GaussDB

Best Practices

Issue	01
Date	2025-09-12

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Huawei Cloud Computing Technologies Co., Ltd.

Address:	Huawei Cloud Data Center Jiaoxinggong Road Qianzhong Avenue Gui'an New District Gui Zhou 550029 People's Republic of China
Website:	https://www.huaweicloud.com/intl/en-us/

Contents

1 Best Practices Overview

This section describes best practices for working with GaussDB and provides operational guidelines that you can follow when using this service.

Table 1-1 GaussDB best practices

Document	Description
Suggestions on GaussDB Security Configuration	Provides guidance on GaussDB security configurations.
Best Practices for Scaling	Describes the use cases of different scaling operations.
Best Practices for Backup and Restoration	Describes typical accidental operations and their corresponding recovery methods.
Suggestions on GaussDB Metric Alarm Configuration	Describes how to configure GaussDB metric alarm rules.
Best Practices for Row Compression	Describes how to use manual and automatic scheduling to compress data.
Best Practices for SQL Queries	Describes how to adjust SQL statements to improve SQL execution efficiency.
Best Practices for Permission Configuration	Describes the responsibilities and capabilities of different permission types and roles, as well as how to configure permissions.
Best Practices for Data Skew Query (Distributed Instances)	Describes how to identify tables that cause data skew.
Best Practices for Stored Procedures	Describes key aspects of stored procedures, including permission management, naming conventions, access to database objects, statement functions, and transaction management.

Document	Description
Best Practices for Import and Export Using COPY	Describes how to use the COPY syntax to import and export data.
Best Practices for Import and Export Using Tools	Describes how to use gs_dump and gs_dumpall to import and export data.
Best Practices for JDBC	Describes how to use JDBC for efficient data operations, including inserting data in batches, executing streaming queries, handling user-defined data types, and performing batch queries.
Best Practices for ODBC	Describes how to use the ODBC driver for batch data insertion.
Best Practices for Go	Describes how to use the Go driver for batch data insertion.
Best Practices for Index Design	Compares the performance of querying large tables containing over a million rows both with indexes and without them, and evaluates the effectiveness of single-column indexes versus composite indexes for optimizing query efficiency.
Best Practices for Table Design	Describes distribution mode and key design, data type selection, partitioning policies, constraint configuration, index optimization, and tuning of storage parameters.

2 Suggestions on GaussDB Security Configuration

Security is a responsibility shared between Huawei Cloud and yourself. Huawei Cloud ensures the security of cloud services for a secure cloud. As a tenant, you should utilize the security capabilities provided by cloud services to protect data and use the cloud securely. For details, see **Shared Responsibilities**.

This section provides actionable guidance for enhancing the overall security of GaussDB. You can continuously evaluate the security of your GaussDB instances and enhance their overall defensive capabilities by combining different security capabilities provided by GaussDB. By doing this, data stored in GaussDB can be protected from leakage and tampering both at rest and in transit.

You can make security configurations from the following dimensions to match your workloads.

- **Maximum Number of Connections**
- **Security Authentication**
- **Client Authentication Configuration**
- **User Password Security**
- **Permissions Management**
- **Database Audit**
- **WAL Archiving**
- **Backup Management**

Maximum Number of Connections

Excessive GaussDB connections can consume excessive server resources, leading to sluggish operation responses. You can adjust the maximum allowed connections using the **max_connections** parameter. For details, see **Connection Settings**.

max_connections: the maximum number of concurrent connections to the database. This parameter affects the concurrency capability of the cluster.

Security Authentication

To ensure user experience and prevent accounts from being cracked, you can configure the following parameters to limit the number of login attempts before an account is locked and how long it is locked for:

- **failed_login_attempts**: the maximum number of failed login attempts permitted
- **password_lock_time**: the number of days before a locked account is automatically unlocked

  If an account is identified as stolen or the account is used to access a database without proper authorization, administrators can manually lock the account. Administrators can manually unlock the account if the account becomes normal again.

  For example, run the following commands to manually lock and unlock the user, **joe**:

  – Manually lock the account.
  ```
  gaussdb=# ALTER USER joe ACCOUNT LOCK;
  ALTER ROLE
  ```
  – Manually unlock the account.
  ```
  gaussdb=# ALTER USER joe ACCOUNT UNLOCK;
  ALTER ROLE
  ```

Client Authentication Configuration

When a host needs to connect to a database remotely, its information must be added to the database system's configuration file, along with client authentication rules. To simplify the process, GaussDB automatically includes the following default rules in the client authentication configuration file when an instance is created:

- Default configurations for a centralized instance (assuming that the subnet CIDR block of the instance is 192.168.0.0)
  ```
  TYPE   DATABASE        USER    ADDRESS          METHOD
  host   all             all     0:0:0:0/0        sha256
  host   all             all     192.168.0.0/16   sha256
  host   replication     all     192.168.0.0/16   sha256
  host   replication     all     0:0:0:0/0        sha256
  ```
  – The first record allows all users from any IPv4 client to access all databases using the SHA-256 authentication method.

  – The second record allows all users from IPv4 clients within the current instance's subnet to access all databases using SHA-256.

  – The third record allows all users from IPv4 clients within the current instance's subnet to request a replication connection using SHA-256.

  – The fourth record allows all users from any IPv4 client to request a replication connection using SHA-256.
- Default configurations for a distributed instance

  This record allows all users from any IPv4 client to access all databases using the SHA-256 authentication method.
  ```
  TYPE   DATABASE        USER    ADDRESS          METHOD
  host   all             all     0:0:0:0/0        sha256
  ```

In most cases, the default configurations are sufficient for typical remote connection requirements. However, if you require more granular control over client access or the existing authentication settings do not meet your operational requirements, you can **customize custom client authentication configurations**.

For robust security, adjust the default configurations to fit your specific application requirements and adopt fine-grained authentication control to precisely manage client access.

User Password Security

GaussDB enhances user account security in the following ways:

- User passwords are stored in the system catalog **pg_authid**. To prevent password leakage, GaussDB encrypts user passwords before storing them. The cryptographic algorithm is determined by the configuration parameter **password_encryption_type**.
- All passwords in GaussDB must have a validity period. You can configure the **password_effect_time** parameter to set a validity period for each database user password.

Permissions Management

- A VPC provides an isolated virtual network for GaussDB instances. You can configure and manage the network as required. A subnet provides dedicated network resources that are logically isolated from other networks for security. If you need to assign different permissions (also known as privileges) to different employees in your enterprise to access your DB instance resources, IAM is a good choice. For details, see **Permissions Management**.
- To ensure database security and reliability, configure security groups before using a DB instance. For details, see **Configuring Security Group Rules**.
- Run the following SQL statement to check whether the **PUBLIC** role has the **CREATE** privilege in public schema. If yes, any user can create and modify tables or database objects in public schema.

  SELECT CAST(has_schema_privilege('public','public','CREATE') AS TEXT);

  - If **TRUE** is returned, run the following SQL statement to revoke the privilege:

    REVOKE CREATE ON SCHEMA public FROM PUBLIC;
- All users are assigned the **PUBLIC** role. If all privileges of an object are granted to the **PUBLIC** role, any user can inherit all the privileges of the object, which violates the principle of least privilege. For security reasons, this role should have only minimal privileges. Run the following SQL statement to check whether all privileges have been granted to the **PUBLIC** role:

  SELECT relname,relacl FROM pg_class WHERE (CAST(relacl AS TEXT) LIKE '%,=arwdDxt/%}' OR CAST(relacl AS TEXT) LIKE '{=arwdDxt/%}') AND (CAST(relacl AS TEXT) LIKE '%,=APmiv/%}' OR CAST(relacl AS TEXT) LIKE '{=APmiv/%}');

  - If the query returns an empty result set, all privileges have been granted. In this case, run the following SQL statement to revoke the privileges:

    REVOKE ALL ON <OBJECT_NAME> FROM PUBLIC;
- The **pg_authid** system catalog in the pg_catalog schema contains information about all roles in a database. To prevent sensitive information from being disclosed or modified, the **PUBLIC** role is not allowed to have any access to

this system catalog. Run the following SQL statement to check whether privileges on the **pg_authid** system catalog have been granted:

SELECT relname,relacl FROM pg_class WHERE relname = 'pg_authid' AND CAST(relacl AS TEXT) LIKE '%,=%}';

- If the returned result set is not empty, privileges have been granted. In this case, run the following SQL statement to revoke the privileges:

  REVOKE ALL ON pg_authid FROM PUBLIC;

- Regular users are non-administrator users who perform common service operations. Regular users should not have administrative privileges beyond their normal scope of responsibilities. For example, they should not have the privileges needed to create roles, create databases, audit, monitor, perform O&M operations, or manage security policies. To ensure the principle of least privilege is enforced for regular users, unnecessary administrative privileges should be revoked while meeting normal business requirements.
- The SECURITY DEFINER function is executed with the privileges of the creator. Improper use of SECURITY DEFINER may cause the function executor to perform unauthorized operations with the privileges of the creator. For this reason, ensure that this function is not misused. For security purposes, the **PUBLIC** role is not allowed to execute functions of the SECURITY DEFINER type. Run the following SQL statement to check whether the **PUBLIC** role has access to any SECURITY DEFINER functions:

  SELECT a.proname, b.nspname FROM pg_proc a, pg_namespace b where a.pronamespace=b.oid and b.nspname <> 'pg_catalog' and a.prosecdef='t';

  - If the returned result set is not empty, run the following SQL statement to check whether it has the **EXECUTE** privilege:

    SELECT CAST(has_function_privilege('public', 'function_name([arg_type][, ...])', 'EXECUTE') AS TEXT);

    - If **TRUE** is returned, the role has the privilege. In this case, run the following SQL statement to revoke the privilege:

      REVOKE EXECUTE ON FUNCTION function_name([arg_type][, ...]) FROM PUBLIC;

- The SECURITY INVOKER function is executed with the privileges of the invoker. Improper use of SECURITY INVOKER may cause the function creator to perform unauthorized operations with the privileges of the executor. Before invoking a function not created by yourself, check the function content to prevent the function creator from performing unauthorized operations with your privileges.

Database Audit

- GaussDB can record operations you perform on your DB instances. However, only operations supported by Cloud Trace Service (CTS) can be recorded. View the supported operations before performing operations. For details, see **Key Operations Supported by CTS**.
- Ensure that auditing is enabled for the creation, deletion, and modification of database objects. For details, see **Database Audit**.
- To view audit logs in a visualized manner, enable **Upload Audit Logs to LTS**. For details, see **Interconnecting with LTS and Querying Database Audit Logs**.

WAL Archiving

The Write Ahead Log (WAL) is another term for the transaction log, which is also referred to as the Xlog. It records changes made to the database before they are written to the main storage, ensuring data consistency and durability in case of failures. The **wal_level** parameter specifies the level of information to be written into a WAL. To enable read-only queries on a standby node, you need to set the **wal_level** parameter to **hot_standby** on the primary node and set **hot_standby** to **on** on the standby node.

Backup Management

GaussDB provides instance backup and restoration to ensure data reliability. Backups are stored in unencrypted form. To prevent data loss caused by misoperations or service exceptions, you can:

- Configure automated backups and create manual backups. For details, see **Working with Backups**. When you create a GaussDB instance, the instance-level automated backup policy is enabled by default. After your instance is created, you can modify the automated backup policy as needed.
- Configure an automated backup policy to periodically back up databases. For details, see **Configuring an Automated Backup Policy**.
- **Export backup information**.

3 Best Practices for Scaling

With GaussDB, you can scale your instance by changing CPU and memory specifications, increasing or decreasing storage, and adding or removing nodes and shards as needed. This allows for flexible adjustment of database performance and capacity to adapt to changing workload demands.

Changing CPU and Memory Specifications

To adjust to changing workloads, you can scale up or down an instance by changing its CPU and memory specifications. For details, see **Changing the CPU and Memory Specifications of a GaussDB Instance**.

Increasing or Decreasing Storage

You are advised to resize your instance storage in the following scenarios:

- As GaussDB instances continue to operate over time, the amount of data to be stored can grow rapidly, potentially surpassing the original storage capacity. At this point, you can scale up the storage of your DB instance.
- When the storage usage exceeds a certain threshold (85% by default, but this can be modified using the **cms:datastorage_threshold_value_check** parameter), the GaussDB instance is set to a read-only state and no more data can be written to it. You can avoid this situation by scaling up the instance storage to ensure service continuity.

For details, see **Scaling Up Storage Space**.

Adding or Deleting Coordinator Nodes

When the number of concurrent requests increases significantly, you can add more coordinator nodes (CNs) to increase the concurrent processing capacity of your instance. For details, see **Adding Coordinator Nodes for an Instance**.

On the other hand, if business activity decreases, some CNs are left idle. You can reduce the number of CNs to improve resource efficiency. For details, see **Deleting Coordinator Nodes for an Instance**.

CNs can only be added or deleted for distributed instances using independent deployment.

Adding or Deleting Shards

As data volume continues to grow, the existing data nodes (DNs) may become unable to accommodate the increased load. To address this, you can scale out the instance by adding more shards to it to distribute data. For details, see **Adding Shards for an Instance**.

Conversely, there may be more than enough DNs in your instance after read/write splitting is enabled or redundant data is cleared. You can delete shards as needed to avoid cost waste. For details, see **Deleting Shards for an Instance**.

Shards can only be added or deleted for distributed instances using independent deployment.

4 Best Practices for Backup and Restoration

4.1 Overview

GaussDB ensures high availability, but accidental or intentional deletion of a database or table will result in data loss across both primary and standby nodes, making it unrecoverable from the standby node. In this case, you can only restore the deleted data from backup. GaussDB enables data restoration from backup, either to the state it was in when the backup was created or to a specific point in time.

This section outlines typical accidental operations and their corresponding recovery methods. For details, see **Table 4-1**. It also presents typical use cases and performance specifications for backup and restoration. For details, see **Table 4-2**. You can choose different data restoration methods based on service requirements.

Restoration Methods for Misoperations

Table 4-1 Restoration methods for different misoperations

Scenario	Restoration Method	Restoration Scope	Instructions
An instance is deleted by mistake.	Locate the deleted instance in the recycle bin and rebuild it.	All databases and tables	**Restoring an Instance from the Recycle Bin**
	If a manual backup was created before the instance was deleted, restore the instance on the **Backups** page.	All databases and tables	**Restoring an Instance from a Backup**

Scenario	Restoration Method	Restoration Scope	Instructions
A table is deleted by mistake.	Use the database and table restoration method to restore the table.	• All databases and tables • Certain databases and tables	• **Restoring Databases or Tables to a Specific Point in Time** • **Restoring Databases or Tables Using a Backup**
A database is deleted by mistake.	Use the database and table restoration method to restore the database.	• All databases and tables • Certain databases and tables	
An entire table is overwritten, or the columns, rows, or data in a table is deleted or modified by mistake.	Use the database and table restoration method to restore table data.	• All databases and tables • Certain databases and tables	

Backup and Restoration Use Cases and Performance Specifications

Table 4-2 Backup and restoration use cases and performance specifications

Use Case	Key Performance Factor	Typical Data Volume	Performance Specifications

DB instance backup	• Data size
	• Network configuration
		Data volume: Petabytes
		Object quantity: about 1 million
			OBS backup and restoration specifications:
			1. In a standard environment, a full backup or restoration of 2 TB of data can be completed within 8 hours.
			2. With the right hardware, plenty of OBS bandwidth, a high compression ratio, and independent deployment, the full backup or restoration duration can be calculated using the following formula:
			• Distributed instances
			`Backup or restoration duration = (Total data volume of the DB instance/Number of shards)/min(Disk I/O read bandwidth, Compression bandwidth, Single-thread OBS transmission bandwidth/Compression ratio)`
			• Centralized instances
			`Backup or restoration duration = Total data volume of the DB instance/min(Disk I/O read bandwidth, Compression bandwidth, Single-thread OBS transmission bandwidth/Compression ratio)`

Use Case	Key Performance Factor	Typical Data Volume	Performance Specifications
			NOTE ● min() means that the smallest of the values listed is used. ● Disk read bandwidth: ● SATA SSD: 200 MB/s to 300 MB/s ● SAS SSD: ~500 MB/s ● NVMe SSD: ~1 GB/s Reserving enough bandwidth for database workloads is critical, or backup tasks may severely degrade performance. ● Compression bandwidth: LZ4 compression is used by default. Generally, the compression bandwidth ranges from 300 MB/s to 400 MB/s. The compression level ranges from 1 (default) to 9. Higher levels slow down compression and cause the backup to take longer. The exact time varies depending on data attributes. ● Single-thread OBS transmission bandwidth: 100 MB/s to 300 MB/s in unrestricted mode or the specified limit in speed-restricted mode. ● Compression ratio: LZ4 compression is used by default, achieving a compression ratio between 0.1 and 0.5. The compression ratio depends on various data attributes. ● Setting the parallel upload parameter to **2** or higher increases CPU and other resource usage during backups. Backup performance improves based on the ratio of OBS single-stream transmission bandwidth to total OBS bandwidth. However, if single-stream bandwidth multiplied by the parallel upload parameter exceeds the total bandwidth, no further performance gains are achieved. 3. During the restoration of a backup set for hash bucket tables undergoing scale-out and redistribution in a distributed instance: Restoration time (excluding the redistribution process after restoration) ≤ 2 x Restoration duration of a backup set with the same data volume in the same way during non-scale-out + Redistribution duration of hash bucket tables with the same data volume. When restoring a backup set for hash bucket tables undergoing scale-out and redistribution, there are three steps: ● Step 1: Restore the full backups of all nodes, restore all incremental backups of

Use Case	Key Performance Factor	Typical Data Volume	Performance Specifications
			the old DNs before scale-out, and replay logs. ● Step 2: Physically migrate the hash bucket files to be redistributed from the old DNs to the new DNs. ● Step 3: Restore all incremental backups of the new DNs and replay logs. 4. The time it takes to start up a distributed instance after data restoration depends on the number of sequences and databases involved. ● During startup after restoration, the sequence information of each database is obtained and set in ETCD. Most of the time is spent on acquiring sequence information and configuring sequences in ETCD. – Connecting to each database to acquire sequence information: The more the databases, the longer the time required. – Configuring sequences in ETCD: The more the sequences, the longer the time required. ● Updating PGXC catalog information: When you connect to each database to update the pgxc_class and pgxc_slice catalog information, the more the databases, the longer the time required.

Use Case	Key Performance Factor	Typical Data Volume	Performance Specifications
Database-level physical restoration	● Data size ● Network configuration	-	Database-level physical restoration based on OBS consists of four steps: ● Step 1: Read all data for database-level restoration from the backup media. In a standard Huawei Cloud environment, 2 TB of data can be read within 8 hours. ● Step 2: Run VACUUM FREEZE on database-level data in the auxiliary database. The VACUUM FREEZE performance is as follows: – Distributed instances: 1,400 GB/hour per shard. Parallel replication is allowed between shards. – Centralized instances: 1,400 GB/hour. ● Step 3: Replicate the database-level data after VACUUM FREEZE to each DN replica of the production instance. The replication performance is as follows: – Distributed instances: Replication performance = Data volume of a single shard/min(Network bandwidth, Disk I/O bandwidth). Parallel replication is allowed between shards. – Centralized instances: Replication performance = Database-level data volume to be restored/min(Network bandwidth, Disk I/O bandwidth) ● Step 4: Import data to the production instance. The import performance is as follows: – Distributed instances: Depending on the data volume per shard and disk I/O bandwidth. Parallel replication is allowed between shards. – Centralized instances: Depending on the database-level data volume to be restored and disk I/O bandwidth. Recommended scenarios: ● Performance: For equivalent data volumes, database-level physical restoration achieves approximately 70% of the performance of instance-level physical restoration. If the total

Use Case	Key Performance Factor	Typical Data Volume	Performance Specifications
			database-level data requiring restoration is below 70% of the instance-level data volume, database-level physical restoration is recommended. ● Availability: During a database-level restoration, other databases within the same instance remain operational, ensuring higher availability compared to an instance-level restoration. For uninterrupted access to other databases throughout the process, database-level physical restoration is recommended. Impacts: ● Before a database-level data import, ensure that flow control is disabled and the GUC parameter **recovery_time_target** is set to **0**. Note that during this process, the throughput of the production environment may be impacted, typically reduced to 50% of its peak capacity, or, in extreme cases, as low as 25%. ● To avoid impacting services, perform fine-grained restorations during off-peak hours.

Use Case	Key Performance Factor	Typical Data Volume	Performance Specifications
Table-level physical restoration	● Data size ● Network configuration ● Table storage type ● Table attribute (column) type	-	Table-level physical restoration based on OBS consists of three steps: ● Step 1: Read all data for table-level restoration from the backup media. In a standard Huawei Cloud environment, 2 TB of data can be read within 8 hours. ● Step 2: Export table data from the auxiliary database to a local file. The export performance is about 25 MB/s. ● Step 3: Import the locally exported file into the production instance. When the GUC parameter **page_version_check** is set to **off**, the import speed is about 25 MB/s (setting this parameter to **memory** reduces the performance by about 15%). Additionally, factors such as the row count, table indexes, and triggers can further decrease import speeds to roughly 10 MB/s. Recommended scenarios: ● Performance: For equivalent data volumes, table-level physical restoration operates at approximately one-fifth the speed of instance-level restoration. Table-level physical restoration is recommended when the total data requiring restoration is below one-fifth of the instance-level data volume and does not exceed 1 TB. ● Availability: During a table-level restoration, other databases and tables within the same instance remain operational, ensuring higher availability compared to an instance-level restoration. For uninterrupted access to other databases and tables throughout the process, table-level physical restoration is recommended. Impacts: ● During a table-level restoration, the throughput of the production environment may be impacted, typically reduced to 50% of its peak capacity, or, in extreme cases, as low as 25%. ● To avoid impacting services, perform fine-grained restorations during off-peak hours.

4.2 Instance Restoration

4.2.1 Restoring an Instance from the Recycle Bin

Scenarios

The recycle bin retains a backup generated when an instance was deleted. If the backup has not expired, you can restore the deleted instance by rebuilding it from the recycle bin.

Procedure

Step	Description
Step 1: Prepare Data	Use Data Admin Service (DAS) to create a database and table and insert data into the table.
Step 2: Delete the Instance	Delete the instance.
Step 3: Restore the Instance from the Recycle Bin	Restore the deleted instance from the recycle bin.
Step 4: Check the Results	Log in to the DAS console and check whether the data was restored.

Step 1: Prepare Data

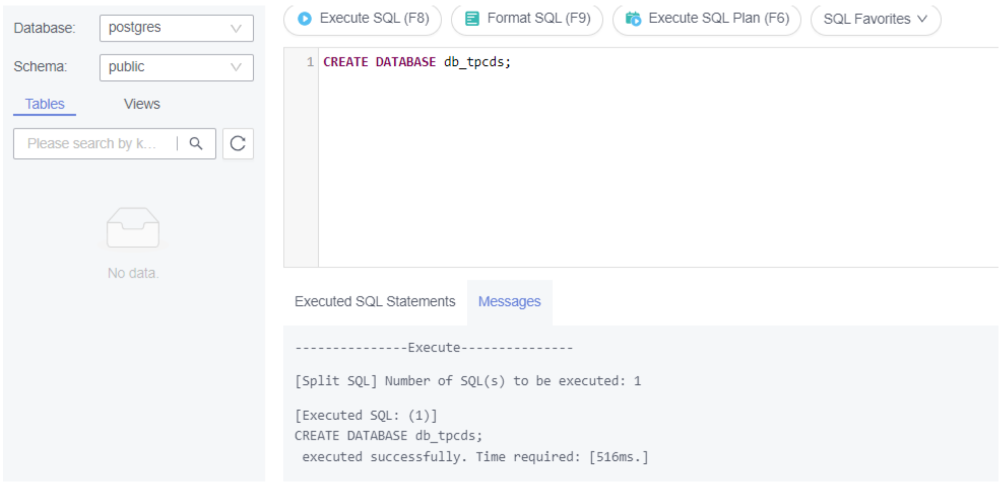
1. **Log in to the management console**.
2. Click in the upper left corner and select a region and project.
3. Click in the upper left corner of the page and choose **Databases** > **GaussDB**.
4. On the **Instances** page, locate the instance and, in the **Operation** column, click **Log In** to access the DAS console.

   Alternatively, click the instance name on the **Instances** page. On the displayed **Basic Information** page, click **Log In** in the upper right corner.
5. Enter the database username and password and click **Test Connection**. After the connection test is successful, click **Log In**.
6. On the menu bar on top, choose **SQL Operations** > **SQL Query**.
7. In the SQL execution window, run the following statement to create a database:

   ```
   CREATE DATABASE db_tpcds;
   ```

   If information shown in the following figure is displayed, the creation was successful.

Figure 4-1 Creating a database



```
CREATE DATABASE db_tpcds;
```

```
--------------Execute--------------

[Split SQL] Number of SQL(s) to be executed: 1

[Executed SQL: (1)]
CREATE DATABASE db_tpcds;
 executed successfully. Time required: [516ms.]
```

Switch to the newly created database **db_tpcds** in the upper left corner.

8. Use SQL statements to create a table and insert data.
   - Create a schema.
     ```
     CREATE SCHEMA myschema;
     ```
     Switch to the newly created schema in the upper left corner.
   - Create a table named **mytable** that has only one column. The column name is **firstcol** and the column type is integer.
     ```
     CREATE TABLE myschema.mytable (firstcol int);
     ```
   - Insert data into the table.
     ```
     INSERT INTO myschema.mytable values (100);
     ```
9. Query table data.
   ```
   SELECT * FROM myschema.mytable;
   ```

Step 2: Delete the Instance

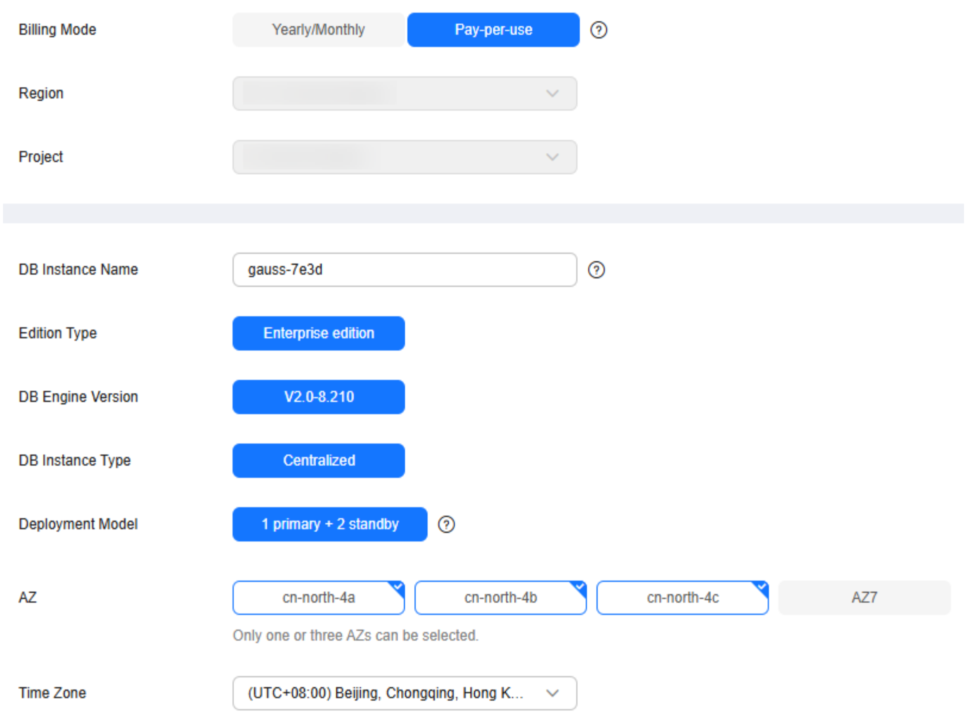
1. **Log in to the management console**.
2. Click in the upper left corner and select a region and project.
3. Click in the upper left corner of the page and choose **Databases** > **GaussDB**.
4. On the **Instances** page, locate the instance you want to delete, click **More** in the **Operation** column, and choose **Delete**.
5. In the **Delete DB Instance** dialog box, enter **DELETE**, select the confirmation check box in the **Confirm** field, and click **OK**. Refresh the **Instances** page later to confirm that the deletion was successful.

Step 3: Restore the Instance from the Recycle Bin

1. **Log in to the management console**.
2. Click in the upper left corner and select a region and project.
3. Click in the upper left corner of the page and choose **Databases** > **GaussDB**.

4. In the navigation pane on the left, choose **Recycle Bin**.
5. Locate the instance to be restored and, in the **Operation** column, click **Rebuild**.
6. On the **Rebuild DB Instance** page, select the billing mode, enter the instance name, and specify other parameters such as the AZs and time zone.

Figure 4-2 Billing mode and basic information



Billing Mode	Yearly/Monthly [✓ Pay-per-use]
Region	
Project	
DB Instance Name	gauss-7e3d
Edition Type	Enterprise edition
DB Engine Version	V2.0-8.210
DB Instance Type	Centralized
Deployment Model	1 primary + 2 standby
AZ	[✓ cn-north-4a] [✓ cn-north-4b] [✓ cn-north-4c] AZ7 Only one or three AZs can be selected.
Time Zone	(UTC+08:00) Beijing, Chongqing, Hong K...

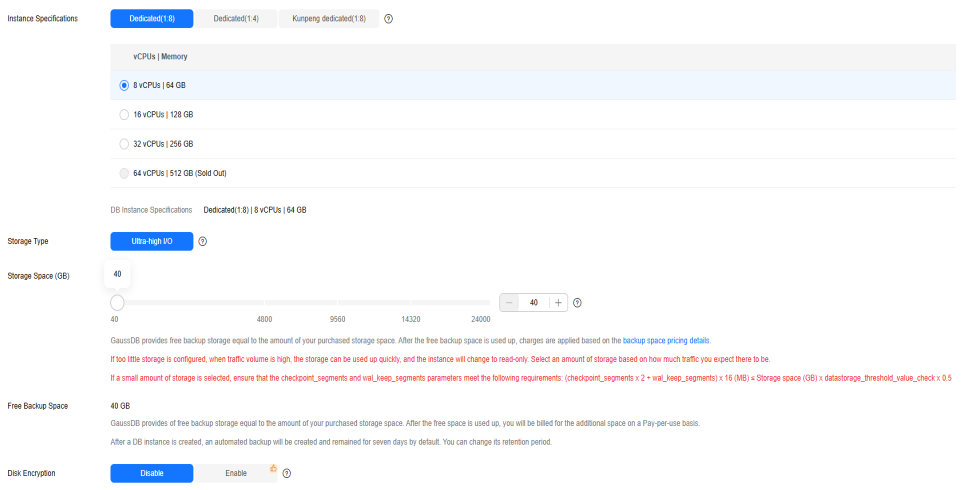
Table 4-3 Parameter description

Parameter	Example Value	Description
Billing Mode	Pay-per-use	GaussDB provides yearly/monthly billing and pay-per-use billing. ● **Yearly/Monthly**: You pay upfront for the amount of time you expect to use the DB instance for. You will need to make sure you have a top-up account with a sufficient balance or have a valid payment method configured first. For distributed instances using the combined deployment model, yearly/monthly billing is only available to authorized users. To apply for the permissions needed, **submit a service ticket**. ● **Pay-per-use**: You can start using the DB instance first and then pay as you go. Pricing is listed on a per-hour basis, but bills are calculated based on the actual usage duration.
DB Instance Name	gauss-7e3d	The instance name is case-sensitive, must start with a letter, and can contain 4 to 64 characters. Only letters, digits, hyphens (-), and underscores (_) are allowed.
Failover Priority	Reliability	This parameter is only available for distributed instances using independent deployment. Additionally, this parameter is only available for authorized users. To apply for the permissions needed, **submit a service ticket**. ● **Reliability**: Data consistency is prioritized during a failover. This is recommended for applications with highest priority for data consistency. ● **Availability**: Database availability is prioritized during a failover. This is recommended for applications that require their databases to provide uninterrupted online services.
AZ	AZ1	An AZ is a physical region where resources have independent power supply and networks. AZs are physically isolated but interconnected through an internal network.

Parameter	Example Value	Description
Time Zone	(UTC+08:00) Beijing, Chongqing, Hong Kong, Urumqi	You need to select a time zone for your instance based on the region it is hosted in.

7. Configure instance specifications.

Figure 4-3 Specifications and storage


Instance Specifications [✓ Dedicated(1:8)] Dedicated(1:4) Kunpeng dedicated(1:8)

vCPUs | Memory
[✓ 8 vCPUs | 64 GB]
16 vCPUs | 128 GB
32 vCPUs | 256 GB
64 vCPUs | 512 GB (Sold Out)

DB Instance Specifications Dedicated(1:8) | 8 vCPUs | 64 GB

Storage Type [✓ Ultra-high I/O]

Storage Space (GB) 40

GaussDB provides free backup storage equal to the amount of your purchased storage space. After the free backup space is used up, charges are applied based on the backup space pricing details.

If too little storage is configured, when traffic volume is high, the storage can be used up quickly, and the instance will change to read-only. Select an amount of storage based on how much traffic you expect there to be.

If a small amount of storage is selected, ensure that the checkpoint_segments and wal_keep_segments parameters meet the following requirements: (checkpoint_segments x 2 + wal_keep_segments) x 16 (MB) ≤ Storage space (GB) x datastorage_threshold_value_check x 0.5

Free Backup Space 40 GB

GaussDB provides of free backup storage equal to the amount of your purchased storage space. After the free space is used up, you will be billed for the additional space on a Pay-per-use basis.

After a DB instance is created, an automated backup will be created and remained for seven days by default. You can change its retention period.

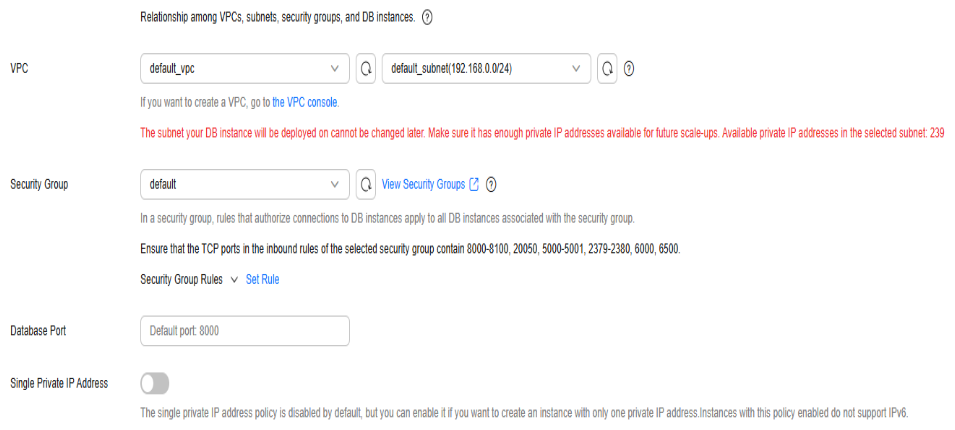
Disk Encryption [✓ Disable] Enable

Table 4-4 Parameter description

Parameter	Example Value	Description
Instance Specifications	Dedicated(1:8); 8 vCPUs | 64 GB	The vCPUs and memory of an instance.
Storage Space	40 GB	The storage space contains the file system overhead required for inodes, reserved blocks, and database operations.
Disk Encryption	Disable	Enabling disk encryption improves data security, but slightly affects the read and write performance of the database. If a shared KMS key is used, the corresponding CTS event is **createGrant**. Only the key owner can receive this event.

8. Retain the default settings for the network configuration.

Figure 4-4 Network configuration



Relationship among VPCs, subnets, security groups, and DB instances.

VPC: default_vpc | default_subnet(192.168.0.0/24)

If you want to create a VPC, go to the VPC console.

The subnet your DB instance will be deployed on cannot be changed later. Make sure it has enough private IP addresses available for future scale-ups. Available private IP addresses in the selected subnet: 239

Security Group: default | View Security Groups

In a security group, rules that authorize connections to DB instances apply to all DB instances associated with the security group.

Ensure that the TCP ports in the inbound rules of the selected security group contain 8000-8100, 20050, 5000-5001, 2379-2380, 6000, 6500.

Security Group Rules | Set Rule

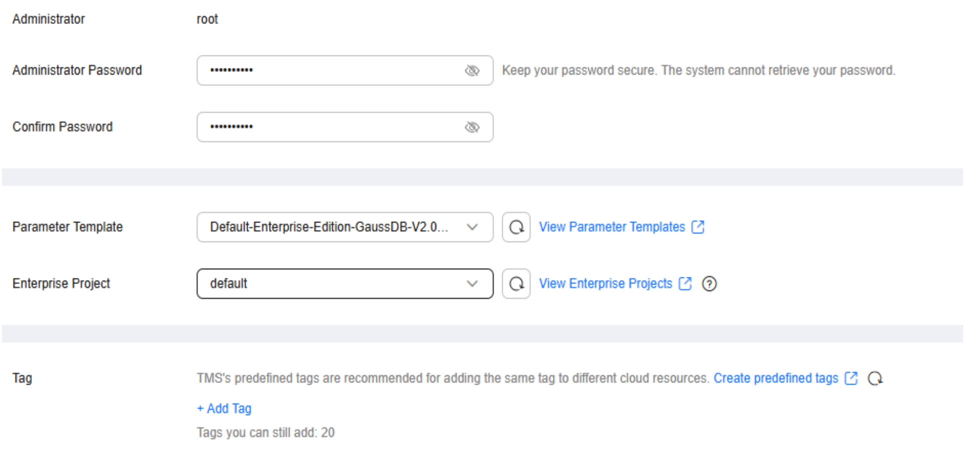
Database Port: Default port: 8000

Single Private IP Address

The single private IP address policy is disabled by default, but you can enable it if you want to create an instance with only one private IP address.Instances with this policy enabled do not support IPv6.

9. Configure the administrator password and enterprise project.

Figure 4-5 Database configuration



Administrator: root

Administrator Password: •••••••••• Keep your password secure. The system cannot retrieve your password.

Confirm Password: ••••••••••

Parameter Template: Default-Enterprise-Edition-GaussDB-V2.0... | View Parameter Templates

Enterprise Project: default | View Enterprise Projects

Tag: TMS's predefined tags are recommended for adding the same tag to different cloud resources. Create predefined tags

+ Add Tag

Tags you can still add: 20

Table 4-5 Parameter description

Parameter	Example Value	Description
Administrator Password	-	Enter a strong password and periodically change it to improve security, preventing security risks such as brute force cracking.
Confirm Password	-	Enter the administrator password again.

Parameter	Example Value	Description
Enterprise Project	default	If the instance has been associated with an enterprise project, select the target project from the **Enterprise Project** drop-down list. You can also go to the Enterprise Project Management console to create a project. For details, see ***Enterprise Management User Guide***.

10. Click **Next**.
11. Confirm the information and click **Submit**.
12. After the task is submitted, check the instance status on the **Instances** page. The rebuild is complete when the status shows **Available**.

Step 4: Check the Results

1. **Log in to the management console**.
2. Click in the upper left corner and select a region and project.
3. Click in the upper left corner of the page and choose **Databases** > **GaussDB**.
4. On the **Instances** page, locate the instance and, in the **Operation** column, click **Log In** to access the DAS console.
5. Enter the database username and password and click **Test Connection**. After the connection test is successful, click **Log In**.
6. Check the database name and table data to verify that the restoration is complete.

4.2.2 Restoring an Instance from a Backup

Scenarios

If a manual backup was created before an instance was deleted, you can restore the instance on the **Backups** page.

Procedure

Step	Description
Step 1: Prepare Data	Use Data Admin Service (DAS) to create a database and table and insert data into the table.
Step 2: Delete the Instance	Delete the instance.

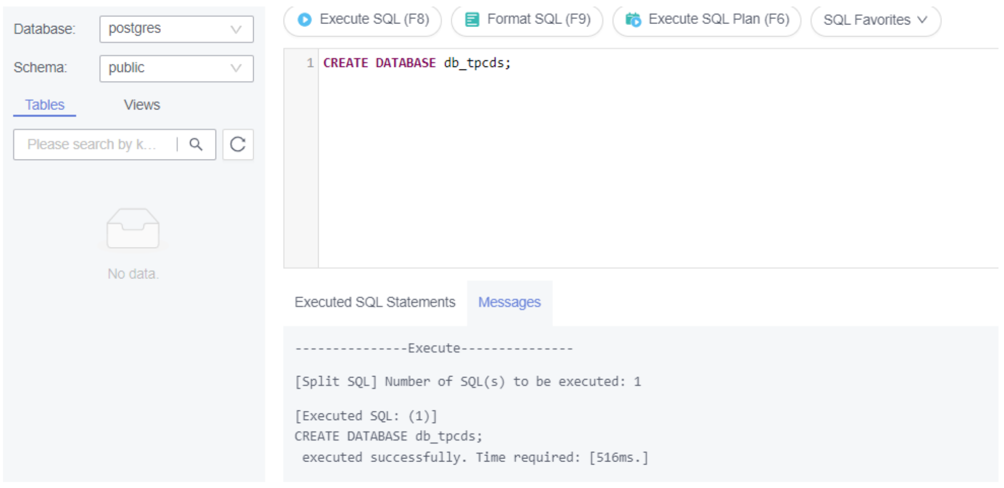
Step	Description
Step 3: Restore an Instance Using a Backup File	Restore your instance data from a backup.
Step 4: Check the Results	Log in to the DAS console and check whether the data was restored.

Step 1: Prepare Data

1. **Log in to the management console**.
2. Click in the upper left corner and select a region and project.
3. Click in the upper left corner of the page and choose **Databases** > **GaussDB**.
4. On the **Instances** page, locate the instance and, in the **Operation** column, click **Log In** to access the DAS console.

   Alternatively, click the instance name on the **Instances** page. On the displayed **Basic Information** page, click **Log In** in the upper right corner.
5. Enter the database username and password and click **Test Connection**. After the connection test is successful, click **Log In**.
6. On the menu bar on top, choose **SQL Operations** > **SQL Query**.
7. In the SQL execution window, run the following statement to create a database:
   ```
   CREATE DATABASE db_tpcds;
   ```
   If information shown in the following figure is displayed, the creation was successful.

   Figure 4-6 Creating a database



```
CREATE DATABASE db_tpcds;
```

```
--------------Execute--------------
[Split SQL] Number of SQL(s) to be executed: 1
[Executed SQL: (1)]
CREATE DATABASE db_tpcds;
 executed successfully. Time required: [516ms.]
```

   Switch to the newly created database **db_tpcds** in the upper left corner.
8. Use SQL statements to create a table and insert data.

- Create a schema.
  ```
  CREATE SCHEMA myschema;
  ```
  Switch to the newly created schema in the upper left corner.
- Create a table named **mytable** that has only one column. The column name is **firstcol** and the column type is integer.
  ```
  CREATE TABLE myschema.mytable (firstcol int);
  ```
- Insert data into the table.
  ```
  INSERT INTO myschema.mytable values (100);
  ```

9. Query table data.
   ```
   SELECT * FROM myschema.mytable;
   ```

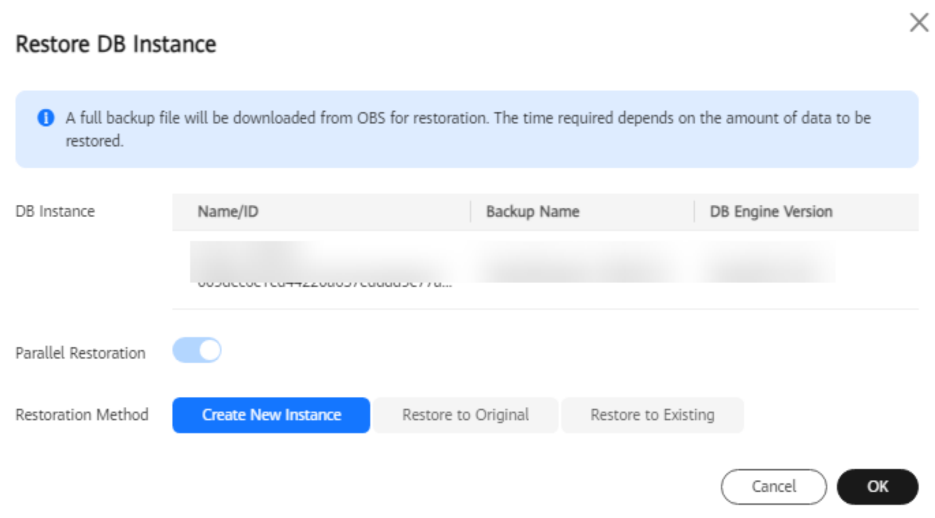
Step 2: Delete the Instance

1. **Log in to the management console**.
2. Click in the upper left corner and select a region and project.
3. Click in the upper left corner of the page and choose **Databases** > **GaussDB**.
4. On the **Instances** page, locate the instance you want to delete, click **More** in the **Operation** column, and choose **Delete**.
5. In the **Delete DB Instance** dialog box, enter **DELETE**, select the confirmation check box in the **Confirm** field, and click **OK**. Refresh the **Instances** page later to confirm that the deletion was successful.

Step 3: Restore an Instance Using a Backup File

1. **Log in to the management console**.
2. Click in the upper left corner and select a region and project.
3. Click in the upper left corner of the page and choose **Databases** > **GaussDB**.
4. In the navigation pane, choose **Backups**. On the **Backups** page, locate the backup to be restored and click **Restore** in the **Operation** column.
5. Set **Restoration Method** to **Create New Instance** and click **OK**.

Figure 4-7 Restoring data from a backup



Restore DB Instance

A full backup file will be downloaded from OBS for restoration. The time required depends on the amount of data to be restored.

DB Instance

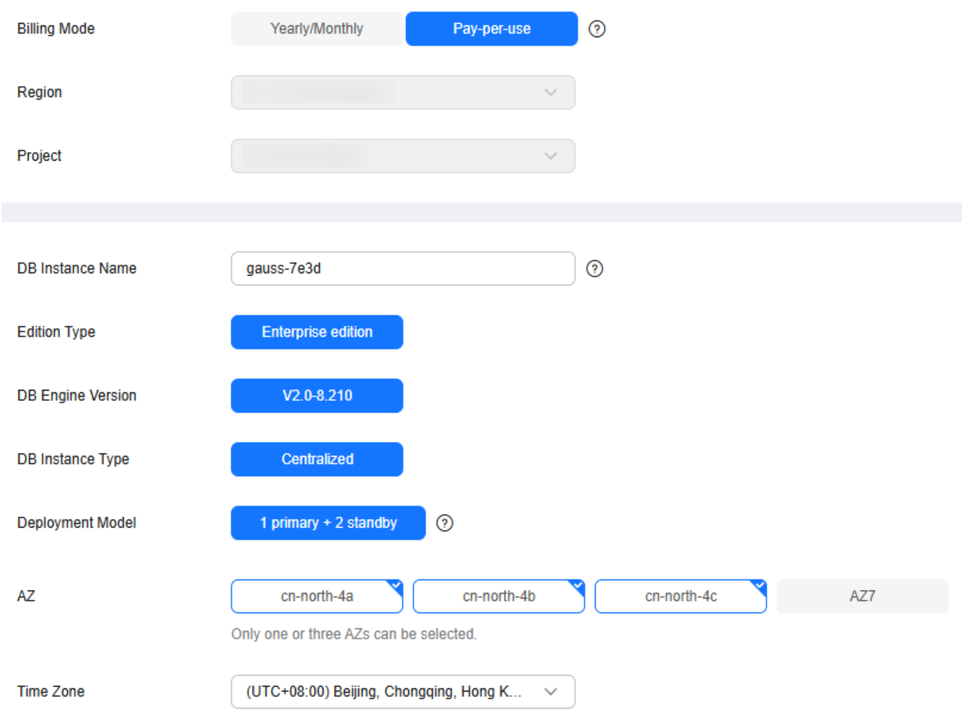
Name/ID	Backup Name	DB Engine Version
[illegible]	[illegible]	[illegible]

Parallel Restoration

Restoration Method: Create New Instance | Restore to Original | Restore to Existing

6. On the **Create New Instance** page, select the billing mode, enter the instance name, and specify other parameters such as the AZs and time zone.

Figure 4-8 Billing mode and basic information



Billing Mode	Yearly/Monthly [✓ Pay-per-use]
Region	
Project	
DB Instance Name	gauss-7e3d
Edition Type	Enterprise edition
DB Engine Version	V2.0-8.210
DB Instance Type	Centralized
Deployment Model	1 primary + 2 standby
AZ	[✓ cn-north-4a] [✓ cn-north-4b] [✓ cn-north-4c] AZ7
	Only one or three AZs can be selected.
Time Zone	(UTC+08:00) Beijing, Chongqing, Hong K...

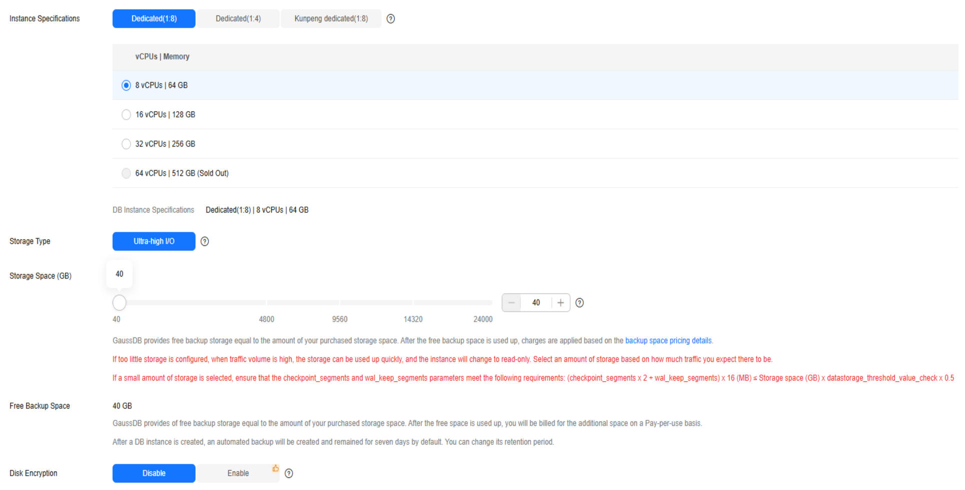
Table 4-6 Parameter description

Parameter	Example Value	Description
Billing Mode	Pay-per-use	GaussDB provides yearly/monthly billing and pay-per-use billing. ● **Yearly/Monthly**: You pay upfront for the amount of time you expect to use the DB instance for. You will need to make sure you have a top-up account with a sufficient balance or have a valid payment method configured first. For distributed instances using the combined deployment model, yearly/monthly billing is only available to authorized users. To apply for the permissions needed, **submit a service ticket**. ● **Pay-per-use**: You can start using the DB instance first and then pay as you go. Pricing is listed on a per-hour basis, but bills are calculated based on the actual usage duration.
DB Instance Name	gauss-7e3d	The instance name is case-sensitive, must start with a letter, and can contain 4 to 64 characters. Only letters, digits, hyphens (-), and underscores (_) are allowed.
Failover Priority	Reliability	This parameter is only available for distributed instances using independent deployment. Additionally, this parameter is only available for authorized users. To apply for the permissions needed, **submit a service ticket**. ● **Reliability**: Data consistency is prioritized during a failover. This is recommended for applications with highest priority for data consistency. ● **Availability**: Database availability is prioritized during a failover. This is recommended for applications that require their databases to provide uninterrupted online services.
AZ	AZ1	An AZ is a physical region where resources have independent power supply and networks. AZs are physically isolated but interconnected through an internal network.

Parameter	Example Value	Description
Time Zone	(UTC+08:00) Beijing, Chongqing, Hong Kong, Urumqi	You need to select a time zone for your instance based on the region it is hosted in.

7. Configure instance specifications.

Figure 4-9 Specifications and storage



Instance Specifications: [✓ Dedicated(1:8)] Dedicated(1:4) Kunpeng dedicated(1:8)

vCPUs | Memory
[✓ 8 vCPUs | 64 GB]
16 vCPUs | 128 GB
32 vCPUs | 256 GB
64 vCPUs | 512 GB (Sold Out)

DB Instance Specifications: Dedicated(1:8) | 8 vCPUs | 64 GB

Storage Type: [✓ Ultra-high I/O]

Storage Space (GB): 40

40 4800 9560 14320 24000

GaussDB provides free backup storage equal to the amount of your purchased storage space. After the free backup space is used up, charges are applied based on the backup space pricing details.

If too little storage is configured, when traffic volume is high, the storage can be used up quickly, and the instance will change to read-only. Select an amount of storage based on how much traffic you expect there to be.

If a small amount of storage is selected, ensure that the checkpoint_segments and wal_keep_segments parameters meet the following requirements: (checkpoint_segments x 2 + wal_keep_segments) x 16 (MB) ≤ Storage space (GB) x datastorage_threshold_value_check x 0.5

Free Backup Space: 40 GB

GaussDB provides of free backup storage equal to the amount of your purchased storage space. After the free space is used up, you will be billed for the additional space on a Pay-per-use basis.

After a DB instance is created, an automated backup will be created and remained for seven days by default. You can change its retention period.

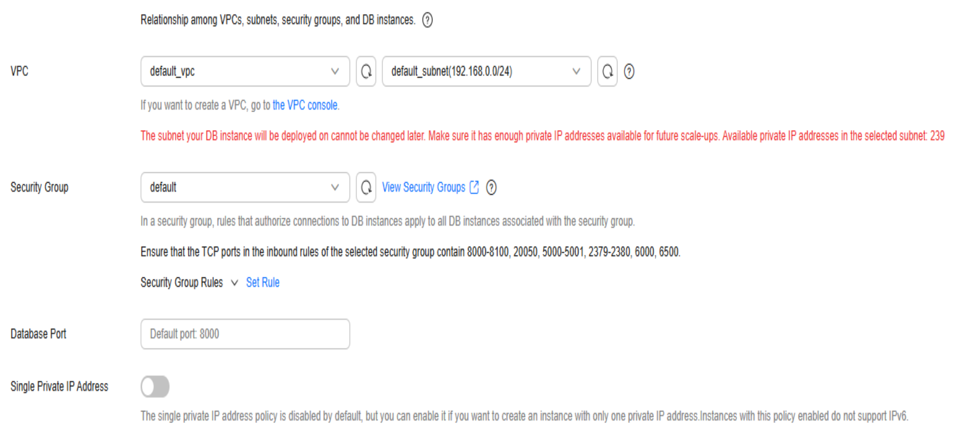
Disk Encryption: [✓ Disable] Enable

Table 4-7 Parameter description

Parameter	Example Value	Description
Instance Specifications	Dedicated(1:8); 8 vCPUs | 64 GB	The vCPUs and memory of an instance.
Storage Space	40 GB	The storage space contains the file system overhead required for inodes, reserved blocks, and database operations.
Disk Encryption	Disable	Enabling disk encryption improves data security, but slightly affects the read and write performance of the database. If a shared KMS key is used, the corresponding CTS event is **createGrant**. Only the key owner can receive this event.

8. Retain the default settings for the network configuration.

Figure 4-10 Network configuration

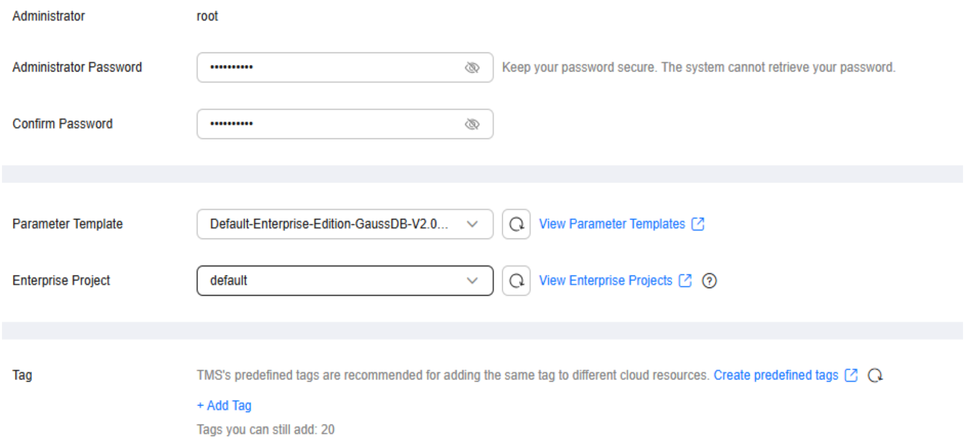


Relationship among VPCs, subnets, security groups, and DB instances.

VPC	default_vpc; default_subnet(192.168.0.0/24)
	If you want to create a VPC, go to the VPC console.
	The subnet your DB instance will be deployed on cannot be changed later. Make sure it has enough private IP addresses available for future scale-ups. Available private IP addresses in the selected subnet: 239
Security Group	default — View Security Groups
	In a security group, rules that authorize connections to DB instances apply to all DB instances associated with the security group.
	Ensure that the TCP ports in the inbound rules of the selected security group contain 8000-8100, 20050, 5000-5001, 2379-2380, 6000, 6500.
	Security Group Rules — Set Rule
Database Port	Default port: 8000
Single Private IP Address	
	The single private IP address policy is disabled by default, but you can enable it if you want to create an instance with only one private IP address.Instances with this policy enabled do not support IPv6.

9. Configure the administrator password and enterprise project.

Figure 4-11 Database configuration



Administrator	root
Administrator Password	•••••••••• — Keep your password secure. The system cannot retrieve your password.
Confirm Password	••••••••••
Parameter Template	Default-Enterprise-Edition-GaussDB-V2.0... — View Parameter Templates
Enterprise Project	default — View Enterprise Projects
Tag	TMS's predefined tags are recommended for adding the same tag to different cloud resources. Create predefined tags
	+ Add Tag
	Tags you can still add: 20

Table 4-8 Parameter description

Parameter	Example Value	Description
Administrator Password	-	Enter a strong password and periodically change it to improve security, preventing security risks such as brute force cracking.
Confirm Password	-	Enter the administrator password again.

Parameter	Example Value	Description
Enterprise Project	default	If the instance has been associated with an enterprise project, select the target project from the **Enterprise Project** drop-down list. You can also go to the Enterprise Project Management console to create a project. For details, see ***Enterprise Management User Guide***.

10. Click **Next**.
11. Confirm the information and click **Submit**.
12. After the task is submitted, check the instance status on the **Instances** page. The restoration is complete when the status shows **Available**.

Step 4: Check the Results

1. **Log in to the management console**.
2. Click ⊙ in the upper left corner and select a region and project.
3. Click ☰ in the upper left corner of the page and choose **Databases** > **GaussDB**.
4. On the **Instances** page, locate the instance and, in the **Operation** column, click **Log In** to access the DAS console.
5. Enter the database username and password and click **Test Connection**. After the connection test is successful, click **Log In**.
6. Check the database name and table data to verify that the restoration is complete.

4.3 Database and Table Restoration

4.3.1 Restoring Databases or Tables to a Specific Point in Time

Scenarios

When a table-level automated backup policy is enabled, you can use existing backups to restore lost table data to a specific point in time in the case of accidental or intentional deletions.

Procedure

Step	Description
Step 1: Prepare Data	Use Data Admin Service (DAS) to create a database and table and insert data into the table.

Step	Description
Step 2: Delete Table Data	Delete the instance.
Step 3: Restore Table Data	Restore the deleted table data using table-level point-in-time recovery (PITR).
Step 4: Check the Results	Log in to the DAS console and check whether the data was restored.

Step 1: Prepare Data

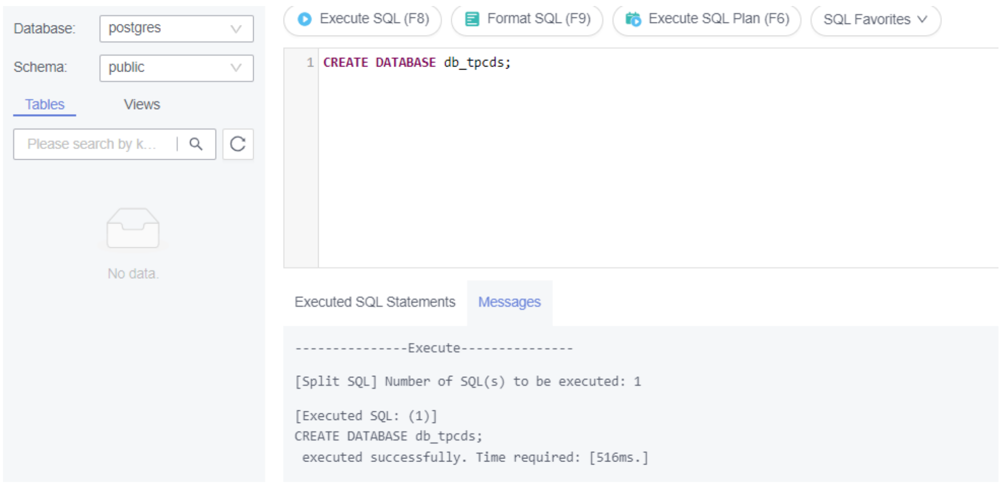
1. Log in to the management console.
2. Click in the upper left corner and select a region and project.
3. Click in the upper left corner of the page and choose **Databases** > **GaussDB**.
4. On the **Instances** page, locate the instance and, in the **Operation** column, click **Log In** to access the DAS console.

   Alternatively, click the instance name on the **Instances** page. On the displayed **Basic Information** page, click **Log In** in the upper right corner.
5. Enter the database username and password and click **Test Connection**. After the connection test is successful, click **Log In**.
6. On the menu bar on top, choose **SQL Operations** > **SQL Query**.
7. In the SQL execution window, run the following statement to create a database:

```
CREATE DATABASE db_tpcds;
```

   If information shown in the following figure is displayed, the creation was successful.

Figure 4-12 Creating a database



```
CREATE DATABASE db_tpcds;
```

```
--------------Execute--------------
[Split SQL] Number of SQL(s) to be executed: 1
[Executed SQL: (1)]
CREATE DATABASE db_tpcds;
 executed successfully. Time required: [516ms.]
```

Switch to the newly created database **db_tpcds** in the upper left corner.

8. Use SQL statements to create a table and insert data.
   - Create a schema.
     ```
     CREATE SCHEMA myschema;
     ```
     Switch to the newly created schema in the upper left corner.
   - Create a table named **mytable** that has only one column. The column name is **firstcol** and the column type is integer.
     ```
     CREATE TABLE myschema.mytable (firstcol int);
     ```
   - Insert data into the table.
     ```
     INSERT INTO myschema.mytable values (100);
     ```
9. Query table data.
   ```
   SELECT * FROM myschema.mytable;
   ```

Step 2: Delete Table Data

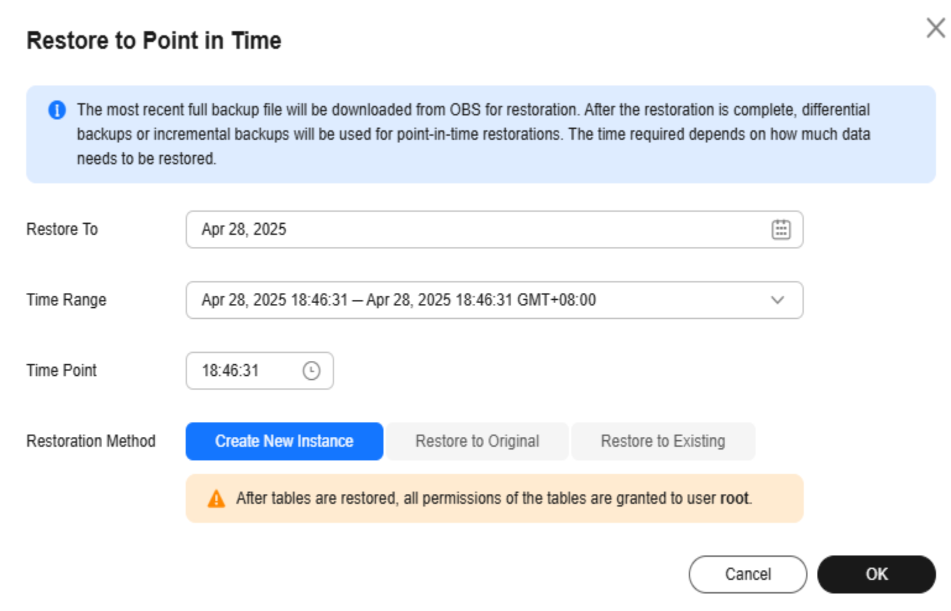
1. **Log in to the management console**.
2. Click in the upper left corner and select a region and project.
3. Click in the upper left corner of the page and choose **Databases** > **GaussDB**.
4. On the **Instances** page, locate the instance and, in the **Operation** column, click **Log In** to access the DAS console.

   Alternatively, click the instance name on the **Instances** page. On the displayed **Basic Information** page, click **Log In** in the upper right corner.
5. Enter the database username and password and click **Test Connection**. After the connection test is successful, click **Log In**.
6. On the menu bar on top, choose **SQL Operations** > **SQL Query**.
7. Delete table data.
   ```
   DELETE FROM myschema.mytable WHERE firstcol = 100;
   ```

Step 3: Restore Table Data

1. **Log in to the management console**.
2. Click in the upper left corner and select a region and project.
3. Click in the upper left corner of the page and choose **Databases** > **GaussDB**.
4. On the **Instances** page, click the name of the instance to access the **Basic Information** page.
5. In the navigation pane, choose **Backups**. On the displayed page, click the **Table Backup** tab.
6. Click **Restore to Point in Time**. In the displayed dialog box, specify the time range and point for restoration, and set **Restoration Method** to **Restore to Original**.

Figure 4-13 Restoring data to a specified point in time


Restore to Point in Time

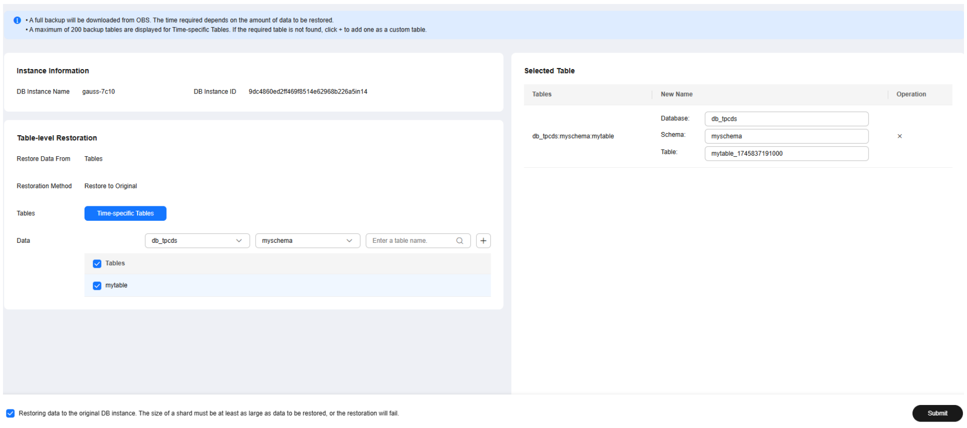
The most recent full backup file will be downloaded from OBS for restoration. After the restoration is complete, differential backups or incremental backups will be used for point-in-time restorations. The time required depends on how much data needs to be restored.

Restore To	Apr 28, 2025
Time Range	Apr 28, 2025 18:46:31 – Apr 28, 2025 18:46:31 GMT+08:00
Time Point	18:46:31
Restoration Method	Create New Instance

After tables are restored, all permissions of the tables are granted to user **root**.

7. Select the tables to be restored and click **Submit**.

Figure 4-14 Selecting tables to be restored


• A full backup will be downloaded from OBS. The time required depends on the amount of data to be restored.
• A maximum of 200 backup tables are displayed for Time-specific Tables. If the required table is not found, click + to add one as a custom table.

Instance Information

DB Instance Name gauss-7c10 DB Instance ID 9dc4860ed2ff469f8514e62968b226a5in14

Table-level Restoration

Restore Data From	Tables
Restoration Method	Restore to Original
Tables	Time-specific Tables
Data	db_tpcds / myschema / Enter a table name.

✓ Tables
✓ mytable

Selected Table

Tables	New Name	Operation
db_tpcds:myschema:mytable	Database: db_tpcds Schema: myschema Table: mytable_1745837191000	×

✓ Restoring data to the original DB instance. The size of a shard must be at least as large as data to be restored, or the restoration will fail.

