分析型数据库

用户指南

为了无法计算的价值 | 阿里云

用户指南

随着企业IT和互联网系统的发展，越来越多的数据被产生了。数据的量的积累带来了质的飞跃，使得数据系统从业务系统的一部分演变得愈发独立，通过对数据的分析和挖掘产生自己独特的价值。在业务系统中，我们通常使用的是OLTP（OnLine Transaction Processing ，联机事务处理）系统，如MySQL, MicroSoft SQL Server等关系数据库系统。这些关系数据库系统擅长事务处理，在数据操作中保持着很强的一致性和原子性，能够很好的支持频繁的数据插入和修改，但是，一旦需要进行计算的数据量过大，达到数千万甚至数十亿条，或需要进行的计算非常复杂的情况下，OLTP类数据库系统便力不从心了。

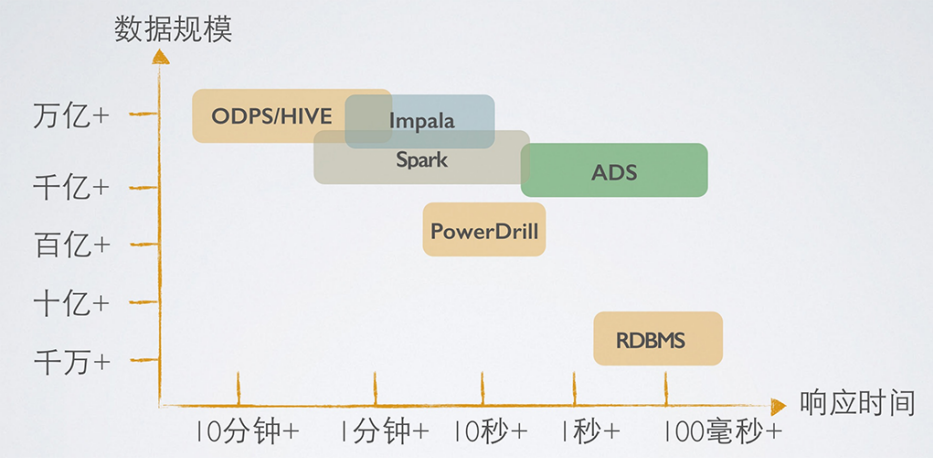


图0-1 分析型数据库和主流数据系统的对比（数据未经严格测试，仅供参考）

这个时候，我们便需要OLAP（On-Line Analytical Processing，联机分析处理）系统，来进行处理。从广义上，OLAP系统是针对OLTP系统而言的，暨不特别关心对数据进行输入、修改等事务性处理，而是关心对已有的大量数据进行多维度的、复杂的分析的一类数据系统。而在具体的产品中，我们通常将OLAP系统分为MOLAP、ROLAP和HOLAP三种。多维OLAP（Multi-Dimensional OLAP，简称MOLAP），是预先根据数据需要分析的维度进行建模，在数据的物理存储层面以“立方体”(Cube)的结构进行存储，具有查询速度快等的优点，但是数据必须预先建模，无法依据使用者的意愿进行即时的修改。而关系型OLAP（RelationalOLAP，简称ROLAP），则使用类似关系数据库的模型进行数据存储，通过类似SQL等语言进行查询和计算，优点是数据查询计算自由，可以灵活的根据使用者的要求进行分析，但是缺点是在海量数据的情况下分析计算缓慢。至于HOLAP，则是MOLAP和ROLAP的混合模式。而阿里云分析型数据库（原名：分析数据库服务ADS），则是一套RT-OLAP（Realtime OLAP，实时OLAP）系统。在数据存储模型上，采用自由灵活的关系模型存储，可以使用SQL进行自由灵活的计算分析，无需预先建模，而利用云计算技术，分析型数据库可以在处理百亿条甚至更多量级的数据上达到甚至超越MOLAP类系统的处理性能，真正实现百亿数据毫秒级计算。

阿里云分析型数据库让海量数据和实时与自由的计算可以兼得，实现了速度驱动的大数据商业变革。一方面，分析型数据库拥有快速处理百亿级别的大数据的能力，使得数据分析中使用的数据可以不再是抽样的，而是业务系统中产生的全量数据，使得数据分析的结果具有最大的代表性。而更重要的是，分析型数据库采用云计

算技术，拥有强大的实时计算能力，通常可以在数百毫秒内完成十亿百亿的数据计算，使得使用者可以根据自己的想法在海量数据中自由的进行探索，而不是根据预先设定好的逻辑查看已有的数据报表。同时，由于分析型数据库能够支撑较高并发查询量，并且通过动态的多副本数据存储计算技术来保证较高的系统可用性，所以能够直接作为面向最终用户（End User）的产品（包括互联网产品和企业内部的分析产品）的后端系统。如淘宝数据魔方、淘宝指数、快的打车、阿里妈妈达摩盘（DMP）、淘宝美食频道等拥有数十万至上千万最终用户的互联网业务系统中，都使用了分析型数据库。分析型数据库作为海量数据下的实时计算系统，给使用者带来极速自由的大数据OLAP分析体验，最终期望为大数据行业带来巨大的变革。我们欢迎有各类相关需求的用户使用分析型数据库并向我们提出宝贵的建议。

第一章 快速开始

在公共云上，满足开通条件的用户可以在 https://buy.aliyun.com/ads 上进行按量付费开通，或访问 https://common-buy.aliyun.com/?commodityCode=prepaid_ads#/buy 购买包月套餐。

在专有云中，开通分析型数据库服务的方式请咨询您的系统管理员或运维人员。

分析型数据库中，需要通过DMS for Analytic DB页面进行创建数据库。

在目前的分析型数据库版本中，创建数据库时，需要填写数据库名，注意这个数据库名称需要在分析型数据库全部集群上全局唯一。然后选择分析型数据库的Region所在地，如杭州、北京等。

分析型数据库以ECU（弹性计算单元）作为资源计量的最小单位。ECU（弹性计算单元）拥有多种型号，每种型号的ECU，标识着不同的vCPU核数、内存大小、磁盘空间大小。用户在创建数据库时需要根据自己的需求选择这个数据库的ECU型号，以及初始的ECU数量（必须是偶数个，至少两个），ECU型号DB创建后不可修改，ECU数量可以在使用中随时调整（扩容/缩容），关于ECU的详细信息，详见 2.4节 ECU详解。

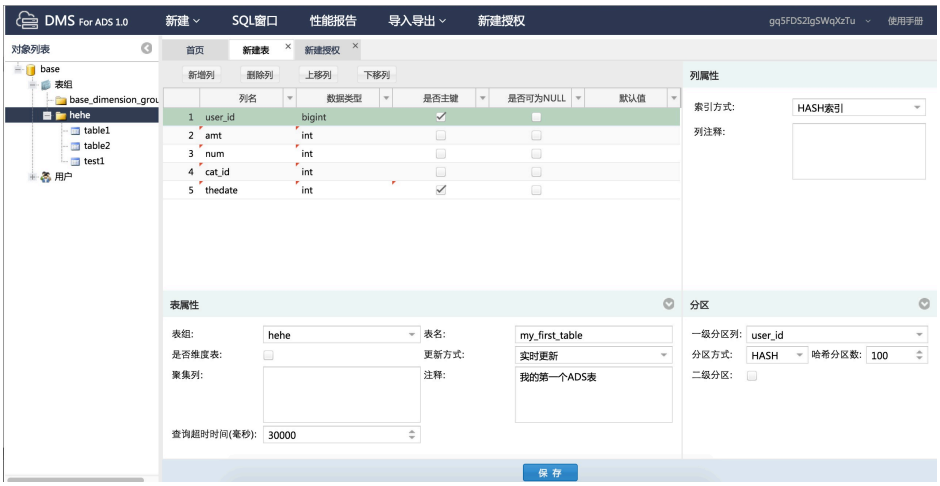
填好所有选项后，点击创建数据库，若返回错误，则根据错误提示进行修正（通常是数据库名称重复或不符合规范，或提交的ECU资源量超过了分析型数据库允许的最大限制），否则则创建成功。十分钟以内DMS界面中会显示出新的数据库的连接地址。

