ALIBABA CLOUD

阿里云

云原生数据仓库AnalyticDB MySQL版 SQL手册

文档版本：20200902

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	**单击设置> 网络> 设置网络类型。**
粗体	表示按键、菜单、页面名称等UI元素。	**在结果确认页面，单击确定。**
`Courier字体`	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.数据类型

本文介绍云原生数据仓库AnalyticDB MySQL版（简称ADB，原分析型数据库MySQL版）支持哪些数据类型以及与MySQL数据类型的差异对比。

ADB支持的数据类型

数据类型	说明
boolean 布尔类型	值只能是 0 或 1 ，存储字节数1比特位。 • 取值 0 的逻辑意义为假。 • 取值 1 的逻辑意义为真。
tinyint 微整数类型	取值范围 -128 ~ 127 ，存储字节数1字节。
smallint 小整数类型	取值范围 -32768 ~ 32767 ，存储字节数2字节。
int 或 integer 整数类型	取值范围 -2147483648 ~ 2147483647 ，存储字节数4字节。
bigint 大整数类型	取值范围 -9223372036854775808 ~ 9223372036854775807 ，存储字节数8字节。
float 单精度浮点数	取值范围 -3.402823466E+38 ~ -1.175494351E-38 ， 0 ， 1.175494351E-38 ~ 3.402823466E+38 ，IEEE标准，存储字节数4字节。
double 双精度浮点数	取值范围 -1.7976931348623157E+308 ~ -2.2250738585072014E-308 ， 0 ， 2.2250738585072014E-308 ~ 1.7976931348623157E+308 ，IEEE标准，存储字节数8字节。
decimal(m,d)	m 是数值的最大精度，取值范围为 1 ~ 1000 ； d 是小数点右侧数字的位数，要求 d ≤ m 。
varchar 变长字符串类型	存储字节数最大为16MB，使用时无需指定存储长度。
date 日期类型	取值范围 '0001-01-01' ~ '9999-12-31' ，支持的数据格式为 'YYYY-MM-DD' ，存储字节数为4字节。
time 时间类型	取值范围 '00:00:00' ~ '23:59:59' ，支持的数据格式为 'HH:MM:SS' ，存储字节数为8字节。

数据类型	说明
datetime 时间戳类型	取值范围 '0001-01-01 00:00:00.000' UTC~ '9999-12-31 23:59:59.999' UTC，支持的数据格式为 'YYYY-MM-DD HH:MM:SS' ，存储字节数为8字节。 说明 datetime 默认UTC时间，且不支持可更改。
timestamp 时间戳类型	时间戳类型，取值范围 '0001-01-01 00:00:00.000' UTC~ '9999-12-31 23:59:59.999' UTC，支持的数据格式为 'YYYY-MM-DD HH:MM:SS' ，存储字节数为4字节。 说明 timestamp 默认为系统时区，可以在SESSION中设置时区。
json	请参见JSON。
multivalue 多值子列类型	支持 1,2,3 或 1:a;2:b 等多字段的多值列，详情请参见多值列。

与MySQL数据类型对比

ADB	MySQL	差异
boolean	bool 、 boolean	一致。
tinyint	tinyint	一致。
smallint	smallint	一致。
int 、 integer	int 、 integer	一致。
bigint	bigint	一致。
float	float[(m,d)]	一致。
double	double[(m,d)]	一致。
decimal	decimal	ADB支持的最大精度为 1000 ，MySQL支持的最大精度为 65 。

ADB	MySQL	差异
varchar	varchar	ADB中的 varchar 对应MySQL中的 char 、 varchar 、 text 、 mediumtext 或者 longtext 。
date	date	MySQL支持 0000-00-00 ；在ADB中写入 0000-00-00 时，系统自动将其转化为 null 。
time	time	ADB精确到毫秒，MySQL支持自定义精度。
datetime	datetime	MySQL支持 0000-00-00 00:00:00 ；在ADB中写入 0000-00-00 00:00:00 时，系统自动将其转化为 null 。 ADB精确到毫秒，MySQL支持自定义精度。
timestamp	timestamp	ADB精确到毫秒，MySQL支持自定义精度。

2.多值列

多值列用于表示一个列中（Cell）有多个不确定的值，一个多值列字段可以包含多种ADB支持的数据类型。一个表中可以定义一个或多个多值列，多值列可以作为筛选条件，也可以进行分组或者参与连接过滤等。

创建表

以下示例创建TEST1表，通过MULTIVALUE 定义 `tags` 字段为多值列，多值列之间以英文逗号（,）分隔，多值列中包含一个Varchar类型的字段 `value_type` 。

```
CREATE TABLE test1 (
user_id BIGINT,
city VARCHAR,
tags MULTIVALUE delimiter_tokenizer ',' value_type 'varchar'
)
DISTRIBUTED BY HASH (user_id)
```

写入数据

通过以下语句向TEST1表中写入三条数据。

```
insert into test1 values(1, 'HZ', 'A,B,C');
insert into test1 values(2, 'BJ', 'B,D');
insert into test1 values(3, 'SH', 'A,C,D,F');
```

```
select * from test1 order by user_id;
+---------+------+---------+
| user_id | city | tags |
+---------+------+---------+
| 1 | HZ | A,B,C |
| 2 | BJ | B,D |
| 3 | SH | A,C,D,F |
```

查询数据

- 过滤查询

  通过以下语句查询TEST1表包含A、B标签的用户数。

  ```
  select count(*) from test1 where ref(tags,0) in ('A', 'B');
  +----------+
  | COUNT(*) |
  +----------+
  | 3 |
  ```

通过以下语句查询TEST1表中同时包含A和B标签的用户数目。

```
select * from test1 where ref(tags,0) in ('A') and ref(tags,0) in ('B');
+---------+------+-------+
| user_id | city | tags |
+---------+------+-------+
| 1 | HZ | A,B,C |
```

- 分组查询

  通过以下语句查询TEST1表中city不在杭州且标签为A或者B的用户在A、B标签下的人数。

```
select ref(tags,0), count(*) from test1
where ref(tags,0) in ('A', 'B') and city != 'HZ' group by ref(tags,0);
+--------------+----------+
| ref(tags, 0) | COUNT(*) |
+--------------+----------+
| A | 1 |
| B | 1 |
```

  通过以下语句查询TEST1表中A标签或者B标签与其对应城市的联合分组。

```
select ref(tags,0), city, count(*) from test1
where ref(tags,0) in ('A', 'B') group by ref(tags,0), city;
+--------------+----------+
| ref(tags, 0) | COUNT(*) |
+--------------+----------+
| B | BJ |
| B | HZ |
| A | SH |
| A | HZ |
```

- 连接查询

```
select count(*) from test1 join test
on test1.user_id = test.id where ref(tags,0) in ('A', 'B') ;
+----------+
| COUNT(*) |
+----------+
| 0 |
```

注意事项

- 多值列条件只能用于单表扫描条件，不能用于Join条件。

- 当Group By分组中包含多值列时，Where条件中也必须包含多值列，否则ADB引擎底层会穷举所有多值列的value，影响ADB查询性能。

  > **说明** 默认穷举1024个多值列value。

3.DDL

3.1. CREATE DATABASE

`CREATE DATABASE` 用于创建数据库。

创建数据库

> 说明 每个集群最多可以创建256个数据库。

- 语法

```
CREATE DATABASE [IF NOT EXISTS] db_name
```

- 参数

  `db_name`：数据库名。以小写字符开头，可包含字母、数字以及下划线（_），但不能包含连续两个及以上的下划线（_），长度不超过64个字符。

  > 说明 数据库名不能是analyticdb，analyticdb是内置数据库。

- 示例

```
CREATE DATABASE adb_demo;
```

使用数据库

数据库创建成功后，您可以通过 `USE db_name` 命令使用数据库。

- 语法

```
USE db_name
```

- 示例

```
use adb_demo;
show tables;
+------------------------------+
| Tables_in_adb_demo |
+------------------------------+
| customer |
| test_table |
```