8. Go to the **Instances** page and check that the target instance is in the **Restoring** state. When the instance status changes to **Available**, the restoration is complete.

Step 4: Check the Results

1. Log in to the management console.
2. Click in the upper left corner and select a region and project.
3. Click in the upper left corner of the page and choose **Databases** > **GaussDB**.
4. On the **Instances** page, locate the instance and, in the **Operation** column, click **Log In** to access the DAS console.

5. Enter the database username and password and click **Test Connection**. After the connection test is successful, click **Log In**.
6. Check the database name and table data to verify that the restoration is complete.

4.3.2 Restoring Databases or Tables Using a Backup

Scenarios

When a table-level automated backup policy is enabled or a manual backup is available, you can restore tables to the exact state captured by an existing backup, recovering any accidentally or intentionally deleted data.

Procedure

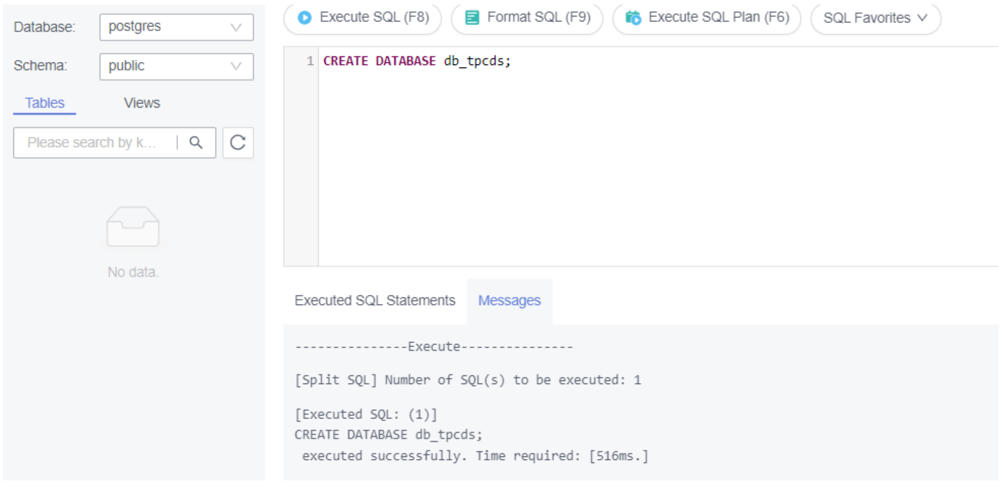
Step	Description
Step 1: Prepare Data	Use Data Admin Service (DAS) to create a database and table and insert data into the table.
Step 2: Delete Table Data	Delete the instance.
Step 3: Restore Table Data	Restore the deleted table data using a table-level backup.
Step 4: Check the Results	Log in to the DAS console and check whether the data was restored.

Step 1: Prepare Data

1. **Log in to the management console**.
2. Click ⊙ in the upper left corner and select a region and project.
3. Click ☰ in the upper left corner of the page and choose **Databases** > **GaussDB**.
4. On the **Instances** page, locate the instance and, in the **Operation** column, click **Log In** to access the DAS console.

   Alternatively, click the instance name on the **Instances** page. On the displayed **Basic Information** page, click **Log In** in the upper right corner.
5. Enter the database username and password and click **Test Connection**. After the connection test is successful, click **Log In**.
6. On the menu bar on top, choose **SQL Operations** > **SQL Query**.
7. In the SQL execution window, run the following statement to create a database:
   ```
   CREATE DATABASE db_tpcds;
   ```
   If information shown in the following figure is displayed, the creation was successful.

Figure 4-15 Creating a database



```
CREATE DATABASE db_tpcds;
```

```
--------------Execute--------------
[Split SQL] Number of SQL(s) to be executed: 1
[Executed SQL: (1)]
CREATE DATABASE db_tpcds;
 executed successfully. Time required: [516ms.]
```

Switch to the newly created database **db_tpcds** in the upper left corner.

8. Use SQL statements to create a table and insert data.
   - Create a schema.
     ```
     CREATE SCHEMA myschema;
     ```
     Switch to the newly created schema in the upper left corner.
   - Create a table named **mytable** that has only one column. The column name is **firstcol** and the column type is integer.
     ```
     CREATE TABLE myschema.mytable (firstcol int);
     ```
   - Insert data into the table.
     ```
     INSERT INTO myschema.mytable values (100);
     ```
9. Query table data.
   ```
   SELECT * FROM myschema.mytable;
   ```

Step 2: Delete Table Data

1. Log in to the management console.
2. Click in the upper left corner and select a region and project.
3. Click in the upper left corner of the page and choose **Databases** > **GaussDB**.
4. On the **Instances** page, locate the instance and, in the **Operation** column, click **Log In** to access the DAS console.

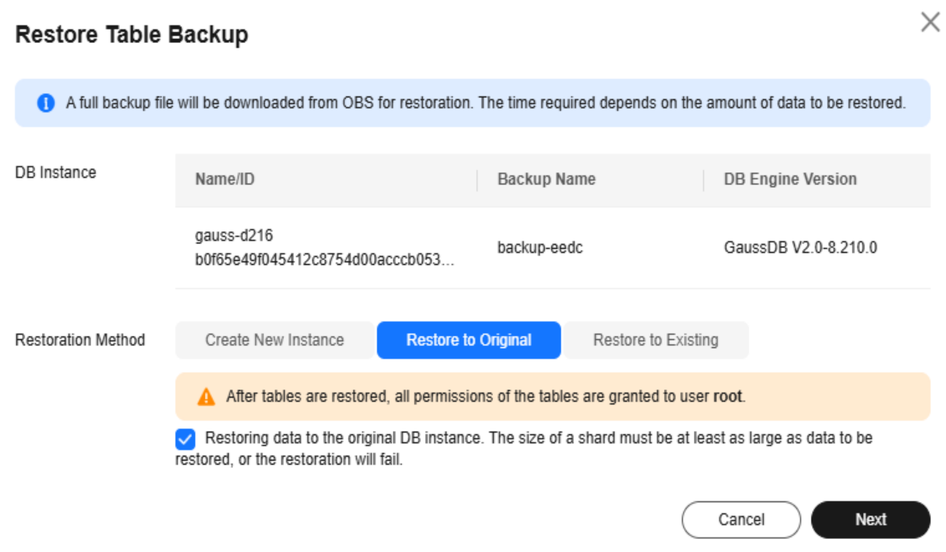
   Alternatively, click the instance name on the **Instances** page. On the displayed **Basic Information** page, click **Log In** in the upper right corner.
5. Enter the database username and password and click **Test Connection**. After the connection test is successful, click **Log In**.
6. On the menu bar on top, choose **SQL Operations** > **SQL Query**.
7. Delete table data.
   ```
   DELETE FROM myschema.mytable WHERE firstcol = 100;
   ```

Step 3: Restore Table Data

1. Log in to the management console.

2. Click in the upper left corner and select a region and project.
3. Click in the upper left corner of the page and choose **Databases** > **GaussDB**.
4. On the **Instances** page, click the name of the instance to access the **Basic Information** page.
5. In the navigation pane, choose **Backups**. On the displayed page, click the **Table Backup** tab.
6. Locate the backup and click **Restore** in the **Operation** column.
7. Set **Restoration Method** to **Restore to Original**, and click **Next**.

Figure 4-16 Restore Table Backup


Restore Table Backup

A full backup file will be downloaded from OBS for restoration. The time required depends on the amount of data to be restored.

DB Instance

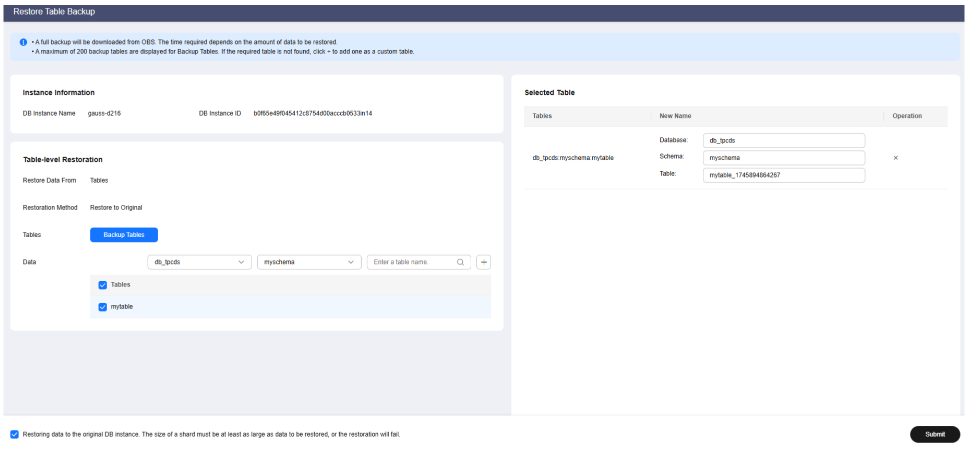
Name/ID	Backup Name	DB Engine Version
gauss-d216 b0f65e49f045412c8754d00acccb053...	backup-eedc	GaussDB V2.0-8.210.0

Restoration Method: Create New Instance [✓ Restore to Original] Restore to Existing

After tables are restored, all permissions of the tables are granted to user root.

☑ Restoring data to the original DB instance. The size of a shard must be at least as large as data to be restored, or the restoration will fail.

8. Select the tables to be restored and click **Submit**.


Restore Table Backup

• A full backup will be downloaded from OBS. The time required depends on the amount of data to be restored.
• A maximum of 200 backup tables are displayed for Backup Tables. If the required table is not found, click + to add one as a custom table.

Instance Information

DB Instance Name gauss-d216 DB Instance ID b0f65e49f045412c8754d00acccb0533in14

Table-level Restoration

Restore Data From	Tables
Restoration Method	Restore to Original
Tables	Backup Tables
Data	db_tpcds / myschema / Enter a table name.

☑ Tables
☑ mytable

Selected Table

Tables	New Name	Operation
db_tpcds:myschema:mytable	Database: db_tpcds Schema: myschema Table: mytable_1745894864267	×

☑ Restoring data to the original DB instance. The size of a shard must be at least as large as data to be restored, or the restoration will fail.

9. Go to the **Instances** page and check that the target instance is in the **Restoring** state. When the instance status changes to **Available**, the restoration is complete.

Step 4: Check the Results

1. **Log in to the management console**.
2. Click in the upper left corner and select a region and project.
3. Click in the upper left corner of the page and choose **Databases** > **GaussDB**.
4. On the **Instances** page, locate the instance and, in the **Operation** column, click **Log In** to access the DAS console.
5. Enter the database username and password and click **Test Connection**. After the connection test is successful, click **Log In**.
6. Check the database name and table data to verify that the restoration is complete.

5 Suggestions on GaussDB Metric Alarm Configuration

You can set alarm rules on the Cloud Eye console to specify the monitored objects and notification policies for your instances and keep track of the instance status. This section describes how to configure GaussDB metric alarm rules.

Creating a Metric Alarm Rule

Step 1 **Log in to the management console**.

Step 2 Under **Management & Governance** of the service list, click **Cloud Eye**.

Step 3 In the navigation pane on the left, choose **Cloud Service Monitoring**.

Step 4 Click **GaussDB** in the list.

Step 5 Locate the instance for which you want to create an alarm rule, click **More** in the **Operation** column, and choose **Create Alarm Rule**.

Step 6 On the displayed page, set parameters as required.

Table 5-1 Alarm rule information

Parameter	Description
Name	Alarm rule name. The system generates a random name, and you can change it if needed. The value can contain only letters, digits, underscores (_), and hyphens (-), and cannot exceed 128 characters.
Description	Description of the alarm rule. The value can contain a maximum of 256 characters. This parameter is optional.

Parameter	Description
Method	Mode for configuring an alarm policy. You can select **Associate template** or **Configure manually**. ● **Configure manually**: You can create a custom alarm policy as needed. ● **Associate template**: If the same alarm rule needs to be configured for multiple GaussDB instances, you can use an alarm template to simplify operation.
Template	This parameter is only available if you select **Associate template** for **Method**. You can select a default alarm template or create a custom one. After an associated template is modified, the policies contained in this alarm rule to be created will be updated accordingly.
Alarm Policy	This parameter is only available if you select **Configure manually** for **Method**. An alarm is triggered when the metric configured for this alarm reaches the preset threshold in consecutive periods. For example, an alarm is triggered if the average CPU usage is 80% or higher for three consecutive 5-minute periods. A maximum of 50 alarm policies can be added to an alarm rule. If any of these alarm policies is met, an alarm will be triggered.

Table 5-2 Alarm notification

Parameter	Description
Alarm Notifications	Specifies whether to notify users when alarms are triggered. Notifications can be sent by email or text messages, or through HTTP/HTTPS requests to servers. This function is enabled by default.
Notified By	The following three options are available: ● **Notification policies**: Flexible alarm notifications by severity and more notification channels are provided. ● **Notification groups**: Configure notification templates on the Cloud Eye console. ● **Topic subscriptions**: Configure notification templates on the Simple Message Notification (SMN) console.
Notification Policies	This parameter is only available if you select **Notification policies** for **Notified By**. Select one or more notification policies. You can specify the notification group, window, template, and other parameters in a notification policy. For how to create a notification policy, see **Creating, Modifying, or Deleting a Notification Policy**.

Parameter	Description
Notification Group	This parameter is only available if you select **Notification groups** for **Notified By**. Select the notification groups to which alarm notifications will be sent. For details about how to create a notification group, see **Creating a Recipient and Notification Group**.
Recipient	This parameter is only available if you select **Topic subscriptions** for **Notified By**. You can select the account contact or a topic as the object to which alarm notifications will be sent. ● The account contact is the mobile phone number and email address of the registered account. ● A topic is a specific event type for publishing messages or subscribing to notifications. If the required topic is unavailable, create one first and add subscriptions to it. For details, see **Creating a Topic** and **Adding Subscriptions**.
Notification Template	This parameter is only available if you select **Notification groups** or **Topic subscriptions** for **Notified By**. You can select an existing template or create a new one to send alarm notifications.
Notification Window	This parameter is only available if you select **Notification groups** or **Topic subscriptions** for **Notified By**. Cloud Eye sends notifications only within the notification window you specified. If **Notification Window** is set to **08:00-20:00**, alarm notifications are sent only from 08:00 to 20:00.
Trigger Condition	This parameter is only available if you select **Notification groups** or **Topic subscriptions** for **Notified By**. You can select either **Generated alarm** or **Cleared alarm**, or both.
Enterprise Project	Enterprise project that the alarm rule belongs to. Only users who have the permissions of the enterprise project can view and manage this alarm rule.
Tags	Key-value pairs that you can use to easily categorize and search for cloud resources. You are advised to create predefined tags in Tag Management Service (TMS). If your organization has configured tag policies for Cloud Eye, you need to add tags to alarm rules based on tag policies. If a tag does not comply with the policies, an alarm rule may fail to be created. Contact your organization administrator to learn more about tag policies. ● A key can contain up to 128 characters, and a value can contain up to 225 characters. ● You can add up to 20 tags.

Step 7 Click **Create**. The alarm rule is created.

For details about how to create alarm rules, see **Creating an Alarm Rule**.

----End

Metric Alarm Configuration Suggestions

Metric ID	Metric Name	Definition	Threshold in Best Practices	Alarm Severity in Best Practices
io_bandwidth_usage	Disk I/O Bandwidth Usage	Percentage of the maximum disk I/O bandwidth currently used	Raw data > 80% for three consecutive periods	Major
iops_usage	IOPS Usage	Percentage of the maximum disk IOPS currently used	Raw data > 80% for three consecutive periods	Major
rds001_cpu_util	CPU Usage	CPU usage of a measured object	Raw data > 80% for three consecutive periods	Major
rds002_mem_util	Memory Usage	Memory usage of a monitored object	Raw data > 90% for three consecutive periods	Major
rds007_instance_disk_usage	Instance Disk Usage	Real-time data disk usage of the monitored instance	Raw data > 75% for three consecutive periods (The threshold should not be set above 80%.)	Major

Metric ID	Metric Name	Definition	Threshold in Best Practices	Alarm Severity in Best Practices
rds020_avg_disk_ms_per_write	Time Required for per Disk Write	Average time required for a data disk write on the monitored node in a measurement period	Raw data > 8 ms for three consecutive periods	Major
rds021_avg_disk_ms_per_read	Time Required for per Disk Read	Average time required for a data disk read on the monitored node in a measurement period	Raw data > 8 ms for three consecutive periods	Major
rds036_deadlocks	Deadlocks	Incremental number of database transaction deadlocks in a measurement period	Raw data > 5 counts for three consecutive periods	Major
rds048_P80	Response Time of 80% SQL Statements	Real-time response time of 80% of database SQL statements	Raw data > 10000000 μs for three consecutive periods	Major
rds049_P95	Response Time of 95% SQL Statements	Real-time response time of 95% of database SQL statements	Raw data > 15000000 μs for three consecutive periods	Major

Metric ID	Metric Name	Definition	Threshold in Best Practices	Alarm Severity in Best Practices
rds060_long_running_transaction_exectime	Maximum Execution Duration of Database Transactions	Real-time maximum execution duration of database transactions execution on a monitored object	Raw data > 7200s for three consecutive periods (You are advisable to manually terminate a transaction if its duration is longer than 2 hours. Adjust this threshold based on workload requirements.)	Major
rds063_slowquery_user	Slow SQL Statements in the User Database	Real-time number of slow SQL statements in the user databases on the primary DN or CN in a measurement period	Raw data > 15 counts for three consecutive periods	Major
rds065_dynamic_used_memory_usage	Dynamic Memory Usage	Real-time dynamic memory usage of a monitored object	Raw data > 80% for three consecutive periods	Major
rds066_replication_slot_wal_log_size	WAL Log Size in the Replication Slot	Real-time size of WAL logs reserved in the replication slot of a primary DN	Raw data > *[10% of the storage]* bytes for three consecutive periods (10% is the recommended value. Adjust this threshold based on the purchased storage.)	Major

Metric ID	Metric Name	Definition	Threshold in Best Practices	Alarm Severity in Best Practices
rds070_thread_pool	Thread Pool Usage	Real-time thread pool usage on a CN or DN	Raw data > 85% for three consecutive periods	Major

6 Best Practices for Row Compression

6.1 Scenario Overview

One of the purposes of row compression is to reuse the space saved after compressing tables. The process can be summarized as follows:

- **Compression execution.** The system traverses each page of tables with Information Lifecycle Management (ILM) enabled and performs compression. The compressed data is referred to as BCA and remains stored in the original page. After compression, the page has more available space. Note that the space saved after compression is not released back to the operating system. For details about ILM, see **Data Lifecycle Management: OLTP Table Compression**.
- **Data insertion.** When new data is inserted, the system prioritizes pages with enough remaining space to accommodate the new rows. By increasing the amount of free space available in pages, row compression reduces the actual storage occupied by the table.

The following examples demonstrate how row compression can be scheduled and executed:

- **Manual scheduling**: You need to manually call the compression interface, and only one task can be generated at a time.
- **Automatic scheduling**: After the prerequisite configuration is complete, the system automatically creates compression tasks in the background through scheduled jobs, and one scheduling cycle can generate multiple tasks.

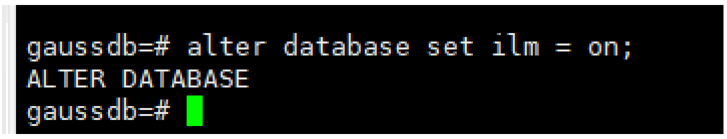
6.2 Manual Scheduling

Step 1 **Log in to the GaussDB management console**. On the **Instances** page, click the name of the target instance to go to the **Basic Information** page.

Step 2 Locate the **Advanced Features** field and click **View and Modify**. On the displayed page, set the value of **Advanced compression** to **on**. For details, see **Viewing and Modifying Advanced Features**.

Step 3 Connect to the database and run the following command to enable ILM: For details about how to connect to the database, see **Using gsql to Connect to an Instance**.

```
alter database set ilm = on;
```

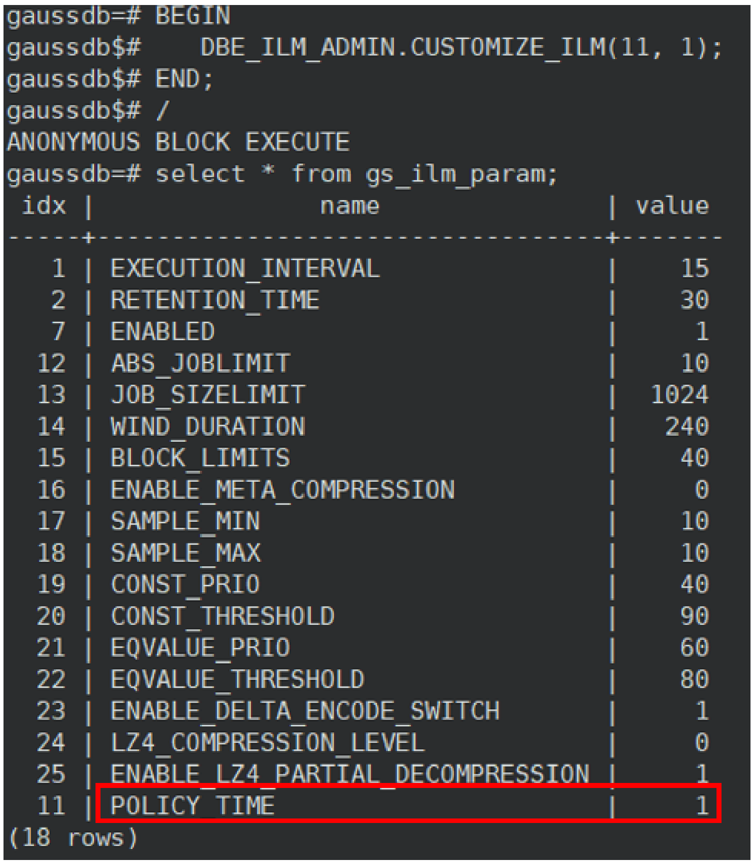


```
gaussdb=# alter database set ilm = on;
ALTER DATABASE
gaussdb=#
```

Step 4 Change the ILM time unit to seconds.

This step is mainly for accelerating testing. By default, ILM uses days as the time unit. When you run manual compression, the first scheduling only records a timestamp, and the second scheduling performs the actual compression. The interval between the two must exceed the cold data threshold to trigger compression. Using seconds as the time unit shortens this waiting period.

1. Change the ILM time unit to seconds.
```
BEGIN
DBE_ILM_ADMIN.CUSTOMIZE_ILM(11, 1);
END;
/
```
2. Check the modification result.
```
select * from gs_ilm_param;
```



```
gaussdb=# BEGIN
gaussdb$#    DBE_ILM_ADMIN.CUSTOMIZE_ILM(11, 1);
gaussdb$# END;
gaussdb$# /
ANONYMOUS BLOCK EXECUTE
gaussdb=# select * from gs_ilm_param;
 idx |               name                | value
-----+-----------------------------------+-------
   1 | EXECUTION_INTERVAL                |    15
   2 | RETENTION_TIME                    |    30
   7 | ENABLED                           |     1
  12 | ABS_JOBLIMIT                      |    10
  13 | JOB_SIZELIMIT                     |  1024
  14 | WIND_DURATION                     |   240
  15 | BLOCK_LIMITS                      |    40
  16 | ENABLE_META_COMPRESSION           |     0
  17 | SAMPLE_MIN                        |    10
  18 | SAMPLE_MAX                        |    10
  19 | CONST_PRIO                        |    40
  20 | CONST_THRESHOLD                   |    90
  21 | EQVALUE_PRIO                      |    60
  22 | EQVALUE_THRESHOLD                 |    80
  23 | ENABLE_DELTA_ENCODE_SWITCH        |     1
  24 | LZ4_COMPRESSION_LEVEL             |     0
  25 | ENABLE_LZ4_PARTIAL_DECOMPRESSION  |     1
  11 | POLICY_TIME                       |     1
(18 rows)
```

Step 5 Adjust the maximum amount of data that can be compressed in a single scheduling task.

In this example, the limit is changed to 4 GB.

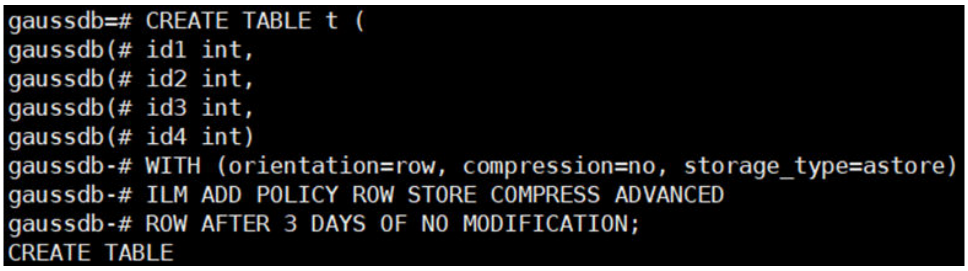
```
BEGIN
    DBE_ILM_ADMIN.CUSTOMIZE_ILM(13, 4096);
END;
/
```

Step 6 Create a table with an ILM policy.

- Method 1: Add the policy when creating the table.

  The clause **3 DAYS OF NO MODIFICATION** defines the threshold for identifying cold data.

```
CREATE TABLE t (
    id1 int,
    id2 int,
    id3 int,
    id4 int)
WITH (orientation=row, compression=no, storage_type=astore)
ILM ADD POLICY ROW STORE COMPRESS ADVANCED
ROW AFTER 3 DAYS OF NO MODIFICATION;
```



```
gaussdb=# CREATE TABLE t (
gaussdb(# id1 int,
gaussdb(# id2 int,
gaussdb(# id3 int,
gaussdb(# id4 int)
gaussdb-# WITH (orientation=row, compression=no, storage_type=astore)
gaussdb-# ILM ADD POLICY ROW STORE COMPRESS ADVANCED
gaussdb-# ROW AFTER 3 DAYS OF NO MODIFICATION;
CREATE TABLE
```

- Method 2: Create the table first, and then add the policy.

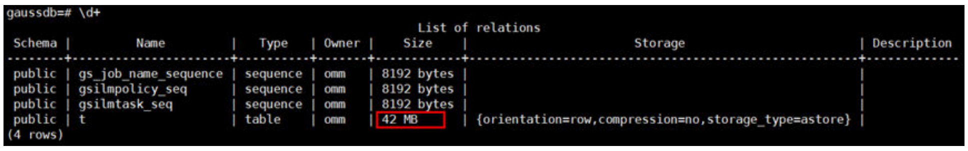
```
CREATE TABLE t (
    id1 int,
    id2 int,
    id3 int,
    id4 int)
WITH (orientation=row, compression=no, storage_type=astore);
ALTER TABLE t ILM ADD POLICY ROW STORE COMPRESS ADVANCED
ROW AFTER 3 DAYS OF NO MODIFICATION;
```

Step 7 Run the following command to insert data into the table:

```
insert into t (id1, id2 ,id3 , id4) select s, s, s, s from generate_series(1, 1000000) AS s;
```

Step 8 Run the following command to check the original size of table **t**, and record the result as **size1**. In this example, **size1** = 42 MB.

```
\d+
```



```
gaussdb=# \d+
                                          List of relations
 Schema |         Name         |   Type   | Owner |    Size    |                        Storage                        | Description
--------+----------------------+----------+-------+------------+-------------------------------------------------------+-------------
 public | gs_job_name_sequence | sequence | omm   | 8192 bytes |                                                       |
 public | gsilmpolicy_seq      | sequence | omm   | 8192 bytes |                                                       |
 public | gsilmtask_seq        | sequence | omm   | 8192 bytes |                                                       |
 public | t                    | table    | omm   | 42 MB      | {orientation=row,compression=no,storage_type=astore} |
(4 rows)
```

Step 9 Execute compression.

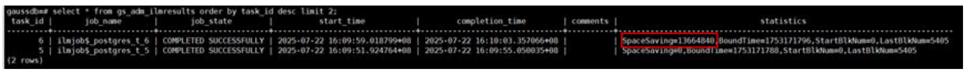
1. Run the following command:

```
declare
v_taskid number;
begin
    dbe_ilm.execute_ilm('public','t',v_taskid,NULL,'ALL POLICIES',2);
end;
/
```

2. Wait for 10 seconds and run the command in **Step 9.1** again.
3. Check the compression result.

```
select * from gs_adm_ilmresults order by task_id desc limit 2;
```

You can view the compression results in the **gs_adm_ilmresults** table. In this example, 13,664,840 bytes of space were saved.



```
gaussdb=# select * from gs_adm_ilmresults order by task_id desc limit 2;
 task_id |        job_name        |        job_state        |          start_time           |         completion_time        | comments |                            statistics
---------+------------------------+-------------------------+-------------------------------+-------------------------------+----------+--------------------------------------------------------------------------
       6 | ilmjob$_postgres_t_6 | COMPLETED SUCCESSFULLY | 2025-07-22 16:09:59.018799+08 | 2025-07-22 16:10:03.357066+08 |          | SpaceSaving=13664840,BoundTime=1753171796,StartBlkNum=0,LastBlkNum=5405
       5 | ilmjob$_postgres_t_5 | COMPLETED SUCCESSFULLY | 2025-07-22 16:09:51.924764+08 | 2025-07-22 16:09:55.050035+08 |          | SpaceSaving=0,BoundTime=1753171788,StartBlkNum=0,LastBlkNum=5405
(2 rows)
```

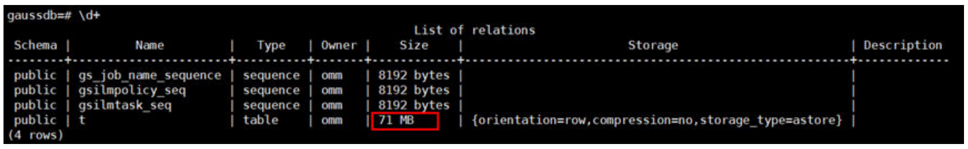
Step 10 Verify reusable space. The saved space obtained in **Step 9** may not be fully reusable. To evaluate the real compression ratio, insert the same dataset again and check the space usage of table **t**.

1. Insert data.

```
insert into t select * from t;
```

2. Check the size of table **t**, and record the result as **size2**. In this example, **size2** = 71 MB.

```
\d+
```



```
gaussdb=# \d+
                                              List of relations
 Schema |         Name          |   Type   | Owner |    Size    |                       Storage                        | Description
--------+-----------------------+----------+-------+------------+------------------------------------------------------+-------------
 public | gs_job_name_sequence | sequence | omm   | 8192 bytes |                                                      |
 public | gsilmpolicy_seq       | sequence | omm   | 8192 bytes |                                                      |
 public | gsilmtask_seq         | sequence | omm   | 8192 bytes |                                                      |
 public | t                     | table    | omm   | 71 MB      | {orientation=row,compression=no,storage_type=astore} |
(4 rows)
```

Step 11 Compare the space usage of table **t** after the first and second data insertions to calculate the compression ratio.

Compression ratio = **size1**/(**size2** – **size1**) = 42/(71 – 42) = 1.45

----End

6.3 Automatic Scheduling

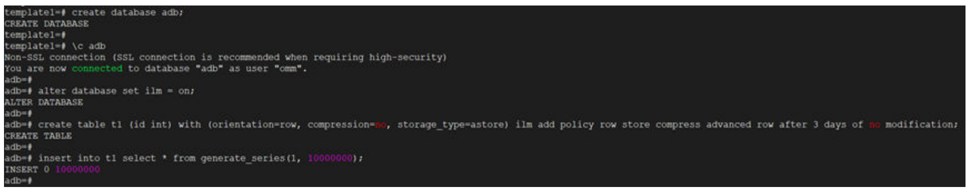
Step 1 **Log in to the GaussDB management console**. On the **Instances** page, click the name of the target instance to go to the **Basic Information** page.

Step 2 Locate the **Advanced Features** field and click **View and Modify**. On the displayed page, set the value of **Advanced compression** to **on**.

Step 3 Connect to the database. Create a data table with an ILM policy, and insert data into the table. For details about how to connect to the database, see **Using gsql to Connect to an Instance**.

```
create database adb;
\c adb
alter database set ilm = on;
create table t1 (id int) with (orientation=row, compression=no, storage_type=astore) ilm add policy row
store compress advanced row after 3 days of no modification;
insert into t1 select * from generate_series(1, 10000000);
```

The following is an example of the command output.



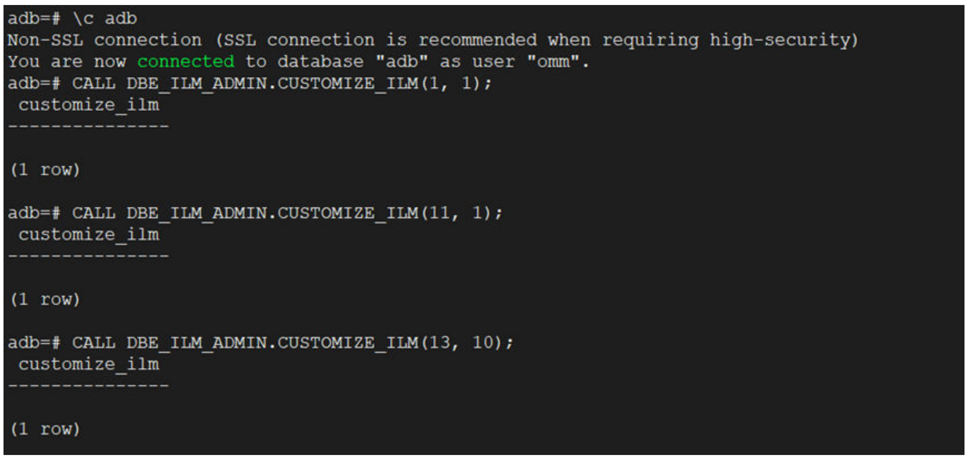
```
template1=# create database adb;
CREATE DATABASE
template1=#
template1=# \c adb
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "adb" as user "omm".
adb=#
adb=# alter database set ilm = on;
ALTER DATABASE
adb=#
adb=# create table t1 (id int) with (orientation=row, compression=no, storage_type=astore) ilm add policy row store compress advanced row after 3 days of no modification;
CREATE TABLE
adb=#
adb=# insert into t1 select * from generate_series(1, 10000000);
INSERT 0 10000000
adb=#
```

Step 4 Configure system parameters for enabling automatic ILM scheduling.

Call the **DBE_ILM_ADMIN.CUSTOMIZE_ILM** interface to modify the scheduling-related system parameters by passing the corresponding parameter IDs and values. In this example, three parameters are modified: Set the automatic scheduling frequency to once per minute; change the time unit for identifying cold data rows to seconds; set the maximum data size compressed per job to 10 MB. Retain the default values of other parameters.

```
\c adb
CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(1, 1);
CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(11, 1);
CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(13, 10);
```

The following is an example of the command output.

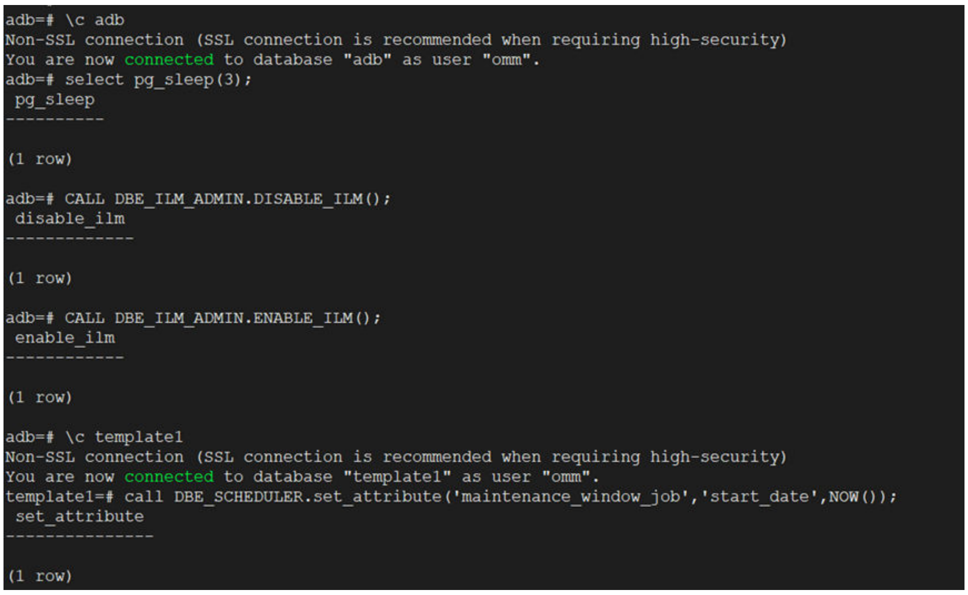


```
adb=# \c adb
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "adb" as user "omm".
adb=# CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(1, 1);
 customize_ilm
---------------

(1 row)

adb=# CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(11, 1);
 customize_ilm
---------------

(1 row)

adb=# CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(13, 10);
 customize_ilm
---------------

(1 row)
```

Step 5 Wait 3 seconds for data rows to become cold, and then enable automatic scheduling.

```
\c adb
select pg_sleep(3);
CALL DBE_ILM_ADMIN.DISABLE_ILM();
CALL DBE_ILM_ADMIN.ENABLE_ILM();
\c template1
call DBE_SCHEDULER.set_attribute('maintenance_window_job','start_date',NOW());
```

The following is an example of the command output.



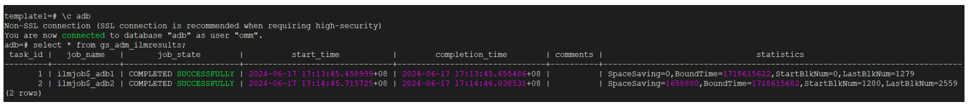
```
adb=# \c adb
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "adb" as user "omm".
adb=# select pg_sleep(3);
 pg_sleep
----------

(1 row)

adb=# CALL DBE_ILM_ADMIN.DISABLE_ILM();
 disable_ilm
-------------

(1 row)

adb=# CALL DBE_ILM_ADMIN.ENABLE_ILM();
 enable_ilm
------------

(1 row)

adb=# \c template1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "template1" as user "omm".
template1=# call DBE_SCHEDULER.set_attribute('maintenance_window_job','start_date',NOW());
 set_attribute
---------------

(1 row)
```

Step 6 Check the compression results of scheduling tasks.

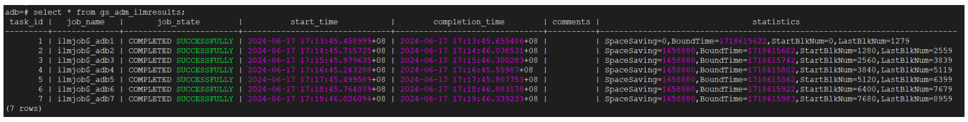
```
\c adb
select * from gs_adm_ilmresults;
```

The following is an example of the command output.

Automatic scheduling runs once per minute, so a new compression job is generated every minute. **SpaceSaving** indicates the amount of space saved by compression. A larger value indicates a greater compression benefit.



```
template1=# \c adb
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "adb" as user "omm".
adb=# select * from gs_adm_ilmresults;
 task_id |   job_name    |       job_state        |          start_time           |        completion_time        | comments |                               statistics
---------+---------------+------------------------+-------------------------------+-------------------------------+----------+-------------------------------------------------------------------------------
       1 | ilmjob$_adb1 | COMPLETED SUCCESSFULLY | 2024-06-17 17:13:45.458999+08 | 2024-06-17 17:13:45.655486+08 |          | SpaceSaving=0,BoundTime=1718615622,StartBlkNum=0,LastBlkNum=1279
       2 | ilmjob$_adb2 | COMPLETED SUCCESSFULLY | 2024-06-17 17:14:45.715725+08 | 2024-06-17 17:14:46.038531+08 |          | SpaceSaving=1658880,BoundTime=1718615682,StartBlkNum=1280,LastBlkNum=2559
(2 rows)
```

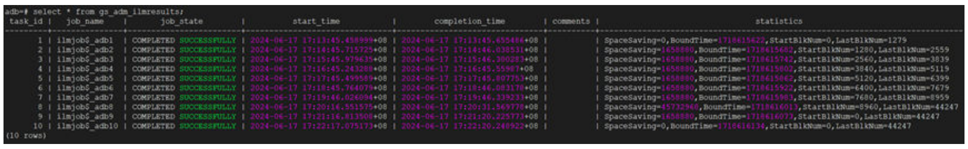


```
adb=# select * from gs_adm_ilmresults;
 task_id |   job_name    |       job_state        |          start_time           |        completion_time        | comments |                               statistics
---------+---------------+------------------------+-------------------------------+-------------------------------+----------+-------------------------------------------------------------------------------
       1 | ilmjob$_adb1 | COMPLETED SUCCESSFULLY | 2024-06-17 17:13:45.458999+08 | 2024-06-17 17:13:45.655486+08 |          | SpaceSaving=0,BoundTime=1718615622,StartBlkNum=0,LastBlkNum=1279
       2 | ilmjob$_adb2 | COMPLETED SUCCESSFULLY | 2024-06-17 17:14:45.715725+08 | 2024-06-17 17:14:46.038531+08 |          | SpaceSaving=1658880,BoundTime=1718615682,StartBlkNum=1280,LastBlkNum=2559
       3 | ilmjob$_adb3 | COMPLETED SUCCESSFULLY | 2024-06-17 17:15:45.979635+08 | 2024-06-17 17:15:46.300283+08 |          | SpaceSaving=1658880,BoundTime=1718615742,StartBlkNum=2560,LastBlkNum=3839
       4 | ilmjob$_adb4 | COMPLETED SUCCESSFULLY | 2024-06-17 17:16:45.243288+08 | 2024-06-17 17:16:45.55987+08  |          | SpaceSaving=1658880,BoundTime=1718615802,StartBlkNum=3840,LastBlkNum=5119
       5 | ilmjob$_adb5 | COMPLETED SUCCESSFULLY | 2024-06-17 17:17:45.499589+08 | 2024-06-17 17:17:45.807753+08 |          | SpaceSaving=1658880,BoundTime=1718615862,StartBlkNum=5120,LastBlkNum=6399
       6 | ilmjob$_adb6 | COMPLETED SUCCESSFULLY | 2024-06-17 17:18:45.764079+08 | 2024-06-17 17:18:46.083178+08 |          | SpaceSaving=1658880,BoundTime=1718615922,StartBlkNum=6400,LastBlkNum=7679
       7 | ilmjob$_adb7 | COMPLETED SUCCESSFULLY | 2024-06-17 17:19:46.026094+08 | 2024-06-17 17:19:46.339233+08 |          | SpaceSaving=1658880,BoundTime=1718615983,StartBlkNum=7680,LastBlkNum=8959
(7 rows)
```

If you want the table to be compressed more quickly, you can increase the data size processed by each compression job, for example, set it to **1024**. Then, restart the scheduling task and check the compression results again after one minute.

```
\c adb
CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(13, 1024);
select pg_sleep(3);
CALL DBE_ILM_ADMIN.DISABLE_ILM();
CALL DBE_ILM_ADMIN.ENABLE_ILM();
\c template1
call DBE_SCHEDULER.set_attribute('maintenance_window_job','start_date',NOW());
```

The following is an example of the command output.



```
adb=# select * from gs_adm_ilmresults;
 task_id |   job_name    |       job_state        |          start_time           |        completion_time        | comments |                               statistics
---------+---------------+------------------------+-------------------------------+-------------------------------+----------+-------------------------------------------------------------------------------
       1 | ilmjob$_adb1  | COMPLETED SUCCESSFULLY | 2024-06-17 17:13:45.458999+08 | 2024-06-17 17:13:45.655486+08 |          | SpaceSaving=0,BoundTime=1718615622,StartBlkNum=0,LastBlkNum=1279
       2 | ilmjob$_adb2  | COMPLETED SUCCESSFULLY | 2024-06-17 17:14:45.715725+08 | 2024-06-17 17:14:46.038531+08 |          | SpaceSaving=1658880,BoundTime=1718615682,StartBlkNum=1280,LastBlkNum=2559
       3 | ilmjob$_adb3  | COMPLETED SUCCESSFULLY | 2024-06-17 17:15:45.979635+08 | 2024-06-17 17:15:46.300283+08 |          | SpaceSaving=1658880,BoundTime=1718615742,StartBlkNum=2560,LastBlkNum=3839
       4 | ilmjob$_adb4  | COMPLETED SUCCESSFULLY | 2024-06-17 17:16:45.243288+08 | 2024-06-17 17:16:45.55987+08  |          | SpaceSaving=1658880,BoundTime=1718615802,StartBlkNum=3840,LastBlkNum=5119
       5 | ilmjob$_adb5  | COMPLETED SUCCESSFULLY | 2024-06-17 17:17:45.499589+08 | 2024-06-17 17:17:45.807753+08 |          | SpaceSaving=1658880,BoundTime=1718615862,StartBlkNum=5120,LastBlkNum=6399
       6 | ilmjob$_adb6  | COMPLETED SUCCESSFULLY | 2024-06-17 17:18:45.764079+08 | 2024-06-17 17:18:46.083178+08 |          | SpaceSaving=1658880,BoundTime=1718615922,StartBlkNum=6400,LastBlkNum=7679
       7 | ilmjob$_adb7  | COMPLETED SUCCESSFULLY | 2024-06-17 17:19:46.026094+08 | 2024-06-17 17:19:46.339233+08 |          | SpaceSaving=1658880,BoundTime=1718615983,StartBlkNum=7680,LastBlkNum=8959
       8 | ilmjob$_adb8  | COMPLETED SUCCESSFULLY | 2024-06-17 17:20:16.551575+08 | 2024-06-17 17:20:31.569778+08 |          | SpaceSaving=45732960,BoundTime=1718616013,StartBlkNum=8960,LastBlkNum=44247
       9 | ilmjob$_adb9  | COMPLETED SUCCESSFULLY | 2024-06-17 17:21:16.813508+08 | 2024-06-17 17:21:20.225773+08 |          | SpaceSaving=1658880,BoundTime=1718616073,StartBlkNum=0,LastBlkNum=44247
      10 | ilmjob$_adb10 | COMPLETED SUCCESSFULLY | 2024-06-17 17:22:17.075173+08 | 2024-06-17 17:22:20.248922+08 |          | SpaceSaving=0,BoundTime=1718616134,StartBlkNum=0,LastBlkNum=44247
(10 rows)
```

----End

7 Best Practices for SQL Queries

7.1 Best Practices for SQL Queries (Distributed Instances)

Based on the SQL execution mechanism and a large number of practices, SQL statements can be optimized by following certain rules to enable the database to execute SQL statements more quickly and obtain correct results.

- Replace UNION with UNION ALL.

  UNION eliminates duplicate rows while merging two result sets but UNION ALL merges the two result sets without deduplication. Deduplication takes a long time. Therefore, use UNION ALL instead of UNION if you are sure that the two result sets do not contain duplicate rows based on the service logic.
- Add NOT NULL to the **JOIN** columns.

  If there are many NULL values in the JOIN columns, you can add the filter criterion IS NOT NULL to filter data in advance to improve the JOIN efficiency.
- Convert NOT IN to NOT EXISTS.

  The NOT IN statement needs to be implemented using NESTLOOP ANTI JOIN, and the NOT EXISTS statement can be implemented using HASH ANTI JOIN. If no NULL value exists in the join columns, NOT IN is equivalent to NOT EXISTS. Therefore, if you are sure that no NULL value exists, you can convert NOT IN to NOT EXISTS to generate hash join and to improve the query performance.

  The statements for creating a foreign table are as follows:

  ```
  gaussdb=# DROP SCHEMA IF EXISTS no_in_to_no_exists_test CASCADE;
  gaussdb=# CREATE SCHEMA no_in_to_no_exists_test;
  gaussdb=# SET CURRENT_SCHEMA=no_in_to_no_exists_test;
  gaussdb=# CREATE TABLE t1(c1 int, c2 int, c3 int);
  gaussdb=# CREATE TABLE t2(d1 int, d2 int NOT NULL, d3 int);
  ```

  The statement for implementing the query using NOT IN is as follows:

  ```
  gaussdb=# SELECT * FROM t1 WHERE c1 NOT IN (SELECT d2 FROM t2);
  ```

  The plan is as follows:

  ```
  gaussdb=# EXPLAIN SELECT * FROM t1 WHERE c1 NOT IN (SELECT d2 FROM t2);
   id |                operation                | E-rows | E-width | E-costs
  ----+-----------------------------------------+--------+---------+---------
  ```

```
  1 | ->  Streaming (type: GATHER)            |     15 |      12 | 29.65
  2 |    ->  Seq Scan on t1                   |     15 |      12 | 28.77
  3 |       ->  Materialize  [2, SubPlan 1]   |    270 |       4 | 14.37
  4 |          ->  Streaming(type: BROADCAST) |     90 |       4 | 14.22
  5 |             ->  Seq Scan on t2          |     30 |       4 | 14.14
(5 rows)

 Predicate Information (identified by plan id)
-----------------------------------------------
   2 --Seq Scan on t1
         Filter: (NOT (hashed SubPlan 1))
(2 rows)
```

Because there is no null value in the **t2.d2** column (the **t2.d2** column is **NOT NULL** in the table definition), the query can be equivalently modified as follows:

```
gaussdb=# SELECT * FROM t1 WHERE NOT EXISTS (SELECT * FROM t2 WHERE t1.c1=t2.d2);
```

The generated plan is as follows:

```
gaussdb=# EXPLAIN SELECT * FROM t1 WHERE NOT EXISTS (SELECT * FROM t2 WHERE t1.c1=t2.d2);
 id |                operation                 | E-rows | E-width | E-costs
----+------------------------------------------+--------+---------+---------
  1 | ->  Streaming (type: GATHER)             |      3 |      12 | 29.99
  2 |    ->  Hash Right Anti Join (3, 5)       |      3 |      12 | 29.86
  3 |       ->  Streaming(type: REDISTRIBUTE) |     30 |       4 | 15.49
  4 |          ->  Seq Scan on t2              |     30 |       4 | 14.14
  5 |       ->  Hash                           |     29 |      12 | 14.14
  6 |          ->  Seq Scan on t1              |     30 |      12 | 14.14
(6 rows)

 Predicate Information (identified by plan id)
-----------------------------------------------
   2 --Hash Right Anti Join (3, 5)
         Hash Cond: (t2.d2 = t1.c1)
(2 rows)

-- Drop.
gaussdb=# DROP TABLE t1,t2;
gaussdb=# DROP SCHEMA IF EXISTS no_in_to_no_exists_test CASCADE;
```

- Use hashagg.

  If the GROUP BY condition exists in the query statement, the generated plan may contain sorting operations, that is, the plan contains the GroupAgg+Sort operator. As a result, the performance is poor. You can set the GUC parameter **work_mem** to increase the available memory and generate a plan with HashAgg to avoid sorting operations and improve performance. For details about how to set **work_mem**, contact the administrator.

- Replace functions with CASE statements.

  The GaussDB performance greatly deteriorates if a large number of functions are called. In this case, you can modify the pushdown functions to CASE statements.

- Do not use functions or expressions for indexes.

  Using functions or expressions for indexes will stop indexing and enable scanning on the full table.

- Do not use operator (!=, <, or >), NULL, OR, or implicit parameter conversion in WHERE clauses.

- For tables with frequent data changes, add hints to related SQL statements to fix an execution plan.

  For a table with frequent data changes, the statistics may not be the latest before the automatic ANALYZE is triggered. As a result, the execution plan may not be optimal. You are advised to add hints to related SQL statements to fix the execution plan.

- Split complex SQL statements.

  You can split an SQL statement into several ones and save the execution result to a temporary table if the SQL statement is too complex to be tuned using the solutions above, including but not limited to the following scenarios:

  – The same subquery is involved in multiple SQL statements of a job and the subquery contains a large amount of data.
  – Incorrect plan cost causes a small hash bucket of subquery. For example, the actual number of rows is 10 million, but only 1000 rows are in hash bucket.
  – Functions such as substr and to_number cause incorrect measures for subqueries containing a large amount of data.
  – BROADCAST subqueries are performed on large tables in multi-DN environment.

For more optimization methods, refer to "SQL Optimization > Typical SQL Optimization Methods" in *Developer Guide*.

7.2 Best Practices for SQL Queries (Centralized Instances)

Based on the SQL execution mechanism and a large number of practices, SQL statements can be optimized by following certain rules to enable the database to execute SQL statements more quickly and obtain correct results.

- Replace UNION with UNION ALL.

  UNION eliminates duplicate rows while merging two result sets but UNION ALL merges the two result sets without deduplication. Deduplication takes a long time. Therefore, use UNION ALL instead of UNION if you are sure that the two result sets do not contain duplicate rows based on the service logic.

- Add not null to the join columns.

  If there are many NULL values in the JOIN columns, you can add the filter criterion IS NOT NULL to filter data in advance to improve the JOIN efficiency.

- Convert NOT IN to NOT EXISTS.

  The NOT IN statement needs to be implemented using NESTLOOP ANTI JOIN, and the NOT EXISTS statement can be implemented using HASH ANTI JOIN. If no NULL value exists in the join columns, NOT IN is equivalent to NOT EXISTS. Therefore, if you are sure that no NULL value exists, you can convert NOT IN to NOT EXISTS to generate hash join and to improve the query performance.

  The statements for creating a foreign table are as follows:

  ```
  gaussdb=# DROP SCHEMA IF EXISTS no_in_to_no_exists_test CASCADE;
  gaussdb=# CREATE SCHEMA no_in_to_no_exists_test;
  gaussdb=# SET CURRENT_SCHEMA=no_in_to_no_exists_test;
  gaussdb=# CREATE TABLE t1(c1 int, c2 int, c3 int);
  gaussdb=# CREATE TABLE t2(d1 int, d2 int NOT NULL, d3 int);
  ```

  The statement for implementing the query using NOT IN is as follows:

  ```
  gaussdb=# SELECT * FROM t1 WHERE c1 NOT IN (SELECT d2 FROM t2);
  ```

The plan is as follows:

```
gaussdb=# EXPLAIN SELECT * FROM t1 WHERE c1 NOT IN (SELECT d2 FROM t2);
 id |                 operation                 | E-rows | E-width |    E-costs
----+-------------------------------------------+--------+---------+----------------
  1 | ->  Seq Scan on t1                        |    972 |      12 | 34.312..68.625
  2 |    ->  Seq Scan on t2  [1, SubPlan 1] |   1945 |       4 | 0.000..29.450
(2 rows)

 Predicate Information (identified by plan id)
-----------------------------------------------
   1 --Seq Scan on t1
         Filter: (NOT (hashed SubPlan 1))
(2 rows)
```

Because there is no null value in the **t2.d2** column (the **t2.d2** column is **NOT NULL** in the table definition), the query can be equivalently modified as follows:

```
gaussdb=# SELECT * FROM t1 WHERE NOT EXISTS (SELECT * FROM t2 WHERE t1.c1=t2.d2);
```

The generated plan is as follows:

```
gaussdb=# EXPLAIN SELECT * FROM t1 WHERE NOT EXISTS (SELECT * FROM t2 WHERE t1.c1=t2.d2);
 id |          operation          | E-rows | E-width |    E-costs
----+-----------------------------+--------+---------+----------------
  1 | ->  Hash Anti Join (2, 3) |    972 |      12 | 53.763..99.142
  2 |    ->  Seq Scan on t1      |   1945 |      12 | 0.000..29.450
  3 |    ->  Hash                |   1945 |       4 | 29.450..29.450
  4 |       ->  Seq Scan on t2  |   1945 |       4 | 0.000..29.450
(4 rows)

 Predicate Information (identified by plan id)
-----------------------------------------------
   1 --Hash Anti Join (2, 3)
         Hash Cond: (t1.c1 = t2.d2)
(2 rows)
```

- Use hashagg.

  If the GROUP BY condition exists in the query statement, the generated plan may contain sorting operations, that is, the plan contains the GroupAgg+Sort operator. As a result, the performance is poor. You can set the GUC parameter **work_mem** to increase the available memory and generate a plan with HashAgg to avoid sorting operations and improve performance. For details about how to set **work_mem**, contact the administrator.

- Replace functions with CASE statements.

  The database performance greatly deteriorates if a large number of functions are called. In this case, you can change the pushdown functions to CASE statements.

- Do not use functions or expressions for indexes.

  Using functions or expressions for indexes will stop indexing and enable scanning on the full table.

- Do not use operator (!=, <, or >), NULL, OR, or implicit parameter conversion in WHERE clauses.

- For tables with frequent data changes, add hints to related SQL statements to fix an execution plan.

  For a table with frequent data changes, the statistics may not be the latest before the automatic ANALYZE is triggered. As a result, the execution plan may not be optimal. You are advised to add hints to related SQL statements to fix the execution plan.

- Split complex SQL statements.

  You can split an SQL statement into several ones and save the execution result to a temporary table if the SQL statement is too complex to be tuned using the solutions above, including but not limited to the following scenarios:

  - The same subquery is involved in multiple SQL statements of a job and the subquery contains a large amount of data.
  - Incorrect plan cost causes a small hash bucket of subquery. For example, the actual number of rows is 10 million, but only 1000 rows are in hash bucket.
  - Functions such as substr and to_number cause incorrect measures for subqueries containing a large amount of data.
  - BROADCAST subqueries are performed on large tables in multi-DN environment.

For more optimization methods, refer to "SQL Optimization > Typical SQL Optimization Methods" in *Developer Guide*.

8 Best Practices for Permission Configuration

8.1 Best Practices for Permission Configuration (Distributed Instances)

Context

A database may be used by many users, and users are grouped into a database role for easy management. A database role can be regarded as one or a group of database users.

For databases, users and roles are basically the same. The difference is that when CREATE ROLE is used to create a role, no schema with the same name is created and the user does not have the LOGIN permission by default. When CREATE USER is used to create a user, a schema with the same name is automatically created. By default, the user has the LOGIN permission. That is, a role with the LOGIN permission can be considered to be a user. In service design, you are advised to use a role to manage permissions rather than accessing databases.

Overview

Improper permission configuration may cause permission exploitation. This section describes the functions of each permission role.

Solution

- **Database user**

  Database users are used to connect databases, access database objects, and run SQL statements. Only an existing database user can be used to connect databases. Therefore, a database administrator must plan a database user for each user who wants to connect to a database.

  Specify at least an account and the corresponding password for a database user.

  By default, database users can be classified into two types, as listed in **Table 8-1**.

Table 8-1 User types

Type	Description
Initial User	Has the highest-level database rights, that is, has all system and object permissions. The initial user is not affected by the settings of the object permissions. This is comparable to the permissions of **root** in a Unix system. For security purposes, you are advised not to operate as an initial user unless necessary. When installing or initializing a database, you can specify the initial username and password. If you do not specify the username, an initial user with the same name as the OS user who installs the database is automatically generated. If no password is specified, the initial user password is empty after the installation. You need to set the initial user password on the gsql client before performing other operations. Note: For security purposes, remote login to GaussDB Kernel in trust mode is prohibited for all users, and remote login in any mode is prohibited for the initial user.
Common User	By default, a user can access the default database system catalogs (excluding pg_authid, pg_largeobject, pg_user_status, and pg_auth_history) and views and connect to the default database **postgres**, as well as the objects in the public schema, including tables, views, and functions. ● You can run CREATE USER and ALTER USER to specify system permissions, or run GRANT ALL PRIVILEGE to grant the SYSADMIN permission. ● You can run the GRANT statement to assign object permissions to a common user. ● The user can run the GRANT statement to assign other user permissions to a common user.

- **Database permission types**

  Permissions and roles work together to specify accessible data and executable SQL statements. For details, see **Table 8-2**.

  System permissions are specified by using the CREATE USER/ALTER USER and CREATE ROLE/ALTER ROLE statements and cannot be inherited from roles. The SYSADMIN permission can be granted or revoked by using the GRANT/REVOKE ALL PRIVILEGES statement.

Table 8-2 Permission types

Type	Description
System permission	System permissions are also regarded as user attributes, which can be specified when a user is being created or modified. System permissions include SYSADMIN, MONADMIN, OPRADMIN, POLADMIN, CREATEDB, CREATEROLE, AUDITADMIN, and LOGIN. They can be specified only by the CREATE USER or ALTER USER statement. System permissions except SYSADMIN cannot be granted or revoked by the GRANT or REVOKE statement. In addition, system permissions cannot be inherited from roles.
Object permission	Object permissions are operation permissions for tables, views, indexes, sequences, and functions. These permissions include SELECT, INSERT, UPDATE, and DELETE. Only an object owner or SYSADMIN can use the GRANT/REVOKE statement to grant or revoke object permissions.
Role	A role is a group of permissions. If a role consists of system permissions, these permissions cannot be granted to other users or roles. If a role consists of object permissions, these permissions can be granted to other users or roles.

- **Database permission model**
  - **System permission model**
    - **Default permission mechanism**

Figure 8-1 Permission architecture

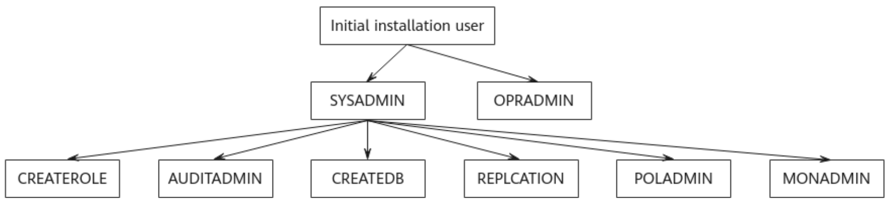


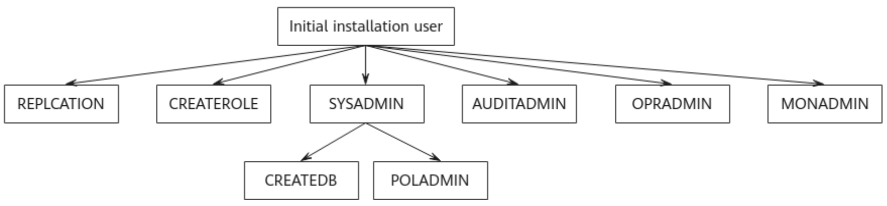
Figure 8-1 shows the permission architecture. In the default permission mechanism, the SYSADMIN has most permissions.

- **Initial installation user**: an account automatically generated during cluster installation. This account has the highest permissions in the system and can perform all operations.
- **SYSADMIN**: system administrator permissions, which are only inferior to those of the initial installation user. By default, the system administrator has the same permissions as the object owner, excluding the MONADMIN and OPRADMIN permissions. However, SYSADMIN can grant the MONADMIN permissions to itself.
- **MONADMIN**: monitor administrator permissions, including the permissions to access and grant views and functions in the monitor schema DBE_PERF.

- **OPRADMIN**: O&M administrator permissions, including the permission to use Roach to perform backup and restoration.
- **CREATEROLE**: security administrator permissions, including the permissions to create, modify, and delete users and roles.
- **AUDITADMIN**: audit administrator permissions, including the permissions to view and maintain database audit logs.
- **CREATEDB**: permission to create databases.
- **POLADMIN**: security policy administrator permissions, including the permissions to create resource labels, dynamic data masking policies, and unified audit policies.

- **Separation of duties**

Figure 8-2 Separation of duties



- **SYSADMIN**: system administrator permission. The user with this attribute no longer has the permissions to create, modify, delete users or roles, or view or maintain database audit logs.
- **CREATEROLE**: security administrator permissions, including the permissions to create, modify, and delete users and roles.
- **AUDITADMIN**: audit administrator permissions, including the permissions to view and maintain database audit logs.
- A user or role can only have the system permissions of either SYSADMIN, CREATEROLE, o AUDITADMIN.

– **Object permission model**

- Object permissions refer to the permissions to perform operations for database objects (such as databases, schemas, and tables), including SELECT, INSERT, UPDATE, DELETE, and CONNECT.
- The permissions vary by object. Object permissions can be granted to users or roles.
- You can use GRANT or REVOKE to grant permissions to a user or revoke them from the user. Object permissions can be inherited by a role.

– **Role permission model**

GaussDB Kernel provides a group of default roles whose names start with **gs_role_**. These roles are provided to access to specific, typically high-privileged operations. You can grant these roles to other users or roles within the database so that they can use specific functions. These roles should be given with great care to ensure that they are used where they are needed. **Table 8-3** describes the permissions of built-in roles.

Table 8-3 Permissions of built-in roles

Role	Permission
gs_role_signal_backend	Permission to call the pg_cancel_backend(), pg_terminate_backend(), and pg_terminate_session() functions to cancel or terminate other sessions. However, this role cannot perform operations on sessions of the initial user or users with the PERSISTENCE attribute.
gs_role_tablespace	Permission to create a tablespace.
gs_role_replication	Permission to call logical replication functions, such as kill_snapshot(), pg_create_logical_replication_slot(), pg_create_physical_replication_slot(), pg_drop_replication_slot(), pg_replication_slot_advance(), pg_create_physical_replication_slot_extern(), pg_logical_slot_get_changes(), pg_logical_slot_peek_changes(), pg_logical_slot_get_binary_changes(), and pg_logical_slot_peek_binary_changes().
gs_role_account_lock	Permission to lock and unlock users. However, this role cannot lock or unlock the initial user or users with the PERSISTENCE attribute.
gs_role_pldebugger	Permission to debug functions in dbe_pldebugger.
gs_role_public_dblink_drop	Permission to delete public database links.
gs_role_public_dblink_alter	Permission to modify public database links.
gs_role_seclabel	Permission to create, delete, and apply security labels.
gs_role_public_synonym_create	Permission to create public synonyms.
gs_role_public_synonym_drop	Permission to drop public synonyms.

- System permission configuration
  - **Configuring the default permission mechanism**
    - **Initial user**

      The account automatically generated during database installation is called an initial user. The initial user is also the SYSADMIN, MONADMIN, OPRADMIN, and POLADMIN. It has the highest permissions in the system and can perform all operations. If the initial username is not specified during installation, the username is the same as the name of the OS user who installs the database. If the password of the initial user is not specified during the installation, the password is empty after the installation. In this case, you need to change the password of the initial user on the gsql client before performing other operations. If the initial user password is empty, you cannot perform other SQL operations, such as upgrade, capacity expansion, and node replacement, except changing the password.

An initial user bypasses all permission checks. You are advised to use the initial user as a database administrator only for database management other than service running.

- **SYSADMIN**

```
gaussdb=#CREATE USER u_sysadmin WITH SYSADMIN password '********';

-- Alternatively, run the following SQL statement when the user already exists:
gaussdb=#ALTER USER u_sysadmin01 SYSADMIN;
```

- **MONADMIN**

```
gaussdb=#CREATE USER u_monadmin WITH MONADMIN password '********';

-- Alternatively, run the following SQL statement when the user already exists:
gaussdb=#ALTER USER u_monadmin01 MONADMIN;
```

- **OPRADMIN**

```
gaussdb=#CREATE USER u_opradmin WITH OPRADMIN password "xxxxxxxxx";

-- Alternatively, run the following SQL statement when the user already exists:
gaussdb=#ALTER USER u_opradmin01 OPRADMIN;
```

- **POLADMIN**

```
gaussdb=#CREATE USER u_poladmin WITH POLADMIN password "xxxxxxxxx";

-- Alternatively, run the following SQL statement when the user already exists:
gaussdb=#ALTER USER u_poladmin01 POLADMIN;
```

– **Configuring the separation of duties**

To configure this mode, you need to set the GUC parameter **enableSeparationOfDuty** to **on**. This is a POSTMASTER parameter. After this parameter is modified, you need to restart the database.

```
gs_guc set -Z coordinator -Z datanode -N all -I all -c "enableSeparationOfDuty=on"
gs_om -t stop
gs_om -t start
```

The syntax for creating and configuring user permissions is the same as that for default permissions.

- Role permission configuration

```
-- Create the database test.
gaussdb=#CREATE DATABASE test;
-- Create role1 and user1.
gaussdb=#CREATE ROLE role1 PASSWORD '********';
gaussdb=#CREATE USER user1 PASSWORD '********';
-- Grant the CREATE ANY TABLE permission to role1.
gaussdb=#GRANT CREATE ON DATABASE test TO role1;

-- If role1 is assigned to user1, user1 belongs to group role1 and inherits the permissions of role1 to
create schemas in the database test.
gaussdb=#GRANT role1 TO user1;

-- Query user and role information.
```

```
gaussdb=#\du role1|user1;
           List of roles
 Role name |  Attributes  | Member of
-----------+--------------+-----------
 role1     | Cannot login | {}
 user1     |              | {role1}
```

Practice Effect

None.