前文中，我们已经创建了一个分析型数据库数据库，分析型数据库采用关系模型存储数据，也就是使用二维表来进行数据的组织和存储。像MySQL一样，将数据灌入分析型数据库前需要需要建立对应的数据表。而分析型数据库为了管理相关联的数据表，又引入了表组的概念。

表组是数据库的下一级实体，也是表的上一级。在分析型数据库中一个表必须从属于一个表组。关于表组的具体介绍我们会在3.2节中进行。在这里，我们首先创建一个表组。

在DMS for 分析型数据库中，右击左侧表组对象，选择新建表组，弹出新建表组对话框，弹出如下图所示内容，我们填写表组名为test_group，其余参数先暂时使用默认值。

点击确定建立好表组后，我们右击表组，选择新建表。在这里，我们根据测试数据的情况，建立一张有五列和一级分区的数据表。如下图所示：



在分析型数据库中，数据根据分区列进行分布式的存储和计算。举例来讲，我们在这里的原始数据是按照user_id较均匀的进行分布的，所以我们指定按照user_id进行hash分区，分区数调整为40个（一般来讲，每个分区的数据不超过800万条为宜，当然也不绝对，分区数不能超过256个）。表名和列名根实际情况填写，目前需要和源头数据表的字段名称一致。

另外，如果这个表的数据来源是批量的从其他系统导入（例如从ODPS），那么在更新方式一项，则选择批量更新，随后阅读1.4节中的导入数据部分。如果这个表的数据来源来自于直接的insert插入，那么在更新方式一项选择实时更新，随后阅读1.4节中的插入数据一项。

分析型数据库拥有强大的自动索引功能，用户在创建表时通常无需关心一个列的索引情况，分析型数据库会根据实际数据的分布情况来自动进行索引。所以这里我们先不调整列的索引设置。而表名任意填写，表组名选择我们刚刚创建的test_group，然后点击保存，弹出实际的建表DDL供校验。

有关表和列的详细说明，我们会在3.3和3.4节中稍后叙述。创建表完毕后，右击已有的表可以进行编辑。

分析型数据库支持多种接入数据的方式，您可以直接将数据通过insert/delete SQL写入实时表（详见使用手册第四章），或通过Kettle等ETL工具将本地文件写入分析型数据库，或是通过阿里云数据传输从阿里云RDS中实时同步数据变更（见使用手册8.5节），或者建立批量导入表从阿里云MaxCompute（原名ODPS）大批量的导入数据。

如果在建立表时选择数据来源是批量导入，则分析型数据库提供多种数据导入的方式，如通过data pipeline系列命令（详见5.1），等方式。在这里，作为测试使用，我们通过控制台界面进行数据导入。

在操作导入数据之前，我们需要对数据的来源表进行授权，例如数据的来源表在odps上，在公有云上则需要在ODPS上对 garuda_build@aliyun.com 与 garuda_data@aliyun.com 授予describe和select权限（各个专有云授权的账号名参照专有云的相关配置文档，不一定是这个账号）。另外要注意，分析型数据库目前仅允许操作者导入自身为Project Owner的ODPS Project中，或者操作者是ODPS表的Table Creator的数据。

进入DMS页面，选择菜单栏上的导入按钮，弹出导入对话框。这里我们的数据源表在阿里云ODPS上。因此数据导入路径按照 “odps://project_name/table_name/partition_spec” 的格式来填写。关于导入数据的分区信息，在仅有Hash分区的情况下iDB Cloud会帮我们自动识别并填写。填写完毕后，如下图所示，点击“确定”按钮。

之后页面会展示导入状态一览，分析型数据库会对导入任务进行调度，根据当前系统繁忙情况和待导入数据的大小和结构不同，二十分钟至数个小时内数据导入会结束。

如果建表时数据来源选择的是实时写入，那么则可以在建表后直接在SQL窗口中编写SQL:

```
insert into my_first_table (user_id,amt,num,cat_id,thedate) values (12345, 80, 900, 1555, 20140101);
```

实时写入的表刚刚建立后，会有一分钟左右（公共云当前版本：0.9.*版本数据，若使用0.8版本的ADS一般为十五分钟左右）的准备时间，这时候写入的数据需要在准备完成后才能查询，否则查询会报错。在准备完成后

，实时进行insert/delete的数据变更一般延迟一分钟内可查。

需要注意的是，分析型数据库进行实时插入和删除时，不支持事务，并且仅遵循最终一致性的设计，所以分析型数据库并不能作为OLTP系统使用。

若表的数据是离线批量导入，但是数据源是RDS等其它的云上系统，那么我们可以通过阿里云的CDP产品进行数据同步，在 http://www.aliyun.com/product/cdp/ 上开通CDP（可能需要申请公测）后，按照如下示例配置同步Job：

```
{
"type": "job",
"traceId": "rds to ads job test",
"version": "1.0",
"configuration": {
"setting": {
},
"reader": {
"plugin": "mysql",
"parameter": {
"instanceName": "你的RDS的实例名",
"database": "RDS数据库名",
"table": "RDS表名",
"splitPk": "任意一个列的名字",
"username": "RDS用户名",
"password": "RDS密码",
"column": ["*"],
}
},
"writer": {
"plugin": "ads",
"parameter": {
"url": "在分析型数据库的控制台中选择数据库时提供的连接信息",
"schema": "分析型数据库数据库名",
"table": "分析型数据库表名",
"username": "你的access key id",
"password": "你的access key secret",
"partition": "",
"lifeCycle": 2,
"overWrite": true
}
}
}
}
```

需要注意的是，在运行这个任务之前，需要在分析型数据库中对cloud-data-pipeline@aliyun-inner.com 至少授予表的Load Data权限。

若表是实时写入的，但是仍需要通过数据集成CDP从其他数据源导入到分析型数据库中，则>上述reader配置不变，但是writer配置需要变为（以原表列和目标表列相同为例，若不同需要在column选项中配置Mapping）：

```
"writer": {
"name": "adswriter",
"parameter": {
"writeMode": "insert",
"url": "在分析型数据库的控制台中选择数据库时提供的连接信息",
"schema": "分析型数据库数据库名",
"table": "分析型数据库表名",
"column": ["*"],
"username": "你的access key id",
"password": "你的access key secret",
"partition": "id,ds=2015"
}
```

首次成功导入数据到分析型数据库后，我们便希望我们的应用系统能够连接到分析型数据库来进行数据查询。分析型数据库可以通过任何支持 5.1.x 5.4.x 5.6.x协议的客户端进行连接。连接所使用的域名和端口号可以在iDB Cloud的右上角进行查看。连接使用的用户名和密码为用户在阿里云的Access Key，可以在https://i.aliyun.com/access_key/ 查看和管理。其中Access Key ID为用户名，Access Key Secret为密码（分析型数据库承诺不会保存用户的Access Key信息）。

若需要使用阿里云访问控制(RAM)子账号连接分析型数据库，请参阅使用手册的8.6节。

在PHP中连接分析型数据库

在PHP环境下，假设我们已经安装好了php-mysql 5.1.x模块（Windows下为php_MySQL.dll），那么我们新建一个ads_conn.php，内容如下：

```
$ads_server_name="mydbname-xxxx.ads-cn-hangzhou-1.aliyuncs.com "; //数据库的连接url，请在控制台中的连接信息中获取
$ads_username="my_access_key_id"; // 连接数据库用户名
$ads_password="my_access_key_secret"; // 连接数据库密码
$ads_database="my_ads_db"; // 数据库的名字
$ads_port=3003; //数据库的端口号，请在控制台中的连接信息中获取
// 连接到数据库
$ads_conn=mysqli_connect($ads_server_name, $ads_username, $ads_password, $ads_database, $ads_port);
```

