GaussDB

MySQL Compatibility(Distributed)

Issue	01
Date	2025-06-30

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Huawei Cloud Computing Technologies Co., Ltd.

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

Contents

1 Overview......1
1.1 Overview of MYSQL-compatible Mode......1
1.2 Overview of M-compatible Mode......1
2 MYSQL-compatible Mode......3
2.1 Data Types......3
2.1.1 Numeric Data Types......3
2.1.2 Date and Time Data Types......11
2.1.3 String Data Types......26
2.1.4 Binary Data Types......31
2.1.5 JSON Data Type......34
2.1.6 Attributes Supported by Data Types......34
2.1.7 Data Type Conversion......34
2.2 System Functions......37
2.2.1 Flow Control Functions......38
2.2.2 Date and Time Functions......40
2.2.3 String Functions......53
2.2.4 Forced Conversion Functions......59
2.2.5 Encryption Functions......59
2.2.6 JSON Functions......59
2.2.7 Aggregate Functions......62
2.2.8 Arithmetic Functions......64
2.2.9 Other Functions......65
2.3 Operators......65
2.4 Character Sets......67
2.5 Collation Rules......67
2.6 SQL......68
2.6.1 DDL......68
2.6.2 DML......79
2.6.3 DCL......92
2.7 Drivers......92
2.7.1 JDBC......92
2.7.1.1 JDBC API Reference......92
3 M-compatible Mode......94

3.1 Data Types....94
3.1.1 Numeric Data Types....94
3.1.2 Date and Time Data Types....97
3.1.3 String Data Types....99
3.1.4 Binary Data Types....103
3.1.5 Attributes Supported by Data Types....109
3.1.6 Data Type Conversion....110
3.2 System Functions....114
3.2.1 System Function Compatibility Overview....114
3.2.2 Flow Control Functions....116
3.2.3 Date and Time Functions....116
3.2.4 String Functions....122
3.2.5 Forced Conversion Functions....126
3.2.6 Encryption Functions....127
3.2.7 Comparison Functions....128
3.2.8 Aggregate Functions....129
3.2.9 Arithmetic Functions....130
3.2.10 Other Functions....131
3.3 Operators....132
3.4 Character Sets....143
3.5 Collation Rules....144
3.6 Transactions....145
3.7 SQL....149
3.7.1 Keywords....150
3.7.2 Identifiers....150
3.7.3 DDL....152
3.7.4 DML....175
3.7.5 DCL....206
3.7.6 Other Statements....207
3.7.7 Users and Permissions....209
3.7.8 System Catalogs and System Views....215
3.8 Drivers....219
3.8.1 ODBC....219
3.8.1.1 ODBC API Reference....220

1 Overview

1.1 Overview of MYSQL-compatible Mode

MYSQL-compatible Mode compares GaussDB in MYSQL-compatible mode (that is, when **sql_compatibility** is set to **'MYSQL'**, **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

The implementation logic of MYSQL-compatible mode (**sql_compatibility** set to **'MYSQL'**) in distributed mode is similar to that of the B-compatible mode (**sql_compatibility** set to **'B'**) in centralized mode. For details, see "Overview > B-compatible Mode" in the centralized *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 505.2.0 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.

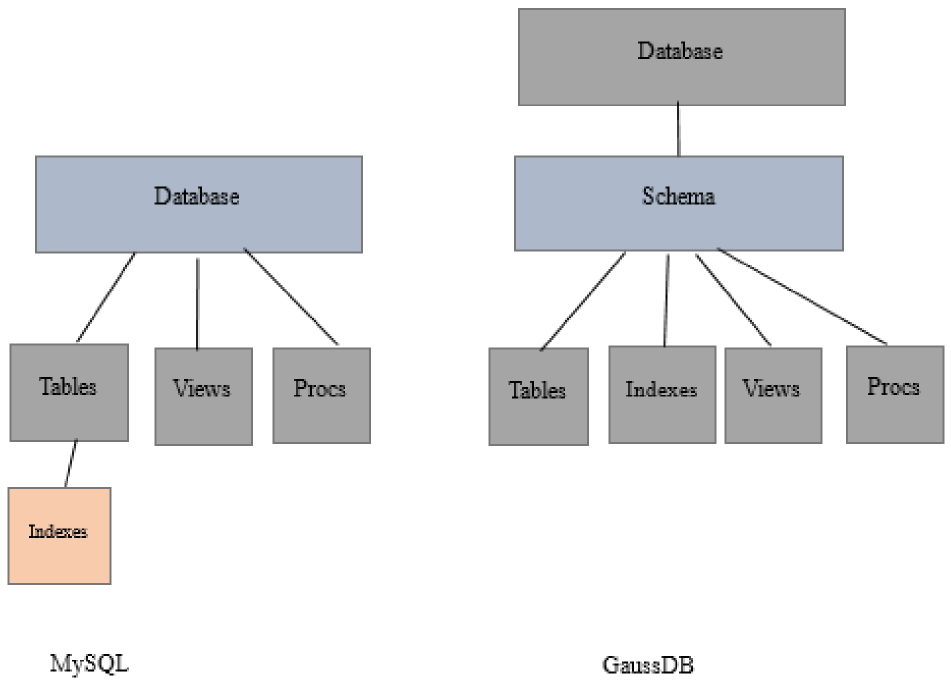
NOTE

The underlying architecture of GaussDB 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 MYSQL-compatible Mode

2.1 Data Types

2.1.1 Numeric Data Types

Integer

Unless otherwise specified, the precision, scale, and number of bits cannot be defined as the floating-point values in MYSQL-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.

- 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 between integer types in GaussDB and MySQL, 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 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 formats – 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 formats 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 ranges
		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.
		● Operators
		– 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: In GaussDB, 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 ranges 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. ● Operators – 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), varchar(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 zones
		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 formats: – 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 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, 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 ranges 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. ● Operators – 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 zones 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 formats: – 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 formats: 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 ranges –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.
		● Operators
		– 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: ● Operators – 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	<ul><li>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.</li><li>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.</li><li>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.</li></ul>

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',...)	Not 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 MYSQL-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 MYSQL-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
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.

- Using CAST(expr AS datatype) can explicitly convert 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 one input parameter is of the FLOAT4 type and the other is of any type in the numeric category, GaussDB returns the DOUBLE type. In MySQL, if one input parameter is of FLOAT4 type and the other is of the TINYINT, UNSIGNED TINYINT, SMALLINT, UNSIGNED SMALLINT, MEDIUMINT, UNSIGNED MEDIUMINT, or BOOL type, the FLOAT4 type is returned. If the first is of FLOAT4 type and the second is of BIGINT or UNSIGNED BIGINT type, the FLOAT type is returned.

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 MySQL-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, dates beyond the MySQL ranges 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 MySQL-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 an input 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 an input 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 an input 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	<ul><li>MySQL returns NULL if the second input parameter is a string in the DATETIME format. GaussDB can calculate the value.</li><li>The value range of an input parameter is ['0001-01-01 00:00:00', 9999-12-31 23:59:59.999999].</li><li>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:<ul><li>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.</li><li>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.</li><li>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.</li><li>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.</li><li>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.</li><li>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.</li><li>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.</li><li>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.</li></ul></li></ul>

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 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,
999999999999999999999999999999999999999999999999999999999999999999999999999]','$[6]',json
_insert('[1,4]','$[5]',999999999999999999999999999999999999999999999999999999999999999999999
999999));
        json_insert
------------------------------
 [1, 4, 1e+74, [1, 4, 1e+74]]
(1 row)
```

Table 2-15 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, with differences	GaussDB returns values of the int type, while MySQL returns values of the bigint type.
JSON_EXTRACT()	Supported.	-
JSON_INSERT()	Supported.	-
JSON_KEYS()	Supported.	-
JSON_LENGTH()	Supported, with differences	Return value difference: In GaussDB, int is returned. In MySQL, bigint is returned.
JSON_MERGE()	Supported.	-

MySQL	GaussDB	Difference
JSON_OBJECT()	Supported.	-
JSON_QUOTE()	Supported, with differences	Return value difference: In GaussDB, JSON is returned. In MySQL, varchar or text is returned.
JSON_REMOVE()	Supported.	-
JSON_REPLACE()	Supported.	-
JSON_SEARCH()	Supported, with differences	Return value difference: In GaussDB, text is returned. In MySQL, JSON is returned.
JSON_SET()	Supported.	-
JSON_TYPE()	Supported, with differences	JSON values of the numeric type are identified as number, which is different from MySQL.
JSON_UNQUOTE()	Supported.	-
JSON_VALID()	Supported.	-

2.2.7 Aggregate Functions

Table 2-16 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.

2.2.8 Arithmetic Functions

Table 2-17 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.

2.2.9 Other Functions

Table 2-18 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 MYSQL-compatible mode is the native GaussDB behavior by default.

Table 2-19 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 are converted to the bytea type. The REGEXP operator does not support the bytea type. Therefore, the two types cannot be matched.
[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 2-20 Character sets

MySQL	GaussDB
utf8mb4	Supported
gbk	Supported
gb18030	Supported

NOTE

Currently, GaussDB does not perform strict encoding logic verification on invalid characters that do not belong to the current character set. As a result, such invalid characters may be successfully entered. However, an error is reported during verification in MySQL.

2.5 Collation Rules

GaussDB allows you to specify the following collation rules for 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 can be specified only when the corresponding character set is the same as the database-level character set.
- The default collation of the utf8mb4 character set is utf8mb4_general_ci, which is the same as that in MySQL 5.7. utf8mb4_0900_ai_ci is not the default collation of utf8mb4.
- In GaussDB, utf8 and utf8mb4 are the same character set.

Table 2-21 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 SQL

2.6.1 DDL

Table 2-22 DDL syntax compatibility

Description	Syntax	Difference
Create primary keys and UNIQUE indexes 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. However, MySQL supports these functions. ● 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.

Description	Syntax	Difference
Support prefix indexes.	CREATE INDEX	● 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 and unique key indexes.
Specify character sets and collation rules.	ALTER SCHEMA, ALTER TABLE, CREATE SCHEMA, and CREATE TABLE	-
Create a partitioned table.	CREATE TABLE PARTITION	-
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	-

Description	Syntax	Difference
Exchange the partition data of an ordinary table and a partitioned table.	ALTER TABLE PARTITION	Differences in ALTER TABLE EXCHANGE PARTITION: ● 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 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.

Description	Syntax	Difference
		● 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
Support auto-increment columns.	ALTER TABLE and CREATE TABLE	● Currently, only local auto-increment columns of each DN are supported. ● 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

Description	Syntax	Difference
		similar to that when the MySQL system variable *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. ● In different execution plans, the auto-increment sequence and reserved auto-increment values may be different from those in MySQL. For example, "INSERT INTO table VALUES(...),(...),..." is distributed to different DNs. Therefore, in some execution plans, DNs cannot obtain the number of rows to be inserted. 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-

Description	Syntax	Difference
		increment value generated by parallel insertion cannot be guaranteed to be the same as that generated in MySQL. ● 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

Description	Syntax	Difference
		statistics may not be the same as that in MySQL. ● 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. ● The last_insert_id function is not supported. ● Currently, local temporary tables do not support auto-increment columns. ● 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.
Delete the primary key constraints of a table.	ALTER TABLE	-

Description	Syntax	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. ● For the set data type, MySQL supports replication while GaussDB does not during table creation. ● 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

Description	Syntax	Difference
		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. MySQL fails to create the table because it is controlled by the strict mode. ● MySQL supports cross-database table creation, but GaussDB does not. ● If the source table is a temporary table, you can create a non-temporary table in MySQL but not in GaussDB.

Description	Syntax	Difference

Compatible with syntax for changing table names.	ALTER TABLE[IF EXISTS] 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.

Description	Syntax	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 ALTER TABLE table_name ADD PARTITION (partition_definition1, partition_definition1,…); syntax cannot be used to add multiple partitions. • Only the original syntax for adding multiple partitions is supported: ALTER TABLE table_name ADD PARTITION (partition_definition1), ADD PARTITION (partition_definition2[y1]), …;.

2.6.2 DML

Table 2-23 DML syntax compatibility

Description	Syntax	Difference
DELETE supports ORDER BY and LIMIT.	DELETE	-
UPDATE supports ORDER BY and LIMIT.	UPDATE	-

Description	Syntax	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)`
		`DISTRIBUTE BY HASH(f1);`
		`CREATE TABLE`
		`gaussdb=# REPLACE INTO`
		`test VALUES(f1, f2, f3);`
		`REPLACE 0 1`
		`gaussdb=# SELECT * FROM`
		`test;`
		`f1          |          f2          |      f3`
		`---------------------`
		`+---------------------+------------`

Description	Syntax	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 type when NOT NULL exists. 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 int, f2 BIT(3) NOT NULL) DISTRIBUTE BY HASH(f1);`
		`CREATE TABLE`
		`gaussdb=# REPLACE INTO test VALUES(1, f2);`
		`ERROR:  null value in column "f2" violates not-null constraint`
		`DETAIL:  Failing row contains (1, null).`

SELECT supports multi-partition query.	SELECT	-

Description	Syntax	Difference
UPDATE supports multi-partition update.	UPDATE	-

Description	Syntax	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

Description	Syntax	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. ● Columns can only be specified by column name instead of user variables. ● The LOAD DATA SET expression does not support the calculation of a specified column name. ● If no implicit conversion function exists between the return value type of the SET expression and the corresponding column type, an error is reported. ● LOAD DATA does not support the INSERT or DELETE trigger. ● 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.

Description	Syntax	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. For example:
		● 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 DATETIME NOT NULL, f3 TIMESTAMP NOT NULL);`

Description	Syntax	Difference

