e.g., `Dynamic Assert`, `Dynamic Sequence`, `Dynamic Index Scan`), it was generated by GPORCA. The PostgreSQL planner does not produce these node types.
- For partitioned table queries, GPORCA’s EXPLAIN output only displays the number of pruned partitions, without listing them individually. In contrast, the PostgreSQL planner lists every scanned partition.

In addition to the EXPLAIN output, the optimizer type is also recorded in the logs. If GPORCA cannot support a query and falls back to the PostgreSQL planner, the system logs a NOTICE with an explanation.

You can also enable the `optimizer_trace_fallback` parameter to display detailed fallback reasons directly in `psql`.

Note

Setting the server parameter `optimizer_trace_fallback` to `on` allows you to view detailed fallback reasons in the command-line interface when GPORCA falls back.

Example

The following example demonstrates the behavior of a query on a partitioned table when GPORCA is enabled.

The CREATE TABLE statement below creates a range-partitioned table based on the date column:

```
CREATE TABLE sales (trans_id int, date date,
    amount decimal(9,2), region text)
DISTRIBUTED BY (trans_id)
PARTITION BY RANGE (date)
    (START (date '2016-01-01')
        INCLUSIVE END (date '2017-01-01')
        EXCLUSIVE EVERY (INTERVAL '1 month'),
    DEFAULT PARTITION outlying_dates );
```

The plan generated by GPORCA only shows the number of selected partitions, without listing their names:

```
-> Partition Selector for sales (dynamic scan id: 1) (cost=10.00..100.00)
rows=50 width=4)
Partitions selected: 13 (out of 13)
```

If a particular partitioned table query is not supported by GPORCA, the system automatically falls back to the PostgreSQL optimizer. In such cases, the EXPLAIN output lists all accessed partitions. For example:

```
-> Append  (cost=0.00..0.00 rows=26 width=53)
   -> Seq Scan on sales2_1_prt_7_2_prt_usa sales2  (cost=0.00..0.00 rows=1
width=53)
      -> Seq Scan on sales2_1_prt_7_2_prt_asia sales2  (cost=0.00..0.00 rows=1
width=53)
         ...

```

The following example shows the log output when a query falls back to the PostgreSQL optimizer:

Run the following query:

```
EXPLAIN SELECT * FROM pg_class;
```

The system uses the PostgreSQL planner and logs a NOTICE message indicating why GPORCA did not handle the query:

```
INFO: GPORCA failed to produce a plan, falling back to Postgres-based planner
DETAIL: Falling back to Postgres-based planner because GPORCA does not
support the following feature: Non-default collation
```

GPORCA Features and Enhancements

GPORCA provides enhanced support for certain types of queries and operations:

- Queries on partitioned tables
- Queries with subqueries
- Queries with common table expressions (CTEs)
- Optimized DML operations

Enhancements for partitioned table queries

GPORCA applies the following optimizations when handling queries on partitioned tables:

- Improved partition pruning capabilities.
- Query plans can include the Partition Selector operator.
- EXPLAIN plans no longer enumerate all partitions.

For queries using static partition pruning (for example, comparing partition keys with constants), GPORCA displays a Partition Selector operator in the EXPLAIN output, including the filter condition and the number of selected partitions. Example:

```
Partition Selector for Part_Table (dynamic scan id: 1)
  Filter: a > 10
  Partitions selected: 1 (out of 3)
```

For dynamic partition pruning (for example, comparing partition keys with variables), the partitions are determined at execution time, and the EXPLAIN output does not list the selected partitions.

- The size of the query plan does not grow with the number of partitions.
- Significantly reduces the risk of out-of-memory errors caused by a large number of partitions.

The following CREATE TABLE example creates a range-partitioned table:

```
CREATE TABLE sales(order_id int, item_id int, amount numeric(15,2),
    date date, yr_qtr int)
    PARTITION BY RANGE (yr_qtr) (start (201501) INCLUSIVE end (201504)
INCLUSIVE,
    start (201601) INCLUSIVE end (201604) INCLUSIVE,
    start (201701) INCLUSIVE end (201704) INCLUSIVE,
    start (201801) INCLUSIVE end (201804) INCLUSIVE,
    start (201901) INCLUSIVE end (201904) INCLUSIVE,
    start (202001) INCLUSIVE end (202004) INCLUSIVE);
```

GPORCA optimizes the following types of queries on partitioned tables:

- Full table scans: partitions are not enumerated in the plan.

```
SELECT * FROM sales;
```

- Queries with constant filter conditions: partition pruning is applied.

```
SELECT * FROM sales WHERE yr_qtr = 201501;
```

- Range queries: also trigger partition pruning.

```
SELECT * FROM sales WHERE yr_qtr BETWEEN 201601 AND 201704;
```

- Joins on partitioned tables: for example, joining the dimension table date_dim with the fact table catalog_sales.

```
SELECT * FROM catalog_sales
    WHERE date_id IN (SELECT id FROM date_dim WHERE month=12);
```

Subquery optimization

GPORCA processes subqueries more efficiently. A subquery is a query nested within an outer query block, such as the SELECT inside the WHERE clause below:

```
SELECT * FROM part
    WHERE price > (SELECT avg(price) FROM part);
```

GPORCA also optimizes correlated subqueries (CSQs), which reference columns from the outer

query. For example, both the inner and outer queries in the following statement use the price column:

```
SELECT * FROM part p1 WHERE price > (SELECT avg(price) FROM part p2 WHERE p2.brand = p1.brand);
```

GPORCA can generate more efficient plans for the following types of subqueries:

- Correlated subqueries in the SELECT list:

```
SELECT *,  
    (SELECT min(price) FROM part p2 WHERE p1.brand = p2.brand)  
    AS foo  
FROM part p1;
```

- Correlated subqueries inside OR conditions:

```
SELECT FROM part p1 WHERE p_size > 40 OR  
    p_retailprice >  
    (SELECT avg(p_retailprice)  
        FROM part p2  
        WHERE p2.p_brand = p1.p_brand)
```

- Nested subqueries with jump correlations:

```
SELECT * FROM part p1 WHERE p1.p_partkey  
    IN (SELECT p_partkey FROM part p2 WHERE p2.p_retailprice =  
        (SELECT min(p_retailprice)  
            FROM part p3  
            WHERE p3.p_brand = p1.p_brand)  
    );
```

Note

The PostgreSQL planner does not support nested correlated subqueries with jump correlations.

- Correlated subqueries with aggregation and inequality:

```
SELECT * FROM part p1 WHERE p1.p_retailprice =
  (SELECT min(p_retailprice) FROM part p2 WHERE p2.p_brand <> p1.p_brand);
```

- Correlated subqueries expected to return a single row:

```
SELECT p_partkey,
  (SELECT p_retailprice FROM part p2 WHERE p2.p_brand = p1.p_brand )
FROM part p1;
```

Common table expression (CTE) optimization

GPORCA efficiently handles queries with WITH clauses. A WITH clause, also known as a common table expression (CTE), defines a temporary logical table used within the query. The following is an example of a query with a CTE:

```
WITH v AS (SELECT a, sum(b) as s FROM T WHERE c < 10 GROUP BY a)
SELECT * FROM v AS v1, v AS v2
WHERE v1.a <> v2.a AND v1.s < v2.s;
```

As part of query optimization, GPORCA supports predicate pushdown into CTEs. In the example below, equality predicates are pushed into the CTE:

```
WITH v AS (SELECT a, sum(b) as s FROM T GROUP BY a)
SELECT *
  FROM v as v1, v as v2, v as v3
 WHERE v1.a < v2.a
   AND v1.s < v3.s
   AND v1.a = 10
   AND v2.a = 20
   AND v3.a = 30;
```

GPORCA supports the following types of CTEs:

- CTEs that define multiple logical tables. In the example below, the CTE defines two logical tables:

```
WITH cte1 AS (SELECT a, sum(b) as s FROM T
               WHERE c < 10 GROUP BY a),
```

(continues on next page)

(continued from previous page)

```

cte2 AS (SELECT a, s FROM cte1 WHERE s > 1000)
SELECT *
FROM cte1 as v1, cte2 as v2, cte2 as v3
WHERE v1.a < v2.a AND v1.s < v3.s;

```

DML operation optimization

GPORCA also enhances DML operations such as INSERT, UPDATE, and DELETE:

- DML operations appear as regular operator nodes in the execution plan.
 - They can appear at any position in the plan (currently limited to the top-level slice).
 - They can have downstream nodes (consumers).
- UPDATE operations are implemented using the Split operator and support the following:
 - Updates to distribution key columns.
 - Updates to partition key columns. The following example shows a plan containing a Split operator:

```

QUERY PLAN
-----
Update  (cost=0.00..5.46 rows=1 width=1)
  -> Redistribute Motion 2:2  (slice1; segments: 2)
    Hash Key: a
    -> Result  (cost=0.00..3.23 rows=1 width=48)
      -> Split   (cost=0.00..2.13 rows=1 width=40)
        -> Result  (cost=0.00..1.05 rows=1 width=40)
          -> Seq Scan on dmltest

```

Other optimization capabilities

GPORCA also includes the following optimization features:

- Improved join order selection.
- Support for reordering joins and aggregation operations.
- Optimization of sort order.
- Consideration of data skew estimation during optimization.

GPORCA Optimizer Update Notes

This document describes feature enhancements and behavioral changes to the GPORCA optimizer in each version of SynxDB.

v4.0.0

Starting from v4.0.0, the GPORCA optimizer includes the following new features and improvements:

- In addition to previously supported index-only scans on B-tree indexes, GPORCA now supports index-only scans on more index types. Specifically:
 - Index-only scans are now supported on AO tables and PAX tables.
 - PostgreSQL-style `INCLUDE` columns are supported, enabling the creation of covering indexes on AO and PAX tables to improve performance in repeated-read workloads.
- GPORCA can now automatically choose backward index scans based on the sort direction of a query. This applies to both regular and index-only scans. This optimization reduces the need for `Sort` nodes and can improve performance in certain top-N queries.
- Support has been added for pushing down `ScalarArrayOp` predicates (e.g., `col IN (...)` or `col = ANY(array)`) to index paths, including:
 - B-tree or hash indexes;
 - Index Scan or Bitmap Index Scan paths.

The optimizer decides whether to use a Bitmap scan based on cost estimation. In low-selectivity cases, it may still fall back to sequential scan. Note that `ScalarArrayOp` pushdown only applies to the leading column of an index, and not to non-leading columns in composite indexes. Example:

```
CREATE INDEX idx ON t(col1, col2);
SELECT * FROM t WHERE col1 = ANY('{1,2,3}'); -- Pushdown supported
SELECT * FROM t WHERE col2 = ANY('{1,2,3}'); -- No pushdown, used as
filter only
```

- Support for `FULL JOIN` has been added, using the `Hash Full Join` execution strategy.

This implementation does not require sorting of join keys and is well-suited for large datasets, high-cardinality join keys, or mismatched distribution keys.

Merge Full Join is not currently supported, so all FULL JOIN queries use Hash Full Join.

Compared to traditional Merge Join, Hash Full Join offers the following advantages:

- No sorting required on join keys;
- Reduced data movement (Motion) overhead;
- Potentially better performance when join keys are highly skewed or have high cardinality.

Example:

```
EXPLAIN SELECT * FROM t1 FULL JOIN t2 ON t1.id = t2.id;
```

Might produce the following plan:

```
Hash Full Join
Hash Cond: t1.id = t2.id
...
```

- GPORCA introduces a query rewrite rule that pushes JOIN operations below each branch of a UNION ALL. When enabled, the optimizer may rewrite a JOIN over a UNION ALL into multiple smaller joins. This can significantly improve performance by:
 - Converting a large join over a UNION ALL into multiple smaller joins.
 - Allowing each sub-join to use indexes independently, reducing Motion and Hash Join overhead.
 - Pushing the JOIN to either the left or right side of the UNION ALL, enabling more flexible query structures.

This optimization is disabled by default. You can enable it with the following GUC:

```
SET optimizer_enable_push_join_below_union_all = on;
```

The following example shows how the optimizer pushes the JOIN below each UNION ALL

branch when the optimization is enabled:

```
-- Creates test tables
CREATE TABLE dist_small_1(c1 int);
INSERT INTO dist_small_1 SELECT generate_series(1, 1000);
CREATE INDEX dist_small_1_index ON dist_small_1 USING btree (c1);
ANALYZE dist_small_1;

CREATE TABLE dist_small_2(c1 int);
INSERT INTO dist_small_2 SELECT generate_series(1, 1000);
ANALYZE dist_small_2;

CREATE TABLE inner_1(cc int);
INSERT INTO inner_1 VALUES(1);
ANALYZE inner_1;

-- Creates a view
CREATE VIEW dist_view_small AS
SELECT c1 FROM dist_small_1
UNION ALL
SELECT c1 FROM dist_small_2;

-- Enables the optimization and run the query
SET optimizer_enable_push_join_below_union_all = on;
EXPLAIN ANALYZE
SELECT c1 FROM dist_view_small JOIN inner_1 ON c1 < cc;
```

The optimizer might produce a plan like the following:

```
-> Append
    -> Nested Loop
        ...
        -> Index Scan using dist_small_1_index on dist_small_1
    -> Nested Loop
        ...
        -> Seq Scan on dist_small_2
```

This optimization is especially useful for the following types of queries:

- A UNION ALL over large tables joined with a small table.
- Each branch has indexes that the optimizer can use.

- The JOIN is applied to a view or subquery that contains a UNION ALL.

i Note

- This optimization does not support FULL JOIN or Common Table Expressions (CTEs).
- It also does not support JOIN of UNION ALL or UNION ALL of JOIN structures.

- By default, GPORCA assigns a higher cost to broadcast paths (Broadcast Motion) using the optimizer_penalize_broadcast_threshold GUC parameter, to avoid choosing expensive plans when data volumes are large.

Starting from v4.0.0, for NOT IN queries (for example, Left Anti Semi Join, LASJ), broadcast paths are no longer penalized. This prevents the optimizer from concentrating large tables on the coordinator, which can otherwise lead to severe performance issues or even out-of-memory (OOM) errors.

Allowing broadcast paths helps preserve parallel execution and significantly improves the performance of NOT IN queries on large datasets.

Feature details:

- Applies only to NOT IN queries (LASJ).
- Ignores the setting of optimizer_penalize_broadcast_threshold.
- The penalty strategy remains in place for other types of joins (such as IN or EXISTS).

Example:

```
SELECT * FROM foo WHERE a NOT IN (SELECT a FROM bar);
```

Sample query plan:

```
Gather Motion 2:1
-> Hash Left Anti Semi (Not-In) Join
    -> Seq Scan on foo
    -> Broadcast Motion
```

(continues on next page)

(continued from previous page)

```
-> Seq Scan on bar
```

- GPORCA now supports index-only scans inside common table expressions (CTEs). In the example below, the CTE query can trigger an index-only scan:

```
CREATE TABLE t(a int, b int);
CREATE INDEX i ON t(a);
INSERT INTO t SELECT i, i+i FROM generate_series(1, 10)i;
VACUUM ANALYZE t;

EXPLAIN WITH cte AS (SELECT a FROM t WHERE a > 42) SELECT * FROM cte;
```

Optimization for self-joins with multiple outer joins

Starting from v4.0.0, GPORCA can identify specific patterns involving multiple outer joins and skip unnecessary Redistribute Motion operations to improve execution efficiency:

- The query contains multiple LEFT OUTER JOIN or RIGHT OUTER JOIN operations.
- All joined tables are aliases of the same base table.
- The join conditions are symmetric (e.g., `t1.a = t2.a`).
- All tables use the same distribution key and satisfy locality constraints.

Example:

```
CREATE TABLE o1 (a1 int, b1 int) DISTRIBUTED BY (a1);

EXPLAIN (COSTS OFF)
SELECT * FROM (SELECT DISTINCT a1 FROM o1) t1
    LEFT OUTER JOIN o1 t2 ON t1.a1 = t2.a1
    LEFT OUTER JOIN o1 t3 ON t2.a1 = t3.a1;
```

In earlier versions, this query would insert Redistribute Motion between each join level. Starting from v4.0.0, GPORCA can detect this multi-level self-join pattern and avoid unnecessary data redistribution, improving overall query performance.

Use Automatic Materialized Views for Query Optimization

SynxDB supports automatically using *materialized views* to process some or all queries (called “Answer Query Using Materialized Views” , or “AQUMV”) during the query planning phase. This feature is suitable for queries on large tables and can greatly reduce query processing time. AQUMV uses incremental materialized views (IMVs) because IMVs usually keep the latest data when related tables have write operations.

Usage scenarios

- Aggregation queries on large data sets: For queries that need to aggregate results from millions of records, AQUMV can significantly reduce query time.
- Frequently updated large tables: In an environment where data is frequently updated, using IMV can ensure that the query results are real-time and accurate.
- Complex calculations: For queries with complex calculations (such as square root and absolute value calculations), AQUMV can speed up the queries by pre-calculating these values in materialized views.

Implementation

AQUMV implements query optimization by equivalently transforming the query tree.

SynxDB automatically uses materialized views only when the table query meets the following conditions:

- Materialized views must contain all the rows required by the query expression.
- If the materialized view contains more rows than the query, you might need to add additional filters.
- All output expressions must be able to be calculated from the output of the view.
- The output expression can fully or partially match the target list of the materialized view.

When there are multiple valid materialized view candidates, or the cost of querying from the materialized view is higher than querying directly from the original table (for example, the original table has indexes), you can let the planner make the best choice based on cost estimates.

Comparison with dynamic tables

AQUMV and *dynamic tables* have these differences:

Feature	Dynamic table	AQUMV
Purpose	A special table that automatically refreshes, processes data pipelines, and simplifies ETL.	Uses materialized views to improve query efficiency and automatically rewrite queries.
Base and structure	Can be based on ordinary tables, external tables, materialized views.	Based on materialized views, usually targeting a single table.
Query rewrite	Not supported	Supports single-table rewrite
Data refresh mechanism	Users can define automatic refresh interval through SQL	Requires manual refresh of materialized view data

Restrictions

- Only SELECT queries for a single relationship are supported, which is applicable to materialized view queries and the original queries.
- You need to explicitly enable the support for external tables through the `aqumv_allow_foreign_table` parameter.
- The support for aggregate queries have the following limitations: - Aggregates with GROUP BY are not supported. - The HAVING clause must be calculable from the target list of the materialized view.
- Currently, unsupported features include: subqueries, sorting of the original query (ORDER BY), joins (JOIN), sublinks (SUBLINK), grouping (GROUP BY), window functions, common table expressions (CTE), distinct on (DISTINCT ON), refreshing materialized views (REFRESH MATERIALIZED VIEW), and CREATE AS statements.

Usage examples

To enable AQUMV, you need to create a materialized view, turn off the GPORCA optimizer, and set the value of the system parameter `enable_answer_query_using_materialized_views` to ON. The following example compares the results of the same complex query without AQUMV and with AQUMV.

1. Turn off the GPORCA planner to use the Postgres-based planner.

```
SET optimizer TO off;
```

2. Create a table aqumv_t1.

```
CREATE TABLE aqumv_t1(c1 INT, c2 INT, c3 INT) DISTRIBUTED BY (c1);
```

3. Insert data into the table and collect statistics from the table.

```
INSERT INTO aqumv_t1 SELECT i, i+1, i+2 FROM generate_series(1, 1000000)
i;
ANALYZE aqumv_t1;
```

4. Execute a query without enabling AQUMV. The query takes 7384.329 ms.

```
SELECT SQRT(ABS(ABS(c2) - c1 - 1) + ABS(c2)) FROM aqumv_t1 WHERE c1 > 30
AND c1 < 40 AND SQRT(ABS(c2)) > 5.8;

sqrt
-----
66.08276253029821
96.24499799839839
85.91607978309961
66.16441400296897
66.32455532033675
95.8309518948453
(7 rows)

Time: 7384.329 ms (00:07.384)
```

From the following query plan, you can see that the optimizer scans the table (Seq Scan on aqumv_t1).

```
EXPLAIN(COSTS OFF) SELECT SQRT(ABS(ABS(c2) - c1 - 1) + ABS(c2)) FROM
aqumv_t1 WHERE c1 > 30 AND c1 < 40 AND SQRT(ABS(c2)) > 5.8;

QUERY PLAN
-----
Gather Motion 3:1 (slice1; segments: 3)
-> Seq Scan on aqumv_t1
```

(continues on next page)

(continued from previous page)

```

Filter: ((c1 > 30) AND (c1 < 40) AND (sqrt((abs(c2))::double
precision) > '5.8'::double precision))
Optimizer: Postgres query optimizer
(4 rows)

```

5. Create a materialized view mvt1 based on aqumv_t1 and collect statistics on the view.

```

CREATE INCREMENTAL MATERIALIZED VIEW mvt1 AS SELECT c1 AS mc1, c2 AS mc2,
ABS(c2) AS mc3, ABS(ABS(c2) - c1 - 1) AS mc4
FROM aqumv_t1 WHERE c1 > 30 AND c1 < 40;

ANALYZE mvt1;

```

6. Enable the AQUMV-related configuration parameter.

```
SET enable_answer_query_using_materialized_views = ON;
```

7. Now AQUMV is enabled. Execute the same query again, which takes only 45.701 ms.

```

SELECT SQRT(ABS(ABS(c2) - c1 - 1) + ABS(c2)) FROM aqumv_t1 WHERE c1 > 30
AND c1 < 40 AND SQRT(ABS(c2)) > 5.8;

sqrt
-----
66.08276253029821
96.24499799839839
85.8309518948453
5.916079783099616
66.16441400296897
66.32455532033675
(7 rows)

Time: 45.701 ms

```

From the following query plan, you can see that the optimizer does not scan the aqumv_t1 table, but scans the materialized view mvt1 (Seq Scan on public.mvt1) instead.

```

EXPLAIN(VERBOSE, COSTS OFF) SELECT SQRT(ABS(ABS(c2) - c1 - 1) + ABS(c2))
FROM aqumv_t1 WHERE c1 > 30 AND c1 < 40 AND SQRT(ABS(c2)) > 5.8;

```

(continues on next page)

(continued from previous page)

```

QUERY PLAN
-----
Gather Motion 3:1  (slice1; segments: 3)
  Output: (sqrt(((mc4 + mc3))::double precision))
-> Seq Scan on public.mvt1
    Output: sqrt(((mc4 + mc3))::double precision)
    Filter: (sqrt((mvt1.mc3)::double precision) > '5.8'::double
precision)
Settings: enable_answer_query_using_materialized_views = 'on', optimizer =
'off'
Optimizer: Postgres query optimizer
(7 rows)

```

In the above example, the query takes 7384.329 ms without AQUMV and without using the materialized views. In contrast, the same query takes only 45.701 ms with AQUMV enabled and using the materialized view. This means that the materialized view pre-calculates and stores relevant calculation result, so that the view only contains rows that meet the specific condition ($c1 > 30$ AND $c1 < 40$).

Therefore, the above table query `SELECT SQRT(ABS(ABS(c2) - c1 - 1) + ABS(c2)) FROM aqumv_t1 WHERE c1 > 30 AND c1 < 40 AND SQRT(ABS(c2)) > 5.8;` is equivalent to querying from the materialized view `SELECT SQRT(ABS(ABS(c2) - c1 - 1) + ABS(c2)) FROM aqumv_t1 WHERE c1 > 30 AND c1 < 40 AND SQRT(ABS(c2)) > 5.8;.`

When the same query is executed, the data can be obtained directly from the materialized view, rather than from the original table. In this way, AQUMV can significantly improve query performance, especially when dealing with large data volumes and complex calculations.

Use materialized views to query external tables

SynxDB supports querying external tables using materialized views. In general, querying external tables faces challenges like data synchronization latency and network bottlenecks. Materialized views can cache query results to avoid accessing external data sources every time, which improves query speed.

Here is an example:

1. Create an external table and its corresponding materialized view.

```
CREATE READABLE EXTERNAL TABLE aqumv_ext_r(id int)
LOCATION ('demoprot://aqumvtextfile.txt')
FORMAT 'text';

CREATE MATERIALIZED VIEW aqumv_ext_mv AS
SELECT * FROM aqumv_ext_r;
```

In the above external table creation statement, demoprot://aqumvtextfile.txt is an example file path and example protocol.

2. Enable the AQUMV support in a transaction for the external table:

```
BEGIN;
SET optimizer = OFF;
SET aqumv_allow_foreign_table = ON;
SET enable_answer_query_using_materialized_views = ON;
```

3. Check the query plan to see that the following query uses the materialized view instead of directly scanning the external table:

```
EXPLAIN (COSTS OFF, VERBOSE)
SELECT * FROM aqumv_ext_r;
                                         QUERY PLAN
-----
Gather Motion 3:1  (slice1; segments: 3)
  Output: id
    -> Seq Scan on aqumv.aqumv_ext_mv
      Output: id
Optimizer: Postgres query optimizer
```

4. Create an index on the materialized view to further optimize queries:

```
CREATE INDEX ON aqumv_ext_mv(id);
```

5. When the conditional query is executed, the query plan shows the use of the index:

```
EXPLAIN (COSTS OFF, VERBOSE)
SELECT * FROM aqumv_ext_r WHERE id = 5;
                                         QUERY PLAN
-----
-- 
Gather Motion 1:1  (slice1; segments: 1)
  Output: id
    -> Index Only Scan using aqumv_ext_mv_id_idx on aqumv.aqumv_ext_mv
      Output: id
      Index Cond: (aqumv_ext_mv.id = 5)
Optimizer: Postgres query optimizer
```

Commit the transaction:

```
COMMIT;
```

When this feature is enabled, queries on external tables automatically uses materialized views, including leveraging indexes on materialized views to accelerate queries.

Note

- Because external table data might change outside the database and the planner cannot detect these changes, AQUMV is disabled for external tables by default.
- It is recommended to enable `aqumv_allow_foreign_table` only when you are certain that external table data has not changed.
- For some external tables (like Iceberg and Hudi), access speed can be 10 times slower or more than internal tables. Using materialized views in these cases can greatly improve query performance.
- Compared to importing external table data into internal tables, using materialized views avoids data loading latency and time overhead.

Aggregate query support

SynxDB supports optimizing queries that contain aggregate functions using materialized views. This feature can greatly improve query performance when dealing with aggregate operations on large datasets.

Aggregate query support

1. Create a table t.

```
CREATE TABLE t(c1 INT, c2 INT, c3 INT) DISTRIBUTED BY (c1);
```

2. Insert data into the table and collect statistics.

```
INSERT INTO t SELECT i, i+1, i+2 FROM generate_series(1, 1000000) i;
ANALYZE t;
```

3. Create a materialized view with aggregate functions.

```
CREATE MATERIALIZED VIEW mv AS
SELECT sum(c1) AS mc1, count(c2) AS mc2,
       avg(c3) AS mc3, count(*) AS mc4
FROM t
WHERE c1 > 90;
```

4. Query transformation example.

Original query:

```
SELECT count(*), sum(c1), count(c2), avg(c3),
       abs(count(*) - 21)
FROM t
WHERE c1 > 90;
```

AQUMV will automatically rewrite it as:

```
SELECT mc4, mc1, mc2, mc3, abs((mc4 - 21))
FROM mv;
```

Seq Scan on mv in the query plan shows that the optimizer directly uses the materialized view:

```
EXPLAIN (VERBOSE, COSTS OFF)
SELECT count(*), sum(c1), count(c2), avg(c3), abs(count(*) - 21)
FROM t
WHERE c1 > 90;
```

QUERY PLAN

```
Gather Motion 3:1 (slice1; segments: 3)
  Output: mc4, mc1, mc2, mc3, (abs((mc4 - 21)))
  -> Seq Scan on mv
    Output: mc4, mc1, mc2, mc3, abs((mc4 - 21))
Settings: enable_answer_query_using_materialized_views = 'on', optimizer = 'off'
Optimizer: Postgres query optimizer
(6 rows)
```

HAVING clause processing

AQUMV supports queries with HAVING clauses. If the conditions in the HAVING clause can be calculated from the target list of the materialized view, these conditions will be converted to WHERE clauses.

Example:

1. Create a table and materialized view.

```
CREATE TABLE t(c1 int, c2 int, c3 int, c4 int);
INSERT INTO t SELECT i, i+1, i+2, i+3 FROM generate_series(1, 1000000) i;
ANALYZE t;
CREATE MATERIALIZED VIEW mv AS
SELECT sum(c1) AS mc1, count(c2) AS mc2,
       avg(c3) AS mc3, count(*) AS mc4
FROM t
WHERE c1 > 90;
```

2. Query transformation with HAVING clause.

Original query:

```
SELECT count(*), sum(c1)
FROM t
WHERE c1 > 90
HAVING abs(count(*) - 21) > 0 AND 2 > 1 AND avg(c3) > 97;
```

AQUMV will automatically rewrite it as:

```
SELECT mc4, mc1
FROM mv
WHERE mc3 > 97 AND abs(mc4 - 21) > 0;
```

The query plan shows that HAVING conditions are converted to filter conditions in the WHERE clause:

```
EXPLAIN (VERBOSE, COSTS OFF)
SELECT count(*), sum(c1)
FROM t
WHERE c1 > 90
HAVING abs(count(*) - 21) > 0 AND 2 > 1 AND avg(c3) > 97;
                                         QUERY PLAN
-----
Gather Motion 3:1 (slice1; segments: 3)
  Output: mc4, mc1
    -> Seq Scan on aqumv.mv
      Output: mc4, mc1
      Filter: ((mv.mc3 > '97'::numeric) AND (abs((mv.mc4 - 21)) > 0))
Optimizer: Postgres query optimizer
(7 rows)
```

Note:

- The condition $2 > 1$ in the HAVING clause is optimized out during planning.
- $\text{abs}(\text{count}(\text{*}) - 21) > 0$ is converted to $\text{abs}(\text{mc4} - 21) > 0$ and used as a filter condition.

LIMIT and ORDER BY Support

Because only aggregate queries without GROUP BY are currently supported, aggregate results will have at most one row, therefore:

- ORDER BY clauses do not affect the results in this case.
- LIMIT and OFFSET clauses with constant values can be directly applied to materialized view queries.

Example:

```
-- Original query
SELECT count(*), sum(c1)
FROM t
WHERE c1 > 90
LIMIT 2;

-- Rewritten as
SELECT mc4, mc1
FROM mv
LIMIT 2;
```

Notes for aggregate query support

- Currently, only aggregate queries without GROUP BY clauses are supported.
- Conditions in HAVING clauses must be calculable from the target list of the materialized view.
- For aggregate queries, the result will be either one row or zero rows.

5.2 Manage Resources Using Resource Groups

You can use resource groups to manage and protect the resource allocation of CPU, memory, concurrent transaction limits, and disk I/O in SynxDB. Once you define a resource group, you assign the group to one or more SynxDB roles, or to an external component such as PL/Container, in order to control the resources used by them.

When you assign a resource group to a role, the resource limits that you define for the group apply to all of the roles to which you assign the group. For example, the memory limit for a resource group identifies the maximum memory usage for all running transactions submitted by SynxDB users in all roles to which you assign the group.

SynxDB uses Linux-based control groups for CPU resource management, and Runaway Detector for statistics, tracking and management of memory.

Role and component resource groups

SynxDB supports two types of resource groups: groups that manage resources for roles, and groups that manage resources for external components such as PL/Container.

The most common application for resource groups is to manage the number of active queries that different roles might run concurrently in your SynxDB cluster. You can also manage the amount of CPU, memory resources, and disk I/O that SynxDB allocates to each query.

When a user runs a query, SynxDB evaluates the query against a set of limits defined for the resource group. SynxDB runs the query immediately if the group's resource limits have not yet been reached and the query does not cause the group to exceed the concurrent transaction limit. If these conditions are not met, SynxDB queues the query. For example, if the maximum number of concurrent transactions for the resource group has already been reached, a subsequent query is queued and must wait until other queries complete before it runs. SynxDB might also run a pending query when the resource group's concurrency and memory limits are altered to large enough values.

Within a resource group for roles, transactions are evaluated on a first in, first out basis. SynxDB periodically assesses the active workload of the system, reallocating resources and starting/queuing jobs as necessary.

You can also use resource groups to manage the CPU and memory resources of external

components such as PL/Container. Resource groups for external components use Linux cgroups to manage the total CPU resources for the component.

Resource group attributes and limits

When you create a resource group, you provide a set of limits that determine the amount of CPU and memory resources available to the group. The following table lists the available limits for resource groups:

Limit Type	Description	Value Range	Default
CONCURR	The maximum number of concurrent transactions, including active and idle transactions, that are permitted in the resource group.	[0 - max_connecti	20
CPU_MAX	The maximum percentage of CPU resources the group can use.	[1 - 100]	-1 (not set)
CPU_WEIGHT	The scheduling priority of the resource group.	[1 - 500]	100
CPUSET	The specific CPU logical core (or logical thread in hyperthreading) reserved for this resource group.	It depends on system core configuration	-1
IO_LIMIT	The limit for the maximum read/write disk I/O throughput, and maximum read/write I/O operations per second. Set the value on a per-tablespace basis.	[2 - 4294967295 or max]	-1
MEMORY_LIMIT	The memory limit value specified for the resource group.	Integer (MB)	-1 (not set, use statement_mem as the memory limit for a single query)
MIN_COST	The minimum cost of a query plan to be included in the resource group.	Integer	0

Note

Resource limits are not enforced on SET, RESET, and SHOW commands.

Transaction concurrency limit

The CONCURRENCY limit controls the maximum number of concurrent transactions permitted for a resource group.

Each resource group is logically divided into a fixed number of slots equal to the CONCURRENCY limit. SynxDB allocates these slots an equal, fixed percentage of memory resources.

The default CONCURRENCY limit value for a resource group for roles is 20. A value of 0 means that no query is allowed to run for this resource group.

SynxDB queues any transactions submitted after the resource group reaches its CONCURRENCY limit. When a running transaction completes, SynxDB un-queues and runs the earliest queued transaction if sufficient memory resources exist. Note that if a transaction is in `idle in transaction` state, even if no statement is running, the concurrency slot is still in use.

You can set the server configuration parameter `gp_resource_group_queuing_timeout` to specify the amount of time a transaction remains in the queue before SynxDB cancels the transaction. The default timeout is zero, SynxDB queues transactions indefinitely.

Bypass limits and unassign from resource groups

A query bypasses the resource group concurrency limit if you set the server configuration parameter `gp_resource_group_bypass`. This parameter enables or disables the concurrent transaction limit for the resource group so a query can run immediately. The default value is false, which enforces the limit of the CONCURRENCY limit. You might only set this parameter for a single session, not within a transaction or a function. If you set `gp_resource_group_bypass` to true, the query no longer enforces the CPU or memory limits assigned to its resource group. Instead, the memory limit assigned to this query is `statement_mem` per query. If there is not enough memory to satisfy the memory allocation request, the query will fail.

You might bypass queries that only use catalog tables, such as the database Graphical User Interface (GUI) client, which runs catalog queries to obtain metadata. If the server configuration parameter `gp_resource_group_bypass_catalog_query` is set to true (the default), SynxDB's resource group scheduler bypasses all queries that read exclusively from system catalogs, or queries that contain in their query text `pg_catalog` schema tables only. These queries are automatically unassigned from its current resource group; they do not enforce the limits of the

resource group and do not account for resource usage. The query resources are assigned out of the resource groups and run immediately. The memory limit is `statement_mem` per the query.

You might bypass direct dispatch queries with the server configuration parameter `gp_resource_group_bypass_direct_dispatch`. A direct dispatch query is a special type of query that only requires a single segment to participate in the execution. In order to improve efficiency, SynxDB optimizes this type of query, using direct dispatch optimization. The system sends the query plan to the execution of a single segment that needs to execute the plan, instead of sending it to all segments for execution. If you set `gp_resource_group_bypass_direct_dispatch` to true, the query no longer enforces the CPU or memory limits assigned to its resource group, so it runs immediately. Instead, the memory limit assigned to this query is `statement_mem` per query. If there is not enough memory to satisfy the memory allocation request, the query will fail. You might only set this parameter for a single session, not within a transaction or a function.

