GaussDB

MySQL Compatibility(Centralized)

Issue	01
Date	2025-06-30

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

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 Overview

1.1 Overview of B-compatible Mode

B-compatible Mode compares GaussDB in B-compatible mode (that is, when **sql_compatibility** is set to **'B'**, **b_format_version** is set to **'5.7'**, and **b_format_dev_version** is set to **'s1'**) with MySQL 5.7. Only compatibility features added later than GaussDB Kernel 503.0.0 are described. You are advised to view the specifications and restrictions of the features in *Developer Guide*.

NOTE

Currently, the implementation logic of B-compatible mode (**sql_compatibility** set to **'B'**) is similar to that of the MYSQL-compatible mode (**sql_compatibility** set to **'MYSQL'**) in the distributed mode.

The B-compatible mode here matches the MYSQL-compatible mode in the distributed GaussDB. For details, see "Overview > Overview of MYSQL-compatible Mode" in the distributed *MySQL Compatibility Description*.

GaussDB is compatible with MySQL in terms of data types, SQL functions, and database objects.

GaussDB and MySQL implement different underlying frameworks. Therefore, there are still some differences between GaussDB and MySQL.

1.2 Overview of M-compatible Mode

M-compatible Mode compares GaussDB in M-compatible mode (**sql_compatibility** set to **'M'**) with MySQL 5.7. Only compatibility features added later than GaussDB Kernel 505.1.0 are described. You are advised to view the specifications and restrictions of the features in *M Compatibility Developer Guide*.

GaussDB is compatible with MySQL in terms of data types, SQL functions, and database objects.

The execution plan, optimization, and EXPLAIN result in GaussDB are different from those in MySQL.

GaussDB and MySQL implement different underlying frameworks. Therefore, there are still some differences between GaussDB and MySQL.

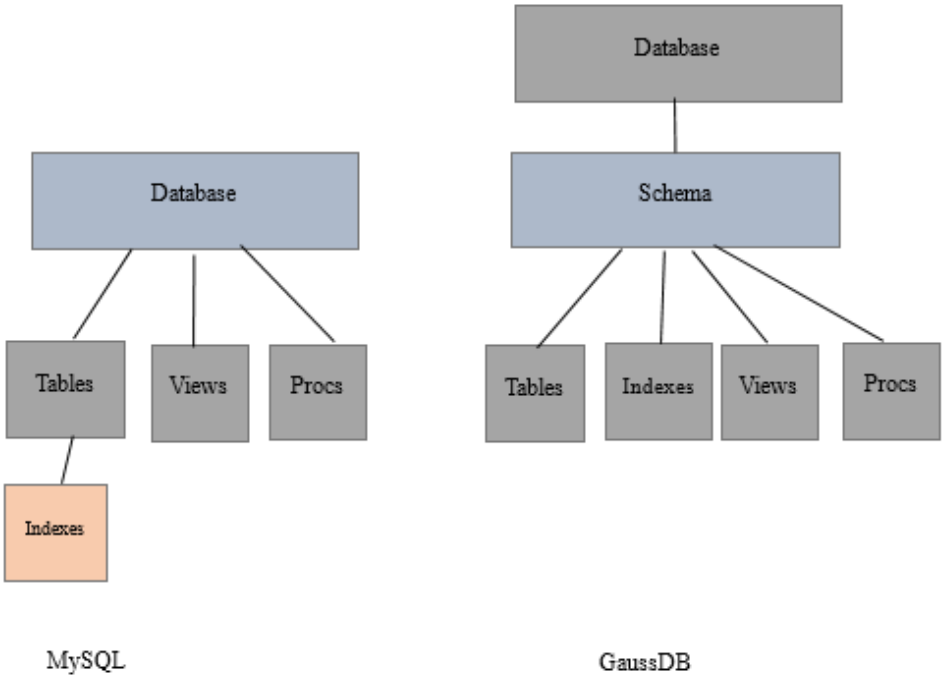
NOTE

- The M-compatible mode is recommended because it is more compatible with MySQL in terms of syntax, data types, metadata, and protocols. B-compatible mode is not recommended and will not evolve because it is not compatible with the MySQL architecture.
- The underlying architecture of GaussDB in M-compatible mode is different from that of MySQL. Therefore, the performance of querying the same schemas under information_schema and m_schema may be different from that in MySQL. For details, see "Schemas" in *M Compatibility Developer Guide*. For example, the execution of the count function cannot be optimized. The time consumed by the SELECT * and SELECT COUNT(*) statements is similar.

Database and Schema Design

MySQL data objects include database, table, index, view, trigger, and proc, mapping those in GaussDB hierarchically and maybe in a 1:N relationship, as shown in the following figure.

Figure 1-1 Differences between databases and schemas in MySQL and GaussDB



- In MySQL, database and schema are synonyms. In GaussDB, a database can have multiple schemas. In this feature, each database in MySQL is mapped to a schema in GaussDB.
- In MySQL, an index belongs to a table. In GaussDB, an index belongs to a schema. As a result, an index name must be unique in a schema in GaussDB and must be unique in a table in MySQL. This difference will be retained as a current constraint.

2 B-compatible Mode

2.1 Data Types

2.1.1 Numeric Data Types

Integer

Unless otherwise specified, considering the characteristics of the floating-point type in terms of precision, scale, and number of digits after the decimal point, the floating-point type is not supported in B-compatible mode by default. You are advised to use a valid integer type.

Differences in terms of the integer types:

- Input format:
  - MySQL

    For characters such as "asbd", "12dd", and "12 12", the system truncates them or returns 0 and reports a WARNING. Data fails to be inserted into a table in strict mode.
  - GaussDB
    - For integer types (TINYINT, SMALLINT, MEDIUMINT, INT, INTEGER, and BIGINT), if the invalid part of a character string is truncated, for example, "12@3", no message is displayed. Data is successfully inserted into a table.
    - If the whole integer is truncated (for example, "@123") or the character string is empty, 0 is returned and data is successfully inserted into a table.
- Operators:
  - +, -, and *

    GaussDB: When INT, INTEGER, SMALLINT, or BIGINT is used for calculation, a value of the original type is returned and is not changed to a larger type. If the return value exceeds the range, an error is reported.

MySQL: The value can be changed to BIGINT for calculation.

- |, &, ^, and ~

  GaussDB: The value is calculated in the bits occupied by the type. In GaussDB, ^ indicates the exponentiation operation. If the XOR operator is required, replace it with #.

  MySQL: The value is changed to a larger type for calculation.

- Explicit type conversion of negative numbers:

  GaussDB: The result is **0** in loose mode and an error is reported in strict mode.

  MySQL: The most significant bit is replaced with a numeric bit based on the corresponding binary value, for example, (-1)::uint4 = 4294967295.

- Other differences:

  The precision of INT[(M)] controls formatted output in MySQL. GaussDB supports only the syntax but does not support the function.

- Aggregate function:
  - variance: indicates the sample variance in GaussDB and the population variance in MySQL.
  - stddev: indicates the sample standard deviation in GaussDB and the overall standard deviation in MySQL.
- Display width:
  - If **ZEROFILL** is not specified when the width information is specified for an integer column, the width information is not displayed in the table structure description.
  - When the INSERT statement is used to insert a column of the character type, GaussDB pads 0s before inserting the column.
  - The JOIN USING statement involves type derivation. In MySQL, the first table column is used by default. In GaussDB, if the result is of the signed type, the width information is invalid. Otherwise, the width of the first table column is used.
  - For GREATEST/LEAST, IFNULL/IF, and CASE WHEN/DECODE, MySQL does not pad 0s. In GaussDB, 0s are padded when the type and width information is consistent.
  - MySQL supports this function when it is used as the input or output parameter or return value of a function or stored procedure. GaussDB neither reports syntax errors nor supports this function.

For details about the differences in GaussDB and MySQL in terms of the integer types, see **Table 2-1**.

Table 2-1 Integer types

MySQL	GaussDB	Difference
BOOL	Supported, with differences	MySQL: The BOOL/BOOLEAN type is actually mapped to the TINYINT type. GaussDB: BOOL is supported. • Valid literal values for the "true" state include: **TRUE**, **'t'**, **'true'**, **'y'**, **'yes'**, **'1'**, **'TRUE'**, **true**, **'on'**, and all non-zero values. • Valid literal values for the "false" state include: **FALSE**, **'f'**, **'false'**, **'n'**, **'no'**, **'0'**, **0**, **'FALSE'**, **false**, and **'off'**. **TRUE** and **FALSE** are standard expressions, compatible with SQL statements.
BOOLEAN	Supported, with differences	
TINYINT[(M)] [UNSIGNED]	Supported, with differences	For details, see **Differences in terms of the integer types**.
SMALLINT[(M)] [UNSIGNED]	Supported, with differences	For details, see **Differences in terms of the integer types**.
MEDIUMINT[(M)] [UNSIGNED]	Supported, with differences	MySQL requires 3 bytes to store MEDIUMINT data. • The signed range is –8388608 to +8388607. • The unsigned range is 0 to +16777215. GaussDB maps data to the INT type and requires 4 bytes for storage. • The signed range is –2147483648 to +2147483647. • The unsigned range is 0 to +4294967295. For details about other differences, see **Differences in terms of the integer types**.
INT[(M)] [UNSIGNED]	Supported, with differences	For details, see **Differences in terms of the integer types**.
INTEGER[(M)] [UNSIGNED]	Supported, with differences	For details, see **Differences in terms of the integer types**.
BIGINT[(M)] [UNSIGNED]	Supported, with differences	For details, see **Differences in terms of the integer types**.

Arbitrary Precision Types

Table 2-2 Arbitrary precision types

MySQL	GaussDB	Difference
DECIMAL[(M[,D])]	Supported, with differences	• Operator: In GaussDB, "^" indicates the exponentiation operation. If the XOR operator is required, replace it with "#". In MySQL, "^" indicates the XOR operation. • Value range: The precision **M** and scale **D** support only integers and do not support floating-point values. • Input format: No error is reported when all input parameters of a character string (for example, "@123") are truncated. An error is reported only when it is partially truncated, for example, "12@3".
NUMERIC[(M[,D])]	Supported, with differences	
DEC[(M[,D])]	Supported, with differences	
FIXED[(M[,D])]	Not supported	-

Floating-Point Types

Table 2-3 Floating-point types

MySQL	GaussDB	Difference
FLOAT[(M,D)]	Supported, with differences	• Partitioned table: The FLOAT data type does not support partitioned tables with the key partitioning policy. • Operator: In GaussDB, "^" indicates the exponentiation operation. If the XOR operator is required, replace it with "#". In MySQL, "^" indicates the XOR operation. • Value range: The precision **M** and scale **D** support only integers and do not support floating-point values. • Output format: An ERROR message is reported for invalid input parameters. No WARNING is reported in loose mode (that is, **sql_mode** is set to **''**).

MySQL	GaussDB	Difference
FLOAT(p)	Supported, with differences	• Partitioned table: The FLOAT data type does not support partitioned tables with the key partitioning policy. • Operator: The ^ operator is used for the numeric types, which is different from that in MySQL. In GaussDB, the ^ operator is used for exponential calculation. • Value range: When the precision **p** is defined, only valid integer data types are supported. • Output format: An ERROR message is reported for invalid input parameters. No WARNING is reported in loose mode (that is, **sql_mode** is set to **''**).
DOUBLE[(M, D)]	Supported, with differences	• Partitioned table: The DOUBLE data type does not support partitioned tables with the key partitioning policy. • Operator: In GaussDB, "^" indicates the exponentiation operation. If the XOR operator is required, replace it with "#". In MySQL, "^" indicates the XOR operation. • Value range: The precision **M** and scale **D** support only integers and do not support floating-point values. • Output format: An ERROR message is reported for invalid input parameters. No WARNING is reported in loose mode (that is, **sql_mode** is set to **''**).
DOUBLE PRECISION[(M,D)]	Supported, with differences	• Operator: In GaussDB, "^" indicates the exponentiation operation. If the XOR operator is required, replace it with "#". In MySQL, "^" indicates the XOR operation. • Value range: The precision **M** and scale **D** support only integers and do not support floating-point values. • Output format: An ERROR message is reported for invalid input parameters. No WARNING is reported in loose mode (that is, **sql_mode** is set to **''**).

MySQL	GaussDB	Difference
REAL[(M,D)]	Supported, with differences	• Partitioned table: The REAL data type does not support partitioned tables with the key partitioning policy. • Operator: In GaussDB, "^" indicates the exponentiation operation. If the XOR operator is required, replace it with "#". In MySQL, "^" indicates the XOR operation. • Value range: The precision **M** and scale **D** support only integers and do not support floating-point values. • Output format: An ERROR message is reported for invalid input parameters. No WARNING message is reported in loose mode (that is, **sql_mode** is set to **''**).

Sequential Integers

Table 2-4 Sequential integers

MySQL	GaussDB	Difference
SERIAL	Supported, with differences	For details about SERIAL in GaussDB, see "SQL Reference > Data Types > Value Types" in *Developer Guide*. The differences in specifications are as follows: `CREATE TABLE test(f1 serial, f2 CHAR(20));` • The SERIAL of MySQL is mapped to BIGINT(20) UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE, and the SERIAL of GaussDB is mapped to INTEGER NOT NULL DEFAULT nextval('test_f1_seq'::regclass). For example: `-- Definition of MySQL SERIAL:` `mysql> SHOW CREATE TABLE test\G` `*************************** 1. row ***************************` `Table: test` `Create Table: CREATE TABLE `test` (` `` `f1` bigint(20) unsigned NOT NULL AUTO_INCREMENT, `` `` `f2` char(20) DEFAULT NULL, `` `` UNIQUE KEY `f1` (`f1`) `` `) ENGINE=InnoDB DEFAULT CHARSET=utf8` `1 row in set (0.00 sec)` `-- Definition of GaussDB SERIAL` `gaussdb=# \d+ test` `Table "public.test"` `Column	Type	Modifiers	Storage	Stats target	Description` `--------+---------------+----------+---------+--------------+-------------` `f1	integer	not null default nextval('test_f1_seq'::regclass)	plain		` `f2	character(20)		extended		` `Has OIDs: no` `Options: orientation=row, compression=no, storage_type=USTORE` • Differences in using INSERT to insert default values of the SERIAL type. For example: `-- Inserting default values of the SERIAL type in MySQL` `mysql> INSERT INTO test VALUES(DEFAULT, 'aaaa');` `Query OK, 1 row affected (0.00 sec)` `mysql> INSERT INTO test VALUES(10, 'aaaa');` `Query OK, 1 row affected (0.00 sec)` `mysql> INSERT INTO test VALUES(DEFAULT, 'aaaa');` `Query OK, 1 row affected (0.00 sec)` `mysql> SELECT * FROM test;` `+----+------+` `| f1	f2 |` `+----+------+` `| 1	aaaa |` `| 10	aaaa |`

MySQL	GaussDB	Difference
			11	aaaa | +----+------+ 3 rows in set (0.00 sec) -- Inserting default values of the SERIAL type in GaussDB gaussdb=# INSERT INTO test VALUES(DEFAULT, 'aaaa'); INSERT 0 1 gaussdb=# INSERT INTO test VALUES(10, 'aaaa'); INSERT 0 1 gaussdb=# INSERT INTO test VALUES(DEFAULT, 'aaaa'); INSERT 0 1 gaussdb=# SELECT * FROM test; f1	f2 ----+---------------------- 1	aaaa 2	aaaa 10	aaaa (3 rows) • Differences in performing REPLACE on referencing columns of the SERIAL type. For details about GaussDB referencing columns, see "SQL Reference > SQL Syntax > R > REPLACE" in *Developer Guide*. For example: -- Inserting values of the referencing columns of the SERIAL type in MySQL mysql> REPLACE INTO test VALUES(f1, 'aaaa'); Query OK, 1 row affected (0.00 sec) mysql> REPLACE INTO test VALUES(f1, 'bbbb'); Query OK, 1 row affected (0.00 sec) mysql> SELECT * FROM test; +----+------+ | f1	f2 | +----+------+ | 1	aaaa
2	bbbb	 +----+------+ 2 rows in set (0.00 sec) -- Inserting values of the referencing columns of the SERIAL type in GaussDB gaussdb=# REPLACE INTO test VALUES(f1, 'aaaa'); REPLACE 0 1 gaussdb=# REPLACE INTO test VALUES(f1, 'bbbb'); REPLACE 0 1 gaussdb=# SELECT * FROM test; f1	f2 ----+---------------------- 0	aaaa 0	bbbb (2 rows)

2.1.2 Date and Time Data Types

Table 2-5 Date and time data types

MySQL	GaussDB	Difference
DATE	Supported, with differences	GaussDB supports the date data type. Compared with MySQL, GaussDB has the following differences in specifications: ● Input format – GaussDB supports only the character type and does not support the numeric type. For example, the format can be '2020-01-01' or '20200101', but cannot be 20200101. MySQL supports conversion from numeric input to the date type. – Separator: GaussDB does not support the plus sign (+) or colon (:) as the separator between the year, month, and day. Other symbols are supported. MySQL supports all symbols as separators. Sometimes, the mixed use of separators is not supported, which is different from MySQL, such as '2020-01>01' and '2020/01+01'. You are advised to use hyphens (-) or slashes (/) as separators. – No separator: You are advised to use the complete format, for example, 'YYYYMMDD' or 'YYMMDD'. The parsing rules of incomplete formats (including the ultra-long format) are different from those of MySQL. An error may be reported or the parsing result may be inconsistent with that of MySQL. Therefore, the incomplete format is not recommended. ● Output format If the **sql_mode** parameter of GaussDB does not contain **'strict_trans_tables'** (the strict mode is used unless otherwise defined as the loose mode), the year, month, and day can be set to **0**. However, the value is converted to a valid value in the sequence of year, month, and day. For example, date '0000-00-10' is converted to 0002-12-10 BC. If the input is invalid or exceeds the range, a warning message is reported and the value **0000-00-00** is returned. MySQL outputs the date value as it is, even if the year, month, and day are set to **0**.

MySQL	GaussDB	Difference
		• Value range The value range of GaussDB is 4713-01-01 BC to 5874897-12-31 AD. BC dates are supported. In loose mode, if the value exceeds the range, **0000-00-00** is returned. In strict mode, an error is reported. In MySQL, the value range is 0000-00-00 to 9999-12-31. In loose mode, if the value exceeds the range, the performance varies in different scenarios. An error may be reported (for example, in the SELECT statement) or the value **0000-00-00** may be returned (for example, in the INSERT statement). As a result, when the date type is used as the input parameter of the function, the results returned by the function are different. • Operator – GaussDB supports only the comparison operators =, !=, <, <=, >, and >= between date types and returns **true** or **false**. For the addition operation between the date and interval types, the return result is of the date type. For the subtraction operation between the date and interval types, the return result is of the date type. For the subtraction operation between date types, the return result is of the interval type. – When the MySQL date type and other numeric types are calculated, the date type is converted to the numeric type, and then the calculation is performed based on the numeric type. The result is also of the numeric type. It is different from GaussDB. For example: `-- MySQL: date+numeric. Convert the date type to 20200101 and add it to 1. The result is 20200102.` `mysql> SELECT date'2020-01-01' + 1;` `+----------------------+` `| date'2020-01-01' + 1 |` `+----------------------+` `| 20200102 |` `+----------------------+` `1 row in set (0.00 sec)` `-- GaussDB: date+numeric. Convert the numeric type to the interval type (1 day), and then add them up to obtain a new date.` `gaussdb=# SELECT date'2020-01-01' + 1;` `?column?` `------------` `2020-01-02` `(1 row)`

MySQL	GaussDB	Difference
		• Type conversion Compared with MySQL, GaussDB supports conversion between the date type and char(n), nchar(n), datetime, or timestamp type, but does not support conversion between the date type and binary, decimal, JSON, integer, unsigned integer, or time type. The principles for determining common types in scenarios such as collections and complex expressions are different from those in MySQL. For details, see **Data Type Conversion**.

MySQL	GaussDB	Difference
DATETIME[(fsp)]	Supported, with differences.	GaussDB supports the datetime data type. Compared with MySQL, GaussDB has the following differences in specifications: ● Input formats: – GaussDB supports only the character type and does not support the numeric type. For example, '2020-01-01 10:20:30.123456' or '20200101102030.123456' is supported, but 20200101102030.123456 is not supported. MySQL supports conversion from numeric input to the datetime type. – Separator: GaussDB does not support the plus sign (+) or colon (:) as the separator between the year, month, and day. Other symbols are supported. Only colons (:) can be used as separators between hours, minutes, and seconds. Sometimes, the mixed use of separators is not supported, which is different from MySQL. Therefore, it is not recommended. MySQL supports all symbols as separators. – No separator: The complete format 'YYYYMMDDhhmiss.ffffff' is recommended. The parsing rules of incomplete formats (including the ultra-long format) may be different from those of MySQL. An error may be reported or the parsing result may be inconsistent with that of MySQL. Therefore, the incomplete format is not recommended. ● Output formats: – The format is 'YYYY-MM-DD hh:mi:ss.ffffff', which is the same as that of MySQL and is not affected by the **DateStyle** parameter. However, for the precision part, if the last several digits are 0, they are not displayed in GaussDB but displayed in MySQL. – If the **sql_mode** parameter of GaussDB does not contain **'strict_trans_tables'** (the strict mode is used unless otherwise defined as the loose mode), the year, month, and day can be set to **0**. However, the value is converted to a valid value in the sequence of year, month, and day. For example, datetime '0000-00-10 00:00:00'

MySQL	GaussDB	Difference
		is converted to 0002-12-10 00:00:00 BC. If the input is invalid or exceeds the range, a warning message is reported and the value **0000-00-00 00:00:00** is returned. MySQL outputs the datetime value as it is, even if the year, month, and day are set to **0**. ● Value range 4713-11-24 00:00:00.000000 BC to 294277-01-09 04:00:54.775806 AD. If the value is **294277-01-09 04:00:54.775807 AD**, **infinity** is returned. If the value exceeds the range, GaussDB reports an error in strict mode. Whether MySQL reports an error depends on the application scenario. Generally, no error is reported in the query scenario. However, an error is reported when the DML or SQL statement is executed to change the value of a table attribute. In loose mode, GaussDB returns **0000-00-00 00:00:00**. MySQL may report an error, return **0000-00-00 00:00:00**, or return **null** based on the application scenario. As a result, the execution result of the function that uses the datetime type as the input parameter is different from that of MySQL. ● Precision The value ranges from 0 to 6. For a table column, the default value is **0**, which is the same as that in MySQL. In the datetime[(p)]'str' expression, GaussDB parses **(p)** as the precision. The default value is **6**, indicating that **'str'** is formatted to the datetime type based on the precision specified by **p**. MySQL does not support the datetime[(p)]'str' expression. ● Operator – GaussDB supports only the comparison operators =, !=, <, <=, >, and >= between datetime types and returns **true** or **false**. For the addition operation between the datetime and interval types, the return result is of the datetime type. For the subtraction operation between the datetime and interval types, the return result is of the datetime type. For the subtraction operation between datetime types, the return result is of the interval type.

MySQL	GaussDB	Difference
		– When the MySQL datetime type and other numeric types are calculated, the datetime type is converted to the numeric type, and then the calculation is performed based on the numeric type. The result is also of the numeric type. It is different from GaussDB. For example: -- MySQL: datetime+numeric. Convert the datetime type to 20201010123456 and add it to 1. The result is 20201010123457. mysql> SELECT cast('2020-10-10 12:34:56.123456' AS datetime) + 1; +--+ | cast('2020-10-10 12:34:56.123456' AS datetime) + 1 | +--+ | 20201010123457 | +--+ 1 row in set (0.00 sec) -- GaussDB: datetime+numeric. Convert the numeric type to the interval type (1 day), and then add them up to obtain the new datetime. gaussdb=# SELECT cast('2020-10-10 12:34:56.123456' AS datetime) + 1; ?column? --------------------- 2020-10-11 12:34:56 (1 row) If the calculation result of the datetime type and numeric type is used as the input parameter of a function, the result of the function may be different from that of MySQL. • Type conversion Compared with MySQL, GaussDB supports only conversion between the datetime type and char(n), nchar(n), and timestamp types, and conversion from datetime to date and time types (only value assignment and explicit conversion). The conversion between the datetime type and the binary, decimal, json, integer, or unsigned integer type is not supported. The principles for determining common types in scenarios such as collections and complex expressions are different from those in MySQL. For details, see **Data Type Conversion**. • Time zone In GaussDB, the datetime value can carry the time zone information (time zone offset or time zone name), for example, '2020-01-01 12:34:56.123456 +01:00' or '2020-01-01 2:34:56.123456 CST'. GaussDB converts the time to the time of the current server time

MySQL	GaussDB	Difference
		zone. MySQL 5.7 does not support this function. MySQL 8.0 and later versions support this function. • The table columns of the datetime data type in GaussDB are actually converted to the timestamp(p) without time zone. When you query the table information or use a tool to export the table structure, the data type of columns is timestamp(p) without time zone instead of datetime. For MySQL, datetime(p) is displayed.

MySQL	GaussDB	Difference
TIMESTAMP[(fsp)]	Supported, with differences	GaussDB supports the timestamp data type. Compared with MySQL, GaussDB has the following differences in specifications: ● Input format – It supports only the character type and does not support the numeric type. For example, '2020-01-01 10:20:30.123456' or '20200101102030.123456' is supported, but 20200101102030.123456 is not supported. MySQL supports conversion from numeric input to the timestamp type. – Separator: It does not support the plus sign (+) or colon (:) as the separator between the year, month, and day. Other symbols are supported. Only colons (:) can be used as separators between hours, minutes, and seconds. Sometimes, the mixed use of separators is not supported, which is different from MySQL. Therefore, it is not recommended. MySQL supports all symbols as separators. – No separator: The complete format 'YYYYMMDDhhmiss.ffffff' is recommended. The parsing rules of incomplete formats (including the ultra-long format) may be different from those of MySQL. An error may be reported or the parsing result may be inconsistent with that of MySQL. Therefore, the incomplete format is not recommended. ● Output format – The format is 'YYYY-MM-DD hh:mi:ss.ffffff', which is the same as that of MySQL and is not affected by the **DateStyle** parameter. However, for the precision part, if the last several digits are 0, they are not displayed in GaussDB but displayed in MySQL. – If the **sql_mode** parameter of GaussDB does not contain **'strict_trans_tables'** (the strict mode is used unless otherwise defined as the loose mode), the year, month, and day can be set to **0**. However, the value is converted to a valid value in the sequence of year, month, and day. For example, timestamp '0000-00-10

MySQL	GaussDB	Difference
		00:00:00' is converted to 0002-12-10 00:00:00 BC. If the input is invalid or exceeds the range, a warning message is reported and the value **0000-00-00 00:00:00** is returned. MySQL outputs the timestamp value as it is, even if the year, month, and day are set to **0**. ● Value range 4713-11-24 00:00:00.000000 BC to 294277-01-09 04:00:54.775806 AD. If the value is **294277-01-09 04:00:54.775807 AD**, **infinity** is returned. If the value exceeds the range, GaussDB reports an error in strict mode. Whether MySQL reports an error depends on the application scenario. Generally, no error is reported in the query scenario. However, an error is reported when the DML or SQL statement is executed to change the value of a table attribute. In loose mode, GaussDB returns **0000-00-00 00:00:00**. MySQL may report an error, return **0000-00-00 00:00:00**, or return **null** based on the application scenario. As a result, the execution result of the function that uses the timestamp type as the input parameter is different from that of MySQL. ● Precision The value ranges from 0 to 6. For a table column, the default value is **0**, which is the same as that in MySQL. In the timestamp[(p)] 'str' expression: – GaussDB parses **(p)** as the precision. The default value is **6**, indicating that **'str'** is formatted to the timestamp type based on the precision specified by **p**. – The meaning of timestamp 'str' in MySQL is the same as that in GaussDB. The default precision is **6**. However, timestamp(p) 'str' is parsed as a function call. **p** is used as the input parameter of the timestamp function. The result returns a value of the timestamp type, and **'str'** is used as the alias of the projection column. ● Operator – GaussDB supports only the comparison operators =, !=, <, <=, >, and >= between timestamp types and returns **true** or **false**. For the addition operation between

MySQL	GaussDB	Difference
		the timestamp and interval types, the return result is of the timestamp type. For the subtraction operation between the timestamp and interval types, the return result is of the timestamp type. For the subtraction operation between timestamp types, the return result is of the interval type. – When the MySQL timestamp type and other numeric types are calculated, the timestamp type is converted to the numeric type, and then the calculation is performed based on the numeric type. The result is also of the numeric type. It is different from GaussDB. For example: `-- MySQL: timestamp+numeric. Convert the timestamp type to 20201010123456.123456 and add it to 1. The result is 20201010123457.123456.` `mysql> SELECT timestamp '2020-10-10 12:34:56.123456' + 1;` `+-------------------------------------------+` `| timestamp '2020-10-10 12:34:56.123456' + 1 |` `+-------------------------------------------+` `| 20201010123457.123456 |` `+-------------------------------------------+` `1 row in set (0.00 sec)` `-- GaussDB: timestamp+numeric. Convert the numeric type to the interval type (1 day), and then add them up to obtain a new timestamp.` `gaussdb=# SELECT timestamp '2020-10-10 12:34:56.123456' + 1;` `?column?` `---------------------------` `2020-10-11 12:34:56.123456` `(1 row)` If the calculation result of the timestamp type and numeric type is used as the input parameter of a function, the result of the function may be different from that of MySQL. ● Type conversion Compared with MySQL, GaussDB supports only conversion between timestamp and char(n), varchar(n), and datetime, and conversion from timestamp to date and time (only value assignment and explicit conversion). The conversion between the timestamp type and the binary, decimal, json, integer, or unsigned integer type is not supported. The principles for determining common types in scenarios such as collections and complex expressions are

MySQL	GaussDB	Difference
		different from those in MySQL. For details, see **Data Type Conversion**. • Time zone In GaussDB, the timestamp value can carry the time zone information (time zone offset or time zone name), for example, '2020-01-01 12:34:56.123456 +01:00' or '2020-01-01 2:34:56.123456 CST'. GaussDB converts the time to the time of the current server time zone. If the time zone of the server is changed, the timestamp value is converted to the timestamp of the new time zone. MySQL 5.7 does not support this function. MySQL 8.0 and later versions support this function. • The table columns of the timestamp data type in GaussDB are actually converted to the timestamp(p) with time zone. When you query the table information or use a tool to export the table structure, the data type of columns is timestamp(p) with time zone instead of timestamp. For MySQL, timestamp(p) is displayed.

MySQL	GaussDB	Difference
TIME[(fsp)]	Supported, with differences	GaussDB supports the time data type. Compared with MySQL, GaussDB has the following differences in specifications: ● Input format – It supports only the character type and does not support the numeric type. For example, '1 10:20:30' or '102030' is supported, but 102030 is not supported. MySQL supports conversion from numeric input to the time type. – Separator: GaussDB supports only colons (:) as separators between hours, minutes, and seconds. MySQL supports all symbols as separators. – No separator: The complete format 'hhmiss.ffffff' is recommended. The parsing rules of incomplete formats (including the ultra-long format) may be different from those of MySQL. An error may be reported or the parsing result may be inconsistent with that of MySQL. Therefore, the incomplete format is not recommended. – When a negative value is entered for minute, second, or precision, GaussDB may ignore the first part of the negative value, which is parsed as 0. For example, '00:00:-10' is parsed as '00:00:00'. An error may also be reported. For example, if '00:00:-10000' is parsed, an error will be reported. The result depends on the range of the input value. However, MySQL reports an error in both cases. ● Output format The format is hh:mi:ss.ffffff, which is the same as that of MySQL. However, for the precision part, if the last several digits are 0, they are not displayed in GaussDB but displayed in MySQL. ● Value range –838:59:59.000000 to 838:59:59.000000, which is the same as that of MySQL. For values that exceed the range, when GaussDB performs DML operations such as SELECT, INSERT, and UPDATE in loose mode, it returns the nearest boundary values such as **-838:59:59** or **838:59:59**. In MySQL, an error is reported during query, or the nearest

MySQL	GaussDB	Difference
		boundary value is returned after a DML operation. As a result, when the time type is used as the input parameter of the function, the results returned by the function are different. ● Precision The value ranges from 0 to 6. For a table column, the default value is **0**, which is the same as that in MySQL. In the time(p) 'str' expression, GaussDB parses **(p)** as the precision. The default value is **6**, indicating that **'str'** is formatted to the time type based on the precision specified by **p**. MySQL parses it as a time function, **p** is an input parameter, and **'str'** is the alias of the projection column. ● Operator – GaussDB supports only the comparison operators =, !=, <, <=, >, and >= between time types and returns **true** or **false**. For the addition operation between the time and interval types, the return result is of the time type. For the subtraction operation between the time and interval types, the return result is of the time type. For the subtraction operation between time types, the return result is of the interval type. – When the MySQL time type and other numeric types are calculated, the time type is converted to the numeric type, and then the calculation is performed based on the numeric type. The result is also of the numeric type. It is different from GaussDB. For example: `-- MySQL: time+numeric. Convert the time type to 123456 and add it to 1. The result is 123457.` `mysql> SELECT time '12:34:56' + 1;` `+---------------------+` `| time '12:34:56' + 1 |` `+---------------------+` `| 123457 |` `+---------------------+` `1 row in set (0.00 sec)` `-- GaussDB: time+numeric. Convert the numeric type to the interval type (1 day), and then add them up to obtain the new time. Because 24 hours are added, the obtained time is still 12:34:56.` `gaussdb=# SELECT time '12:34:56' + 1;` ` ?column?` `----------`

MySQL	GaussDB	Difference
		12:34:56 (1 row) If the calculation result of the time type and numeric type is used as the input parameter of a function, the result of the function may be different from that of MySQL. • Type conversion Compared with MySQL, GaussDB supports only conversion between the time type and char(n) or nchar(n) type, and conversion between the datetime or timestamp type and time type. The conversion between the time type and binary, decimal, date, JSON, integer, or unsigned integer type is not supported. The principles for determining common types in scenarios such as collections and complex expressions are different from those in MySQL. For details, see **Data Type Conversion**.
YEAR[(4)]	Supported, with differences	GaussDB supports the year data type. Compared with MySQL, GaussDB has the following differences in specifications: • Operator – GaussDB supports only the comparison operators =, !=, <, <=, >, and >= between year types and returns **true** or **false**. – GaussDB supports only the arithmetic operators + and - between the year and int4 types and returns integer values. MySQL returns unsigned integer values. • Type conversion Compared with MySQL, GaussDB supports only the conversion between the year type and int4 type, and supports only the conversion from the int4, varchar, numeric, date, time, timestamp, or timestamptz type to the year type. The principles for determining common types in scenarios such as collections and complex expressions are different from those in MySQL. For details, see **Data Type Conversion**.

MySQL	GaussDB	Difference
INTERVAL	Supported, with differences	GaussDB supports the INTERVAL data type, but INTERVAL is an expression in MySQL. The differences are as follows: ● The date input of the character string type cannot be used as an operation, for example, SELECT '2023-01-01' + interval 1 day. ● In the INTERVAL expr unit syntax, **expr** cannot be a negative integer or floating-point number, for example, SELECT date'2023-01-01' + interval -1 day. ● In the INTERVAL expr unit syntax, **expr** cannot be the input of an operation expression, for example, SELECT date'2023-01-01' + interval 4/2 day. ● When the INTERVAL expression is used for calculation, the return value is of the datetime type. For MySQL, the return value is of the datetime or date type. The calculation logic is the same as that of GaussDB but different from that of MySQL. ● In the INTERVAL expr unit syntax, the value range of **expr** varies with the unit. The maximum value range is [–2147483648, 2147483647]. If the value exceeds the range, an error is reported in strict mode, and a warning is reported in loose mode and **0** is returned. ● In the INTERVAL expr unit syntax, if the number of columns specified by **expr** is greater than the expected number of columns in unit, an error is reported in strict mode, and a warning is reported in loose mode and **0** is returned. For example, if the value of **unit** is **DAY_HOUR**, the expected number of columns is 2. If the value of **expr** is **'1-2-3'**, the expected number of columns is 3.

2.1.3 String Data Types

Table 2-6 String data types

MySQL	GaussDB	Difference
CHAR[(M)]	Supported, with differences	• Input format – The length of parameters and return values of GaussDB user-defined functions cannot be verified. The length of stored procedure parameters cannot be verified. In addition, correct spaces cannot be supplemented when **PAD_CHAR_TO_FULL_LENGTH** is enabled. However, MySQL supports these functions. – GaussDB does not support escape characters or double quotation marks (""). MySQL supports these inputs. • Syntax The CAST(expr as char) syntax of GaussDB cannot convert the input string to the corresponding type based on the string length. It can only be converted to the varchar type. CAST('' as char) and CAST('' as char(0)) cannot convert an empty string to the char(0) type. MySQL supports conversion to the corresponding type by length. • Operator – After performing addition, subtraction, multiplication, division, or modulo operations on a string (that can be converted to a floating-point value) and an integer value, GaussDB returns an integer, while MySQL returns a floating-point value. – If a value is divided by 0, GaussDB reports an error, and MySQL returns **null**. – "~": returns a negative number in GaussDB and an 8-byte unsigned integer in MySQL. – "^": indicates a power in GaussDB and a bitwise XOR in MySQL.

MySQL	GaussDB	Difference
VARCHAR(M)	Supported, with differences	• Input format – The length of parameters and return values of GaussDB user-defined functions cannot be verified. The length of stored procedure parameters cannot be verified. However, MySQL supports these functions. – The length of temporary variables in GaussDB user-defined functions and stored procedures can be verified, and an error or truncation alarm is reported in strict or loose mode. However, MySQL does not support these functions. – GaussDB does not support escape characters or double quotation marks (""). MySQL supports these inputs. • Operator – After performing addition, subtraction, multiplication, division, or modulo operations on a string (that can be converted to a floating-point value) and an integer value, GaussDB returns an integer, while MySQL returns a floating-point value. – If a value is divided by 0, GaussDB reports an error, and MySQL returns **null**. – "~": returns a negative number in GaussDB and an 8-byte unsigned integer in MySQL. – "^": indicates a power in GaussDB and a bitwise XOR in MySQL.

MySQL	GaussDB	Difference
TINYTEXT	Supported, with differences	• Input format – In GaussDB, the length of this type cannot exceed 1 GB. If the length exceeds 1 GB, an error is reported. In MySQL, the length of this type cannot exceed 255 bytes. Otherwise, an error is reported in strict mode, and data is truncated and an alarm is generated in loose mode. – GaussDB does not support escape characters or double quotation marks (""). MySQL supports these inputs. • Operator – After performing addition, subtraction, multiplication, division, or modulo operations on a string (that can be converted to a floating-point value) and an integer value, GaussDB returns an integer, while MySQL returns a floating-point value. – If a value is divided by 0, GaussDB reports an error, and MySQL returns **null**. – "~": returns a negative number in GaussDB and an 8-byte unsigned integer in MySQL. – "^": indicates a power in GaussDB and a bitwise XOR in MySQL.

MySQL	GaussDB	Difference
TEXT	Supported, with differences	• Input format – In GaussDB, the length of this type cannot exceed 1 GB. If the length exceeds 1 GB, an error is reported. In MySQL, the length of this type cannot exceed 65535 bytes. Otherwise, an error is reported in strict mode, and data is truncated and an alarm is generated in loose mode. – GaussDB does not support escape characters or double quotation marks (""). MySQL supports these inputs. • Operator – After performing addition, subtraction, multiplication, division, or modulo operations on a string (that can be converted to a floating-point value) and an integer value, GaussDB returns an integer, while MySQL returns a floating-point value. – If a value is divided by 0, GaussDB reports an error, and MySQL returns **null**. – "~": returns a negative number in GaussDB and an 8-byte unsigned integer in MySQL. – "^": indicates a power in GaussDB and a bitwise XOR in MySQL.

MySQL	GaussDB	Difference
MEDIUMTEXT	Supported, with differences	• Input format – In GaussDB, the length of this type cannot exceed 1 GB. If the length exceeds 1 GB, an error is reported. In MySQL, the length of this type cannot exceed 16777215 bytes. Otherwise, an error is reported in strict mode, and data is truncated and an alarm is generated in loose mode. – GaussDB does not support escape characters or double quotation marks (""). MySQL supports these inputs. • Operator – After performing addition, subtraction, multiplication, division, or modulo operations on a string (that can be converted to a floating-point value) and an integer value, GaussDB returns an integer, while MySQL returns a floating-point value. – If a value is divided by 0, GaussDB reports an error, and MySQL returns **null**. – "~": returns a negative number in GaussDB and an 8-byte unsigned integer in MySQL. – "^": indicates a power in GaussDB and a bitwise XOR in MySQL.

MySQL	GaussDB	Difference
LONGTEXT	Supported, with differences	• Input format – GaussDB supports a maximum of 1 GB, and MySQL supports a maximum of 4 GB minus 1 byte. – GaussDB does not support escape characters or double quotation marks (""). MySQL supports these inputs. • Operator – After performing addition, subtraction, multiplication, division, or modulo operations on a string (that can be converted to a floating-point value) and an integer value, GaussDB returns an integer, while MySQL returns a floating-point value. – If a value is divided by 0, GaussDB reports an error, and MySQL returns **null**. – "~": returns a negative number in GaussDB and an 8-byte unsigned integer in MySQL. – "^": indicates a power in GaussDB and a bitwise XOR in MySQL.
ENUM('value1','value2',...)	Not supported	-
SET('value1','value2',...)	Supported	-

2.1.4 Binary Data Types

Table 2-7 Binary data types

MySQL	GaussDB	Difference
BINARY[(M)]	Not supported	-
VARBINARY(M)	Not supported	-

MySQL	GaussDB	Difference
TINYBLOB	Supported, with differences	• Value range: In GaussDB, this type is mapped from the BYTEA type. Its length cannot exceed 1 GB. Otherwise, an error is reported. In MySQL, the length of this type cannot exceed 255 bytes. Otherwise, an error is reported in strict mode, and data is truncated and an alarm is generated in loose mode. • Input format: Escape characters and double quotation marks ("") are not supported. • Output format: For the '\0' character, the query result is displayed as "\000". If the getBytes API of the JDBC driver is used, the result is the '\0' character. • Operator: Arithmetic operators (+ - * / %) are not supported. Common logical operators OR, AND, NOT (|| && !) are not supported. Common bitwise operators (~ & | ^) are not supported.
BLOB	Supported, with differences	• Value range: In GaussDB, this type is mapped from the BYTEA type. Its length cannot exceed 1 GB. Otherwise, an error is reported. In MySQL, the length of this type cannot exceed 65535 bytes. Otherwise, an error is reported in strict mode, and data is truncated and an alarm is generated in loose mode. • Input format: Escape characters and double quotation marks ("") are not supported. • Output format: For the '\0' character, the query result is displayed as "\000". If the getBytes API of the JDBC driver is used, the result is the '\0' character. • Operator: Arithmetic operators (+ - * / %) are not supported. Common logical operators OR, AND, NOT (|| && !) are not supported. Common bitwise operators (~ & | ^) are not supported.

MySQL	GaussDB	Difference
MEDIUMBLOB	Supported, with differences	• Value range: In GaussDB, this type is mapped from the BYTEA type. Its length cannot exceed 1 GB. Otherwise, an error is reported. In MySQL, the length of this type cannot exceed 16777215 bytes. Otherwise, an error is reported in strict mode, and data is truncated and an alarm is generated in loose mode. • Input format: Escape characters and double quotation marks ("") are not supported. • Output format: For the '\0' character, the query result is displayed as "\000". If the getBytes API of the JDBC driver is used, the result is the '\0' character. • Operator: Arithmetic operators (+ - * / %) are not supported. Common logical operators OR, AND, NOT (|| && !) are not supported. Common bitwise operators (~ & | ^) are not supported.
LONGBLOB	Supported, with differences	• Value range: In GaussDB, this type is mapped from the BYTEA type. Its length cannot exceed 1 GB. For details, see the centralized and distributed specifications of the BYTEA data type. • Input format: Escape characters and double quotation marks ("") are not supported. • Output format: For the '\0' character, the query result is displayed as "\000". If the getBytes API of the JDBC driver is used, the result is the '\0' character. • Operator: Arithmetic operators (+ - * / %) are not supported. Common logical operators OR, AND, NOT (|| && !) are not supported. Common bitwise operators (~ & | ^) are not supported.
BIT[(M)]	Not supported	-

2.1.5 JSON Data Type

Table 2-8 JSON Data Type

MySQL	GaussDB	Difference
JSON	Supported, with differences	• The JSON types in GaussDB in B-compatible mode are the same as the native JSON type of GaussDB but greatly different from that of MySQL. Therefore, the JSON types are not listed one by one. • For details about the JSON types in GaussDB in B-compatible mode, see "SQL Reference > Data Types > JSON/JSONB Types" in *Developer Guide*.

2.1.6 Attributes Supported by Data Types

Table 2-9 Attributes Supported by Data Types

MySQL	GaussDB
NULL	Supported
NOT NULL	Supported
DEFAULT	Supported
ON UPDATE	Supported
PRIMARY KEY	Supported
AUTO_INCREMENT	Supported
CHARACTER SET name	Supported
COLLATE name	Supported

2.1.7 Data Type Conversion

Conversion between different data types is supported. Data type conversion is involved in the following scenarios:

- The data types of operands of operators (such as comparison and arithmetic operators) are inconsistent. It is commonly used for comparison operations in query conditions or join conditions.
- The data types of arguments and parameters are inconsistent when a function is called.

- The data types of target columns to be updated by DML statements (including INSERT, UPDATE, MERGE, and REPLACE) and the defined column types are inconsistent.
- Explicit type conversion: cast(expr as datatype), which converts an expression to a data type.
- After the target data type of the final projection column is determined by set operations (UNION, MINUS, EXCEPT, and INTERSECT), the type of the projection column in each SELECT statement is inconsistent with the target data type.
- In other expression calculation scenarios, the target data type used for comparison or final result is determined based on the data type of different expressions.
  - DECODE
  - CASE WHEN
  - lexpr [NOT] IN (expr_list)
  - BETWEEN AND
  - JOIN USING(a,b)
  - GREATEST and LEAST
  - NVL and COALESCE

GaussDB and MySQL have different rules for data type conversion and target data types. The following examples show the differences between the two processing modes:

```
-- MySQL: The execution result of IN is 0, indicating false. According to the rule, '1970-01-01' is compared with the expressions in the list in sequence. The results are all 0s. Therefore, the final result is 0.
mysql> SELECT '1970-01-01' IN ('1970-01-02', 1, '1970-01-02');
+-----------------------------------------------+
| '1970-01-01' IN ('1970-01-02', 1, '1970-01-02') |
+-----------------------------------------------+
|                                             0 |
+-----------------------------------------------+

-- GaussDB: The execution result of IN is true, which is opposite to the MySQL result. The common type selected based on the rule is int. Therefore, the left expression '1970-01-01' is converted to the int type and compared with the value after the expression in the list is converted to the int type.
-- When '1970-01-01' and '1970-01-02' are converted to the int type, the values are 1970. (In MySQL-compatible mode, invalid characters and the following content are ignored during conversion, and the previous part is converted to the int type.) The comparison result is equal. Therefore, the returned result is true.
gaussdb=# SELECT '1970-01-01' IN ('1970-01-02', 1::int, '1970-01-02') AS result;
 result
--------
 t
(1 row)
```

Differences in data type conversion rules:

- The GaussDB clearly defines the conversion rules between different data types.
  - Whether to support conversion: Conversion is supported only when the conversion path of two types is defined in the pg_cast system catalog.
  - Conversion scenarios: conversion in any scenario, conversion only in CAST expressions, and conversion only during value assignment. In scenarios that are not supported, data type conversion cannot be performed even if the conversion path is defined.

- MySQL supports conversion between any two data types.

Due to the preceding differences, when MySQL-based applications are migrated to GaussDB, an error may be reported because the SQL statement does not support the conversion between different data types. In the scenario where conversion is supported, different conversion rules result in different execution results of SQL statements.

You are advised to use the same data type in SQL statements for comparison or value assignment to avoid unexpected results or performance loss caused by data type conversion.

Differences in target data type selection rules:

In some scenarios, the data type to be compared or returned can be determined only after the types of multiple expressions are considered. For example, in the UNION operation, projection columns at the same position in different SELECT statements are of different data types. The final data type of the query result needs to be determined based on the data type of the projection columns in each SELECT statement.

GaussDB and MySQL have different rules for determining the target data types.