执行查询时，可以使用：

```
$strsql="SELECT user_id FROM my_ads_db.my_first_table limit 20;"; $result=mysqli_query($ads_conn, $strsql);

while($row = mysqli_fetch_array($result)) {
echo $row["user_id"] ; //user_id为列名
}
```

上述代码即可取出任意十条记录的user_id并打印出。注意分析型数据库在数据查询中是不支持SELECT *方式查询所有列的。

在JAVA中连接分析型数据库

通常，在JAVA中，我们通过连接池来使用分析型数据库。在这里我们以国产的高性能连接池Druid为例来演示连接分析型数据库的方式。

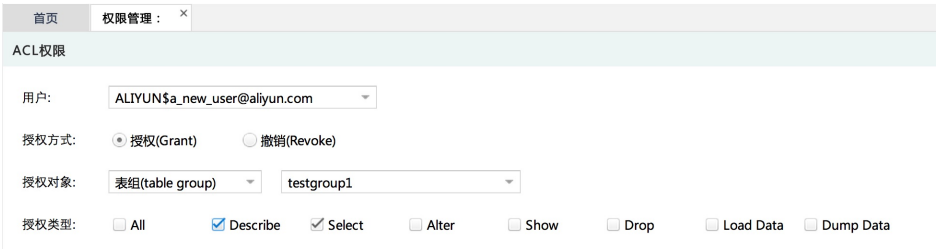
```
import com.alibaba.druid.pool.*;
DruidDataSource dataSource = new DruidDataSource();
dataSource.setDriverClassName("com.mysql.jdbc.Driver");
dataSource.setUsername("my_access_key_id");
dataSource.setPassword("my_access_key_secret");
dataSource.setUrl("jdbc:mysql://mydbname-xxxx.ads-hz.aliyuncs.com:5544/my_ads_db");
// 连接数配置
dataSource.setInitialSize(5);
dataSource.setMinIdle(1);
dataSource.setMaxActive(10);
// 启用监控统计功能
dataSource.setFilters("stat");
// for mysql
dataSource.setPoolPreparedStatements(false);
// 使用心跳语句检测空闲连接
dataSource.setValidationQuery("show status like '%Service_Status%';");
dataSource.setTestWhileIdle(true);
```

如上，需要注意的是，若是在任何语言中需要使用心跳SQL来进行分析型数据库服务状态检测，请使用 show status like ‘%Service_Status%’ 语句，若返回一行两列且第二列为1，则分析型数据库服务正常。

在分析型数据库中，创建一个数据库的账号为这个数据库的owner账号，拥有最大化的权限。在很多情况下，我们不希望这个账号被更多的人掌握，所以需要授权其他人仅使用较小的权限连接分析型数据库。这时我们便需要进行权限管理，完整的权限体系我们将会在第六章详细描述，这里先给出一个简单的例子。

在DMS中，左侧的对象列表中，点击用户可以看到可以访问该数据库的全部用户列表。若要新增一个授权，则我们点击菜单栏上的“权限”按钮。

在打开的窗口中，我们输入待授权的用户账号，注意由于分析型数据库将会支持多种账号来源，所以需要在输入账号时声明账号来源，例如 ALIYUN$ 前缀代表该账号是阿里云账号。我们输入一个待授权的云账号例如 ALIYUN$a_new_user@aliyun.com 。然后假设我们希望这个账号对testgroup1表组下的所有表有查看元信息和只读权限，那么我们进行如下选项操作：



然后点击保存，完成授权操作。随后 a_new_user@aliyun.com 该用户便可使用自己的Access Key登录分析型数据库执行有权限的操作了。

第二章 基本概念

在传统的关系数据库系统中，表隶属于某个数据库。而分析型数据库为了方便的进行数据的关联的管理，以及进行资源分配，引入了表组的概念。

数据库

在分析型数据库中，数据库是用户和系统管理员的管理职权的分界点。

系统管理员：可管理的最小范围即数据库粒度的参数，但未经授权无法查看和管理数据库内部的结构和信息。

用户：对于数据库级别的参数，默认只能查看而不能修改。

在分析型数据库中，一个数据库对应一个用于访问的域名和端口号，同时有且只有一个onwer即数据库的创建者。

- 分析型数据库是以数据库为粒度对用户的宏观资源进行配置，因此创建数据库时用户需要输入业务预估的QPS、数据量、Query类型等信息 用于智能的判断初始的资源分配。
- 分析型数据库的用户不能直接通过CREATE DATABASE的DDL语句创建数据库，只能通过DMS控制台界面来创建需要的业务数据库。

表组

表组是一系列可发生关联的数据表的集合。分析型数据库采用关系模型存储数据，也就是使用二维表来进行数据的组织和存储。分析型数据库为了管理相关联的数据表，引入了表组的概念。

- 表组是数据库的下一级实体，也是表的上一级。
- 在分析型数据库中一个表必须从属于一个表组。

分析型数据库中表组分为两类：维度表组和普通表组。

- 维度表组，用于存放维度表，目前有且仅有一个，在数据库建立时会自动创建（表组名称：”数据库名_dimension_group”），用户不可修改和删除。（维度表特征上是一种数据量较小但是需要和任何表进行关联的表。）
- 普通表组，有如下特征：
    - 表组是数据物理分配的最小单元，数据的物理分布情况通常无需用户关心，但是数据的副本数必须在表组上进行设定，一个表组所有表副本数一致。注：副本数指数据在分析型数据库中同时存在的份数。
    - 同一个表组内的普通表才能进行快速的hash join。（0.9版本之前分析型数据库对普通表仅支持同表组内join。0.9.5版本后支持不同普通表组内的普通表进行join）

- 一个表组的表可以共享一些配置，例如查询超时时间，如果表组中的单表也进行了这些配置的个性化，那么进行表关联时会通表组级别的配置进行覆盖。
- 一个表组中的所有表的一级Hash分区的分区数建议一致，但非强制性要求。

表

分析型数据库中表分为维度表和普通表，普通表也称实时表或分区表。

- 维度表可以和任意表组的任意表进行关联，并且创建时不需要配置分区信息，但是对单表数据量大小有所限制，并且需要消耗更多的存储资源。（一般要求维度表单表不超过1000万）
- 普通表创建时至少要指定一级分区列和相关分区信息，并且指定存放在一个表组中。

分区

在分析型数据库中，维度表无分区，普通表的分区目前最多为两级。

分区种类有Hash分区和List分区两种。目前分析型数据库支持将普通表的一级分区设置为Hash分区，二级分区设置为List分区，即可支持 Hash join又可支持增量数据导入。

Hash分区是根据导入数据时已有的一列的内容进行散列后进行分区的，目前多张事实表进行快速的Hash Join时JoinKey必须有分区列参与，同时这些表的Hash分区数必须一致。仅采用Hash分区的数据表，在数据装载时，将进行全量覆盖历史数据。

注：0.9版本之前，不支持事实表在没有Hash分区键参与关联条件的情况下进行关联，0.9.5版本后，通 过Full MPP Mode或小表广播进行计算时无此限制，关于 join的详细内容请看《多计算引擎和Hint》章 节。

List分区是根据导入操作时所填写的分区列值来进行分区的，即一次导入的数据会进入同一个List分区中，因此List分区是支持增量的数据导入的。

注：无论采用何种分区形态，分析型数据库均不需要在用户查询时指定分区列，但是指定分区列或分区列的范围进行查询可能会提高查询性能。

根据表的更新方式不同，分析型数据库的表分为离线批量更新的表和实时更新的表：

- 离线批量更新的表，适合从离线系统如MaxCompute（原ODPS）产出的数据结果导入到分析型数据库供在线系统使用。
- 实时更新的表，可以直接insert/delete单条数据，适合业务系统直接写入数据，不支持odps批量load。