Queries whose plan cost is less than the limit `MIN_COST` are automatically unassigned from their resource group and do not enforce any of its limits. The resources used by the query do not account for the resources of the resource group. The query has a memory limit of `statement_mem`. The default value of `MIN_COST` is 0.

CPU limits

SynxDB leverages Linux control groups to implement CPU resource management. SynxDB allocates CPU resources in two ways:

- By assigning a percentage of CPU resources to resource groups
- By assigning specific CPU cores to resource groups

When you set one of the allocation modes for a resource group, SynxDB deactivates the other allocation mode. You might employ both modes of CPU resource allocation simultaneously for different resource groups on the same SynxDB cluster. You might also change the CPU resource allocation mode for a resource group at runtime.

SynxDB uses the server configuration parameter `gp_resource_group_cpu_limit` to identify the maximum percentage of system CPU resources to allocate to resource groups on each SynxDB segment node. The remaining unreserved CPU resources are used for the operating system kernel and SynxDB daemons. The amount of CPU available to SynxDB queries per host is

then divided equally among each segment on the SynxDB node.

 **Note**

The default `gp_resource_group_cpu_limit` value might not leave sufficient CPU resources if you are running other workloads on your SynxDB cluster nodes, so be sure to adjust this server configuration parameter accordingly.

 **Caution**

Avoid setting `gp_resource_group_cpu_limit` to a value higher than .9. Doing so might result in high workload queries taking near all CPU resources, potentially starving SynxDB auxiliary processes.

Assign CPU resources by core

You identify the CPU cores that you want to reserve for a resource group with the CPUSED property. The CPU cores that you specify must be available in the system and cannot overlap with any CPU cores that you reserved for other resource groups. Although SynxDB uses the cores that you assign to a resource group exclusively for that group, note that those CPU cores might also be used by non-SynxDB processes in the system. When you configure CPUSED for a resource group, SynxDB deactivates `CPU_MAX_PERCENT` and `CPU_WEIGHT` for the group and sets their value to -1.

Specify CPU cores separately for the coordinator host and segment hosts, separated by a semicolon. Use a comma-separated list of single core numbers or number intervals when you configure cores for CPUSED. You must enclose the core numbers/intervals in single quotes, for example, '`1;1,3-4`' uses core 1 on the coordinator host, and cores 1, 3, and 4 on segment hosts.

When you assign CPU cores to CPUSED groups, consider the following:

- A resource group that you create with CPUSED uses the specified cores exclusively. If there are no running queries in the group, the reserved cores are idle and cannot be used by queries in other resource groups. Consider minimizing the number of CPUSED groups to avoid wasting system CPU resources.

- Consider keeping CPU core 0 unassigned. CPU core 0 is used as a fallback mechanism in the following cases:
 - `admin_group` and `default_group` require at least one CPU core. When all CPU cores are reserved, SynxDB assigns CPU core 0 to these default groups. In this situation, the resource group to which you assigned CPU core 0 shares the core with `admin_group` and `default_group`.
 - If you restart your SynxDB cluster with one node replacement and the node does not have enough cores to service all CPuset resource groups, the groups are automatically assigned CPU core 0 to avoid system start failure.
- Use the lowest possible core numbers when you assign cores to resource groups. If you replace a SynxDB node and the new node has fewer CPU cores than the original, or if you back up the database and want to restore it on a cluster with nodes with fewer CPU cores, the operation might fail. For example, if your SynxDB cluster has 16 cores, assigning cores 1-7 is optimal. If you create a resource group and assign CPU core 9 to this group, database restore to an 8 core node will fail.

Resource groups that you configure with CPuset have a higher priority on CPU resources. The maximum CPU resource usage percentage for all resource groups configured with CPuset on a segment host is the number of CPU cores reserved divided by the number of all CPU cores, multiplied by 100.

Note

You must configure CPuset for a resource group *after* you have enabled resource group-based resource management for your SynxDB cluster with the `gp_resource_manager` server configuration parameter.

Assign CPU resources by percentage

You configure a resource group with `CPU_MAX_PERCENT` in order to assign CPU resources by percentage. When you configure `CPU_MAX_PERCENT` for a resource group, SynxDB deactivates CPuset for the group.

The parameter `CPU_MAX_PERCENT` sets a hard upper limit for the percentage of the segment

CPU for resource management. The minimum CPU_MAX_PERCENT percentage you can specify for a resource group is 1, the maximum is 100. The sum of CPU_MAX_PERCENTs specified for all resource groups that you define in your SynxDB cluster can exceed 100. It specifies the total time ratio that all tasks in a resource group can run in a given CPU time period. Once the tasks in the resource group have used up all the time specified by the limit, they are throttled for the remainder of the time specified in that time period, and are not allowed to run until the next time period.

When tasks in a resource group are idle and not using any CPU time, the leftover time is collected in a global pool of unused CPU cycles. Other resource groups can borrow CPU cycles from this pool. The actual amount of CPU time available to a resource group might vary, depending on the number of resource groups present on the system.

The parameter CPU_MAX_PERCENT enforces a hard upper limit of CPU usage. For example, if it is set to 40%, it indicates that although the resource group can temporarily use some idle CPU resources from other groups, the maximum it can use is 40% of the CPU resources available to SynxDB.

You set the parameter CPU_WEIGHT to assign the scheduling priority of the current group. The default value is 100, and the range of values is 1 to 500. The value specifies the relative share of CPU time available to tasks in the resource group. For example, if one resource group has a relative share of 100 and another two groups have a relative share of 50, when processes in all the resource groups try to use 100% of the CPU (that means, the value of CPU_MAX_PERCENT for all groups is set to 100), the first resource group gets 50% of all CPU time, and the other two get 25% each. However, if you add another group with a relative share of 100, the first group is only allowed to use 33% of the CPU, and the remaining groups get 16.5%, 16.5%, and 33% respectively.

For example, consider the following groups:

Group Name	CONCURRENCY	CPU_MAX_PERCENT	CPU_WEIGHT
default_group	20	50	10
admin_group	10	70	30
system_group	10	30	10
test	10	10	10

Roles in default_group have an available CPU ratio (determined by CPU_WEIGHT) of $10/(10+30+10+10)=16\%$. This means that they can use at least 16% of the CPU when the system

workload is high. When the system has idle CPU resources, they can use more resources, as the hard limit (set by `CPU_MAX_PERCENT`) is 50%.

Roles in `admin_group` have an available CPU ratio of $30/(10+30+10+10)=50\%$ when the system workload is high. When the system has idle CPU resources, they can use resources up to the hard limit of 70%.

Roles in `test` have a CPU ratio of $10/(10+30+10+10)=16\%$. However, as the hard limit determined by `CPU_MAX_PERCENT` is 10%, they can only use up to 10% of the resources even when the system is idle.

Memory limits

When you enable resource groups, memory usage is managed at the SynxDB segment and resource group levels. You can also manage memory at the transaction level.

The amount of memory allocated to a query is determined by the following parameters:

The parameter `MEMORY_QUOTA` of a resource group sets the maximum amount of memory reserved for this resource group on a segment. This determines the total amount of memory that all worker processes for a query can consume on the segment host during query execution. The amount of memory allotted to a query is the group memory limit divided by the group concurrency limit: `MEMORY_QUOTA / CONCURRENCY`.

If a query requires a large amount of memory, you might use the server configuration parameter `gp_resgroup_memory_query_fixed_mem` to set a fixed memory amount for the query at the session level. This parameter overrides and can surpass the allocated memory of the resource group.

SynxDB allocates memory for an incoming query using the `gp_resgroup_memory_query_fixed_mem` value, if set, to bypass the resource group settings. Otherwise, it uses `MEMORY_QUOTA / CONCURRENCY` as the memory allocated for the query. If `MEMORY_QUOTA` is not set, the value for the query memory allocation defaults to `statement_mem`.

For all queries, if there is not enough memory in the system, they spill to disk. When the limit `gp_workfile_limit_files_per_query` is reached, SynxDB generates an out of memory (OOM) error.

For example, consider a resource group named adhoc with MEMORY_QUOTA set to 1.5 GB and CONCURRENCY set to 3. By default, each statement submitted to the group is allocated 500 MB of memory. Now consider the following series of events:

1. User ADHOC_1 submits query Q1, overriding gp_resgroup_memory_query_fixed_mem to 800MB. The Q1 statement is admitted into the system.
2. User ADHOC_2 submits query Q2, using the default 500MB.
3. With Q1 and Q2 still running, user ADHOC3 submits query Q3, using the default 500MB.

Queries Q1 and Q2 have used 1300MB of the group's 1500MB. However, if there is enough system memory available for query Q3 in the segment at that time, it can run normally.

4. User ADHOC4 submits query Q4, using gp_resgroup_memory_query_fixed_mem set to 700 MB.

Query Q4 runs immediately as it bypasses the resource group limits.

There are some special usage considerations regarding memory limits:

- If the configuration parameters gp_resource_group_bypass or gp_resource_group_bypass_catalog_query are enabled to bypass resource group limits, the query's memory allocation will be determined by the value of statement_mem.
- When (MEMORY_QUOTA / CONCURRENCY) is less than statement_mem, the system uses statement_mem as the fixed memory allocation for the query.
- The maximum value of statement_mem is limited by max_statement_mem.
- If a query's planned cost is lower than the configured MIN_COST, it will also be assigned memory based on statement_mem.

Disk I/O limits

SynxDB leverages Linux control groups to implement disk I/O limits. The parameter IO_LIMIT limits the maximum read/write disk I/O throughput, and the maximum read/write I/O operations per second for the queries assigned to a specific resource group. It allocates bandwidth, ensures the use of high-priority resource groups, and avoids excessive use of disk bandwidth. The value of the parameter is set on a per-tablespace basis.

Note

Disk I/O limits are only available when you use Linux Control Groups v2.

When you limit disk I/O you specify:

- The tablespace name or the tablespace object ID (OID) you set the limits for. Use * to set limits for all tablespaces.
- The values for `rbps` and `wbps` to limit the maximum read and write disk I/O throughput in the resource group, in MB/sec. The default value is `max`, which means there is no limit.
- The values for `riops` and `wiops` to limit the maximum read and write I/O operations per second in the resource group. The default value is `max`, which means there is no limit.

If the parameter `IO_LIMIT` is not set, the default value for `rbps`, `wbps`, `riops`, and `wiops` is set to `max`, which means that there are no disk I/O limits. In this scenario, the `gp_toolkit.gp_resgroup_config` system view displays its value as `-1`. If only some of the values of `IO_LIMIT` are set (for example. `rbps`), the parameters that are not set default to `max` (in this example, `wbps`, `riops`, `wiops`).

Configure and use resource groups

Prerequisites

SynxDB resource groups use Linux Control Groups (cgroups) to manage CPU resources and disk I/O. There are two versions of cgroups: cgroup v1 and cgroup v2. SynxDB supports both versions, but it only supports the parameter `IO_LIMIT` for cgroup v2. The version of Linux Control Groups shipped by default with your Linux distribution depends on the operating system version. For Enterprise Linux 8 and older, the default version is v1. For Enterprise Linux 9 and later, the default version is v2.

Verify what version of cgroup is configured in your environment by checking what filesystem is mounted by default during system boot:

```
stat -fc %T /sys/fs/cgroup/
```

For cgroup v1, the output is `tmpfs`. For cgroup v2, output is `cgroup2fs`.

You do not need to change your version of cgroup, you can simply skip to [Configure cgroup v1](#) or [Configure cgroup v2](#) in order to complete the configuration prerequisites.

However, if you prefer to switch from cgroup v1 to v2, run the following commands as root:

Note

- If you are using an older version of CentOS, such as Centos 7.x or earlier, you will only be able to use the default cgroup v1 control groups; if you are using Centos 8.x, you can follow the steps below to switch the control group from v1 to v2.
- Starting from v4.0.0, for `cpu.pressure`, even when `gp_resource_manager` is set to `group-v2`, SynxDB no longer checks the permissions or existence of the `/proc/pressure/cpu` interface. Therefore, the `group-v2` mode can still function properly as long as the kernel version is compatible, even if PSI (Pressure Stall Information) is not enabled.

- Red Hat 8/Rocky 8/Oracle 8 systems:

```
grubby --update-kernel=/boot/vmlinuz-$$(uname -r) --args="systemd.unified_cgroup_hierarchy=1"
```

- Ubuntu systems:

```
vim /etc/default/grub      # add or modify: GRUB_CMDLINE_LINUX="systemd.unified_cgroup_hierarchy=1"
update-grub
```

If you want to switch from cgroup v2 to v1, run the following commands as root:

- Red Hat 8/Rocky 8/Oracle 8 systems:

```
grubby --update-kernel=/boot/vmlinuz-$$(uname -r) --args="systemd.unified_cgroup_hierarchy=0 systemd.legacy_systemd_cgroup_controller"
```

- Ubuntu systems:

```
vim /etc/default/grub
# add or modify: GRUB_CMDLINE_LINUX="systemd.unified_cgroup_hierarchy=0"
```

(continues on next page)

(continued from previous page)

```
update-grub
```

After that, reboot your host in order for the changes to take effect.

Configure cgroup v1

Complete the following tasks on each node in your SynxDB cluster to set up cgroups v1 for use with resource groups:

1. Install libcgroup and libcgroup-tools:

```
sudo yum install libcgroup
sudo yum install libcgroup-tools
```

2. Locate the cgroups configuration file `/etc/cgconfig.conf`. You must be the superuser or have `sudo` access to edit this file:

```
vi /etc/cgconfig.conf
```

3. Add the following configuration information to the file:

```
group gpdb {
    perm {
        task {
            uid = gpadmin;
            gid = gpadmin;
        }
        admin {
            uid = gpadmin;
            gid = gpadmin;
        }
    }
    cpu {
    }
    cpuacct {
    }
    cpuset {
    }
```

(continues on next page)

(continued from previous page)

```
memory {  
}  
}
```

This content configures CPU, CPU accounting, CPU core set, and memory control groups managed by the gpadmin user. SynxDB uses the memory control group only for monitoring the memory usage.

4. Start the cgroups service on each SynxDB node. For Redhat/Oracle/Rocky 8.x and older, run the following as root:

```
cgconfigparser -l /etc/cgconfig.conf
```

5. To automatically recreate SynxDB required cgroup hierarchies and parameters when your system is restarted, configure your system to enable the Linux cgroup service daemon cgconfig.service at node start-up. To ensure the configuration is persistent after reboot, run the following commands as user root. For Redhat/Oracle/Rocky 8.x and older:

```
systemctl enable cgconfig.service
```

To start the service immediately (without having to reboot) enter:

```
systemctl start cgconfig.service
```

6. Identify the cgroup directory mount point for the node. For Redhat/Oracle/Rocky 8.x and older, run the following as root:

```
grep cgroup /proc/mounts
```

The first line of output identifies the cgroup mount point.

7. Verify that you set up the SynxDB cgroups configuration correctly by running the following commands. Replace <cgroup_mount_point> with the mount point that you identified in the previous step:

```
ls -l <cgroup_mount_point>/cpu/gpdb  
ls -l <cgroup_mount_point>/cpuset/gpdb  
ls -l <cgroup_mount_point>/memory/gpdb
```

If these directories exist and are owned by gpadmin:gpadmin, you have successfully configured cgroups for SynxDB resource management.

Configure cgroup v2

1. Configure the system to mount `cgroups-v2` by default during system boot by the `systemd` system and service manager as user root.

```
grubby --update-kernel=ALL --args="systemd.unified_cgroup_hierarchy=1"
```

2. Reboot the system for the changes to take effect. `reboot now`
3. Create the directory `/sys/fs/cgroup/gpdb`, add all the necessary controllers, and ensure `gpadmin` user has read and write permission on it.

```
mkdir -p /sys/fs/cgroup/gpdb
echo "+cpuset +io +cpu +memory" | tee -a /sys/fs/cgroup/cgroup.subtree_
control
chown -R gpadmin:gpadmin /sys/fs/cgroup/gpdb
```

You might encounter the error `Invalid argument` after running the above commands. This is because cgroups v2 do not support control of real-time processes, and the `cpu` controller can only be enabled when all the real-time processes are in the root cgroup. In this situation, find all real-time processes and move them to the root cgroup before you re-enable the controllers.

1. Ensure that `gpadmin` has write permission on `/sys/fs/cgroup/cgroup.procs`. This is required to move the SynxDB processes from the user slices to `/sys/fs/cgroup/gpdb/` after the cluster is started in order to manage the postmaster services and all its auxiliary processes.

```
chmod a+w /sys/fs/cgroup/cgroup.procs
```

Because resource groups manually manage cgroup files, the above settings will become ineffective after a system reboot. Add the following bash script for `systemd` so it runs automatically during system startup. Perform the following steps as user root:

2. Create `gpdb.service`.

```
vim /etc/systemd/system/gpdb.service
```

3. Write the following content into `gpdb.service`, if the user is not `gpadmin`, replace it with the appropriate user.

```
[Unit]
Description=|product_name| Cgroup v2 Configuration Service
[Service]
Type=simple
WorkingDirectory=/sys/fs/cgroup/gpdb.service
Delegate=yes
Slice=-.slice

# set hierarchies only if cgroup v2 mounted
ExecCondition=bash -c '[ xcgroup2fs = x$(stat -fc "%%T" /sys/fs/cgroup) ] || exit 1'
ExecStartPre=bash -ec " \
chown -R gpadmin:gpadmin .; \
chmod a+w ../*.proc; \
mkdir -p helper.scope"
ExecStart=sleep infinity
ExecStartPost=bash -ec "echo $MAINPID > ./helper.scope/cgroup.procs;"
[Install]
WantedBy=basic.target
```

4. Reload systemd daemon and enable the service:

```
systemctl daemon-reload
systemctl enable gpdb.service
```

Enable resource groups

When you install SynxDB, no resource management policy is enabled by default. To use resource groups, set the `gp_resource_manager` server configuration parameter.

1. Set the `gp_resource_manager` server configuration parameter to the value "`group`" or "`group-v2`", depending on the version of cgroup configured on your Linux distribution.
For example:

```
gpconfig -c gp_resource_manager -v "group"
gpconfig -c gp_resource_manager -v "group-v2"
```

2. Restart SynxDB:

```
gpstop
gpstart
```

Once enabled, any transaction submitted by a role is directed to the resource group assigned to the role, and is governed by that resource group's concurrency, memory, CPU, and disk I/O limits.

SynxDB creates three default resource groups for roles named `admin_group`, `default_group`, and `system_group`. When you enable resources groups, any role that was not explicitly assigned a resource group is assigned the default group for the role's capability. SUPERUSER roles are assigned the `admin_group`, non-admin roles are assigned the group named `default_group`. The resources of the SynxDB system processes are assigned to the `system_group`. You cannot manually assign any roles to the `system_group`.

Use the following command to retrieve details of all the resource groups, including the default resource groups `admin_group`, `default_group`, and `system_group`:

```
SELECT * FROM gp_toolkit.gp_resgroup_config;
```

Create resource groups

When you create a resource group for a role, you provide a name and a CPU resource allocation mode (core or percentage). You can optionally provide a concurrent transaction limit, a memory limit, a CPU soft priority, disk I/O limits, and a minimum cost. Use the `CREATE RESOURCE GROUP` command to create a new resource group.

When you create a resource group for a role, you must provide a `CPU_MAX_PERCENT` or `CPUSED` limit value. These limits identify the percentage of SynxDB CPU resources to allocate to this resource group. You might specify a `MEMORY_QUOTA` to reserve a fixed amount of memory for the resource group.

For example, to create a resource group named `rgroup1` with a CPU limit of 20, a memory limit of 25, a CPU soft priority of 500, a minimum cost of 50, and disk I/O limits for the `pg_default` tablespace:

```
CREATE RESOURCE GROUP rgroup1 WITH (CONCURRENCY=20, CPU_MAX_
PERCENT=20, MEMORY_QUOTA=250, CPU_WEIGHT=500, MIN_COST=50,
IO_LIMIT='pg_default: wbps=1000, rbps=1000, wiops=100, riops=100');
```

i Note

If you use cgroup v1 but specified IO_LIMIT. The resource group would still be created but with the following warning: “resource group io limit only can be used in cgroup v2” .

The CPU limit of 20 is shared by every role to which rgroup1 is assigned. Similarly, the memory limit of 25 is shared by every role to which rgroup1 is assigned. rgroup1 utilizes the default CONCURRENCY setting of 20.

The ALTER RESOURCE GROUP command updates the limits of a resource group. To change the limits of a resource group, specify the new values that you want for the group. For example:

```
ALTER RESOURCE GROUP rg_role_light SET CONCURRENCY 7;
ALTER RESOURCE GROUP exec SET MEMORY_QUOTA 30;
ALTER RESOURCE GROUP rgroup1 SET CPuset '1;2,4';
ALTER RESOURCE GROUP sales SET IO_LIMIT 'tablespace1:wbps=2000,
wiops=2000;tablespace2:rbps=2024,riops=2024';
```

i Note

You cannot set or alter the CONCURRENCY value for the admin_group to zero (0).

The DROP RESOURCE GROUP command drops a resource group. To drop a resource group for a role, the group cannot be assigned to any role, nor can there be any transactions active or waiting in the resource group.

To drop a resource group:

```
DROP RESOURCE GROUP exec;
```

Configure automatic query termination based on memory usage

SynxDB supports the Runaway detector, which automatically terminates queries based on the amount of memory used by the query. For resource group-managed queries, SynxDB terminates a running query based on the amount of memory used by the query. The relevant configuration parameters are:

- `gp_vmem_protect_limit` sets the amount of memory that all `postgres` processes of the active segment instance can consume. If a query causes this limit to be exceeded, no memory will be allocated and the query will fail.
- `runaway_detector_activation_percent` When resource groups are enabled, if the used memory exceeds the specified value `gp_vmem_protect_limit * runaway_detector_activation_percent`, SynxDB terminates queries based on memory usage, selecting queries from the queries managed by user resource groups (excluding those in the `system_group` resource group). SynxDB starts with the query that consumes the largest amount of memory. The query will terminate until the percentage of memory used falls below the specified percentage.

Assign a resource group to a role

You assign a resource group to a database role using the `RESOURCE GROUP` clause of the `CREATE ROLE` or `ALTER ROLE` commands. If you do not specify a resource group for a role, the role is assigned the default group for the role's capability. `SUPERUSER` roles are assigned the `admin_group`, non-admin roles are assigned the group named `default_group`.

Use the `ALTER ROLE` or `CREATE ROLE` commands to assign a resource group to a role. For example:

```
ALTER ROLE bill RESOURCE GROUP rg_light;
CREATE ROLE mary RESOURCE GROUP exec;
```

You can assign a resource group to one or more roles. If you have defined a role hierarchy, assigning a resource group to a parent role does not propagate down to the members of that role group.

If you wish to remove a resource group assignment from a role and assign the role the default group, change the role's group name assignment to `NONE`. For example:

```
ALTER ROLE mary RESOURCE GROUP NONE;
```

Monitor resource group status

Monitoring the status of your resource groups and queries might involve the following tasks.

View resource group limits

The `gp_toolkit.gp_resgroup_config` system view displays the current limits for a resource group. To view the limits of all resource groups:

```
SELECT * FROM gp_toolkit.gp_resgroup_config;
```

View resource group query status

The `gp_toolkit.gp_resgroup_status` system view enables you to view the status and activity of a resource group. The view displays the number of running and queued transactions. To view this information:

```
SELECT * FROM gp_toolkit.gp_resgroup_status;
```

View resource group memory usage per host

The `gp_toolkit.gp_resgroup_status_per_host` system view enables you to view the real-time memory usage of a resource group on a per-host basis. To view this information:

```
SELECT * FROM gp_toolkit.gp_resgroup_status_per_host;
```

View the resource group assigned to a role

To view the resource group-to-role assignments, perform the following query on the pg_roles and pg_resgroup system catalog tables:

```
SELECT rolname, rsgname FROM pg_roles, pg_resgroup  
WHERE pg_roles.rolresgroup=pg_resgroup.oid;
```

View resource group disk I/O usage per host

The gp_resgroup_iostats_per_host gp_toolkit system view enables you to view the real-time disk I/O usage of a resource group on a per-host basis. To view this information:

```
SELECT * FROM gp_toolkit.gp_resgroup_iostats_per_host;
```

View a resource group's running and pending queries

To view a resource group's running queries, pending queries, and how long the pending queries have been queued, examine the pg_stat_activity system catalog table:

```
SELECT query, rsgname, wait_event_type, wait_event  
FROM pg_stat_activity;
```

pg_stat_activity displays information about the user/role that initiated a query. A query that uses an external component such as PL/Container is composed of two parts: the query operator that runs in SynxDB and the UDF that runs in a PL/Container instance. SynxDB processes the query operators under the resource group assigned to the role that initiated the query. A UDF running in a PL/Container instance runs under the resource group assigned to the PL/Container runtime. The latter is not represented in the pg_stat_activity view; SynxDB does not have any insight into how external components such as PL/Container manage memory in running instances.

Cancel a running or queued transaction in a resource group

There might be cases when you want to cancel a running or queued transaction in a resource group. For example, you might want to remove a query that is waiting in the resource group queue but has not yet been run. Or, you might want to stop a running query that is taking too long to run, or one that is sitting idle in a transaction and taking up resource group transaction slots that are needed by other users.

By default, transactions can remain queued in a resource group indefinitely. If you want SynxDB to cancel a queued transaction after a specific amount of time, set the server configuration parameter `gp_resource_group_queuing_timeout`. When this parameter is set to a value (milliseconds) greater than 0, SynxDB cancels any queued transaction when it has waited longer than the configured timeout.

To manually cancel a running or queued transaction, you must first determine the process id (pid) associated with the transaction. Once you have obtained the process id, you can invoke `pg_cancel_backend()` to end that process, as shown below.

For example, to view the process information associated with all statements currently active or waiting in all resource groups, run the following query. If the query returns no results, then there are no running or queued transactions in any resource group.

```
SELECT rolname, g.rsgname, pid, waiting, state, query, datname  
FROM pg_roles, gp_toolkit.gp_resgroup_status g, pg_stat_activity  
WHERE pg_roles.rolresgroup=g.groupid  
AND pg_stat_activity.username=pg_roles.rolname;
```

Sample partial query output:

rolname	rsgname	pid	waiting	state	query
datname					
-----	-----	-----	-----	-----	-----
-----	-----	-----	-----	-----	-----
sammy	rg_light	31861	f	idle	SELECT * FROM mytesttbl;
testdb					
billy	rg_light	31905	t	active	SELECT * FROM topten;
testdb					

Use this output to identify the process id (`pid`) of the transaction you want to cancel, and then

cancel the process. For example, to cancel the pending query identified in the sample output above:

```
SELECT pg_cancel_backend(31905);
```

You can provide an optional message in a second argument to `pg_cancel_backend()` to indicate to the user why the process was cancelled.

 **Note**

Do not use an operating system `KILL` command to cancel any SynxDB process.

Move a query to a different resource group

A user with SynxDB superuser privileges can run the `gp_toolkit.pg_resgroup_move_query()` function to move a running query from one resource group to another, without stopping the query. Use this function to expedite a long-running query by moving it to a resource group with a higher resource allotment or availability.

 **Note**

You can move only an active or running query to a new resource group. You cannot move a queued or pending query that is in an idle state due to concurrency or memory limits.

`pg_resgroup_move_query()` requires the process id (pid) of the running query, as well as the name of the resource group to which you want to move the query. The signature of the function follows:

```
pg_resgroup_move_query( pid int4, group_name text );
```

You can obtain the pid of a running query from the `pg_stat_activity` system view as described in [Cancel a running or queued transaction in a resource group](#). Use the `gp_toolkit.gp_resgroup_status` view to list the name, id, and status of each resource group.

When you invoke `pg_resgroup_move_query()`, the query is subject to the limits configured for the destination resource group:

- If the group has already reached its concurrent task limit, SynxDB queues the query until a slot opens or for `gp_resource_group_queuing_timeout` milliseconds if set.
- If the group has a free slot, `pg_resgroup_move_query()` tries to give slot control away to the target process for up to `gp_resource_group_move_timeout` milliseconds. If target process can't handle movement request until `gp_resource_group_queuing_timeout` exceeds, SynxDB returns the error: `target process failed to move to a new group`.
- If `pg_resgroup_move_query()` was cancelled, but target process already got all slot controls, segment's processes will not be moved to new group, and target process will hold the slot. Such inconsistent state will be fixed by the end of transaction or by any next command dispatched by target process inside same transaction.
- If the destination resource group does not have enough memory available to service the query's current memory requirements, SynxDB returns the error: `group <group_name> doesn't have enough memory . . .` In this situation, you might choose to increase the group shared memory allotted to the destination resource group, or perhaps wait a period of time for running queries to complete and then invoke the function again.

After SynxDB moves the query, there is no way to guarantee that a query currently running in the destination resource group does not exceed the group memory limit. In this situation, one or more running queries in the destination group might fail, including the moved query. Reserve enough resource group global shared memory to minimize the potential for this scenario to occur.

`pg_resgroup_move_query()` moves only the specified query to the destination resource group. SynxDB assigns subsequent queries that you submit in the session to the original resource group.

Frequently asked questions

Why is CPU usage lower than the ``CPU_MAX_PERCENT`` configured for the resource group?

You might run into this situation when a low number of queries and slices are running in the resource group, and these processes are not utilizing all of the cores on the system.

My resource group has a `CPU_WEIGHT` equivalent to 40%. Why is the CPU usage never reaching this limit?

The value of CPU_MAX_PERCENT might be lower than 40, hence it might be limiting the CPU usage even with idle resources.

Why is the number of running transactions lower than the ``CONCURRENCY`` limit configured for the resource group?

SynxDB considers memory availability before running a transaction, and will queue the transaction if there is not enough memory available to serve it. If you use ALTER RESOURCE GROUP to increase the CONCURRENCY limit for a resource group but do not also adjust memory limits, currently running transactions might be consuming all allotted memory resources for the group. When in this state, SynxDB queues subsequent transactions in the resource group.

Why is the number of running transactions in the resource group higher than the configured ``CONCURRENCY`` limit?

This behaviour is expected. There are several reasons why this might happen:

- Resource groups do not enforce resource restrictions on SET, RESET and SHOW commands.
- The server configuration parameter gp_resource_group_bypass disables the concurrent transaction limit for the resource group so a query can run immediately.
- If the server configuration parameter gp_resource_group_bypass_catalog_query is set to true (the default), all queries that read exclusively from system catalogs, or queries that contain in their query text pg_catalog schema tables only will not enforce the limits of the resource group.
- Queries whose plan cost is less than the limit MIN_COST will be automatically unassigned from their resource group and will not enforce any of the limits set for this.

Why did my query return a “memory limit reached” error?

SynxDB automatically adjusts transaction and group memory to the new settings when you use ALTER RESOURCE GROUP to change a resource group’s memory and/or concurrency limits. An “out of memory” error might occur if you recently altered resource group attributes and there is no longer a sufficient amount of memory available for a currently running query.

My query cannot run due to insufficient memory, resulting in memory leak Out of Memory (OOM).

First, ensure that the resource group is allocating enough memory required by the query by tuning

resource group parameters such as CONCURRENCY and MEMORY_QUOTA. Analyze the type of query, whether there will be a lot of intermediate results using memory. If it does exist, you can set a reasonable gp_resgroup_memory_query_fixed_mem to allocate more memory at the session level for this specific query.

After a memory leak OOM the system has a high concurrent load.

When the system starts to clean up the sessions left over by the memory leak, the concurrent load of the system is high at this time, and the OOM error message might reappear. Due to the current design, we cannot expedite the cleanup process of the Runaway Session. The solution to this problem is to adjust the runaway_detector_activation_percent to 0.85 or 0.8, or even lower, in order to increase the available memory of the segment host.

Some transaction requests only run during a certain period of time, and do not run at other times.

You might change the configuration of resource groups can be changed dynamically at regular intervals to match the requirements of your workload, and customize resource allocation at different times to achieve higher efficiency. For example, change the configuration of resources within a group, add or delete resource groups.

5.3 Use Dynamic Tables to Speed Up Queries and Auto-Refresh Data

Dynamic tables are database objects similar to materialized views that refresh data automatically and speed up queries. SynxDB introduces dynamic tables to make query processing faster and data updates automatic.

Dynamic tables can be created from base tables, external tables, or materialized views. They update data automatically based on a schedule, keeping the data current.

Use cases

Dynamic tables are suitable for these scenarios:

- **Faster queries in lakehouse setups:** Replace external table queries with dynamic table queries to improve performance.
- **Automatic data updates:** Schedule refresh tasks to keep data updated without manual effort.
- **Real-time analysis:** Do well for scenarios that need frequent and up-to-date queries, such as financial analysis or operations monitoring.

Comparison with materialized views

Dynamic tables and Answer Query Using Materialized Views have these differences:

Feature	Dynamic table	AQUMV
Purpose	A special table that automatically refreshes, processes data pipelines, and simplifies ETL.	Uses materialized views to improve query efficiency and automatically rewrite queries.
Base and structure	Can be based on ordinary tables, external tables, materialized views.	Based on materialized views, usually targeting a single table.
Query rewrite	Not supported	Supports single-table rewrite
Data refresh mechanism	Users can define automatic refresh interval through SQL	Requires manual refresh of materialized view data

Usage

Create a dynamic table

To create a dynamic table, use the CREATE DYNAMIC TABLE statement:

```
CREATE DYNAMIC TABLE table_name AS <select_query> [WITH NO DATA] [SCHEDULE '  
<cron_string>'] [DISTRIBUTED BY <distribution_key>];
```

Parameter details:

- <select_query>: The SQL query with which the dynamic table is defined. Supports joins with base tables, materialized views, or other dynamic tables.
- WITH NO DATA: Creates the table without initial data.
- SCHEDULE '<cron_string>': Sets the refresh schedule using Linux cron expressions. See [Cron Expressions](#) for more details.
- DISTRIBUTED BY: Defines the distribution key to improve query performance.

Full syntax:

```
CREATE DYNAMIC TABLE [ IF NOT EXISTS ] table_name  
[ (column_name [, ...] ) ]  
[ USING method ]  
[ WITH ( storage_parameter [= value] [, ... ] ) ]  
[ TABLESPACE tablespace_name ]  
[ SCHEDULE schedule_string ]  
AS query  
[ WITH [ NO ] DATA ]  
[ DISTRIBUTED BY (keys [, ...]) ];
```

Example:

```
CREATE DYNAMIC TABLE dt0 SCHEDULE '5 * * * *' AS SELECT a, b, sum(c) FROM t1  
GROUP BY a, b WITH NO DATA DISTRIBUTED BY(b);
```

```
\d dt0
```

List **of** relations

(continues on next page)

(continued from previous page)

Schema	Name	Type	Owner	Storage
public	dt0	dynamic table	gpadmin	heap (1 rows)

This example creates a dynamic table `dt0` and schedules it to refresh every 5 minutes.

Refresh a dynamic table

Dynamic tables refresh automatically based on their `SCHEDULE` setting. To manually refresh a table, use the `REFRESH DYNAMIC TABLE` statement. Replace `table_name` with the actual table name:

```
REFRESH DYNAMIC TABLE table_name;
```

To clear the table and make it unreadable, use the `WITH NO DATA` option. Replace `table_name` with the actual table name:

```
REFRESH DYNAMIC TABLE table_name WITH NO DATA;
```

View schedule information

To view the refresh schedule of a dynamic table, use the `pg_get_dynamic_table_schedule` function. Replace `table_name` with the actual table name:

```
SELECT pg_get_dynamic_table_schedule('table_name'::regclass::oid);
```

Drop a dynamic table

To delete a dynamic table and its associated refresh tasks, use the `DROP DYNAMIC TABLE` statement. Replace `table_name` with the actual table name:

```
DROP DYNAMIC TABLE table_name;
```

View distribution key

You can specify a distribution key when creating a dynamic table to optimize query performance. Use the \d+ command to view distribution keys and query definitions.