- GaussDB rules:
  - If the operand types of operators are inconsistent, the operand types are not converted to the target type before calculation. Instead, operators of two data types are directly registered, and two types of processing rules are defined during operator processing. In this mode, implicit type conversion does not exist, but the customized processing rule implies the conversion operation.
  - Rules for determining the target data type in the set operation and expression scenarios:
    - If all types are the same, it is the target type.
    - If the two data types are different, check whether the data types are of the same type, such as the numeric type, character type, and date and time type. If they do not belong to the same type, the target type cannot be determined. In this case, an error is reported during SQL statement execution.
    - For data types with the same **category** attribute (defined in the pg_type system catalog), the data type with the **preferred** attribute (defined in the pg_type system catalog) is selected as the target type. If operand 1 can be converted to operand 2 (no conversion path), but operand 2 cannot be converted to operand 1 or the priority of the numeric type is lower than that of operand 2, then operand 2 is selected as the target type.
    - If three or more data types are involved, the rule for determining the target type is as follows: common_type(type1,type2,type3) = common_type(common_type(type1,type2),type3). Perform iterative processing in sequence to obtain the final result.
    - For IN and NOT IN expressions, if the target type cannot be determined based on the preceding rules, each expression in **lexpr**

and **expr_list** is compared one by one based on the equivalent operator (=).

- Precision determination: The precision of the finally selected expression is used as the final result.

- MySQL rules:
  - If the operand types of operators are inconsistent, determine the target type based on the following rules. Then, convert the inconsistent operand types to the target type and then process the operands.
    - If both parameters are of the string type, they are compared based on the string type.
    - If both parameters are of the integer type, they are compared based on the integer type.
    - If a hexadecimal value is not compared with a numeric value, they are compared based on the binary string.
    - If one parameter is of the datetime/timestamp type, and the other parameter is a constant, the constant is converted to the timestamp type for comparison.
    - If one parameter is of the decimal type, the data type used for comparison depends on the other parameter. If the other type is decimal or integer, the decimal type is used. If the other type is not decimal, the real type is used.
    - In other scenarios, the data type is converted to the real type for comparison.
  - Rules for determining the target data type in the set operation and expression scenarios:
    - Establish a target type matrix between any two types. Given two types, the target type can be determined by using the matrix.
    - If three or more data types are involved, the rule for determining the target type is as follows: common_type(type1,type2,type3) = common_type(common_type(type1,type2),type3). Perform iterative processing in sequence to obtain the final result.
    - If the target type is integer and each expression type contains signed and unsigned integers, the type is promoted to an integer type with higher precision. The result is unsigned only when all expressions are unsigned. Otherwise, the result is signed.
    - The highest precision in the expression is used as the final result.

According to the preceding rules, GaussDB and MySQL differ greatly in data type conversion rules and types cannot be directly compared. In the preceding scenario, the execution result of SQL statements may be different from that in MySQL. In the current version, you are advised to use the same type for all expressions or use CAST to convert the type to the required type in advance to avoid differences.

2.2 System Functions

2.2.1 Flow Control Functions

Table 2-10 Flow control functions

MySQL	GaussDB	Difference
IF()	Supported, with differences	• The **expr1** input parameter supports only the Boolean type. If an input parameter of the non-Boolean type cannot be converted to the Boolean type, an error is reported. • If the types of **expr2** and **expr3** are different and no implicit conversion function exists between the two types, an error is reported. • If the two input parameters are of the same type, the input parameter type is returned. • If the **expr2** and **expr3** input parameters are of the NUMERIC, STRING, or TIME type respectively, GaussDB outputs the TEXT type, while MySQL outputs the VARCHAR type.
IFNULL()	Supported, with differences	• If the types of **expr1** and **expr2** are different and no implicit conversion function exists between the two types, an error is reported. • If the two input parameters are of the same type, the input parameter type is returned. • If the **expr1** and **expr2** input parameters are of the NUMERIC, STRING, or TIME type respectively, GaussDB outputs the TEXT type, while MySQL outputs the VARCHAR type. • If the first input parameter is FLOAT4 and the other is BIGINT or UNSIGNED BIGINT, GaussDB returns the DOUBLE type, while MySQL returns the FLOAT type.

MySQL	GaussDB	Difference
NULLIF()	Supported, with differences	• The NULLIF() type derivation in GaussDB complies with the following logic: – If the data types of two parameters are different and the two input parameter types have an equality comparison operator, the left value type corresponding to the equality comparison operator is returned. Otherwise, the two input parameter types are forcibly compatible. – If an equality comparison operator exists after forcible type compatibility, the left value type of the equality comparison operator after forcible type compatibility is returned. – If the corresponding equality operator cannot be found after forcible type compatibility, an error is reported. -- The two input parameter types have an equality comparison operator. gaussdb=# SELECT pg_typeof(nullif(1::int2, 2::int8)); pg_typeof ---------- smallint (1 row) -- The two input parameter types do not have the equality comparison operator, but the equality comparison operator can be found after forcible type compatibility. gaussdb=# SELECT pg_typeof(nullif(1::int1, 2::int2)); pg_typeof ---------- bigint (1 row) -- The two input parameter types do not have the equality comparison operator, and the equality comparison operator does not exist after forcible type compatibility. gaussdb=# SELECT nullif(1::bit, '1'::MONEY); ERROR: operator does not exist: bit = money LINE 1: SELECT nullif(1::bit, '1'::MONEY); ^ HINT: No operator matches the given name and argument type(s). You might need to add explicit type casts. CONTEXT: referenced column: nullif • The MySQL output type is related only to the type of the first input parameter. – If the type of the first input parameter is TINYINT, SMALLINT, MEDIUMINT, INT, or BOOL, the output is of the INT type. – If the type of the first input parameter is BIGINT, the output is of the BIGINT type.

MySQL	GaussDB	Difference
		– When the type of the first input parameter is UNSIGNED TINYINT, UNSIGNED SMALLINT, UNSIGNED MEDIUMINT, UNSIGNED INT, or BIT, the output is of the UNSIGNED INT type. – If the type of the first input parameter is UNSIGNED BIGINT, the output is of the UNSIGNED BIGINT type. – If the type of the first input parameter is of the FLOAT, DOUBLE, or REAL type, the output is of the DOUBLE type. – If the type of the first input parameter DECIMAL or NUMERIC, the output is of the DECIMAL type. – If the type of the first input parameter is DATE, TIME, DATE, DATETIME, TIMESTAMP, CHAR, VARCHAR, TINYTEXT, ENUM, or SET, the output is of the VARCHAR type. – If the type of the first input parameter is TEXT, MEDIUMTEXT, or LONGTEXT, the output is of the LONGTEXT type. – If the type of the first input parameter is TINYBLOB, the output is of the VARBINARY type. – If the type of the first input parameter is MEDIUMBLOB or LONGBLOB, the output is of the LONGBLOB type. – If the type of the first input parameter is BLOB, the output is of the BLOB type.
ISNULL()	Supported, with differences	In GaussDB, the return value is **t** or **f** of the BOOLEAN type. In MySQL, the return value is **1** or **0** of the INT type.

2.2.2 Date and Time Functions

The date and time functions in GaussDB in B-compatible mode, with the same behavior as MySQL, are described as follows:

- Functions may use time expressions as their input parameters.

  Time expressions mainly include text, datetime, date, and time. Besides, all types that can be implicitly converted to time expressions can be input parameters. For example, a number can be implicitly converted to text and then used as a time expression.

  However, different functions take effect in different ways. For example, the datediff function calculates only the difference between dates. Therefore, time

expressions are parsed as the date type. The timestampdiff function parses time expressions as date, time, or datetime based on the **unit** parameter before calculating the time difference.

- The input parameters of functions may contain an invalid date.

  Generally, the supported date and datetime ranges are the same as those of MySQL. The value of date ranges from '0000-01-01' to '9999-12-31', and the value of datetime ranges from '0000-01-01 00:00:00' to '9999-12-31 23:59:59'. Although GaussDB supports larger date and datetime ranges than MySQL, dates out of range are still considered invalid.

  In most cases, time functions report an alarm and return NULL if the input date is invalid, unless the invalid date can be converted by CAST.

- Separators for input parameters of functions:

  For a time function, all non-digit characters are regarded as separators when input parameters are processed. The standard format is recommended: Use hyphens (-) to separate year, month, and day, use colons (:) to separate hour, minute, and second, and use a period (.) before milliseconds.

  Error-prone scenario: When **SELECT timestampdiff(hour, '2020-03-01 00:00:00', '2020-02-28 00:00:00+08');** is executed in a B-compatible database, the time function does not automatically calculate the time zone. Therefore, +08 is not identified as the time zone. Instead, + is used as the separator for calculation as seconds.

Most function scenarios of GaussDB date and time functions are the same as those of MySQL, but there are still differences. Some differences are as follows:

- If an input parameter of a function is NULL, the function returns NULL, and no warning or error is reported. These functions include:

  from_days, date_format, str_to_date, datediff, timestampdiff, date_add, subtime, month, time_to_sec, to_days, to_seconds, dayname, monthname, convert_tz, sec_to_time, addtime, adddate, date_sub, timediff, last_day, weekday, from_unixtime, unix_timestamp, subdate, day, year, weekofyear, dayofmonth, dayofyear, week, yearweek, dayofweek, time_format, hour, minute, second, microsecond, quarter, get_format, extract, makedate, period_add, timestampadd, period_diff, utc_time, utc_timestamp, maketime, and curtime.

  Example:

  ```
  gaussdb=# SELECT day(null);
   day
  -----

  (1 row)
  ```

- Some functions with pure numeric input parameters are different from those of MySQL. Numeric input parameters without quotation marks are converted into text input parameters for processing.

  Example:

  ```
  gaussdb=# SELECT day(19231221.123141);
  WARNING:  Incorrect datetime value: "19231221.123141"
  CONTEXT:  referenced column: day
   day
  -----

  (1 row)
  ```


- Time and date calculation functions are adddate, subdate, date_add, and date_sub. If the calculation result is a date, the supported range is [0000-01-01,9999-12-31]. If the calculation result is a date and time, the supported range is [0000-01-01 00:00:00.000000,9999-12-31 23:59:59.999999]. If the calculation result exceeds the supported range, an ERROR is reported in strict mode, or a WARNING is reported in loose mode. If the date result after calculation is within the range [0000-01-01,0001-01-01], GaussDB returns the result normally. MySQL returns '0000-00-00'.

  Example:

  ```
  gaussdb=# SELECT subdate('0000-01-01', interval 1 hour);
  ERROR:  Datetime function: datetime field overflow
  CONTEXT:  referenced column: subdate

  gaussdb=# SELECT subdate('0001-01-01', interval 1 day);
    subdate
  -------------
   0000-12-31

  (1 row)
  ```

- If the input parameter of the date or datetime type of the date and time function contains month 0 or day 0, the value is invalid. In strict mode, an error is reported. In loose mode, if the input is a character string or number, a warning is reported. If the input is of the date or datetime type, the system processes the input as December of the previous year or the last day of the previous month.

  If the type of the CAST function is converted to date or datetime, an error is reported in strict mode. In loose mode, no warning is reported. Instead, the system processes the input as December of the previous year or the last day of the previous month. Pay attention to this difference. MySQL outputs the value as it is, even if the year, month, and day are set to **0**.

  Example:

  ```
  gaussdb=# SELECT adddate('2023-01-00', 1);-- Strict mode
  ERROR:  Incorrect datetime value: "2023-01-00"
  CONTEXT:  referenced column: adddate

  gaussdb=# SELECT adddate('2023-01-00', 1); -- Loose mode
  WARNING:  Incorrect datetime value: "2023-01-00"
  CONTEXT:  referenced column: adddate
   adddate
  ---------

  (1 row)

  gaussdb=# SELECT adddate(date'2023-00-00', 1); -- Loose mode
    adddate
  ------------
   2022-12-01
  (1 row)

  gaussdb=# SELECT cast('2023/00/00' as date); -- Loose mode
      date
  ------------
   2022-11-30
  (1 row)

  gaussdb=# SELECT cast('0000-00-00' as datetime); -- Loose mode
        timestamp
  ---------------------
   0000-00-00 00:00:00
  (1 row)
  ```


- If the input parameter of the function is of the numeric data type, no error is reported in the case of invalid input, and the input parameter is processed as 0.

  Example:

```
gaussdb=# SELECT from_unixtime('aa');
    from_unixtime
---------------------
 1970-01-01 08:00:00
(1 row)
```

- A maximum of six decimal places are allowed. Decimal places with all 0s are not allowed.

  Example:

```
gaussdb=# SELECT from_unixtime('1234567899.00000');
    from_unixtime
---------------------
 2009-02-14 07:31:39
(1 row)
```

- If the time function parameter is a character string, the result is correct only when the year, month, and day are separated by a hyphen (-) and the hour, minute, and second are separated by a colon (:).

  Example:

```
gaussdb=# SELECT adddate('20-12-12',interval 1 day);
  adddate
------------
 2020-12-13
(1 row)
```

- If the return value of a function is of the varchar type in MySQL, the return value of the function is of the text type in GaussDB.

```
-- Return value of a function in GaussDB.
gaussdb=# SELECT pg_typeof(adddate('2023-01-01', 1));
 pg_typeof
-----------
 text
(1 row)

-- Return value of a function in MySQL.
mysql> CREATE VIEW v1 AS SELECT adddate('2023-01-01', 1);
Query OK, 0 rows affected (0.00 sec)

mysql> DESC v1;
+--------------------------+-------------+------+-----+---------+-------+
| Field                    | Type        | Null | Key | Default | Extra |
+--------------------------+-------------+------+-----+---------+-------+
| adddate('2023-01-01', 1) | varchar(29) | YES  |     | NULL    |       |
+--------------------------+-------------+------+-----+---------+-------+
1 row in set (0.00 sec)
```

Table 2-11 Date and time functions

MySQL	GaussDB	Difference
ADDDATE()	Supported, with differences	The performance of this function is different from that of MySQL due to interval expression differences. For details, see **INTERVAL**.

MySQL	GaussDB	Difference
ADDTIME()	Supported, with differences	• MySQL returns NULL if the second input parameter is a string in the DATETIME format. GaussDB can calculate the value. • The value range of an input parameter is ['0001-01-01 00:00:00', 9999-12-31 23:59:59.999999]. • If the first parameter of the ADDTIME function in MySQL is a dynamic parameter (for example, in a prepared statement), the return type is TIME. Otherwise, the parse type of the function is derived from the parse type of the first parameter. The return value rules of the ADDTIME function in GaussDB are as follows: – The first input parameter is of the date type, the second input parameter is of the date type, and the return value is of the time type. – The first input parameter is of the date type, the second input parameter is of the text type, and the return value is of the text type. – The first input parameter is of the date type, the second input parameter is of the datetime type, and the return value is of the time type. – The first input parameter is of the date type, the second input parameter is of the time type, and the return value is of the time type. – The first input parameter is of the text type, the second input parameter is of the date type, and the return value is of the text type. – The first input parameter is of the text type, the second input parameter is of the text type, and the return value is of the text type. – The first input parameter is of the text type, the second input parameter is of the datetime type, and the return value is of the text type. – The first input parameter is of the text type, the second input parameter is of the time type, and the return value is of the text type.

MySQL	GaussDB	Difference
		– The first input parameter is of the datetime type, the second input parameter is of the date type, and the return value is of the datetime type. – The first input parameter is of the datetime type, the second input parameter is of the text type, and the return value is of the text type. – The first input parameter is of the datetime type, the second input parameter is of the datetime type, and the return value is of the datetime type. – The first input parameter is of the datetime type, the second input parameter is of the time type, and the return value is of the datetime type. – The first input parameter is of the time type, the second input parameter is of the date type, and the return value is of the time type. – The first input parameter is of the time type, the second input parameter is of the text type, and the return value is of the text type. – The first input parameter is of the time type, the second input parameter is of the datetime type, and the return value is of the time type. – The first input parameter is of the time type, the second input parameter is of the time type, and the return value is of the time type.
CONVERT_TZ()	Supported	-
CURDATE()	Supported	-
CURRENT_DATE(), CURRENT_DATE	Supported	-

MySQL	GaussDB	Difference
CURRENT_TIME(), CURRENT_TIME	Supported, with differences	The time value (after the decimal point) output by precision is rounded off in GaussDB and directly truncated in MySQL. The trailing 0s of the time value (after the decimal point) output by precision are not displayed in GaussDB but displayed in MySQL. GaussDB supports only an integer value within the range of [0,6] as the precision of the returned time. For other values, an error is reported. In MySQL, a precision value within [0,6] is valid, but an input integer value is divided by 256 to get a remainder. For example, if the input integer value is **257**, the time value with precision 1 is returned.
CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP	Supported, with differences	The time value (after the decimal point) output by precision is rounded off in GaussDB and directly truncated in MySQL. The trailing 0s of the time value (after the decimal point) output by precision are not displayed in GaussDB but displayed in MySQL. GaussDB supports only an integer value within the range of [0,6] as the precision of the returned time. If the input integer value is greater than 6, an alarm is generated and the time value is output based on the precision 6. In MySQL, a precision value within [0,6] is valid, but an input integer value is divided by 256 to get a remainder. For example, if the input integer value is **257**, the time value with precision 1 is returned.
CURTIME()	Supported, with differences	In GaussDB, if a character string or a non-integer value is entered, the value is implicitly converted into an integer and then the precision is verified. If the value is beyond the [0,6] range, an error is reported. If the value is within the range, the time value is output normally. In MySQL, an error is reported. The time value (after the decimal point) output by precision is rounded off in GaussDB and directly truncated in MySQL. The trailing 0s of the time value (after the decimal point) output by precision are not displayed in GaussDB but displayed in MySQL. GaussDB supports only an integer value within the range of [0,6] as the precision of the returned time. For other values, an error is reported. In MySQL, a precision value within [0,6] is valid, but an input integer value is divided by 256 to get a remainder. For example, if the input integer value is **257**, the time value with precision 1 is returned.

MySQL	GaussDB	Difference
YEARWEEK()	Supported	-
DATE_ADD()	Supported, with differences	The performance of this function is different from that of MySQL due to interval expression differences. For details, see **INTERVAL**.
DATE_FORMAT()	Supported	-
DATE_SUB()	Supported, with differences	The performance of this function is different from that of MySQL due to interval expression differences. For details, see **INTERVAL**.
DATEDIFF()	Supported	-
DAY()	Supported	-
DAYNAME()	Supported	-
DAYOFMONTH()	Supported	-
DAYOFWEEK()	Supported	-
DAYOFYEAR()	Supported	-
EXTRACT()	Supported	-
FROM_DAYS()	Supported	-
FROM_UNIXTIME()	Supported	-
GET_FORMAT()	Supported	-
HOUR()	Supported	-
LAST_DAY	Supported	-

MySQL	GaussDB	Difference
LOCALTIME(), LOCALTIME	Supported, with differences	The time value (after the decimal point) output by precision is rounded off in GaussDB and directly truncated in MySQL. The trailing 0s of the time value (after the decimal point) output by precision are not displayed in GaussDB but displayed in MySQL. GaussDB supports only an integer value within the range of [0,6] as the precision of the returned time. For other integer values, an error is reported. In MySQL, a precision value within [0,6] is valid, but an input integer value is divided by 256 to get a remainder. For example, if the input integer value is **257**, the time value with precision 1 is returned.
LOCALTIMESTAMP, LOCALTIMESTAMP()	Supported, with differences	The time value (after the decimal point) output by precision is rounded off in GaussDB and directly truncated in MySQL. The trailing 0s of the time value (after the decimal point) output by precision are not displayed in GaussDB but displayed in MySQL. GaussDB supports only an integer value within the range of [0,6] as the precision of the returned time. If the input integer value is greater than 6, an alarm is generated and the time value is output based on the precision 6. In MySQL, a precision value within [0,6] is valid, but an input integer value is divided by 256 to get a remainder. For example, if the input integer value is **257**, the time value with precision 1 is returned.
MAKEDATE()	Supported	-
MAKETIME()	Supported, with differences	When the input parameter is NULL, GaussDB does not support self-nesting of the maketime function, but MySQL supports.
MICROSECOND()	Supported	-
MINUTE()	Supported	-
MONTH()	Supported	-
MONTHNAME()	Supported	-

MySQL	GaussDB	Difference
NOW()	Supported, with differences	The time value (after the decimal point) output by precision is rounded off in GaussDB and directly truncated in MySQL. The trailing 0s of the time value (after the decimal point) output by precision are not displayed in GaussDB but displayed in MySQL. GaussDB supports only an integer value within the range of [0,6] as the precision of the returned time. If the input integer value is greater than 6, an alarm is generated and the time value is output based on the precision 6. In MySQL, a precision value within [0,6] is valid, but an input integer value is divided by 256 to get a remainder. For example, if the input integer value is **257**, the time value with precision 1 is returned.
PERIOD_ADD()	Supported, with differences	If the input parameter **period** or result is less than 0, GaussDB reports an error by referring to the performance in MySQL 8.0.x. Integer wrapping occurs in MySQL 5.7. As a result, the calculation result is abnormal.
PERIOD_DIFF()	Supported, with differences	If the input parameter or result is less than 0, GaussDB reports an error by referring to the performance in MySQL 8.0.x. Integer wrapping occurs in MySQL 5.7. As a result, the calculation result is abnormal.
QUARTER()	Supported	-
SEC_TO_TIME()	Supported	-
SECOND()	Supported	-
STR_TO_DATE()	Supported, with differences	GaussDB returns values of the text type, while MySQL returns values of the datetime or date type.
SUBDATE()	Supported, with differences	The performance of this function is different from that of MySQL due to interval expression differences. For details, see **INTERVAL**.

MySQL	GaussDB	Difference
SUBTIME()	Supported, with differences	• MySQL returns NULL if the second input parameter is a string in the DATETIME format. GaussDB can calculate the value. • The value range of an input parameter is ['0001-01-01 00:00:00', 9999-12-31 23:59:59.999999]. • If the first parameter of the SUBTIME function in MySQL is a dynamic parameter (for example, in a prepared statement), the return type is TIME. Otherwise, the parse type of the function is derived from the parse type of the first parameter. The return value rules of the SUBTIME function in GaussDB are as follows: – The first input parameter is of the date type, the second input parameter is of the date type, and the return value is of the time type. – The first input parameter is of the date type, the second input parameter is of the text type, and the return value is of the text type. – The first input parameter is of the date type, the second input parameter is of the datetime type, and the return value is of the time type. – The first input parameter is of the date type, the second input parameter is of the time type, and the return value is of the time type. – The first input parameter is of the text type, the second input parameter is of the date type, and the return value is of the text type. – The first input parameter is of the text type, the second input parameter is of the text type, and the return value is of the text type. – The first input parameter is of the text type, the second input parameter is of the datetime type, and the return value is of the text type. – The first input parameter is of the text type, the second input parameter is of the time type, and the return value is of the text type.

MySQL	GaussDB	Difference
		– The first input parameter is of the datetime type, the second input parameter is of the date type, and the return value is of the datetime type. – The first input parameter is of the datetime type, the second input parameter is of the text type, and the return value is of the text type. – The first input parameter is of the datetime type, the second input parameter is of the datetime type, and the return value is of the datetime type. – The first input parameter is of the datetime type, the second input parameter is of the time type, and the return value is of the datetime type. – The first input parameter is of the time type, the second input parameter is of the date type, and the return value is of the time type. – The first input parameter is of the time type, the second input parameter is of the text type, and the return value is of the text type. – The first input parameter is of the time type, the second input parameter is of the datetime type, and the return value is of the time type. – The first input parameter is of the time type, the second input parameter is of the time type, and the return value is of the time type.
SYSDATE()	Supported, with differences	In MySQL, an integer input value is wrapped when it reaches **255** (maximum value of a one-byte integer value), while GaussDB does not.
YEAR()	Supported	-
TIME_FORMAT()	Supported	-
TIME_TO_SEC()	Supported	-
TIMEDIFF()	Supported	-
WEEKOFYEAR()	Supported	-

MySQL	GaussDB	Difference
TIMESTAMPADD()	Supported	-
TIMESTAMPDIFF()	Supported	-
TO_DAYS()	Supported	-
TO_SECONDS()	Supported	-
UNIX_TIMESTAMP()	Supported, with differences	GaussDB returns values of the numeric type, while MySQL returns values of the int type.
UTC_DATE()	Supported, with differences	• MySQL supports calling without parentheses, but GaussDB does not. In MySQL, an integer input value is wrapped when it reaches **255** (maximum value of a one-byte integer value). • MySQL input parameters support only integers ranging from 0 to 6. GaussDB supports input parameters that can be implicitly converted to integers ranging from 0 to 6.
UTC_TIME()	Supported, with differences	
UTC_TIMESTAMP()	Supported, with differences	
WEEK()	Supported	-
WEEKDAY()	Supported	-

2.2.3 String Functions

Table 2-12 String functions

MySQL	GaussDB	Difference
BIN()	Supported, with differences	In GaussDB, the types supported by function input parameters are as follows: • Integer types: tinyint, smallint, mediumint, int, and bigint. • Unsigned integer types: tinyint unsigned, smallint unsigned, int unsigned, and bigint unsigned. • Character and text types: char, varchar, tinytext, text, mediumtext, and longtext. Only numeric integer strings are supported, and the integer range is within the bigint range. • Floating-point types: float, real, and double. • Fixed-point types: numeric, decimal, and dec. • Boolean type: bool.
CONCAT()	Supported, with differences	The data type of the return value of CONCAT is always text regardless of the data type of the parameter. However, in MySQL, if CONCAT contains binary parameters, the return value is binary.
CONCAT_WS()	Supported, with differences	The data type of the return value of CONCAT_WS is always text regardless of the data type of the parameter. However, in MySQL, if CONCAT_WS contains binary parameters, the return value is binary. In other cases, the return value is a string.

MySQL	GaussDB	Difference
ELT()	Supported, with differences	• In GaussDB, the types supported by function input parameter 1 are as follows: – Integer types: tinyint, smallint, mediumint, int, and bigint. – Unsigned integer types: tinyint unsigned, smallint unsigned, and int unsigned. – Character and text types: char, varchar, tinytext, text, mediumtext, and longtext. Only numeric integer strings are supported, and the integer range is within the bigint range. – Floating-point types: float, real, and double. – Fixed-point types: numeric, decimal, and dec. – Boolean type: bool. • In GaussDB, the types supported by function input parameter 2 are as follows: – Integer types: tinyint, smallint, mediumint, int, and bigint. – Unsigned integer types: tinyint unsigned, smallint unsigned, int unsigned, and bigint unsigned. – Character and text types: char, varchar, tinytext, text, mediumtext, and longtext. – Floating-point types: float, real, and double. – Fixed-point types: numeric, decimal, and dec. – Boolean type: bool. – Large object types: tinyblob, blob, mediumblob, and longblob. – Date types: datetime, timestamp, date, and time.
FIELD()	Supported, with differences	When function input parameters range from the maximum bigint value to the maximum bigint unsigned value, incompatibility occurs. When function input parameters are of the float(m, d), double(m, d), or real(m, d) type, the precision is higher and incompatibility occurs.

MySQL	GaussDB	Difference
FIND_IN_SET()	Supported, with differences	When the database encoding is set to **'SQL_ASCII'**, the default case sensitivity rule is not supported. That is, if no character set rule is specified, uppercase and lowercase letters are treated as distinct.
INSERT()	Supported, with differences	• The range of input parameters of the Int64 type is from –9223372036854775808 to +9223372036854775807. If a value is out of range, an error is reported. MySQL does not limit the range of input parameters of the numeric type. If an exception occurs, an alarm is generated, indicating that the value is set to the upper or lower limit. • The maximum length of the input parameter of the text type is 2^30 – 5 bytes, and the maximum length of the input parameter of the bytea type is 2^30 – 512 bytes. • If any of the **s1** and **s2** parameters is of the bytea type and the result contains invalid characters, the displayed result may be different from that of MySQL, but the character encoding is the same as that of MySQL.
LOCATE()	Supported, with differences	When input parameter 1 is of the bytea type and input parameter 2 is of the text type, the behavior of GaussDB is different from that of MySQL.

MySQL	GaussDB	Difference
MAKE_SET()	Supported, with differences	• When the **bits** parameter is an integer, the maximum range is int128, which is smaller than the MySQL range. • When the **bits** parameter is of the date type (datetime, timestamp, date, or time), it is not supported because the conversion from the date type to the integer type is different from that in MySQL. • GaussDB and MySQL are inherently different in the bit and Boolean types, causing different returned results. When the **bits** input parameter is of the Boolean type, and the **str** input parameter is of the bit or Boolean type, they are not supported. • When the **bits** input parameter is of the character string or text type, only the pure integer format is supported. In addition, the value range of pure integers is limited to bigint. • The integer value of the **str** input parameter exceeds the range from 81 negative nines to 81 positive nines. The return value is different from that of MySQL. • When the **str** input parameter is expressed in scientific notation, trailing zeros are displayed in GaussDB but not displayed in MySQL. This is an inherent difference.

MySQL	GaussDB	Difference
QUOTE()	Supported, with differences	• If the **str** character string contains "\Z", "\r", "\%", or "_", GaussDB does not escape it, which is different from MySQL. The slash followed by digits may also cause differences, for example, "\563". This function difference is the escape character difference between GaussDB and MySQL. • The output format of "\b" in the **str** character string is different from that in MySQL. This is an inherent difference between GaussDB and MySQL. • If the **str** character string contains "\0", GaussDB cannot identify the character because the UTF-8 character set cannot identify the character. As a result, the input fails. This is an inherent difference between GaussDB and MySQL. • If **str** is of the bit or Boolean type, this type is not supported because it is different in GaussDB and MySQL. • GaussDB supports a maximum of 1 GB data transfer. The maximum length of the **str** input parameter is 536870908 bytes, and the maximum size of the result string returned by the function is 1 GB. • The integer value of the **str** input parameter exceeds the range from 81 negative nines to 81 positive nines. The return value is different from that of MySQL. • When the **str** input parameter is expressed in scientific notation, trailing zeros are displayed in GaussDB but not displayed in MySQL. This is an inherent difference.

MySQL	GaussDB	Difference
SPACE()	Supported, with differences	• GaussDB allows an input parameter of no more than 1073741818 bytes. If the length exceeds the limit, an empty string is returned. By default, MySQL allows an input parameter of no more than 4194304 bytes. If the length exceeds the limit, an alarm is generated. • In GaussDB, the types supported by function input parameters are as follows: – Integer types: tinyint, smallint, mediumint, int, and bigint. – Unsigned integer types: tinyint unsigned, smallint unsigned, and int unsigned. – Character and text types: char, varchar, tinytext, text, mediumtext, and longtext. Only numeric integer strings are supported, and the integer range is within the bigint range. – Floating-point types: float, real, and double. – Fixed-point types: numeric, decimal, and dec. – Boolean type: bool.
SUBSTR()	Supported.	-
SUBSTRING()	Supported.	-
SUBSTRING_INDEX()	Supported.	-
STRCMP()	Supported, with differences	• In GaussDB, the types supported by function input parameters are as follows: – Character types: char, varchar, nvarchar2, and text – Binary type: BYTEA – Value type: tinying [unsigned], smallint [unsigned], integer [unsigned], bigint [unsigned], float4, float8, and numeric – Date and time type: date, time without time zone, datetime, and timestamptz • For the floating-point type in the numeric type, the precision may be different from that in MySQL due to different connection parameter settings. Therefore, this scenario is not recommended, or the NUMERIC type is used instead.

MySQL	GaussDB	Difference
SHA() / SHA1()	Supported.	-
SHA2()	Supported.	-

2.2.4 Forced Conversion Functions

Table 2-13 Forced conversion functions

MySQL	GaussDB	Difference
CAST()	Supported, with differences	The data type conversion rules and supported conversion types are subject to the conversion scope and rules supported by GaussDB.
CONVERT()	Supported, with differences	The data type conversion rules and supported conversion types are subject to the conversion scope and rules supported by GaussDB.

2.2.5 Encryption Functions

Table 2-14 Encryption functions

MySQL	GaussDB	Difference
AES_DECRYPT()	Supported.	-
AES_ENCRYPT()	Supported.	-

2.2.6 Information Functions

Table 2-15 Information functions

MySQL	GaussDB	Difference
LAST_INSERT_ID()	Supported	-

2.2.7 JSON Functions

JSON function differences:

- If you add escape characters as input parameters to JSON functions and other functions that allow character inputs, the processing is different from that in MySQL by default. To be compatible with MySQL, set the GUC parameter **standard_conforming_strings** to **off**. In this case, the processing of escape characters is compatible with MySQL, but a warning is generated for non-standard character input. The escape characters \t and \u and escape digits are different from those in MySQL. The JSON_UNQUOTE () function is compatible with MySQL. Even if the GUC parameter is not set, no alarm is generated.
- When processing an ultra-long number (the number contains more than 64 characters), the JSON function of GaussDB parses the number as a DOUBLE and uses scientific notation for counting. The input parameters of the non-JSON type are the same as those of MySQL. However, when input parameters of the JSON type are used, the JSON type is not completely compatible with MySQL. As a result, differences occur in this scenario. MySQL displays complete numbers. (When the number length exceeds 82, MySQL displays an incorrect result.) GaussDB still parses an ultra-long number into a double-precision value. Long numbers are stored using floating-point numbers. During calculation, precision loss occurs in both GaussDB and MySQL. Therefore, you are advised to use character strings to store long numbers.

```
gaussdb=# SELECT json_insert('[1, 4,
99999999999999999999999999999999999999999999999999999999999999999999999999999]','$[6]',json
_insert('[1,4]','$[5]',99999999999999999999999999999999999999999999999999999999999999999999999999999
999999));
        json_insert
-------------------------------
 [1, 4, 1e+74, [1, 4, 1e+74]]
(1 row)
```

Table 2-16 JSON functions

MySQL	GaussDB	Difference
JSON_APPEND()	Supported.	-
JSON_ARRAY()	Supported.	-
JSON_ARRAY_APPEND()	Supported.	-
JSON_ARRAY_INSERT()	Supported.	-
JSON_CONTAINS()	Supported.	-
JSON_CONTAINS_PATH()	Supported.	-
JSON_DEPTH()	Supported.	-
JSON_EXTRACT()	Supported.	-

MySQL	GaussDB	Difference
JSON_INSERT()	Supported.	-
JSON_KEYS()	Supported.	-
JSON_LENGTH()	Supported.	-
JSON_MERGE()	Supported.	-
JSON_OBJECT()	Supported.	-
JSON_QUOTE()	Supported.	-
JSON_REMOVE()	Supported.	-
JSON_REPLACE()	Supported.	-
JSON_SEARCH()	Supported, with differences	GaussDB returns values of the text type, while MySQL returns values of the JSON type.
JSON_SET()	Supported.	-
JSON_TYPE()	Supported.	-
JSON_UNQUOTE()	Supported.	-
JSON_VALID()	Supported.	-

2.2.8 Aggregate Functions

Table 2-17 Aggregate functions

MySQL	GaussDB	Difference
GROUP_CONCAT()	Supported, with differences	• If the **group_concat** parameter contains both the DISTINCT and ORDER BY syntaxes, all expressions following ORDER BY must be in the DISTINCT expression. • **group_concat(... order by** *Number***)** does not indicate the sequence of the parameter. The number is only a constant expression, which is equivalent to no sorting. • The data type of the return value of **group_concat** is always text regardless of the data type of the parameter. For MySQL, if **group_concat** contains binary parameters, the return value is binary. In other cases, the return value is a character string. If the return value length is greater than 512 bytes, the data type is a character large object or binary large object. • The value of **group_concat_max_len** ranges from 0 to 1073741823. The maximum value is smaller than that of MySQL.

MySQL	GaussDB	Difference
DEFAULT()	Supported, with differences	• The default value of a column is an array. GaussDB returns an array. MySQL does not support the array type. • GaussDB columns are hidden columns (such as **xmin** and **cmin**). The default function returns a null value. • GaussDB supports default values of partitioned tables, temporary tables, and multi-table join query. • GaussDB supports the query of nodes whose column names contain character string values (indicating names) and A_Star nodes (indicating that asterisks [*] appear), for example, **default(tt.t4.id)** and **default(tt.t4.*)**. For invalid query column names and A_Star nodes, the error information reported by GaussDB is different from that reported by MySQL. • When the default value of a column is created in GaussDB, the range of the column type is not verified. As a result, an error may be reported when the default function is used. • If the default value of a column is a function expression, the default function in GaussDB returns the calculated value of the default expression of the column during table creation. The default function in MySQL returns NULL.

2.2.9 Numeric Operation Functions

Table 2-18 Numeric operation functions

MySQL	GaussDB	Difference
log2()	Supported, with differences	• The display of decimal places is different from that in MySQL. Due to the limitation of the GaussDB floating-point data type, the **extra_float_digits** parameter is used to control the number of decimal places to be displayed. • Due to the internal processing difference of the input precision, the calculation results of GaussDB and MySQL are different. • The following data types are supported: – Integer types: bigint, int16, int, smallint, and tinyint. – Unsigned integer types: bigint unsigned, integer unsigned, smallint unsigned, and tinyint unsigned. – Floating-point number type: numeric and real. – Character string type: character, character varying, clob, and text. Only numeric integer strings are supported. – SET type. – NULL type.

MySQL	GaussDB	Difference
log10()	Supported, with differences	• The display of decimal places is different from that in MySQL. Due to the limitation of the GaussDB floating-point data type, the **extra_float_digits** parameter is used to control the number of decimal places to be displayed. • Due to the internal processing difference of the input precision, the calculation results of GaussDB and MySQL are different. • The following data types are supported: – Integer types: bigint, int16, int, smallint, and tinyint. – Unsigned integer types: bigint unsigned, integer unsigned, smallint unsigned, and tinyint unsigned. – Floating-point number type: numeric and real. – Character string type: character, character varying, clob, and text. Only numeric integer strings are supported. – SET type. – NULL type.

2.2.10 Other Functions

Table 2-19 Other functions

MySQL	GaussDB	Difference
UUID()	Supported	-
UUID_SHORT()	Supported	-

2.3 Operators

GaussDB is compatible with most MySQL operators, but there are some differences. Unless otherwise specified, the operator behavior in B-compatible mode is the native GaussDB behavior by default.

Table 2-20 Operators

MySQL	GaussDB	Difference
NULL-safe equal (<=>)	Supported.	-
[NOT] REGEXP	Supported, with differences	• If the GUC parameter **b_format_dev_version** is set to **'s2'** and a pattern string with escape characters such as "\\a", "\\d", "\\e", "\\n", "\\Z", or "\\u" is matched with source character strings "\a", "\d", "\e", "\n", "\Z", or "\u", the behavior of GaussDB is different from that of MySQL 5.7 but the same as that of MySQL 8.0. • When the GUC parameter **b_format_dev_version** is set to **'s2'**, "\b" in GaussDB can match "\\b", but the matching will fail in MySQL. • If the input parameter of the pattern string is invalid with only the right parenthesis ()), GaussDB and MySQL 5.7 will report an error, but MySQL 8.0 will not. • In the rule of matching the de|abc sequence with de or abc, when there are empty values on the left and right of the pipe symbol (|), MySQL 5.7 will report an error, but GaussDB and MySQL 8.0 will not. • The regular expression of the tab character "\t" can match the character class [:blank:] in GaussDB and MySQL 8.0 but cannot in MySQL 5.7. • GaussDB supports non-greedy pattern matching. That is, the number of matching characters is as small as possible. A question mark (?) is added after some special characters, for example, ?? *? +? {n}? {n,}? {n,m}? MySQL 5.7 does not support non-greedy pattern matching, and the error message "Got error 'repetition-operator operand invalid' from regexp" is displayed. MySQL 8.0 already supports this function. • In the binary character set, the text and BLOB types will be converted to the bytea type. However, the REGEXP operator does not support the bytea type. Therefore, the matching will fail.
[NOT] RLIKE	Supported, with differences	Same as [NOT] REGEXP.

2.4 Character Sets

GaussDB allows you to specify the following character sets for databases, schemas, tables, or columns.

Table 2-21 Character sets

MySQL	GaussDB
utf8mb4	Supported
gbk	Supported
gb18030	Supported
utf8	Supported
binary	Supported.

2.5 Collation Rules

GaussDB allows you to specify the following collation rules for databases, schemas, tables, or columns.

NOTE

Differences in collation rules:

- Currently, only the character string type and some binary types support the specified collation rules. You can check whether the **typcollation** attribute of a type in the pg_type system catalog is not 0 to determine whether the type supports the collation. The collation can be specified for all types in MySQL. However, collation rules are meaningless except those for character strings and binary types.
- The current collation rules (except binary) can be specified only when the corresponding character set is the same as the database-level character set. In GaussDB, the character set must be the same as the database character set, and multiple character sets cannot be used together in a table.
- The default collation of the utf8mb4 character set is utf8mb4_general_ci, which is the same as that in MySQL 5.7.
- In GaussDB, utf8 and utf8mb4 are the same character set.

Table 2-22 Collation rules

MySQL	GaussDB
utf8mb4_general_ci	Supported.
utf8mb4_unicode_ci	Supported.
utf8mb4_bin	Supported.
gbk_chinese_ci	Supported.

MySQL	GaussDB
gbk_bin	Supported.
gb18030_chinese_ci	Supported.
gb18030_bin	Supported.
binary	Supported.
utf8mb4_0900_ai_ci	Supported.
utf8_general_ci	Supported.
utf8_bin	Supported.

2.6 Expressions

GaussDB is compatible with most MySQL expressions, but there are some differences. If not listed, the expression behavior is the native GaussDB behavior by default.

Table 2-23 Expressions

MySQL	GaussDB
User-defined variable @var_name	Partially supported
Global variable @@var_name	Partially supported

2.7 SQL

2.7.1 DDL

Table 2-24 DDL syntax compatibility

MySQL Function	Syntax	GaussDB Implementation Difference
Create primary keys, UNIQUE indexes, and foreign keys during table creation and modification.	ALTER TABLE and CREATE TABLE	• GaussDB does not support the UNIQUE INDEX|KEY index_name syntax. An error will be reported when the UNIQUE INDEX| KEY index_name syntax is used. • When a constraint is created as a global secondary index and USING BTREE is specified in the SQL statement, the underlying index is created as UB-tree. • When the table joined with the constraint is Ustore and USING BTREE is specified in the SQL statement, the underlying index is created as UB-tree.

MySQL Function	Syntax	GaussDB Implementation Difference
Support auto-increment columns.	ALTER TABLE and CREATE TABLE	• It is recommended that the auto-increment column be the first column of a non-global secondary index. Otherwise, an alarm is generated when a table is created, and errors may occur when some operations are performed on a table that contains auto-increment columns, for example, ALTER TABLE EXCHANGE PARTITION. The auto-increment column in MySQL must be the first column of the index. • In the syntax AUTO_INCREMENT = value, **value** must be a positive number less than 2^127. MySQL does not verify the value. • An error occurs if the auto-increment continues after an auto-increment value reaches the maximum value of a column data type. In MySQL, errors or warnings may be generated during auto-increment, and sometimes auto-increment continues until the maximum value is reached. • GaussDB does not support the *innodb_autoinc_lock_mode* system variable, but when its GUC parameter **auto_increment_cache** is set to **0**, the behavior of inserting auto-increment columns in batches is similar to that when the MySQL system variable

MySQL Function	Syntax	GaussDB Implementation Difference
		innodb_autoinc_lock_mode is set to **1**. • When 0s, NULLs, and definite values are imported or batch inserted into auto-increment columns, the auto-increment values inserted after an error occurs in GaussDB may not be the same as those in MySQL. The **auto_increment_cache** parameter is provided to control the number of reserved auto-increment values. • When auto-increment is triggered by parallel import or insertion of auto-increment columns, the cache value reserved for each parallel thread is used only in the thread. If the cache value is not used up, the values of auto-increment columns in the table are discontinuous. The auto-increment value generated by parallel insertion cannot be guaranteed to be the same as that generated in MySQL. • When auto-increment columns are batch inserted into a local temporary table, no auto-increment value is reserved. In normal scenarios, auto-increment values are not discontinuous. In MySQL, the auto-increment result of an auto-increment column in a temporary table is the same as that in an ordinary table.

MySQL Function	Syntax	GaussDB Implementation Difference
		• The SERIAL data type of GaussDB is an original auto-increment column, which is different from the **AUTO_INCREMENT** column. The SERIAL data type of MySQL is the **AUTO_INCREMENT** column. • The value of **auto_increment_offset** cannot be greater than that of **auto_increment_increment**. Otherwise, an error occurs. MySQL allows it and states that **auto_increment_offset** will be ignored. • If a table has a primary key or index, the sequence in which the **ALTER TABLE** command rewrites table data may be different from that in MySQL. GaussDB rewrites table data based on the table data storage sequence, while MySQL rewrites table data based on the primary key or index sequence. As a result, the auto-increment sequence may be different. • When the **ALTER TABLE** command is used to add or modify auto-increment columns, the number of auto-increment values reserved for the first time is the number of rows in the table statistics. The number of rows in the statistics may not be the same as that in MySQL. • The return value of the last_insert_id function is a 128-bit integer.

MySQL Function	Syntax	GaussDB Implementation Difference
		• When auto-increment is performed in a trigger or user-defined function, the return value of last_insert_id is updated. MySQL does not update it. • If the values of the GUC parameters **auto_increment_offset** and **auto_increment_increment** are out of range, an error occurs. MySQL automatically changes the value to a boundary value. • If **sql_mode** is set to **no_auto_value_on_zero**, the auto-increment columns of the table are not subject to NOT NULL constraints. In GaussDB and MySQL, when the value of an auto-increment column is not specified, **NULL** will be inserted into the auto-increment column, but auto-increment is triggered for the former and not triggered for the latter.

MySQL Function	Syntax	GaussDB Implementation Difference
Support prefix indexes.	CREATE INDEX, ALTER TABLE, and CREATE TABLE	• The prefix length cannot exceed 2676. The actual length of the key value is restricted by the internal page. If a column contains multi-byte characters or an index has multiple keys, an error may be reported when the index line length exceeds the threshold. • In the CREATE INDEX syntax, the following keywords cannot be used as prefix keys for column names: COALESCE, EXTRACT, GREATEST, LEAST, LNNVL, NULLIF, NVL, NVL2, OVERLAY, POSITION, REGEXP_LIKE, SUBSTRING, TIMESTAMPDIFF, TREAT, TRIM, XMLCONCAT, XMLELEMENT, XMLEXISTS, XMLFOREST, XMLPARSE, XMLPI, XMLROOT, and XMLSERIALIZE. • Prefix keys are not supported in primary key indexes.
Specify character sets and collation rules.	ALTER SCHEMA, ALTER TABLE, CREATE SCHEMA, and CREATE TABLE	-
Add columns before the first column of a table or after a specified column during table modification.	ALTER TABLE	-
Compatible with the column name modification and the definition syntax.	ALTER TABLE	-

MySQL Function	Syntax	GaussDB Implementation Difference
Compatible with the EVENT syntax of a scheduled task.	ALTER EVENT, CREATE EVENT, DROP EVENT, and SHOW EVENTS	-
Create a partitioned table.	CREATE TABLE PARTITION and CREATE TABLE SUBPARTITION	-
Specify table-level and column-level comments during table creation and modification.	CREATE TABLE and ALTER TABLE	-
Specify index-level comments during index creation.	CREATE INDEX	-

MySQL Function	Syntax	GaussDB Implementation Difference
Exchange the partition data of an ordinary table and a partitioned table.	ALTER TABLE PARTITION	Differences in ALTER TABLE EXCHANGE PARTITION: • After **ALTER TABLE EXCHANGE PARTITION** is executed, the auto-increment columns are reset in MySQL, but in GaussDB, they are not reset and continue the auto-increment based on their old values. • If MySQL tables or partitions use tablespaces, data in partitions and ordinary tables cannot be exchanged. If GaussDB tables or partitions use different tablespaces, data in partitions and ordinary tables can still be exchanged. • MySQL does not verify the default values of columns. Therefore, data in partitions and ordinary tables can be exchanged even if the default values are different. GaussDB verifies the default values. If the default values are different, data in partitions and ordinary tables cannot be exchanged. • After the DROP COLUMN operation is performed on a partitioned table or an ordinary table in MySQL, if the table structure is still consistent, data can be exchanged between partitions and ordinary tables. In GaussDB, data can be exchanged between partitions and ordinary tables only when the deleted columns of ordinary tables and

MySQL Function	Syntax	GaussDB Implementation Difference
		partitioned tables are strictly aligned. • MySQL and GaussDB use different hash algorithms. Therefore, data stored in the same hash partition may be inconsistent. As a result, the exchanged data may also be inconsistent. • MySQL partitioned tables do not support foreign keys. If an ordinary table contains foreign keys or other tables reference foreign keys of an ordinary table, data in partitions and ordinary tables cannot be exchanged. GaussDB partitioned tables support foreign keys. If the foreign key constraints of two tables are the same, data in partitions and ordinary tables can be exchanged. If a GaussDB partitioned table does not contain foreign keys, an ordinary table is referenced by other tables, and the partitioned table is the same as the ordinary table, data in the partitioned table can be exchanged with that in the ordinary table.
Delete the primary key and foreign key constraints of a table.	ALTER TABLE DROP [PRIMARY | FOREIGN]KEY	-

MySQL Function	Syntax	GaussDB Implementation Difference
Support the CREATE TABLE... LIKE syntax.	CREATE TABLE ... LIKE	• In versions earlier than MySQL 8.0.16, CHECK constraints are parsed but their functions are ignored. In this case, CHECK constraints are not replicated. GaussDB supports replication of CHECK constraints. • When a table is created, all primary key constraint names in MySQL are fixed to **PRIMARY KEY**. GaussDB does not support replication of primary key constraint names. • When a table is created, MySQL supports replication of unique key constraint names, but GaussDB does not. • When a table is created, MySQL versions earlier than 8.0.16 do not have CHECK constraint information, but GaussDB supports replication of CHECK constraint names. • When a table is created, MySQL supports replication of index names, but GaussDB does not. • When a table is created across sql_mode, MySQL is controlled by the loose mode and strict mode. The strict mode may become invalid in GaussDB. For example, if the source table has the default value **"0000-00-00"**, GaussDB can create a table that contains the default value **"0000-00-00"** in "no_zero_date" strict mode, which means that

MySQL Function	Syntax	GaussDB Implementation Difference
		the strict mode is invalid. MySQL fails to create the table because it is controlled by the strict mode. • MySQL supports cross-database table creation, but GaussDB does not.
Compatible with syntax for changing table names.	ALTER TABLE tbl_name RENAME [TO | AS | =] new_tbl_name; RENAME {TABLE | TABLES} tbl_name TO new_tbl_name [, tbl_name2 TO new_tbl_name2, ...];	• The ALTER RENAME syntax in GaussDB supports only the function of changing the table name and cannot be coupled with other function operations. • In GaussDB, only the old table name column supports the schema.table_name format, and the new and old table names belong to the same schema. • GaussDB does not support renaming of old and new tables across schemas. However, if you have the permission, you can modify the names of tables in other schemas in the current schema. • The syntax for renaming multiple groups of tables in GaussDB supports renaming of all local temporary tables, but does not support the combination of local temporary tables and non-local temporary tables. • The RENAME TABLE verification sequence in GaussDB is different from that in MySQL. As a result, the error information is inconsistent.