		```
		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
		column f1.
		gaussdb=# INSERT IGNORE
		INTO test VALUES('1010');
		```

Description	Syntax	Difference

		`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`
		`----------+---------------------+---------------------`
		` 00:00:00 | 1970-01-01 00:00:00 | 1970-01-01 00:00:00`
		`(1 row)`

Description	Syntax	Difference

		– 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`
		`DETAIL:  Failing row contains (2, null).`
		`WARNING:  division by zero`
		`CONTEXT:  referenced column: f2`
		`WARNING:  null value in column "f2" violates not-null`

Description	Syntax	Difference

		```
		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 $$
		gaussdb$# BEGIN
		gaussdb$# INSERT IGNORE INTO test1 VALUES(NULL);
		gaussdb$# RETURN NEW;
		gaussdb$# END;
		gaussdb$# $$ LANGUAGE plpgsql;
		CREATE FUNCTION
		gaussdb=# CREATE TRIGGER trig2 BEFORE INSERT ON test2 FOR EACH ROW EXECUTE PROCEDURE
		```

Description	Syntax	Difference

		```
		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
		'f1' cannot be null
		mysql> INSERT IGNORE INTO
		test2 VALUES(NULL);
		Query OK, 1 row affected
		(0.00 sec)

		mysql> SELECT * FROM test1;
		+----+
		| f1 |
		```

Description	Syntax	Difference