Example:

```
\d+ dt0
```

Output:

```
Dynamic table "public.dt0"
Column | Type      | Collation | Nullable | Default | Storage | Compression |
Stats target | Description
-----+-----+-----+-----+-----+-----+-----+
-----+-----+
a    | integer   |           |           | plain   |           |
     |           |           |           |         |
b    | integer   |           |           | plain   |           |         |
     |           |           |           |         |
sum | bigint    |           |           | plain   |           |         |
     |           |           |           |         |
View definition:
SELECT t1.a,
       t1.b,
       sum(t1.c) AS sum
  FROM t1
 GROUP BY t1.a, t1.b;
Distributed by: (b)
Access method: heap
```

In this example, the dynamic table `dt0` uses column `b` as its distribution key. This ensures efficient query and aggregation operations, because data is distributed across nodes based on `b`.

Examples

Example 1: Accelerate external table queries in lake-house architecture

SynxDB automatically replaces queries to external tables with queries on dynamic tables when processing data from data lakes or external storage, thereby accelerating the query process.

1. Create a readable external table `ext_r` with data sourced from the specified file

`dynamic_table_text_file.txt`.

```
CREATE READABLE EXTERNAL TABLE ext_r(id int)
  LOCATION('demoprot://dynamic_table_text_file.txt')
FORMAT 'text';
```

In the above statement for creating an external table,

`demoprot://dynamic_table_text_file.txt` is an example protocol and file path.

2. Use EXPLAIN to view the query execution plan for the table with the query condition `id > 5`.

```
EXPLAIN(COSTS OFF, VERBOSE)
SELECT sum(id) FROM ext_r WHERE id > 5;
```

Query plan output. In the following plan, Foreign Scan on `dynamic_table_schema.ext_r` appears, which means that the planner directly scanned the external table `ext_r`.

```
QUERY PLAN
-----
Finalize Aggregate
  Output: sum(id)
    -> Gather Motion 3:1 (slice1; segments: 3)
      Output: (PARTIAL sum(id))
        -> Partial Aggregate
          Output: PARTIAL sum(id)
            -> Foreign Scan on dynamic_table_schema.ext_r
              Output: id
              Filter: (ext_r.id > 5)
```

3. Create a dynamic table `dt_external` to store the filtered data from the external table `ext_r`, and execute `ANALYZE` to update the table statistics.

```
CREATE DYNAMIC TABLE dt_external AS
SELECT * FROM ext_r WHERE id > 5;
ANALYZE dt_external;
```

4. Set the parameters in a transaction to enable dynamic table query acceleration.

```
BEGIN;
SET optimizer = OFF;
SET LOCAL enable_answer_query_using_materialized_views = ON;
SET LOCAL aqumv_allow_foreign_table = ON;
```

5. Again, use `EXPLAIN` to view the query execution plan for the table with the query condition `id > 5`.

Use dynamic table to automatically replace external table queries, replacing the query to the external table `ext_r` with a query to the dynamic table `dt_external`, thereby accelerating the query.

```
EXPLAIN (COSTS OFF, VERBOSE)
SELECT sum(id) FROM ext_r WHERE id > 5;

COMMIT;
```

Query plan output. In the following plan, `Seq Scan` on `dynamic_table_schema.dt_external` indicates that the planner automatically replaced the query on the external table with the query on the dynamic table `dt_external`.

```
QUERY PLAN
-----
Finalize Aggregate
  Output: sum(id)
    -> Gather Motion 3:1 (slice1; segments: 3)
      Output: (PARTIAL sum(id))
        -> Partial Aggregate
          Output: PARTIAL sum(id)
            -> Seq Scan on dynamic_table_schema.dt_external
              Output: id
```

(continues on next page)

(continued from previous page)

```
Settings: enable_answer_query_using_materialized_views = 'on',
optimizer = 'off'
Optimizer: Postgres query optimizer
(10 rows)
```

Through this process, query efficiency is improved because the operation that originally required scanning the external table is replaced by scanning the dynamic table, which reduces query response time.

Example 2: Create an empty dynamic table

1. Create a base table `existing_table` and insert data:

```
CREATE TABLE existing_table (
    id INT,
    name VARCHAR(100),
    value INT
);

INSERT INTO existing_table (id, name, value) VALUES
(1, 'Alice', 100),
(2, 'Bob', 150),
(3, 'Charlie', 200);
```

2. Create an empty dynamic table:

```
CREATE DYNAMIC TABLE empty_table AS
SELECT * FROM existing_table
WITH NO DATA;
```

This statement creates an empty dynamic table, which can be populated with data through manual refresh later.

3. Check that this table is empty:

```
SELECT * FROM empty_table;

-- ERROR: materialized view "empty_table" has not been populated
```

(continues on next page)

(continued from previous page)

-- HINT: Use the `REFRESH MATERIALIZED VIEW` command.

Notes

- Refresh strategy: Ensure to set an appropriate refresh interval. If the refresh interval is too frequent, it might increase system load; if the refresh interval is too sparse, the query data might not be timely.
- Query replacement: The performance advantage of dynamic tables lies in automatically replacing external table queries, making them suitable for query scenarios that frequently require external table data.
- Data consistency: Because dynamic tables are a type of materialized view, pay attention to the balance between refresh strategy and data consistency to ensure the accuracy of application logic.

Chapter 6

AI & ML (HashML)

6.1 Deploy HashML Platform

HashML Platform is an integrated platform designed for data science and AI development. It provides full lifecycle management from data preparation to production deployment, including data management, model development, training, and deployment. The platform offers a web-based visual interface.

This document explains how to deploy HashML Platform for a SynxDB cluster.

Steps to deploy HashML Platform

Preparation

Before deploying HashML Platform, make sure the following requirements are met:

- A machine running CentOS 7 with at least 16 GB of available memory.
- A SynxDB database cluster has been successfully deployed and is accessible.
- The database is configured to allow access from HashML Platform nodes in the `pg_hba.conf` file.

- Contact SynxDB technical support to obtain the following RPM installation packages:
 - hashml-runtime (runtime dependencies)
 - hashml-platform (platform services)

Step 1: Install the RPM packages

1. Install the RPM packages. Replace <version> with the actual file name, for example:

hashml-platform-0.1.2-1.noarch.rpm.

```
# Installs the runtime component.
sudo yum install -y hashml-runtime-<version>.x86_64.rpm

# Installs the platform component.
sudo yum install -y hashml-platform-<version>.noarch.rpm
```

2. Initialize the environment. Switch to the hashml user and activate the hashml Python environment:

```
su - hashml
source /usr/local/hashml/bin/activate # Enter the hashml user environment
```

3. Verify the installation. If the installation is successful, the following command returns a result.

```
rpm -qa | grep hashml-platform
```

Step 2: Configure HashML Platform

1. Modify the HashML Platform configuration file. Edit the /etc/hashml/platform.toml file. For example:

```
[database] # Database connection settings
name = "hashml" # Database name
user = "gpadmin" # Database user
password = "gpadmin" # Database password
host = "localhost" # Database host
port = 5432 # Database port
schema = "hashml" # Database schema
```

(continues on next page)

(continued from previous page)

```

[service] # HashML Platform service settings
allowed_hosts = ["*"]
ray_service_url = "http://localhost:8000" # Ray service URL
service_host = "0.0.0.0" # HashML Platform service bind address
service_port = 8813 # HashML Platform service port
web_workers = 1

[storage]
file_path = "static"

[system]
thread_pool_max_workers = 20

```

Make sure the port number in `ray_service_url` matches the one configured later in `services.yml`.

2. Modify the service configuration file. Edit the `services.yml` file located at:

```
/usr/local/hashml/lib/python3.10/site-packages/hashml/platform/core/
services.yml
```

Update the `DBCONN_PREFIX` and `DBCONN` fields according to your actual database connection information:

```

DBCONN_PREFIX: postgresql://
DBCONN: gpadmin:gpadmin@localhost:5432/hashml

```

3. Set the following environment variables in the `hashml` user environment:

```

# Sets the shared storage directory.
export SHARED_STORAGE_PATH=/home/hashml/ray-logs

# Sets the model directory.
export ROOT_MODEL_DIR=/home/hashml/models

# Sets the data directory.
export ROOT_DATA_DIR=/home/hashml/

```

For multi-node deployments, you can set the shared storage to an S3-compatible path:

```
# Set the shared storage directory
export SHARED_STORAGE_PATH=s3://ak:sk@bucket/path?endpoint_override=http://
/ip:port
```

- Grant read and write permissions to the static resources directory (run as root):

```
sudo chmod 777 /usr/local/hashml/lib/python3.10/site-packages/hashml/
platform/static
```

Step 3: Start the HashML Platform service

Run the following commands in the hashml user environment.

- Start the Ray service.

```
ray start --head
ray status # Checks the status.
```

- Start the model backend service.

```
serve deploy /usr/local/hashml/lib/python3.10/site-packages/hashml/
platform/core/services.yml
serve status # Checks the status.
```

- Start the API backend service.

```
systemctl start hashml-platform.service
```

Step 4: Access the HashML Platform console in your browser

Open a browser and navigate to `http://<ip>:<port>/hp` to access the HashML Platform login page. Here:

- <ip> is the IP address of the machine where HashML Platform is running.
- <port> is the service port configured in the `service_port` field of the `/etc/hashml/platform.toml` file.

The default login username and password are both `admin`.

Manage HashML Platform services

The following are commonly used commands to manage HashML Platform:

- Check the service status:

```
systemctl status hashml-platform.service
```

- Stop the service:

```
systemctl stop hashml-platform.service
```

- Restart the service (run this after updating the configuration):

```
systemctl restart hashml-platform.service
```

- Enable the service to start on boot:

```
sudo systemctl enable hashml-platform.service
```

- View logs in real time:

```
sudo journalctl -u hashml-platform.service -f
```

- View logs within a specific time range, for example from 2025-04-22 to 2025-04-23:

```
sudo journalctl -u hashml-platform.service --since "2025-04-22" --until "2025-04-23"
```

- View all logs:

```
sudo journalctl -u hashml-platform.service -e
```

Additional notes

- For assistance with obtaining RPM packages or configuring the platform, contact SynxDB technical support.
- It is recommended to record logs during installation and deployment to help with troubleshooting.

6.2 Use the HashML Platform

HashML Platform is a high-performance distributed computing platform designed for enterprise-grade AI applications. It integrates the strengths of open-source technologies and enhances them with enterprise-grade features. The platform offers a complete end-to-end solution covering data preparation, feature engineering, model training, and production deployment.

Core features of HashML Platform

- **Visual modeling:** Offers an intuitive graphical interface that lets users train models by selecting algorithms, datasets, and setting hyperparameters. After training, models can be deployed as services with a single click. The platform also supports A/B testing and staged rollouts.
- **Notebook-based development:** Comes with a built-in professional-grade Jupyter environment. Developers can write and debug Python code flexibly, enabling customization and experimentation with complex algorithms.
- **Algorithm and model library:** Includes a rich collection of built-in algorithms such as Logistic Regression and XGBoost, deep learning models like MLP, ResNet, and Bert, and fine-tuning algorithms for large language models (supporting QLoRA and Unsloth). Users can also extend the library with custom algorithms.

Use cases for HashML Platform

- **Private model training for enterprises:** Enables organizations to train machine learning models on sensitive internal data without leaving the enterprise network. The platform ensures secure end-to-end management and high-performance training, helping businesses build accurate, compliant, and domain-specific models.
- **Fine-tuning large language models:** Integrates efficient fine-tuning methods such as QLoRA and Unsloth to support popular LLMs. Enterprises can quickly adapt LLMs to specialized domains using few-shot learning, reducing training costs while improving task accuracy and semantic understanding.
- **Model deployment:** Allows trained models to be deployed as RESTful API services in one click. It seamlessly integrates with business systems and supports A/B testing, phased rollouts,

and auto-scaling to maintain stable performance under high concurrency.

Use the HashML Platform console

Log into the HashML console

After deploying the HashML Platform (see [Deploy HashML Platform](#)), open a browser and go to `http://<node_ip>:<port>/hp/`. Enter your username and password, then click **Login**.

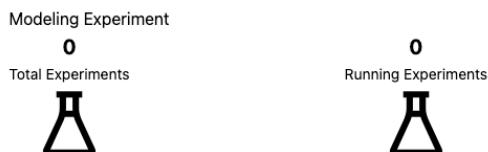
Tip

- <ip> is the IP address of the machine running HashML Platform. <port> is the service port configured in the `service_port` field of the `/etc/hashml/platform.toml` file.
- The default username is `admin`, and the default password is also `admin`.

After logging in, you can see an overview of modeling experiments and model services on the [Overview](#) page.

Welcome to HashML Machine Learning Platform

Model Training



Model Service



Train a model

In the **Model Training** module in the left navigation menu, you can create training tasks, track their progress and status, and perform data insights.

Create a training task

On the **New Task** page, follow the instructions to fill in or select the required information, such as task name, model algorithm, training configuration, and description. Then click **Create Task** to confirm.

To view details about specific parameters, hover your mouse over the ? icon next to the parameter.

Create Training Task

Task Name

Model Algorithm

Training Configuration

num_workers	1	?
num_boost_round	100	?
early_stopping_rounds	10	?
objective	reg:squarederror	?
max_depth	6	?
lambda	1	?

Data Configuration

DB Name	testdb	?
Training Set	docmind_test_split	?
Validation Set	Please select a validation set	?
Target column	Please select	?
Feature columns	Please select	?

Description

Create Task

Once a task is created, you can check its status on the **Task List** page. After training is complete, you can deploy and manage the model in the **Model Factory** module.

View training tasks

The **Task List** page displays all tasks. You can view detailed task information, including status, creation time, last updated time, creator, model algorithm, and more. Click **Get Logs** to see detailed logs for a task.

You can also stop or delete a task from this page.

Task Name	Description	Create Time	Update Time	Creator	Model Algorithm	Training Config	Status	Actions
test-traini...		2025-04-24 15:20:11	2025-04-24 15:20:11	wpy	XGBoost	{"alpha":0,"lambda":1,"m...	RUNNING	<button>Get Logs</button> <button>Stop Training</button> <button>Delete</button>

Model Algorithm

- XGBoost
- LightGBM
- MLP
- SVC
- SVR
- LogisticRegression
- LinearRegression
- RNNBlockRegressor

Data exploration and insights

The **Data Insight** page provides visual data analysis tools.

1. Select a dataset.
2. View dataset summary: Click **Data Overview** to see dataset characteristics such as data summary, statistical description, missing value report, and correlation matrix.

The screenshot shows the Data Insight interface with the following configuration:

- DB Name:** testdb
- Dataset:** diabetes
- X Axis:** Please select X axis
- Y Axis:** 请选择y轴
- Chart Type:** Line Chart
- Data Overview:** A large table titled "Data Overview" is displayed under the heading "1. Data Summary". It contains two sections: "1. Data Summary" and "2. Data Description".

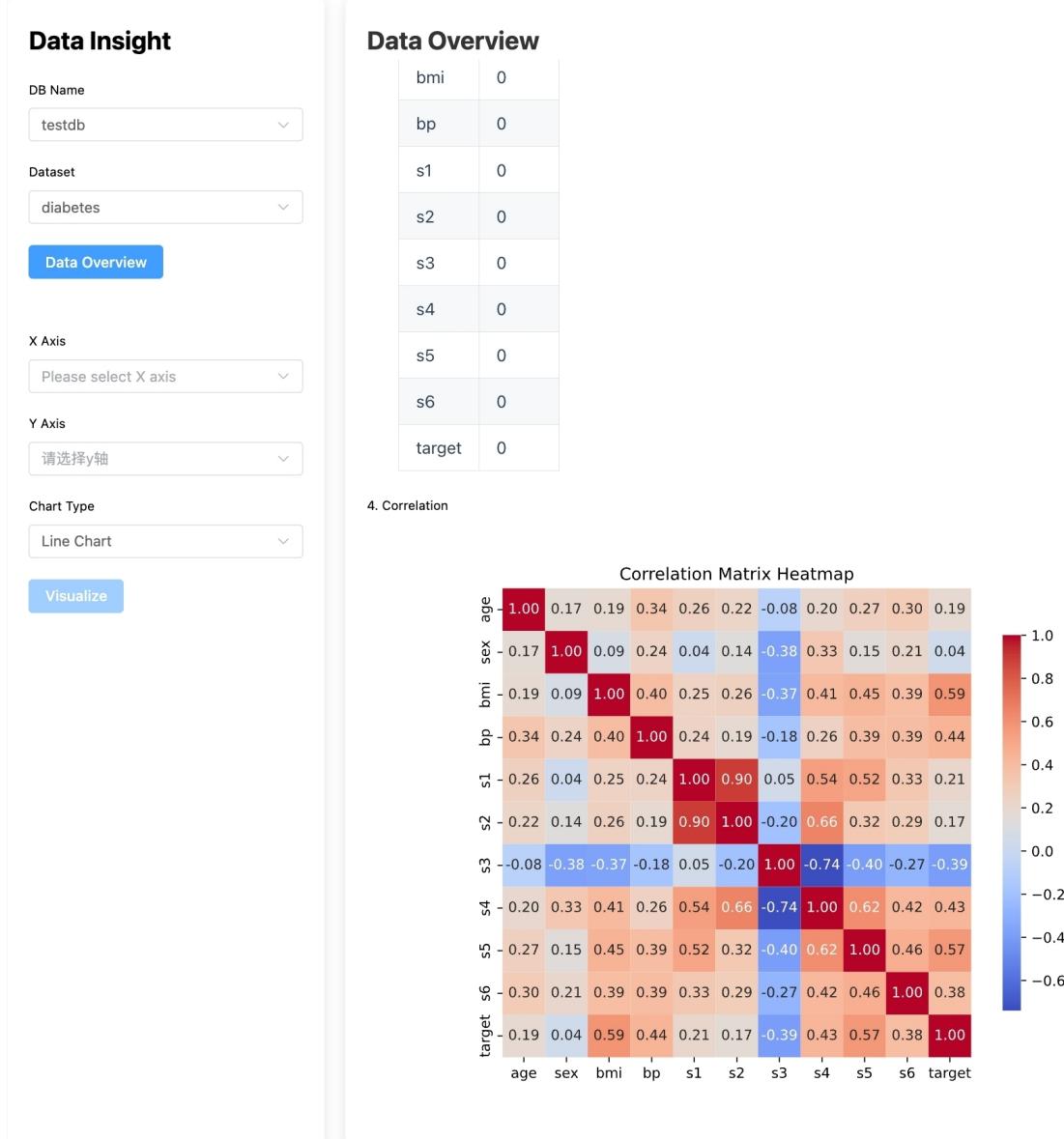
1. Data Summary:

#	Column	Non-Null Count	Dtype
1	age	442	float64
2	sex	442	float64
3	bmi	442	float64
4	bp	442	float64
5	s1	442	float64
6	s2	442	float64
7	s3	442	float64
8	s4	442	float64
9	s5	442	float64
10	s6	442	float64
11	target	442	float64

2. Data Description:

	age	sex	bmi	bp	s1	s2	s3	s4	s5	s6	target
count	442	442	442	442	442	442	442	442	442	442	442
mean	2.51182e-19	1.15544e-17	-2.24305e-16	-4.79757e-17	-1.41918e-17	3.96867e-17	-5.77718e-18	-9.16813e-18	9.30628e-17	1.06752e-17	152.133
std	0.047619	0.047619	0.047619	0.047619	0.047619	0.047619	0.047619	0.047619	0.047619	0.047619	77.093
min	-0.107226	-0.0446416	-0.0902753	-0.112399	-0.126781	-0.115613	-0.102307	-0.0763945	-0.126097	-0.137767	25
25%	-0.0372993	-0.0446416	-0.0342291	-0.0366561	-0.0342478	-0.0303584	-0.0351172	-0.0394934	-0.0332456	-0.033179	87
50%	0.00538306	-0.0446416	-0.00728377	-0.00567042	-0.00432087	-0.00381907	-0.00658447	-0.00259226	-0.00194717	-0.0010777	140.5
75%	0.0380759	0.0506801	0.031248	0.0356438	0.028358	0.0298444	0.0293115	0.0343089	0.0324323	0.0279171	211.5

3. Visualize specific variables: Choose X-axis, Y-axis, and chart type, then click **Visualize** to explore selected data visually.



Model factory

The **Model Factory** module allows you to import external models, deploy them as services, and manage existing models.

Import a model

HashML Platform supports importing models trained outside the platform.

The screenshot shows the 'Import Model' form. At the top is the title 'Import Model'. Below it is a 'Model Name' field with the placeholder 'Please enter the model name'. Underneath is a 'Description' field containing 'XGBoost'. The third section is 'Import Model File', which includes a dashed rectangular area for file upload with the instruction 'Drag the file here, or click to upload'. At the bottom is a large blue 'Submit' button.

1. On the **Import Model** page, add a description for the model and select the corresponding algorithm.
2. Upload the model file by clicking or dragging it into the upload area.
3. Click **Submit**. The imported model will appear on the model list page.

Deploy a model

You can view existing models and deploy them with one click on the **Model Factory > Model List** page. The list displays model descriptions, creation and update times, source, algorithm, and other information.

To view more details about a model, click the model name in the **Model Name** column to open a model card with additional information.

```
{
  "model_id": 4,
  "model_name": "XGBoost_66e5dbae2a7d47139ca008e18e35c6f0",
  "model_type": "xgboost",
  "algorithm": "XGBoost",
  "description": "",
  "model_config": {
    "num_features": 1,
    "feature_names": [
      "mean_perimeter"
    ],
    "feature_types": [
      "float"
    ]
  },
  "created_at": "2025-04-24T15:20:39.289Z",
  "updated_at": "2025-04-24T15:20:39.289Z"
}
```

To deploy a model service, click **Deploy**:

1. Enter the model name, ID (optional), version (optional), and other relevant information.
2. Optionally specify the traffic ratio to assign to this model. For example, entering 40 means

the model will handle 40% of the traffic.

Tip

This step allows you to define the percentage of online traffic that the newly deployed model will process. It is a key part of implementing a canary release strategy, which helps minimize risk by testing the new model with a small portion of traffic before full rollout.

For example, if you enter 40, the new model will handle 40% of incoming requests, while the remaining 60% will continue to be processed by the currently deployed model (if any). To route all traffic to the new model, enter 100. To temporarily assign no traffic (for example, for internal testing), enter 0.

3. Click **Confirm Deploy**. You can also delete unnecessary models from the **Model List** page.

Deploy Model Service ×

Service Name

Service ID

Version

Ray Service

Weight

Cancel Confirm Deploy

View deployed services

You can view deployed model services on the **Model Services** page.

To view detailed information about a service, click the service ID to open a service card, as shown below:

```
{
  "service_id": "s1",
  "version": "v1",
  "weight": 0.3,
  "model_name": "XGBoost_14b6e381467348c3982090944c0e4fef",
  "model_config": {
    "num_features": 8,
    "feature_names": [
      "longitude",
      "latitude",
      "housing_median_age",
      "total_rooms",
      "total_bedrooms",
      "population",
      "median_income",
      "median_house_value"
    ],
    "feature_types": [
      "float",
      "float",
      "float",
      "float",
      "float",
      "float",
      "float"
    ]
  },
  "total_call_count": 0,
  "successful_call_count": 0
}
```

Use notebook-based modeling

HashML Platform provides an interactive notebook environment for developing large models. In the **Notebook Modeling** module, you can create and manage notebooks.

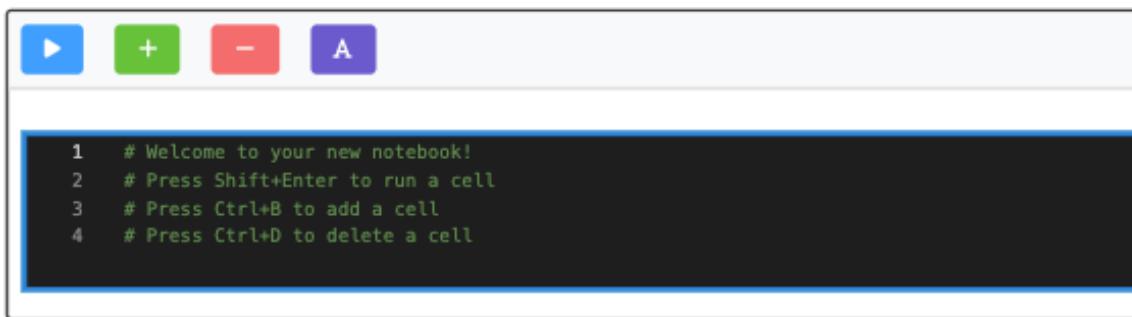
Create a notebook

1. In the left navigation menu, go to **Notebook Modeling > Create notebook**.
2. Enter a name for the notebook, then click **Create Notebook**.

Manage and operate notebooks

1. In the left navigation menu, go to **Notebook Modeling > Notebook List**.
2. Locate the target notebook in the list, then click **Open** or **Delete** as needed.

After opening a notebook, use the buttons in the upper left corner of the editor to perform operations:



Chapter 7

Manage System

7.1 View Monitoring Data Using the Web Console

CBCC is a monitoring platform for SynxDB clusters. It provides dashboards and metrics for both the cluster and the database, helping users monitor and diagnose the current status of their systems more effectively. CBCC supports monitoring multiple database clusters, and users can switch between clusters in the console.

Install Monitoring Console

CBCC is a console tool for monitoring SynxDB clusters and databases. It provides dashboards and metrics to help users diagnose the current state of the cluster and databases.

This document describes how to install and deploy the CBCC console components to enable monitoring for SynxDB clusters.

The CBCC console consists of two main components: CBCC Server and CBCC Agent. The CBCC Server is the dashboard server that receives and displays metrics in a unified view. The CBCC Agent runs on each cluster node, and collects and reports monitoring data.

The CBCC Agent monitors local system and services on each node, and reports data to the CBCC Server via the gRPC port. The CBCC Server is responsible for receiving, processing, and visualizing the operational status and performance metrics of all nodes in the cluster, and presents the results in the monitoring dashboard.

Prerequisites

- A SynxDB database cluster has been installed and configured, and `gpperfmon` is enabled on all nodes.
- The installation package and dependency packages of the CBCC components are available.
- The target servers run Linux and support systemd.
- `root` or `sudo` privileges.
- `curl` (used for health checks).
- `glibc 2.17` or later.
- Hardware requirements: CPU of at least 1 core, RAM of at least 2 GB.

Tip

`gpperfmon` is a performance monitoring component that helps administrators and developers view, analyze, and diagnose various metrics during cluster runtime. In SynxDB, you can enable `gpperfmon` following these steps:

1. Enable the monitoring service:

```
gpperfmon_install --enable --password <database_password> --port
<database_port>
```

2. Restart the database:

```
gpstop -ar
```

Step 1: Deploy the Server component

Tip

CBCC Server uses the following default administrator credentials:

- Username: admin
- Password: admin

To change these default credentials, edit the `config/cbcc-server/application.yml` configuration file in the installation package before running the installation script. Update the following settings:

```
spring:
security:
  user:
    name: admin      # Changes to your preferred username
    password: admin  # Changes to your preferred password
```

Upload the CBCC Server installation tool to the target server, navigate to the tool directory, and run the following command:

```
sudo ./deploy.sh install
```

This command performs the following tasks:

1. Creates required directories.
2. Copies binaries and configuration files.

3. Adds systemd service files.
4. Starts the CBCC Server, Prometheus, and AlertManager services.

After deploying the CBCC Server, open a browser and access the control panel at:

```
http://<CBCC_SERVER_IP>:8080
```

<CBCC_SERVER_IP> refers to the IP address of the server where the CBCC Server is installed.

The following table lists the default ports used by the CBCC Server.

Default ports

Component	Port	Description
CBCC Server UI	8080	Web dashboard access port
CBCC gRPC	28080	Port used by Agents to report data
Prometheus	9090	Prometheus dashboard
AlertManager	9093	Alert management dashboard

Start and maintain the Server

Example commands:

```
# Starts all services
sudo ./deploy.sh start

# Starts a specific service
sudo ./deploy.sh start cbcc-server
sudo ./deploy.sh start prometheus
sudo ./deploy.sh start alertmanager

# Stops all services
sudo ./deploy.sh stop

# Stops a specific service
sudo ./deploy.sh start prometheus

# Restarts all services
```

(continues on next page)

(continued from previous page)

```
sudo ./deploy.sh restart

# Restarts a specific service
sudo ./deploy.sh restart alertmanager

# Checks the status of all services
sudo ./deploy.sh status

# Checks the status of a specific service
sudo ./deploy.sh status alertmanager

# Checks the version of the monitoring server package and its components
sudo ./deploy.sh version

# Checks the version of a specific service, for example, alertmanager
sudo ./deploy.sh version alertmanager
```

Configuration file paths

- The main configuration file of CBCC Server is located at /etc/cbcc/cbcc-server/application.yml. This file contains important server operation settings.

Tip

You can customize the server logo, icon, and name by adding the following configuration to the /etc/cbcc/cbcc-server/application.yml file:

```
cbcc:
  title: 'custom name'          # custom display name
  distribution:
    distributor: 'distributor' # only supports SynxDB or BlueBerry
```

- Prometheus service configuration files:
 - Main configuration: /etc/cbcc/prometheus/prometheus.yml
 - Alert rules: /etc/cbcc/prometheus/alert_rule.yml

- Scrape targets: /etc/cbcc/prometheus/scrape_config.yml
- AlertManager service configuration file: /etc/cbcc/alertmanager/alertmanager.yml

Step 2: Deploy the Agent on each node

Upload the CBCC Agent installation tool to each node in the SynxDB cluster, then go to the tool directory on each node and run:

```
sudo ./deploy.sh install --console-url <CBCC_SERVER_IP>:28080
```

<CBCC_SERVER_IP> is the gRPC IP address of the CBCC Server. For example:

```
sudo ./deploy.sh install --console-url 192.168.0.1:28080
```

This command performs the following tasks:

1. Creates necessary directories.
2. Copies binaries and configuration files.
3. Sets up systemd service files.
4. Starts the Agent service.

Attention

The `console-url` parameter is critical for communication between the Agent and the CBCC Server. This parameter includes the server IP address and the gRPC port (`CBCC_SERVER_GRPC_PORT`, default: 28080).

The format of the `console-url` parameter is `<server_ip>:<grpc_port>`, for example: 192.168.0.1:28080.

The console URL is saved in the Agent configuration file located at `/etc/cbcc/cbcc-agent/config.yml`, for example:

```
# Console configuration
console:
  # Console service address
  url: "192.168.0.1:28080"
```

Start and maintain the Agent

Example commands:

```
# Starts the Agent service
sudo ./deploy.sh start

# Stops the Agent service
sudo ./deploy.sh stop

# Restarts the Agent service
sudo ./deploy.sh restart

# Checks the service status
sudo ./deploy.sh status

# Checks the version information of the Agent package and its components
sudo ./deploy.sh version
```

Next step

After deploying CBCC, you can open the dashboard to view the status of each node in the cluster, including performance metrics such as CPU, memory, and disk I/O. For details, see [View Monitoring Data Using the Web Console](#).

Troubleshooting

Server service fails to start

1. Check the service status:

```
sudo ./deploy.sh status cbcc-server
```

2. View detailed logs:

```
sudo journalctl -u cbcc-server -f
```

3. Verify port availability:

```
sudo netstat -tulpn | grep 8080  
sudo netstat -tulpn | grep 28080
```

4. Check for errors in the configuration file:

```
cat /etc/cbcc/cbcc-server/application.yml
```

Server health check fails

If the health check fails:

1. Make sure the service is running.
2. Check if the ports are accessible.
3. Check the service logs for errors:

```
sudo journalctl -u cbcc-server -f
```

4. Verify the configuration file is correct.

Agent service fails to start

1. Check the service status:

```
sudo ./deploy.sh status
```

2. View detailed logs:

```
sudo journalctl -u cbcc-agent -f
```

3. Verify the configuration:

```
cat /etc/cbcc/cbcc-agent/config.yml
```

4. Check if the console URL is correctly set and the server is reachable:

```
ping <server_ip>
telnet <server_ip> <grpc_port>
```

Agent health check fails

If the health check fails:

1. Make sure the service is running.
2. Check whether the CBCC Server is accessible.
3. Check the service logs for errors:

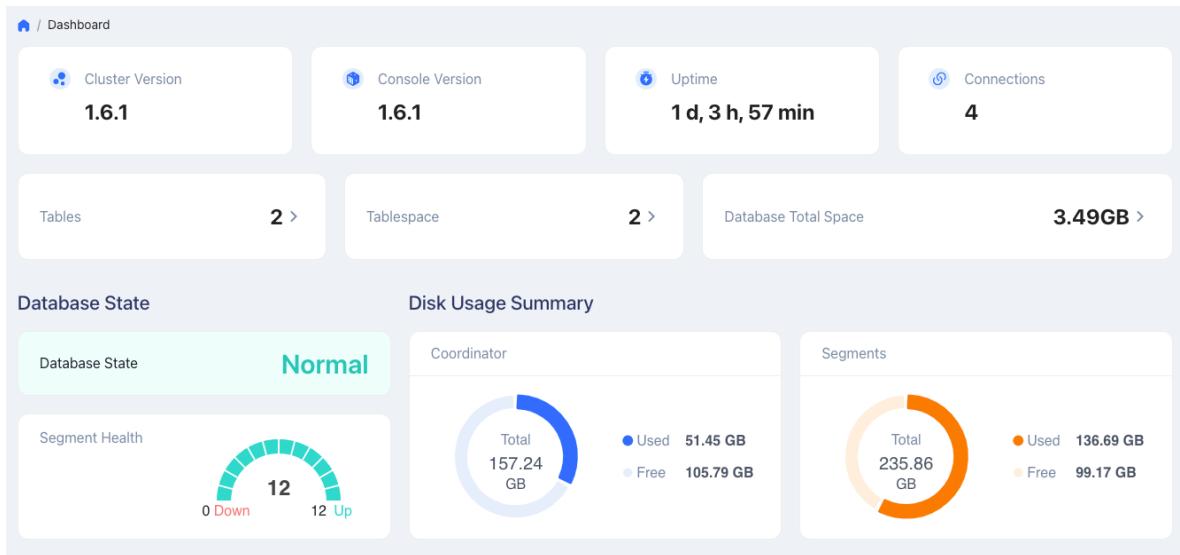
```
sudo journalctl -u cbcc-agent -f
```

4. Verify that the console URL in the configuration file is correct.

View Cluster Information

Steps

1. Access `http://<ip>:8080/` to log into the CBCC console.
2. Click **Dashboard** in the left navigation menu to view the cluster overview.



Display item	Description
Overview	Cluster version, console version, cluster uptime, the number of cluster connection sessions, total number of tables, number of tablespaces, and total disk in use. Note that some metrics contain a detailed page. Click on the metrics to view more.
Database State	The overall status of the database and the number of normal and abnormal segments.
Disk Usage Summary	Disk usage of the coordinator and segments.



Display item	Description
CPU	Shows the average CPU and maximum CPU usage of the cluster in the past 2 hours. - The orange and blue lines represent the system processes and user processes, respectively. Click the legend in the upper left corner to show or hide the line. - Hover the cursor over the chart to display the CPU usage percentage at specific time points.
Memory	Shows the average memory usage percentage of the cluster in the past 2 hours. - Hover the cursor over the chart to display the average memory and maximum memory usage percentage at specific time points.

i Note

CBCC supports monitoring multiple database clusters. You can switch between clusters using the dropdown menu in the top-right corner of the console.

Cluster Status and Metrics

You can view the near-real-time status and metric data of the cluster on the **Cluster Metrics** and **Cluster Status** pages.

i Note

CBCC supports monitoring multiple database clusters. You can switch between clusters using the dropdown menu in the top-right corner of the console.

Access the pages

To access the **Cluster Metrics** and **Cluster Status** pages, you need to:

1. Access the CBCC dashboard in your browser via
`http://<cluster_node_IP>:8080/`.
2. Click **Cluster Metrics** or **Cluster Status** in the left navigation menu to enter the page.

View the overall status and data of the cluster



The **Cluster Metrics** page displays the cluster's CPU, memory, disk I/O, and network data.

You can adjust the time range of the data displayed through the drop-down menu on the upper-right corner of the page. Available time range options are “2 hours”, “6 hours”, “1 day” and “7 days”. Data calculation time units vary with the selected time range.

The charts have time as the X axis and numerical value or percentage as the Y axis. When you hover over a certain time point, the data of that time will pop up. You can view data from different time points by moving the cursor horizontally. Corresponding legends are provided in the upper right corner of each chart.