注意，分析型数据库不支持读写事务，并且数据实时更新时一分钟左右才可查询，另外在一致性方面分 析型数据库遵循最终一致性。

支持的数据类型

boolean布尔类型，值只能是0或1；取值0的逻辑意义为假，取值1的逻辑意义为真；存储字节数1比特位。

tinyint微整数类型，取值范围-128到127； 存储字节数1字节。

smallint整数类型，取值范围-32768到32767；存储字节数2字节。

int整数类型，取值范围-2147483648到2147483647； 存储字节数4字节。

bigint大整数类型，取值范围-9223372036854775808到9223372036854775807; 存储字节数8字节。

float单精度浮点数，取值范围-3.402823466E+38到-1.175494351E-38, 0, 1.175494351E-38到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到-2.2250738585072014E-308, 0, 2.2250738585072014E-308 到 1.7976931348623157E+308. IEEE标准; 存储字节数4字节。

varchar变长字符串类型。

date日期类型，取值范围： '1000-01-01' 到 '9999-12-31', 支持的数据格式为'YYYY-MM-DD'存储字节数为4字节。

timestamp时间戳类型， 取值范围： 1970-01-01 00:00:01.000' UTC 到 '2038-01-19 03:14:07.999' UTC., 支持的的数据格式为：'YYYY-MM-DD HH:MM:SS'存储字节数为4字节。

multivalue多值列类型，即一个cell中包含多个值，多个值直接的分隔符默认是英文半角逗号，当然也可以通过delimeter关键字指定。有关多值列说明请看后面《分析型数据库特色功能》介绍。

与mysql数据类型对比

注： *分析型数据库所有的数据类型都不支持unsigned， 下表不包含该差异*

分析型数据库数据类型	MySQL数据类型	差异
boolean	bool,boolean	一致
tinyint	tinyint	一致
smallint	smallint	一致
int	int,integer	一致

bigint	bigint	一致
float	float[(m,d)]	分析型数据库不支持自定义m和d，而mysql可以
double	double[(m,d)]	分析型数据库不支持自定义m和d，而mysql可以
varchar	varchar	一致
date	date	一致
timestamp	timestamp	分析型数据库只支持到精确到毫秒，而mysql是可以定经度的
multivalue	-	分析型数据库特有的，MYSQL无此类型

作为创新型的实时OLAP服务，分析型数据库提供很多独有的特色功能，这里便就其中常用的部分进行简要的介绍。

聚集列

在创建表时，用户可以指定一列或者若干列作为聚集列。

- 在物理上，一个分区内聚集列内容相同的数据会尽可能的分布在同样的区块内存放。因此当用户的查询Query的条件指定了聚集列的内容或范围时，查询性能会有较大的提升。
- 需要注意的是如果用户指定多列为聚集列，那么指定的聚集列的顺序就是比较数据是否相同的顺序。
- 聚集列可以在建表后进行修改，但是目前修改后需重新装载数据完毕后才生效。

多值列

分析型数据库有一个特殊的数据类型：多值列。

- 多值列可以存入String类型的多个值。在原始数据中多值列为一列用分隔符（默认为半角逗号，也可以建表时进行配置）分隔的String类型。
- 多值列数据存入分析型数据库后，可使用in, contains条件对该列的单个值进行查询，枚举查询后该列的每个值可像一个普通列一样进行各类操作。但是不允许在没有进行枚举查询时对该列直接select或在group by中使用该列。

多值列的通常使用场景：

已有一个实体属性表均为普通列并以实体编号为主键的情况下，需要新增一个用于进行实体筛选的属性，而这个属性和实体编号为多对多的对应关系，按照通常的做法，则新建一张该实体编号和属性两列的数据表，和原有的实体属性表进行Join操作查询。而使用多值列后，**性能会被进行Join操作计算高数倍。**

一个具体的例子如下：

一张用户表中已有用户id和性别、年龄、职业、籍贯的数据。

用户ID	性别	年龄	职业	籍贯

bigint	varchar	int	varchar	varchar

而现在需要增加用户购买的商品品类的数据用于筛选，于是可以将这个商品的品类id存放在用户表的一个新增的多值列中使用。

用户ID	性别	年龄	职业	籍贯	购买的商品品类ID
bigint	varchar	int	varchar	varchar	multivalue

智能自动索引

分析型数据库拥有智能的自动索引创建机制。

通常情况下，分析型数据库会根据导入的数据的每一列分布情况自动为每一列创建符合该列情况的索引类型，无需用户指定创建索引或索引类型。

> 注：导入数据每一列分布情况如该列的枚举值数量的多少，该列的数据是连续或离散。

但是，如果用户认为某一列不需要进行筛选查询，可以指定该列不需要创建索引来节省数据存储空间。

> 注：0.9版本之前如果一个列需要作为表关联的hash 分区列，那么用户需要人工指定该列创建Hash索引。0.9版本已废弃Hash索引，可动态根据Join SQL进行优化，完全无需用户干预。

查询CBO优化

分析型数据库拥有高度智能的CBO(Cost-Based Optimization)优化策略。

在用户发起的一个Query达到分析型数据库后，分析型数据库会智能的判断该Query涉及的数据的分布情况、索引使用情况、缓存使用情况、Query条件分布等，进行逻辑和物理执行计划优化和重组，尽可能的以最优的路径执行Query查询。

因此大部分情况下用户无需关心Query的具体写法，只要保证语义正确即可。另外分析型数据库提供一些Hint参数，可以在Query时对执行计划进行部分调整，详见《用户指南》的《多计算引擎和Hint》章节。

ECU（弹性计算单元），是分析型数据库中存储和计算资源的分配单位。

分析型数据库对每个用户的每一个DB会分配若干个计算节点（COMPUTENODE），以及若干个接入节点（FRONTNODE），还有若干个用于放置实时化数据写入缓冲的缓冲节点（BUFFERNODE）。

- FRONTNODE节点用于接收用户的应用前端连接等工作。
- COMPUTENODE节点用于存储用户的数据和进行计算。
- BUFFERNODE用于实时数据写入缓冲。

目前分析型数据库仅计算节点是用户可按ECU模式配置，分析型数据库会自动根据用户的计算节点的量来配置接入节点等其它角色的数量。

计算节点的ECU具有如下属性：

- 内存容量：该ECU的内存大小。
- 磁盘容量：该ECU的磁盘容量，用户在一个DB中存储的物理数据总量不能超过该DB的全部的磁盘容量，并且由于分析型数据库将用户的数据分布到每一个ECU中，若用户的数据倾斜导致单个ECU的磁盘空间占满，也会导致数据无法再进入分析型数据库。各个ECU的磁盘使用情况可以在云监控（公共云地址:http://cms.console.aliyun.com/）中查看。

目前分析型数据库公共云提供的ECU规格为（专有云参照此标准灵活执行）：

型号	内存	磁盘类型	磁盘容量(SSD)	磁盘容量(SATA)
c1	7.5GB	SSD	60GB	
c8	45GB	SSD	480GB	
s1n	25GB	SSD+SATA	250GB	1.5TB
s2n	45GB	SSD+SATA	480GB	4TB

0.9版本的分析型数据库，提供基于SATA存储的大容量实例（目前为邀请测试功能，后续开放购买），采用SATA和SSD混合存储，能够大幅度降低存储成本，但是同时查询性能也以数量级而下降。

大容量实例的ECU型号通常以字母s开头。专有云中原则上仅万兆网物理机能够运行大容量实例。

ECU数量，可以通过DMS for AnalyticDB界面的扩容/缩容功能，或相应DDL动态修改。

在分析型数据库中拥有两套计算引擎：

- COMPUTENODE Local/Merge(简称LM)：先前版本的旧计算引擎，优点是计算性能很好，并发能力强，缺点是对部分跨一级分区列的计算支持较差。
- Full MPP Mode（简称MPP）：0.9.5版本新增的下一代计算引擎，优点是计算功能全面，支持SQL:2003标准，支持任意列join、window function等，对跨一级分区列的计算有良好的支持，可以通过全部TPC-H查询测试用例（22个）和全部的TPC-DS查询测试用例。