		```
		+----+
		|  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> 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)
		```

Description	Syntax	Difference
		```
		mysql> SELECT * FROM test;
		+----+----+
		| f1 | f2 |
		+----+----+
		|  1 |  0 |
		+----+----+
		1 row in set (0.00 sec)
		```

2.6.3 DCL

Table 2-24 DCL syntax compatibility

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*.

2.7 Drivers

2.7.1 JDBC

2.7.1.1 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 MySQL-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 2-25 describes the common methods for obtaining data. If you want to know more about other methods, see JDK official documents.

Table 2-25 Common methods for obtaining data from a result set

Method	Description	Difference
int getInt(int columnIndex)	Obtains int data by column index.	-

Method	Description	Difference
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.
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, the data type precision, scale, and number of bits in M-compatible mode of MySQL compatibility cannot be defined as floating-point values by default. You are advised to use valid integer values.

Table 3-1 Integer types

MySQL	GaussDB	Difference
BOOL	Supported, with differences	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	Supported, with differences	

MySQL	GaussDB	Difference
TINYINT[(M)] [UNSIGNED] [ZEROFILL]	Supported, with differences	Input format: ● MySQL: If a character string with multiple decimal points (such as "1.2.3.4.5") is entered, MySQL will misparse the character string in loose mode, throw a warning, and insert the character string into the table successfully. For example, after "1.2.3.4.5" is inserted into the table, the value is **12**. ● GaussDB: If a character string with multiple decimal points (such as "1.2.3.4.5") is entered in loose mode, the characters after the second decimal point are truncated as invalid characters, a warning is thrown, and the character string is inserted into the table successfully. For example, after "1.2.3.4.5" is inserted into the table, the value is **1**. After "1.6.3.4.5" is inserted into the table, the value is **2**.
SMALLINT[(M)] [UNSIGNED] [ZEROFILL]	Supported, with differences	
MEDIUMINT[(M)] [UNSIGNED] [ZEROFILL]	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 is mapped to the INT type. Four bytes are required for storage. The value range is determined based on boundary values. ● The signed range is –8388608 to +8388607. ● The unsigned range is 0 to +16777215. For other differences, see the description below the table.

MySQL	GaussDB	Difference
INT[(M)] [UNSIGNED] [ZEROFILL]	Supported, with differences	Input format: ● MySQL: If a character string with multiple decimal points (such as "1.2.3.4.5") is entered, MySQL will misparse the character string in loose mode, throw a warning, and insert the character string into the table successfully. For example, after "1.2.3.4.5" is inserted into the table, the value is **12**. ● GaussDB: If a character string with multiple decimal points (such as "1.2.3.4.5") is entered in loose mode, the characters after the second decimal point are truncated as invalid characters, a warning is thrown, and the character string is inserted into the table successfully. For example, after "1.2.3.4.5" is inserted into the table, the value is **1**. After "1.6.3.4.5" is inserted into the table, the value is **2**.
INTEGER[(M)] [UNSIGNED] [ZEROFILL]	Supported, with differences	
BIGINT[(M)] [UNSIGNED] [ZEROFILL]	Supported, with differences	

Table 3-2 Arbitrary precision types

MySQL	GaussDB	Difference
DECIMAL[(M[,D])] [ZEROFILL]	Supported, with differences	MySQL DECIMAL uses a 9 x 9 array to store values. The integer part and decimal part are stored separately. If the length exceeds the value, the decimal part is truncated first. GaussDB truncates an integer that contains more than 81 digits.
NUMERIC[(M[,D])] [ZEROFILL]	Supported, with differences	
DEC[(M[,D])] [ZEROFILL]	Supported, with differences	
FIXED[(M[,D])] [ZEROFILL]	Supported, with differences	

Table 3-3 Floating-point types

MySQL	GaussDB	Difference
FLOAT[(M,D)] [ZEROFILL]	Supported, with differences	The FLOAT data type does not support partitioned tables with the key partitioning policy.

MySQL	GaussDB	Difference
FLOAT(p) [ZEROFILL]	Supported, with differences	The FLOAT data type does not support partitioned tables with the key partitioning policy.
DOUBLE[(M, D)] [ZEROFILL]	Supported, with differences	The DOUBLE data type does not support partitioned tables with the key partitioning policy.
DOUBLE PRECISION[(M,D)] [ZEROFILL]	Supported, with differences	The DOUBLE PRECISION data type does not support partitioned tables with the key partitioning policy.
REAL[(M,D)] [ZEROFILL]	Supported, with differences	The REAL data type does not support partitioned tables with the key partitioning policy.

3.1.2 Date and Time Data Types

Table 3-4 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: A backslash (\) is regarded as an escape character in both MySQL and GaussDB. However, MySQL supports \0, but GaussDB does not support \0. Therefore, GaussDB reports an error when the backslash is used as a separator and the separator is followed by 0.
DATETIME[(fsp)]	Supported, with differences.	GaussDB supports the datetime data type. Compared with MySQL, GaussDB has the following differences in specifications: A backslash (\) is regarded as an escape character in both MySQL and GaussDB. However, MySQL supports \0, but GaussDB does not support \0. Therefore, GaussDB reports an error when the backslash is used as a separator and the separator is followed by 0.

MySQL	GaussDB	Difference
TIMESTAMP[(fsp)]	Supported, with differences.	GaussDB supports the timestamp data type. Compared with MySQL, GaussDB has the following differences in specifications: ● A backslash (\) is regarded as an escape character in both MySQL and GaussDB. However, MySQL supports \0, but GaussDB does not support \0. Therefore, GaussDB reports an error when the backslash is used as a separator and the separator is followed by 0. ● 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**.
TIME[(fsp)]	Supported, with differences.	GaussDB supports the time data type. Compared with MySQL, GaussDB has the following differences in specifications: ● A backslash (\) is regarded as an escape character in both MySQL and GaussDB. However, MySQL supports \0, but GaussDB does not support \0. Therefore, GaussDB reports an error when the backslash is used as a separator and the separator is followed by 0. ● When the hour, minute, second, and nanosecond of the time type are 0, the sign bits of GaussDB and MySQL may be different.
YEAR[(4)]	Supported.	-

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

MySQL	GaussDB	Difference
CHAR(M)	Supported, with differences	Input format: If a binary or hexadecimal character string is input, GaussDB outputs a hexadecimal character string, and MySQL escapes the character string based on the ASCII code table. If the character string cannot be escaped, the output is empty.
VARCHAR(M)	Supported, with differences	Input formats: ● 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. ● After a binary or hexadecimal character string is entered, GaussDB outputs a hexadecimal character string, and MySQL escapes the character string based on the ASCII code table. If the character string cannot be escaped, the output is empty.

MySQL	GaussDB	Difference
TINYTEXT	Supported, with differences	● Input formats: – Default value: When creating a table column, you can set a default value in the syntax. MySQL does not allow you to set a default value. – After a binary or hexadecimal character string is entered, GaussDB outputs a hexadecimal character string, and MySQL escapes the character string based on the ASCII code table. If the character string cannot be escaped, the output is empty. ● Primary key: In MySQL, the TINYTEXT type does not support primary keys, but GaussDB supports. ● Index: In MySQL, the TINYTEXT type does not support other index methods except prefix indexes. GaussDB supports these index methods. ● Foreign key: In MySQL, the TINYTEXT type cannot be used as the referencing column or referenced column of a foreign key, but GaussDB supports this operation.
TEXT	Supported, with differences	● Input formats: – Default value: When creating a table column, you can set a default value in the syntax. MySQL does not allow you to set a default value. – After a binary or hexadecimal character string is entered, GaussDB outputs a hexadecimal character string, and MySQL escapes the character string based on the ASCII code table. If the character string cannot be escaped, the output is empty. ● Primary key: In MySQL, the TEXT type does not support primary keys, but GaussDB supports. ● Index: In MySQL, the TEXT type does not support other index methods except prefix indexes. GaussDB supports these index methods. ● Foreign key: In MySQL, the TINYTEXT type cannot be used as the referencing column or referenced column of a foreign key, but GaussDB supports this operation.

MySQL	GaussDB	Difference
MEDIUMTEXT	Supported, with differences	● Input formats: – Default value: When creating a table column, you can set a default value in the syntax. MySQL does not allow you to set a default value. – After a binary or hexadecimal character string is entered, GaussDB outputs a hexadecimal character string, and MySQL escapes the character string based on the ASCII code table. If the character string cannot be escaped, the output is empty. ● Primary key: In MySQL, the MEDIUMTEXT type does not support primary keys, but GaussDB supports. ● Index: In MySQL, the MEDIUMTEXT type does not support other index methods except prefix indexes. GaussDB supports these index methods. ● Foreign key: In MySQL, the TINYTEXT type cannot be used as the referencing column or referenced column of a foreign key, but GaussDB supports this operation.

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. – Default value: When creating a table column, you can set a default value in the syntax. MySQL does not allow you to set a default value. – After a binary or hexadecimal character string is entered, GaussDB outputs a hexadecimal character string, and MySQL escapes the character string based on the ASCII code table. If the character string cannot be escaped, the output is empty. ● Primary key: In MySQL, the LONGTEXT type does not support primary keys, but GaussDB supports. ● Index: In MySQL, the LONGTEXT type does not support other index methods except prefix indexes. GaussDB supports these index methods. ● Foreign key: In MySQL, the TINYTEXT type cannot be used as the referencing column or referenced column of a foreign key, but GaussDB supports this operation.

3.1.4 Binary Data Types

Table 3-6 Binary data types

MySQL	GaussDB	Difference
BINARY[(M)]	Supported, with differences	• Input format: – After a binary or hexadecimal character string is entered, GaussDB outputs a hexadecimal character string, and MySQL escapes the character string based on the ASCII code table. If the character string cannot be escaped, the output is empty. – 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. – In MySQL 8.0 and later versions, results starting with 0x are returned by default. In GaussDB, results in the format of "\x...\x...\x..." are returned. **NOTE** Due to the differences between GaussDB and MySQL in BINARY fillers and \0 truncation, 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 difference scenarios, see the examples in this section.

MySQL	GaussDB	Difference
VARBINARY(M)	Supported, with differences	• Input format: If a binary or hexadecimal character string is input, GaussDB outputs a hexadecimal character string, and MySQL escapes the character string based on the ASCII code table. If the character string cannot be escaped, the output is empty. • 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 MySQL 8.0 and later versions, results starting with 0x are returned by default. In GaussDB, results in the format of "\x...\x...\x..." are returned.
TINYBLOB	Supported, with differences	• Input formats: – Default value: When creating a table column, you can set a default value in the syntax. MySQL does not allow you to set a default value. – After a binary or hexadecimal character string is entered, GaussDB outputs a hexadecimal character string, and MySQL escapes the character string based on the ASCII code table. If the character string cannot be escaped, the output is empty. • Primary key: In MySQL, the TINYBLOB type does not support primary keys, but GaussDB supports. • Index: In MySQL, the TINYBLOB type does not support other index methods except prefix indexes. GaussDB supports these index methods. • Foreign key: In MySQL, the TINYTEXT type cannot be used as the referencing column or referenced column of a foreign key, but GaussDB supports this operation. • Output format: In MySQL 8.0 and later versions, results starting with 0x are returned by default. In GaussDB, results in the format of "\x...\x...\x..." are returned.

MySQL	GaussDB	Difference
BLOB	Supported, with differences	● Input formats: – Default value: When creating a table column, you can set a default value in the syntax. MySQL does not allow you to set a default value. – After a binary or hexadecimal character string is entered, GaussDB outputs a hexadecimal character string, and MySQL escapes the character string based on the ASCII code table. If the character string cannot be escaped, the output is empty. ● Primary key: In MySQL, the BLOB type does not support primary keys, but GaussDB supports. ● Index: In MySQL, the BLOB type does not support other index methods except prefix indexes. GaussDB supports these index methods. ● Foreign key: In MySQL, the TINYTEXT type cannot be used as the referencing column or referenced column of a foreign key, but GaussDB supports this operation. ● Output format: In MySQL 8.0 and later versions, results starting with 0x are returned by default. In GaussDB, results in the format of "\x...\x...\x..." are returned.

MySQL	GaussDB	Difference
MEDIUMBLOB	Supported, with differences	● Input formats: – Default value: When creating a table column, you can set a default value in the syntax. MySQL does not allow you to set a default value. – After a binary or hexadecimal character string is entered, GaussDB outputs a hexadecimal character string, and MySQL escapes the character string based on the ASCII code table. If the character string cannot be escaped, the output is empty. ● Primary key: In MySQL, the MEDIUMBLOB type does not support primary keys, but GaussDB supports. ● Index: In MySQL, the MEDIUMBLOB type does not support other index methods except prefix indexes. GaussDB supports these index methods. ● Foreign key: In MySQL, the TINYTEXT type cannot be used as the referencing column or referenced column of a foreign key, but GaussDB supports this operation. ● Output format: In MySQL 8.0 and later versions, results starting with 0x are returned by default. In GaussDB, results in the format of "\x...\x...\x..." are returned.

MySQL	GaussDB	Difference
LONGBLOB	Supported, with differences	● Value range: a maximum of 1 GB. MySQL supports a maximum of 4 GB minus 1 byte. ● Input format: – Default value: When creating a table column, you can set a default value in the syntax. MySQL does not allow you to set a default value. – After a binary or hexadecimal character string is entered, GaussDB outputs a hexadecimal character string, and MySQL escapes the character string based on the ASCII code table. If the character string cannot be escaped, the output is empty. ● Primary key: In MySQL, the LONGBLOB type does not support primary keys, but GaussDB supports. ● Index: In MySQL, the LONGBLOB type does not support other index methods except prefix indexes. GaussDB supports these index methods. ● Foreign key: In MySQL, the TINYTEXT type cannot be used as the referencing column or referenced column of a foreign key, but GaussDB supports this operation. ● Output format: In MySQL 8.0 and later versions, results starting with 0x are returned by default. In GaussDB, results in the format of "\x...\x...\x..." are returned.
BIT[(M)]	Supported, with differences	Output formats: ● All outputs are displayed as binary character strings. MySQL escapes the character string based on the ASCII code table. If the character string cannot be escaped, the output is empty. ● In MySQL 8.0 and later versions, 0 is added at the beginning of each result by default. In GaussDB, 0 is not added.

Example:

```
-- GaussDB
m_db=# CREATE TABLE test(a BINARY(10)) DISTRIBUTE BY REPLICATION;
CREATE TABLE
m_db=# INSERT INTO test VALUES(0x8000);
INSERT 0 1
m_db=# SELECT hex(a) FROM test;
        hex
----------------------
```

```
 80202020202020202020
(1 row)

m_db=# SELECT * FROM test WHERE hex(a) = 80000000000000000000;
 a
---
(0 rows)

m_db=# CREATE TABLE test2(a BINARY(10)) DISTRIBUTE BY REPLICATION;
CREATE TABLE
m_db=# INSERT INTO test2 VALUES(0x80008000);
INSERT 0 1
m_db=# SELECT hex(a) FROM test2;
         hex
----------------------
 80202020202020202020
(1 row)

m_db=# DROP TABLE test;
DROP TABLE
m_db=# DROP TABLE test2;
DROP TABLE

-- MySQL
mysql> CREATE TABLE test(a BINARY(10));
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO test VALUES(0x8000);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT hex(a) FROM test;
+----------------------+
| hex(a)               |
+----------------------+
| 80000000000000000000 |
+----------------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM test WHERE hex(a) = 80000000000000000000;
+------------+
| a          |
+------------+
| ▒          |
+------------+
1 row in set (0.00 sec)

mysql> CREATE TABLE test2(a binary(10));
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO test2 VALUES(0x80008000);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT hex(a) FROM test2;
+----------------------+
| hex(a)               |
+----------------------+
| 80008000000000000000 |
+----------------------+
1 row in set (0.00 sec)
mysql> DROP TABLE test;
Query OK, 0 rows affected (0.00 sec)
mysql> DROP TABLE test2;
Query OK, 0 rows affected (0.00 sec)
```

3.1.5 Attributes Supported by Data Types

Table 3-7 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.
ZEROFILL	Supported.

When CREATE TABLE AS is used to create a table and default values are set for columns of the VARBINARY type, the command output of **SHOW CREATE TABLE**, **DESC**, or **\d** is different from that of MySQL. The value displayed in GaussDB is a hexadecimal value, but MySQL displays the original value.

Example:

```
m_db=# CREATE TABLE test_int(
       int_col INT
);
m_db=# CREATE TABLE test_varbinary(
       varbinary_col VARBINARY(20) default 'gauss'
) AS SELECT * FROM test_int;
m_db=# SHOW CREATE TABLE test_varbinary;
    Table      |                          Create Table
----------------+---------------------------------------------------------------------------
 test_varbinary | SET search_path = public;                                             +
                | CREATE TABLE test_varbinary (                                         +
                |     varbinary_col varbinary(20) DEFAULT X'6761757373',                +
                |     int_col integer                                                   +
                | )                                                                     +
                | CHARACTER SET = "UTF8" COLLATE = "utf8mb4_general_ci"                 +
                | WITH (orientation=row, compression=no, storage_type=USTORE, segment=off);
(1 row)
m_db=# DROP TABLE test_int, test_varbinary;

mysql> CREATE TABLE test_int(
       int_col INT
);
mysql> CREATE TABLE test_varbinary(
       varbinary_col VARBINARY(20) default 'gauss'
) AS SELECT * FROM test_int;
mysql> SHOW CREATE TABLE test_varbinary;
+----------------
+-------------------------------------------------------------------------------------------------------------------------+
| Table          | Create
Table                                                                                                                     |
+----------------
```