Cluster CPU usage status

The chart displays the following metrics:

- Average, skewness, and maximum value of the total CPU usage (%) of all user processes.
- Average, skewness, and maximum value of the total CPU usage (%) of all system processes.
- Average, skewness, and maximum of the total CPU usage (%).
- Name of the busiest host.

Cluster memory usage status

The chart displays the following metrics:

- Average, skewness, and maximum of the total memory in use (%).
- Average, skewness, and maximum of the total buffer and cache memory (%).
- Average, skewness, and maximum value of the total available memory (%).
- Name of the busiest host

Cluster disk I/O rate

The chart displays the following indicators:

- Average, skewness, and maximum value of the total disk read rate (MB/s).
- Average, skewness, and maximum value of the total disk write rate (MB/s).
- Name of the busiest host by disk read.
- Name of the busiest host by disk write.

Attention

The part above the X axis of the chart displays disk read data, and the part below the X axis displays disk write data.

Network I/O rate

- Average, skewness, and maximum value of the total network read rate (MB/s).
- Average, skewness, and maximum value of the total network write rate (MB/s).
- Name of the busiest host by network read.
- Name of the busiest host by network write.

i Note

The part above the X axis of the chart displays network read data, and the part below the X axis displays network write data.

View the status and data of nodes and hosts

The **Cluster Status** page displays the status and data of the hosts and nodes in the cluster. You can switch between host and segment tabs from the upper-left corner of the page.

Coordinator and Standby Hosts							
Hostname	CPU	User/System/Idle	Memory in Use(GB)	Disk R(MB/s)	Disk W(MB/s)	Net R(MB/s)	Net W(MB/s)
i-f5ots2ip	● User ● System ● Idle	1.16% 0.82% 97.99%	Used Free	1.89 GB 13.62 GB	0 MB/s	0.1 MB/s	0.01 MB/s 0.03 MB/s

Segment Hosts							
Hostname	CPU	User/System/Idle	Memory in Use(GB)	Disk R(MB/s)	Disk W(MB/s)	Net R(MB/s)	Net W(MB/s)
i-c7hg27f	● User ● System ● Idle	0.02% 0.07% 99.9%	Used Free	1.29 GB 14.22 GB	0 MB/s	0.01 MB/s	0 MB/s 0 MB/s
i-f5ots2ip	● User ● System ● Idle	1.16% 0.82% 97.99%	Used Free	1.89 GB 13.62 GB	0 MB/s	0.1 MB/s	0.01 MB/s 0.03 MB/s

Host metrics

The **Host Metrics** tab displays data of the coordinator host, its backup host, and the segment hosts:

i Note

Users can search for a specific segment host by hostname.

Field	Description
Hostname	The name of the coordinator or segment host
CPU User/System/Idle	% of the user processes CPU usage, system processes CPU usage, and idle CPU
Memory in Use (GB)	In-use and available memory
Disk R (MB/s)	Disk read rate
Disk W (MB/s)	Disk write rate
Net R (MB/s)	Network read rate
Net W (MB/s)	Network write rate

Segment status

The **Segment Status** tab displays the status and data of each segment.

The top of the tab shows the overall status of the database, segment count, segment-specific host count, and a segment status indicator.

The table displays the following metrics and status:

Field	Description
Hostname	The name of the segment host
Address	The address of the segment on the segment host
Port	The listening TCP port of the segment host
DB ID	Database ID
Content ID	The content identifier of the segment
Status	Segment status. Values: Up or Down
Role	The current role of the segment. Values: p (Primary) or m (Mirror)
Preferred Role	The role that is set for the segment when it is initialized. Values: p (Primary) or m (Mirror)
Replication Mode	The synchronization status of the segment with its mirror copy. Values: s (Synchronized) or n (Not In Sync)

View Database Object Information

CBCC provides information of database objects. On the **Tables** page, you can view detailed information about tables in the database, such as the schema to which the table belongs, the table name, whether the table contains partitions, the table size, the user, and the estimated number of rows. An example page is as follows:

Schema	Relation Name	Include Partitions	Size	User	Est. Rows	Last Analyzed
public	orders	false	0B	gpadmin	0	0001-01-01T08:05:43+08:00
public	products	false	32KB	gpadmin	0	0001-01-01T08:05:43+08:00
public	users	false	0B	gpadmin	0	0001-01-01T08:05:43+08:00

To access the **Tables** page, you need to:

1. Access the CBCC dashboard in your browser via

`http://<cluster_node_IP>:8080/.`

2. Click **Tables** in the left navigation menu to enter the page.

View table objects

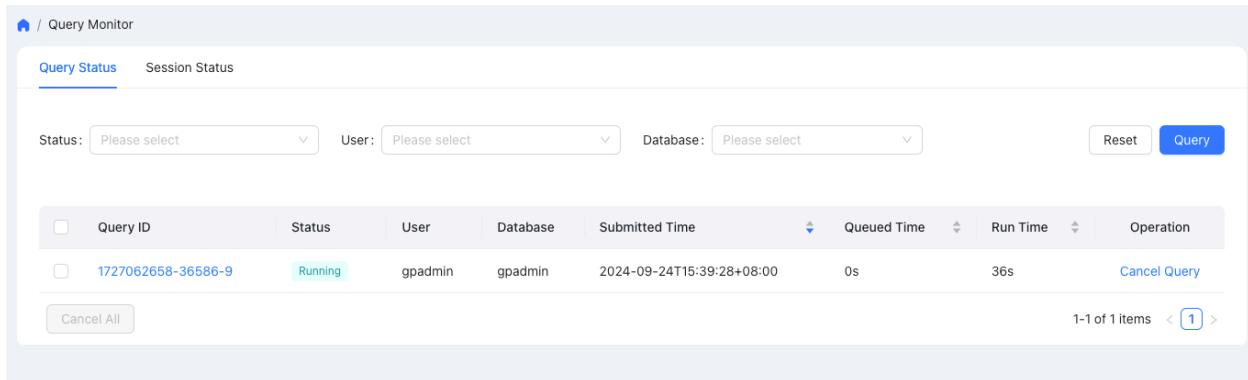
To query the information of a table, you can fill in the drop-down option boxes according to the database, the schema, the table name, and the user. Then click **Query**. The **User** box supports multiple selections.

The fields for the table information are described as follows:

Field name	Description
Schema	The schema to which the table belongs.
Relation Name	The table or view name.
Include Partitions	Indicates whether the table contains partitions. <code>False</code> means no, and <code>true</code> means yes.
Size	The storage size that the table occupies.
User	The database user to which the table belongs.
Est. Rows	The estimated number of rows.
Last Analyzed	The last time the table was analyzed for updated statistics.

View SQL Monitoring Information

CBCC provides monitoring information for in-progress and history SQL statements in the database. On the **Query Monitor** and **Query History** pages, you can see the execution status and details of each SQL statement, and the status of each database session.



The screenshot shows the CBCC Query Monitor interface. At the top, there are three dropdown filters: Status (Please select), User (Please select), and Database (Please select). Below these are two buttons: 'Reset' and 'Query'. The main area displays a table with the following columns: Query ID, Status, User, Database, Submitted Time, Queued Time, Run Time, and Operation. A single row is visible, representing a query with ID 1727062658-36586-9, status Running, user gpadmin, database gpadmin, submitted at 2024-09-24T15:39:28+08:00, queued for 0s, and run for 36s. There is a 'Cancel Query' link next to the row. At the bottom left is a 'Cancel All' button, and at the bottom right is a pagination indicator showing 1-1 of 1 items with a page number of 1.

i Note

CBCC currently doesn't support displaying information of the Prepare statements.

Access the page

To access the **Query Monitor** page, you need to:

1. Access the CBCC dashboard in your browser via
`http://<cluster_node_IP>:8080/`.
2. Click **Query Monitor** in the left navigation menu to enter the page.

View SQL execution status

To view the execution status of a SQL statement that is being executed, click the **Query Status** tab. You can view the information of history queries on **Query History**.

View in-progress SQL

To check the execution status of an ongoing SQL statement, click on the **Query Execution Status** section.

Note

You can pause or resume monitoring of in-progress query status by using the following commands:

```
psql gpperfmon -- Connect to the "gpperfmon" data warehouse.  
select query_state_pause(); -- Pause monitoring of ongoing queries.  
select query_state_resume(); -- Resume monitoring of ongoing queries.
```

In the search area, you can filter by the execution status of the SQL statement, user, and database. Click **Query** to search. The **User** filter supports multiple selections.

The options in the **Status** dropdown are described as follows:

Option Name	Description
Running	SQL statement is executing.
Cancelling	SQL statement is being cancelled.
Unknown	SQL statement execution status is unknown.

After clicking **Query**, a list of SQL statements will be displayed in the area below.

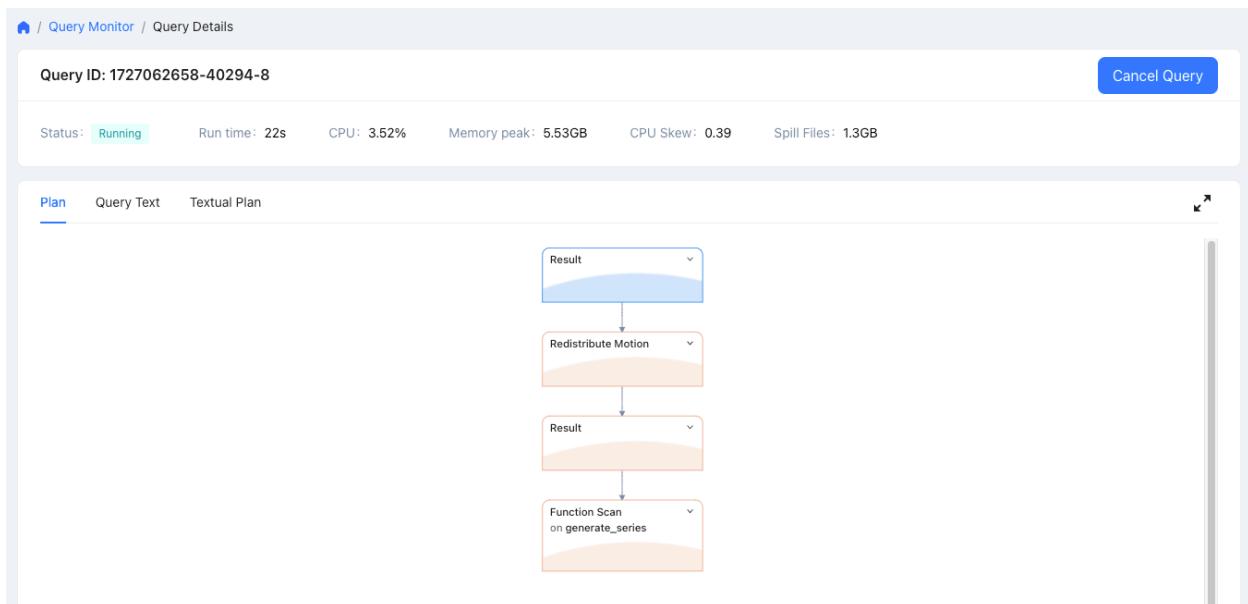
View SQL Details

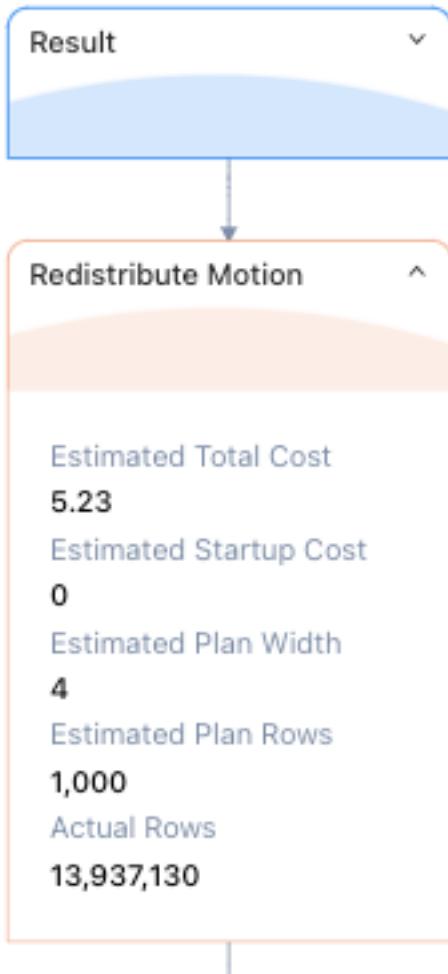
To view the details of a specific SQL statement, click on the query ID of that SQL statement to enter the details page.

The details page displays information about the SQL execution, and you can click on different tabs to view the query plan diagram, SQL text, and query plan text. In the query plan diagram section, ongoing queries are dynamic, and the animation effect disappears once the query ends.

You can click on a module to view specific information, which updates automatically as the query progresses.

Example:





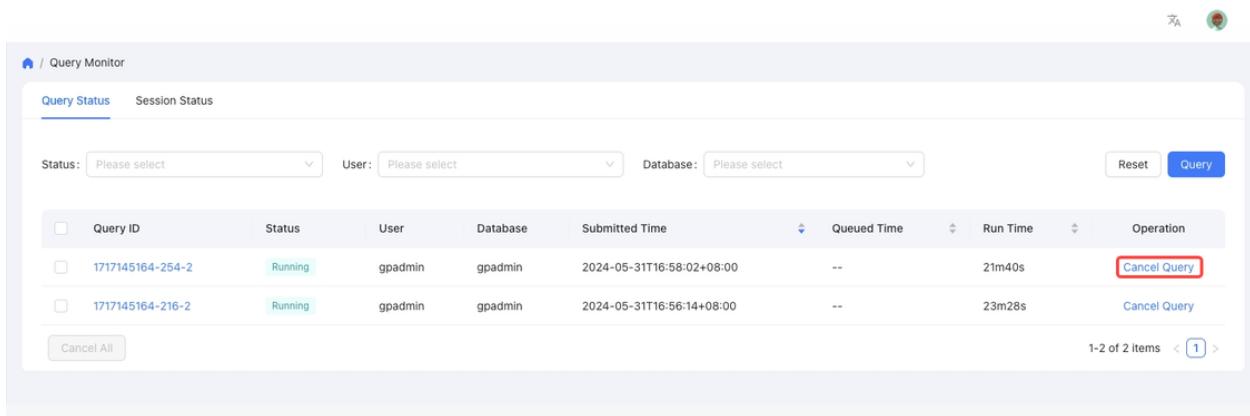
In the **Query Plan Text** section, you can view the query plan information in text format.

i Note

In the **Query History** page, you can see SQL queries that took longer than a specific duration to execute. By default, the system records queries that take longer than 20 seconds. You can modify this value by changing the `min_query_time` parameter in the `gpperfmon.conf` configuration file on the Coordinator node. You can print the directory of this configuration file using `echo $COORDINATOR_DATA_DIRECTORY`.

Cancel SQL execution

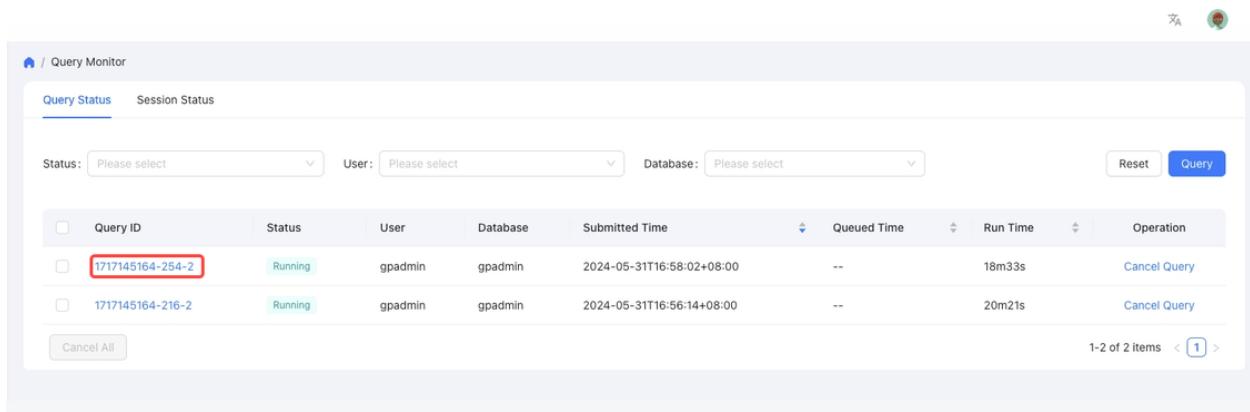
To cancel one or more SQL statements, locate the **Operation** column of the corresponding SQL statement in the SQL list, and then click **Cancel Query**.



<input type="checkbox"/> Query ID	Status	User	Database	Submitted Time	Queued Time	Run Time	Operation
<input type="checkbox"/> 1717145164-254-2	Running	gpadmin	gpadmin	2024-05-31T16:58:02+08:00	--	21m40s	Cancel Query
<input type="checkbox"/> 1717145164-216-2	Running	gpadmin	gpadmin	2024-05-31T16:56:14+08:00	--	23m28s	Cancel Query

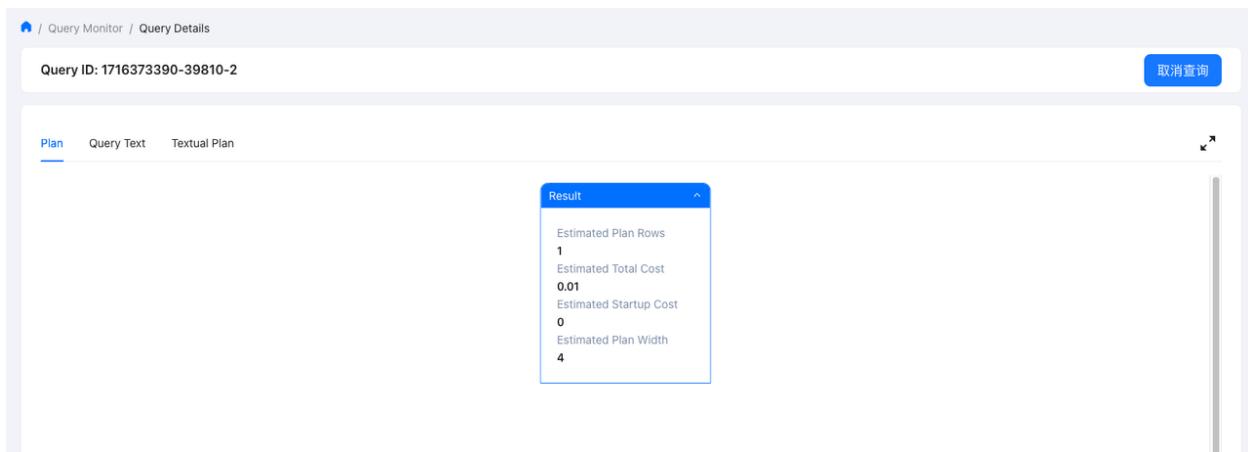
View SQL details

To view the details of a SQL statement, click the query ID of the SQL statement, and then enter the details page.



<input type="checkbox"/> Query ID	Status	User	Database	Submitted Time	Queued Time	Run Time	Operation
<input type="checkbox"/> 1717145164-254-2	Running	gpadmin	gpadmin	2024-05-31T16:58:02+08:00	--	18m33s	Cancel Query
<input type="checkbox"/> 1717145164-216-2	Running	gpadmin	gpadmin	2024-05-31T16:56:14+08:00	--	20m21s	Cancel Query

The details page displays the details of SQL execution. You can click different tabs to view the query plan diagram, SQL text, and the query plan text of the SQL statement. An example is as follows:



View session status

To view session status in the database, click the **Session Status** tab on the **Query Monitor** page.

A list of real-time sessions running in the database is displayed, including session ID, execution status, the user who operates, the database where the session is running, the start time, the application, and idle time.

<input type="checkbox"/>	Session ID	Status	User	Database	Start Time	Application	Idle Time	Operation
<input type="checkbox"/>	708	Idle	gpadmin	gpadmin	2024-05-31T17:18:43+08:00	psql	--	
<input type="checkbox"/>	553	Idle	gpmon	gpadmin	2024-05-31T17:12:02+08:00	--	--	
<input type="checkbox"/>	254	Active	gpadmin	gpadmin	2024-05-31T16:57:50+08:00	psql	--	Cancel Query
<input type="checkbox"/>	216	Active	gpadmin	gpadmin	2024-05-31T16:56:04+08:00	psql	--	Cancel Query

To view the details of a session, in the search area, fill in the corresponding drop-down option box according to the execution status, user, database, and application name. Then click **Query** to search. The **User** box supports multiple selections.

The options in the **Status** drop-down box are described as follows:

Option name	Description
Active	The backend is running the session.
Idle	The backend is waiting for new client commands.
Idle in transaction (aborted)	The backend is in a transaction, but currently, no query is running.
Fastpath function call	The backend is executing the fast path function.
Disabled	The status is reported when <code>track_activities</code> is disabled in the backend.
Unknown	The session status is unknown.

After clicking **Query**, a list of sessions is displayed in the area below, and you should be able to find the target session from the list.

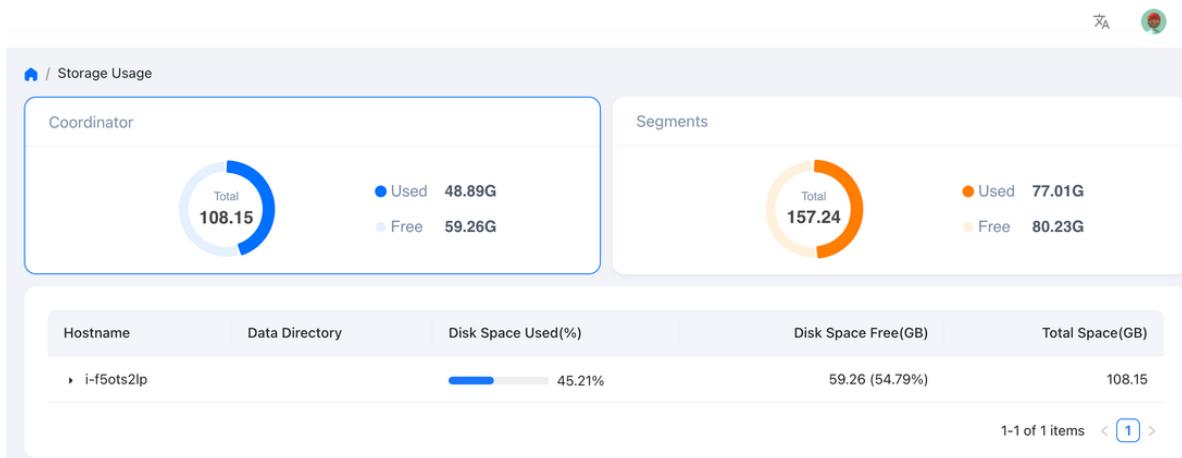
By default, the session list is sorted by **Start Time** in descending order. You can click **Start Time** to sort in ascending order, or sort by **Idle Time**. The description of the fields in the list is as follows:

Option name	Description
Session ID	Identifies the unique ID of the session being executed in the database.
Status	The status of the session.
User	The user who performs the session operation.
Database	The database where the session is running.
Start Time	The start time of the session.
Application	The client application for executing the session.
Idle time	The idle time of the session.
Operation	For running sessions, you can click Cancel Query to cancel the session.

View Storage Information

Steps

1. Access `http://<ip>:8080/` to log into the CBCC console.
2. Click **Storage Usage** in the left navigation menu to view the storage overview information.



- Click the **Coordinator** and **Segments** cards at the top of the page to view the disk usage of the machines hosting the coordinator and segment nodes respectively.

.. list-table:: :header-rows: 1 :align: left

* - Display item - Description * - Host Name - Host names of the coordinator or segments. There may be multiple hosts. * - Data Directory - The mounting point and path information of each machine. * - Used Disk Space (- The disk usage of each machine and its percentage. * - Disk Space Free (GB) - The available amount of disks on each machine and its percentage. * - Total Space (GB) - The total capacity of disks on each machine.

.. raw:: latex

- Click the small triangle to the left of the **Hostname** to expand and view the disk usage under different mount points.

View and Create Alert Rules

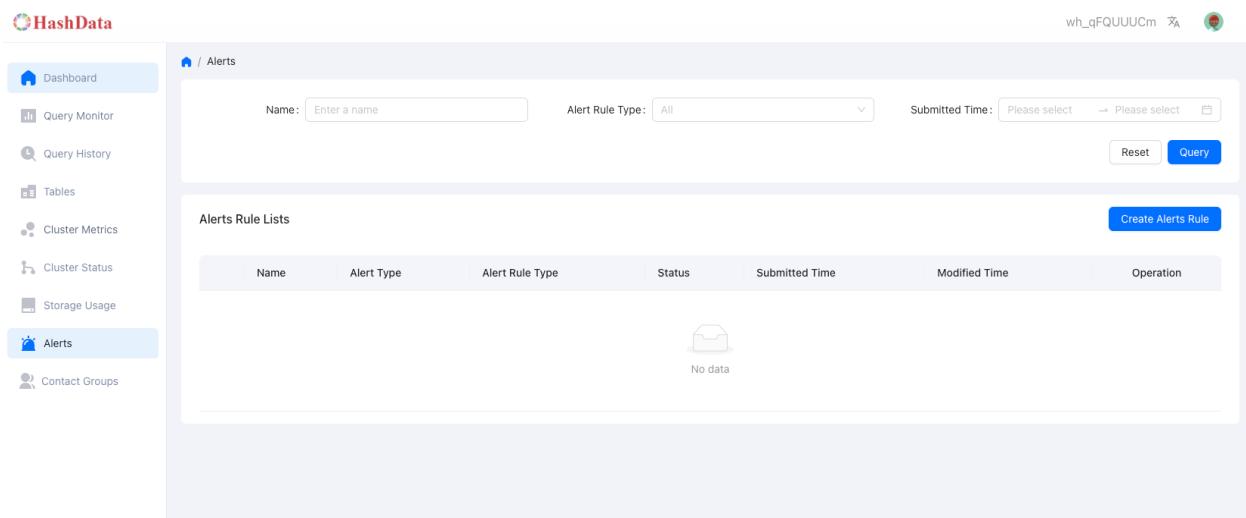
You can view and create alert rules in the CBCC. The system continuously monitors system events or resource usage and promptly notifies relevant personnel when anomalies occur.

Access the alert page

- Make sure you are logged in to the CBCC:

`http://<ip>:8080/`

- In the left navigation bar, click **Alerts** to enter the alert page.



View existing alert rules

At the top of the page, search for target alert rules by name, rule type, or submitted time.

Create an alert rule

1. Click **Create Alert Rule** to open the rule creation page.
2. Fill in the basic information for the alert at the top of the page:
 - **Name:** Enter the name of the alert. This field is required.
 - **Description:** Optional. Use this field to describe the purpose of the alert.

It is recommended to use a name with business importance to simplify identification and management.

The screenshot shows a web-based configuration interface for creating a new alert. It consists of three main sections:

- Basic Information:** Contains fields for "Name" (with placeholder "Enter a name" and character count "0 / 50") and "Description" (with placeholder "Please enter description" and character count "0 / 200").
- Alert Rules:** Shows the "Alert Type" as "Event" (selected) and "Metrics". A dropdown for "Related Template" is set to "Please select".
- Notification Settings:** Includes fields for "Contact Group" (dropdown), "Effective Time" (set to "00:00:00 → 23:59:59"), and "Priority Level" (dropdown).

At the bottom right of the form are "Cancel" and "Create" buttons.

Select alert type

You can choose one of the following alert types:

- **Event:** Detects specific system or database events.
- **Metrics:** Monitors resource usage in the cluster.

Click the corresponding button to switch the type. The setting steps vary depending on the selected type.

Set alert rules

Set the trigger conditions based on the selected alert type.

If you select the “Event” type:

Choose **Related Template** from the dropdown list to select a predefined event template. The system triggers alerts based on the events defined in the template.

If you select the “Metrics” type:

1. Choose a Trigger Rule:

- **Related Template:** Uses an existing policy template.
 - **Custom Template:** Manually defines the alert policy.
2. (If you choose **Related Template**) Select the appropriate template from the dropdown list.
 3. (If you choose **Custom Template**) Follow the on-screen instructions to define the alert policy.

Set notification policy

Specify who will receive alerts and under what conditions:

- **Contact Group:** Select an existing contact group. This field is required.
- **Effective Time:** Set the time range for notifications, such as all day (00:00:00 - 23:59:59).
- **Repetition Notification:** Set how frequently the alert is sent.
- **Priority Level:** Set the severity level of the alert.

Save the configuration

After confirming that the information is correct, click **Create** in the lower-right corner of the page to complete the alert configuration. To cancel the operation, click **Cancel** to return to the previous page.

When the alert is triggered, the system automatically sends notifications to the contact group based on your configuration, helping you detect potential issues in time.

View and Create Contact Groups

You can view and create contact groups in the CBCC. When an alert is triggered, the system promptly notifies the relevant members of the contact group.

Access the contact group page

1. Make sure you are logged in to the CBCC:

```
http://<ip>:8080/
```

2. In the left navigation bar, click **Contact Groups** to enter the contact group page.

Group Name	Created	Operation
		No data

View existing contact groups

At the top of the page, search for the target contact group by group name or creation time.

Create a contact group

1. Click **Create Contact Group** to open the creation window.
2. Fill in the group name, contact, and contact channel (email).
3. To add multiple contacts, click **+ Create Contacts**.

Create Contact Group X

* Group Name:

* Contact: 

* Channel:

+ Create Contacts

Cancel Create

7.2 Configure Security and Permission

Manage Roles and Privileges in SynxDB

The SynxDB authorization mechanism stores roles and privileges to access database objects in the database and is administered using SQL statements or command-line utilities.

SynxDB manages database access privileges using *roles*. The concept of roles subsumes the concepts of *users* and *groups*. A role can be a database user, a group, or both. Roles can own database objects (for example, tables) and can assign privileges on those objects to other roles to control access to the objects. Roles can be members of other roles, thus a member role can inherit the object privileges of its parent role.

Every SynxDB system contains a set of database roles (users and groups). Those roles are separate from the users and groups managed by the operating system on which the server runs. However, for convenience you might want to maintain a relationship between operating system user names and SynxDB role names, since many of the client applications use the current operating system user name as the default.

In SynxDB, users log in and connect through the coordinator instance, which then verifies their role and access privileges. The coordinator then issues commands to the segment instances behind the scenes as the currently logged-in role.

Roles are defined at the system level, meaning they are valid for all databases in the system.

To bootstrap the SynxDB system, a freshly initialized system always contains one predefined *superuser* role (also referred to as the system user). This role will have the same name as the operating system user that initialized the SynxDB system. Customarily, this role is named `gpadmin`. In order to create more roles you first have to connect as this initial role.

Create new roles (users)

A user-level role is considered to be a database role that can log in to the database and initiate a database session. Therefore, when you create a new user-level role using the `CREATE ROLE` command, you must specify the `LOGIN` privilege. For example:

```
CREATE ROLE jsmith WITH LOGIN;
```

A database role might have a number of attributes that define what sort of tasks that role can perform in the database. You can set these attributes when you create the role, or later using the `ALTER ROLE` command.

Alter role attributes

A database role might have a number of attributes that define what sort of tasks that role can perform in the database.

Attributes	Description
SUPERUSER or NOSUPERUSER	Determines if the role is a superuser. You must yourself be a superuser to create a new superuser. <code>NOSUPERUSER</code> is the default.
CREATEDB or NOCREATEDB	Determines if the role is allowed to create databases. <code>NOCREATEDB</code> is the default.
CREATEROLE or NOCREATEROLE	Determines if the role is allowed to create and manage other roles. <code>NOCREATEROLE</code> is the default.
INHERIT or NOINHERIT	Determines whether a role inherits the privileges of roles it is a member. A role with the <code>INHERIT</code> attribute can automatically use whatever database privileges have been granted to all roles it is directly or indirectly a member of. <code>INHERIT</code> is the default.
LOGIN or NOLOGIN	Determines whether a role is allowed to log in. A role having the <code>LOGIN</code> attribute can be thought of as a user. Roles without this attribute are useful for managing database privileges (groups). <code>NOLOGIN</code> is the default.
CONNECTION LIMIT *connlimit*	If role can log in, this specifies how many concurrent connections the role can make. -1 (the default) means no limit.
CREATEEXTTABLE or NOCREATEEXTTABLE	Determines whether a role is allowed to create external tables. <code>NOCREATEEXTTABLE</code> is the default. For a role with the <code>CREATEEXTTABLE</code> attribute, the default external table type is <code>readable</code> and the default protocol is <code>gpfdist</code> . Note that external tables that use the <code>file</code> protocol can only be created by superusers.
PASSWORD '*password*'	Sets the role's password. If you do not plan to use password authentication you can omit this option. If no password is specified, the password will be set to null and password authentication will always fail for that user. A null password can optionally be written explicitly as <code>PASSWORD NULL</code> .
DENY deny_interval or DENY deny_point	Restricts access during an interval, specified by day or day and time.

You can set these attributes when you create the role, or later using the `ALTER ROLE` command. For example:

```
ALTER ROLE jsmith WITH PASSWORD 'passwd123';
ALTER ROLE jsmith LOGIN;
ALTER ROLE jsmith DENY DAY 'Sunday';
```

A role can also have role-specific defaults for many of the server configuration settings. For example, to set the default schema search path for a role:

```
ALTER ROLE admin SET search_path TO myschema, public;
```

Role membership

It is frequently convenient to group users together to ease management of object privileges: that way, privileges can be granted to, or revoked from, a group as a whole. In SynxDB, this is done by creating a role that represents the group, and then granting membership in the group role to individual user roles.

Use the `CREATE ROLE` SQL command to create a new group role. For example:

```
CREATE ROLE admin CREATEROLE CREATEDB;
```

Once the group role exists, you can add and remove members (user roles) using the `GRANT` and `REVOKE` commands. For example:

```
GRANT admin TO john, sally;  
REVOKE admin FROM bob;
```

For managing object privileges, you would then grant the appropriate privileges to the group-level role only (see [Manage object privileges](#)). The member user roles then inherit the object privileges of the group role. For example:

```
GRANT ALL ON TABLE mytable TO admin;  
GRANT ALL ON SCHEMA myschema TO admin;  
GRANT ALL ON DATABASE mydb TO admin;
```

The role attributes `LOGIN`, `SUPERUSER`, `CREATEDB`, `CREATEROLE`, `CREATEEXTTABLE`, and `RESOURCE QUEUE` are never inherited as ordinary privileges on database objects are. User members must actually `SET ROLE` to a specific role having one of these attributes in order to make use of the attribute. In the above example, we gave `CREATEDB` and `CREATEROLE` to the `admin` role. If `sally` is a member of `admin`, then `sally` could issue the following command to assume the role attributes of the parent role:

```
SET ROLE admin;
```

Manage object privileges

When an object (table, view, sequence, database, function, language, schema, or tablespace) is created, it is assigned an owner. The owner is normally the role that ran the creation statement. For most kinds of objects, the initial state is that only the owner (or a superuser) can do anything with the object. To allow other roles to use it, privileges must be granted. SynxDB supports the following privileges for each object type:

Object Type	Privileges
Tables, External Tables, Views	SELECT, INSERT, UPDATE, DELETE, REFERENCES, TRIGGER, TRUNCATE, ALL
Columns	SELECT, INSERT, UPDATE, REFERENCES, ALL
Sequences	USAGE, SELECT, UPDATE, ALL
Databases	CREATE, CONNECT, TEMPORARY, TEMP, ALL
Domains	USAGE, ALL
Foreign Data Wrappers	USAGE, ALL
Foreign Servers	USAGE, ALL
Functions	EXECUTE, ALL
Procedural Languages	USAGE, ALL
Schemas	CREATE, USAGE, ALL
Tablespaces	CREATE, ALL
Types	USAGE, ALL
Protocols	SELECT, INSERT, ALL

Note

You must grant privileges for each object individually. For example, granting ALL on a database does not grant full access to the objects within that database. It only grants all of the database-level privileges (CONNECT, CREATE, TEMPORARY) to the database itself.