3.2. CREATE TABLE

ADB支持通过 `CREATE TABLE` 创建表，也支持通过 `CTAS` 将查询到的数据写入新表中。

语法

```
CREATE TABLE [IF NOT EXISTS] table_name
({column_name column_type [column_attributes] [ column_constraints ] [COMMENT 'string']
| table_constraints}
[, ... ] )
table_attribute
[partition_options]
[AS] query_expression
COMMENT 'string'

column_attributes:
[DEFAULT default_expr]
[AUTO_INCREMENT]

column_constraints:
[{NOT NULL|NULL} ]
[PRIMARY KEY]

table_constraints:
[{INDEX|KEY} [index_name] (column_name,...)]
[PRIMARY KEY [index_name] (column_name,...)]
[CLUSTERED KEY [index_name] (column_name,...)]

table_attribute:
DISTRIBUTED BY HASH(column_name,...) | DISTRIBUTED BY BROADCAST

partition_options:
PARTITION BY
{VALUE(column_name) | VALUE(date_format(column_name, ?))}
LIFECYCLE N
```

参数

参数	说明
table_name	表名。 表名以字母或下划线（_）开头，可包含字母、数字以及下划线（_），长度为1到127个字符。 支持 db_name.table_name 格式，区分不同数据库下相同名字的表。

参数	说明
column_name	列名。 列名以字母或下划线（_）开头，可包含字母、数字以及下划线（_），长度为1到127个字符。
column_type	要添加的列的数据类型。 AnalyticDB for MySQL支持的数据类型请参见数据类型。
column_attributes	• DEFAULT default_expr：设置列的默认值，DEFAULT为无变量表达式，例如current_timestamp。 如果未指定默认值，则列的默认值为NULL。 • AUTO_INCREMENT：定义自增列，可选项。 自增列的数据类型必须是BIGINT类型，AnalyticDB for MySQL为自增列提供唯一值，但自增列的值不是顺序递增。
column_constraints	• NOT NULL | NULL：定义了NOT NULL的列不允许值为NULL；定义了NULL（默认值）的列允许值为NULL。 • PRIMARY KEY：定义主键。 如果有多个主键，语法为PRIMARY KEY(column_name [, ...])。
table_constraints	INDEX|KEY：倒排索引。 AnalyticDB for MySQL默认为表创建全索引，一般情况下无须手动创建索引。
PRIMARY KEY	主键索引。 • 只有定义过主键的表支持DELETE和UPDATE操作。 • 主键中必须包含分区键，建议把分区键放到组合主键之前。

参数	说明
CLUSTERED KEY	聚集索引，定义表中的排序列，聚集索引中键值的逻辑顺序决定了表中相应行的物理顺序。 例如， clustered key col5_col6_cls_index(col5,col6) 定义了 col5 col6 的聚集索引， col5 col6 和 col6 col5 是不同的聚集索引。 聚集索引会将该列或者多列进行排序，保证与该列相同或者相近的数据存储在磁盘的相同或相近位置。当以聚集列做为查询条件时，查询结果存储在磁盘的相同位置，这样可以减少磁盘的IO，提高查询性能。 如何判断是否需要聚集索引：查询一定会携带的字段可以作为聚集索引。例如，SAAS类应用中，用户通常只访问自己的数据，用户ID可以定义为聚集索引，保证数据的局部性，提升数据查询性能。 聚集列有以下限制： • 每张表中只支持创建一个聚集列索引。 • 由于聚集索引会进行全表排序，导致数据写入性能下降、CPU占用较高，因此一般不建议使用聚集索引。
DISTRIBUTED BY HASH(column_name,...)	在普通表中定义表的分布键，按照 column_name 的HASH值进行分区。 AnalyticDB for MySQL支持将多个字段作为分区键。
DISTRIBUTED BY BROADCAST	用于定义维度表，维度表会在集群的每个节点存储一份数据，因此建议维度表的数据量不宜太大。
partition_options	普通表中定义分区。 AanlyticDB for MySQL通过 LIFECYCLE N 方式实现表生命周期管理，即对分区进行排序，超出 N 的分区将被过滤掉。 例如， PARTITION BY VALUE(column_name) 表示使用 column_name 的值来做分区， PARTITION BY VALUE(DATE_FORMAT(column_name, '%Y%m%d')) 表示将 column_name 格式化为类似 20190101 的日期格式做分区。 LIFECYCLE 365 表示每个节点最多保留的分区个数为365，即如果数据保存天数为365天，则第366天写入数据后，系统会自动删除第1天写入的数据。

注意事项

- 创建表时，AnalyticDB for MySQL集群默认编码格式为utf-8，相当于MySQL中的utf8mb4编码，暂不支持其他编码格式。
- 目前AnalyticDB for MySQL集群支持创建的最大表数目如下所示：
  - 集群版：min（节点组数量*256,10000）。

- 基础版：
  - T8，500。
  - T16和T32，1500。
  - T52，2500。

示例

- 新建TEST表。

```
create table test (
id bigint auto_increment,
name varchar,
value int,
ts timestamp
)
DISTRIBUTED BY HASH(id)
```

TEST为普通表，`id`为自增列，分布键为`id`，按照`id`值进行HASH分区。

- 新建CUSTOMER表。

```
CREATE TABLE customer (
customer_id bigint NOT NULL COMMENT '顾客ID',
customer_name varchar NOT NULL COMMENT '顾客姓名',
phone_num bigint NOT NULL COMMENT '电话',
city_name varchar NOT NULL COMMENT '所属城市',
sex int NOT NULL COMMENT '性别',
id_number varchar NOT NULL COMMENT '身份证号码',
home_address varchar NOT NULL COMMENT '家庭住址',
office_address varchar NOT NULL COMMENT '办公地址',
age int NOT NULL COMMENT '年龄',
login_time timestamp NOT NULL COMMENT '登录时间',
PRIMARY KEY (login_time,customer_id，phone_num)
)
DISTRIBUTED BY HASH(customer_id)
PARTITION BY VALUE(DATE_FORMAT(login_time, '%Y%m%d')) LIFECYCLE 30
COMMENT '客户信息表';
```

CUSTOMER表为普通表，`customer_id`为分布键，`login_time`为分区键，`login_time`、`customer_id`、`phone_num`为组合主键。

MySQL语法兼容性说明

AnalyticDB for MySQL标准建表语法中必须包含 `DISTRIBUTED BY ...` ，而MySQL建表语法中没有 `DISTRIBUTED BY ...` 。AnalyticDB for MySQL默认兼容MySQL建表语法，您可以根据实际情况通过以下两种方式处理 `DISTRIBUTED BY ...` 不兼容问题。

- 如果MySQL表含有主键，AnalyticDB for MySQL默认将主键作为 `DISTRIBUTED BY COLUMN` 。

```
mysql> create table t (c1 bigint, c2 int, c3 varchar, primary key(c1,c2));
Query OK, 0 rows affected (2.37 sec)
mysql> show create table t;
+-------+-----------------------------------------------------------------------------------------------------------------------
------------------+
| Table | Create Table |
+-------+-----------------------------------------------------------------------------------------------------------------------
------------------+
| t | Create Table `t` (
`c1` bigint,
`c2` int,
`c3` varchar,
primary key (c1,c2)
) DISTRIBUTED BY HASH(`c1`,`c2`) INDEX_ALL='Y' |
+-------+-----------------------------------------------------------------------------------------------------------------------
------------------+
1 row in set (0.04 sec)
```

- 如果MySQL表不含主键，AnalyticDB for MySQL将添加一个 `__adb_auto_id__` 字段作为主键和 `DISTRIBUTED BY COLUMN` 。

```
mysql> create table t (c1 bigint, c2 int, c3 varchar);
Query OK, 0 rows affected (0.50 sec)
mysql> show create table t;
+-------+------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------+
| Table | Create Table |
+-------+------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------+
| t | Create Table `t` (
`c1` bigint,
`c2` int,
`c3` varchar,
`__adb_auto_id__` bigint AUTO_INCREMENT,
primary key (__adb_auto_id__)
) DISTRIBUTED BY HASH(`__adb_auto_id__`) INDEX_ALL='Y' |
+-------+------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------+
1 row in set (0.04 sec)
```

3.3. 全文检索

ADB支持通过SQL定义全文检索列实现全文检索功能， 全文检索列由多值列实现，分词器类型为nlp_tokenizer。

说明 全文检索不同于SQL中的like，全文检索是在SQL中通过分词搜索匹配；like是模糊匹配，只需要为字符串列创建索引即可。

创建表

通过以下SQL创建TEST3表，定义text列为多值列类型，分词器类型为nlp_tokenizer。

```
CREATE TABLE test3 (
user_id BIGINT,
city VARCHAR,
text MULTIVALUE nlp_tokenizer 'ik' value_type 'varchar'
)
DISTRIBUTE BY HASH (user_id)
```