8.2 Best Practices for Permission Configuration (Centralized Instances)

Context

A database may be used by many users, and users are grouped into a database role for easy management. A database role can be regarded as one or a group of database users.

For databases, users and roles are basically the same. The difference is that when CREATE ROLE is used to create a role, no schema with the same name is created and the user does not have the LOGIN permission by default. When CREATE USER is used to create a user, a schema with the same name is automatically created. By default, the user has the LOGIN permission. That is, a role with the LOGIN permission can be considered to be a user. In service design, you are advised to use a role to manage permissions rather than accessing databases.

Overview

Improper permission configuration may cause permission exploitation. This section describes the functions of each permission role.

Solution

- **Database user**

  Database users are used to connect databases, access database objects, and run SQL statements. Only an existing database user can be used to connect databases. Therefore, a database administrator must plan a database user for each user who wants to connect to a database.

  Specify at least an account and the corresponding password for a database user.

  By default, database users can be classified into two types, as listed in **Table 8-4**.

Table 8-4 User types

Type	Description
Initial User	Has the highest-level database rights, that is, has all system and object permissions. The initial user is not affected by the settings of the object permissions. This is comparable to the permissions of **root** in a Unix system. For security purposes, you are advised not to operate as an initial user unless necessary. When installing or initializing a database, you can specify the initial username and password. If you do not specify the username, an initial user with the same name as the OS user who installs the database is automatically generated. If no password is specified, the initial user password is empty after the installation. You need to set the initial user password on the gsql client before performing other operations. Note: For security purposes, remote login to GaussDB Kernel in trust mode is prohibited for all users, and remote login in any mode is prohibited for the initial user.
Common User	By default, a user can access the default database system catalogs (excluding pg_authid, pg_largeobject, pg_user_status, and pg_auth_history) and views and connect to the default database **postgres**, as well as the objects in the public schema, including tables, views, and functions. ● You can run CREATE USER and ALTER USER to specify system permissions, or run GRANT ALL PRIVILEGE to grant the SYSADMIN permission. ● You can run the GRANT statement to assign object permissions to a common user. ● The user can run the GRANT statement to assign other user permissions to a common user.

- **Database permission types**

  Permissions and roles work together to specify accessible data and executable SQL statements. For details, see **Table 8-5**.

  System permissions are specified by using the CREATE USER/ALTER USER and CREATE ROLE/ALTER ROLE statements and cannot be inherited from roles. The SYSADMIN permission can be granted or revoked by using the GRANT/REVOKE ALL PRIVILEGES statement.

Table 8-5 Permission types

Type	Description
System permission	System permissions are also regarded as user attributes, which can be specified when a user is being created or modified. System permissions include SYSADMIN, MONADMIN, OPRADMIN, POLADMIN, CREATEDB, CREATEROLE, AUDITADMIN, and LOGIN. They can be specified only by the CREATE USER or ALTER USER statement. System permissions except SYSADMIN cannot be granted or revoked by the GRANT or REVOKE statement. In addition, system permissions cannot be inherited from roles.
Object permission	Object permissions are operation permissions for tables, views, indexes, sequences, and functions. These permissions include SELECT, INSERT, UPDATE, and DELETE. Only an object owner or SYSADMIN can use the GRANT/REVOKE statement to grant or revoke object permissions.
Role	A role is a group of permissions. If a role consists of system permissions, these permissions cannot be granted to other users or roles. If a role consists of object permissions, these permissions can be granted to other users or roles.

- **Database permission model**
  - **System permission model**
    - **Default permission mechanism**

Figure 8-3 Permission architecture

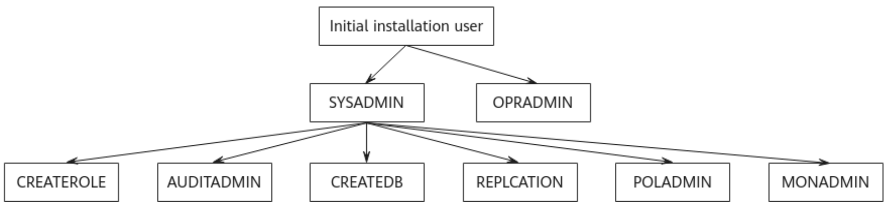


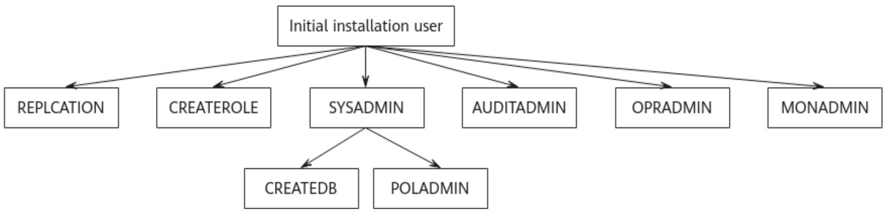
Figure 8-3 shows the permission architecture. In the default permission mechanism, the SYSADMIN has most permissions.

- **Initial installation user**: an account automatically generated during cluster installation. This account has the highest permissions in the system and can perform all operations.
- **SYSADMIN**: system administrator permissions, which are only inferior to those of the initial installation user. By default, the system administrator has the same permissions as the object owner, excluding the MONADMIN and OPRADMIN permissions. However, SYSADMIN can grant the MONADMIN permissions to itself.
- **MONADMIN**: monitor administrator permissions, including the permissions to access and grant views and functions in the monitor schema DBE_PERF.

- **OPRADMIN**: O&M administrator permissions, including the permission to use Roach to perform backup and restoration.
- **CREATEROLE**: security administrator permissions, including the permissions to create, modify, and delete users and roles.
- **AUDITADMIN**: audit administrator permissions, including the permissions to view and maintain database audit logs.
- **CREATEDB**: permission to create databases.
- **POLADMIN**: security policy administrator permissions, including the permissions to create resource labels, dynamic data masking policies, and unified audit policies.

- **Separation of duties**

Figure 8-4 Separation of duties



- **SYSADMIN**: system administrator permission. The user with this attribute no longer has the permissions to create, modify, delete users or roles, or view or maintain database audit logs.
- **CREATEROLE**: security administrator permissions, including the permissions to create, modify, and delete users and roles.
- **AUDITADMIN**: audit administrator permissions, including the permissions to view and maintain database audit logs.
- A user or role can only have the system permissions of either SYSADMIN, CREATEROLE, o AUDITADMIN.

– **Object permission model**

- Object permissions refer to the permissions to perform operations for database objects (such as databases, schemas, and tables), including SELECT, INSERT, UPDATE, DELETE, and CONNECT.
- The permissions vary by object. Object permissions can be granted to users or roles.
- You can use GRANT or REVOKE to grant permissions to a user or revoke them from the user. Object permissions can be inherited by a role.

– **Role permission model**

GaussDB Kernel provides a group of default roles whose names start with **gs_role_**. These roles are provided to access to specific, typically high-privileged operations. You can grant these roles to other users or roles within the database so that they can use specific functions. These roles should be given with great care to ensure that they are used where they are needed. **Table 8-6** describes the permissions of built-in roles.

Table 8-6 Permissions of built-in roles

Role	Permission
gs_role_signal_backend	Permission to call the pg_cancel_backend(), pg_terminate_backend(), and pg_terminate_session() functions to cancel or terminate other sessions. However, this role cannot perform operations on sessions of the initial user or users with the PERSISTENCE attribute.
gs_role_tablespace	Permission to create a tablespace.
gs_role_replication	Permission to call logical replication functions, such as kill_snapshot(), pg_create_logical_replication_slot(), pg_create_physical_replication_slot(), pg_drop_replication_slot(), pg_replication_slot_advance(), pg_create_physical_replication_slot_extern(), pg_logical_slot_get_changes(), pg_logical_slot_peek_changes(), pg_logical_slot_get_binary_changes(), and pg_logical_slot_peek_binary_changes().
gs_role_account_lock	Permission to lock and unlock users. However, this role cannot lock or unlock the initial user or users with the PERSISTENCE attribute.
gs_role_pldebugger	Permission to debug functions in dbe_pldebugger.
gs_role_public_dblink_drop	Permission to delete public database links.
gs_role_public_dblink_alter	Permission to modify public database links.
gs_role_seclabel	Permission to create, delete, and apply security labels.
gs_role_public_synonym_create	Permission to create public synonyms.
gs_role_public_synonym_drop	Permission to drop public synonyms.
gs_role_pdb_create	Permission to create a PDB.

- System permission configuration
  - **Configuring the default permission mechanism**

- **Initial user**

  The account automatically generated during database installation is called an initial user. The initial user is also the SYSADMIN, MONADMIN, OPRADMIN, and POLADMIN. It has the highest permissions in the system and can perform all operations. If the initial username is not specified during installation, the username is the same as the name of the OS user who installs the database. If the password of the initial user is not specified during the installation, the password is empty after the installation. In this case, you need to change the password of the initial user on the gsql client before performing other operations. If the initial user password is empty, you cannot perform other SQL operations, such as upgrade, capacity expansion, and node replacement, except changing the password.

  An initial user bypasses all permission checks. You are advised to use the initial user as a database administrator only for database management other than service running.

- **SYSADMIN**

```
gaussdb=#CREATE USER u_sysadmin WITH SYSADMIN password '********';

-- Alternatively, run the following SQL statement when the user already exists:
gaussdb=#ALTER USER u_sysadmin01 SYSADMIN;
```

- **MONADMIN**

```
gaussdb=#CREATE USER u_monadmin WITH MONADMIN password '********';

-- Alternatively, run the following SQL statement when the user already exists:
gaussdb=#ALTER USER u_monadmin01 MONADMIN;
```

- **OPRADMIN**

```
gaussdb=#CREATE USER u_opradmin WITH OPRADMIN password "xxxxxxxxx";

-- Alternatively, run the following SQL statement when the user already exists:
gaussdb=#ALTER USER u_opradmin01 OPRADMIN;
```

- **POLADMIN**

```
gaussdb=#CREATE USER u_poladmin WITH POLADMIN password "xxxxxxxxx";

-- Alternatively, run the following SQL statement when the user already exists:
gaussdb=#ALTER USER u_poladmin01 POLADMIN;
```

– **Configuring the separation of duties**

To configure this mode, you need to set the GUC parameter **enableSeparationOfDuty** to **on**. This is a POSTMASTER parameter. After this parameter is modified, you need to restart the database.

```
gs_guc set -Z datanode -N all -I all -c "enableSeparationOfDuty=on"
gs_om -t stop
gs_om -t start
```

The syntax for creating and configuring user permissions is the same as that for default permissions.

- Role permission configuration

```
-- Create the database test.
gaussdb=#CREATE DATABASE test;
-- Create role1 and user1.
gaussdb=#CREATE ROLE role1 PASSWORD '********';
gaussdb=#CREATE USER user1 PASSWORD '********';
-- Grant the CREATE ANY TABLE permission to role1.
gaussdb=#GRANT CREATE ON DATABASE test TO role1;
```

```
-- If role1 is assigned to user1, user1 belongs to group role1 and inherits the permissions of role1 to
create schemas in the database test.
gaussdb=#GRANT role1 TO user1;

-- Query user and role information.
gaussdb=#\du role1|user1;
           List of roles
 Role name |  Attributes  | Member of
-----------+--------------+-----------
 role1     | Cannot login | {}
 user1     |              | {role1}
```

Practice Effect

None.

9 Best Practices for Data Skew Query (Distributed Instances)

9.1 Quickly Locating Tables That Cause Data Skew

Currently, you can choose from the following ways based on your service needs to query data skew: the table_distribution(schemaname text, tablename text) function, the table_distribution() function, and the PGXC_GET_TABLE_SKEWNESS view. For further details, refer to the corresponding function and view sections in *Developer Guide*.

Scenario 1: Data Skew Caused by a Full Disk

First, use the pg_stat_get_last_data_changed_time(oid) function to identify tables with recent data changes. Since the last modification time of tables is recorded only on the CN where INSERT, UPDATE, and DELETE operations are executed, you can use the following encapsulated function to pinpoint tables that were modified within the past day (adjustable in the function):

```
gaussdb=# CREATE OR REPLACE FUNCTION get_last_changed_table(OUT schemaname text, OUT relname
text)
RETURNS setof record
AS $$
DECLARE
 row_data record;
 row_name record;
 query_str text;
 query_str_nodes text;
 BEGIN
  query_str_nodes := 'SELECT node_name FROM pgxc_node where node_type = ''C''';
  FOR row_name IN EXECUTE(query_str_nodes) LOOP
   query_str := 'EXECUTE DIRECT ON (' || row_name.node_name || ') ''SELECT b.nspname,a.relname FROM
pg_class a INNER JOIN pg_namespace b on a.relnamespace = b.oid where
pg_stat_get_last_data_changed_time(a.oid) BETWEEN current_timestamp - 1 AND current_timestamp;''';
    FOR row_data IN EXECUTE(query_str) LOOP
     schemaname = row_data.nspname;
     relname = row_data.relname;
     return next;
    END LOOP;
  END LOOP;
  return;
 END; $$
LANGUAGE 'plpgsql';
```

Then execute table_distribution(schemaname text, tablename text) to query the storage space occupied by the tables on each DN.

```
gaussdb=# SELECT table_distribution(schemaname,relname) FROM get_last_changed_table();
```

Scenario 2: Routine Data Skew Inspection

- If the number of tables in the database is less than 10,000, use the skew view to query data skew of all tables in the database.

  ```
  gaussdb=#SELECT * FROM pgxc_get_table_skewness ORDER BY totalsize DESC;
  ```

- When there are a substantial number of tables (at least over 10,000) in the database, using the PGXC_GET_TABLE_SKEWNESS view involves thorough skewness calculations across the entire database, potentially requiring a significant amount of time (hours). To optimize the calculations and reduce the output columns, you can use the table_distribution() function to customize the output based on the definition of the PGXC_GET_TABLE_SKEWNESS view.

  ```
  gaussdb=#SELECT schemaname,tablename,max(dnsize) AS maxsize, min(dnsize) AS minsize
  FROM pg_catalog.pg_class c
  INNER JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
  INNER JOIN pg_catalog.table_distribution() s ON s.schemaname = n.nspname AND s.tablename =
  c.relname
  INNER JOIN pg_catalog.pgxc_class x ON c.oid = x.pcrelid AND x.pclocatortype = 'H'
  GROUP BY schemaname,tablename;
  ```

10 Best Practices for Stored Procedures

10.1 Best Practices for Stored Procedures (Distributed Instances)

In GaussDB, business rules and logics are saved as stored procedures.

A stored procedure is a combination of SQL, PL/SQL, and Java statements. Stored procedures can move the code that executes business rules from applications to databases. Therefore, the code storage can be used by multiple programs at a time.

For the basic usage of stored procedures, refer to the "Stored Procedures" section in *Developer Guide*.

10.1.1 Permission Management

By default, stored procedures are granted the SECURITYINVOKER permission. To change this default to the SECURITYDEFINER permission, set the GUC parameter **behavior_compat_options** to **'plsql_security_definer'**. For details about permissions, see "SQL Reference > SQL Syntax > C > CREATE FUNCTION" in *Developer Guide*.

Improper permission mode may cause unauthorized access to sensitive data or unauthorized resource operations. Therefore, select and configure the permission mode with caution to ensure system security.

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA
-- Create two different users.
gaussdb=# CREATE USER test_user1 PASSWORD '********';
CREATE ROLE
gaussdb=# CREATE USER test_user2 PASSWORD '********';
CREATE ROLE
-- Set the permissions of the two users on schema best_practices_for_procedure.
gaussdb=# GRANT usage, create ON SCHEMA best_practices_for_procedure TO test_user1;
GRANT
gaussdb=# GRANT usage, create ON SCHEMA best_practices_for_procedure TO test_user2;
GRANT
-- Switch to the test_user1 user and create a table and a stored procedure.
gaussdb=# SET SESSION AUTHORIZATION test_user1 PASSWORD '********';
SET
```

```
gaussdb=> CREATE TABLE best_practices_for_procedure.user1_tb (a int, b int);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=> CREATE OR REPLACE PROCEDURE best_practices_for_procedure.user1_proc() AS
  BEGIN
    INSERT INTO best_practices_for_procedure.user1_tb VALUES(1,1);
  END;
/
CREATE PROCEDURE
-- Switch to the test_user2 user to execute the stored procedure created by the test_user1 user. An error is reported, indicating that the user does not have the permission on the user1_tb table because stored procedures are executed with the caller's permissions by default.
gaussdb=> RESET SESSION AUTHORIZATION;
RESET
gaussdb=# SET SESSION AUTHORIZATION test_user2 PASSWORD '********';
SET
gaussdb=> CALL best_practices_for_procedure.user1_proc();
ERROR:  Permission denied for relation user1_tb.
DETAIL:  N/A.
CONTEXT:  SQL statement "insert into best_practices_for_procedure.user1_tb values(1,1)"
PL/pgSQL function best_practices_for_procedure.user1_proc() line 3 at SQL statement
-- Set the GUC parameter to use the creator's permissions by default when creating stored procedures.
gaussdb=> SET behavior_compat_options='plsql_security_definer';
SET
-- Switch to the test_user1 user and re-create the stored procedure.
gaussdb=> RESET SESSION AUTHORIZATION;
RESET
gaussdb=# SET SESSION AUTHORIZATION test_user1 password '********';
SET
gaussdb=> CREATE OR REPLACE PROCEDURE best_practices_for_procedure.user1_proc() AS
  BEGIN
    INSERT INTO best_practices_for_procedure.user1_tb VALUES(1,1);
  END;
/
CREATE PROCEDURE
-- Switch to the test_user2 user and execute the stored procedure. The execution is successful.
gaussdb=> RESET SESSION AUTHORIZATION;
RESET
gaussdb=# SET SESSION AUTHORIZATION test_user2 PASSWORD '********';
SET
gaussdb=> CALL best_practices_for_procedure.user1_proc();
 user1_proc
------------

(1 row)

-- Switch to the test_user1 user and view the table content.
gaussdb=> RESET SESSION AUTHORIZATION;
RESET
gaussdb=# SET SESSION AUTHORIZATION test_user1 PASSWORD '********';
SET
gaussdb=> SELECT * FROM best_practices_for_procedure.user1_tb;
 a | b
---+---
 1 | 1
(1 row)

-- Clean the environment.
gaussdb=> RESET behavior_compat_options;
RESET
gaussdb=> RESET SESSION AUTHORIZATION;
RESET
gaussdb=# DROP SCHEMA best_practices_for_procedure CASCADE;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.user1_tb
drop cascades to function best_practices_for_procedure.user1_proc()
DROP SCHEMA
gaussdb=# DROP USER test_user1;
```

```
DROP ROLE
gaussdb=# DROP USER test_user2;
DROP ROLE
```

10.1.2 Naming Convention

Improper stored procedure and variable naming may adversely affect system usage.

- The name of a stored procedure, variable, or type can contain a maximum of 63 characters. If this limit is exceeded, the name is automatically truncated to 63 characters.

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

-- When a stored procedure name containing 66 characters is created, a message is displayed,
indicating that the name is truncated to 63 characters.
gaussdb=# CREATE OR REPLACE PROCEDURE
best_practices_for_procedure.abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz0123456789101
1() AS
BEGIN
    NULL;
END;
/
NOTICE:  identifier "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891011" will
be truncated to "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891"
CREATE PROCEDURE

-- When a variable name containing 66 characters is created, a message is displayed, indicating that
the name is truncated to 63 characters.
gaussdb=# CREATE OR REPLACE PROCEDURE
best_practices_for_procedure.proc1(abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz0123456
7891011 int) AS
BEGIN
    NULL;
END;
/
NOTICE:  identifier "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891011" will
be truncated to "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891"
CREATE PROCEDURE

gaussdb=# DROP SCHEMA best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to function
best_practices_for_procedure.abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891()
drop cascades to function best_practices_for_procedure.proc1(integer)
DROP SCHEMA
```

- When creating a stored procedure, avoid using variables or types with the same name in different variable scopes. For details, see "Stored Procedures > Basic Statements > Variable Definition Statements > Scope of a Variable" in *Developer Guide*. Using variables and types with the same name in different variable scopes may reduce the readability of stored procedures and increase the maintenance difficulty.

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

-- Create a stored procedure, create the same variable name in different variable scopes, and assign
values.
gaussdb=# CREATE OR REPLACE PROCEDURE best_practices_for_procedure.proc1() AS
    name varchar2(10) := 'outer';
    age int := 2025;
BEGIN
    DECLARE
    name varchar2(10) := 'inner'; -- This is only an example and is not recommended.
    age int := 2024; -- This is only an example and is not recommended.
    BEGIN
        dbe_output.print_line('inner name =' || name);
        dbe_output.print_line('inner age =' || age);
    END;
    dbe_output.print_line('outer name =' || name);
    dbe_output.print_line('outer age =' || age);
END;
/
CREATE PROCEDURE

-- Execute the stored procedure. The same variable name in different scopes actually refers to
different variables.
gaussdb=# CALL best_practices_for_procedure.proc1();
inner name =inner
inner age =2024
outer name =outer
outer age =2025
 proc1
-------

(1 row)

gaussdb=# DROP SCHEMA best_practices_for_procedure cascade;
NOTICE:  drop cascades to function best_practices_for_procedure.proc1()
DROP SCHEMA
```

- Do not use SQL keywords in stored procedure, internal variable, and data type names to ensure that the stored procedure can run properly in all scenarios.

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

gaussdb=# cREATE OR REPLACE PROCEDURE best_practices_for_procedure."as"() AS -- This is only an
example and is not recommended.
BEGIN
    NULL;
END;
/
CREATE PROCEDURE

-- A direct call will result in an error.
gaussdb=# CALL as();
ERROR:  syntax error at or near "as"
LINE 1: call as();
             ^
gaussdb=# CALL best_practices_for_procedure."as"();
 as
----

(1 row)

gaussdb=# DROP SCHEMA best_practices_for_procedure cascade;
NOTICE:  drop cascades to function best_practices_for_procedure."as"()
DROP SCHEMA
```

- When creating a stored procedure, avoid using the same name as system functions to prevent confusion. If the same name must be used, specify the schema during a call.

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

-- Create an abs function with the same name as the abs system function in the schema. This is only
an example and is not recommended.
gaussdb=# CREATE OR REPLACE FUNCTION best_practices_for_procedure.abs(a int) RETURN int AS
BEGIN
    dbe_output.print_line('my abs funciton.');
    RETURN abs(a);
END;
/
CREATE FUNCTION
```

```
-- Call a stored procedure. If no schema is added, the abs system function is called.
gaussdb=# CALL abs(-1);
 abs
-----
   1
(1 row)


-- You are advised to add a schema.
gaussdb=# CALL best_practices_for_procedure.abs(-1);
my abs funciton.
 abs
-----
   1
(1 row)

gaussdb=# DROP SCHEMA best_practices_for_procedure cascade;
NOTICE:  drop cascades to function best_practices_for_procedure.abs(integer)
DROP SCHEMA
```

10.1.3 Access Object

If no schema is specified for a stored procedure, the stored procedure searches for objects based on the sequence specified by **SEARCH_PATH**. As a result, unexpected objects may be accessed. If tables, stored procedures, and other database objects with the same name exist in different schemas, unexpected results may occur if the schema is not specified. Therefore, it is recommended that you always explicitly specify a schema when a stored procedure accesses a data object.

Example:

```
-- Create two different schemas.
gaussdb=# CREATE SCHEMA best_practices_for_procedure1;
CREATE SCHEMA
gaussdb=# CREATE SCHEMA best_practices_for_procedure2;
CREATE SCHEMA

-- Create the same stored procedure in two different schemas.
gaussdb=# CREATE OR REPLACE PROCEDURE best_practices_for_procedure1.proc1() as
BEGIN
    dbe_output.print_line('in schema best_practices_for_procedure1');
END;
/
CREATE PROCEDURE

gaussdb=# CREATE OR REPLACE procedure best_practices_for_procedure2.proc1() as
BEGIN
    dbe_output.print_line('in schema best_practices_for_procedure2');
END;
/
CREATE PROCEDURE

-- Calling the same stored procedure with different search_path settings may lead to differences.
gaussdb=# SET search_path TO best_practices_for_procedure1, best_practices_for_procedure2;
SET
gaussdb=# CALL proc1();
in schema best_practices_for_procedure1
 proc1
-------

(1 row)

gaussdb=# RESET search_path;
RESET
gaussdb=# SET search_path TO best_practices_for_procedure2, best_practices_for_procedure1;
```

```
SET
gaussdb=# CALL proc1();
in schema best_practices_for_procedure2
 proc1
-------

(1 row)

gaussdb=# RESET search_path;
RESET

gaussdb=# DROP SCHEMA best_practices_for_procedure1 cascade;
NOTICE:  drop cascades to function best_practices_for_procedure1.proc1()
DROP SCHEMA

gaussdb=# DROP SCHEMA best_practices_for_procedure2 cascade;
NOTICE:  drop cascades to function best_practices_for_procedure2.proc1()
DROP SCHEMA
```

10.1.4 Statement Functions

10.1.4.1 Package Variables

A package variable is a global variable defined in a package. Its lifecycle covers the entire database session. Improper use may cause the following problems:

- If a variable is completely transparent to users who have the package access permission, the variable may be shared among multiple stored procedures and modified unexpectedly.
- The lifecycle of a package variable is at the session level. Improper operations may cause residual data and affect other stored procedures.
- Caching a large number of package variables in a session may consume a substantial amount of memory.

Therefore, you are advised to use package variables with caution and ensure that their access and lifecycle are properly managed.

```
-- Create an ORA-compatible database.
gaussdb=# CREATE DATABASE db_test DBCOMPATIBILITY 'ORA';

-- Switch to the ORA-compatible database.
gaussdb=# \c db_test
db_test=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

db_test=# CREATE OR REPLACE PACKAGE best_practices_for_procedure.pkg1 AS
    id int;
    name varchar2(20);
    arg int;
    procedure p1();
END pkg1;
/
CREATE PACKAGE

db_test=# CREATE OR REPLACE PACKAGE BODY best_practices_for_procedure.pkg1 AS
    procedure p1() as
    BEGIN
        id := 1;
        name := 'huawei';
    arg := 20;
    END;
END pkg1;
/
```

```
CREATE PACKAGE BODY

-- Create a stored procedure and modify package variables.
db_test=# CREATE OR REPLACE PROCEDURE best_practices_for_procedure.pro1 () AS
BEGIN
    best_practices_for_procedure.pkg1.id := 2;
    best_practices_for_procedure.pkg1.name := 'gaussdb';
    best_practices_for_procedure.pkg1.arg := 18;
END;
/
CREATE PROCEDURE

-- Change the value of a package variable.
db_test=# CALL best_practices_for_procedure.pro1();
 pro1
------

(1 row)

-- In practice, it is found that the parameters have been modified.
db_test=# DECLARE
BEGIN
    dbe_output.print_line('id = ' || best_practices_for_procedure.pkg1.id || ' name = ' ||
best_practices_for_procedure.pkg1.name || ' arg = ' || best_practices_for_procedure.pkg1.arg);
    best_practices_for_procedure.pkg1.p1();
    dbe_output.print_line('id = ' || best_practices_for_procedure.pkg1.id || ' name = ' ||
best_practices_for_procedure.pkg1.name || ' arg = ' || best_practices_for_procedure.pkg1.arg);
END;
/
id = 2 name = gaussdb arg = 18
id = 1 name = huawei arg = 20
ANONYMOUS BLOCK EXECUTE

db_test=# DROP SCHEMA best_practices_for_procedure CASCADE;
NOTICE:  drop cascades to 3 other objects
DETAIL:  drop cascades to package 16443
drop cascades to function best_practices_for_procedure.p1()
drop cascades to function best_practices_for_procedure.pro1()
DROP SCHEMA

db_test=#  \c postgres
gaussdb=# DROP DATABASE db_test;
```

10.1.4.2 Cursors

In stored procedures, cursors are important resources. Improper use of cursors may cause the following problems:

- Unclosed cursors will consume system resources, and a large number of cursors that are not closed promptly will severely impact database memory and performance, especially in high-concurrency or iterative operations.

Therefore, you are advised to close the cursor immediately after it is used in a stored procedure.

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

gaussdb=# CREATE TABLE best_practices_for_procedure.tb1 (a int);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

gaussdb=# INSERT INTO best_practices_for_procedure.tb1 VALUES (1),(2),(3);
INSERT 0 3

-- Create a stored procedure that uses a cursor.
```

```
gaussdb=# CREATE OR REPLACE PROCEDURE best_practices_for_procedure.pro_cursor () AS
    my_cursor CURSOR FOR SELECT *FROM best_practices_for_procedure.tb1;
    a int;
BEGIN
    OPEN my_cursor;
    FETCH my_cursor INTO a;
    CLOSE my_cursor; -- Close the cursor promptly.
END;
/
CREATE PROCEDURE

gaussdb=# CALL best_practices_for_procedure.pro_cursor();
 pro_cursor
------------

(1 row)

gaussdb=# DROP SCHEMA best_practices_for_procedure CASCADE;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.pro_cursor()
DROP SCHEMA
```

10.1.4.3 Compatibility

Due to differences in database compatibility, the behavior of stored procedures may be inconsistent under different compatibility settings or GUC parameters. Exercise caution when using these compatibility functions. Example:

- Due to differences in compatibility, functions with output parameters may ignore output values in certain cases. After the GUC parameter **behavior_compat_options** is set to **'proc_outparam_override'**, some scenarios can ensure the correct return of output values and return values. However, since this function behaves differently under different compatibility settings, you are advised not to use functions with output parameters. Instead, you can use procedures with output parameters.

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

-- Create a function with output parameters.
gaussdb=#  CREATE OR REPLACE FUNCTION best_practices_for_procedure.func (a out int, b out int)
RETURN int AS -- This is only an example and is not recommended.
    c int;
BEGIN
    a := 1;
    b := 2;
    c := 3;
    RETURN c;
END;
/
CREATE FUNCTION

-- When a function with output parameters is called, it is found that no value is assigned to
parameters a and b.
gaussdb=# DECLARE
    a int;
    b int;
    c int;
BEGIN
    c := best_practices_for_procedure.func(a, b);
    dbe_output.print_line('a := ' || a || ' b := ' || b || ' c := ' || c);
END;
/
a :=  b :=  c := 3
ANONYMOUS BLOCK EXECUTE
```

```
-- Set the GUC parameter.
gaussdb=# SET behavior_compat_options='proc_outparam_override';
SET

-- When the function with output parameters is called again, values are assigned to parameters a, b,
and c.
gaussdb=# DECLARE
    a int;
    b int;
    c int;
BEGIN
    c := best_practices_for_procedure.func(a, b);
    dbe_output.print_line('a := ' || a || ' b := ' || b || ' c := ' || c);
END;
/
a := 1 b := 2 c := 3
ANONYMOUS BLOCK EXECUTE

-- You are advised to use a stored procedure with output parameters to replace the function with
output parameters. You can change the preceding function to the following stored procedure.
gaussdb=# RESET behavior_compat_options;
gaussdb=# CREATE OR REPLACE PROCEDURE best_practices_for_procedure.proc (a OUT int, b OUT
int, c OUT int) AS
BEGIN
    a := 1;
    b := 2;
    c := 3;
END;
/
CREATE PROCEDURE

gaussdb=# DECLARE
    a int;
    b int;
    c int;
BEGIN
    best_practices_for_procedure.proc(a, b, c);
    dbe_output.print_line('a := ' || a || ' b := ' || b || ' c := ' || c);
END;
/
a := 1 b := 2 c := 3
ANONYMOUS BLOCK EXECUTE

gaussdb=# DROP SCHEMA best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to function best_practices_for_procedure.func()
drop cascades to function best_practices_for_procedure.proc()
DROP SCHEMA
```

- In dynamic statements, if placeholder names are the same, compatibility settings across different databases may cause placeholders to be bound to different variables, thereby affecting expected behavior. After the GUC parameter **behavior_compat_options** is set to **'dynamic_sql_compat'**, you can use placeholders with the same name to bind different variables. However, since this function behaves differently under different compatibility settings, you are advised not to use placeholders with the same name.

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

gaussdb=# CREATE TABLE best_practices_for_procedure.tb1 (a int, b int);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

-- Create a stored procedure that uses dynamic statements and bind the same placeholders to the
same variables.
gaussdb=# CREATE OR REPLACE PROCEDURE best_practices_for_procedure.pro_dynexecute() AS
    a int := 1;
    b int := 2;
BEGIN
    EXECUTE IMMEDIATE 'INSERT INTO best_practices_for_procedure.tb1 VALUES(:1, :1),(:2, :2);' USING
IN a, IN b;
END;
/
CREATE PROCEDURE

gaussdb=# CALL best_practices_for_procedure.pro_dynexecute();
 pro_dynexecute
----------------

(1 row)

-- Check the table and find that the same placeholders are bound to the same variables.
gaussdb=# SELECT * FROM best_practices_for_procedure.tb1;
 a | b
---+---
 1 | 1
 2 | 2
(2 rows)

-- Set the GUC parameter.
gaussdb=# SET behavior_compat_options='dynamic_sql_compat';
SET
gaussdb=# TRUNCATE TABLE best_practices_for_procedure.tb1;
TRUNCATE TABLE

-- Create a stored procedure that uses dynamic statements and bind the same placeholders to
different variables.
gaussdb=# CREATE OR REPLACE PROCEDURE best_practices_for_procedure.pro_dynexecute() AS
    a int := 1;
    b int := 2;
    c int := 3;
    d int := 4;
BEGIN
    EXECUTE IMMEDIATE 'INSERT INTO best_practices_for_procedure.tb1 VALUES(:1, :1),(:2, :2);' USING
IN a, IN b, IN c, IN d;
END;
/
CREATE PROCEDURE

gaussdb=# CALL best_practices_for_procedure.pro_dynexecute();
 pro_dynexecute
----------------

(1 row)

-- After the GUC parameter is set and the function is called, the same placeholders can be bound to
different variables.
gaussdb=# SELECT * FROM best_practices_for_procedure.tb1;
 a | b
---+---
 1 | 2
 3 | 4
(2 rows)

gaussdb=# RESET behavior_compat_options;
RESET

gaussdb=# DROP SCHEMA best_practices_for_procedure CASCADE;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.pro_dynexecute()
DROP SCHEMA
```

10.1.4.4 Exception Handling

Using the exception handling mechanism in stored procedures can improve code fault tolerance, but frequently catching and handling exceptions may lead to performance degradation. Each exception handling involves context creation and destruction, which consumes extra memory and resources. In addition, since exceptions are caught, error information will not be logged, making it more difficult to diagnose issues.

You are advised to use the EXCEPTION processing mechanism only when necessary and ensure that sufficient context information is passed to facilitate fault locating and rectification.

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1(id int, name varchar2(20));
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# create unique index id1 on best_practices_for_procedure.tb1(id);
CREATE INDEX
-- Create a stored procedure with an exception.
gaussdb=# create or replace procedure best_practices_for_procedure.proc1(oi_flag OUT int, os_msg OUT
varchar) as
begin
oi_flag := 0;
os_msg := 'insert into tb1 some data.';
for i in 1..10 loop
if i = 5 then
insert into best_practices_for_procedure.tb1 values(i - 1, 'name'|| i - 1);-- Intentionally create an error.
end if;
insert into best_practices_for_procedure.tb1 values(i, 'name'|| i);
end loop;
exception when others then
oi_flag := 1;
os_msg := SQLERRM; -- Pass the error message out.
end;
/
CREATE PROCEDURE
gaussdb=# declare
oi_flag int;
os_msg varchar(1000);
begin
best_practices_for_procedure.proc1(oi_flag, os_msg);
if oi_flag = 1 then
dbe_output.print_line('Exception for ' || os_msg);
end if;
end;
/
Exception for Duplicate key value violates unique constraint "id1".
ANONYMOUS BLOCK EXECUTE
gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.proc1()
DROP SCHEMA
```

10.1.4.5 User-defined Types

Variables of user-defined types in stored procedures cannot be pushed down. If user-defined types need to be pushed down, use variables to receive elements of user-defined types.

Below is an example:

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# SET CURRENT_SCHEMA=best_practices_for_procedure;
SET
gaussdb=#
gaussdb=# CREATE TABLE tb1(c1 INT, c2 VARCHAR(20));
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'c1' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# INSERT INTO tb1 VALUES(1, 'a'),(2,'b'), (3, 'c');
INSERT 0 3
gaussdb=# -- SELECT statement pushdown
gaussdb=# EXPLAIN SELECT c1 FROM tb1 WHERE c2 = 'a';
                 QUERY PLAN
--------------------------------------------------
 Data Node Scan  (cost=0.00..0.00 rows=0 width=0)
   Node/s: All datanodes
(2 rows)
gaussdb=# -- Variables receiving elements of user-defined types
gaussdb=# CREATE OR REPLACE PROCEDURE proc1() AS
gaussdb$# TYPE ta IS VARRAY(10) OF VARCHAR(30);
gaussdb$# v ta := ta();
gaussdb$# b VARCHAR;
gaussdb$# a INT;
gaussdb$# BEGIN
gaussdb$#     v(1) := 'a';
gaussdb$#     v(2) := 'b';
gaussdb$#     FOR i IN 1..v.count LOOP
gaussdb$#         b := v(i);  -- Use a variable to receive an element of the user-defined type.
gaussdb$#         SELECT c1 INTO a FROM tb1 WHERE c2 = b; -- The execution is successful.
gaussdb$#     END LOOP;
gaussdb$# END;
gaussdb$# /
CREATE PROCEDURE
gaussdb=# CALL proc1();
 proc1
-------

(1 row)
gaussdb=#
gaussdb=# -- User-defined type pushdown
gaussdb=# CREATE OR REPLACE PROCEDURE proc2() AS
gaussdb$# TYPE ta IS VARRAY(10) OF VARCHAR(30);
gaussdb$# v ta := ta();
gaussdb$# b VARCHAR;
gaussdb$# a INT;
gaussdb$# BEGIN
gaussdb$#     v(1) := 'a';
gaussdb$#     v(2) := 'b';
gaussdb$#     FOR i IN 1..v.count LOOP
gaussdb$#         SELECT c1 INTO a FROM tb1 WHERE c2 = v(1); -- User-defined types do not support
pushdown. An error is reported.
gaussdb$#     END LOOP;
gaussdb$# END;
gaussdb$# /
CREATE PROCEDURE
gaussdb=# CALL proc2();
ERROR:  Function v(integer) does not exist.
LINE 1: SELECT c1        FROM tb1 WHERE c2 = v(1)
                                          ^
HINT:  No function matches the given name and argument types. You might need to add explicit type casts.
QUERY:  SELECT c1        FROM tb1 WHERE c2 = v(1)
CONTEXT:  PL/pgSQL function proc2() line 10 at SQL statement
gaussdb=#
gaussdb=# DROP SCHEMA best_practices_for_procedure cascade;
NOTICE:  drop cascades to 3 other objects
DETAIL:  drop cascades to table tb1
drop cascades to function proc1()
drop cascades to function proc2()
DROP SCHEMA
```

10.1.5 Transaction Management

10.1.5.1 Transactions

Stored procedures can use SAVEPOINT and COMMIT/ROLLBACK to manage transactions. Improper use of SAVEPOINT and COMMIT/ROLLBACK may cause the following problems:

- Resources are allocated each time a savepoint is created in a transaction. If the resources are not released promptly, resource consumption will gradually accumulate.
- The COMMIT and ROLLBACK operations of a transaction require synchronization of the database's metadata and logs, and frequent execution may increase I/O overhead, thereby affecting performance.

Suggestions:

- After using a savepoint, use RELEASE SAVEPOINT to release resources promptly.
- Do not create savepoints in a loop because savepoints with the same name will not overwrite each other but will be created again, potentially leading to rapid resource accumulation.

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1(id int, name varchar2(20));
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
-- Create a stored procedure that uses a savepoint.
gaussdb=# create or replace procedure best_practices_for_procedure.proc1() as
begin
savepoint sp1; -- Do not use the savepoint in a loop.
for i in 1..10 loop
insert into best_practices_for_procedure.tb1 values(i, 'name'|| i);
end loop;
release savepoint sp1; -- Release the savepoint.
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc1();
 proc1
-------

(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.proc1()
DROP SCHEMA
```

- Do not perform COMMIT or ROLLBACK frequently.

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1(id int, name varchar2(20));
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# create or replace procedure best_practices_for_procedure.proc1() as
```

```
begin
for i in 1..10 loop
insert into best_practices_for_procedure.tb1 values(i, 'name'|| i);
end loop;
commit; -- Commit after the loop is executed, instead of repeatedly committing in the loop.
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc1();
 proc1
-------

(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.proc1()
DROP SCHEMA
```

10.1.5.2 Autonomous Transactions

An autonomous transaction is an independent transaction started in a stored procedure. The transaction is independent of the primary transaction and can continue its operations even after the primary transaction is committed or rolled back. Executing a stored procedure by starting a new database session may increase the usage of system resources, including memory, CPU, and database connections.

It is recommended that autonomous transactions be used to record service logs instead of being used as the entry or core of a service process. Frequent use of autonomous transactions should be avoided to minimize consumption of system resources.

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.log_table(log_time timestamptz, message text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'log_time' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# create table best_practices_for_procedure.work_table(company text, balance float);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'company' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# insert into best_practices_for_procedure.work_table values('huawei', 100000);
INSERT 0 1
-- Create a stored procedure for an autonomous transaction.
gaussdb=# create or replace procedure best_practices_for_procedure.proc_auto(log_time timestamptz,
message text) as
PRAGMA AUTONOMOUS_TRANSACTION;
begin
insert into best_practices_for_procedure.log_table values (log_time, message); -- Record only logs.
end;
/
CREATE PROCEDURE
-- Call an autonomous transaction in a stored procedure.
gaussdb=# create or replace procedure best_practices_for_procedure.proc1(companys text, turnover float) as
message text;
begin
    update best_practices_for_procedure.work_table set balance = balance + turnover where company =
companys;
    message := 'Company turnover ' || turnover;
    best_practices_for_procedure.proc_auto(current_timestamp, message);
end;
/
CREATE PROCEDURE
```

```
gaussdb=# call best_practices_for_procedure.proc1('huawei', 1000);
 proc1
-------

(1 row)

gaussdb=# select * from best_practices_for_procedure.log_table;
          log_time             |        message
-------------------------------+-----------------------
 2024-11-25 15:23:27.202458+08 | Company turnover 1000
(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 4 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.log_table
drop cascades to table best_practices_for_procedure.work_table
drop cascades to function best_practices_for_procedure.proc_auto(timestamp with time zone,text)
drop cascades to function best_practices_for_procedure.proc1(text,double precision)
DROP SCHEMA
```

10.1.6 Others

10.1.6.1 DDL

Data definition language (DDL) operations (such as CREATE, ALTER, and DROP) are usually locked to ensure atomicity and consistency of changes. In a high-concurrency environment, DDL operations may cause lock conflicts or long-time blocking, affecting the normal execution of other service operations.

You are advised to suspend related service operations when performing DDL changes to prevent adverse impacts on system performance and stability.

10.1.6.2 Complex Dependencies

If there are complex dependencies between stored procedures or packages, the dependent objects may not be created or initialized during creation. As a result, the stored procedure fails to be compiled. In addition, when an object is modified or rebuilt, other stored procedures and packages that directly or indirectly depend on the object become invalid and need to be recompiled, which affects system performance.

To improve system stability and performance, do not create complex dependencies between stored procedures and packages.

```
-- Create an ORA-compatible database.
CREATE DATABASE db_test DBCOMPATIBILITY 'ORA';

-- Switch to the ORA-compatible database.
gaussdb=# \c db_test
db_test=# create schema best_practices_for_procedure;
CREATE SCHEMA

-- An error is reported when pkg1 that depends on pkg2 is created.
db_test=# create or replace package best_practices_for_procedure.pkg1 as
procedure p1();
end pkg1;
/
CREATE PACKAGE

db_test=# create or replace package body best_practices_for_procedure.pkg1 as
procedure p1() as
begin
```

```
best_practices_for_procedure.pkg2.a := 100;
end;
end pkg1;
/
ERROR:  "best_practices_for_procedure.pkg2.a" is not a known variable.
LINE 3: best_practices_for_procedure.pkg2.a := 100;
        ^
QUERY:   DECLARE
begin
best_practices_for_procedure.pkg2.a := 100;
end

-- You can create pkg1 only after pkg2 is created.
db_test=# create or replace package best_practices_for_procedure.pkg2 as
a int;
procedure p1();
end pkg2;
/
CREATE PACKAGE

db_test=# create or replace package body best_practices_for_procedure.pkg2 as
procedure p1() as
begin
null;
end;
end pkg2;
/
CREATE PACKAGE BODY

db_test=# create or replace package best_practices_for_procedure.pkg1 as
procedure p1();
end pkg1;
/
CREATE PACKAGE

db_test=# create or replace package body best_practices_for_procedure.pkg1 as
procedure p1() as
begin
best_practices_for_procedure.pkg2.a := 100;
end;
end pkg1;
/
CREATE PACKAGE BODY

db_test=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 4 other objects
DETAIL:  drop cascades to package 16526
drop cascades to function best_practices_for_procedure.p1()
drop cascades to package 16524
drop cascades to function best_practices_for_procedure.p1()
DROP SCHEMA

templatea=# \c postgres
gaussdb=# DROP DATABASE db_test;
```

10.1.6.3 IMMUTABLE and SHIPPABLE

IMMUTABLE is an attribute used to declare that the result of a stored procedure is determined solely by input parameters and remains independent of the database status. In certain scenarios, stored procedures with the **IMMUTABLE** attribute may be optimized to execute only once, and improper use may lead to unexpected results.

Another attribute for stored procedures is SHIPPABLE, which specifies whether the stored procedures can be pushed down to DNs for execution. If the pushed-down stored procedure accesses the table or database status, data inconsistency may occur.

When using stored procedures and functions with the **IMMUTABLE** and **SHIPPABLE** attributes, you are advised to avoid accessing information in tables or databases to ensure that the behavior meets expectations and maintain data consistency. For details about attributes, see "SQL Reference > SQL Syntax > C > CREATE FUNCTION" in *Developer Guide*.

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1(a int, b int);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# create or replace procedure best_practices_for_procedure.proc1(a int, b int) immutable as
begin
insert into best_practices_for_procedure.tb1 values(a, b); -- This is only an example and is not recommended.
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc1(2, 5);
ERROR:  INSERT is not allowed in a non-volatile function
CONTEXT:  SQL statement "insert into best_practices_for_procedure.tb1 values(a, b)"
PL/pgSQL function best_practices_for_procedure.proc1(integer,integer) line 3 at SQL statement
gaussdb=# create or replace function best_practices_for_procedure.func1(a int, b int) return int immutable
as
begin
return a * b;
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.func1(2, 5);
 func1
-------
    10
(1 row)

gaussdb=# create or replace procedure best_practices_for_procedure.proc2(a int, b int) shippable as
begin
insert into best_practices_for_procedure.tb1 values(a, b); -- This is only an example and is not recommended.
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc2(2, 5);
 proc2
-------

(1 row)

gaussdb=# create or replace function best_practices_for_procedure.func2(a int, b int) return int shippable as
begin
return a * b;
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.func2(2, 5);
 func2
-------
    10
(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 5 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.proc1(integer,integer)
drop cascades to function best_practices_for_procedure.func1(integer,integer)
drop cascades to function best_practices_for_procedure.proc2(integer,integer)
drop cascades to function best_practices_for_procedure.func2(integer,integer)
DROP SCHEMA
```

10.2 Best Practices for Stored Procedures (Centralized Instances)

In GaussDB, business rules and logics are saved as stored procedures.

A stored procedure is a combination of SQL and PL/SQL. Stored procedures can move the code that executes business rules from applications to databases. Therefore, the code storage can be used by multiple programs at a time.

For the basic usage of stored procedures, refer to the "Stored Procedures" section in *Developer Guide*.

10.2.1 Permission Management

By default, stored procedures are granted the SECURITYINVOKER permission. To change this default to the SECURITYDEFINER permission, set the GUC parameter **behavior_compat_options** to **'plsql_security_definer'**. For details about permissions, see "SQL Reference > SQL Syntax > C > CREATE FUNCTION" in *Developer Guide*.

Improper permission mode may cause unauthorized access to sensitive data or unauthorized resource operations. Therefore, select and configure the permission mode with caution to ensure system security.

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA
-- Create two different users.
gaussdb=# CREATE USER test_user1 PASSWORD '********';
CREATE ROLE
gaussdb=# CREATE USER test_user2 PASSWORD '********';
CREATE ROLE
-- Set the permissions of the two users on schema best_practices_for_procedure.
gaussdb=# GRANT usage, create ON SCHEMA best_practices_for_procedure TO test_user1;
GRANT
gaussdb=# GRANT usage, create ON SCHEMA best_practices_for_procedure TO test_user2;
GRANT
-- Switch to the test_user1 user and create a table and a stored procedure.
gaussdb=# SET SESSION AUTHORIZATION test_user1 PASSWORD '********';
SET
gaussdb=> CREATE TABLE best_practices_for_procedure.user1_tb (a int, b int);
CREATE TABLE
gaussdb=> CREATE OR REPLACE PROCEDURE best_practices_for_procedure.user1_proc() AS
  BEGIN
    INSERT INTO best_practices_for_procedure.user1_tb VALUES(1,1);
  END;
/
CREATE PROCEDURE
-- Switch to the test_user2 user to execute the stored procedure created by the test_user1 user. An error is
reported, indicating that the user does not have the permission on the user1_tb table because stored
procedures are executed with the caller's permissions by default.
gaussdb=> RESET SESSION AUTHORIZATION;
RESET
gaussdb=# SET SESSION AUTHORIZATION test_user2 PASSWORD '********';
SET
gaussdb=> CALL best_practices_for_procedure.user1_proc();
ERROR:  Permission denied for relation user1_tb.
DETAIL:  N/A.
CONTEXT:  SQL statement "insert into best_practices_for_procedure.user1_tb values(1,1)"
PL/pgSQL function best_practices_for_procedure.user1_proc() line 3 at SQL statement
-- Set the GUC parameter to use the creator's permissions by default when creating stored procedures.
gaussdb=> SET behavior_compat_options='plsql_security_definer';
```

```
SET
-- Switch to the test_user1 user and re-create the stored procedure.
gaussdb=> RESET SESSION AUTHORIZATION;
RESET
gaussdb=# SET SESSION AUTHORIZATION test_user1 PASSWORD '********';
SET
gaussdb=> CREATE OR REPLACE PROCEDURE best_practices_for_procedure.user1_proc() AS
  BEGIN
    INSERT INTO best_practices_for_procedure.user1_tb VALUES(1,1);
  END;
/
CREATE PROCEDURE
-- Switch to the test_user2 user and execute the stored procedure. The execution is successful.
gaussdb=> RESET SESSION AUTHORIZATION;
RESET
gaussdb=# SET SESSION AUTHORIZATION test_user2 PASSWORD '********';
SET
gaussdb=> CALL best_practices_for_procedure.user1_proc();
 proc_user1
------------

(1 row)

-- Switch to the test_user1 user and view the table content.
gaussdb=> RESET SESSION AUTHORIZATION;
RESET
gaussdb=# SET SESSION AUTHORIZATION test_user1 PASSWORD '********';
SET
gaussdb=> SELECT * FROM best_practices_for_procedure.user1_tb;
 a | b
---+---
 1 | 1
(1 row)

-- Clean the environment.
gaussdb=> RESET behavior_compat_options;
RESET
gaussdb=> RESET SESSION AUTHORIZATION;
RESET
gaussdb=# DROP SCHEMA best_practices_for_procedure CASCADE;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.user1_tb
drop cascades to function best_practices_for_procedure.user1_proc()
DROP SCHEMA
gaussdb=# DROP USER test_user1;
DROP ROLE
gaussdb=# DROP USER test_user2;
DROP ROLE
```

10.2.2 Naming Convention

Improper stored procedure and variable naming may adversely affect system usage.

- The name of a stored procedure, variable, or type can contain a maximum of 63 characters. If this limit is exceeded, the name is automatically truncated to 63 characters.

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

-- When a stored procedure name containing 66 characters is created, a message is displayed,
indicating that the name is truncated to 63 characters.
gaussdb=# CREATE OR REPLACE PROCEDURE
best_practices_for_procedure.abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz0123456789101
1() AS
BEGIN
    NULL;
END;
```

```
/
NOTICE:  identifier "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891011" will
be truncated to "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891"
CREATE PROCEDURE

-- When a variable name containing 66 characters is created, a message is displayed, indicating that
the name is truncated to 63 characters.
gaussdb=# CREATE OR REPLACE PROCEDURE
best_practices_for_procedure.proc1(abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz0123456
7891011 int) as
BEGIN
    NULL;
END;
/
NOTICE:  identifier "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891011" will
be truncated to "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891"
CREATE PROCEDURE

gaussdb=# DROP SCHEMA best_practices_for_procedure CASCADE;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to function
best_practices_for_procedure.abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891()
drop cascades to function best_practices_for_procedure.proc1(integer)
DROP SCHEMA
```

- When creating a stored procedure, avoid using variables or types with the same name in different variable scopes. For details, see "Stored Procedures > Basic Statements > Variable Definition Statements > Scope of a Variable" in *Developer Guide*. Using variables and types with the same name in different variable scopes may reduce the readability of stored procedures and increase the maintenance difficulty.

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

-- Create a stored procedure, create the same variable name in different variable scopes, and assign
values.
gaussdb=# CREATE OR REPLACE PROCEDURE best_practices_for_procedure.proc1() AS
    name varchar2(10) := 'outer';
    age int := 2025;
BEGIN
    DECLARE
    name varchar2(10) := 'inner'; -- This is only an example and is not recommended.
    age int := 2024; -- This is only an example and is not recommended.
    BEGIN
        dbe_output.print_line('inner name =' || name);
        dbe_output.print_line('inner age =' || age);
    END;
    dbe_output.print_line('outer name =' || name);
    dbe_output.print_line('outer age =' || age);
END;
/
CREATE PROCEDURE

-- Execute the stored procedure. The same variable name in different scopes actually refers to
different variables.
gaussdb=# CALL best_practices_for_procedure.proc1();
inner name =inner
inner age =2024
outer name =outer
outer age =2025
 proc1
-------

(1 row)

gaussdb=# DROP SCHEMA best_practices_for_procedure cascade;
NOTICE:  drop cascades to function best_practices_for_procedure.proc1()
DROP SCHEMA
```

- Do not use SQL keywords in stored procedure, internal variable, and data type names to ensure that the stored procedure can run properly in all scenarios.

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

gaussdb=#
CREATE OR REPLACE PROCEDURE best_practices_for_procedure."as"() AS -- This is only an example
and is not recommended.
BEGIN
    NULL;
END;
/
CREATE PROCEDURE

-- A direct call will result in an error.
gaussdb=# CALL as();
ERROR:  syntax error at or near "as"
LINE 1: call as();
             ^
gaussdb=# CALL best_practices_for_procedure."as"();
 as
----

(1 row)

gaussdb=# DROP SCHEMA best_practices_for_procedure CASCADE;
NOTICE:  drop cascades to function best_practices_for_procedure."as"()
DROP SCHEMA
```

- When creating a stored procedure, avoid using the same name as system functions to prevent confusion. If the same name must be used, specify the schema during a call.

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

-- Create an abs function with the same name as the abs system function in the schema. This is only
an example and is not recommended.
gaussdb=#
CREATE OR REPLACE FUNCTION best_practices_for_procedure.abs(a int) RETURN int AS
BEGIN
    dbe_output.print_line('my abs funciton.');
    RETURN abs(a);
END;
/
CREATE FUNCTION

-- Call a stored procedure. If no schema is added, the abs system function is called.
gaussdb=# CALL abs(-1);
 abs
-----
   1
(1 row)

-- You are advised to add a schema.
gaussdb=# CALL best_practices_for_procedure.abs(-1);
my abs funciton.
 abs
-----
   1
(1 row)

gaussdb=# DROP SCHEMA best_practices_for_procedure CASCADE;
NOTICE:  drop cascades to function best_practices_for_procedure.abs(integer)
DROP SCHEMA
```

10.2.3 Access Object

If no schema is specified for a stored procedure, the stored procedure searches for objects based on the sequence specified by **SEARCH_PATH**. As a result, unexpected objects may be accessed. If tables, stored procedures, and other database objects with the same name exist in different schemas, unexpected results may occur if the schema is not specified. Therefore, it is recommended that you always explicitly specify a schema when a stored procedure accesses a data object.

Example:

```
-- Create two different schemas.
gaussdb=# create schema best_practices_for_procedure1;
CREATE SCHEMA
gaussdb=# create schema best_practices_for_procedure2;
CREATE SCHEMA
-- Create the same stored procedure in two different schemas.
gaussdb=# create or replace procedure best_practices_for_procedure1.proc1() as
begin
dbe_output.print_line('in schema best_practices_for_procedure1');
end;
/
CREATE PROCEDURE
gaussdb=# create or replace procedure best_practices_for_procedure2.proc1() as
begin
dbe_output.print_line('in schema best_practices_for_procedure2');
end;
/
CREATE PROCEDURE
-- Calling the same stored procedure with different search_path settings may lead to differences.
gaussdb=# set search_path to best_practices_for_procedure1, best_practices_for_procedure2;
SET
gaussdb=# call proc1();
in schema best_practices_for_procedure1
 proc1
-------

(1 row)

gaussdb=# reset search_path;
RESET
gaussdb=# set search_path to best_practices_for_procedure2, best_practices_for_procedure1;
SET
gaussdb=# call proc1();
in schema best_practices_for_procedure2
 proc1
-------

(1 row)

gaussdb=# reset search_path;
RESET
gaussdb=# drop schema best_practices_for_procedure1 cascade;
NOTICE:  drop cascades to function best_practices_for_procedure1.proc1()
DROP SCHEMA
gaussdb=# drop schema best_practices_for_procedure2 cascade;
NOTICE:  drop cascades to function best_practices_for_procedure2.proc1()
DROP SCHEMA
```

10.2.4 Statement Functions

10.2.4.1 Package Variables

A package variable is a global variable defined in a package. Its lifecycle covers the entire database session. Improper use may cause the following problems:

- If a variable is completely transparent to users who have the package access permission, the variable may be shared among multiple stored procedures and modified unexpectedly.
- The lifecycle of a package variable is at the session level. Improper operations may cause residual data and affect other stored procedures.
- Caching a large number of package variables in a session may consume a substantial amount of memory.

Therefore, you are advised to use package variables with caution and ensure that their access and lifecycle are properly managed.

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create or replace package best_practices_for_procedure.pkg1 as
id int;
name varchar2(20);
arg int;
procedure p1();
end pkg1;
/
CREATE PACKAGE
gaussdb=# create or replace package body best_practices_for_procedure.pkg1 as
procedure p1() as
begin
 id := 1;
 name := 'huawei';
 arg := 20;
end;
end pkg1;
/
CREATE PACKAGE BODY
-- Create a stored procedure and modify package variables.
gaussdb=# create or replace procedure best_practices_for_procedure.pro1 () as
begin
best_practices_for_procedure.pkg1.id := 2;
best_practices_for_procedure.pkg1.name := 'gaussdb';
best_practices_for_procedure.pkg1.arg := 18;
end;
/
CREATE PROCEDURE
-- Change the value of a package variable.
gaussdb=# call best_practices_for_procedure.pro1();
 pro1
------

(1 row)

-- In practice, it is found that the parameters have been modified.
gaussdb=# declare
begin
dbe_output.print_line('id = ' || best_practices_for_procedure.pkg1.id || ' name = ' ||
best_practices_for_procedure.pkg1.name || ' arg = ' || best_practices_for_procedure.pkg1.arg);
best_practices_for_procedure.pkg1.p1();
dbe_output.print_line('id = ' || best_practices_for_procedure.pkg1.id || ' name = ' ||
best_practices_for_procedure.pkg1.name || ' arg = ' || best_practices_for_procedure.pkg1.arg);
end;
/
id = 2 name = gaussdb arg = 18
id = 1 name = huawei arg = 20
ANONYMOUS BLOCK EXECUTE
gaussdb=# drop schema best_practices_for_procedure cascade;
```

```
NOTICE:  drop cascades to 3 other objects
DETAIL:  drop cascades to package 16782
drop cascades to function best_practices_for_procedure.p1()
drop cascades to function best_practices_for_procedure.pro1()
DROP SCHEMA
```

10.2.4.2 Cursors

In stored procedures, cursors are important resources. Improper use of cursors may cause the following problems:

- Unclosed cursors will consume system resources, and a large number of cursors that are not closed promptly will severely impact database memory and performance, especially in high-concurrency or iterative operations.

Therefore, you are advised to close the cursor immediately after it is used in a stored procedure.

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1 (a int);
CREATE TABLE
gaussdb=# insert into best_practices_for_procedure.tb1 values (1),(2),(3);
INSERT 0 3
-- Create a stored procedure that uses a cursor.
gaussdb=# create or replace procedure best_practices_for_procedure.pro_cursor () as
my_cursor cursor for select *from best_practices_for_procedure.tb1;
a int;
begin
open my_cursor;
fetch my_cursor into a;
close my_cursor; -- Close the cursor promptly.
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.pro_cursor();
 pro_cursor
------------

(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.pro_cursor()
DROP SCHEMA
```

10.2.4.3 Compatibility

Due to differences in database compatibility, the behavior of stored procedures may be inconsistent under different compatibility settings or GUC parameters. Exercise caution when using these compatibility functions. Example:

- Due to differences in compatibility, functions with output parameters may ignore output values in certain cases. After the GUC parameter **behavior_compat_options** is set to **'proc_outparam_override'**, some scenarios can ensure the correct return of output values and return values. However, since this function behaves differently under different compatibility settings, you are advised not to use functions with output parameters. Instead, you can use procedures with output parameters.

  ```
  gaussdb=# create schema best_practices_for_procedure;
  CREATE SCHEMA
  -- Create a function with output parameters.
  ```

```
gaussdb=# create or replace function best_practices_for_procedure.func (a out int, b out int) return
int as -- This is only an example and is not recommended.
c int;
begin
a := 1;
b := 2;
c := 3;
return c;
end;
/
CREATE FUNCTION
-- When a function with output parameters is called, it is found that no value is assigned to
parameters a and b.
gaussdb=# declare
a int;
b int;
c int;
begin
c := best_practices_for_procedure.func(a, b);
dbe_output.print_line('a := ' || a || ' b := ' || b || ' c := ' || c);
end;
/
a :=  b :=  c := 3
ANONYMOUS BLOCK EXECUTE
-- Set the GUC parameter.
gaussdb=# set behavior_compat_options='proc_outparam_override';
SET
-- When the function with output parameters is called again, values are assigned to parameters a, b,
and c.
gaussdb=# declare
a int;
b int;
c int;
begin
c := best_practices_for_procedure.func(a, b);
dbe_output.print_line('a := ' || a || ' b := ' || b || ' c := ' || c);
end;
/
a := 1 b := 2 c := 3
ANONYMOUS BLOCK EXECUTE
-- You are advised to use a stored procedure with output parameters to replace the function with
output parameters. You can change the preceding function to the following stored procedure.
gaussdb=# reset behavior_compat_options;
gaussdb=# create or replace procedure best_practices_for_procedure.proc (a out int, b out int, c out
int) as
begin
a := 1;
b := 2;
c := 3;
end;
/
CREATE PROCEDURE
gaussdb=# declare
a int;
b int;
c int;
begin
best_practices_for_procedure.proc(a, b, c);
dbe_output.print_line('a := ' || a || ' b := ' || b || ' c := ' || c);
end;
/
a := 1 b := 2 c := 3
ANONYMOUS BLOCK EXECUTE
gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to function best_practices_for_procedure.func()
drop cascades to function best_practices_for_procedure.proc()
DROP SCHEMA
```

- In dynamic statements, if placeholder names are the same, compatibility settings across different databases may cause placeholders to be bound to different variables, thereby affecting expected behavior. After the GUC parameter **behavior_compat_options** is set to **'dynamic_sql_compat'**, you can use placeholders with the same name to bind different variables. However, since this function behaves differently under different compatibility settings, you are advised not to use placeholders with the same name.

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1 (a int, b int);
CREATE TABLE
-- Create a stored procedure that uses dynamic statements and bind the same placeholders to the
same variables.
gaussdb=# create or replace procedure best_practices_for_procedure.pro_dynexecute() as
a int := 1;
b int := 2;
begin
execute immediate 'insert into best_practices_for_procedure.tb1 values(:1, :1),(:2, :2);' using in a, in b;
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.pro_dynexecute();
 pro_dynexecute
----------------

(1 row)

-- Check the table and find that the same placeholders are bound to the same variables.
gaussdb=# select *from best_practices_for_procedure.tb1;
 a | b
---+---
 1 | 1
 2 | 2
(2 rows)

-- Set the GUC parameter.
gaussdb=# set behavior_compat_options='dynamic_sql_compat';
SET
gaussdb=# truncate table best_practices_for_procedure.tb1;
TRUNCATE TABLE
-- Create a stored procedure that uses dynamic statements and bind the same placeholders to
different variables.
gaussdb=# create or replace procedure best_practices_for_procedure.pro_dynexecute() as
a int := 1;
b int := 2;
c int := 3;
d int := 4;
begin
execute immediate 'insert into best_practices_for_procedure.tb1 values(:1, :1),(:2, :2);' using in a, in b,
in c, in d;
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.pro_dynexecute();
 pro_dynexecute
----------------

(1 row)

-- After the GUC parameter is set and the function is called, the same placeholders can be bound to
different variables.
gaussdb=# select * from best_practices_for_procedure.tb1;
 a | b
---+---
 1 | 2
 3 | 4
(2 rows)
```

```
gaussdb=# reset behavior_compat_options;
RESET
gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.pro_dynexecute()
DROP SCHEMA
```

10.2.4.4 Exception Handling

Using the exception handling mechanism in stored procedures can improve code fault tolerance, but frequently catching and handling exceptions may lead to performance degradation. Each exception handling involves context creation and destruction, which consumes extra memory and resources. In addition, since exceptions are caught, error information will not be logged, making it more difficult to diagnose issues.

You are advised to use the EXCEPTION processing mechanism only when necessary and ensure that sufficient context information is passed to facilitate fault locating and rectification.

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1(id int, name varchar2(20));
CREATE TABLE
gaussdb=# create unique index id1 on best_practices_for_procedure.tb1(id);
CREATE INDEX
-- Create a stored procedure with an exception.
gaussdb=# create or replace procedure best_practices_for_procedure.proc1(oi_flag OUT int, os_msg OUT
varchar) as
begin
oi_flag := 0;
os_msg := 'insert into tb1 some data.';
for i in 1..10 loop
if i = 5 then
insert into best_practices_for_procedure.tb1 values(i - 1, 'name'|| i - 1);-- Intentionally create an error.
end if;
insert into best_practices_for_procedure.tb1 values(i, 'name'|| i);
end loop;
exception when others then
oi_flag := 1;
os_msg := SQLERRM; -- Pass the error message out.
end;
/
CREATE PROCEDURE
gaussdb=# declare
oi_flag int;
os_msg varchar(1000);
begin
best_practices_for_procedure.proc1(oi_flag, os_msg);
if oi_flag = 1 then
dbe_output.print_line('Exception for ' || os_msg);
end if;
end;
/
Exception for Duplicate key value violates unique constraint "id1".
ANONYMOUS BLOCK EXECUTE
gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.proc1()
DROP SCHEMA
```

10.2.5 Transaction Management

10.2.5.1 Transactions

Stored procedures can use SAVEPOINT and COMMIT/ROLLBACK to manage transactions. Improper use of SAVEPOINT and COMMIT/ROLLBACK may cause the following problems:

- Resources are allocated each time a savepoint is created in a transaction. If the resources are not released promptly, resource consumption will gradually accumulate.
- The COMMIT and ROLLBACK operations of a transaction require synchronization of the database's metadata and logs, and frequent execution may increase I/O overhead, thereby affecting performance.

Suggestions:

- After using a savepoint, use RELEASE SAVEPOINT to release resources promptly.
- Do not create savepoints in a loop because savepoints with the same name will not overwrite each other but will be created again, potentially leading to rapid resource accumulation.

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1(id int, name varchar2(20));
CREATE TABLE
-- Create a stored procedure that uses a savepoint.
gaussdb=# create or replace procedure best_practices_for_procedure.proc1() as
begin
savepoint sp1; -- Do not use the savepoint in a loop.
for i in 1..10 loop
insert into best_practices_for_procedure.tb1 values(i, 'name'|| i);
end loop;
release savepoint sp1; -- Release the savepoint.
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc1();
 proc1
-------

(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.proc1()
DROP SCHEMA
```

- Do not perform COMMIT or ROLLBACK frequently.