MySQL Function	Syntax	GaussDB Implementation Difference
Create a partition.	ALTER TABLE [IF EXISTS] { table_name [*] | ONLY table_name | ONLY (table_name)} action [, ...]; action: move_clause | exchange_clause | row_clause | merge_clause | modify_clause | split_clause | add_clause | drop_clause | ilm_clause add_clause: ADD {{partition_less_than_item | partition_start_end_item | partition_list_item} | PARTITION({partition_less_than_item | partition_start_end_item | partition_list_item})}	• The following syntax cannot be used to add multiple partitions: `ALTER TABLE table_name ADD PARTITION (partition_definition1, partition_definition1,...);` • Only the original syntax for adding multiple partitions is supported. `ALTER TABLE table_name ADD PARTITION (partition_definition1), ADD PARTITION (partition_definition2[y1] ), ...;`

2.7.2 DML

Table 2-25 DML syntax compatibility

MySQL Function	Syntax Description	GaussDB Implementation Difference
DELETE supports deleting data from multiple tables.	DELETE	-
DELETE supports ORDER BY and LIMIT.	DELETE	-

MySQL Function	Syntax Description	GaussDB Implementation Difference
DELETE supports deleting data from a specified partition (or subpartition).	DELETE	-
UPDATE supports updating data from multiple tables.	UPDATE	-
UPDATE supports ORDER BY and LIMIT.	UPDATE	-
Support the SELECT INTO syntax.	SELECT	• In GaussDB, you can use SELECT INTO to create a table based on the query result. MySQL does not support this function. • In GaussDB, the SELECT INTO syntax does not support the query result that is obtained after the set operation of multiple queries is performed.

MySQL Function	Syntax Description	GaussDB Implementation Difference
Support the REPLACE INTO syntax.	REPLACE	• Difference between the initial values of the time type. For example: – MySQL is not affected by the strict or loose mode. You can insert time 0 into a table. mysql> CREATE TABLE test(f1 TIMESTAMP NOT NULL, f2 DATETIME NOT NULL, f3 DATE NOT NULL); Query OK, 1 row affected (0.00 sec) mysql> REPLACE INTO test VALUES(f1, f2, f3); Query OK, 1 row affected (0.00 sec) mysql> SELECT * FROM test; +--------------------- +---------------------+------------ + | f1 | f2 | f3 | +--------------------- +---------------------+------------ + | 0000-00-00 00:00:00 | 0000-00-00 00:00:00 | 0000-00-00 | +--------------------- +---------------------+------------ + 1 row in set (0.00 sec) – The time 0 can be successfully inserted only when GaussDB is in loose mode. gaussdb=# SET b_format_version = '5.7'; SET gaussdb=# SET b_format_dev_version = 's1'; SET gaussdb=# SET sql_mode = ''; SET gaussdb=# CREATE TABLE test(f1 TIMESTAMP NOT NULL, f2 DATETIME NOT NULL, f3 DATE NOT NULL); CREATE TABLE gaussdb=# REPLACE INTO test VALUES(f1, f2, f3); REPLACE 0 1 gaussdb=# SELECT * FROM test; f1 | f2 | f3 --------------------- +---------------------+------------

MySQL Function	Syntax Description	GaussDB Implementation Difference
		0000-00-00 00:00:00 | 0000-00-00 00:00:00 | 0000-00-00 (1 row) In strict mode, the error is reported: date/time field value out of range: "0000-00-00 00:00:00". • Difference between the initial values of the bit string type. For example: – The initial value of the BIT type is an empty string '' in MySQL, that is: mysql> CREATE TABLE test(f1 BIT(3) NOT NULL); Query OK, 0 rows affected (0.01 sec) mysql> REPLACE INTO test VALUES(f1); Query OK, 1 row affected (0.00 sec) mysql> SELECT f1, f1 IS NULL FROM test; +----+------------+ | f1 | f1 is null | +----+------------+ | | 0 | | | 0 | +----+------------+ 2 rows in set (0.00 sec) – If the initial value of the BIT type is **NULL** in GaussDB, an error is reported. gaussdb=# CREATE TABLE test(f1 BIT(3) NOT NULL); CREATE TABLE gaussdb=# REPLACE INTO test VALUES(f1); ERROR: null value in column "f1" violates not-null constraint DETAIL: Failing row contains (null).
SELECT supports multi-partition query.	SELECT	-
UPDATE supports multi-partition update.	UPDATE	-

MySQL Function	Syntax Description	GaussDB Implementation Difference
Import data by using LOAD DATA.	LOAD DATA	• The execution result of the LOAD DATA syntax is the same as that in MySQL strict mode. The loose mode is not adapted currently. • The **IGNORE** and **LOCAL** parameters are used only to ignore the conflicting rows when the imported data conflicts with the data in the table and to automatically fill default values for other columns when the number of columns in the file is less than that in the table. Other functions are not supported currently. • If the keyword LOCAL is specified and the file path is a relative path, the file is searched from the binary directory. If the keyword LOCAL is not specified and the file path is a relative path, the file is searched from the data directory. • If single quotation marks are specified as separators, escape characters, and newline characters in the syntax, lexical parsing errors occur. • The **[(col_name_or_user_var [, col_name_or_user_var]...)]** parameter cannot be used to specify a column repeatedly. • The newline character specified by **[FIELDS TERMINATED BY 'string']** cannot be the same as the separator specified by

MySQL Function	Syntax Description	GaussDB Implementation Difference
		[LINES TERMINATED BY'string']. ● If the data written to a table by running **LOAD DATA** cannot be converted to the data type of the table, an error is reported. ● The LOAD DATA SET expression does not support the calculation of a specified column name. ● If there is no implicit conversion function between the return value type of the SET expression and the corresponding column type, an error is reported. ● LOAD DATA applies only to tables but not views. ● The default newline character of the file in Windows is different from that in Linux. LOAD DATA cannot identify this scenario and reports an error. You are advised to check the newline character at the end of lines in the file to be imported.

MySQL Function	Syntax Description	GaussDB Implementation Difference
Compatible with INSERT IGNORE.	INSERT IGNORE	• GaussDB displays the error information after the downgrade. MySQL records the error information after the downgrade to the error stack and runs the **show warnings;** command to view the error information. • Time type difference. For example: – The default values of **date**, **datetime**, and **timestamp** in GaussDB are **0**. gaussdb=# CREATE TABLE test(f1 DATE NOT NULL, f2 DATETIME NOT NULL, f3 TIMESTAMP NOT NULL); CREATE TABLE gaussdb=# INSERT IGNORE INTO test VALUES(NULL, NULL, NULL); WARNING: null value in column "f1" violates not-null constraint DETAIL: Failing row contains (null, null, null, null). WARNING: null value in column "f2" violates not-null constraint DETAIL: Failing row contains (null, null, null, null). WARNING: null value in column "f3" violates not-null constraint DETAIL: Failing row contains (null, null, null, null). INSERT 0 1 gaussdb=# SELECT * FROM test; f1 | f2 | f3 ------------+---------------------+--------------------- 1970-01-01 | 1970-01-01 00:00:00 | 1970-01-01 00:00:00 (1 row) – The default values of **date**, **datetime**, and **timestamp** in MySQL are **0**. mysql> CREATE TABLE test(f1 DATE NOT NULL, f2

MySQL Function	Syntax Description	GaussDB Implementation Difference
		DATETIME NOT NULL, f3 TIMESTAMP NOT NULL); Query OK, 0 rows affected (0.00 sec) mysql> INSERT IGNORE INTO test VALUES(NULL, NULL, NULL); Query OK, 1 row affected, 3 warnings (0.00 sec) mysql> show warnings; +---------+------ +----------------------------+ | Level | Code | Message | +---------+------ +----------------------------+ | Warning | 1048 | Column 'f1' cannot be null | | Warning | 1048 | Column 'f2' cannot be null | | Warning | 1048 | Column 'f3' cannot be null | +---------+------ +----------------------------+ 3 rows in set (0.00 sec) mysql> SELECT * FROM test; +------------+--------------------- +---------------------+ | f1 | f2 | f3 | +------------+--------------------- +---------------------+ | 0000-00-00 | 0000-00-00 00:00:00 | 0000-00-00 00:00:00 | +------------+--------------------- +---------------------+ 1 row in set (0.00 sec) • GaussDB does not support the MySQL bit type. Therefore, the INSERT IGNORE error downgrade is not supported when the NOT NULL constraint of the bit type is ignored and the length of the inserted bit type is different from that defined. – Bit type in GaussDB gaussdb=# CREATE TABLE test(f1 BIT(10) NOT NULL); CREATE TABLE gaussdb=# INSERT IGNORE INTO test VALUES(NULL); ERROR: Un-support feature DETAIL: ignore null for insert statement is not supported in

MySQL Function	Syntax Description	GaussDB Implementation Difference
		column f1. gaussdb=# INSERT IGNORE INTO test VALUES('1010'); ERROR: bit string length 4 does not match type bit(10) CONTEXT: referenced column: f1 – Bit type in MySQL mysql> CREATE TABLE test(f1 BIT(10) NOT NULL); Query OK, 0 rows affected (0.00 sec) mysql> INSERT IGNORE INTO test VALUES(NULL); Query OK, 1 row affected, 1 warning (0.00 sec) mysql> INSERT IGNORE INTO test VALUES('1010'); Query OK, 1 row affected, 1 warning (0.01 sec) • If the precision is specified for the time type in MySQL, the precision is displayed when the zero value is inserted. It is not displayed in GaussDB. For example: – Time precision specified in GaussDB gaussdb=# CREATE TABLE test(f1 TIME(3) NOT NULL, f2 DATETIME(3) NOT NULL, f3 TIMESTAMP(3) NOT NULL); CREATE TABLE gaussdb=# INSERT IGNORE INTO test VALUES(NULL,NULL,NULL); WARNING: null value in column "f1" violates not-null constraint DETAIL: Failing row contains (null, null, null). WARNING: null value in column "f2" violates not-null constraint DETAIL: Failing row contains (null, null, null). WARNING: null value in column "f3" violates not-null constraint DETAIL: Failing row contains (null, null, null). INSERT 0 1 gaussdb=# SELECT * FROM test; f1 | f2 | f3

MySQL Function	Syntax Description	GaussDB Implementation Difference
		----------+---------- +---------- 00:00:00 | 1970-01-01 00:00:00 | 1970-01-01 00:00:00 (1 row) – Time precision specified in MySQL mysql> CREATE TABLE test(f1 TIME(3) NOT NULL, f2 DATETIME(3) NOT NULL, f3 TIMESTAMP(3) NOT NULL); Query OK, 0 rows affected (0.00 sec) mysql> INSERT IGNORE INTO test VALUES(NULL,NULL,NULL); Query OK, 1 row affected, 3 warnings (0.00 sec) mysql> SELECT * FROM test; +-------------- +------------------------- +-------------------------+ | f1 | f2 | f3 | +-------------- +------------------------- +-------------------------+ | 00:00:00.000 | 0000-00-00 00:00:00.000 | 0000-00-00 00:00:00.000 | +-------------- +------------------------- +-------------------------+ 1 row in set (0.00 sec) • The execution process in MySQL is different from that in GaussDB. Therefore, the number of generated warnings may be different. For example: – Number of warnings generated in GaussDB gaussdb=# CREATE TABLE test(f1 INT, f2 INT not null); CREATE TABLE gaussdb=# INSERT INTO test VALUES(1,0),(3,0),(5,0); INSERT 0 3 gaussdb=# INSERT IGNORE INTO test SELECT f1+1, f1/f2 FROM test; WARNING: division by zero CONTEXT: referenced column: f2 WARNING: null value in column "f2" violates not-null constraint

MySQL Function	Syntax Description	GaussDB Implementation Difference
		DETAIL: Failing row contains (2, null). WARNING: division by zero CONTEXT: referenced column: f2 WARNING: null value in column "f2" violates not-null constraint DETAIL: Failing row contains (4, null). WARNING: division by zero CONTEXT: referenced column: f2 WARNING: null value in column "f2" violates not-null constraint DETAIL: Failing row contains (6, null). INSERT 0 3 – Number of warnings generated in MySQL mysql> CREATE TABLE test(f1 INT, f2 INT not null); Query OK, 0 rows affected (0.01 sec) mysql> INSERT INTO test VALUES(1,0),(3,0),(5,0); Query OK, 3 rows affected (0.00 sec) Records: 3 Duplicates: 0 Warnings: 0 mysql> INSERT IGNORE INTO test SELECT f1+1, f1/f2 FROM test; Query OK, 3 rows affected, 4 warnings (0.00 sec) Records: 3 Duplicates: 0 Warnings: 4 • The differences between MySQL's and GaussDB's INSERT IGNORE in triggers are as follows: – INSERT IGNORE used in a GaussDB trigger gaussdb=# CREATE TABLE test1(f1 INT NOT NULL); CREATE TABLE gaussdb=# CREATE TABLE test2(f1 INT); CREATE TABLE gaussdb=# CREATE OR REPLACE FUNCTION trig_test() RETURNS TRIGGER AS $$ gaussdb$# BEGIN gaussdb$# INSERT IGNORE INTO test1 VALUES(NULL); gaussdb$# RETURN NEW;

MySQL Function	Syntax Description	GaussDB Implementation Difference
		gaussdb$# END; gaussdb$# $$ LANGUAGE plpgsql; CREATE FUNCTION gaussdb=# CREATE TRIGGER trig2 BEFORE INSERT ON test2 FOR EACH ROW EXECUTE PROCEDURE trig_test(); CREATE TRIGGER gaussdb=# INSERT INTO test2 VALUES(NULL); WARNING: null value in column "f1" violates not-null constraint DETAIL: Failing row contains (null). CONTEXT: SQL statement "INSERT IGNORE INTO test1 VALUES(NULL)" PL/pgSQL function trig_test() line 3 at SQL statement INSERT 0 1 gaussdb=# SELECT * FROM test1; f1 ---- 0 (1 rows) gaussdb=# SELECT * FROM test2; f1 ---- (1 rows) – INSERT IGNORE used in a MySQL trigger mysql> CREATE TABLE test1(f1 INT NOT NULL); Query OK, 0 rows affected (0.01 sec) mysql> CREATE TABLE test2(f1 INT); Query OK, 0 rows affected (0.00 sec) mysql> DELIMITER || mysql> CREATE TRIGGER trig2 BEFORE INSERT ON test2 FOR EACH ROW -> BEGIN -> INSERT IGNORE into test1 values(NULL); -> END|| Query OK, 0 rows affected (0.01 sec) mysql> DELIMITER ; mysql> INSERT INTO test2 VALUES(NULL); ERROR 1048 (23000): Column

MySQL Function	Syntax Description	GaussDB Implementation Difference
		'f1' cannot be null mysql> INSERT IGNORE INTO test2 VALUES(NULL); Query OK, 1 row affected (0.00 sec) mysql> SELECT * FROM test1; +----+ | f1 | +----+ | 0 | +----+ 1 row in set (0.00 sec) mysql> SELECT * FROM test2; +------+ | f1 | +------+ | NULL | +------+ 1 row in set (0.00 sec) • The implementation mechanism of Boolean and serial in GaussDB is different from that in MySQL. Therefore, the default zero value in GaussDB is different from that in MySQL. For example: – Behavior in GaussDB gaussdb=# CREATE TABLE test(f1 SERIAL, f2 BOOL NOT NULL); NOTICE: CREATE TABLE will create implicit sequence "test_f1_seq" for serial column "test.f1" CREATE TABLE gaussdb=# INSERT IGNORE INTO test values(NULL,NULL); WARNING: null value in column "f1" violates not-null constraint DETAIL: Failing row contains (null, null). WARNING: null value in column "f2" violates not-null constraint DETAIL: Failing row contains (null, null). INSERT 0 1 gaussdb=# SELECT * FROM test; f1 | f2 ----+---- 0 | f (1 row) – Behavior in MySQL

MySQL Function	Syntax Description	GaussDB Implementation Difference
		mysql> CREATE TABLE test(f1 SERIAL, f2 BOOL NOT NULL); Query OK, 0 rows affected (0.00 sec) mysql> INSERT IGNORE INTO test values(NULL,NULL); Query OK, 1 row affected, 1 warning (0.00 sec) mysql> SELECT * FROM test; +----+----+ | f1 | f2 | +----+----+ | 1 | 0 | +----+----+ 1 row in set (0.00 sec)

2.7.3 DCL

Table 2-26 DCL syntax compatibility

Description	Syntax	Difference
Set user-defined variables.	SET	Difference in the length of a user-defined variable. For example: • There is no restriction on the length of MySQL user-defined variable names. • The length of a user-defined GaussDB variable name cannot exceed 64 bytes. If the length exceeds 64 bytes, the excess part will be truncated and an alarm will be generated.
Support the SET TRANSACTION syntax.	SET TRANSACTION	In MySQL, you can set the transaction isolation level and read/write mode for the current session and global. In GaussDB, you need to set the **b_format_behavior_compat_options** parameter to include **set_session_transaction** for the current session. The global setting takes effect only for the current database.

Description	Syntax	Difference
Set names with COLLATE specified.	SET [SESSION | LOCAL] NAMES {'charset_name' [COLLATE 'collation_name'] | DEFAULT};	GaussDB does not allow **charset_name** to be different from the database character set. For details, see "SQL Reference > SQL Syntax > S > SET" in *Developer Guide*.

2.8 Drivers

2.8.1 JDBC

2.8.1.1 JDBC API Reference

The JDBC API definitions in GaussDB are the same as those in MySQL and comply with industry standards. This section describes the behavior differences of JDBC APIs between the GaussDB in B-compatible mode and MySQL.

Obtaining Data from a Result Set

ResultSet objects provide a variety of methods to obtain data from a result set. **Table 2-27** describes the common methods for obtaining data. If you want to know more about other methods, see JDK official documents.

Table 2-27 Common methods for obtaining data from a result set

Method	Description	Difference
int getInt(int columnIndex)	Obtains int data by column index.	-
int getInt(String columnLabel)	Obtains int data by column name.	-
String getString(int columnIndex)	Obtains string data by column index.	If the column type is integer and the column contains the **ZEROFILL** attribute, GaussDB pads 0s to meet the width required by the **ZEROFILL** attribute and outputs the result. MySQL directly outputs the result.

Method	Description	Difference
String getString(String columnLabel)	Obtains string data by column name.	If the column type is integer and the column contains the **ZEROFILL** attribute, GaussDB pads 0s to meet the width required by the **ZEROFILL** attribute and outputs the result. MySQL directly outputs the result.
Date getDate(int columnIndex)	Obtains date data by column index.	-
Date getDate(String columnLabel)	Obtains date data by column name.	-

3 M-compatible Mode

3.1 Data Types

3.1.1 Numeric Data Types

Unless otherwise specified, considering the characteristics of the floating-point type in terms of precision, scale, and number of digits after the decimal point, the floating-point type is not supported in GaussDB by default. You are advised to use a valid integer type.

Table 3-1 Integer types

Data Type	Differences Compared with MySQL
BOOL	Output format: The output of SELECT TRUE/FALSE in GaussDB is **t** or **f**, and that in MySQL is **1** or **0**. MySQL: The BOOL/BOOLEAN type is actually mapped to the TINYINT type.
BOOLEAN	
TINYINT[(M)] [UNSIGNED] [ZEROFILL]	For details about the differences, see the examples below the table.
SMALLINT[(M)] [UNSIGNED] [ZEROFILL]	
MEDIUMINT[(M)] [UNSIGNED] [ZEROFILL]	For details about the differences, see the examples below the table.
INT[(M)] [UNSIGNED] [ZEROFILL]	For details about the differences, see the examples below the table.

Data Type	Differences Compared with MySQL
INTEGER[(M)] [UNSIGNED] [ZEROFILL]	
BIGINT[(M)] [UNSIGNED] [ZEROFILL]	

Difference 1: In MySQL 5.7, when a string like '1.2.3.4' that contains multiple decimal points is entered, the content of the first decimal point is incorrectly inserted in loose mode (when **sql_mode** does not contain the **'strict_trans_tables'** option). This issue has been resolved in MySQL 8.0. GaussDB is consistent with MySQL 8.0.

```
-- GaussDB
m_db=# SET SQL_MODE='';
SET
m_db=# CREATE TABLE test_int(a int);
CREATE TABLE
m_db=# INSERT INTO test_int VALUES('1.2.3');
WARNING:  Data truncated for column
LINE 1: INSERT INTO test_int VALUES('1.2.3');
                                    ^
CONTEXT:  referenced column: a
INSERT 0 1
m_db=# SELECT * FROM test_int;
 a
---
 1
(1 row)

m_db=# DROP TABLE test_int;
DROP TABLE

-- MySQL 5.7
mysql> SET SQL_MODE='';
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> CREATE TABLE test_int(a int);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO test_int VALUES('1.2.3');
Query OK, 1 row affected, 1 warning (0.00 sec)

mysql> SELECT * FROM test_int;
+------+
| a    |
+------+
|   12 |
+------+
1 row in set (0.00 sec)

mysql> DROP TABLE test_int;
Query OK, 0 rows affected (0.01 sec)
```

Difference 2: When the parameter that specifies the precision is enabled (**m_format_behavior_compat_options** contains the **'enable_precision_decimal'** option), in the CREATE TABLE AS scenario of UNION, GaussDB calculates the column length using the maximum length of a directly input integer (11 for INT

and 20 for BIGINT), while MySQL calculates it based on the actual length of the integer.

```
-- GaussDB
m_db=# set m_format_behavior_compat_options= 'enable_precision_decimal';
SET
m_db=# CREATE TABLE test_int AS SELECT 1234567 UNION ALL SELECT '456789';
INSERT 0 2
m_db=# DESC test_int;
  Field   |     Type     | Null | Key | Default | Extra
----------+--------------+------+-----+---------+-------
 ?column? | varchar(11) | YES  |     |         |
(1 row)

m_db=# DROP TABLE test_int;
DROP TABLE
m_db=# CREATE TABLE test_int AS SELECT 1234567890 UNION ALL SELECT '456789';
INSERT 0 2
m_db=# DESC test_int;
  Field   |     Type     | Null | Key | Default | Extra
----------+--------------+------+-----+---------+-------
 ?column? | varchar(20) | YES  |     |         |
(1 row)

m_db=# DROP TABLE test_int;
DROP TABLE

-- MySQL
mysql> CREATE TABLE test_int AS SELECT 1234567 UNION ALL SELECT '456789';
Query OK, 2 rows affected (0.02 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> DESC test_int;
+---------+------------+------+-----+---------+-------+
| Field   | Type       | Null | Key | Default | Extra |
+---------+------------+------+-----+---------+-------+
| 1234567 | varchar(7) | NO   |     |         |       |
+---------+------------+------+-----+---------+-------+
1 row in set (0.00 sec)

mysql> DROP TABLE test_int;
Query OK, 0 rows affected (0.01 sec)

mysql> CREATE TABLE test_int AS SELECT 1234567890 UNION ALL SELECT '456789';
Query OK, 2 rows affected (0.02 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> DESC test_int;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| 1234567890 | varchar(10) | NO   |     |         |       |
+------------+-------------+------+-----+---------+-------+
1 row in set (0.00 sec)

mysql> DROP TABLE test_int;
Query OK, 0 rows affected (0.00 sec)
```


Table 3-2 Arbitrary precision types

Data Type	Differences Compared with MySQL
DECIMAL[(M[,D])] [ZEROFILL]	MySQL uses a 9 x 9 array to store values of these types. The integer part and decimal part are stored separately. If the length exceeds the value, the decimal part will be truncated first. In GaussDB, an integer with more than 81 digits will be truncated.
NUMERIC[(M[,D])] [ZEROFILL]	
DEC[(M[,D])] [ZEROFILL]	
FIXED[(M[,D])] [ZEROFILL]	

Table 3-3 Floating-point types

Data Type	Differences Compared with MySQL
FLOAT[(M,D)] [ZEROFILL]	In the scenario where the driver adopts FLOAT and DOUBLE types with a precision scale, no error is reported when the input data exceeds the range.
FLOAT(p) [ZEROFILL]	
DOUBLE[(M,D)] [ZEROFILL]	
DOUBLE PRECISION[(M,D)] [ZEROFILL]	
REAL[(M,D)] [ZEROFILL]	

3.1.2 Date and Time Data Types

Table 3-4 Date and Time Data Types

Data Type	Differences Compared with MySQL
DATE	None
DATETIME[(fsp)]	For details about the differences, see the description below the table.

Data Type	Differences Compared with MySQL
TIMESTAMP[(fsp)]	GaussDB supports the timestamp data type. Compared with MySQL, GaussDB has the following differences in specifications: ● In MySQL 5.7, the default value of the **timestamp** column is the real time when data is inserted. Same as MySQL 8.0, GaussDB has no default value set for this column. That is, when **null** is inserted, the value is **null**. ● For other differences, see the description below the table.
TIME[(fsp)]	GaussDB supports the time data type. Compared with MySQL, GaussDB has the following differences in specifications: ● When the hour, minute, second, and nanosecond of the time type are 0, the sign bits of GaussDB and MySQL may be different. ● For other differences, see the description below the table.
YEAR[(4)]	GaussDB supports the year data type. Compared with MySQL, GaussDB has the following differences in specifications: In MySQL 5.7, the **year** column is displayed as 'year(4)' by default. GaussDB is consistent with MySQL 8.0, displaying only 'year'.

NOTE

- GaussDB does not support ODBC syntax literals:

  { d 'str' }

  { t 'str' }

  { ts 'str' }
- GaussDB supports standard SQL literals, and precision can be added after type keywords, but MySQL does not support the following:

  DATE[(n)] 'str'

  TIME[(n)] 'str'

  TIMESTAMP[(n)] 'str'
- If you specify a precision for the DATETIME, TIME, or TIMESTAMP data type greater than the maximum precision supported by the data type, GaussDB truncates the precision to the maximum precision supported by the data type, whereas MySQL reports an error.

3.1.3 String Data Types

Table 3-5 String Data Types

Data Type	Differences Compared with MySQL
CHAR(M)	For details about the differences, see the description below the table.
VARCHAR(M)	For details about the differences, see the description below the table.
TINYTEXT	For details about the differences, see the description below the table.
TEXT	For details about the differences, see the description below the table.
MEDIUMTEXT	For details about the differences, see the description below the table.
LONGTEXT	Input format: • GaussDB supports a maximum of 1 GB – 512 bytes, and MySQL supports a maximum of 4 GB – 1 byte. • For other differences, see the description below the table.

NOTE

- For binary or hexadecimal strings that cannot be escaped, MySQL outputs an empty string, while GaussDB outputs a hexadecimal result.
- For the TINYTEXT, TEXT, MEDIUMTEXT, and LONGTEXT types:
  - In MySQL, a default value cannot be set. However, in GaussDB, you can set a default value when creating a table column.
  - Primary key: MySQL does not support primary keys, but GaussDB does.
  - Index: MySQL supports only prefix indexes. GaussDB supports all index methods.
  - Foreign key: MySQL does not support any of these types to be the referencing column or referenced column of a foreign key, but GaussDB supports.

Example:

```
-- GaussDB
m_db=# CREATE TABLE test_text(a text);
CREATE TABLE

m_db=# INSERT INTO test_text VALUES(0x1);
INSERT 0 1

m_db=# INSERT INTO test_text VALUES(0x111111);
INSERT 0 1

m_db=# INSERT INTO test_text VALUES(0x61);
INSERT 0 1

m_db=# SELECT * FROM test_text;
     a
```



```
--------------
 \x01
 \x11\x11\x11
 a
(3 rows)

m_db=# DROP TABLE test_text;
DROP TABLE

-- MySQL 5.7
mysql> CREATE TABLE test_text(a text);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO test_text VALUES(0x1);
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO test_text VALUES(0x111111);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO test_text VALUES(0x61);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM test_text;
+------+
| a    |
+------+
|      |
|   |
| a    |
+------+
3 rows in set (0.00 sec)

mysql> DROP TABLE test_text;
Query OK, 0 rows affected (0.01 sec)
```


3.1.4 Binary Data Types

Table 3-6 Binary Data Types

Data Type	Differences Compared with MySQL
BINARY[(M)]	• Input format: – If the length of the inserted string is less than the target length, the padding character is 0x20 in GaussDB and 0x00 in MySQL. • Character set: The default character set is the initialized character set of the database. For MySQL, the default character set is BINARY. • Output formats: – When the JDBC protocol is used, a space at the end of the BINARY type is displayed as a space, and that in MySQL is displayed as \x00. – In loose mode, if characters (such as Chinese characters) of the BINARY type exceed *n* bytes, the excess characters will be truncated. MySQL retains the first *n* bytes. However, garbled characters are displayed in the output. For other differences, see the description below the table. **NOTE** Due to the differences between GaussDB and MySQL in BINARY fillers, GaussDB and MySQL have different performance in scenarios such as operator comparison calculation, character string-related system function calculation, index matching, and data import and export. For details about the specific difference scenarios, see the examples in this section.
VARBINARY(M)	Character set: The default character set is the initialized character set of the database. For MySQL, the default character set is BINARY. For other differences, see the description below the table.
TINYBLOB	For details about the differences, see the description below the table.
BLOB	For details about the differences, see the description below the table.
MEDIUMBLOB	For details about the differences, see the description below the table.
LONGBLOB	Value range: a maximum of 1 GB – 512 bytes. MySQL supports a maximum of 4 GB – 1 byte.

Data Type	Differences Compared with MySQL
BIT[(M)]	Output formats: • GaussDB outputs a binary character string. In MySQL 5.7, the result is escaped based on the ASCII code table. If the result cannot be escaped, a character string that cannot be displayed is output. In MySQL 8.0, the hexadecimal result is output in the 0x format. • In MySQL 8.0 and later versions, 0 is added at the beginning of each result by default. In GaussDB, 0 is not added. For other differences, see the description below the table.

NOTE

- For unescapable binary or hexadecimal strings, MySQL 5.7 outputs undisplayable strings; MySQL 8.0 outputs a hexadecimal result in the format of 0x; GaussDB outputs a hexadecimal result in the format of multiple \x.
- For the TINYBLOB, BLOB, MEDIUMBLOB, and LONGBLOB types:
  - In MySQL, a default value cannot be set. However, in GaussDB, you can set a default value when creating a table column.
  - Primary key: MySQL does not support primary keys, but GaussDB does.
  - Index: MySQL supports only prefix indexes. GaussDB supports all index methods.
  - Foreign key: MySQL does not support any of these types to be the referencing column or referenced column of a foreign key, but GaussDB supports.

Example:

```
-- GaussDB
m_db=# CREATE TABLE test_blob(a blob);
CREATE TABLE

m_db=# INSERT INTO test_blob VALUES(0x1);
INSERT 0 1

m_db=# INSERT INTO test_blob VALUES(0x111111);
INSERT 0 1

m_db=# INSERT INTO test_blob VALUES(0x61);
INSERT 0 1

m_db=# SELECT * FROM test_blob;
      a
--------------
 \x01
 \x11\x11\x11
 a
(3 rows)

m_db=# DROP TABLE test_blob;
DROP TABLE

-- MySQL 5.7
mysql> CREATE TABLE test_blob(a blob);
Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO test_blob VALUES(0x1);
Query OK, 1 row affected (0.00 sec)
```



```
mysql> INSERT INTO test_blob VALUES(0x111111);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO test_blob VALUES(0x61);
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM test_blob;
+------+
| a    |
+------+
|      |
|  |
| a    |
+------+
3 rows in set (0.00 sec)

mysql> DROP TABLE test_blob;
Query OK, 0 rows affected (0.00 sec)

-- MySQL 8.0
mysql> CREATE TABLE test_blob(a blob);
Query OK, 0 rows affected (0.08 sec)

mysql> INSERT INTO test_blob VALUES(0x1);
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO test_blob VALUES(0x111111);
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO test_blob VALUES(0x61);
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM test_blob;
+------------+
| a          |
+------------+
| 0x01       |
| 0x111111   |
| 0x61       |
+------------+
3 rows in set (0.00 sec)

mysql> DROP TABLE test_blob;
Query OK, 0 rows affected (0.04 sec)
```


3.1.5 JSON

Table 3-7 JSON Data Type

Data Type	Differences Compared with MySQL
JSON	GaussDB supports the JSON data type. Compared with MySQL, GaussDB has the following differences in specifications: • Value range: In MySQL, the maximum size of the JSON data type is 4 GB. However, in GaussDB, the maximum size of the JSON data type is 1 GB – 512 bytes, and the maximum number of key-value pairs of an object and the maximum number of elements in an array are also less than those in MySQL. • Difference in collation: In MySQL, when the collation function is used to separately query columns of the JSON type, the returned collation is BINARY. However, utf8mb4_bin is returned in GaussDB. In other scenarios, utf8mb4_bin is used, which is the same as that of MySQL. • The differences in behavior when ORDER BY is performed on non-scalar JSON types are as follows: MySQL has no rule for using ORDER BY to sort non-scalar JSON data and does not support this operation. It reports an alarm for attempts of sorting non-scalar JSON data. In GaussDB, non-scalar JSON data can be sorted and arranged according to specific collation rules. – Compare the JSON data types (OPAQUE > TIMESTAMP = DATETIME > TIME > DATE > BOOL > array & > object & > string type > DOUBLE = UINT = INT = DECIMAL > NULL). If the results are the same, compare the contents. – Compare values between scalars. For OPAQUE, compare types first (TYPE_STRING > TYPE_VAR_STRING > TYPE_BLOB > TYPE_BIT > TYPE_VARCHAR > TYPE_YEAR). If the types are the same, compare each byte. – Compare the size of arrays. The size of arrays can be compared by comparing the number of the elements between arrays. The array with more elements is larger. When the number of elements are the same, compare the lengths of elements. If the lengths are the same, compare each key-value pair in sequence. Compare the key first and then the value.

NOTE

- In GaussDB, the BLOB, TINYBLOB, MEDIUMBLOB, LONGBLOB, BINARY, VARBINARY, BIT, and YEAR types are converted to the JSON type. The result is different from that in MySQL.

  Example:

```
-- GaussDB
m_db=# CREATE TABLE test_blob (c1 BLOB, c2 TINYBLOB, c3 MEDIUMBLOB, c4 LONGBLOB, c5
BINARY(32), c6 VARBINARY(100), c7 BIT(64), c8 YEAR);
CREATE TABLE
m_db=# INSERT INTO test_blob VALUES('[1, "json"]', 'true', 'abc', '{"jsnid": 1, "tag": "ab"}', '[1,
"json"]', '{"jsnid": 1, "tag": "ab"}', '20', '2020');
INSERT 0 1
m_db=# SELECT CAST(c1 AS JSON), CAST(c2 AS JSON), CAST(c3 AS JSON), CAST(c4 AS JSON),
CAST(c5 AS JSON), CAST(c6 AS JSON), CAST(c7 AS JSON), CAST(c8 AS JSON) FROM test_blob;
      CAST       |  CAST  | CAST  |               CAST                |                CAST                |
CAST               | CAST |  CAST
-----------------+--------+-------+-----------------------------------+------------------------------------
+-----------------------------------+------+--------
 "[1, \"json\"]" | "true" | "abc" | "{\"jsnid\": 1, \"tag\": \"ab\"}" | "[1, \"json\"]                    " |
"{\"jsnid\": 1, \"tag\": \"ab\"}" | "20" | "2020"
(1 row)
m_db=# DROP TABLE test_blob;
DROP TABLE

-- MySQL
mysql> CREATE TABLE test_blob (c1 BLOB, c2 TINYBLOB, c3 MEDIUMBLOB, c4 LONGBLOB, c5
BINARY(32), c6 VARBINARY(100), c7 BIT(64), c8 YEAR);
Query OK, 0 rows affected (0.02 sec)
mysql> INSERT INTO test_blob VALUES('[1, "json"]', 'true', 'abc', '{"jsnid": 1, "tag": "ab"}', '[1,
"json"]', '{"jsnid": 1, "tag": "ab"}', '20', '2020');
Query OK, 1 row affected (0.00 sec)
mysql> SELECT CAST(c1 AS JSON), CAST(c2 AS JSON), CAST(c3 AS JSON), CAST(c4 AS JSON),
CAST(c5 AS JSON), CAST(c6 AS JSON), CAST(c7 AS JSON), CAST(c8 AS JSON) FROM test_blob;
+-----------------------------------+---------------------------+-----------------------
+-------------------------------------------------
+-------------------------------------------------------------
+-----------------------------------------------------+-----------------------------+-------------------------
+
| CAST(c1 AS JSON)                  | CAST(c2 AS JSON)          | CAST(c3 AS JSON)      | CAST(c4 AS
JSON)                                | CAST(c5 AS JSON)                                    | CAST(c6 AS
JSON)                                | CAST(c7 AS JSON)          | CAST(c8 AS JSON)      |
+-----------------------------------+---------------------------+-----------------------
+-------------------------------------------------
+-------------------------------------------------------------
+-----------------------------------------------------+-----------------------------+-------------------------
+
| "base64:type252:WzEsICJqc29uIl0=" | "base64:type249:dHJ1ZQ==" | "base64:type250:YWJj" |
"base64:type251:eyJqc25pZCI6IDEsICJ0YWciOiAiYWIifQ==" |
"base64:type254:WzEsICJqc29uIl0AAAAAAAAAAAAAAAAAAAAAAAAAAAAA=" |
"base64:type15:eyJqc25pZCI6IDEsICJ0YWciOiAiYWIifQ==" | "base64:type16:AAAAAAAAMjA=" |
"base64:type13:MjAyMA==" |
+-----------------------------------+---------------------------+-----------------------
+-------------------------------------------------
+-------------------------------------------------------------
+-----------------------------------------------------+-----------------------------+-------------------------
+
1 row in set (0.00 sec)
mysql> DROP TABLE test_blob;
Query OK, 0 rows affected (0.01 sec)
```

- If a forcible type conversion (::JSON) is performed on data in the current version, the actual converted JSON type depends on whether the GUC parameter **m_format_behavior_compat_options** is set to **cast_as_new_json**.

3.1.6 Attributes Supported by Data Types

Table 3-8 Attributes Supported by Data Types

Attributes Supported by Data Types
NULL
NOT NULL
DEFAULT
ON UPDATE
PRIMARY KEY
AUTO_INCREMENT
CHARACTER SET name
COLLATE name
ZEROFILL

Difference:

During table creation, default values are set for fields of the VARBINARY type. When querying the table structure using DESC or similar methods, GaussDB displays the converted hexadecimal values, whereas MySQL displays original values.

Example:

```
-- GaussDB
m_db=# CREATE TABLE test_varbinary(a varbinary(20) DEFAULT 'GaussDB');
CREATE TABLE

m_db=# DESC test_varbinary;
 Field |     Type      | Null | Key |      Default       | Extra
-------+---------------+------+-----+--------------------+-------
 a     | varbinary(20) | YES  |     | X'47617573734442' |
(1 row)

m_db=# DROP TABLE test_varbinary;
DROP TABLE

-- MySQL
mysql> CREATE TABLE test_varbinary(a varbinary(20) DEFAULT 'GaussDB');
Query OK, 0 rows affected (0.02 sec)

mysql> DESC test_varbinary;
+-------+---------------+------+-----+---------+-------+
| Field | Type          | Null | Key | Default | Extra |
+-------+---------------+------+-----+---------+-------+
| a     | varbinary(20) | YES  |     | GaussDB |       |
+-------+---------------+------+-----+---------+-------+
1 row in set (0.00 sec)

mysql> DROP TABLE test_varbinary;
Query OK, 0 rows affected (0.01 sec)
```


3.1.7 Data Type Conversion

Conversion between different data types is supported. Data type conversion is involved in the following scenarios:

- The data types of operands of operators (such as comparison and arithmetic operators) are inconsistent. It is commonly used for comparison operations in query conditions or join conditions.
- The data types of arguments and parameters are inconsistent when a function is called.
- The data types of target columns to be updated by DML statements (including INSERT, UPDATE, MERGE, and REPLACE) and the defined column types are inconsistent.
- After the target data type of the final projection column is determined by set operations (UNION and EXCEPT), the type of the projection column in each SELECT statement is inconsistent with the target data type.
- In other expression calculation scenarios, the target data type used for comparison or final result is determined based on the data type of different expressions.
- When the collation of a common character string is BINARY, the character string is converted to the corresponding binary type (for example, TEXT is converted to BLOB, and VARCHAR is converted to VARBINARY).

There are three types of data type conversion differences: implicit conversion, UNION/CASE, and decimal type.

Differences in Implicit Type Conversion

- In GaussDB, the conversion rules from small types to small types are used. In MySQL, the conversion rules from small types to large types and from large types to small types are used.
- In a WHERE clause with only character strings, GaussDB identifies 't', 'true', 'y', 'yes', 'on', '1', and their uppercase counterpart as **TRUE** and a query result can be obtained. Conversely, 'f', 'false', 'n', 'no', 'off', and '0', including their uppercase counterparts, are identified as **FALSE** and a query result cannot be obtained. Errors are reported for other character strings. In MySQL, the query result can be obtained only when the first digit of a character string is not 0; otherwise, the query result cannot be obtained.

  Example:

```
-- GaussDB
m_db=# CREATE TABLE test_where(a int);
CREATE TABLE

m_db=# INSERT INTO test_where VALUES(1);
INSERT 0 1

m_db=# SELECT * FROM test_where WHERE 't';
 a
---
 1
(1 row)

m_db=# SELECT * FROM test_where WHERE '1';
 a
---
```



```
 1
(1 row)

m_db=# SELECT * FROM test_where WHERE '1a';
ERROR:  invalid input syntax for type boolean: "1a"
LINE 1: SELECT * FROM test_where WHERE '1a';
                                       ^
m_db=# SELECT * FROM test_where WHERE 'f';
 a
---
(0 rows)

m_db=# SELECT * FROM test_where WHERE '0';
 a
---
(0 rows)

m_db=# DROP TABLE test_where;
DROP TABLE

-- MySQL
mysql> CREATE TABLE test_where(a int);
Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO test_where VALUES(1);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM test_where WHERE 't';
Empty set, 1 warning (0.00 sec)

mysql> SELECT * FROM test_where WHERE '1';
+------+
| a    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)

mysql> SELECT * FROM test_where WHERE '1a';
+------+
| a    |
+------+
|    1 |
+------+
1 row in set, 1 warning (0.00 sec)

mysql> SELECT * FROM test_where WHERE 'f';
Empty set, 1 warning (0.00 sec)

mysql> SELECT * FROM test_where WHERE '0';
Empty set (0.00 sec)

mysql> DROP TABLE test_where;
Query OK, 0 rows affected (0.01 sec)
```

- For a YEAR type table, if a character string contains 'e' or 'E', MySQL processes it using scientific notation, while GaussDB reports an error or truncates it.

  Example:

```
-- GaussDB
m_db=# SET SQL_MODE='';
SET

m_db=# CREATE TABLE test_year(a year);
CREATE TABLE

m_db=# INSERT INTO test_year VALUES('2E3');
WARNING:  Data truncated for column.
LINE 1: INSERT INTO test_year VALUES('2E3');
                                     ^
```



```
CONTEXT:  referenced column: a
INSERT 0 1
m_db=# SELECT * FROM test_year;
  a
------
 2002
(1 row)

m_db=# DROP TABLE test_year;
DROP TABLE

-- MySQL
mysql> CREATE TABLE test_year(a year);
Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO test_year VALUES('2E3');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM test_year;
+------+
| a    |
+------+
| 2000 |
+------+
1 row in set (0.00 sec)

mysql> DROP TABLE test_year;
Query OK, 0 rows affected (0.01 sec)
```

- In the function nesting scenarios in GaussDB, if aggregate functions (such as max, min, sum, and avg) contain non-numeric characters in the string type, the characters of this type are truncated or set to zeros during implicit conversion to the numeric type. If operator comparison and HAVING comparison are also involved, GaussDB converts types and generates alarms in a unified manner, but MySQL may not generate alarms in the same scenarios.