写入数据

通过以下SQL向TEST3表中写入测试数据。

```
insert into test3 values(1, 'HZ', '中华人民共和国');
insert into test3 values(2, 'BJ', 'AnalyticDB3.0是全新一代OLAP数据库');
insert into test3 values(3, 'SH', 'hello, world');
```

```
select * from test3 order by user_id;
+---------+------+-------------------------------+
| user_id | city | text |
+---------+------+------------------------------ +
| 1 | HZ |中华人民共和国 |
| 2 | BJ |AnalyticDB3.0是全新一代OLAP数据库|
| 3 | SH | hello, world |
```

检索查询

> **说明** 全文检索中，默认将所有大写字母统一转换成小写字母。

```
select * from test3 where text in ('中华', '数据库');
+---------+------+-------------------------------+
| user_id | city | text |
+---------+------+-------------------------------+
| 1 | HZ | 中华人民共和国 |
| 2 | BJ | AnalyticDB3.0是全新一代OLAP数据库|
```

```
select * from test3 where text in ('hello') and city != 'HZ';
+---------+------+--------------+
| user_id | city | text |
+---------+------+--------------+
| 3 | SH | hello, world |
```

```
select * from test3 where ref(text,0) in ('analyticdb');
+---------+------+---------------------------------+
| user_id | city | text |
+---------+------+---------------------------------+
| 2 | BJ | AnalyticDB3.0是全新一代OLAP数据库 |
```

```
# 普通字符串列也可以进行like查询
mysql> select * from test3 where city like '%J%';
+---------+------+-----------------------------------+
| user_id | city | text |
+---------+------+-----------------------------------+
| 2 | BJ | AnalyticDB3.0是全新一代OLAP数据库 |
```

3.4. CTAS

ADB支持通过 `CREATE TABLE` 创建表，也支持通过 `CTAS` 将查询到的数据写入新表中。

语法

```
CREATE TABLE [IF NOT EXISTS] <table_name> [table_definition]
[IGNORE|REPLACE] [AS] <query_statement>
```

> **说明** 该建表方式默认与CREATE TABLE一致，支持语法也相同，例如默认为表创建全索引等。

参数

参数	说明
`table_name`	表名。 表名以字母或下划线（_）开头，可包含字母、数字以及下划线（_），长度为1到127个字符。 支持 `db_name.table_name` 格式，区分不同数据库下相同名字的表。
`IF NOT EXISTS`	判断 `table_name` 指定的表是否存在，若存在，则不执行建表语句。
`IGNORE`	可选参数，若表中已有相同主键的记录，新记录不会被写入。
`REPLACE`	可选参数，若表中已有相同主键的记录，新记录将替换已有相同主键的记录。

示例

```
CREATE TABLE customer (
customer_id bigint NOT NULL COMMENT '顾客ID',
customer_name varchar NOT NULL COMMENT '顾客姓名',
phone_num bigint NOT NULL COMMENT '电话',
city_name varchar NOT NULL COMMENT '所属城市',
sex int NOT NULL COMMENT '性别',
id_number varchar NOT NULL COMMENT '身份证号码',
home_address varchar NOT NULL COMMENT '家庭住址',
office_address varchar NOT NULL COMMENT '办公地址',
age int NOT NULL COMMENT '年龄',
login_time timestamp NOT NULL COMMENT '登录时间',
PRIMARY KEY (login_time,customer_id,phone_num)
)
DISTRIBUTED BY HASH(customer_id)
PARTITION BY VALUE(DATE_FORMAT(login_time, '%Y%m%d')) LIFECYCLE 30
COMMENT '客户信息表';
```

```
INSERT INTO
customer values
(002367,'李四','13678973421','杭州',0,'987300','西湖','转塘云栖小镇',23,'2018-03-02 10:00:00'),(002368,'张三','138
78971234','杭州',0,'987300','西湖','转塘云栖小镇',28,'2018-08-01 11:00:00'),(002369,'王五','13968075284','杭州',1,
'987300','西湖','转塘云栖小镇',35,'2018-09-12 08:11:00');
```

通过以下SQL从CUSTOMER表中读取数据，并将数据写入新表NWE_CUSTOMER表中。

```
CREATE TABLE new_customer AS SELECT * FROM customer;
```

```
CREATE TABLE new_table AS SELECT a, b FROM base_table
```

```
CREATE TABLE new_table (PRIMARY KEY (a)) DISTRIBUTE BY HASH (b)
AS SELECT a, b FROM base_table
```

```
CREATE TABLE new_table (c varchar, PRIMARY KEY (a)) DISTRIBUTE BY HASH (b)
AS SELECT a, b, c FROM base_table
```

```
CREATE TABLE new_table (INDEX a_idx (a))
AS SELECT a, b, c FROM base_table
```

3.5. ALTER TABLE

ALTER TABLE用于修改表。

语法

```
ALTER TABLE table_name
ADD COLUMN (column_name column_definition,...)
| ADD {INDEX|KEY} [index_name] (column_name,...)
| ADD CLUSTERED [INDEX|KEY] [index_name] (column_name,...)
| DROP COLUMN column_name
| DROP {INDEX|KEY} index_name
| DROP CLUSTERED [INDEX|KEY] index_name
| MODIFY COLUMN column_name column_definition
| RENAME new_table_name
| TRUNCATE PARTITION {partition_names | ALL}
```

增加列

- 语法

  ```
  ALTER TABLE db_name.table_name ADD column_name data_type;
  ```

- 示例

  在CUSTOMER表中增加一列province，数据类型为VARCHAR。

  ```
  ALTER TABLE adb_demo.customer ADD COLUMN province varchar comment '省份';
  ```

删除列

- 语法

  ```
  ALTER TABLE db_name.table_name DROP column_name data_type;
  ```

- 示例

  在CUSTOMER表中删除类型为VARCHAR的province列。

  ```
  ALTER TABLE adb_demo.customer DROP COLUMN province;
  ```

更改COMMENT

- 语法

  ```
  ALTER TABLE db_name.table_name MODIFY COLUMN column_name data_type comment 'new_commen
  t';
  ```

- 示例

  将CUSTOMER表中province列的COMMENT修改为顾客所属省份。

```
ALTER TABLE adb_demo.customer MODIFY COLUMN province varchar comment '顾客所属省份';
```

设置NULL

> 说明 仅支持将NOT NULL变更为NULL。

- 语法

```
ALTER TABLE db_name.table_name MODIFY COLUMN column_name data_type {NULL};
```

- 示例

  将CUSTOMER表中province列的值更改为可空（NULL）。

```
ALTER TABLE adb_demo.customer MODIFY COLUMN province varchar NULL;
```

更改DEFAULT值

- 语法

```
ALTER TABLE db_name.table_name MODIFY COLUMN column_name data_type DEFAULT 'default';
```

- 示例

  将CUSTOMER表中性别sex的默认值设置为0（性别为男）。

```
ALTER TABLE adb_demo.customer MODIFY COLUMN sex int(11) NOT NULL DEFAULT 0;
```

更改列类型

- 语法

```
ALTER TABLE db_name.table_name MODIFY COLUMN column_name new_data_type;
```

- 注意事项

  仅支持整型数据类型之间，以及浮点数据类型之间的列类型更改，并且只能将取值范围小的数据类型更改为取值范围大的数据类型，或者将单精度数据类型更改为双精度数据类型。

  - 整型数据类型：支持Tinyint、Smallint、 Int 、 Bigint间，小类型到大类型的更改，例如支持将Tinyint更改为Bigint，不支持将Bigint更改为Tinyint。
  - 浮点数据类型：支持将Float更改为Double类型，不支持将Double更改为Float类型。

- 示例

  将TEST表中order_number列由Int类型更改为Bigint类型。

```
CREATE TABLE adb_demo.test(id int, order_number int NOT NULL DEFAULT 100, name varchar) DISTRI
BUTE BY HASH(id);
```

```
ALTER TABLE adb_demo.test MODIFY COLUMN order_number BIGINT NOT NULL DEFAULT 100;
```