Use the GRANT SQL command to give a specified role privileges on an object. For example, to grant the role named `jsmith` insert privileges on the table named `mytable`:

```
GRANT INSERT ON mytable TO jsmith;
```

Similarly, to grant `jsmith` select privileges only to the column named `col1` in the table named

table2:

```
GRANT SELECT (col1) on TABLE table2 TO jsmith;
```

To revoke privileges, use the REVOKE command. For example:

```
REVOKE ALL PRIVILEGES ON mytable FROM jsmith;
```

You can also use the **DROP OWNED** and **REASSIGN OWNED** commands for managing objects owned by deprecated roles (Note: only an object's owner or a superuser can drop an object or reassign ownership). For example:

```
REASSIGN OWNED BY sally TO bob;  
DROP OWNED BY visitor;
```

Security best practices for roles and privileges

- **Secure the gpadmin system user.**

SynxDB requires a UNIX user ID to install and initialize the SynxDB system. This system user is referred to as `gpadmin` in the SynxDB documentation. This `gpadmin` user is the default database superuser in SynxDB, as well as the file system owner of the SynxDB installation and its underlying data files. This default administrator account is fundamental to the design of SynxDB. The system cannot run without it, and there is no way to limit the access of this `gpadmin` user ID.

Use roles to manage who has access to the database for specific purposes. You should only use the `gpadmin` account for system maintenance tasks such as expansion and upgrade. Anyone who logs on to a SynxDB host as this user ID can read, alter or delete any data, including system catalog data and database access rights. Therefore, it is very important to secure the `gpadmin` user ID and only provide access to essential system administrators. Administrators should only log in to SynxDB as `gpadmin` when performing certain system maintenance tasks (such as upgrade or expansion). Database users should never log on as `gpadmin`, and ETL or production workloads should never run as `gpadmin`.

- **Assign a distinct role to each user that logs in.**

For logging and auditing purposes, each user that is allowed to log in to SynxDB should be

given their own database role. For applications or web services, consider creating a distinct role for each application or service. See [Create New Roles \(Users\)](#).

- **Use groups to manage access privileges.**

See [Role membership](#).

- **Limit users who have the SUPERUSER role attribute.**

Roles that are superusers bypass all access privilege checks in SynxDB, as well as resource queuing. Only system administrators should be given superuser rights. See [Alter Role Attributes](#).

Encrypt data

SynxDB is installed with an optional module of encryption/decryption functions called pgcrypto. The pgcrypto functions allow database administrators to store certain columns of data in encrypted form. This adds an extra layer of protection for sensitive data, as data stored in SynxDB in encrypted form cannot be read by anyone who does not have the encryption key, nor can it be read directly from the disks.

Note

The pgcrypto functions run inside the database server, which means that all the data and passwords move between pgcrypto and the client application in clear-text.

To use pgcrypto functions, register the pgcrypto extension in each database in which you want to use the functions. For example:

```
psql -d testdb -c "CREATE EXTENSION pgcrypto"
```

7.3 Backup and Restore

Backup and Restore Overview

SynxDB offers both parallel and non-parallel methods for database backups and restores. Parallel operations handle large systems efficiently because each segment host writes data to its local disk at the same time. Non-parallel operations, however, transfer all data over the network to the coordinator, which then writes it to its storage. This method not only concentrates I/O on a single host but also requires the coordinator to have enough local disk space for the entire database.

Parallel backup with `gpbackup` and `gprestore`

SynxDB provides `gpbackup` and `gprestore` for parallel backup and restore utilities. `gpbackup` uses table-level ACCESS SHARE locks instead of EXCLUSIVE locks on the `pg_class` catalog table. This enables you to execute DDL statements such as CREATE, ALTER, DROP, and TRUNCATE during backups, as long as these statements do not target the current backup set.

Backup files created with `gpbackup` are designed to provide future capabilities for restoring individual database objects along with their dependencies, such as functions and required user-defined data types.

For details about backup and restore using `gpbackup` and `gprestore`, see [Perform Full Backup and Restore](#) and [Perform Incremental Backup and Restore](#).

Command-line flags for `gpbackup` and `gprestore`

The command-line flags for `gpbackup` are as follows:

```
Usage:
gpbackup [flags]

Flags:
  --backup-dir string          The absolute path of the directory to
which all backup files will be written
  --compression-level int      Level of compression to use during data
                                (continues on next page)
```

(continued from previous page)

```

backup. Range of valid values depends on compression type (default 1)
      --compression-type string      Type of compression to use during data
backup. Valid values are 'gzip', 'zstd' (default "gzip")
      --copy-queue-size int         number of COPY commands gpbackup should
enqueue when backing up using the --single-data-file option (default 1)
      --data-only                  Only back up data, do not back up
metadata
      --dbname string              The database to be backed up
      --debug                      Print verbose and debug log messages
      --exclude-schema stringArray Back up all metadata except objects in
the specified schema(s). --exclude-schema can be specified multiple times.
      --exclude-schema-file string A file containing a list of schemas to be
excluded from the backup
      --exclude-table stringArray   Back up all metadata except the specified
table(s). --exclude-table can be specified multiple times.
      --exclude-table-file string  A file containing a list of fully-
qualified tables to be excluded from the backup
      --from-timestamp string      A timestamp to use to base the current
incremental backup off
      --help                        Help for gpbackup
      --include-schema stringArray Back up only the specified schema(s). --include-schema can be specified multiple times.
      --include-schema-file string A file containing a list of schema(s) to
be included in the backup
      --include-table stringArray   Back up only the specified table(s). --include-table can be specified multiple times.
      --include-table-file string  A file containing a list of fully-
qualified tables to be included in the backup
      --incremental                 Only back up data for AO tables that have
been modified since the last backup
      --jobs int                    The number of parallel connections to use
when backing up data (default 1)
      --leaf-partition-data        For partition tables, create one data
file per leaf partition instead of one data file for the whole table
      --metadata-only               Only back up metadata, do not back up
data
      --no-compression             Skip compression of data files
      --plugin-config string       The configuration file to use for a
plugin
      --quiet                      Suppress non-warning, non-error log

```

(continues on next page)

(continued from previous page)

messages	
--single-data-file	Back up all data to a single file instead of one per table
--verbose	Print verbose log messages
--version	Print version number and <code>exit</code>
--with-stats	Back up query plan statistics
--without-globals	Skip backup of global metadata

The command-line flags for `gprestore` are as follows:

Usage:	
<code>gprestore [flags]</code>	
Flags:	
<code>--backup-dir string</code>	The absolute path of the directory <code>in</code> which the backup files to be restored are located
<code>--copy-queue-size int</code>	Number of COPY commands <code>gprestore</code> should enqueue when restoring a backup taken using the <code>--single-data-file</code> option (default <code>1</code>)
<code>--create-db</code>	Create the database before metadata restore
<code>--data-only</code>	Only restore data, <code>do</code> not restore metadata
<code>--debug</code>	Print verbose and debug log messages
<code>--exclude-schema stringArray</code>	Restore all metadata except objects <code>in</code> the specified schema(s). <code>--exclude-schema</code> can be specified multiple times.
<code>--exclude-schema-file string</code>	A file containing a list of schemas that will not be restored
<code>--exclude-table stringArray</code>	Restore all metadata except the specified relation(s). <code>--exclude-table</code> can be specified multiple times.
<code>--exclude-table-file string</code>	A file containing a list of fully-qualified relation(s) that will not be restored
<code>--help</code>	Help <code>for</code> <code>gprestore</code>
<code>--include-schema stringArray</code>	Restore only the specified schema(s). <code>--include-schema</code> can be specified multiple times.
<code>--include-schema-file string</code>	A file containing a list of schemas that will be restored
<code>--include-table stringArray</code>	Restore only the specified relation(s). <code>--include-table</code> can be specified multiple times.
<code>--include-table-file string</code>	A file containing a list of fully-

(continues on next page)

(continued from previous page)

qualified relation(s) that will be restored	
--incremental	BETA FEATURE: Only restore data for all heap tables and only AO tables that have been modified since the last backup
--jobs int	Number of parallel connections to use when restoring table data and post-data (default 1)
--metadata-only	Only restore metadata, do not restore data
--on-error-continue	Log errors and continue restore, instead of exiting on first error
--plugin-config string	The configuration file to use for a plugin
--quiet	Suppress non-warning, non-error log messages
--redirect-db string	Restore to the specified database instead of the database that was backed up
--redirect-schema string	Restore to the specified schema instead of the schema that was backed up
--resize-cluster	Restore a backup taken on a cluster with more or fewer segments than the cluster to which it will be restored
--run-analyze	Run ANALYZE on restored tables
--timestamp string	The timestamp to be restored, in the format YYYYMMDDHHMMSS
--truncate-table	Removes data of the tables getting restored
--verbose	Print verbose log messages
--version	Print version number and exit
--with-globals	Restore global metadata
--with-stats	Restore query plan statistics

Non-parallel backup with pg_dump

You can also use the PostgreSQL non-parallel backup utilities `pg_dump` and `pg_dumpall` to create a single dump file on the coordinator host that contains all data from all active segments.

The PostgreSQL non-parallel utilities should be used only for special cases. They are much slower than using `gpbacup` and `gprestore` because all of the data must pass through the coordinator. In addition, it is often the case that the coordinator host has insufficient disk space to save a backup of an entire distributed SynxDB.

The `pg_restore` utility requires compressed dump files created by `pg_dump` or `pg_dumpall`. Before starting the restore, you should modify the `CREATE TABLE` statements in the dump files to include the SynxDB `DISTRIBUTED` clause. If you do not include the `DISTRIBUTED` clause, SynxDB assigns default values, which might not be optimal.

To perform a non-parallel restore using parallel backup files, you can copy the backup files from each segment host to the coordinator host, and then load them through the coordinator.

Another non-parallel method for backing up SynxDB data is to use the `COPY TO SQL` command to copy all or a portion of a table out of the database to a delimited text file on the coordinator host.

Backup and recovery with CBDR at the WAL level

CBDR is a backup and restore tool built on top of WAL-G, designed for SynxDB. It supports both full and incremental backups via the command line and is well suited for disaster recovery and long-term archival use cases.

Key features of CBDR include:

- **Full backup:** Supports full backup of the entire database cluster.
- **Incremental backup:** Captures only changes since the last backup based on WAL logs.
- **Restore point support:** Allows creating and restoring to named restore points for point-in-time recovery.
- **Backup listing and management:** Lists all full and incremental backups as well as restore points.
- **Storage support:** Supports S3-compatible object storage.
- **Compression options:** Supports lz4, lzma, zstd, and brotli compression algorithms.
- **Backup encryption:** Enables encryption of backup data.
- **Configuration management:** Automatically generates and manages required backup and restore configuration files.

For details of the CBDR tool, see the document [CBDR](#).

 **Tip**

Compared to traditional tools like `gpbackup` and `gprestore`, CBDR offers more flexible incremental backup capabilities and supports storing backup files in remote object storage, reducing reliance on local disk space. In addition, CBDR uses WAL-based physical backup and recovery, making it suitable for large-scale clusters and cross-region disaster recovery scenarios.

Perform Full Backup and Restore

SynxDB supports backing up and restoring the full database in parallel. Parallel operations scale regardless of the number of segments in your system, because segment hosts each write their data to local disk storage at the same time.

`gpbackup` and `gprestore` are SynxDB command-line utilities that create and restore backup sets for SynxDB. By default, `gpbackup` stores only the object metadata files and DDL files for a backup in the SynxDB coordinator data directory. SynxDB segments use the `COPY ... ON SEGMENT` command to store their data for backed-up tables in compressed CSV data files, located in each segment's backups directory.

The backup metadata files contain all of the information that `gprestore` needs to restore a full backup set in parallel. Each `gpbackup` task uses a single transaction in SynxDB. During this transaction, metadata is backed up on the coordinator host, and data for each table on each segment host is written to CSV backup files using `COPY ... ON SEGMENT` commands in parallel. The backup process acquires an `ACCESS SHARE` lock on each table that is backed up.

Back up the full database

To perform a complete backup of a database, as well as SynxDB system metadata, use the command:

```
gpbackup --dbname <database_name>
```

For example:

```
$ gpbackup --dbname test_04

20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-gpbackup
version = 1.2.7-beta1+dev.7
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-|product_
name| Version = oudberry Database 1.0.0 build 5551471267
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Starting
backup of database test_04
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Backup
Timestamp = 20240108171718
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Backup
```

(continues on next page)

(continued from previous page)

```
Database = test_04
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Gathering
table state information
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Acquiring
ACCESS SHARE locks on tables
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Gathering
additional table metadata
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Getting
storage information
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[WARNING]:-No
tables in backup set contain data. Performing metadata-only backup instead.
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Metadata
will be written to /data0/coordinator/gpseg-1/backups/20240108/20240108171718/
gpbackup_20240108171718_metadata.sql
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Writing
global database metadata
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Global
database metadata backup complete
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Writing
pre-data metadata
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Pre-data
metadata backup complete
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Writing
post-data metadata
20240108:17:17:18 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Post-data
metadata backup complete
20240108:17:17:19 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Found
neither /usr/local/cloudberry-db-1.0.0/bin/gp_email_contacts.yaml nor /home/
gpadmin//gp_email_contacts.yaml
20240108:17:17:19 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Email
containing gpbackup report /data0/coordinator/gpseg-1/backups/20240108/
20240108171718/gpbackup_20240108171718_report will not be sent
20240108:17:17:19 gpbackup:gpadmin:cbdb-coordinator:001945-[INFO]:-Backup
completed successfully
```

The above command creates a file that contains global and database-specific metadata on the SynxDB coordinator host in the default directory, \$COORDINATOR_DATA_DIRECTORY/backups/<YYYYMMDD>/<YYYYMMDDHHMMSS>/. For example:

```
ls $COORDINATOR_DATA_DIRECTORY/backups/20240108/20240108171718

gpbackup_20240108171718_config.yaml      gpbackup_20240108171718_report
gpbackup_20240108171718_metadata.sql     gpbackup_20240108171718_toc.yaml
```

By default, each segment stores each table's data for the backup in a separate compressed CSV file in <seg_dir>/backups/<YYYYMMDD>/<YYYYMMDDHHMMSS>/. For example:

```
ls /data1/primary/gpseg1/backups/20240108/20240108171718/

gpbackup_0_20240108171718_17166.gz  gpbackup_0_20240108171718_26303.gz
gpbackup_0_20240108171718_21816.gz
```

To consolidate all backup files into a single directory, include the --backup-dir option. Note that you need to specify an absolute path with this option:

```
$ gpbackup --dbname test_04 --backup-dir /home/gpadmin/backups

20240108:17:34:10 gpbackup:gpadmin:cbdb-coordinator:003348-[INFO] :-gpbackup
version = 1.2.7-beta1+dev.7
20240108:17:34:10 gpbackup:gpadmin:cbdb-coordinator:003348-[INFO] :-|product_
name| Version = oudberry Database 1.0.0 build 5551471267
...
20240108:17:34:12 gpbackup:gpadmin:cbdb-coordinator:003348-[INFO] :-Backup
completed successfully

$ find /home/gpadmin/backups/ -type f

/home/gpadmin/backups/gpseg0/backups/20240108/20240108173410/gpbackup_0_
20240108173410_16593.gz
/home/gpadmin/backups/gpseg-1/backups/20240108/20240108173410/gpbackup_
20240108173410_config.yaml
/home/gpadmin/backups/gpseg-1/backups/20240108/20240108173410/gpbackup_
20240108173410_report
/home/gpadmin/backups/gpseg-1/backups/20240108/20240108173410/gpbackup_
20240108173410_toc.yaml
/home/gpadmin/backups/gpseg-1/backups/20240108/20240108173410/gpbackup_
20240108173410_metadata.sql
/home/gpadmin/backups/gpseg1/backups/20240108/20240108173410/gpbackup_1_
20240108173410_16593.gz
```

When performing a backup operation, you can use the `--single-data-file` in situations where the additional overhead of multiple files might be prohibitive. For example, if you use a third party storage solution such as Data Domain with backups.

Tip

Backing up a materialized view does not back up the materialized view data. Only the materialized view definition is backed up.

Restore the full database

To use `gprestore` to restore from a backup set, you must use the `--timestamp` option to specify the exact timestamp value (YYYYMMDDHHMMSS) to restore. Include the `--create-db` option if the database does not exist in the cluster. For example:

```
$ dropdb demo
$ gprestore --timestamp 20240108171718 --create-db

20240108:17:42:26 gprestore:gadmin:cbdb-coordinator:004115-[INFO]:-Restore
Key = 20240108171718
20240108:17:42:26 gprestore:gadmin:cbdb-coordinator:004115-[INFO]:-gpbackup
version = 1.2.7-beta1+dev.7
20240108:17:42:26 gprestore:gadmin:cbdb-coordinator:004115-[INFO]:-gprestore
version = 1.2.7-beta1+dev.7
20240108:17:42:26 gprestore:gadmin:cbdb-coordinator:004115-[INFO]:-|product_
name| Version = oudberry Database 1.0.0 build 5551471267
20240108:17:42:26 gprestore:gadmin:cbdb-coordinator:004115-[INFO]:-Creating
database
20240108:17:42:26 gprestore:gadmin:cbdb-coordinator:004115-[INFO]:-Database
creation complete for: test_04
20240108:17:42:26 gprestore:gadmin:cbdb-coordinator:004115-[INFO]:-Restoring
pre-data metadata
Pre-data objects restored: 3 / 3 [=====] 100.00%
0s
20240108:17:42:26 gprestore:gadmin:cbdb-coordinator:004115-[INFO]:-Pre-data
metadata restore complete
20240108:17:42:26 gprestore:gadmin:cbdb-coordinator:004115-[INFO]:-Restoring
post-data metadata
```

(continues on next page)

(continued from previous page)

```
20240108:17:42:26 gprestore:gadmin:cbdb-coordinator:004115-[INFO]:-Post-data  
metadata restore complete  
20240108:17:42:26 gprestore:gadmin:cbdb-coordinator:004115-[INFO]:-Found  
neither /usr/local/cloudberry-db-1.0.0/bin/gp_email_contacts.yaml nor /home/  
gadmin/gp_email_contacts.yaml  
20240108:17:42:26 gprestore:gadmin:cbdb-coordinator:004115-[INFO]:-Email  
containing gprestore report /data0/coordinator/gpseg-1/backups/20240108/  
20240108171718/gprestore_20240108171718_20240108174226_report will not be sent  
20240108:17:42:26 gprestore:gadmin:cbdb-coordinator:004115-[INFO]:-Restore  
completed successfully
```

If you specified a custom `--backup-dir` to consolidate the backup files, include the same `--backup-dir` option when using `gprestore` to locate the backup files:

```
$ dropdb test_04  
$ gprestore --backup-dir /home/gadmin/backups/ --timestamp 20240109102646 --  
create-db  
  
20240109:10:33:17 gprestore:gadmin:cbdb-coordinator:017112-[INFO]:-Restore  
Key = 20240109102646  
...  
20240109:10:33:17 gprestore:gadmin:cbdb-coordinator:017112-[INFO]:-Restore  
completed successfully
```

`gprestore` does not attempt to restore global metadata for the SynxDB system by default. If this is required, include the `--with-globals` argument.

By default, `gprestore` uses 1 connection to restore table data and metadata. If you have a large backup set, you can improve performance of the restore by increasing the number of parallel connections with the `--jobs` option. For example:

```
$ gprestore --backup-dir /home/gadmin/backups/ --timestamp 20240109102646 --  
create-db --jobs 4
```

Test the number of parallel connections with your backup set to determine the ideal number for fast recovery.

Tip

You cannot perform a parallel restore operation with `gprestore` if the backup combines table backups into a single file per segment with the `gpbackup` option `--single-data-file`.

Restoring a materialized view does not restore materialized view data. Only the materialized view definition is restored. To populate the materialized view with data, use `REFRESH MATERIALIZED VIEW`. When you refresh the materialized view, the tables that are referenced by the materialized view definition must be available. The `gprestore` log file lists the materialized views that have been restored and the `REFRESH MATERIALIZED VIEW` commands that are used to populate the materialized views with data.

Filter the contents of a backup or restore

Filter by schema

`gpbackup` backs up all schemas and tables in the specified database, unless you exclude or include individual schema or table objects with schema level or table level filter options.

The schema level options are `--include-schema`, `--include-schema-file`, or `--exclude-schema`, `--exclude-schema-file` command-line options to `gpbackup`. For example, if the `test_04` database includes only 2 schemas, `schema1` and `schema2`, both of the following commands back up only the `schema1` schema:

```
$ gpbackup --dbname test_04 --include-schema schema1
$ gpbackup --dbname test_04 --exclude-schema schema2
```

You can include multiple `--include-schema` options in a `gpbackup` or multiple `--exclude-schema` options. For example:

```
$ gpbackup --dbname test_04 --include-schema schema1 --include-schema schema2
```

If you have a large number of schemas, you can list the schemas in a text file and specify the file with the `--include-schema-file` or `--exclude-schema-file` options in a `gpbackup` command. Each line in the file must define a single schema, and the file cannot contain trailing lines. For example, this command uses a file in the `gpadmin` home directory to include a set of schemas.

```
$ gpbackup --dbname test_04 --include-schema-file /home/gpadmin/backup-schemas.txt --backup-dir /home/gpadmin/backups
```

Filter by table

To filter the individual tables that are included in a backup set, or excluded from a backup set, specify individual tables with the `--include-table` option or the `--exclude-table` option. The table must be schema qualified, `<schema-name>.<table-name>`. The individual table filtering options can be specified multiple times. However, `--include-table` and `--exclude-table` cannot both be used in the same command.

Tip

If you have used the `--include-table` option in a `gpbackup` command, the database, schema, and the sequence related to the target table you specified are not backed up. Before you restore the backup set, you must create the database, schema, and sequence manually.

For example, if you have used `gpbackup --dbname test_04 --include-table schema1.table1 --backup-dir /home/gpadmin/backups` to back up the `test_04` database, you must create the `test_04` database, `schema1` schema, and `schema1.table1_id_seq` sequence manually before you restore the backup set. Otherwise, the restore operation fails with the error message indicating that the database, schema, or sequence does not exist.

If a table or schema name uses any character other than a lowercase letter, number, or an underscore character, then you must include that name in double quotes. For example:

```
gpbackup --dbname test1 --include-table "schema1"."ComplexName Table" --backup-dir /home/gpadmin/backups
```

You can create a list of qualified table names in a text file. When listing tables in a file, each line in the text file must define a single table using the format `<schema-name>.<table-name>`. The file must not include trailing lines or double quotes even if table or schema name uses any character other than a lowercase letter, number, or an underscore character. For example:

```
schema1.table1
schema2.table2
schema1.ComplexName Table
```

After creating the file, you can use it either to include or exclude tables with the `gpbackup` options `--include-table-file` or `--exclude-table-file`. For example:

```
$ gpbackup --dbname test_04 --include-table-file /home/gpadmin/table-list.txt
```

You can combine `--include-schema` with `--exclude-table` or `--exclude-table-file` for a backup. The following example uses `--include-schema` with `--exclude-table` to back up a schema except for a single table.

```
$ gpbackup --dbname test_04 --include-schema schema1 --exclude-table schema2.table2
```

You cannot combine `--include-schema` with `--include-table` or `--include-table-file`, and you cannot combine `--exclude-schema` with any table filtering option such as `--exclude-table` or `--include-table`.

When you use `--include-table` or `--include-table-file` dependent objects are not automatically backed up or restored, you must explicitly specify the dependent objects that are required. For example, if you back up or restore a view or materialized view, you must also specify the tables that the view or the materialized view uses. If you backup or restore a table that uses a sequence, you must also specify the sequence.

Filter with `gprestore`

After creating a backup set with `gpbackup`, you can filter the schemas and tables that you want to restore from the backup set using the `gprestore` `--include-schema` and `--include-table-file` options. These options work in the same way as their `gpbackup` counterparts, but have the following restrictions:

- The tables that you attempt to restore must not already exist in the database.
- If you attempt to restore a schema or table that does not exist in the backup set, the `gprestore` does not execute.

- If you use the `--include-schema` option, `gprestore` cannot restore objects that have dependencies on multiple schemas.
- If you use the `--include-table-file` option, `gprestore` does not create roles or set the owner of the tables. The utility restores table indexes and rules. Triggers are also restored but are not supported in SynxDB.
- The file that you specify with `--include-table-file` cannot include a leaf partition name, as it can when you specify this option with `gpbackup`. If you specified leaf partitions in the backup set, specify the partitioned table to restore the leaf partition data.

When restoring a backup set that contains data from some leaf partitions of a partitioned table, the partitioned table is restored along with the data for the leaf partitions. For example, you create a backup with the `gpbackup` option `--include-table-file` and the text file lists some leaf partitions of a partitioned table. Restoring the backup creates the partitioned table and restores the data only for the leaf partitions listed in the file.

Filter by leaf partition

By default, `gpbackup` creates one file for each table on a segment. You can specify the `--leaf-partition-data` option to create one data file per leaf partition of a partitioned table, instead of a single file. You can also filter backups to specific leaf partitions by listing the leaf partition names in a text file to include. For example, consider a table that was created using the statement:

```
CREATE TABLE sales (id int, date date, amt decimal(10,2))
DISTRIBUTED BY (id)
PARTITION BY RANGE (date)
( PARTITION Jan23 START (date '2023-01-01') INCLUSIVE ,
  PARTITION Feb23 START (date '2023-02-01') INCLUSIVE ,
  PARTITION Mar23 START (date '2023-03-01') INCLUSIVE ,
  PARTITION Apr23 START (date '2023-04-01') INCLUSIVE ,
  PARTITION May23 START (date '2023-05-01') INCLUSIVE ,
  PARTITION Jun23 START (date '2023-06-01') INCLUSIVE ,
  PARTITION Jul23 START (date '2023-07-01') INCLUSIVE ,
  PARTITION Aug23 START (date '2023-08-01') INCLUSIVE ,
  PARTITION Sep23 START (date '2023-09-01') INCLUSIVE ,
  PARTITION Oct23 START (date '2023-10-01') INCLUSIVE ,
```

(continues on next page)

(continued from previous page)

```

PARTITION Nov23 START (date '2023-11-01') INCLUSIVE ,
PARTITION Dec23 START (date '2023-12-01') INCLUSIVE
END (date '2024-01-01') EXCLUSIVE );

NOTICE: CREATE TABLE will create partition "sales_1_prt_jan23" for table
"sales"
NOTICE: CREATE TABLE will create partition "sales_1_prt_feb23" for table
"sales"
NOTICE: CREATE TABLE will create partition "sales_1_prt_mar23" for table
"sales"
NOTICE: CREATE TABLE will create partition "sales_1_prt_apr23" for table
"sales"
NOTICE: CREATE TABLE will create partition "sales_1_prt_may23" for table
"sales"
NOTICE: CREATE TABLE will create partition "sales_1_prt_jun23" for table
"sales"
NOTICE: CREATE TABLE will create partition "sales_1_prt_jul23" for table
"sales"
NOTICE: CREATE TABLE will create partition "sales_1_prt_aug23" for table
"sales"
NOTICE: CREATE TABLE will create partition "sales_1_prt_sep23" for table
"sales"
NOTICE: CREATE TABLE will create partition "sales_1_prt_oct23" for table
"sales"
NOTICE: CREATE TABLE will create partition "sales_1_prt_nov23" for table
"sales"
NOTICE: CREATE TABLE will create partition "sales_1_prt_dec23" for table
"sales"
CREATE TABLE

```

To back up only data for the last quarter of the year, first create a text file that lists those leaf partition names instead of the full table name:

```

public.sales_1_prt_oct23
public.sales_1_prt_nov23
public.sales_1_prt_dec23

```

Then specify the file with the `--include-table-file` option to generate one data file per leaf partition:

```
$ gpbackup --dbname test_04 --include-table-file last-quarter.txt --leaf-partition-data
```

When you specify `--leaf-partition-data`, `gpbackup` generates one data file per leaf partition when backing up a partitioned table. For example, this command generates one data file for each leaf partition:

```
$ gpbackup --dbname test_04 --include-table public.sales --leaf-partition-data
```

When leaf partitions are backed up, the leaf partition data is backed up along with the metadata for the entire partitioned table.

Check report files

When performing a backup or restore operation, `gpbackup` and `gprestore` generate a report file that contains the detailed information of the operations. When email notification is configured, the email sent contains the contents of the report file. For information about email notification, see [Configure email notifications](#).

The report file is placed in the SynxDB coordinator backup directory. The report file name contains the timestamp of the operation. The following are the formats of the `gpbackup` and `gprestore` report file names.

```
gpbackup_<backup_timestamp>_report  
gprestore_<backup_timestamp>_<restore_timestamp>_report
```

For these example report file names, 20240109111719 is the timestamp of the backup and 20240109112545 is the timestamp of the restore operation.

```
gpbackup_20240109111719_report  
gprestore_20240109111719_20240109112545_report
```

This backup directory on a SynxDB coordinator host contains both a `gpbackup` and `gprestore` report file.

```
$ ls -l /data0/coordinator/gpseg-1/backups/20240109/20240109111719/  
total 24
```

(continues on next page)

(continued from previous page)

```
-r--r--r-- 1 gpadmin gpadmin 715 Jan  9 11:17 gpbackup_20240109111719_config.yaml
-r--r--r-- 1 gpadmin gpadmin 895 Jan  9 11:17 gpbackup_20240109111719_metadata.sql
-r--r--r-- 1 gpadmin gpadmin 1441 Jan  9 11:17 gpbackup_20240109111719_report
-r--r--r-- 1 gpadmin gpadmin 1226 Jan  9 11:17 gpbackup_20240109111719_toc.yaml
-r--r--r-- 1 gpadmin gpadmin 569 Jan  9 11:21 gprestore_20240109111719_20240109112128_report
-r--r--r-- 1 gpadmin gpadmin 514 Jan  9 11:25 gprestore_20240109111719_20240109112545_report
```

The contents of the report files are similar. This is an example of the contents of a gprestore report file.

```
|product_name| Restore Report

timestamp key:          20240109111719
gpdb version:           oudberry Database 1.0.0 build 5551471267
gprestore version:       1.2.7-beta1+dev.7

database name:          test_04
command line:            gprestore --timestamp 20240109111719 --create-db

backup segment count:    2
restore segment count:   2
start time:              Tue Jan 09 2024 11:25:45
end time:                Tue Jan 09 2024 11:25:46
duration:                0:00:01

restore status:          Success
```

Configure email notifications

`gpbackup` and `gprestore` can send email notifications after a back up or restore operation completes.

To have `gpbackup` or `gprestore` send out status email notifications, you need place a file named `gp_email_contacts.yaml` in the home directory of the user running `gpbackup` or `gprestore` in the same directory as the utilities (`$GPHOME/bin`). A utility issues a message if it cannot locate a `gp_email_contacts.yaml` file in either location. If both locations contain a `.yaml` file, the utility uses the file in user `$HOME`.

The email subject line includes the utility name, timestamp, job status (Success or Failure), and the name of the SynxDB host `gpbackup` or `gprestore` is called from. These are example subject lines for `gpbackup` emails.

```
gpbackup 20180202133601 on gp-master completed: Success
```

or

```
gpbackup 20200925140738 on mdw completed: Failure
```

The email contains summary information about the operation including options, duration, and number of objects backed up or restored. For information about the contents of a notification email, see Report Files.

Tip

The UNIX mail utility must be running on the SynxDB host and must be configured to allow the SynxDB superuser (`gpadmin`) to send email. Also ensure that the mail program executable is locatable via the `gpadmin` user's `$PATH`.

gpbackup and gprestore email file format

The gpbackup and gprestore email notification YAML file `gp_email_contacts.yaml` uses indentation (spaces) to determine the document hierarchy and the relationships of the sections to one another. The use of white space is significant. White space should not be used simply for formatting purposes, and tabs should not be used at all.

If the status parameters are not specified correctly, the utility does not issue a warning. For example, if the success parameter is misspelled and is set to true, a warning is not issued and an email is not sent to the email address after a successful operation. To ensure email notification is configured correctly, run tests with email notifications configured.

This is the format of the `gp_email_contacts.yaml` YAML file for gpbackup email notifications:

```
contacts:
  gpbackup:
    - address: <user>@<domain>
      status:
        success: [true | false]
        success_with_errors: [true | false]
        failure: [true | false]
  gprestore:
    - address: <user>@<domain>
      status:
        success: [true | false]
        success_with_errors: [true | false]
        failure: [true | false]
```

Email YAML file sections

contacts

Required. The section that contains the gpbackup and gprestore sections. The YAML file can contain a gpbackup section, a gprestore section, or one of each.

gpbackup

Optional. Begins the gpbackup email section.

address

Required. At least one email address must be specified. Multiple email address parameters can be specified. Each address requires a status section. `user@domain` is a single, valid email address.

status

Required. Specify when the utility sends an email to the specified email address. The default is to not send email notification. You specify sending email notifications based on the completion status of a backup or restore operation. At least one of these parameters must be specified and each parameter can appear at most once.

success

Optional. Specify whether an email is sent if the operation completes without errors. If the value is `true`, an email is sent if the operation completes without errors. If the value is `false` (the default), an email is not sent.

success_with_errors

Optional. Specify whether an email is sent if the operation completes with errors. If the value is `true`, an email is sent if the operation completes with errors. If the value is `false` (the default), an email is not sent.

failure

Optional. Specify if an email is sent if the operation fails. If the value is `true`, an email is sent if the operation fails. If the value is `false` (the default), an email is not sent.

gprestore

Optional. Begins the gprestore email section. This section contains the address and status parameters that are used to send an email notification after a gprestore operation. The syntax is the same as the gpbackup section.

Email examples

This example YAML file specifies sending email to email addresses depending on the success or failure of an operation. For a backup operation, an email is sent to a different address depending on the success or failure of the backup operation. For a restore operation, an email is sent to gpadmin@example.com only when the operation succeeds or completes with errors.

```
contacts:  
  gpbackup:  
    - address: gpadmin@example.com  
      status:  
        success: true  
        success_with_errors: true  
        failure: true  
  gprestore:  
    - address: gpadmin@example.com  
      status:  
        success: true  
        success_with_errors: true  
        failure: true
```

Perform Incremental Backup and Restore

Before reading this document, you are expected to first read the [Perform Full Backup and Restore](#) document.

To back up and restore tables incrementally, use the `gpbackup` and `gprestore` utilities. Incremental backups include all specified heap tables, and append-optimized tables (including column-oriented ones) that have changed. Even a single row change triggers a backup of the entire append-optimized table. For partitioned append-optimized tables, only the modified leaf partitions are backed up.

Incremental backups are efficient when the total amount of data in append-optimized tables or table partitions that changed is small compared to the data that has not changed since the last backup.

An incremental backup backs up an append-optimized table only if one of the following operations have been performed on the table after the last full or incremental backup:

- `ALTER TABLE`
- `DELETE`
- `INSERT`
- `TRUNCATE`
- `UPDATE`
- `DROP` and then re-create the table

To restore data from incremental backups, you need a complete incremental backup set.

About incremental backup sets

An incremental backup set includes the following backups:

- A full backup. This is the full backup that the incremental backups are based on.
- The set of incremental backups that capture the changes to the database from the time of the full backup.

For example, you can create a full backup and then create 3 daily incremental backups. The full backup and all 3 incremental backups are the backup set. For information about using an

incremental backup set, see [Example using incremental backup sets](#).

When you create or add to an incremental backup set, `gpbackup` ensures that the backups in the set are created with a consistent set of backup options to ensure that the backup set can be used in a restore operation. For information about backup set consistency, see [Use incremental backups](#).

When you create an incremental backup you include these options with the other `gpbackup` options to create a backup:

- `--leaf-partition-data`: required for all backups in the incremental backup set.
 - Required when you create a full backup that will be the base backup for an incremental backup set.
 - Required when you create an incremental backup.
- `--incremental`: required when you create an incremental backup.

You cannot combine `--data-only` or `--metadata-only` with `--incremental`.

- `--from-timestamp`: optional. This option can be used with `--incremental`. The timestamp you specify is an existing backup. The timestamp can be either a full backup or incremental backup. The backup being created must be compatible with the backup specified with the `--from-timestamp` option.