```
+------------------------------------------------------------------------------------------------------------------------+
| test_varbinary | CREATE TABLE `test_varbinary` (
  `varbinary_col` varbinary(20) DEFAULT 'gauss',
  `int_vol` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 |
+----------------
+------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
mysql> DROP TABLE test_int, test_varbinary;
```

3.1.6 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 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.

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

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 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.
- Due to data type differences, some output formats of implicit conversion in GaussDB are inconsistent.
- During implicit conversion from the BIT data type to the character data type and binary data type in GaussDB, some output behaviors are inconsistent. GaussDB outputs a hexadecimal character string, and MySQL escapes the character string based on the ASCII code table. If the character string cannot be escaped, the output is empty.

  Example:

```
m_db=# CREATE TABLE bit_storage (
    VS_COL1 BIT(4),
    VS_COL2 BIT(4),
    VS_COL3 BIT(4),
    VS_COL4 BIT(4),
    VS_COL5 BIT(4),
    VS_COL6 BIT(4),
    VS_COL7 BIT(4),
    VS_COL8 BIT(4)
) DISTRIBUTE BY REPLICATION;
m_db=# CREATE TABLE string_storage (
    VS_COL1 BLOB,
    VS_COL2 TINYBLOB,
    VS_COL3 MEDIUMBLOB,
    VS_COL4 LONGBLOB,
    VS_COL5 TEXT,
    VS_COL6 TINYTEXT,
    VS_COL7 MEDIUMTEXT,
    VS_COL8 LONGTEXT
) DISTRIBUTE BY REPLICATION;
m_db=# INSERT INTO bit_storage VALUES(B'101', B'101', B'101', B'101', B'101', B'101', B'101', B'101');
m_db=# INSERT INTO string_storage SELECT * FROM bit_storage;
m_db=# SELECT * FROM string_storage;
 VS_COL1 | VS_COL2 | VS_COL3 | VS_COL4 | VS_COL5 | VS_COL6 | VS_COL7 | VS_COL8
---------+---------+---------+---------+---------+---------+---------+---------
 \x05    | \x05    | \x05    | \x05    | \x05    | \x05    | \x05    | \x05
(1 row)
m_db=# DROP TABLE bit_storage, string_storage;

mysql> CREATE TABLE bit_storage (
    VS_COL1 BIT(4),
    VS_COL2 BIT(4),
    VS_COL3 BIT(4),
    VS_COL4 BIT(4),
    VS_COL5 BIT(4),
    VS_COL6 BIT(4),
    VS_COL7 BIT(4),
    VS_COL8 BIT(4)
);
mysql> CREATE TABLE bit_storage (
    VS_COL1 BIT(4),
    VS_COL2 BIT(4),
    VS_COL3 BIT(4),
    VS_COL4 BIT(4),
    VS_COL5 BIT(4),
    VS_COL6 BIT(4),
    VS_COL7 BIT(4),
    VS_COL8 BIT(4)
);
mysql> INSERT INTO bit_storage VALUES(B'101', B'101', B'101', B'101', B'101', B'101', B'101', B'101');
mysql> INSERT INTO string_storage SELECT * FROM bit_storage;
mysql> SELECT * FROM string_storage;
```

```
+---------+---------+---------+---------+---------+---------+---------+---------+
| VS_COL1 | VS_COL2 | VS_COL3 | VS_COL4 | VS_COL5 | VS_COL6 | VS_COL7 | VS_COL8 |
+---------+---------+---------+---------+---------+---------+---------+---------+
|         |         |         |         |         |         |         |         |
+---------+---------+---------+---------+---------+---------+---------+---------+
1 row in set (0.00 sec)
mysql> DROP TABLE bit_storage, string_storage;
```

- When a binary or hexadecimal character string with 0x00 is inserted into the binary data type, GaussDB inserts part of the string and truncates the characters following 0x00. MySQL can insert the entire string.

  Example:

```
m_db=# CREATE TABLE blob_storage (
       A BLOB
) DISTRIBUTE BY REPLICATION;
m_db=# INSERT INTO blob_storage VALUES (0xBB00BB);
m_db=# SELECT hex(A) FROM blob_storage;
 hex
-----
 BB
(1 row)
m_db=# DROP TABLE blob_storage;

mysql> CREATE TABLE blob_storage (
       A BLOB
);
mysql> INSERT INTO blob_storage VALUES (0xBB00BB);
mysql> SELECT hex(A) FROM blob_storage;
+--------+
| hex(a) |
+--------+
| BB00BB |
+--------+
1 row in set (0.01 sec)
mysql> DROP TABLE blob_storage;
```

- When a binary or hexadecimal string with 0x00 in the middle is inserted into the string data type, GaussDB inserts part of the string and truncates the characters following 0x00. In MySQL, the string cannot be inserted in strict mode, and an empty string is inserted in loose mode.

  Example:

```
m_db=# CREATE TABLE text_storage (
       A TEXT
);
m_db=# INSERT INTO text_storage VALUES (b'101110110000000010111011');
m_db=# SELECT hex(A) FROM text_storage;
 hex
-----
 BB
(1 row)
m_db=# DROP TABLE text_storage;

mysql> CREATE TABLE text_storage (
       A TEXT
);
mysql> INSERT INTO text_storage VALUES (b'101110110000000010111011');
ERROR 1366 (HY000): Incorrect string value: '\xBB\x00\xBB' for column 'A' at row 1
mysql> SELECT hex(A) FROM text_storage;
Empty set (0.00 sec)
mysql> SET SQL_MODE='';
mysql> INSERT INTO text_storage VALUES (b'101110110000000010111011');
mysql> SELECT hex(A) FROM text_storage;
+--------+
| hex(A) |
+--------+
|        |
+--------+
1 row in set (0.01 sec)
mysql> DROP TABLE text_storage;
```

- The WHERE clause contains only common character strings. GaussDB returns **TRUE** for **'t'**, **'true'**, **'yes'**, **'y'**, and **'on'**, returns **FALSE** for **'no'**, **'f'**, **'off'**, **'false'**, and **'n'**, and reports an error for other character strings. MySQL determines whether to return **TRUE** or **FALSE** by converting a character string to an INT1 value.

  Example:

```
m_db=# CREATE TABLE test_where (
       A INT
);
m_db=# INSERT INTO test_where VALUES (1);
m_db=# SELECT * FROM test_where WHERE '111';
ERROR:  invalid input syntax for type boolean: "111"
LINE 1: SELECT * FROM test_where WHERE '111';
m_db=# DROP TABLE test_where;

mysql> CREATE TABLE test_where (
       A INT
);
mysql> INSERT INTO test_where VALUES (1);
mysql> SELECT * FROM test_where WHERE '111';
+------+
| a    |
+------+
|    1 |
+------+
1 row in set (0.01 sec)
mysql> DROP TABLE test_where;
```

- When converting strings of the YEAR type to integers, MySQL uses scientific notation, but GaussDB does not support scientific notation and truncates the strings.

  Example:

```
m_db=# CREATE TABLE test_year (
       A YEAR
);
m_db=# SET sql_mode = '';
m_db=# INSERT INTO test_year VALUES ('2E3x');
WARNING:  Data truncated for column.
LINE 1: INSERT INTO test_year VALUES ('2E3x');
                                      ^
CONTEXT:  referenced column: a
m_db=# SELECT * FROM test_year ORDER BY A;
  a
------
 2002
(1 row)
m_db=# DROP TABLE test_year;

mysql> CREATE TABLE test_year (
       A YEAR
);
mysql> INSERT INTO test_year VALUES ('2E3x');
mysql> SELECT * FROM test_year ORDER BY A;
+------+
| a    |
+------+
| 2000 |
+------+
1 row in set (0.01 sec)
mysql> DROP TABLE test_year;
```

Differences in Explicit Type Conversion

- In GaussDB, the conversion rules for each target type are used. In MySQL, C++ polymorphic overloading functions are used, causing inconsistent behavior in nesting scenarios.

  Example:
  ```
  m_db=# SELECT CAST(GREATEST(date'2023-01-01','2023-01-01') AS SIGNED);
  WARNING:  Truncated incorrect INTEGER value: '2023-01-01'
  CONTEXT:  referenced column: cast
   cast
  ------
   2023
  (1 row)