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1(id int, name varchar2(20));
CREATE TABLE
gaussdb=# create or replace procedure best_practices_for_procedure.proc1() as
begin
for i in 1..10 loop
insert into best_practices_for_procedure.tb1 values(i, 'name'|| i);
end loop;
commit; -- Commit after the loop is executed, instead of repeatedly committing in the loop.
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc1();
 proc1
-------
```

```
(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.proc1()
DROP SCHEMA
```

10.2.5.2 Autonomous Transactions

An autonomous transaction is an independent transaction started in a stored procedure. The transaction is independent of the primary transaction and can continue its operations even after the primary transaction is committed or rolled back. Executing a stored procedure by starting a new database session may increase the usage of system resources, including memory, CPU, and database connections.

It is recommended that autonomous transactions be used to record service logs instead of being used as the entry or core of a service process. Frequent use of autonomous transactions should be avoided to minimize consumption of system resources.

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.log_table(log_time timestamptz, message text);
CREATE TABLE
gaussdb=# create table best_practices_for_procedure.work_table(company text, balance float);
CREATE TABLE
gaussdb=# insert into best_practices_for_procedure.work_table values('huawei', 100000);
INSERT 0 1
-- Create a stored procedure that contains an autonomous transaction.
gaussdb=# create or replace procedure best_practices_for_procedure.proc_auto(log_time timestamptz,
message text) as
PRAGMA AUTONOMOUS_TRANSACTION;
begin
insert into best_practices_for_procedure.log_table values (log_time, message); -- Record only logs.
end;
/
CREATE PROCEDURE
-- Call an autonomous transaction in a stored procedure.
gaussdb=# create or replace procedure best_practices_for_procedure.proc1(companys text, turnover float) as
message text;
begin
    update best_practices_for_procedure.work_table set balance = balance + turnover where company =
companys;
    message := 'Company turnover ' || turnover;
    best_practices_for_procedure.proc_auto(current_timestamp, message);
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc1('huawei', 1000);
 proc1
-------

(1 row)

gaussdb=# select * from best_practices_for_procedure.log_table;
           log_time            |        message
-------------------------------+----------------------
 2024-11-22 16:21:35.27499+08 | Company turnover 1000
(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 4 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.log_table
drop cascades to table best_practices_for_procedure.work_table
drop cascades to function best_practices_for_procedure.proc_auto(timestamp with time zone,text)
drop cascades to function best_practices_for_procedure.proc1(text,double precision)
DROP SCHEMA
```

10.2.6 Others

10.2.6.1 DDL

Data definition language (DDL) operations (such as CREATE, ALTER, and DROP) are usually locked to ensure atomicity and consistency of changes. In a high-concurrency environment, DDL operations may cause lock conflicts or long-time blocking, affecting the normal execution of other service operations.

You are advised to suspend related service operations when performing DDL changes to prevent adverse impacts on system performance and stability.

10.2.6.2 Complex Dependencies

If there are complex dependencies between stored procedures or packages, the dependent objects may not be created or initialized during creation. As a result, the stored procedure fails to be compiled. In addition, when an object is modified or rebuilt, other stored procedures and packages that directly or indirectly depend on the object become invalid and need to be recompiled, which affects system performance.

To improve system stability and performance, do not create complex dependencies between stored procedures and packages.

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
-- An error is reported when pkg1 that depends on pkg2 is created.
gaussdb=# create or replace package best_practices_for_procedure.pkg1 as
procedure p1();
end pkg1;
/
CREATE PACKAGE
gaussdb=# create or replace package body best_practices_for_procedure.pkg1 as
procedure p1() as
begin
best_practices_for_procedure.pkg2.a := 100;
end;
end pkg1;
/
ERROR:  "best_practices_for_procedure.pkg2.a" is not a known variable.
LINE 3: best_practices_for_procedure.pkg2.a := 100;
        ^
QUERY:   DECLARE
begin
best_practices_for_procedure.pkg2.a := 100;
end
-- You can create pkg1 only after pkg2 is created.
gaussdb=# create or replace package best_practices_for_procedure.pkg2 as
a int;
procedure p1();
end pkg2;
/
CREATE PACKAGE
gaussdb=# create or replace package body best_practices_for_procedure.pkg2 as
procedure p1() as
begin
null;
end;
```

```
end pkg2;
/
CREATE PACKAGE BODY
gaussdb=# create or replace package best_practices_for_procedure.pkg1 as
procedure p1();
end pkg1;
/
CREATE PACKAGE
gaussdb=# create or replace package body best_practices_for_procedure.pkg1 as
procedure p1() as
begin
best_practices_for_procedure.pkg2.a := 100;
end;
end pkg1;
/
CREATE PACKAGE BODY
gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 4 other objects
DETAIL:  drop cascades to package 16836
drop cascades to function best_practices_for_procedure.p1()
drop cascades to package 16834
drop cascades to function best_practices_for_procedure.p1()
DROP SCHEMA
```

10.2.6.3 IMMUTABLE

IMMUTABLE is an attribute used to declare that the result of a stored procedure is determined solely by input parameters and remains independent of the database status. In certain scenarios, stored procedures with the **IMMUTABLE** attribute may be optimized to execute only once, and improper use may lead to unexpected results.

When using stored procedures and functions with the **IMMUTABLE** attribute, you are advised to avoid accessing information in tables or databases to ensure that the behavior meets expectations. For details about attributes, see "SQL Reference > SQL Syntax > C > CREATE FUNCTION" in *Developer Guide*.

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1(a int, b int);
CREATE TABLE
gaussdb=# create or replace procedure best_practices_for_procedure.proc1(a int, b int) immutable as
begin
insert into best_practices_for_procedure.tb1 values(a, b); -- This is only an example and is not recommended.
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc1(2, 5);
ERROR:  INSERT is not allowed in a non-volatile function
CONTEXT:  SQL statement "insert into best_practices_for_procedure.tb1 values(a, b)"
PL/pgSQL function best_practices_for_procedure.proc1(integer,integer) line 3 at SQL statement
gaussdb=# create or replace function best_practices_for_procedure.func1(a int, b int) return int immutable
as
begin
return a * b;
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.func1(2, 5);
 func1
-------
    10
(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 3 other objects
```

```
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.proc1(integer,integer)
drop cascades to function best_practices_for_procedure.func1(integer,integer)
DROP SCHEMA
```

11 Best Practices for Import and Export Using COPY

11.1 Best Practices for Import and Export Using COPY (Distributed Instances)

The COPY syntax is a useful tool for exporting data from GaussDB to data files in CSV, BINARY, FIXED, or TEXT format. It can also import data from the four types of files to a specified table. For details about the COPY syntax, see "SQL Reference > SQL Syntax > C > COPY" in *Developer Guide*.

11.1.1 Typical Scenarios

Preparations

During import or export using COPY, the database requires file read/write permissions on the server. To grant these permissions, enable the GUC parameter **enable_copy_server_files**.

```
gs_guc reload -I all -N all -Z datanode -Z coordinator -c "enable_copy_server_files=on"
```

NOTICE

- Parameters in the COPY command must match the actual data in files, including the format, delimiter, newline character, and character set.
- If the specified client encoding matches the server encoding, the COPY command preserves the binary form of data during import and export, preventing any structural damage that may result from transcoding. This mechanism guarantees the integrity and originality of data migration, making it suitable for scenarios that demand absolute fidelity.

11.1.1.1 Using the Recommended CSV Format

In CSV format, each file is divided into multiple records using end-of-line (EOL) characters. Each record is further split into multiple fields using delimiters, which are commas by default. Each field can be enclosed within a pair of quote characters, which are double quotation marks by default. This eliminates the need to escape special characters such as EOL characters and delimiters within the field content. Furthermore, CSV is a widely recognized standard with exceptional cross-platform compatibility and cross-industry universality, solidifying its position as the top recommended format.

Recommended export command:

```
COPY {data_source} TO '/path/export.csv' delimiter ',' quote '"' escape '"' encoding {server_encoding} csv;
-- data_source can be a table name or a SELECT statement.
-- server_encoding can be obtained using SHOW server_encoding.
```

Corresponding import command:

```
COPY {data_destination} FROM '/path/export.csv' delimiter ',' quote '"' escape '"' encoding {file_encoding}
csv;
-- data_destination can only be a table name.
-- file_encoding indicates the encoding format used during file export.
```

NOTICE

To import a manually created CSV data file into the database, ensure that the file complies with CSV standards. Additionally, specify the correct parameters, including delimiters, quote characters, and EOL characters, in the **COPY** command.

Examples

Step 1 Prepare data.

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# CREATE TABLE test_copy(id int, name text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(2, e'bb\nb');
INSERT 0 1
db1=# insert into test_copy values(3, e'cc\tc');
INSERT 0 1
db1=# insert into test_copy(name) values('ddd');
INSERT 0 1
db1=# insert into test_copy values(5, e'ee\\e');
INSERT 0 1
db1=# insert into test_copy values(6, ',');
INSERT 0 1
db1=# insert into test_copy values(7, '"');
INSERT 0 1
db1=#  SELECT * FROM test_copy;
 id |   name
----+-----------
  1 | aaa
  2 | bb       +
    | b
  3 | cc      c
    | ddd
  5 | ee\e
  6 | ,
  7 | "
(7 rows)
```

Step 2 Export data from the entire table.

```
db1=# copy test_copy to '/home/xy/test.csv' delimiter ',' quote '"' escape '"' encoding 'UTF-8' csv;
COPY 7
```

The content of the exported CSV file is as follows:

```
1,aaa
2,"bb
b"
3,cc c
,ddd
5,ee\e
6,","
7,""""
```

Step 3 Import data.

```
db1=# truncate test_copy;
TRUNCATE TABLE
db1=# copy test_copy from '/home/xy/test.csv' delimiter ',' quote '"' escape '"' encoding 'UTF-8' csv;
COPY 7
```

Step 4 (Custom data set export) Export the **name** column for all rows in **test_copy**, excluding those with an empty ID.

```
db1=# copy (select name from test_copy where id is not null) to '/home/xy/test.csv' delimiter ',' quote '"'
escape '"' encoding 'UTF-8' csv;
COPY 6
```

The content of the exported CSV file is as follows:

```
aaa
"bb
b"
cc c
ee\e
","
""""
```

----End

11.1.1.2 Importing and Exporting Data with Extreme Performance

When there are strict demands for data import and export performance, and the import and export occur between clusters of the same version, you can use the BINARY format to store and read data as binary numbers other than regular text. Despite offering performance advantages over other formats, this format also comes with drawbacks, such as:

1. The BINARY format is specific to GaussDB, making it non-portable. It is advisable to use this format only for importing and exporting data between databases of the same version.
2. The BINARY format is tightly coupled with specific data types. For example, while the TEXT format allows exporting data from a smallint field and importing it into an integer column, this is not possible with the BINARY format.
3. Certain data types cannot be imported or exported using the BINARY format. For details, see BINARY limitations.

Recommended export command:

```
set client_encoding = '{server_encoding}';
copy {data_source} to '/path/export.bin' binary;
-- data_source can be a table name or a SELECT statement.
-- server_encoding can be obtained using SHOW server_encoding.
```

Corresponding import command:

```
set client_encoding = '{file_encoding}';
copy {data_destination} from '/path/export.bin' binary;
-- data_destination can only be a table name.
-- file_encoding indicates the encoding format used during binary file export.
```

NOTICE

Before choosing the BINARY format, carefully review its limitations. Use it to enhance import and export performance only when you are completely certain that these limitations will not impact the data to be exported.

Examples

Step 1 Prepare data.

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# CREATE TABLE test_copy(id int, name text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(3, e'cc\tc');
INSERT 0 1
db1=# insert into test_copy(name) values('ddd');
INSERT 0 1
db1=# insert into test_copy values(5, e'ee\\e');
INSERT 0 1
db1=# insert into test_copy values(6, ',');
INSERT 0 1
db1=# insert into test_copy values(7, '"');
INSERT 0 1
db1=# SELECT * FROM test_copy;
 id |   name
----+-----------
  1 | aaa
  3 | cc      c
    | ddd
  5 | ee\e
  6 | ,
  7 | "
(6 rows)
```

Step 2 Export data.

```
db1=# set client_encoding = 'UTF-8';
SET
db1=# COPY test_copy TO '/home/xy/test.bin' BINARY;
COPY 6
```

Step 3 Import data.

```
db1=# truncate test_copy;
TRUNCATE TABLE
```

```
db1=# set client_encoding = 'UTF-8';
SET
db1=# copy test_copy from '/home/xy/test.bin' BINARY;
COPY 6
```

----End

11.1.1.3 Exporting Data Files for Manual Parsing

The FIXED format maintains a fixed data structure to simplify file parsing: Each row represents a record, and within each row, the starting offset and length of each field value are fixed. Therefore, it is advisable to utilize the FIXED format when you need to manually parse a data file exported from the database. Despite this, the FIXED format has its limitations, as detailed in FIXED description.

Recommended export command:

```
COPY {data_source} FROM '/path/export.fixed' encoding {server_encoding} FIXED
FORMATTER(col1_name(col1_offset, col1_length), col2_name(col2_offset, col2_length));
-- data_source can be a table name or a SELECT statement.
-- server_encoding can be obtained using SHOW server_encoding.
-- col1_name(col1_offset, col1_length) indicates that the data named col1_name in each row of the data
file starts at the position with an offset of col1_offset and extends for a length of col1_length.
```

Corresponding import command:

```
COPY {data_destination} from '/path/export.fixed' encoding {file_encoding} FIXED
FORMATTER(col1_name(col1_offset, col1_length), col2_name(col2_offset, col2_length));
-- data_destination can only be a table name.
-- file_encoding indicates the encoding format used during file export.
-- col1_name(col1_offset, col1_length) indicates that the data named col1_name in each row of the data
file starts at the position with an offset of col1_offset and extends for a length of col1_length.
```

NOTICE

The fixed column width format has limitations: It requires maintaining a relatively uniform width for each column in the table, allowing for the selection of an appropriate fixed **col_length** to prevent data truncation and eliminate unnecessary spaces in all columns. In cases where column widths vary significantly or change dynamically, it is advisable to prioritize flexible formats such as TEXT or CSV to ensure data integrity and efficient use of storage space.

Examples

Step 1 Prepare data.

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# CREATE TABLE test_copy(id int, name text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(2, 'bb"b');
INSERT 0 1
db1=# insert into test_copy values(3, 'cc  c');
```

```
INSERT 0 1
db1=# insert into test_copy values('', e'dd\td');
INSERT 0 1
db1=# insert into test_copy values('5', e'ee\e');
INSERT 0 1
db1=# select * from test_copy;
 id |   name
----+-----------
  1 | aaa
  2 | bb"b
  3 | cc  c
    | dd      d
  5 | eee
(5 rows)
```

Step 2 Export data.

```
db1=# COPY test_copy TO '/home/xy/test.fixed' encoding 'UTF-8' FIXED FORMATTER(id(0,1), name(1,5));
COPY 5
```

The content of the exported data file is as follows:

```
1  aaa
2 bb"b
3cc  c
  dd    d
5  eee
```

Step 3 Import data.

```
db1=# truncate test_copy;
TRUNCATE TABLE
db1=# copy test_copy from '/home/xy/test.fixed' encoding 'UTF-8' FIXED FORMATTER(id(0,1), name(1,5));
COPY 5
```

----End

11.1.1.4 Importing and Exporting Data When Only the TEXT Format Is Available

In TEXT format, each file is divided into multiple records using EOL characters. Each record is further split into multiple fields using delimiters, which are tab characters ('\t') by default.

Using the TEXT format in GaussDB necessitates special logic, for example:

- The backspace (0x08), form-feed (0x0C), newline (0x0A), carriage return (0x0D), horizontal tab (0x09), and vertical tab (0x0B) characters are escaped as **'\b'**, **'\f'**, **'\n'**, **'\r'**, **'\t'**, and **'\v'**, respectively.
- By default, the EOL character is configured as the first identified **'\n'**, **'\r'**, or **'\r\n'** during import, and as **'\n'** during export.
- A single backslash is escaped as double backslashes.
- NULL values are escaped as **'\N'**.

Exporting Data from and Importing Data into GaussDB

Recommended export command:

```
copy {data_source} to '/path/export.txt' eol e'\n' delimiter e'\t' encoding '{server_encoding}';
-- data_source can be a table name or a SELECT statement.
-- server_encoding can be obtained using SHOW server_encoding.
```

Corresponding import command:

```
copy {data_destination} from '/path/export.txt' eol e'\n' delimiter e'\t' encoding '{file_encoding}';
-- data_destination can only be a table name.
-- file_encoding indicates the encoding format used during binary file export.
```

Example:

Step 1 Prepare data.

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is\C  recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# create table test_copy(id int, name text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(3, e'cc\tc');
INSERT 0 1
db1=# insert into test_copy(name) values('ddd');
INSERT 0 1
db1=# insert into test_copy values(5, e'ee\\e');
INSERT 0 1
db1=# insert into test_copy values(6, e',');
INSERT 0 1
db1=# insert into test_copy values(7, e'"');
INSERT 0 1
db1=# select * from test_copy;
 id |   name
----+-----------
  1 | aaa
  3 | cc      c
    | ddd
  5 | ee\e
  6 | ,
  7 | "
(6 rows)
```

Step 2 Export data.

```
db1=# copy test_copy to '/home/xy/test.txt' eol e'\n' delimiter e'\t' encoding 'UTF-8';
COPY 6
```

Step 3 Import data.

```
db1=# truncate test_copy;
TRUNCATE TABLE
db1=# copy test_copy from '/home/xy/test.txt' eol e'\n' delimiter e'\t' encoding 'UTF-8';
COPY 6
```

----End

Exporting Data Files from GaussDB for Manual Parsing

In this scenario, you will not want the exported TEXT files to exhibit any escape behavior specific to GaussDB. Follow these steps:

- Check for EOL characters or delimiters in the field data.
- If they exist, modify the **eol** or **delimiter** parameter to use other characters that do not appear in the field data. It is advisable to select from invisible characters (0x01 to 0x1F) for this purpose.
- You can use the **null** option to specify how NULL values should be represented during export.

- Finally, include the **without escaping** parameter to prevent escaping in the output.

Recommended export command:

```
copy {data_source} to '/path/export.txt' without escaping eol e'\x1E' delimiter e'\x1F' null '\N' encoding
'{server_encoding}';
-- data_source can be a table name or a SELECT statement.
-- server_encoding can be obtained using SHOW server_encoding.
```

NOTICE

The primary function of the escaping mechanism is to prevent special characters (such as delimiters and newline characters) in fields from damaging the file structure. When choosing to disable the escaping mechanism, be sure to isolate special characters by carefully choosing delimiters and newline characters for non-escaping scenarios and ensuring that these characters are absent in the data content. This is essential for non-escaped file parsing, as any character conflicts can lead to data parsing failure or structural disorder.

Example:

Step 1 Prepare data.

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# create table test_copy(id int, name text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(3, e'cc\tc');
INSERT 0 1
db1=# insert into test_copy(name) values('ddd');
INSERT 0 1
db1=# insert into test_copy values(5, e'ee\\e');
INSERT 0 1
db1=# insert into test_copy values(6, e',');
INSERT 0 1
db1=# insert into test_copy values(7, e'"');
INSERT 0 1
db1=# select * from test_copy;
 id |   name
----+-----------
  1 | aaa
  3 | cc      c
    | ddd
  5 | ee\e
  6 | ,
  7 | "
(6 rows)
```

Step 2 Export data.

```
db1=# copy test_copy to '/home/xy/test.txt' without escaping eol e'\x1E' delimiter e'\x1F' null '\N' encoding
'UTF-8';
COPY 6
```

Step 3 Read data. In this step, data is first split into rows based on the selected EOL character (0x1E). Then each row of data stream is further divided based on the delimiter (0x1F) to extract the field values within each row.

----End

Importing User-Created Data Files into GaussDB

If you only have raw field data without a complete data file, you can create a data file manually. To minimize the need for extensive modifications—such as escaping or other special character processing—it is advisable to use the TEXT format. When creating a data file in TEXT format, you can select custom delimiters and EOL characters to ensure accurate data import.

Step 1 Select an EOL character. If the field data contains 0x0A, do not use 0x0A as the EOL character. Otherwise, newlines within the field data will be misinterpreted as EOL characters, resulting in a single line of data being split into two. To prevent this issue, ensure that the selected EOL character is not present in the field data. For instance, consider using 0x1E as the EOL character when it is not found in the field data. It is advisable to select from invisible characters (0x01 to 0x1F) for this purpose.

Step 2 Select a delimiter. If the field data contains tab characters, do not use them as delimiters. Otherwise, tabs within the field data will be misinterpreted as delimiters, resulting in a single field being split into two. To prevent this issue, ensure that the selected delimiter is not present in the field data. For instance, consider using 0x1F as the delimiter when it is not found in the field data. It is advisable to select from invisible characters (0x01 to 0x1F) for this purpose.

Step 3 Build data. Create a data file using the EOL character and delimiter selected in steps 1 and 2. Check the character set of the server. You need to generate a data file that matches the server's character set.

Step 4 Import data. In this example, the character set of the data file is UTF-8.

```
db1=# copy test_copy to '/home/xy/test.txt' without escaping eol e'\x1E' delimiter e'\x1F' null '\N' encoding
'UTF-8';
COPY 6
```

----End

11.1.1.5 Importing and Exporting Data Files on a GSQL Client

When you execute the **COPY** command to export data on a GSQL client, the generated data files are saved on the database server by default. This can make it difficult for users to access the files. To offer easier file access, the **\COPY** command directly generates data files on the local client.

Differences Between COPY and \COPY

1. File location: The files generated and read during COPY import are located on the server. Conversely, in \COPY import, both the generated and read files are located on the client.
2. Performance: To complete \COPY import, the client must read file streams and transmit them to the server. This results in decreased performance in comparison to COPY.

3. Functionality: Compared to COPY, \COPY offers an additional capability—client-based parallel import. For detailed specifications and limitations, consult "Database Connection Tools > gsql for Connecting to a Database > Meta-Command Reference" in *Tool Reference*.

Example \COPY Commands

To export with **\COPY**, simply replace **COPY** with **\COPY** in your commands. Below is a straightforward example of converting CSV export commands from **COPY** to their **\COPY** equivalents:

```
-- COPY commands:
COPY {data_source} to '/path/export.csv' encoding {server_encoding} csv;
COPY {data_source} from '/path/export.csv' encoding {server_encoding} csv;
-- Corresponding \COPY commands:
\COPY {data_source} to '/path/export.csv' encoding {server_encoding} csv;
\COPY {data_source} from '/path/export.csv' encoding {server_encoding} csv;
```

Example Commands for Parallel Import

```
-- Import command for the CSV format:
\COPY {data_destination} from '/path/export.txt' encoding {file_encoding} parallel {parallel_num} csv;
-- Import command for the FIXED format:
\COPY {data_destination} from '/path/export.txt' encoding {file_encoding} parallel {parallel_num} fixed;
-- Import command for the TEXT format:
\COPY {data_destination} from '/path/export.txt' encoding {file_encoding} parallel {parallel_num};
-- data_destination can only be a table name.
-- file_encoding indicates the encoding format used during binary file export.
-- parallel_num indicates the number of clients for data import. When there are sufficient cluster
resources, it is advisable to set it to 8.
```

11.1.1.6 Importing and Exporting Data Through the JDBC Driver

In the presence of the JDBC driver, you can call its CopyManager APIs to implement data import and export. CopyManager can import data into tables and export data from databases in batches.

11.1.1.7 Importing Erroneous Data Through Error Tolerance

Both the **COPY** and **\COPY** commands will halt the data import process upon detecting any data exceptions. To overcome this limitation, GaussDB provides two error tolerance modes: intelligent correction mode and strict verification mode. The strict verification mode (Level 1 error tolerance) is preferred because it can skip abnormal records, maintaining data integrity and minimizing the impact on import performance. For details, see **Guide to Importing Erroneous Data**.

Below are import commands executed in strict verification mode (Level 1 error tolerance):

```
-- CSV format
\COPY {data_destination} from '/path/export.txt' log errors reject limit '{limit_num}' encoding
{file_encoding} CSV;
-- BINARY format
\COPY {data_destination} from '/path/export.txt' log errors reject limit '{limit_num}' encoding
{file_encoding} BINARY;
-- FIXED format
\COPY {data_destination} from '/path/export.txt' log errors reject limit '{limit_num}' encoding
{file_encoding} FIXED;
-- TEXT format
\COPY {data_destination} from '/path/export.txt' log errors reject limit '{limit_num}' encoding
```

```
{file_encoding};
-- data_destination can only be a table name.
-- file_encoding indicates the encoding format used during binary file export.
-- limit_num specifies the maximum number of erroneous rows that the COPY FROM statement can
tolerate during data import. If this limit is exceeded, errors will be reported as usual according to the
original mechanism.
```

NOTICE

To use the error tolerance feature, regular users require permissions on the two system catalogs of the feature. Execute the following SQL statements to grant them these permissions:

```
grant insert,select,delete on pg_catalog.gs_copy_error_log to {user_name};
```

11.1.2 Guide to Exporting Erroneous Data

Data errors during export typically occur when character strings or binary data that does not match the server-side encoding is inserted into the database. To address this, you are advised to keep the client-side encoding consistent with the server-side encoding, eliminating the need for validity checks against the server-side encoding and data transcoding.

Encoding Consistency Principle During Export

1. When the client-side encoding is consistent with the server-side encoding:
   - Native data is exported.
   - Data integrity and originality are guaranteed.
   - Character set conversion is not required.
2. When the client-side encoding is inconsistent with the server-side encoding:
   - The client-side encoding is employed as the target encoding standard for the exported files.
   - The kernel first checks existing data against the server-side encoding. Upon detecting any data encoded in an illegal format, it will report an error.
   - The kernel then proceeds to transcode the data. If it encounters any characters that cannot be transcoded (due to code bits present in the source character set but not in the target character set), it will report an error.

Solutions to Illegal Encoding

If your database contains any data encoded in an illegal format and you wish to export the data without triggering an error, consider the following methods:

Preferred method: Keep the client-side encoding consistent with the server-side encoding. Then export data using the server-side encoding without performing any transcoding.

Step 1 Query the database server-side encoding.

```
gaussdb=# show server_encoding;
```

Step 2 Query the database client-side encoding.

```
gaussdb=# show client_encoding;
```

Step 3 Keep the client-side encoding consistent with the server-side encoding.

```
gaussdb=# set client_encoding = '{server_encoding}';
```

Step 4 Execute **COPY** to export data to a file in standard CSV format.

```
gaussdb=# COPY test_copy TO '/data/test_copy.csv' CSV;
```

----End

Alternative solution: Use placeholders ('?') to replace any bytes encoded in an illegal format. This solution depends on the transcoding capability of the database kernel and will alter the content of the exported data.

Step 1 Query the database server-side encoding.

```
gaussdb=# show server_encoding;
```

Step 2 Set the database client-side encoding as the target encoding.

```
gaussdb=# set client_encoding = {target_encoding};
```

Step 3 Leverage the kernel's transcoding capability to replace any bytes encoded in an illegal format while exporting data.

```
gaussdb=# COPY test_copy TO '/data/test_copy.csv' CSV COMPATIBLE_ILLEGAL_CHARS;
```

----End

NOTICE

- Consider enabling the **COMPATIBLE_ILLEGAL_CHARS** parameter to correct any exported data encoded in an illegal format during export, while keeping the data in the database unchanged. Kindly use this parameter as necessary.
- When enabled, the **COMPATIBLE_ILLEGAL_CHARS** parameter:
  - Replaces illegal characters with the ones specified in the **convert_illegal_char_mode** parameter. The default replacement character is **'?'** (U+003F).
  - Replaces zero characters (U+0000) with spaces (U+0020). If you do not need this replacement, configure the zero-character functionality in different compatible modes.
- For detailed limitations on the **COMPATIBLE_ILLEGAL_CHARS** parameter, refer to the **COMPATIBLE_ILLEGAL_CHARS** description in the COPY section.

Typical Scenario Examples

The basic processing logic for independent zero characters is simple. This document focuses on streamlining the error tolerance process for complex exception scenarios where both zero characters (\0) and characters encoded in an illegal format are present in data streams.

1. Build UTF-8 zero characters and illegal characters.

```
gaussdb=# create database db_utf8 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE
='en_US.UTF-8' dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c db_utf8
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db_utf8" as user "omm".
```

```
db_utf8=# create table test_encodings(id int, content text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
db_utf8=# insert into test_encodings values(1,
dbe_raw.cast_to_varchar2(dbe_raw.concat(hextoraw('2297'),
    dbe_raw.cast_from_varchar2_to_raw('Import/Export'))));
INSERT 0 1
db_utf8=# show client_encoding;
 client_encoding
-----------------
 UTF8
(1 row)
-- The content in row 1 includes zero characters, while that in row 2 includes characters that are not
encoded by UTF-8.
db_utf8=# select *, dbe_raw.cast_from_varchar2_to_raw(content) from test_encodings;
 id |  content  |  cast_from_varchar2_to_raw
----+-----------+------------------------------
  1 | "Import/Export | 2297E5AFBCE585A5E5AFBCE587BA
(1 row)
```

2. Selecting the server's character set for file export allows for direct export without transcoding. However, if a different character set is selected, transcoding will be required. During transcoding, the system will report an error upon detecting the illegal UTF-8 character 0x97. In this situation, simply enable the **compatible_illegal_chars** parameter to ensure successful file export.

```
db_utf8=# copy test_encodings to '/home/xy/encodings.txt.utf8' encoding 'utf-8';
COPY 1
db_utf8=# copy test_encodings to '/home/xy/encodings.txt.gb18030' encoding 'gb18030';
ERROR:  invalid byte sequence for encoding "UTF8": 0x97
db_utf8=# copy test_encodings to '/home/xy/encodings.txt.gb18030' encoding 'gb18030'
compatible_illegal_chars;
COPY 1
```

3. Open the **/home/xy/encodings.txt.utf8** file with UTF-8 encoding. In this example, the **support_zero_character** option and **compatible_illegal_chars** parameter are disabled. You will notice that there are garbled characters in the second column of the first row. Although no explicit exception is shown, the **hexdump** command reveals the presence of garbled characters. You can refer to this example to reproduce the problem, but specific data details are not provided here.

```
1    "Import/Export
```

4. Open the **/home/xy/encodings.txt.gb18030** file with GB18030 encoding. You will notice that the illegal character in the second column of the first row has been replaced by a question mark (?).

```
1    "?Import/Export
```

11.1.3 Guide to Importing Erroneous Data

The error tolerance mechanism during data import provides two modes.

Intelligent Correction Mode (Adaptive Import)

- Principle: Prioritize data integrity and leverage intelligent correction to ensure that the highest possible volume of accurate data is imported.
- Scenario: Apply this mode when the imported data contains column count exceptions (extra columns) or character exceptions.
- Procedure:
  a. Rectify the column count exceptions by truncating extra columns.

   b. Transcode the character sets and clean illegal characters.
   c. Write all data into the target database.

- Output: corrected data set, along with logs recorded by GaussDB for character exceptions
- Example:

   a. Extra columns: The number of data columns is greater than that of table columns. In this situation, you can specify the **ignore_extra_data** option in the COPY statement, and GaussDB will import data from the correct number of columns into the table while discarding the extra columns.

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE
='en_US.UTF-8' dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# create table test_copy(id int, content text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by
default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
db1=# copy test_copy from stdin delimiter ',';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1,Import,Export
>>\.
ERROR:  extra data after last expected column
CONTEXT:  COPY test_copy, line 1: "1,Import,Export"
-- When ignore_extra_data is not specified, the import fails. However, specifying it ensures a
successful import.
db1=# copy test_copy from stdin delimiter ',' ignore_extra_data;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1,Import,Export
>> \.
COPY 1
db1=# select * from test_copy;
 id | content
----+---------
  1 | Import
(1 row)
```

   b. Character exceptions: When dealing with character exceptions, take appropriate actions based on the consistency of server-side encoding with client-side encoding.

      ▪ Consistent encoding: If the data being imported contains any characters encoded in an illegal format, consider setting the GUC parameter **copy_special_character_version** to **'no_error'** for error tolerance. In this setting, GaussDB will accept data that does not comply with the encoding format. Instead of reporting an error, it will directly insert the data into the table according to the original encoding format.

        For details, see the example data file, **/home/xy/encodings.txt.utf8**, generated in **Guide to Exporting Erroneous Data**. To simulate the scenario where files with encoding exceptions are not transcoded during import, you can create a database with UTF-8 encoding.

```
gaussdb=# create database db_utf8 encoding='UTF-8' LC_COLLATE='en_US.UTF-8'
LC_CTYPE ='en_US.UTF-8' dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c db_utf8
Non-SSL connection (SSL connection is recommended when requiring high-security)
```

```
You are now connected to database "db_utf8" as user "omm".
db_utf8=# create table test_encodings(id int, content text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column
by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
db_utf8=# show copy_special_character_version;
 copy_special_character_version
--------------------------------

(1 row)
db_utf8=# copy test_encodings from '/home/omm/temp/encodings.txt.utf8';
ERROR:  invalid byte sequence for encoding "UTF8": 0x97
CONTEXT:  COPY test_encodings, line 2
db_utf8=# set copy_special_character_version = 'no_error';
SET
db_utf8=# copy test_encodings from '/home/xy/encodings.txt.utf8';
COPY 2
db_utf8=# select * from test_encodings;
 id |  content
----+-----------
  1 | Import
  2 | "Import/Export
(2 rows)
```

- Inconsistent encoding: Transcoding is required if the server-side encoding does not match the client-side encoding. COPY implements transcoding through the **compatible_illegal_chars** parameter. If illegal characters are imported into GaussDB, the error tolerance mechanism will convert them and store the resulting characters in GaussDB. The entire import process will proceed without any errors or interruptions. This ensures efficient and stable data import, even in complex environments with inconsistent encoding.

Strict Verification Mode (Precise Import)

- Principle: Prioritize data accuracy and ensure the standardization of imported data.
- Scenario: Apply this mode in fields that demand strict data accuracy, particularly for sensitive data like medical records and financial transactions. This mode helps alleviate concerns about intelligent correction potentially compromising data accuracy.
- Procedure:
  a. Conduct a series of verifications at multiple levels (column count exceptions, character exceptions, data type conversion exceptions, and constraint conflicts).
  b. Generate an error diagnosis report (which includes row numbers, error types, and error data).
  c. Create an erroneous data isolation area.
  d. Only import the original data that successfully passes these verifications into the database.
- Output: pure data set and error details report (further details available in pg_catalog.pgxc_copy_error_log)
- Error tolerance level: applies to all the exceptions processed by the intelligent correction mode, including extra columns, data type conversion errors, overlong columns, and transcoding exceptions. The process is as follows:

```
-- When the number of data type errors during data import does not exceed 100, no error will be
reported, and GaussDB will proceed to the next row. However, if the number exceeds 100, an error
```

```
will be reported. The details and row number of the erroneous data will be recorded in the
gs_copy_error_log table.
gaussdb=# copy test_copy from '/home/omm/temp/test.csv' log errors reject limit '100' csv;
-- In comparison to the previous statement, the next one includes an additional action: recording all
data from the erroneous row in gs_copy_error_log. This action is recommended when there is no risk
to data security. Note that SYSADMIN permissions are required.
gaussdb=# copy test_copy from '/home/omm/temp/test.csv' log errors data reject limit '100' csv;
```

Conclusion

To optimize data import processes, you can combine the intelligent correction mode with the strict verification mode, with the intelligent correction mode taking priority. When intelligent correction is enabled during a COPY import and it corrects a data item, strict verification will not be triggered for that specific row. Consequently, the error table will not record relevant data, nor will it update **reject limit**. Before importing data, carefully evaluate whether to automatically correct column and character exceptions or discard them, depending on specific requirements.

11.2 Best Practices for Import and Export Using COPY (Centralized Instances)

The COPY syntax is a useful tool for exporting data from GaussDB to data files in CSV, BINARY, FIXED, or TEXT format. It can also import data from the four types of files to a specified table. For details about the COPY syntax, see "SQL Reference > SQL Syntax > C > COPY" in *Developer Guide*.

11.2.1 Typical Scenarios

Preparations

During import or export using **COPY**, the database requires file read/write permissions on the server. To grant these permissions, enable the **enable_copy_server_files** parameter.

```
gs_guc reload -I all -N all -Z datanode -c "enable_copy_server_files=on"
```

NOTICE

- Parameters in the COPY import command must match the actual data in files, including the format, delimiter, newline character, and character set.
- If the specified client-side encoding matches the server-side encoding, the **COPY** command preserves the binary form of data during import and export, preventing any structural damage that may result from transcoding. This mechanism guarantees the integrity and originality of data migration, making it suitable for scenarios that demand high fidelity.

11.2.1.1 Using the Recommended CSV Format

In CSV format, each file is divided into multiple records using end-of-line (EOL) characters. Each record is further split into multiple fields using delimiters, which are commas by default. Each field can be enclosed within a pair of quote characters, which are double quotation marks by default. This eliminates the need to escape special characters such as EOL characters and delimiters within the field content. Furthermore, it is a widely recognized standard with exceptional cross-platform compatibility and cross-industry universality, solidifying its position as the top recommended format.

Recommended export command:

```
COPY {data_source} TO '/path/export.csv' delimiter ',' quote '"' escape '"' encoding {server_encoding} csv;
-- data_source can be a table name or a SELECT statement.
-- server_encoding can be obtained using SHOW server_encoding.
```

Corresponding import command:

```
COPY {data_destination} FROM '/path/export.csv' delimiter ',' quote '"' escape '"' encoding {file_encoding}
csv;
-- data_destination can only be a table name.
-- file_encoding indicates the encoding format used during file export.
```

NOTICE

To import a manually created CSV data file into the database, ensure that the file complies with CSV standards. Additionally, specify the correct parameters, including delimiters, quote characters, and EOL characters, in the **COPY** command.

Examples

Step 1 Prepare data.

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'A';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# CREATE TABLE test_copy(id int, name text);
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(2, e'bb\nb');
INSERT 0 1
db1=# insert into test_copy values(3, e'cc\tc');
INSERT 0 1
db1=# insert into test_copy(name) values('ddd');
INSERT 0 1
db1=# insert into test_copy values(5, e'ee\\e');
INSERT 0 1
db1=# insert into test_copy values(6, ',');
INSERT 0 1
db1=# insert into test_copy values(7, '"');
INSERT 0 1
db1=#  SELECT * FROM test_copy;
 id |   name
----+-----------
  1 | aaa
  2 | bb       +
    | b
  3 | cc      c
    | ddd
  5 | ee\e
  6 | ,
  7 | "
(7 rows)
```

Step 2 Export data from the entire table.

```
db1=# copy test_copy to '/home/xy/test.csv' delimiter ',' quote '"' escape '"' encoding 'UTF-8' csv;
COPY 7
```

The content of the exported CSV file is as follows:

```
1,aaa
2,"bb
b"
3,cc    c
,ddd
5,ee\e
6,","
7,""""""
```

Step 3 Import data.

```
db1=# truncate test_copy;
TRUNCATE TABLE
db1=# copy test_copy from '/home/xy/test.csv' delimiter ',' quote '"' escape '"' encoding 'UTF-8' csv;
COPY 7
```

Step 4 (Custom data set export) Export the **name** column for all rows in **test_copy**, excluding those with an empty ID.

```
db1=# copy (select name from test_copy where id is not null) to '/home/xy/test.csv' delimiter ',' quote '"'
escape '"' encoding 'UTF-8' csv;
COPY 6
```

The content of the exported CSV file is as follows:

```
aaa
"bb
b"
cc    c
ee\e
","
""""""
```

----End

11.2.1.2 Importing and Exporting Data with Extreme Performance

When there are strict demands for data import and export performance, and the import and export occur between database instances of the same version, you can use the BINARY format to store and read data as binary numbers other than regular text. Despite offering performance advantages over other formats, this format also comes with drawbacks, such as:

1. The format is specific to GaussDB, making it non-portable. It is advisable to use this format only for importing and exporting data between databases of the same version.
2. The BINARY format is tightly coupled with specific data types. For example, while the TEXT format allows exporting data from a smallint field and importing it into an integer column, this is not possible with the BINARY format.
3. Certain data types cannot be imported or exported using the BINARY format. For details, see BINARY limitations.

Recommended export command:

```
set client_encoding = '{server_encoding}';
copy {data_source} to '/path/export.bin' binary;
```

```
-- data_source can be a table name or a SELECT statement.
-- server_encoding can be obtained using SHOW server_encoding.
```

Corresponding import command:

```
set client_encoding = '{file_encoding}';
copy {data_destination} from '/path/export.bin' binary;
-- data_destination can only be a table name.
-- file_encoding indicates the encoding format used during binary file export.
```

NOTICE

Before choosing this format, carefully review its limitations. Use it to enhance import and export performance only when you are completely certain that these limitations will not impact the data to be exported.

Examples

Step 1 Prepare data.

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'A';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# CREATE TABLE test_copy(id int, name text);
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(3, e'cc\tc');
INSERT 0 1
db1=# insert into test_copy(name) values('ddd');
INSERT 0 1
db1=# insert into test_copy values(5, e'ee\\e');
INSERT 0 1
db1=# insert into test_copy values(6, ',');
INSERT 0 1
db1=# insert into test_copy values(7, '"');
INSERT 0 1
db1=# SELECT * FROM test_copy;
 id |   name
----+-----------
  1 | aaa
  3 | cc      c
    | ddd
  5 | ee\e
  6 | ,
  7 | "
(6 rows)
```

Step 2 Export data.

```
db1=# set client_encoding = 'UTF-8';
SET
db1=# COPY test_copy TO '/home/xy/test.bin' BINARY;
COPY 6
```

Step 3 Import data.

```
db1=# truncate test_copy;
TRUNCATE TABLE
db1=# set client_encoding = 'UTF-8';
SET
```

```
db1=# copy test_copy from '/home/xy/test.bin' BINARY;
COPY 6
```

----End

11.2.1.3 Exporting Data Files for Manual Parsing

The FIXED format maintains a fixed data structure to simplify file parsing: Each row represents a record, and within each row, the starting offset and length of each field value are fixed. Therefore, it is advisable to utilize the FIXED format when you need to manually parse a data file exported from the database. Despite this, the FIXED format has its limitations, as detailed in FIXED description.

Recommended export command:

```
COPY {data_source} FROM '/path/export.fixed' encoding {server_encoding} FIXED
FORMATTER(col1_name(col1_offset, col1_length), col2_name(col2_offset, col2_length));
-- data_source can be a table name or a SELECT statement.
-- server_encoding can be obtained using SHOW server_encoding.
-- col1_name(col1_offset, col1_length) indicates that the data named col1_name in each row of the data
file starts at the position with an offset of col1_offset and extends for a length of col1_length.
```

Corresponding import command:

```
COPY {data_destination} from '/path/export.fixed' encoding {file_encoding} FIXED
FORMATTER(col1_name(col1_offset, col1_length), col2_name(col2_offset, col2_length));
-- data_destination can only be a table name.
-- file_encoding indicates the encoding format used during file export.
-- col1_name(col1_offset, col1_length) indicates that the data named col1_name in each row of the data
file starts at the position with an offset of col1_offset and extends for a length of col1_length.
```

NOTICE

The fixed column width format has limitations: It requires maintaining a relatively uniform width for each column in the table, allowing for the selection of an appropriate fixed **col_length** to prevent data truncation and eliminate unnecessary spaces in all columns. In cases where column widths vary significantly or change dynamically, it is advisable to prioritize flexible formats such as TEXT or CSV to ensure data integrity and efficient use of storage space.

Examples

Step 1 Prepare data.

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'A';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# CREATE TABLE test_copy(id int, name text);
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(2, 'bb"b');
INSERT 0 1
db1=# insert into test_copy values(3, 'cc  c');
INSERT 0 1
db1=# insert into test_copy values('', e'dd\td');
INSERT 0 1
db1=# insert into test_copy values('5', e'ee\e');
```

```
INSERT 0 1
db1=# select * from test_copy;
 id |   name
----+-----------
  1 | aaa
  2 | bb"b
  3 | cc  c
    | dd      d
  5 | eee
(5 rows)
```

Step 2 Export data.

```
db1=# COPY test_copy TO '/home/xy/test.fixed' encoding 'UTF-8' FIXED FORMATTER(id(0,1), name(1,5));
COPY 5
```

The content of the exported data file is as follows:

```
1  aaa
2 bb"b
3cc  c
  dd    d
5  eee
```

Step 3 Import data.

```
db1=# truncate test_copy;
TRUNCATE TABLE
db1=# copy test_copy from '/home/xy/test.fixed' encoding 'UTF-8' FIXED FORMATTER(id(0,1), name(1,5));
COPY 5
```

----End

11.2.1.4 Importing and Exporting Data When Only the TEXT Format Is Available

In TEXT format, each file is divided into multiple records using EOL characters. Each record is further split into multiple fields using delimiters, which are tab characters ('\t') by default.

Using the TEXT format in GaussDB necessitates special logic, for example:

- The backspace (0x08), form-feed (0x0C), newline (0x0A), carriage return (0x0D), horizontal tab (0x09), and vertical tab (0x0B) characters are escaped as **'\b'**, **'\f'**, **'\n'**, **'\r'**, **'\t'**, and **'\v'**, respectively.
- By default, the EOL character is configured as the first identified **'\n'**, **'\r'**, or **'\r\n'** during import, and as **'\n'** during export.
- A single backslash is escaped as double backslashes.
- NULL values are escaped as **'\N'**.

Exporting Data from and Importing Data into GaussDB

Recommended export command:

```
copy {data_source} to '/path/export.txt' eol e'\n' delimiter e'\t' encoding '{server_encoding}';
-- data_source can be a table name or a SELECT statement.
-- server_encoding can be obtained using SHOW server_encoding.
```

Corresponding import command:

```
copy {data_destination} from '/path/export.txt' eol e'\n' delimiter e'\t' encoding '{file_encoding}';
-- data_destination can only be a table name.
-- file_encoding indicates the encoding format used during binary file export.
```

Example:

Step 1 Prepare data.

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'A';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# create table test_copy(id int, name text);
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(3, e'cc\tc');
INSERT 0 1
db1=# insert into test_copy(name) values('ddd');
INSERT 0 1
db1=# insert into test_copy values(5, e'ee\\e');
INSERT 0 1
db1=# insert into test_copy values(6, e',');
INSERT 0 1
db1=# insert into test_copy values(7, e'"');
INSERT 0 1
db1=# select * from test_copy;
 id |   name
----+-----------
  1 | aaa
  3 | cc      c
    | ddd
  5 | ee\e
  6 | ,
  7 | "
(6 rows)
```

Step 2 Export data.

```
db1=# copy test_copy to '/home/xy/test.txt' eol e'\n' delimiter e'\t' encoding 'UTF-8';
COPY 6
```

Step 3 Import data.

```
db1=# truncate test_copy;
TRUNCATE TABLE
db1=# copy test_copy from '/home/xy/test.txt' eol e'\n' delimiter e'\t' encoding 'UTF-8';
COPY 6
```

----End

Exporting Data Files from GaussDB for Manual Parsing

In this scenario, you will not want the exported TEXT files to exhibit any escape behavior specific to GaussDB. Follow these steps:

- Check for EOL characters or delimiters in the field data.
- If they exist, modify the **eol** or **delimiter** parameter to use other characters that do not appear in the field data. It is advisable to select from invisible characters (0x01 to 0x1F) for this purpose.
- You can use the **null** option to specify how NULL values should be represented during export.
- Finally, include the **without escaping** parameter to prevent escaping in the output.

Recommended export command:

```
copy {data_source} to '/path/export.txt' without escaping eol e'\x1E' delimiter e'\x1F' null '\N' encoding
'{server_encoding}';
```

```
-- data_source can be a table name or a SELECT statement.
-- server_encoding can be obtained using SHOW server_encoding.
```

NOTICE

The primary function of the escaping mechanism is to prevent special characters (such as delimiters and newline characters) in fields from damaging the file structure. When choosing to disable the escaping mechanism, be sure to isolate special characters by carefully choosing delimiters and newline characters for non-escaping scenarios and ensuring that these characters are absent in the data content. This is essential for non-escaped file parsing, as any character conflicts can lead to data parsing failure or structural disorder.

Example:

Step 1 Prepare data.

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'A';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# create table test_copy(id int, name text);
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(3, e'cc\tc');
INSERT 0 1
db1=# insert into test_copy(name) values('ddd');
INSERT 0 1
db1=# insert into test_copy values(5, e'ee\\e');
INSERT 0 1
db1=# insert into test_copy values(6, e',');
INSERT 0 1
db1=# insert into test_copy values(7, e'"');
INSERT 0 1
db1=# select * from test_copy;
 id |   name
----+-----------
  1 | aaa
  3 | cc      c
    | ddd
  5 | ee\e
  6 | ,
  7 | "
(6 rows)
```

Step 2 Export data.

```
db1=# copy test_copy to '/home/xy/test.txt' without escaping eol e'\x1E' delimiter e'\x1F' null '\N' encoding
'UTF-8';
COPY 6
```

Step 3 Read data. In this step, data is first split into rows based on the selected EOL character (0x1E). Then each row of data stream is further divided based on the delimiter (0x1F) to extract the field values within each row.

----End

Importing User-Created Data Files into GaussDB

If you only have raw field data without a complete data file, you can create a data file manually. To minimize the need for extensive modifications—such as escaping or other special character processing—it is advisable to use the TEXT format. When creating a data file in TEXT format, you can select custom delimiters and EOL characters to ensure accurate data import.

Step 1 Select an EOL character. If the field data contains 0x0A, do not use 0x0A as the EOL character. Otherwise, newlines within the field data will be misinterpreted as EOL characters, resulting in a single line of data being split into two. To prevent this issue, ensure that the selected EOL character is not present in the field data. For instance, consider using 0x1E as the EOL character when it is not found in the field data. It is advisable to select from invisible characters (0x01 to 0x1F) for this purpose.

Step 2 Select a delimiter. If the field data contains tab characters, do not use them as delimiters. Otherwise, tabs within the field data will be misinterpreted as delimiters, resulting in a single field being split into two. To prevent this issue, ensure that the selected delimiter is not present in the field data. For instance, consider using 0x1F as the delimiter when it is not found in the field data. It is advisable to select from invisible characters (0x01 to 0x1F) for this purpose.

Step 3 Build data. Create a data file using the EOL character and delimiter selected in steps 1 and 2. Check the character set of the server. You need to generate a data file that matches the server's character set.

Step 4 Import data. In this example, the character set of the data file is UTF-8.

```
db1=# copy test_copy to '/home/xy/test.txt' without escaping eol e'\x1E' delimiter e'\x1F' null '\N' encoding
'UTF-8';
COPY 6
```

----End

11.2.1.5 Importing and Exporting Data Files on a GSQL Client

When you execute the **COPY** command to export data on a GSQL client, the generated data files are saved on the database server by default. This can make it difficult for users to access the files. To offer easier file access, the **\COPY** command directly generates data files on the local client.

Differences Between COPY and \COPY

1. File location: The files generated and read during COPY import are located on the server. Conversely, in \COPY import, both the generated and read files are located on the client.
2. Performance: To complete \COPY import, the client must read file streams and transmit them to the server. This results in decreased performance in comparison to COPY.
3. Functionality: Compared to COPY, \COPY offers an additional capability—client-based parallel import. For detailed specifications and limitations, consult "Database Connection Tools > gsql for Connecting to a Database > Meta-Command Reference" in *Tool Reference*.

Example \COPY Commands

To export with **\COPY**, simply replace **COPY** with **\COPY** in your commands. Below is a straightforward example of converting CSV export commands from **COPY** to their **\COPY** equivalents:

```
-- COPY commands:
COPY {data_source} to '/path/export.csv' encoding {server_encoding} csv;
COPY {data_source} from '/path/export.csv' encoding {server_encoding} csv;
-- Corresponding \COPY commands:
\COPY {data_source} to '/path/export.csv' encoding {server_encoding} csv;
\COPY {data_source} from '/path/export.csv' encoding {server_encoding} csv;
```

Example Commands for Parallel Import

```
-- Import command for the CSV format:
\COPY {data_destination} from '/path/export.txt' encoding {file_encoding} parallel {parallel_num} csv;
-- Import command for the FIXED format:
\COPY {data_destination} from '/path/export.txt' encoding {file_encoding} parallel {parallel_num} fixed;
-- Import command for the TEXT format:
\COPY {data_destination} from '/path/export.txt' encoding {file_encoding} parallel {parallel_num};
-- data_destination can only be a table name.
-- file_encoding indicates the encoding format used during binary file export.
-- parallel_num indicates the number of clients for data import. When there are sufficient cluster
resources, it is advisable to set it to 8.
```

11.2.1.6 Importing and Exporting Data Through the JDBC Driver

In the presence of the JDBC driver, you can call its CopyManager APIs to implement data import and export. CopyManager can import data into tables and export data from databases in batches.

11.2.1.7 Importing Erroneous Data Through Error Tolerance

Both the **COPY** and **\COPY** commands will halt the data import process upon detecting any data exceptions. To overcome this limitation, GaussDB provides two error tolerance modes: intelligent correction mode and strict verification mode. The strict verification mode (Level 1 error tolerance) is preferred because it can skip abnormal records, maintaining data integrity and minimizing the impact on import performance. For details, see **Guide to Importing Erroneous Data**.

Below are import commands executed in strict verification mode (Level 1 error tolerance):

```
-- CSV format
\COPY {data_destination} from '/path/export.txt' log errors reject limit '{limit_num}' encoding
{file_encoding} CSV;
-- BINARY format
\COPY {data_destination} from '/path/export.txt' log errors reject limit '{limit_num}' encoding
{file_encoding} BINARY;
-- FIXED format
\COPY {data_destination} from '/path/export.txt' log errors reject limit '{limit_num}' encoding
{file_encoding} FIXED;
-- TEXT format
\COPY {data_destination} from '/path/export.txt' log errors reject limit '{limit_num}' encoding
{file_encoding};
-- data_destination can only be a table name.
-- file_encoding indicates the encoding format used during binary file export.
-- limit_num specifies the maximum number of erroneous rows that the COPY FROM statement can
tolerate during data import. If this limit is exceeded, errors will be reported as usual according to the
original mechanism.
```

NOTICE

To use the error tolerance feature, regular users require permissions on the two system catalogs of the feature. Execute the following SQL statements to grant them these permissions:

```
grant insert,select,delete on pgxc_copy_error_log to {user_name};
grant insert,select,delete on gs_copy_summary to {user_name};
```

11.2.2 Guide to Exporting Erroneous Data

Data errors during export typically occur when character strings or binary data that does not match the server-side encoding is inserted into the database. To address this, you are advised to keep the client-side encoding consistent with the server-side encoding, eliminating the need for validity checks against the server-side encoding and data transcoding.

Encoding Consistency Principle During Export

1. When the client-side encoding is consistent with the server-side encoding:
   - Native data is exported.
   - Data integrity and originality are guaranteed.
   - Character set conversion is not required.
2. When the client-side encoding is inconsistent with the server-side encoding:
   - The client-side encoding is employed as the target encoding standard for the exported files.
   - The kernel first checks existing data against the server-side encoding. Upon detecting any data encoded in an illegal format, it will report an error.
   - The kernel then proceeds to transcode the data. If it encounters any characters that cannot be transcoded (due to code bits present in the source character set but not in the target character set), it will report an error.

Solutions to Illegal Encoding

If your database contains any data encoded in an illegal format and you wish to export the data without triggering an error, consider the following methods:

Preferred method: Keep the client-side encoding consistent with the server-side encoding. Then export data using the server-side encoding without performing any transcoding.

Step 1 Query the database server-side encoding.

```
gaussdb=# show server_encoding;
```

Step 2 Query the database client-side encoding.

```
gaussdb=# show client_encoding;
```

Step 3 Keep the client-side encoding consistent with the server-side encoding.

```
gaussdb=# set client_encoding = '{server_encoding}';
```

Step 4 Execute **COPY** to export data to a file in standard CSV format.

```
gaussdb=# COPY test_copy TO '/data/test_copy.csv' CSV;
```

----End

Alternative solution: Use placeholders ('?') to replace any bytes encoded in an illegal format. This solution depends on the transcoding capability of the database kernel and will alter the content of the exported data.

Step 1 Query the database server-side encoding.

```
gaussdb=# show server_encoding;
```

Step 2 Set the database client-side encoding as the target encoding.

```
gaussdb=# set client_encoding = {target_encoding};
```

Step 3 Leverage the kernel's transcoding capability to replace any bytes encoded in an illegal format while exporting data.

```
gaussdb=# COPY test_copy TO '/data/test_copy.csv' CSV COMPATIBLE_ILLEGAL_CHARS;
```

----End

NOTICE

- Consider enabling the **COMPATIBLE_ILLEGAL_CHARS** parameter to correct any exported data encoded in an illegal format during export, while keeping the data in the database unchanged. Kindly use this parameter as necessary.
- When enabled, the **COMPATIBLE_ILLEGAL_CHARS** parameter:
  - Replaces illegal characters with the ones specified in the **convert_illegal_char_mode** parameter. The default replacement character is **'?'** (U+003F).
  - Replaces zero characters (U+0000) with spaces (U+0020). If you do not need this replacement, configure the zero-character functionality in different compatible modes.
- For detailed limitations on the **COMPATIBLE_ILLEGAL_CHARS** parameter, refer to the **COMPATIBLE_ILLEGAL_CHARS** description in the COPY section.

11.2.3 Guide to Importing Erroneous Data

The error tolerance mechanism during data import provides two modes.

Intelligent Correction Mode (Adaptive Import)

- Principle: Prioritize data integrity and leverage intelligent correction to ensure that the highest possible volume of accurate data is imported.
- Scenario: Apply this mode when the imported data contains column count exceptions, including extra, missing, or deprecated columns, as well as abnormal characters.
- Procedure:
  a. Rectify the column count exceptions by truncating extra columns, padding missing columns, and discarding deprecated columns.
  b. Transcode the character sets and clean illegal characters.
  c. Write all data into the target database.

- Output: corrected data set, along with logs recorded by GaussDB for character exceptions

Strict Verification Mode (Precise Import)

- Principle: Prioritize data accuracy and enforce strict compliance when importing data into the database.
- Scenario: Apply this mode in fields that demand strict data accuracy, particularly for sensitive data like medical records and financial transactions. This mode helps alleviate concerns about **intelligent correction** potentially compromising data accuracy.
- Procedure:
  a. Conduct a series of verifications at multiple levels (column count exceptions, character exceptions, data type conversion exceptions, and constraint conflicts).
  b. Generate an error diagnosis report (which includes row numbers, error types, and error data).
  c. Create an erroneous data isolation area.
  d. Only import the original data that successfully passes these verifications into the database.
- Output: pure data set and error details report (further details available in gs_copy_error_log and gs_copy_summary)
- Error tolerance levels:
  a. Level 1 error tolerance: applies to all the exceptions processed by the intelligent correction mode, including extra data source columns, data type conversion errors, overlong columns, and transcoding exceptions. The process is as follows:
```
-- When the number of data type errors during data import does not exceed 100, no error will be reported, and GaussDB will proceed to the next row. However, if the number exceeds 100, an error will be reported. The details and row number of the erroneous data will be recorded in the gs_copy_error_log table.
gaussdb=# copy test_copy from '/home/omm/temp/test.csv' log errors reject limit '100' csv;
-- In comparison to the previous statement, the next one includes an additional action: recording all data from the erroneous row in gs_copy_error_log. This action is recommended when there is no risk to data security. Note that SYSADMIN permissions are required.
gaussdb=# copy test_copy from '/home/omm/temp/test.csv' log errors data reject limit '100' csv;
```
  b. Level 2 error tolerance: Building on Level 1, Level 2 extends support to address constraint conflicts in data, including NOT NULL constraints, conditional constraints, PRIMARY KEY constraints, UNIQUE constraints, and unique index constraints. The process is as follows:
```
-- To support a new error type, constraint conflict, while keeping the same COPY statement and error tolerance logic as Level 1, set the GUC parameter as follows:
gaussdb=# SET a_format_load_with_constraints_violation = 's2';
gaussdb=# copy test_copy from '/home/omm/temp/test.csv' log errors data reject limit '100' csv;
```

Conclusion

To optimize data import processes, you can combine the intelligent correction mode with the strict verification mode, with the intelligent correction mode taking priority. When intelligent correction is enabled during a COPY import and it corrects a data item, strict verification will not be triggered for that specific row. Consequently, the error table will not record relevant data, nor will it update **reject limit**. Before importing data, carefully evaluate whether to automatically correct column and character exceptions or discard them, depending on specific requirements.

For strict verification mode, Level 1 is recommended by default, as it effectively identifies the most common errors without compromising import performance. However, if you are operating in centralized A-compatible mode, you can consider Level 2. Be mindful that Level 2 can degrade import performance and consume additional memory resources. Therefore, you are advised not to use Level 2 by default. Enable Level 2 only when constraint type conflicts occur.

NOTE

- This default option is that error tolerance does not support constraint conflicts. To make constraint conflicts tolerated, set the session-level GUC parameter **a_format_load_with_constraints_violation** to **"s2"** and import the file again.
  - The conflicts of the NOT NULL constraints, conditional constraints, PRIMARY KEY constraints, UNIQUE constraints, and unique index constraints can be tolerated.
  - This function is valid only in centralized A-compatible mode.
  - A statement-level trigger cannot handle any constraint conflict above, so the attempt to import data into a table with such trigger will fail with an error reported.
  - Under this feature, data will be inserted row by row instead of in batches, which deteriorates the import performance.
  - Under this feature, the UB-tree indexes will be built row by row instead of in batches, degrading the index building performance.
  - This feature is still valid even if a constraint conflict is triggered by an operation on a table with a row-level trigger. Constraint conflicts of row-level triggers are implemented in sub-transactions, which use more memory resources and increase execution time. Therefore, you are advised to use this feature when constraint conflicts are very likely to occur. In this scenario, the amount of data to be imported at a time by using COPY should be less than or equal to 1 GB.

12 Best Practices for Import and Export Using Tools

12.1 Best Practices for Import and Export Using Tools (Distributed Instances)

12.1.1 Database-Level Import and Export

The gs_dump tool can back up a single database and supports four archive formats. The application scenarios of these archive formats are described in **Table 12-1**. You can choose a proper archive format as required.

For details about how to use the gs_dump tool, see "Data Import and Export Tools > gs_dump for Exporting Database Information" in *Tool Reference*.

Table 12-1 Formats of exported files

Format	Value of -F	Description	Suggestion	Import Tool
Plain-text	p	A plain-text script file that contains SQL statements and commands. The commands can be executed on gsql, a command line terminal, to rebuild database objects and load table data.	For a small-sized database or the exported SQL file needs to be modified, the plain-text format is recommended.	Before using gsql to restore database objects, you can use a text editor to edit the exported plain-text file as needed. For details about how to use the gsql tool, see "Database Connection Tools > gsql for Connecting to a Database > gsql Usage Guide" in *Tool Reference*.
Custom	c	A binary file that allows the restoration of all or selected database objects from an exported file.	For medium- or large-sized databases, the backup result needs to be exported to a single file. In this case, the custom format is recommended.	You can use gs_restore to import database objects from a custom-, directory-, or tar-format archive. For details about how to use the gs_restore tool, see "Data Import and Export Tools > gs_restore for Importing Data" in *Tool Reference*.
Directory	d	A directory containing directory files of database objects and the data files of tables and BLOBs.	For medium- or large-sized databases, database objects and data files need to be stored and exported in different directories. In this case, the directory format is recommended.	

Format	Value of -F	Description	Suggestion	Import Tool
.tar archive	t	A tar-format archive that allows the restoration of all or selected database objects from an exported file. The .tar file cannot be further compressed and has an 8-GB limitation on the size of each single file.	For a small-sized database, you need to export the archive result and pack it. In this case, the .tar format is recommended.	

NOTICE

- gs_dump does not back up global objects (roles, tablespaces, and corresponding permissions) shared with all databases. Therefore, ensure that global objects have been created on the target database or new instance before the restoration. The **-g** command of gs_dumpall can be used to export global objects and use gsql to import global objects at the target end. For details about how to use the gs_dumpall tool, see "Data Import and Export Tools > gs_dumpall for Exporting All Database Information" in *Tool Reference*.
- gs_dump and gs_restore do not support import and export across database compatibility modes. Ensure that the database compatibility mode and compatibility configuration parameters of the source and target databases are the same. For details about how to query and create a database of a specified compatibility mode, see "SQL Reference > SQL Syntax > C > CREATE DATABASE" in *Developer Guide*.
- Do not modify the files and contents exported using the **-F c**, **-F d**, or **-F t** format. Otherwise, the restoration may fail. If you need to modify or replace the file exported using the **-F p** format, edit them with caution.
- After restoration, you are advised to run **ANALYZE** on the database to provide useful statistics for the optimizer.

You are advised to run the following command as a user with the SYSADMIN permission to back up data. The source database is **my_database**, and the exported data contains data and object definitions.

```
-- Plain-text
nohup gs_dump my_database -U root -W ******** -p 8000 -F p -f /data/backup/my_database_backup.sql > /
data/backup/my_database_backup.log &
-- Custom
nohup gs_dump my_database -U root -W ******** -p 8000 -F c -f /data/backup/my_database_backup.dmp > /
data/backup/my_database_backup.log &
-- Directory
nohup gs_dump my_database -U root -W ******** -p 8000 -F d -f /data/backup/my_database_backup > /data/
backup/my_database_backup.log &
-- .tar archive
nohup gs_dump my_database -U root -W ******** -p 8000 -F t -f /data/backup/my_database_backup.tar > /
data/backup/my_database_backup.log &
```

Before restoration, you need to create a target database that has the same attributes as the source database and does not contain any data.

```
-- Run the following gsql meta-command to view the database attribute information:
\l+
-- Create a target database based on the queried attribute information.
create database my_database2 encoding='xxxxx' LC_COLLATE='xxxxx' LC_CTYPE ='xxxxx' TEMPLATE=xxx
DBCOMPATIBILITY 'xxx';
```

Run the following command as a user with the SYSADMIN permission to restore the database:

```
-- Plain-text
nohup gsql -d my_database2 -p 8000 -U root -W ******** -f /data/backup/my_database_backup.sql -a > /
data/backup/my_database_restore.log &
-- Custom
nohup gs_restore /data/backup/my_database_backup.dmp -d my_database2 -p 8000 -U root -W ******** -F c
-v > /data/backup/my_database_restore.log &
-- Directory
nohup gs_restore /data/backup/my_database_backup -d my_database2 -p 8000 -U root -W ******** -F d -v
> /data/backup/my_database_restore.log &
-- .tar archive
nohup gs_restore /data/backup/my_database_backup.tar -d my_database2 -p 8000 -U root -W ******** -F t -
v > /data/backup/my_database_restore.log &
```

12.1.2 Schema-Level Import and Export

The gs_dump tool can be used to back up a single schema and you are advised to use the gs_dump tool and the **-n** parameter to do so. Multiple **-n** parameters can be used to back up multiple schemas.

For details about how to use the gs_dump tool, see "Data Import and Export Tools > gs_dump for Exporting Database Information" in *Tool Reference*.

NOTICE

If the exported schema depends on objects that are not exported, an error message may be displayed indicating that the dependent objects are missing when the schema is imported. Therefore, ensure that the dependent objects have been created before importing the schema.

You are advised to run the following command as a user with the SYSADMIN permission to back up data. The source database is **my_database**, and the exported data contains data and object definitions.