Full MPP Mode计算引擎目前(2016年12月起)已向全体用户公测，除非已申请关闭Full MPP Mode，否则用户可以自行判断自己的Query合适哪一种计算引擎，使用Hint强制将查询路由到某一个计算引擎中：

```
--强制使用COMPUTENODE LM计算引擎（默认）
/*+engine=COMPUTENODE*/
select * from ....

--强制使用Full MPP Mode计算引擎
/*+engine=MPP*/
select * from ....
```

目前使用Full MPP Mode时请注意以下问题：

Full MPP Mode查询响应时间和并发能力较LM模式稍差，建议用户仅在较低频调用的、对性能不太

敏感的、必须使用Full MPP Mode计算引擎的情况下使用，通常适用于交互式BI场景，实时ETL场景；

Full MPP Mode计算引擎拥有较丰富的数学函数、字符串处理函数、窗口函数等支持。

分析型数据库亦支持自动对查询Query进行路由功能，可以将LM引擎不支持的查询路由给MPP引擎，尽可能兼顾性能和通用性。

开启自动路由功能后（默认不开启），目前以下几种情况会自动路由到Full MPP Mode，这些场景同时也是必须使用Full MPP Mode的场景：

特定函数，如row_number over等，LM模式不识别，捕获异常；

Join类：

- 事实表 join 事实表，join key全部在非分区列上；
- 不同表组的事实表 join 事实表；
- 维度表在前，left join 事实表；
- 事实表 right join 维度表（同上）；
- 事实表 join 事实表，一级分区数不同。

Group By、Order By、Having类：

- Group By仅含非分区列，外层套子查询；
- Group By仅含非分区列，带Order By；
- Group By仅含非分区列，带Having。

UNION/INTERSECT/MINUS不含分区列；

SELECT 复杂表达式，如SUM/SUM, 任何带聚合函数的计算表达式等；

COUNT DISTINCT或DISTINCT非分区列。

管理MPP查询任务：

通过如下命令可以查询目前系统正在运行的MPP查询任务，由于有多个FRONTNODE节点实例，如下命令查询是目前连接的FRONTNODE实例上运行MPP查询任务:

```
SHOW PROCESSLIST MPP
```

如果想查询跨FRONTNODE所有实例上正在运行的MPP查询任务，可运行如下命令:

```
/*+cross-frontnode=true*/SHOW PROCESSLIST MPP
```

每个MPP查询会有一个Query ID, 通过SHOW PROCESSLIST MPP 可以看到MPP查询任务的Query ID, 如果想杀掉该MPP查询任务，可运行如下命令:

```
/*+cross-frontnode=true*/KILL MPP '$query_id'
```

如果想杀掉当前连接的FRONTNODE实例上所有正在运行的MPP查询任务，可运行如下命令:

```
KILL MPP ALL
```

如果想跨FRONTNODE所有实例杀掉所有正在运行的MPP查询任务，可运行如下命令:

```
/*+cross-frontnode=true*/KILL MPP ALL
```

第三章 DDL

目前，公共云计算上分析型数据库的用户不能直接通过CREATE DATABASE的DDL语句创建数据库，只能通过DMS控制台界面来创建需要的业务数据库。

用户需要根据自己的业务需求来估算使用分析型数据库时所需的查询类型等配置参数，并在DMS的界面上进行填写。由于已在1.2节中叙述过相关流程，所以再次不再叙述。

专有云中若需要创建数据库，请联系您的运维管理员或DBA。

在第一章中，我们已经介绍了如何通过DMS界面创建和修改表组，本节中，我们详细描述如何通过DDL来创建和修改表组以及表组各属性的含义。

创建表组时，我们提交如下SQL：

```
create tablegroup db_name.tablegroup_name options(minRedundancy=2 executeTimeout=30000);
```

其中db_name为数据库名称，tablegroup_name为表组名称（不能和现有表组重叠）。

options部分为可选项：minRedundancy表示该表组的副本数，默认为2，可配置为1、2、4、8。需要注意的是，如果将一个表组配置为1副本，那么这个表组中的表在数据导入时会有不可用的时间。而将表组副本数配置为4或更高，可以一定程度的增加分析型数据库的最大承受的QPS，但是数据存储费用也会相应增加。在绝大部分情况下，不推荐修改任何表组默认配置。

executeTimeout表示该表组的全局Query超时时间，默认为30000，单位毫秒。

当我们需要删除一个表组时，我们可以提交如下SQL：

```
drop tablegroup db_name.tablegroup_name;
```

注意，仅允许删除没有任何表的空表组，维度表组不允许删除。

当我们需要修改表组的两个属性中的任何一个时，我们可以提交如下SQL：

```
alter tablegroup db_name.tablegroup_name key=value;
```

其中key为属性名，value为新的属性值。需要注意的是，minRedundancy修改后需要下次装载数据时才会生效。

在快速开始中，我们已经大概了解了在如何通过iDB Cloud创建和修改表。在这一节中，我们将详细描述如何通过DDL创建和修改表。

由于分析型数据库有一些传统关系型数据库没有的特性，所以分析型数据库的DDL在遵循类MySQL规范的基础上，有不少的独有的属性和语法。分析型数据库中创建事实表DDL语法结构如下：

```
CREATE TABLE db_name.table_name (
col1 bigint COMMENT 'col1',
col2 varchar COMMENT 'col2',
col3 int COMMENT 'col3',
col4 bigint COMMENT 'col4',
col5 multivalue COMMENT 'col5 多值列',
[primary key (col1, col3)]
)
PARTITION BY HASH KEY(col1)
PARTITION NUM 50
[SUBPARTITION BY LIST (part_col2 long)]
[SUBPARTITION OPTIONS (available_partition_num = 30)]
[CLUSTERED BY (col3,col4)]
TABLEGROUP ads_test ;
[options (updateType='{realtime | batch}')]
```

其中’ []’ 中的内容为视情况填写的可选项。创建事实表（也就是非维度表）时，必须指定的是一个数据库名（db_name）和表名（table_name），以及至少一列的列信息，至少一个分区的信息以及一个表组。

首先，根据表的数据更新方式不同，分析型数据库的表根据updateType分为批量更新表（仅能够离线批量更新数据）和实时更新表（能够通过insert/delete实时更新数据），用updateType以区分，如果updateType选项不填则默认为批量更新表。需要注意的是，updateType=realtime暨为实时更新表时，必须指定合法的主键并且不能有二级分区。

列信息的基本格式为”列名 类型 COMMENT ‘注释’ ”，关于列的更多属性，请参照后文3.4节。”[primary key (col1, col3)]”一段指定了表的主键，如果表为批量更新的，则主键没有意义，而表为实时更新的，则必须指定主键并且主键中必须含有一级Hash分区列（可以是联合主键）。

关于分区方面，目前分析型数据库支持最多两级分区，并且一级分区仅支持HASH分区，二级分区仅支持

LIST分区。HASH分区是一种动态分区值类型，暨根据实际数据中的某一列的内容进行分区。所以在语法上，一级HASH分区的用法是：

```
PARTITION BY HASH KEY(col1)
PARTITION NUM 50
```

其中col1为需要进行分区的列名，必须是表中实际存在的列。’ PARTITION NUM’为分区数量，一般根据该表的数据量确定，每个分区一般不超过1500万条记录为宜（亦可通过划分二级分区实现无限扩展）。另外HASH分区列的数据分布要尽可能均匀，不能有非常明显的倾斜（暨分区列值），否则会较严重的影响查询性能。

一定需要注意的一点：目前分析型数据库要求一个表组下所有表的一级分区数目一致，所以建立第一张表时指定一级分区数量时请谨慎。一级分区数量默认不能超过256个。

另外需要注意的是，若一张表仅有一级HASH分区并且是批量导入的表，则每次导入数据时会对已有进行全量覆盖。若需要每次导入数据时增量导入，则需要再指定二级List分区信息：

```
SUBPARTITION BY LIST (part_col2 bigint)
SUBPARTITION OPTIONS (available_partition_num = 30)
```