新增索引

> 说明
> - AnalyticDB for MySQL建表时默认创建全列索引 `index_all='Y'` 。
> - 若建表时未创建全列索引，可以通过以下方式新增索引。

- 语法

```
ALTER TABLE db_name.table_name ADD KEY index_name(column_name);
```

- 示例

  在CUSTOMER表中为age列新增索引。

```
ALTER TABLE adb_demo.customer ADD KEY age_idx(age);
```

删除索引

- 语法

```
ALTER TABLE db_name.table_name DROP KEY index_name;
```

- 参数说明

  可以通过以下命令获取 `index_name` 。

```
SHOW INDEXES FROM db_name.table_name;
```

- 示例

  删除CUSTOMER表中age列的索引。

```
ALTER TABLE adb_demo.customer DROP KEY age_idx;
```

更改列名

> 说明 不支持更改主键列的列名。

- 语法

```
ALTER TABLE db_name.table_name rename column column_name to column_newname;
```

- **示例**

  将CUSTOMER表中的age列更名为new_age。

```
ALTER TABLE customer rename column age to new_age;
```

修改表的生命周期

- **语法**

```
ALTER TABLE db_name.table_name partitions N;
```

- **示例**

  将CUSTOMER表的生命周期由30改为40。

```
CREATE TABLE customer (
customer_id bigint NOT NULL COMMENT '顾客ID',
customer_name varchar NOT NULL COMMENT '顾客姓名',
phone_num bigint NOT NULL COMMENT '电话',
city_name varchar NOT NULL COMMENT '所属城市',
sex int NOT NULL COMMENT '性别',
id_number varchar NOT NULL COMMENT '身份证号码',
home_address varchar NOT NULL COMMENT '家庭住址',
office_address varchar NOT NULL COMMENT '办公地址',
age int NOT NULL COMMENT '年龄',
login_time timestamp NOT NULL COMMENT '登录时间',
PRIMARY KEY (login_time,customer_id，phone_num)
)
DISTRIBUTED BY HASH(customer_id)
PARTITION BY VALUE(DATE_FORMAT(login_time, '%Y%m%d')) LIFECYCLE 30
COMMENT '客户信息表';
```

```
alter table customer partitions 40;
```

3.6. CREATE VIEW

`CREATE VIEW` 用于创建视图。

语法

```
CREATE VIEW view_name AS select_stmt
```

参数

- `view_name`：视图的名字，视图名前可加上数据库名。
- `select_stmt`：视图中的数据来源。

示例

创建视图v，视图数据来源为CUSTOMER表中的数据。

```
CREATE VIEW adb_demo.v AS SELECT * FROM customer;
```

3.7. DROP DATABASE

`DROP DATABASE` 用于删除数据库。

语法

```
DROP DATABASE db_name;
```

> 说明 删除数据库之前，必须先删除数据库中的表。

示例

```
use adb_demo2;
+------------------------------+
show tables;
+------------------------------+
| Tables_in_adb_demo2 |
+------------------------------+
| test2 |
+------------------------------+
drop table test2;
drop database adb_demo2;
```

3.8. DROP TABLE

`DROP TABLE` 用于删除表。

语法

```
DROP TABLE db_name.table_name;
```

> 说明 执行该命令会同时删除表数据和表结构。

示例

```
DROP TABLE adb_demo.customer;
```

3.9. DROP VIEW

`DROP VIEW` 用于删除视图。

语法

```
DROP VIEW [IF EXISTS] view_name, [, view_name] ...
```

参数

`view_name` ：视图名字，视图名前可以加上数据库名，区分不同数据库中的同名视图。

示例

删除视图v。

```
DROP VIEW v;
```

4.DML

4.1. INSERT INTO

`INSERT INTO` 用于向表中插入数据，遇到主键重复时会自动忽略当前写入数据，不做更新，作用等同于 `INSERT IGNORE INTO` 。

语法

```
INSERT [IGNORE]
INTO table_name
[( column_name [, ...] )]
[VALUES]
[(value_list[, ...])]
[query];
```

参数

- `IGNORE` ：可选参数，若系统中已有相同主键的记录，新记录不会被写入。
- `column_name` ：可选参数，列名。
- `query` ：通过定义查询，将一行或多行数据插入表中。

注意事项

如果插入数据时不指定列名，则要插入的数据必须和CREATE TABLE语句中声明的列的顺序一致。

示例

创建CUSTOMER和COURSES表。

```
CREATE TABLE customer (
customer_id bigint NOT NULL COMMENT '顾客ID',
customer_name varchar NOT NULL COMMENT '顾客姓名',
phone_num bigint NOT NULL COMMENT '电话',
city_name varchar NOT NULL COMMENT '所属城市',
sex int NOT NULL COMMENT '性别',
id_number varchar NOT NULL COMMENT '身份证号码',
home_address varchar NOT NULL COMMENT '家庭住址',
office_address varchar NOT NULL COMMENT '办公地址',
age int NOT NULL COMMENT '年龄',
login_time timestamp NOT NULL COMMENT '登录时间',
PRIMARY KEY (login_time，customer_id，phone_num)
)
DISTRIBUTED BY HASH(customer_id)
PARTITION BY VALUE(DATE_FORMAT(login_time, '%Y%m%d')) LIFECYCLE 30
COMMENT '客户信息表';
```

```
CREATE TABLE courses(
id bigint AUTO_INCREMENT PRIMARY KEY,
name varchar(20) NOT NULL,
grade varchar(20) default '三年级',
submission_date timestamp
)
DISTRIBUTED BY HASH(id)
```

- 向CUSTOMER表中插入一条数据。

```
INSERT INTO customer(customer_id,customer_name,phone_num,city_name,sex,id_number,home_addr
ess,office_address,age,login_time)
values
(002367,'杨过','13678973421','杭州',0,'987300','西湖','转塘云栖小镇',23,'2018-03-02 10:00:00');
```

- 向CUSTOMER表中插入多条数据。

```
INSERT INTO customer(customer_id,customer_name,phone_num,city_name,sex,id_number,home_addr
ess,office_address,age,login_time)
values
(002367,'李四','13678973421','杭州',0,'987300','西湖','转塘云栖小镇',23,'2018-03-02 10:00:00'),(002368,'张三','
13878971234','杭州',0,'987300','西湖','转塘云栖小镇',28,'2018-08-01 11:00:00'),(002369,'王五','13968075284','
杭州',1,'987300','西湖','转塘云栖小镇',35,'2018-09-12 08:11:00');
```

- 向CUSTOMER表中插入多条数据时，可以省略列名。

```
INSERT INTO
customer values
(002367,'李四','13678973421','杭州',0,'987300','西湖','转塘云栖小镇',23,'2018-03-02 10:00:00'),(002368,'张三','13878971234','杭州',0,'987300','西湖','转塘云栖小镇',28,'2018-08-01 11:00:00'),(002369,'王五','13968075284','杭州',1,'987300','西湖','转塘云栖小镇',35,'2018-09-12 08:11:00');
```

- 向COURSES表中插入一条数据。

```
insert into courses (name,submission_date) values("Jams",NOW());
```

- `INSERT query` 示例请参见INSERT SELECT FROM。

4.2. REPLACE INTO

`REPLACE INTO` 用于实时覆盖写入数据。写入数据时，根据主键判断待写入的数据是否已经存在于表中，如果已经存在，则先删除该行数据，然后插入新的数据；如果不存在，则直接插入新数据。

语法

```
REPLACE INTO table_name [(column_name,...)] VALUES ({常量|NULL|DEFAULT},...),(...),...
```

示例

- 通过 `REPLACE INTO` 向CUSTOMER表中插入一条数据。

```
REPLACE INTO customer(customer_id,customer_name,phone_num,city_name,sex,id_number,home_address,office_address,age,login_time)
values
(002367,'杨过','13678973421','杭州',0,'987300','西湖','转塘云栖小镇',23,'2018-03-02 10:00:00');
```

- 向CUSTOMER表中插入多条数据时，可以省略列名。

```
REPLACE INTO
customer values
(002367,'李四','13678973421','杭州',0,'987300','西湖','转塘云栖小镇',23,'2018-03-02 10:00:00'),(002368,'张三','13878971234','杭州',0,'987300','西湖','转塘云栖小镇',28,'2018-08-01 11:00:00'),(002369,'王五','13968075284','杭州',1,'987300','西湖','转塘云栖小镇',35,'2018-09-12 08:11:00');
```