```
-- Plain-text
nohup gs_dump my_database -U root -W ******** -p 8000 -F p -f /data/backup/my_schema_backup.sql -n
my_schema > /data/backup/my_schema_backup.log &
-- Custom
nohup gs_dump my_database -U root -W ******** -p 8000 -F c -f /data/backup/my_schema_backup.dmp -n
my_schema > /data/backup/my_schema_backup.log &
-- Directory
nohup gs_dump my_database -U root -W ******** -p 8000 -F d -f /data/backup/my_schema_backup -n
my_schema > /data/backup/my_schema_backup.log &
-- .tar archive
nohup gs_dump my_database -U root -W ******** -p 8000 -F t -f /data/backup/my_schema_backup.tar -n
my_schema > /data/backup/my_schema_backup.log &
```

Before restoration, you need to create a target database that has the same attributes as the source database and does not contain the target schema.

```
-- Run the following gsql meta-command to view the database attribute information:
\l+
-- Create a target database based on the queried attribute information.
```

```
create database my_database2 encoding='xxxxx' LC_COLLATE='xxxxx' LC_CTYPE ='xxxxx' TEMPLATE=xxx
DBCOMPATIBILITY 'xxx';
```

Run the following command as a user with the SYSADMIN permission to restore the database:

```
-- Plain-text
nohup gsql -d my_database2 -p 8000 -U root -W ******** -f /data/backup/my_schema_backup.sql -a > /data/
backup/my_schema_restore.log &
-- Custom
nohup gs_restore /data/backup/my_database_backup.dmp -d my_database2 -p 8000 -U root -W ******** -F c
-v -n my_schema > /data/backup/my_schema_restore.log &
-- Directory
nohup gs_restore /data/backup/my_database_backup -d my_database2 -p 8000 -U root -W ******** -F d -v -
n my_schema > /data/backup/my_schema_restore.log &
-- .tar archive
nohup gs_restore /data/backup/my_database_backup.tar -d my_database2 -p 8000 -U root -W ******** -F t -
v -n my_schema > /data/backup/my_schema_restore.log &
```

12.1.3 Table-Level Import and Export

Many tools are available for table-level import and export. You can select a proper tool based on the following scenarios:

1. If you need to export the definition and data of a single table to the same file, you are advised to use the plain-text archive of the gs_dump tool and the **-t** parameter. You can use multiple **-t** parameters to back up multiple tables. For details about how to use the gs_dump tool, see "Data Import and Export Tools > gs_dump for Exporting Database Information" in *Tool Reference*.

   NOTICE

   If the exported table depends on objects that are not exported, an error message may be displayed indicating that the dependent objects are missing when the table is imported. Therefore, ensure that the dependent objects have been created before importing the table.

   You are advised to run the following command as a user with the SYSADMIN permission to back up data. The source database is **my_database**, and the target table is **my_table** in **my_schema**.

   ```
   nohup gs_dump my_database -U root -W ******** -p 8000 -F p -f /data/backup/my_table_backup.sql -t
   my_schema.my_table > /data/backup/my_table_backup.log &
   ```

   Before restoration, create a target database with the same attributes as the source database, and ensure that the target schema exists in the database and no target table exists. Then, run the following command as a user with SYSADMIN permissions to restore the database:

   ```
   nohup gsql -d my_database2 -p 8000 -U root -W ******** -f /data/backup/my_table_backup.sql -a > /
   data/backup/my_table_restore.log &
   ```

2. Only the definition of a single table needs to be exported and no data in the table is required.

   NOTICE

   If the exported table depends on objects that are not exported, an error message may be displayed indicating that the dependent objects are missing when the table is imported. Therefore, ensure that the dependent objects have been created before importing the table.

You are advised to use the plain-text archive of the gs_dump tool together with the **-s** parameter. The command is as follows:

```
nohup gs_dump my_database -U root -W ******** -p 8000 -F p -f /data/backup/my_table_backup.sql -t
my_schema.my_table -s > /data/backup/my_table_backup.log &
```

Before restoration, create a target database with the same attributes as the source database, and ensure that the target schema exists in the database and no target table exists. Then, run the following command as a user with SYSADMIN permissions to restore the database:

```
nohup gsql -d my_database2 -p 8000 -U root -W ******** -f /data/backup/my_table_backup.sql -a > /
data/backup/my_table_restore.log &
```

12.2 Best Practices for Import and Export Using Tools (Centralized Instances)

12.2.1 Database-Level Import and Export

The gs_dump tool can back up a single database and supports four archive formats. The application scenarios of these archive formats are described in **Table 12-2**. You can choose a proper archive format as required.

For details about how to use the gs_dump tool, see "Data Import and Export Tools > gs_dump for Exporting Database Information" in *Tool Reference*.

Table 12-2 Formats of exported files

Format	Value of -F	Description	Suggestion	Import Tool
Plain-text	p	A plain-text script file that contains SQL statements and commands. The commands can be executed on gsql, a command line terminal, to rebuild database objects and load table data.	For a small-sized database or the exported SQL file needs to be modified, the plain-text format is recommended.	Before using gsql to restore database objects, you can use a text editor to edit the exported plain-text file as needed. For details about how to use the gsql tool, see "Database Connection Tools > gsql for Connecting to a Database > gsql Usage Guide" in *Tool Reference*.

Format	Value of -F	Description	Suggestion	Import Tool
Custom	c	A binary file that allows the restoration of all or selected database objects from an exported file.	For medium- or large-sized databases, the backup result needs to be exported to a single file. In this case, the custom format is recommended.	You can use gs_restore to import database objects from a custom-, directory-, or tar-format archive. For details about how to use the gs_restore tool, see "Data Import and Export Tools > gs_restore for Importing Data" in *Tool Reference*.
Directory	d	A directory containing directory files of database objects and the data files of tables and BLOBs.	For medium- or large-sized databases, database objects and data files need to be stored and exported in different directories. In this case, the directory format is recommended.	
.tar archive	t	A tar-format archive that allows the restoration of all or selected database objects from an exported file. The .tar file cannot be further compressed and has an 8-GB limitation on the size of each single file.	For a small-sized database, you need to export the archive result and pack it. In this case, the .tar format is recommended.	

NOTICE

- gs_dump does not back up global objects (roles, tablespaces, and corresponding permissions) shared with all databases. Therefore, ensure that global objects have been created on the target database or new instance before the restoration. The **-g** command of gs_dumpall can be used to export global objects and use gsql to import global objects at the target end. For details about how to use the gs_dumpall tool, see "Data Import and Export Tools > gs_dumpall for Exporting All Database Information" in *Tool Reference*.
- gs_dump and gs_restore do not support import and export across database compatibility modes. Ensure that the database compatibility mode and compatibility configuration parameters of the source and target databases are the same. For details about how to query and create a database of a specified compatibility mode, see "SQL Reference > SQL Syntax > C > CREATE DATABASE" in *Developer Guide*.
- Do not modify the files and contents exported using the **-F c**, **-F d**, or **-F t** format. Otherwise, the restoration may fail. If you need to modify or replace the file exported using the **-F p** format, edit them with caution.
- After restoration, you are advised to run **ANALYZE** on the database to provide useful statistics for the optimizer.

You are advised to run the following command as a user with the SYSADMIN permission to back up data. The source database is **my_database**, and the exported data contains data and object definitions.

```
-- Plain-text
nohup gs_dump my_database -U root -W ******** -p 8000 -F p -f /data/backup/my_database_backup.sql > /
data/backup/my_database_backup.log &
-- Custom
nohup gs_dump my_database -U root -W ******** -p 8000 -F c -f /data/backup/my_database_backup.dmp > /
data/backup/my_database_backup.log &
-- Directory
nohup gs_dump my_database -U root -W ******** -p 8000 -F d -f /data/backup/my_database_backup > /data/
backup/my_database_backup.log &
-- .tar archive
nohup gs_dump my_database -U root -W ******** -p 8000 -F t -f /data/backup/my_database_backup.tar > /
data/backup/my_database_backup.log &
```

Before restoration, you need to create a target database that has the same attributes as the source database and does not contain any data.

```
-- Run the following gsql meta-command to view the database attribute information:
\l+
-- Create a target database based on the queried attribute information.
create database my_database2 encoding='xxxxx' LC_COLLATE='xxxxx' LC_CTYPE ='xxxxx' TEMPLATE=xxx
DBCOMPATIBILITY 'xxx';
```

Run the following command as a user with the SYSADMIN permission to restore the database:

```
-- Plain-text
nohup gsql -d my_database2 -p 8000 -U root -W ******** -f /data/backup/my_database_backup.sql -a > /
data/backup/my_database_restore.log &
-- Custom
nohup gs_restore /data/backup/my_database_backup.dmp -d my_database2 -p 8000 -U root -W ******** -F c
-v > /data/backup/my_database_restore.log &
-- Directory
nohup gs_restore /data/backup/my_database_backup -d my_database2 -p 8000 -U root -W ******** -F d -v
> /data/backup/my_database_restore.log &
-- .tar archive
nohup gs_restore /data/backup/my_database_backup.tar -d my_database2 -p 8000 -U root -W ******** -F t -
v > /data/backup/my_database_restore.log &
```

12.2.2 Schema-Level Import and Export

The gs_dump tool can be used to back up a single schema and you are advised to use the gs_dump tool and the **-n** parameter to do so. Multiple **-n** parameters can be used to back up multiple schemas.

For details about how to use the gs_dump tool, see "Data Import and Export Tools > gs_dump for Exporting Database Information" in *Tool Reference*.

NOTICE

If the exported schema depends on objects that are not exported, an error message may be displayed indicating that the dependent objects are missing when the schema is imported. Therefore, ensure that the dependent objects have been created before importing the schema.

You are advised to run the following command as a user with the SYSADMIN permission to back up data. The source database is **my_database**, and the exported data contains data and object definitions.

```
-- Plain-text
nohup gs_dump my_database -U root -W ******** -p 8000 -F p -f /data/backup/my_schema_backup.sql -n
my_schema > /data/backup/my_schema_backup.log &
-- Custom
nohup gs_dump my_database -U root -W ******** -p 8000 -F c -f /data/backup/my_schema_backup.dmp -n
my_schema > /data/backup/my_schema_backup.log &
-- Directory
nohup gs_dump my_database -U root -W ******** -p 8000 -F d -f /data/backup/my_schema_backup -n
my_schema > /data/backup/my_schema_backup.log &
-- .tar archive
nohup gs_dump my_database -U root -W ******** -p 8000 -F t -f /data/backup/my_schema_backup.tar -n
my_schema > /data/backup/my_schema_backup.log &
```

Before restoration, you need to create a target database that has the same attributes as the source database and does not contain the target schema.

```
-- Run the following gsql meta-command to view the database attribute information:
\l+
-- Create a target database based on the queried attribute information.
create database my_database2 encoding='xxxxx' LC_COLLATE='xxxxx' LC_CTYPE ='xxxxx' TEMPLATE=xxx
DBCOMPATIBILITY 'xxx';
```

Run the following command as a user with the SYSADMIN permission to restore the database:

```
-- Plain-text
nohup gsql -d my_database2 -p 8000 -U root -W ******** -f /data/backup/my_schema_backup.sql -a > /data/
backup/my_schema_restore.log &
-- Custom. For details about how to use the gs_restore tool, see "Data Import and Export Tools > gs_restore
for Importing Data" in Tool Reference.
nohup gs_restore /data/backup/my_database_backup.dmp -d my_database2 -p 8000 -U root -W ******** -F c
-v -n my_schema > /data/backup/my_schema_restore.log &
-- Directory
nohup gs_restore /data/backup/my_database_backup -d my_database2 -p 8000 -U root -W ******** -F d -v -
n my_schema > /data/backup/my_schema_restore.log &
-- .tar archive
nohup gs_restore /data/backup/my_database_backup.tar -d my_database2 -p 8000 -U root -W ******** -F t -
v -n my_schema > /data/backup/my_schema_restore.log &
```

12.2.3 Table-Level Import and Export

Many tools are available for table-level import and export. You can select a proper tool based on the following scenarios:

1. If you need to export the definition and data of a single table to the same file, you are advised to use the plain-text archive of the gs_dump tool and the **-t** parameter. You can use multiple **-t** parameters to back up multiple tables. For details about how to use the gs_dump tool, see "Data Import and Export Tools > gs_dump for Exporting Database Information" in *Tool Reference*.

   NOTICE

   If the exported table depends on objects that are not exported, an error message may be displayed indicating that the dependent objects are missing when the table is imported. Therefore, ensure that the dependent objects have been created before importing the table.

   You are advised to run the following command as a user with the SYSADMIN permission to back up data. The source database is **my_database**, and the target table is **my_table** in **my_schema**.
   ```
   nohup gs_dump my_database -U root -W ******** -p 8000 -F p -f /data/backup/my_table_backup.sql -t my_schema.my_table > /data/backup/my_table_backup.log &
   ```
   Before restoration, create a target database with the same attributes as the source database, and ensure that the target schema exists in the database and no target table exists. Then, run the following command as a user with SYSADMIN permissions to restore the database:
   ```
   nohup gsql -d my_database2 -p 8000 -U root -W ******** -f /data/backup/my_table_backup.sql -a > /data/backup/my_table_restore.log &
   ```
2. Only the definition of a single table needs to be exported and no data in the table is required.

   NOTICE

   If the exported table depends on objects that are not exported, an error message may be displayed indicating that the dependent objects are missing when the table is imported. Therefore, ensure that the dependent objects have been created before importing the table.

   You are advised to use the plain-text archive of the gs_dump tool together with the **-s** parameter. The command is as follows:
   ```
   nohup gs_dump my_database -U root -W ******** -p 8000 -F p -f /data/backup/my_table_backup.sql -t my_schema.my_table -s > /data/backup/my_table_backup.log &
   ```
   Before restoration, create a target database with the same attributes as the source database, and ensure that the target schema exists in the database and no target table exists. Then, run the following command as a user with SYSADMIN permissions to restore the database:
   ```
   nohup gsql -d my_database2 -p 8000 -U root -W ******** -f /data/backup/my_table_backup.sql -a > /data/backup/my_table_restore.log &
   ```
3. If you are used to using Oracle SQLLDR or need to save the data import logs (such as import result, discarded data, and error data) to the client, you can use gs_loader, which supports importing data files in csv, text, and fixed formats. CSV is recommended for data import. For details about how to use the gs_loader tool, see "Data Import and Export Tools > gs_loader for Importing Data" in *Tool Reference*.

NOTICE

- The data files in the three formats must use "\n" or "\r\n" as the line terminator, and the line terminators of the entire file must be the same (either "\n" or "\r\n").
- In text format, gs_loader cannot identify escaped special characters (for example, '\n', which is still '\n' after import and will not be escaped to 0x0A), compared with COPY. By default, the TEXT file exported using COPY is escaped. If gs_loader is used to import the file, the escaped characters cannot be converted back, causing inconsistency. Therefore, gs_loader can import only the TEXT files that are exported by COPY and are not escaped (using WITHOUT ESCAPING).

The typical application scenarios are as follows:

– The format of the upstream data file can be customized or determined as CSV.

  The CSV format is compatible with multiple platforms and is universal in the industry. Many data sources support the export of data files in CSV format. Unless the upstream data file format is not CSV, you need to select the import method based on the actual file format (TEXT or FIXED). In addition, CSV files are recommended. If the column data contains special characters (such as commas and newline characters), the data file can only be saved as CSV because the CSV format uses quote characters to enclose the data containing special characters to prevent the special characters in the field data from conflicting with separators and line terminators.

  When importing a standard CSV file, add the *FIELDS CSV* statement to the control file. The following is an example:

```
-- Create a table.
gaussdb=# create table test_loader(id int, name name);

-- View the control file test.ctl.
LOAD DATA
TRUNCATE INTO TABLE test_loader
FIELDS CSV
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
(
  id integer external,
  name char
)

-- View the data file test.csv.
1,"aa
a"
2,"bb b"
3,"cc,c"
4,ddd

-- Import data.
gs_loader -p xxx host=xxx control=test.ctl data=test.csv -d testdb -W xxx

-- The import is successful. View the import result.
gaussdb=# select * from t1;
 id |   name
----+-----------
  1 | aa       +
    | a
  2 | bb      b
  3 | cc,c
  4 | ddd
(4 rows)
```

NOTE

- If the column quote characters are not standard double quotation marks, you can set it using the *OPTIONALLY ENCLOSED BY* clause.
- If the column separator is not a standard comma (,), you can set it using the *FIELDS TERMINATED BY* clause.

– The upstream data file is in TEXT format.

In text format, gs_loader cannot identify escaped special characters (for example, '\n', which is still '\n' after import and will not be escaped to 0x0A), compared with COPY. Therefore, data containing special characters such as delimiters and line terminators is not supported in text format. Otherwise, the structure will be disordered.

The following is an example of importing a valid data file in TEXT format:

```
-- Create a table.
gaussdb=# create table test_loader(id int, name name);

-- View the control file test.ctl.
LOAD DATA
TRUNCATE INTO TABLE test_loader
(
  id integer external,
  name char
)

-- View the data file test.dat.
1 aaa
2 bbb
3 ccc

-- Import data.
gs_loader -p xxx host=xxx control=test.ctl data=test.dat -d testdb -W xxx

-- The import is successful. View the import result.
gaussdb=# select * from t1;
 id | name
----+------
  1 | aaa
  2 | bbb
  3 | ccc
(3 rows)
```

NOTE

If the column separator is not a standard horizontal tab character, you can set it using the *FIELDS TERMINATED BY* clause.

– The upstream data file is in FIXED format.

In FIXED format, each column has fixed length and is not separated by delimiters. Therefore, column data does not conflict with delimiters. However, if the column data contains newline characters, the data cannot be processed.

An import example is as follows:

```
-- Create a table.
gaussdb=# create table test_loader(id int, name name);

-- View the control file test.ctl.
LOAD DATA
TRUNCATE INTO TABLE test_loader
```

```
(
  id POSITION(1:1) integer external,
  name POSITION(2:5) char
)

-- View the data file test.fixed.
1aaaa
2bb b
3cc,c

-- Import data.
gs_loader -p xxx host=xxx control=test.ctl data=test.fixed -d testdb -W xxx

-- The import is successful. View the import result.
gaussdb=# select * from t1;
 id |   name
----+-----------
  1 | aaaa
  2 | bb      b
  3 | cc,c
(3 rows)
```

13 Best Practices for JDBC

13.1 Best Practices for JDBC (Distributed Instances)

13.1.1 Batch Insertion

13.1.1.1 Scenario Overview

This section explains how to use the JDBC driver for batch data insertion.

13.1.1.1.1 Usage Scenarios

Scenario Description

When you have a large data size to insert into a database, using batch insertion is much more efficient than executing SQL statements multiple times, one at a time.

This section illustrates various operations using the JDBC driver, including establishing database connections, utilizing transactions, executing batch insertion, and obtaining column information in the result set.

Trigger Conditions

JDBC adds SQL statements to the batch execution list through its addBatch API. The batch operation is then executed through the executeBatch API.

Impact on Services

- Lower network interaction costs

  Combining multiple INSERT statements into one batch operation significantly reduces the number of round trips between the client and the database. This enhances the overall throughput and minimizes the impact of network congestion on performance.
- Higher data processing efficiency

  In single-record insertion, the database must parse the syntax and generate an execution plan for each SQL statement. In contrast, batch insertion requires parsing the syntax and generating the plan only once, eliminating repetitive tasks and saving CPU cycles and memory allocation time.

- Reduced system resource usage and overhead

  In single-record insertion, transaction commits or Xlog writes occur at least once. In contrast, batch insertion allows multiple records to be inserted within a single transaction, significantly reducing the frequency of transaction commits, Xlog pressure, and transaction management overhead. In addition, it decreases the total number of network packet processing, transaction management, log write, and row format conversion tasks, which in turn lowers the CPU loads and temporary memory usage of the database server. This results in more resources being available for core query and computing operations.

- Higher memory usage

  When large data sizes are involved, constructing SQL statements for batch insertion can significantly increase memory usage. This is particularly noticeable when you construct SQL statements through string concatenation, as it can lead to a sharp rise in memory consumption. Large-size batch processing may exceed the maximum SQL length limit of the database or driver, or trigger other parameter restrictions, potentially leading to errors or performance issues.

Here is a detailed comparison between batch insertion and single-record insertion.

Mode	Advantages	Disadvantages
Single-record insertion	● Its code is simple, straightforward, and easy to implement. ● If any single record fails, it can be accurately identified and handled without impacting other records. ● This mode is less demanding in terms of database and driver compatibility.	● Extensive network interactions are needed. Each INSERT operation requires connecting, parsing, and committing, leading to suboptimal performance. ● Inserting a large number of records is likely to cause a bottleneck. ● Not using transactions may result in failure to guarantee the consistency of INSERT operations.
Batch insertion	● This mode greatly reduces the number of network round trips and SQL parsing instances, leading to a notable improvement in insertion throughput. ● Multiple rows can be committed within a single transaction to guarantee atomicity.	● Its code is complex, requiring manual concatenation of placeholders and parameters. ● If a single statement encounters an error, all data will be rolled back, complicating the error recovery process. ● The number of placeholders is limited; therefore, it is essential to carefully manage the batch size.

Applicable Versions

This applies only to GaussDB 503.1.0 and later versions.

13.1.1.1.2 Requirements and Objectives

Service Pain Points

When dealing with large data sizes, single-record insertion generates numerous network requests and consumes substantial system resources. Moreover, the database server has to repeatedly parse similar statements, leading to a decline in service performance. Batch insertion is introduced as a solution to these issues.

Service Objectives

Utilize JDBC to implement batch insertion with transactions, obtain column data in the result set, and output the result information.

13.1.1.2 Architecture Principles

Core Principles

Batch processing of the JDBC driver allows adding multiple SQL statements to the batch execution list and collectively sending them to the database in one go. All insert or update operations in the transaction are carried in a single U packet. Consequently, completing the batch operation only necessitates once instance of network connection establishment and data exchange.

Solution Advantages

Sending all batch updates at once in a single U packet significantly decreases network communication overhead and enhances execution efficiency when compared to sending PBE packets multiple times.

13.1.1.3 Preparations

- JDK version: 1.7 or later.
- Database environment: GaussDB 503.1.0 or later.
- JDBC driver environment:

  Refer to "Application Development Guide > Development Based on JDBC > Development Procedure > Obtaining the JAR Package of the Driver and Configuring the JDK Environment" in *Developer Guide*.
- Data: Create a test table and insert test data, as follows:

```
gaussdb=# CREATE TABLE TEST_BATCH(
V1 TEXT,
V2 TEXT)
CREATE TABLE
```

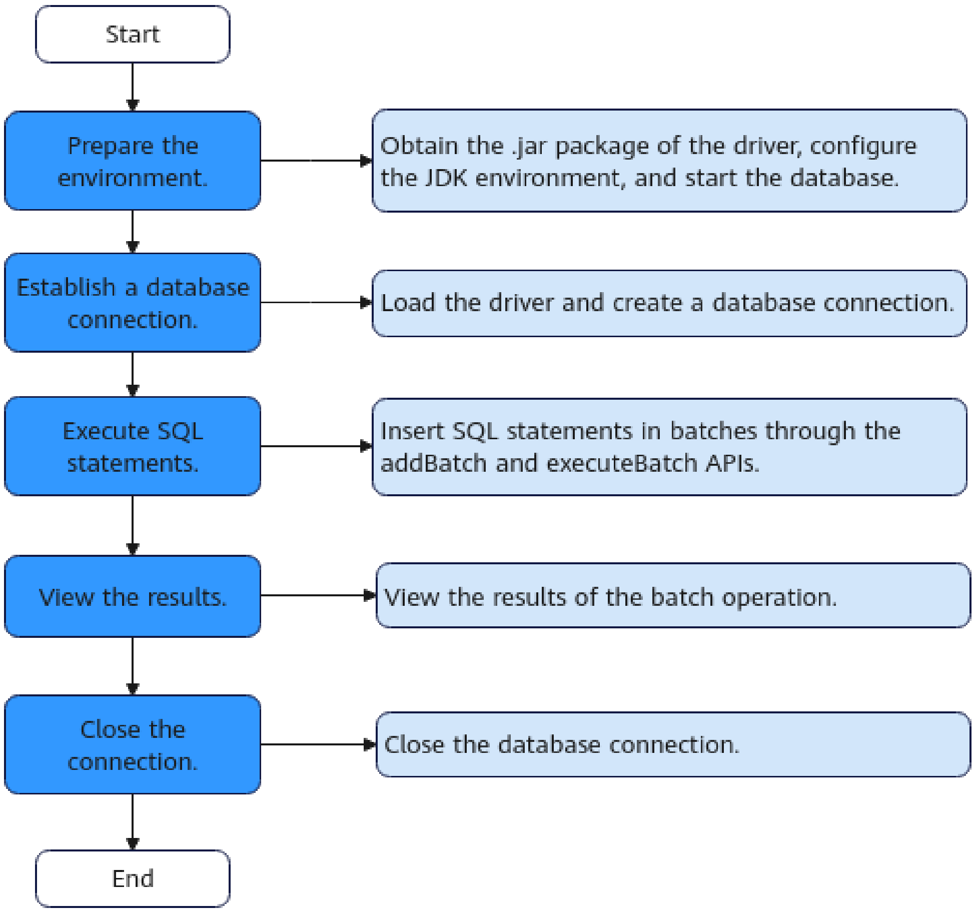
13.1.1.4 Procedure

13.1.1.4.1 Process Overview

The process of batch data insertion using JDBC includes preparing the environment, establishing a database connection, executing batch insertion through APIs, querying the execution results, and closing the connection.

Figure 13-1 shows the overall process.

Figure 13-1 Process of batch data insertion using JDBC


Start
Prepare the environment.
Obtain the .jar package of the driver, configure the JDK environment, and start the database.
Establish a database connection.
Load the driver and create a database connection.
Execute SQL statements.
Insert SQL statements in batches through the addBatch and executeBatch APIs.
View the results.
View the results of the batch operation.
Close the connection.
Close the database connection.
End

13.1.1.4.2 Detailed Procedure

Step 1 Establish a database connection.

Here are suggestions for commonly used parameters in the connection string. For more detailed settings, refer to "Application Development Guide > Development Based on JDBC > Development Procedure > Connecting to a Database > Connection Parameter Reference" in *Developer Guide*.

- **connectTimeout**: timeout interval (in seconds) for connecting to the server's OS. If the time taken for JDBC to establish a TCP connection with the database exceeds this interval, the connection will be closed. It is advisable to set this parameter based on network conditions. The default value is **0**, whereas the recommended value is **2**.
- **socketTimeout**: timeout interval (in seconds) for socket reads. If the time taken to read data streams from the server exceeds this interval, the

connection will be closed. Not setting this parameter may lead to prolonged waiting times for the client in the event of abnormal database processes. It is advisable to set this parameter based only the acceptable SQL execution time for services. The default value is **0**, with no specific recommended value provided.

- **connectionExtraInfo**: specifies whether the driver reports its deployment path, process owner, and URL connection configurations to the database. The default value is **false**, whereas the recommended value is **true**.
- **logger**: specifies a third-party log framework as needed by your application. It is advisable to choose one that incorporates slf4j APIs. These APIs can record JDBC logs to facilitate exception locating. The recommended value is **Slf4JLogger** when a third-party log framework is needed.
- **autoBalance**: specifies whether to enable load balancing for establishing new connections. The default value is **false**, but it is advisable to set it to **true** to utilize the polling load balancing policy.
- **batchMode**: specifies whether the connection operates in batch mode. When **batchMode** is set to **on**, batch insertion and modification are allowed, with the data type of each column determined by the type specified in the first data record.

```
String url = "jdbc:gaussdb://$ip:$port/database?
connectTimeout=xx&socketTimeout=xx&connectionExtraInfo=true&logger=Slf4JLogger&autoBalance=t
rue&batchMode=on"
Connection conn = DriverManager.getConnection("url",userName,password);
```

Step 2 Prepare SQL statements for batch execution.

Use preparedStatement to prepare statements. For example, to insert data into a test table, prepare the statements provided below. You can substitute these statements with the necessary ones for your services.

```
String sql = "INSERT INTO  TEST_BATCH(v1,v2) VALUES(?,?)";
PreparedStatement preparedStatement = conn.prepareStatement(sql);
```

Step 3 Bind parameters in batches.

Use preparedStatement to bind parameters. Then call the addBatch API to add the SQL statements to the batch execution list.

```
for(int i=0;i<5;i++){
    preparedStatement.setString(1,"value1_"+i);
    preparedStatement.setString(2,"value2_"+i);
    preparedStatement.addBatch();
}
```

Step 4 Execute the batch operation.

By calling the executeBatch API, preparedStatement executes the batch operation. Upon completion, it returns an array of int results, from which you can determine the number of data records affected by the batch operation.

```
Int[] results = preparedStatement.executeBatch();
System.out.println(Arrays.toString(results));
```

Step 5 Release resources and close the database connection.

Use try-with-resources to automatically close any open file resources.

```
try (Connection conn = getConnection(); PreparedStatement preparedStatement =
conn.prepareStatement(sql))
```

Step 6 Handle exceptions if any.

If your program encounters any SQL exception during runtime, utilize the try-catch module to handle them and add the necessary exception handling logic for the actual services.

```
try {
// Service code
} catch (SQLException e) {
// Exception handling logic
}
```

----End

13.1.1.4.3 Complete Example

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Arrays;
import java.sql.DriverManager;
public class TestBatch {
    public static Connection getConnection() throws ClassNotFoundException, SQLException{
 String driver = "com.huawei.gaussdb.jdbc.Driver";
        // Specify the source URL of the database. (Adjust $ip, $port, and database based on the actual 
services.)
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";
        // Obtain the username and password from the environment variables.
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        Class.forName(driver);
        return DriverManager.getConnection(sourceURL, userName, password);
    }
    public static void main(String[] args) {
        String sql = "insert into test_batch(v1,v2) values(?,?)";
        try (Connection conn = getConnection(); PreparedStatement preparedStatement = 
conn.prepareStatement(sql)) {
            conn.setAutoCommit(false);
            for (int i = 0; i < 5; i++) {
                preparedStatement.setInt(1, 1);
                preparedStatement.setString(2, "value2_" + i);
                preparedStatement.addBatch();
            }
            int[] results = preparedStatement.executeBatch();
            conn.commit();
            System.out.println(Arrays.toString(results));
        } catch (ClassNotFoundException | SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Result Verification

The execution results in **Complete Example** show the number of data records affected by the batch operation.

When you use preparedStatement for batch insertion, it returns an array of INT results, with **results[0]** indicating the total number of data records affected by the batch operation.

```
[5, 0, 0, 0, 0]
```

Rollback Method

To roll back operations within a specific transaction, call the Rollback API of the transaction object.

13.1.1.5 Typical Issues

1. Symptom: When preparedStatement binds parameters multiple times, the following error occurs during batch insertion:
   ```
   java.lang.RuntimeException: java.sql.BatchUpdateException: Batch entry 0 - 5 insert into
   test_batch1(v1,v2) values(('1'),('value2_0')) was aborted: [*.*.*.*:*/*.*.*.*:*] ERROR: invalid input syntax
   for integer: "ss"  Call getNextException to see other errors in the batch.
   ```
   Cause: The data type of a bound parameter differs from the initially bound parameter.

   Solution: To ensure successful batch insertion using preparedStatement, maintain consistency in the data types of bound parameters.
2. Symptom: The array of results returned by preparedStatement after batch insertion does not match the results returned by Oracle Database.

   Cause: When **batchMode** is set to **on**, JDBC utilizes GaussDB's distinct packet processing logic, which differs from Oracle Database's logic but offers faster speed. Setting **batchMode** to **off** in the connection string will ensure consistency with Oracle Database's results.

   When **batchMode** is set to **on**, **Complete Example** will produce results that differ from those generated by Oracle Database's corresponding API.
   ```
   [5, 0, 0, 0, 0]
   ```
   Oracle Database's results:
   ```
   [1, 1, 1, 1, 1]
   ```
   When **batchMode** is set to **off**, the results become consistent with Oracle Database's corresponding API.
   ```
   [1, 1, 1, 1, 1]
   ```

13.1.2 Streaming Query

13.1.2.1 Scenario Overview

This section explains how to execute a streaming query with GaussDB JDBC.

13.1.2.1.1 Usage Scenarios

Scenario Description

GaussDB's streaming query mechanism processes results one by one rather than loading them all at once. It is designed for big data query scenarios with limited memory resources, such as big data export, offline analysis tasks, and pagination/on-demand loading. This mechanism helps prevent excessive memory consumption and potential memory overflow, ultimately enhancing processing speed.

Trigger Conditions

The JDBC connection parameter **enableStreamingQuery** is set to **true**. In addition, **fetchSize** is set to **Integer.MIN_VALUE** before the executeQuery method of Statement and PreparedStatement is called.

Impact on Services

Streaming query offers the following advantages:

- Low memory consumption: Streaming query significantly reduces the memory usage of applications.
- Fast response: Clients can start processing immediately upon receiving the first batch of data, without having to wait for the entire data set to be ready.

However, there are also associated risks:

- Prolonged connection occupation: Streaming query requires the database connection to remain open throughout the transmission of data streams. This prolonged connection occupation may result in connection pool depletion or other resource scheduling issues.
- Extended transaction duration: Transactions involving streaming queries may hold locks or MVCC snapshots for an extended period, increasing the probability of deadlocks or impacting write performance.

Applicable Versions

This applies only to GaussDB 505.1.0 and later versions.

13.1.2.1.2 Requirements and Objectives

Service Pain Points

In a standard query, JDBC will receive a significant amount of data when querying massive data. Reading all this data in full can potentially cause memory overflow.

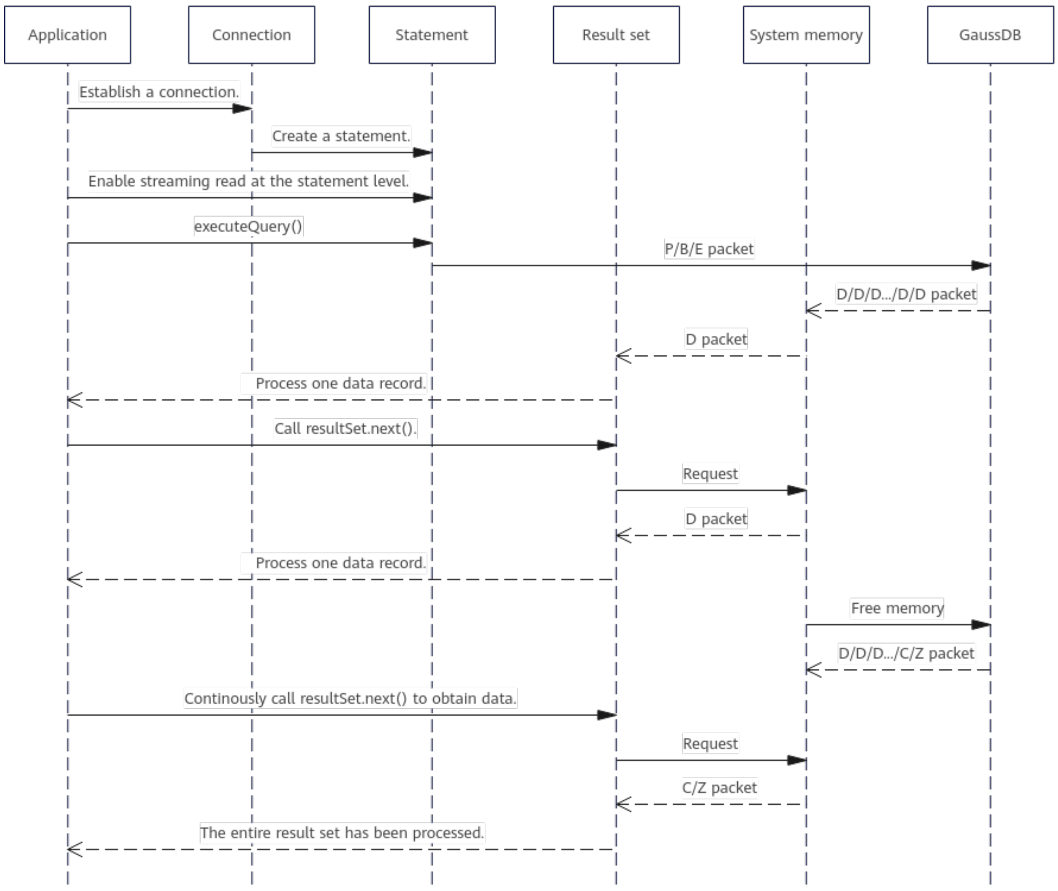
Service Objectives

Execute streaming queries with JDBC to prevent memory overflow.

13.1.2.2 Architecture Principles

Core Principles

Figure 13-2 Principles of streaming query



- When a streaming query is executed, the server keeps sending data to the client's socket buffer until the buffer is full. Upon the arrival of the first data record in the buffer, the first D packet is returned. JDBC then begins loading and processing data from the buffer row by row.
- The result set is designed to store only one row of data. The next data record is read from the buffer and stored into the result set only when the "next" method is called. This process repeats until all records have been read.

Solution Advantages

Streaming query is recommended for big data query scenarios with limited memory resources. If streaming query does not meet your service requirements, consider the following alternative methods:

- Standard query: All data is read in one go.

  Advantages: This method allows traversal of the entire result set in both forward and backward directions. In addition, data can be reprocessed after being consumed.

Disadvantages: Processing is slow, and there is a risk of memory overflow with large result sets.

- Batch query: Multiple rows are read at a time. For details, see **Batch Query**.

  Advantages: Data is returned in the specified size, which helps prevent memory overflow.

  Disadvantages: Processing is slow, and multiple query requests need to be sent to the server.

13.1.2.3 Preparations

- Environment: Ensure that GaussDB is running properly, obtain the JDBC driver, and configure the environment. For details, see "Application Development Guide > Development Based on JDBC > Development Procedure > Obtaining the JAR Package of the Driver and Configuring the JDK Environment" in *Developer Guide*.
- Data: Create a test table and insert test data, as follows:

```
gaussdb=# CREATE TABLE tab_test(id int,context varchar(1000),PRIMARY KEY(id));
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "tab_test_pkey" for table "tab_test"
CREATE TABLE
gaussdb=# INSERT INTO tab_test SELECT generate_series(1,1000000),repeat('GaussDB Test', 50);
INSERT 0 1000000
```

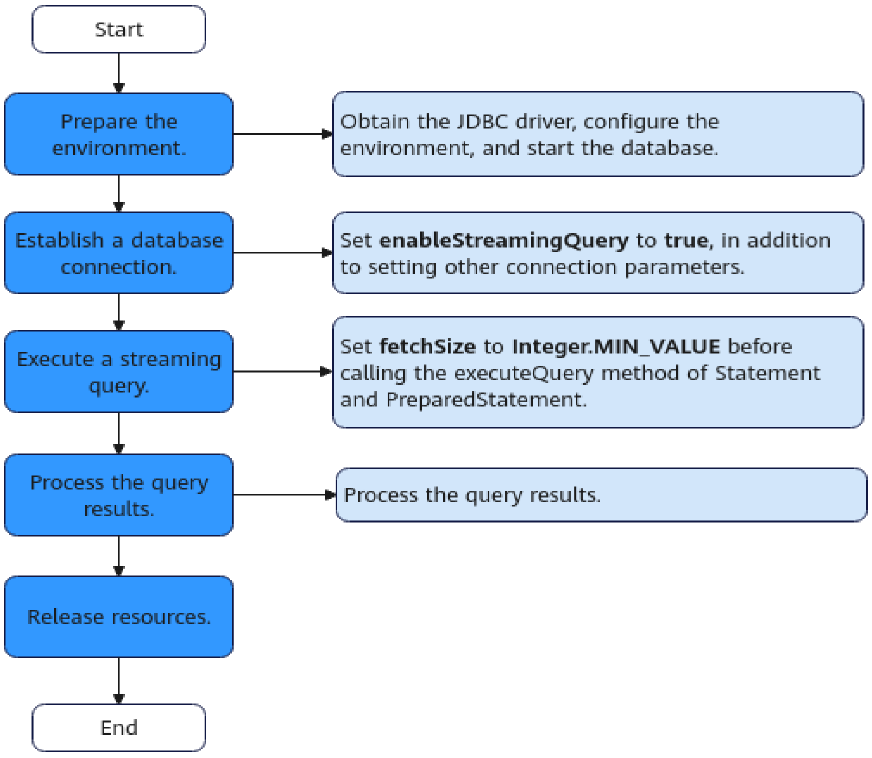
13.1.2.4 Procedure

13.1.2.4.1 Process Overview

The process of executing a streaming query with GaussDB JDBC includes preparing the environment, establishing a database connection, executing a streaming query, processing the query results, and releasing resources.

Figure 13-3 shows the overall process.

Figure 13-3 Process of executing a streaming query with GaussDB JDBC


Start
Prepare the environment.
Obtain the JDBC driver, configure the environment, and start the database.
Establish a database connection.
Set **enableStreamingQuery** to **true**, in addition to setting other connection parameters.
Execute a streaming query.
Set **fetchSize** to **Integer.MIN_VALUE** before calling the executeQuery method of Statement and PreparedStatement.
Process the query results.
Process the query results.
Release resources.
End

13.1.2.4.2 Detailed Procedure

Step 1 Database connection: Set **enableStreamingQuery** to **true**, in addition to setting other connection parameters. For details, see "Application Development Guide > Development Based on JDBC > Development Procedure > Connecting to a Database" in *Developer Guide*.

Step 2 Streaming query: Set **fetchSize** to **Integer.MIN_VALUE** before calling the executeQuery method of Statement and PreparedStatement.

Step 3 Result processing: Process the query results based on the actual services.

Step 4 Resource release: Upon completion, release the result set, statement, connection, and other resources. It is advisable to use the try-with-resources syntax or the close method in the try-finally block to release resources.

----End

13.1.2.4.3 Complete Example

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class StreamQueryTest {
    // Establish a database connection.
    public static Connection getConnection() throws ClassNotFoundException, SQLException {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        // Specify the source URL of the database. (Adjust $ip, $port, and database as needed.) Set the
connection parameter enableStreamingQuery to true.
        String sourceURL = "jdbc:gaussdb://$ip:$port/database?enableStreamingQuery=true";
        // Obtain the username and password from the environment variables.
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        Class.forName(driver);
        return DriverManager.getConnection(sourceURL, userName, password);
    }

    public static void main(String[] args) {
        String selectSql = "select * from tab_test order by id asc limit ?";
        // Use the try-with-resources syntax to release resources.
        try (Connection conn = getConnection(); PreparedStatement preparedStatement = 
conn.prepareStatement(selectSql,
            ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY)) {
            preparedStatement.setInt(1, 100000);
            // Set fetchSize to Integer.MIN_VALUE and call executeQuery to query data.
            preparedStatement.setFetchSize(Integer.MIN_VALUE);
            int totalCount = 0;
            try (ResultSet resultSet = preparedStatement.executeQuery()) {
                while (resultSet.next()) {
                    // Process the query results. The sample code below prints only a portion of the query results 
along with the total number of data records.
                    if (totalCount++ < 5) {
                        System.out.println("row:" + resultSet.getRow() + ",id :" + resultSet.getInt(1));
                    }
                }
                System.out.println("totalCount:" + totalCount);
            }
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Result Verification

1. Place **gaussdbjdbc.jar** and **StreamQueryTest.java** from **Complete Example** into the same directory.
2. Compile and execute the sample code. During execution, set the maximum heap memory of JVM to 16 MB.
```
javac -classpath ".:gaussdbjdbc.jar" StreamQueryTest.java
java -Xmx16M -classpath ".:gaussdbjdbc.jar" StreamQueryTest
```
3. Run the example code. It completes successfully without encountering memory overflow. During the streaming query, the result set stores only one data record at a time. Therefore, the return value of **resultSet.getRow()** is **1**. The execution result of the code is like this:
```
row:1,id :1
row:1,id :2
row:1,id :3
row:1,id :4
row:1,id :5
totalCount:100000
```

Rollback Method

- To disable streaming query for a single statement, delete **setFetchSize(Integer.MIN_VALUE)** from Statement and PreparedStatement.
- To disable streaming query for the current connection, set the connection parameter **enableStreamingQuery** to **false**.

13.1.2.5 Typical Issues

1. Symptom: The following exception occurs when **fetchSize** is set to **Integer.MIN_VALUE** in Statement and PreparedStatement:

```
org.postgresql.util.PSQLException: Fetch size must be a value greater to or equal to 0.
    at org.postgresql.jdbc.PgStatement.setFetchSize(PgStatement.java:1623)
```

Cause: The JDBC connection parameter **enableStreamingQuery** is not set to **true**.

2. Symptom: The following exception occurs when the execute, executeQuery, and executeUpdate methods of Statement and PreparedStatement are called to query, insert, and update data:

```
org.postgresql.util.PSQLException: Streaming query statement is still active. No statements may be
issued when any streaming result sets are open and in use on a given connection. Ensure that you
have called .close() on any active streaming statement sets before attempting more queries.
    at org.postgresql.jdbc.PgStatement.execute(PgStatement.java:492)
    at org.postgresql.jdbc.PgPreparedStatement.executeWithFlags(PgPreparedStatement.java:210)
    at org.postgresql.jdbc.PgPreparedStatement.executeQuery(PgPreparedStatement.java:147)
```

   Cause: The result set from the previous streaming query has not been completely read or has not been closed.

3. Sequential access only: Streaming query utilizes the FORWARD_ONLY result set, which means that data can only be accessed row by row in the forward direction. It does not support randomly locating or rolling back to previous records. If your services involve frequent traversal or require random data access, do not use streaming query.
4. Configuration and compatibility requirements: Streaming query requires special configurations (**Trigger Conditions**) in the JDBC driver or ORM framework. Without proper configurations, the desired outcome cannot be achieved.

13.1.3 User-defined Type

13.1.3.1 Scenario Overview

This section explains how JDBC can use user-defined data types.

13.1.3.1.1 Usage Scenarios

Scenario Description

In real-world production, users' service systems often involve complex data analysis and operations. Traditional data types may not fully or efficiently represent or process their data scenarios. To address this, JDBC provides user-defined data types, Struct and Array, enabling the creation of new data types from existing ones. These user-defined types offer users a more convenient way to add, delete, and modify data.

Trigger Conditions

JDBC operations involve user-defined data types.

Impact on Services

- Improved data consistency and integrity

  User-defined types allow users to declare data types and check constraints for columns during the type definition phase, ensuring that all columns referencing these types automatically inherit the same verification logic.

- Improved data reusability and encapsulation

  User-defined types encapsulate frequently used data structures for reuse across multiple tables or functions. This minimizes redundant definitions and enhances maintenance efficiency.
- Additional performance overhead

  Using user-defined types will take up additional storage space and necessitate object assembly and disassembly during access, potentially leading to increased CPU and I/O usage.

Applicable Versions

This applies only to GaussDB 503.0 and later versions.

13.1.3.1.2 Requirements and Objectives

Service Pain Points

When dealing with complex data structures in services, basic data types may not accurately express complex service concepts, leading to potential code redundancy and impacting development efficiency.

Service Objectives

Utilize user-defined types in JDBC's stored procedures.

13.1.3.2 Architecture Principles

Core Principles

The database system creates metadata records for each user-defined type, including the type name, structure, and constraints. When working with user-defined types, JDBC converts them based on their structures as well as the user-defined type objects in the programming language.

Solution Advantages

- Support for complex data modeling

  The ability to process complex data structures, such as nested objects and arrays, improves the expression capability of data models.
- Improved data quality and development efficiency

  When querying complex data structures, you can utilize user-defined types for filtering. Reusing user-defined types helps enhance development efficiency.

13.1.3.3 Preparations

- JDK version: 1.7 or later.
- Database environment: GaussDB 503.0 or later.
- JDBC driver environment:

  Refer to "Application Development Guide > Development Based on JDBC > Development Procedure > Obtaining the JAR Package of the Driver and Configuring the JDK Environment" in *Developer Guide*.

- Data: Create a user-defined type and stored procedure, as follows:

```
// Create a user-defined type, PUBLIC.COMPFOO, that includes two records.
gaussdb=# CREATE TYPE PUBLIC.COMPFOO AS(
    ID INTEGER,
    NAME TEXT
);
CREATE TYPE
// Create a user-defined table type, PUBLIC.COMPFOO_TABLE.
gaussdb=# CREATE TYPE PUBLIC.COMPFOO_TABLE IS TABLE OF PUBLIC.COMPFOO;
CREATE TYEP
// Create the stored procedure public.test_proc with two input parameters of a user-defined type and
two output parameters of a user-defined type.
gaussdb=# CREATE OR REPLACE PROCEDURE public.test_proc(
    IN INPUT_COMPFOO PUBLIC.COMPFOO,
    IN INPUT_COMPFOO_TABLE PUBLIC.COMPFOO_TABLE,
    OUT OUTPUT_COMPFOO PUBLIC.COMPFOO,
    OUT OUTPUT_COMPFOO_TABLE PUBLIC.COMPFOO_TABLE
)
AS
BEGIN
    OUTPUT_COMPFOO := INPUT_COMPFOO;
    OUTPUT_COMPFOO_TABLE := INPUT_COMPFOO_TABLE;
END;
/
CREATE PROCEDURE
```

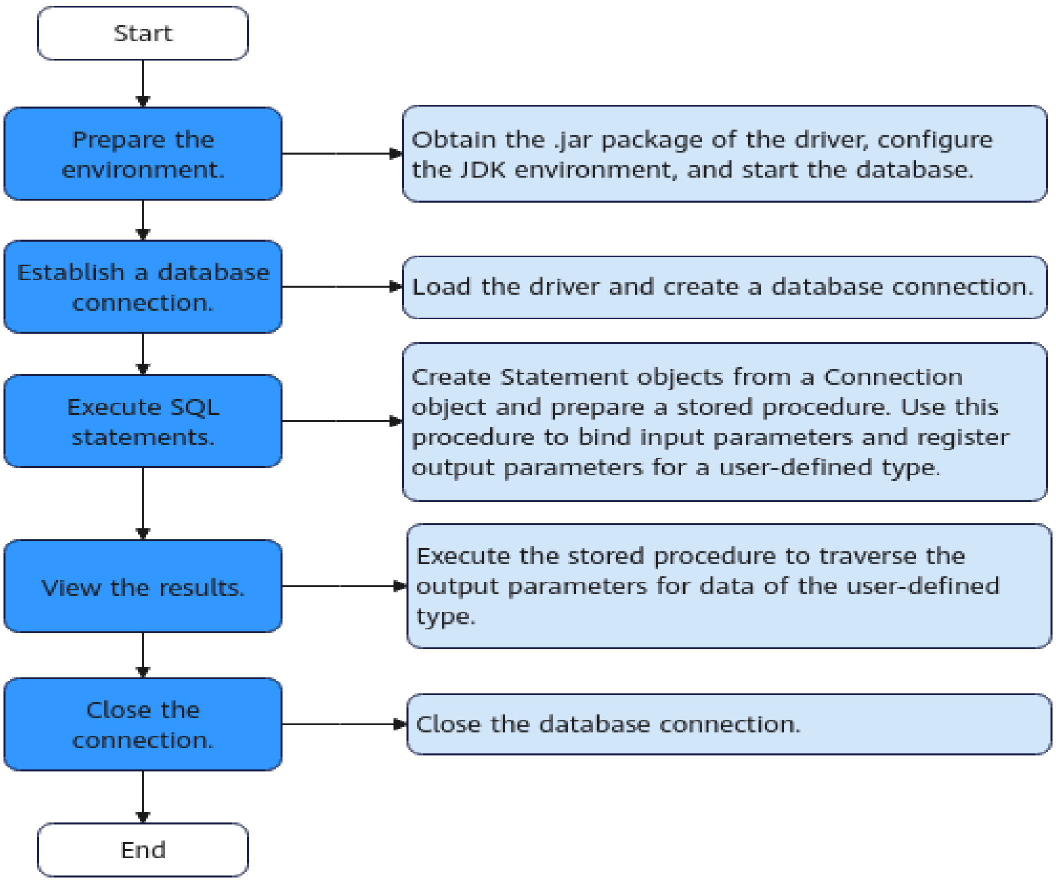
13.1.3.4 Procedure

13.1.3.4.1 Process Overview

Figure 13-4 shows the process for JDBC to insert and query data of user-defined types using a stored procedure.

This process includes preparing the environment, establishing a database connection, executing SQL statements of user-defined types, viewing the results, and closing the connection.

Figure 13-4 Process of using stored procedures in JDBC to handle user-defined types


Start
Prepare the environment.
Obtain the .jar package of the driver, configure the JDK environment, and start the database.
Establish a database connection.
Load the driver and create a database connection.
Execute SQL statements.
Create Statement objects from a Connection object and prepare a stored procedure. Use this procedure to bind input parameters and register output parameters for a user-defined type.
View the results.
Execute the stored procedure to traverse the output parameters for data of the user-defined type.
Close the connection.
Close the database connection.
End

13.1.3.4.2 Detailed Procedure

Step 1 Establish a database connection.

Here are suggestions for commonly used parameters in the connection string. For more detailed settings, refer to "Application Development Guide > Development Based on JDBC > Development Procedure > Connecting to a Database" in *Developer Guide*.

- **connectTimeout**: timeout interval (in seconds) for connecting to the server's OS. If the time taken for JDBC to establish a TCP connection with the database exceeds this interval, the connection will be closed. It is advisable to set this parameter based on network conditions. The default value is **0**, whereas the recommended value is **2**.
- **socketTimeout**: timeout interval (in seconds) for socket reads. If the time taken to read data streams from the server exceeds this interval, the connection will be closed. Not setting this parameter may lead to prolonged waiting times for the client in the event of abnormal database processes. It is advisable to set this parameter based only the acceptable SQL execution time for services. The default value is **0**, with no specific recommended value provided.
- **connectionExtraInfo**: specifies whether the driver reports its deployment path, process owner, and URL connection configurations to the database. The default value is **false**, whereas the recommended value is **true**.

- **logger**: specifies a third-party log framework as needed by your application. It is advisable to choose one that incorporates slf4j APIs. These APIs can record JDBC logs to facilitate exception locating. The recommended value is **Slf4JLogger** when a third-party log framework is needed.
- **autoBalance**: specifies whether to enable load balancing for establishing new connections. The default value is **false**, but it is advisable to set it to **true** to utilize the polling load balancing policy.

```
String url = "jdbc:gaussdb://$ip:$port/database?
connectTimeout=xx&socketTimeout=xx&connectionExtraInfo=true&logger=Slf4JLogger&autoBalance=t
rue"
Connection conn = DriverManager.getConnection("url",userName,password);
```

Step 2 Set GUC parameters.

Execute the SQL statement **SET behavior_compat_options='proc_outparam_override';** to enable **proc_outparam_override**, that is, overloading of stored procedures.

```
Statement statement = conn.createStatement();
statement.execute("SET behavior_compat_options='proc_outparam_override'");
statement.close();
```

Step 3 Prepare a stored procedure.

Use the Call syntax to declare the SQL statement for calling the stored procedure TEST_PROC. Then use prepareCall to prepare the statement.

```
CallableStatement cs = conn.prepareCall("{CALL PUBLIC.TEST_PROC(?,?,?,?)}");
```

Step 4 Bind input parameters.

Use PGobject to assemble data of the user-defined type. Then use prepareCall to bind input parameters.

```
PGobject pgObject = new PGobject();
pgObject.setType("public.compfoo"); // Set the name of a composite type.
pgObject.setValue("(1,demo)"); // Bind values of the composite type, formatted as "(value1,value2)".
cs.setObject(1, pgObject);
pgObject = new PGobject();
pgObject.setType("public.compfoo_table"); // Set the name of a table type.
pgObject.setValue("{\"(10,demo10)\",\"(11,demo111)\"}"); // Bind values of the table type, formatted as
"{\"(value1,value2)\",\"(value1,value2)\",...}".
cs.setObject(2, pgObject);
```

Step 5 Register the output type.

Use prepareCall to register the output type of the stored procedure. For the composite type, register **Types.STRUCT**. For the table type, register **Types.ARRAY**.

```
// Register output parameters for the composite type.
cs.registerOutParameter(3, Types.STRUCT, "public.compfoo");
// Register output parameters for the table type, in the "schema.typename" format.
cs.registerOutParameter(4, Types.ARRAY, "public.compfoo_table");
```

Step 6 Execute SQL statements and view the results.

Call the stored procedure and view the results corresponding to the output parameters.

```
cs.execute();
// Obtain output parameters.
// The return structure is of the user-defined type.
PGobject result = (PGobject) cs.getObject(3);  // Obtain output parameters.
result.getValue(); // Obtain character string values of the composite type.
String[] arrayValue = result.getArrayValue(); // Obtain array values of the composite type and sort them
based on columns of the composite type.
result.getStruct(); // Obtain the names of subtypes in the composite type and sort them based on the order
```

```
in which they were created.
result.getAttributes(); // Return objects of the user-defined type in each column. For the array and table
types, PgArray is returned. For the user-defined type, PGobject is encapsulated. Other types of data are
stored as character strings.
for (String s : arrayValue) {
    System.out.println(s);
}
PgArray pgArray = (PgArray) cs.getObject(4);
ResultSet rs = pgArray.getResultSet();
Object[] array = (Object[]) pgArray.getArray();
for (Object element : array) {
    System.out.println(element);
}
```

Step 7 Release resources and close the database connection.

```
cs.close();
conn.close();
```

Step 8 (Optional) Handle exceptions if any.

If your program encounters any SQL exception during runtime, utilize the try-catch module to handle them and add the necessary exception handling logic for the actual services.

```
try {
   // Service code
} catch (SQLException e) {
   // Exception handling logic
}
```

----End

13.1.3.4.3 Complete Example

```
import com.huawei.gaussdb.jdbc.jdbc.PgArray;
import com.huawei.gaussdb.jdbc.util.PGobject;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;
import java.sql.DriverManager;
public class TypesTest {
 public static Connection getConnection() throws ClassNotFoundException, SQLException {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        // Specify the source URL of the database. (Adjust $ip, $port, and database based on the actual
services.)
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";
        // Obtain the username and password from the environment variables.
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        Class.forName(driver);
        return DriverManager.getConnection(sourceURL, userName, password);
    }
 public static void main(String[] args) {
  try {
   Connection conn = getConnection();
   Statement statement = conn.createStatement();
   statement.execute("set behavior_compat_options='proc_outparam_override'");
   statement.close();
   CallableStatement cs = conn.prepareCall("{CALL PUBLIC.TEST_PROC(?,?,?,?)}");
    // Set parameters.
   PGobject pgObject = new PGobject();
   pgObject.setType("public.compfoo"); // Set the name of a composite type.
   pgObject.setValue("(1,demo)"); // Bind values of the composite type.
   cs.setObject(1, pgObject);
   pgObject = new PGobject();
   pgObject.setType("public.compfoo_table"); // Set the name of a table type.
   pgObject.setValue("{\"(10,demo10)\",\"(11,demo111)\"}"); // Bind values of the table type, formatted as
"{\"(value1,value2)\",\"(value1,value2)\",...}".
   cs.setObject(2, pgObject);
   // Register output parameters.
   // Register output parameters for the composite type.
   cs.registerOutParameter(3, Types.STRUCT, "public.compfoo");
   // Register output parameters for the table type.
   cs.registerOutParameter(4, Types.ARRAY, "public.compfoo_table");
   cs.execute();
   // Obtain output parameters.
  // The return structure is of the user-defined type.
   PGobject result = (PGobject) cs.getObject(3); // Obtain output parameters.
   result.getValue(); // Obtain character string values of the composite type.
   String[] arrayValue = result.getArrayValue(); // Obtain array values of the composite type and sort them
based on columns of the composite type.
   result.getStruct(); // Obtain the names of subtypes in the composite type and sort them based on the
order in which they were created.
   result.getAttributes(); // Return objects of the user-defined type in each column. For the array and table
types, PgArray is returned. For the user-defined type, PGobject is encapsulated. Other types of data are
stored as character strings.
   for (String s : arrayValue) {
    System.out.println(s);
   }
   PgArray pgArray = (PgArray) cs.getObject(4);
   ResultSet rs = pgArray.getResultSet();
   Object[] array = (Object[]) pgArray.getArray();
   for (Object element : array) {
    System.out.println(element);
   }
   cs.close();
   conn.close();
  } catch (ClassNotFoundException | SQLException e) {
   throw new RuntimeException(e);
  }
 }
}
```

Result Verification

Data of user-defined types is correctly obtained in **Complete Example**. Below are the execution results for **Complete Example**:

```
1
demo
(10,demo10)
(11,demo111)
```

Rollback Method

N/A

13.1.3.5 Typical Issues

1. Symptom: Users encounter the following exception when calling a stored procedure, even though they have bound input parameters of a user-defined type and registered output parameters for the procedure:

   ```
   ERROR: Function public.test_proc(compfoo, public.compfoo_table, compfoo, public.compfoo_table)
   does not exist.
     ???No function matches the given name and argument types. You might need to add explicit type
   casts.
   ```

   Cause: The user-defined type bound to the input and output parameters does not match the one set in the stored procedure.

Solution: Keep the user-defined type bound to the input and output parameters consistent with the one set in the stored procedure.

2. Symptom: Users encounter the following exception when calling a stored procedure, even though they have bound input parameters of a user-defined type and registered output parameters for the procedure:

```
java.lang.RuntimeException: org.postgresql.util.PSQLException: ?? CallableStatement ?????????????
java.sql.Types=1111 ?? 1?????????? java.sql.Types=2002?
```

Cause: The **enableGaussArrayAndStruct** parameter is set to **true** when pgArray compatibility is in effect. However, it is impossible to maintain compatibility with both pgArray and GaussArray simultaneously.

Solution: When pgArray compatibility is in effect, check for and delete **enableGaussArrayAndStruct=true**.

13.1.4 Batch Query

13.1.4.1 Scenario Overview

This section explains how to use JDBC for batch query.

13.1.4.1.1 Usage Scenarios

Scenario Description

Returning a large number of query results to JDBC at once may lead to JVM memory overflow. To decrease JVM memory usage, consider using batch queries, where you can specify the number of data records to be returned to JDBC each time.

Trigger Conditions

Java applications connect to the database through JDBC and query data in large batches.

Impact on Services

Developers must start and stop transactions correctly prior to database operations. In batch query scenarios, data is processed in smaller batches to effectively reduce JVM memory usage. This approach can prevent the problem of Java application memory overflow caused by clients receiving all data at once in the traditional full data transmission mode. However, it is worth noting that traversing the result set increases the number of network interactions between the database and the client (especially when each record needs to be transmitted independently). This can lead to additional performance loss, necessitating a thoughtful balance during system design.

13.1.4.1.2 Requirements and Objectives

Service Pain Points

In data-intensive service scenarios, traditional queries face the following challenges:

1. Large result sets can deplete application memory, resulting in JVM memory overflow and query failures.
2. Obtaining a large amount of data at once may saturate the network bandwidth, impacting data transmission efficiency.
3. The continuous occupation of database connections and related resources impedes overall throughput.

Service Objectives

Execute batch queries to prevent memory overflow.

13.1.4.2 Architecture Principles

Core Principles

During batch queries, JDBC leverages the cursor traversal capability of GaussDB Kernel to obtain a specific number of rows of data from the database server. This number is predetermined. The database server then returns result data to JDBC in batches based on the specified number until all query results are returned.

Solution Advantages and Disadvantages

1. During batch queries, the database returns a specific number of rows of data each time. This helps prevent memory overflow in Java applications when dealing with large result sets.
2. Batch queries involve multiple interactions with the database network, which can lead to a certain level of performance loss.

13.1.4.3 Preparations

- JDK version: 1.7 or later.
- Database environment: GaussDB 503.0 or later.
- JDBC driver environment:

  Refer to "Application Development Guide > Development Based on JDBC > Development Procedure > Obtaining the JAR Package of the Driver and Configuring the JDK Environment" in *Developer Guide*.

  Data: Create a test table and insert test data, as follows:

```
gaussdb=# CREATE TABLE tab_test(id int,context varchar(1000),PRIMARY KEY(id));
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "tab_test_pkey" for table "tab_test"
CREATE TABLE
gaussdb=# INSERT INTO tab_test SELECT generate_series(1,5),repeat('GaussDB Test', 50);
INSERT 0 5
```

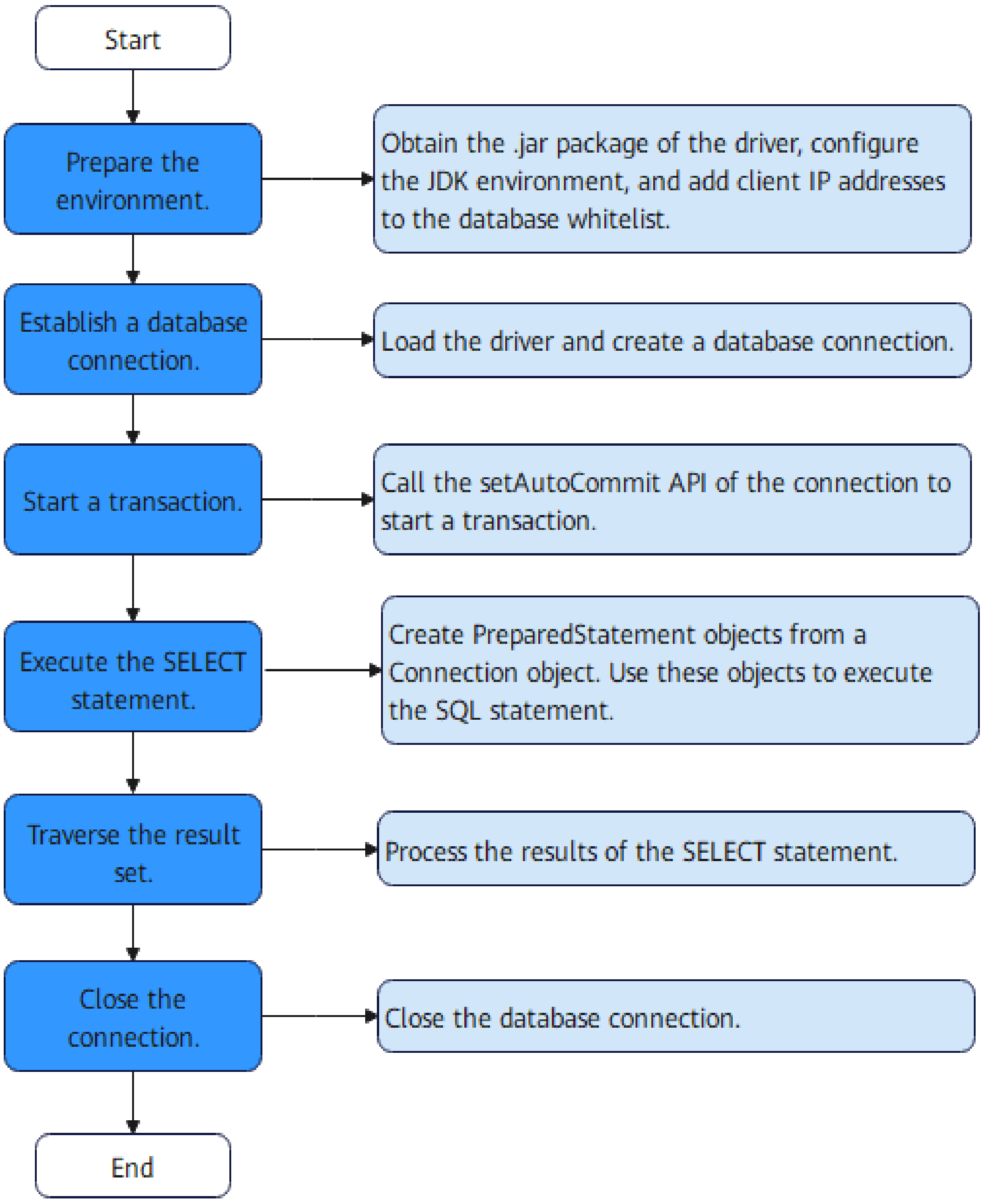
13.1.4.4 Procedure

13.1.4.4.1 Process Overview

Figure 13-5 shows the process of executing a batch query with JDBC.

This process includes preparing the environment, establishing a database connection, starting a transaction, executing SELECT statements, traversing the result set, and closing the connection.

Figure 13-5 Process of executing a batch query with JDBC


Start
Prepare the environment.
Obtain the .jar package of the driver, configure the JDK environment, and add client IP addresses to the database whitelist.
Establish a database connection.
Load the driver and create a database connection.
Start a transaction.
Call the setAutoCommit API of the connection to start a transaction.
Execute the SELECT statement.
Create PreparedStatement objects from a Connection object. Use these objects to execute the SQL statement.
Traverse the result set.
Process the results of the SELECT statement.
Close the connection.
Close the database connection.
End

13.1.4.4.2 Detailed Procedure

Step 1 Create a Connection object to connect to the database.

Here are suggestions for commonly used parameters in the connection string. For more detailed settings, refer to "Application Development Guide > Development Based on JDBC > Development Procedure > Connecting to a Database > Connection Parameter Reference" in *Developer Guide*.

- **connectTimeout**: timeout interval (in seconds) for connecting to the server's OS. If the time taken for JDBC to establish a TCP connection with the database exceeds this interval, the connection will be closed. It is advisable to set this parameter based on network conditions. The default value is **0**, whereas the recommended value is **2**.

- **socketTimeout**: timeout interval (in seconds) for socket reads. If the time taken to read data streams from the server exceeds this interval, the connection will be closed. Not setting this parameter may lead to prolonged waiting times for the client in the event of abnormal database processes. It is advisable to set this parameter based only the acceptable SQL execution time for services. The default value is **0**, with no specific recommended value provided.
- **connectionExtraInfo**: specifies whether the driver reports its deployment path, process owner, and URL connection configurations to the database. The default value is **false**, whereas the recommended value is **true**.
- **logger**: specifies a third-party log framework as needed by your application. It is advisable to choose one that incorporates slf4j APIs. These APIs can record JDBC logs to facilitate exception locating. The recommended value is **Slf4JLogger** when a third-party log framework is needed.
- **autoBalance**: specifies whether to enable load balancing for establishing new connections. The default value is **false**, but it is advisable to set it to **true** to utilize the polling load balancing policy.