  Example:

```
-- GaussDB
m_db=# SET m_format_behavior_compat_options= 'enable_precision_decimal';
SET

m_db=# SELECT max(c4) <> 0 FROM ((SELECT 2.22 id, '2006-04-27 20:19:02.132' c4)) tb_1;
 ?column?
----------
 t
(1 row)

m_db=# SELECT sum(c4) <> 0 FROM ((SELECT 2.22 id, '2006-04-27 20:19:02.132' c4)) tb_1;
WARNING:  The double value '2006-04-27 20:19:02.132' is incorrect.
 ?column?
----------
 t
(1 row)

m_db=# SELECT (SELECT max(c4) f5 FROM ((SELECT 2.22 id, '2006-04-27 20:19:08.132' c4) UNION
ALL (SELECT 2.22 id, '1985-09-01 07:59:59' c4)) tb_1 WHERE EXISTS (SELECT max(c4) FROM
((SELECT 2.22 id, '2006-04-27 20:19:08.132' c4) UNION ALL (SELECT 2.22 id, '1985-09-01 07:59:59'
c4)) tb_2) GROUP BY id WITH rollup HAVING f5<>0 LIMIT 0,1) + INTERVAL '33.22'
SECOND_MICROSECOND col5;
           col5
----------------------------
 2006-04-27 20:19:41.352000
(1 row)

m_db=# SELECT (SELECT sum(c4) f5 FROM ((SELECT 2.22 id, '2006-04-27 20:19:08.132' c4) UNION
ALL (SELECT 2.22 id, '1985-09-01 07:59:59' c4)) tb_1 WHERE EXISTS (SELECT sum(c4) FROM ((SELECT
2.22 id, '2006-04-27 20:19:08.132' c4) UNION ALL (SELECT 2.22 id, '1985-09-01 07:59:59' c4)) tb_2)
```



```
GROUP BY id WITH rollup HAVING f5<>0 LIMIT 0,1) + INTERVAL '33.22' SECOND_MICROSECOND
col5;
WARNING:  The double value '2006-04-27 20:19:08.132' is incorrect.
CONTEXT:  referenced column: col5
WARNING:  The double value '1985-09-01 07:59:59' is incorrect.
CONTEXT:  referenced column: col5
WARNING:  The double value '2006-04-27 20:19:08.132' is incorrect.
CONTEXT:  referenced column: col5
WARNING:  The double value '2006-04-27 20:19:08.132' is incorrect.
CONTEXT:  referenced column: col5
WARNING:  The double value '1985-09-01 07:59:59' is incorrect.
CONTEXT:  referenced column: col5
WARNING:  The double value '1985-09-01 07:59:59' is incorrect.
CONTEXT:  referenced column: col5
WARNING:  Incorrect datetime value: '3991'
CONTEXT:  referenced column: col5
 col5
------

(1 row)

-- MySQL
mysql> SELECT max(c4) <> 0 FROM ((SELECT 2.22 id, '2006-04-27 20:19:02.132' c4)) tb_1;
+--------------+
| max(c4) <> 0 |
+--------------+
|            1 |
+--------------+
1 row in set (0.00 sec)

mysql> SELECT sum(c4) <> 0 FROM ((SELECT 2.22 id, '2006-04-27 20:19:02.132' c4)) tb_1;
+--------------+
| sum(c4) <> 0 |
+--------------+
|            1 |
+--------------+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW warnings;
+---------+------+-----------------------------------------------------------------+
| Level   | Code | Message                                                         |
+---------+------+-----------------------------------------------------------------+
| Warning | 1292 | Truncated incorrect DOUBLE value: '2006-04-27 20:19:02.132' |
+---------+------+-----------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT (SELECT max(c4) f5 FROM ((SELECT 2.22 id, '2006-04-27 20:19:08.132' c4) UNION
ALL (SELECT 2.22 id, '1985-09-01 07:59:59' c4)) tb_1
    -> WHERE EXISTS (SELECT max(c4) FROM ((SELECT 2.22 id, '2006-04-27 20:19:08.132' c4) UNION
ALL (SELECT 2.22 id, '1985-09-01 07:59:59' c4)) tb_2)
    -> GROUP BY id WITH rollup HAVING f5<>0 limit 0,1) + INTERVAL '33.22'
SECOND_MICROSECOND col5;
+----------------------------+
| col5                       |
+----------------------------+
| 2006-04-27 20:19:41.352000 |
+----------------------------+
1 row in set (0.00 sec)

mysql> SELECT (SELECT sum(c4) f5 FROM ((SELECT 2.22 id, '2006-04-27 20:19:08.132' c4) UNION
ALL (SELECT 2.22 id, '1985-09-01 07:59:59' c4)) tb_1
    -> WHERE EXISTS (SELECT sum(c4) FROM ((SELECT 2.22 id, '2006-04-27 20:19:08.132' c4) UNION
ALL (SELECT 2.22 id, '1985-09-01 07:59:59' c4)) tb_2)
    -> GROUP BY id WITH rollup HAVING f5<>0 LIMIT 0,1) + INTERVAL '33.22'
SECOND_MICROSECOND col5;
+------+
| col5 |
+------+
| NULL |
```



```
+------+
1 row in set, 7 warnings (0.01 sec)

mysql> SHOW warnings;
+---------+------+-----------------------------------------------------------------+
| Level   | Code | Message                                                         |
+---------+------+-----------------------------------------------------------------+
| Warning | 1292 | Truncated incorrect DOUBLE value: '2006-04-27 20:19:08.132' |
| Warning | 1292 | Truncated incorrect DOUBLE value: '1985-09-01 07:59:59'     |
| Warning | 1292 | Truncated incorrect DOUBLE value: '2006-04-27 20:19:08.132' |
| Warning | 1292 | Truncated incorrect DOUBLE value: '2006-04-27 20:19:08.132' |
| Warning | 1292 | Truncated incorrect DOUBLE value: '1985-09-01 07:59:59'     |
| Warning | 1292 | Truncated incorrect DOUBLE value: '1985-09-01 07:59:59'     |
| Warning | 1292 | Incorrect datetime value: '3991'                            |
+---------+------+-----------------------------------------------------------------+
7 rows in set (0.00 sec)
```

Differences Between UNION, CASE, and Related Structures

- In MySQL, POLYGON+NULL, POINT+NULL, and POLYGON+POINT return the GEOMETRY type. They are not involved in GaussDB and considered as errors.
- The SET and ENUM types are not supported currently and are considered as errors.
- For UNION or UNION ALL that combines the JSON and binary types (BINARY, VARBINARY, TINYBLOB, BLOB, MEDIUMBLOB, and LONGBLOB), BIT, or YEAR, MySQL returns the LONGBLOB or LONGTEXT type while GaussDB returns the JSON type. In addition, binary types (BINARY, VARBINARY, TINYBLOB, BLOB, MEDIUMBLOB, and LONGBLOB), BIT, or YEAR, can be implicitly converted to JSON.
- If **m_format_behavior_compat_options** is not set to **enable_precision_decimal**, when the constant type is aggregated with other types, the precision of the output type is the precision of other types. For example, the precision of the result of "SELECT "helloworld" UNION SELECT p FROM t;" is the precision of attribute p.
- If **m_format_behavior_compat_options** is not set to **enable_precision_decimal**, when fixed-point constants and types without precision constraints (non-string types such as int, bool, and year, and the fixed-point type of aggregation result type) are aggregated, the precision constraint is output based on the default precision 31 of fixed-point numbers.
- Differences in merge rules:

  MySQL 5.7 has some improper type derivation. For example, the VARBINARY type is derived from the BIT type and integer/YEAR type, and the UNSIGNED type is derived from the UNSIGNED type and non-UNSIGNED type. In addition, the aggregation results of CASE WHEN and UNION are different. If the type derivation result is too small, data overflow may occur. The preceding issues have been resolved in MySQL 8.0. Therefore, the merge rule in MySQL 8.0 prevails.
- In MySQL, BINARY and CHAR use different padding characters. BINARY is padded with '\0', and CHAR is padded with spaces. In GaussDB, BINARY and CHAR are padded with spaces.
- In the precision transfer scenario, when the CASE WHEN statement is used, type conversion and precision recalculation are performed. As a result, trailing zeros may be inconsistent with those in the output result of the CASE clause.
  - More trailing zeros: The CASE node calculates the precision of the CASE node based on the precision of the CASE clause. If the precision of the

THEN clause is lower than that of the CASE node, zeros are added to the end of the CASE node.

- Less trailing zeros: When multiple layers of CASE WHEN are nested, only the precision of the inner CASE is retained after the inner CASE performs type conversion. The outer CASE cannot obtain the precision information of the THEN clause. Therefore, the outer CASE performs type conversion based on the precision calculated according to that of the inner CASE. When the outer CASE clause is converted, if the precision of the inner CASE clause is less than that of the THEN clause, there will be less trailing zeros.

Example:

- -- Trailing zeros

```
-- More trailing zeros
m_db=# SELECT 15.6 AS result;
 result
--------
   15.6
(1 row)

m_db=# SELECT CASE WHEN 1 < 2 THEN 15.6 ELSE  23.578 END AS result;
 result
--------
 15.600
(1 row)

m_db=# SELECT greatest(12, 3.4, 15.6) AS result;
 result
--------
   15.6
(1 row)

m_db=# SELECT CASE WHEN 1 < 2 THEN greatest(12, 3.4, 15.6) ELSE greatest(123.4, 23.578,
36) END AS result;
 result
--------
 15.600
(1 row)

-- Less trailing zeros
m_db=# CREATE TABLE t1 AS SELECT (false/-timestamp '2008-12-31 23:59:59.678') AS result;
INSERT 0  1
m_db=# DESC t1;
 Field  |     Type     | Null | Key | Default | Extra
--------+--------------+------+-----+---------+-------
 result | double(8,7) | YES  |     |         |
(1 row)

m_db=# SELECT (false/-timestamp '2008-12-31 23:59:59.678') AS result;
   result
------------
 -0.0000000
(1 row)

m_db=# CREATE TABLE t1 AS SELECT (CASE WHEN 1<2 THEN false/-timestamp '2008-12-31
23:59:59.678' ELSE 0016.11e3/'22.2' END) AS result;
INSERT 0 1
m_db=# DESC t1;
 Field  |  Type  | Null | Key | Default | Extra
--------+--------+------+-----+---------+-------
 result | double | YES  |     |         |
(1 row)

m_db=# SELECT (CASE WHEN 1<2 THEN false/-timestamp '2008-12-31 23:59:59.678' ELSE
0016.11e3/'22.2' END) AS result;
 result
```



```
--------
    -0
(1 row)

m_db=# DROP TABLE t1;
DROP TABLE
m_db=# CREATE TABLE t1 AS SELECT (CASE WHEN 1+1=2 THEN CASE WHEN 1<2 THEN false/-
timestamp '2008-12-31 23:59:59.678' ELSE 0016.11e3/'22.2' END ELSE 'test' END) AS result;
INSERT 0 1
m_db=# DESC t1;
 Field  |    Type     | Null | Key | Default | Extra
--------+-------------+------+-----+---------+-------
 result | varchar(23) | YES  |     |         |
(1 row)

m_db=# SELECT (CASE WHEN 1+1=2 THEN CASE WHEN 1<2 THEN false/-timestamp
'2008-12-31 23:59:59.678' ELSE 0016.11e3/'22.2' END ELSE 'test' END) AS result;
 result
--------
 -0
(1 row)
```

- When the precision transfer parameter is enabled, set operations (UNION and EXCEPT) are used. If the columns queried by the query statements involved in set operations are functions and expressions instead of directly using columns in the table, if the data type of the query result is INT or INT UNSIGNED, the return data type is different. In MySQL, the returned data type is BIGINT or BIGINT UNSIGNED. In GaussDB, the returned data type is INT/INT UNSIGNED.

```
-- Execution result in GaussDB
m_db=# SET
m_format_behavior_compat_options='select_column_name,enable_precision_decimal';
SET
m_db=# DROP TABLE IF EXISTS t1,t2,ctas1,ctas2;
DROP TABLE
m_db=# CREATE TABLE t1(a INT, b INT);
CREATE TABLE
m_db=# CREATE TABLE t2(c INT UNSIGNED, d INT UNSIGNED);
CREATE TABLE
m_db=# CREATE TABLE ctas1 AS (SELECT a, ABS(a) FROM t1) UNION (SELECT b, ABS(b) FROM
t1);
INSERT 0 0
m_db=# DESC ctas1;
 Field  |    Type     | Null | Key | Default | Extra
--------+-------------+------+-----+---------+-------
 a      | integer(11) | YES  |     |         |
 ABS(a) | integer(11) | YES  |     |         |
(2 rows)

m_db=# CREATE TABLE ctas2 AS (SELECT c, ABS(c) FROM t2) UNION (SELECT d, ABS(d) FROM
t2);
INSERT 0 0
m_db=# DESC ctas2;
 Field  |         Type         | Null | Key | Default | Extra
--------+----------------------+------+-----+---------+-------
 c      | integer(11) unsigned | YES  |     |         |
 ABS(c) | integer(11) unsigned | YES  |     |         |
(2 rows)

m_db=# DROP TABLE IF EXISTS t1,t2,ctas1,ctas2;
DROP TABLE

-- Execution result in MySQL
mysql> DROP TABLE IF EXISTS t1,t2,ctas1,ctas2;
Query OK, 0 rows affected, 4 warnings (0.00 sec)

mysql> CREATE TABLE t1(a INT, b INT);
Query OK, 0 rows affected (0.05 sec)
```



```
mysql> CREATE TABLE t2(c INT UNSIGNED, d INT UNSIGNED);
Query OK, 0 rows affected (0.03 sec)

mysql> CREATE TABLE ctas1 AS (SELECT a, ABS(a) FROM t1) UNION (SELECT b, ABS(b) FROM
t1);
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESC ctas1;
+--------+------------+------+-----+---------+-------+
| Field  | Type       | Null | Key | Default | Extra |
+--------+------------+------+-----+---------+-------+
| a      | int(11)    | YES  |     | NULL    |       |
| ABS(a) | bigint(20) | YES  |     | NULL    |       |
+--------+------------+------+-----+---------+-------+
2 rows in set (0.01 sec)

mysql> CREATE TABLE ctas2 AS (SELECT c, ABS(c) FROM t2) UNION (SELECT d, ABS(d) FROM
t2);
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESC ctas2;
+--------+----------------------+------+-----+---------+-------+
| Field  | Type                 | Null | Key | Default | Extra |
+--------+----------------------+------+-----+---------+-------+
| c      | int(11) unsigned     | YES  |     | NULL    |       |
| ABS(c) | bigint(20) unsigned | YES  |     | NULL    |       |
+--------+----------------------+------+-----+---------+-------+
2 rows in set (0.00 sec)

mysql> DROP TABLE IF EXISTS t1,t2,ctas1,ctas2;
Query OK, 0 rows affected (0.07 sec)
```

- When precision transfer is enabled, the result in the CASE WHEN nesting scenario is different from that in MySQL. In MySQL, a type can be directly converted despite multiple layers. However, in GaussDB, the result precision is determined and the type is converted layer by layer. As a result, the decimal places or carry of the result may be inconsistent with that of MySQL.

```
-- GaussDB:
m_db=# SET m_format_behavior_compat_options='enable_precision_decimal';
SET
m_db=# SELECT (CASE WHEN 1+1=3 THEN 'test' ELSE CASE WHEN 1>2 THEN '-1.5'%06.6600e1
ELSE -TIME '10:10:10.456'%2.2 END END) RES;
        res
---------------------
 -1.8559999999974321
(1 row)

-- MySQL:
mysql> SELECT (CASE WHEN 1+1=3 THEN 'test' ELSE CASE WHEN 1>2 THEN '-1.5'%06.6600e1
ELSE -TIME '10:10:10.456'%2.2 END END) RES;
+--------+
| res    |
+--------+
| -1.856 |
+--------+
1 row in set (0.00 sec)
```

- If operators of the int type (such as ~, &, |, <<, and >>) are nested in a CASE WHEN statement and the return type of the CASE WHEN statement is VARCHAR, truncation may occur in actual situations (you can determine whether truncation will occur by analyzing the original table data). In GaussDB, a warning is reported when SELECT is used for query and an error is reported when a table is created by CREATE. MySQL

does not report an error in this case. (If you want to CREATE TABLE in GaussDB, you can set **sql_mode** to disable the strict mode.)

```
-- GaussDB:
m_db=# CREATE TABLE t_base (num_var numeric(20, 10), time_var time(6));
CREATE TABLE
m_db=# INSERT INTO t_base VALUES ('-2514.1441000000','12:10:10.125000'),
('-417.2147000000',' 11:30:25.258000');
INSERT 0 2
m_db=# SELECT (~(CASE WHEN false THEN time_var ELSE num_var END)) AS res2 FROM
t_base;
WARNING:  Truncated incorrect INTEGER value: '-2514.1441000000'
CONTEXT:  referenced column: res2
WARNING:  Truncated incorrect INTEGER value: '-417.2147000000'
CONTEXT:  referenced column: res2
 res2
------
 2513
 416
(2 rows)
m_db=# CREATE TABLE t1 AS SELECT (~(CASE WHEN false THEN time_var ELSE num_var
END)) AS res2 FROM t_base;
ERROR:  Truncated incorrect INTEGER value: '-2514.1441000000'
CONTEXT:  referenced column: res2
m_db=# SET sql_mode="";
SET
m_db=# CREATE TABLE t1 AS SELECT (~(CASE WHEN false THEN time_var ELSE num_var
END)) AS res2 FROM t_base;
WARNING:  Truncated incorrect INTEGER value: '-2514.1441000000'
CONTEXT:  referenced column: res2
WARNING:  Truncated incorrect INTEGER value: '-417.2147000000'
CONTEXT:  referenced column: res2
INSERT 0 2
m_db=# DESC t1;
 Field |        Type         | Null | Key | Default | Extra
-------+---------------------+------+-----+---------+-------
 res2  | bigint(21) unsigned | YES  |     |         |
(1 row)

-- MySQL:
mysql> CREATE TABLE t_base (num_var numeric(20, 10), time_var time(6));
Query OK, 0 rows affected (0.01 sec)
mysql> INSERT INTO t_base VALUES ('-2514.1441000000','12:10:10.125000'),
('-417.2147000000',' 11:30:25.258000');
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0
mysql> SELECT (~(CASE WHEN false THEN time_var ELSE num_var END)) AS res2 FROM t_base;
+------+
| res2 |
+------+
| 2513 |
|  416 |
+------+
2 rows in set (0.00 sec)
mysql> CREATE TABLE t1 AS SELECT (~(CASE WHEN false THEN time_var ELSE num_var END))
AS res2 FROM t_base;
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> DESC t1;
+-------+---------------------+------+-----+---------+-------+
| Field | Type                | Null | Key | Default | Extra |
+-------+---------------------+------+-----+---------+-------+
| res2  | bigint(21) unsigned | YES  |     | NULL    |       |
+-------+---------------------+------+-----+---------+-------+
1 row in set (0.00 sec)
```

- When precision transfer is enabled, if constants are nested in CREATE VIEW AS SELECT CASE WHEN and SELECT CASE WHEN statements (including constant calculation and nesting functions with constants), the

values in GaussDB are the same. In MySQL, some precision may be lost in SELECT CASE WHEN statements.

```
-- GaussDB:
m_db=# CREATE OR REPLACE VIEW test_view AS
m_db-# SELECT (CASE WHEN 1<2 THEN 3.33/4.46 ELSE 003.3630/002.2600 END) c1,(CASE
WHEN 1>2 THEN IFNULL(null,3.363/2.2) ELSE NULLIF(3.33/4.46,3.363/2.2) END) c2;
CREATE VIEW
m_db=# SELECT * FROM test_view;
    c1     |    c2
-----------+-----------
 0.74663677 | 0.7466368
(1 row)
m_db=# SELECT (CASE WHEN 1<2 THEN 3.33/4.46 ELSE 003.3630/002.2600 END) c1,(CASE
WHEN 1>2 THEN IFNULL(null,3.363/2.2) ELSE NULLIF(3.33/4.46,3.363/2.2) END) c2;
    c1     |    c2
-----------+-----------
 0.74663677 | 0.7466368
(1 row)

-- MySQL:
mysql> CREATE OR REPLACE VIEW test_view AS
    -> SELECT (CASE WHEN 1<2 THEN 3.33/4.46 ELSE 003.3630/002.2600 END) c1,(CASE WHEN
1>2 THEN IFNULL(null,3.363/2.2) ELSE NULLIF(3.33/4.46,3.363/2.2) END) c2;
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT * FROM test_view;
+------------+-----------+
| c1         | c2        |
+------------+-----------+
| 0.74663677 | 0.7466368 |
+------------+-----------+
1 row in set (0.00 sec)
mysql> SELECT (CASE WHEN 1<2 THEN 3.33/4.46 ELSE 003.3630/002.2600 END) c1,(CASE
WHEN 1>2 THEN IFNULL(null,3.363/2.2) ELSE NULLIF(3.33/4.46,3.363/2.2) END) c2;
+----------+----------+
| c1       | c2       |
+----------+----------+
| 0.746637 | 0.746637 |
+----------+----------+
1 row in set (0.00 sec)
```

- When precision transfer is enabled, GaussDB supports table creation using the UNION/CASE WHEN statement. However, due to different architectures, GaussDB cannot ensure that all types of created tables are the same as those of MySQL 8.0. The scenarios where character strings and binary-related types are returned and some function nesting scenarios in MySQL are different from those in GaussDB.

```
-- GaussDB:
m_db=# CREATE TABLE IF NOT EXISTS testcase (id int, col_text1 tinytext, col_text2 text,
col_blob1 tinyblob, col_blob2 blob, col_blob3 mediumblob, col_blob4 longblob);
CREATE TABLE
m_db=# CREATE TABLE t1 AS SELECT id,(CASE WHEN id=2 THEN col_text1 ELSE 'test' END)
f35,  (CASE WHEN id=2 THEN col_text2 ELSE 'test' END) f36,(CASE WHEN id=2 THEN col_blob1
ELSE 'test' END) f41,  (CASE WHEN id=2 THEN col_blob2 ELSE 'test' END) f42,  (CASE WHEN
id=2 THEN col_blob3 ELSE 'test' END) f43,  (CASE WHEN id=2 THEN col_blob4 ELSE 'test' END)
f44 FROM testcase;
INSERT 0 0
m_db=# DESC t1;
Field |      Type      | Null | Key | Default | Extra
-------+----------------+------+-----+---------+-------
id    | integer(11)    | YES  |     |         |
f35   | varchar(255)   | YES  |     |         |
f36   | mediumtext     | YES  |     |         |
f41   | varbinary(255) | YES  |     |         |
f42   | blob           | YES  |     |         |
f43   | mediumblob     | YES  |     |         |
f44   | longblob       | YES  |     |         |
(7 rows)
```



```
m_db=# CREATE TABLE IF NOT EXISTS testtext1 (col10 text);
CREATE TABLE
m_db=# CREATE TABLE IF NOT EXISTS testtext2 (col10 text);
CREATE TABLE
m_db=# CREATE TABLE testtext AS (SELECT * FROM testtext1) UNION (SELECT * FROM
testtext2);
CREATE TABLE
m_db=# DESC testtext;
m_db=#
 Field | Type | Null | Key | Default | Extra
-------+------+------+-----+---------+-------
 col10 | text | YES  |     |         |
(1 row)

m_db=# CREATE TABLE testchar AS SELECT (SELECT lcase(-6873.4354)) a, (SELECT
sec_to_time(-485769.567)) b UNION ALL SELECT (SELECT bin(-58768923.21321)), (SELECT
asin(-0.7237465));
INSERT 0 2
m_db=# DESC testchar;
 Field |    Type     | Null | Key | Default | Extra
-------+-------------+------+-----+---------+-------
 a     | text        | YES  |     |         |
 b     | varchar(23) | YES  |     |         |
(2 rows)

m_db=# CREATE TABLE test_func (col_text char(29));
CREATE TABLE
m_db=# CREATE TABLE test1 AS SELECT * FROM ( SELECT
        GREATEST(2.22, col_text) f1, LEAST(2.22, col_text) f2,
        ADDDATE(col_text, INTERVAL '1.28.16.31' HOUR_MICROSECOND) f3,
        SUBDATE(col_text, INTERVAL '39.49.15' MINUTE_MICROSECOND) f4,
        DATE_SUB(col_text, INTERVAL '45' MICROSECOND) f5,
        DATE_ADD(col_text, INTERVAL '12.00.00.00.001' DAY_MICROSECOND) f6,
        ADDTIME(col_text, '8:20:20.3554') f7,
        SUBTIME(col_text, '8:20:20.3554') f8 FROM test_func) t1
UNION ALL
        SELECT * FROM ( SELECT
        GREATEST(2.22, col_text) f1, LEAST(2.22, col_text) f2,
        ADDDATE(col_text, INTERVAL '1.28.16.31' HOUR_MICROSECOND) f3,
        SUBDATE(col_text, INTERVAL '39.49.15' MINUTE_MICROSECOND) f4,
        DATE_SUB(col_text, INTERVAL '45' MICROSECOND) f5,
        DATE_ADD(col_text, INTERVAL '12.00.00.00.001' DAY_MICROSECOND) f6,
        ADDTIME(col_text, '8:20:20.3554') f7,
        SUBTIME(col_text, '8:20:20.3554') f8 FROM test_func) t2;
INSERT 0 0
m_db=# DESC test1;
 Field |    Type     | Null | Key | Default | Extra
-------+-------------+------+-----+---------+-------
 f1    | double      | YES  |     |         |
 f2    | double      | YES  |     |         |
 f3    | varchar(29) | YES  |     |         |
 f4    | varchar(29) | YES  |     |         |
 f5    | varchar(29) | YES  |     |         |
 f6    | varchar(29) | YES  |     |         |
 f7    | varchar(29) | YES  |     |         |
 f8    | varchar(29) | YES  |     |         |
(8 rows)

-- MySQL:
mysql> CREATE TABLE IF NOT EXISTS testcase (id int, col_text1 tinytext, col_text2 text,
col_blob1 tinyblob, col_blob2 blob, col_blob3 mediumblob, col_blob4 longblob);
Query OK, 0 rows affected (0.01 sec)
mysql> CREATE TABLE t1 AS SELECT id,(CASE WHEN id=2 THEN col_text1 ELSE 'test' END) f35,
(CASE WHEN id=2 THEN col_text2 ELSE 'test' END) f36,(CASE WHEN id=2 THEN col_blob1 ELSE
'test' END) f41,  (CASE WHEN id=2 THEN col_blob2 ELSE 'test' END) f42,  (CASE WHEN id=2
THEN col_blob3 else 'test' END) f43,  (CASE WHEN id=2 THEN col_blob4 ELSE 'test' END) f44
FROM testcase;
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0
```



```
mysql> DESC t1;
+-------+----------+------+-----+---------+-------+
| Field | Type     | Null | Key | Default | Extra |
+-------+----------+------+-----+---------+-------+
| id    | int      | YES  |     | NULL    |       |
| f35   | longtext | YES  |     | NULL    |       |
| f36   | longtext | YES  |     | NULL    |       |
| f41   | longblob | YES  |     | NULL    |       |
| f42   | longblob | YES  |     | NULL    |       |
| f43   | longblob | YES  |     | NULL    |       |
| f44   | longblob | YES  |     | NULL    |       |
+-------+----------+------+-----+---------+-------+
7 rows in set (0.00 sec)

mysql> CREATE TABLE IF NOT EXISTS testtext1 (col10 text);
Query OK, 0 rows affected (0.01 sec)

mysql> CREATE TABLE IF NOT EXISTS testtext2 (col10 text);
Query OK, 0 rows affected (0.02 sec)

mysql>  CREATE TABLE testtext AS (SELECT * FROM testtext1) UNION (SELECT * FROM
testtext2);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESC testtext;
+-------+------------+------+-----+---------+-------+
| Field | Type       | Null | Key | Default | Extra |
+-------+------------+------+-----+---------+-------+
| col10 | mediumtext | YES  |     | NULL    |       |
+-------+------------+------+-----+---------+-------+
1 row in set (0.00 sec)

mysql> SET sql_mode='';
Query OK, 0 rows affected, 1 warning (0.01 sec)

mysql> CREATE TABLE testchar AS SELECT (SELECT lcase(-6873.4354)) a, (SELECT
sec_to_time(-485769.567)) b UNION ALL SELECT (SELECT bin(-58768923.21321)), (SELECT
asin(-0.7237465));
Query OK, 2 rows affected, 1 warning (0.02 sec)
Records: 2  Duplicates: 0  Warnings: 1

mysql> DESC testchar;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| a     | varchar(21) | YES  |     | NULL    |       |
| b     | varchar(53) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
2 rows in set (0.00 sec)

mysql> CREATE TABLE test_func (col_text char(29));
Query OK, 0 rows affected (0.02 sec)

mysql> CREATE TABLE test1 AS SELECT * FROM ( SELECT
    -> GREATEST(2.22, col_text) f1, LEAST(2.22, col_text) f2,
    -> ADDDATE(col_text, INTERVAL '1.28.16.31' HOUR_MICROSECOND) f3,
    -> SUBDATE(col_text, INTERVAL '39.49.15' MINUTE_MICROSECOND) f4,
    -> DATE_SUB(col_text, INTERVAL '45' MICROSECOND) f5,
    -> DATE_ADD(col_text, INTERVAL '12.00.00.00.001' DAY_MICROSECOND) f6,
    -> ADDTIME(col_text, '8:20:20.3554') f7,
    -> SUBTIME(col_text, '8:20:20.3554') f8 FROM test_func) t1
    -> UNION ALL
    -> SELECT * FROM ( SELECT
    -> GREATEST(2.22, col_text) f1, LEAST(2.22, col_text) f2,
    -> ADDDATE(col_text, INTERVAL '1.28.16.31' HOUR_MICROSECOND) f3,
    -> SUBDATE(col_text, INTERVAL '39.49.15' MINUTE_MICROSECOND) f4,
    -> DATE_SUB(col_text, INTERVAL '45' MICROSECOND) f5,
```



```
   -> DATE_ADD(col_text, INTERVAL '12.00.00.00.001' DAY_MICROSECOND) f6,
   -> ADDTIME(col_text, '8:20:20.3554') f7,
   -> SUBTIME(col_text, '8:20:20.3554') f8 FROM test_func) t2;
   -> SUBTIME(col_text, '8:20:20.3554') f8 FROM test_func) t1
   -> UNION ALL
   -> SELECT * FROM ( SELECT
   -> GREATEST(2.22, col_text) f1, LEAST(2.22, col_text) f2,
   -> ADDDATE(col_text, INTERVAL '1.28.16.31' HOUR_MICROSECOND) f3,
   -> SUBDATE(col_text, INTERVAL '39.49.15' MINUTE_MICROSECOND) f4,
   -> DATE_SUB(col_text, INTERVAL '45' MICROSECOND) f5,
   -> DATE_ADD(col_text, INTERVAL '12.00.00.00.001' DAY_MICROSECOND) f6,
   -> ADDTIME(col_text, '8:20:20.3554') f7,
   -> SUBTIME(col_text, '8:20:20.3554') f8 FROM test_func) t2;
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESC test1;
+-------+------------+------+-----+---------+-------+
| Field | Type       | Null | Key | Default | Extra |
+-------+------------+------+-----+---------+-------+
| f1    | binary(23) | YES  |     | NULL    |       |
| f2    | binary(23) | YES  |     | NULL    |       |
| f3    | char(29)   | YES  |     | NULL    |       |
| f4    | char(29)   | YES  |     | NULL    |       |
| f5    | char(29)   | YES  |     | NULL    |       |
| f6    | char(29)   | YES  |     | NULL    |       |
| f7    | char(29)   | YES  |     | NULL    |       |
| f8    | char(29)   | YES  |     | NULL    |       |
+-------+------------+------+-----+---------+-------+
8 rows in set (0.01 sec)
```

- In the scenario where precision transfer is enabled, for the CREATE TABLE AS SELECT A % (CASE WHEN) statement, if *A* is of the DECIMAL type and the result of CASE WHEN is of the date type (DATE, TIME, or DATETIME), the two databases are different in the precision obtained by performing the modulo operation (%). The precision obtained by GaussDB is the same as that obtained by performing modulo operations on the decimal type and date type.

```
-- GaussDB: (decimal % date type case) and (numeric%date) have the same precision, that is, decimal(24,10).
m_db=# SET m_format_behavior_compat_options = 'enable_precision_decimal';
SET
m_db=# DROP TABLE IF EXISTS t1, t2;
DROP TABLE
m_db=# CREATE TABLE t1 (num_var numeric(20, 10), date_var date, time_var time(6), dt_var datetime(6));
CREATE TABLE
m_db=# CREATE TABLE t2 AS SELECT num_var % (CASE WHEN true THEN dt_var ELSE dt_var END) AS res1 FROM t1;
INSERT 0 0
m_db=# DESC t2;
 Field |      Type      | Null | Key | Default | Extra
-------+----------------+------+-----+---------+-------
 res1  | decimal(24,10) | YES  |     |         |
(1 row)

m_db=# DROP TABLE IF EXISTS t1, t2;
DROP TABLE
m_db=# CREATE TABLE t1 (num_var numeric(20, 10), date_var date, time_var time(6), dt_var datetime(6));
CREATE TABLE
m_db=# CREATE TABLE t2 AS SELECT num_var % dt_var AS res1 FROM t1;
INSERT 0 0
m_db=# DESC t2;
 Field |      Type      | Null | Key | Default | Extra
-------+----------------+------+-----+---------+-------
 res1  | decimal(24,10) | YES  |     |         |
(1 row)
```



```
-- MySQL 5.7: The precision is different. The precision of (decimal % date type case) is
decimal(65,10), and that of (numeric%date) is decimal(24,10).
mysql> DROP TABLE IF EXISTS t1, t2;
Query OK, 0 rows affected (0.02 sec)

mysql> CREATE TABLE t1 (num_var numeric(20, 10), date_var date, time_var time(6), dt_var
datetime(6));
Query OK, 0 rows affected (0.02 sec)

mysql> CREATE TABLE t2 AS SELECT num_var % (CASE WHEN true THEN dt_var ELSE dt_var
END) AS res1 FROM t1;
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESC t2;
+-------+----------------+------+-----+---------+-------+
| Field | Type           | Null | Key | Default | Extra |
+-------+----------------+------+-----+---------+-------+
| res1  | decimal(65,10) | YES  |     | NULL    |       |
+-------+----------------+------+-----+---------+-------+
1 row in set (0.00 sec)

mysql> DROP TABLE IF EXISTS t1, t2;
Query OK, 0 rows affected (0.02 sec)

mysql> CREATE TABLE t1 (num_var numeric(20, 10), date_var date, time_var time(6), dt_var
datetime(6));
Query OK, 0 rows affected (0.02 sec)

mysql> CREATE TABLE t2 AS SELECT num_var % dt_var AS res1 FROM t1;
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESC t2;
+-------+----------------+------+-----+---------+-------+
| Field | Type           | Null | Key | Default | Extra |
+-------+----------------+------+-----+---------+-------+
| res1  | decimal(24,10) | YES  |     | NULL    |       |
+-------+----------------+------+-----+---------+-------+
1 row in set (0.00 sec)

-- MySQL 8.0: The precision of (decimal % date type case) and (numeric%date) is
decimal(20,10).
mysql> DROP TABLE IF EXISTS t1, t2;
Query OK, 0 rows affected (0.02 sec)

mysql> CREATE TABLE t1 (num_var numeric(20, 10), date_var date, time_var time(6), dt_var
datetime(6));
Query OK, 0 rows affected (0.02 sec)

mysql> CREATE TABLE t2 AS SELECT num_var % (CASE WHEN true THEN dt_var ELSE dt_var
END) AS res1 FROM t1;
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESC t2;
+-------+----------------+------+-----+---------+-------+
| Field | Type           | Null | Key | Default | Extra |
+-------+----------------+------+-----+---------+-------+
| res1  | decimal(20,10) | YES  |     | NULL    |       |
+-------+----------------+------+-----+---------+-------+
1 row in set (0.00 sec)

mysql> DROP TABLE IF EXISTS t1, t2;
Query OK, 0 rows affected (0.02 sec)

mysql> CREATE TABLE t1 (num_var numeric(20, 10), date_var date, time_var time(6), dt_var
datetime(6));
```



```
Query OK, 0 rows affected (0.02 sec)

mysql> CREATE TABLE t2 AS SELECT num_var % dt_var AS res1 FROM t1;
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESC t2;
+-------+----------------+------+-----+---------+-------+
| Field | Type           | Null | Key | Default | Extra |
+-------+----------------+------+-----+---------+-------+
| res1  | decimal(20,10) | YES  |     | NULL    |       |
+-------+----------------+------+-----+---------+-------+
1 row in set (0.00 sec)
```

- When precision transfer is enabled and UNION is used, if the query statement participates in set calculation, the queried column is a constant, and the query result data type is INT or DECIMAL, the returned precision is different. In MySQL 5.7, the returned precision is related to the left/right sequence of UNION. In MySQL 8.0 and GaussDB, they are irrelevant, which means this issue is addressed.

```
-- GaussDB:
m_db=# CREATE TABLE t1 AS (SELECT -23.45 c2) UNION ALL (SELECT -45.678 c2);
INSERT 0 2
m_db=# DESC t1;
Field |     Type     | Null | Key | Default | Extra
-------+--------------+------+-----+---------+-------
c2    | decimal(5,3) | YES  |     |         |
(1 row)
m_db=# CREATE TABLE t2 AS (SELECT -45.678 c2) UNION ALL (SELECT -23.45 c2);
INSERT 0 2
m_db=# DESC t2;
Field |     Type     | Null | Key | Default | Extra
-------+--------------+------+-----+---------+-------
c2    | decimal(5,3) | YES  |     |         |
(1 row)

-- MySQL 5.7:
mysql> CREATE TABLE t1 AS (SELECT -23.45 c2) UNION ALL (SELECT -45.678 c2);
Query OK, 2 rows affected (2.28 sec)
Records: 2  Duplicates: 0  Warnings: 0
mysql> DESC t1;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| c2    | decimal(6,3) | NO   |     | 0.000   |       |
+-------+--------------+------+-----+---------+-------+
1 row in set (0.00 sec)
mysql> CREATE TABLE t2 AS (SELECT -45.678 c2) UNION ALL (SELECT -23.45 c2);
Query OK, 2 rows affected (2.22 sec)
Records: 2  Duplicates: 0  Warnings: 0
mysql> DESC t2;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| c2    | decimal(5,3) | NO   |     | 0.000   |       |
+-------+--------------+------+-----+---------+-------+
1 row in set (0.00 sec)

-- MySQL 8.0:
mysql> CREATE TABLE t1 AS (SELECT -23.45 c2) UNION ALL (SELECT -45.678 c2);
Query OK, 2 rows affected (0.02 sec)
Records: 2  Duplicates: 0  Warnings: 0
mysql> DESC t1;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| c2    | decimal(5,3) | NO   |     | 0.000   |       |
```



```
+-------+--------------+------+-----+---------+-------+
1 row in set (0.03 sec)
mysql>  CREATE TABLE t2 AS (SELECT -45.678 c2) UNION ALL (SELECT -23.45 c2);
Query OK, 2 rows affected (0.03 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> DESC t2;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| c2    | decimal(5,3) | NO   |     | 0.000   |       |
+-------+--------------+------+-----+---------+-------+
1 row in set (0.02 sec)
```

Differences in Double Colon Conversion

In GaussDB, if you use double colons to convert input parameters of a function to another type, the result may be unexpected. In MySQL, double colons do not take effect.

Example:

```
m_db=# SELECT POW("12"::VARBINARY,"12"::VARBINARY);
ERROR:  value out of range: overflow
CONTEXT:  referenced column: pow

varbinary col
m_db=# CREATE TABLE test_varbinary (
        A VARBINARY(10)
);
m_db=# INSERT INTO test_varbinary VALUES ('12');
m_db=# SELECT POW(A, A) FROM test_varbinary;
      pow
---------------
 8916100448256
(1 row)
```

Differences in Decimal Types

In CREATE TABLE ... AS (SELECT ...) statement, if data of the decimal type has 0s in the prefix, GaussDB ignores and excludes 0s during the length calculation. In MySQL 5.7, the number of 0s in the prefix is added to the total length. In MySQL 8.0, despite the number of 0s in the prefix, only 1 is added to the total length.

```
-- GaussDB
m_db=# CREATE TABLE test AS SELECT 004.01 col1;
INSERT 0 1
m_db=# DESC test;
 Field |     Type     | Null | Key | Default | Extra
-------+--------------+------+-----+---------+-------
 col1  | decimal(3,2) | YES  |     |         |
(1 row)

-- MySQL 5.7
mysql> CREATE TABLE test AS SELECT 004.01 col1;
Query OK, 1 row affected (0.02 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> DESC test;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| col1  | decimal(5,2) | NO   |     | 0.00    |       |
+-------+--------------+------+-----+---------+-------+
1 row in set (0.00 sec)

-- MySQL 8.0
mysql> CREATE TABLE test AS SELECT 004.01 col1;
```



```
Query OK, 1 row affected (0.23 sec)
Records: 1  Duplicates: 0  Warnings: 0
mysql> DESC test;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| col1  | decimal(4,2) | NO   |     | 0.00    |       |
+-------+--------------+------+-----+---------+-------+
1 row in set (0.01 sec)
```

3.2 System Functions

3.2.1 System Function Compatibility Overview

GaussDB is compatible with most MySQL system functions, but there are some differences. You are advised to use system functions supported in M-compatible mode. Do not use the original GaussDB system functions.

Currently, there are system functions in GaussDB with the same names as MySQL system functions, but these functions are not yet supported in M-compatible mode; functions listed in **Table 3-9** will prompt users that they are not supported in M-compatible mode, while functions listed in **Table 3-10** retain the behavior of the original GaussDB system functions. The behavior of functions with the same name is greatly different from that of MySQL. Therefore, you are advised to avoid using them but use only system functions in M-compatible mode.

Table 3-9 Same-name functions for which a message indicating that they are not supported in M-compatible mode is displayed

isEmpty	variance	overlaps	point	stddev_pop
stddev_samp	var_pop	var_samp	-	-

Table 3-10 Same-name functions that retain the behaviors of the original GaussDB system functions in M-compatible mode

ceil	decode	encode	format	instr
position	round	stddev	row_number	regexp_instr
regexp_like	regexp_replace	regexp_substr	-	-

When the value of the parameter **m_format_dev_version** is **s2** or later and the value of **m_format_behavior_compat_options** includes **enable_conflict_funcs**, the behavior of the functions with the same name in the following table will be modified to that in M-compatible mode.

Table 3-11 Same-name functions controlled by GUC parameters in M-compatible mode

ceil	format	instr	position	row_number

NOTE

- When the function regexp_instr, regexp_like, regexp_replace, or regexp_substr is used, if the value of the **m_format_dev_version** parameter is **'s2'** or a value indicating a later version and the value of the **m_format_behavior_compat_options** parameter contains **'enable_conflict_funcs'**, an error is reported, indicating that the behavior is not supported in M-compatible mode. Other behaviors of these functions are the same as those of functions with the same name in "SQL Reference > Functions and Operators > Character Processing Functions and Operators" in *Developer Guide*.
- In M-compatible mode, system functions have the following differences:
  - The return value type of a system function is the same as that of MySQL only when the node type of the input parameter is Var (table data) or Const (constant input). In other cases (for example, the input parameter is a calculation expression or function expression), the return value type may be different from that of MySQL.
  - In the table query scenario where LIMIT and OFFSET are used at the same time, execution layer mechanisms of GaussDB and MySQL are different. GaussDB calls functions line by line. Therefore, if an error occurs, it is reported and the execution is interrupted. However, MySQL does not execute functions line by line. Therefore, errors are not reported line by line and the execution is not interrupted, which may lead to inconsistent returned results.
  - Calling system functions by pg_catalog.func_name() is not recommended. If the called function has input parameters in the format of syntax (such as SELECT pg_catalog.substr('demo' FROM 1 FOR 2)), an error may occur when the function is called.

3.2.2 Flow Control Functions

Table 3-12 Flow control functions

Function	Differences Compared with MySQL
IF()	If the first parameter is **TRUE** and the third parameter expression contains an implicit type conversion error, or if the first parameter is **FALSE** and the second parameter expression contains an implicit type conversion error, MySQL ignores the error while GaussDB displays a type conversion error.

Function	Differences Compared with MySQL
IFNULL()	• If the first parameter is not **NULL** and the expression of the second parameter contains an implicit type conversion error, MySQL ignores the error while GaussDB displays a type conversion error. • When the input parameter is of the float type, the precision of the GaussDB result value is the same as that of MySQL 8.0. For example: m_db=# CREATE TABLE t1(c1 float); CREATE TABLE m_db=# INSERT INTO t1 VALUES(2.123); INSERT 0 1 -- Behavior in GaussDB: m_db=# SELECT ifnull(c1, c1) FROM t1; ifnull -------- 2.123 (1 row) -- Behavior in MySQL 5.7: mysql> SELECT ifnull(c1, c1) FROM t1; +--------------------+ | ifnull(c1, c1) | +--------------------+ | 2.122999906539917 | +--------------------+ 1 row in set (0.00 sec) -- Behavior in MySQL 8.0: mysql> SELECT ifnull(c1, c1) FROM t1; +-----------------+ | ifnull(c1, c1) | +-----------------+ | 2.123 | +-----------------+ 1 row in set (0.00 sec)

Function	Differences Compared with MySQL
NULLIF()	• In MySQL 5.7 and MySQL 8.0, the return value types of functions are different. Those in MySQL 8.0 are more reasonable. Therefore, the return value types of GaussDB functions are compatible with MySQL 8.0. • When the input parameter is of the float type, the precision of the GaussDB result value is the same as that of MySQL 8.0. For example: m_db=# CREATE TABLE t1(c1 float); CREATE TABLE m_db=# INSERT INTO t1 VALUES(2.123); INSERT 0 1 -- Behavior in GaussDB: m_db=# SELECT nullif(c1, 1) FROM t1; nullif -------- 2.123 (1 row) -- Behavior in MySQL 5.7: mysql> SELECT nullif(c1, 1) SELECT t1; +--------------------+ | nullif(c1, 1) | +--------------------+ | 2.122999906539917 | +--------------------+ 1 row in set (0.00 sec) -- Behavior in MySQL 8.0: mysql> SELECT nullif(c1, 1) SELECT t1; +---------------+ | nullif(c1, 1) | +---------------+ | 2.123 | +---------------+ 1 row in set (0.00 sec)

3.2.3 Date and Time Functions

The following describes the date and time functions in GaussDB M-compatible mode:

- If a SELECT subquery contains only a time function and the input parameters of the function contain columns in the table, when arithmetic operators (such as +, -, *, /, and the negation operator) are used to calculate the result, the return values of the date and time functions are truncated before the arithmetic operation.

```
m_db=# CREATE TABLE t1(int_var int);
CREATE TABLE
m_db=# INSERT INTO t1 VALUES(100);
INSERT 0 1
m_db=# SELECT (SELECT (1 * DATE_ADD('2020-10-20', interval int_var microsecond))) AS a FROM t1;
-- Truncate is not performed.
       a
----------------
 20201020000000
(1 row)

m_db=# SELECT (1 * (SELECT DATE_ADD('2020-10-20', interval int_var microsecond))) AS a FROM t1;
```



```
-- Truncation is performed.
  a
------
 2020
(1 row)

m_db=# SELECT 1 * a FROM (SELECT (SELECT 1 * DATE_ADD('2020-10-20', interval int_var
microsecond)) AS a FROM t1) AS t2; -- Truncation is not performed.
     1 * a
----------------
 20201020000000
(1 row)

m_db=# SELECT 1 * a FROM (SELECT (SELECT DATE_ADD('2020-10-20', interval int_var microsecond))
AS a FROM t1) AS t2; -- Truncation is performed.
 1 * a
-------
  2020
(1 row)
```

Table 3-13 Date and time functions

Function	Differences Compared with MySQL
ADDDATE()	-
ADDTIME()	-
CONVERT_TZ()	-
CURDATE()	-
CURRENT_DATE() /CURRENT_DATE	-
CURRENT_TIME() /CURRENT_TIME	In MySQL, an integer input value is wrapped when it reaches **255** (maximum value of a one-byte integer value), for example, **SELECT CURRENT_TIME(257) == SELECT CURRENT_TIME(1)**. GaussDB supports only valid values ranging from 0 to 6. For other values, an error is reported.
CURRENT_TIMESTAMP()/ CURRENT_TIMESTAMP	
CURTIME()	
LOCALTIME()/ LOCALTIME	
LOCALTIMESTAMP/ LOCALTIMESTAMP()	
NOW()	
SYSDATE()	
UTC_TIME()	
UTC_TIMESTAMP()	

Function	Differences Compared with MySQL
DATE()	-
DATE_ADD()	-
DATE_FORMAT()	-
DATE_SUB()	-
DATEDIFF()	-
DAY()	-
DAYNAME()	-
DAYOFMONTH()	-
DAYOFWEEK()	-
DAYOFYEAR()	-
EXTRACT()	-
FROM_DAYS()	-
FROM_UNIXTIME()	-
GET_FORMAT()	-
HOUR()	-
LAST_DAY()	-
MAKEDATE()	-
MAKETIME()	-
MICROSECOND()	-
MINUTE()	-
MONTH()	-
MONTHNAME()	-

Function	Differences Compared with MySQL
PERIOD_ADD()	In MySQL 8.0, the aforementioned issues have been resolved; consequently, the behavior of this function remains consistent with that of MySQL 8.0 in the following scenarios: • Processing integer overflow: In MySQL 5.7, the maximum value of an input parameter result of this function is **2^32=4294967296**. When the accumulated value of the month corresponding to **period** and the value of **month_number** in the input parameter or result exceed the uint32 range, integer wraparound occurs. • Performance when the value of **period** is negative: In MySQL 5.7, a negative year is parsed as an abnormal value instead of an error. Conversely, GaussDB reports an error when any input parameter or result is negative (for example, January 100 minus 10000 months). • Performance when the month in **period** exceeds the range: When dealing with a month greater than 12 or equal to 0, for example, **200013** or **199900**, MySQL 5.7 postpones it to the next year or views month 0 as December of the previous year.
PERIOD_DIFF()	
QUARTER()	-
SEC_TO_TIME()	-
SECOND()	-
STR_TO_DATE()	-
SUBDATE()	-
SUBTIME()	-
TIME()	-
TIME_FORMAT()	-
TIME_TO_SEC()	-
TIMEDIFF()	-
TIMESTAMP()	-
TIMESTAMPADD()	-
TIMESTAMPDIFF()	-
TO_DAYS()	-

Function	Differences Compared with MySQL
TO_SECONDS()	In MySQL 5.7, the precision of this function is incorrect. When the precision transfer parameter is enabled, the GaussDB precision information is normal and consistent with that in MySQL 8.0.
UNIX_TIMESTAMP()	MySQL determines whether to return a fixed-point value or an integer based on whether an input parameter contains decimal places. When operators or functions are nested in the input parameter, GaussDB may return a value of the type different from that in MySQL. If the inner node returns a value of the fixed-point, floating-point, string, or time type (excluding the DATE type), MySQL may return an integer, while GaussDB returns a fixed-point value.
UTC_DATE()	-
WEEK()	-
WEEKDAY()	-
WEEKOFYEAR()	-
YEAR()	-
YEARWEEK()	-

3.2.4 String Functions

When GaussDB uses the SQL_ASCII, the server interprets byte values 0 to 127 according to the ASCII standard, and byte values 128 to 255 are regarded as characters that cannot be parsed. If the input and output of the function contain any non-ASCII data, the database cannot convert or verify non-ASCII characters. As a result, the behavior of the function is greatly different from that of MySQL.