二级List分区为非动态分区，暨分区值不是由数据本身决定的，而是由每次导入/写入数据时用户指定的。所以在进行分区信息定义时需要指定一个和现有数据中的列不同的新列名，以及这个列的类型（目前仅支持long）。二级分区有一个可选属性，available_partition_num，即为最大保留的二级分区数，当新的数据装载进来后，若线上存在的二级分区数大于这个值，那么会根据二级分区的值进行排序，下线最小的若干分区的数据。

0.8版本的实时更新表暂不支持二级分区，0.9版本支持。但是实时更新表的二级分区仅用于极大的扩展单表容量，以及进行生命周期管理，实时更新表的增量更新不依赖于二级分区。

惯常的用法是，将经常需要进行Join的列（例如买家ID）作为一级Hash分区列，而将日期列作为二级分区列。这样的表既可以进行大表Join的加速，又可以每天进行增量数据导入，并且指定保留若干天的数据在线上来进行生命周期管理。

但是，一级分区数量和二级分区数量的设置，并不是越多越好的。而是要看表的数据量，以及数据库拥有的资源数。若一级二级分区过多而数据库的资源数过少，则很容易让分区的数据Meta将内存耗尽。

CLUSTERED BY子句用于指定聚集列，用户可以把一列或者多列指定为聚集列，注意如果指定多列，那么该表的数据聚集顺序按照DDL中这个子句中指定的列组合顺序进行排序。通常，我们将一个表的查询中肯定会涉及到的并且数据区分度很大的列设置为聚集列，有时候能较显著的提升查询性能。

事实表的创建上，默认有如下限制：（1）一张事实表至少有一级Hash分区并且分区数不能小于8个；（2）一个事实表组最多可以创建256个事实表；（3）一个事实表最多不能超过1024个列。

与创建事实表相比，创建维度表要简单的多：

```
CREATE DIMENSION TABLE db_name.dim_table_name (
col1 int comment 'col1',
col2 varchar comment 'col2'
```

```
[primary key (col1)]
)
[options (updateType='{realtime | batch}')]
;
```

创建维度表时只需要指定数据库名和表名即可，维度表会创建到统一的维度表组，并且无需指定分区信息。

在表已经创建好之后，目前支持有限的修改：主要是支持增加列。

```
ALTER TABLE db_name.table_name ADD column col_new varchar;
```

注意，批量导入表新增加的列在数据重新装载后才会生效，实时写入表新增列后，一般需要几分钟后才可以读写。

最大二级分区数目前可以在建表后进行修改，但是修改后的下一次数据导入发起后才会生效：

```
ALTER TABLE db_name.table_name subpartition_available_partition_num = N
```

N为新的二级分区数。

在DDL中，建表时一列的定义是：

```
col_name type [NOT NULL | NULL] [DEFAULT default_value] [PRIMARY KEY] [COMMENT 'string']
[column_options(precision, scale, disableIndex...)]
```

其中col_name为列名，type为列的数据类型，详见2.3节。[NOT NULL | NULL]是否可为空，以及[DEFAULT default_value]定义的列的默认值和标准的MySQL DDL中无甚不同。

关于[PRIMARY KEY] 主键部分，对于批量更新表，分析型数据库中主键的概念是弱化的，分析型数据库不要求一个表有主键，有主键的表的性能和用法上和没有主键的表之间没有任何区别。若一个表进行数据导入时该次导入的数据中存在主键冲突，则该次导入会失败并且报错。对于实时更新表，请使用在所有列尾部的"primary key (col1, col3)"语法指定主键。

列属性上，一个列可以设置列属性disableIndex = true，用于屏蔽分析型数据库的默认列索引，不过需要注意的是，要如此做，则这个列应该不在实际查询中所筛选和计算的。precision和scale属性则是针对decimal数据类型（目前暂未上线，未来会上线该功能）特有的属性，precision为数字整体有效数字个数，scale为小数点后的数字个数。

同主键一样，分析型数据库中索引的概念也是弱化的。在前文中介绍分析型数据库拥有高度智能的自动化索引机制，所以通常用户无需亲自为自己的数据表配置索引。但是有一种情况例外：暨0.8版本下用户需要对某列进行Hash Join时，无论是事实表之间的Join还是事实表和维度表的Join，都需要为事实表的该列建立索引（公共云当前版本无需）。

建立索引的语句如下：

```
ALTER TABLE tbl_name ADD INDEX [index_name] [index_type] (index_col_name) [comment=’’];
```

其中，[index_type] 为索引类型，需要指定为HashMap，index_col_name 为被索引列的列名。例如：

```
ALTER TABLE db_name.table_name
ADD INDEX user_id_index HashMap (user_id)
```

批量导入表索引修改后，需要重新导入数据后生效。实时写入表一般在24小时内自动生效，或如果要加速这个过程，可以执行optimize table 命令：

```
optimize table <tablename>;
```

执行成功一段时间后，新的索引会生效。

0.9及以后版本中分析型数据库会自动处理Hash Join时的索引构建，故无需用户自行创建HashMap索引也可以进行Hash Join，因此HashMap索引被废弃。

在DMS for 分析型数据库中，用户可以在图形界面上进行扩容/缩容，以及查看扩容/缩容执行状态等操作(实例管理->DB容量管理->容量变更 按钮)。同时，用户可以通过DDL来进行ECU变更，也就是扩容/缩容（需要Alter Database权限或DB owner角色）。

```
ALTER DATABASE SET ecu_count = N;
```

N为要设置的目标ECU数量，若目标ECU数量大于目前当前的ECU数量，则为扩容行为，若目标ECU数量小于当前的ECU数量，则为缩容行为。若用户的数据量大于目标ECU的总存储，则缩容会失败。

缩容和扩容都不是瞬时的同步操作，可以使用元数据查询状态：

```
select * from information_schema.resource_request;
```

查看目前已有的ECU状态：

```
select * from information_schema.current_instances;
```

SHOW DATABASES

查询用户的database列表。指定 EXTRA 参数，输出关于database的更多信息（创建者ID，数据库连接串IP:PORT等）。

```
SHOW DATABASES [LIKE 'name_pattern'] [EXTRA]
```

SHOW TABLEGROUPS

查询用户当前database下的表组列表。

```
SHOW TABLEGROUPS [IN db_name]
```

SHOW TABLES

查询用户当前database（或者表组）下的表的列表。

```
SHOW TABLES [IN db_name[.tablegroup_name]]
```

SHOW COLUMNS

查询表的列信息。

```
SHOW COLUMNS IN table_name
```

SHOW CREATE TABLE

查询指定表的DDL。

```
SHOW CREATE TABLE [db_name.]table_name
```

SHOW ALL CREATE TABLE

查询当前数据库里所有表的DDL。

```
SHOW ALL CREATE TABLE db_name
```

SHOW PROCESSLIST MPP

查询当前正在运行的MPP任务。指定 */+cross-frontnode=true/* 时，查询当前数据库实例所有正在运行的MPP任务，不指定时，仅仅查询当前连接的FRONTNODE节点实例运行的MPP任务。

```
[/*+cross-frontnode=true*/] SHOW PROCESSLIST MPP
```

第四章 DML

在分析型数据库中拥有两套计算引擎：

- COMPUTENODE Local/Merge(简称LM)：先前版本的旧计算引擎，优点是计算性能很好，并发能力强，缺点是对部分跨一级分区列的计算支持较差。
- Full MPP Mode（简称MPP）：0.9.5版本新增的下一代计算引擎，优点是计算功能全面，支持SQL:2003标准，支持任意列join、window function等，对跨一级分区列的计算有良好的支持，可以通过全部TPC-H查询测试用例（22个）和全部的TPC-DS查询测试用例。