```
String url = "jdbc:gaussdb://$ip:$port/database?
connectTimeout=xx&socketTimeout=xx&connectionExtraInfo=true&logger=Slf4JLogger&autoBalance=t
rue"
Connection conn = DriverManager.getConnection("url",userName,password);
```

Step 2 Start a transaction.

Set **AutoCommit** to **false** so that JDBC will deliver "BEGIN" to proactively start a transaction prior to executing a query from the database.

```
conn.setAutoCommit(false);
```

Step 3 Create PreparedStatement objects and specify the number of rows to be returned by the database each time.

Use the setFetchSize method to specify this number at the statement level. If **fetchsize** has been set in the connection string, it will be overridden by the setFetchSize method.

```
String selectSql = "select * from tab_test";
PreparedStatement preparedStatement = conn.prepareStatement(selectSql);
preparedStatement.setFetchSize(3);
```

Step 4 Execute a query to obtain a result set.

```
ResultSet resultSet = preparedStatement.executeQuery();
```

Step 5 Process the result set and check data in the first column of the table.

```
while (resultSet.next()) {
    int id = resultSet.getInt(1);
    System.out.println("row:" + resultSet.getRow() + ",id :" + id);
}
```

Step 6 Obtain metadata in the result set, including the column count and types.

Obtain metadata from the **resultSet** returned by executeQuery.

```
ResultSetMetaData metaData = resultSet.getMetaData();
System.out.println("Result column: " + metaData.getColumnCount());
System.out.println("Type ID: " + metaData.getColumnType(1));
System.out.println("Type name: " + metaData.getColumnTypeName(1));
System.out.println("Column name: " + metaData.getColumnName(1));
```

Step 7 Close resources.

Use try-with-resources to automatically close any open file resources.

```
try (Connection conn = getConnection(); PreparedStatement preparedStatement =
conn.prepareStatement(selectSql))
```

Step 8 (Optional) Handle exceptions if any.

If your program encounters any exception during runtime, utilize the try-catch module to handle them and add the necessary exception handling logic for your services.

```
try {
// Service code
} catch (Exception e) {
// Exception handling logic
}
```

----End

13.1.4.4.3 Complete Example

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

public class BatchQuery {
    public static Connection getConnection() throws ClassNotFoundException, SQLException {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        // Specify the source URL of the database. (Adjust $ip, $port, and database based on the actual
services.)
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";
        // Obtain the username and password from the environment variables.
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        Class.forName(driver);
        return DriverManager.getConnection(sourceURL, userName, password);
    }

    public static void main(String[] args) {
        String selectSql = "select * from tab_test";
        try (Connection conn = getConnection(); PreparedStatement preparedStatement =
conn.prepareStatement(selectSql)) {
            conn.setAutoCommit(false);
            preparedStatement.setFetchSize(3);
            try (ResultSet resultSet = preparedStatement.executeQuery()) {
                while (resultSet.next()) {
                    // Print only a portion of the query results.
                    int id = resultSet.getInt(1);
                    System.out.println("row:" + resultSet.getRow() + ",id :" + id);
                }
                ResultSetMetaData metaData = resultSet.getMetaData();
                System.out.println("Result column: " + metaData.getColumnCount());
                System.out.println("Type ID: " + metaData.getColumnType(1));
                System.out.println("Type name: " + metaData.getColumnTypeName(1));
                System.out.println("Column name: " + metaData.getColumnName(1));
            }
            conn.commit();
        } catch (ClassNotFoundException | Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Result Verification

Below are the execution results for **Complete Example**:

```
row:1,id :1
row:2,id :2
row:3,id :3
row:4,id :4
row:5,id :5
Result column: 2
Type ID: 4
Type name: int4
Column name: id
```

Rollback Method

To disable batch query, remove the **fetchsize** parameter from the connection string and reset **setFetchSize** to the default value of **0**.

13.1.4.5 Typical Issues

1. Symptom: Customers using the Spring framework have set the **fetchsize** parameter in the connection string, but OutOfMemoryError is reported during a batch query.

   Cause: Transactions are not started prior to the batch query. Generally, connection transactions are managed by the Spring framework. However, if transactions are not started, the database will return all data to JDBC in one go.

   Solution: Check the service code and be sure to start transactions before executing batch queries.
2. Symptom: The **fetchsize** parameter has been set in the connection string, but OutOfMemoryError is reported during a batch query.

   Cause: Other call points in the application code consume substantial memory. In this situation, JDBC may encounter OutOfMemoryError when it attempts to read only a small amount of data.

   Solution: Use JDK to check memory usage and determine whether the memory overflow is caused by table data.
3. Symptom: The table query speed is fast with gsql but slow with JDBC.

   Cause: The **fetchsize** parameter has been set to a small value in the connection string. This results in a high number of packet interactions between Kernel and the result set, leading to performance degradation.

   Solution: Before querying data, determine whether to start transactions based on the table size or call the prepareStatement.setFetchSize() method to adjust the number of rows to be returned by the database each time. If this number is set to **0**, all query results will be returned in one go.

13.2 Best Practices for JDBC (Centralized Instances)

13.2.1 Batch Insertion

13.2.1.1 Scenario Overview

This section explains how to use JDBC for batch data insertion.

13.2.1.1.1 Usage Scenarios

Scenario Description

When you have a large data size to insert into a database, using batch insertion is much more efficient than executing SQL statements multiple times, one at a time.

This section illustrates various operations using the JDBC driver, including establishing database connections, utilizing transactions, executing batch insertion, and obtaining column information in the result set.

Trigger Conditions

JDBC adds SQL statements to the batch execution list through its addBatch API. The batch operation is then executed through the executeBatch API.

Impact on Services

- Lower network interaction costs

  Combining multiple INSERT statements into one batch operation significantly reduces the number of round trips between the client and the database. This enhances the overall throughput and minimizes the impact of network congestion on performance.

- Higher data processing efficiency

  In single-record insertion, the database must parse the syntax and generate an execution plan for each SQL statement. In contrast, batch insertion requires parsing the syntax and generating the plan only once, eliminating repetitive tasks and saving CPU cycles and memory allocation time.

- Reduced system resource usage and overhead

  In single-record insertion, transaction commits or Xlog writes occur at least once. In contrast, batch insertion allows multiple records to be inserted within a single transaction, significantly reducing the frequency of transaction commits, Xlog pressure, and transaction management overhead. In addition, it decreases the total number of network packet processing, transaction management, log write, and row format conversion tasks, which in turn lowers the CPU loads and temporary memory usage of the database server. This results in more resources being available for core query and computing operations.

- Higher memory usage

  When large data sizes are involved, constructing SQL statements for batch insertion can significantly increase memory usage. This is particularly noticeable when you construct SQL statements through string concatenation, as it can lead to a sharp rise in memory consumption. Large-size batch processing may exceed the maximum SQL length limit of the database or driver, or trigger other parameter restrictions, potentially leading to errors or performance issues.

Here is a detailed comparison between batch insertion and single-record insertion.

Mode	Advantages	Disadvantages
Single-record insertion	● Its code is simple, straightforward, and easy to implement. ● If any single record fails, it can be accurately identified and handled without impacting other records. ● This mode is less demanding in terms of database and driver compatibility.	● Extensive network interactions are needed. Each INSERT operation requires connecting, parsing, and committing, leading to suboptimal performance. ● Inserting a large number of records is likely to cause a bottleneck. ● Not using transactions may result in failure to guarantee the consistency of INSERT operations.
Batch insertion	● This mode greatly reduces the number of network round trips and SQL parsing instances, leading to a notable improvement in insertion throughput. ● Multiple rows can be committed within a single transaction to guarantee atomicity.	● Its code is complex, requiring manual concatenation of placeholders and parameters. ● If a single statement encounters an error, all data will be rolled back, complicating the error recovery process. ● The number of placeholders is limited; therefore, it is essential to carefully manage the batch size.

Applicable Versions

This applies only to GaussDB 503.1.0 and later versions.

13.2.1.1.2 Requirements and Objectives

Service Pain Points

When dealing with large data sizes, single-record insertion generates numerous network requests and consumes substantial system resources. Moreover, the database server has to repeatedly parse similar statements, leading to a decline in service performance. Batch insertion is introduced as a solution to these issues.

Service Objectives

Utilize JDBC to implement batch insertion with transactions, obtain column data in the result set, and output the result information.

13.2.1.2 Architecture Principles

Core Principles

Batch processing of the JDBC driver allows adding multiple SQL statements to the batch execution list and collectively sending them to the database in one go. All insert or update operations in the transaction are carried in a single U packet. Consequently, completing the batch operation only necessitates once instance of network connection establishment and data exchange.

Solution Advantages

Sending all batch updates at once in a single U packet significantly decreases network communication overhead and enhances execution efficiency when compared to sending PBE packets multiple times.

13.2.1.3 Preparations

- JDK version: 1.7 or later.
- Database environment: GaussDB 503.1.0 or later.
- JDBC driver environment:

  Refer to "Application Development Guide > Development Based on JDBC > Development Procedure > Obtaining the JAR Package of the Driver and Configuring the JDK Environment" in *Developer Guide*.
- Data: Create a test table and insert test data, as follows:

```
gaussdb=# CREATE TABLE TEST_BATCH(
V1 TEXT,
V2 TEXT);
CREATE TABLE
```

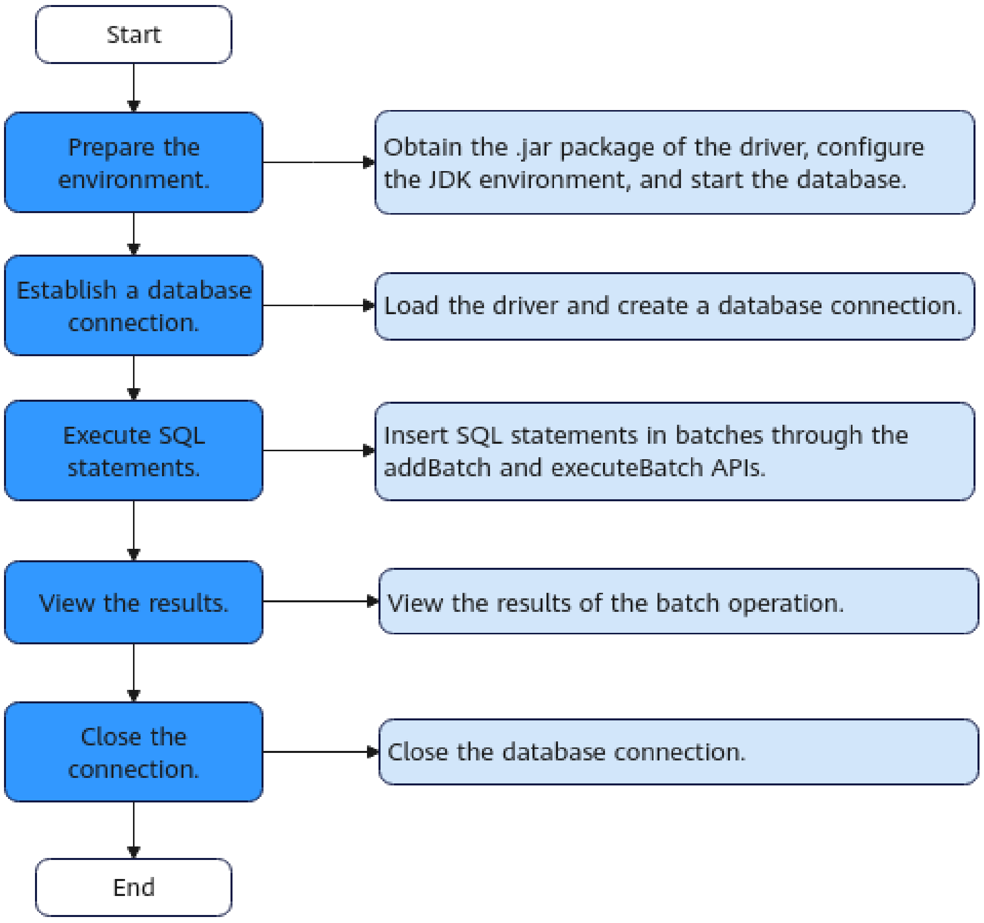
13.2.1.4 Procedure

13.2.1.4.1 Process Overview

The process of batch data insertion using JDBC includes preparing the environment, establishing a database connection, executing batch insertion through APIs, querying the execution results, and closing the connection.

Figure 13-6 shows the overall process.

Figure 13-6 Process of batch data insertion using JDBC


Start
Prepare the environment.
Obtain the .jar package of the driver, configure the JDK environment, and start the database.
Establish a database connection.
Load the driver and create a database connection.
Execute SQL statements.
Insert SQL statements in batches through the addBatch and executeBatch APIs.
View the results.
View the results of the batch operation.
Close the connection.
Close the database connection.
End

13.2.1.4.2 Detailed Procedure

Step 1 Establish a database connection.

Here are suggestions for commonly used parameters in the connection string. For more detailed settings, refer to "Application Development Guide > Development Based on JDBC > Development Procedure > Connecting to a Database > Connection Parameter Reference" in *Developer Guide*.

- **connectTimeout**: timeout interval (in seconds) for connecting to the server's OS. If the time taken for JDBC to establish a TCP connection with the database exceeds this interval, the connection will be closed. It is advisable to set this parameter based on network conditions. The default value is **0**, whereas the recommended value is **2**.
- **socketTimeout**: timeout interval (in seconds) for socket reads. If the time taken to read data streams from the server exceeds this interval, the connection will be closed. Not setting this parameter may lead to prolonged waiting times for the client in the event of abnormal database processes. It is advisable to set this parameter based only the acceptable SQL execution time for services. The default value is **0**, with no specific recommended value provided.
- **connectionExtraInfo**: specifies whether the driver reports its deployment path, process owner, and URL connection configurations to the database. The default value is **false**, whereas the recommended value is **true**.

- **logger**: specifies a third-party log framework as needed by your application. It is advisable to choose one that incorporates slf4j APIs. These APIs can record JDBC logs to facilitate exception locating. The recommended value is **Slf4JLogger** when a third-party log framework is needed.
- **batchMode**: specifies whether the connection operates in batch mode. When **batchMode** is set to **on**, batch insertion and modification are allowed, with the data type of each column determined by the type specified in the first data record.

```
String url = "jdbc:gaussdb://$ip:$port/database?
connectTimeout=xx&socketTimeout=xx&connectionExtraInfo=true&logger=Slf4JLogger&autoBalance=t
rue&batchMode=on"
Connection conn = DriverManager.getConnection("url",userName,password);
```

Step 2 Prepare SQL statements for batch execution.

Use preparedStatement to prepare statements. For example, to insert data into a test table, prepare the statements provided below. You can substitute these statements with the necessary ones for your services.

```
String sql = "INSERT INTO  TEST_BATCH(v1,v2) VALUES(?,?)";
PreparedStatement preparedStatement = conn.prepareStatement(sql);
```

Step 3 Bind parameters in batches.

Use preparedStatement to bind parameters. Then call the addBatch API to add the SQL statements to the batch execution list.

```
for(int i=0;i<5;i++){
    preparedStatement.setString(1,"value1_"+i);
    preparedStatement.setString(2,"value2_"+i);
    preparedStatement.addBatch();
}
```

Step 4 Execute the batch operation.

By calling the executeBatch API, preparedStatement executes the batch operation. Upon completion, it returns an array of int results, from which you can determine the number of data records affected by the batch operation.

```
Int[] results = preparedStatement.executeBatch();
System.out.println(Arrays.toString(results));
```

Step 5 Release resources and close the database connection.

```
preparedStatement.close();
conn.close();
```

Step 6 Handle exceptions if any.

If your program encounters any SQL exception during runtime, utilize the try-catch module to handle them and add the necessary exception handling logic for your services.

```
try {
// Service code
} catch (SQLException e) {
// Exception handling logic
}
```

----End

13.2.1.4.3 Complete Example

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Arrays;
import java.sql.DriverManager;
public class TestBatch {
    public static Connection getConnection() throws ClassNotFoundException, SQLException{
 String driver = "com.huawei.gaussdb.jdbc.Driver";
        // Specify the source URL of the database. (Adjust $ip, $port, and database based on the actual services.)
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";
        // Obtain the username and password from the environment variables.
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        Class.forName(driver);
        return DriverManager.getConnection(sourceURL, userName, password);
    }
    public static void main(String[] args) {
 String sql = "insert into test_batch(v1,v2) values(?,?)";
        try (Connection conn = getConnection(); PreparedStatement preparedStatement = 
conn.prepareStatement(sql)) {
            conn.setAutoCommit(false);
            for (int i = 0; i < 5; i++) {
                preparedStatement.setInt(1, 1);
                preparedStatement.setString(2, "value2_" + i);
                preparedStatement.addBatch();
            }
            int[] results = preparedStatement.executeBatch();
            conn.commit();
            System.out.println(Arrays.toString(results));
        } catch (ClassNotFoundException | SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Result Verification

The preceding execution result shows the number of data records affected by the batch operation.

When you use preparedStatement for batch insertion, it returns an array of INT results, with **results[0]** indicating the total number of data records affected by the batch operation.

```
[5, 0, 0, 0, 0]
```

Rollback Method

To roll back operations within a specific transaction, call the Rollback API of the transaction object.

13.2.1.5 Typical Issues

1. Symptom: When preparedStatement binds parameters multiple times, the following error occurs during batch insertion:
   ```
   java.lang.RuntimeException: java.sql.BatchUpdateException: Batch entry 0 - 5 insert into
   test_batch1(v1,v2) values(('1'),('value2_0')) was aborted: [*.*.*.*:*/*.*.*.*:*] ERROR: invalid input syntax
   for integer: "ss"  Call getNextException to see other errors in the batch.
   ```
   Cause: The data type of a bound parameter differs from the initially bound parameter.

   Solution: To ensure successful batch insertion using preparedStatement, maintain consistency in the data types of bound parameters.
2. Symptom: The array of results returned by preparedStatement after batch insertion does not match the results returned by Oracle Database.

Cause: When **batchMode** is set to **on**, JDBC utilizes GaussDB's distinct packet processing logic, which differs from Oracle Database's logic but offers faster speed. Setting **batchMode** to **off** in the connection string will ensure consistency with Oracle Database's results.

When **batchMode** is set to **on**, **Complete Example** will produce results that differ from those generated by Oracle Database's corresponding API.

```
[5, 0, 0, 0, 0]
```

Oracle Database's results:

```
[1, 1, 1, 1, 1]
```

When **batchMode** is set to **off**, the results become consistent with Oracle Database's corresponding API.

```
[1, 1, 1, 1, 1]
```

13.2.2 Streaming Query

13.2.2.1 Scenario Overview

This section explains how to execute a streaming query with GaussDB JDBC.

13.2.2.1.1 Usage Scenarios

Scenario Description

GaussDB JDBC's streaming query mechanism processes results one by one rather than loading them all at once. It is designed for big data query scenarios with limited memory resources, such as big data export, offline analysis tasks, and pagination/on-demand loading. This mechanism helps prevent excessive memory consumption and potential memory overflow, ultimately enhancing processing speed.

Trigger Conditions

The JDBC connection parameter **enableStreamingQuery** is set to **true**. In addition, **fetchSize** is set to **Integer.MIN_VALUE** before the executeQuery method of Statement and PreparedStatement is called.

Impact on Services

Streaming query offers the following advantages:

- Low memory consumption: Streaming query significantly reduces the memory usage of applications.
- Fast response: Clients can start processing immediately upon receiving the first batch of data, without having to wait for the entire data set to be ready.

However, there are also associated risks:

- Prolonged connection occupation: Streaming query requires the database connection to remain open throughout the transmission of data streams. This prolonged connection occupation may result in connection pool depletion or other resource scheduling issues.

- Extended transaction duration: Transactions involving streaming queries may hold locks or MVCC snapshots for an extended period, increasing the probability of deadlocks or impacting write performance.

Applicable Versions

This applies only to GaussDB 505.1.0 and later versions.

13.2.2.1.2 Requirements and Objectives

Service Pain Points

In a standard query, JDBC will receive a significant amount of data when querying massive data. Reading all this data in full can potentially cause memory overflow.

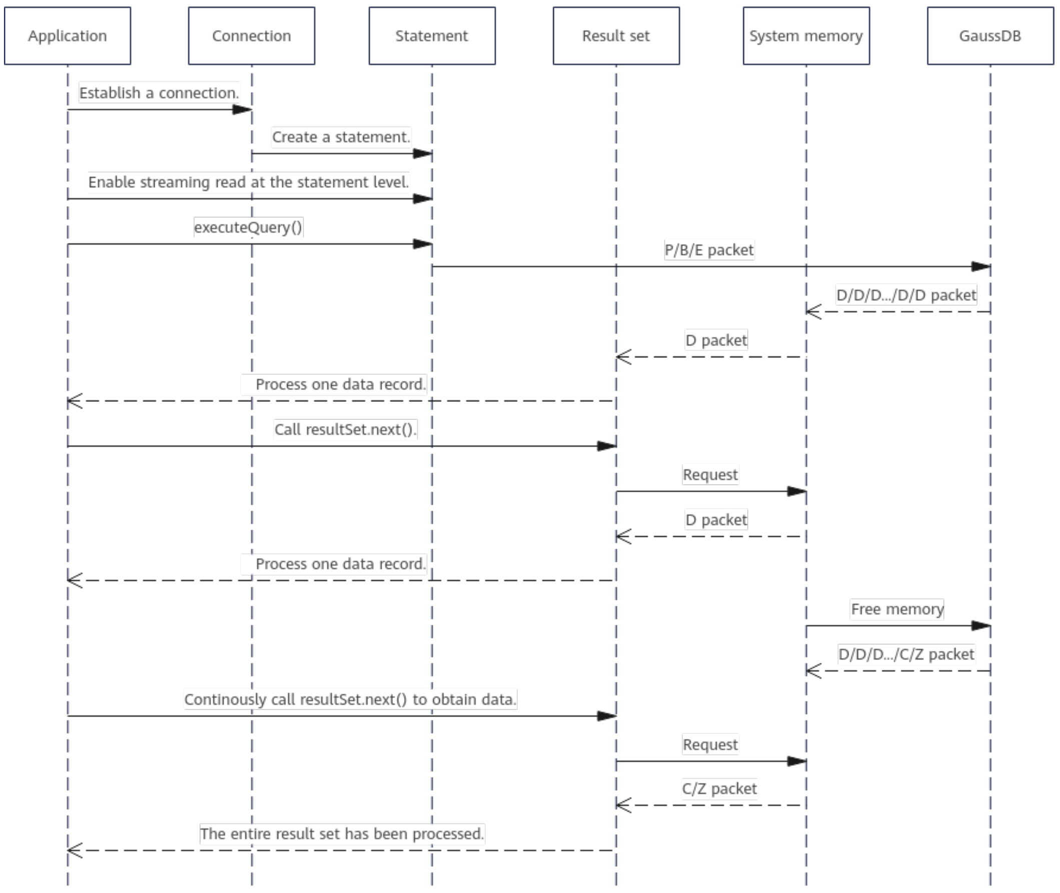
Service Objectives

Execute streaming queries with JDBC to prevent memory overflow.

13.2.2.2 Architecture Principles

Core Principles

Figure 13-7 Principles of streaming query



- When a streaming query is executed, the server keeps sending data to the client's socket buffer until the buffer is full. Upon the arrival of the first data record in the buffer, the first D packet is returned. JDBC then begins loading and processing data from the buffer row by row.
- The result set is designed to store only one row of data. The next data record is read from the buffer and stored into the result set only when the "next" method is called. This process repeats until all records have been read.

Solution Advantages

Streaming query is recommended for big data query scenarios with limited memory resources. If streaming query does not meet your service requirements, consider the following alternative methods:

- Standard query: All data is read in one go.

  Advantages: This method allows traversal of the entire result set in both forward and backward directions. In addition, data can be reprocessed after being consumed.

  Disadvantages: Processing is slow, and there is a risk of memory overflow with large result sets.
- Batch query: Multiple rows are read at a time. For details, see **Batch Query**.

  Advantages: Data is returned in the specified size, which helps prevent memory overflow.

  Disadvantages: Processing is slow, and multiple query requests need to be sent to the server.

13.2.2.3 Preparations

- Environment: Ensure that the database is running properly, obtain the JDBC driver, and configure the environment. For details, see "Application Development Guide > Development Based on JDBC > Development Procedure > Obtaining the JAR Package of the Driver and Configuring the JDK Environment" in *Developer Guide*.
- Data: Create a test table and insert test data, as follows:

```
gaussdb=# CREATE TABLE tab_test(id int,context varchar(1000),PRIMARY KEY(id));
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "tab_test_pkey" for table "tab_test"
CREATE TABLE
gaussdb=# INSERT INTO tab_test SELECT generate_series(1,1000000),repeat('GaussDB Test', 50);
INSERT 0 1000000
```

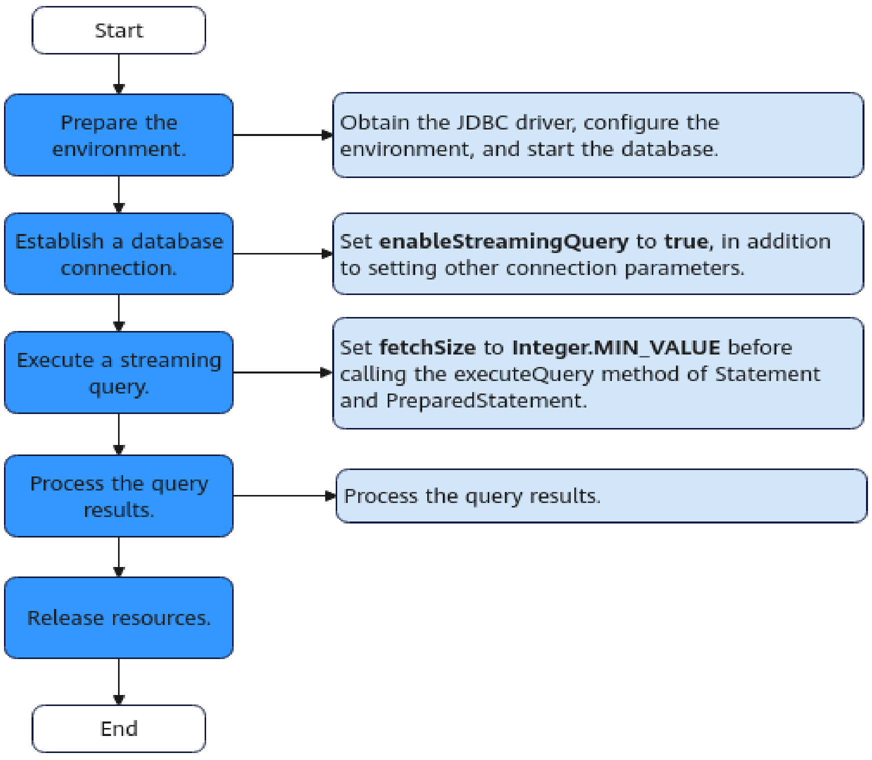
13.2.2.4 Procedure

13.2.2.4.1 Process Overview

The process of executing a streaming query with GaussDB JDBC includes preparing the environment, establishing a database connection, executing a streaming query, processing the query results, and releasing resources.

Figure 13-8 shows the overall process.

Figure 13-8 Process of executing a streaming query with GaussDB JDBC


Start
Prepare the environment.
Obtain the JDBC driver, configure the environment, and start the database.
Establish a database connection.
Set **enableStreamingQuery** to **true**, in addition to setting other connection parameters.
Execute a streaming query.
Set **fetchSize** to **Integer.MIN_VALUE** before calling the executeQuery method of Statement and PreparedStatement.
Process the query results.
Process the query results.
Release resources.
End

13.2.2.4.2 Detailed Procedure

Step 1 Database connection: Set **enableStreamingQuery** to **true**, in addition to setting other connection parameters. For details, see "Application Development Guide > Development Based on JDBC > Development Procedure > Connecting to a Database" in *Developer Guide*.

Step 2 Streaming query: Set **fetchSize** to **Integer.MIN_VALUE** before calling the executeQuery method of Statement and PreparedStatement.

Step 3 Result processing: Process the query results based on the actual services.

Step 4 Resource release: Upon completion, release the result set, statement, connection, and other resources. It is advisable to use the try-with-resources syntax or the close method in the try-finally block to release resources.

----End

13.2.2.4.3 Complete Example

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class StreamQueryTest {
    // Establish a database connection.
    public static Connection getConnection() throws ClassNotFoundException, SQLException {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        // Specify the source URL of the database. (Adjust $ip, $port, and database as needed.) Set the
connection parameter enableStreamingQuery to true.
        String sourceURL = "jdbc:gaussdb://$ip:$port/database?enableStreamingQuery=true";
        // Obtain the username and password from the environment variables.
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        Class.forName(driver);
        return DriverManager.getConnection(sourceURL, userName, password);
    }

    public static void main(String[] args) {
        String selectSql = "select * from tab_test order by id asc limit ?";
        // Use the try-with-resources syntax to release resources.
        try (Connection conn = getConnection(); PreparedStatement preparedStatement = 
conn.prepareStatement(selectSql,
            ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY)) {
            preparedStatement.setInt(1, 100000);
            // Set fetchSize to Integer.MIN_VALUE and call executeQuery to query data.
            preparedStatement.setFetchSize(Integer.MIN_VALUE);
            int totalCount = 0;
            try (ResultSet resultSet = preparedStatement.executeQuery()) {
                while (resultSet.next()) {
                    // Process the query results. The sample code below prints only a portion of the query results 
along with the total number of data records.
                    if (totalCount++ < 5) {
                        System.out.println("row:" + resultSet.getRow() + ",id :" + resultSet.getInt(1));
                    }
                }
                System.out.println("totalCount:" + totalCount);
            }
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Result Verification

1. Place **gaussdbjdbc.jar** and **StreamQueryTest.java** from **Complete Example** into the same directory.
2. Compile and execute the sample code. During execution, set the maximum heap memory of JVM to 16 MB.
```
javac -classpath ".:gaussdbjdbc.jar" StreamQueryTest.java
java -Xmx16M -classpath ".:gaussdbjdbc.jar" StreamQueryTest
```
3. Run **Complete Example**. It completes successfully without encountering memory overflow. During the streaming query, the result set stores only one data record at a time. Therefore, the return value of **resultSet.getRow()** is **1**. The execution result of the code is like this:
```
row:1,id :1
row:1,id :2
row:1,id :3
row:1,id :4
row:1,id :5
totalCount:100000
```

Rollback Method

- To disable streaming query for a single statement, delete **setFetchSize(Integer.MIN_VALUE)** from Statement and PreparedStatement.
- To disable streaming query for the current connection, set the connection parameter **enableStreamingQuery** to **false**.

13.2.2.5 Typical Issues

1. Symptom: The following exception occurs when **fetchSize** is set to **Integer.MIN_VALUE** in Statement and PreparedStatement:

```
org.postgresql.util.PSQLException: Fetch size must be a value greater to or equal to 0.
    at org.postgresql.jdbc.PgStatement.setFetchSize(PgStatement.java:1623)
```

Cause: The JDBC connection parameter **enableStreamingQuery** is not set to **true**.

2. Symptom: The following exception occurs when the execute, executeQuery, and executeUpdate methods of Statement and PreparedStatement are called to query, insert, and update data:

```
org.postgresql.util.PSQLException: Streaming query statement is still active. No statements may be
issued when any streaming result sets are open and in use on a given connection. Ensure that you
have called .close() on any active streaming statement sets before attempting more queries.
    at org.postgresql.jdbc.PgStatement.execute(PgStatement.java:492)
    at org.postgresql.jdbc.PgPreparedStatement.executeWithFlags(PgPreparedStatement.java:210)
    at org.postgresql.jdbc.PgPreparedStatement.executeQuery(PgPreparedStatement.java:147)
```

Cause: The result set from the previous streaming query has not been completely read or has not been closed.

3. Sequential access only: Streaming query utilizes the FORWARD_ONLY result set, which means that data can only be accessed row by row in the forward direction. It does not support randomly locating or rolling back to previous records. If your services involve frequent traversal or require random data access, do not use streaming query.
4. Configuration and compatibility requirements: Streaming query requires special configurations (**Trigger Conditions**) in the JDBC driver or ORM framework. Without proper configurations, the desired outcome cannot be achieved.

13.2.3 User-defined Type

13.2.3.1 Scenario Overview

This section explains how JDBC can use user-defined data types.

13.2.3.1.1 Usage Scenarios

Scenario Description

In real-world production, users' service systems often involve complex data analysis and operations. Traditional data types may not fully or efficiently represent or process their data scenarios. To address this, JDBC provides user-defined data types, Struct and Array, enabling the creation of new data types from existing ones. These user-defined types offer users a more convenient way to add, delete, and modify data.

Trigger Conditions

JDBC operations involve user-defined data types.

Impact on Services

- Improved data consistency and integrity

  User-defined types allow users to declare data types and check constraints for columns during the type definition phase, ensuring that all columns referencing these types automatically inherit the same verification logic.

- Improved data reusability and encapsulation

  User-defined types encapsulate frequently used data structures for reuse across multiple tables or functions. This minimizes redundant definitions and enhances maintenance efficiency.
- Additional performance overhead

  Using user-defined types will take up additional storage space and necessitate object assembly and disassembly during access, potentially leading to increased CPU and I/O usage.

Applicable Versions

This applies only to GaussDB 503.0 and later versions.

13.2.3.1.2 Requirements and Objectives

Service Pain Points

When dealing with complex data structures in services, basic data types may not accurately express complex service concepts, leading to potential code redundancy and impacting development efficiency.

Service Objectives

Utilize user-defined types in JDBC's stored procedures.

13.2.3.2 Architecture Principles

Core Principles

The database system creates metadata records for each user-defined type, including the type name, structure, and constraints. When working with user-defined types, JDBC converts them based on their structures as well as the user-defined type objects in the programming language.

Solution Advantages

- Support for complex data modeling

  The ability to process complex data structures, such as nested objects and arrays, improves the expression capability of data models.
- Improved data quality and development efficiency

  When querying complex data structures, you can utilize user-defined types for filtering. Reusing user-defined types helps enhance development efficiency.

13.2.3.3 Preparations

- JDK version: 1.7 or later.
- Database environment: GaussDB 503.0 or later.
- JDBC driver environment:

  Refer to "Application Development Guide > Development Based on JDBC > Development Procedure > Obtaining the JAR Package of the Driver and Configuring the JDK Environment" in *Developer Guide*.

- Data: Create a user-defined type and stored procedure, as follows:

```
// Create a user-defined type, PUBLIC.COMPFOO, that includes two records.
gaussdb=# CREATE TYPE PUBLIC.COMPFOO AS(
    ID INTEGER,
    NAME TEXT
);
CREATE TYPE
// Create a user-defined table type, PUBLIC.COMPFOO_TABLE.
gaussdb=# CREATE TYPE PUBLIC.COMPFOO_TABLE IS TABLE OF PUBLIC.COMPFOO;
CREATE TYEP
// Create the stored procedure public.test_proc with two input parameters of a user-defined type and
two output parameters of a user-defined type.
gaussdb=# CREATE OR REPLACE PROCEDURE public.test_proc(
    IN INPUT_COMPFOO PUBLIC.COMPFOO,
    IN INPUT_COMPFOO_TABLE PUBLIC.COMPFOO_TABLE,
    OUT OUTPUT_COMPFOO PUBLIC.COMPFOO,
    OUT OUTPUT_COMPFOO_TABLE PUBLIC.COMPFOO_TABLE
)
AS
BEGIN
    OUTPUT_COMPFOO := INPUT_COMPFOO;
    OUTPUT_COMPFOO_TABLE := INPUT_COMPFOO_TABLE;
END;
/
CREATE PROCEDURE
```

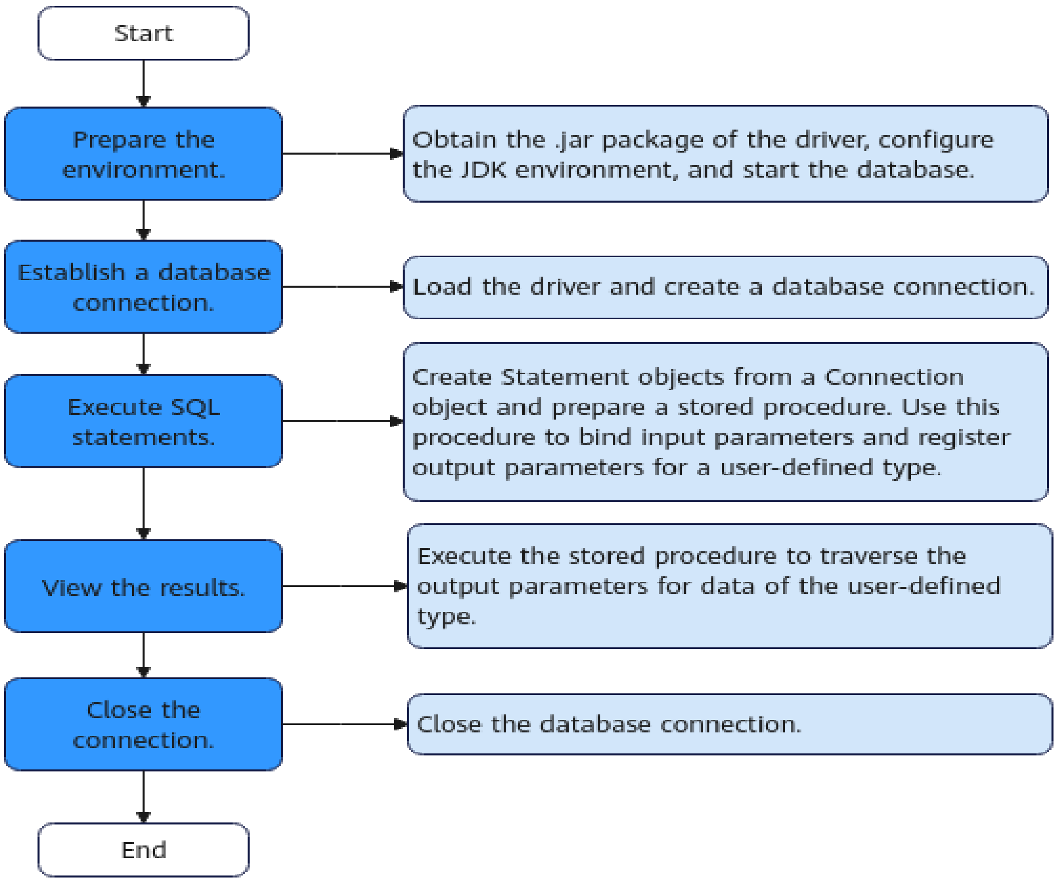
13.2.3.4 Procedure

13.2.3.4.1 Process Overview

Figure 13-9 shows the process for JDBC to insert and query data of user-defined types using a stored procedure.

This process includes preparing the environment, establishing a database connection, executing SQL statements of user-defined types, viewing the results, and closing the connection.

Figure 13-9 Process of using stored procedures in JDBC to handle user-defined types


Start
Prepare the environment.
Obtain the .jar package of the driver, configure the JDK environment, and start the database.
Establish a database connection.
Load the driver and create a database connection.
Execute SQL statements.
Create Statement objects from a Connection object and prepare a stored procedure. Use this procedure to bind input parameters and register output parameters for a user-defined type.
View the results.
Execute the stored procedure to traverse the output parameters for data of the user-defined type.
Close the connection.
Close the database connection.
End

13.2.3.4.2 Detailed Procedure

Step 1 Establish a database connection.

Here are suggestions for commonly used parameters in the connection string. For more detailed settings, refer to "Application Development Guide > Development Based on JDBC > Development Procedure > Connecting to a Database" in *Developer Guide*.

- **connectTimeout**: timeout interval (in seconds) for connecting to the server's OS. If the time taken for JDBC to establish a TCP connection with the database exceeds this interval, the connection will be closed. It is advisable to set this parameter based on network conditions. The default value is **0**, whereas the recommended value is **2**.
- **socketTimeout**: timeout interval (in seconds) for socket reads. If the time taken to read data streams from the server exceeds this interval, the connection will be closed. Not setting this parameter may lead to prolonged waiting times for the client in the event of abnormal database processes. It is advisable to set this parameter based only the acceptable SQL execution time for services. The default value is **0**, with no specific recommended value provided.
- **connectionExtraInfo**: specifies whether the driver reports its deployment path, process owner, and URL connection configurations to the database. The default value is **false**, whereas the recommended value is **true**.

- **logger**: specifies a third-party log framework as needed by your application. It is advisable to choose one that incorporates slf4j APIs. These APIs can record JDBC logs to facilitate exception locating. The recommended value is **Slf4JLogger** when a third-party log framework is needed.

```
String url = "jdbc:gaussdb://$ip:$port/database?
connectTimeout=xx&socketTimeout=xx&connectionExtraInfo=true&logger=Slf4JLogger&autoBalance=t
rue"
Connection conn = DriverManager.getConnection("url",userName,password);
```

Step 2 Set GUC parameters.

Execute the SQL statement **SET behavior_compat_options='proc_outparam_override';** to enable **proc_outparam_override**, that is, overloading of stored procedures.

```
Statement statement = conn.createStatement();
statement.execute("SET behavior_compat_options='proc_outparam_override'");
statement.close();
```

Step 3 Prepare a stored procedure.

Use the Call syntax to declare the SQL statement for calling the stored procedure TEST_PROC. Then use prepareCall to prepare the statement.

```
CallableStatement cs = conn.prepareCall("{CALL PUBLIC.TEST_PROC(?,?,?,?)}");
```

Step 4 Bind input parameters.

Use PGobject to assemble data of the user-defined type. Then use prepareCall to bind input parameters.

```
PGobject pgObject = new PGobject();
pgObject.setType("public.compfoo"); // Set the name of a composite type.
pgObject.setValue("(1,demo)"); // Bind values of the composite type, formatted as "(value1,value2)".
cs.setObject(1, pgObject);
pgObject = new PGobject();
pgObject.setType("public.compfoo_table"); // Set the name of a table type.
pgObject.setValue("{\"(10,demo10)\",\"(11,demo111)\"}"); // Bind values of the table type, formatted as
"{\"(value1,value2)\",\"(value1,value2)\",...}".
cs.setObject(2, pgObject);
```

Step 5 Register the output type.

Use prepareCall to register the output type of the stored procedure. For the composite type, register **Types.STRUCT**. For the table type, register **Types.ARRAY**.

```
// Register output parameters for the composite type.
cs.registerOutParameter(3, Types.STRUCT, "public.compfoo");
// Register output parameters for the table type.
cs.registerOutParameter(4, Types.ARRAY, "public.compfoo_table");
```

Step 6 Execute SQL statements and view the results.

Call the stored procedure and view the results corresponding to the output parameters.

```
cs.execute();
// Obtain output parameters.
// The return structure is of the user-defined type.
PGobject result = (PGobject) cs.getObject(3);  // Obtain output parameters.
result.getValue(); // Obtain character string values of the composite type.
String[] arrayValue = result.getArrayValue(); // Obtain array values of the composite type and sort them
based on columns of the composite type.
result.getStruct(); // Obtain the names of subtypes in the composite type and sort them based on the order
in which they were created.
result.getAttributes(); // Return objects of the user-defined type in each column. For the array and table
types, PgArray is returned. For the user-defined type, PGobject is encapsulated. Other types of data are
stored as character strings.
```

```
for (String s : arrayValue) {
    System.out.println(s);
}
PgArray pgArray = (PgArray) cs.getObject(4);
ResultSet rs = pgArray.getResultSet();
Object[] array = (Object[]) pgArray.getArray();
for (Object element : array) {
    System.out.println(element);
}
```

Step 7 Release resources and close the database connection.

```
cs.close();
conn.close();
```

Step 8 (Optional) Handle exceptions if any.

If your program encounters any SQL exception during runtime, utilize the try-catch module to handle them and add the necessary exception handling logic for your services.

```
try {
   // Service code
} catch (SQLException e) {
   // Exception handling logic
}
```

----End

13.2.3.4.3 Complete Example

```
import com.huawei.gaussdb.jdbc.jdbc.PgArray;
import com.huawei.gaussdb.jdbc.util.PGobject;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;
import java.sql.DriverManager;
public class TypesTest {
 public static Connection getConnection() throws ClassNotFoundException, SQLException {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        // Specify the source URL of the database. (Adjust $ip, $port, and database based on the actual 
services.)
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";
        // Obtain the username and password from the environment variables.
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        Class.forName(driver);
        return DriverManager.getConnection(sourceURL, userName, password);
    }
 public static void main(String[] args) {
  try {
   Connection conn = getConnection();
   Statement statement = conn.createStatement();
   statement.execute("set behavior_compat_options='proc_outparam_override'");
   statement.close();
   CallableStatement cs = conn.prepareCall("{CALL PUBLIC.TEST_PROC(?,?,?,?)}");
   // Set parameters.
   PGobject pgObject = new PGobject();
   pgObject.setType("public.compfoo"); // Set the name of a composite type.
   pgObject.setValue("(1,demo)"); // Bind values of the composite type.
   cs.setObject(1, pgObject);
   pgObject = new PGobject();
   pgObject.setType("public.compfoo_table"); // Set the name of a table type.
   pgObject.setValue("{\"(10,demo10)\",\"(11,demo111)\"}"); // Bind values of the table type, formatted as 
"{\"(value1,value2)\",\"(value1,value2)\",...}".
   cs.setObject(2, pgObject);
   // Register output parameters.
   // Register output parameters for the composite type.
   cs.registerOutParameter(3, Types.STRUCT, "public.compfoo");
   // Register output parameters for the table type.
   cs.registerOutParameter(4, Types.ARRAY, "public.compfoo_table");
   cs.execute();
   // Obtain output parameters.
  // The return structure is of the user-defined type.
   PGobject result = (PGobject) cs.getObject(3); // Obtain output parameters.
   result.getValue(); // Obtain character string values of the composite type.
   String[] arrayValue = result.getArrayValue(); // Obtain array values of the composite type and sort them
based on columns of the composite type.
   result.getStruct(); // Obtain the names of subtypes in the composite type and sort them based on the
order in which they were created.
   result.getAttributes(); // Return objects of the user-defined type in each column. For the array and table
types, PgArray is returned. For the user-defined type, PGobject is encapsulated. Other types of data are
stored as character strings.
   for (String s : arrayValue) {
    System.out.println(s);
   }
   PgArray pgArray = (PgArray) cs.getObject(4);
   ResultSet rs = pgArray.getResultSet();
   Object[] array = (Object[]) pgArray.getArray();
   for (Object element : array) {
    System.out.println(element);
   }
   cs.close();
   conn.close();
  } catch (ClassNotFoundException | SQLException e) {
   throw new RuntimeException(e);
  }
 }
}
```

Result Verification

Data of user-defined types is correctly obtained in **Complete Example**. Below are the execution results for **Complete Example**:

```
1
demo
(10,demo10)
(11,demo111)
```

Rollback Method

N/A

13.2.3.5 Typical Issues

1. Symptom: Users encounter the following exception when calling a stored procedure, even though they have bound input parameters of a user-defined type and registered output parameters for the procedure:

   ```
   ERROR: Function public.test_proc(compfoo, public.compfoo_table, compfoo, public.compfoo_table)
   does not exist.
     ???No function matches the given name and argument types. You might need to add explicit type
   casts.
   ```

   Cause: The user-defined type bound to the input and output parameters does not match the one set in the stored procedure.

   Solution: Keep the user-defined type bound to the input and output parameters consistent with the one set in the stored procedure.

2. Symptom: Users encounter the following exception when calling a stored procedure, even though they have bound input parameters of a user-defined type and registered output parameters for the procedure:

```
java.lang.RuntimeException: org.postgresql.util.PSQLException: ?? CallableStatement ?????????????
java.sql.Types=1111 ?? 1?????????? java.sql.Types=2002?
```

   Cause: The **enableGaussArrayAndStruct** parameter is set to **true** when pgArray compatibility is in effect. However, it is impossible to maintain compatibility with both pgArray and GaussArray simultaneously.

   Solution: When pgArray compatibility is in effect, check for and delete **enableGaussArrayAndStruct=true**.

13.2.4 Batch Query

13.2.4.1 Scenario Overview

This section explains how to use JDBC for batch query.

13.2.4.1.1 Usage Scenarios

Scenario Description

Returning a large number of query results to JDBC at once may lead to JVM memory overflow. To decrease JVM memory usage, consider using batch queries, where you can specify the number of data records to be returned to JDBC each time.

Trigger Conditions

Java applications connect to the database through JDBC and query data in large batches.

Impact on Services

Developers must start and stop transactions correctly prior to database operations. In batch query scenarios, data is processed in smaller batches to effectively reduce JVM memory usage. This approach can prevent the problem of Java application memory overflow caused by clients receiving all data at once in the traditional full data transmission mode. However, it is worth noting that traversing the result set increases the number of network interactions between the database and the client (especially when each record needs to be transmitted independently). This can lead to additional performance loss, necessitating a thoughtful balance during system design.

13.2.4.1.2 Requirements and Objectives

Service Pain Points

In data-intensive service scenarios, traditional queries face the following challenges:

1. Large result sets can deplete application memory, resulting in JVM memory overflow and query failures.

2. Obtaining a large amount of data at once may saturate the network bandwidth, impacting data transmission efficiency.
3. The continuous occupation of database connections and related resources impedes overall throughput.

Service Objectives

Execute batch queries to prevent memory overflow.

13.2.4.2 Architecture Principles

Core Principles

During batch queries, JDBC leverages the cursor traversal capability of GaussDB Kernel to obtain a specific number of rows of data from the database server. This number is predetermined. The database server then returns result data to JDBC in batches based on the specified number until all query results are returned.

Solution Advantages and Disadvantages

1. During batch queries, the database returns a specific number of rows of data each time. This helps prevent memory overflow in Java applications when dealing with large result sets.
2. Batch queries involve multiple interactions with the database network, which can lead to a certain level of performance loss.

13.2.4.3 Preparations

- JDK version: 1.7 or later.
- Database environment: GaussDB 503.0 or later.
- JDBC driver environment:

  Refer to "Application Development Guide > Development Based on JDBC > Development Procedure > Obtaining the JAR Package of the Driver and Configuring the JDK Environment" in *Developer Guide*.

  Data: Create a test table and insert test data, as follows:

```
gaussdb=# CREATE TABLE tab_test(id int,context varchar(1000),PRIMARY KEY(id));
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "tab_test_pkey" for table "tab_test"
CREATE TABLE
gaussdb=# INSERT INTO tab_test SELECT generate_series(1,5),repeat('GaussDB Test', 50);
INSERT 0 5
```

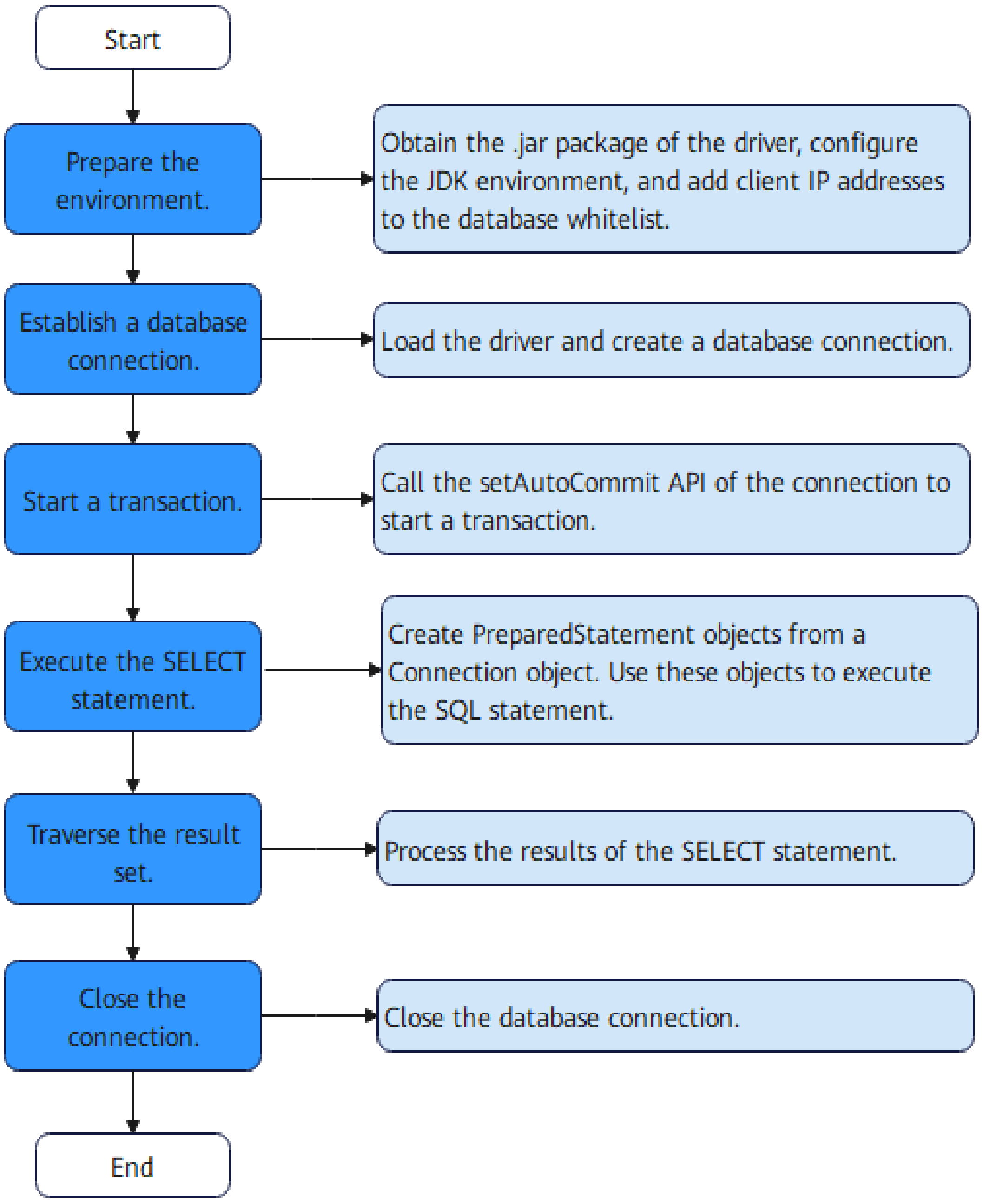
13.2.4.4 Procedure

13.2.4.4.1 Process Overview

Figure 13-10 shows the process of executing a batch query with JDBC.

This process includes preparing the environment, establishing a database connection, starting a transaction, executing SELECT statements, traversing the result set, and closing the connection.

Figure 13-10 Process of executing a batch query with JDBC


Start
Prepare the environment.
Obtain the .jar package of the driver, configure the JDK environment, and add client IP addresses to the database whitelist.
Establish a database connection.
Load the driver and create a database connection.
Start a transaction.
Call the setAutoCommit API of the connection to start a transaction.
Execute the SELECT statement.
Create PreparedStatement objects from a Connection object. Use these objects to execute the SQL statement.
Traverse the result set.
Process the results of the SELECT statement.
Close the connection.
Close the database connection.
End

13.2.4.4.2 Detailed Procedure

Step 1 Create a Connection object to connect to the database.

Here are suggestions for commonly used parameters in the connection string. For more detailed settings, refer to "Application Development Guide > Development Based on JDBC > Development Procedure > Connecting to a Database > Connection Parameter Reference" in *Developer Guide*.

- **connectTimeout**: timeout interval (in seconds) for connecting to the server's OS. If the time taken for JDBC to establish a TCP connection with the database exceeds this interval, the connection will be closed. It is advisable to set this parameter based on network conditions. The default value is **0**, whereas the recommended value is **2**.

- **socketTimeout**: timeout interval (in seconds) for socket reads. If the time taken to read data streams from the server exceeds this interval, the connection will be closed. Not setting this parameter may lead to prolonged waiting times for the client in the event of abnormal database processes. It is advisable to set this parameter based only the acceptable SQL execution time for services. The default value is **0**, with no specific recommended value provided.
- **connectionExtraInfo**: specifies whether the driver reports its deployment path, process owner, and URL connection configurations to the database. The default value is **false**, whereas the recommended value is **true**.
- **logger**: specifies a third-party log framework as needed by your application. It is advisable to choose one that incorporates slf4j APIs. These APIs can record JDBC logs to facilitate exception locating. The recommended value is **Slf4JLogger** when a third-party log framework is needed.

```
String url = "jdbc:gaussdb://$ip:$port/database?
connectTimeout=xx&socketTimeout=xx&connectionExtraInfo=true&logger=Slf4JLogger&autoBalance=t
rue"
Connection conn = DriverManager.getConnection("url",userName,password);
```

Step 2 Start a transaction.

Set **AutoCommit** to **false** so that JDBC will deliver "BEGIN" to proactively start a transaction prior to executing a query from the database.

```
conn.setAutoCommit(false);
```

Step 3 Create PreparedStatement objects and specify the number of rows to be returned by the database each time.

Use the setFetchSize method to specify this number at the statement level. If **fetchsize** has been set in the connection string, it will be overridden by the setFetchSize method.

```
String selectSql = "select * from tab_test";
PreparedStatement preparedStatement = conn.prepareStatement(selectSql);
preparedStatement.setFetchSize(20);
```

Step 4 Execute a query to obtain a result set.

```
ResultSet resultSet = preparedStatement.executeQuery();
```

Step 5 Process the result set and check data in the first column of the table.

```
while (resultSet.next()) {
    int id = resultSet.getInt(1);
    System.out.println("row:" + resultSet.getRow() + ",id :" + id);
}
```

Step 6 Obtain metadata in the result set, including the column count and types.

Obtain metadata from the **resultSet** returned by executeQuery.

```
ResultSetMetaData metaData = resultSet.getMetaData();
System.out.println("Result column: " + metaData.getColumnCount());
System.out.println("Type ID: " + metaData.getColumnType(1));
System.out.println("Type name: " + metaData.getColumnTypeName(1));
System.out.println("Column name: " + metaData.getColumnName(1));
```

Step 7 Close resources.

Use try-with-resources to automatically close any open file resources.

```
try (Connection conn = getConnection(); PreparedStatement preparedStatement =
conn.prepareStatement(selectSql))
```

Step 8 (Optional) Handle exceptions if any.

If your program encounters any exception during runtime, utilize the try-catch module to handle them and add the necessary exception handling logic for your services.