Table 3-14 String functions

Function	Differences Compared with MySQL
ASCII()	-
BIT_LENGTH()	-
CHAR_LENGTH()	If the database character set is SQL_ASCII, this function returns the number of bytes instead of the number of characters.
CHARACTER_LENGTH()	
CONCAT()	-
CONCAT_WS()	-
HEX()	-

Function	Differences Compared with MySQL
LENGTH()	-
LPAD()	The default maximum padding length in MySQL is **1398101**, and that in GaussDB is **1048576**. The maximum padding length varies depending on the character set. For example, if the character set is GBK, the default maximum padding length in GaussDB is **2097152**.
RPAD()	
MD5()	When the length of the inserted string of the BINARY type is less than the target length, the padding characters in GaussDB are different from those in MySQL. Therefore, when the input parameter is of the BINARY type, the function result in GaussDB is different from that in MySQL.
RANDOM_BYTES()	Both GaussDB and MySQL use OpenSSL to generate random character strings. GaussDB uses OpenSSL 3.x.x to generate random character strings. Compared with MySQL using OpenSSL 1.x.x, the performance in GaussDB may deteriorate.
REPEAT()	-
REPLACE()	If the third input parameter is **null** and the string length of the second input parameter is not 0, GaussDB returns **NULL** and MySQL may return the characters of the first parameter. For example: -- Behavior in GaussDB: m_db=# select replace('1.23', binary(1.1), null); replace --------- (1 row) -- Behavior in MySQL: mysql> select replace('1.23', binary(1.1), null); +------------------------------------+ | replace('1.23', binary(1.1), null) | +------------------------------------+ | 1.23 | +------------------------------------+ 1 row in set (0.00 sec)
SHA()/SHA1()	-
SHA2()	-
SPACE()	-
STRCMP()	-
FIND_IN_SET()	-
LCASE()	-
LEFT()	-
LOWER()	-
LTRIM()	-

Function	Differences Compared with MySQL
REVERSE()	-
RIGHT()	-
RTRIM()	-
SUBSTR()	When the collation returned by the first input parameter node is **BINARY**, MySQL may still use different collation logic (depending on the nested function), but GaussDB processes functions based on BINARY collation. As a result, the length of truncated bytes is different.
SUBSTRING()	
SUBSTRING_INDEX()	• When the third input parameter is a negative number, the comparison logic of MySQL is different from that of GaussDB, which may lead to different results. • When the third input parameter is a positive number, wraparound may occur because MySQL 5.7 stores data in int32 format, leading to an incorrect result. In MySQL 8.0, int64 is used for storage, which rectifies the problem. Therefore, GaussDB follows the setting of MySQL 8.0. However, when the input parameter value exceeds 2^63 – 1, wraparound also occurs. As a result, the obtained value of the third parameter may be a negative number, and the results are different.
TRIM()	-
UCASE()	-
UPPER()	-
UNHEX()	-
FIELD()	-
COMPRESS()	-
UNCOMPRESS()	-
UNCOMPRESS_LENGTH()	-
EXPORT_SET()	-
POSITION()	-
LOCATE()	-

Function	Differences Compared with MySQL
CHAR()	• When the CHAR function is used to specify a character set, if the transcoding fails, GaussDB reports an error, and MySQL reports a WARNING and returns **NULL**. • In MySQL, if the parameter value is the 0th to 31st or 127th code in the ASCII table, the returned result is invisible. GaussDB returns the value in hexadecimal format, such as \x01 and \x02. • In MySQL, the number of input parameters of the CHAR function is not limited. In GaussDB, the number of input parameters of the function cannot exceed 8192.
ELT()	In MySQL, the number of input parameters of the ELT function is not limited. In GaussDB, the number of input parameters of the function cannot exceed 8192.
FORMAT()	-
BIN()	-
MAKE_SET()	In MySQL 5.7, if the first parameter selected by the MAKE_SET function is of the integer, floating-point, or fixed-point type and the returned result contains non-ASCII characters, garbled characters may be displayed. In GaussDB, the displayed result is normal, which is the same as that in MySQL 8.0.
TO_BASE64()	-
FROM_BASE64()	-
ORD()	-
MID()	-
QUOTE()	• An input parameter string contains "\0" cannot be entered, because it is not supported by the character set in GaussDB. It is the escape character instead of the function itself that makes the function different in GaussDB and MySQL. • GaussDB supports a maximum of 1 GB data transfer. The maximum length of the **str** input parameter is 536870908 bytes, and the maximum size of the result string returned by the function is 1 GB. • For characters that are not padded, if the input parameter is of the BINARY type with a fixed length, null characters \0 are padded in MySQL and spaces are padded in GaussDB by default.
INSERT()	-
INSTR()	-

Function	Differences Compared with MySQL
OCTET_LENGTH( )	-

3.2.5 Type Conversion Functions

Table 3-15 Type conversion functions

Function	Differences Compared with MySQL
CAST()	• Due to different function execution mechanisms, when other functions (such as greatest and least) are nested in the cast function, the inner function returns a value less than 1. The result is different from that in MySQL. --GaussDB: m_db=# SELECT cast(least(1.23, 1.23, 0.23400) AS date); WARNING: Incorrect datetime value: '0.23400' CONTEXT: referenced column: cast cast ------ (1 row) --MySQL 5.7: mysql> SELECT cast(least(1.23, 1.23, 0.23400) AS date); +--+ | cast(least(1.23, 1.23, 0.23400) as date) | +--+ | 0000-00-00 | +--+ 1 row in set (0.00 sec) • In GaussDB, you can use **CAST(expr AS FLOAT[(p)])** or **CAST(expr AS DOUBLE)** to convert an expression to the one of the floating-point type. MySQL 5.7 does not support this conversion. • In the CAST nested subquery scenario, if the subquery statement returns the FLOAT type, an accurate value is returned in GaussDB while a distorted value is returned in MySQL 5.7. The same rule applies to the BINARY function implemented using CAST. --GaussDB m_db=# CREATE TABLE sub_query_table(myfloat float); CREATE TABLE m_db=# INSERT INTO sub_query_table(myfloat) VALUES (1.23); INSERT 0 1 m_db=# SELECT binary(SELECT myfloat FROM sub_query_table) FROM sub_query_table; binary -------- 1.23 (1 row) m_db=# SELECT cast((SELECT myfloat FROM sub_query_table) AS char); cast ------ 1.23 (1 row) --MySQL 5.7 mysql> CREATE TABLE sub_query_table(myfloat float); Query OK, 0 rows affected (0.02 sec) mysql> INSERT INTO sub_query_table(myfloat) VALUES (1.23); Query OK, 1 row affected (0.00 sec)

Function	Differences Compared with MySQL
	mysql> SELECT binary(SELECT myfloat FROM sub_query_table) FROM sub_query_table; +---+ | binary(SELECT myfloat FROM sub_query_table) | +---+ | 1.2300000190734863 | +---+ 1 row in set (0.00 sec) mysql> SELECT cast((SELECT myfloat FROM sub_query_table) AS char); +---+ | cast((SELECT myfloat FROM sub_query_table) AS char) | +---+ | 1.2300000190734863 | +---+ 1 row in set (0.00 sec) • When the JSON data type is converted and used for precision calculation, the precision consistent with that of the JSON table is used in GaussDB, which is different from that in MySQL 5.7 but the same as that in MySQL 8.0. Example: --GaussDB test=# SET m_format_behavior_compat_options='enable_precision_decimal'; SET test=# DROP TABLE tt01; DROP TABLE test=# CREATE TABLE tt01 AS SELECT -cast('98.7654321' AS json) AS c1; INSERT 0 1 test=# DESC tt01; Field | Type | Null | Key | Default | Extra -------+--------+------+-----+---------+------- c1 | double | YES | | | (1 row) test=# SELECT * FROM tt01; c1 ------------- -98.7654321 (1 row) --MySQL 5.7 mysql> SELECT version(); +------------------+ | version() | +------------------+ | 5.7.44-debug-log | +------------------+ 1 row in set (0.00 sec) mysql> DROP TABLE tt01; Query OK, 0 rows affected (0.02 sec) mysql> CREATE TABLE tt01 AS SELECT -cast('98.7654321' AS json) AS c1; Query OK, 1 row affected (0.03 sec) Records: 1 Duplicates: 0 Warnings: 0 mysql> DESC tt01; +-------+--------------+------+-----+---------+-------+ | Field | Type | Null | Key | Default | Extra |

Function	Differences Compared with MySQL
	+-------+--------------+------+-----+---------+-------+ | c1 | double(17,0) | YES | | NULL | | | +-------+--------------+------+-----+---------+-------+ 1 row in set (0.00 sec) mysql> SELECT * FROM tt01; +------+ | c1 | +------+ | -99 | +------+ 1 row in set (0.00 sec) --MySQL 8.0 mysql> SELECT version(); +--------------+ | version() | +--------------+ | 8.0.36-debug | +--------------+ 1 row in set (0.00 sec) mysql> DROP TABLE tt01; Query OK, 0 rows affected (0.05 sec) mysql> CREATE TABLE tt01 AS SELECT -cast('98.7654321' AS json) AS c1; Query OK, 1 row affected (0.12 sec) Records: 1 Duplicates: 0 Warnings: 0 mysql> DESC tt01; +-------+--------+------+-----+---------+-------+ | Field | Type | Null | Key | Default | Extra | +-------+--------+------+-----+---------+-------+ | c1 | double | YES | | NULL | | | +-------+--------+------+-----+---------+-------+ 1 row in set (0.01 sec) mysql> SELECT * FROM tt01; +-------------+ | c1 | +-------------+ | -98.7654321 | +-------------+ 1 row in set (0.00 sec)
CONVERT()	In GaussDB, you can use **CONVERT(expr, FLOAT[(p)])** or **CONVERT(expr, DOUBLE)** to convert an expression to the one of the floating-point type. MySQL 5.7 does not support this conversion.

3.2.6 Encryption Functions

Table 3-16 Encryption functions

Function	Differences Compared with MySQL
AES_DECRYPT() AES_ENCRYPT()	• GaussDB does not support ECB mode, which is an insecure encryption mode, but uses CBC mode by default. • When characters are specified to be encoded in SQL_ASCII for GaussDB, the server parses byte values 0 to 127 according to the ASCII standard, and byte values 128 to 255 cannot be parsed. If the input and output of the function contain any non-ASCII characters, the database cannot convert or verify them. • The GUC parameter **block_encryption_mode** cannot be set to a number.
PASSWORD()	• In MySQL, the GUC parameter **old_passwords** can be used to control how the hash generates passwords. – The default value of **old_passwords** is **0**. – If **old_passwords** is set to **0**, MySQL 4.1 native hashing is used for encryption. – If **old_passwords** is set to **2**, SHA-256 hashing is used for encryption. • The GUC parameter **old_passwords** is not supported in GaussDB. The behavior of the password function is only consistent with the default behavior (that is, when the value of **old_passwords** is **0**). • When the length of the inserted string of the BINARY type is less than the target length, the padding characters in GaussDB are different from those in MySQL. Therefore, when the input parameter is of the BINARY type, the function result in GaussDB is different from that in MySQL.

3.2.7 Comparison Functions

Table 3-17 Comparison functions

Function	Differences Compared with MySQL
COALESCE()	In the union distinct scenario, the precision of the return value is different from that in MySQL. If there is an implicit type conversion error in the subsequent parameter expression of the first parameter that is not **NULL**, MySQL ignores the error while GaussDB displays a type conversion error. When the parameter is a MIN or MAX function, the return value type is different from that in MySQL.
INTERVAL()	-
GREATEST()	If the input parameter of the function contains **NULL** and the function is called after the WHERE keyword, the returned result is inconsistent with that of MySQL 5.7. This problem lies in MySQL 5.7. Since MySQL 8.0 has resolved this problem, GaussDB are consistent with MySQL 8.0.
LEAST()	

Function	Differences Compared with MySQL
ISNULL()	• The return value type of a function differs in MySQL 5.7 and MySQL 8.0. Return types are compatible with MySQL 8.0 because its behavior is more appropriate. • When some aggregate functions are nested, the returned results in MySQL 5.7 and MySQL 8.0 are different in some scenarios. Return values are compatible with MySQL 8.0 because its behavior is more appropriate. m_db=# SELECT isnull(avg(1.23)); ?column? ---------- f (1 row) m_db=# SELECT isnull(group_concat(1.23)); ?column? ---------- f (1 row) m_db=# SELECT isnull(max('1.23')); ?column? ---------- f (1 row) m_db=# SELECT isnull(min(1/2)); ?column? ---------- f (1 row) m_db=# SELECT isnull(std(3.14159 * 1.2345)); ?column? ---------- f (1 row) m_db=# SELECT isnull(sum('0.23400')); ?column? ---------- f (1 row)

3.2.8 Aggregate Functions

Table 3-18 Aggregate functions

Function	Differences Compared with MySQL
AVG()	• In GaussDB, if the columns in **expr** are of the BIT, BOOL, or integer type and the sum of all rows exceeds the range of BIGINT, overflow occurs, reversing integers. • In GaussDB, the behavior is different when the input parameter of the AVG function is of the TEXT or BLOB type. – In MySQL 5.7, the return value type of AVG(TEXT/BLOB) is MEDIUMTEXT. In MySQL 8.0, the return value type of AVG(TEXT/BLOB) is DOUBLE. – In GaussDB, the return value type of AVG(TEXT/BLOB) is the same as that in MySQL 8.0.
BIT_AND()	When the input parameter of the BIT_AND function is **NULL** and the BIT_AND function is nested by other functions, the result is **–1** in MySQL 5.7 and **NULL** in MySQL 8.0. In GaussDB, the function nesting is the same as that in MySQL 8.0. -- GaussDB: m_db=# SELECT acos(bit_and(null)); acos ------ (1 row) -- MySQL 5.7: mysql> SELECT acos(bit_and(null)); +----------------------+ | acos(bit_and(null)) | +----------------------+ | 3.141592653589793 | +----------------------+ 1 row in set (0.03 sec) -- MySQL 8.0 mysql> SELECT acos(bit_and(null)); +----------------------+ | acos(bit_and(null)) | +----------------------+ | NULL | +----------------------+ 1 row in set (0.01 sec)
BIT_OR()	-
BIT_XOR()	-
COUNT()	GaussDB supports the count(tablename.*) syntax, but MySQL does not.

Function	Differences Compared with MySQL
GROUP_CONCAT()	• In GaussDB, if the parameters in GROUP_CONCAT contain both the DISTINCT and ORDER BY syntaxes, all expressions following ORDER BY must be in the DISTINCT expression. • In GaussDB, **GROUP_CONCAT(... ORDER BY** *Number***)** does not indicate the sequence of the parameter. The number is only a constant expression, which is equivalent to no sorting. • In GaussDB, the **group_concat_max_len** parameter is used to limit the maximum return length of GROUP_CONCAT. If the return length exceeds the maximum, the length is truncated. Currently, the maximum length that can be returned is **1073741823**, which is smaller than that in MySQL. • When the default UTF-8 character set is used, the maximum number of bytes of UTF-8 character set in GaussDB is different from that in MySQL. As a result, the created table structure in GaussDB is different from that in MySQL. `-- GaussDB:` `m_db=# SET` `m_format_behavior_compat_options='enable_precision_decimal';` `SET` `m_db=# CREATE TABLE t1 AS SELECT * FROM (SELECT CASE WHEN 1 < 2` `THEN group_concat(1.23, 3.24) ELSE 12.34 END v1) c1;` `INSERT 0 1` `m_db=# DESC t1;` ` Field	Type	Null	Key	Default	Extra` `-------+--------------+------+-----+---------+-------` ` v1	varchar(256)	YES		|` `(1 row)` `-- MySQL 5.7:` `mysql> CREATE TABLE t1 AS SELECT * FROM (SELECT CASE WHEN 1 < 2` `THEN group_concat(1.23, 3.24) ELSE 12.34 END v1) c1;` `Query OK, 1 row affected (0.01 sec)` `Records: 1 Duplicates: 0 Warnings: 0`  `mysql> DESC t1;` `+-------+--------------+------+-----+---------+-------+` `| Field	Type	Null	Key	Default	Extra |` `+-------+--------------+------+-----+---------+-------+` `| v1	varchar(341)	YES		NULL	|` `+-------+--------------+------+-----+---------+-------+` `1 row in set (0.00 sec)` • When the GROUP_CONCAT function is used as the input parameter of the NULLIF function, the behavior in nested scenarios is different. In MySQL 5.7, no matter whether the GROUP_CONCAT function is nested in the input parameters of NULLIF, the values of **GROUP_CONCAT** in both occasions are regarded as the same and **NULL** is returned. In MySQL 8.0, the values are regarded as unequal due to precision differences. In GaussDB, this function is nested in the same way as that in MySQL 8.0.

Function	Differences Compared with MySQL
	-- GaussDB: m_db=# SELECT nullif(group_concat(1/7), 1/7); nullif -------- 0.1429 (1 row) -- MySQL 5.7: mysql> SELECT nullif(group_concat(1/7), 1/7); +--------------------------------+ | nullif(group_concat(1/7), 1/7) | +--------------------------------+ | NULL | +--------------------------------+ 1 row in set (0.00 sec) -- MySQL 8.0: mysql> SELECT nullif(group_concat(1/7), 1/7); +--------------------------------+ | nullif(group_concat(1/7), 1/7) | +--------------------------------+ | 0.1429 | +--------------------------------+ 1 row in set (0.00 sec)
MAX() MIN()	• When the parameter is not a table column, the return value type of the MAX function is different from that of MySQL 5.7. • When precision transfer is enabled, the MAX function is nested with the time interval calculation of the time, date, datetime, or timestamp type. The return value and return type are the same as those in MySQL 8.0. • When precision transfer is enabled, the return value and return type of the MAX and INTERVAL functions are the same as those of MySQL 8.0.
SUM()	In GaussDB, if the columns in **expr** are of the BIT, BOOL, or integer type and the sum of all rows exceeds the range of BIGINT, overflow occurs, reversing integers.
STD()	-

Function	Differences Compared with MySQL
Aggregate Functions	• If **DISTINCT** is specified and the SQL statement contains the GROUP BY clause, GaussDB does not sort the results, while MySQL sorts the results. • If the ORDER BY statement contains an aggregate function, no error is reported in GaussDB, but an error is reported in MySQL. • If precision transfer is disabled (**m_format_behavior_compat_options** is not set to **'enable_precision_decimal'**), when an aggregate function uses other functions, operators, or expressions such as SELECT clauses as input parameters, for example, SELECT sum(abs(n)) FROM t, but cannot obtain the precision information transferred by the input parameter expression, the result precision of the function is different from that of MySQL. • The result of the aggregate function varies depending on the data input sequence. – For example, if ORDER BY is used together with the aggregate function, the execution sequence of the function is changed. As a result, the result is inconsistent with that in MySQL. -- Prepare a base table. CREATE TABLE test_n(col_unumeric1 decimal(4,3) unsigned, col_znumeric2 decimal(3,2) unsigned zerofill, col_znumeric3 decimal(5,3) unsigned zerofill); Query OK, 0 rows affected (0.01 sec) INSERT INTO test_n VALUES(1.010, 2.02, 3.303),(1.190, 2.29, 3.339), (1.180, 2.28, 3.338); Query OK, 3 rows affected (0.00 sec) Records: 3 Duplicates: 0 Warnings: 0 CREATE TABLE test_n_2(col_unumeric1 decimal(4,3) unsigned, col_znumeric2 decimal(3,2) unsigned zerofill, col_znumeric3 decimal(5,3) unsigned zerofill); Query OK, 0 rows affected (0.02 sec) INSERT INTO test_n_2 VALUES(1.180, 2.28, 3.338),(1.190, 2.29, 3.339), (1.010, 2.02, 3.303); Query OK, 3 rows affected (0.00 sec) Records: 3 Duplicates: 0 Warnings: 0 CREATE TABLE IF NOT EXISTS fun_op_case_tb_1 (id int, name varchar(20), col_unumeric1 NUMERIC(4,3) unsigned, col_znumeric2 DECIMAL(3,2) zerofill,col_znumeric3 DEC(5,3) zerofill); CREATE TABLE INSERT INTO fun_op_case_tb_1 (id, name, col_unumeric1, col_znumeric2, col_znumeric3) VALUES (1, 'Computer', 1.11, 2.12, 3.133), (2, 'Computer', 2.11, 2.22, 3.233), (3, 'Computer', 3.11, 2.32, 3.333), (4, 'Computer', 1.41, 2.42, 3.343), (5, 'Computer', 1.51, 2.52, 3.353), (6, 'Computer', 1.61, 2.26, 3.363), (7, 'Computer', 1.17, 2.27, 3.337), (8, 'Computer', 1.18, 2.28, 3.338),

Function	Differences Compared with MySQL

```
(9, 'Computer', 1.19, 2.29, 3.339),
(10, 'Computer', 1.01, 2.02, 3.303),
(1, 'Software', 1.11, 2.12, 3.133),
(2, 'Software', 2.11, 2.22, 3.233),
(3, 'Software', 3.11, 2.32, 3.333),
(4, 'Software', 1.41, 2.42, 3.343),
(5, 'Software', 1.51, 2.52, 3.353),
(6, 'Software', 1.61, 2.26, 3.363),
(7, 'Software', 1.17, 2.27, 3.337),
(8, 'Software', 1.18, 2.28, 3.338),
(9, 'Software', 1.19, 2.29, 3.339),
(10, 'Software', 1.01, 2.02, 3.303),
(1, 'Database', 1.11, 2.12, 3.133),
(2, 'Database', 2.11, 2.22, 3.233),
(3, 'Database', 3.11, 2.32, 3.333),
(4, 'Database', 1.41, 2.42, 3.343),
(5, 'Database', 1.51, 2.52, 3.353),
(6, 'Database', 1.61, 2.26, 3.363),
(7, 'Database', 1.17, 2.27, 3.337),
(8, 'Database', 1.18, 2.28, 3.338),
(9, 'Database', 1.19, 2.29, 3.339),
(10, 'Database', 1.01, 2.02, 3.303);
INSERT 0 30
-- GaussDB:
m_db=# SELECT * FROM test_n;
 col_unumeric1 | col_znumeric2 | col_znumeric3
---------------+---------------+---------------
         1.010 |          2.02 |        03.303
         1.190 |          2.29 |        03.339
         1.180 |          2.28 |        03.338
m_db=# SELECT * FROM test_n_2;
 col_unumeric1 | col_znumeric2 | col_znumeric3
---------------+---------------+---------------
         1.180 |          2.28 |        03.338
         1.190 |          2.29 |        03.339
         1.010 |          2.02 |        03.303
m_db=# SELECT std(col_unumeric1*(col_znumeric2 | col_znumeric3))
FROM test_n_2 ;
         std
---------------------
 0.24779023386727736
(1 row)
m_db=# SELECT std(col_unumeric1*(col_znumeric2 | col_znumeric3))
FROM test_n ;
         std
---------------------
 0.24779023386727742
(1 row)

m_db=# SELECT std(col_unumeric1*(col_znumeric2 | col_znumeric3))
FROM fun_op_case_tb_1 GROUP BY name ORDER BY name;
        std
--------------------
 1.8167446160646796
 1.8167446160646794
 1.8167446160646796
(3 rows)

--MySQL:
mysql> SELECT * FROM test_n;
+---------------+---------------+---------------+
| col_unumeric1 | col_znumeric2 | col_znumeric3 |
+---------------+---------------+---------------+
|         1.010 |          2.02 |        03.303 |
|         1.190 |          2.29 |        03.339 |
```

Function	Differences Compared with MySQL
	<pre>| 1.180 | 2.28 | 03.338 | +---------------+---------------+---------------+ 3 rows in set (0.00 sec) mysql> SELECT *FROM test_n_2; +---------------+---------------+---------------+ | col_unumeric1 | col_znumeric2 | col_znumeric3 | +---------------+---------------+---------------+ | 1.180 | 2.28 | 03.338 | | 1.190 | 2.29 | 03.339 | | 1.010 | 2.02 | 03.303 | +---------------+---------------+---------------+ 3 rows in set (0.00 sec) mysql> SELECT std(col_unumeric1*(col_znumeric2 | col_znumeric3)) FROM test_n_2 ; +---+ | std(col_unumeric1*(col_znumeric2 | col_znumeric3)) | +---+ | 0.24779023386727736 | +---+ 1 row in set (0.00 sec) mysql> SELECT std(col_unumeric1*(col_znumeric2 | col_znumeric3)) FROM test_n; +---+ | std(col_unumeric1*(col_znumeric2 | col_znumeric3)) | +---+ | 0.24779023386727742 | +---+ 1 row in set (0.00 sec) mysql> SELECT std(col_unumeric1*(col_znumeric2 | col_znumeric3)) FROM fun_op_case_tb_1 GROUP BY name ORDER BY name; +---+ | std(col_unumeric1*(col_znumeric2 | col_znumeric3)) | +---+ | 1.8167446160646794 | | 1.8167446160646794 | | 1.8167446160646794 | +---+ 3 rows in set (0.00 sec) -- Delete the base table. DROP TABLE test_n; DROP TABLE DROP TABLE test_n_2; DROP TABLE DROP TABLE fun_op_case_tb_1; DROP TABLE</pre> – For example, if WITH ROLLUP is used together with the aggregate function, the execution sequence of the function is changed. As a result, the result is inconsistent with that in MySQL. <pre>-- Prepare a base table. CREATE TABLE IF NOT EXISTS t1 (name VARCHAR(20), c1 INT(100), c2 FLOAT(7,5)); INSERT INTO t1 VALUES ('Computer', 666,-55.155), ('Computer', 789, -15.593), ('Computer', 928,-53.963), ('Computer', 666, -54.555), ('Computer', 666,-55.555), ('Database', 666,-55.155), ('Database', 789, -15.593), ('Database', 928,-53.963), ('Database', 666, -54.555),</pre>

Function	Differences Compared with MySQL
	('Database', 666,-55.555); -- GaussDB: m_db=# SELECT name, std(c1/c2) c5 FROM t1 GROUP BY name WITH rollup; name	c5 --------+-------------------- Database	15.02396266299967 Computer	15.023962662999669 	15.02396266299967 (3 rows) --MySQL mysql> SELECT name, std(c1/c2) c5 FROM t1 GROUP BY name WITH rollup; +-----------+--------------------+ | name	c5 | +-----------+--------------------+ | Database	15.023962662999669
Computer	15.023962662999669
NULL	15.02396266299967 | +-----------+--------------------+ 3 rows in set (0.00 sec) -- Delete the base table. DROP TABLE t1; DROP TABLE • If GROUP BY is used together with the aggregate function and the intermediate result of the DECIMAL data type is involved in calculation, data distortion occurs in MySQL, and GaussDB retains data in full precision. -- Prepare a base table. CREATE TABLE IF NOT EXISTS fun_op_case_tb_1 (id int,name varchar(20),col_znumeric2 DECIMAL(3,2) zerofill,col_znumeric3 DEC(5,3) zerofill, col_bit1 BIT(3), col_time2 time); INSERT INTO fun_op_case_tb_1 VALUES (1, 'Computer', 0.01, 3.130, b'101', '08:30:23.01'), (2, 'Computer', 1.20, 30.990, b'101', '08:30:23.01'), (3, 'Computer', 1.33, 43.500, b'101', '08:30:23.01'), (4, 'Computer', 2.24, 30.990, b'101', '08:30:23.01'), (5, 'Computer', 1.25, 43.600, b'101', '08:30:23.01'), (6, 'Computer', 2.20, '20.900', b'101', '08:30:23.01'), (7, 'Computer', 2.20, '20.900', b'101', '08:30:23.01'), (8, 'Computer', 2.20, '20.900', b'101', '08:30:23.01'), (9, 'Computer', 2.29, '22.780', b'101', '08:30:23.01'), (10, 'Computer', 2.02, '20.900', b'101', '08:30:23.01') -- GaussDB: m_db=# SET m_format_behavior_compat_options='enable_precision_decimal'; m_db=# SELECT avg(col_znumeric3/col_znumeric2) FROM fun_op_case_tb_1 WHERE id<=10 GROUP BY name; avg ---------------- 46.90407212526 (1 row) m_db=# SELECT sum(col_bit1/col_time2) FROM fun_op_case_tb_1 WHERE id<=10 GROUP BY name; sum -------- 0.0006 (1 rows)

Function	Differences Compared with MySQL
	--MySQL: mysql> SELECT avg(col_znumeric3/col_znumeric2) FROM fun_op_case_tb_1 WHERE id<=10 GROUP BY name; +----------------------------------+ | avg(col_znumeric3/col_znumeric2) | +----------------------------------+ | 46.90407213000 | +----------------------------------+ 1 row in set (0.00 sec) mysql> SELECT sum(col_bit1/col_time2) FROM fun_op_case_tb_1 WHERE id<=10 GROUP BY name; +--------------------------+ | sum(col_bit1/col_time2) | +--------------------------+ | 0.0010 | +--------------------------+ 1 row in set (0.00 sec) -- Delete the base table. DROP TABLE fun_op_case_tb_1; DROP TABLE

3.2.9 JSON Functions

JSON function differences: If you add escape characters as input parameters to JSON functions and other functions that allow characters, the processing is different from that in MySQL by default. To be compatible with MySQL, set the GUC parameter **standard_conforming_strings** to **off**. In this case, the escape character processing is compatible with MySQL, except when the escape character \f, \Z, \0, or \uxxxx is used.

Table 3-19 JSON functions

Function	Differences Compared with MySQL
JSON_APPEND()	-
JSON_ARRAY()	-
JSON_ARRAY_APPEND()	-
JSON_ARRAY_INSERT()	-
JSON_CONTAINS()	-
JSON_CONTAINS_PATH()	-
JSON_DEPTH()	-
JSON_EXTRACT()	-
JSON_INSERT()	-

Function	Differences Compared with MySQL
JSON_KEYS()	-
JSON_LENGTH()	-
JSON_MERGE()	-
JSON_MERGE_PATCH()	-
JSON_MERGE_PRESERVE()	-
JSON_OBJECT()	-
JSON_QUOTE()	-
JSON_REMOVE()	-
JSON_REPLACE()	-
JSON_SEARCH()	-
JSON_SET()	-
JSON_TYPE()	-
JSON_UNQUOTE()	The scenarios where escape characters \0 and \uxxxx are used are different from those in MySQL. `SELECT json_unquote('"\0"');` `mysql> SELECT json_unquote('"\0"');` `ERROR 3141 (22032): Invalid JSON text in argument 1 to function json_unquote: "Missing a closing quotation mark in string." at position 1.` `m_db=# SELECT json_unquote('"\0"');` `ERROR:  invalid byte sequence for encoding "UTF8": 0x00`
JSON_VALID()	-

3.2.10 Window Functions

Table 3-20 Window functions

Function	Differences Compared with MySQL
LAG() LEAD()	• The value range of offset *N* is different. In MySQL, *N* must be an integer in the range [0, 2^{63}-1]. In GaussDB, *N* must be an integer in the range [0, 2^{31}-1]. • The value of offset *N* varies in terms of the value format. – In MySQL, the value format is as follows: – Unsigned integer of a constant literal. – Parameter marker denoted by a question mark (?) in the PREPARE statement. – User-defined variable. – Local variable in a stored procedure. – In GaussDB, the value format is as follows: – Unsigned integer of a constant literal. – Parameter markers denoted by a question mark (?) in the PREPARE statement are not supported (current difference in the PREPARE statement). – User-defined variable. – Local variables in stored procedures are not supported. (Currently, PL/SQL does not support local variables.) • When this function is used as a subquery together with CREATE TABLE AS and no error or alarm is reported when the subquery statement of this function is executed independently, the difference is as follows: – If GaussDB is in strict or loose mode, the **CREATE TABLE AS** statement is successfully executed and a table is successfully created. – If MySQL is in strict mode, an error may be reported when the **CREATE TABLE AS** statement is executed, and table creation fails.
ROW_NUMBER()	-
RANK()	-
DENSE_RANK()	-
FIRST_VALUE()	-
LAST_VALUE()	-
PERCENT_RANK()	-
NTILE()	-

Function	Differences Compared with MySQL
Window Functions	• When the ORDER BY clause is used for sorting, the sorting of **NULL** values is different. – In MySQL, NULL values are placed at the front by default when sorted in ascending order. – In GaussDB, NULL values are placed at the end by default when sorted in ascending order. • Column aliases are used in the OVER clauses including ORDER BY and PARTITION BY. – MySQL does not support column aliases. – GaussDB supports column aliases. • When the input parameter is an expression (for example, 1 / col1), the precision of the result is different. – MySQL first calculates the expression result and rounds it off. As a result, the precision of the final result decreases. – GaussDB does not round off the result of the expression. • The binary character strings are displayed differently. – In MySQL, a binary string is encoded into a hexadecimal value. For example, '–4' is displayed as 0x2D34 after encoding. – In GaussDB, the value of the original character string is displayed. For example, '–4' is displayed as '–4'. • When the CREATE TABLE AS syntax is used to create a table and **DESC** is specified to view the table structure, the differences are as follows: – In MySQL 8.0: – If a column type in a table is BIGINT or INT, the width is not displayed. – If the width of a column type (Type) in a table is 0, the width is displayed, for example, binary(0). – In GaussDB: – If a column type in a table is BIGINT or INTEGER, the width is displayed. – If the width of a column type in a table is 0, the width is not displayed. For example, binary(0) is displayed as binary. • The execution result of a window function depends on the sequence of table data. In some scenarios (for example, GROUP BY, WHERE, or HAVING), if the table data sequence of GaussDB is different from that of MySQL, the execution result of the window function may be different. For example: – Behavior in GaussDB:

Function	Differences Compared with MySQL
	-- Preset table data. m_db=# CREATE TABLE t1(id int,name varchar(20),age int); CREATE TABLE m_db=# INSERT INTO t1(id, name,age) VALUES (1, 'zwt',90), (2, 'dda',85), (3, 'aab',90), (4, 'aac',78), (5, 'aad',85), (6, 'aae',92), (7, 'aaf',78); INSERT 0 7 m_db=# INSERT INTO t1(id, name,age) VALUES (1, 'zwt',90), (2, 'dda',85), (3, 'aab',90), (4, 'aac',78), (5, 'aad',85), (6, 'aae',92), (7, 'aaf',78); INSERT 0 7 -- The sequence of table data is different from that in MySQL. As a result, the value of **last_value** is different. m_db=# SELECT age, last_value(age) over() FROM t1 WHERE id > 0 GROUP BY age HAVING age > 10; age | last_value -----+------------ 78 | 85 90 | 85 92 | 85 85 | 85 (4 rows) m_db=# DROP TABLE IF EXISTS t1; DROP TABLE – Behavior in MySQL: # Preset table data. mysql> CREATE TABLE t1(id int,name varchar(20),age int); Query OK, 0 rows affected (0.12 sec) mysql> INSERT INTO t1(id, name,age) VALUES (1, 'zwt',90), (2, 'dda',85), (3, 'aab',90), (4, 'aac',78), (5, 'aad',85), (6, 'aae',92), (7, 'aaf',78); Query OK, 7 rows affected (0.01 sec) Records: 7 Duplicates: 0 Warnings: 0 mysql> INSERT INTO t1(id, name,age) VALUES (1, 'zwt',90), (2, 'dda',85), (3, 'aab',90), (4, 'aac',78), (5, 'aad',85), (6, 'aae',92), (7, 'aaf',78); Query OK, 7 rows affected (0.01 sec) Records: 7 Duplicates: 0 Warnings: 0 # The sequence of table data is different from that in GaussDB. As a result, the value of **last_value** is different. mysql> SELECT age, last_value(age) over() FROM t1 WHERE id > 0 GROUP BY age HAVING age > 10; +------+------------------------+ | age | last_value(age) over() | +------+------------------------+ | 90 | 92 | | 85 | 92 | | 78 | 92 | | 92 | 92 | +------+------------------------+ 4 rows in set (0.00 sec) mysql> DROP TABLE IF EXISTS t1; Query OK, 0 rows affected (0.10 sec)

3.2.11 Arithmetic Functions

Table 3-21 Arithmetic functions

Function	Differences Compared with MySQL
ABS()	-
ACOS()	-
ASIN()	-
ATAN()	-
ATAN2()	-
CEILING()	In some scenarios, the return type of the function in GaussDB is different from that in MySQL. Therefore, table columns generated by CREATE TABLE AS compared to MySQL. ● If the input parameter is of the BIGINT or BIGINT UNSIGNED type and its value contains 20 or more characters (including the sign bit), GaussDB returns an integer, whereas MySQL 5.7 returns a decimal. For example: `SET m_format_behavior_compat_options='enable_precision_decimal';` `CREATE TABLE tt AS SELECT ceiling(-9223372036854775808);` `DESC tt;` The return type of MySQL table columns is DECIMAL(16,0). The return type of GaussDB table columns is BIGINT(17). ● If the input parameter is of the NUMERIC type, the return type may be different from that in MySQL. If the parameter is a constant or table column, the same type as that in MySQL 5.7 is returned. For other types of input parameters, such as nested input parameters, the results are different. GaussDB returns the result of the NUMERIC type, whereas MySQL may return an integer. For example: `SET m_format_behavior_compat_options='enable_precision_decimal';` `CREATE TABLE t AS SELECT ceiling(abs(5.5));` `DESC t;` The return type of MySQL table columns is INT(4). The return type of GaussDB table columns is DECIMAL(3,0).
CEIL()	
FLOOR()	
COS()	-
DEGREES()	-
EXP()	-
LN()	-
LOG()	-

Function	Differences Compared with MySQL
LOG10()	-
LOG2()	-
PI()	When the precision transfer function is disabled, that is, **m_format_behavior_compat_options** is not set to **enable_precision_decimal**, the returned value of the PI function is rounded off to six decimal places in MySQL, but is rounded off to 15 decimal places in GaussDB.
POW()	-
POWER()	-
RAND()	-
SIGN()	-
SIN()	-
SQRT()	-
TAN()	-
TRUNCATE()	-
CRC32()	When the length of the inserted string of the BINARY type is less than the target length, the padding characters in GaussDB are different from those in MySQL. Therefore, when the input parameter is of the BINARY type, the function result in GaussDB is different from that in MySQL.
CONV()	-
COT()	-
RADIANS()	-

3.2.12 Network Address Functions

Table 3-22 Network address functions

Function	Differences Compared with MySQL
INET_ATON()	-
INET_NTOA()	-
INET6_ATON()	-
INET6_NTOA()	-
IS_IPV6()	-

Function	Differences Compared with MySQL
IS_IPV4()	-

3.2.13 Other Functions

Table 3-23 Other functions

Function	Differences Compared with MySQL
DATABASE()	-
UUID()	-
UUID_SHORT()	-

Function	Differences Compared with MySQL
ANY_VALUE()	• The first data record in a group is uncertain, depending on the underlying operator. For example, for the same SQL statement, GaussDB returns 5 and 4, and MySQL returns 5 and 2. CREATE TABLE t1(a INT, b INT); INSERT INTO t1 VALUES(1, 5); INSERT INTO t1 VALUES(2, 4); INSERT INTO t1 VALUES(2, 2); CREATE TABLE t2(a INT, b INT); INSERT INTO t2 VALUES(2, 7); INSERT INTO t2 VALUES(3, 9); m_db=# SELECT ANY_VALUE(t1.b) FROM t1 LEFT JOIN t2 ON t1.a=t1.b GROUP BY t1.a; any_value ---------- 5 4 (2 rows) mysql> SELECT ANY_VALUE(t1.b) FROM t1 LEFT JOIN t2 ON t1.a=t1.b GROUP BY t1.a; +-----------------+ | ANY_VALUE(t1.b) | +-----------------+ | 5 | | 2 | +-----------------+ 2 rows in set (0.04 sec) DROP TABLE t1; DROP TABLE t2; • When used with the DISTINCT keyword, if the columns to be sorted in ORDER BY are not included in the columns of the result set retrieved by the SELECT statement, the ANY_VALUE function cannot be used in GaussDB in case of errors. CREATE TABLE t1(a INT, b INT); INSERT INTO t1 VALUES(1, 2); INSERT INTO t1 VALUES(1, 3); m_db=# SELECT DISTINCT a FROM t1 ORDER BY ANY_VALUE(b); ERROR: For SELECT DISTINCT, ORDER BY expressions must appear in select list. LINE 1: SELECT DISTINCT a FROM t1 ORDER BY ANY_VALUE(b); ^ mysql> SELECT DISTINCT a FROM t1 ORDER BY ANY_VALUE(b); +------+ | a | +------+ | 1 | +------+ 1 row in set (0.00 sec) DROP TABLE t1; • When an input parameter of the ANY_VALUE function is **NULL** or of the string type, the return value type is different from that in MySQL. For example: – If the input parameter is **NULL**, the return value type in GaussDB is BIGINT and that in MySQL is binary. – If the input parameter is of the VARCHAR type, the return value type is VARCHAR in GaussDB, but may be VARCHAR or TEXT in MySQL.

Function	Differences Compared with MySQL
SLEEP()	• When the SLEEP function is being called, if you press **Ctrl**+**C** to end the process in advance, only "Cancel request sent" is displayed in GaussDB, which is different from the display information in MySQL. • In addition to the above situation, when the SLEEP function is being called in other SQL statements, if you press **Ctrl**+**C** to end the statement in advance and the operation is obtained by the SLEEP function, no error is reported; if the value is obtained by other functions in the system, an error is reported. This behavior is different from that in MySQL. • During the execution of the SLEEP function, if the process is ended by a related command (for example, **SELECT PG_TERMINATE_BACKEND(xxx);**), GaussDB reports an error, which is different from MySQL.
COLLATION()	GaussDB supports only the collation in the utf8, utf8mb4, gbk, gb18030, and latin1 character sets.
FOUND_ROWS()	-
ROW_COUNT()	• GaussDB does not have SIGNAL statements, but MySQL supports SIGNAL statements. • In GaussDB, the connection parameter **CLIENT_FOUND_ROWS** does not exist. Even if this parameter is set, it does not take effect and the number of matched rows is returned instead of the number of affected rows. Therefore, the number of affected rows is returned in a unified manner. In MySQL, the number of affected rows is affected by this parameter. • In the scenario where INSERT ON DUPLICATE KEY UPDATE triggers a conflict, the number of affected rows returned in GaussDB is different from that in MySQL. For details, see the INSERT... ON DUPLICATE KEY UPDATE syntax in difference description of **DML**.
SYSTEM_USER()	In MySQL, if **skip-name-resolve** is included in a configuration file, **127.0.0.1** or **::1** is not parsed as localhost, but GaussDB does not have related parameters and always parses **127.0.0.1** and **::1** as localhost.
DEFAULT()	GaussDB supports column aliases, but MySQL does not.

Function	Differences Compared with MySQL
BENCHMARK()	• The execution layer frameworks of MySQL and GaussDB are different. Therefore, the execution time of the same expression estimated by the function in MySQL and GaussDB is not comparable. This function is used only to compare the execution efficiency of different GaussDB expressions. • If the execution takes a long time, when you press **Ctrl** +**C** on the client, the MySQL returns **0** and ends the task. The GaussDB displays "Cancel request sent" and ends the task.
LAST_INSERT_ID( )	-
CONNECTION_I D()	-

3.3 Operators

GaussDB is compatible with most MySQL operators, but there are some differences. If not listed, the operator behavior is the native behavior of GaussDB by default. Currently, there are statements that are not supported by MySQL but supported by GaussDB. In GaussDB, they are usually used inside the system, so they are not recommended.

Operator Differences

- NULL values in ORDER BY are sorted in different ways. MySQL sorts NULL values first, while GaussDB sorts NULL values last. In GaussDB, you can use **NULLS FIRST** and **NULLS LAST** to set the sorting sequence of NULL values.
- If ORDER BY is used, the output sequence in GaussDB is consistent with that in MySQL except for the aforementioned special scenarios. If ORDER BY is not used, the output sequence in GaussDB may be different from that in MySQL.
- When using MySQL operators, use parentheses to ensure the combination of expressions. Otherwise, an error is reported. For example, SELECT 1 regexp ('12345' regexp '123').

  The GaussDB operators can be successfully executed without using parentheses to strictly combine expressions.
- NULL values are displayed in different ways. MySQL displays a NULL value as **"NULL"**. GaussDB displays a NULL value as empty.

  MySQL output:

```
mysql> SELECT NULL;
+------+
| NULL |
+------+
| NULL |
+------+
1 row in set (0.00 sec)
```

  GaussDB output:

```
m_db=# SELECT NULL;
 ?column?
----------

(1 row)
```

- After the operator is executed, the column names are displayed in different ways. MySQL displays a NULL value as **"NULL"**. GaussDB displays a NULL value as empty.
- When character strings are being converted to the double type but there is an invalid one, the alarm is reported differently. MySQL reports an error when there is an invalid constant character string, but does not report an error for an invalid column character string. GaussDB reports an error in either situation.
- The results returned by the comparison operator are different. For MySQL, **1** or **0** is returned. For GaussDB, **t** or **f** is returned.

Table 3-24 Operators

Operators	Differences Compared with MySQL
<>	MySQL supports indexes, but GaussDB does not.
<=>	MySQL supports indexes, but GaussDB does not support indexes, hash joins, or merge joins.

Operators	Differences Compared with MySQL
Row expressions	• MySQL supports row comparison using the <=> operator, but GaussDB does not support row comparison using the <=> operator. • MySQL does not support comparison between row expressions and NULL values. In GaussDB, the <, <=, =, >=, >, and <> operators can be used to compare row expressions with NULL values. • IS NULL or ISNULL operations on row expressions are not supported in MySQL. However, they are supported in GaussDB. • For operations by using operators that cannot be performed on row expressions, the error information in GaussDB is inconsistent with that in MySQL. • MySQL does not support ROW(*values*), in which *values* contains only one column of data, but GaussDB supports. GaussDB: `m_db=# SELECT (1,2) <=> row(2,3);` `ERROR:  could not determine interpretation of row comparison operator <=>` `LINE 1: SELECT (1,2) <=> row(2,3);` `^` `HINT:  unsupported operator.` `m_db=# SELECT (1,2) < NULL;` ` ?column?` `----------` `(1 row)` `m_db=# SELECT (1,2) <> NULL;` ` ?column?` `----------` `(1 row)` `m_db=# SELECT (1, 2)  IS NULL;` ` ?column?` `----------` ` f` `(1 row)` `m_db=# SELECT ISNULL((1, 2));` ` ?column?` `----------` ` f` `(1 row)` `m_db=# SELECT ROW(0,0) BETWEEN ROW(1,1) AND ROW(2,2);` `ERROR:  un support type` `m_db=#  SELECT ROW(NULL) AS x;` ` x` `----` ` ()` `(1 row)` MySQL: `mysql> SELECT (1,2) <=> row(2,3);` `+--------------------+` `| (1,2) <=> row(2,3) |` `+--------------------+` `|                  0 |` `+--------------------+` `1 row in set (0.00 sec)` `mysql> SELECT (1,2) < NULL;` `ERROR 1241 (21000): Operand should contain 2 column(s)` `mysql> SELECT (1,2) <> NULL;`

Operators	Differences Compared with MySQL
	ERROR 1241 (21000): Operand should contain 2 column(s) mysql> SELECT (1, 2) IS NULL; ERROR 1241 (21000): Operand should contain 1 column(s) mysql> SELECT ISNULL((1, 2)); ERROR 1241 (21000): Operand should contain 1 column(s) mysql> SELECT NULL BETWEEN NULL AND ROW(2,2); ERROR 1241 (21000): Operand should contain 1 column(s) mysql> SELECT ROW(NULL) AS x; ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near ') as x' at line 1
--	MySQL indicates that an operand is negated twice and the result is equal to the original operand. GaussDB indicates a comment.
!!	MySQL: The meaning of !! is the same as that of !, indicating NOT. GaussDB: ! indicates NOT. If there is a space between two exclamation marks (! !), it indicates NOT for twice. If there is no space between them (!!), it indicates factorial. **NOTE** • In GaussDB, when both factorial (!!) and NOT (!) are used, a space must be added between them. Otherwise, an error is reported. • In GaussDB, when multiple NOT operations are required, use a space between exclamation marks (! !).