4.3. INSERT SELECT FROM

如果您的数据在其他表中已经存在，可以通过 `INSERT SELECT FROM` 将数据复制到另外一张表。

语法

```
INSERT INTO table_name
[( column_name [, ...] )]
query;
```

参数

- `column_name` ：列名，如果需要将源表中的部分列数据插入到目标表中，SELECT子句中的列必须与INSERT子句中列的顺序、数据类型一致。
- `query` ：可以是 `SELECT FROM TABLE` 或者 `SELECT FROM VIEW` 。

示例

- 以指定列名的方式，从CUSTOMER表中复制某几列数据到NEW_CUSTOMER表中。

```
INSERT INTO new_customer (customer_id, customer_name, phone_num)
SELECT customer_id, customer_name, phone_num FROM customer
WHERE customer.customer_name = '杨过';
```

- 不指定列名，从CUSTOMER表中复制所有列数据到NEW_CUSTOMER表中。

```
INSERT INTO new_customer
SELECT customer_id,customer_name,phone_num,city_name,sex,id_number,home_address,office_add
ress,age,login_time) FROM customer
WHERE customer.customer_name = '杨过';
```

4.4. REPLACE SELECT FROM

`REPLACE SELECT FROM` 用于将其他表中的数据实时覆盖写入目标表中。写入数据时，根据主键判断待写入的数据是否已经存在于表中，如果已经存在，则先删除该行数据，然后插入新的数据；如果不存在，则直接插入新数据。

语法

```
REPLACE INTO table_name
[(column_name,...)]
query;
```

参数

- `query` ：可以是 `SELECT FROM TABLE` 或者 `SELECT FROM VIEW` 。
- `column_name` ：列名，如果需要将源表中的部分列数据插入到目标表中， `SELECT` 子句中的列必须与 `REPLACE` 子句中列的顺序、数据类型一致。

注意事项

执行 `REPLACE SELECT FROM` 命令时，需先创建待写入数据的目标表。

示例

以指定列名的方式，从CUSTOMER表中复制某几列数据到NEW_CUSTOMER表中。

```
REPLACE INTO new_customer (customer_id, customer_name, phone_num)
SELECT customer_id, customer_name, phone_num FROM customer
WHERE customer.customer_name = '杨过';
```

4.5. INSERT OVERWRITE INTO SELECT

`INSERT OVERWRITE INTO SELECT` 用于向表中批量插入数据。

语法

```
INSERT OVERWRITE INTO table_name [(column_name,...)]
SELECT select_statement FROM from_statement
```

注意事项

- 执行 `INSERT OVERWRITE INTO SELECT` 命令时，需要提前创建目标表。
- 如果目标表中已存在数据， `INSERT OVERWRITE INTO SELECT` 命令执行结束之前，目标表中的数据不会发生任何变化； `INSERT OVERWRITE INTO SELECT` 命令执行结束后，系统自动一键切换将数据写入目标表中。
- 执行 `INSERT OVERWRITE INTO SELECT` 遇到主键重复时，以第一条写入数据为准。
- 向同一个表中写入数据时， `INSERT OVERWRITE INTO SELECT` 不能与实时写入（INSERT INTO、REPLACE INTO、DELETE、UPDATE）混用，否则实时写入的数据会被丢弃。

4.6. UPDATE

`UPDATE` 用于更新数据。

语法

```
UPDATE table_reference
SET assignment_list
[WHERE where_condition]
[ORDER BY ...]
```

注意事项

执行 `UPDATE` 命令时，要求表中存在主键。

示例

将CUSTOMER表中 customer_id ='2369' 客户的姓名更改为黄蓉。

```
update customer set customer_name ='黄蓉' where customer_id ='2369';
```

4.7. DELETE

DELETE 用于删除表中的记录。

语法

```
DELETE FROM table_name
[ WHERE condition ]
```

注意事项

- 执行 DELETE 命令时，表中必须存在主键。
- DELETE 暂不支持使用表的别名。
- 不建议通过 DELETE 命令做全表、全分区删除，建议使用 TRUNCATE TABLE 、 TRUNCATE PARTITION 。

示例

- 删除CUSTOMER表中 name 为 张三 的数据。

  ```
  DELETE FROM customer WHERE customer_name='张三';
  ```

- 删除CUSTOMER表的中多行。

  ```
  DELETE FROM customer WHERE age<18;
  ```

4.8. TRUNCATE TABLE

TRUNCATE TABLE 用于清空表数据或者表分区数据。

语法

- 清空表数据

  ```
  TRUNCATE TABLE db_name.table_name;
  ```

- 清空表中的指定分区

  ```
  TRUNCATE TABLE db_name.table_name PARTITION partition_name;
  ```

  分区名的数据类型为bigint，您可以通过以下SQL获取某个表的所有分区名。

```
select partition_name from information_schema.partitions where table_name = 'your_table_name' order by partition_name desc limit 100;
```

注意事项

> 说明
>
> - 数据库备份期间无法执行，执行 TRUNCATE TABLE 将会报错。
> - 执行 TRUNCATE TABLE 命令清空表中的数据，表结构不会被删除。

示例

- 清空CUSTOMER表中的数据。

```
TRUNCATE TABLE adb_demo.customer;
```

- 清空表中的指定分区。

```
TRUNCATE TABLE adb_demo.customer partition 20170103,20170104,20170108;
```

4.9. KILL PROCESS

KILL PROCESS 用于终止正在运行的PROCESS。

语法

```
KILL PROCESS process_id
```

参数

process_id：来源于SHOW PROCESSLIST返回结果中的ProcessId字段。

权限

- 默认您可以通过 KILL PROCESS 终止您当前账号下正在运行的PROCESS。
- 高权限账号通过GRANT语句授予普通账号PROCESS权限，普通账号可以终止集群下所有用户正在运行的PROCESS。

```
GRANT process on *.* to account_name;
```

4.10. SHOW PROCESSLIST

SHOW PROCESSLIST 用于查看正在运行的PROCESS。

> 说明 您也可以通过 INFORMATION_SCHEMA PROCESSLIST 表查看正在运行的PROCESS。

语法

```
SHOW [FULL] PROCESSLIST
```

返回参数

执行 `SHOW FULL PROCESSLIST` 或者 `SHOW PROCESSLIST` 后，返回结果中包含以下参数。

- Id：PROCESS的Id。
- ProcessId：任务的唯一标识，执行 `KILL PROCESS` 时需要使用ProcessId。
- User：当前用户。
- Host：显示发出这个语句的客户端的主机名，包含IP和端口号。
- DB：显示该PROCESS目前连接的是哪个数据库。
- Command：显示当前连接所执行的命令，即休眠（sleep）、查询（query）以及连接（connect）三种类型的命令。
- Time：显示Command执行的时间，单位为秒。
- State：显示当前连接下SQL语句的执行状态。
- Info：显示SQL语句。

> 说明 如果不使用 `FULL` 关键字，只能查看每个记录中Info字段的前100个字符。

权限

- 默认您可以通过 `SHOW PROCESSLIST` ，查看您当前账号下正在运行的PROCESS。
- 高权限账号通过GRANT语句授予普通账号PROCESS权限，普通账号可以查看集群下所有用户正在运行的PROCESS。

```
GRANT process on *.* to account_name;
```

5.SELECT

5.1. 语法

`SELECT` 语句用于从一个或多个表中查询数据，具体语法如下所示。

```
[ WITH with_query [, ...] ]
SELECT
[ ALL | DISTINCT ] select_expr [, ...]
[ FROM table_reference [, ...] ]
[ WHERE condition ]
[ GROUP BY [ ALL | DISTINCT ] grouping_element [, ...] ]
[ HAVING condition]
[ WINDOW window_name AS (window_spec) [, window_name AS (window_spec)] ...]
[ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
[ ORDER BY {column_name | expr | position} [ASC | DESC], ... [WITH ROLLUP]]
[LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

- `table_reference` ：查询的数据源，可以是表、视图、关联表或者子查询。
- 表名和列名不区分大小写。
- 表名和列名中如果含有关键字或者空格等字符，可以使用反引号（``）将其引起来。

WHERE

`WHERE` 关键字后跟 `BOOLEAN` 表达式，用于从表中查询满足条件的数据。例如，在CUSTOMER表中查询 `customer_id` 为 `2368` 的顾客信息。