Use incremental backups

When you add an incremental backup to a backup set, `gpbackup` ensures that the full backup and the incremental backups are consistent by checking these `gpbackup` options:

- `--dbname`: the database must be the same.
- `--backup-dir`: the directory must be the same. The backup set, the full backup and the incremental backups, must be in the same location.
- `--single-data-file`: this option must be either specified or absent for all backups in the set.
- `--include-table-file`, `--include-schema`, or any other options that filter tables and schemas must be the same. When checking schema filters, only the schema names are checked, not the objects contained in the schemas.

- `--no-compression`: if this option is specified, it must be specified for all backups in the backup set.

If compression is used on the full backup, compression must be used on the incremental backups. Different compression levels are allowed for the backups in the backup set. For a backup, the default is compression level 1.

If you try to add an incremental backup to a backup set, the backup operation fails if the `gpbackup` options are not consistent.

Example using incremental backup sets

Each backup has a timestamp taken when the backup is created. For example, if you create a backup on May 14, 2023, the backup file names contain `20230514hhmmss`. The `hhmmss` represents the time: hour, minute, and second.

This example assumes that you have created two full backups and incremental backups of the database `mytest`. To create the full backups, you used this command:

```
gpbackup --dbname mytest --backup-dir /mybackup --leaf-partition-data
```

You created incremental backups with this command:

```
gpbackup --dbname mytest --backup-dir /mybackup --leaf-partition-data --  
incremental
```

When you specify the `--backup-dir` option, the backups are created in the `/mybackup` directory on each SynxDB host.

In the example, the full backups have the timestamp keys `20230514054532` and `20231114064330`. The other backups are incremental backups. The example consists of two backup sets, the first with two incremental backups, and second with one incremental backup. The backups are listed from earliest to most recent.

- `20230514054532` (full backup)
- `20230714095512`
- `20230914081205`

- 20231114064330 (full backup)
- 20230114051246

To create a new incremental backup based on the latest incremental backup, you must include the same `--backup-dir` option as the incremental backup as well as the options `--leaf-partition-data` and `--incremental`.

```
gpbackup --dbname mytest --backup-dir /mybackup --leaf-partition-data --  
incremental
```

You can specify the `--from-timestamp` option to create an incremental backup based on an existing incremental or full backup. Based on the example, this command adds a fourth incremental backup to the backup set that includes 20230914081205 as an incremental backup and uses 20230514054532 as the full backup.

```
gpbackup --dbname mytest --backup-dir /mybackup --leaf-partition-data --  
incremental --from-timestamp 20230914081205
```

This command creates an incremental backup set based on the full backup 20231114064330 and is separate from the backup set that includes the incremental backup 20230114051246.

```
gpbackup --dbname mytest --backup-dir /mybackup --leaf-partition-data --  
incremental --from-timestamp 20231114064330
```

To restore a database with the incremental backup 20230914081205, you need the incremental backups 20120914081205 and 20230714095512, and the full backup 20230514054532. This would be the `gprestore` command.

```
gprestore --backup-dir /backupdir --timestamp 20230914081205
```

Create an incremental backup with `gpbackup`

The `gpbackup` output displays the timestamp of the backup on which the incremental backup is based. In this example, the incremental backup is based on the backup with timestamp 20230802171642. The backup 20230802171642 can be an incremental or full backup.

```
$ gpbackup --dbname test --backup-dir /backups --leaf-partition-data --
incremental

20230803:15:40:51 gpbackup:gpadmin:mdw:002907-[ INFO] :-Starting backup of
database test
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[ INFO] :-Backup Timestamp =
20230803154051
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[ INFO] :-Backup Database = test
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[ INFO] :-Gathering list of tables
for backup
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[ INFO] :-Acquiring ACCESS SHARE
locks on tables
Locks acquired: 5 / 5
[=====] 100.00% 0s
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[ INFO] :-Gathering additional
table metadata
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[ INFO] :-Metadata will be written
to /backups/gpseg-1/backups/20230803/20230803154051/gpbackup_20230803154051_
metadata.sql
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[ INFO] :-Writing global database
metadata
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[ INFO] :-Global database metadata
backup complete
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[ INFO] :-Writing pre-data
metadata
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[ INFO] :-Pre-data metadata backup
complete
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[ INFO] :-Writing post-data
metadata
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[ INFO] :-Post-data metadata
backup complete
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[ INFO] :-Basing incremental
backup off of backup with timestamp = 20230802171642
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[ INFO] :-Writing data to file
Tables backed up: 4 / 4
[=====] 100.00% 0s
20230803:15:40:52 gpbackup:gpadmin:mdw:002907-[ INFO] :-Data backup complete
20230803:15:40:53 gpbackup:gpadmin:mdw:002907-[ INFO] :-Found neither /usr/
local/greenplum-db./bin/gp_email_contacts.yaml nor /home/gpadmin/gp_email_
contacts.yaml
```

(continues on next page)

(continued from previous page)

```
20230803:15:40:53 gpbackup:gpadmin:mdw:002907-[INFO]:-Email containing
gpbackup report /backups/gpseg-1/backups/20230803/20230803154051/gpbackup_
20230803154051_report will not be sent
20230803:15:40:53 gpbackup:gpadmin:mdw:002907-[INFO]:-Backup completed
successfully
```

Restore from an incremental backup with gprestore

When restoring an from an incremental backup, you can specify the --verbose option to display the backups that are used in the restore operation on the command line. For example, the following gprestore command restores a backup using the timestamp 20230807092740, an incremental backup. The output includes the backups that were used to restore the database data.

```
$ gprestore --create-db --timestamp 20230807162904 --verbose
...
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[INFO]:-Pre-data metadata
restore complete
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Verifying backup file
count
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Restoring data from
backup with timestamp: 20230807162654
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for table
public.tbl_ao from file (table 1 of 1)
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Checking whether
segment agents had errors during restore
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Restoring data from
backup with timestamp: 20230807162819
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for table
public.test_ao from file (table 1 of 1)
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Checking whether
segment agents had errors during restore
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Restoring data from
backup with timestamp: 20230807162904
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for table
public.homes2 from file (table 1 of 4)
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for table
public.test2 from file (table 2 of 4)
20230807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for table
```

(continues on next page)

(continued from previous page)

```
public.homes2a from file (table 3 of 4)
20230807:16:31:56 gprestore:gadmin:mdw:008603-[DEBUG]:-Reading data for table
public.test2a from file (table 4 of 4)
20230807:16:31:56 gprestore:gadmin:mdw:008603-[DEBUG]:-Checking whether
segment agents had errors during restore
20230807:16:31:57 gprestore:gadmin:mdw:008603-[INFO]:-Data restore complete
20230807:16:31:57 gprestore:gadmin:mdw:008603-[INFO]:-Restoring post-data
metadata
20230807:16:31:57 gprestore:gadmin:mdw:008603-[INFO]:-Post-data metadata
restore complete
...
```

The output shows that the restore operation used three backups.

When restoring an from an incremental backup, `gprestore` also lists the backups that are used in the restore operation in the `gprestore` log file.

During the restore operation, `gprestore` displays an error if the full backup or other required incremental backup is not available.

Incremental backup notes

To create an incremental backup, or to restore data from an incremental backup set, you need the complete backup set. When you archive incremental backups, the complete backup set must be archived. You must archive all the files created on the coordinator and all segments.

If you do not specify the `--from-timestamp` option when you create an incremental backup, `gpbackup` uses the most recent backup with a consistent set of options.

If you specify the `--from-timestamp` option when you create an incremental backup, `gpbackup` ensures that the options of the backup that is being created are consistent with the options of the specified backup.

The `gpbackup` option `--with-stats` is not required to be the same for all backups in the backup set. However, to perform a restore operation with the `gprestore` option `-with-stats` to restore statistics, the backup you specify must have used the `--with-stats` when creating the backup.

You can perform a restore operation from any backup in the backup set. However, changes

captured in incremental backups later than the backup use to restore database data will not be restored.

When restoring from an incremental backup set, `gprestore` checks the backups and restores each append-optimized table from the most recent version of the append-optimized table in the backup set and restores the heap tables from the latest backup.

The incremental back up set, a full backup and associated incremental backups, must be on a single device. For example, the backups in a backup set must all be on a file system or must all be on a Data Domain system.

If you specify the `gprestore` option `--incremental` to restore data from a specific incremental backup, you must also specify the `--data-only` option. Before performing the restore operation, `gprestore` ensures that the tables being restored exist. If a table does not exist, `gprestore` returns an error and exits.



Warning

Changes to the SynxDB segment configuration invalidate incremental backups. After you change the segment configuration (add or remove segment instances), you must create a full backup before you can create an incremental backup.

CBDR

CBDR is a backup and recovery tool for SynxDB and Apache Cloudberry, built on top of WAL-G. It provides a simple command-line interface for performing backup and recovery operations, helping ensure data safety and enabling disaster recovery.

CBDR offers the following features:

- **Full backup:** Supports full backup of the entire database cluster.
- **Incremental backup:** Backs up only the changes made since the last backup.
- **Backup listing:** Displays all available backups.
- **Data recovery:** Restores data from a specified backup.
- **Storage support:** Supports S3-compatible object storage.
- **Configuration management:** Generates and manages the configuration files needed for backup and restore.

Tip

Compared to peer tools like `gpbackup` and `gprestore`, CBDR also supports storing backups to S3, multiple compression algorithms (lz4, lzma, zstd, brotli), and backup encryption.

Full backup and restore procedure

Before using CBDR to back up or restore a SynxDB cluster, make sure the following requirements are met:

- SynxDB is properly installed and running.
- The `wal-g` binary is installed under `/usr/local/bin/` or `/usr/bin/`.
- If using S3 storage, the appropriate credentials have been configured.

The general procedure for performing a backup using CBDR is as follows:

Backup process

1. Create a backup configuration file named config.yaml. Assume the file is located at /path/to/config.yaml. For the configuration file template, see [Configuration file reference](#).
2. Distribute the configuration file to all Segment nodes and update the archive command in postgresql.conf:

```
cbdr configure backup --config=/path/to/config.yaml
```

3. Restart the SynxDB cluster:

```
gpstop -ari
```

4. Perform the backup:

```
cbdr backup --config=/path/to/config.yaml
```

5. View the list of available backups:

```
cbdr backup-list --config=/path/to/config.yaml
```

Restore process

1. Prepare a new SynxDB cluster as the target for restoration, and create the required configuration file config.yaml. Assume the file is located at /path/to/config.yaml.
2. Generate the restore configuration file restore_cfg.json. Before running this command, make sure the new cluster is reachable:

```
cbdr configure restore --config=/path/to/config.yaml --restore-config=/path/to/restore_cfg.json
```

3. Delete all existing data directories on the new cluster, including both Coordinator and Segment nodes. For example:

```
rm -rf /data202502111728221784/coordinator/gpseg-1
rm -rf /data202502111728221784/segment/gpseg-0
rm -rf /data202502111728221784/segment/gpseg-1
rm -rf /data202502111728221784/segment/gpseg-2
```

4. Perform the restore:

```
cbdr restore --config=/path/to/config.yaml --restore-config=/path/to/
restore_cfg.json
```

5. Start the Coordinator node in admin mode and update the `gp_segment_configuration` system table to set the correct hostname, address, mirror, and other fields:

```
gpstart -c -a
```

If the following error occurs during startup, use `ps -ef | grep postgres` to check if the Coordinator process is running. If it is, you can safely ignore the error:

```
gpstart failed. (Reason='connection to server at "localhost" (127.0.0.1),
port 7000 failed: FATAL:  the database system is not accepting connections
DETAIL:  Hot standby mode is disabled.')
```

Once the Coordinator starts successfully, you can query the segment configuration:

```
select * from gp_segment_configuration;
```

If you have exited admin mode, restart the cluster in admin mode:

```
gpstop -c -a
```

6. Start the full cluster:

```
gpstart -a
```

You may encounter the following error during startup:

```
invalid IP mask "trust": Name or service not known
```

This happens because the `pg_hba.conf` file generated by WAL-G is missing CIDR masks (e.g., `/32`). You need to manually fix the configuration files for the Coordinator, Segment, and Mirror nodes.

Example of incorrect configuration:

host	all	all	<code>192.168.199.42</code>	trust
host	all	gpadmin	<code>192.168.192.159</code>	trust
host	all	gpadmin	<code>192.168.197.5</code>	trust

Corrected configuration:

host	all	all	<code>192.168.199.42/32</code>	trust
host	all	gpadmin	<code>192.168.192.159/32</code>	trust
host	all	gpadmin	<code>192.168.197.5/32</code>	trust

⚠ Attention

- Before backing up, make sure the database is running, configuration is correct, and there is enough available disk space.
- Prepare the restore environment in advance. Do not interrupt the restore process. After restoring, always verify data integrity.
- For storage management, regularly clean up invalid backups and monitor storage usage. If using S3, ensure a stable network connection.

Incremental backup and restore procedure

Before performing an incremental backup, make sure that you have completed at least one full backup.

1. On the source cluster, run the following command to start an incremental backup based on a specific full backup. Example:

```
cbdr backup --config=/path/to/config.yaml --delta-from-name=backup_
20250409T153036Z
```

⚠️ Attention

If you have not run the `cbdr configure backup` command on the current machine, or if you have run it before but the configuration file has changed, run `cbdr configure backup --config=/path/to/config.yaml` first. This command distributes the `/path/to/config.yaml` file from the coordinator node to the same file path on all segment nodes.

If you have run the `cbdr configure backup` command on the current machine, and the configuration file has not changed since then, you can simply run `cbdr backup`.

2. View all available backups (including full and incremental):

```
cbdr backup-list --config=/path/to/config.yaml
```

Sample output:

backup_name	modified	wal_
file_name	storage_name	
backup_20250409T153036Z	2025-04-09T15:31:36+08:00	
ZZZZZZZZZZZZZZZZZZZZZZZZ	default	
backup_20250409T153136Z_D_20250409T153036Z	2025-04-09T15:32:36+08:00	
ZZZZZZZZZZZZZZZZZZZZZZZZ	default	

3. Prepare a new SynxDB cluster as the target for restore. The preparation process is the same as for full backup restore.

4. Run the restore on the new cluster:

```
cbdr restore backup_20250409T153136Z_D_20250409T153036Z \
--config=/path/to/config.yaml \
--restore-config=/path/to/restore_cfg.json
```

Restore procedure using a restore point

CBDR supports creating logical restore points based on full or incremental backups. A restore point is essentially a named timestamp marker that allows a new cluster to be restored to a specific point in time.

Currently, CBDR only supports restoring to a restore point on a newly prepared cluster. It does not yet support incremental continuation from a restore point on an already restored cluster.

Below is an example of restoring a new cluster to a specified restore point:

1. Create a restore point on the source cluster:

```
cbdr create-restore-point "restore-point-1" --config=/path/to/config.yaml
```

- ## 2. View all backups and restore points:

```
cbdr backup-list --config=/path/to/config.yaml
```

Sample output:

```
backup_name           modified           wal_file_name
storage_name
backup_20250409T155552Z 2025-04-09T15:56:52+08:00 zzzzzzzzzzzzzzzzzzzzzzzzzzz
default
restore-point-1        2025-04-09T15:56:52+08:00 zzzzzzzzzzzzzzzzzzzzzzzzzzz
default
```

3. Prepare a new SynxDB cluster as the restore target. The preparation steps are the same as in the full backup restore scenario and are not repeated here.
 4. Run the restore on the new cluster, specifying the restore point name:

```
cbdr restore \
--config=/path/to/config.yaml \
--restore-config=/path/to/config.yaml \
--restore-point="restore-point-1"
```

Configuration file reference

CBDR uses YAML configuration files. Currently, it only supports configuring S3 storage parameters and does not support storing backups on the local file system. The following is a sample configuration.

```

# Database connection settings
PGHOST: "localhost"
PGPORT: 7000
PGUSER: "gpadmin"
PGDATABASE: "postgres"

# Concurrency settings
GOMAXPROCS: 6

# Relative path to the recovery configuration file
WALG_GP_RELATIVE_RECOVERY_CONF_PATH: "conf.d/recovery.conf"

# Polling interval for segment status
WALG_GP_SEG_POLL_INTERVAL: "1m"

# Compression method: supports lz4, lzma, zstd, brotli
WALG_COMPRESSION_METHOD: "lz4"

# Upload and download concurrency
WALG_UPLOAD_CONCURRENCY: 5
WALG_DOWNLOAD_CONCURRENCY: 5

# Retry attempts for file download
WALG_DOWNLOAD_FILE_RETRIES: 5

# Required settings for using S3 storage
WALE_S3_PREFIX: "xxxxxxxxxxxxxx"
AWS_ENDPOINT: "xxxxxxxxxxxxxx"
AWS_SECRET_ACCESS_KEY: "xxxxxxxxxxxxxx"
AWS_ACCESS_KEY_ID: "xxxxxxxxxxxxxx"

# Directory for log files
WALG_GP_LOGS_DIR: "/var/log/cbdr"

```

(continues on next page)

(continued from previous page)

```
# Incremental backup limit: maximum number of incremental backups allowed
# after a full backup. For example, if set to 6, a new full backup will be
# forced after 6 consecutive incremental backups to prevent long backup
# chains.

WALG_DELTA_MAX_STEPS: 10
```

Command usage

The basic syntax for running CBDR commands is:

```
cbdr <command> [options] --config=<config_file>
```

The following sections describe the main usage of each CBDR command.

Configure commands

The `cbdr configure` command is used to distribute the backup configuration to all Segment nodes or to generate a restore configuration file.

```
# Distribute the backup configuration and update the archive command in
postgresql.conf
cbdr configure backup --config=<config_file>

# Generate the restore configuration file
cbdr configure restore --config=<config_file> --restore-config=<restore_
config_file>
```

The restore configuration file can be automatically generated using `cbdr configure restore` (requires the cluster to be reachable), or it can be written manually. A sample JSON format is shown below:

```
{
  "segments": {
    "-1": {
      "hostname": "localhost",
      "port": 7000,
      "data_dir": "/tmp/tests/gpdemo/datadirs1/qddir/demoDataDir-1"
    }
  }
}
```

(continues on next page)

(continued from previous page)

```
        },
        "0": {
            "hostname": "localhost",
            "port": 7002,
            "data_dir": "/tmp/tests/gpdemo/datadirs1/dbfast1/demoDataDir0"
        },
        "1": {
            "hostname": "localhost",
            "port": 7003,
            "data_dir": "/tmp/tests/gpdemo/datadirs1/dbfast2/demoDataDir1"
        },
        "2": {
            "hostname": "localhost",
            "port": 7004,
            "data_dir": "/tmp/tests/gpdemo/datadirs1/dbfast3/demoDataDir2"
        }
    }
}
```

Cluster backup

The `cbdr backup` command is used to back up the database cluster. Syntax:

```
cbdr backup [options] --config=<config_file>
```

Optional parameters:

- **--permanent**: Marks the backup as permanent. It cannot be deleted unless forced.
 - **--full**: Performs a full backup.
 - **--add-user-data=<json>**: Attaches custom metadata to the backup in JSON format.
 - **--delta-from-user-data=<json>**: Specifies the base backup for incremental backup using metadata.
 - **--delta-from-name=<backup_name>**: Specifies the base backup for incremental backup by name.

View backup list

The cbdr backup-list command displays the list of available backups.

```
cbdr backup-list --config=<config_file> [options]
```

Optional parameters:

- **--pretty**: Outputs the list in a more readable format.
- **--json**: Outputs the list in JSON format.
- **--detail**: Displays detailed information.

Restore command

The cbdr restore command restores a backup to the target cluster. Usage:

```
cbdr restore <backup_name> --config=<config_file> [--restore-config=<restore_config_file>] [--target-user-data=<json>]
```

Parameter descriptions:

- **--restore-config**: Path to the restore configuration file.
- **--target-user-data**: Restores a backup that matches the specified user-defined metadata.

Delete command

The cbdr delete command removes an existing backup.

```
cbdr delete --config=<config_file> [--confirm] [--force-delete]
```

Parameter descriptions:

- **--confirm**: Must be explicitly set to execute the deletion.
- **--force-delete**: Forces deletion even for permanent backups.

Restore point commands

Restore points are used to support incremental recovery.

To create a restore point:

```
cbdr create-restore-point <restore-point-name> --config=<config_file>
```

To view existing restore points:

```
cbdr restore-point-list --config=<config_file> [--pretty] [--json] [--detail]
```

7.4 Configure Database System

Server configuration parameters affect the behavior of SynxDB. They are part of the PostgreSQL “Grand Unified Configuration” system, so they are sometimes called “GUCs”. Most of the SynxDB server configuration parameters are the same as the PostgreSQL configuration parameters, but some are specific to SynxDB.

Coordinator and local parameters

Server configuration files contain parameters that configure server behavior. The SynxDB configuration file, `postgresql.conf`, resides in the data directory of the database instance.

The coordinator and each segment instance have their own `postgresql.conf` file. Some parameters are local: each segment instance examines its `postgresql.conf` file to get the value of that parameter. Set local parameters on the coordinator and on each segment instance.

Other parameters are coordinator parameters that you set on the coordinator instance. The value is passed down to (or in some cases ignored by) the segment instances at query run time.

Set configuration parameters

Many configuration parameters limit who can change them and where or when they can be set. For example, to change certain parameters, you must be a SynxDB superuser. Other parameters can be set only at the system level in the `postgresql.conf` file or require a system restart to take effect.

Many configuration parameters are session parameters. You can set session parameters at the system level, the database level, the role level or the session level. Database users can change most session parameters within their session, but some require superuser permissions.

Set a local configuration parameter

To change a local configuration parameter across multiple segments, update the parameter in the `postgresql.conf` file of each targeted segment, both primary and mirror. Use the `gpconfig` utility to set a parameter in all SynxDB `postgresql.conf` files. For example:

```
$ gpconfig -c gp_vmem_protect_limit -v 4096
```

Restart SynxDB to make the configuration changes effective:

```
$ gpstop -r
```

Set a coordinator configuration parameter

To set a coordinator configuration parameter, set it at the SynxDB coordinator instance. If it is also a session parameter, you can set the parameter for a particular database, role or session. If a parameter is set at multiple levels, the most granular level takes precedence. For example, session overrides role, role overrides database, and database overrides system.

Set parameters at the system level

Coordinator parameter settings in the coordinator `postgresql.conf` file are the system-wide default. To set a coordinator parameter:

1. Edit the `$COORDINATOR_DATA_DIRECTORY/postgresql.conf` file.
2. Find the parameter to set, uncomment it (remove the preceding `#` character), and type the desired value.
3. Save and close the file.
4. For session parameters that do not require a server restart, upload the `postgresql.conf` changes as follows:

```
$ gpstop -u
```

For parameter changes that require a server restart, restart SynxDB as follows:

```
$ gpstop -r
```

Set parameters at the database level

Use `ALTER DATABASE` to set parameters at the database level. For example:

```
ALTER DATABASE mydatabase SET search_path TO myschema;
```

When you set a session parameter at the database level, every session that connects to that database uses that parameter setting. Settings at the database level override settings at the system level.

Set parameters at the row level

Use `ALTER ROLE` to set a parameter at the role level. For example:

```
ALTER ROLE bob SET search_path TO bobschema;
```

When you set a session parameter at the role level, every session initiated by that role uses that parameter setting. Settings at the role level override settings at the database level.

Set parameters in a session level

Any session parameter can be set in an active database session using the `SET` command. For example:

```
SET statement_mem TO '200MB';
```

The parameter setting is valid for the rest of that session or until you issue a `RESET` command. For example:

```
RESET statement_mem;
```

Settings at the session level override those at the role level.

View server configuration parameter settings

The SQL command `SHOW` allows you to see the current server configuration parameter settings. For example, to see the settings for all parameters:

```
$ psql -c 'SHOW ALL;'
```

`SHOW` lists the settings for the coordinator instance only. To see the value of a particular parameter across the entire system (and all segments), use the `gpconfig` utility. For example:

```
$ gpconfig --show max_connections
```

View parameters using the `pg_settings` view

In addition, you can use the `pg_settings` view to check all parameter settings:

```
SELECT name, setting, unit, category, short_desc, context, vartype, min_val,
max_val
FROM pg_settings
ORDER BY category, name;
```

Field descriptions:

- **name:** Parameter name.
- **setting:** Current parameter value.
- **unit:** Parameter unit (if applicable, such as ms and kB).
- **category:** The configuration category that the parameter belongs to.
- **short_desc:** Brief description of the parameter.
- **context:** Required context for parameter modification, possible values include:
 - **internal:** Used internally by the system, cannot be modified directly.
 - **postmaster:** Requires database instance restart to take effect.
 - **sighup:** Loads configuration via `pg_ctl reload` or `SELECT pg_reload_conf();`.

- superuser: Only superusers can modify.
- user: Regular users can modify.
- vartype: Parameter type, such as bool, integer, real, string.
- min_val and max_val: Minimum and maximum allowed values (if applicable).

Use cases:

- View parameters of a specific category:

```
SELECT name, setting, short_desc
FROM pg_settings
WHERE category = 'Resource Usage / Memory';
```

- Search for parameters containing specific keywords:

```
SELECT name, setting, short_desc
FROM pg_settings
WHERE name LIKE '%work_mem%';
```

- Check parameters requiring restart to take effect:

```
SELECT name, setting, context
FROM pg_settings
WHERE context = 'postmaster';
```

- Display parameters effective in the current database session:

```
SELECT name, setting
FROM pg_settings
WHERE context = 'user';
```

Note

In SynxDB, because of its distributed architecture, some parameters can only be configured on the coordinator node or cluster-wide. When modifying parameters using the `gpconfig` tool, pay attention to synchronization between primary, mirror, and segment nodes.

To change configuration parameters, you can use SQL commands, for example:

```
ALTER SYSTEM SET parameter_name = 'value';
```

Or edit the `postgresql.conf` file and make it effective through `pg_reload_conf()` or service restart.

Chapter 8

Reference Guides

8.1 System Utilities

gpdemo

You can use the built-in `gpdemo` utility to quickly deploy a cluster with a specified number of segments only with a single command.

`gpdemo` is installed with other system utilities (such as `gpinitsystem`, `gpstart`, and `gpstop`) in the `GPHOME/bin` directory.

 **Warning**

It is not recommended to use `gpdemo` for production environments, except when deploying a single node cluster.

How to use

You can run `gpdemo` once the RPM package is installed on the target server and `greenplum_path.sh` is correctly loaded.

Deploy with default settings

To quickly deploy a cluster without special requirements for the data directory location, initial port number, or segment count, you can run `gpdemo` without any parameter:

```
gpdemo
```

After running this command, the script creates a cluster in the current path. The default configuration of the cluster is as follows:

- Segment count: 3
- Data directory: `./datadirs`
- The initial port number of coordinator: 7000

Note

Once a cluster is deployed, the script generates the `gpdemo-env.sh` file in the current directory containing the basic information of the cluster. Before using `gpdemo` to operate on the cluster, run `source gpdemo-env.sh` to load this information.

You need to run this command for each new shell session. To load the cluster information automatically, you can add this command to `~/.bashrc` or `~/.zshrc`.

Deploy with customized settings

If you need to customize the number of segments, data directory location, and the initial port number of the cluster, refer to the following instructions.

Specify segment count

For the default deployment, the number of segments in a cluster is 3.

To specify the number of segments in a cluster, you can add the NUM_PRIMARY_MIRROR_PAIRS parameter to the gpdemo command. For example, to specify the number of segments as 3, you can run:

```
NUM_PRIMARY_MIRROR_PAIRS=3 gpdemo
```

Note

- Each segment node consists of a primary node and a mirror node. So every time the parameter value increases by 1, 2 more nodes will be created. To better capture data distribution issues, it is recommended to set the value to an odd number.
- When the parameter value is set to 0, a single-computing-node cluster is deployed. See [Deploy |product_name| with a Single Computing Node](#) for details.

Specify the data directory of a node

Each node's data in a cluster is stored in the ./datadirs subdirectory of the current directory. To set the cluster's data directory location, use one of the following methods:

- Switch to the target directory using cd and run gpdemo.
- In the gpdemo command, specify the DATADIRS parameter to set the data directory. For example:

```
DATADIRS=/target/directory/ gpdemo
```

Note

After a cluster is deployed, if you want to manually adjust the location of the data directory (by modifying the `pg_hba.conf` configuration file), you might need to know the default directories of the coordinator node and segment nodes.

- Regular multi-node mode (which means `NUM_PRIMARY_MIRROR_PAIRS > 0`)
 - Coordinator: `./datadirs/qddir`
 - Coordinator Standby: `./datadirs/standby`
 - Segments
 - * Primary: `./datadirs/dbfast<N>`, where `<N>` is the segment number.
 - * Mirror: `./datadirs/dbfast_mirror<N>`, where `<N>` is the segment number.
- Single-node mode (which means `NUM_PRIMARY_MIRROR_PAIRS = 0`)
 - Coordinator: `./datadirs/singlenodedir`
 - Coordinator Standby: `./datadirs/standby`

Specify the initial port number of the coordinator node

By default, the port number of the Coordinator node is 7000. Starting from this number, all other nodes in the cluster are assigned consecutive port numbers.

To specify the initial port number of a cluster's coordinator node, you can add the `PORt_BASE` parameter to the `gpdemo` command. For example:

```
PORt_BASE=7000 gpdemo
```

Specify additional initialization parameter for each node in the cluster

`BLDWRAP_POSTGRES_CONF_ADDONS` specifies the additional initialization parameter for each node in a cluster. The default value `fsync=off` is for better concurrency performance.

In a production environment, use `export BLDWRAP_POSTGRES_CONF_ADDONS = "fsync=on"` to enable fsync. Otherwise, the ACID characteristics of the database will be affected. In the single-node mode, fsync is automatically enabled.

Command-line options

`gpdemo` provides the following command-line options. You can also check them using `gpdemo -H`.

Option	Description
<code>-K</code>	Skips the data consistency check when deploying a cluster. This option is not recommended.
<code>-c</code>	Checks the port occupancy and confirms whether a test cluster can be deployed. Before running <code>gpdemo -c</code> , you need to run <code>source gpdemo-env.sh</code> to load the basic information of a cluster. The <code>sh</code> file is located in the directory where the cluster is created.
<code>-d</code>	Deletes the test cluster. Before running <code>gpdemo -c</code> , you need to run <code>source gpdemo-env.sh</code> to load the basic information of a cluster. The <code>sh</code> file is located in the directory where the cluster is created.
<code>-p</code>	Views cluster status, version and other information of the coordinator and segment nodes. Before running <code>gpdemo -c</code> , you need to run <code>source gpdemo-env.sh</code> to load the basic information of a cluster. The <code>sh</code> file is located in the directory where the cluster is created.
<code>-H or -H -H</code>	<code>gpdemo -h</code> outputs a brief help description. <code>gpdemo -H</code> outputs a more detailed help description, including additional configurations such as environment variables.
<code>-v</code>	Checks the current <code>gpdemo</code> version.

8.2 System Catalogs

System Views

This document introduces selected system views available in SynxDB.

pg_stat_progress_create_index

`pg_stat_progress_create_index` is a system view that shows real-time progress for ongoing `CREATE INDEX` or `REINDEX` operations in the current database.

This view displays detailed status information for each backend process building an index, including the current execution phase, the number of tuples processed, blocks scanned, and more. Once the operation completes, the corresponding entries are automatically removed from the view.

In SynxDB, this view also supports AO (Append-Optimized) tables and can be used to observe the phase-wise progress of index creation on such tables.

Typical use cases include:

- Monitoring index creation or rebuild operations in real time.
- Analyzing performance bottlenecks of long-running `CREATE INDEX` or `REINDEX` commands.
- Checking if any index operations are currently consuming system resources.
- Correlating with `pg_stat_activity` to trace backend process details.

Example queries:

```
-- Views all ongoing index creation tasks
SELECT * FROM pg_stat_progress_create_index;

-- Views index progress for a specific table
SELECT * FROM pg_stat_progress_create_index
WHERE relid = 'ao_test'::regclass;
```

Field descriptions

Field	Description
gp_segment_id	ID of the segment where this entry resides. Only applicable in a distributed environment.
pid	Process ID of the backend. Can be joined with pg_stat_activity for session details.
datid	OID of the database, corresponding to pg_database.oid.
datname	Name of the database.
relid	OID of the table being indexed, corresponding to pg_class.oid.
index_relid	OID of the index being built.
command	Type of command being executed: either CREATE INDEX or REINDEX.
phase	Current phase of execution, such as: <ul style="list-style-type: none"> • initializing: Initialization • scanning heap: Scanning table data • sorting: Sorting phase • building index: loading tuples in tree: Building the index structure • waiting for locks: Waiting for table or metadata locks
lockers_total	Total number of sessions holding locks (if any).
lockers_done	Number of sessions that have released their locks.
current_locker_pid	Process ID of the session currently holding the lock (if waiting).
blocks_total	Total number of data blocks to scan (may be 0 if unavailable or not started).
blocks_done	Number of data blocks scanned so far.
tuples_total	Estimated total number of tuples to process (if computable).
tuples_done	Number of tuples processed so far.
partitions_total	Total number of partitions (for partitioned tables, if applicable).
partitions_done	Number of partitions processed (if applicable).

Note

- This view only displays currently running index operations. Entries are removed once the operation completes.
- For small tables, index creation may complete instantly, and the view might return no rows.
- To better observe progress, try creating an index on a large table or use complex columns

(e.g., large text) to slow down execution.

- Progress is also reported when building indexes on AO (Append-Optimized) tables.
- You can join this view with pg_stat_activity using the pid field.

```
SELECT a.username, a.query, p.phase, p.blocks_done, p.blocks_total
FROM pg_stat_activity a
JOIN pg_stat_progress_create_index p ON a.pid = p.pid;
```

The gp_toolkit Administrative Schema

SynxDB provides an administrative schema called `gp_toolkit` that you can use to query the system catalogs, log files, and operating environment for system status information. The `gp_toolkit` schema contains a number of views that you can access using SQL commands. The `gp_toolkit` schema is accessible to all database users, although some objects may require superuser permissions. For convenience, you may want to add the `gp_toolkit` schema to your schema search path. For example:

```
=> ALTER ROLE myrole SET search_path TO myschema, gp_toolkit;
```

This document describes the most useful views and user-defined functions (UDFs) in `gp_toolkit`. You may notice other objects (views, functions, and external tables) within the `gp_toolkit` schema that are not described in this documentation (these are supporting objects to the views described in this section).

Caution

Do not change database objects in the `gp_toolkit` schema. Do not create database objects in the schema. Changes to objects in the schema might affect the accuracy of administrative information returned by schema objects. Any changes made in the `gp_toolkit` schema are lost when the database is backed up and then restored with the `gpbackup` and `gprestore` utilities.

These are the categories for views in the `gp_toolkit` schema.

- *Check for tables that need routine maintenance*

- *Check for locks*
- *Check append-optimized tables*
- *View server log files*
- *Check server configuration files*
- *Check for failed segments*
- *Check resource group activity and status*
- *Check resource queue activity and status*
- *Check query disk spill space usage*
- *View users and groups (roles)*
- *Check database object sizes and disk space*
- *Check for missing and orphaned data files*
- *Move orphaned data files*
- *Check for uneven data distribution*
- *Maintain partitions*

About the extension

`gp_toolkit` is implemented as an extension in SynxDB. Because this extension is registered in the `template1` database, it is both registered and immediately available to use in every SynxDB database that you create.

Upgrade the extension

The `gp_toolkit` extension is installed when you install or upgrade SynxDB. A previous version of the extension will continue to work in existing databases after you upgrade SynxDB. To upgrade to the most recent version of the extension, you must:

```
ALTER EXTENSION gp_toolkit UPDATE TO '1.4';
```

in **every** database in which you use the extension.

Check for tables that need routine maintenance

The following views can help identify tables that need routine table maintenance (VACUUM and/or ANALYZE).

- *gp_bloat_diag*
- *gp_stats_missing*

The VACUUM or VACUUM FULL command reclaims disk space occupied by deleted or obsolete rows. Because of the MVCC transaction concurrency model used in SynxDB, data rows that are deleted or updated still occupy physical space on disk even though they are not visible to any new transactions. Expired rows increase table size on disk and eventually slow down scans of the table.