Operators	Differences Compared with MySQL
[NOT] REGEXP	• GaussDB and MySQL support different metacharacters in regular expressions. For example, GaussDB allows \d to indicate digits, \w to indicate letters, digits, and underscores (_), and \s to indicate spaces. However, MySQL does not support these metacharacters and considers them as normal character strings. • In GaussDB, "\b" can match "\\b", but in MySQL, the matching will fail. • In GaussDB, a backslash (\) indicates an escape character. In MySQL, two backslashes (\\) are used. • MySQL does not support two operators to be used together. • If the input parameter of the pattern string is invalid with only the right parenthesis ()), GaussDB and MySQL 5.7 will report an error, but MySQL 8.0 will not. • In the rule of matching the de|abc sequence with de or abc, when there are empty values on the left and right of the pipe symbol (|), MySQL 5.7 will report an error, but GaussDB and MySQL 8.0 will not. • The regular expression of the tab character "\t" can match the character class [:blank:] in GaussDB and MySQL 8.0 but cannot in MySQL 5.7. • GaussDB supports non-greedy pattern matching. That is, the number of matching characters is as small as possible. A question mark (?) is added after some special characters, for example, ?? *? +? {n}? {n,}? {n,m}? MySQL 5.7 does not support non-greedy pattern matching, and the error message "Got error 'repetition-operator operand invalid' from regexp" is displayed. MySQL 8.0 already supports this function. • In the BINARY character set, the text and BLOB types are converted to the BYTEA type. The REGEXP operator does not support the BYTEA type. Therefore, the two types cannot be matched.
LIKE	MySQL: The left operand of LIKE can only be an expression of a bitwise or arithmetic operation, or expression consisting of parentheses. The right operand of LIKE can only be an expression consisting of unary operators (excluding NOT) or parentheses. GaussDB: The left and right operands of LIKE can be any expression.

Operators	Differences Compared with MySQL
[NOT] BETWEEN AND	MySQL: [NOT] BETWEEN AND is nested from right to left. The first and second operands of [NOT] BETWEEN AND can only be expressions of bitwise or arithmetic operations, or expressions consisting of parentheses. GaussDB: [NOT] BETWEEN AND is nested from left to right. The first and second operands of [NOT] BETWEEN AND can be any expression.
IN	MySQL: The left operand of IN can only be an expression of a bitwise or arithmetic operation, or expression consisting of parentheses. GaussDB: The left operand of IN can be any expression. The query in ROW IN (ROW,ROW...) format is not supported. When precision transfer is enabled and the in operator is used for data in a table, if the data in the table is of the FLOAT or DOUBLE type and includes the corresponding precision and scale, such as float (4,2) or double (4,2), GaussDB compares values based on the precision and scale, but MySQL reads values in the memory, which are distorted values, causing unequal comparison results. -- GaussDB: m_db=# CREATE TABLE test1(t_float float(4,2)); CREATE TABLE m_db=# INSERT INTO test1 VALUES(1.42),(2.42); INSERT 0 2 m_db=# SELECT t_float, t_float in (1.42,2.42) FROM test1; t_float | ?column? ---------+---------- 1.42 | t 2.42 | t (2 rows) --MySQL: mysql> CREATE TABLE test1(t_float float(4,2)); Query OK, 0 rows affected (0.01 sec) mysql> INSERT INTO test1 VALUES(1.42),(2.42); Query OK, 2 rows affected (0.00 sec) Records: 2 Duplicates: 0 Warnings: 0 mysql> SELECT t_float, t_float in (1.42,2.42) FROM test1; +---------+------------------------+ | t_float | t_float in (1.42,2.42) | +---------+------------------------+ | 1.42 | 0 | | 2.42 | 0 | +---------+------------------------+ 2 rows in set (0.00 sec)
!	MySQL: The operand of ! can only be an expression consisting of unary operators (excluding NOT) or parentheses. GaussDB: The operand of ! can be any expression.
#	MySQL supports the comment tag (#), but GaussDB does not.

Operators	Differences Compared with MySQL
BINARY	Expressions (including some functions and operators) supported by GaussDB are different from those supported by MySQL. For GaussDB-specific expressions such as "~" and "IS DISTINCT FROM", due to the higher priority of the BINARY keyword, when BINARY expr is used, BINARY is combined with the left parameters of "~" and "IS DISTINCT FROM" first. As a result, an error is reported.

Operators	Differences Compared with MySQL
Negation (-)	The type and precision of the negation result are inconsistent with those in the MySQL. `CREATE TABLE t AS SELECT - -1;` • The return type of MySQL table fields is decimal(2,0). • The return type of GaussDB table fields is integer(1). When precision transfer is enabled (**m_format_behavior_compat_options** is set to **'enable_precision_decimal'**), the precision of the negative constant data type may be different from that in MySQL. In MySQL 5.7, when the expression contains negation operators, the **max_length** of the result precision increases based on the number of the negation operators, but this will not happen in GaussDB. For example: • GaussDB: `m_db=# DROP TABLE IF EXISTS test;` `NOTICE: table "test" does not exist, skipping` `DROP TABLE` `m_db=# CREATE TABLE test as` `m_db-# SELECT format(-4.4600e3,1) f9;` `INSERT 0 1` `m_db=# DESC test;` ` Field	Type	Null	Key	Default	Extra` `-------+-------------+------+-----+---------+-------` ` f9	varchar(45)	YES		|` `(1 row)`  `m_db=# DROP TABLE IF EXISTS t1;` `NOTICE: table "t1" does not exist, skipping` `DROP TABLE` `m_db=# CREATE TABLE t1 AS SELECT cast(- -4.46 AS BINARY) c4,convert(- - -002.2600,BINARY) c14;` `INSERT 0 1` `m_db=# DESC t1;` ` Field	Type	Null	Key	Default	Extra` `-------+---------------+------+-----+---------+-------` ` c4	varbinary(5)	YES		|` ` c14	varbinary(10)	YES		|` `(2 rows)`  `m_db=# DROP VIEW IF EXISTS v2;` `NOTICE: view "v2" does not exist, skipping` `DROP VIEW` `m_db=# CREATE VIEW v2 AS SELECT cast(- -4.46 AS BINARY) c4,convert(- - -002.2600,BINARY) c14;` `CREATE VIEW` `m_db=# DESC v2;` ` Field	Type	Null	Key	Default	Extra` `-------+--------------+------+-----+---------+-------` ` c4	varbinary(5)	YES		|` ` c14	varbinary(8)	YES		|` `(2 rows)` • MySQL: `mysql> DROP TABLE IF EXISTS test;` `Query OK, 0 rows affected (0.01 sec)`  `mysql> CREATE TABLE test AS` ` -> SELECT format(-4.4600e3,1) f9;` `Query OK, 1 row affected (0.01 sec)` `Records: 1 Duplicates: 0 Warnings: 0`

Operators	Differences Compared with MySQL
	mysql> DESC test; +-------+-------------+------+-----+---------+-------+ | Field | Type | Null | Key | Default | Extra | +-------+-------------+------+-----+---------+-------+ | f9 | varchar(63) | YES | | NULL | | | +-------+-------------+------+-----+---------+-------+ 1 row in set (0.00 sec) mysql> DROP TABLE IF EXISTS t1; Query OK, 0 rows affected, 1 warning (0.00 sec) mysql> CREATE TABLE t1 AS SELECT cast(- -4.46 AS BINARY) c4,convert(- - -002.2600,BINARY) c14; Query OK, 1 row affected (0.02 sec) Records: 1 Duplicates: 0 Warnings: 0 mysql> DESC t1; +-------+---------------+------+-----+---------+-------+ | Field | Type | Null | Key | Default | Extra | +-------+---------------+------+-----+---------+-------+ | c4 | varbinary(7) | YES | | NULL | | | | c14 | varbinary(12) | YES | | NULL | | | +-------+---------------+------+-----+---------+-------+ 2 rows in set (0.00 sec) mysql> DROP VIEW IF EXISTS v2; Query OK, 0 rows affected, 1 warning (0.00 sec) mysql> CREATE VIEW v2 AS SELECT cast(- -4.46 AS BINARY) c4,convert(- - -002.2600,BINARY) c14; Query OK, 0 rows affected (0.03 sec) mysql> DESC v2; +-------+---------------+------+-----+---------+-------+ | Field | Type | Null | Key | Default | Extra | +-------+---------------+------+-----+---------+-------+ | c4 | varbinary(7) | YES | | NULL | | | | c14 | varbinary(10) | YES | | NULL | | | +-------+---------------+------+-----+---------+-------+ 2 rows in set (0.00 sec)
/**/	Comments enclosed by /**/ are not supported in GaussDB statements.

Operators	Differences Compared with MySQL
xor	The behavior of XOR in GaussDB is different from that in MySQL. The GaussDB optimizer performs constant optimization. As a result, the results that are constants are calculated first. GaussDB: m_db=# SELECT 1 xor null xor pow(200, 2000000) FROM dual; ERROR: value out of range: overflow m_db=# CREATE TABLE t1(a int, b int); CREATE TABLE m_db=# INSERT INTO t1 VALUES(2,2), (200, 2000000000); INSERT 0 2 m_db=# m_db=# m_db=# SELECT 1 xor null xor pow(a, b) FROM t1; ?column? ---------- (2 rows) MySQL: mysql> SELECT 1 xor null xor pow(200, 2000000) FROM dual; +----------+ | 1 xor null xor pow(200, 2000000) | +----------+ | NULL | +----------+ 1 row in set (0.00 sec) ysql> CREATE TABLE t1(a int, b int); Query OK, 0 rows affected (0.04 sec) mysql> INSERT INTO t1 VALUES(2,2), (200, 2000000000); Query OK, 2 rows affected (0.01 sec) Records: 2 Duplicates: 0 Warnings: 0 mysql> SELECT 1 xor null xor pow(a, b) FROM t1; +----------+ | 1 xor null xor pow(a, b) | +----------+ | NULL | | NULL | +----------+ 2 rows in set (0.00 sec)
IS NULL and IS NOT NULL	In MySQL, these operators have a lower priority than logical operators, but they have a higher priority than logical operators in GaussDB.

Operators	Differences Compared with MySQL
AND(&&), OR (||), XOR, |, & , < , > , <=, >=, =, and !=	The execution mechanism of MySQL is as follows: After the left operand is executed, the system checks whether the result is empty and then determines whether to execute the right operand. As for the execution mechanism of GaussDB, after the left and right operands are executed, the system checks whether the result is empty. If the result of the left operand is empty and an error is reported during the execution of the right operand, MySQL does not report an error but directly returns an error. GaussDB reports an error during the execution. Behavior in MySQL: `mysql> SELECT version();` `+------------------+` `| version()  |` `+------------------+` `| 5.7.44-debug-log |` `+------------------+` `1 row in set (0.00 sec)` `mysql> DROP TABLE IF EXISTS data_type_table;` `Query OK, 0 rows affected (0.02 sec)` `mysql> CREATE TABLE data_type_table (` `  -> MyBool BOOL,` `  -> MyBinary BINARY(10),` `  -> MyYear YEAR` `  -> );` `Query OK, 0 rows affected (0.02 sec)` `mysql> INSERT INTO data_type_table VALUES (TRUE, 0x1234567890, '2021');` `Query OK, 1 row affected (0.00 sec)` `mysql> SELECT (MyBool % MyBinary) | (MyBool - MyYear) FROM data_type_table;` `+----------------------------------------+` `| (MyBool % MyBinary) | (MyBool - MyYear) |` `+----------------------------------------+` `|  NULL |` `+----------------------------------------+` `1 row in set, 2 warnings (0.00 sec)` Behavior in GaussDB: `m_db=# DROP TABLE IF EXISTS data_type_table;` `DROP TABLE` `m_db=# CREATE TABLE data_type_table (` `m_db(# MyBool BOOL,` `m_db(# MyBinary BINARY(10),` `m_db(# MyYear YEAR` `m_db(# );` `CREATE TABLE` `m_db=# INSERT INTO data_type_table VALUES (TRUE, 0x1234567890, '2021');` `INSERT 0 1` `m_db=# SELECT (MyBool % MyBinary) | (MyBool - MyYear) FROM data_type_table;` `WARNING:  Truncated incorrect double value: '4Vx  '` `CONTEXT:  referenced column: (MyBool % MyBinary) | (MyBool - MyYear)` `WARNING:  division by zero` `CONTEXT:  referenced column: (MyBool % MyBinary) | (MyBool - MyYear)` `ERROR:  Bigint is out of range.` `CONTEXT:  referenced column: (MyBool % MyBinary) | (MyBool - MyYear)`

Operators	Differences Compared with MySQL
+, -, *, /, %, mod, div	When the b "constant is embedded in the CREATE VIEW AS SELECT *arithmetic operator* ('+', '-', '*', '/', '%', 'mod', or 'div'), the return type in MySQL 5.7 may contain the unsigned identifier, but in GaussDB, the return type does not contain the unsigned identifier. MySQL output: mysql> CREATE VIEW v22 as SELECT b'101' / b'101' c22; Query OK, 0 rows affected (0.00 sec) mysql> DESC v22; +-------+-----------------------+------+-----+---------+-------+ | Field | Type | Null | Key | Default | Extra | +-------+-----------------------+------+-----+---------+-------+ | c22 | decimal(5,4) unsigned | YES | | NULL | | +-------+-----------------------+------+-----+---------+-------+ 1 row in set (0.01 sec) GaussDB output: m_db=# CREATE VIEW v22 AS SELECT b'101' / b'101' c22; CREATE VIEW m_db=# DESC v22; Field | Type | Null | Key | Default | Extra -------+--------------+------+-----+---------+------- c22 | decimal(5,4) | YES | | | (1 row)

Table 3-25 Differences in operator combinations

Example of Operator Combination	MySQL	GaussDB	Description
SELECT 1 LIKE 3 & 1;	Not supported	Supported	The right operand of LIKE cannot be an expression consisting of bitwise operators.
SELECT 1 LIKE 1 +1;	Not supported	Supported	The right operand of LIKE cannot be an expression consisting of arithmetic operators.
SELECT 1 LIKE NOT 0;	Not supported	Supported	The right operand of LIKE can only be an expression consisting of unary operators (such as +, -, or ! but except NOT) or parentheses.
SELECT 1 BETWEEN 1 AND 2 BETWEEN 2 AND 3;	Right-to-left combination	Left-to-right combination	You are advised to add parentheses to specify the calculation priority to prevent result deviation caused by sequence differences.

Example of Operator Combination	MySQL	GaussDB	Description
SELECT 2 BETWEEN 1=1 AND 3;	Not supported	Supported	The second operand of BETWEEN cannot be an expression consisting of comparison operators.
SELECT 0 LIKE 0 BETWEEN 1 AND 2;	Not supported	Supported	The first operand of BETWEEN cannot be an expression consisting of pattern matching operators.
SELECT 1 IN (1) BETWEEN 0 AND 3;	Not supported	Supported	The first operand of BETWEEN cannot be an expression consisting of IN operators.
SELECT 1 IN (1) IN (1);	Not supported	Supported	The second left operand of the IN expression cannot be an expression consisting of INs.
SELECT ! NOT 1;	Not supported	Supported	The operand of ! can only be an expression consisting of unary operators (such as +, -, or ! but except NOT) or parentheses.

Index Differences

- Currently, GaussDB supports only UB-tree and B-tree indexes.
- When LIKE fuzzy match is executed and the execution plan is printed using EXPLAIN, the execution plan displays the minimum/maximum character weight codes of the collation corresponding to the current index column. The displayed codes may vary depending on the character set of a client, but it does not affect the accuracy of the LIKE fuzzy match query result.
- In the B-tree/UB-tree index scenario, the original logic of the native GaussDB is retained. That is, index scan supports comparison of types in the same operator family, but does not support other index types currently.
- When GaussDB JDBC is used to connect to the database, the YEAR type of GaussDB cannot use indexes in the PBE scenario that contains bind parameters.
- In the operation scenarios involving index column type and constant type, the conditions that indexes of a WHERE clause are supported in GaussDB is different from those in MySQL, as shown in **Table 3-26**. For example, GaussDB does not support indexes in the following statement:

```
CREATE TABLE t(_int int);
CREATE INDEX idx ON t(_int) USING BTREE;
SELECT * FROM t WHERE _int >  2.0;
```


NOTE

- In the operation scenarios involving index column type and constant type in the WHERE clause, you can use the cast function to explicitly convert the constant type to the column type for indexing.
  ```
  SELECT * FROM t WHERE _int > cast(2.0 AS signed);
  ```
- During LIKE fuzzy match, the maximum length of a prefix index created in GaussDB is 2676 bytes (3072 bytes for MySQL) in the single-byte character set scenario. If the length exceeds the maximum, you are advised to create a non-prefix index.

Table 3-26 Differences in index support

Index Column Type	Constant Type	Supported in GaussDB	Supported in MySQL
BIT	BIT	No	Yes
	Integer	No	Yes
	Floating-point	No	Yes
	String	No	Yes
	Binary	No	Yes
	Time with date	No	No
	TIME	No	Yes
	YEAR	No	Yes
SET/ENUM	String	No	No
	Binary	No	No
	Integer	No	No
	Floating-point	No	No
	Fixed-point	No	No
	Time with date	No	No
	TIME	No	No
	YEAR	No	No
TIME	TIME	Yes	Yes
	Time with date	Yes	Yes
	YEAR	Yes	Yes

Index Column Type	Constant Type	Supported in GaussDB	Supported in MySQL
	Integer (that can be converted to the TIME type)	Yes	Yes
	Integer (that cannot be converted to the TIME type)	No	No
	Character string (that can be converted to the TIME type)	Yes	Yes
	Character string (that cannot be converted to the TIME type)	No	No
	Binary (that can be converted to the TIME type)	Yes	Yes
	Binary (that cannot be converted to the TIME type)	No	No
	Floating-point (that can be converted to the TIME type)	Yes	Yes
	Floating-point (that cannot be converted to the TIME type)	No	No

Index Column Type	Constant Type	Supported in GaussDB	Supported in MySQL
	Fixed-point (that can be converted to the TIME type)	Yes	Yes
	Fixed-point (that cannot be converted to the TIME type)	No	No
YEAR	YEAR	Yes	Yes
	Integer (that can be converted to the YEAR type)	Yes	Yes
	Integer (that cannot be converted to YEAR type and is negative)	No	Yes
	Floating-point (that can be converted to the YEAR type)	Yes	Yes
	Floating-point (that cannot be converted to the YEAR type)	No	No
	Fixed-point (that can be converted to the YEAR type)	Yes	Yes
	Fixed-point (that cannot be converted to the YEAR type)	No	No

Index Column Type	Constant Type	Supported in GaussDB	Supported in MySQL
	Character string (that can be converted to the YEAR type)	Yes	Yes
	Character string (that cannot be converted to the YEAR type)	No	No
	Binary (that can be converted to the YEAR type)	Yes	Yes
	Binary (that cannot be converted to the YEAR type)	No	No
	Time with date	No	Yes
	TIME	No	Yes
Time with date	Time with date	Yes	Yes
	TIME	Yes	Yes
	YEAR	Yes	No
	Integer (that can be converted to the time type with date)	Yes	Yes
	Integer (that cannot be converted to the time type with date)	No	No

Index Column Type	Constant Type	Supported in GaussDB	Supported in MySQL
	Floating-point (that can be converted to the time type with date)	Yes	Yes
	Floating-point (that cannot be converted to the time type with date)	No	No
	Fixed-point (that can be converted to the time type with date)	Yes	Yes
	Fixed-point (that cannot be converted to the time type with date)	No	No
	Character string (that can be converted to the time type with date)	Yes	Yes
	Character string (that cannot be converted to the time type with date)	No	No
	Binary (that can be converted to the time type with date)	Yes	Yes
	Binary (that cannot be converted to the time type with date)	No	No

Index Column Type	Constant Type	Supported in GaussDB	Supported in MySQL
Fixed-point	Fixed-point	Yes	Yes
	Integer	Yes	Yes
	Floating-point	No	Yes
	String	No	Yes
	Binary	No	Yes
	Time with date	No	Yes
	TIME	No	Yes
	YEAR	Yes	Yes
Floating-point	Floating-point	Yes	Yes
	Integer	Yes	Yes
	Fixed-point	Yes	Yes
	String	Yes	Yes
	Binary	Yes	Yes
	Time with date	Yes	Yes
	TIME	Yes	Yes
	YEAR	Yes	Yes
Integer	Integer	Yes	Yes
	Floating-point	No	Yes
	Fixed-point	Yes	Yes
	Character string (can be converted to the integer type)	Yes	Yes
	Character string (that cannot be converted to the integer type)	No	Yes

Index Column Type	Constant Type	Supported in GaussDB	Supported in MySQL
	Binary type (can be converted to the integer type)	Yes	Yes
	Binary type (cannot be converted to the integer type)	No	Yes
	Time with date	Yes	Yes
	TIME	Yes	Yes
	YEAR	Yes	Yes
Binary	Binary	Yes	Yes
	String	Yes	Yes
	Time with date	No	No
	TIME	Yes	No
	YEAR	No	No
	Integer	No	No
	Floating-point	No	No
	Fixed-point	No	No
String	String	Yes	Yes
	Time with date	No	No
	TIME	Yes	No
	YEAR	No	No
	Binary	No	No
	Integer	No	No
	Floating-point	No	No
	Fixed-point	No	No

NOTE

- Only when the **disable_int_cmp_num_index** option is not enabled for the GUC parameter **m_format_behavior_compat_options**, indexes can be used to compare integer index columns with fixed-point constants.
- Fixed-point constants cannot be user-defined variables, case when expressions, or subquery return values if indexes are used to compare integer index columns with fixed-point constants.

3.4 Character Sets

GaussDB allows you to specify the following character sets for databases, schemas, tables, or columns. The default one is utf8.

Table 3-27 Character sets

MySQL	GaussDB
utf8mb4	Supported.
utf8	Supported.
gbk	Supported.
gb18030	Supported.
binary	Supported.
latin1	Supported.

NOTE

- GaussDB regards utf8 and utf8mb4 as the same character set. The maximum length of the code is 4 bytes. If the current character set is **utf8** and the collation is set to **utf8mb4_bin**, **utf8mb4_general_ci**, **utf8mb4_unicode_ci**, or **utf8mb4_0900_ai_ci** (for example, by running **SELECT _utf8'a' collate utf8mb4_bin**), MySQL reports an error but GaussDB does not. The difference also exists when the character set is **utf8mb4** and the collation is set to **utf8_bin**, **utf8_general_ci**, or **utf8_unicode_ci**.
- The lexical syntax is parsed based on the byte stream. When multi-byte characters contain codes consistent with symbols like '\', '\'' and '\\', the behavior in GaussDB is inconsistent with that in MySQL. You are advised to temporarily disable the escape character feature. For details, see the **enable_escape_string** option of the GUC parameter **m_format_behavior_compat_options** in "Configuring GUC Parameters > GUC Parameters > Version and Platform Compatibility > Platform and Client Compatibility" in *Administrator Guide*.
- GaussDB does not strictly verify the encoding logic of invalid characters that do not belong to the current character set, which may allow such invalid characters to be successfully entered. Conversely, MySQL will report an error upon verifying such characters.

3.5 Collation Rules

GaussDB allows you to specify the following collation rules for schemas, tables, or columns.

NOTE

Differences in collation rules:

- In GaussDB, only the character string type and some binary types support the specified collation rules. You can check whether the typcollation attribute of a type in the pg_type system catalog is 0 to determine whether the type supports the collation. The collation can be specified for all types in MySQL. However, collation rules are meaningless except those for character strings and binary types.
- In GaussDB, a character set must be the same as the database-level character set, which is also the prerequisite of specifying collation rules (except binary). Multiple character sets cannot be used together in a table.
- The default collation of the utf8mb4 character set is utf8mb4_general_ci, which is the same as that in MySQL 5.7.
- To use the latin1 collation, you need to set the compatibility parameter **m_format_dev_version** to **'s2'**.

Table 3-28 Collation rules

MySQL	GaussDB
utf8mb4_general_ci	Supported.
utf8mb4_unicode_ci	Supported.
utf8mb4_bin	Supported.
gbk_chinese_ci	Supported.
gbk_bin	Supported.
gb18030_chinese_ci	Supported.
gb18030_bin	Supported.
binary	Supported.
utf8mb4_0900_ai_ci	Supported.
utf8_general_ci	Supported.
utf8_bin	Supported.
utf8_unicode_ci	Supported.
latin1_swedish_ci	Supported.
latin1_bin	Supported.

3.6 Transactions

GaussDB is compatible with MySQL transactions, but there are some differences. This section describes transaction-related differences in GaussDB.

Default Transaction Isolation Levels

The default isolation level in GaussDB is READ COMMITTED, and that of MySQL is REPEATABLE-READ.

```
-- View the current transaction isolation level.
m_db=# SHOW transaction_isolation;
 transaction_isolation
-----------------------
 read committed
(1 row)
```

Sub-transactions

In GaussDB, you can use SAVEPOINT to create a savepoint (sub-transaction) in the current transaction, and ROLLBACK TO SAVEPOINT to roll back to the savepoint. After the rollback, the current transaction can continue to run, whose status will not be affected.

No savepoint (sub-transaction) can be created in MySQL.

Nested Transactions

A nested transaction refers to a new transaction started in a transaction block.

In GaussDB, if you run a command to start a new transaction in a normal transaction block, a warning is displayed, indicating an ongoing transaction exists and the command will be ignored; while if you do so in an abnormal transaction block, an error is reported and the transaction can only be started after **ROLLBACK** is executed or **COMMIT** is used to commit other statements.

In MySQL, if a new transaction is started in a normal transaction block, the previous transaction is committed and then the new transaction is started. If a new transaction is started in an abnormal transaction block, the error is ignored, and the previous error-free statement is committed and the new transaction is started.

```
-- In GaussDB, if a new transaction is started in a normal transaction block, a warning is generated and the transaction is ignored.
m_db=# DROP TABLE IF EXISTS test_t;
m_db=# CREATE TABLE test_t(a int, b int);
m_db=# BEGIN;
m_db=# INSERT INTO test_t values(1, 2);
m_db=# BEGIN; -- The warning "There is already a transaction in progress" is displayed.
m_db=# SELECT * FROM test_t ORDER BY 1;
m_db=# COMMIT;

-- In GaussDB, if a new transaction is started in an abnormal transaction block, an error is reported. The transaction can be executed only after ROLLBACK or COMMIT is executed.
m_db=# BEGIN;
m_db=# ERROR sql; -- Error statement.
m_db=# BEGIN; -- An error is reported.
m_db=# COMMIT; -- It can be executed only after ROLLBACK/COMMIT is executed.
```

Statements Committed Implicitly

Databases under GaussDB use GaussDB for storage and inherits the GaussDB transaction mechanism. If a DDL or DCL statement is executed in a transaction, the transaction is not automatically committed.

In MySQL, if DDL, DCL, management-related, or lock-related statements are executed, the transaction is automatically committed.

```
-- In GaussDB, table creation and GUC parameter settings can be rolled back.
m_db=# DROP TABLE IF EXISTS test_table_rollback;
m_db=# BEGIN;
m_db=# CREATE TABLE test_table_rollback(a int, b int);
m_db=# \d test_table_rollback;
m_db=# ROLLBACK;
m_db=# \d test_table_rollback; -- This table does not exist.
Did not find any relation named "test_table_rollback".
```

Differences in SET TRANSACTION

In GaussDB, if SET TRANSACTION is used to set the isolation level or transaction access mode for multiple times, only the last setting takes effect. Transaction features can be separated by spaces and commas (,).

In MySQL, SET TRANSACTION cannot be used to set the isolation level or transaction access mode for multiple times. Transaction features can only be separated by commas (,).

Table 3-29 Differences in SET TRANSACTION

Syntax	Description	Difference
SET TRANSACTION	Sets transactions.	In GaussDB, if the **m_format_dev_version** parameter is not set to **'s2'**, SET TRANSACTION takes effect at the session level, with the same functionality as SET SESSION TRANSACTION. If the **m_format_dev_version** parameter is set to **'s2'**, SET TRANSACTION sets the next transaction feature. In MySQL, SET TRANSACTION takes effect in the next transaction.
SET SESSION TRANSACTION	Sets session-level transactions.	-
SET GLOBAL TRANSACTION	Sets global session-level transactions. This feature applies to subsequent sessions and has no impact on the current session.	In GaussDB, GLOBAL takes effect in transactions at the global session level and applies only to the current database instance. In MySQL, this feature takes effect in all databases.

```
-- SET TRANSACTION takes effect in session-level transactions.
m_db=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
m_db=# SHOW transaction_isolation;
m_db=# SHOW transaction_read_only;
```



```
-- In GaussDB, if the isolation level or transaction access mode is set for multiple times, only the last one
takes effect.
m_db=# SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED, ISOLATION LEVEL
REPEATABLE READ, READ WRITE, READ ONLY;
m_db=# SHOW transaction_isolation; -- repeatable read
 transaction_isolation
-----------------------
 repeatable read
(1 row)
m_db=# SHOW transaction_read_only; -- on
 transaction_read_only
-----------------------
 on
(1 row)
```

Differences in START TRANSACTION

In GaussDB, when START TRANSACTION is used to start a transaction, the isolation level can be set. If the isolation level or transaction access mode is set for multiple times, only the last setting takes effect. In the current version, consistency snapshot cannot be enabled immediately. Transaction features can be separated by spaces or commas (,).

In MySQL, if START TRANSACTION is used to start a transaction, the isolation level cannot be set and the transaction access mode cannot be set for multiple times. Transaction features can only be separated by commas (,).

In MySQL, a transaction at the repeatable read isolation level starts snapshot read only after the first SELECT statement is executed. In GaussDB, once a transaction is started, not only the first SELECT statement performs snapshot read, but also the first executed DDL, DML, or DCL statement creates a consistent read snapshot of the transaction.

```
-- Start a transaction and set the isolation level.
m_db=# START TRANSACTION ISOLATION LEVEL READ COMMITTED;
m_db=# COMMIT;
-- Set the access mode for multiple times.
m_db=# START TRANSACTION READ ONLY, READ WRITE;
m_db=# COMMIT;
```

Transaction-related GUC Parameters

Table 3-30 Differences in transaction-related GUC parameters

GUC Parameter	Description	Difference
autocommit	Sets the automatic transaction commit mode.	-

GUC Parameter	Description	Difference
transaction_isolation	Sets the isolation level of the current transaction in GaussDB. Sets the isolation level of a session-level transaction in MySQL.	• In GaussDB, you can only change the isolation level of the current transaction by running the **SET transaction_isolation =** *value* command. To change the session-level isolation level, use **default_transaction_isolation**. In MySQL, you can run the **SET** command to change the isolation level of a session-level transaction. • The supported range is different. MySQL supports the following isolation levels, which are case-insensitive but space-sensitive: – READ-COMMITTED – READ-UNCOMMITTED – REPEATABLE-READ – SERIALIZABLE GaussDB supports the following isolation levels, which are case-sensitive and space-sensitive: – read committed – read uncommitted – repeatable read – serializable – default (The level is set to be the same as the default isolation level in the session.) • In GaussDB, the value of **transaction_isolation** of a new transaction is initialized to the value of **default_transaction_isolation**. • When **m_format_dev_version** is set to **'s2'**: – You can set the features of the next transaction by running **set @@transaction_isolation = value; set transaction isolation level value**. – You can modify the features of a session-level transaction by running **set [local| session|@@session.] transaction_isolation = value**. – The features of the next transaction cannot be used within the current transaction. If an error is reported for an implicit transaction, that is, a single SQL statement, the features of the next transaction are retained.

GUC Parameter	Description	Difference
tx_isolation	Sets the transaction isolation level. **tx_isolation** and **transaction_isolation** are synonyms.	In GaussDB, only the select @@ syntax can be used for query. The show syntax cannot be used for query or modification.
default_transaction_isolation	Sets the transaction isolation level.	In GaussDB, the **SET** command is used to change the isolation level of a session-level transaction. MySQL does not support this system parameter.
transaction_read_only	In GaussDB, this parameter is used to set the access mode of the current transaction. In MySQL, this parameter is used to set the access mode of session-level transactions.	• In GaussDB, only the access mode of the current transaction can be changed by using the **SET** command. If you want to change the access mode of a session-level transaction, you can use **default_transaction_read_only**. In MySQL, you can run the **SET** command to change the isolation level of a session-level transaction. • In GaussDB, the value of **transaction_read_only** of a new transaction is initialized to the value of **default_transaction_read_only**. • When **m_format_dev_version** is set to **'s2'**: – You can set the next transaction feature by running **set @@transaction_read_only = value; set transaction {read write | read only}**. – You can modify the features of a session-level transaction by running **set [local| session|@@session.] transaction_read_only = value**. – The features of the next transaction cannot be used within the current transaction. If an error is reported for an implicit transaction, that is, a single SQL statement, the features of the next transaction are retained.

GUC Parameter	Description	Difference
tx_read_only	Sets the access mode of a transaction. **tx_read_only** and **transaction_read_only** are synonyms.	In GaussDB, only the select @@ syntax can be used for query. The show syntax cannot be used for query or modification.
default_transaction_read_only	Sets the access mode of a transaction.	In GaussDB, the **SET** command is used to change the access mode of a session-level transaction. MySQL does not support this system parameter.

3.7 SQL

3.7.1 Keywords

The constraint differences are as follows:

- If a keyword is a reserved one in GaussDB but non-reserved in MySQL, it cannot be a table name, column name, column alias, AS column alias, AS table alias, table alias, function name, or variable name in GaussDB, but can be any of these names or aliases in MySQL.
- If a keyword is a non-reserved one in GaussDB but reserved in MySQL, it can be a table name, column name, column alias, AS column alias, AS table alias, table alias, function name, or variable name in GaussDB, but cannot be any of these names or aliases in MySQL.
- If a keyword is a reserved one (function or type) both in GaussDB and MySQL, it can be a column alias, AS column alias, function name, or variable name in GaussDB, but cannot be any of these names or aliases in MySQL.
- If a keyword is a reserved one (function or type) in GaussDB but non-reserved in MySQL, it cannot be a table name, column name, AS table alias, or table alias in GaussDB, but can be one of these names or aliases in MySQL.
- If a keyword is a non-reserved one (excluding function and type) in GaussDB but reserved in MySQL, it can be a table name, column name, column alias, AS column alias, AS table alias, table alias, function name, or variable name in GaussDB, but cannot be any of these names or aliases in MySQL.
- If a keyword is a non-reserved one (excluding function and type) both in GaussDB and MySQL, it cannot be a function name in GaussDB, but can be a function name in MySQL.

NOTE

Among non-reserved keywords, reserved keywords (functions or types), and non-reserved keywords (not functions or types) in GaussDB, the following keywords cannot be column aliases:

BETWEEN, BIGINT, BLOB, CHAR, CHARACTER, CROSS, DEC, DECIMAL, DIV, DOUBLE, EXISTS, FLOAT, FLOAT4, FLOAT8, GROUPING, INNER, INOUT, INT, INT1, INT2, INT3, INT4, INT8, INTEGER, JOIN, LEFT, LIKE, LONGBLOB, LONGTEXT, MEDIUMBLOB, MEDIUMINT, MEDIUMTEXT, MOD, NATURAL, NUMERIC, OUT, OUTER, PRECISION, REAL, RIGHT, ROW, ROW_NUMBER, SIGNED, SMALLINT, SOUNDS, TINYBLOB, TINYINT, TINYTEXT, VALUES, VARCHAR, VARYING, and WITHOUT.

SIGNED and WITHOUT can be used as column aliases in MySQL.

3.7.2 Identifiers

Differences in identifiers of GaussDB are as follows:

- In GaussDB, unquoted identifiers cannot start with a dollar sign ($). In MySQL unquoted identifiers can start with a dollar sign ($).
- GaussDB unquoted identifiers support case-sensitive database objects.
- GaussDB identifiers support extended characters from U+0080 to U+00FF. MySQL identifiers support extended characters from U+0080 to U+FFFF.
- As for unquoted identifier, a table that starts with a digit and ends with an e or E as the identifier cannot be created in GaussDB. For example:

```
-- GaussDB reports an error indicating that this operation is not supported. MySQL supports this operation.
m_db=# CREATE TABLE 23e(c1 int);
ERROR:  syntax error at or near "23"
LINE 1: CREATE TABLE 23e(c1 int);
                     ^
m_db=# CREATE TABLE t1(23E int);
ERROR:  syntax error at or near "23"
LINE 1: CREATE TABLE t1(23E int);
                        ^
```

- As for quoted identifiers, tables whose column names contain only digits or scientific computing cannot be directly used in GaussDB. You need to use them in quotes. This rule also applies to the dot operator (.) scenarios. For example:

```
-- Create a table whose column names contain only numbers or scientific computing.
m_db=# CREATE TABLE t1(`123` int, `1e3` int, `1e` int);
CREATE TABLE

-- Insert data into the table.
m_db=# INSERT INTO t1 VALUES(7, 8, 9);
INSERT 0 1

-- The result is not as expected, but is the same as that in MySQL.
m_db=# SELECT 123 FROM t1;
 ?column?
----------
      123
(1 row)

-- The result is not as expected, but is the same as that in MySQL.
m_db=# SELECT 1e3 FROM t1;
 ?column?
----------
     1000
(1 row)

-- The result is not as expected and is not the same as that in MySQL.
```



```
m_db=# SELECT 1e FROM t1;
 e
---
 1
(1 row)

-- The correct way to use is as follows:
m_db=# SELECT `123` FROM t1;
 123
-----
   7
(1 row)

m_db=# SELECT `1e3` FROM t1;
 1e3
-----
   8
(1 row)

m_db=# SELECT `1e` FROM t1;
 1e
----
  9
(1 row)

-- Dot operator scenarios are not supported by GaussDB but supported by MySQL.
m_db=# SELECT t1.123 FROM t1;
ERROR:  syntax error at or near ".123"
LINE 1: SELECT t1.123 FROM t1;
                ^
m_db=# SELECT t1.1e3 FROM t1;
ERROR:  syntax error at or near "1e3"
LINE 1: SELECT t1.1e3 FROM t1;
                 ^
m_db=# SELECT t1.1e FROM t1;
ERROR:  syntax error at or near "1"
LINE 1: SELECT t1.1e FROM t1;
                 ^
-- The correct way to use in dot operator scenarios is as follows:
m_db=# SELECT t1.`123` FROM t1;
 123
-----
   7
(1 row)

m_db=# SELECT t1.`1e3` FROM t1;
 1e3
-----
   8
(1 row)

m_db=# SELECT t1.`1e` FROM t1;
 1e
----
  9
(1 row)

m_db=# DROP TABLE t1;
DROP TABLE
```

- In GaussDB, the partition name is case-sensitive when it is enclosed in double quotation marks (**SQL_MODE** must include **ANSI_QUOTES**) or backquotes, but in MySQL the partition name is case-insensitive.
- The maximum length of a MySQL identifier is 64 characters, while that of a GaussDB identifier is 63 bytes. If the length of an identifier exceeds the limit, MySQL reports an error, while GaussDB truncates the identifier and generates an alarm.

- GaussDB does not support executable comments.

3.7.3 DDL

Table 3-31 DDL syntax compatibility

Description	Syntax	Difference
Primary keys/Indexes	ALTER TABLE, CREATE TABLE, CREATE INDEX, and DROP INDEX	• In GaussDB, when the table joined with the constraint is Ustore and USING BTREE is specified in the SQL statement, the underlying index is created as UB-tree. • The index name, constraint name, and key name must be unique within the same schema in GaussDB and within the same table in MySQL. • The maximum number of columns supported by the primary keys of MySQL is different from those of GaussDB. • The index name created after being specified by a primary key in GaussDB is the index name specified by a user. In MySQL, the index name is **PRIMARY**. • **ASC | DESC** specifies whether an index is in ascending or descending order. In MySQL 5.7, it is parsed but ignored, and the default behavior is **ASC**. In MySQL 8.0 and GaussDB, **ASC | DESC** is parsed and takes effect. • In GaussDB, the prefix length cannot exceed 2676. The actual length of the key value is restricted by the internal page. If a column contains multi-byte characters or an index has multiple keys, an error may be reported when the index line length exceeds the threshold.

Description	Syntax	Difference
		• In GaussDB, the primary key index does not support prefix keys. The prefix length cannot be specified when a primary key is created or added. • In GaussDB, the index options **algorithm_option** and **lock_option** in the CREATE/DROP INDEX statement are only supported in syntax. No error is reported during creation, but they do not take effect.

Description	Syntax	Difference
Support auto-increment columns.	ALTER TABLE and CREATE TABLE	• It is recommended that an auto-increment column in GaussDB be the first column of an index. Otherwise, an alarm is generated during table creation. The auto-increment column in MySQL must be the first column of the index. Otherwise, an error is reported during table creation. In GaussDB, an error occurs when some operations (such as ALTER TABLE EXCHANGE PARTITION) are performed on a table that contains auto-increment columns. • For the syntax AUTO_INCREMENT = value, **value** must be a positive number less than 2^127 in GaussDB, but **value** can be **0** in MySQL. • In GaussDB, an error occurs if the auto-increment continues after an auto-increment value reaches the maximum value of a column data type. In MySQL, errors or warnings may be generated during auto-increment, and sometimes auto-increment continues until the maximum value is reached. • GaussDB does not support the *innodb_autoinc_lock_mode* system variable, but when its GUC parameter **auto_increment_cache** is set to **0**, the behavior of inserting auto-increment columns in batches is similar to that when the

Description	Syntax	Difference
		MySQL system variable *innodb_autoinc_lock_mode* is set to **1**. • In GaussDB, when 0s, NULLs, and definite values are imported or batch inserted into auto-increment columns, the auto-increment values inserted after an error occurs in GaussDB may not be the same as those in MySQL. You can use the GUC parameter **auto_increment_cache** to control the number of reserved auto-increment values. • In GaussDB, when auto-increment is triggered by parallel import or insertion of auto-increment columns, the cache value reserved for each parallel thread is used only in the thread. If the cache value is not used up, the values of auto-increment columns in the table are discontinuous. The auto-increment value generated by parallel insertion cannot be guaranteed to be the same as that generated in MySQL. • In GaussDB, when auto-increment columns are batch inserted into a local temporary table, no auto-increment value is reserved. In normal scenarios, auto-increment values are not discontinuous. In MySQL, the auto-increment result of an auto-increment column in a temporary

Description	Syntax	Difference
		table is the same as that in an ordinary table. • The SERIAL data type of GaussDB is an original auto-increment column, which is different from the **AUTO_INCREMENT** column. The SERIAL data type of MySQL is the **AUTO_INCREMENT** column. • GaussDB does not allow the value of **auto_increment_offset** to be greater than that of **auto_increment_increment**. Otherwise, an error occurs. MySQL allows it and states that **auto_increment_offset** will be ignored. • If a table has a primary key or index, the sequence in which the **ALTER TABLE** command rewrites table data may be different from that in MySQL. GaussDB rewrites table data based on the table data storage sequence, while MySQL rewrites table data based on the primary key or index sequence. As a result, the auto-increment sequence may be different. • When the **ALTER TABLE** command in GaussDB is used to add or modify auto-increment columns, the number of auto-increment values reserved for the first time is the number of rows in the table statistics. The number of rows in the statistics may not be the same as that in MySQL.

Description	Syntax	Difference
		• The return value of the last_insert_id function in GaussDB is a 128-bit integer. • When GaussDB performs auto-increment in a trigger or user-defined function, the return value of last_insert_id is updated. MySQL does not update it. • If the values of the GUC parameters **auto_increment_offset** and **auto_increment_increment** in GaussDB are out of range, an error occurs. MySQL automatically changes the value to a boundary value. • If **sql_mode** is set to **no_auto_value_on_zero**, the auto-increment columns of the table are not subject to NOT NULL constraints. In GaussDB and MySQL, when the value of an auto-increment column is not specified, **NULL** will be inserted into the auto-increment column, but auto-increment is triggered for the former and not triggered for the latter.

Description	Syntax	Difference
Specify character sets and collation rules.	ALTER SCHEMA, ALTER TABLE, CREATE SCHEMA, and CREATE TABLE	• When you specify a database-level character set, except binary character sets, the character set of a new database or schema cannot be different from that specified by **server_encoding** of the database. • When you specify a table-level or column-level character set and collation, MySQL allows you to specify a character set and collation that are different from the database-level character set and collation. In GaussDB, the table-level and column-level character sets and collations support only the binary character sets and collations or can be the same as the database-level character sets and collations. • If the character set or collation is specified repeatedly, only the last one takes effect.

Description	Syntax	Difference
Basic table definition syntax	CREATE TABLE and ALTER TABLE	• GaussDB does not support the following options: **AVG_ROW_LENGTH**, **CHECKSUM**, **COMPRESSION**, **CONNECTION**, **DATA DIRECTORY**, **INDEX DIRECTORY**, **DELAY_KEY_WRITE**, **ENCRYPTION**, **INSERT_METHOD**, **KEY_BLOCK_SIZE**, **MAX_ROWS**, **MIN_ROWS**, **PACK_KEYS**, **PASSWORD**, **STATS_AUTO_RECALC**, **STATS_PERSISTENT**, and **STATS_SAMPLE_PAGES**. • The following options do not report errors in GaussDB and do not take effect: **ENGINE** and **ROW_FORMAT**. • In GaussDB, ALTER TABLE does not support DROP INDEX, DROP KEY, and ORDER BY. • When ALTER TABLE is used to add a column, if the specified column in MySQL is **NOT NULL**, the **NULL** value is converted to the default value of the corresponding type and inserted into the column. GaussDB checks the **NULL** value. • In GaussDB, the *tablename* of **ALTER TABLE** *tablename***;** cannot be empty. • In MySQL loose mode, NULL is converted and data is successfully inserted. In MySQL strict mode, NULL values cannot be inserted. GaussDB does not support this feature. NULL values

Description	Syntax	Difference
		cannot be inserted in loose or strict mode. • CREATE TABLE that contains the CHECK constraint takes effect in MySQL 8.0. MySQL 5.7 parses the syntax but the syntax does not take effect. GaussDB synchronizes this function of MySQL 8.0, and the GaussDB CHECK constraint can reference other columns, but MySQL cannot. • A maximum of 32767 CHECK constraints can be added to a table in GaussDB.