```
try {
// Service code
} catch (Exception e) {
// Exception handling logic
}
```

----End

13.2.4.4.3 Complete Example

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

public class BatchQuery {
    public static Connection getConnection() throws ClassNotFoundException, SQLException {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        // Specify the source URL of the database. (Adjust $ip, $port, and database based on the actual
services.)
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";
        // Obtain the username and password from the environment variables.
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        Class.forName(driver);
        return DriverManager.getConnection(sourceURL, userName, password);
    }

    public static void main(String[] args) {
        String selectSql = "select * from tab_test";
        try (Connection conn = getConnection(); PreparedStatement preparedStatement =
conn.prepareStatement(selectSql)) {
            conn.setAutoCommit(false);
            preparedStatement.setFetchSize(3);
            try (ResultSet resultSet = preparedStatement.executeQuery()) {
                while (resultSet.next()) {
                    // Print only a portion of the query results.
                    int id = resultSet.getInt(1);
                    System.out.println("row:" + resultSet.getRow() + ",id :" + id);
                }
                ResultSetMetaData metaData = resultSet.getMetaData();
                System.out.println("Result column: " + metaData.getColumnCount());
                System.out.println("Type ID: " + metaData.getColumnType(1));
                System.out.println("Type name: " + metaData.getColumnTypeName(1));
                System.out.println("Column name: " + metaData.getColumnName(1));
            }
            conn.commit();
        } catch (ClassNotFoundException | Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Result Verification

Below are the execution results for **Complete Example**:

```
row:1,id :1
row:2,id :2
row:3,id :3
```

```
row:4,id :4
row:5,id :5
Result column: 2
Type ID: 4
Type name: int4
Column name: id
```

Rollback Method

To disable batch query, remove the **fetchsize** parameter from the connection string and reset **setFetchSize** to the default value of **0**.

13.2.4.5 Typical Issues

1. Symptom: Customers using the Spring framework have set the **fetchsize** parameter in the connection string, but OutOfMemoryError is reported during a batch query.

   Cause: Transactions are not started prior to the batch query. Generally, connection transactions are managed by the Spring framework. However, if transactions are not started, the database will return all data to JDBC in one go.

   Solution: Check the service code and be sure to start transactions before executing batch queries.

2. Symptom: The **fetchsize** parameter has been set in the connection string, but OutOfMemoryError is reported during a batch query.

   Cause: Other call points in the application code consume substantial memory. In this situation, JDBC may encounter OutOfMemoryError when it attempts to read only a small amount of data.

   Solution: Use JDK to check memory usage and determine whether the memory overflow is caused by table data.

3. Symptom: The table query speed is fast with gsql but slow with JDBC.

   Cause: The **fetchsize** parameter has been set to a small value in the connection string. This results in a high number of packet interactions between Kernel and the result set, leading to performance degradation.

   Solution: Before querying data, determine whether to start transactions based on the table size or call the prepareStatement.setFetchSize() method to adjust the number of rows to be returned by the database each time. If this number is set to **0**, all query results will be returned in one go.

14 Best Practices for ODBC

14.1 Best Practices for ODBC (Distributed Instances)

14.1.1 Scenario Overview

This section explains how to use the ODBC driver for batch data insertion.

14.1.1.1 Usage Scenarios

Scenario Description

Batch insertion offers a more efficient way to write multiple records into the database in a single operation. Compared to single-record insertion, batch insertion decreases interactions between the application and the database, leading to lower network latency and system resource usage, while significantly improving data write efficiency.

This section illustrates various operations using the ODBC driver, including establishing database connections, utilizing transactions, executing batch insertion, and obtaining column information in the result set.

Trigger Conditions

The **UseServerSidePrepare** and **UseBatchProtocol** parameters are enabled in the ODBC configurations (both are enabled by default). Batch binding parameters are set and batch data is initialized in the application code. Batch insertion is then executed under these settings.

Impact on Services

- Lower network interaction costs

  Combining multiple INSERT statements into one batch operation significantly reduces the number of round trips between the client and the database. This enhances the overall throughput and minimizes the impact of network congestion on performance.

- Higher data processing efficiency

  In single-record insertion, the database must parse the syntax and generate an execution plan for each SQL statement. In contrast, batch insertion requires parsing the syntax and generating the plan only once, eliminating repetitive tasks and saving CPU cycles and memory allocation time.
- Reduced system resource usage and overhead

  In single-record insertion, transaction commits or Xlog writes occur at least once. In contrast, batch insertion allows multiple records to be inserted within a single transaction, significantly reducing the frequency of transaction commits, Xlog pressure, and transaction management overhead. In addition, it decreases the total number of network packet processing, transaction management, log write, and row format conversion tasks, which in turn lowers the CPU loads and temporary memory usage of the database server. This results in more resources being available for core query and computing operations.
- Higher memory usage

  When large data sizes are involved, constructing SQL statements for batch insertion can significantly increase memory usage. This is particularly noticeable when you construct SQL statements through string concatenation, as it can lead to a sharp rise in memory consumption. Large-size batch processing may exceed the maximum SQL length limit of the database or driver, or trigger other parameter restrictions, potentially leading to errors or performance issues.

Here is a detailed comparison between batch insertion and single-record insertion.

Mode	Advantages	Disadvantages
Single-record insertion	● Its code is simple, straightforward, and easy to implement. ● If any single record fails, it can be accurately identified and handled without impacting other records. ● This mode is less demanding in terms of database and driver compatibility.	● Extensive network interactions are needed. Each INSERT operation requires connecting, parsing, and committing, leading to suboptimal performance. ● Inserting a large number of records is likely to cause a bottleneck. ● Not using transactions may result in failure to guarantee the consistency of INSERT operations.

Mode	Advantages	Disadvantages
Batch insertion	● This mode greatly reduces the number of network round trips and SQL parsing instances, leading to a notable improvement in insertion throughput. ● Multiple rows can be committed within a single transaction to guarantee atomicity.	● Its code is complex, requiring manual concatenation of placeholders and parameters. ● If a single statement encounters an error, all data will be rolled back, complicating the error recovery process. ● The number of placeholders is limited; therefore, it is essential to carefully manage the batch size.

Applicable Versions

This applies only to GaussDB V500R002C10 and later versions.

14.1.1.2 Requirements and Objectives

Service Pain Points

When dealing with large data sizes, single-record insertion generates numerous network requests and consumes substantial system resources. Moreover, the database server has to repeatedly parse similar statements, leading to a decline in service performance. Batch insertion is introduced as a solution to these issues.

Service Objectives

Use the ODBC driver to initialize the target table, and insert the required data in batches through transactions for future queries. Compared to inserting data individually through SQL statements, this approach decreases interactions with the database and alleviates the database load.

14.1.2 Architecture Principles

Core Principles

When the **UseBatchProtocol** and **UseServerSidePrepare** parameters are enabled, batch processing of ODBC can reuse the same execution plan for a prepared SQL statement. All batch data to be committed within the current transaction is carried in a single U packet. Consequently, completing the batch operation only necessitates once instance of network connection establishment and data exchange.

Solution Advantages

- Optimized network communication

Sending all batch updates at once in a single U packet significantly decreases network communication overhead when compared to sending PBE packets multiple times.

- Improved execution efficiency

  Due to the reduced number and frequency of network communication, the overall execution efficiency is significantly improved, especially for large data sizes.

- Optimized resource utilization

  Batch insertion optimizes the utilization of database server resources, cutting down on unnecessary system overhead associated with single-record insertion.

14.1.3 Preparations

- ODBC version: V500R002C10 or later.
- Database environment: GaussDB V500R002C10 or later.
- ODBC driver environment:

  Refer to "Application Development Guide > Development Based on ODBC > Development Procedure > Obtaining the Source Code Package, ODBC Packages, and Dependency Libraries" in *Developer Guide*.

- ODBC data source:

  Refer to "Configuring a Data Source in the Linux OS" or "Configuring a Data Source in the Windows OS" under "Application Development Guide > Development Based on ODBC > Development Procedure > Connecting to a Database" in *Developer Guide*.

  Taking Linux environments as an example, you are advised to set parameters in the **odbc.ini** file as follows:

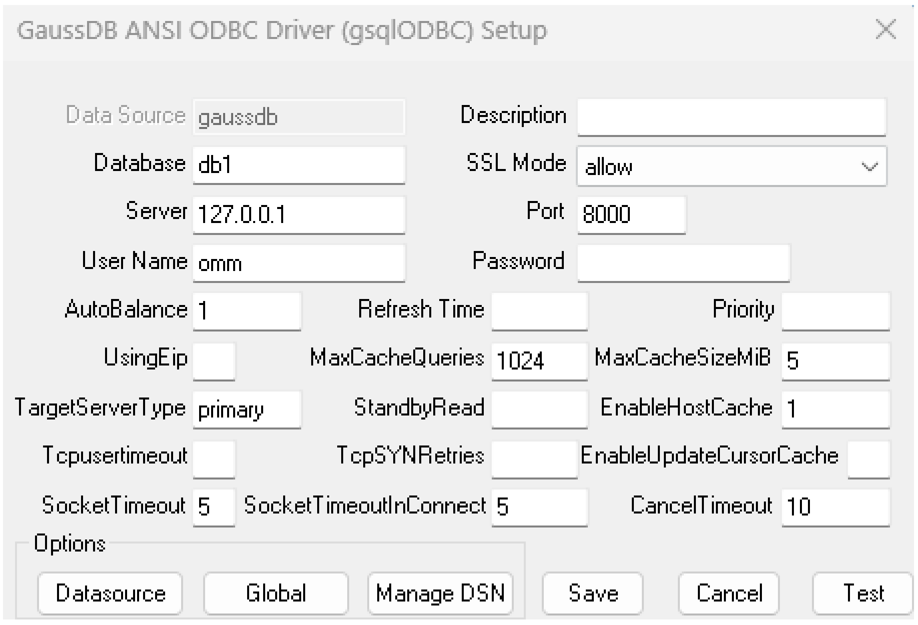
```
[gaussdb]
Driver=GaussMPP
Servername=127.0.0.1  # IP address of the database server.
Database=db1  # Database name.
Username=omm  # Name of a database user.
Password=******  # Password for the database user.
Port=8000  # Database listening port.
Sslmode=allow  # Specifies whether to enable SSL encryption. When set to allow, it means that the
database server can use SSL encryption as required, but the server's authenticity is not verified.
UseServerSidePrepare=1  # This parameter is enabled by default. If it is set to 1, the client sends
PU/PBE packets in soft parsing mode. If it is set to 0, the client sends Q packets in hard parsing mode.
UseBatchProtocol=1  # Specifies whether to enable batch query. This parameter is enabled by default.
MaxCacheQueries=1024  # Number of prepared statements cached for each connection.
MaxCacheSizeMiB=5  # Total size of prepared statements cached for each connection. This parameter
takes effect when MaxCacheQueries is greater than 0.
ConnSettings=set client_encoding=UTF8  # Client-side encoding, which must be consistent with server-
side encoding.
SocketTimeout=5  # Timeout interval for socket reads/writes after a connection is successfully
established between the client and the server.
TargetServerType=primary  # Type of the target server to connect to. A connection can be successfully
established only when the actual server type matches the value of this parameter. primary indicates
that only the primary node in a primary/standby system can be connected to.
AutoBalance=1  # Specifies whether to enable load balancing for ODBC. Load balancing is
unavailable for DR clusters if the connected database version is earlier than 506.0.
```

NOTE

- The **AutoBalance** parameter is supported in GaussDB V500R002C20 and later versions.
- The **MaxCacheQueries** and **MaxCacheSizeMiB** parameters are supported in GaussDB 503.1 and later versions.
- The **SocketTimeout** parameter is supported in GaussDB 505.2 and later versions.
- The **TargetServerType** options of **cluster-primary**, **cluster-standby**, and **cluster-mainnode** are available only from GaussDB 506.0. Other options have been supported since GaussDB 505.2.

Figure 14-1 shows the recommended configurations for the data source manager in Windows environments.

Figure 14-1 Data source manager configurations in Windows environments



GaussDB ANSI ODBC Driver (gsqlODBC) Setup

Data Source	gaussdb	Description			
Database	db1	SSL Mode	allow		
Server	127.0.0.1	Port	8000		
User Name	omm	Password			
AutoBalance	1	Refresh Time		Priority	
UsingEip		MaxCacheQueries	1024	MaxCacheSizeMiB	5
TargetServerType	primary	StandbyRead		EnableHostCache	1
Tcpusertimeout		TcpSYNRetries		EnableUpdateCursorCache	
SocketTimeout	5	SocketTimeoutInConnect	5	CancelTimeout	10

NOTE

Adjust the preceding data source configurations, including but not limited to the database server IP address, port number, and other connection parameters, to match the actual services.

14.1.4 Procedure

14.1.4.1 Process Overview

The process of batch binding and insertion includes steps such as configuring a connection, setting batch binding parameters, and executing batch insertion. Typical APIs involved include SQLSetConnectAttr, SQLSetStmtAttr, SQLPrepare, SQLBindParameter, SQLExecute, and SQLRowCount.

For details about these APIs, see "Application Development Guide > Development Based on ODBC > ODBC API Reference" in *Developer Guide*.

14.1.4.2 Detailed Procedure

Step 1 Configure a connection.

1. Set the connection timeout interval.

   To manage the timeout interval (in seconds) for clients to connect to the server, adjust the **SQL_LOGIN_TIMEOUT** parameter in the SQLSetConnectAttr function. This parameter corresponds to the libpq parameter **connect_timeout**. A default value of **0** indicates that the parameter is not in effect. You are advised to set it based on the actual network conditions.

   ```
   SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
   ```

2. Disable the autocommit option in order to use transactions for commit or rollback.

   To use transactions for commit or rollback, disable autocommit by setting **SQL_AUTOCOMMIT_OFF** in the SQLSetConnectAttr function.

   ```
   SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF, 0);
   ```

3. Establish a database connection.

   Establish a database connection through the SQLConnect function. Below is the function prototype:

   ```
   SQLRETURN  SQLConnect(SQLHDBC        ConnectionHandle,
                         SQLCHAR        *ServerName,
                         SQLSMALLINT    NameLength1,
                         SQLCHAR        *UserName,
                         SQLSMALLINT    NameLength2,
                         SQLCHAR        *Authentication,
                         SQLSMALLINT    NameLength3);
   ```

   If **ServerName** of the data source was set to **gaussdb** in **Preparations**, ODBC automatically obtains connection parameters from the **odbc.ini** file (in Linux environments) or from the data source manager (in Windows environments).

   After obtaining the data source, the function utilizes the connection handle **hdbc** to access all details about the connected data source, including program running status, transaction processing status, and error information. Subsequently, the function employs the appropriate parameters to connect to the database.

   ```
   SQLConnect(hdbc, (SQLCHAR *)"gaussdb", SQL_NTS, (SQLCHAR *)"", 0, (SQLCHAR *)"", 0);
   ```

Step 2 Set batch binding parameters.

1. Set batch binding parameters.

   Set the total number of rows in the batch binding parameter array. The *batchCount* variable indicates the total number of rows to be inserted in batches.

   ```
   SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMSET_SIZE, (SQLPOINTER)batchCount, sizeof(batchCount));
   ```

   Set the number of processed rows. The *processRows* variable indicates the number of rows that have been inserted in batches.

   ```
   SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMS_PROCESSED_PTR, (SQLPOINTER)&processRows,
   sizeof(processRows));
   ```

2. Prepare statements and bind parameters.

   Use the SQLPrepare function to prepare SQL statements. The *sql* variable holds the SQL statement string. *SQL_NTS* indicates that the string ends with a null character. Use the SQLBindParameter function to bind parameters to the prepared statements. *ids* and *cols* correspond to the two arrays in the **id** column (INT type) and **col** column (VARCHAR type).

```
SQLPrepare(hstmt, (SQLCHAR *)sql, SQL_NTS);
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, sizeof(ids[0]), 0,
&(ids[0]), 0, bufLenIds);
SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, 50, 50, cols, 50,
bufLenCols);
```

Step 3 Execute batch insertion.

1. Execute batch insertion.

   Use the SQLExecute function to execute the prepared SQL statements for batch insertion. The return value of **retcode** indicates the result of the insertion.

   ```
   retcode = SQLExecute(hstmt);
   ```

2. Manually commit or roll back the transaction.

   If **retcode** returns **SQL_SUCCESS** or **SQL_SUCCESS_WITH_INFO**, the insertion was successful. In this case, call the SQLEndTran function to commit the transaction. However, if **retcode** returns any other value, the insertion has failed. In this case, call the SQLEndTran function to roll back the transaction.

   ```
   SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT); // Commit the transaction.
   SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK); // Roll back the transaction.
   ```

3. Obtain the number of rows processed in batches.

   Use the SQLRowCount function to obtain the number of rows actually inserted and store the number in the *rowsCount* variable.

   ```
   SQLRowCount(hstmt, &rowsCount);
   ```

----End

14.1.4.3 Complete Example

```
/**********************************************************************
 * Enable UseBatchProtocol in the data source and set the database parameter support_batch_bind to on.
 * CHECK_ERROR and CHECK_ERROR_VOID are used to check for and print error information.
 * This example interactively obtains the DSN and the data size for batch binding, and inserts the final data
into test_odbc_batch_insert.
 **********************************************************************/
#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <sql.h>
#include <sqlext.h>
#include <string.h>
#define CHECK_ERROR(e, s, t, h)                                    \
    ({                                                             \
        if (e != SQL_SUCCESS && e != SQL_SUCCESS_WITH_INFO) {      \
            fprintf(stderr, "FAILED:\t");                          \
            print_diag(s, h, t);                                   \
            goto exit;                                             \
        }                                                          \
    })
#define CHECK_ERROR_VOID(e, s, t, h)                               \
    ({                                                             \
        if (e != SQL_SUCCESS && e != SQL_SUCCESS_WITH_INFO) {      \
            fprintf(stderr, "FAILED:\t");                          \
            print_diag(s, h, t);                                   \
        }                                                          \
    })
#define BATCH_SIZE 100  // Data size to be bound in batches.
// Print error information.
void print_diag(char *msg, SQLSMALLINT htype, SQLHANDLE handle);
// Execute SQL statements.
```

```
void Exec(SQLHDBC hdbc, SQLCHAR *sql)
{
    SQLRETURN retcode;                  // Returned error code.
    SQLHSTMT hstmt = SQL_NULL_HSTMT;  // Statement handle.
    SQLCHAR loginfo[2048];
    // Allocate a statement handle.
    retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLAllocHandle(SQL_HANDLE_STMT) failed");
        return;
    }
    // Prepare statements.
    retcode = SQLPrepare(hstmt, (SQLCHAR *)sql, SQL_NTS);
    sprintf((char *)loginfo, "SQLPrepare log: %s", (char *)sql);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLPrepare(hstmt, (SQLCHAR*) sql, SQL_NTS) failed");
        return;
    }
    // Execute statements.
    retcode = SQLExecute(hstmt);
    sprintf((char *)loginfo, "SQLExecute stmt log: %s", (char *)sql);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLExecute(hstmt) failed");
        return;
    }
    // Release the handle.
    retcode = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
    sprintf((char *)loginfo, "SQLFreeHandle stmt log: %s", (char *)sql);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLFreeHandle(SQL_HANDLE_STMT, hstmt) failed");
        return;
    }
}
int main()
{
    SQLHENV henv = SQL_NULL_HENV;
    SQLHDBC hdbc = SQL_NULL_HDBC;
    SQLLEN rowsCount = 0;
    int i = 0;
    SQLRETURN retcode;
    SQLCHAR dsn[1024] = {'\0'};
    SQLCHAR loginfo[2048];
    // Allocate an environment handle.
    retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLAllocHandle failed");
        goto exit;
    }
    CHECK_ERROR(retcode, "SQLAllocHandle henv", henv, SQL_HANDLE_ENV);
    // Set the ODBC version.
    retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER *)SQL_OV_ODBC3, 0);
    CHECK_ERROR(retcode, "SQLSetEnvAttr", henv, SQL_HANDLE_ENV);
    // Allocate connections.
    retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    CHECK_ERROR(retcode, "SQLAllocHandle hdbc", hdbc, SQL_HANDLE_DBC);
    // Set the login timeout.
    retcode = SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
    CHECK_ERROR(retcode, "SQLSetConnectAttr SQL_LOGIN_TIMEOUT", hdbc, SQL_HANDLE_DBC);
    // Disable the autocommit option in order to use transactions for commit.
    retcode = SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF, 0);
    CHECK_ERROR(retcode, "SQLSetConnectAttr SQL_ATTR_AUTOCOMMIT", hdbc, SQL_HANDLE_DBC);
    // Establish a database connection.
    retcode = SQLConnect(hdbc, (SQLCHAR *)"gaussdb", SQL_NTS, (SQLCHAR *)"", 0, (SQLCHAR *)"", 0);
    CHECK_ERROR(retcode, "SQLSetConnectAttr SQL_ATTR_AUTOCOMMIT", hdbc, SQL_HANDLE_DBC);
    printf("SQLConnect success\n");
    // Initialize table information.
    Exec(hdbc, "DROP TABLE IF EXISTS test_odbc_batch_insert");
    Exec(hdbc, "CREATE TABLE test_odbc_batch_insert (id INT PRIMARY KEY, col VARCHAR2(50))");
    // Commit the transaction in segments for other SQL operations.
    retcode = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
    CHECK_ERROR(retcode, "SQLEndTran", hdbc, SQL_HANDLE_DBC);
    // Construct the data to be inserted in batches based on the user-specified data size.
    {
        SQLRETURN retcode;
        SQLHSTMT hstmt = SQL_NULL_HSTMT;
        SQLCHAR *sql = NULL;
        SQLINTEGER *ids = NULL;
        SQLCHAR *cols = NULL;
        SQLLEN *bufLenIds = NULL;
        SQLLEN *bufLenCols = NULL;
        SQLUSMALLINT *operptr = NULL;
        SQLUSMALLINT *statusptr = NULL;
        SQLULEN process = 0;
        // Construct fields by column.
        ids = (SQLINTEGER *)malloc(sizeof(ids[0]) * BATCH_SIZE);
        cols = (SQLCHAR *)malloc(sizeof(cols[0]) * BATCH_SIZE * 50);
        // Memory length for each row of data with each field.
        bufLenIds = (SQLLEN *)malloc(sizeof(bufLenIds[0]) * BATCH_SIZE);
        bufLenCols = (SQLLEN *)malloc(sizeof(bufLenCols[0]) * BATCH_SIZE);
        if (NULL == ids || NULL == cols || NULL == bufLenCols || NULL == bufLenIds) {
            fprintf(stderr, "FAILED:\tmalloc data memory failed\n");
            goto exit;
        }
        // Assign values to data.
        for (i = 0; i < BATCH_SIZE; i++) {
            ids[i] = i;
            sprintf(cols + 50 * i, "column test value %d", i);
            bufLenIds[i] = sizeof(ids[i]);
            bufLenCols[i] = strlen(cols + 50 * i);
        }
        // Allocate a statement handle.
        retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
        CHECK_ERROR(retcode, "SQLSetConnectAttr SQL_ATTR_AUTOCOMMIT", hstmt, SQL_HANDLE_STMT);
        // Prepare statements.
        sql = (SQLCHAR *)"INSERT INTO test_odbc_batch_insert VALUES(?, ?)";
        retcode = SQLPrepare(hstmt, (SQLCHAR *)sql, SQL_NTS);
        CHECK_ERROR(retcode, "SQLPrepare", hstmt, SQL_HANDLE_STMT);
        // Bind parameters.
        retcode = SQLBindParameter(
            hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, sizeof(ids[0]), 0, &(ids[0]), 0,
bufLenIds);
        CHECK_ERROR(retcode, "SQLBindParameter 1", hstmt, SQL_HANDLE_STMT);
        retcode = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, 50, 50, cols, 50,
bufLenCols);
        CHECK_ERROR(retcode, "SQLBindParameter 2", hstmt, SQL_HANDLE_STMT);
        // Set the total number of rows in the parameter array.
        retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMSET_SIZE, (SQLPOINTER)BATCH_SIZE,
sizeof(BATCH_SIZE));
        CHECK_ERROR(retcode, "SQLSetStmtAttr", hstmt, SQL_HANDLE_STMT);
        // Set the number of processed rows.
        retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMS_PROCESSED_PTR, (SQLPOINTER)&process,
sizeof(process));
        CHECK_ERROR(retcode, "SQLSetStmtAttr SQL_ATTR_PARAMS_PROCESSED_PTR", hstmt,
SQL_HANDLE_STMT);
        // Execute batch insertion.
        retcode = SQLExecute(hstmt);
        // Manually commit the transaction.
        if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
            retcode = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
            CHECK_ERROR(retcode, "SQLEndTran", hdbc, SQL_HANDLE_DBC);
        }
        // On failure, roll back the transaction.
        else {
            CHECK_ERROR_VOID(retcode, "SQLExecute", hstmt, SQL_HANDLE_STMT);
            retcode = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK);
            printf("Transaction rollback\n");
            CHECK_ERROR(retcode, "SQLEndTran", hdbc, SQL_HANDLE_DBC);
        }
        // Obtain the number of rows processed in batches.
        SQLRowCount(hstmt, &rowsCount);
        sprintf((char *)loginfo, "SQLRowCount : %ld", rowsCount);
        puts(loginfo);
        // Check whether the number of inserted rows matches the number of processed rows.
        if (rowsCount != process) {
            sprintf(loginfo, "process(%d) != rowsCount(%d)", process, rowsCount);
            puts(loginfo);
        } else {
            sprintf(loginfo, "process(%d) == rowsCount(%d)", process, rowsCount);
        }
        retcode = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
        CHECK_ERROR(retcode, "SQLFreeHandle", hstmt, SQL_HANDLE_STMT);
    }
exit:
    (void)printf("Complete.\n");
    // Close the connection.
    if (hdbc != SQL_NULL_HDBC) {
        SQLDisconnect(hdbc);
        SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
    }
    // Release the environment handle.
    if (henv != SQL_NULL_HENV)
        SQLFreeHandle(SQL_HANDLE_ENV, henv);
    return 0;
}
void print_diag(char *msg, SQLSMALLINT htype, SQLHANDLE handle)
{
    char sqlstate[32];
    char message[1000];
    SQLINTEGER nativeerror;
    SQLSMALLINT textlen;
    SQLRETURN ret;
    SQLSMALLINT recno = 0;
    if (msg)
        printf("%s\n", msg);
    do {
        recno++;
        // Obtain diagnostic information.
        ret = SQLGetDiagRec(
            htype, handle, recno, (SQLCHAR *)sqlstate, &nativeerror, (SQLCHAR *)message, sizeof(message),
&textlen);
        if (ret == SQL_INVALID_HANDLE)
            printf("Invalid handle\n");
        else if (SQL_SUCCEEDED(ret))
            printf("%s=%s\n", sqlstate, message);
    } while (ret == SQL_SUCCESS);
    if (ret == SQL_NO_DATA && recno == 1)
        printf("No error information\n");
}
```

Result Verification

After successfully connecting to the database, ODBC inserts 100 data records in batches. Below are the expected results for **Complete Example**:

```
SQLConnect success
SQLRowCount : 100
Complete.
```

Rollback Method

If any abnormal operations occur during the transaction, call the SQLEndTran API to roll them back, as follows:

```
SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK);
```

14.1.5 Typical Issues

1. Question: After setting **ignoreCount** (which specifies the data size to exclude), can I select which specific data to exclude from the database?

   Answer: Sure. In **Complete Example**, you can adjust the operator pointer **operptr** to specify which data should be inserted and which should be excluded. Specifically, when **operptr** is set to **SQL_PARAM_IGNORE**, the data will be excluded; when **operptr** is set to **SQL_PARAM_PROCEED**, the data will be inserted.

2. Question: If batch insertion fails, can the transaction be committed for successfully inserted data and rolled back only for data that encounters an error?

   Answer: No, but in the event of insertion failure, the transaction must be rolled back for all scheduled data. However, to address the issue, one possible solution is to define smaller data batches in the application code to commit the data in batches for better exception capture.

3. Question: I have set **UseServerSidePrepare** to **0** and inserted *n* records in batches, but why does the SQLRowCount API return **1** as the number of inserted records?

   Answer: This result is as expected. Let's revisit the explanation of **UseServerSidePrepare** provided in **Preparations** for the **odbc.ini** file for Linux environments: When **UseServerSidePrepare** is set to **1**, PU/PBE packets will be sent for soft parsing. In this case, **SQLRowCount** returns the number of records updated last time. Conversely, when **UseServerSidePrepare** is set to **0**, Q packets will be sent for hard parsing, where **SQLRowCount** still returns the number of records updated last time. However, because only one SQL statement is carried in each Q packet, the number of inserted records is 1.

14.2 Best Practices for ODBC (Centralized Instances)

14.2.1 Scenario Overview

This section explains how to use the ODBC driver for batch data insertion.

14.2.1.1 Usage Scenarios

Scenario Description

Batch insertion offers a more efficient way to write multiple records into the database in a single operation. Compared to single-record insertion, batch insertion decreases interactions between the application and the database, leading to lower network latency and system resource usage, while significantly improving data write efficiency.

This section illustrates various operations using the ODBC driver, including establishing database connections, utilizing transactions, executing batch insertion, and obtaining column information in the result set.

Trigger Conditions

The **UseServerSidePrepare** and **UseBatchProtocol** parameters are enabled in the ODBC configurations (both are enabled by default). Batch binding parameters are set and batch data is initialized in the application code. Batch insertion is then executed under these settings.

Impact on Services

- Lower network interaction costs

  Combining multiple INSERT statements into one batch operation significantly reduces the number of round trips between the client and the database. This enhances the overall throughput and minimizes the impact of network congestion on performance.
- Higher data processing efficiency

  In single-record insertion, the database must parse the syntax and generate an execution plan for each SQL statement. In contrast, batch insertion requires parsing the syntax and generating the plan only once, eliminating repetitive tasks and saving CPU cycles and memory allocation time.
- Reduced system resource usage and overhead

  In single-record insertion, transaction commits or Xlog writes occur at least once. In contrast, batch insertion allows multiple records to be inserted within a single transaction, significantly reducing the frequency of transaction commits, Xlog pressure, and transaction management overhead. In addition, it decreases the total number of network packet processing, transaction management, log write, and row format conversion tasks, which in turn lowers the CPU loads and temporary memory usage of the database server. This results in more resources being available for core query and computing operations.
- Higher memory usage

  When large data sizes are involved, constructing SQL statements for batch insertion can significantly increase memory usage. This is particularly noticeable when you construct SQL statements through string concatenation, as it can lead to a sharp rise in memory consumption. Large-size batch processing may exceed the maximum SQL length limit of the database or driver, or trigger other parameter restrictions, potentially leading to errors or performance issues.

Here is a detailed comparison between batch insertion and single-record insertion.

Mode	Advantages	Disadvantages
Single-record insertion	● Its code is simple, straightforward, and easy to implement. ● If any single record fails, it can be accurately identified and handled without impacting other records. ● This mode is less demanding in terms of database and driver compatibility.	● Extensive network interactions are needed. Each INSERT operation requires connecting, parsing, and committing, leading to suboptimal performance. ● Inserting a large number of records is likely to cause a bottleneck. ● Not using transactions may result in failure to guarantee the consistency of INSERT operations.
Batch insertion	● This mode greatly reduces the number of network round trips and SQL parsing instances, leading to a notable improvement in insertion throughput. ● Multiple rows can be committed within a single transaction to guarantee atomicity.	● Its code is complex, requiring manual concatenation of placeholders and parameters. ● If a single statement encounters an error, all data will be rolled back, complicating the error recovery process. ● The number of placeholders is limited; therefore, it is essential to carefully manage the batch size.

Applicable Versions

This applies only to GaussDB V500R002C10 and later versions.

14.2.1.2 Requirements and Objectives

Service Pain Points

When dealing with large data sizes, single-record insertion generates numerous network requests and consumes substantial system resources. Moreover, the database server has to repeatedly parse similar statements, leading to a decline in service performance. Batch insertion is introduced as a solution to these issues.

Service Objectives

Use the ODBC driver to initialize the target table, and insert the required data in batches through transactions for future queries. Compared to inserting data individually through SQL statements, this approach decreases interactions with the database and alleviates the database load.

14.2.2 Architecture Principles

Core Principles

When the **UseBatchProtocol** and **UseServerSidePrepare** parameters are enabled, batch processing of ODBC can reuse the same execution plan for a prepared SQL statement. All batch data to be committed within the current transaction is carried in a single U packet. Consequently, completing the batch operation only necessitates once instance of network connection establishment and data exchange.

Solution Advantages

- Optimized network communication

  Sending all batch updates at once in a single U packet significantly decreases network communication overhead when compared to sending PBE packets multiple times.
- Improved execution efficiency

  Due to the reduced number and frequency of network communication, the overall execution efficiency is significantly improved, especially for large data sizes.
- Optimized resource utilization

  Batch insertion optimizes the utilization of database server resources, cutting down on unnecessary system overhead associated with single-record insertion.

14.2.3 Preparations

- ODBC version: V500R002C10 or later.
- Database environment: GaussDB V500R002C10 or later.
- ODBC driver environment:

  Refer to "Application Development Guide > Development Based on ODBC > Development Procedure > Obtaining the Source Code Package, ODBC Packages, and Dependency Libraries" in *Developer Guide*.
- ODBC data source:

  Refer to "Configuring a Data Source in the Linux OS" or "Configuring a Data Source in the Windows OS" under "Application Development Guide > Development Based on ODBC > Development Procedure > Connecting to a Database" in *Developer Guide*.

  Taking Linux environments as an example, you are advised to set parameters in the **odbc.ini** file as follows:

```
[gaussdb]
Driver=GaussMPP
Servername=127.0.0.1 # IP address of the database server.
Database=db1 # Database name.
Username=omm # Name of a database user.
Password=****** # Password for the database user.
Port=8000 # Database listening port.
Sslmode=allow # Specifies whether to enable SSL encryption. When set to allow, it means that the database server can use SSL encryption as required, but the server's authenticity is not verified.
UseServerSidePrepare=1 # This parameter is enabled by default. If it is set to 1, the client sends PU/PBE packets in soft parsing mode. If it is set to 0, the client sends Q packets in hard parsing mode.
```

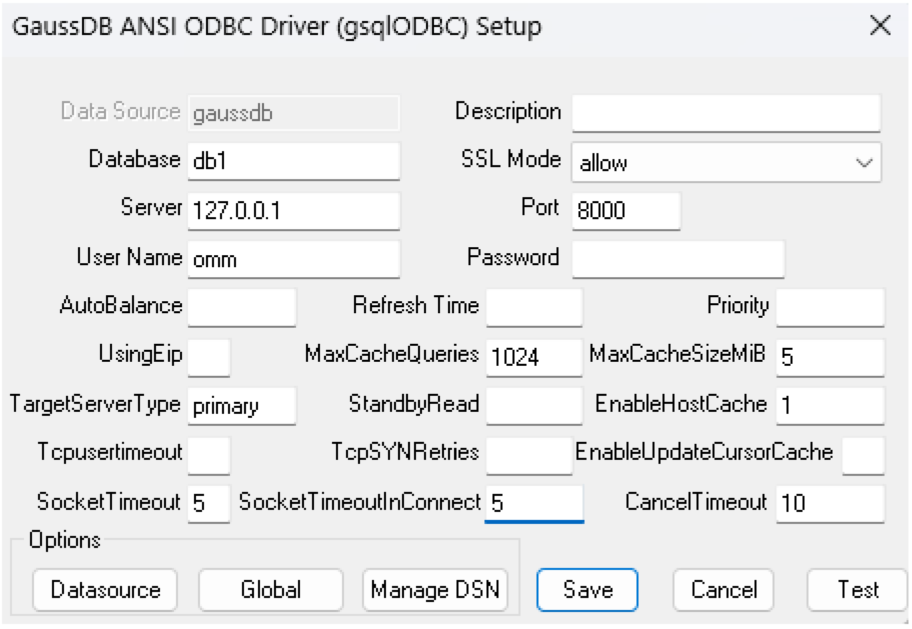
```
UseBatchProtocol=1 # Specifies whether to enable batch query. This parameter is enabled by default.
MaxCacheQueries=1024 # Number of prepared statements cached for each connection.
MaxCacheSizeMiB=5 # Total size of prepared statements cached for each connection. This parameter takes effect when MaxCacheQueries is greater than 0.
ConnSettings=set client_encoding=UTF8 # Client-side encoding, which must be consistent with server-side encoding.
SocketTimeout=5 # Timeout interval for socket reads/writes after a connection is successfully established between the client and the server.
TargetServerType=primary # Type of the target server to connect to. A connection can be successfully established only when the actual server type matches the value of this parameter. primary indicates that only the primary node in a primary/standby system can be connected to.
```

NOTE

- The **MaxCacheQueries** and **MaxCacheSizeMiB** parameters are supported in GaussDB 503.1 and later versions.
- The **SocketTimeout** parameter is supported in GaussDB 505.2 and later versions.
- The **TargetServerType** options of **cluster-primary**, **cluster-standby**, and **cluster-mainnode** are available only from GaussDB 506.0. Other options have been supported since GaussDB 505.2.

Figure 14-2 shows the recommended configurations for the data source manager in Windows environments.

Figure 14-2 Data source manager configurations in Windows environments


GaussDB ANSI ODBC Driver (gsqlODBC) Setup

Data Source	gaussdb	Description			
Database	db1	SSL Mode	allow		
Server	127.0.0.1	Port	8000		
User Name	omm	Password			
AutoBalance		Refresh Time		Priority	
UsingEip		MaxCacheQueries	1024	MaxCacheSizeMiB	5
TargetServerType	primary	StandbyRead		EnableHostCache	1
Tcpusertimeout		TcpSYNRetries		EnableUpdateCursorCache	
SocketTimeout	5	SocketTimeoutInConnect	5	CancelTimeout	10

NOTE

Adjust the preceding data source configurations, including but not limited to the database server IP address, port number, and other connection parameters, to match the actual services.

14.2.4 Procedure

14.2.4.1 Process Overview

The process of batch binding and insertion includes steps such as configuring a connection, setting batch binding parameters, and executing batch insertion.

Typical APIs involved include SQLSetConnectAttr, SQLSetStmtAttr, SQLPrepare, SQLBindParameter, SQLExecute, and SQLRowCount.

For details about these APIs, see "Application Development Guide > Development Based on ODBC > ODBC API Reference" in *Developer Guide*.

14.2.4.2 Detailed Procedure

Step 1 Configure a connection.

1. Set the connection timeout interval.

   To manage the timeout interval (in seconds) for clients to connect to the server, adjust the **SQL_LOGIN_TIMEOUT** parameter in the SQLSetConnectAttr function. This parameter corresponds to the libpq parameter **connect_timeout**. A default value of **0** indicates that the parameter is not in effect. You are advised to set it based on the actual network conditions.

   ```
   SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
   ```

2. Disable the autocommit option in order to use transactions for commit or rollback.

   To use transactions for commit or rollback, disable autocommit by setting **SQL_AUTOCOMMIT_OFF** in the SQLSetConnectAttr function.

   ```
   SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF, 0);
   ```

3. Establish a database connection.

   Establish a database connection through the SQLConnect function. Below is the function prototype:

   ```
   SQLRETURN  SQLConnect(SQLHDBC        ConnectionHandle,
                         SQLCHAR        *ServerName,
                         SQLSMALLINT    NameLength1,
                         SQLCHAR        *UserName,
                         SQLSMALLINT    NameLength2,
                         SQLCHAR        *Authentication,
                         SQLSMALLINT    NameLength3);
   ```

   If **ServerName** of the data source was set to **gaussdb** in **Preparations**, ODBC automatically obtains connection parameters from the **odbc.ini** file (in Linux environments) or from the data source manager (in Windows environments).

   After obtaining the data source, the function utilizes the connection handle **hdbc** to access all details about the connected data source, including program running status, transaction processing status, and error information. Subsequently, the function employs the appropriate parameters to connect to the database.

   ```
   SQLConnect(hdbc, (SQLCHAR *)"gaussdb", SQL_NTS, (SQLCHAR *)"", 0, (SQLCHAR *)"", 0);
   ```

Step 2 Set batch binding parameters.

1. Set batch binding parameters.

   Set the total number of rows in the batch binding parameter array. The *batchCount* variable indicates the total number of rows to be inserted in batches.

   ```
   SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMSET_SIZE, (SQLPOINTER)batchCount, sizeof(batchCount));
   ```

   Set the number of processed rows. The *processRows* variable indicates the number of rows that have been inserted in batches.

   ```
   SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMS_PROCESSED_PTR, (SQLPOINTER)&processRows,
   sizeof(processRows));
   ```

2. Prepare statements and bind parameters.

   Use the SQLPrepare function to prepare SQL statements. Use the SQLBindParameter function to bind parameters to the prepared statements. The *sql* variable holds the SQL statement string. *SQL_NTS* indicates that the string ends with a null character. *ids* and *cols* correspond to the two arrays in the **id** column (INT type) and **col** column (VARCHAR type).

   ```
   SQLPrepare(hstmt, (SQLCHAR *)sql, SQL_NTS);
   SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, sizeof(ids[0]), 0,
   &(ids[0]), 0, bufLenIds);
   SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, 50, 50, cols, 50,
   bufLenCols);
   ```

Step 3 Execute batch insertion.

1. Execute batch insertion.

   Use the SQLExecute function to execute the prepared SQL statements for batch insertion. The return value of **retcode** indicates the result of the insertion.

   ```
   retcode = SQLExecute(hstmt);
   ```

2. Manually commit or roll back the transaction.

   If **retcode** returns **SQL_SUCCESS** or **SQL_SUCCESS_WITH_INFO**, the insertion was successful. In this case, call the SQLEndTran function to commit the transaction. However, if **retcode** returns any other value, the insertion has failed. In this case, call the SQLEndTran function to roll back the transaction.

   ```
   SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT); // Commit the transaction.
   SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK); // Roll back the transaction.
   ```

3. Obtain the number of rows processed in batches.

   Use the SQLRowCount function to obtain the number of rows actually inserted and store the number in the *rowsCount* variable.

   ```
   SQLRowCount(hstmt, &rowsCount);
   ```

----End

14.2.4.3 Complete Example

```
/**********************************************************************
 * Enable UseBatchProtocol in the data source and set the database parameter support_batch_bind to on.
 * CHECK_ERROR and CHECK_ERROR_VOID are used to check for and print error information.
 * This example interactively obtains the DSN and the data size for batch binding, and inserts the final data
into test_odbc_batch_insert.
 **********************************************************************/
#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <sql.h>
#include <sqlext.h>
#include <string.h>
#define CHECK_ERROR(e, s, t, h)                                 \
    ({                                                          \
        if (e != SQL_SUCCESS && e != SQL_SUCCESS_WITH_INFO) {   \
            fprintf(stderr, "FAILED:\t");                       \
            print_diag(s, h, t);                                \
            goto exit;                                          \
        }                                                       \
    })
#define CHECK_ERROR_VOID(e, s, t, h)                            \
    ({                                                          \
        if (e != SQL_SUCCESS && e != SQL_SUCCESS_WITH_INFO) {   \
```

```
            fprintf(stderr, "FAILED:\t");                       \
            print_diag(s, h, t);                                \
        }                                                       \
    })
#define BATCH_SIZE 100  // Data size to be bound in batches.
// Print error information.
void print_diag(char *msg, SQLSMALLINT htype, SQLHANDLE handle);
// Execute SQL statements.
void Exec(SQLHDBC hdbc, SQLCHAR *sql)
{
    SQLRETURN retcode;                 // Returned error code.
    SQLHSTMT hstmt = SQL_NULL_HSTMT;  // Statement handle.
    SQLCHAR loginfo[2048];
    // Allocate a statement handle.
    retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLAllocHandle(SQL_HANDLE_STMT) failed");
        return;
    }
    // Prepare statements.
    retcode = SQLPrepare(hstmt, (SQLCHAR *)sql, SQL_NTS);
    sprintf((char *)loginfo, "SQLPrepare log: %s", (char *)sql);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLPrepare(hstmt, (SQLCHAR*) sql, SQL_NTS) failed");
        return;
    }
    // Execute statements.
    retcode = SQLExecute(hstmt);
    sprintf((char *)loginfo, "SQLExecute stmt log: %s", (char *)sql);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLExecute(hstmt) failed");
        return;
    }
    // Release the handle.
    retcode = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
    sprintf((char *)loginfo, "SQLFreeHandle stmt log: %s", (char *)sql);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLFreeHandle(SQL_HANDLE_STMT, hstmt) failed");
        return;
    }
}
int main()
{
    SQLHENV henv = SQL_NULL_HENV;
    SQLHDBC hdbc = SQL_NULL_HDBC;
    SQLLEN rowsCount = 0;
    int i = 0;
    SQLRETURN retcode;
    SQLCHAR dsn[1024] = {'\0'};
    SQLCHAR loginfo[2048];
    // Allocate an environment handle.
    retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLAllocHandle failed");
        goto exit;
    }
    CHECK_ERROR(retcode, "SQLAllocHandle henv", henv, SQL_HANDLE_ENV);
    // Set the ODBC version.
    retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER *)SQL_OV_ODBC3, 0);
    CHECK_ERROR(retcode, "SQLSetEnvAttr", henv, SQL_HANDLE_ENV);
    // Allocate connections.
    retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    CHECK_ERROR(retcode, "SQLAllocHandle hdbc", hdbc, SQL_HANDLE_DBC);
    // Set the login timeout.
    retcode = SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
    CHECK_ERROR(retcode, "SQLSetConnectAttr SQL_LOGIN_TIMEOUT", hdbc, SQL_HANDLE_DBC);
    // Disable the autocommit option in order to use transactions for commit.
    retcode = SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF, 0);
    CHECK_ERROR(retcode, "SQLSetConnectAttr SQL_ATTR_AUTOCOMMIT", hdbc, SQL_HANDLE_DBC);
    // Establish a database connection.
    retcode = SQLConnect(hdbc, (SQLCHAR *)"gaussdb", SQL_NTS, (SQLCHAR *)"", 0, (SQLCHAR *)"", 0);
    CHECK_ERROR(retcode, "SQLSetConnectAttr SQL_ATTR_AUTOCOMMIT", hdbc, SQL_HANDLE_DBC);
    printf("SQLConnect success\n");
    // Initialize table information.
    Exec(hdbc, "DROP TABLE IF EXISTS test_odbc_batch_insert");
    Exec(hdbc, "CREATE TABLE test_odbc_batch_insert (id INT PRIMARY KEY, col VARCHAR2(50))");
    // Commit the transaction in segments for other SQL operations.
    retcode = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
    CHECK_ERROR(retcode, "SQLEndTran", hdbc, SQL_HANDLE_DBC);
    // Construct the data to be inserted in batches based on the user-specified data size.
    {
        SQLRETURN retcode;
        SQLHSTMT hstmt = SQL_NULL_HSTMT;
        SQLCHAR *sql = NULL;
        SQLINTEGER *ids = NULL;
        SQLCHAR *cols = NULL;
        SQLLEN *bufLenIds = NULL;
        SQLLEN *bufLenCols = NULL;
        SQLUSMALLINT *operptr = NULL;
        SQLUSMALLINT *statusptr = NULL;
        SQLULEN process = 0;
        // Construct fields by column.
        ids = (SQLINTEGER *)malloc(sizeof(ids[0]) * BATCH_SIZE);
        cols = (SQLCHAR *)malloc(sizeof(cols[0]) * BATCH_SIZE * 50);
        // Memory length for each row of data with each field.
        bufLenIds = (SQLLEN *)malloc(sizeof(bufLenIds[0]) * BATCH_SIZE);
        bufLenCols = (SQLLEN *)malloc(sizeof(bufLenCols[0]) * BATCH_SIZE);
        if (NULL == ids || NULL == cols || NULL == bufLenCols || NULL == bufLenIds) {
            fprintf(stderr, "FAILED:\tmalloc data memory failed\n");
            goto exit;
        }
        // Assign values to data.
        for (i = 0; i < BATCH_SIZE; i++) {
            ids[i] = i;
            sprintf(cols + 50 * i, "column test value %d", i);
            bufLenIds[i] = sizeof(ids[i]);
            bufLenCols[i] = strlen(cols + 50 * i);
        }
        // Allocate a statement handle.
        retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
        CHECK_ERROR(retcode, "SQLSetConnectAttr SQL_ATTR_AUTOCOMMIT", hstmt, SQL_HANDLE_STMT);
        // Prepare statements.
        sql = (SQLCHAR *)"INSERT INTO test_odbc_batch_insert VALUES(?, ?)";
        retcode = SQLPrepare(hstmt, (SQLCHAR *)sql, SQL_NTS);
        CHECK_ERROR(retcode, "SQLPrepare", hstmt, SQL_HANDLE_STMT);
        // Bind parameters.
        retcode = SQLBindParameter(
            hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, sizeof(ids[0]), 0, &(ids[0]), 0,
bufLenIds);
        CHECK_ERROR(retcode, "SQLBindParameter 1", hstmt, SQL_HANDLE_STMT);
        retcode = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, 50, 50, cols, 50,
bufLenCols);
        CHECK_ERROR(retcode, "SQLBindParameter 2", hstmt, SQL_HANDLE_STMT);
        // Set the total number of rows in the parameter array.
        retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMSET_SIZE, (SQLPOINTER)BATCH_SIZE,
sizeof(BATCH_SIZE));
        CHECK_ERROR(retcode, "SQLSetStmtAttr", hstmt, SQL_HANDLE_STMT);
        // Set the number of processed rows.
        retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMS_PROCESSED_PTR, (SQLPOINTER)&process,
sizeof(process));
        CHECK_ERROR(retcode, "SQLSetStmtAttr SQL_ATTR_PARAMS_PROCESSED_PTR", hstmt,
SQL_HANDLE_STMT);
        // Execute batch insertion.
        retcode = SQLExecute(hstmt);
        // Manually commit the transaction.
        if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
            retcode = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
            CHECK_ERROR(retcode, "SQLEndTran", hdbc, SQL_HANDLE_DBC);
        }
        // On failure, roll back the transaction.
        else {
            CHECK_ERROR_VOID(retcode, "SQLExecute", hstmt, SQL_HANDLE_STMT);
            retcode = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK);
            printf("Transaction rollback\n");
            CHECK_ERROR(retcode, "SQLEndTran", hdbc, SQL_HANDLE_DBC);
        }
        // Obtain the number of rows processed in batches.
        SQLRowCount(hstmt, &rowsCount);
        sprintf((char *)loginfo, "SQLRowCount : %ld", rowsCount);
        puts(loginfo);
        // Check whether the number of inserted rows matches the number of processed rows.
        if (rowsCount != process) {
            sprintf(loginfo, "process(%d) != rowsCount(%d)", process, rowsCount);
            puts(loginfo);
        } else {
            sprintf(loginfo, "process(%d) == rowsCount(%d)", process, rowsCount);
        }
        retcode = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
        CHECK_ERROR(retcode, "SQLFreeHandle", hstmt, SQL_HANDLE_STMT);
    }
exit:
    (void)printf("Complete.\n");
    // Close the connection.
    if (hdbc != SQL_NULL_HDBC) {
        SQLDisconnect(hdbc);
        SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
    }
    // Release the environment handle.
    if (henv != SQL_NULL_HENV)
        SQLFreeHandle(SQL_HANDLE_ENV, henv);
    return 0;
}
void print_diag(char *msg, SQLSMALLINT htype, SQLHANDLE handle)
{
    char sqlstate[32];
    char message[1000];
    SQLINTEGER nativeerror;
    SQLSMALLINT textlen;
    SQLRETURN ret;
    SQLSMALLINT recno = 0;
    if (msg)
        printf("%s\n", msg);
    do {
        recno++;
        // Obtain diagnostic information.
        ret = SQLGetDiagRec(
            htype, handle, recno, (SQLCHAR *)sqlstate, &nativeerror, (SQLCHAR *)message, sizeof(message),
&textlen);
        if (ret == SQL_INVALID_HANDLE)
            printf("Invalid handle\n");
        else if (SQL_SUCCEEDED(ret))
            printf("%s=%s\n", sqlstate, message);
    } while (ret == SQL_SUCCESS);
    if (ret == SQL_NO_DATA && recno == 1)
        printf("No error information\n");
}
```

Result Verification

After successfully connecting to the database, ODBC inserts 100 data records in batches. Below are the expected results for **Complete Example**:

```
SQLConnect success
SQLRowCount : 100
Complete.
```

Rollback Method

If any abnormal operations occur during the transaction, call the SQLEndTran API to roll them back, as follows:

```
SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK);
```

14.2.5 Typical Issues

1. Question: After setting **ignoreCount** (which specifies the data size to exclude), can I select which specific data to exclude from the database?

   Answer: Sure. In **Complete Example**, you can adjust the operator pointer **operptr** to specify which data should be inserted and which should be excluded. Specifically, when **operptr** is set to **SQL_PARAM_IGNORE**, the data will be excluded; when **operptr** is set to **SQL_PARAM_PROCEED**, the data will be inserted.

2. Question: If batch insertion fails, can the transaction be committed for successfully inserted data and rolled back only for data that encounters an error?

   Answer: No, but in the event of insertion failure, the transaction must be rolled back for all scheduled data. However, to address the issue, one possible solution is to define smaller data batches in the application code to commit the data in batches for better exception capture.

3. Question: I have set **UseServerSidePrepare** to **0** and inserted *n* records in batches, but why does the SQLRowCount API return **1** as the number of inserted records?

   Answer: This result is as expected. Let's revisit the explanation of **UseServerSidePrepare** provided in **Preparations** for the **odbc.ini** file for Linux environments: When **UseServerSidePrepare** is set to **1**, PU/PBE packets will be sent for soft parsing. In this case, **SQLRowCount** returns the number of records updated last time. Conversely, when **UseServerSidePrepare** is set to **0**, Q packets will be sent for hard parsing, where **SQLRowCount** still returns the number of records updated last time. However, because only one SQL statement is carried in each Q packet, the number of inserted records is 1.

15 Best Practices for Go

15.1 Best Practices for Go (Distributed Instances)

15.1.1 Scenario Overview

This section explains how to use the Go driver for batch data insertion.

15.1.1.1 Usage Scenarios

Scenario Description

Batch insertion offers a more efficient way to write multiple records into the database in a single operation. Compared to single-record insertion, batch insertion decreases interactions between the application and the database, leading to lower network latency and system resource usage, while significantly improving data write efficiency.

This section illustrates various operations using the Go driver, including establishing database connections, utilizing transactions, executing batch insertion, and obtaining column information in the result set.

Trigger Conditions

The application code has generated an SQL statement for batch insertion and bound the necessary parameters. Subsequently, the Exec API is called within a transaction to execute the SQL statement.

Impact on Services

- Lower network interaction costs

  Combining multiple INSERT statements into one batch operation significantly reduces the number of round trips between the client and the database. This enhances the overall throughput and minimizes the impact of network congestion on performance.
- Higher data processing efficiency

In single-record insertion, the database must parse the syntax and generate an execution plan for each SQL statement. In contrast, batch insertion requires parsing the syntax and generating the plan only once, eliminating repetitive tasks and saving CPU cycles and memory allocation time.

- Reduced system resource usage and overhead

  In single-record insertion, transaction commits or Xlog writes occur at least once. In contrast, batch insertion allows multiple records to be inserted within a single transaction, significantly reducing the frequency of transaction commits, Xlog pressure, and transaction management overhead. In addition, it decreases the total number of network packet processing, transaction management, log write, and row format conversion tasks, which in turn lowers the CPU loads and temporary memory usage of the database server. This results in more resources being available for core query and computing operations.

- Higher memory usage

  When large data sizes are involved, constructing SQL statements for batch insertion can significantly increase memory usage. This is particularly noticeable when you construct SQL statements through string concatenation, as it can lead to a sharp rise in memory consumption. Large-size batch processing may exceed the maximum SQL length limit of the database or driver, or trigger other parameter restrictions, potentially leading to errors or performance issues.

Here is a detailed comparison between batch insertion and single-record insertion.

Mode	**Advantages**	**Disadvantages**
Single-record insertion	● Its code is simple, straightforward, and easy to implement. ● If any single record fails, it can be accurately identified and handled without impacting other records. ● This mode is less demanding in terms of database and driver compatibility.	● Extensive network interactions are needed. Each INSERT operation requires connecting, parsing, and committing, leading to suboptimal performance. ● Inserting a large number of records is likely to cause a bottleneck. ● Not using transactions may result in failure to guarantee the consistency of INSERT operations.

Mode	Advantages	Disadvantages
Batch insertion	● This mode greatly reduces the number of network round trips and SQL parsing instances, leading to a notable improvement in insertion throughput. ● Multiple rows can be committed within a single transaction to guarantee atomicity.	● Its code is complex, requiring manual concatenation of placeholders and parameters. ● If a single statement encounters an error, all data will be rolled back, complicating the error recovery process. ● The number of placeholders is limited; therefore, it is essential to carefully manage the batch size.

Applicable Versions

This applies only to GaussDB 503.1.0 and later versions.

15.1.1.2 Requirements and Objectives

Service Pain Points

When dealing with large data sizes, single-record insertion generates numerous network requests and consumes substantial system resources. Moreover, the database server has to repeatedly parse similar statements, leading to a decline in service performance. Batch insertion is introduced as a solution to these issues.

Service Objectives

Use the Go driver to initialize the target table, and insert the required data in batches through transactions for future queries. Once the batch insertion is complete, obtain column data from the result set and output the result information.

15.1.2 Architecture Principles

Core Principles

Batch processing of Go allows using one SQL statement to send multiple records into the database in one go. All insert or update operations in the transaction are carried in a single U packet. Consequently, completing the batch operation only necessitates once instance of network connection establishment and data exchange.

Solution Advantages

- Optimized network communication: Sending all batch updates at once in a single U packet significantly decreases network communication overhead when compared to sending PBE packets multiple times.

- Improved execution efficiency: Due to the reduced number and frequency of network communication, the overall execution efficiency is significantly improved, especially for large data sizes.
- Optimized resource utilization: Batch insertion optimizes the utilization of database server resources, cutting down on unnecessary system overhead associated with single-record insertion.

15.1.3 Preparations

- Golang version: 1.13 or later.
- Database environment:

  GaussDB 503.1.0 or later
- Go driver environment:

  Refer to "Application Development Guide > Development Based on the Go Driver > Development Procedure > Preparing the Environment" in *Developer Guide*.
- Environment variables needed by code:

  Taking Linux environments as an example:

```
export GOHOSTIP='127.0.0.1'              # IP address. Adjust it based on the actual services.
export GOPORT='5432'                    # Port number. Adjust it based on the actual services.
export GOUSRNAME='test_user'             # Name of a database user. Adjust it based on the actual
services.
export GOPASSWD='xxxxxxxx'              # Password for the database user. Adjust it based on the
actual services.
export GODBNAME='gaussdb'               # Database name. Adjust it based on the actual services.
export GOCONNECT_TIMEOUT='3'            # Timeout interval for database connection. Adjust it
based on the actual services.
export GOSOCKET_TIMEOUT='1'             # Timeout interval for a single SQL statement. Adjust it
based on the actual services.
export GOSSLMODE='verify-full'          # Specifies whether to enable SSL encryption. Adjust it
based on the actual services.
export GOROOTCERT='certs/cacert.pem'     # Path to the root certificate. Adjust it based on the
actual services.
export GOSSLKEY='certs/client-key.pem'    # Path to the client key. Adjust it based on the actual
services.
export GOSSLCERT='certs/client-cert.pem'   # Path to the client certificate. Adjust it based on the
actual services.
```

NOTE

Adjust the values of environment variables as needed. However, if your code does not intend to obtain connection parameter values from environment variables, you may skip this step.

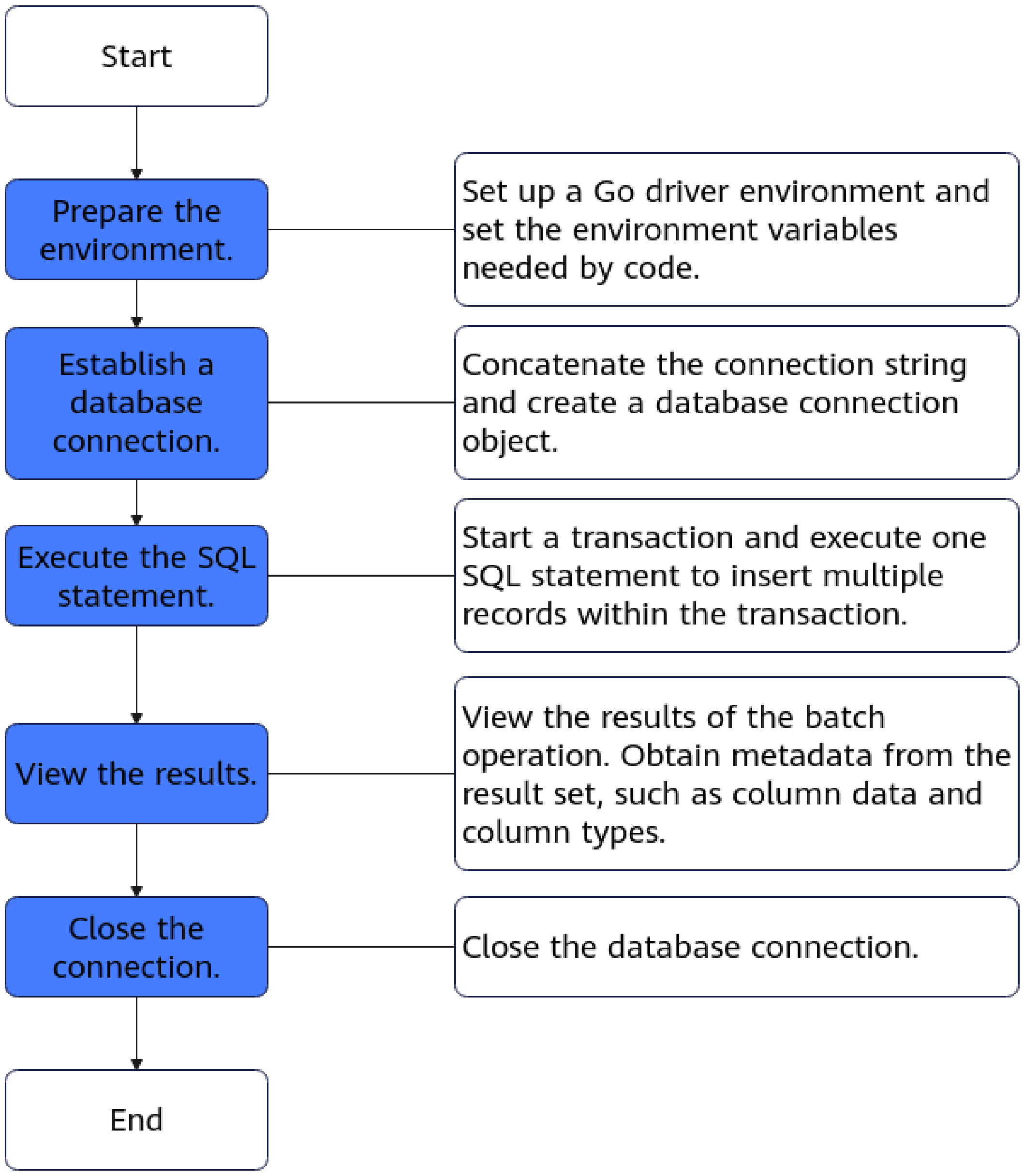
15.1.4 Procedure

15.1.4.1 Process Overview

The Go driver can create database connections and insert data in batches within a transaction.

Figure 15-1 shows the overall process.

Figure 15-1 Process of batch insertion


Start
Prepare the environment.
Set up a Go driver environment and set the environment variables needed by code.
Establish a database connection.
Concatenate the connection string and create a database connection object.
Execute the SQL statement.
Start a transaction and execute one SQL statement to insert multiple records within the transaction.
View the results.
View the results of the batch operation. Obtain metadata from the result set, such as column data and column types.
Close the connection.
Close the database connection.
End

15.1.4.2 Detailed Procedure

Step 1 Obtain the variable values needed by connection parameters and concatenate them to create a connection string.

NOTE

- The connection string can be in DSN or URL format.
- For details about database connection parameters, refer to "Application Development Guide > Development Based on the Go Driver > Development Procedure > Preparing the Environment" in *Developer Guide*.

The parameter values involved in **Detailed Procedure** can be obtained from the environment variables set in **Preparations** and concatenated, as shown in the following code. You can obtain the values of connection parameters from environment variables by using os.Getenv. Alternatively, you can set these values by reading configuration files or writing fixed values.

```
hostip := os.Getenv("GOHOSTIP")                 // GOHOSTIP: IP address written to environment variables.
port := os.Getenv("GOPORT")                     // GOPORT: port number written to environment variables.
usrname := os.Getenv("GOUSRNAME")               // GOUSRNAME: username written to environment variables.
passwd := os.Getenv("GOPASSWD")                 // GOPASSWD: user password written to environment variables.
dbname := os.Getenv("GODBNAME")                 // GODBNAME: name of the target database written to environment variables.
connect_timeout := os.Getenv("GOCONNECT_TIMEOUT") // GOCONNECT_TIMEOUT: timeout interval for connecting to the database written to environment variables.
socket_timeout := os.Getenv("GOSOCKET_TIMEOUT")   // GOSOCKET_TIMEOUT: maximum duration of the SQL statement written to environment variables.
rootcertPath := os.Getenv("GOROOTCERT")         // GOROOTCERT: path to the root certificate written to environment variables.
sslkeyPath := os.Getenv("GOSSLKEY")             // GOSSLKEY: path to the key of the client certificate written to environment variables.
sslcertPath := os.Getenv("GOSSLCERT")           // GOSSLCERT: path to the client SSL certificate written to environment variables.
sslmode := os.Getenv("GOSSLMODE")               // GOSSLMODE: SSL encryption written to environment variables.
```

- In case of a DSN connection string, refer to the following recommended connection settings and format when assigning values to the *dsn* variable:

```
dsn := fmt.Sprintf("host=%s port=%s user=%s password=%s dbname=%s connect_timeout=%s socketTimeout=%s sslmode=%s sslrootcert=%s sslkey=%s sslcert=%s target_session_attrs=master autoBalance=true",
 hostip,
 port,
 usrname,
 passwd,
 dbname,
 connect_timeout,
 socket_timeout,
 sslmode,
 rootcertPath,
 sslkeyPath,
 sslcertPath,
)
```

- In case of a URL connection string, refer to the following recommended URL connection settings and format when assigning values to the *url* variable:

```
url := fmt.Sprintf("gaussdb://%s:%s@%s:%s/%s?connect_timeout=%s&socketTimeout=%s&sslmode=%s&sslrootcert=%s&sslkey=%s&sslcert=%s&target_session_attrs=master&autoBalance=true",
 usrname,
 passwd,
 hostip,
 port,
 dbname,
 connect_timeout,
 socket_timeout,
 sslmode,
 rootcertPath,
 sslkeyPath,
 sslcertPath,
)
```

NOTE

- **connect_timeout**: timeout interval (in seconds) for connecting to the database server. The timeout interval must be set based on the actual network conditions. A default value of **0** indicates that no timeout will occur.
- **socket_timeout**: maximum duration of a single SQL statement. If a statement exceeds this limit, it will be interrupted and reconnected. You are advised to set this parameter based on service characteristics. If not specified, the default value **0** will be applied, indicating that no timeout will occur.
- **sslmode**: specifies whether to enable SSL encryption.
- **target_session_attrs**: connection type of the database. This parameter is used to identify the primary and standby nodes. The default value is **any**.
- **autoBalance**: load balancing policy. It is a character string, with a default value of **false**.

Step 2 Create a database connection object using the connection string concatenated in **Step 1**.

Golang's database/sql standard library provides the sql.Open API for creating a database connection object. Upon completion, the API returns the database connection object and any error information.

```
func Open(driverName, dataSourceName string) (*DB, error)
```

- Define a DSN connection string as follows:
  ```
  db, err := sql.Open("gaussdb", dsn)
  ```
- Define a URL connection string as follows:
  ```
  db, err := sql.Open("gaussdb", url)
  ```

Step 3 Create a transaction object using the database connection object created in **Step 2**.

The database connection object provides the Begin API for creating a transaction object. Upon completion, the API returns the transaction object and any error information.

```
func (db *DB) Begin() (*Tx, error)
```

The following creates a transaction object and receives the transaction object through the *tx* variable:

```
tx, err := db.Begin()
```

Step 4 Execute batch insertion using the transaction object created in **Step 3**.

The Exec API is used as an example. For details, see "Application Development Guide > Development Based on the Go Driver > Go API Reference > type Tx" in *Developer Guide*.

```
(tx *Tx)Exec(query string, args ...interface{})
```

The Exec API is called to insert the user-specified data size into the **employee** table in batches by using the transaction object **"tx"** created in **Step 3**. This involves concatenating the SQL statement for batch insertion and passing the necessary values.

```
employee := []struct {
 Name string
 Age  uint8
}{{Name: "zhangsan", Age: 21}, {Name: "lisi", Age: 22}, {Name: "zhaowu", Age: 23}}
```

```
batchSql := "INSERT INTO employee (name, age) VALUES "
vals := []interface{}{}

placeholders := ""
for i, u := range employee {
 placeholders += "(?, ?)"
 if i < len(employee)-1 {
  placeholders += ","
 }
 vals = append(vals, u.Name, u.Age)
}

stmt := batchSql + placeholders
result, err := tx.Exec(stmt, vals...)
```

Step 5 (Optional) Roll back the transaction using the transaction object created in **Step 3**.

The transaction object provides the Rollback API for rolling back the transaction.

```
func (tx *Tx) Rollback() error
```

If an error occurs in the transaction, call the Rollback API of the transaction object **"tx"** created in **Step 3** to roll back the transaction.

```
tx.Rollback()
```

Step 6 Commit the transaction using the transaction object created in **Step 3**.

The transaction object provides the Commit API for committing the transaction.

```
func (tx *Tx) Commit() error
```

Commit the transaction through the Commit API of the transaction object **"tx"** created in **Step 3**.

```
err := tx.Commit()
```

Step 7 (Optional) Execute a query using the database connection object created in **Step 2**.

Both the database object and the transaction object provide the Query API. For details, see "type DB" and "type Tx" under "Application Development Guide > Development Based on the Go Driver > Go API Reference" in *Developer Guide*.

For example, call the Query API provided by the database object **"db"** created in **Step 2** to query the batch insertion results in **Step 4** and receive the result object through the *"rows"* variable.

```
rows, err := db.Query("SELECT id, name, created_at FROM users;"
```

Step 8 (Optional) Obtain the column count and column name list in the result set using the result object in **Step 7**.

The result object in **Step 7** is of the Rows type in database/sql of Golang. This type provides a Columns API to return the list of column names in the query result set. For details, see "Application Development Guide > Development Based on the Go Driver > Go API Reference > type Rows".

```
func (rs *Rows) Columns() ([]string, error)
```

The following calls the Columns API provided by the result object in **Step 7** to obtain the list of queried column names and assign values to the *columns* variable.

```
columns, err := rows.Columns()
```

The column count in the result set can be obtained by calling the len function to calculate **columns**.

```
len(columns)
```

Step 9 (Optional) Obtain metadata such as column types in the result set by using the result object in **Step 7**.

The result object in **Step 7** is of the Rows type in database/sql of Golang. This type provides a ColumnTypes API to return the list of column names in the query result set. For details, see "Application Development Guide > Development Based on the Go Driver > Go API Reference > type Rows".

```
func (rs *Rows) ColumnTypes() ([]*ColumnType, error)
```

Obtain column information from the result in **Step 7**.

The following calls the ColumnTypes API provided by the result object in **Step 7** to obtain the list of queried column types (**[]*ColumnType**) and assign values to the *columnTypes* variable.

```
columnTypes, err := rows.ColumnTypes()
```

By traversing the **columnTypes** list, the application code can determine the types of returned columns.

The *type ColumnType* variable provides APIs to describe the column types available in database tables.

Table 15-1 Common methods for using the APIs provided by *type ColumnType*

Method	Description	Return Type
(ci *ColumnType)DatabaseTypeName()	Returns a column-type database system name. If an empty string is returned, that type of name is not supported.	Error
(ci *ColumnType)DecimalSize()	Returns the scale and precision of the decimal type. If the value of **ok** is **false**, the specified type is unavailable or unsupported.	Precision and scale: int64; **ok**: Boolean
(ci *ColumnType)Length()	Returns the length of a data column type. If the value of **ok** is **false**, the specified type does not have a variable length.	Length: int64; **ok**: Boolean
(ci *ColumnType)ScanType()	Returns a Go type that can be used for Rows.scan.	reflect.Type
(ci *ColumnType)Name()	Returns the name of a data column.	String

Step 10 Close the connection using the database connection object created in **Step 2**.

The database connection object provides the Close API for closing the database connection.