```
SELECT * FROM CUSTOMER where customer_id=2368;
```

ALL和DISTINCT

ALL和DISTINCT关键字用于指定查询结果是否返回重复的行，默认值为ALL，即返回所有匹配的行，DISTINCT将从结果集中删除重复的行。

```
SELECT col1, col2 FROM t1;
SELECT DISTINCT col1, col2 FROM t1;
```

以下为SELECT中的其他关键字用法。

- LIMIT
- UNION、INTERSECT和EXCEPT
- WITH
- GROUP BY
- HAVING

- JOIN
- ORDER BY
- 子查询

5.2. WITH

本文介绍如何在 SELECT 语句中使用 WITH 子句。

查询中可以使用 WITH 子句来创建通用表达式（Common Table Express， 简称CTE）， WITH 子句定义的子查询，供 SELECT 查询引用。 WITH 子句可以扁平化嵌套查询或者简化子查询， SELECT 只需执行一遍子查询，提高查询性能。

> 说明
> - CTE是一个命名的临时结果集，仅在单个SQL语句（例如SELECT、INSERT或DELETE）的执行范围内存在。
> - CTE仅在查询执行期间持续。

注意事项

- CTE之后可以接SQL语句（例如 SELECT 、 INSERT 或 UPDATE 等）或者其他的CTE（只能使用一个 WITH ），多个CTE中间用逗号（,）分隔，否则CTE将失效。
- CTE语句中暂不支持分页功能。

WITH使用方法

- 以下两个查询等价

```
SELECT a, b
FROM (SELECT a, MAX(b) AS b FROM t GROUP BY a) AS x;
```

```
WITH x AS (SELECT a, MAX(b) AS b FROM t GROUP BY a)
SELECT a, b FROM x;
```

- WITH子句可用于多子查询：

```
WITH
t1 AS (SELECT a, MAX(b) AS b FROM x GROUP BY a),
t2 AS (SELECT a, AVG(d) AS d FROM y GROUP BY a)
SELECT t1.*, t2.*
FROM t1 JOIN t2 ON t1.a = t2.a;
```

- WITH子句中定义的关系可以互相连接

```
WITH
x AS (SELECT a FROM t),
y AS (SELECT a AS b FROM x),
z AS (SELECT b AS c FROM y)
SELECT c FROM z;
```

5.3. GROUP BY

`GROUP BY` 子句用于对查询结果进行分组，可以在 `GROUP BY` 中使用 `GROUPING SETS` 、 `CUBE` 、 `ROLLUP` 以不同的形式展示分组结果。

```
GROUP BY expression [, ...]
```

GROUPING SETS

`GROUPING SETS` 用于在同一结果集中指定多个 `GROUP BY` 选项，作用相当于多个 `GROUP BY` 查询的 `UNION` 组合形式。

```
SELECT origin_state, origin_zip, destination_state, sum(package_weight)
FROM shipping
GROUP BY GROUPING SETS (
(origin_state),
(origin_state, origin_zip),
(destination_state));
```

上述示例等同于：

```
SELECT origin_state, NULL, NULL, sum(package_weight)
FROM shipping GROUP BY origin_state
UNION ALL
SELECT origin_state, origin_zip, NULL, sum(package_weight)
FROM shipping GROUP BY origin_state, origin_zip
UNION ALL
SELECT NULL, NULL, destination_state, sum(package_weight)
FROM shipping GROUP BY destination_state;
```

CUBE

`CUBE` 用于列出所有可能的分组集。

```
SELECT origin_state, destination_state, sum(package_weight)
FROM shipping
GROUP BY origin_state, destination_state WITH CUBE
```

上述示例等同于：

```
SELECT origin_state, destination_state, sum(package_weight)
FROM shipping
GROUP BY GROUPING SETS (
(origin_state, destination_state),
(origin_state),
(destination_state),
())
```

ROLLUP

`ROLLUP` 可以以层级的方式列出分组集。

```
SELECT origin_state, origin_zip, sum(package_weight)
FROM shipping
GROUP BY ROLLUP (origin_state, origin_zip)
```

上述示例等同于：

```
SELECT origin_state, origin_zip, sum(package_weight)
FROM shipping
GROUP BY GROUPING SETS ((origin_state, origin_zip), (origin_state), ())
```

注意事项

- 查询中必须使用标准聚合函数（ `SUM` 、 `AVG` 或 `COUNT` ）声明非分组列，否则无法使用 `GROUP BY` 子句。
- `GROUP BY` 中的列或表达式列表必须与查询列表中的非聚合表达式的列相同。

示例

例如，以下查询列表中包含两个聚合表达式，第一个聚合表达式使用 `SUM` 函数，第二个聚合表达式使用 `COUNT` 函数，其余两列（ `LISTID` 、 `EVENTID` ）声明为分组列。

```
select listid, eventid, sum(pricepaid) as revenue,
count(qtysold) as numtix
from sales
group by listid, eventid
order by 3, 4, 2, 1
limit 5;
listid | eventid | revenue | numtix
-------+---------+---------+--------
89397| 47 | 20.00 | 1
106590 | 76 | 20.00 | 1
124683 | 393 | 20.00 | 1
103037 | 403 | 20.00 | 1
147685 | 429 | 20.00 | 1
(5 rows)
```

`GROUP BY` 子句中的表达式也可以使用序号来引用所需的列。

例如，上述示例可改写为以下形式。

```
select listid, eventid, sum(pricepaid) as revenue,
count(qtysold) as numtix
from sales
group by 1,2
order by 3, 4, 2, 1
limit 5;

listid | eventid | revenue | numtix
-------+---------+---------+--------
89397 | 47 | 20.00 | 1
106590 | 76 | 20.00 | 1
124683 | 393 | 20.00 | 1
103037 | 403 | 20.00 | 1
147685 | 429 | 20.00 | 1
```

5.4. HAVING

`HAVING` 子句与聚合函数以及 `GROUP BY` 子句一起使用，在分组和聚合计算完成后，HAVING子句对分组进行过滤，去掉不满足条件的分组。

```
[ HAVING condition ]
```

注意事项

- `HAVING` 条件中引用的列必须为分组列或引用了聚合函数结果的列。
- `HAVING` 子句必须与聚合函数以及 `GROUP BY` 子句一起使用，用于对 `GROUP BY` 分组进行过滤，去掉不满足条件的分组。

示例

在CUSTOMER表中，进行分组查询，查询账户余额大于指定值的记录。

```
SELECT count(*), mktsegment, nationkey,
CAST(sum(acctbal) AS bigint) AS totalbal
FROM customer
GROUP BY mktsegment, nationkey
HAVING sum(acctbal) > 5700000
ORDER BY totalbal DESC;

_col0 | mktsegment | nationkey | totalbal
-------+------------+-----------+----------
1272 | AUTOMOBILE | 19 | 5856939
1253 | FURNITURE | 14 | 5794887
1248 | FURNITURE | 9 | 5784628
1243 | FURNITURE | 12 | 5757371
1231 | HOUSEHOLD | 3 | 5753216
1251 | MACHINERY | 2 | 5719140
1247 | FURNITURE | 8 | 5701952
```

5.5. JOIN

以下查询由 `FROM` 子句中的两个子查询联接组成，查询不同类别活动（音乐会和演出）的已售门票数和未售门票数。

```
join_table:
table_reference [INNER] JOIN table_factor [join_condition]
| table_reference {LEFT|RIGHT|FULL} [OUTER] JOIN table_reference join_condition
| table_reference CROSS JOIN table_reference [join_condition])

table_reference:
table_factor
| join_table

table_factor:
tbl_name [alias]
| table_subquery alias
| ( table_references )

join_condition:
ON expression
```

示例

```
select catgroup1, sold, unsold
from
(select catgroup, sum(qtysold) as sold
from category c, event e, sales s
where c.catid = e.catid and e.eventid = s.eventid
group by catgroup) as a(catgroup1, sold)
join
(select catgroup, sum(numtickets)-sum(qtysold) as unsold
from category c, event e, sales s, listing l
where c.catid = e.catid and e.eventid = s.eventid
and s.listid = l.listid
group by catgroup) as b(catgroup2, unsold)
on a.catgroup1 = b.catgroup2
order by 1;
```

5.6. LIMIT

`LIMIT` 子句用于限制最终结果集的行数， `LIMIT` 子句中通常会携带一个或两个数字参数，第一个参数指定要返回数据行的第一行的偏移量，第二个参数指定要返回的最大行数。

以下示例查询ORDERS表，通过LIMIT限制返回结果，仅返回5行数据。