Description	Syntax	Difference
Create or alter a partitioned table.	CREATE TABLE PARTITION, CREATE TABLE SUBPARTITION, and ALTER TABLE	• MySQL supports expressions but does not support multiple partition keys in the following scenarios: – The LIST/RANGE partitioning policy is used and the COLUMNS keyword is not specified. – The hash partitioning policy is used. • MySQL does not support expressions but supports multiple partition keys in the following scenarios: – The LIST/RANGE partitioning policy is used and the COLUMNS keyword is specified. – The KEY partitioning policy is used. • In GaussDB, expressions cannot be used as partition keys. • GaussDB supports multiple partition keys only when the LIST or RANGE partitioning policy is used and subpartitions are not specified. • In GaussDB partitioned tables, virtual generated columns cannot be used as partition keys. • In GaussDB, **column_list** of a partition key cannot be empty. • GaussDB: Partitioned tables do not support LINEAR/KEY hash. • In GaussDB, the hash functions used by hash partitioned tables and level-2 partitioned tables in the CREATE TABLE

Description	Syntax	Difference
		statement are different from those used in MySQL. Therefore, the storage of hash partitioned tables and level-2 partitioned tables is different from that in MySQL. • MySQL allows you to modify the partition key information of a partitioned table, but GaussDB does not. • GaussDB: If the table is partitioned by key in the CREATE/ALTER TABLE statement, algorithms cannot be specified. The syntaxes that do not support expressions as input parameters are as follows: – PARTITION BY HASH() – PARTITION BY KEY() – VALUES LESS THAN()

Description	Syntax	Difference
Exchange the partition data of an ordinary table and a partitioned table.	ALTER TABLE PARTITION	Differences in ALTER TABLE EXCHANGE PARTITION: ● After **ALTER TABLE EXCHANGE PARTITION** is executed, the auto-increment columns are reset in MySQL, but in GaussDB, they are not reset and continue the auto-increment based on their old values. ● If MySQL tables or partitions use tablespaces, data in partitions and ordinary tables cannot be exchanged. If GaussDB tables or partitions use different tablespaces, data in partitions and ordinary tables can still be exchanged. ● MySQL does not verify the default values of columns. Therefore, data in partitions and ordinary tables can be exchanged even if the default values are different. GaussDB verifies the default values. If they are different, data in partitions and ordinary tables cannot be exchanged. ● After the DROP COLUMN operation is performed on a partitioned table or an ordinary table in MySQL, if the table structure is still consistent, data can be exchanged between partitions and ordinary tables. In GaussDB, data can be exchanged between partitions and ordinary tables only when the deleted columns of ordinary tables and partitioned tables are strictly aligned.

Description	Syntax	Difference
		• MySQL and GaussDB use different hash algorithms. Therefore, data stored in the same hash partition may be inconsistent. As a result, the exchanged data may also be inconsistent. • MySQL partitioned tables do not support foreign keys. If an ordinary table contains foreign keys or other tables reference foreign keys of an ordinary table, data in partitions and ordinary tables cannot be exchanged. GaussDB partitioned tables support foreign keys. If the foreign key constraints of two tables are the same, data in partitions and ordinary tables can be exchanged. If a GaussDB partitioned table does not contain foreign keys, an ordinary table is referenced by other tables, and the partitioned table is the same as the ordinary table, data in the partitioned table can be exchanged with that in the ordinary table.

Description	Syntax	Difference
CREATE TABLE... LIKE syntax	CREATE TABLE ... LIKE	• In versions earlier than MySQL 8.0.16, CHECK constraints are parsed but their functions are ignored. In this case, CHECK constraints are not replicated. GaussDB supports replication of CHECK constraints. • When a table is created, all primary key constraint names in MySQL are fixed to **PRIMARY KEY**. GaussDB does not support replication of primary key constraint names. • When a table is created, MySQL supports replication of unique key constraint names, but GaussDB does not. • When a table is created, MySQL versions earlier than 8.0.16 do not have CHECK constraint information, but GaussDB supports replication of CHECK constraint names. • When a table is created, MySQL supports replication of index names, but GaussDB does not. • When a table is created across sql_mode, MySQL is controlled by the loose mode and strict mode. The strict mode may become invalid in GaussDB. For example, if the source table has the default value **"0000-00-00"**, GaussDB can create a table that contains the default value **"0000-00-00"** in "no_zero_date" strict mode, which means that the strict mode is invalid.

Description	Syntax	Difference
		MySQL fails to create the table because it is controlled by the strict mode.
Truncate a partition.	ALTER TABLE [IF EXISTS] table_name truncate_clause;	For truncate_clause, the supported subitems are different: • GaussDB: TRUNCATE PARTITION { { ALL | partition_name [, ...] } | FOR (partition_value [, ...]) } [UPDATE GLOBAL INDEX] • MySQL: TRUNCATE PARTITION {partition_names | ALL}
Delete dependent objects.	DROP drop_type name CASCADE;	In GaussDB, CASCADE needs to be added to delete dependent objects. In MySQL, CASCADE is not required.

Description	Syntax	Difference
Partitioned table index	CREATE INDEX	• GaussDB partitioned table indexes are classified into local and global indexes. A local index is bound to a specific partition, and a global index corresponds to the entire partitioned table. • For details about how to create local and global indexes and the default rules, see "SQL Reference > SQL Syntax > C > CREATE INDEX " in *M Compatibility Developer Guide*. For example, if a unique index is created on a non-partition key, a global index is created by default. • MySQL does not have global indexes. In GaussDB, if the partitioned table index is a global index, the global index is not updated by default when operations such as DROP, TRUNCATE, and EXCHANGE are performed on table partitions. As a result, the global index becomes invalid and cannot be selected in subsequent statements. To avoid this problem, you are advised to explicitly specify the UPDATE GLOBAL INDEX clause at the end of the partition syntax or set the global GUC parameter **enable_gpi_auto_update** to **true** (recommended) so that global indexes can be automatically updated during partition operations. • In MySQL, both unique and ordinary indexes can

Description	Syntax	Difference
		be created on the same column simultaneously. In GaussDB, partitioned LOCAL and GLOBAL indexes can be simultaneously created on index columns with different sequences; however, the operation is not supported when the sequences are the same.

Description	Syntax	Difference
Add foreign key constraints and modify referencing columns and referenced columns of the foreign key constraints.	CREATE TABLE and ALTER TABLE	• GaussDB foreign key constraints are insensitive to types. If the data types of the fields in the main and child tables are implicitly converted, foreign keys can be created. MySQL are sensitive to foreign key types. If the column types of the two tables are different, foreign keys cannot be created. • MySQL does not allow you to modify the data type or name of a table column where the foreign key of the column is located by running **MODIFY COLUMN** or **CHANGE COLUMN**, but GaussDB supports such operation. • GaussDB: Foreign keys can be used as partition keys. • In GaussDB, the **MATCH FULL** and **MATCH SIMPLE** options can be specified when you are creating a foreign key. However, if you specify the **MATCH PARTIAL** option, an error is reported. In MySQL, the preceding options can be specified, but will not be effective. Their behavior ends up being the same as that of **MATCH SIMPLE**. • In GaussDB, the **ON [UPDATE | DELETE] SET DEFAULT** option can be specified when you are creating a foreign key. In MySQL, if you specify the **ON [UPDATE | DELETE] SET DEFAULT** option when creating a foreign key, an error is reported.

Description	Syntax	Difference
		• When creating a foreign key in GaussDB, you must create a unique index on the referenced column of the referenced table. When creating a foreign key in MySQL, you need to create an index on the referenced column of the referenced table. The index can be not unique. • When creating a foreign key in GaussDB, you do not need to create an index on the referencing column of the referencing table. When creating a foreign key in MySQL, you need to create an index on the referencing column of the referencing table. Otherwise, a corresponding index is automatically added. If the foreign key is deleted, this index is not deleted. • In GaussDB, referencing tables and referenced tables can be temporary tables. Foreign keys cannot be created between temporary tables and non-temporary tables. In MySQL, temporary tables cannot be used as referencing tables or referenced tables. When a foreign key is created to specify a referenced table, MySQL does not match the temporary table created in the current session. • When you are creating a foreign key in GaussDB, it is optional to specify the referenced field name of the referenced table. In this case, the primary key

Description	Syntax	Difference
		in the referenced table is used as the referenced field of the foreign key. In MySQL, the referenced field of the referenced table must be specified. • In GaussDB, the data type of a referencing field or referenced field can be modified regardless of whether **foreign_key_checks** is disabled. In MySQL, you can change the data type of a referencing field or referenced field only when **foreign_key_checks** is set to **off**. • In GaussDB, you can delete referencing fields from a referencing table. In this case, related foreign key constraints are deleted cascadingly. Attempts to delete referencing field in a referencing table will fail in MySQL. • In GaussDB, if **foreign_key_checks** is set to **on** and a referenced table and a referencing table are in different schemas, when the schema that contains the referenced table is deleted, foreign key constraints on the referencing table are deleted cascadingly. In MySQL, if **foreign_key_checks** is set to **on**, the deletion fails.
Encrypt the CMKs of CEKs in round robin (RR) mode and encrypt the plaintext of CEKs.	ALTER COLUMN ENCRYPTION KEY	The M-compatible mode does not support the full encryption. Therefore, this syntax is not supported.

Description	Syntax	Difference
The encrypted equality query feature adopts a multi-level encryption model. The master key encrypts the column key, and the column key encrypts data. This syntax is used to create a master key object.	CREATE CLIENT MASTER KEY	The M-compatible mode does not support the full encryption. Therefore, this syntax is not supported.
Create a CEK that can be used to encrypt a specified column in a table.	CREATE COLUMN ENCRYPTION KEY	The M-compatible mode does not support the full encryption. Therefore, this syntax is not supported.
Send keys to the server for caching. This function is used only when the memory decryption emergency channel is enabled. This is a fully-encrypted function.	\send_token	The M-compatible mode does not support the full encryption. Therefore, this syntax is not supported.
Send keys to the server for caching. This function is used only when the memory decryption emergency channel is enabled. This is a fully-encrypted function.	\st	The M-compatible mode does not support the full encryption. Therefore, this syntax is not supported.
Destroy the keys cached on the server. This function is used only when the memory decryption emergency channel is enabled. This is a fully-encrypted function.	\clear_token	The M-compatible mode does not support the full encryption. Therefore, this syntax is not supported.

Description	Syntax	Difference
Destroy the keys cached on the server. This function is used only when the memory decryption emergency channel is enabled. This is a fully-encrypted function.	\ct	The M-compatible mode does not support the full encryption. Therefore, this syntax is not supported.
Set the parameters for accessing the external key manager in the fully-encrypted database features.	\key_info KEY_INFO	The M-compatible mode does not support the full encryption. Therefore, this syntax is not supported.
Enable third-party dynamic libraries and set related parameters. This is a fully-encrypted function.	\crypto_module_info MODULE_INFO	The M-compatible mode does not support the full encryption. Therefore, this syntax is not supported.
Enable third-party dynamic libraries and set related parameters. This is a fully-encrypted function.	\cmi MODULE_INFO	The M-compatible mode does not support the full encryption. Therefore, this syntax is not supported.

Description	Syntax	Difference
Support syntaxes that change table names.	ALTER TABLE tbl_name RENAME [TO | AS | =] new_tbl_name; Or RENAME {TABLE | TABLES} tbl_name TO new_tbl_name [, tbl_name2 TO new_tbl_name2, ...];	• The ALTER RENAME syntax in GaussDB supports only the function of changing the table name and cannot be coupled with other function operations. • In GaussDB, only the old table name column supports the schema.table_name format, and the new and old table names belong to the same schema. • GaussDB does not support renaming of old and new tables across schemas. However, if you have the permission, you can modify the names of tables in other schemas in the current schema. • The syntax for renaming multiple groups of tables in GaussDB supports renaming of all local temporary tables, but does not support the combination of local temporary tables and non-local temporary tables.

Description	Syntax	Difference
Support the CREATE VIEW AS SELECT syntax.	CREATE VIEW table_name AS query;	• When the precision transfer function is disabled (**m_format_behavior_compat_options** is not set to **enable_precision_decimal**), the "query" in the CREATE VIEW view_name AS query syntax cannot contain calculation operations (such as function calling and calculation using operators) for the following types. Only direct column calling is supported (such as SELECT col1 FROM table1). It can be used when the precision transfer function is enabled (**m_format_behavior_compat_options** is set to **enable_precision_decimal**). BINARY[(n)], VARBINARY(n), CHAR[(n)], VARCHAR(n), TIME[(p)], DATETIME[(p)], TIMESTAMP[(p)], BIT[(n)], NUMERIC[(p[,s])], DECIMAL[(p[,s])], DEC[(p[,s])], FIXED[(p[,s])], FLOAT4[(p, s)], FLOAT8[(p,s)], FLOAT[(p)], REAL[(p, s)], FLOAT[(p, s)], DOUBLE[(p,s)], DOUBLE PRECISION[(p,s)], TEXT, TINYTEXT, MEDIUMTEXT, LONGTEXT, BLOB, TINYBLOB, MEDIUMBLOB, and LONGBLOB • In simple queries, an error message is displayed for the preceding calculation

Description	Syntax	Difference
		operations in GaussDB. For example: m_db=# CREATE TABLE TEST (salary int(10)); CREATE TABLE m_db=# INSERT INTO TEST VALUES(8000); INSERT 0 1 m_db=# CREATE VIEW view1 AS SELECT salary/10 as te FROM TEST; ERROR: Unsupported type numeric used with expression in CREATE VIEW statement. m_db=# CREATE VIEW view2 AS SELECT sec_to_time(salary) as te FROM TEST; ERROR: Unsupported type time used with expression in CREATE VIEW statement. • In non-simple queries such as composite queries and subqueries, the calculation operations of the preceding types in GaussDB are different from those in MySQL. GaussDB does not retain the data type column precision attribute of a created table. • CREATE VIEW AS SELECT. When a UNION is nested with a subquery, MySQL creates a temporary table for the subquery. If the return type of a temporary table is tinytext, text, mediumtext, or longtext, MySQL performs calculation based on the default maximum byte length of the type. However, GaussDB performs calculation based on the actual byte length of the created temporary table. Therefore, the text type of the GaussDB aggregation result may be smaller

Description	Syntax	Difference
		than that of the MySQL aggregation result. For example, longtext is returned for MySQL, and mediumtext is returned for GaussDB. For example: Behavior in MySQL 5.7: mysql> CREATE TABLE IF NOT EXISTS tb_1 (id int,col_text2 text); Query OK, 0 rows affected (0.02 sec) mysql> CREATE TABLE IF NOT EXISTS tb_2 (id int,col_text2 text); Query OK, 0 rows affected (0.02 sec) mysql> CREATE VIEW v1 AS SELECT * FROM (SELECT cast(col_text2 AS char) c37 FROM tb_1) t1 -> UNION ALL SELECT * FROM (SELECT cast(col_text2 as char) c37 FROM tb_2) t2; Query OK, 0 rows affected (0.00 sec) mysql> DESC v1; +-------+----------+------+-----+---------+-------+ | Field	Type	Null	Key	Default	Extra | +-------+----------+------+-----+---------+-------+ | c37	longtext	YES		NULL	| +-------+----------+------+-----+---------+-------+ 1 row in set (0.00 sec) Behavior in GaussDB: mysql_regression=# CREATE TABLE IF NOT EXISTS tb_1 (id int,col_text2 text); CREATE TABLE mysql_regression=# CREATE TABLE IF NOT EXISTS tb_2 (id int,col_text2 text); CREATE TABLE mysql_regression=# CREATE VIEW v1 AS SELECT * FROM (SELECT cast(col_text2 AS char) c37 FROM tb_1) t1 mysql_regression-# UNION ALL SELECT * FROM (SELECT cast(col_text2 AS char) c37 FROM tb_2) t2; CREATE VIEW mysql_regression=# DESC v1; Field	Type	Null	Key	Default	Extra -------+------------+------+-----+---------+------- c37	mediumtext	YES

Description	Syntax	Difference
		<pre>| | (1 row)</pre> • When the bitstring constant is used to create a view, the constant is converted into hexstring for creation in MySQL, whereas the bitstring constant is used directly to create a view in GaussDB. The bitstring constant is an unsigned value. Therefore, the attribute of the view created in GaussDB is unsigned. – Behavior in MySQL 5.7: <pre>mysql> SELECT version(); +------------------+ | version() | +------------------+ | 5.7.44-debug-log | +------------------+ 1 row in set (0.00 sec) mysql> DROP VIEW IF EXISTS v1; Query OK, 0 rows affected, 1 warning (0.00 sec) mysql> CREATE VIEW v1 AS SELECT b'101'/b'101' AS c22; Query OK, 0 rows affected (0.01 sec) mysql> DESC v1; +-------+----------------------+------+-----+---------+-------+ | Field | Type | Null | Key | Default | Extra | +-------+----------------------+------+-----+---------+-------+ | c22 | decimal(5,4) unsigned | YES | | NULL | | +-------+----------------------+------+-----+---------+-------+ 1 row in set (0.00 sec) mysql> SHOW CREATE VIEW v1; +------ +-- +---------------------- +----------------------+ | View | Create View | character_set_client | collation_connection | +------ +----------------------------------</pre>

Description	Syntax	Difference
		`` ---------- ---------- +---------- +----------+ | v1	CREATE ALGORITHM=UNDEFINED DEFINER=`omm`@`%` SQL SECURITY DEFINER VIEW `v1` AS SELECT (0x05 / 0x05) AS `c22` | utf8mb4 | utf8mb4_general_ci | +------ +---------- ---------- ---------- +---------- +----------+ 1 row in set (0.00 sec) ``` – Behavior in GaussDB: ``` m_db=# DROP VIEW IF EXISTS v1; DROP VIEW m_db=# CREATE VIEW v1 AS SELECT b'101'/b'101' AS c22; CREATE VIEW m_db=# DESC v1; Field	Type	Null	Key | Default	Extra -------+--------------+------+----- +---------+------- c22	decimal(5,4)	YES
	 (1 row) ```

Description	Syntax	Difference
View dependency differences	CREATE VIEW and ALTER TABLE	In MySQL, view storage records only the table name, column name, and database name of the target table, but does not record the unique identifier of the target table. GaussDB parses the SQL statement used for creating a view and stores the unique identifier of the target table. Therefore, the differences are as follows: • In MySQL, you can modify the data type of a column on which a view depends because the view is unaware of the modification of the target table. In GaussDB, such modification is forbidden and the attempt will fail. • In MySQL, you can rename a column on which a view depends because the view is unaware of the modification of the target table, but the view cannot be queried after the operation. In GaussDB, each column precisely stores the unique identifier of the corresponding table and column. Therefore, the column name in the table can be modified successfully without changing the column name in the view. In addition, the view can be queried after the operation.

Description	Syntax	Difference
Modifying a view definition	CREATE OR REPLACE VIEW and ALTER VIEW	• In MySQL, you can modify any attribute of a view. In GaussDB, names and types of columns in non-updatable views cannot be modified and the columns cannot be deleted, but these operations are allowed in updatable views. • In MySQL, after a column in the underlying view of a nested view is modified, the upper-level views can be used as long as the column name exists. In GaussDB, after the name or type of a column in the underlying view of a nested view is modified or the column is deleted, the upper-layer views are unavailable.
ANALYZE partition syntax	ALTER TABLE tbl_name ANALYZE PARTITION {partition_names | ALL}	• In GaussDB, this syntax supports only partition statistics collection. • In MySQL, **partition_names** is case-insensitive. In GaussDB, **partition_names** with backquotes are case-insensitive, but the one without backquotes are case-sensitive. • In GaussDB, ALTER TABLE is displayed if the execution is successful. The execution error is reported based on the existing error code. In MySQL, the execution result is displayed in a table.

Description	Syntax	Difference
Supports the syntax of virtual generated columns.	[GENERATED ALWAYS] AS (generation_expr) [STORED | VIRTUAL]	• Indexes can be created for virtual generated columns in MySQL, but cannot in GaussDB. • Virtual generated columns can be used as partition keys in MySQL, but cannot in GaussDB. • The CHECK constraint of generated columns in GaussDB is compatible with that in MySQL 8.0. Therefore, the CHECK constraint is effective. • In MySQL, ALTER TABLE can be used to modify the stored generated columns that are considered as partition keys. GaussDB does not support this operation. • In MySQL, when data in generated columns of an updatable view is updated, the DEFAULT keyword can be specified. In GaussDB, this operation is not supported. • IGNORE feature is supported by virtual generated columns in MySQL, but not in GaussDB. • Querying a virtual generated column in GaussDB is equivalent to querying the expression of the virtual generated column. (If the data type, character set, or collation defined in the expression is inconsistent with that defined in the column, the expression type is implicitly converted to the type defined in the column) This behavior is to query virtual generated

Description	Syntax	Difference
		columns that are used for creating tables or views or other behaviors. As a result, the data type of such columns may be different from those in MySQL. For example, when CREATE TABLE AS is used to create a table, if the virtual generated column in the source table is defined as the FLOAT type, the data type of the corresponding column in the target table may be DOUBLE, which is different from that in MySQL. • In GaussDB, the GENERATED ALWAYS AS statement cannot reference columns generated by GENERATED ALWAYS AS, but it can in MySQL.

Description	Syntax	Difference
Create a table and insert data into the table using CREATE TABLE SELECT.	CREATE TABLE [AS] SELECT	• Partitioned tables cannot be created. • REPLACE/IGNORE is not supported. • If the SELECT column is not a direct table column, **NULL** is allowed by default, and there is no default value. For example, the column **a** of the new table created by **CREATE TABLE t1 SELECT unix_timestamp('2008-01-02 09:08:07.3465') AS a;** can be **NULL** and does not have the default value. • To use all functions, you need to set the GUC parameter **m_format_behavior_compat_options** to **enable_precision_decimal**. Otherwise, a behavior error will be reported for types related to data type precision due to version compatibility issues. For example, an error is reported in the UNION scenario or when a SELECT column contains a non-direct table column (such as expressions, functions, and constants). • When CREATE TABLE AS SELECT is used to create a table, the maximum length of a column name in the table is 63 characters. If the length exceeds 63 characters, the excess part will be truncated. If the length exceeds 64 characters (the maximum) in MySQL, an error is reported.

Description	Syntax	Difference
The maximum length of the UTF-8 character set code is different. As a result, the column length of a created table or view is different.	CREATE TABLE [AS] SELECT; CREATE VIEW [AS] SELECT	If MySQL uses the utf8 character set, whose code allows a maximum of 3 bytes, and GaussDB uses utf8 (utf8mb4), whose code allows a maximum of 4 bytes, when the GUC parameter **m_format_behavior_compat_options** is set to **'enable_precision_decimal'**, CREATE TABLE AS (CTAS) and CREATE VIEW AS (CVAS) may create different text types (including binary text). The returned character length in the CTAS and CVAS scenarios depends on the maximum length of the character set. For example, if the maximum length of the character set returned by a node is 1024 bytes for both GaussDB and MySQL, 341 (1024/3) characters are returned for MySQL and 256 (1024/4) characters are returned for GaussDB. For example: Behavior in MySQL 5.7: mysql> CREATE TABLE t1 AS SELECT (CASE WHEN true THEN min(521.2312) ELSE group_concat(115.0414) END) res1; Query OK, 1 row affected (0.06 sec) Records: 1 Duplicates: 0 Warnings: 0 mysql> DESC t1; +-------+--------------+------+-----+---------+-------+ | Field | Type | Null | Key | Default | Extra | +-------+--------------+------+-----+---------+-------+ | res1 | varchar(341) | YES | | NULL | | +-------+--------------+------+-----+---------+-------+ 1 row in set (0.01 sec) Behavior in GaussDB: mysql_regression=# CREATE TABLE t1 AS SELECT (CASE WHEN true THEN min(521.2312) ELSE group_concat(115.0414) END) res1; INSERT 0 1 mysql_regression=# DESC t1;

Description	Syntax	Difference
		Field | Type | Null | Key | Default | Extra -------+--------------+------+-----+---------+------- res1 | varchar(256) | YES | | | (1 row)

Description	Syntax	Difference
Setting default values of columns	CREATE TABLE and ALTER TABLE	• For MySQL 5.7, only the default value without parentheses is supported. MySQL 8.0 and GaussDB support default values in parentheses. -- GaussDB m_db=# DROP TABLE IF EXISTS t1, t2; DROP TABLE m_db=# CREATE TABLE t1(a DATETIME DEFAULT NOW()); CREATE TABLE m_db=# CREATE TABLE t2(a DATETIME DEFAULT (NOW())); CREATE TABLE -- MySQL 5.7 mysql> DROP TABLE IF EXISTS t1, t2; Query OK, 0 rows affected (0.04 sec) mysql> CREATE TABLE t1(a DATETIME DEFAULT NOW()); Query OK, 0 rows affected (0.04 sec) mysql> CREATE TABLE t2(a DATETIME DEFAULT (NOW())); ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '(NOW()))' at line 1 -- MySQL 8.0 mysql> DROP TABLE IF EXISTS t1, t2; Query OK, 0 rows affected (0.17 sec) mysql> CREATE TABLE t1(a DATETIME DEFAULT NOW()); Query OK, 0 rows affected (0.19 sec) mysql> CREATE TABLE t2(a DATETIME DEFAULT (NOW())); Query OK, 0 rows affected (0.20 sec) • In MySQL, when specifying default values for BLOB, TEXT, and JSON data types, you must add parentheses to the default values. In GaussDB, you do not need to add parentheses when specifying default values

Description	Syntax	Difference
		for the preceding data types. • GaussDB does not check whether a specified default value overflows except a value of the varbianry type. MySQL checks whether a default value specified without parentheses overflows, but does not check whether the one specified with parentheses overflows. • In GaussDB, time constants starting with DATE, TIME, or TIMESTAMP can be used to specify default values for columns. In MySQL, when time constants starting with DATE, TIME, or TIMESTAMP are used to specify default values for columns, parentheses must be added to the default values. -- GaussDB m_db=# DROP TABLE IF EXISTS t1, t2; DROP TABLE m_db=# CREATE TABLE t1(a TIMESTAMP DEFAULT TIMESTAMP '2000-01-01 00:00:00'); CREATE TABLE m_db=# CREATE TABLE t2(a TIMESTAMP DEFAULT (TIMESTAMP '2000-01-01 00:00:00')); CREATE TABLE -- MySQL 5.7 mysql> DROP TABLE IF EXISTS t1, t2; Query OK, 0 rows affected (0.02 sec) mysql> CREATE TABLE t1(a TIMESTAMP DEFAULT TIMESTAMP '2000-01-01 00:00:00'); ERROR 1067 (42000): Invalid default value for 'a' mysql> CREATE TABLE t2(a TIMESTAMP DEFAULT (TIMESTAMP '2000-01-01 00:00:00')); ERROR 1064 (42000): You have an error in your SQL syntax; check

Description	Syntax	Difference
		the manual that corresponds to your MySQL server version for the right syntax to use near '(TIMESTAMP '2000-01-01 00:00:00'))' at line 1 -- MySQL 8.0 mysql> DROP TABLE IF EXISTS t1, t2; Query OK, 0 rows affected (0.14 sec) mysql> CREATE TABLE t1(a TIMESTAMP DEFAULT TIMESTAMP '2000-01-01 00:00:00'); ERROR 1067 (42000): Invalid default value for 'a' mysql> CREATE TABLE t2(a TIMESTAMP DEFAULT (TIMESTAMP '2000-01-01 00:00:00')); Query OK, 0 rows affected (0.19 sec)

3.7.4 DML

Table 3-32 DML syntax compatibility

Description	Syntax	Difference
DELETE supports deleting data from multiple tables.	DELETE	• During multi-table deletion, if a tuple to be deleted is concurrently modified by other sessions, the latest values of all tuples in the session are used for matching again. If the conditions are still met, the tuple is deleted. During this process, MySQL deletes all target tables in the same way. However, GaussDB only rematches tuples in the target tables that involve concurrent updates, which may cause data inconsistency. • The verification rules of target tables and range tables in the multi-table operation syntax are different from those in MySQL. After the GUC compatibility parameter **m_format_dev_version** is set to **'s2'**, the verification rules become consistent with MySQL.
UPDATE supports updating data from multiple tables.	UPDATE	During multi-table update, if a tuple to be updated is concurrently modified by other sessions, the latest values of all tuples in the session are used for matching again. If the conditions are still met, the tuple is updated. During this process, MySQL updates all target tables consistently. However, GaussDB only rematches tuples of target tables that involve concurrent updates, which may cause data inconsistency.

Description	Syntax	Difference
SELECT INTO syntax	SELECT	• In GaussDB, you can use SELECT INTO to create a table based on the query result. MySQL does not support this function. • In GaussDB, the SELECT INTO syntax does not support the query result that is obtained after the set operation of multiple queries is performed.

Description	Syntax	Difference
REPLACE INTO syntax	REPLACE	Difference between the initial values of the time type. For example: • MySQL is not affected by the strict or loose mode. You can insert time 0 into a table. mysql> CREATE TABLE test(f1 TIMESTAMP NOT NULL, f2 DATETIME NOT NULL, f3 DATE NOT NULL); Query OK, 1 row affected (0.00 sec) mysql> REPLACE INTO test VALUES(f1, f2, f3); Query OK, 1 row affected (0.00 sec) mysql> SELECT * FROM test; +--------------------- +---------------------+------------+ | f1 | f2 | f3 | +--------------------- +---------------------+------------+ | 0000-00-00 00:00:00 | 0000-00-00 00:00:00 | 0000-00-00 | +--------------------- +---------------------+------------+ 1 row in set (0.00 sec) • The time 0 can be successfully inserted only when GaussDB is in loose mode. gaussdb=# SET sql_mode = ''; SET gaussdb=# CREATE TABLE test(f1 TIMESTAMP NOT NULL, f2 DATETIME NOT NULL, f3 DATE NOT NULL); CREATE TABLE gaussdb=# REPLACE INTO test VALUES(f1, f2, f3); REPLACE 0 1 gaussdb=# SELECT * FROM test; f1 | f2 | f3 ---------------------+---------------------+------------ 0000-00-00 00:00:00 | 0000-00-00 00:00:00 | 0000-00-00 (1 row) In strict mode, the following error is reported: The date, time, datetime, timestamp, or year is incorrect.

Description	Syntax	Difference
Import data by using LOAD DATA.	LOAD DATA	When LOAD DATA is used to import data, GaussDB differs from MySQL in the following aspects: • The execution result of the LOAD DATA syntax is the same as that in MySQL strict mode. The loose mode is not adapted currently. • The **IGNORE** and **LOCAL** parameters are used only to ignore the conflicting rows when the imported data conflicts with the data in the table and to automatically fill default values for other columns when the number of columns in the file is less than that in the table. Other functions are not supported currently. • The **[(col_name_or_user_var [, col_name_or_user_var]...)]** parameter cannot be used to specify a column repeatedly. • The newline character specified by **[FIELDS TERMINATED BY 'string']** cannot be the same as the separator specified by **[LINES TERMINATED BY'string']**. • If the data written to a table by running **LOAD DATA** cannot be converted to the data type of the table, an error is reported. • The LOAD DATA SET expression does not support the calculation of a specified column name. • LOAD DATA applies only to tables but not views. • The default newline character of the file in Windows is different from that in Linux. LOAD DATA

Description	Syntax	Difference
		cannot identify this scenario and reports an error. You are advised to check the newline character at the end of lines in the file to be imported. • In GaussDB, when the GUC parameter **m_format_behavior_compat_options** is not set, data can be imported only from the server using LOAD DATA, regardless of whether the **LOCAL** parameter is specified. In MySQL, if the **LOCAL** parameter is specified, data can be imported from the client; otherwise, it is imported from the server. After you specify the value of this GUC parameter that includes **enable_load_data_remote_transmission** in GaussDB, the **LOCAL** parameter behavior of LOAD DATA becomes consistent with that in MySQL.
LIMIT clause differences	DELETE, SELECT, and UPDATE	The LIMIT clauses of each statement in GaussDB are different from those in MySQL. The maximum parameter value of LIMIT (of the BIG INT type) in GaussDB is **9223372036854775807**. If the actual value exceeds the number, an error is reported. In MySQL, the maximum value of LIMIT (of the unsigned LONGLONG type) is **18446744073709551615**. If the actual value exceeds the number, an error is reported. You can set a small value in LIMIT, which is rounded off during execution. The value cannot be a decimal in MySQL.

Description	Syntax	Difference
Difference in using backslashes (\)	INSERT	The usage of backslashes (\) can be determined by parameters in GaussDB and MySQL, but their default usages are different. In MySQL, the **NO_BACKSLASH_ESCAPES** parameter is used to determine whether backslashes (\) in character strings and identifiers are parsed as common characters or escape characters. By default, backslashes (\) are parsed as escape characters in character strings and identifiers. If **SET sql_mode='NO_BACKSLASH_ESCAPES';** is used, the backslashes (\) cannot be parsed as escape characters in strings and identifiers. In GaussDB, the **standard_conforming_strings** parameter is used to determine whether backslashes (\) in character strings and identifiers are parsed as common characters or escape characters. The default value is **on**, indicating that backslashes (\) are parsed as common text in common character string texts according to the SQL standard. If **SET standard_conforming_strings=off;** is used, backslashes (\) can be parsed as escape characters in character strings and identifiers.

Description	Syntax	Difference
If the inserted value is less than the number of columns, MySQL reports an error while GaussDB supplements null values.	INSERT	In GaussDB, if the column list is not specified and the inserted value is less than the number of columns, values are assigned based on the column sequence when the table is created by default. If a column has a NOT NULL constraint, an error is reported. If no NOT NULL constraint exists and a default value is specified, the default value is added to the column. If no default value is specified, **null** is added.
The columns sorted in ORDER BY must be included in the columns of the result set.	SELECT	In GaussDB, when used with the GROUP BY clause, the columns to be sorted in ORDER BY must be included in the columns of the result set retrieved by the SELECT statement. When used with the DISTINCT keyword, the columns to be sorted in ORDER BY must be included in the columns of the result set retrieved by the SELECT statement.
If the foreign key data type is timestamp or datetime, an error is reported for attempts to perform UPDATE or DELETE on a foreign table.	UPDATE/DELETE	If the foreign key data type is timestamp or datetime, an error is reported for attempts to perform UPDATE or DELETE on a foreign table, but such operations are allowed in MySQL.

Description	Syntax	Difference
NATURAL JOIN syntax	SELECT	• In GaussDB, NATURAL [[LEFT | RIGHT] OUTER] JOIN allows you not to specify **LEFT | RIGHT**. If **LEFT | RIGHT** is not specified, **NATURAL OUTER JOIN** is **NATURAL JOIN**. You can use JOIN consecutively. • In GaussDB, join sequence is strictly from left to right. MySQL may adjust the sequence. • In GaussDB and MySQL, columns involving **join** in the left or right table cannot be ambiguous during natural join or using. (Generally, ambiguity is caused by duplicate names of columns in the left or right temporary table.) The join sequence differs in two databases, which may lead to different behaviors. – Behavior in GaussDB: m_regression=# CREATE TABLE t1(a int,b int); CREATE TABLE m_regression=# CREATE TABLE t2(a int,b int); CREATE TABLE m_regression=# CREATE TABLE t3(a int,b int); CREATE TABLE m_regression=# SELECT * FROM t1 JOIN t2; a | b | a | b ---+---+---+--- (0 rows) m_regression=# SELECT * FROM t1 JOIN t2 natural join t3; -- Failed. Duplicate contents exist in columns **a** and **b** of the temporary table obtained by **t1 join t2**. Therefore, there is ambiguity in nature join. ERROR: common column name "a" appears more than once in left table – Behavior in MySQL: mysql> SELECT * FROM t1 JOIN t2 NATURAL JOIN t3; Empty set (0.00 sec) mysql> SELECT * FROM (t1 join t2) NATURAL JOIN t3; ERROR 1052 (23000): Column 'a' in from clause is ambiguous

Description	Syntax	Difference
JOIN syntax	SELECT	Commas (,) cannot be used as a way of JOIN in GaussDB, but can be used in MySQL. GaussDB does not support USE INDEX FOR JOIN. The execution plans generated in the multi-table join STRAIGHT_JOIN in GaussDB may be different from those in MySQL.

Description	Syntax	Difference
Display column names by using SELECT.	SELECT	• To ensure that the column names displayed by using the SELECT statement in GaussDB are the same as those in MySQL, you need to enable the parameter to display the column name output. `SET m_format_behavior_compat_options = 'select_column_name'` • If this configuration item is not set in GaussDB: – **SELECT** *System function*: The output is the system function name. – **SELECT** *Expression*: The output is **?***column***?**. – **SELECT** *Boolean value*: The output is a Boolean value. • If this configuration item is set in GaussDB, the column name is displayed as all functions or expressions. – The MySQL client ignores common comments, but the gsql and PyMySQL clients do not. – The MySQL server converts comments starting with /*! into executable statements. An M-compatible database does not support such comments and processes them as common comments. – If an expression contains two hyphens (--) that is not followed by a space, an M-compatible database cannot identify the two hyphens as a comment, whereas the MySQL server identifies it as two hyphens (--).

Description	Syntax	Difference
		– If the displayed column name string contains escape characters, the escaped characters are displayed only when **m_format_behavior_compat_options** is set to a value that includes **enable_escape_string**. Otherwise, the escape characters are displayed. For example, in an M-compatible database, "SELECT"abc\tdef";" is displayed as abc\tdef when the preceding option is disabled. m_db=# SET m_format_behavior_compat_options='select_column_name,enable_escape_string'; SET m_db=# SELECT "abc\tdef"; abc def ---------- abc def (1 row) m_db=# SET m_format_behavior_compat_options='select_column_name'; SET m_db=# SELECT "abc\tdef"; abc\tdef ---------- abc\tdef (1 row) – If a column name contains more than 63 characters, the extra characters will be truncated. – If the last part of an expression is a comment, the last comment and the space connected to the comment are not displayed. m_db=# SELECT 123 /* 456 */; 123 ----- 123 (1 row)

Description	Syntax	Difference
		– If an expression is a Boolean value, the command output is **TRUE** or **FALSE** regardless of the input case. m_db=# SELECT true; TRUE ------ t (1 row) – If an expression is null, the command output is **NULL** regardless of the input case. m_db=# SELECT null; NULL ------ (1 row) – If an expression contains a hyphen (-), all inputs are output as column names. m_db=# SELECT (+-+1); (+-+1) -------- -1 (1 row) m_db=# SELECT -true; -true ------- -1 (1 row) m_db=# SELECT -null; -null ------- (1 row) • When pymysql is used to execute the SELECT statement, the prefix of the queried character string does not use ASCII characters, and the database is not encoded in UTF-8, the displayed column names are different from those in MySQL.

Description	Syntax	Difference
SELECT export file (into outfile)	SELECT ... INTO OUFILE ...	In the file exported by using the SELECT INTO OUTFILE syntax, the display precision of values of the FLOAT, DOUBLE, and REAL types in GaussDB is different from that in MySQL. The syntax does not affect the import using COPY the values after import.

Description	Syntax	Difference
Specify schema names and table names by using SELECT/UPDATE/ INSERT/REPLACE.	SELECT/UPDATE/ INSERT/REPLACE	• When the SELECT statement is used to the projection column, MySQL supports the three-segment format of *schema name.table alias.column name*, but GaussDB does not. m_db=# CREATE SCHEMA test; CREATE SCHEMA m_db=# CREATE TABLE test.t1(a int); CREATE TABLE m_db=# SELECT test.alias1.a FROM t1 alias1; ERROR: invalid reference to FROM-clause entry for table "alias1" LINE 1: SELECT test.alias1.a FROM t1 alias1; ^ HINT: There is an entry for table "alias1", but it cannot be referenced from this part of the query. CONTEXT: referenced column: a • The three-segment format for UPDATE/REPLACE SET is *database.table.column* in MySQL, and is *table.column.field* in GaussDB, where *field* indicates the attribute in the specified composite type. • For INSERT ... SET, MySQL supports *column*, *table.column*, and *database.table.column*. GaussDB supports only *column* and does not support *table.column* and *database.table.column*. • For INSERT... SET, you can reference column names and expressions that contain column names on the right of the SET clause in MySQL. GaussDB does not support this operation. – Behavior in GaussDB: m_db=# CREATE TABLE t2 (a int default 3, b int default 5); CREATE TABLE m_db=# INSERT INTO t2 SET a = b + 1; ERROR: Column "b" does not exist.

Description	Syntax	Difference
		LINE 1: INSERT INTO t2 SET a = b + 1; ^ HINT: There is a column named "b" in table "t2", but it cannot be referenced from this part of the query. m_db=# INSERT INTO t2 SET a = b + 1, b = 0; ERROR: Column "b" does not exist. LINE 1: INSERT INTO t2 SET a = b + 1, b = 0; ^ HINT: There is a column named "b" in table "t2", but it cannot be referenced from this part of the query. m_db=# INSERT INTO t2 SET b = 0, a = b + 1; ERROR: Column "b" does not exist. LINE 1: INSERT INTO t2 SET b = 0, a = b + 1; ^ HINT: There is a column named "b" in table "t2", but it cannot be referenced from this part of the query. m_db=# INSERT INTO t2 SET a = a + 1; ERROR: Column "a" does not exist. LINE 1: INSERT INTO t2 SET a = a + 1; ^ HINT: There is a column named "a" in table "t2", but it cannot be referenced from this part of the query. m_db=# DROP TABLE t2; DROP TABLE – Behavior in MySQL: mysql> CREATE TABLE t2 (a int default 3, b int default 5); Query OK, 0 rows affected (0.07 sec) mysql> INSERT INTO t2 SET a = b + 1; Query OK, 1 row affected (0.02 sec) mysql> SELECT * FROM t2; +------+------+ | a | b | +------+------+ | 6 | 5 | +------+------+ 1 row in set (0.00 sec)

Description	Syntax	Difference
		mysql> INSERT INTO t2 SET a = b + 1, b = 0; Query OK, 1 row affected (0.00 sec) mysql> SELECT * FROM t2; +------+------+ | a | b | +------+------+ | 6 | 5 | | 6 | 0 | +------+------+ 2 rows in set (0.00 sec) mysql> INSERT INTO t2 SET b = 0, a = b + 1; Query OK, 1 row affected (0.00 sec) mysql> SELECT * FROM t2; +------+------+ | a | b | +------+------+ | 6 | 5 | | 6 | 0 | | 1 | 0 | +------+------+ 3 rows in set (0.00 sec) mysql> INSERT INTO t2 SET a = a + 1; Query OK, 1 row affected (0.02 sec) mysql> SELECT * FROM t2; +------+------+ | a | b | +------+------+ | 6 | 5 | | 6 | 0 | | 1 | 0 | | 4 | 5 | +------+------+ 4 rows in set (0.00 sec) mysql> DROP TABLE t2; Query OK, 4 rows affected (0.40 sec)

Description	Syntax	Difference
The execution sequence of UPDATE SET is different from that of MySQL.	UPDATE ... SET	In MySQL, UPDATE SET is performed in sequence. The results of UPDATE at the front affect subsequent results of UPDATE, and the same column can be set for multiple times. In GaussDB, all related data is obtained first, and then UPDATE is performed on the data at a time. The same column cannot be updated for multiple times. After the GUC compatibility parameter **m_format_dev_version** is set to **'s2'**, the behavior can be the same as that in MySQL only in the single-table scenario. That is, the same column can be updated for multiple times and the updated result is referenced.
IGNORE feature	UPDATE/DELETE/ INSERT	The execution process in MySQL is different from that in GaussDB. Therefore, the number and information of generated warnings may be different.

Description	Syntax	Difference
SHOW COLUMNS syntax	SHOW	• User permission verification is different from that of MySQL. – In GaussDB, you need the USAGE permission on the schema of a specified table and table-level or column-level permissions on the specified table. Only information about columns with the SELECT, INSERT, UPDATE, REFERENCES, and COMMENT permissions is displayed. – In MySQL, you need table-level or column-level permissions on a specified table. Only information about columns with the SELECT, INSERT, UPDATE, REFERENCES, and COMMENT permissions is displayed. • When the LIKE and WHERE clauses involve string comparison, the fields **Field**, **Collation**, **Null**, **Extra**, and **Privileges** use the character set utf8mb4 and the collation utf8mb4_general_ci, and the fields **Type**, **Key**, **Default**, and **Comment** use the character set utf8mb4 and the collation utf8mb4_bin. • In GaussDB, you are advised not to select columns other than the returned fields in the WHERE clause. Otherwise, unexpected errors may occur. `-- Expected error` `m_db=# SHOW FULL COLUMNS FROM t02 WHERE `b`='pri';` `ERROR: Column "b" does not exist.` `LINE 1: SHOW FULL COLUMNS FROM t02 WHERE `b`='pri';` `^`

Description	Syntax	Difference
		-- Unexpected error m_db=# SHOW FULL COLUMNS FROM t02 WHERE `c`='pri'; ERROR: input of anonymous composite types is not implemented LINE 1: SHOW FULL COLUMNS FROM t02 WHERE `c`='pri'; ^
SHOW CREATE DATABASE syntax	SHOW	User permission verification is different from that of MySQL. • In GaussDB, you need the USAGE permission on a specified schema. • In MySQL, you need database-level permissions (except GRANT OPTION and USAGE), table-level permissions (except GRANT OPTION), or column-level permissions.

Description	Syntax	Difference
SHOW CREATE TABLE syntax	SHOW	• User permission verification is different from that of MySQL. – In GaussDB, you need the USAGE permission on the schema where a specified table is located and table-level permissions on the specified table. – Table-level permissions (except GRANT OPTION) of the specified table are required in MySQL. • The returned statements for table creation are different from those in MySQL. – In GaussDB, indexes are returned as CREATE INDEX statements. In MySQL, indexes are returned as CREATE TABLE statements. In GaussDB, the range of optional parameters supported by the CREATE INDEX syntax is different from that supported by the CREATE TABLE syntax. Therefore, some indexes cannot be created in CREATE TABLE statements. – In GaussDB, the **ENGINE** and **ROW_FORMAT** options of CREATE TABLE are adapted only for the syntax but do not take effect. Therefore, they are not displayed in the returned statements for table creation. • These statements are compatible with MySQL only after the compatibility parameter **m_format_dev_version** is set to **'s2'**. The compatibility parameter takes effect by

Description	Syntax	Difference
		changing the positions of column comments, table comments, **ON COMMIT** option for global temporary tables, primary key and unique constraints (where the **USING INDEX TABLESPACE** option is no longer displayed), and index comments.
SHOW CREATE VIEW syntax	SHOW	• User permission verification is different from that of MySQL. – In GaussDB, you need the USAGE permission on the schema where a specified view is located and table-level permissions on the specified view. – In MySQL, you need the table-level SELECT and table-level SHOW VIEW permissions on the specified view. • The returned statements for view creation are different from those in MySQL. If a view is created in the format of **SELECT * FROM** *tbl_name*, * is not expanded in GaussDB but expanded in MySQL. • The *character_set_client* and *collation_connection* fields in the returned result are different from those in MySQL. – The session values of system variables *character_set_client* and *collation_connection* are displayed during view creation in MySQL – Related metadata is not recorded in GaussDB and **NULL** is displayed.

Description	Syntax	Difference
SHOW PROCESSLIST syntax	SHOW	In GaussDB, the field content and case in the query result of this command are the same as those in the information_schema.processlist view. In MySQL, the field content and case may be different. • In GaussDB, common users can access only their own thread information. Users with the SYSADMIN permission can access thread information of all users. • In MySQL, common users can access only their own thread information. Users with the PROCESS permission can access thread information of all users.
SHOW [STORAGE] ENGINES	SHOW	In GaussDB, the field content and case of the query result of this command are the same as those in the information_schema.engines view. In MySQL, they may be different from those in the view. The query results of this command are different in MySQL and GaussDB because the databases have different storage engines.
SHOW [SESSION] STATUS	SHOW	In GaussDB, the field content and case of the query result of this command are the same as those in the information_schema.session_status view. In MySQL, they may be different from those in the view. Currently, GaussDB supports only **Threads_connected** and **Uptime**.

Description	Syntax	Difference
SHOW [GLOBAL] STATUS	SHOW	In GaussDB, the field content and case of the query result of this command are the same as those in the information_schema.global_status view. In MySQL, they may be different from those in the view. Currently, GaussDB supports only **Threads_connected** and **Uptime**.

Description	Syntax	Difference
SHOW INDEX	SHOW	• User permission verification is different from that of MySQL. – In GaussDB, you need the USAGE permission on a specified schema and table-level or column-level permissions on a specified table. – In MySQL, you need table-level (except GRANT OPTION) or column-level permission on the specified table. • Temporary tables in GaussDB are stored in independent temporary schemas. When using the FROM or IN db_name condition to display the index information of a specified temporary table, you must specify **db_name** as the schema where the temporary table is located. Otherwise, the system displays a message indicating that the temporary table does not exist. This is different from MySQL in some cases. • In the query result of GaussDB, the **Table**, **Index_type**, and **Index_comment** columns use the character set **utf8mb4** and collation **utf8mb4_bin**. The **Key_name**, **Column_name**, **Collation**, **Null**, and **Comment** columns use the character set **utf8mb4** and collation **utf8mb4_general_ci**.

Description	Syntax	Difference
SHOW SESSION VARIABLES	SHOW	In GaussDB, the field content and case of the query result are the same as those in the information_schema.session_variables view. In MySQL, they may be different from those in the view. In GaussDB, when LIKE and WHERE are used to select fields in the query result, the sorting rule is the same as that of the corresponding fields in the information_schema.session_variables view.
SHOW GLOBAL VARIABLES	SHOW	In GaussDB, the field content and case of the query result are the same as those in the information_schema.global_variables view. In MySQL, they may be different from those in the view. In GaussDB, when LIKE and WHERE are used to select fields in the query result, the sorting rule is the same as that of the corresponding fields in the information_schema.global_variables view.
SHOW CHARACTER SET	SHOW	In GaussDB, the field content and case of the query result are the same as those in the information_schema.character_sets view. In MySQL, they may be different from those in the view. In GaussDB, when LIKE and WHERE are used to select fields in the query result, the sorting rule is the same as that of the corresponding fields in the information_schema.character_sets view.