The ANALYZE command collects column-level statistics needed by the query optimizer. SynxDB uses a cost-based query optimizer that relies on database statistics. Accurate statistics allow the query optimizer to better estimate selectivity and the number of rows retrieved by a query operation in order to choose the most efficient query plan.

gp_bloat_diag

This view shows regular heap-storage tables that have bloat (the actual number of pages on disk exceeds the expected number of pages given the table statistics). Tables that are bloated require a VACUUM or a VACUUM FULL in order to reclaim disk space occupied by deleted or obsolete rows. This view is accessible to all users, however non-superusers will only be able to see the tables that they have permission to access.

Note

For diagnostic functions that return append-optimized table information, see *Check append-optimized tables*.

The columns of this view are described as follows:

- bdirelId: Table object id.

- bdinspname: Schema name.
- bdirelname: Table name.
- bdirelpages: Actual number of pages on disk.
- bdiexppages: Expected number of pages given the table data.
- bdidiag: Bloat diagnostic message.

gp_stats_missing

This view shows tables that do not have statistics and therefore may require an ANALYZE be run on the table.

The columns of this view are described as follows:

- smischema: Schema name.
- smitable: Table name.
- smisize: Does this table have statistics? False if the table does not have row count and row sizing statistics recorded in the system catalog, which may indicate that the table needs to be analyzed. This will also be false if the table does not contain any rows. For example, the parent tables of partitioned tables are always empty and will always return a false result.
- smicols: Number of columns in the table.
- smirecs: The total number of columns in the table that have statistics recorded.

Check for locks

When a transaction accesses a relation (such as a table), it acquires a lock. Depending on the type of lock acquired, subsequent transactions may have to wait before they can access the same relation. For more information on the types of locks, see the [Transaction](#). SynxDB resource queues (used for resource management) also use locks to control the admission of queries into the system.

The gp_locks_* family of views can help diagnose queries and sessions that are waiting to access an object due to a lock.

- [*gp_locks_on_relation*](#)

- *gp_locks_on_resqueue*

gp_locks_on_relation

This view shows any locks currently being held on a relation, and the associated session information about the query associated with the lock. For more information on the types of locks, see the [Transaction](#). This view is accessible to all users, however non-superusers will only be able to see the locks for relations that they have permission to access.

The columns of this view are described as follows:

- `lorlocktype`: Type of the lockable object: `relation`, `extend`, `page`, `tuple`, `transactionid`, `object`, `userlock`, `resource queue`, or `advisory`.
- `lорdatabase`: Object ID of the database in which the object exists, zero if the object is a shared object.
- `lorrelname`: The name of the relation.
- `lorrelation`: The object ID of the relation.
- `lortransaction`: The transaction ID that is affected by the lock.
- `lorpid`: Process ID of the server process holding or awaiting this lock. `NULL` if the lock is held by a prepared transaction.
- `lormode`: Name of the lock mode held or desired by this process.
- `lorgranted`: Displays whether the lock is granted (`true`) or not granted (`false`).
- `lorcurrentquery`: The current query in the session.

gp_locks_on_resqueue

Note

The `gp_locks_on_resqueue` view is valid only when resource queue-based resource management is active.

This view shows any locks currently being held on a resource queue, and the associated session information about the query associated with the lock. This view is accessible to all users, however non-superusers will only be able to see the locks associated with their own sessions.

The columns of this view are described as follows:

- `lorusename`: Name of the user running the session.
- `lorrsqname`: The resource queue name.
- `lorlocktype`: Type of the lockable object: `resource queue`.
- `lorobjid`: The ID of the locked transaction.
- `lortransaction`: The ID of the transaction that is affected by the lock.
- `lorpid`: The process ID of the transaction that is affected by the lock.
- `lormode`: The name of the lock mode held or desired by this process.
- `lorgranted`: Displays whether the lock is granted (`true`) or not granted (`false`).
- `lorwaiteventtype`: The type of waiting event (for example, I/O, lock, network).
- `lorwaitevent`: The name or description of the waiting event (for example, the I/O operation or lock object of a file).

Check append-optimized tables

The `gp_toolkit` schema includes a set of diagnostic functions you can use to investigate the state of append-optimized tables.

When an append-optimized table (or column-oriented append-optimized table) is created, another table is implicitly created, containing metadata about the current state of the table. The metadata includes information such as the number of records in each of the table's segments.

Append-optimized tables may have non-visible rows—rows that have been updated or deleted, but remain in storage until the table is compacted using `VACUUM`. The hidden rows are tracked using an auxiliary visibility map table, or `visimap`.

The following functions let you access the metadata for append-optimized and column-oriented tables and view non-visible rows.

For most of the functions, the input argument is `regclass`, either the table name or the `oid` of a table.

`__gp_aovisimap_compaction_info(oid)`

This function displays compaction information for an append-optimized table. The information is for the on-disk data files on database segments that store the table data. You can use the information to determine the data files that will be compacted by a VACUUM operation on an append-optimized table.

Note

Until a VACUUM operation deletes the row from the data file, deleted or updated data rows occupy physical space on disk even though they are hidden to new transactions. The configuration parameter `gp_appendonly_compaction` controls the functionality of the VACUUM command.

This table describes the `__gp_aovisimap_compaction_info` function output table. The columns of this table are described as follows:

- `content`: Database segment ID.
- `datafile`: ID of the data file on the segment.
- `compaction_possible`: The value is either `t` or `f`. The value `t` indicates that the data in data file be compacted when a VACUUM operation is performed. The server configuration parameter `gp_appendonly_compaction_threshold` affects this value.
- `hidden_tupcount`: In the data file, the number of hidden (deleted or updated) rows.
- `total_tupcount`: In the data file, the total number of rows.
- `percent_hidden`: In the data file, the ratio (as a percentage) of hidden (deleted or updated) rows to total rows.

__gp_aoseg(regclass)

This function returns metadata information contained in the append-optimized table’s on-disk segment file.

The input argument is the name or the oid of an append-optimized table. The columns of the output are described as follows:

- `segment_id`: The ID of the data segment in the cluster.
- `segno`: The file segment number.
- `eof`: The effective end of file for this file segment.
- `tupcount`: The total number of tuples in the segment, including invisible tuples.
- `varblockcount`: The total number of varblocks in the file segment.
- `eof_uncompressed`: The end of file if the file segment were uncompressed.
- `modcount`: The number of data modification operations.
- `formatversion`: The version of the AO file storage format, which indicates the format version of the file.
- `state`: The state of the file segment. Indicates if the segment is active or ready to be dropped after compaction.

__gp_aoseg_history(regclass)

This function returns metadata information contained in the append-optimized table’s on-disk segment file. It displays all different versions (heap tuples) of the aoseg meta information. The data is complex, but users with a deep understanding of the system may find it useful for debugging.

The input argument is the name or the oid of an append-optimized table. The columns of the output are described as follows:

- `segment_id`: The ID of the data segment in the cluster.
- `segno`: The number of the segment in the segment file.
- `tupcount`: The number of tuples, including hidden tuples.

- `eof`: The effective end of file for the segment.
- `eof_uncompressed`: The end of file for the segment if data were uncompressed.
- `modcount`: A count of data modifications.
- `formatversion`: The version of the AO file storage format, which indicates the format version of the file.
- `state`: The status of the segment.

__gp_aocsseg(regclass)

This function returns metadata information contained in a column-oriented append-optimized table's on-disk segment file, excluding non-visible rows. Each row describes a segment for a column in the table.

The input argument is the name or the oid of a column-oriented append-optimized table. The columns of the output are described as follows:

- `segment_id`: The ID of the data segment in the cluster.
- `segno`: The segment number.
- `column_num`: The column number.
- `physical_segno`: The number of the segment in the segment file.
- `tupcount`: The number of rows in the segment, excluding hidden tuples.
- `eof`: The effective end of file for the segment.
- `eof_uncompressed`: The end of file for the segment if the data were uncompressed.
- `modcount`: A count of data modification operations for the segment.
- `formatversion`: The version of the AO file storage format, which indicates the format version of the file.
- `state`: The status of the segment.

__gp_aocsseg_history(regclass)

This function returns metadata information contained in a column-oriented append-optimized table's on-disk segment file. Each row describes a segment for a column in the table. The data is complex, but users with a deep understanding of the system may find it useful for debugging.

The input argument is the name or the oid of a column-oriented append-optimized table. The columns of the output are described as follows:

- **segment_id**: The ID of the data segment in the cluster.
- **segno**: The segment number in the segment file.
- **column_num**: The column number.
- **physical_segno**: The segment containing data for the column.
- **tupcount**: The total number of tuples in the segment.
- **eof**: The effective end of file for the segment.
- **eof_uncompressed**: The end of file for the segment if the data were uncompressed.
- **modcount**: A count of the data modification operations.
- **formatversion**: The version of the AO file storage format, which indicates the format version of the file.
- **state**: The state of the segment.

__gp_aovisimap(regclass)

This function returns the tuple ID, the segment file, and the row number of each non-visible tuple according to the visibility map.

The input argument is the name or the oid of an append-optimized table. The columns of the output are described as follows:

- **tid**: The tuple id.
- **segno**: The number of the segment file.

- `row_num`: The row number of a row that has been deleted or updated.

`__gp_aovisimap_hidden_info(regclass)`

This function returns the numbers of hidden and visible tuples in the segment files for an append-optimized table.

The input argument is the name or the oid of an append-optimized table. The columns of the output are described as follows:

- `segno`: The number of the segment file.
- `hidden_tupcount`: The number of hidden tuples in the segment file.
- `total_tupcount`: The total number of tuples in the segment file.

`__gp_aovisimap_entry(regclass)`

This function returns information about each visibility map entry for the table.

The input argument is the name or the oid of an append-optimized table. The columns of the output are described as follows:

- `segno`: Segment number of the visibility map entry.
- `first_row_num`: The first row number of the entry.
- `hidden_tupcount`: The number of hidden tuples in the entry.
- `bitmap`: A text representation of the visibility bitmap.

`__gp_aoblkdir(regclass)`

For a given AO/AOCO table that had or has an index, this function returns a row for each block directory entry recorded in the block directory relation; it flattens the `minipage` column of block directory relations and returns a row for each `minipage` entry.

The input argument is the name or the oid of an append-optimized table.

You must execute this function in utility mode against every segment, or with

`gp_dist_random()` as shown here:

```
SELECT (gp_toolkit.__gp_aobkdir('<table_name>')).*
FROM gp_dist_random('gp_id');
```

The columns of the output are described as follows:

- `tupleid`: The tuple id of the block directory row containing this block directory entry.
- `segno`: The physical segment file number.
- `columngroup_no`: The `attnum` of the column described by this minipage entry (always 0 for row-oriented tables).
- `entry_no`: The entry serial number inside this minipage containing this block directory entry.
- `first_row_no`: The first row number of the rows covered by this block directory entry.
- `file_offset`: The starting file offset of the rows covered by this block directory entry.
- `row_count`: The count of rows covered by this block directory entry.

get_column_size(oid)

For a given AOCO table, this function returns the column size and compression ratio for all columns in the table.

The input argument is the object identifier of a column-oriented append-optimized table. The columns of the output are described as follows:

- `segment`: The segment id.
- `attnum`: The attribute number of the column.
- `size`: The size of the column in bytes.
- `size_uncompressed`: The size of the column in bytes if the column was not compressed.
- `compression_ratio`: The compression ratio.

gp_column_size

This view gathers the column size and compression ratio for column-oriented append-optimized tables from all segments. The columns of the view are described as follows:

- gp_segment_id: Segment ID.
- relid: The object identifier (OID) of the table.
- schema: The schema to which the table belongs.
- relname: The table name.
- attnum: The attribute number of the column.
- attname: The column name.
- size: The size of the column in bytes.
- size_uncompressed: The size of the column in bytes if the column was not compressed.
- compression_ratio: The compression ratio.

gp_column_size_summary

This view shows a summary of the gp_column_size view. It aggregates the column size and compression ratio for each column in each column-oriented append-optimized table from all segments. The columns of the view are described as follows:

- relid: The object identifier (OID) of the table.
- schema: The schema to which the table belongs.
- relname: The table name.
- attnum: The attribute number of the column.
- attname: The column name.
- size: The size of the column in bytes.
- size_uncompressed: The size of the column in bytes if the column were uncompressed.

- `compression_ratio`: The compression ratio.

View server log files

Each component of a SynxDB system (coordinator, standby coordinator, primary segments, and mirror segments) keeps its own server log files. The `gp_log_*` family of views allows you to issue SQL queries against the server log files to find particular entries of interest. The use of these views require superuser permissions.

- `gp_log_command_timings`
- `gp_log_database`
- `gp_log_coordinator_concise`
- `gp_log_system`

`gp_log_command_timings`

This view uses an external table to read the log files on the coordinator and report the run time of SQL commands in a database session. The use of this view requires superuser permissions. The columns of the view are described as follows:

- `logsession`: The session identifier (prefixed with “con”).
- `logcmdcount`: The command number within a session (prefixed with “cmd”).
- `logdatabase`: The name of the database.
- `loguser`: The name of the database user.
- `logpid`: The process id (prefixed with “p”).
- `logtimemin`: The time of the first log message for this command.
- `logtimemax`: The time of the last log message for this command.
- `logduration`: Statement duration from start to end time.

gp_log_database

This view uses an external table to read the server log files of the entire SynxDB system (coordinator, segments, and mirrors) and lists log entries associated with the current database. Associated log entries can be identified by the session id (logsession) and command id (logcmdcount). The use of this view requires superuser permissions. The columns of the view are described as follows:

- **logtime**: The timestamp of the log message.
- **loguser**: The name of the database user.
- **logdatabase**: The name of the database.
- **logpid**: The associated process id (prefixed with “p”).
- **logthread**: The associated thread count (prefixed with “th”).
- **loghost**: The segment or coordinator host name.
- **logport**: The segment or coordinator port.
- **logsessiontime**: Time session connection was opened.
- **logtransaction**: Global transaction id.
- **logsession**: The session identifier (prefixed with “con”).
- **logcmdcount**: The command number within a session (prefixed with “cmd”).
- **logsegment**: The segment content identifier (prefixed with “seg” for primary or “mir” for mirror. The coordinator always has a content id of -1).
- **logslice**: The slice id (portion of the query plan being run).
- **logdistxact**: Distributed transaction id.
- **loglocalxact**: Local transaction id.
- **logsubxact**: Subtransaction id.
- **logseverity**: LOG, ERROR, FATAL, PANIC, DEBUG1 or DEBUG2.
- **logstate**: SQL state code associated with the log message.

- **logmessage:** Log or error message text.
- **logdetail:** Detail message text associated with an error message.
- **loghint:** Hint message text associated with an error message.
- **logquery:** The internally-generated query text.
- **logquerypos:** The cursor index into the internally-generated query text.
- **logcontext:** The context in which this message gets generated.
- **logdebug:** Query string with full detail for debugging.
- **logcursorpos:** The cursor index into the query string.
- **logfunction:** The function in which this message is generated.
- **logfile:** The log file in which this message is generated.
- **logline:** The line in the log file in which this message is generated.
- **logstack:** Full text of the stack trace associated with this message.

gp_log_coordinator_concise

This view uses an external table to read a subset of the log fields from the coordinator log file. The use of this view requires superuser permissions. The columns of the view are described as follows:

- **logtime:** The timestamp of the log message.
- **logdatabase:** The name of the database.
- **logsession:** The session identifier (prefixed with “con”).
- **logcmdcount:** The command number within a session (prefixed with “cmd”).
- **logseverity:** The log severity level.
- **logmessage:** Log or error message text.

gp_log_system

This view uses an external table to read the server log files of the entire SynxDB system (coordinator, segments, and mirrors) and lists all log entries. Associated log entries can be identified by the session id (logsession) and command id (logcmdcount). The use of this view requires superuser permissions. The columns of the view are described as follows:

- **logtime:** The timestamp of the log message.
- **loguser:** The name of the database user.
- **logdatabase:** The name of the database.
- **logpid:** The associated process id (prefixed with “p”).
- **logthread:** The associated thread count (prefixed with “th”).
- **loghost:** The segment or coordinator host name.
- **logport:** The segment or coordinator port.
- **logsessiontime:** Time session connection was opened.
- **logtransaction:** Global transaction id.
- **logsession:** The session identifier (prefixed with “con”).
- **logcmdcount:** The command number within a session (prefixed with “cmd”).
- **logsegment:** The segment content identifier (prefixed with “seg” for primary or “mir” for mirror. The coordinator always has a content id of -1).
- **logslice:** The slice id (portion of the query plan being run).
- **logdistxact:** Distributed transaction id.
- **loglocalxact:** Local transaction id.
- **logsubxact:** Subtransaction id.
- **logseverity:** LOG, ERROR, FATAL, PANIC, DEBUG1 or DEBUG2.
- **logstate:** SQL state code associated with the log message.

- `logmessage`: Log or error message text.
- `logdetail`: Detail message text associated with an error message.
- `loghint`: Hint message text associated with an error message.
- `logquery`: The internally-generated query text.
- `logquerypos`: The cursor index into the internally-generated query text.
- `logcontext`: The context in which this message gets generated.
- `logdebug`: Query string with full detail for debugging.
- `logcursorpos`: The cursor index into the query string.
- `logfunction`: The function in which this message is generated.
- `logfile`: The log file in which this message is generated.
- `logline`: The line in the log file in which this message is generated.
- `logstack`: Full text of the stack trace associated with this message.

Check server configuration files

Each component of a SynxDB system (coordinator, standby coordinator, primary segments, and mirror segments) has its own server configuration file (`postgresql.conf`). The following `gp_toolkit` objects can be used to check parameter settings across all primary `postgresql.conf` files in the system:

- `gp_param_setting('parameter_name')`
- `gp_param_settings_seg_value_diffs`

`gp_param_setting('parameter_name')`

This function takes the name of a server configuration parameter and returns the `postgresql.conf` value for the coordinator and each active segment. This function is accessible to all users. The columns of the view are described as follows:

- `paramsegment`: The segment content id (only active segments are shown). The

coordinator content id is always -1.

- paramname: The name of the parameter.
- paramvalue: The value of the parameter.

Example:

```
SELECT * FROM gp_toolkit.gp_param_setting('max_connections');
```

gp_param_settings_seg_value_diffs

Server configuration parameters that are classified as *local* parameters (meaning each segment gets the parameter value from its own `postgresql.conf` file), should be set identically on all segments. This view shows local parameter settings that are inconsistent. Parameters that are supposed to have different values (such as `port`) are not included. This view is accessible to all users.

The columns of the view are described as follows:

- psdname: The name of the parameter.
- psdvalue: The value of the parameter.
- psdcount: The number of segments that have this value.

Check for failed segments

The `gp_pgdatabase_invalid` view can be used to check for down segments.

gp_pgdatabase_invalid

This view shows information about segments that are marked as down in the system catalog. This view is accessible to all users. The columns of the view are described as follows:

- pgdbidbid: The segment dbid. Every segment has a unique dbid.
- pgdbiisprimary: Is the segment currently acting as the primary (active) segment? (t or f).

- pgdbicontent: The content id of this segment. A primary and mirror will have the same content id.
- pgdbivalid: Is this segment up and valid? (t or f).
- pgdbidefinedprimary: Was this segment assigned the role of primary at system initialization time? (t or f).

Check resource group activity and status

Note

The resource group activity and status views described in this section are valid only when resource group-based resource management is active.

Resource groups manage transactions to avoid exhausting system CPU and memory resources. Every database user is assigned a resource group. SynxDB evaluates every transaction submitted by a user against the limits configured for the user's resource group before running the transaction.

You can use the `gp_resgroup_config` view to check the configuration of each resource group. You can use the `gp_resgroup_status*` views to display the current transaction status and resource usage of each resource group.

- `gp_resgroup_config`
- `gp_resgroup_role`
- `gp_resgroup_status`
- `gp_resgroup_status_per_host`
- `gp_resgroup_status_per_segment`

gp_resgroup_config

The `gp_resgroup_config` view allows administrators to see the current CPU, memory, and concurrency limits for a resource group.

This view is accessible to all users. The columns of the view are described as follows:

- `groupid`: The ID of the resource group.
- `groupname`: The name of the resource group.
- `concurrency`: The concurrency (CONCURRENCY) value specified for the resource group.
- `cpu_max_percent`: The CPU limit (CPU_MAX_PERCENT) value specified for the resource group, or -1.
- `cpu_weight`: The scheduling priority of the resource group (CPU_WEIGHT).
- `cpuset`: The CPU cores reserved for the resource group (CPUSET), or -1.
- `memory_quota`: The memory limit value specified for the resource group.
- `min_cost`: The minimum cost of a query plan to be included in the resource group (MIN_COST).
- `io_limit`: The maximum read/write sequential disk I/O throughput, and the maximum read/write I/O operations per second for the queries assigned to a specific tablespace (shown as the tablespace oid) and resource group (IO_LIMIT).

gp_resgroup_role

The `gp_resgroup_role` view allows administrators to see the resource group assigned to every role.

This view is accessible to all users. The columns of the view are described as follows:

- `rrroiname`: The name of the role.
- `rrrsgname`: The name of the resource group.

gp_resgroup_status

The `gp_resgroup_status` view allows administrators to see status and activity for a resource group. It shows how many queries are waiting to run and how many queries are currently active in the system for each resource group. The view also displays current memory and CPU usage for the resource group.

Note

Resource groups use the Linux control groups (cgroups) configured on the host systems. The cgroups are used to manage host system resources. When resource groups use cgroups that are as part of a nested set of cgroups, resource group limits are relative to the parent cgroup allotment. For information about nested cgroups and SynxDB resource group limits, see the *Use Resource Groups* document.

This view is accessible to all users. The columns of the view are described as follows:

- `groupid`: The ID of the resource group.
- `groupname`: The name of the resource group.
- `num_running`: The number of transactions currently running in the resource group.
- `num_queueing`: The number of currently queued transactions for the resource group.
- `num_queued`: The total number of queued transactions for the resource group since the SynxDB cluster was last started, excluding the `num_queueing`.
- `num_executed`: The total number of transactions run in the resource group since the SynxDB cluster was last started, excluding the `num_running`.
- `total_queue_duration`: The total time any transaction was queued since the SynxDB cluster was last started.

Sample output for the `gp_resgroup_status` view:

```
select * from gp_toolkit.gp_resgroup_status;

groupid | groupname | num_running | num_queueing | num_queued | num_executed |
(continues on next page)
```

(continued from previous page)

```
total_queue_duration
```

```
+-----+-----+-----+-----+
```

```
+-----+
```

```
(0 rows)
```

gp_resgroup_status_per_host

The `gp_resgroup_status_per_host` view displays the real-time CPU and memory usage (MBs) for each resource group on a per-host basis. The view also displays available and granted group fixed and shared memory for each resource group on a host.

The columns of the view are described as follows:

- `groupid`: The ID of the resource group.
- `groupname`: The name of the resource group.
- `hostname`: The hostname of the segment host.
- `cpu_usage`: The real-time CPU core usage by the resource group on a host. The value is the sum of the percentages (as a float value) of the CPU cores that are used by the resource group on the host.
- `memory_usage`: The real-time memory usage of the resource group on each database segment's host, in MB.

Sample output for the `gp_resgroup_status_per_host` view:

```
select * from gp_toolkit.gp_resgroup_status_per_host;
```

```
groupid | groupname | hostname | cpu_usage | memory_usage
```

```
+-----+-----+-----+-----+
```

```
(0 rows)
```

gp_resgroup_status_per_segment

The `gp_toolkit.gp_resgroup_status_per_segment` view shows memory usage by resource group on each segment. The statistics are reported by the vmem tracker and measured in MB. This view helps database administrators monitor memory consumption of resource groups across segments.

 **Note**

This view is available only when resource group-based resource management is enabled.

Column descriptions:

- `groupid` (type: `oid`): Unique identifier of the resource group, corresponding to `pg_resgroup.oid`.
- `groupname` (type: `name`): Name of the resource group, corresponding to `pg_resgroup.rsgname`.
- `segment_id` (type: `smallint`): Content ID of the corresponding segment instance, taken from `gp_segment_configuration.content`.
- `vmem_usage` (type: `smallint`): Real-time memory usage (in MB) of the resource group on the corresponding segment.

Check resource queue activity and status

 **Note**

The resource queue activity and status views described in this section are valid only when resource queue-based resource management is active.

The purpose of resource queues is to limit the number of active queries in the system at any given time in order to avoid exhausting system resources such as memory, CPU, and disk I/O. All database users are assigned to a resource queue, and every statement submitted by a user is first evaluated against the resource queue limits before it can run. The `gp_resq_*` family of

views can be used to check the status of statements currently submitted to the system through their respective resource queue. Note that statements issued by superusers are exempt from resource queuing.

- *gp_resq_activity*
- *gp_resq_activity_by_queue*
- *gp_resq_priority_statement*
- *gp_resq_role*
- *gp_resqueue_status*

gp_resq_activity

For the resource queues that have active workload, this view shows one row for each active statement submitted through a resource queue. This view is accessible to all users.

The columns of the view are described as follows:

- **resqprocpid**: Process ID assigned to this statement (on the coordinator).
- **resqrole**: User name.
- **resqid**: Resource queue object id.
- **resqname**: Resource queue name.
- **resqstart**: Time statement was issued to the system.
- **resqstatus**: Status of statement: running, waiting or cancelled.

gp_resq_activity_by_queue

For the resource queues that have active workload, this view shows a summary of queue activity. This view is accessible to all users.

The columns of the view are described as follows:

- **resqid**: Resource queue object id.

- `resqname`: Resource queue name.
- `resqlast`: Time of the last statement issued to the queue.
- `resqstatus`: Status of last statement: running, waiting or cancelled.
- `resqtotal`: Total statements in this queue.

gp_resq_priority_statement

This view shows the resource queue priority, session ID, and other information for all statements currently running in the SynxDB system. This view is accessible to all users.

The columns of the view are described as follows:

- `rqpdatname`: The database name that the session is connected to.
- `rqpusename`: The user who issued the statement.
- `rqpsession`: The session ID.
- `rqpcommand`: The number of the statement within this session (the command id and session id uniquely identify a statement).
- `rqppriority`: The resource queue priority for this statement (MAX, HIGH, MEDIUM, LOW).
- `rqpweight`: An integer value associated with the priority of this statement.
- `rqpquery`: The query text of the statement.

gp_resq_role

This view shows the resource queues associated with a role. This view is accessible to all users.

The columns of the view are described as follows:

- `rrrolname`: Role (user) name.
- `rrrsqname`: The resource queue name assigned to this role. If a role has not been explicitly assigned to a resource queue, it will be in the default resource queue (*pg_default*).

gp_resqueue_status

This view allows administrators to see status and activity for a resource queue. It shows how many queries are waiting to run and how many queries are currently active in the system from a particular resource queue.

The columns of the view are described as follows:

- **queueid**: The ID of the resource queue.
- **rsgname**: The name of the resource queue.
- **rsqcountlimit**: The active query threshold of the resource queue. A value of -1 means no limit.
- **rsqcountvalue**: The number of active query slots currently being used in the resource queue.
- **rsqcostlimit**: The query cost threshold of the resource queue. A value of -1 means no limit.
- **rsqcostvalue**: The total cost of all statements currently in the resource queue.
- **rsqmemorylimit**: The memory limit for the resource queue.
- **rsqmemoryvalue**: The total memory used by all statements currently in the resource queue.
- **rsqwaiters**: The number of statements currently waiting in the resource queue.
- **rsqholders**: The number of statements currently running on the system from this resource queue.

Check query disk spill space usage

The *gp_workfile_* views show information about all the queries that are currently using disk spill space. SynxDB creates work files on disk if it does not have sufficient memory to run the query in memory. This information can be used for troubleshooting and tuning queries. The information in the views can also be used to specify the values for the SynxDB configuration parameters *gp_workfile_limit_per_query* and *gp_workfile_limit_per_segment*.

- *gp_workfile_entries*
- *gp_workfile_usage_per_query*
- *gp_workfile_usage_per_segment*

gp_workfile_entries

This view contains one row for each operator using disk space for workfiles on a segment at the current time. The view is accessible to all users, however non-superusers only to see information for the databases that they have permission to access.

The columns of the view are described as follows:

Column	Type	Description
datname	name	Database name.
pid	integer	Process ID of the server process.
sess_id	integer	Session ID.
command_cnt	integer	Command ID of the query.
username	name	Role name.
query	text	Current query that the process is running.
segid	integer	Segment ID.
slice	integer	The query plan slice. The portion of the query plan that is being run.
optype	text	The query operator type that created the work file.
size	bigint	The size of the work file in bytes.
numfiles	integer	The number of files created.
prefix	text	Prefix used when naming a related set of workfiles.

gp_workfile_usage_per_query

This view contains one row for each query using disk space for workfiles on a segment at the current time. The view is accessible to all users, however non-superusers only to see information for the databases that they have permission to access.

The columns of the view are described as follows:

Column	Type	Description
datname	name	Database name.
pid	integer	Process ID of the server process.
sess_id	integer	Session ID.
command_cnt	integer	Command ID of the query.
username	name	Role name.
query	text	Current query that the process is running.
segid	integer	Segment ID.
size	numeric	The size of the work file in bytes.
numfiles	bigint	The number of files created.

gp_workfile_usage_per_segment

This view contains one row for each segment. Each row displays the total amount of disk space used for workfiles on the segment at the current time. The view is accessible to all users, however non-superusers only to see information for the databases that they have permission to access.

The columns of the view are described as follows:

Column	Type	Description
segid	smallint	Segment ID.
size	numeric	The total size of the work files on a segment.
numfiles	bigint	The number of files created.

View users and groups (roles)

It is frequently convenient to group users (roles) together to ease management of object privileges: that way, privileges can be granted to, or revoked from, a group as a whole. In SynxDB, this is done by creating a role that represents the group, and then granting membership in the group role to individual user roles.

The *gp_roles_assigned* view can be used to see all of the roles in the system, and their assigned members (if the role is also a group role).

gp_roles_assigned

This view shows all of the roles in the system, and their assigned members (if the role is also a group role). This view is accessible to all users.

The columns of the view are described as follows:

- `raroleid`: The role object ID. If this role has members (users), it is considered a *group* role.
- `rarolename`: The role (user or group) name.
- `ramemberid`: The role object ID of the role that is a member of this role.
- `ramembername`: Name of the role that is a member of this role.

Check database object sizes and disk space

The `gp_size_*` family of views can be used to determine the disk space usage for a distributed SynxDB database, schema, table, or index. The following views calculate the total size of an object across all primary segments (mirrors are not included in the size calculations).

- `gp_size_of_all_table_indexes`
- `gp_size_of_database`
- `gp_size_of_index`
- `gp_size_of_schema_disk`
- `gp_size_of_table_and_indexes_disk`
- `gp_size_of_table_and_indexes_licensing`
- `gp_size_of_table_disk`
- `gp_size_of_table_uncompressed`
- `gp_disk_free`

The table and index sizing views list the relation by object ID (not by name). To check the size of a table or index by name, you must look up the relation name (`relname`) in the `pg_class` table.

For example:

```
SELECT relname as name, sotdsize as size, sotdtoastsize as
toast, sotdadditionalsize as other
FROM gp_size_of_table_disk as sotd, pg_class
WHERE sotd.sotdoid=pg_class.oid ORDER BY relname;
```

gp_size_of_all_table_indexes

This view shows the total size of all indexes for a table. This view is accessible to all users, however non-superusers will only be able to see relations that they have permission to access.

The columns of the view are described as follows:

- soatiod: The object ID of the table.
- soatisize: The total size of all table indexes in bytes.
- soatischemaname: The schema name.
- soatitablename: The table name.

gp_size_of_database

This view shows the total size of a database. This view is accessible to all users, however non-superusers will only be able to see databases that they have permission to access.

The columns of the view are described as follows:

- soddatname: The name of the database.
- soddatsize: The size of the database in bytes.

gp_size_of_index

This view shows the total size of an index. This view is accessible to all users, however non-superusers will only be able to see relations that they have permission to access.

The columns of the view are described as follows:

- soioid: The object ID of the index.
- soitableoid: The object ID of the table to which the index belongs.
- soisize: The size of the index in bytes.
- soiindexschemaname: The name of the index schema.
- soiindexname: The name of the index.
- soitableschemaname: The name of the table schema.
- soitablename: The name of the table.

gp_size_of_schema_disk

This view shows schema sizes for the public schema and the user-created schemas in the current database. This view is accessible to all users, however non-superusers will be able to see only the schemas that they have permission to access.

The columns of the view are described as follows:

- sosdnbsp: The name of the schema.
- sosdschematablesize: The total size of tables in the schema in bytes.
- sosdschemaidxsize: The total size of indexes in the schema in bytes.

gp_size_of_table_and_indexes_disk

This view shows the size on disk of tables and their indexes. This view is accessible to all users, however non-superusers will only be able to see relations that they have permission to access.

The columns of the view are described as follows:

- sotaidoid: The object ID of the parent table.
- sotaidtablesize: The disk size of the table.
- sotaididxsize: The total size of all indexes on the table.
- sotaidschemaname: The name of the schema.
- sotaidtablename: The name of the table.

gp_size_of_table_and_indexes_licensing

This view shows the total size of tables and their indexes for licensing purposes. The use of this view requires superuser permissions.

The columns of the view are described as follows:

- sotailoid: The object ID of the table.
- sotailtablesizedisk: The total disk size of the table.
- sotailtablesizeuncompressed: If the table is a compressed append-optimized table, shows the uncompressed table size in bytes.
- sotailindexessize: The total size of all indexes in the table.
- sotailschemaname: The schema name.
- sotailtablename: The table name.

gp_size_of_table_disk

This view shows the size of a table on disk. This view is accessible to all users, however non-superusers will only be able to see tables that they have permission to access. The columns of the view are described as follows:

- `sotdoid`: The object ID of the table.
- `sotdsizze`: The size of the table in bytes. The size is only the main table size. The size does not include auxiliary objects such as oversized (toast) attributes, or additional storage objects for AO tables.
- `sotdtoastsize`: The size of the TOAST table (oversized attribute storage), if there is one.
- `sotdadditionalsize`: Reflects the segment and block directory table sizes for append-optimized (AO) tables.
- `sotdschemaname`: The schema name.
- `sotdtablename`: The table name.

gp_size_of_table_uncompressed

This view shows the uncompressed table size for append-optimized (AO) tables. Otherwise, the table size on disk is shown. The use of this view requires superuser permissions. The columns of the view are described as follows:

- `sotuoid`: The object ID of the table.
- `sotusize`: The uncompressed size of the table in bytes if it is a compressed AO table. Otherwise, the table size on disk.
- `sotuschemaname`: The schema name.
- `sotutablename`: The table name.

gp_disk_free

This external table runs the `df` (disk free) command on the active segment hosts and reports back the results. Inactive mirrors are not included in the calculation. The use of this external table requires superuser permissions. The columns of the view are described as follows:

- `dfsegment`: The content id of the segment (only active segments are shown).
- `dfhostname`: The hostname of the segment host.
- `dfdevice`: The device name.
- `dfspace`: Free disk space in the segment file system in kilobytes.

Check for missing and orphaned data files

SynxDB considers a relation data file that is present in the catalog, but not on disk, to be missing. Conversely, when SynxDB encounters an unexpected data file on disk that is not referenced in any relation, it considers that file to be orphaned.

SynxDB provides the following views to help identify if missing or orphaned files exist in the current database:

- `gp_check_orphaned_files`
- `gp_check_missing_files`
- `gp_check_missing_files_ext`

Consider it a best practice to check for these conditions prior to expanding the cluster or before offline maintenance.

By default, the views identified in this section are available to PUBLIC.

gp_check_orphaned_files

The `gp_check_orphaned_files` view scans the default and user-defined tablespaces for orphaned data files. SynxDB considers normal data files, files with an underscore (_) in the name, and extended numbered files (files that contain a `.<N>` in the name) in this check. `gp_check_orphaned_files` gathers results from the SynxDB coordinator and all segments.

The columns of the view are described as follows:

- `gp_segment_id`: The database segment identifier.
- `tablespace`: The identifier of the tablespace in which the orphaned file resides.
- `filename`: The file name of the orphaned data file.
- `filepath`: The file system path of the orphaned data file, relative to the data directory of the coordinator or segment.