  mysql> SELECT CAST(GREATEST(date'2023-01-01','2023-01-01') AS SIGNED);
  +---------------------------------------------------------+
  | CAST(GREATEST(date'2023-01-01','2023-01-01') AS SIGNED) |
  +---------------------------------------------------------+
  |                                                20230101 |
  +---------------------------------------------------------+
  ```

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.
- 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.
- When fixed-point constants and types without precision constraints (non-string types such as int, bool, and year, and the type of the aggregation result is the fixed-point type) are aggregated, the precision constraint is output based on the default precision 31 of fixed-point numbers.
- Differences in merge rules:

  In MySQL 5.7, if YEAR is aggregated with TINYINT, INT, MEDIUMINT, BIGINT, or BOOL, the result is of the type with UNSIGNED. In GaussDB, it is of the type without UNSIGNED. In MySQL, if BIT is aggregated with a numeric type such as INT, NUMERIC, FLOAT, or DOUBLE, the result type is VARBINARY. In GaussDB, the result type is NUMERIC for aggregation between BIT and INT or NUMERIC, DOUBLE for aggregation between BIT and FLOAT or DOUBLE, and UINT8 for aggregation between BIT and unsigned integers.
- 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.

3.2 System Functions

3.2.1 System Function Compatibility Overview

GaussDB is compatible with most MySQL system functions, but there are some differences.

Currently, some system functions in GaussDB with the same names as those in MySQL are not supported in M-compatible mode. For some of them, the message indicating that they are not supported in M-compatible mode is displayed. Other functions still retain the behaviors 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.

The following table lists the functions with the same name.

Table 3-8 Same-name functions for which a message indicating that they are not supported in M-compatible mode is displayed

cot	isEmpty	last_insert_id	mod	octet_length
overlaps	point	radians	regexp_instr	regexp_like
regexp_replace	regexp_substr	stddev_pop	stddev_samp	var_pop
var_samp	variance	-	-	-

Table 3-9 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_num	-

NOTE

- MySQL allows you to add user-defined functions to the database through the loadable functions. When such functions are called, aliases can be specified in the input parameters of the functions. GaussDB does not support loadable functions. When a function is called, aliases cannot be specified for input parameters of the function.
- 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.
  - When an aggregate function uses an expression such as another function, operator, or SELECT clause as the input parameter (for example, **SELECT sum(abs(n)) FROM t;**), the aggregate function cannot obtain the precision transferred by the input parameter expression. As a result, the precision of the function result is different from that of MySQL.
  - 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 substr('demo' from 1 for 2)), an error may occur when the function is called.

3.2.2 Flow Control Functions

Table 3-10 Flow control functions

MySQL	GaussDB	Difference
IF()	Supported, with differences.	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.
IFNULL()	Supported, with differences.	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.
NULLIF()	Supported, with differences.	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 it is more appropriate.

3.2.3 Date and Time Functions

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

- Functions may use time expressions as their input parameters.

  Time expressions (mainly including TEXT, DATETIME, DATE, and TIME) and types that can be implicitly converted to time expressions can be used as 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, the time expression is parsed as the date type. The TIMESTAMPDIFF function parses the time expression 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 in 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 the DATE and DATETIME ranges supported by GaussDB are greater than those supported by MySQL, out-of-bounds dates are still 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.

Most date and time functions in the GaussDB M-compatible framework are the same as those in MySQL. The following table lists the differences between them in terms of some functions.

Table 3-11 Date and time functions

MySQL	GaussDB	Difference
ADDDATE()	Supported	-
ADDTIME()	Supported	-
CONVERT_TZ()	Supported	-
CURDATE()	Supported	-
CURRENT_DATE()/ CURRENT_DATE	Supported	-
CURRENT_TIME()/ CURRENT_TIME	Supported, with differences.	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	Supported, with differences.	In MySQL, an integer input value is wrapped when it reaches **255** (maximum value of a one-byte integer value), for example, **SELECT CURRENT_TIMESTAMP(257) == SELECT CURRENT_TIMESTAMP(1)**. GaussDB supports only valid values ranging from 0 to 6. For other values, an error is reported.
CURTIME()	Supported, with differences.	In MySQL, an integer input value is wrapped when it reaches **255** (maximum value of a one-byte integer value), for example, **SELECT CURTIME(257) == SELECT CURTIME(1)**. GaussDB supports only valid values ranging from 0 to 6. For other values, an error is reported.
DATE()	Supported	-
DATE_ADD()	Supported	-
DATE_FORMAT()	Supported	-

MySQL	GaussDB	Difference
DATE_SUB()	Supported	-
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	-
LOCALTIME()/LOCALTIME	Supported, with differences.	In MySQL, an integer input value is wrapped when it reaches **255** (maximum value of a one-byte integer value), for example, **SELECT LOCALTIME(257) == SELECT LOCALTIME(1)**. GaussDB supports only valid values ranging from 0 to 6. For other values, an error is reported.
LOCALTIMESTAMP/ LOCALTIMESTAMP()	Supported, with differences.	In MySQL, an integer input value is wrapped when it reaches **255** (maximum value of a one-byte integer value), for example, **SELECT LOCALTIMESTAMP(257) == SELECT LOCALTIMESTAMP(1)**. GaussDB supports only valid values ranging from 0 to 6. For other values, an error is reported.
MAKEDATE()	Supported	-
MAKETIME()	Supported, with differences.	In the distributed pushdown scenario, if no second precision is specified for the TIME type, MySQL supplements six trailing zeros by default, but GaussDB does not supplement anything.

MySQL	GaussDB	Difference
MICROSECOND()	Supported	-
MINUTE()	Supported	-
MONTH()	Supported	-
MONTHNAME()	Supported	-
NOW()	Supported, with differences.	In MySQL, an integer input value is wrapped when it reaches **255** (maximum value of a one-byte integer value), for example, **SELECT NOW(257)==SELECT NOW(1)**. GaussDB supports only valid values ranging from 0 to 6. For other values, an error is reported.
PERIOD_ADD()	Supported, with differences.	● Processing of 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 **month_number** value in the input parameter or result exceed the uint32 range, integer wraparound occurs. This issue has been resolved in MySQL 8.0. The performance of this function in GaussDB is the same as that in MySQL 8.0. ● 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). This issue has been resolved in MySQL 8.0. The performance of this function in GaussDB is the same as that in MySQL 8.0. ● 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. GaussDB reports an error for months beyond the range. This issue has been resolved in MySQL 8.0. The performance of this function in GaussDB is the same as that in MySQL 8.0.

MySQL	GaussDB	Difference
PERIOD_DIFF()	Supported, with differences.	● Processing of 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 **month_number** value in the input parameter or result exceed the uint32 range, integer wraparound occurs. This issue has been resolved in MySQL 8.0. The performance of this function in GaussDB is the same as that in MySQL 8.0. ● 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). This issue has been resolved in MySQL 8.0. The performance of this function in GaussDB is the same as that in MySQL 8.0. ● 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. GaussDB reports an error for months beyond the range. This issue has been resolved in MySQL 8.0. The performance of this function in GaussDB is the same as that in MySQL 8.0.
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	-
SUBTIME()	Supported	-
SYSDATE()	Supported, with differences.	In MySQL, an integer input value is wrapped when it reaches **255** (maximum value of a one-byte integer value). GaussDB does not support wraparound.

MySQL	GaussDB	Difference
TIME()	Supported	-
TIME_FORMAT()	Supported	-
TIME_TO_SEC()	Supported	-
TIMEDIFF()	Supported	-
TIMESTAMP()	Supported	-
TIMESTAMPADD()	Supported	-
TIMESTAMPDIFF()	Supported	-
TO_DAYS()	Supported	-
TO_SECONDS()	Supported	-
UNIX_TIMESTAMP()	Supported, with differences.	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()	Supported	-
UTC_TIME()	Supported, with differences.	In MySQL, an integer input value is wrapped when it reaches **255** (maximum value of a one-byte integer value). GaussDB supports only valid values ranging from 0 to 6. For other values, an error is reported.
UTC_TIMESTAMP()	Supported, with differences.	In MySQL, an integer input value is wrapped when it reaches **255** (maximum value of a one-byte integer value). GaussDB supports only valid values ranging from 0 to 6. For other values, an error is reported.
WEEK()	Supported	-
WEEKDAY()	Supported	-
WEEKOFYEAR()	Supported	-

MySQL	GaussDB	Difference
YEAR()	Supported	-
YEARWEEK()	Supported	-

3.2.4 String Functions

Table 3-12 String functions

MySQL	GaussDB	Difference
ASCII()	Supported.	-
BIT_LENGTH()	Supported.	-
CHAR_LENGTH()	Supported, with differences.	In GaussDB, if the character set is SQL_ASCII, CHAR_LENGTH() returns the number of bytes instead of characters.
CHARACTER_LENGTH()	Supported, with differences.	In GaussDB, if the character set is SQL_ASCII, CHARACTER_LENGTH() returns the number of bytes instead of characters.
CONCAT()	Supported, with differences.	For binary return values, MySQL offers various options (including BINARY, VARBINARY, and BLOB), while GaussDB offers only one—LONGBLOB. For non-binary return values, MySQL offers various options (including CHAR, VARCHAR, and TEXT), while GaussDB only offers TEXT.
CONCAT_WS()	Supported, with differences.	For binary return values, MySQL offers various options (including BINARY, VARBINARY, and BLOB), while GaussDB offers only one—LONGBLOB. For non-binary return values, MySQL offers various options (including CHAR, VARCHAR, and TEXT), while GaussDB only offers TEXT.
HEX()	Supported.	-
LENGTH()	Supported.	-

MySQL	GaussDB	Difference
LPAD()	Supported, with differences.	• 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**. • 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. • For binary return values, MySQL offers various options (including BINARY, VARBINARY, and BLOB), while GaussDB offers only one—LONGBLOB. For non-binary return values, MySQL offers various options (including CHAR, VARCHAR, and TEXT), while GaussDB only offers TEXT.
REPEAT()	Supported, with differences.	For binary return values, MySQL offers various options (including BINARY, VARBINARY, and BLOB), while GaussDB offers only one—LONGBLOB. For non-binary return values, MySQL offers various options (including CHAR, VARCHAR, and TEXT), while GaussDB only offers TEXT.
REPLACE()	Supported, with differences.	For binary return values, MySQL offers various options (including BINARY, VARBINARY, and BLOB), while GaussDB offers only one—LONGBLOB. For non-binary return values, MySQL offers various options (including CHAR, VARCHAR, and TEXT), while GaussDB only offers TEXT.

MySQL	GaussDB	Difference
RPAD()	Supported, with differences.	● 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**. ● 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. ● For binary return values, MySQL offers various options (including BINARY, VARBINARY, and BLOB), while GaussDB offers only one—LONGBLOB. For non-binary return values, MySQL offers various options (including CHAR, VARCHAR, and TEXT), while GaussDB only offers TEXT.
SPACE()	Supported.	-
STRCMP()	Supported, with differences.	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.

MySQL	GaussDB	Difference
FIND_IN_SET()	Supported, with differences.	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. For binary return values, MySQL offers various options (including BINARY, VARBINARY, and BLOB), while GaussDB offers only one—LONGBLOB. For non-binary return values, MySQL offers various options (including CHAR, VARCHAR, and TEXT), while GaussDB only offers TEXT.
LCASE()		
LEFT()		
LOWER()		
LTRIM()		
REVERSE()		
RIGHT()		
RTRIM()		
SUBSTR()		
SUBSTRING()		
SUBSTRING_INDEX()		
TRIM()		
UCASE()		
UPPER()		
UNHEX()	Supported, with differences.	The return value type in MySQL is BINARY, VARBINARY, BLOB, MEDIUMBLOB, or LONGBLOB, while the return value type in GaussDB is fixed to LONGBLOB.
FIELD()	Supported.	-
FORMAT()	Supported.	-

3.2.5 Forced Conversion Functions

Table 3-13 Forced conversion functions

MySQL	GaussDB	Difference

CAST()	Supported, with differences	• In GaussDB, CAST(expr AS CHAR[(N)] charset_info or CAST(expr AS NCHAR[(N)]) cannot be used to convert character sets.
		• 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 GaussDB, CAST(expr AS JSON) cannot be used to convert expressions to JSON.
		• 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) DISTRIBUTE BY REPLICATION;`
		`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)`
		`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)  |`
		`+------------------------------------------------------+`

MySQL	GaussDB	Difference

		`| 1.2300000190734863 |`
		`+-----------------------------------------------------+`
		`1 row in set (0.00 sec)`

CONVERT()	Supported, with differences	● In GaussDB, CONVERT(expr, CHAR[(N)] charset_info or CAST(expr, NCHAR[(N)]) cannot be used to convert character sets. ● 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. ● In GaussDB, CONVERT(expr, JSON) cannot be used to convert expressions to JSON.

3.2.6 Encryption Functions

Table 3-14 Encryption functions

MySQL	GaussDB	Difference
AES_DECRYPT()	Supported, with differences	● 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 return value type in MySQL is BINARY, VARBINARY, BLOB, MEDIUMBLOB, or LONGBLOB, while the return value type in GaussDB is fixed to LONGBLOB.
AES_ENCRYPT()	Supported, with differences	
SHA()/ SHA1()	Supported.	-
SHA2()	Supported.	-

3.2.7 Comparison Functions

Table 3-15 Comparison functions

MySQL	GaussDB	Difference
COALESCE()	Supported, with differences.	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()	Supported.	-
GREATEST()	Supported, with differences.	For binary return values, MySQL offers various options (including BINARY, VARBINARY, and BLOB), while GaussDB offers only one—LONGBLOB. For non-binary return values, MySQL offers various options (including CHAR, VARCHAR, and TEXT), while GaussDB only offers TEXT. 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()	Supported, with differences.	For binary return values, MySQL offers various options (including BINARY, VARBINARY, and BLOB), while GaussDB offers only one—LONGBLOB. For non-binary return values, MySQL offers various options (including CHAR, VARCHAR, and TEXT), while GaussDB only offers TEXT. 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.
ISNULL()	Supported.	-

3.2.8 Aggregate Functions

Table 3-16 Aggregate functions

MySQL	GaussDB	Difference
AVG()	Supported, with differences.	● If **DISTINCT** is specified and the SQL statement contains the GROUP BY clause, GaussDB does not sort the results, while MySQL sorts the results. ● 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.
BIT_AND()	Supported.	-
BIT_OR()	Supported.	-
BIT_XOR()	Supported.	-
COUNT()	Supported, with differences.	If **DISTINCT** is specified and the SQL statement contains the GROUP BY clause, GaussDB does not sort the results, while MySQL sorts the results.
GROUP_CONCAT()	Supported, with differences.	● If **DISTINCT** is specified and the SQL statement contains the GROUP BY clause, GaussDB does not sort the results, while MySQL sorts the results. ● 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.

MySQL	GaussDB	Difference
MAX()	Supported, with differences.	If **DISTINCT** is specified and the SQL statement contains the GROUP BY clause, GaussDB does not sort the results, while MySQL sorts the results. When the parameter is not a table column, the return value type of the MAX function is different from that of MySQL 5.7.
MIN()	Supported, with differences.	If **DISTINCT** is specified and the SQL statement contains the GROUP BY clause, GaussDB does not sort the results, while MySQL sorts the results. When the parameter is not a table column, the return value type of the MIN function is different from that of MySQL 5.7.
SUM()	Supported, with differences.	● If **DISTINCT** is specified and the SQL statement contains the GROUP BY clause, GaussDB does not sort the results, while MySQL sorts the results. ● 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.

3.2.9 Arithmetic Functions

Table 3-17 Arithmetic functions

MySQL	GaussDB	Difference
ABS()	Supported.	-
ACOS()	Supported.	-
ASIN()	Supported.	-
ATAN()	Supported.	-
ATAN2()	Supported.	-
CEILING()	Supported, with differences.	Some operation result types are inconsistent with those in MySQL. If the derived result is of the NUMERIC or integer type and can be stored as an integer type, the result type in MySQL is integer, but is still NUMERIC in GaussDB.
COS()	Supported.	-
DEGREES()	Supported.	-
EXP()	Supported.	-

MySQL	GaussDB	Difference
FLOOR()	Supported, with differences.	The return value types of the FLOOR function in GaussDB are different from those in MySQL. When the input parameter type is INT, the return value type is BIGINT in GaussDB, but is INT in MySQL. Some operation result types are inconsistent with those in MySQL. If the derived result is of the NUMERIC or integer type and can be stored as an integer type, the result type in MySQL is integer, but is still NUMERIC in GaussDB.
LN()	Supported.	-
LOG()	Supported.	-
LOG10()	Supported.	-
LOG2()	Supported.	-
PI()	Supported, with differences.	The precision of the return value of the PI function in GaussDB is different from that in MySQL. It is rounded off to 15 decimal places in GaussDB but to six decimal places in MySQL.
POW()	Supported.	-
POWER()	Supported.	-
RAND()	Supported.	-
SIGN()	Supported.	-
SIN()	Supported.	-
SQRT()	Supported.	-
TAN()	Supported.	-
TRUNCATE()	Supported.	-
CEIL()	Supported.	-

3.2.10 Other Functions

Table 3-18 Other functions

MySQL	GaussDB	Difference
DATABASE()	Supported.	-
UUID()	Supported.	-

MySQL	GaussDB	Difference
UUID_SHORT()	Supported.	-

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 MySQL compatibility, 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 of GaussDB is the same as that of MySQL. Without ORDER BY, GaussDB does not guarantee that the results are ordered.
- 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 M-compatible 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 3-19 Operators

MySQL	GaussDB	Difference
<>	Supported, with differences.	MySQL supports indexes, but GaussDB does not.
<=>	Supported, with differences.	MySQL supports indexes, but GaussDB does not support indexes, hash joins, or merge joins.

MySQL	GaussDB	Difference

Row expressions	Supported, with differences.	● 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, but 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.
		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`
		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;`
		`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	GaussDB	Difference

		`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)`

--	Supported, with differences.	MySQL indicates that an operand is negated twice and the result is equal to the original operand. GaussDB indicates a comment.
!!	Supported, with differences.	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 (! !).

MySQL	GaussDB	Difference
[NOT] REGEXP	Supported, with differences.	• 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 will be converted to the bytea type. However, the REGEXP operator does not support the bytea type. Therefore, the matching will fail.

MySQL	GaussDB	Difference
LIKE	Supported, with differences.	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.
[NOT] BETWEEN AND	Supported, with differences.	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	Supported, with differences.	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.
!	Supported, with differences.	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.
#	Not supported.	MySQL supports the comment tag (#), but GaussDB does not.
BINARY	Supported, with differences.	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.
Negation (-)	Supported, with differences.	If the number of consecutive negation times exceeds 1, GaussDB identifies the negations as comments. As a result, it returns results different from MySQL.

MySQL	GaussDB	Difference

XOR, |, & , < , > , <=, >=, =, and !=	Supported, with differences.	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) |`

MySQL	GaussDB	Difference

		`(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)`

Table 3-20 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.
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.

Example of Operator Combination	MySQL	GaussDB	Description
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.
- For fuzzy match (LIKE operator), the default index created can be used in MySQL, but cannot be used in GaussDB. You need to use the following syntax to specify **opclass** to, for example, **text_pattern_ops**, so that LIKE operators can be used as indexes:
```
CREATE INDEX indexname ON tablename(col [opclass]);
```
- 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 3-21**. 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);
```