Full MPP Mode计算引擎目前(2016年12月起)已向全体用户公测，除非已申请关闭Full MPP Mode，否则用户可以自行判断自己的Query合适哪一种计算引擎，使用Hint强制将查询路由到某一个计算引擎中：

```
--强制使用COMPUTENODE LM计算引擎（默认）
/*+engine=COMPUTENODE*/
select * from ....

--强制使用Full MPP Mode计算引擎
/*+engine=MPP*/
select * from ....
```

目前使用Full MPP Mode时请注意以下问题：

Full MPP Mode查询响应时间和并发能力较LM模式稍差，建议用户仅在较低频调用的、对性能不太敏感的、必须使用Full MPP Mode计算引擎的情况下使用，通常适用于交互式BI场景，实时ETL场景；

Full MPP Mode计算引擎拥有较丰富的数学函数、字符串处理函数、窗口函数等支持。

分析型数据库亦支持自动对查询Query进行路由功能，可以将LM引擎不支持的查询路由给MPP引擎，尽可能兼顾性能和通用性。

开启自动路由功能后（默认不开启），目前以下几种情况会自动路由到Full MPP Mode，这些场景同时也是必须使用Full MPP Mode的场景：

特定函数，如row_number over等，LM模式不识别，捕获异常；

Join类：

- 事实表 join 事实表，join key全部在非分区列上；
- 不同表组的事实表 join 事实表；
- 维度表在前，left join 事实表；
- 事实表 right join 维度表（同上）；
- 事实表 join 事实表，一级分区数不同。

Group By、Order By、Having类：

- Group By仅含非分区列，外层套子查询；
- Group By仅含非分区列，带Order By；
- Group By仅含非分区列，带Having。

UNION/INTERSECT/MINUS不含分区列；

SELECT 复杂表达式，如SUM/SUM, 任何带聚合函数的计算表达式等；

COUNT DISTINCT或DISTINCT非分区列。

管理MPP查询任务：

通过如下命令可以查询目前系统正在运行的MPP查询任务，由于有多个FRONTNODE节点实例，如下命令查询是目前连接的FRONTNODE实例上运行MPP查询任务:

```
SHOW PROCESSLIST MPP
```

如果想查询跨FRONTNODE所有实例上正在运行的MPP查询任务，可运行如下命令:

```
/*+cross-frontnode=true*/SHOW PROCESSLIST MPP
```

每个MPP查询会有一个Query ID, 通过SHOW PROCESSLIST MPP 可以看到MPP查询任务的Query ID, 如果想杀掉该MPP查询任务，可运行如下命令:

```
/*+cross-frontnode=true*/KILL MPP '$query_id'
```

如果想杀掉当前连接的FRONTNODE实例上所有正在运行的MPP查询任务，可运行如下命令:

```
KILL MPP ALL
```

如果想跨FRONTNODE所有实例杀掉所有正在运行的MPP查询任务，可运行如下命令:

```
/*+cross-frontnode=true*/KILL MPP ALL
```

4.2 LM计算引擎

当表定义为实时表时（指定 options(updateType='realtime')），只能采用实时写入的方式录入数据。 目前ADS支持的实时写入语句包括：INSERT和DELETE两种类型，不支持UPDATE。 另外，实时表必须定义主键

primary key，主键可包含一列或多列，主键保证唯一性。

实时数据可见性

实时数据通常在INSERT语句执行成功后，一分钟内可查。目前还在持续优化实时数据的可见性时延。针对大批量写入和大批量delete的场景，时效性在数分钟级别，若对时效性有要求，可执行命令 optimize table xxx (参见本节“OPTIMIZE TABLE操作”内容)来加速数据的生效时间。

INSERT IGNORE vs INSERT

ADS实时数据INSERT的默认行为为主键覆盖，即后INSERT的记录将覆盖系统中已有的相同主键的记录。 如果INSERT指定 IGNORE 关键字，若在系统中已经有相同主键的记录，则当前INSERT IGNORE语句执行能成功，但是新记录将会被丢弃掉。 用户可以根据业务应用的实际需求，选择INSERT IGNORE或者INSERT。

INSERT语句

INSERT语句符合标准SQL INSERT语法，支持单条插入:

```
INSERT INTO db_name.table_name (col1, col2, col3) VALUES ('xxx', 123, 'xxx');
```

多条批量插入:

```
INSERT INTO db_name.table_name (col1, col2, col3) VALUES ('xxx', 111, 'xxx'), ('xxx', 222, 'xxx'), ('xxx', 333, 'xxx');
```

如果插入数据指定了所有列，可以省略INSERT语句中的列名部分:

```
INSERT INTO db_name.table_name VALUES ('xxx', 111, 'xxx'), ('xxx', 222, 'xxx'), ('xxx', 333, 'xxx');
```

> 注意：VALUES中的数据对应的列顺序必须符合表定义时的列顺序，可通过如下语句查询表的列定义的顺序:

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

数据表示

ADS的INSERT语句中的实时数据，采用强类型的策略，如不符合强类型的数据表征，INSERT语句将抛出异常：

VARCHAR, DATE, TIME, TIMESTAMP, MULTIVALUE类型的数据必须加上'', 例如：'abc', '2017-01-01';

数值类型的数据不能加上'', 例如：1, 1.9, 0.009

特殊字符

由于ADS实时数据写入链路设计，如下特殊字符是不能出现在INSERT语句的字段数据中：

#, ', \n, \r, \

应用程序应该在数据字段写入ADS前进行特殊字符清洗或过滤处理，通常包含复杂字符的字段为明细信息字段， 最终用来在业务层做展示使用，不参与关联或聚合计算。有一种解决方案为，在数据入库前对这类字段进行Base64编码， 最终查询这些字段出去后，进行Base64解码获得原始数据。

实时数据写入性能优化

为了提高实时数据写入性能，通常采用如下两种优化策略：

1 批量写入：

默认单条INSERT语句的最大限制为1MB，但基于实践经验，批量写入的一批的记录数通常控制在100 ~ 200比较合适；

若表的列非常多，酌情减少一批的记录数，控制一条INSERT语句的大小在1MB以内。

2 分区聚合：

在批量写入的前提下，尽量保证一批记录的目标hash分区相同，这样可以充分利用批量写入提交的性能优化；

业务开发者可向ADS的技术支持获取ADS的数据分区算法，用来预先进行数据分区聚合。

删除数据

通过DELETE语句执行删除操作，必须指定WHERE条件:

```
DELETE FROM db_name.table_name WHERE col1 = xxx;
```

删除全表数据（不包含二级分区，请谨慎使用）:

```
DELETE FROM db_name.table_name WHERE 1=1;
```

如果目标表包含二级分区，WHERE条件必须包含二级分区列的等值条件:

```
DELETE FROM db_name.table_name WHERE sub_part_col = xxx;
```

通过系统元数据查询实时数据写入量

查询语句:

```
SELECT * FROM information_schema.realtime_table_traffic_cross;
```

```
+--------------+-------------+-----------+----------+------------------+--------------------+-------------+-------------
----+-----------------------------+------------------------+------------------------+------------------------+
| TABLE_SCHEMA | TABLE_NAME | SCHEMA_ID | TABLE_ID | INSERT_STMT_COUNT | INSERT_RECORD_COUNT |
DELETE_COUNT | SUB_PART_COUNT | SUB_PART_RECORD_COUNT_STATS | LAST_RESET_TIME | CREATE_TIME |
UPDATE_TIME |
+--------------+-------------+-----------+----------+------------------+--------------------+-------------+-------------
----+-----------------------------+------------------------+------------------------+------------------------+
| xxxxxxxxxxx | xxxxxxxxxxx | 9096 | 14 | 0 | 0 | 0 | 0 | {} | 2017-07-24 14:15:09.294 | 2017-07-25 09:39:38.848 | 2017-
07-25 09:39:38.848 |
+--------------+-------------+-----------+----------+------------------+--------------------+-------------+-------------
----+-----------------------------+------------------------+------------------------+------------------------+
```