Description	Syntax	Difference
SHOW COLLATION	SHOW	In GaussDB, the field content and case of the query result are the same as those in the information_schema.collations view. In MySQL, they may be different from those in the view. In GaussDB, when LIKE and WHERE are used to select columns in the query result, the sorting rule is the same as that of the corresponding columns in the information_schema.collations view.

Description	Syntax	Difference
SHOW TABLES	SHOW	• The LIKE behavior is different. For details, see "LIKE" in **Operators**. • The WHERE expression behavior is different. For details, see "WHERE" in GaussDB. • In GaussDB, permissions on tables and databases must be assigned to users separately. The database to be queried must be available to users on the SHOW SCHEMAS. Users must have permissions on both tables and databases. MySQL can be accessed as long as you have table permissions. • In GaussDB, the verification logic preferentially checks whether a schema exists and then checks whether the current user has the permission on the schema, which is different from that in MySQL. • In GaussDB, fields in the query result use the character set utf8mb4 and collation utf8mb4_bin. • In the LIKE clause of GaussDB, if the target database is information_schema, the pattern is converted to lowercase letters before matching. In MySQL 8.0, when the target database is information_schema, the pattern is converted to uppercase letters before matching.

Description	Syntax	Difference
SHOW TABLE STATUS	SHOW	• In GaussDB, the syntax displays data depending on the tables view under information_schema. In MySQL, the tables view specifies tables. • In GaussDB, permissions on tables and databases must be assigned to users separately. The database to be queried must be available to users on the SHOW SCHEMAS. Users must have permissions on both tables and databases. MySQL can be accessed as long as you have table permissions. • In GaussDB, the verification logic preferentially checks whether a schema exists and then checks whether the current user has the permission on the schema, which is different from that in MySQL. • In GaussDB, when LIKE and WHERE are used to select fields in the query result, the sorting rule is the same as that of the corresponding fields in the information_schema.tables view. • In the LIKE clause of GaussDB, if the target database is information_schema, the pattern is converted to lowercase letters before matching. In MySQL 8.0, when the target database is information_schema, the pattern is converted to uppercase letters before matching.

Description	Syntax	Difference
SHOW DATABASES	SHOW	In GaussDB, fields in the query result use the character set utf8mb4 and collation utf8mb4_bin.
Support the **ONLY_FULL_GROUP_BY** option in **SQL_MODE**.	SELECT	If the non-aggregate function column in the SELECT list is inconsistent with the GROUP BY column, when all non-aggregate function columns are in the GROUP BY list or WHERE list and the column in the WHERE clause is equal to a constant, no error is reported. For the column in the WHERE clause, GaussDB supports function column expressions whose input parameter is **1**, but MySQL does not support function column expressions. In GaussDB, the column following GROUP BY must be a positive integer.
Query system parameters and user variables by using SELECT.	SELECT @variable, SELECT @@variable	• In MySQL, user variables can be queried without adding specific variable names (that is, **SELECT @**). GaussDB does not support this feature. Behavior in MySQL: `mysql> SELECT @;` `+------------+` `| @ |` `+------------+` `| NULL |` `+------------+` `1 row in set (0.00 sec)` Behavior in GaussDB: `m_db=# SELECT @;` `ERROR: syntax error at or near "@"` `LINE 1: SELECT @;` `^` • When the type of the queried system variable is BOOLEAN, the output result is **t** or **f** in GaussDB and **1** or **0** in MySQL. The BOOLEAN type is actually mapped to the TINYINT type.

Description	Syntax	Difference
Subqueries	SELECT	• In GaussDB, the subquery result cannot contain multiple columns. If the subquery result contains multiple columns, an error is reported. In MySQL, the subquery result can contain multiple columns. Behavior in MySQL: `mysql> SELECT row(1,2) = (SELECT 1,2);` `+-------------------------+` `| row(1,2) = (SELECT 1,2) |` `+-------------------------+` `| 1 |` `+-------------------------+` `1 row in set (0.00 sec)` Behavior in GaussDB: `m_db=# SELECT row(1,2) = (SELECT 1,2);` `ERROR: subquery must return only one column` `LINE 1: SELECT row(1,2) = (SELECT 1,2); ^` • In the scenario where precision transfer is enabled, if the return type in the FROM clause of a subquery is numeric in MySQL, one of the following conditions is met: – The SELECT clause contains GROUP BY. – The SELECT clause contains HAVING. – The SELECT clause contains DISTINCT. – The SELECT clause contains LIMIT. – The SELECT clause does not contain FROM *table*. – The SELECT clause contains a statement that assigns a value to a user-defined variable. Precision truncation may occur. If this type of subquery is used as the intermediate calculation value for the next operation,

Description	Syntax	Difference
		the precision of GaussDB is higher than that of MySQL. Behavior in MySQL: mysql> SELECT greatest((SELECT * FROM (SELECT DISTINCT c2/1.61 FROM t_time) t4), 1.00000000000000000); +---------- ---------- --+ | greatest((SELECT * FROM (SELECT DISTINCT c2/1.61 FROM t_time) t4), 1.00000000000000000) | +---------- ---------- --+ | 39144.72670800000000000 | +---------- ---------- --+ 1 row in set (0.00 sec) Behavior in GaussDB: m_db=# SELECT greatest((SELECT * FROM (SELECT DISTINCT c2/1.61 FROM t_time) t4), 1.00000000000000000); greatest ---------- 39144.72670807453416149 (1 row) In addition, PBE is used together with user-defined variables. If the preceding conditions are met, MySQL outputs results with the precision of 30 decimal places. Otherwise, the MySQL outputs results with the original precision, but GaussDB always outputs results with the precision of 30 decimal places. For example: Behavior in MySQL: -- The preceding conditions are met: mysql> SET @var6=12.1234567891; Query OK, 0 rows affected (0.00 sec) mysql> PREPARE p1 FROM "SELECT * FROM (SELECT @var6) t"; Query OK, 0 rows affected (0.00 sec) Statement prepared mysql> EXECUTE p1; +----------+ | @var6 | +----------+ | 12.123456789100000000000000000000 |

Description	Syntax	Difference
		+----------+ 1 row in set (0.00 sec) -- The preceding conditions are not met: mysql> PREPARE p1 FROM "SELECT * FROM (SELECT @var6 FROM (SELECT 1) v1) t"; Query OK, 0 rows affected (0.00 sec) Statement prepared mysql> EXECUTE p1; +---------------+ | @var6 | +---------------+ | 12.1234567891 | +---------------+ 1 row in set (0.00 sec) Behavior in GaussDB: -- The preceding conditions are met: m_db=# SET @var6=12.1234567891; SET m_db=# PREPARE p1 FROM "SELECT * FROM (SELECT @var6) t"; PREPARE m_db=# EXECUTE p1; @var6 ---------- 12.123456789100000000000000000000 (1 row) -- The preceding conditions are not met: m_db=# PREPARE p1 FROM "SELECT * FROM (SELECT @var6 FROM (SELECT 1) v1) t"; PREPARE m_db=# EXECUTE p1; @var6 ---------- 12.123456789100000000000000000000 (1 row)
SELECT followed by a row expression	SELECT	In MySQL, SELECT cannot be followed by a row expression, but in GaussDB, SELECT can be followed by a row expression. Behavior in MySQL: mysql> SELECT row(1,2); ERROR 1241 (21000): Operand should contain 1 column(s) Behavior in GaussDB: m_db=# SELECT row(1,2); row(1,2) ---------- (1,2) (1 row)

Description	Syntax	Difference
SELECT view query, subquery, or UNION involves the carry difference when NUMERIC is converted to TIME or DATETIME.	SELECT	In some SELECT scenarios, the results of the TIME/DATETIME type are different from those of MySQL. Difference scenarios involving conversion from NUMERIC to TIME/DATETIME: view query, subquery, and UNION. Differential behavior: The SELECT behavior of GaussDB is unified. When the NUMERIC type is converted to the TIME or DATETIME type, only the maximum precision bit(6) is carried. In MySQL view query, subquery, and UNION scenarios, carry is performed based on the actual precision of a result. Behavior in MySQL: -- In a simple query, carry is performed only on the result with the precision of 6, which is the maximum. Therefore, 11:11:00.00002 is output. mysql> SELECT maketime(11, 11, 2.2/ time '08:30:23.01'); +----------+ | maketime(11, 11, 2.2/time '08:30:23.01') | +----------+ | 11:11:00.00002 | +----------+ 1 row in set (0.01 sec) -- In a subquery, carry is performed based on the actual result precision. Therefore, 11:11:00.00003 is output. mysql> SELECT * FROM (SELECT maketime(11, 11, 2.2/time '08:30:23.01')) f1; +----------+ | maketime(11, 11, 2.2/time '08:30:23.01') | +----------+ | 11:11:00.00003 | +----------+ 1 row in set (0.00 sec) Behavior in GaussDB: m_db=# SET m_format_behavior_compat_options= 'enable_precision_decimal'; SET -- In a simple query, carry is performed only on the result with the precision of 6, which is the maximum. Therefore, 11:11:00.00002 is output. m_db=# SELECT maketime(11, 11, 2.2/ time '08:30:23.01');

Description	Syntax	Difference
		maketime ---------- 11:11:00.00002 (1 row) -- In a simple query, carry is performed only on the result with the precision of 6, which is the maximum, and the result precision is 5. Therefore, 11:11:00.00002 is output. m_db=# SELECT * FROM (SELECT maketime(11, 11, 2.2/time '08:30:23.01')) f1; maketime ---------- 11:11:00.00002 (1 row)
Differences of SELECT in calculating and processing date and time functions of the numeric type and subquery	SELECT	When date and time functions of the numeric type and subquery are calculated using SELECT, if the GUC parameter **m_format_behavior_compat_options** is set to **enable_precision_decimal**, GaussDB converts the value of the date and time type returned by the function to the one of the numeric type and then performs calculation based on the numeric type. The result is also of the numeric type. MySQL truncates the values returned by the date and time functions in scenarios such as subquery condition query and group query. Behavior in MySQL: mysql> SELECT 1.5688 * (SELECT adddate('2020-10-20', interval 1 day) WHERE true GROUP BY 1 HAVING true); +----------+ | 1.5688 * (SELECT adddate('2020-10-20', interval 1 day) WHERE true HAVING true) | +----------+ | 3168.976 | Behavior in GaussDB: m_db=# SELECT 1.5688 * (SELECT adddate('2020-10-20', interval 1 day) WHERE true GROUP BY 1 HAVING true); ?column? ---------- 31691361.744799998 (1 row)

Description	Syntax	Difference
Differences in unsigned types when SELECT nests subqueries	SELECT	When SELECT nests subqueries, the unsigned type is not overwritten, which is different from MySQL 5.7. Behavior in MySQL 5.7: mysql> DROP TABLE IF EXISTS t1; Query OK, 0 rows affected (0.02 sec) mysql> CREATE TABLE t1 (-> c10 real(10, 4) zerofill ->); Query OK, 0 rows affected (0.03 sec) mysql> INSERT INTO t1 VALUES(123.45); Query OK, 1 row affected (0.00 sec) mysql> DESC t1; +-------+--------------------------------- +------+-----+---------+-------+ | Field | Type | Null | Key | Default | Extra | +-------+--------------------------------- +------+-----+---------+-------+ | c10 | double(10,4) unsigned zerofill | YES | | NULL | | +-------+--------------------------------- +------+-----+---------+-------+ 1 row in set (0.01 sec) mysql> CREATE TABLE t1_sub_1 AS SELECT (SELECT * FROM t1); Query OK, 1 row affected (0.03 sec) Records: 1 Duplicates: 0 Warnings: 0 mysql> DESC t1_sub_1; +--------------------+--------------+------ +-----+---------+-------+ | Field | Type | Null | Key | Default | Extra | +--------------------+--------------+------ +-----+---------+-------+ | (SELECT * FROM t1) | double(10,4) | YES | | NULL | | +--------------------+--------------+------ +-----+---------+-------+ 1 row in set (0.00 sec) Behavior in GaussDB: test=# DROP TABLE IF EXISTS t1; DROP TABLE test=# CREATE TABLE t1 (test(# c10 real(10, 4) ZEROFILL test(#); CREATE TABLE test=# INSERT INTO t1 VALUES(123.45); INSERT 0 1 test=# DESC t1; Field | Type | Null | Key | Default | Extra -------+---------------------------------+------ +-----+---------+------- c10 | double(10,4) unsigned zerofill | YES | | | (1 row) test=# CREATE TABLE t1_sub_1 AS

Description	Syntax	Difference
		SELECT (SELECT * FROM t1); INSERT 0 1 test=# DESC t1_sub_1; Field | Type | Null | Key | Default | Extra -------+-----------------------+------+-----+---------+------- c10 | double(10,4) unsigned | YES | | | (1 row)

Description	Syntax	Difference
SELECT FOR SHARE/FOR UPDATE/ LOCK IN SHARE MODE	SELECT	• The FOR SHARE/FOR UPDATE/LOCK IN SHARE MODE and UNION/EXCEPT/ DISTINCT/GROUP BY/ HAVING clauses cannot be used together in GaussDB. They can be used together in MySQL 5.7 (except in the FOR SHARE/EXCEPT syntax) and MySQL 8.0. • When a lock clause is used together with the LEFT/ RIGHT [OUTER] JOIN clause, the LEFT JOIN cannot be used to lock the right table, and the RIGHT JOIN clause cannot be used to lock the left table. In MySQL, tables on both sides of JOIN can be locked at the same time. • In MySQL, multiple lock clauses cannot be specified for the same table, while GaussDB supports this operation and the strongest lock will take effect. -- GaussDB m_db=# DROP TABLE IF EXISTS t1; DROP TABLE m_db=# CREATE TABLE t1(a INT, b INT); CREATE TABLE m_db=# INSERT INTO t1 VALUES(1,2); INSERT 0 1 m_db=# SELECT * FROM t1 FOR UPDATE OF t1 LOCK IN SHARE MODE; a | b ---+--- 1 | 2 (1 row) m_db=# DROP TABLE t1; DROP TABLE -- MySQL mysql> DROP TABLE IF EXISTS t1; Query OK, 0 rows affected (0.05 sec) mysql> CREATE TABLE t1(a INT, b INT); Query OK, 0 rows affected (0.09 sec)

Description	Syntax	Difference
		mysql> INSERT INTO t1 VALUES(1,2); Query OK, 1 row affected (0.01 sec) mysql> SELECT * FROM t1 FOR UPDATE OF t1 LOCK IN SHARE MODE; ERROR 3569 (HY000): Table t1 appears in multiple locking clauses. mysql> DROP TABLE t1; Query OK, 0 rows affected (0.05 sec)

Description	Syntax	Difference
SELECT syntax	SELECT	• In GaussDB, HAVING can only reference columns in the GROUP BY clause or columns used in aggregate functions. However, MySQL supports more: it allows HAVING to reference **SELECT** columns in the list and columns in external subqueries. • In GaussDB, when an empty table is queried by using the specified WITH ROLLUP statement, the query result is an empty row. In contrast, the query result in MySQL is empty. • In GaussDB, a table alias with the column name can be specified by using the FROM clause. In MySQL 5.7, a table alias with the column name cannot be specified. In MySQL 8.0, it is allowed only in a subquery. -- GaussDB m_db=# DROP TABLE IF EXISTS t1; DROP TABLE m_db=# CREATE TABLE t1(a INT, b INT); CREATE TABLE m_db=# INSERT INTO t1 VALUES(1,2); INSERT 0 1 m_db=# SELECT * FROM t1 t2(a, b); a | b ---+--- 1 | 2 (1 row) m_db=# SELECT * FROM (SELECT * FROM t1) t2(a, b); a | b ---+--- 1 | 2 (1 row) -- MySQL 5.7 mysql> DROP TABLE IF EXISTS t1; Query OK, 0 rows affected, 1 warning (0.00 sec) mysql> CREATE TABLE t1(a INT, b INT); Query OK, 0 rows affected (0.03 sec) mysql> INSERT INTO t1 VALUES(1,2);

Description	Syntax	Difference
		Query OK, 1 row affected (0.01 sec) mysql> SELECT * FROM t1 t2(a, b); ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '(a, b)' at line 1 mysql> SELECT * FROM (SELECT * FROM t1) t2(a, b); ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '(a, b)' at line 1 -- MySQL 8.0 mysql> DROP TABLE IF EXISTS t1; Query OK, 0 rows affected (0.10 sec) mysql> CREATE TABLE t1(a INT, b INT); Query OK, 0 rows affected (0.18 sec) mysql> INSERT INTO t1 VALUES(1,2); Query OK, 1 row affected (0.03 sec) mysql> SELECT * FROM t1 t2(a, b); ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '(a, b)' at line 1 mysql> SELECT * FROM (SELECT * FROM t1) t2(a, b); +------+------+ | a | b | +------+------+ | 1 | 2 | +------+------+ 1 row in set (0.00 sec) • If a query statement does not contain the FROM clause, GaussDB supports the WHERE clause, which is the same as that in MySQL 8.0. MySQL 5.7 does not support the WHERE clause. -- GaussDB m_db=# SELECT 1 WHERE true; 1 --- 1 (1 row) -- MySQL 5.7 mysql> SELECT 1 WHERE true; ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'where true' at line 1

Description	Syntax	Difference
		-- MySQL 8.0 mysql> SELECT 1 WHERE true; +---+ | 1 | +---+ | 1 | +---+ 1 row in set (0.00 sec)

Description	Syntax	Difference
When statements such as UNION and GROUP BY that do not carry the ORDER BY clause are used to merge or aggregate data, the output data sequence may not be the same as that in MySQL because the executor operators are different.	SELECT	Take the GROUP BY scenario as an example. If the hashagg operator is used, the sequence is different from the original one. You are advised to add the ORDER BY clause in the scenario where the data sequence needs to be ensured. -- Initialize data. DROP TABLE IF EXISTS test; CREATE TABLE test(id INT); INSERT INTO test VALUES (1),(2),(3),(4),(5); -- GaussDB -- If precision transfer is disabled, the ID sequence is (1 3 2 4 5). m_db=# SET m_format_behavior_compat_options= ''; SET m_db=# SELECT /*+ use_hash_agg*/ id, pi() FROM test GROUP BY 1,2; id | pi ----+------------------ 1 | 3.141592653589793 3 | 3.141592653589793 2 | 3.141592653589793 4 | 3.141592653589793 5 | 3.141592653589793 (5 rows) -- When the precision transfer function is enabled, the ID sequence changes to (5 4 2 3 1) due to the value change. m_db=# SET m_format_behavior_compat_options= 'enable_precision_decimal'; SET m_db=# SELECT /*+ use_hash_agg*/ id, pi() FROM test GROUP BY 1,2; id | pi ----+---------- 5 | 3.141593 4 | 3.141593 2 | 3.141593 3 | 3.141593 1 | 3.141593 (5 rows) -- In MySQL, the ID sequence is the original one. mysql> SELECT id, pi() FROM test GROUP BY 1,2; +------+----------+ | id | pi() | +------+----------+ | 1 | 3.141593 | | 2 | 3.141593 | | 3 | 3.141593 | | 4 | 3.141593 | | 5 | 3.141593 | +------+----------+ 5 rows in set (0.00 sec)

Description	Syntax	Difference
Support the WITH AS statement.	SELECT UPDATE DELETE	• In the WITH RECURSIVE scenario of GaussDB, when the type, typmod, and collation of the non-recursive columns and subqueries are inconsistent with those of the result columns obtained by the subquery, a syntax error occurs, because this scenario does not support such inconsistency. • For the WITH recursion part, GaussDB does not support aggregate functions, window functions, FOR UPDATE/ SHARE, LIMIT, and OFFSET; MySQL supports FOR UPDATE/SHARE, and MySQL 8.0.19 and later versions support LIMIT and OFFSET. • For the WITH RECURSIVE part, GaussDB supports DISTINCT and GROUP BY, but MySQL does not. • In the WITH RECURSIVE scenario, when the value generated by the WITH RECURSIVE column is wider than that generated by the non-recursive column, MySQL truncates data in loose mode, or reports an error in strict mode; GaussDB does not truncate data and generates results consistent with those after MySQL widens data length. • When the WITH RECURSIVE part is used for an outer join, the supported scope of GaussDB differs from that of MySQL. -- The differences are as follows: -- Initialize data. DROP TABLE IF EXISTS t2; CREATE TABLE t2(c INT); INSERT INTO t2 VALUES (5); -- In GaussDB, the non-recursive part uses UNION SELECT 1::bigint to forcibly convert the column type of

Description	Syntax	Difference
		the non-recursive part. m_db=# WITH RECURSIVE cte AS (SELECT 1 AS a UNION SELECT 1::bigint UNION SELECT a+1 FROM cte RIGHT JOIN t2 ON t2.c>cte.a WHERE cte.a<3) SELECT * FROM cte; ERROR: recursive reference to query "cte" must not appear within an outer join LINE 1: ...AS a UNION SELECT 1::bigint UNION SELECT a+1 FROM cte RIGHT ... ^ -- MySQL 8.0 mysql> WITH RECURSIVE cte AS (SELECT 1 AS a UNION SELECT a+1 FROM cte RIGHT JOIN t2 ON t2.c>cte.a WHERE cte.a<3) SELECT * FROM cte; +------+ | a | +------+ | 1 | | 2 | | 3 | +------+ 3 rows in set (0.00 sec) DROP TABLE IF EXISTS t2;

Description	Syntax	Difference
INSERT ... ON DUPLICATE KEY UPDATE syntax	INSERT	• The format of *table-name.column-name* is not supported by VALUES() in the ON DUPLICATE KEY UPDATE clause in GaussDB, but is supported in MySQL. • INSERT ... In the query ON DUPLICATE KEY UPDATE statement, if the query is a UNION or EXCEPT subquery, MySQL 5.7 allows the UPDATE clause to reference column names in the subquery; however, this is not allowed in MySQL 8.0 and GaussDB. • In MySQL, when you use the ON DUPLICATE KEY UPDATE clause to update multiple columns, the result of the previous UPDATE statement affects the subsequent results. In addition, you can update the same column for multiple times. In GaussDB, the result of the previous UPDATE operation does not affect the subsequent results. In addition, the same column cannot be updated for multiple times. You can set the GUC compatibility parameter **m_format_dev_version** to **'s2'** to make its behavior the same as that in MySQL. That is, the same column can be updated for multiple times and the updated result is referenced. • When the UPDATE operation is performed on inserted data that violates the unique constraint, the number of affected rows returned by GaussDB is different from that returned by MySQL. When a data record is updated, GaussDB returns **1**

Description	Syntax	Difference
		and MySQL returns **2**. If such update does not change the value of an existing row, GaussDB returns **1** and MySQL returns **0**. • When the ON DUPLICATE KEY UPDATE clause updates the auto-increment column to **NULL** and the column contains the NOT NULL constraint, GaussDB reports the error "The null value in column xxx violates the not-null constraint." However, the operation can be performed in MySQL, and the corresponding auto-increment column is updated to **0**. • When a table column with a string is implicitly converted from VARCHAR to a numeric type, if the part that fails to be converted is 2 characters shorter than the whole string (for example, if no character of "AA" is successfully converted, the length difference is 2 – 0 = 2; if the first character of "4XY" is converted, the length difference is 3 – 1 = 2), no error is reported in MySQL. However, MySQL still reports an error when the length difference is not 2. GaussDB always reports an error if the string fails to be converted in the preceding scenario. m_db=# CREATE TABLE t1(a INT PRIMARY KEY, b VARCHAR(10)); NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "t1_pkey" for table "t1" CREATE TABLE m_db=# INSERT INTO t1 values(1, 'A'); INSERT 0 1 m_db=# INSERT INTO t1 VALUES(1, 'X') ON DUPLICATE KEY UPDATE b=values(a)+values(b); ERROR: The double value 'X' is incorrect. CONTEXT: referenced column: b m_db=# INSERT INTO t1 VALUES(1, 'YY')

Description	Syntax	Difference
		ON DUPLICATE KEY UPDATE b=values(a) +values(b); ERROR: The double value 'YY' is incorrect. CONTEXT: referenced column: b m_db=# INSERT INTO t1 VALUES(1, 'ZZZ') ON DUPLICATE KEY UPDATE b=values(a)+values(b); ERROR: The double value 'ZZZ' is incorrect. CONTEXT: referenced column: b m_db=# DROP TABLE t1; DROP TABLE mysql> CREATE TABLE t1(a INT PRIMARY KEY, b VARCHAR(10)); Query OK, 0 rows affected (0.00 sec) mysql> TRUNCATE TABLE t1; Query OK, 0 rows affected (0.01 sec) mysql> INSERT INTO t1 values(1, 'A'); Query OK, 1 row affected (0.00 sec) mysql> INSERT INTO t1 VALUES(1, 'X') ON DUPLICATE KEY UPDATE b=values(a) +values(b); ERROR 1292 (22007): Truncated incorrect DOUBLE value: 'X' mysql> INSERT INTO t1 VALUES(1, 'YY') ON DUPLICATE KEY UPDATE b=values(a) +values(b); Query OK, 2 rows affected, 2 warnings (0.00 sec) mysql> INSERT INTO t1 VALUES(1, 'ZZZ') ON DUPLICATE KEY UPDATE b=values(a) +values(b); ERROR 1292 (22007): Truncated incorrect DOUBLE value: 'ZZZ' mysql> DROP TABLE t1; Query OK, 0 rows affected (0.00 sec)

3.7.5 DCL

Table 3-33 DCL syntax compatibility

Description	Syntax	Difference
Set names with COLLATE specified.	SET [SESSION | LOCAL] NAMES {'charset_name' [COLLATE 'collation_name'] | DEFAULT};	In GaussDB, you cannot specify **charset_name** to be different from that of the database character set. For details, see "SQL Reference > SQL Syntax > SQL Statements > S > SET" in M Compatibility Developer Guide. If no character set is specified, MySQL reports an error but GaussDB does not.

Description	Syntax	Difference
DESCRIBE statements are supported.	{DESCRIBE | DESC} tbl_name [col_name | wild]	• User permission verification is different from that of MySQL. – In GaussDB, you need the USAGE permission on the schema of a specified table and table-level or column-level permissions on the specified table. Only information about columns with the SELECT, INSERT, UPDATE, REFERENCES, and COMMENT permissions is displayed. – In MySQL, you need table-level or column-level permissions on a specified table. Only information about columns with the SELECT, INSERT, UPDATE, REFERENCES, and COMMENT permissions is displayed. • If character string comparison is involved in fuzzy match, the **Field** field uses the character set utf8mb4 and collation utf8mb4_general_ci.

Description	Syntax	Difference
SET sets user variables.	SET @var_name := expr	• In MySQL, user-defined variable names can be escaped using escape characters or double quotation marks, but this feature is not supported in GaussDB. Variable names enclosed in single quotation marks cannot contain other single quotation marks. For example, @'', @''', and @'\'' are not supported. During parsing, the single quotation marks (') cannot be matched or an error will be reported. For example: -- An error is reported during parsing. db_mysql=# SET @'''' = 1; ERROR: syntax error at or near "@" LINE 1: SET @'''' = 1; -- The single quotation marks (') cannot be matched during parsing. db_mysql=# SET @'\'' = 1; db_mysql'# Variable names enclosed in double quotation marks cannot contain double quotation marks ("). For example, @"", @"""" and @"\"" are not supported. The double quotation marks cannot be matched or an error will be reported during parsing. For example: -- An error is reported during parsing. db_mysql=# SET @"""" = 1; ERROR: syntax error at or near "@" LINE 1: SET @"""" = 1; -- The double quotation marks (") cannot be matched during parsing. db_mysql=# SET @"\"" = 1; db_mysql"# The variable name enclosed by backquotes cannot contain backquotes. For example,

Description	Syntax	Difference
		@````, @`````, and @`\`` are not supported. During parsing, the backquotes (`) cannot be matched or an error will be reported. For example: -- An error is reported during parsing. db_mysql=# SET @```` = 1; ERROR: syntax error at or near "@" LINE 1: SET @```` = 1; -- The backquotes (`) cannot be matched during parsing. db_mysql=# SET @`\`` = 1; db_mysql`# ● For example, **set @var_name1 = @var_name2 := @var_name3 = @var_name4 := expr;** can be used to assign consecutive values in MySQL, but cannot in GaussDB. db_mysql=# set @a := @b := @c = @d := 1; ERROR: user_defined variables cannot be set, such as @var_name := expr is not supported. ● expr can be an aggregate function in GaussDB but not in MySQL.

Description	Syntax	Difference
SET sets system parameters.	SET [SESSION | @@SESSION. | @@ | LOCAL | @@LOCAL.] {config_parameter { TO | = } { expr | DEFAULT } | FROM CURRENT }};	• When **config_parameter** is a system parameter of the BOOLEAN type: – The parameter value can be set to **'1'** or **'0'** or **'true'** or **'false'** in the character string format in GaussDB but cannot in MySQL. – If the parameter value is set to the subquery result, when the result is **'true'** or **'false'** and the non-integer type is **1** or **0**, the setting will be successful in GaussDB but will fail in MySQL. When the query result is **NULL**, the setting in GaussDB will fail but will be successful in MySQL.
Switch the current mode with USE.	USE schema_name	If the USE statement is used to specify a schema and the user does not have USAGE permissions on the schema, MySQL reports an error while GaussDB specifies the current schema as null. -- MySQL mysql> USE test; ERROR 1044 (42000): Access denied for user 'u1'@'%' to database 'test' -- GaussDB m_db=> USE test; SET m_db=> SELECT database(); ERROR: function returned NULL CONTEXT: referenced column: database

3.7.6 Other Statements

Table 3-34 Compatibility of other syntaxes

Description	Syntax	Difference
Lock mechanism	Lock mechanism	• The GaussDB lock mechanism can be used only in transaction blocks. There is no such restriction in MySQL. • After the read lock is obtained, write operations cannot be performed on the current session in MySQL, but write operations can be performed on the current session in GaussDB. • After MySQL locks a table, an error is reported when other tables are read. GaussDB does not have such restriction. • In MySQL, if the lock of the same table is obtained in the same session, the previous lock is automatically released and the transaction is committed. GaussDB does not have this mechanism. • In GaussDB, LOCK TABLE can be used only inside a transaction block, and UNLOCK TABLE will not be used. Locks are always released at the end of transactions.

Description	Syntax	Difference
PBE	PBE	• In GaussDB, if a PREPARE statement with the same name is repeatedly created, an error is reported, indicating that the statement already exists. You need to delete the existing statement first. In MySQL, the old statement will be overwritten. • GaussDB and MySQL report errors in different phases, such as parsing and execution, during SQL statement execution. PREPARE statements process prepared statements till the parsing phrase. Therefore, in abnormal scenarios in PBE, GaussDB may be different from MySQL in terms of whether the error is reported in the PREPARE or EXECUTE phase.
Single-line comment syntax	Single-line comment syntax	The single-line comment syntax is consistent with MySQL only when the **m_format_behavior_compat_options** parameter includes the **'forbid_none_space_comment'** option.

3.7.7 Users and Permissions

Overview

For details about user and permission management of GaussDB, see "Database Security Management > Managing Users and Their Permissions" in *Developer Guide*.

For details about the user and permission syntax, see "SQL Reference > SQL Syntax > SQL Statements" in *M Compatibility Developer Guide*.

Differences

- Syntax format differences

  For details about the authorization syntax of GaussDB, see "SQL Reference > SQL Syntax > SQL Statements > G > GRANT" in *M Compatibility Developer Guide*.

  The authorization syntax in MySQL is as follows:

```
-- Global, database-level, table-level, and stored procedure–level permission granting syntax
GRANT
    priv_type [(column_list)]
      [, priv_type [(column_list)]] ...
    ON [object_type] priv_level
    TO user [auth_option] [, user [auth_option]] ...
    [REQUIRE {NONE | tls_option [[AND] tls_option] ...}]
    [WITH {GRANT OPTION | resource_option} ...]

-- Syntax for granting permissions to a user proxy
GRANT PROXY ON user
    TO user [, user] ...
    [WITH GRANT OPTION]

object_type: {
    TABLE
  | FUNCTION
  | PROCEDURE
}

priv_level: {
    *
  | *.*
  | db_name.*
  | db_name.tbl_name
  | tbl_name
  | db_name.routine_name
}

user:
    'user_name'@'host_name'

auth_option: {
    IDENTIFIED BY 'auth_string'
  | IDENTIFIED WITH auth_plugin
  | IDENTIFIED WITH auth_plugin BY 'auth_string'
  | IDENTIFIED WITH auth_plugin AS 'auth_string'
  | IDENTIFIED BY PASSWORD 'auth_string'
}

tls_option: {
    SSL
  | X509
  | CIPHER 'cipher'
  | ISSUER 'issuer'
  | SUBJECT 'subject'
}

resource_option: {
  | MAX_QUERIES_PER_HOUR count
  | MAX_UPDATES_PER_HOUR count
  | MAX_CONNECTIONS_PER_HOUR count
  | MAX_USER_CONNECTIONS count
}
```

- Differences in types of permissions granted

  In MySQL, the following types of permissions can be granted.

Table 3-35 Types of permissions that can be granted in MySQL

Permission Type	Definition and Permission Level
ALL [PRIVILEGES]	Grants all permissions of a specified access level, except **GRANT OPTION** and **PROXY**.
ALTER	Enables **ALTER TABLE**. Level: global, database, and table.
ALTER ROUTINE	Allows you to modify or delete stored procedures. Level: global, database, and routine.
CREATE	Enables database and table creation. Level: global, database, and table.
CREATE ROUTINE	Enables stored procedure creation. Level: global and database.
CREATE TABLESPACE	Allows you to create, modify, or delete tablespaces or log file groups. Level: global.
CREATE TEMPORARY TABLES	Enables **CREATE TEMPORARY TABLE**. Level: global and database.
CREATE USER	Enable **CREATE USER**, **DROP USER**, **RENAME USER**, and **REVOKE ALL PRIVILEGES**. Level: global.
CREATE VIEW	Allows you to create or modify views. Level: global, database, and table.
DELETE	Enable **DELETE**. Level: global, database, and table.
DROP	Allows you to delete databases, tables, or views. Level: global, database, and table.
EVENT	Enable scheduled tasks. Level: global and database.
EXECUTE	Allows you to execute stored procedures. Level: global, database, and stored procedure.
FILE	Allows you to enable the server to read or write files. Level: global.
GRANT OPTION	Allows you to grant permissions to or remove permissions from other accounts. Level: global, database, table, stored procedure, and proxy.
INDEX	Allows you to create or delete indexes. Level: global, database, and table.
INSERT	Enables **INSERT**. Level: global, database, table, and column.

Permission Type	Definition and Permission Level
LOCK TABLES	LOCK TABLES is enabled on tables with the SELECT permission. Level: global and database.
PROCESS	Allows you to view all running threads through **SHOW PROCESSLIST**. Level: global.
PROXY	Enables a user proxy. Level: from user to user.
REFERENCES	Enables foreign key creation. Level: global, database, table, and column.
RELOAD	Enables **FLUSH**. Level: global.
REPLICATION CLIENT	Allows you to query the location of the source server or replica server. Level: global.
REPLICATION SLAVE	Allows replicas to read binary logs from the source. Level: global.
SELECT	Enables **SELECT**. Level: global, database, table, and column.
SHOW DATABASES	Enables **SHOW DATABASES** to display all databases. Level: global.
SHOW VIEW	Enables **SHOW CREATE VIEW**. Level: global, database, and table.
SHUTDOWN	Enables **mysqladmin shutdown**. Level: global.
SUPER	Enables other management operations, such as the **CHANGE MASTER TO**, **KILL**, **PURGE BINARY LOGS**, **SET GLOBAL**, and **mysqladmin debug** commands. Level: global.
TRIGGER	Enables **TRIGGER**. Level: global, database, and table.
UPDATE	Enables **UPDATE**. Level: global, database, table, and column.
USAGE	Equivalent to "no privilege".

GaussDB supports the following permissions based on objects.

Table 3-36 Types of permissions that can be granted in GaussDB

Object	Permissions That Can Be Granted
Database	CREATE, CONNECT, TEMPORARY, TEMP, ALTER, DROP, and COMMENT
Schema	CREATE, USAGE, ALTER, DROP, and COMMENT

Object	Permissions That Can Be Granted
Table and view	SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, ALTER, DROP, COMMENT, INDEX, and VACUUM
Column	SELECT, INSERT, UPDATE, REFERENCES, and COMMENT
Sequence	SELECT, USAGE, UPDATE, ALTER, DROP, and COMMENT

- In MySQL, *'*.*'* represents a global-level authorization object; in GaussDB, *'{DATABASE} db_name'* represents a database-level authorization object. The database level of GaussDB corresponds to the global level of MySQL.
- In MySQL, *'schema_name.*'* represents a database/schema-level authorization object; in GaussDB, *'{SCHEMA} schema_name'* represents a schema-level authorization object. The schema level of GaussDB corresponds to the database/schema level of MySQL.
- In MySQL, a username consists of two parts: *username*@*hostname*, but a username is only itself in GaussDB.
- MySQL allows you to modify user authentication, secure connection, and resource parameter attributes (including **auth_option**, **tls_option**, and **resource option**) with the GRANT syntax. In GaussDB, permission granting syntax does not support this function, and you need to use CREATE USER and ALTER USER to set user attributes.
- MySQL supports permission granting with a user proxy. GRANT PROXY ON is used to manage permissions of users in a unified manner. MySQL 5.7 does not provide the role mechanism, while both MySQL 8.0 and GaussDB provide the role mechanism. If a role can manage and control the permissions of users in a unified manner, it can replace GRANT PROXY ON.
- GaussDB has a concept called public. All users have public permissions and they can query some system catalogs and system views. Users can grant or revoke public permissions. In MySQL, newly created users have only the global usage permission, which is almost low to none. They have only the permission to connect to the database and query the information_schema database.
- In GaussDB, the owner of an object has all permissions on the object by default. For security purposes, the owner can discard some permissions. However, ALTER, DROP, COMMENT, INDEX, VACUUM, and re-grantable permissions on the object are implicitly inherent permissions of the owner: MySQL does not have a concept called owner. Even if a user creates a table, the user cannot perform operations such as IUD on the table without being granted the corresponding permissions.
- In MySQL, All users have the USAGE permission, which indicates no permission. When **REVOKE** or **GRANT USAGE** is executed, no modification is performed. In GaussDB, the USAGE permission has the following meanings:
  - For schemas, USAGE allows access to objects contained in the schema. Without this permission, it is still possible to see the object names.
  - For sequences, USAGE allows use of the nextval function.

- In GaussDB, administrator roles can be set for users, including system administrator (SYSADMIN), security administrator (CREATEROLE), audit administrator (AUDITADMIN), monitor administrator (MONADMIN), O&M administrator (OPRADMIN), and security policy administrator (POLADMIN). By default, system administrators with the SYSADMIN attribute have the highest permission in the system. After separation of duties is enabled, a system administrator does not have the CREATEROLE or AUDITADMIN attribute. That is, the system administrator can neither create roles or users, nor view or maintain database audit logs. In MySQL, administrator roles cannot be set for users, and there is no design for separation of duties.
- In GaussDB, the ANY permission can be granted to a user, indicating that the user can have the corresponding permission in non-system mode, including CREATE ANY TABLE, SELECT ANY TABLE, and CREATE ANY INDEX. In MySQL, ANY permission cannot be granted.
- MySQL provides **SHOW GRANTS** to query user permissions. In GaussDB, you can run a gsql client meta-command **'\l+'**, **'\dn+'**, or **'\dp'** to query permission information, or query related columns in system catalogs such as pg_namespace, pg_class, and pg_attribute for permission information.
- After a database, table, or column is deleted, the related permission granting information is still retained in system catalogs in MySQL. If an object with the same name is created again, the user still has the original permissions. However, related permission granting information will be deleted from GaussDB. If an object with the same name is created again, permissions need to be granted again.
- When database-level permissions are granted, MySQL supports fuzzy match of database names using underscores (_) and percent signs (%). However, GaussDB does not support fuzzy match of object names using special characters such as underscores (_) or percent signs (%), which are identified as common characters.
- In MySQL, if a user specified in the GRANT statement does not exist, a user account is created by default (this feature has been removed from MySQL 8.0). In GaussDB, permissions cannot be granted to users who are not created.

3.7.8 System Catalogs and System Views

Table 3-37 Differences in system catalogs or views between GaussDB and MySQL

System Catalog or System View	Column	Differences between GaussDB and MySQL
information_schema.columns	generation_expression	The output of this column varies due to different string concatenation logics of expressions in GaussDB and MySQL.
	data_type	The output result of this column in GaussDB, having not been modified due to the data type format_type involved, is different from that in MySQL.

System Catalog or System View	Column	Differences between GaussDB and MySQL
	column_type	The output result of this column in GaussDB, having not been modified due to the data type format_type involved, is different from that in MySQL.
information_schema.tables	engine	In GaussDB: • ENGINE is aligned with data of information_schema.engines. • In some system catalogs, ENGINE is left empty. • If the default table is an ASTORE table and **STORAGE_TYPE** is not specified, **ENGINE** is empty.
	version	GaussDB does not support this column.
	row_format	GaussDB does not support this column.
	avg_row_length	In GaussDB, the result of dividing the size of the data files by the number of all tuples (including live tuples and dead tuples) is used. If there is no tuple in the table, the value is **null**.
	max_data_length	GaussDB does not support this column.
	data_free	In GaussDB, it indicates the result of (number of dead tuples/total number of tuples) multiplied by data file size. If there is no tuple in the table, the value is **null**.
	check_time	GaussDB does not support this column.
	create_time	In GaussDB, the behavior of this column is different from that in MySQL. When a view is created in MySQL, this column is set to **null**. In GaussDB, the actual table creation time is displayed. The value is **null** if it is a table or view provided by the database.
	update_time	The value is **null** if it is a table or view provided by GaussDB.
	table_collation	The behavior of this column in GaussDB is different from that in MySQL. The value is **null** if the table specifies a view. The value is **null** if the COLLATE clause is not used to specify the collation of columns when the specified table is created.

System Catalog or System View	Column	Differences between GaussDB and MySQL
information_schema.statistics	collation	GaussDB supports only values **A** and **D** and does not support **NULL**.
	packed	GaussDB does not support this column.
	sub_part	GaussDB does not support this column.
	comment	GaussDB does not support this column.
information_schema.partitions	subpartition_name	In GaussDB, if the partition is not a level-2 partition, the value is **null**.
	subpartition_ordinal_position	In GaussDB, if the partition is not a level-2 partition, the value is **null**.
	partition_method	Partitioning policy in GaussDB. If the partition is not a level-1 partition, the value is **null**. ● **'r'**: range partition. ● **'i'**: interval partition. ● **'l'**: list partition. ● **'h'**: hash partition
	subpartition_method	Level-2 partitioning policy in GaussDB. If the partition is not a level-2 partition, the value is **null**. ● **'r'**: range partition. ● **'i'**: interval partition. ● **'l'**: list partition. ● **'h'**: hash partition
	partition_description	GaussDB classifies partitions as level-1 and level-2 partitions.
	partition_expression	GaussDB does not support this column.
	subpartition_expression	GaussDB does not support this column.
	data_length	GaussDB does not support this column.
	max_data_length	GaussDB does not support this column.
	index_length	GaussDB does not support this column.
	data_free	GaussDB does not support this column.
	create_time	GaussDB does not support this column.

System Catalog or System View	Column	Differences between GaussDB and MySQL
	update_time	GaussDB does not support this column.
	check_time	GaussDB does not support this column.
	checksum	GaussDB does not support this column.
	partition_comment	GaussDB does not support this column.
	nodegroup	GaussDB does not support this column.

NOTE

- The precision range cannot be specified for the command output of the integer type in a view. For example, the bigint(1) type in MySQL corresponds to the bigint type in GaussDB, and the bigint(21) unsigned type in MySQL corresponds to the bigint unsigned type in GaussDB.
- The int type in MySQL corresponds to the integer type in GaussDB.
- This version does not support or display **Column_priv** column in the m_schema.columns_priv view, **Table_priv,Column_priv** column in the m_schema.tables_priv view, **Routine_type,Proc_priv** column in the m_schema.procs_priv view, the **type,language,sql_data_access,is_deterministic,security_type,sql_mode** column in the m_schema.proc view, or the **type** column in the m_schema.func view.
- Some columns in information_schema.tables, information_schema.statistics, and information_schema.partitions are obtained based on statistics. Therefore, run **ANALYZE** to update statistics before viewing them. (If data is updated in the database, you are advised to delay running **ANALYZE**.)
- The index columns contained in information_schema.statistics must be complete table columns in the created indexes. If the index columns are expressions, they are not in this view.
- In information_schema.partitions, level-1 and level-2 partitions are displayed separately.
- In MySQL, the format of the **grantee** column in the view is '*user_name*'**@**'*host_name*'. In GaussDB, it is the name of the user or role to which the permission is granted.
- The **host** column in the view returns the host name of the current node in GaussDB.
- In MySQL, you need the permission before viewing m_schema.tables_priv, information_schema.user_privileges, information_schema.schema_privileges, information_schema.table_privileges, information_schema.column_privileges, m_schema.columns_priv, m_schema.func, and m_schema.procs_priv. In GaussDB, you can view them with the default permission. For example, for table **t1**, you need the corresponding permission in MySQL so that you can view the corresponding permission information in the permission view. In GaussDB, you can view the permission information related to table **t1** in the view.
- A system view in m_schema is a system catalog in MySQL.
- The collations of **VIEW_DEFINITION** in information_schema.views and **ROUTINE_DEFINITION** in information_schema.routines are not controlled.
- For the view columns of the character type listed in "Schemas" in *M Compatibility Developer Guide*, the character set is utf8mb4, and the collation is utf8mb4_bin or utf8mb4_general_ci, and the collation priority is the priority of columns of data types that support collation described in "SQL Reference > Character Set and Collations > Rules for Combining Character Sets and Collations" in *M Compatibility Developer Guide*. These features are different from those in MySQL.

3.8 Drivers

In GaussDB, when the PBE driver parameters are set in text mode, if the compatibility parameter **m_format_behavior_compat_options** does not contain the **disable_zero_chars_conversion** option, the server replaces the \0 character in the parameter with a space, which is different from MySQL. If the compatibility parameter **m_format_behavior_compat_options** contains the **disable_zero_chars_conversion** option, the character \0 cannot be converted to a space, which is the same as MySQL.

3.8.1 ODBC

3.8.1.1 ODBC API Reference

Obtaining Parameter Description

SQLDescribeParam is a function in the ODBC API. It is used to obtain the description of parameters related to prepared SQL statements (for example, calling SQLPrepare). It can return metadata such as the type, size, and whether **NULL** values are allowed for parameters, which is useful for dynamically building SQL statements and binding parameters.

Prototype

```
SQLRETURN SQLDescribeParam(
    SQLHSTMT        StatementHandle,
    SQLUSMALLINT    ParameterNumber,
    SQLSMALLINT    *DataTypePtr,
    SQLULEN        *ParameterSizePtr,
    SQLSMALLINT    *DecimalDigitsPtr,
    SQLSMALLINT    *NullablePtr);
```

Table 3-38 Parameters of SQLDescribeParam

Parameter	Description	Difference
StatementHandle	Statement handle.	-
ParameterNumber	Parameter marker number, starting with 1 and increasing in ascending order.	-
DataTypePtr	Points to the data type of the returned parameter.	In MySQL, ODBC returns **SQL_VARCHAR** for any type. In GaussDB, ODBC returns the data type to an application based on that returned by the kernel.
ParameterSizePtr	Points to the size of the returned parameter.	If MySQL allows the ODBC driver to use a larger data packet for data transmission, **24M** is returned. Otherwise, **255** is returned. In GaussDB, ODBC returns the parameter size based on the actual type.
DecimalDigitsPtr	Points to the number of decimal digits of the returned parameter.	-

Parameter	Description	Difference
NullablePtr	Points to whether **NULL** values are allowed for the returned parameter.	In MySQL, ODBC directly returns **SQL_NULLABLE_UNKNOWN**. In GaussDB, ODBC directly returns **SQL_NULLABLE**.