Table 3-21 Differences in index support

Index Column Type	Constant Type	Supported by GaussDB	Supported by MySQL
Integer	Integer	Yes	Yes
Floating-point	Floating-point	Yes	Yes
Fixed-point	Fixed-point	Yes	Yes
String	String	Yes	Yes
Binary	Binary	Yes	Yes
Time with date	Time with date	Yes	Yes

Index Column Type	Constant Type	Supported by GaussDB	Supported by MySQL
TIME	TIME	Yes	Yes
Time with date	Type that can be converted to time type with date (for example, integers such as 20231130)	Yes	Yes
Time with date	TIME	Yes	Yes
TIME	Constants that can be converted to the TIME type (for example, integers such as 203008)	Yes	Yes
Floating-point	Integer	Yes	Yes
Floating-point	Fixed-point	Yes	Yes
Floating-point	String	Yes	Yes
Floating-point	Binary	Yes	Yes
Floating-point	Time with date	Yes	Yes
Floating-point	TIME	Yes	Yes
Fixed-point	Integer	Yes	Yes
String	Time with date	Yes	No
String	TIME	Yes	No
Binary	String	Yes	Yes
Binary	Time with date	Yes	No
Binary	TIME	Yes	No
Integer	Floating-point	No	Yes
Integer	Fixed-point	No	Yes
Integer	String	No	Yes
Integer	Binary	No	Yes
Integer	Time with date	No	Yes
Integer	TIME	No	Yes
Fixed-point	Floating-point	No	Yes

Index Column Type	Constant Type	Supported by GaussDB	Supported by MySQL
Fixed-point	String	No	Yes
Fixed-point	Binary	No	Yes
Fixed-point	Time with date	No	Yes
Fixed-point	TIME	No	Yes
String	Binary	No	Yes
Time with date	Integer (that cannot be converted to the time type with date)	No	Yes
Time with date	Floating-point (that cannot be converted to the time type with date)	No	Yes
Time with date	Fixed-point (that cannot be converted to the time type with date)	No	Yes
TIME	Integer (that cannot be converted to the TIME type)	No	Yes
TIME	Character string (that cannot be converted to the TIME type)	No	Yes
TIME	Binary (that cannot be converted to the TIME type)	No	Yes
TIME	Time with date	No	Yes
YEAR	YEAR	Yes	Yes
YEAR	Constants that can be converted to the YEAR type (for example, integers such as 2034)	Yes	Yes

Index Column Type	Constant Type	Supported by GaussDB	Supported by MySQL
BIT	BIT	No	Yes

Table 3-22 Whether index use is supported

Index Column Type	Constant Type	Use Index or Not in GaussDB	Use Index or Not in MySQL
String	Integer	No	No
String	Floating-point	No	No
String	Fixed-point	No	No
Binary	Integer	No	No
Binary	Floating-point	No	No
Binary	Fixed-point	No	No
Time with date	Character string (that cannot be converted to the time type with date)	No	No
Time with date	Binary (that cannot be converted to the time type with date)	No	No
TIME	Floating-point (that cannot be converted to the TIME type)	No	No
TIME	Fixed-point (that cannot be converted to the TIME type)	No	No
BIT	String	No	No

3.4 Character Sets

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

Table 3-23 Character sets

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

NOTE

- **utf8** and **utf8mb4** refer to the same character set in GaussDB. The maximum code length 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 byte streams. If a multi-byte character contains code that is consistent with symbols such as '\', '\'', and '\\', the behavior of the multi-byte character is inconsistent with that in MySQL. In this case, you are advised to disable the escape character function temporarily.

3.5 Collation Rules

GaussDB allows you to specify the following collation rules for 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.

Table 3-24 Collation rules

MySQL	GaussDB
utf8mb4_general_ci	Supported.
utf8mb4_unicode_ci	Supported.

MySQL	GaussDB
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.

3.6 Transactions

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

Default Transaction Isolation Levels

The default isolation level of an M-compatible database is READ COMMITTED, and that of MySQL is REPEATABLE READ.

```
-- View the current transaction isolation level.
m_db=# SHOW transaction_isolation;
```

Sub-transactions

In an M-compatible database, SAVEPOINT is used to create a savepoint (sub-transaction) in the current transaction, and ROLLBACK TO SAVEPOINT is used to roll back to a savepoint (sub-transaction). After the sub-transaction is rolled back, the parent transaction can continue to run, the rollback of a sub-transaction does not affect the transaction status of the parent transaction.

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 an M-compatible database, if a new transaction is started in a normal transaction block, a warning is displayed indicating that an ongoing transaction exists and the start command is ignored. 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. If **ROLLBACK** or **COMMIT** is executed, the previous statement is rolled back.