```
func (db *DB) Close() error
```

Execute the following statement to close the database connection object **"db"** created in **Step 2**:

```
db.Close()
```

----End

15.1.4.3 Complete Example

The following uses a DSN connection string as an example to explain how to initialize the **employee** table, insert data in batches, and obtain column information in the result set:

```
// main.go
package main

import (
 "database/sql"
 "fmt"
 _ "gitee.com/opengauss/openGauss-connector-go-pq"
 "log"
 "os"
)

func main() {
 // Create a database object.
 hostip := os.Getenv("GOHOSTIP")                  // GOHOSTIP: IP address written to environment variables.
 port := os.Getenv("GOPORT")                      // GOPORT: port number written to environment variables.
 usrname := os.Getenv("GOUSRNAME")                // GOUSRNAME: username written to environment
variables.
 passwd := os.Getenv("GOPASSWD")                  // GOPASSWD: user password written to environment
variables.
 dbname := os.Getenv("GODBNAME")                  // GODBNAME: name of the target database written to
environment variables.
 connect_timeout := os.Getenv("GOCONNECT_TIMEOUT") // GOCONNECT_TIMEOUT: timeout interval for
connecting to the database written to environment variables.
 socket_timeout := os.Getenv("GOSOCKET_TIMEOUT")   // GOSOCKET_TIMEOUT: maximum duration of
the SQL statement written to environment variables.
 rootcertPath := os.Getenv("GOROOTCERT")           // GOROOTCERT: path to the root certificate written to
environment variables.
 sslkeyPath := os.Getenv("GOSSLKEY")               // GOSSLKEY: path to the key of the client certificate
written to environment variables.
 sslcertPath := os.Getenv("GOSSLCERT")             // GOSSLCERT: path to the client SSL certificate written to
environment variables.
 sslmode := os.Getenv("GOSSLMODE")                 // GOSSLMODE: SSL encryption written to environment
variables.

dsn := fmt.Sprintf("host=%s port=%s user=%s password=%s dbname=%s connect_timeout=%s
socketTimeout=%s "+
   "sslmode=%s sslrootcert=%s sslkey=%s sslcert=%s target_session_attrs=master",
   hostip,
   port,
   usrname,
   passwd,
   dbname,
   connect_timeout,
   socket_timeout,
   sslmode,
   rootcertPath,
   sslkeyPath,
   sslcertPath,
)
```

```
db, err := sql.Open("gaussdb", dsn)
if err != nil {
 panic(err)
}
defer db.Close()
err = db.Ping()
if err != nil {
 panic(err)
}
fmt.Println("connect success.")
// Start a transaction.
tx, err := db.Begin()
if err != nil {
 log.Fatal(err)
 return
}
// Initialize a data table.
_, err = tx.Exec("drop table if exists employee;")
if err != nil {
 fmt.Println("drop table employee failed, err:", err)
 err = tx.Rollback() // On error, roll back the transaction.
 return
}
fmt.Println("drop table employee success.")
_, err = tx.Exec("create table employee (id SERIAL PRIMARY KEY, name varchar(20), age int, created_at
TIMESTAMP DEFAULT CURRENT_TIMESTAMP);")
if err != nil {
 fmt.Println("create table employee failed, err:", err)
 err = tx.Rollback()
 return
}
fmt.Println("create table employee success.")
// Insert data in batches.
employee := []struct {
 Name string
 Age  uint8
}{{Name: "zhangsan", Age: 21}, {Name: "lisi", Age: 22}, {Name: "zhaowu", Age: 23}}

batchSql := "INSERT INTO employee (name, age) VALUES "
vals := []interface{}{}

placeholders := "(?, ?)"
for _, u := range employee {
 vals = append(vals, u.Name, u.Age)
}

stmt := batchSql + placeholders
_, err = tx.Exec(stmt, vals...)

if err != nil {
 fmt.Println("batch insert into table employee failed, err:", err)
 err = tx.Rollback()
 return
}
fmt.Println("batch insert into table employee success.")
// Commit the transaction.
err = tx.Commit()
if err != nil {
 fmt.Println("commit failed, err:", err)
 err = tx.Rollback()
 log.Fatal(err)
 return
}
fmt.Println("commit success.")
// Obtain column information in the result set.
rows, err := db.Query("SELECT id, name, created_at FROM employee;")
if err != nil {
 fmt.Println("query table employee failed, err:", err)
 return
}
columns, err := rows.Columns()
if err != nil {
 fmt.Println("get query rows columns failed, err:", err)
 return
}
fmt.Println("Column count: ", len(columns))
fmt.Println("Column name list: ", columns)
fmt.Println("--------------------------")
// Obtain column types.
columnTypes, err := rows.ColumnTypes()
if err != nil {
 fmt.Println("get query rows ColumnTypes failed, err:", err)
 return
}

for _, ct := range columnTypes {
 fmt.Println("Column name: ", ct.Name())
 fmt.Println("Database type: ", ct.DatabaseTypeName())
 length, ok := ct.Length()
 if ok {
  fmt.Println("Length: ", length)
 }
 precision, scale, ok := ct.DecimalSize()
 if ok {
  fmt.Printf("Precision/Scale: %d/%d\n", precision, scale)
 }
 nullable, ok := ct.Nullable()
 if ok {
  fmt.Println("Nullable: ", nullable)
 }
 fmt.Println("Go type: ", ct.ScanType())
 fmt.Println("-----")
}
}
```

Result Verification

Below are the execution results for **Complete Example**:

```
connect success.
drop table employee success.
create table employee success.
batch insert into table employee success.
commit success.
Column count: 3
Column name list: [id name created_at]
--------------------------
Column name: id
Database type: INT4
Go type: int32
-----
Column name: name
Database type: VARCHAR
Length: 20
Go type: string
-----
Column name: created_at
Database type: TIMESTAMP
Go type: time.Time
-----
```

The following tasks have been completed as expected:

1. Concatenate the database connection string, create a connection object using sql.Open, and verify the connection status through the db.Ping() method.
2. Start a transaction and initialize the test table **employee** in the transaction.

3. Construct employee test data in the transaction, generate an SQL statement for batch insertion, bind parameters through the Exec API provided by the transaction object, and send packets to the database to execute the SQL statement.
4. After a successful batch insertion, call the Commit API to commit the transaction. (If the insertion fails, call the Rollback API to roll back the transaction.)
5. Call the Query API provided by the database connection object **"db"** to query the batch insertion results. Call the Columns API provided by the result object **"rows"** to obtain the list of all column names. Call the ColumnTypes API provided by the result object **"rows"** to obtain metadata about columns in the result set.

Rollback Method

To roll back operations within a specific transaction, call the Rollback API of the transaction object.

15.1.5 Typical Issues

1. SQL injection risks during the construction of batch insertion statements

   Placeholders and parameter binding are used rather than directly concatenating user-specified values. When inserting data, multi-line VALUES statements use placeholders (?) and a parameter list to prevent injection attacks. All dynamic data must be passed as parameters. For example, the Prepare or Exec API of the database object *DB can be used to pass variable parameter forms.
2. Batch insertion failure

   If a record fails to be inserted during batch insertion, the database returns only general error information (such as primary key conflict, foreign key constraint violated, or data type mismatch). However, it does not indicate which specific record is causing the error. If an SQL statement contains multiple records and one of them fails to be inserted, the entire transaction may fail (unless the error ignoring mechanism is enabled). To pinpoint the specific row causing the error, it is common practice to divide the batch into smaller batches or insert data row by row for better error capture.
3. Increased memory usage

   When large data sizes are involved, constructing SQL statements for batch insertion can significantly increase memory usage. This is particularly noticeable when you construct SQL statements through string concatenation, as it can lead to a sharp rise in memory consumption. Large-size batch processing may exceed the maximum SQL length limit of the database or Go driver, or trigger other parameter restrictions, potentially leading to errors or performance issues.

15.2 Best Practices for Go (Centralized Instances)

15.2.1 Scenario Overview

This section explains how to use the Go driver for batch data insertion.

15.2.1.1 Usage Scenarios

Scenario Description

Batch insertion offers a more efficient way to write multiple records into the database in a single operation. Compared to single-record insertion, batch insertion decreases interactions between the application and the database, leading to lower network latency and system resource usage, while significantly improving data write efficiency.

This section illustrates various operations using the Go driver, including establishing database connections, utilizing transactions, executing batch insertion, and obtaining column information in the result set.

Trigger Conditions

The application code has generated an SQL statement for batch insertion and bound the necessary parameters. Subsequently, the Exec API is called within a transaction to execute the SQL statement.

Impact on Services

- Lower network interaction costs

  Combining multiple INSERT statements into one batch operation significantly reduces the number of round trips between the client and the database. This enhances the overall throughput and minimizes the impact of network congestion on performance.
- Higher data processing efficiency

  In single-record insertion, the database must parse the syntax and generate an execution plan for each SQL statement. In contrast, batch insertion requires parsing the syntax and generating the plan only once, eliminating repetitive tasks and saving CPU cycles and memory allocation time.
- Reduced system resource usage and overhead

  In single-record insertion, transaction commits or Xlog writes occur at least once. In contrast, batch insertion allows multiple records to be inserted within a single transaction, significantly reducing the frequency of transaction commits, Xlog pressure, and transaction management overhead. In addition, it decreases the total number of network packet processing, transaction management, log write, and row format conversion tasks, which in turn lowers the CPU loads and temporary memory usage of the database server. This results in more resources being available for core query and computing operations.
- Higher memory usage

  When large data sizes are involved, constructing SQL statements for batch insertion can significantly increase memory usage. This is particularly noticeable when you construct SQL statements through string concatenation, as it can lead to a sharp rise in memory consumption. Large-size batch processing may exceed the maximum SQL length limit of the database or driver, or trigger other parameter restrictions, potentially leading to errors or performance issues.

Here is a detailed comparison between batch insertion and single-record insertion.

Mode	Advantages	Disadvantages
Single-record insertion	● Its code is simple, straightforward, and easy to implement. ● If any single record fails, it can be accurately identified and handled without impacting other records. ● This mode is less demanding in terms of database and driver compatibility.	● Extensive network interactions are needed. Each INSERT operation requires connecting, parsing, and committing, leading to suboptimal performance. ● Inserting a large number of records is likely to cause a bottleneck. ● Not using transactions may result in failure to guarantee the consistency of INSERT operations.
Batch insertion	● This mode greatly reduces the number of network round trips and SQL parsing instances, leading to a notable improvement in insertion throughput. ● Multiple rows can be committed within a single transaction to guarantee atomicity.	● Its code is complex, requiring manual concatenation of placeholders and parameters. ● If a single statement encounters an error, all data will be rolled back, complicating the error recovery process. ● The number of placeholders is limited; therefore, it is essential to carefully manage the batch size.

Applicable Versions

This applies only to GaussDB V500R002C10 and later versions.

15.2.1.2 Requirements and Objectives

Service Pain Points

When dealing with large data sizes, single-record insertion generates numerous network requests and consumes substantial system resources. Moreover, the database server has to repeatedly parse similar statements, leading to a decline in service performance. Batch insertion is introduced as a solution to these issues.

Service Objectives

Use the Go driver to initialize the target table, and insert the required data in batches through transactions for future queries. Once the batch insertion is complete, obtain column data from the result set and output the result information.

15.2.2 Architecture Principles

Core Principles

Batch processing of Go allows using one SQL statement to send multiple records into the database in one go. All insert or update operations in the transaction are carried in a single U packet. Consequently, completing the batch operation only necessitates once instance of network connection establishment and data exchange.

Solution Advantages

- Optimized network communication

  Sending all batch updates at once in a single U packet significantly decreases network communication overhead when compared to sending PBE packets multiple times.
- Improved execution efficiency

  Due to the reduced number and frequency of network communication, the overall execution efficiency is significantly improved, especially for large data sizes.
- Optimized resource utilization

  Batch insertion optimizes the utilization of database server resources, cutting down on unnecessary system overhead associated with single-record insertion.

15.2.3 Preparations

- Golang version: 1.13 or later.
- Database environment: GaussDB V500R002C10 or later.
- Go driver environment:

  Refer to "Application Development Guide > Development Based on the Go Driver > Development Procedure > Preparing the Environment" in *Developer Guide*.
- Environment variables needed by code:

  Taking Linux environments as an example:

```
export GOHOSTIP='127.0.0.1'                 # IP address. Adjust it based on the actual services.
export GOPORT='5432'                        # Port number. Adjust it based on the actual services.
export GOUSRNAME='test_user'                # Name of a database user. Adjust it based on the actual services.
export GOPASSWD='xxxxxxxx'                  # Password for the database user. Adjust it based on the actual services.
export GODBNAME='gaussdb'                   # Database name. Adjust it based on the actual services.
export GOCONNECT_TIMEOUT='3'                # Timeout interval for database connection. Adjust it based on the actual services.
export GOSOCKET_TIMEOUT='1'                 # Timeout interval for a single SQL statement. Adjust it based on the actual services.
export GOSSLMODE='verify-full'              # Specifies whether to enable SSL encryption. Adjust it based on the actual services.
export GOROOTCERT='certs/cacert.pem'        # Path to the root certificate. Adjust it based on the actual services.
export GOSSLKEY='certs/client-key.pem'      # Path to the client key. Adjust it based on the actual services.
export GOSSLCERT='certs/client-cert.pem'    # Path to the client certificate. Adjust it based on the actual services.
```

NOTE

Adjust the values of environment variables as needed. However, if your code does not intend to obtain connection parameter values from environment variables, you may skip this step.

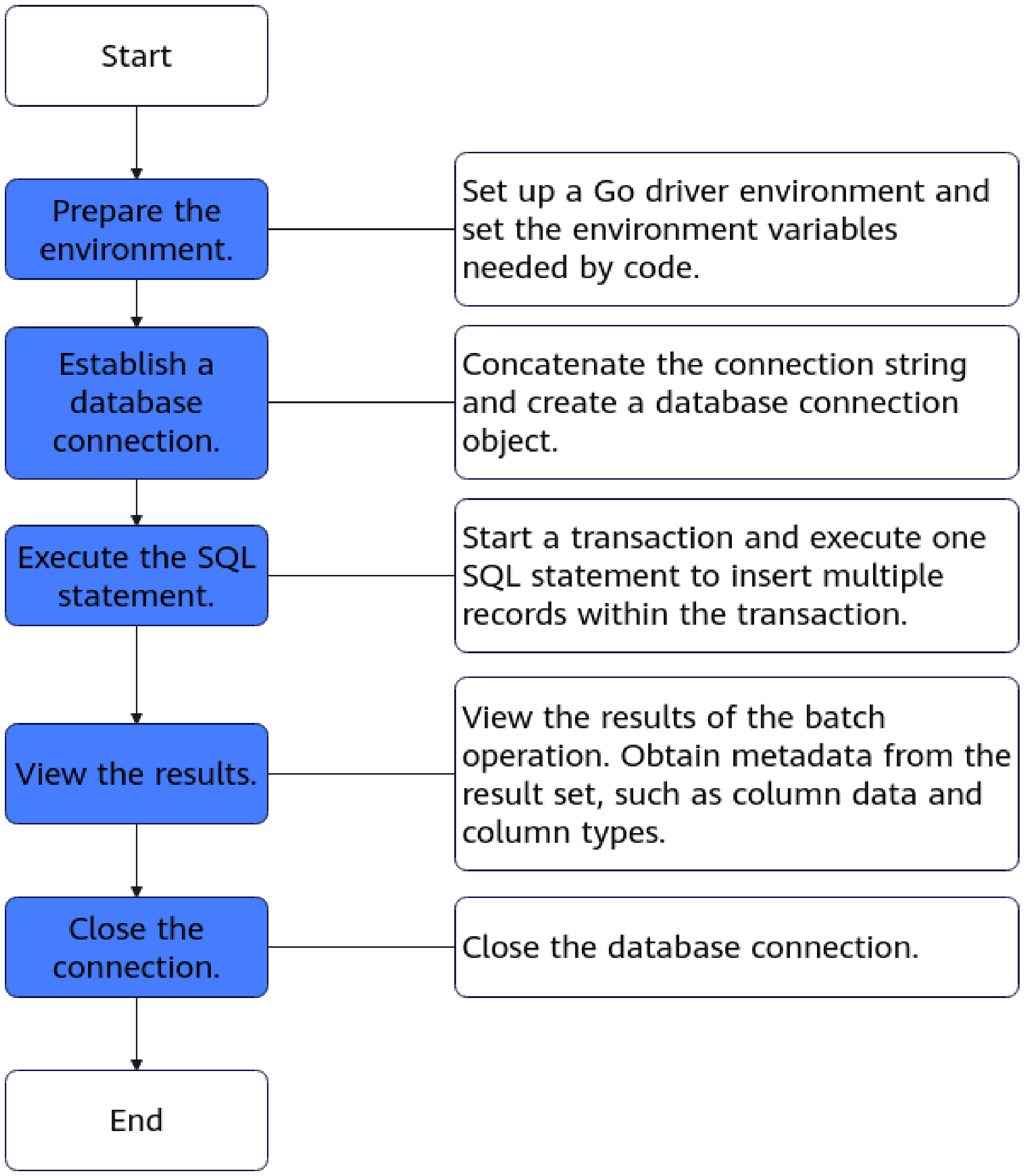
15.2.4 Procedure

15.2.4.1 Process Overview

The Go driver can create database connections and insert data in batches within a transaction.

Figure 15-2 shows the overall process.

Figure 15-2 Process of batch insertion


Start
Prepare the environment.
Set up a Go driver environment and set the environment variables needed by code.
Establish a database connection.
Concatenate the connection string and create a database connection object.
Execute the SQL statement.
Start a transaction and execute one SQL statement to insert multiple records within the transaction.
View the results.
View the results of the batch operation. Obtain metadata from the result set, such as column data and column types.
Close the connection.
Close the database connection.
End

15.2.4.2 Detailed Procedure

Step 1 Obtain the variable values needed by connection parameters and concatenate them to create a connection string.

NOTE

- The connection string can be in DSN or URL format.
- For details about database connection parameters, refer to "Application Development Guide > Development Based on the Go Driver > Development Procedure > Preparing the Environment" in *Developer Guide*.

The parameter values involved in **Detailed Procedure** can be obtained from the environment variables set in **Preparations** and concatenated, as shown in the following code. You can obtain the values of connection parameters from environment variables by using os.Getenv. Alternatively, you can set these values by reading configuration files or writing fixed values.

```
hostip := os.Getenv("GOHOSTIP")                    // GOHOSTIP: IP address written to environment
variables.
port := os.Getenv("GOPORT")                    // GOPORT: port number written to environment
variables.
usrname := os.Getenv("GOUSRNAME")                // GOUSRNAME: username written to environment
variables.
passwd := os.Getenv("GOPASSWD")                  // GOPASSWD: user password written to environment
variables.
dbname := os.Getenv("GODBNAME")                 // GODBNAME: name of the target database
written to environment variables.
connect_timeout := os.Getenv("GOCONNECT_TIMEOUT") // GOCONNECT_TIMEOUT: timeout interval
for connecting to the database written to environment variables.
socket_timeout := os.Getenv("GOSOCKET_TIMEOUT")   // GOSOCKET_TIMEOUT: maximum duration
of the SQL statement written to environment variables.
rootcertPath := os.Getenv("GOROOTCERT")          // GOROOTCERT: path to the root certificate
written to environment variables.
sslkeyPath := os.Getenv("GOSSLKEY")              // GOSSLKEY: path to the key of the client certificate
written to environment variables.
sslcertPath := os.Getenv("GOSSLCERT")            // GOSSLCERT: path to the client SSL certificate
written to environment variables.
sslmode := os.Getenv("GOSSLMODE")               // GOSSLMODE: SSL encryption written to
environment variables.
```

- In case of a DSN connection string, refer to the following recommended connection settings and format when assigning values to the *dsn* variable:

```
dsn := fmt.Sprintf("host=%s port=%s user=%s password=%s dbname=%s connect_timeout=%s
socketTimeout=%s sslmode=%s sslrootcert=%s sslkey=%s sslcert=%s target_session_attrs=master",
 hostip,
 port,
 usrname,
 passwd,
 dbname,
 connect_timeout,
 socket_timeout,
 sslmode,
 rootcertPath,
 sslkeyPath,
 sslcertPath,
)
```

- In case of a URL connection string, refer to the following recommended URL connection settings and format when assigning values to the *url* variable:

```
url := fmt.Sprintf("gaussdb://%s:%s@%s:%s/%s?connect_timeout=%s&socketTimeout=%s&sslmode=
%s&sslrootcert=%s&sslkey=%s&sslcert=%s&target_session_attrs=master",
 usrname,
 passwd,
 hostip,
 port,
 dbname,
 connect_timeout,
 socket_timeout,
 sslmode,
 rootcertPath,
 sslkeyPath,
 sslcertPath,
)
```

NOTE

- **connect_timeout**: timeout interval (in seconds) for connecting to the database server. The timeout interval must be set based on the actual network conditions. A default value of **0** indicates that no timeout will occur.
- **socket_timeout**: maximum duration of a single SQL statement. If a statement exceeds this limit, it will be interrupted and reconnected. You are advised to set this parameter based on service characteristics. If not specified, the default value **0** will be applied, indicating that no timeout will occur.
- **sslmode**: specifies whether to enable SSL encryption.
- **target_session_attrs**: connection type of the database. This parameter is used to identify the primary and standby nodes. The default value is **any**.

Step 2 Create a database connection object using the connection string concatenated in **Step 1**.

Golang's database/sql standard library provides the sql.Open API for creating a database connection object. Upon completion, the API returns the database connection object and any error information.

```
func Open(driverName, dataSourceName string) (*DB, error)
```

- Define a DSN connection string as follows:
  ```
  db, err := sql.Open("gaussdb", dsn)
  ```
- Define a URL connection string as follows:
  ```
  db, err := sql.Open("gaussdb", url)
  ```

Step 3 Create a transaction object using the database connection object created in **Step 2**.

The database connection object provides the Begin API for creating a transaction object. Upon completion, the API returns the transaction object and any error information.

```
func (db *DB) Begin() (*Tx, error)
```

The following creates a transaction object and receives the transaction object through the *tx* variable:

```
tx, err := db.Begin()
```

Step 4 Execute batch insertion using the transaction object created in **Step 3**.

The Exec API is used as an example. For details, see "Application Development Guide > Development Based on the Go Driver > Go API Reference > type Tx" in *Developer Guide*.

```
(tx *Tx)Exec(query string, args ...interface{})
```

The Exec API is called to insert the user-specified data size into the **employee** table in batches by using the transaction object **"tx"** created in **Step 3**. This involves concatenating the SQL statement for batch insertion and passing the necessary values.

```
employee := []struct {
 Name string
```

```
 Age  uint8
}{{Name: "zhangsan", Age: 21}, {Name: "lisi", Age: 22}, {Name: "zhaowu", Age: 23}}

batchSql := "INSERT INTO employee (name, age) VALUES "
vals := []interface{}{}

placeholders := ""
for i, u := range employee {
 placeholders += "(?, ?)"
 if i < len(employee)-1 {
  placeholders += ","
 }
 vals = append(vals, u.Name, u.Age)
}

stmt := batchSql + placeholders
result, err := tx.Exec(stmt, vals...)
```

Step 5 (Optional) Roll back the transaction using the transaction object created in **Step 3**.

The transaction object provides the Rollback API for rolling back the transaction.

```
func (tx *Tx) Rollback() error
```

If an error occurs in the transaction, call the Rollback API of the transaction object **"tx"** created in **Step 3** to roll back the transaction.

```
tx.Rollback()
```

Step 6 Commit the transaction using the transaction object created in **Step 3**.

The transaction object provides the Commit API for committing the transaction.

```
func (tx *Tx) Commit() error
```

Commit the transaction through the Commit API of the transaction object **"tx"** created in **Step 3**.

```
err := tx.Commit()
```

Step 7 (Optional) Execute a query using the database connection object created in **Step 2**.

Both the database object and the transaction object provide the Query API. For details, see "type DB" and "type Tx" under "Application Development Guide > Development Based on the Go Driver > Go API Reference" in *Developer Guide*.

For example, call the Query API provided by the database object **"db"** created in **Step 2** to query the batch insertion results in **Step 4** and receive the result object through the *"rows"* variable.

```
rows, err := db.Query("SELECT id, name, created_at FROM users;"
```

Step 8 (Optional) Obtain the column count and column name list in the result set using the result object in **Step 7**.

The result object in **Step 7** is of the Rows type in database/sql of Golang. This type provides a Columns API to return the list of column names in the query result set. For details, see "Application Development Guide > Development Based on the Go Driver > Go API Reference > type Rows".

```
func (rs *Rows) Columns() ([]string, error)
```

The following calls the Columns API provided by the result object in **Step 7** to obtain the list of queried column names and assign values to the *columns* variable.

```
columns, err := rows.Columns()
```

The column count in the result set can be obtained by calling the len function to calculate **columns**.

```
len(columns)
```

Step 9 (Optional) Obtain metadata such as column types in the result set by using the result object in **Step 7**.

The result object in **Step 7** is of the Rows type in database/sql of Golang. This type provides a ColumnTypes API to return the list of column names in the query result set. For details, see "Application Development Guide > Development Based on the Go Driver > Go API Reference > type Rows".

```
func (rs *Rows) ColumnTypes() ([]*ColumnType, error)
```

Obtain column information from the result in **Step 7**.

The following calls the ColumnTypes API provided by the result object in **Step 7** to obtain the list of queried column types (**[]*ColumnType**) and assign values to the *columnTypes* variable.

```
columnTypes, err := rows.ColumnTypes()
```

By traversing the **columnTypes** list, the application code can determine the types of returned columns.

The *type ColumnType* variable provides APIs to describe the column types available in database tables.

Table 15-2 Common methods for using the APIs provided by *type ColumnType*

Method	Description	Return Type
(ci *ColumnType)DatabaseTypeName()	Returns a column-type database system name. If an empty string is returned, that type of name is not supported.	Error
(ci *ColumnType)DecimalSize()	Returns the scale and precision of the decimal type. If the value of **ok** is **false**, the specified type is unavailable or unsupported.	Precision and scale: int64; **ok**: Boolean
(ci *ColumnType)Length()	Returns the length of a data column type. If the value of **ok** is **false**, the specified type does not have a variable length.	Length: int64; **ok**: Boolean
(ci *ColumnType)ScanType()	Returns a Go type that can be used for Rows.scan.	reflect.Type
(ci *ColumnType)Name()	Returns the name of a data column.	String

Step 10 Close the connection using the database connection object created in **Step 2**.

The database connection object provides the Close API for closing the database connection.

```
func (db *DB) Close() error
```

Execute the following statement to close the database connection object created in **Step 2**:

```
db.Close()
```

----End

15.2.4.3 Complete Example

The following uses a DSN connection string as an example to explain how to initialize the **employee** table, insert data in batches, and obtain column information in the result set:

```
// main.go
package main

import (
 "database/sql"
 "fmt"
 _ "gitee.com/opengauss/openGauss-connector-go-pq"
 "log"
 "os"
)

func main() {
 // Create a database object.
 hostip := os.Getenv("GOHOSTIP")                    // GOHOSTIP: IP address written to environment variables.
 port := os.Getenv("GOPORT")                     // GOPORT: port number written to environment variables.
 usrname := os.Getenv("GOUSRNAME")                  // GOUSRNAME: username written to environment
variables.
 passwd := os.Getenv("GOPASSWD")                    // GOPASSWD: user password written to environment
variables.
 dbname := os.Getenv("GODBNAME")                    // GODBNAME: name of the target database written to
environment variables.
 connect_timeout := os.Getenv("GOCONNECT_TIMEOUT") // GOCONNECT_TIMEOUT: timeout interval for
connecting to the database written to environment variables.
 socket_timeout := os.Getenv("GOSOCKET_TIMEOUT")   // GOSOCKET_TIMEOUT: maximum duration of
the SQL statement written to environment variables.
 rootcertPath := os.Getenv("GOROOTCERT")           // GOROOTCERT: path to the root certificate written to
environment variables.
 sslkeyPath := os.Getenv("GOSSLKEY")               // GOSSLKEY: path to the key of the client certificate
written to environment variables.
 sslcertPath := os.Getenv("GOSSLCERT")             // GOSSLCERT: path to the client SSL certificate written to
environment variables.
 sslmode := os.Getenv("GOSSLMODE")                 // GOSSLMODE: SSL encryption written to environment
variables.

dsn := fmt.Sprintf("host=%s port=%s user=%s password=%s dbname=%s connect_timeout=%s
socketTimeout=%s " +
   "sslmode=%s sslrootcert=%s sslkey=%s sslcert=%s target_session_attrs=master",
   hostip,
   port,
   usrname,
   passwd,
   dbname,
   connect_timeout,
   socket_timeout,
   sslmode,
   rootcertPath,
   sslkeyPath,
   sslcertPath,
)
```

```
db, err := sql.Open("gaussdb", dsn)
if err != nil {
 panic(err)
}
defer db.Close()
err = db.Ping()
if err != nil {
 panic(err)
}
fmt.Println("connect success.")
// Start a transaction.
tx, err := db.Begin()
if err != nil {
 log.Fatal(err)
 return
}
// Initialize a data table.
_, err = tx.Exec("drop table if exists employee;")
if err != nil {
 fmt.Println("drop table employee failed, err:", err)
 err = tx.Rollback() // On error, roll back the transaction.
 return
}
fmt.Println("drop table employee success.")
_, err = tx.Exec("create table employee (id SERIAL PRIMARY KEY, name varchar(20), age int, created_at
TIMESTAMP DEFAULT CURRENT_TIMESTAMP);")
if err != nil {
 fmt.Println("create table employee failed, err:", err)
 err = tx.Rollback()
 return
}
fmt.Println("create table employee success.")
// Insert data in batches.
employee := []struct {
 Name string
 Age  uint8
}{{Name: "zhangsan", Age: 21}, {Name: "lisi", Age: 22}, {Name: "zhaowu", Age: 23}}

batchSql := "INSERT INTO employee (name, age) VALUES "
vals := []interface{}{}

placeholders := "(?, ?)"
for _, u := range employee {
 vals = append(vals, u.Name, u.Age)
}

stmt := batchSql + placeholders
_, err = tx.Exec(stmt, vals...)

if err != nil {
 fmt.Println("batch insert into table employee failed, err:", err)
 err = tx.Rollback()
 return
}
fmt.Println("batch insert into table employee success.")
// Commit the transaction.
err = tx.Commit()
if err != nil {
 fmt.Println("commit failed, err:", err)
 err = tx.Rollback()
 log.Fatal(err)
 return
}
fmt.Println("commit success.")
// Obtain column information in the result set.
rows, err := db.Query("SELECT id, name, created_at FROM employee;")
if err != nil {
 fmt.Println("query table employee failed, err:", err)
 return
}
columns, err := rows.Columns()
if err != nil {
 fmt.Println("get query rows columns failed, err:", err)
 return
}
fmt.Println("Column count: ", len(columns))
fmt.Println("Column name list: ", columns)
fmt.Println("--------------------------")
// Obtain column types.
columnTypes, err := rows.ColumnTypes()
if err != nil {
 fmt.Println("get query rows ColumnTypes failed, err:", err)
 return
}

for _, ct := range columnTypes {
 fmt.Println("Column name: ", ct.Name())
 fmt.Println("Database type: ", ct.DatabaseTypeName())
 length, ok := ct.Length()
 if ok {
  fmt.Println("Length: ", length)
 }
 precision, scale, ok := ct.DecimalSize()
 if ok {
  fmt.Printf("Precision/Scale: %d/%d\n", precision, scale)
 }
 nullable, ok := ct.Nullable()
 if ok {
  fmt.Println("Nullable: ", nullable)
 }
 fmt.Println("Go type: ", ct.ScanType())
 fmt.Println("-----")
}
}
```

Result Verification

Below are the execution results for **Complete Example**:

```
connect success.
drop table employee success.
create table employee success.
batch insert into table employee success.
commit success.
Column count: 3
Column name list: [id name created_at]
--------------------------
Column name: id
Database type: INT4
Go type: int32
-----
Column name: name
Database type: VARCHAR
Length: 20
Go type: string
-----
Column name: created_at
Database type: TIMESTAMP
Go type: time.Time
-----
```

The following tasks have been completed as expected:

1. Concatenate the database connection string, create a connection object using sql.Open, and verify the connection status through the db.Ping() method.
2. Start a transaction and initialize the test table **employee** in the transaction.

3. Construct employee test data in the transaction, generate an SQL statement for batch insertion, bind parameters through the Exec API provided by the transaction object, and send packets to the database to execute the SQL statement.
4. After a successful batch insertion, call the Commit API to commit the transaction. (If the insertion fails, call the Rollback API to roll back the transaction.)
5. Call the Query API provided by the database connection object **"db"** to query the batch insertion results. Call the Columns API provided by the result object **"rows"** to obtain the list of all column names. Call the ColumnTypes API provided by the result object **"rows"** to obtain metadata about columns in the result set.

Rollback Method

To roll back operations within a specific transaction, call the Rollback API of the transaction object.

15.2.5 Typical Issues

1. SQL injection risks during the construction of batch insertion statements

   Placeholders and parameter binding are used rather than directly concatenating user-specified values. When inserting data, multi-line VALUES statements use placeholders (?) and a parameter list to prevent injection attacks. All dynamic data must be passed as parameters. For example, the Prepare or Exec API of the database object *DB can be used to pass variable parameter forms.
2. Batch insertion failure

   If a record fails to be inserted during batch insertion, the database returns only general error information (such as primary key conflict, foreign key constraint violated, or data type mismatch). However, it does not indicate which specific record is causing the error. If an SQL statement contains multiple records and one of them fails to be inserted, the entire transaction may fail (unless the error ignoring mechanism is enabled). To pinpoint the specific row causing the error, it is common practice to divide the batch into smaller batches or insert data row by row for better error capture.
3. Increased memory usage

   When large data sizes are involved, constructing SQL statements for batch insertion can significantly increase memory usage. This is particularly noticeable when you construct SQL statements through string concatenation, as it can lead to a sharp rise in memory consumption. Large-size batch processing may exceed the maximum SQL length limit of the database or Go driver, or trigger other parameter restrictions, potentially leading to errors or performance issues.

16 Best Practices for Index Design

16.1 Best Practices for Index Design (Distributed Instances)

16.1.1 Scenario Overview

Usage Scenarios

When you query large tables with over a million data records, not having indexes or having overly simple indexes can lead to slow query performance. To enhance efficiency, it is advisable to create appropriate indexes. This best practice primarily compares the performance of querying such a large table with indexes to one without indexes, as well as evaluates the effectiveness of single-column indexes versus composite indexes.

Requirements and Objectives

Create appropriate indexes to prevent full table scans and enhance query efficiency.

16.1.2 Preparations

Create a table in the current database and insert data at the million-row scale.

Machine configurations: 8-core CPU; 32 GB of memory

```
-- Create the test_table table.
gaussdb=# CREATE TABLE test_table ( id SERIAL PRIMARY KEY,name VARCHAR(100),email
VARCHAR(100),created_at TIMESTAMP);
CREATE TABLE

-- Insert a million data records into the table.
gaussdb=# INSERT INTO test_table (name,email,created_at) select 'User_' || i,'User_' || i ||
'@example.com',NOW() - (i * INTERVAL '1 minute') FROM generate_series(1, 1000000) AS i;
INSERT 0 1000000

-- Create another table.
gaussdb=# CREATE TABLE sales_records (record_id BIGSERIAL PRIMARY KEY,region_id INT NOT
NULL,store_id INT NOT NULL,product_id INT NOT NULL,sale_date DATE NOT NULL,amount DECIMAL(12,2)
NOT NULL,is_refund BOOLEAN DEFAULT false);

-- Insert 2 million data records.
gaussdb=# INSERT INTO sales_records (region_id, store_id, product_id, sale_date, amount) SELECT
(random()*9)::INT + 1,(random()*99)::INT + 1,(random()*499)::INT + 1,current_date - (random()*1095)::INT,
(random()*9900)::DECIMAL + 100 FROM generate_series(1,2000000);
INSERT 0 2000000
```

16.1.3 Procedure

Performance Comparison Between Tables with and Without Indexes

Step 1 Log in to the database as the root user.

Step 2 Check the execution plan of the **test_table** table.

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM test_table WHERE email = 'user_500000@example.com';
 id |          operation           | A-time  | A-rows | E-rows | Peak Memory | A-width | E-width |      E-cost
s
----+----------------------------+---------+--------+--------+-------------+---------+---------+-----------
-------
  1 | ->  Seq Scan on test_table | 382.457 |      0 |   1989 | 19KB        |         |     148 | 0.000..136
44.650
(1 row)

          Predicate Information (identified by plan id)
-----------------------------------------------------------------
   1 --Seq Scan on test_table
         Filter: ((email)::text = 'user_500000@example.com'::text)
         Rows Removed by Filter: 1000000
(3 rows)

       ====== Query Summary =====
---------------------------------------
 Datanode executor start time: 0.037 ms
 Datanode executor run time: 382.544 ms
 Datanode executor end time: 0.017 ms
 Planner runtime: 0.391 ms
 Query Id: 1945836514001883020
 Total runtime: 382.624 ms
(6 rows)
```

The execution plan indicates that the query needs 382.624 ms (full table scan).

Step 3 Create an index.

```
gaussdb=# CREATE INDEX idx_test_table_email ON test_table(email);
CREATE INDEX
```

Step 4 Check the execution plan of the **test_table** table again.

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM test_table WHERE email = 'user_500000@example.com';
 id |                         operation                          | A-time | A-rows | E-rows | Peak Memory | A-
width | E-width |   E-costs
----+----------------------------------------------------------+--------+--------+--------+-------------+---
------+---------+--------------
  1 | ->  Index Scan using idx_test_table_email on test_table | 0.163  |      0 |      1 | 75KB        |
      |      46 | 0.000..8.268
(1 row)

            Predicate Information (identified by plan id)
---------------------------------------------------------------------
   1 --Index Scan using idx_test_table_email on test_table
         Index Cond: ((email)::text = 'user_500000@example.com'::text)
(2 rows)

       ====== Query Summary =====
---------------------------------------
```

```
 Datanode executor start time: 0.063 ms
 Datanode executor run time: 0.190 ms
 Datanode executor end time: 0.013 ms
 Planner runtime: 0.936 ms
 Query Id: 1945836514001885197
 Total runtime: 0.293 ms
(6 rows)

 ====== Query Others =====
---------------------------
 Bypass: Yes
(1 row)
```

The query time decreases from 382.624 ms (without indexes) to 0.293 ms (with an index).

----End

Performance Comparison Between Single-Column Indexes and Composite Indexes

Step 1 Log in to the database as the root user.

Step 2 Create single-column indexes.

```
gaussdb=# CREATE INDEX idx_region ON sales_records(region_id);
CREATE INDEX
gaussdb=# CREATE INDEX idx_store ON sales_records(store_id);
CREATE INDEX
```

Step 3 Check the execution plan.

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM sales_records WHERE region_id = 5 AND store_id = 42;
                                          QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Data Node Scan  (cost=0.00..0.00 rows=0 width=0) (actual time=23.482..37.694 rows=2221 loops=1)
   Node/s: All datanodes
 Total runtime: 37.945 ms
(3 rows)
```

The execution plan indicates that the query needs 37.945 ms.

Step 4 Create a composite index.

```
gaussdb=# CREATE INDEX idx_region_store ON sales_records(region_id, store_id);
CREATE INDEX
```

Step 5 Check the execution plan again.

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM sales_records WHERE region_id = 5 AND store_id = 42;
                                         QUERY PLAN
---------------------------------------------------------------------------------------------------
 Data Node Scan  (cost=0.00..0.00 rows=0 width=0) (actual time=4.266..6.390 rows=2221 loops=1)
   Node/s: All datanodes
 Total runtime: 6.616 ms
(3 rows)
```

The query time decreases from 37.945 ms (with single-column indexes) to 6.616 ms (with a composite index).

----End

16.2 Best Practices for Index Design (Centralized Instances)

16.2.1 Scenario Overview

Usage Scenarios

When you query large tables with over a million data records, not having indexes or having overly simple indexes can lead to slow query performance. To enhance efficiency, it is advisable to create appropriate indexes. This best practice primarily compares the performance of querying such a large table with indexes to one without indexes, as well as evaluates the effectiveness of single-column indexes versus composite indexes.

Requirements and Objectives

Create appropriate indexes to prevent full table scans and enhance query efficiency.

16.2.2 Preparations

Create a table in the current database and insert data at the million-row scale.

Machine configurations: 8-core CPU; 32 GB of memory

```
-- Create the test_table table.
gaussdb=# CREATE TABLE test_table ( id SERIAL PRIMARY KEY,name VARCHAR(100),email
VARCHAR(100),created_at TIMESTAMP);
CREATE TABLE

-- Insert a million data records into the table.
gaussdb=# INSERT INTO test_table (name,email,created_at) select 'User_' || i,'User_' || i ||
'@example.com',NOW() - (i * INTERVAL '1 minute') FROM generate_series(1, 1000000) AS i;
INSERT 0 1000000

-- Create another table.
gaussdb=# CREATE TABLE sales_records (record_id BIGSERIAL PRIMARY KEY,region_id INT NOT
NULL,store_id INT NOT NULL,product_id INT NOT NULL,sale_date DATE NOT NULL,amount DECIMAL(12,2)
NOT NULL,is_refund BOOLEAN DEFAULT false);

-- Insert 2 million data records.
gaussdb=# INSERT INTO sales_records (region_id, store_id, product_id, sale_date, amount) SELECT
(random()*9)::INT + 1,(random()*99)::INT + 1,(random()*499)::INT + 1,current_date - (random()*1095)::INT,
(random()*9900)::DECIMAL + 100 FROM generate_series(1,2000000);
INSERT 0 2000000
```

16.2.3 Procedure

Performance Comparison Between Tables with and Without Indexes

Step 1 Log in to the database as the **root** user.

Step 2 Check the execution plan of the **test_table** table.

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM test_table WHERE email = 'user_500000@example.com';
 id |         operation          | A-time  | A-rows | E-rows | Peak Memory | A-width | E-width |      E-cost
s
----+----------------------------+---------+--------+--------+-------------+---------+---------+-----------
-------
  1 | ->  Seq Scan on test_table | 382.457 |      0 |   1989 | 19KB        |         |     148 | 0.000..136
44.650
(1 row)

          Predicate Information (identified by plan id)
```

```
-----------------------------------------------------------------------
   1 --Seq Scan on test_table
         Filter: ((email)::text = 'user_500000@example.com'::text)
         Rows Removed by Filter: 1000000
(3 rows)

       ====== Query Summary =====
-----------------------------------------
 Datanode executor start time: 0.037 ms
 Datanode executor run time: 382.544 ms
 Datanode executor end time: 0.017 ms
 Planner runtime: 0.391 ms
 Query Id: 1945836514001883020
 Total runtime: 382.624 ms
(6 rows)
```

The execution plan indicates that the query needs 382.624 ms (full table scan).

Step 3 Create an index.

```
gaussdb=# CREATE INDEX idx_test_table_email ON test_table(email);
CREATE INDEX
```

Step 4 Check the execution plan of the **test_table** table again.

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM test_table WHERE email = 'user_500000@example.com';
 id |                          operation                          | A-time | A-rows | E-rows | Peak Memory | A-
width | E-width |    E-costs
----+---------------------------------------------------------+--------+--------+--------+-------------+---
------+---------+--------------
  1 | ->  Index Scan using idx_test_table_email on test_table | 0.163  |      0 |      1 | 75KB        |
     |      46 | 0.000..8.268
(1 row)

            Predicate Information (identified by plan id)
-----------------------------------------------------------------------
   1 --Index Scan using idx_test_table_email on test_table
         Index Cond: ((email)::text = 'user_500000@example.com'::text)
(2 rows)

       ====== Query Summary =====
-----------------------------------------
 Datanode executor start time: 0.063 ms
 Datanode executor run time: 0.190 ms
 Datanode executor end time: 0.013 ms
 Planner runtime: 0.936 ms
 Query Id: 1945836514001885197
 Total runtime: 0.293 ms
(6 rows)

 ====== Query Others =====
---------------------------
 Bypass: Yes
(1 row)
```

The query time decreases from 382.624 ms (without indexes) to 0.293 ms (with an index).

----End

Performance Comparison Between Single-Column Indexes and Composite Indexes

Step 1 Log in to the database as the **root** user.

Step 2 Create single-column indexes.

```
gaussdb=# CREATE INDEX idx_region ON sales_records(region_id);
CREATE INDEX
```

```
gaussdb=# CREATE INDEX idx_store ON sales_records(store_id);
CREATE INDEX
```

Step 3 Check the execution plan.

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM sales_records WHERE region_id = 5 AND store_id = 42;
                                        QUERY PLAN
-----------------------------------------------------------------------------------------------
 Data Node Scan  (cost=0.00..0.00 rows=0 width=0) (actual time=23.482..37.694 rows=2221 loops=1)
   Node/s: All datanodes
 Total runtime: 37.945 ms
(3 rows)
```

The execution plan indicates that the query needs 37.945 ms.

Step 4 Create a composite index.

```
gaussdb=# CREATE INDEX idx_region_store ON sales_records(region_id, store_id);
CREATE INDEX
```

Step 5 Check the execution plan again.

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM sales_records WHERE region_id = 5 AND store_id = 42;
                                       QUERY PLAN
---------------------------------------------------------------------------------------------
 Data Node Scan  (cost=0.00..0.00 rows=0 width=0) (actual time=4.266..6.390 rows=2221 loops=1)
   Node/s: All datanodes
 Total runtime: 6.616 ms
(3 rows)
```

The query time decreases from 37.945 ms (with single-column indexes) to 6.616 ms (with a composite index).

----End

17 Best Practices for Table Design

17.1 Best Practices for Table Design (Distributed Instances)

17.1.1 Scenario Overview

Usage Scenarios

The following scenarios need to be considered during table design:

- The query efficiency needs to be enhanced through table structure design.
- The maintenance efficiency needs to be enhanced in massive data cases.
- Table designs must be able to handle frequent data updates.
- The best trade-off must be found between storage costs and query performance.

17.1.2 Architecture Principles

Core Principles

- Data type optimization: Integer > Floating-point number > Numeric (in order of priority).
- Index balancing mechanism: trade-off between query acceleration and update costs.
- Partition storage policy: logically unified + physically dispersed.
- Storage engine features: Ustore supports in-place update, and Astore supports append-only.

Solution Advantages

- Improved query performance: Index design and partitioning policies help reduce the scanned data size.

- Reduced storage costs: Selecting appropriate data types can save more than 30% of space.
- Enhanced maintenance efficiency: Independent partition maintenance minimizes impact on services.
- Enhanced concurrent processing: Concurrent access to multiple partitions improves the service throughput.

17.1.3 Preparations

- Confirm the service scenario characteristics.
  - Data size estimation (consider using partitioned tables for large data sizes)
  - Read/write ratio analysis (design the storage engine and indexes based on this ratio)
- Check the environment.

  Check the **track_counts** and **track_activities** parameters.
- Prepare tools.
  - Database client tool
  - Performance monitoring tool

17.1.4 Procedure

Table design involves distribution mode and key design, data type design, partitioning policies, constraint configuration, index design, and storage parameter optimization.

Distribution Mode and Key Design

- Adhere to the following principles when selecting a table distribution mode.

Distribution Mode	Description	Applicable Scenario
Hash	Table data is distributed across all DNs in the cluster by hash.	Tables with a large data size
Replication	Each DN in the cluster holds a complete set of table data.	Dimension tables and tables with a small data size
Range	Table data is distributed to related DNs by mapping the specified columns based on a specific range.	Custom distribution rules
List	Table data is distributed to related DNs by mapping the specified columns based on specific values.	Custom distribution rules

- Distribution key selection

  When working with a hash table, selecting the right distribution key is essential. An inappropriate distribution key can result in data skew, causing heavy I/O loads on specific DNs and impacting overall query performance. Therefore, after determining the distribution policy of a distributed table, you need to check the table data skew to ensure that data is evenly distributed. When selecting a distribution key, adhere to the following principles:

- Use a column with discrete data as the distribution key to evenly distribute data across all DNs. If a column lacks sufficient discreteness, consider using multiple columns as distribution keys. The primary key of a table can also serve as a distribution key, such as the ID number column in an employee information table.
- When the first principle is satisfied, avoid selecting a column with constant filter conditions as the distribution key.
- When both the first and second principles are satisfied, use the join conditions in queries as distribution keys. This will result in data from join tasks being distributed on the local DN, significantly reducing data flow costs among DNs.

Below is simple example that shows how to design a distribution mode and key using syntax.

Step 1 Log in to the database as the root user.

Step 2 Create a table and select a distribution key and mode.

```
-- Replication distribution
gaussdb=#CREATE TABLE tb_t1(c1 int, c2 int)DISTRIBUTE BY REPLICATION;

-- Hash distribution
gaussdb=#CREATE TABLE tb_t2(c1 int,c2 int)DISTRIBUTE BY HASH(c1);

-- Range distribution
gaussdb=#CREATE TABLE tb_t3(c1 int,c2 int)
DISTRIBUTE BY RANGE(c1)(
    SLICE s1 VALUES LESS THAN (100),
    SLICE s2 VALUES LESS THAN (200),
    SLICE s3 VALUES LESS THAN (MAXVALUE)
);
gaussdb=#CREATE TABLE tb_t4(c1 int,c2 int)
DISTRIBUTE BY RANGE(c1)(
    SLICE s1 START (1) END (100),
    SLICE s2 START (100) END (200),
    SLICE s3 START (200) END (MAXVALUE)
);

-- List distribution
gaussdb=#CREATE TABLE tb_t5(id INT,name VARCHAR(20),country VARCHAR(30))
DISTRIBUTE BY LIST(country)(
    SLICE s1 VALUES ('China'),
    SLICE s2 VALUES ('USA'),
    SLICE s3 VALUES (DEFAULT)
);

-- Drop the created table objects.
gaussdb=#DROP TABLE tb_t1,tb_t2,tb_t3,tb_t4,tb_t5;
```

----End

Data Type Design

To improve query efficiency, adhere to the following principles when designing data types:

- Select efficient data types in the following order of priority: Integer > Floating-point number > Numeric, provided that they all meet the required service precision.
- In tables that are logically related, columns with the same meaning should use the same data type.
- When dealing with string data, it is essential to choose between fixed-length or variable-length character types based on the specific situation. Data types like varchar and char require specifying a maximum length. This length must be sufficient to store all potential data while also considering storage space to prevent resource wastage.

When designing a specific column, select a data type that matches its data characteristics. For details about the data types supported by GaussDB, see "SQL Reference > Data Types" in *Developer Guide*.

Partitioning Policies

- **Overview**

  Partitioning is a database optimization technology that divides a large table into multiple partitions based on specific rules to enhance query and maintenance efficiency. The partitioned table functions as a logical table that does not store data directly. Instead, data is stored within these partitions and can be distributed across different storage devices. GaussDB currently supports range partitioning, hash partitioning, and list partitioning. Here are the advantages and disadvantages of using partitioned tables:

  - Advantages:
    - Improved query performance: Reducing the scanned data size significantly enhances query performance.
    - Optimized storage: Distributing partitions across various storage media helps balance performance and costs.
    - Improved maintainability: Maintenance operations such as data cleanup and index rebuild for partitioned tables can be carried out at the partition level, minimizing the impact on the overall system.
    - Improved concurrency: Partitioned tables enable parallel processing of multiple partitions, resulting in improved concurrency. For instance, multiple queries can access different partitions simultaneously without causing interference.
  - Disadvantages:
    - Memory usage: A partitioned table typically consumes around (Number of partitions × 3/1,024) MB of memory. If there are too many partitions causing memory shortages, performance may decline significantly.
    - Complexity of partitioning policies: Technical knowledge and experience are required to develop and implement appropriate partitioning policies. Selecting an inappropriate partitioning policy can lead to uneven data distribution, thereby impacting performance.

- Complexity of backup and restoration: While it is possible to back up and restore partitions individually, this also implies the need for more detailed backup policies and management efforts.

- **Usage scenarios**
  - High query performance: When a table contains a significant amount of data and certain data features are frequently accessed in a particular scenario, you can reduce the scanned data size during queries to enhance query performance. This is particularly useful for tables that are regularly analyzed on a monthly, quarterly, or yearly basis.
  - Balance between performance and costs: When a table contains a significant amount of data, it is advisable to store cold data (infrequently accessed data) on low-cost storage, while keeping hot data (frequently accessed data) on high-performance storage.
  - Large table management: Tables containing a significant amount of data may need to be stored across multiple storage media.
- **Precautions about design**
  - Selection of partition keys:

    Selecting partition keys for partitioned tables is a critical design decision as it directly affects the performance, maintainability, and data management efficiency of the database.
    - Query optimization: Specify frequently queried columns as partition keys. For instance, if a table is often queried by date, it is advisable to select the date column as the partition key.
    - Data distribution: Consider the distribution of data when selecting partition keys. This helps prevent situations where some partitions store an excessive amount of data while others store only a small amount.
    - Partition quantity and management: Limit the number of partitions. Creating excessive partitions can lead to increased management complexity and performance decline.
  - Selection of partition types:
    - Range partitioning: This type is ideal for partition keys with consecutive values, such as time.
    - List partitioning: This type is suitable for partition keys with discrete values that fall into a limited number of categories, such as regions or status codes.
    - Hash partitioning: This type is designed for evenly distributed data, such as user IDs.

Below is simple example that shows how to design a partitioning policy using syntax.

Step 1 Log in to the database as the root user.

Step 2 Create partitioned tables.

```
-- Range partitioned table
gaussdb=#CREATE TABLE tb_t1(id INT,info VARCHAR(20))
```

```
PARTITION BY RANGE (id) (
    PARTITION p1 START(1) END(600) EVERY(200),
    PARTITION p2 START(600) END(800),
    PARTITION pmax START(800) END(MAXVALUE)
);
gaussdb=#CREATE TABLE tb_t2(
    id INT,
    info VARCHAR(20)
) PARTITION BY RANGE (id) (
    PARTITION p1 VALUES LESS THAN (100),
    PARTITION p2 VALUES LESS THAN (200),
    PARTITION p3 VALUES LESS THAN (MAXVALUE)
);

-- List partitioned table
gaussdb=#CREATE TABLE tb_t3(NAME VARCHAR ( 50 ), area VARCHAR ( 50 ))
PARTITION BY LIST (area) (
    PARTITION p1 VALUES ('bj'),
    PARTITION p2 VALUES ('sh'),
    PARTITION pdefault VALUES (DEFAULT)
);


-- Hash partitioned table
gaussdb=#CREATE TABLE tb_t4(c1 int) PARTITION BY HASH(c1) PARTITIONS 3;
gaussdb=#CREATE TABLE tb_t5(c1 int) PARTITION BY HASH(C1)(
    PARTITION pa,
    PARTITION pb,
    PARTITION pc
);

-- Drop the created table objects.
gaussdb=#DROP TABLE tb_t1,tb_t2,tb_t3,tb_t4,tb_t5;
```

----End

Constraint Configuration

- When creating a constraint, clearly indicate the type of constraint and the table name where the constraint is applied in the constraint name. For example, name the PRIMARY KEY constraint with **PK**, the table name, and the column names that make up the key.
- Exercise caution when selecting the DEFAULT constraint. If column values can be completed at the service level, it is not recommended to use the DEFAULT constraint.
- If the NOT NULL constraint is applied to columns that are meant to always contain non-null values, the optimizer will conduct automatic optimization in certain scenarios.

Below is a simple example that shows how to add constraints using syntax.

Step 1 Log in to the database as the root user.

Step 2 Create a table and add constraints to it.

```
-- NOT NULL constraint
gaussdb=#CREATE TABLE tb_t1(id int not null,name varchar(50));

-- UNIQUE constraint
gaussdb=#CREATE TABLE tb_t2(id int UNIQUE,name varchar(50));
gaussdb=#CREATE TABLE tb_t3(id int, name varchar(50),CONSTRAINT unq_t3_id UNIQUE(id));

-- PRIMARY KEY constraint
gaussdb=#CREATE TABLE tb_t4(id int PRIMARY KEY, name varchar(50));
gaussdb=#CREATE TABLE tb_t5(
    id int,
    name varchar(50),
    CONSTRAINT pk_person5_id PRIMARY KEY(id)
);

-- CHECK constraint
gaussdb=#CREATE TABLE tb_t6(name varchar(50),age int CHECK(age > 0 AND age < 200));
gaussdb=#CREATE TABLE tb_t7(
    name varchar(50),
    age int,
    CONSTRAINT chk_t6_age CHECK (age > 0 AND age < 200)
);

-- Drop the created table objects.
gaussdb=#DROP TABLE tb_t1,tb_t2,tb_t3,tb_t4,tb_t5,tb_t6,tb_t7;
```

----End

Index Design

Using indexes helps accelerate data access but can also prolong the time needed to insert, update, or delete data. Therefore, it is crucial to carefully evaluate whether to add indexes to a table and which specific columns to index. It is advisable to adhere to the following principles when setting up indexes:

- Creating indexes for frequently joined columns can enhance join speed.
- Creating indexes for frequently sorted columns can enhance sorting and query speed since indexes are already sorted.
- Creating indexes for columns frequently used in the WHERE clause can enhance the speed of condition judgment.
- A composite index consists of multiple columns. However, including more columns will result in a larger index size and higher maintenance costs.
- Do not apply indexes to frequently updated columns as they can increase the maintenance costs of data updates.
- All functions and operators used in an index definition must be "immutable", that is, their results must depend only on their input parameters and never on any outside influence (such as the contents of another table or the current time). This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index or WHERE clause, remember to mark the function immutable when you create it.
- There are two types of indexes for partitioned tables: local index and global index. A local index is specific to a partition within a partitioned table, while a global index spans the entire partitioned table.
- It is crucial to regularly maintain indexes, and there are several scenarios in which to use REINDEX:
  - An index has become corrupted, and no longer contains valid data.
  - An index has become "bloated", that is, it contains many empty or nearly-empty pages.
  - You have altered a storage parameter (such as fill factor) for an index, and wish to ensure that the change has taken full effect.
  - An index build with the CONCURRENTLY option failed, leaving an "invalid" index.

- When naming an index, make sure to include the table name and the key columns involved. For example, **idx_test_c1** indicates that the index is created on the **c1** column of the **test** table.

Below is a simple example that shows how to add an index to a table using syntax.

Step 1 Log in to the database as the root user.

Step 2 Create a table and add an index to it.

```
gaussdb=#CREATE TABLE tb_t1(id int not null,name varchar(50));

-- Add an index to the table.
gaussdb=#CREATE INDEX idx_t1_id ON tb_t1(id);

-- Drop the created table object.
gaussdb=#DROP TABLE tb_t1;
```

----End

Storage Parameter Optimization

- Fill factor

  The fill factor for a table is a percentage between 10 and 100. If Ustore is in use, the default value is **92**. If Astore is in use, the default value is **100** (complete packing). When you specify a smaller fillfactor, INSERT operations pack table pages only to the indicated percentage. The remaining space on each page is reserved for updating rows on that page. This gives UPDATE a chance to place the updated copy of a row on the same page as the original, which is more efficient than placing it on a different page. For tables that are never updated, it is best to use a fill factor of **100**. However, for heavily updated tables, smaller fill factors are appropriate. Below is an example:

  ```
  CREATE TABLE test(c1 int,c2 int) WITH (FILLFACTOR = 80);
  ```

- Storage engine

  Specifies the storage engine type. Once set, this parameter cannot be modified. Below is an example:

  ```
  CREATE TABLE test(c1 int,c2 int) WITH (STORAGE_TYPE = USTORE);
  ```

  - **USTORE**: The table uses an in-place update storage engine. To prevent space expansion, be sure to enable the **track_counts** and **track_activities** parameters.
  - **ASTORE**: The table uses an append-only storage engine.

17.2 Best Practices for Table Design (Centralized Instances)

17.2.1 Scenario Overview

Usage Scenarios

The following scenarios need to be considered during table design:

- The query efficiency needs to be enhanced through table structure design.
- The maintenance efficiency needs to be enhanced in massive data cases.
- Table designs must be able to handle frequent data updates.
- The best trade-off must be found between storage costs and query performance.

17.2.2 Architecture Principles

Core Principles

- Data type optimization: Integer > Floating-point number > Numeric (in order of priority).
- Index balancing mechanism: trade-off between query acceleration and update costs.
- Partition storage policy: logically unified + physically dispersed.
- Storage engine features: Ustore supports in-place update, and Astore supports append-only.

Solution Advantages

- Improved query performance: Index design and partitioning policies help reduce the scanned data size.
- Reduced storage costs: Selecting appropriate data types can save more than 30% of space.
- Enhanced maintenance efficiency: Independent partition maintenance minimizes impact on services.
- Enhanced concurrent processing: Concurrent access to multiple partitions improves the throughput.

17.2.3 Preparations

- Confirm the service scenario characteristics.
  - Data size estimation (consider using partitioned tables for large data sizes)
  - Read/write ratio analysis (design the storage engine and indexes based on this ratio)
- Check the environment.

  Check the **track_counts** and **track_activities** parameters.
- Prepare tools.
  - Database client tool
  - Performance monitoring tool

17.2.4 Procedure

Table design involves data type design, partitioning policies, constraint configuration, index design, and storage parameter optimization.

Data Type Design

To improve query efficiency, adhere to the following principles when designing data types:

- Select efficient data types in the following order of priority: Integer > Floating-point number > Numeric, provided that they all meet the required service precision.
- In tables that are logically related, columns with the same meaning should use the same data type.
- When dealing with string data, it is essential to choose between fixed-length or variable-length character types based on the specific situation. Data types like varchar and char require specifying a maximum length. This length must be sufficient to store all potential data while also considering storage space to prevent resource wastage.

When designing a specific column, select a data type that matches its data characteristics. For details about the data types supported by GaussDB, see "SQL Reference > Data Types" in *Developer Guide*.

Partitioning Policies

- **Overview**

  Partitioning is a database optimization technology that divides a large table into multiple partitions based on specific rules to enhance query and maintenance efficiency. The partitioned table functions as a logical table that does not store data directly. Instead, data is stored within these partitions and can be distributed across different storage devices. GaussDB currently supports range partitioning, interval partitioning, list partitioning, and hash partitioning. Here are the advantages and disadvantages of using partitioned tables:

  - Advantages:
    - Improved query performance: Reducing the scanned data size significantly enhances query performance.
    - Optimized storage: Distributing partitions across various storage media helps balance performance and costs.
    - Improved maintainability: Maintenance operations such as data cleanup and index rebuild for partitioned tables can be carried out at the partition level, minimizing the impact on the overall system.
    - Improved concurrency: Partitioned tables enable parallel processing of multiple partitions, resulting in improved concurrency. For instance, multiple queries can access different partitions simultaneously without causing interference.
  - Disadvantages:
    - Memory usage: A partitioned table typically consumes around (Number of partitions × 3/1,024) MB of memory. If there are too many partitions causing memory shortages, performance may decline significantly.

        - Complexity of partitioning policies: Technical knowledge and experience are required to develop and implement appropriate partitioning policies. Selecting an inappropriate partitioning policy can lead to uneven data distribution, thereby impacting performance.
        - Complexity of backup and restoration: While it is possible to back up and restore partitions individually, this also implies the need for more detailed backup policies and management efforts.
- **Usage scenarios**
    - High query performance: When a table contains a significant amount of data and certain data features are frequently accessed in a particular scenario, you can reduce the scanned data size during queries to enhance query performance. This is particularly useful for tables that are regularly analyzed on a monthly, quarterly, or yearly basis.
    - Balance between performance and costs: When a table contains a significant amount of data, it is advisable to store cold data (infrequently accessed data) on low-cost storage, while keeping hot data (frequently accessed data) on high-performance storage.
    - Large table management: Tables containing a significant amount of data may need to be stored across multiple storage media.
- **Precautions about design**
    - Selection of partition keys:

      Selecting partition keys for partitioned tables is a critical design decision as it directly affects the performance, maintainability, and data management efficiency of the database.
        - Query optimization: Specify frequently queried columns as partition keys. For instance, if a table is often queried by date, it is advisable to select the date column as the partition key.
        - Data distribution: Consider the distribution of data when selecting partition keys. This helps prevent situations where some partitions store an excessive amount of data while others store only a small amount.
        - Partition quantity and management: Limit the number of partitions. Creating excessive partitions can lead to increased management complexity and performance decline.
    - Selection of partition types:
        - Range partitioning: This type is ideal for partition keys with consecutive values, such as time.
        - Interval partitioning: This is a unique variation of range partitioned tables. In contrast to regular range partitioned tables, interval partitioned tables introduce an additional element—the interval value. If a newly inserted record does not match any existing partition, a new partition will be automatically created based on the interval value.
        - List partitioning: This type is suitable for partition keys with discrete values that fall into a limited number of categories, such as regions or status codes.

- Hash partitioning: This type is designed for evenly distributed data, such as user IDs.

Below is simple example that shows how to design a partitioning policy using syntax.

Step 1 Log in to the database as the root user.

Step 2 Create partitioned tables.

```
-- Range partitioned table
gaussdb=#CREATE TABLE tb_t1(id INT,info VARCHAR(20))
PARTITION BY RANGE (id) (
    PARTITION p1 START(1) END(600) EVERY(200),
    PARTITION p2 START(600) END(800),
    PARTITION pmax START(800) END(MAXVALUE)
);
gaussdb=#CREATE TABLE tb_t2(
    id INT,
    info VARCHAR(20)
) PARTITION BY RANGE (id) (
    PARTITION p1 VALUES LESS THAN (100),
    PARTITION p2 VALUES LESS THAN (200),
    PARTITION p3 VALUES LESS THAN (MAXVALUE)
);

-- List partitioned table
gaussdb=#CREATE TABLE tb_t3(NAME VARCHAR ( 50 ), area VARCHAR ( 50 ))
PARTITION BY LIST (area) (
    PARTITION p1 VALUES ('bj'),
    PARTITION p2 VALUES ('sh'),
    PARTITION pdefault VALUES (DEFAULT)
);


-- Hash partitioned table
gaussdb=#CREATE TABLE tb_t4(c1 int) PARTITION BY HASH(c1) PARTITIONS 3;
gaussdb=#CREATE TABLE tb_t5(c1 int) PARTITION BY HASH(C1)(
    PARTITION pa,
    PARTITION pb,
    PARTITION pc
);

-- Drop the created table objects.
gaussdb=#DROP TABLE tb_t1,tb_t2,tb_t3,tb_t4,tb_t5;
```

----End

Constraint Configuration

- When creating a constraint, clearly indicate the type of constraint and the table name where the constraint is applied in the constraint name. For example, name the PRIMARY KEY constraint with **PK**, the table name, and the column names that make up the key.
- Exercise caution when selecting the DEFAULT constraint. If column values can be completed at the service level, it is not recommended to use the DEFAULT constraint.
- If the NOT NULL constraint is applied to columns that are meant to always contain non-null values, the optimizer will conduct automatic optimization in certain scenarios.

Below is a simple example that shows how to add constraints using syntax.

Step 1 Log in to the database as the root user.

Step 2 Create a table and add constraints to it.

```
-- NOT NULL constraint
gaussdb=#CREATE TABLE tb_t1(id int not null,name varchar(50));

-- UNIQUE constraint
gaussdb=#CREATE TABLE tb_t2(id int UNIQUE,name varchar(50));
gaussdb=#CREATE TABLE tb_t3(id int, name varchar(50),CONSTRAINT unq_t3_id UNIQUE(id));

-- PRIMARY KEY constraint
gaussdb=#CREATE TABLE tb_t4(id int PRIMARY KEY, name varchar(50));
gaussdb=#CREATE TABLE tb_t5(
    id int,
    name varchar(50),
    CONSTRAINT pk_person5_id PRIMARY KEY(id)
);

-- CHECK constraint
gaussdb=#CREATE TABLE tb_t6(name varchar(50),age int CHECK(age > 0 AND age < 200));
gaussdb=#CREATE TABLE tb_t7(
    name varchar(50),
    age int,
    CONSTRAINT chk_t6_age CHECK (age > 0 AND age < 200)
);

-- Drop the created table objects.
gaussdb=#DROP TABLE tb_t1,tb_t2,tb_t3,tb_t4,tb_t5,tb_t6,tb_t7;
```

----End

Index Design

Using indexes helps accelerate data access but can also prolong the time needed to insert, update, or delete data. Therefore, it is crucial to carefully evaluate whether to add indexes to a table and which specific columns to index. It is advisable to adhere to the following principles when setting up indexes:

- Creating indexes for frequently joined columns can enhance join speed.
- Creating indexes for frequently sorted columns can enhance sorting and query speed since indexes are already sorted.
- Creating indexes for columns frequently used in the WHERE clause can enhance the speed of condition judgment.
- A composite index consists of multiple columns. However, including more columns will result in a larger index size and higher maintenance costs.
- Do not apply indexes to frequently updated columns as they can increase the maintenance costs of data updates.
- All functions and operators used in an index definition must be "immutable", that is, their results must depend only on their input parameters and never on any outside influence (such as the contents of another table or the current time). This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index or WHERE clause, remember to mark the function immutable when you create it.
- There are two types of indexes for partitioned tables: local index and global index. A local index is specific to a partition within a partitioned table, while a global index spans the entire partitioned table.
- It is crucial to regularly maintain indexes, and there are several scenarios in which to use REINDEX:

- An index has become corrupted, and no longer contains valid data.
- An index has become "bloated", that is, it contains many empty or nearly-empty pages.
- You have altered a storage parameter (such as fill factor) for an index, and wish to ensure that the change has taken full effect.
- An index build with the CONCURRENTLY option failed, leaving an "invalid" index.

- When naming an index, make sure to include the table name and the key columns involved. For example, **idx_test_c1** indicates that the index is created on the **c1** column of the **test** table.

Below is a simple example that shows how to add an index to a table using syntax.

Step 1 Log in to the database as the root user.

Step 2 Create a table and add an index to it.

```
gaussdb=#CREATE TABLE tb_t1(id int not null,name varchar(50));

-- Add an index to the table.
gaussdb=#CREATE INDEX idx_t1_id ON tb_t1(id);

-- Drop the created table object.
gaussdb=#DROP TABLE tb_t1;
```

----End

Storage Parameter Optimization

- Fill factor

  The fill factor for a table is a percentage between 10 and 100. If Ustore is in use, the default value is **92**. If Astore is in use, the default value is **100** (complete packing). When you specify a smaller fillfactor, INSERT operations pack table pages only to the indicated percentage. The remaining space on each page is reserved for updating rows on that page. This gives UPDATE a chance to place the updated copy of a row on the same page as the original, which is more efficient than placing it on a different page. For tables that are never updated, it is best to use a fill factor of **100**. However, for heavily updated tables, smaller fill factors are appropriate. Below is an example:

  ```
  CREATE TABLE test(c1 int,c2 int) WITH (FILLFACTOR = 80);
  ```

- Storage engine

  Specifies the storage engine type. Once set, this parameter cannot be modified. Below is an example:

  ```
  CREATE TABLE test(c1 int,c2 int) WITH (STORAGE_TYPE = USTORE);
  ```

  - **USTORE**: The table uses an in-place update storage engine. To prevent space expansion, be sure to enable the **track_counts** and **track_activities** parameters.
  - **ASTORE**: The table uses an append-only storage engine.