Caution

Use this view as one of many data points to identify orphaned data files. Do not delete files based solely on results from querying this view.

gp_check_missing_files

The `gp_check_missing_files` view scans heap and append-optimized, column-oriented tables for missing data files. SynxDB considers only normal data files (files that do not contain a `.` or an `_` in the name) in this check. `gp_check_missing_files` gathers results from the SynxDB coordinator and all segments.

The columns of the view are described as follows:

- `gp_segment_id`: The database segment identifier.
- `tablespace`: The identifier of the tablespace in which the table resides.
- `relname`: The name of the table that has a missing data file(s).
- `filename`: The file name of the missing data file.

gp_check_missing_files_ext

The `gp_check_missing_files_ext` view scans only append-optimized, column-oriented tables for missing extended data files. SynxDB considers both normal data files and extended numbered files (files that contain a .<N> in the name) in this check. Files that contain an _ in the name are not considered. `gp_check_missing_files_ext` gathers results from the SynxDB database segments only.

The columns of the view are described as follows:

- `gp_segment_id`: The database segment identifier.
- `tablespace`: The identifier of the tablespace in which the table resides.
- `relname`: The name of the table that has a missing extended data file(s).
- `filename`: The file name of the missing extended data file.

Move orphaned data files

The `gp_move_orphaned_files()` user-defined function (UDF) moves orphaned files found by the `gp_check_orphaned_files` view into a file system location that you specify.

The function signature is: `gp_move_orphaned_files(<target_directory> TEXT)`.

`<target_directory>` must exist on all segment hosts before you move the files, and the specified directory must be accessible by the `gpadmin` user. If you specify a relative path for `<target_directory>`, it is considered relative to the data directory of the coordinator or segment.

SynxDB renames each moved data file to one that reflects the original location of the file in the data directory. The file name format differs depending on the tablespace in which the orphaned file resides:

Tablespace	Renamed file format
default	<code>seg<num>_base_<database-oid>_<relfilenode></code>
global	<code>seg<num>_global_<relfilenode></code>
user-defined	<code>seg<num>_pg_tblspc_<tablespace-oid>_<gpdb-version>_<database-oid>_<relfilenode></code>

For example, if a file named 12345 in the default tablespace is orphaned on primary segment 2,

```
SELECT * FROM gp_move_orphaned_files('/home/gpadmin/orphaned');
```

moves and renames the file as follows:

- original location: <data_directory>/base/13700/12345
- new location and file name: /home/gpadmin/orphaned/seg2_base_13700_12345

`gp_move_orphaned_files()` returns both the original and the new file system locations for each file that it moves, and also provides an indication of the success or failure of the move operation. For example:

gp_segment_id	move_success	oldpath	newpath
-1	t	/<data_dir>/base/13715/99999	/home/gpadmin/orphaned/seg-1_base_13715_99999
1	t	/<data_dir>/base/13715/99999	/home/gpadmin/orphaned/seg1_base_13715_99999
2	t	/<data_dir>/base/13715/99999	/home/gpadmin/orphaned/seg2_base_13715_99999

(3 rows)

Once you move the files, you may choose to remove them or to back them up.

Check for uneven data distribution

All tables in SynxDB are distributed, meaning their data is divided across all of the segments in the system. If the data is not distributed evenly, then query processing performance may decrease. The following views can help diagnose if a table has uneven data distribution:

- `gp_skew_coefficients`
- `gp_skew_idle_fractions`

gp_skew_coefficients

This view shows data distribution skew by calculating the coefficient of variation (CV) for the data stored on each segment. This view is accessible to all users, however non-superusers will only be able to see tables that they have permission to access.

The columns of the view are described as follows:

- `skcoid`: The object id of the table.
- `skcnamespace`: The namespace where the table is defined.
- `skcrelname`: The table name.
- `skccoeff`: The coefficient of variation (CV) is calculated as the standard deviation divided by the average. It takes into account both the average and variability around the average of a data series. The lower the value, the better. Higher values indicate greater data skew.

gp_skew_idle_fractions

This view shows data distribution skew by calculating the percentage of the system that is idle during a table scan, which is an indicator of processing data skew. This view is accessible to all users, however non-superusers will only be able to see tables that they have permission to access.

The columns of the view are described as follows:

- `sifoid`: The object id of the table.
- `sifnamespace`: The namespace where the table is defined.
- `sifrelname`: The table name.
- `siffraction`: The percentage of the system that is idle during a table scan, which is an indicator of uneven data distribution or query processing skew. For example, a value of `0.1` indicates 10% skew, a value of `0.5` indicates 50% skew, and so on. Tables that have more than 10% skew should have their distribution policies evaluated.

Maintain partitions

If your database employs partitions you will need to perform certain tasks regularly to help maintain those partitions. SynxDB includes a view and a number of user-defined functions to help with these tasks.

The gp_partitions view

The `gp_partitions` view shows all leaf partitions in a database.

This view provides backwards compatibility with the legacy `pg_partitions` view (available in earlier major versions of SynxDB). The columns of the view are described as follows:

Column	Type	Description
<code>schemaname</code>	<code>name</code>	The name of the schema the partitioned table is in.
<code>tablename</code>	<code>name</code>	The name of the top-level parent table.
<code>partitionschemaname</code>	<code>name</code>	The schema of the partition table.
<code>partitiontablename</code>	<code>name</code>	The relation name of the partitioned table (this is the table name to use if accessing the partition directly).
<code>parentpartitiontablename</code>	<code>regclass</code>	The relation name of the parent table one level up from this partition.
<code>partitiontype</code>	<code>text</code>	The type of partition (range or list).
<code>partitionlevel</code>	<code>integer</code>	The level of this partition in the hierarchy.
<code>partitionrank</code>	<code>integer</code>	For range partitions, the rank of the partition compared to other partitions of the same level.
<code>partitionlistvalues</code>	<code>text</code>	For list partitions, the list value(s) associated with this partition.
<code>partitionrangestart</code>	<code>text</code>	For range partitions, the start value of this partition.
<code>partitionrangeend</code>	<code>text</code>	For range partitions, the end value of this partition.
<code>partitionisdefault</code>	<code>boolean</code>	T if this is a default partition, otherwise F.
<code>partitionboundary</code>	<code>text</code>	The entire partition specification for this partition.
<code>parenttablespace</code>	<code>name</code>	The tablespace of the parent table one level up from this partition.
<code>partitiontablespace</code>	<code>name</code>	The tablespace of this partition.

User-defined functions for partition maintenance

The following table summarizes the functions SynxDB provides to help you maintain partitions:

Function	Return type	Description
<code>pg_partition_rank(: integer regclass)</code>		For range partitions, returns the rank of the partition compared to other partitions of the same level. For other partition types, it returns <code>NULL</code> .
<code>pg_partition_range_(: text regclass)</code>		Returns the lower bound of a range partition.
<code>pg_partition_range_(: text regclass)</code>		Returns the upper bound of a range partition.
<code>pg_partition_isdef(: boolean regclass)</code>		Evaluates whether a given partition is a default partition.
<code>pg_partition_lowes(: regclass regclass)</code>		Finds the lowest ranked child of given partition.
<code>pg_partition_higher(: regclass regclass)</code>		Finds the highest ranked child of given partition.

8.3 Configuration Parameters

This document lists the configuration parameters (GUC) of SynxDB database in alphabetical order.

`autovacuum_freeze_max_age`

- Variable Type: Integer
- Default Value: 200000000
- Value Range: [100000,2000000000]
- Setting Category: postmaster
- Description: Sets the “maximum age” of transaction IDs in a table. When the number of transactions accumulated since the transaction ID was allocated reaches this value, the system automatically performs autovacuum on the table to prevent transaction ID wraparound. This operation is enforced even if autovacuum is disabled to ensure data safety.

`autovacuum_vacuum_cost_delay`

- Variable Type: Real
- Default Value: 2
- Unit: ms
- Value Range: [-1,100]
- Setting Category: sighup
- Description: Sets the vacuum cost delay time (in milliseconds) for autovacuum operations.

autovacuum_vacuum_scale_factor

- Variable Type: Real
- Default Value: 0.2
- Value Range: [0,100]
- Setting Category: sighup
- Description: Controls the threshold ratio of updated or deleted tuples to total tuples before autovacuum is performed.

autovacuum_vacuum_threshold

- Variable Type: Integer
- Default Value: 50
- Value Range: [0,2147483647]
- Setting Category: sighup
- Description: Controls the minimum number of updated or deleted tuples required to trigger autovacuum.

checkpoint_timeout

- Value Range: 30 - 86400 (integer, in seconds)
- Default Value: 300 (5 minutes)
- Setting Category: local, system, reload
- Description: Specifies the maximum time interval between automatic WAL checkpoints.

If no unit is specified when setting this parameter, the system defaults to seconds. The allowed range is 30 seconds to 1 day. The default value is 5 minutes (300 seconds or 5min). Increasing this parameter's value will increase the time required for crash recovery.

gp_appendonly_compaction_segfile_limit

- Variable Type: Integer
- Default Value: 10
- Value Range: [0,127]
- Setting Category: user
- Description: Sets the minimum number of append-only segfiles that must be reserved for insert operations.

gp_autostats_lock_wait

- Variable Type: Bool
- Default Value: off
- Setting Category: user
- Description: Controls whether autostats automatically generated ANALYZE waits for lock acquisition.

gp_command_count

- Variable Type: Integer
- Default Value: 0
- Value Range: [0,2147483647]
- Setting Category: internal
- Description: Displays the number of commands sent by the client in the current session.

gp_dynamic_partition_pruning

- Parameter Type: Boolean
- Default Value: on
- Setting Category: coordinator, session, reload
- Description: Enables execution plans that can dynamically eliminate partition scans.

gp_enable_runtime_filter_pushdown

- Value Range: Boolean
- Default Value: off
- Setting Category: user
- Description: Attempts to push down the hash table of hash join as a bloom filter to sequential scan or access methods (AM).

gp_enable_statement_trigger

- Variable Type: Bool
- Default Value: off
- Setting Category: user
- Description: Allows creation of statement-level triggers.

gp_max_partition_level

- Variable Type: Integer
- Default Value: 0
- Value Range: [0,2147483647]
- Setting Category: superuser

- Description: Sets the maximum allowed partition level when creating partitioned tables using classic syntax.

gp_resource_manager

- Value Range: none, group, group-v2, queue
- Default Value: none
- Setting Category: local, system, restart
- Description: Specifies the resource management scheme currently enabled in the SynxDB database cluster.
 - none: No resource manager is used (default).
 - group: Uses resource groups and enables resource group behavior based on Linux cgroup v1 functionality.
 - group-v2: Uses resource groups and enables resource group behavior based on Linux cgroup v2 functionality.
 - queue: Uses resource queues for resource management.

gp_role

- Value Range: dispatch, execute, utility
- Default Value: Undefined (depends on process type)
- Setting Category: read only (automatically set in background)
- Description: This parameter is used to identify the role of the current server process.
 - The role of the Coordinator process is `dispatch`, indicating it is responsible for query dispatch. The role of the Segment process is `execute`, indicating it is responsible for executing query plans. `utility` is used for special maintenance or management sessions. This parameter is automatically set by the system in the background and is mainly used to identify different types of internal worker processes.

gp_session_id

- Variable Type: Integer
- Default Value: -1
- Value Range: [-2147483648,2147483647]
- Setting Category: backend
- Description: Used to uniquely identify a session in the SynxDB cluster.

krb_server_keyfile

- Variable Type: string
- Default Value: FILE:/workspace/dist/database/etc/postgresql krb5.keytab
- Setting Category: sighup
- Description: Sets the location of the Kerberos server key file.

log_checkpoints

- Value Range: Boolean
- Default Value: on
- Setting Category: local, system, reload
- Description: Writes checkpoint and restartpoint information to the server log. The log messages include statistics such as the number of buffers written and the time taken to write them.

max_connections

- Value Range: 10 - 262143
- Default Value: 250 on Coordinator, 750 on Segment
- Setting Category: local, system, restart
- Description: The maximum number of concurrent connections allowed by the database server.

In the SynxDB system, client connections enter only through the Coordinator instance. Segment instances should allow 3 to 10 times the number of connections as the Coordinator. When increasing this parameter's value, the value of `max_prepared_transactions` must be increased accordingly.

The larger this parameter value, the more shared memory SynxDB requires.

max_replication_slots

- Variable Type: Integer
- Default Value: 10
- Value Range: [0,262143]
- Setting Category: postmaster
- Description: Sets the maximum number of replication slots that can be defined simultaneously.

optimizer_array_constraints

- Variable Type: Bool
- Default Value: on
- Setting Category: user
- Description: Allows the optimizer's constraint derivation framework to recognize array-type constraints.

optimizer_array_expansion_threshold

- Value Range: Integer greater than 0
- Default Value: 20
- Setting Category: coordinator, session, reload
- Description: When GPORCA is enabled (default) and executing queries containing constant array predicates, the `optimizer_array_expansion_threshold` parameter limits the optimization process based on the number of constants in the array.

If the number of array elements in the query predicate exceeds the value specified by this parameter, GPORCA will not convert the predicate to disjunctive normal form during query optimization, thereby reducing optimization time and memory consumption. For example, when GPORCA processes a query with an `IN` clause containing more than 20 elements, it will not convert the clause to disjunctive normal form for optimization performance. This behavioral difference can be observed in the execution plan from how the `IN` condition is filtered.

Modifying this parameter's value affects the trade-off between optimization time and memory usage, as well as optimization benefits from constraint derivation, such as conflict detection and partition pruning. This parameter can be set at the database system level, individual database level, or session and query level.

optimizer_cost_model

- Variable Type: Enum
- Default Value: calibrated
- Setting Category: user
- Description: Sets the cost model used by the optimizer.

optimizer_cost_threshold

- Variable Type: Real
- Default Value: 0
- Value Range: [0,2.15E+09]
- Setting Category: user
- Description: Sets the sampling threshold related to the optimal execution plan cost, where 0 means no upper limit.

optimizer_cte_inlining_bound

- Variable Type: Integer
- Default Value: 0
- Value Range: [0,2147483647]
- Setting Category: user
- Description: Sets the size boundary for the optimizer to decide whether to inline CTEs (Common Table Expressions).

optimizer_damping_factor_filter

- Variable Type: Real
- Default Value: 0.75
- Value Range: [0,1]
- Setting Category: user
- Description: Sets the damping factor used for selection predicates in the optimizer, where 1 . 0 means no damping.

optimizer_damping_factor_groupby

- Variable Type: Real
- Default Value: 0.75
- Value Range: [0,1]
- Setting Category: user
- Description: Sets the damping factor for group by operations in the optimizer, where 1.0 means no damping.

optimizer_damping_factor_join

- Variable Type: Real
- Default Value: 0
- Value Range: [0,1]
- Setting Category: user
- Description: Sets the damping factor for join predicates in the optimizer, where 1.0 means no damping and 0.0 means using square root damping.

optimizer_discard_redistribute_hashjoin

- Variable Type: Bool
- Default Value: off
- Setting Category: user
- Description: Controls whether the optimizer discards hash join plans containing redistribute operations.

optimizer_dpe_stats

- Variable Type: Bool
- Default Value: on
- Setting Category: user
- Description: Enables statistics derivation for partitioned tables in dynamic partition elimination scenarios.

optimizer_enable_derive_stats_all_groups

- Variable Type: Bool
- Default Value: off
- Setting Category: user
- Description: Enables statistics derivation for all groups after completing search space exploration.

optimizer_enable_dynamicbitmapscan

- Value Range: Boolean
- Default Value: on
- Setting Category: user
- Description: When enabled, the optimizer uses dynamic bitmap scan plans.

If this parameter is set to `off`, GPORCA will not generate dynamic bitmap scan plans and will fall back to using dynamic sequential scan as an alternative.

optimizer_enable_dynamicindexonlyscan

- Parameter Type: Boolean
- Default Value: on
- Setting Category: coordinator, session, reload
- Description: When GPORCA is enabled (default), the `optimizer_enable_dynamicindexonlyscan` parameter controls whether to generate dynamic index-only scan plans.

The default value is `on`. When planning queries on partitioned tables, if the query does not contain single-row volatile (SIRV) functions, GPORCA may generate dynamic index-only scans as an alternative. If set to `off`, GPORCA will not generate dynamic index-only scan plans. This parameter can be set at the database system level, individual database level, or session and query level.

optimizer_enable_dynamicindexscan

- Value Range: Boolean
- Default Value: on
- Setting Category: user
- Description: This parameter controls whether to enable dynamic index scan in query plans.

When enabled, the optimizer uses dynamic index scan plans. If this parameter is set to `off`, GPORCA will not generate dynamic index scan plans and will fall back to using dynamic sequential scan as an alternative.

optimizer_enable_foreign_table

- Parameter Type: Boolean
- Default Value: true
- Setting Category: coordinator, session, reload

- Description: When GPORCA is enabled (default) and this parameter is set to `true` (default), GPORCA generates execution plans for queries involving foreign tables.

If set to `false`, queries containing foreign tables will fall back to being planned by the PostgreSQL-based optimizer.

optimizer_enable_indexonlyscan

- Parameter Type: Boolean
- Default Value: `true`
- Setting Category: coordinator, session, reload
- Description: When GPORCA is enabled (default) and this parameter is set to `true` (default), GPORCA can generate index-only scan plans for B-tree indexes and any type of index that contains all columns in the query. (GiST indexes only support index-only scans for certain operator classes.)

GPORCA only accesses values in the index and not the actual data blocks of the table. This can improve query execution performance, especially when the table has been vacuumed, contains wide columns, and all visible columns are already in the index, eliminating the need to read any data blocks. If this parameter is set to `false`, GPORCA will not generate index-only scan plans. This parameter can be set at the database system level, individual database level, or session and query level.

optimizer_enable_orderedagg

- Parameter Type: Boolean
- Default Value: `on`
- Setting Category: coordinator, session, reload
- Description: When GPORCA is enabled (default), this parameter controls whether to generate query plans for ordered aggregates.

When set to `on` (default), GPORCA generates execution plans for queries containing ordered aggregates. When set to `off`, such queries will fall back to being planned by the PostgreSQL-based optimizer.

This parameter can be set at the database system level, individual database level, or session and query level.

optimizer_enable_push_join_below_union_all

- Parameter Type: Boolean
- Default Value: off
- Setting Category: coordinator, session, reload
- Description: When GPORCA is enabled (default), the `optimizer_enable_push_join_below_union_all` parameter controls GPORCA's behavior when encountering queries containing `JOIN UNION ALL`.

The default value is `off`. GPORCA will not perform any transformation when the query contains `JOIN UNION ALL`.

If set to `on` and the plan cost meets requirements, GPORCA will transform `JOIN UNION ALL` into multiple subqueries each performing `JOIN` followed by `UNION ALL`. This transformation may improve join performance when the subqueries of `UNION ALL` can benefit from join operations (but don't qualify in the original plan).

For example, in scenarios where indexed nested loop joins are highly efficient, such as when the inner table is large with an index and the outer table is small, or when multiple large tables with indexes are `UNION ALL`'ed with a small table, this transformation can push join conditions down as index conditions, potentially performing better than using hash joins.

Enabling this transformation may increase query planning time, so it's recommended to use `EXPLAIN` to analyze query execution plans with this parameter both enabled and disabled. This parameter can be set at the database system level, individual database level, or session and query level.

optimizer_enable_query_parameter

- Variable Type: Bool
- Default Value: on
- Setting Category: user
- Description: Allows the GPORCA optimizer to handle query parameters.

optimizer_enable_right_outer_join

- Parameter Type: Boolean
- Default Value: on
- Setting Category: coordinator, session, reload
- Description: When GPORCA is enabled (default), this parameter controls whether GPORCA generates right outer joins.

If set to the default value `on`, GPORCA can either directly generate right outer joins or convert left outer joins to right outer joins (when the optimizer deems it appropriate). If set to `off`, GPORCA will convert incoming right outer joins to equivalent left outer joins and completely avoid generating any right outer joins.

If you encounter performance issues with queries using right outer joins, you can disable right outer joins by setting this parameter to `off`.

This parameter can be set at the database system level, individual database level, or session and query level. However, it's more recommended to control it at the query level, as right outer joins may be more appropriate query plan choices in certain scenarios.

optimizer_force_three_stage_scalar_dqa

- Variable Type: Bool
- Default Value: on
- Setting Category: user

- Description: Forces the optimizer to always choose three-stage aggregation plans for scalar distinct qualified aggregates.

optimizer_nestloop_factor

- Variable Type: Real
- Default Value: 1024
- Value Range: [1,1.79769e+308]
- Setting Category: user
- Description: Sets the cost factor for nested loop joins in the optimizer.

optimizer_penalize_broadcast_threshold

- Parameter Type: Integer
- Value Range: [0, 2147483647]
- Default Value: 100000
- Setting Category: user
- Description: Specifies the maximum number of relation rows that can be broadcast without penalty.

If the number of broadcast rows exceeds this threshold, the optimizer will increase its execution cost to reduce the likelihood of selecting that plan.

Setting this parameter to 0 disables the penalty mechanism, meaning no penalty is applied to any broadcast.

optimizer_push_group_by_below_setop_threshold

- Variable Type: Integer
- Default Value: 10
- Value Range: [0,2147483647]
- Setting Category: user
- Description: Sets the maximum number of child nodes to attempt pushing GROUP BY operations below SetOp nodes.

optimizer_replicated_table_insert

- Variable Type: Bool
- Default Value: on
- Setting Category: user
- Description: Omits broadcast operations when inserting data into replicated tables.

optimizer_skew_factor

- Variable Type: Integer
- Default Value: 0
- Value Range: [0,100]
- Setting Category: user
- Description: Sets the source and weight of the skew factor. 0 means disabling skew derivation based on sample statistics, 1–100 means enabling and calculating skew ratio based on samples; the actual skew used for cost estimation is the product of this parameter and the skew ratio.

optimizer_sort_factor

- Variable Type: Real
- Default Value: 1
- Value Range: [0,1.79769e+308]
- Setting Category: user
- Description: Sets the cost factor for sort operations in the optimizer; 1 . 0 means the same as default cost, greater than 1 means higher sort cost, less than 1 means lower cost.

optimizer_trace_fallback

- Variable Type: Bool
- Default Value: off
- Setting Category: user
- Description: Prints information at the INFO log level when GPORCA falls back to using the traditional optimizer.

optimizer_use_gpdb_allocators

- Variable Type: Bool
- Default Value: on
- Setting Category: postmaster
- Description: Allows the GPORCA optimizer to use the database's memory context management mechanism (Memory Contexts).

optimizer_xform_bind_threshold

- Variable Type: Integer
- Default Value: 0
- Value Range: [0,2147483647]
- Setting Category: user
- Description: Limits the maximum number of bindings that can be generated for each transformation rule (xform) on each group expression. Setting to 0 means this limit is not enabled.

superuser_reserved_connections

- Value Range: Integer less than max_connections
- Default Value: 10
- Setting Category: local, system, restart
- Description: Specifies the number of connection slots reserved for SynxDB database superusers.

track_io_timing

- Parameter Type: Boolean
- Default Value: off
- Setting Category: superuser
- Description: Used to collect timing statistics for database I/O activities. When enabled, the system records I/O operation durations during statement execution, which is useful for performance analysis and bottleneck identification. This parameter is off by default and can only be set by superusers.

wal_compression

- Variable Type: Bool
- Default Value: on
- Setting Category: superuser
- Description: Enables compression of full page writes in WAL files.

wal_keep_size

- Parameter Type: integer
- Value Range: 0 - 2147483647 (in MB)
- Default Value: 320
- Setting Category: sighup
- Description: Specifies the maximum size of WAL files to retain for standby servers.

work_mem

- Value Range: Integer in kilobytes
- Default Value: 32MB
- Setting Category: coordinator, session, reload
- Description: Specifies the maximum amount of memory that can be used by each query operation (such as sort or hash table) before writing to temporary disk files. If no unit is specified for this parameter, it defaults to kilobytes. The default value is 32MB.

In complex queries, multiple sort or hash operations may be executed in parallel, each of which can use the amount of memory specified by `work_mem` before writing to temporary files. Additionally, multiple sessions may perform these operations simultaneously, so the total memory consumption may be much higher than the value of `work_mem` itself. This should be taken into special consideration when choosing the value for this parameter.

`work_mem` affects these operations: sort operations for `ORDER BY`, `DISTINCT`, and merge

joins; hash tables for hash joins, hash aggregates, and processing IN subqueries; bitmap index scans; and tuple store-based operations such as function scans, CTEs, PL/pgSQL, and management UDFs.

In addition to allocating memory for specific execution operators, `work_mem` also affects the PostgreSQL optimizer's preference for certain query plans. Note that `work_mem` is independent of the resource queue and resource group memory management mechanisms. It takes effect at the query level and is not affected by resource queue or resource group memory limits.

writable_external_table_bufsize

- Variable Type: Integer
- Default Value: 1024
- Unit: kB
- Value Range: [32,131072]
- Setting Category: user
- Description: Sets the buffer size (in kB) for writing to gpfdist before writing to writable external tables.

Chapter 9

Developer Guides

9.1 Develop Database Extensions Using PGRX

This document explains how to develop database extensions using Rust and the PGRX framework. PGRX is a Rust framework for developing extensions for SynxDB, offering a safe and efficient development experience.

For the core features of PGRX, see [PGRX core features](#). For notes of PGRX, see [Considerations and best practices for PGRX](#).

Requirements for development environment

- Make sure that your OS is one of Debian/Ubuntu and RHEL/CentOS.
- Make sure that your SynxDB cluster is compiled from source code, not installed using RPM package.

Basic software environment

Required software:

- Rust toolchain (`rustc`, `cargo`, and `rustfmt`) - install via <https://rustup.rs>
- Git
- libclang 11 or higher (for bindgen)
- GCC 7 or higher

PostgreSQL dependencies

Install required PostgreSQL dependencies for your OS:

For Debian/Ubuntu:

```
sudo apt-get install build-essential libreadline-dev zlib1g-dev flex bison \
    libxml2-dev libxslt-dev libssl-dev libxml2-utils xsltproc ccache pkg-
config
```

For RHEL/CentOS:

```
sudo yum install -y bison-devel readline-devel zlib-devel openssl-devel wget
ccache
sudo yum groupinstall -y 'Development Tools'
```

After installing the dependencies, you can start developing extensions.

Quick start for PGRX

This section introduces the process of quickly developing extensions using PGRX, including:

- Setting up and installing PGRX
- Creating extension
- Installing and using extension

Set up and install PGRX

1. Set the environment variable for SynxDB's pg_config path, where <pg_config_path> is the path in your SynxDB cluster (for example, /usr/local/cloudberry-db/bin/pg_config):

```
export PGRX_PG_CONFIG_PATH=<pg_config_path>
```

2. Build the PGRX framework:

1. Clone the SynxDB-compatible pgrx repository:

```
git clone https://github.com/cloudberry-contrib/pgrx
cd pgrx
```

2. Build the code with pg14 and cbdb features enabled:

```
cargo build --features "pg14, cbdb"
```

3. Install the SynxDB-compatible cargo-pgrx tool:

```
cargo install --path cargo-pgrx/
```

4. Initialize the environment with your database kernel version:

```
cargo pgrx init --pg14=`which pg_config`
```

Create an extension

1. Generate an extension template. This example creates an extension named my_extension:

```
cargo pgrx new my_extension
cd my_extension
```

The created directory structure is as follows:

```
.
├── Cargo.toml
└── my_extension.control
```

(continues on next page)

(continued from previous page)

```

└── sql
└── src
    └── bin
        └── pgrx_embed.rs
└── lib.rs

```

2. Modify dependencies in `Cargo.toml` to use local PGRX:

- Change `pgrx = "0.12.7"` under `[dependencies]` to point to the `pgrx` directory in your local PGRX repository. For example:

```

[dependencies]
pgrx = { path = "/home/gpadmin/pgrx/pgrx/", features = ["pg14", "cbdb"]
""] }

```

- Add `pgrx-pg-sys` under `[dependencies]` to point to the `pgrx-pg-sys` directory in your local PGRX repository. For example:

```

[dependencies]
pgrx-pg-sys = { path = "/home/gpadmin/pgrx/pgrx-pg-sys/", features = [
"pg14", "cbdb"] }

```

- Change `pgrx-tests = "0.12.7"` under `[dev-dependencies]` to point to the `pgrx-tests` directory in your local PGRX repository:

```

[dev-dependencies]
pgrx-tests = { path = "/home/gpadmin/pgrx/pgrx-tests/" }

```

3. Append the extension name `my_extension` to the `workspace.members` array of the `Cargo.toml` file in the root directory of your local PGRX repository. For example:

```
vi /home/gpadmin/pgrx/Cargo.toml
```

```

[workspace]
resolver = "2"
members = [
    "cargo-pgrx",
    "pgrx",

```

(continues on next page)

(continued from previous page)

```
"pgrx-macros",
"pgrx-pg-config",
"pgrx-pg-sys",
"pgrx-sql-entity-graph",
"pgrx-tests",
"pgrx-bindgen",
"my_extension"
]
```

- Grant the current system user the permissions to the SynxDB directory. For example, if the current user is `gpadmin` and SynxDB directory is `/usr/local/cloudberrydb`:

```
sudo chown -R gpadmin:gpadmin /usr/local/cloudberrydb
```

Install and use the extension

- Install the extension:

```
cargo pgrx install
```

- To use the extension in the database, connect to the database and execute the following statements:

```
CREATE EXTENSION my_extension;

-- Tests example function
SELECT hello_my_extension();
```

PGRX type mapping

The table below lists the complete mapping of SynxDB (PostgreSQL) data types to Rust types:

Database data type	Rust type (Option<T>)
<code>bytea</code>	<code>Vec<u8></code> or <code>&[u8]</code> (zero-copy)
<code>text</code>	<code>String</code> or <code>&str</code> (zero-copy)
<code>varchar</code>	<code>String</code> or <code>&str</code> (zero-copy) or <code>char</code>

continues on next page

Table 1 – continued from previous page

Database data type	Rust type (<code>Option<T></code>)
"char"	<code>i8</code>
smallint	<code>i16</code>
integer	<code>i32</code>
bigint	<code>i64</code>
oid	<code>u32</code>
real	<code>f32</code>
double precision	<code>f64</code>
bool	<code>bool</code>
json	<code>pgrx::Json(serde_json::Value)</code>
jsonb	<code>pgrx::JsonB(serde_json::Value)</code>
date	<code>pgrx::Date</code>
time	<code>pgrx::Time</code>
timestamp	<code>pgrx::Timestamp</code>
time with time zone	<code>pgrx::TimeWithTimeZone</code>
timestamp with time zone	<code>pgrx::TimestampWithTimeZone</code>
anyarray	<code>pgrx::AnyArray</code>
anyelement	<code>pgrx::AnyElement</code>
box	<code>pgrx::pg_sys::BOX</code>
point	<code>pgrx::pg_sys::Point</code>
tid	<code>pgrx::pg_sys::ItemPointerData</code>
cstring	<code>&core::ffi::CStr</code>
numeric	<code>pgrx::Numeric<P, S> or pgrx::AnyNumeric</code>
void	<code>()</code>
<code>ARRAY[]::<type></code>	<code>Vec<Option<T>> or pgrx::Array<T> (zero-copy)</code>
int4range	<code>pgrx::Range<i32></code>
int8range	<code>pgrx::Range<i64></code>
numrange	<code>pgrx::Range<Numeric<P, S>> or pgrx::Range<AnyRange></code>
daterange	<code>pgrx::Range<pgrx::Date></code>
tsrange	<code>pgrx::Range<pgrx::Timestamp></code>
tstzrange	<code>pgrx::Range<pgrx::TimestampWithTimeZone></code>
NULL	<code>Option::None</code>
internal	<code>pgrx::PgBox<T> (where T can be any Rust/Postgres struct)</code>
uuid	<code>pgrx::Uuid([u8; 16])</code>

Custom type conversions

You can implement additional type conversions in the following ways:

- Implement `IntoDatum` and `FromDatum` traits.
- Use `# [derive(PostgresType)]` and `# [derive(PostgresEnum)]` for automatic type conversions.

Type mapping details

PGRX converts `text` and `varchar` to `&str` or `String`, and verifies whether the encoding is UTF-8. If an encoding other than UTF-8 is detected, PGRX triggers a panic to alert the developer. Because UTF-8 validation might affect performance, it is not recommended to rely on UTF-8 validation.

The default encoding for PostgreSQL servers is `SQL_ASCII`, which guarantees neither ASCII nor UTF-8 (SynxDB will accept but ignore non-ASCII bytes). For best results, always use UTF-8 encoding with PGRX and explicitly set the database encoding when creating the database.

PGRX core features

Complete management for development environment

`cargo-pgrx` provides a complete set of command-line tools:

- `cargo pgrx new`: Quickly creates a new extension.
- `cargo pgrx init`: Installs or registers a SynxDB (PostgreSQL) instance.
- `cargo pgrx run`: Interactively tests the extension in `psql` (or `pgcli`).
- `cargo pgrx test`: Performs unit tests across multiple SynxDB (PostgreSQL) versions.
- `cargo pgrx package`: Creates an extension installation package.

Automatic mode generation

- Fully implements the extension using Rust.
- Automatically maps various Rust types to SynxDB (PostgreSQL) types.
- Automatically generates SQL schema (can also be manually generated using `cargo pgrx schema`).
- Uses `extension_sql!` and `extension_sql_file!` to include custom SQL.

Security first

- Converts Rust's `panic!` to SynxDB/PostgreSQL's `ERROR` (abort the transaction, not the process).
- Memory management follows Rust's `DROP` semantics, including handling `panic!` and `elog(ERROR)` cases.
- Uses `# [pg_guard]` procedural macro to ensure safety.
- SynxDB's `Datum` is represented as `Option<T>` where `T: FromDatum`, with `NULL` values safely represented as `Option::<T>::None`.

UDF supports

- Uses `# [pg_extern]` annotation to expose functions to SynxDB.
- Returns `pgrx::iter::SetOfIterator<'a, T>` to implement `RETURNS SETOF`.
- Returns `pgrx::iter::TableIterator<'a, T>` to implement `RETURNS TABLE (...)`.
- Uses `# [pg_trigger]` to create trigger functions.

Simple custom types

- Uses `# [derive(PostgresType)]` to treat Rust structs as SynxDB types.
 - By default, CBOR encoding is used for storage, and JSON is used as a human-readable format.
 - Supports custom memory/disk/readable formats.
- Uses `# [derive(PostgresEnum)]` to treat Rust enums as SynxDB (PostgreSQL) enums.
- Supports composite types via `pgrx::composite_type! ("Sample")` macro.

Server programming interface (SPI)

- Secure access to SPI.
- Transparently returns ownership of Datum from SPI context.

Advanced features

- Securely accesses SynxDB's memory context system via `pgrx::PgMemoryContexts`.
- Supports executor/planner/transaction/subtransaction hooks.
- Securely handles SynxDB pointers using `pgrx::PgBox<T>` (similar to `alloc::boxed::Box<T>`).
- Protects Rust functions passed to SynxDB's `extern "C"` using `# [pg_guard]` procedural macro.
- Accesses SynxDB's logging system via the `eprintln!` macro.
- Directly (unsafe) accesses SynxDB internals via the `pgrx::pg_sys` module.

Considerations and best practices for PGRX

Thread supports:

- SynxDB strictly follows a single-threaded model.
- Custom threads cannot call internal database functions.
- The interaction method for asynchronous contexts is still under exploration.

Encoding requirements:

- It is recommended to use UTF-8 encoding.
- The default server encoding is SQL_ASCII.
- It is recommended to explicitly set the encoding when creating the database.

Debugging and development tips

- Uses `cargo pgrx test` for unit testing.
- Uses `# [pg_guard]` to ensure memory safety.
- For custom types, uses appropriate serialization methods.

Learning resources for PGRX

The following resources can help you gain a deeper understanding of PGRX:

- Learn about all available `cargo-pgrx` subcommands and options: [cargo-pgrx command details](#)
- Learn how to define and use custom data types: [custom type examples](#)
- Explore how to implement custom operators: [operator functions and operator classes/families](#)
- Learn how to use shared memory: [shared memory support](#)
- Browse example code implementations: [more example code](#)