```
SELECT orderdate FROM orders LIMIT 5;
-------------
o_orderdate
-------------
1996-04-14
1992-01-15
1995-02-01
1995-11-12
1992-04-26
```

以下示例查询CUSTOMER表，按照创建时间排序，返回第3个到第7个客户的信息。

```
SELECT * FROM customer ORDER BY create_date LIMIT 2,5
```

5.7. ORDER BY

`ORDER BY` 子句用于对查询结果进行排序， `ORDER BY` 中每个表达式由列名或列序号（从1开始）组成。

```
[ ORDER BY expression
[ ASC | DESC ]
[ NULLS FIRST | NULLS LAST ]
[ LIMIT { count | ALL } ]
```

5.8. 子查询

以下示例查询门票销量排名前10的卖家， `WHERE` 子句中包含一个表子查询，子查询生成多个行，每行包含一列数据。

> **说明** 表子查询可以包含多个列和行。

```
select firstname, lastname, cityname, max(qtysold) as maxsold
from users join sales on users.userid=sales.sellerid
where users.cityname not in(select venuecity from venue)
group by firstname, lastname, cityname
order by maxsold desc, cityname desc
limit 10;

firstname | lastname | cityname | maxsold
-----------+----------+----------------+---------
Noah | Guerrero | Worcester | 8
Isadora | Moss | Winooski | 8
Kieran | Harrison | Westminster | 8
Heidi | Davis | Warwick | 8
Sara | Anthony | Waco | 8
Bree | Buck | Valdez | 8
Evangeline | Sampson | Trenton | 8
Kendall | Keith | Stillwater | 8
Bertha | Bishop | Stevens Point | 8
Patricia | Anderson | South Portland | 8
```

5.9. UNION、INTERSECT和EXCEPT

`UNION` 、 `INTERSECT` 和 `EXCEPT` 用于将多个查询结果集进行组合，从而得到一个最终结果。

语法

```
query
{ UNION [ ALL ] | INTERSECT | EXCEPT }
query
```

参数

- `UNION` ：返回两个查询表达式的集合运算。
- `UNIO ALL` ： `ALL` 关键字用于保留 `UNION` 中产生的重复行。
- `INTERSECT` ：返回只有在两个集合中同时出现的行，返回结果将删除两个集合中的重复行。
- `EXCEPT` ：先删除两个集合中重复的数据行，返回只在第一个集合中出现且不在第二个集合中出现的所有行。

计算顺序

- `UNION` 和 `EXCEPT` 集合运算符为左关联，如果未使用圆括号来改变计算顺序，则按照从左到右的顺序进行集合运算。

例如，以下查询中，首先计算 T1 和 T2 的 UNION ，然后对 UNION 结果执行 EXCEPT 操作。

```
select * from t1
union
select * from t2
except
select * from t3
order by c1;
```

- 在同一查询中，组合使用集合运算符时， INTERSECT 运算符优先于 UNION 和 EXCEPT 运算符。

  例如，以下查询先计算 T2 和 T3 的交集，然后将计算得到的结果与 T1 进行并集。

```
select * from t1
union
select * from t2
intersect
select * from t3
order by c1;
```

- 可以使用圆括号改变集合运算符的计算顺序。

  以下示例中，将 T1 和 T2 的并集结果与 T3 执行交集运算。

```
(select * from t1
union
select * from t2)
intersect
(select * from t3)
order by c1;
```

5.10. CROSS JOIN

通过cross join语句实现列转行功能。能将某一列中的多个元组转换成多行。

数组结构转多行

将某一列转成数组，然后转成多行，示例如下：

```
####建库
CREATE DATABASE mydb;
USE mydb;

###建表
CREATE TABLE test(
userid INT
,user_name VARCHAR
,product VARCHAR
) distributed by hash(userid);

###插入两行测试数据
INSERT INTO test VALUES
(1,'aaa','cat,mat,bat'),(2,'bbb','dog,pog,fog');

####查询数据，产品列转行，其中temp_table为临时表名可以更换，col为返回列名
SELECT userid, col
FROM (select userid, split(product,',') as numbers_array from test)
CROSS JOIN UNNEST(numbers_array) as temp_table(col);

###查询结果
userid col
1 cat
1 mat
1 bat
2 dog
2 pog
2 fog
```

6.CREATE USER

`CREATE USER` 用于创建账号。

```
CREATE USER
[if not exists] user [auth_option] [, [if not exists] user [auth_option]] ...
```

注意事项

通过 `CREATE USER` 创建账号时，您需要拥有 `CREATE_USER` 权限。

示例

创建账号account2，密码为Account2。

```
CREATE USER if not exists 'account2' IDENTIFIED BY 'Account2';
```

7.GRANT

`GRANT` 用于为用户授权。

```
GRANT
priv_type [(column_list)]
[, priv_type [(column_list)]] ...
ON priv_level
TO user [auth_option]
[WITH {GRANT OPTION}]
```

参数

- `priv_type`：权限类型，详情请参见权限模型。
- `column_list`：可选参数，当 `priv_type` 为 `SELECT` 时，可以填写表中的列名，针对具体列授予 `SELECT` 授权。
- `priv_level`：被授权对象层级。
  - `*.*`：整个集群级别的权限。
  - `db_name.*`：数据库级别的权限。
  - `db_name.table_name` 或者 `table_name`：表级别的权限。

注意事项

通过 `GRANT` 授权用户时，您需要拥有 `GRANT OPTION` 权限。

示例

- 为账号account2授予集群级别的 `all` 权限。

  ```
  GRANT all ON *.* TO 'account2';
  ```

- 为账号account3授予数据库级别的 `all` 权限。

  ```
  GRANT all ON adb_demo.* TO 'account3';
  ```

- 可以通过 `GRANT` 创建并授权账号。

  例如，创建全局DML帐号。

  ```
  GRANT insert,select,update,delete on *.* to 'test'@'%' identified by 'Testpassword1';
  ```

  创建数据库级别DML帐号。

  ```
  GRANT insert,select,update,delete on adb_demo.* to 'test123' identified by 'Testpassword123';
  ```

- 创建账号并授予列级别的 `SELECT` 权限。

```
GRANT select (customer_id, sex) ON customer TO 'test321' identified by 'Testpassword321';
```

8.REVOKE

`REVOKE` 用于撤销用户权限。

```
REVOKE
priv_type [(column_list)]
[, priv_type [(column_list)]] ...
ON [object_type] priv_level
FROM user
```

参数

- `priv_type`：权限类型，详情请参见权限模型。
- `column_list`：可选参数，当 `priv_type` 为 `SELECT` 时，可以填写表中的列名，针对具体列撤销 `SELECT` 权限。
- `priv_level`：被撤销权限的对象层级。
  - `*.*`：整个集群级别的权限。
  - `db_name.*`：数据库级别的权限。
  - `db_name.table_name` 或者 `table_name`：表级别的权限。

注意事项

通过 `REVOKE` 撤销用户权限时，您需要拥有 `GRANT OPTION` 权限。

示例

撤销账号account3数据库级别的 `all` 权限。

```
REVOKE all ON adb_demo.* FROM 'account3';
```

9.查询用户

AnalyticDB for MySQL兼容MySQL数据库，AnalyticDB for MySQL中也有一个名为MySQL的内置数据库，该数据库中存储的是AnalyticDB for MySQL中的用户、权限信息以及存储过程等。您可以通过SELECT语句查询AnalyticDB for MySQL中的用户信息。

注意事项

AnalyticDB for MySQL中，只有通过高权限账号查询用户信息。

> 说明 AnalyticDB for MySQL中的高权限账号相当于MySQL中的root账号。

示例

```
USE MYSQL;
SELECT User, Host, Password FROM mysql.user;
+---------+-----+-------------------------+
| User | Host| Password |
+---------+-----+-------------------------+
| account1| % | *61f3777f02386598cd***** |
| account2| % | *0fe79c07e168cabc99***** |
```

10.RENAME USER

RENAME USER 用于更改用户名。

语法