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 an M-compatible database, 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 an M-compatible database, if a new transaction is started in an abnormal transaction block, an error is
reported. The transaction can be executed only after ROLLBACK/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

An M-compatible database uses 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 M-compatible database, table creation and GUC parameter setting support rollback.
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.
```

Differences in SET TRANSACTION

In an M-compatible database, 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 or 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 3-25 Differences in SET TRANSACTION

Syntax	Function	Difference
SET TRANSACTION	Sets transactions.	In M-compatible mode, 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 is applicable to subsequent sessions and has no impact on the current session.	In an M-compatible database, GLOBAL takes effect in global session-level transactions and is applicable 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 an M-compatible database, if the isolation level or transaction access mode is set for multiple times,
only the last setting 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
m_db=# SHOW transaction_read_only; -- on
```

Differences in START TRANSACTION

In an M-compatible database, 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 (,).

```
-- 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 3-26 Differences in transaction-related GUC parameters

GUC Parameter	Function	Difference
autocommit	Sets the automatic transaction commit mode.	-
transaction_isolation	Sets the isolation level of the current transaction in an M-compatible database. 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 transaction isolation level for a session. ● 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.) – If **m_format_dev_version** is set to **'s2'**, the isolation levels of MySQL can be set. ● In GaussDB, the value of **transaction_isolation** of a new transaction is initialized to the value of **default_transaction_isolation**.

GUC Parameter	Function	Difference
tx_isolation	Sets the transaction isolation level. **tx_isolation** and **transaction_isolation** are synonyms.	This parameter does not support query or modification in an M-compatible database. You are advised to use **transaction_isolation** for query.
default_transaction_isolation	Sets the transaction isolation level.	In an M-compatible database, the **SET** command is used to change the transaction isolation level for a session. MySQL does not support this system parameter.
transaction_read_only	Sets the access mode of a transaction.	● In an M-compatible database, 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 transaction isolation level for a session. ● In GaussDB, the value of **transaction_read_only** of a new transaction is initialized to the value of **default_transaction_read_only**.
tx_read_only	Sets the access mode of a transaction. **tx_read_only** and **transaction_read_only** are synonyms.	This parameter does not support query or modification in an M-compatible database. You are advised to use **transaction_read_only** for query.
default_transaction_read_only	Sets the access mode of a transaction.	In an M-compatible database, 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 M-compatible mode 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 M-compatible mode, but can be any of these names or aliases in MySQL.
- If a keyword is a non-reserved one in M-compatible mode 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 M-compatible mode, but cannot be any of these names or aliases in MySQL.
- If a keyword is a reserved one (function or type) both in M-compatible mode and MySQL, it can be a column alias, AS column alias, function name, or variable name in M-compatible mode, but cannot be any of these names or aliases in MySQL.
- If a keyword is a reserved one (function or type) in M-compatible mode but non-reserved in MySQL, it cannot be a table name, column name, AS table alias, or table alias in M-compatible mode, but can be one of these names or aliases in MySQL.
- If a keyword is a non-reserved one (excluding function and type) in M-compatible mode 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 M-compatible mode, 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 M-compatible mode and MySQL, it cannot be a function name in M-compatible mode, 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 M-compatible mode, the following keywords cannot be used as 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 in M-compatible mode 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 be set to **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.

3.7.3 DDL

Table 3-27 DDL syntax compatibility

Description	Syntax	Difference
Create primary keys, UNIQUE indexes, and foreign keys during table creation and modification.	ALTER TABLE and CREATE TABLE	● 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. ● GaussDB: Foreign keys can be used as partition keys. ● The index name, constraint name, and key name are unique in a schema in GaussDB and unique in a table in MySQL.

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.
Support prefix indexes.	CREATE INDEX, ALTER TABLE, and CREATE TABLE	• 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. • 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.

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.
Add columns before the first column of a table or after a specified column during table modification.	ALTER TABLE	-
Alter the column name/definition.	ALTER TABLE	Currently, the DROP INDEX, DROP KEY, or ORDER BY is not supported.

Description	Syntax	Difference
Create a partitioned table.	CREATE TABLE PARTITION	● 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, and partitions cannot be specified. ● GaussDB supports multiple partition keys only when the LIST or RANGE partitioning policy is used. ● In GaussDB partitioned tables, generated columns cannot be used as partition keys.
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	-

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.
Modify the partition key information of a partitioned table.	ALTER TABLE	MySQL allows you to modify the partition key information of a partitioned table, but GaussDB does not.

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: ● M-compatible mode: `TRUNCATE PARTITION { { ALL | partition_name [, ...] } | FOR ( partition_value [, ...] ) } [ UPDATE GLOBAL INDEX ]` ● MySQL: `TRUNCATE PARTITION {partition_names | ALL}`
Index name of a primary key	`CREATE TABLE table_name ( col_definitine ,PRIMARY KEY [index_name] [ USING method ] ( { column_name | ( expression ) }[ ASC | DESC ] } [, ... ] ) index_parameters [USING method| COMMENT 'string'])`	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**.
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.

The NOT NULL constraint does not allow NULL values to be inserted.	`CREATE TABLE t1(id int NOT NULL DEFAULT 8);`
	`INSERT INTO t1 VALUES(NULL);`
	`INSERT INTO t1 VALUES(1),(NULL),(2);`
		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 cannot be inserted in loose or strict mode.

Description	Syntax	Difference
The CHECK constraint takes effect.	CREATE TABLE	● 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.
The **algorithm** and **lock** options of an index do not take effect.	CREATE INDEX ... DROP INDEX ...	Currently, the index options **algorithm_option** and **lock_option** in the CREATE/ DROP INDEX statement in M-compatible mode are supported only in syntax. No error is reported during creation, but they do not take effect.
The storage of hash partitions and level-2 partitions in CREATE TABLE in GaussDB is different from that in MySQL.	CREATE TABLE	In GaussDB, the hash functions used by hash partitioned tables and level-2 partitioned tables in the CREATE TABLE 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.

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 Syntax > SQL Statement > C > CREATE INDEX " in *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.

Description	Syntax	Difference
If the table is partitioned by key in the CREATE/ALTER TABLE statement, algorithms cannot be specified. Input parameters of some partition definition do not support expressions.	CREATE TABLE and ALTER TABLE	In 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()
Partitioned tables do not support LINEAR/KEY hash.	CREATE TABLE ... PARTITION ...	GaussDB: Partitioned tables do not support LINEAR/KEY hash.
The CHECK and AUTO_INCREMENT syntaxes cannot be used in the same column.	CREATE TABLE	The column using CHECK does not take effect in MySQL 5.7. When both CHECK and AUTO_INCREMENT are used on the same column, only AUTO_INCREMENT takes effect. However, GaussDB reports an error.
Delete dependent tables.	DROP TABLE	In GaussDB, CASCADE must be added to delete dependent tables. In MySQL, CASCADE is not required.

Description	Syntax	Difference
Options related to table definition.	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**.
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.
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.

Description	Syntax	Difference
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.
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.

Description	Syntax	Difference
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.
The GENERATED ALWAYS AS statement cannot reference columns generated by GENERATED ALWAYS AS.	Generated Always AS	In GaussDB, the GENERATED ALWAYS AS statement cannot reference columns generated by GENERATED ALWAYS AS, but it can in MySQL.
Support syntaxes that change table names.	ALTER TABLE tbl_name RENAME [TO | AS | =] new_tbl_name;	● 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.
Disable the GUC parameter **enable_expr_fusion**.	SET enable_expr_fusion= ON	In M-compatible mode, the GUC parameter **enable_expr_fusion** cannot be enabled.

Description	Syntax	Difference
Support the CREATE VIEW AS SELECT syntax.	`CREATE VIEW table_name AS query;`	● For the following types, the query using the CREATE VIEW view_name AS query syntax cannot contain calculation operations (such as function call and calculation using operators): – BINARY[(n)] – BOOLEAN/BOOL – 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 – LONGBLOB ● In the simple query scenario, an error message is displayed for the

Description	Syntax	Difference

		preceding calculation operations in M-compatible mode. 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 query scenarios such as composite query and subquery, the calculation operations of the preceding types in M-compatible mode are different from those in MySQL. In M-compatible mode, the data type column precision attribute of the created table is not retained.

Range of index names that can be duplicated	CREATE TABLE, CREATE INDEX	In MySQL, an index name is unique in a table. Different tables can have the same index name. In M-compatible mode, the index name must be unique in the same schema. In M-compatible mode, the same rules apply to constraints and keys that automatically create indexes.

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
Foreign key differences	CREATE 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.
Differences in index ascending and descending orders	CREATE INDEX	In MySQL 5.7, **ASC | DESC** is parsed but ignored, and the default behavior is **ASC**. In MySQL 8.0 and GaussDB, **ASC | DESC** is parsed and takes effect.

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.
		● When the default value is specified, GaussDB does not check whether the default value overflows. When the default value without parentheses is specified in MySQL, MySQL checks whether the default value overflows. When the default value with parentheses is specified, MySQL does not check whether the default value 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`

Description	Syntax	Difference

		`TIMESTAMP DEFAULT (TIMESTAMP '2000-01-01 00:00:00'));`
		`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 '(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 3-28 DML syntax compatibility

Description	Syntax	Difference
DELETE supports deleting data from a specified partition (or subpartition).	DELETE	-
UPDATE supports ORDER BY and LIMIT.	UPDATE	-
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 error "Incorrect Date/Time/Datetime/Timestamp/Year value" is reported.

SELECT supports multi-partition query.	SELECT	-

Description	Syntax	Difference
UPDATE supports multi-partition update.	UPDATE	-

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 M* 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.

Description	Syntax	Difference
		● 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. ● 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.
INSERT supports the VALUES reference column syntax.	INSERT INTO tabname VALUES(1,2,3) ON DUPLICATE KEY UPDATE b = VALUES(column_name)	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.

Description	Syntax	Difference
LIMIT 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.
Modify constraint columns by using INSERT ON DUPLICATE KEY UPDATE.	INSERT	GaussDB does not allow ON DUPLICATE KEY UPDATE to modify constraint columns. But it is allowed in MySQL.
Duplicate column names are allowed in the SELECT result.	SELECT	-
NATURAL JOIN in GaussDB is different from that in MySQL.	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.

Description	Syntax	Difference
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

Compatibility in terms of nature join and using	SELECT	● 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`

The WITH clause is compatible with MySQL 8.0.	SELECT, INSERT, UPDATE, and DELETE	-

Description	Syntax	Difference
Compatibility in terms of join	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.
If the column expression in the SELECT statement is a function expression or arithmetic expression, the column name in the query result is **?column?**.	SELECT	In GaussDB, if the column expression in the SELECT statement is a function expression or arithmetic expression, the column name in the query result is **?column?**. In MySQL, the name is the corresponding expression.
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);`

Description	Syntax	Difference

		```
		CREATE TABLE

		m_db=# INSERT INTO t2 SET a = b + 1;
		ERROR:  Column "b" does not exist.
		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;
		```

Description	Syntax	Difference
		```
		+------+------+
		| a    | b    |
		+------+------+
		|    6 |    5 |
		+------+------+
		1 row in set (0.00 sec)

		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 set for multiple times.
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 operations, the collation is different from that in MySQL. – **utf8_general_ci** is used in MySQL. – The **collation_connection** of the current client is used as the collation in GaussDB. In GaussDB, you are advised not to select columns other than the returned fields in the WHERE clause. Otherwise, unexpected errors may occur.