```
RENAME USER old_user TO new_user
[, old_user TO new_user] ...
```

示例

```
RENAME USER account2 TO account_2;
SELECT User, Host, Password FROM mysql.user;
+----------+-----+---------------------- -+
| User | Host| Password |
+----------+-----+------------------------+
| account2 | % | *61f3777f02386598cda****|
| account_2| % | *0fe79c07e168cabc99b****|
```

11.DROP USER

`DROP USER` 用于删除用户。

语法

```
DROP USER [if exists] user [, [if exists] user] ...
```

注意事项

通过 `DROP USER` 删除用户时，您需要拥有 `CREATE_USER` 权限。

示例

```
DROP USER account_2;
```

12.SHOW

`SHOW` 用于查看数据库相关信息，例如数据库列表、数据库中的表等。

SHOW DATABASES

- 语法

```
SHOW DATABASES [EXTRA];
```

指定 `EXTRA` 参数时，将输出关于数据库的更多信息，例如创建者ID、数据库连接信息等。

- 示例

```
SHOW DATABASES EXTRA;
+--------------------+
| Database |
+--------------------+
| adb_demo |
| MYSQL |
| INFORMATION_SCHEMA |
| adb_demo2 |
```

SHOW TABLES

查看用户当前数据库中的表。

- 语法

```
SHOW TABLES [IN db_name];
```

- 示例

```
SHOW TABLES IN adb_demo;
+------------------------------+
| Tables_in_adb_demo |
+------------------------------+
| customer |
| customer2 |
| new_customer |
| test_table |
| v |
```

SHOW COLUMNS

查看表的列信息。

- 语法

```
SHOW COLUMNS IN db_name.table_name;
```

- 示例

```
SHOW COLUMNS IN adb_demo.test_table;
```

SHOW CREATE TABLE

查看表的建表语句。

- 语法

```
SHOW CREATE TABLE db_name.table_name;
```

- 示例

```
SHOW CREATE TABLE adb_demo.customer;
```

SHOW GRANTS

查看当前登录用户的权限。

```
SHOW GRANTS;
```

13.JSON

为赋能用户、降低用户处理半结构化数据的难度，ADB MySQL版提供了半结构化数据检索功能即JSON索引。

背景信息

大数据时代结构化数据检索已有多元化的、丰富的解决方案。但是，事实上大多数大数据都是半结构化，并且半结构化数据的数据量仍旧急剧增长。理解和分析半结构化数据的难度比结构化数据大很多，急需成熟的解决方案处理半结构化数据。为赋能用户、降低用户处理半结构化数据的难度，云原生数据仓库MySQL版（简称ADB MySQL版）提供了半结构化数据检索功能，即JSON索引。

注意事项

- 创建表时指定某一列的数据类型为JSON之后，AnalyticDB for MySQL自动创建JSON索引，且暂不支持更改JSON索引。
- JSON支持的数据类型有Boolean、Number、Varchar、Array、Object、Null，Number不能超过Double取值范围，否则会写入报错。
- AnalyticDB for MySQL支持标准JSON格式，写入JSON串时必须严格符合标准JSON格式规范。
- 增加列时不支持设置非Null Default值。

创建表

例如，以下示例中的 `vj` 字段为JSON类型，建表成功后AnalyticDB for MySQL自动为 `vj` 列构建JSON索引。

```
CREATE TABLE json_test(
id int,
vj json COMMENT 'json类型，自动创建索引'
)
DISTRIBUTED BY HASH(id);
```

JSON格式要求

写入数据时，AnalyticDB for MySQL对JSON数据中的属性键 `key` 和属性值 `value` 有以下要求。

- **属性键key**：必须使用双引号（ `""` ）将 `key` 引起来。
- **属性值value**：
  - 如果 `value` 是字符串类型，必须使用双引号（ `""` ）将 `value` 引起来。

    如果 `value` 是字符串类型，且 `value` 中包含双引号，需要做转义处理。例如， `value` 为 `{"addr":"xyz"ab"c"}` 时，正确的写法为 `{"addr":"xyz\"ab\"c"}` 。
  - 如果 `value` 是数值类型，直接写数据，不能使用双引号（ `""` ）将 `value` 引起来。
  - 如果 `value` 是 `Boolean` 类型，直接写 `TRUE` 或者 `FALSE` ，不能写成 `1` 或者 `0` 。
  - 如果 `value` 是 `Null` ，直接写 `Null` 。
- AnalyticDB for MySQL支持JSON数组写入：包括PLAIN ARRAY及嵌套ARRAY。

例如 `{"hobby":["basketball", "football"]}` 和 `{"addr":[{"city":"beijing", "no":0}, {"city":"shenzhen", "no":0}]}` 。

写入数据

向表中写入数据时，JSON类型字段的写入方式与VARCHAR类型字段的写入方式相同，即在JSON串两端使用单引号引起来。

以下SQL示例包含多种JSON数据格式，供您参考使用。

```
insert into json_test values(0, '{"id":0, "name":"abc", "age":0}');
insert into json_test values(1, '{"id":1, "name":"abc", "age":10, "gender":"femal"}');
insert into json_test values(2, '{"id":3, "name":"xyz", "age":30, "company":{"name":"alibaba", "place":"hangzhou"}}');
insert into json_test values(3, '{"id":5, "name":"abc", "age":50, "company":{"name":"alibaba", "place":"america"}}');
insert into json_test values (4, '{"a":1, "b":"abc-char", "c":true}');
insert into json_test values (5, '{"uname":{"first":"lily", "last":"chen"}, "addr":[{"city":"beijing", "no":1}, {"city":"shenzhen", "no":0}], "age":10, "male":true, "like":"fish", "hobby":["basketball", "football"]}');
```

查询数据

查询数据时，AnalyticDB for MySQL支持使用函数 `json_extract` 。

- **语法**

  ```
  json_extract(json, jsonpath)
  ```

- **命令说明**

  从 `json` 中返回 `jsonpath` 指定的值。

- **参数说明**
  - `json` 为JSON列的列名。
  - `jsonpath` ，通过点号（ `.` ）进行分割的JSON属性键 `key` 的路径，其中 `$` 表示最外层的路径。

  更多JSON函数用法请参见JSON函数。

- **示例**
  - **基本查询**

    ```
    select json_extract(vj,'$.name') from json_test where id=1;
    ```

  - **等值查询**

    ```
    select id, vj from json_test where json_extract(vj, '$.name') = 'abc';
    select id, vj from json_test where json_extract(vj, '$.c') = true;
    select id, vj from json_test where json_extract(vj, '$.age') = 30;
    select id, vj from json_test where json_extract(vj, '$.company.name') = 'alibaba';
    ```

- **范围查询**

```
select id, vj from json_test where json_extract(vj, '$.age') > 0;
select id, vj from json_test where json_extract(vj, '$.age') < 100;
select id, vj from json_test where json_extract(vj, '$.name') > 'a' and json_extract(vj, '$.name') < 'z'
;
```

- **IS NULL/IS NOT NULL查询**

```
select id, vj from json_test where json_extract(vj, '$.remark') is null;
select id, vj from json_test where json_extract(vj, '$.name') is null;
select id, vj from json_test where json_extract(vj, '$.name') is not null;
```

- **IN 查询**

```
select id, vj from json_test where json_extract(vj, '$.name') in ('abc','xyz');
select id, vj from json_test where json_extract(vj, '$.age') in (10,20);
```

- **LIKE查询**

```
select id, vj from json_test where json_extract(vj, '$.name') like 'ab%';
select id, vj from json_test where json_extract(vj, '$.name') like '%bc%';
select id, vj from json_test where json_extract(vj, '$.name') like '%bc';
```

- **ARRAY查询**

```
select id, vj from json_test where json_extract(vj, '$.addr[0].city') = 'beijing' and json_extract(vj, '$.
addr[1].no') = 0;
select id, vj from json_test where json_extract(vj, '$.addr[1].city') = 'shenzhen' and json_extract(js
on_test, '$.addr.no') = 0;
```

> **说明** 查询ARRAY数据时，支持下标取值，暂不支持遍历整个数组。

> **说明** 同一个 `key` 可以对应不同类型的 `value` 。
>
> 例如示例数据 `"key": 1` ， `"key": "1"` ， `json_extract(col, '$.key')=1` 将返回数字 `"key": 1` ； `json_extract(col, '$.key')='1'` 将返回字符串 `"key": "1"` 。