Description	Syntax	Difference
		-- 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'; ^ -- 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

Description	Syntax	Difference
		only after the compatibility parameter **m_format_dev_version** is set to **'s2'**. The compatibility parameter takes effect by 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.

Description	Syntax	Difference
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**.
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.

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.
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.
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.
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.
EXCEPT Syntax	SELECT	-
SELECT supports the STRAIGHT_JOIN syntax.	SELECT	The execution plans generated in the multi-table JOIN scenarios in GaussDB may be different from those in MySQL.

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.

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.
HAVING 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.

Description	Syntax	Difference

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 FOR SHARE/FOR UPDATE/ LOCK IN SHRAE 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)`

Description	Syntax	Difference
		```
		mysql> CREATE TABLE t1(a INT, b INT);
		Query OK, 0 rows affected (0.09 sec)

		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, 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);`
		`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`

Description	Syntax	Difference

		```
		-- 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

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

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. ● In GaussDB, constraint columns cannot be modified by using ON DUPLICATE KEY UPDATE, but this operation is allowed in MySQL. ● In the INSERT... query ON DUPLICATE KEY UPDATE statement, if query is a subquery, the ON DUPLICATE KEY UPDATE clause cannot reference column names in the subquery in GaussDB, while MySQL supports this behavior. ● 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. ● 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,

Description	Syntax	Difference
		GaussDB returns **1** and MySQL returns **2**. If such update does not change the value of an existing row, GaussDB returns **1** and MySQL returns **0**.

3.7.5 DCL

Table 3-29 DCL syntax compatibility

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 > SQL Statements > S > SET" in *M Compatibility Developer Guide*.

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 3-30 Compatibility of other syntaxes

Description	Syntax	Difference
Transaction-related syntax	Default database isolation level	The default isolation level of an M-compatible database is READ COMMITTED, and that of MySQL is REPEATABLE READ. Only the READ COMMITTED and REPEATABLE READ isolation levels take effect in M-compatible databases.
Transaction-related syntax	Transaction nesting	In M-compatible mode, nested transactions are not automatically committed, but in MySQL, they are automatically committed.
Transaction-related syntax	Autocommit	In M-compatible mode, GaussDB is used for storage and the GaussDB transaction mechanism is inherited. If DDL or DCL 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.
Transaction-related syntax	Rollback is required after an error is reported.	If an error is reported for a transaction in an M-compatible database, rollback needs to be performed. There is no such restriction in MySQL.
Transaction-related syntax	Lock mechanism	The M-compatible lock mechanism can be used only in transaction blocks. There is no such restriction in MySQL.

Description	Syntax	Difference
Lock mechanism	Lock mechanism	• 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 an M-compatible database. • After MySQL locks a table, an error is reported when other tables are read. M-compatible 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. M-compatible databases do not have this mechanism. • M-compatible databases allow LOCK TABLE to be used only inside a transaction block, and have no UNLOCK TABLE command. Locks are always released at the end of transactions.

Description	Syntax	Difference
PBE	PBE	● In an M-compatible database, 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. ● M-compatible databases and MySQL report errors in different phases, such as the parsing layer and execution layer, during SQL statement execution. PREPARE statements process prepared statements till the parsing layer. Therefore, in abnormal scenarios in PBE, an M-compatible database may be different from that in 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 is set to **'forbid_none_space_comment'**.

3.7.7 Users and Permissions

Description

In M-compatible mode, the behaviors and syntaxes related to user and permission control inherit the GaussDB mechanism but are not synchronized with those in MySQL.

User and permission behaviors are the same as those in GaussDB. For details, see "Database Security Management > Managing Users and Their Permissions" in *Developer Guide*.

Some syntaxes for users and permissions are tailored in GaussDB. For details about the syntaxes, see "SQL Reference > SQL Syntax > SQL Statements" in *M Compatibility Developer Guide*. For details about the syntax differences between an M-compatible database and GaussDB, see **Table 3-31**.

Table 3-31 Syntax differences between an M-compatible database and GaussDB

Syntax	Description	Difference
CREATE ROLE	Creates a role.	In M-compatible mode, options involving the following keywords cannot be specified: **ENCRYPTED**, **UNENCRYPTED**, **RESOURCE POOL**, **PERM SPACE**, **TEMP SPACE**, and **SPILL SPACE**. When a user is created, a schema with the same name as the user is automatically created in an M-compatible database, but it is not created in MySQL.
CREATE USER	Creates a user.	
CREATE GROUP	Creates a user group. CREATE GROUP is the alias of CREATE ROLE and is not recommended.	
ALTER ROLE	Modifies role attributes.	
ALTER USER	Modifies user attributes.	
ALTER GROUP	Modifies the attributes of a user group.	-
DROP ROLE	Deletes a role.	-
DROP USER	Deletes a user.	-
DROP GROUP	Deletes a user group.	-
DROP OWNED	Deletes the database objects owned by a database role.	-
REASSIGN OWNED	Changes the owner of a database object.	This syntax is not supported in an M-compatible database.
GRANT	Grants permissions to roles and users.	In an M-compatible database, permissions on objects such as functions, stored procedures, tablespaces, and database links cannot be granted or revoked.
REVOKE	Revokes permissions from one or more roles.	
ALTER DEFAULT PRIVILEGES	Sets the permissions that will be granted to objects created in the future. (It does not affect permissions granted to existing objects.)	This syntax is not supported in an M-compatible database.

Differences

- Syntax format differences

  For details about the M-compatible permission granting syntaxes, see "SQL Reference > SQL Syntax > G > GRANT" in *M Compatibility Developer Guide*. The permission granting 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 3-32 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".

M-compatible databases support the following permissions by level:

Table 3-33 Types of permissions that can be granted in M-compatible databases

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

Object	Permissions That Can Be Granted
Schema	CREATE, USAGE, ALTER, DROP, and COMMENT
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 M-compatible databases.
- 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 M-compatible databases, 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, but MySQL 8.0 and M-compatible databases 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.
- M-compatible databases have 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 M-compatible databases, 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 M-compatible databases, 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 M-compatible databases, 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, the system administrator with the SYSADMIN attribute has 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 M-compatible databases, 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 M-compatible databases, 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.
- When a database, table, or column is deleted from MySQL, the related permission granting information is still retained in the system catalog. If an object with the same name is created again, the user still has the original permissions. In M-compatible databases, when a database, table, or column is deleted, related permission granting information is deleted. If an object with the same name is created again, permissions need to be granted again.
- When granting database-level permissions, MySQL supports fuzzy match of database names using underscores (_) and percent signs (%). However, M-compatible databases do 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 M-compatible databases, permissions cannot be granted to users who are not created.

3.7.8 System Catalogs and System Views

Table 3-34 Differences between M-compatible databases and GaussDB in terms of system catalogs or views

System Catalog or System View	Column	Difference
information_schema.columns	generation_expression	The output of this column varies due to different string concatenation logics of expressions in M-compatible mode and MySQL.

System Catalog or System View	Column	Difference
information_schema.columns	data_type	The output result of this column in M-compatible mode, having not been modified due to the data type format_type involved, is different from that in MySQL.
information_schema.columns	column_type	The output result of this column in M-compatible mode, having not been modified due to the data type format_type involved, is different from that in MySQL.
information_schema.tables	engine	In M-compatible mode: ● 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.
information_schema.tables	version	This column is not supported in M-compatible mode.
information_schema.tables	row_format	This column is not supported in M-compatible mode.
information_schema.tables	avg_row_length	In M-compatible mode, 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**.
information_schema.tables	max_data_length	This column is not supported in M-compatible mode.
information_schema.tables	data_free	In M-compatible mode, it indicates the result of (Number of dead tuples/Total number of tuples) x Data file size. If there is no tuple in the table, the value is **null**.
information_schema.tables	check_time	This column is not supported in M-compatible mode.
information_schema.tables	create_time	The behavior of this column in M-compatible mode is different from that in MySQL. When a view is created in MySQL, this column is set to **null**. In M-compatible mode, the actual table creation time is displayed. The value is **null** if it is a table or view provided by the database.
information_schema.tables	update_time	The value is **null** if it is a table or view provided by the M-compatible database.

System Catalog or System View	Column	Difference
information_schema.statistics	collation	The value can only be **A** or **D** but not NULL in M-compatible mode.
information_schema.statistics	packed	This column is not supported in M-compatible mode.
information_schema.statistics	sub_part	This column is not supported in M-compatible mode.
information_schema.statistics	comment	This column is not supported in M-compatible mode.
information_schema.partitions	subpartition_name	In M-compatible mode, if the partition is not a level-2 partition, the value is **null**.
information_schema.partitions	subpartition_ordinal_position	In M-compatible mode, if the partition is not a level-2 partition, the value is **null**.
information_schema.partitions	partition_method	In M-compatible mode: Partitioning policy. 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
information_schema.partitions	subpartition_method	In M-compatible mode: Level-2 partitioning policy. If a partition is not a level-2 partition, the value is **null**. ● **'r'**: range partition. ● **'i'**: interval partition. ● **'l'**: list partition. ● **'h'**: hash partition
information_schema.partitions	partition_description	In M-compatible mode, level-1 partitions and level-2 partitions are distinguished.
information_schema.partitions	partition_expression	This column is not supported in M-compatible mode.

System Catalog or System View	Column	Difference
information_schema.partitions	subpartition_expression	This column is not supported in M-compatible mode.
information_schema.partitions	data_length	This column is not supported in M-compatible mode.
information_schema.partitions	max_data_length	This column is not supported in M-compatible mode.
information_schema.partitions	index_length	This column is not supported in M-compatible mode.
information_schema.partitions	data_free	This column is not supported in M-compatible mode.
information_schema.partitions	create_time	This column is not supported in M-compatible mode.
information_schema.partitions	update_time	This column is not supported in M-compatible mode.
information_schema.partitions	check_time	This column is not supported in M-compatible mode.
information_schema.partitions	checksum	This column is not supported in M-compatible mode.
information_schema.partitions	partition_comment	This column is not supported in M-compatible mode.
information_schema.partitions	nodegroup	This column is not supported in M-compatible mode.

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 M-compatible mode, and the bigint(21) unsigned type in MySQL corresponds to the bigint unsigned type in M-compatible mode.
- The int type in MySQL corresponds to the integer type in M-compatible mode.
- M-compatible mode does not support columns of the set and enum types that are supported in MySQL. 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.
- **table_rows**, **avg_row_length**, **data_length**, **data_free**, **index_length**, and **cardinality** in information_schema.tables and **cardinality** in information_schema.statistics 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.
- **table_row** in information_schema.partitions is obtained based on statistics. Before viewing the value, run **ANALYZE** to update the statistics. (If data is updated in the database, you are advised to delay running **ANALYZE**.)
- The format of the **grantee** column supported in MySQL is **'***user_name***'@'***host_name***'**. In the M-compatible database, it is the name of the user or role to which the permission is granted.
- For the **host** column supported in the M-compatible database, the **hostname** of the current node is returned.
- 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 the M-compatible database, 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 the M-compatible database, 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

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 3-35 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**.