列定义：

- INSERT_STMT_COUNT: 从LAST_RESET_TIME到目前为止，实时INSERT的语句条数；
- INSERT_RECORD_COUNT: 从LAST_RESET_TIME到目前为止，实时INSERT的记录条数，批量插入的情况下，INSERT_RECORD_COUNT大于INSERT_STMT_COUNT；
- DELETE_COUNT: 从LAST_RESET_TIME到目前为止，DELETE的语句条数；
- SUB_PART_COUNT: 从LAST_RESET_TIME到目前为止，实时INSERT涉及到的二级分区的个数；
- SUB_PART_RECORD_COUNT_STATS: 从LAST_RESET_TIME到目前为止，实时INSERT涉及到的各个二级分区的记录数；
- LAST_RESET_TIME: realtime_table_traffic_cross表默认24小时reset一次，该字段记录reset的时间点。

FLUSH操作

由于AnalyticDB的实时数据写入和计算节点实时数据消费是异步流程，所以，FLUSH操作用来检查FLUSH时间点之前的实时数据是否都被正常入库到计算节点上了。

```
FLUSH db=<schema> table=<table> timeout=<timeout_duration_ms>
```

其中 timeout=<timeout_duration_ms> 用来指定FLUSH操作在多长时间内完成（单位：ms），否则FLUSH超时失败，未能保证FLUSH时间点之前的实时数据被正常入库完成。

```
FLUSH db=<schema> table=<table> timeout=<timeout_duration_ms> return_version_onfailure=[true|false]
```

其中 return_version_onfailure=[true|false] 用来指定在FLUSH失败时是否返回当前计算节点的数据版本号，true 表示返回， false 表示不返回。

```
```

```
FLUSH db=<schema> table=<table> timeout=<timeout_duration_ms> expected_version=<partition_version_list>
```

其中 expected_version=<partition_version_list> 用来传入期望的数据分区版本号，仅仅当期望的数据分区版本号满足时，FLUSH才执行成功。

```
<partition_version_list> pattern should be 0:xxxx,1:xxxx,2:xxxx,...
```

OPTIMIZE TABLE操作

OPTIMIZE TABLE是实时表的的索引构建操作，用来对实时数据构建索引并且与基线数据进行合并，以提升实时表的查询性能。

```
OPTIMIZE TABLE [dbname.]table_name1 [, [dbname.]table_name2]
```

通过DML进行计算是用户在分析型数据库中最常用的操作。目前分析型数据库对DML支持使用SELECT进行查询，以及支持INSERT、DELETE等数据修改操作（INSERT/DELETE见4.2.1节）。另外本节的内容需要配合附录二、4.1中的多计算引擎以及4.2.7中Hint的说明来阅读。

分析型数据库中SELECT语句的基本结构如下：

```
SELECT
[DISTINCT]
select_expr [, select_expr ...]
[FROM table_references
[WHERE filter_condition]
[GROUP BY {col_name | expr | position}
[HAVING having_condition]
[ORDER BY {col_name | expr | position}
[ASC | DESC], ...]
[UNION ALL (SELECT select_expr..)]
[{UNION|INTERSECT|MINUS} (SELECT select_expr..)]
[LIMIT {row_count}]
```

下面，我们将一个完整的SELECT分为几个部分加以介绍：列投射部分（列和列表达式）、FROM/JOIN子句（表扫描、表连接与子查询）、WHERE子句（过滤表达式）、GROUP BY/HAVING子句（分组和分组后过滤）、ORDER BY子句（排序）、两个查询间的UNION [ALL]/INTERSECT/MINUS、LIMIT子句（返回行数限制）。

列投射（列和列表达式）

分析型数据库中，SELECT语句中的列投射的基本结构为：

```
SELECT [DISTINCT] select_expr [, select_expr ...]
```

其中 select_expr 可以是基本列（直接从表或子查询中选出的），也可以是列表达式（包括算数计算、聚合函

数、系统UDF等）。

当 select_expr 为基本列时：

基本列是表中的列，这里的表可以是物理表也可以是虚表（物理表的别名引用或子查询产生的表）。

例如：

- SELECT A.a FROM A：指定表名和列名，表来自FROM子句，列属于FROM子句指定的表
- SELECT a FROM A：仅指定列名，列属于FROM子句指定的表
- SELECT A.a AS x FROM A：指定表名和列名（带别名），表来自FROM子句，列属于FROM子句
- SELECT a AS x FROM A：仅指定列名（带别名），表来自FROM子句，列属于FROM子句
- SELECT b, a FROM A JOIN B ...，仅指定列名，列属于FROM子句指定的第一个包含该列的表
- SELECT A.a FROM (SELECT A.a, b FROM A JOIN B ...)，指定表名和列名，列属于子查询返回的列（<table>.<column>完全匹配）
- SELECT a FROM (SELECT a, b FROM A JOIN B ...)，仅指定列名，列属于子查询返回的列(<column>完全匹配）
- SELECT X.a FROM (SELECT a FROM A) AS X：指定表名和列名，表来自子查询且是别名引用，列属于子查询返回的列(<column>完全匹配）
- SELECT X.x FROM (SELECT A.a AS x, b FROM A JOIN B ...) AS X：指定表名和列名，表来自子查询且是别名引用，列属于子查询返回的列且带别名
- SELECT count(distinct a) from A where b = 123, a不是分区列，且经过where条件筛选后数据量不超过100万条
- SELECT NULL ...：空值列
- SELECT 1, 2 ...：常量值列
- SELECT true ...：常量值（boolean类型的取值使用1或0）列
- SELECT * ...：通配符代表所有列

暂不支持：

- SELECT A.* ...：通配符代表特定表的所有列
- SELECT a, b, a ...：重复列

0.8版本中不支持但是0.9版本的Full MPP Mode支持（详见4.4节）：

- SELECT distinct a | count(distinct a) from A group by b：其中a不是分区列
- SELECT distinct a | count(distinct a) from A where b = 123, a不是分区列，且经过where条件筛选后数据量超过100万条

当 select_expr 中使用列表达式时：

例如：

- SELECT a + b ...：常用算术操作符（+，-，*，/, %），其中表达式（a，b）是合法的值、基本列或列表达式，但必须是数值类型
- SELECT min(a), UDF_EXAMPLE(a, s) ...：系统内置或用户自定义函数，可以是值、基本列或列表达式，但数据类型要符合函数参数要求
- SELECT CASE WHEN a > 0 THEN a + b ELSE 0 END ...：CASE-WHEN表达式，其中表达式（a >

0）是合法的 **行过滤表达式**（参看后面），THEN和ELSE取值是合法的 **列表达式** 而且数据类型相同（int）
- SELECT a IS NULL, a IS NOT TRUE, a >= 0, s LIKE '%foo' ...：合法的 **关系表达式**（参看后面）
- SELECT ! a, NOT s ...：合法的 **逻辑表达式**（参看后面）

暂不支持：

- SELECT a IS [NOT] TRUE, a BETWEEN 1 AND 2, a IN (1, 2), a CONTAINS (1, 2, 3)：不支持部分关系表达
- SELECT a XOR b：不支持逻辑表达式（参看4.3节）
- SELECT a & b FROM A, B：不支持位操作

当 select_expr 为列聚集函数表达式时：

包含任何聚集函数的列表达式，即聚集表达式：

例如：

- SELECT COUNT(*)：常用聚合函数，详见附件中函数表
- SELECT UDF_SYS_COUNT_COLUMN(a) FROM A：系统内置聚合函数（UDF_SYS_...）
- SELECT SUM(CASE WHEN ... THEN ... ELSE ... END)：聚合表达式（SUM(...)）套列普通表达式（CASE WHEN ...）做为其子表达式
- SELECT SUM(a) / COUNT(a) FROM A：列普通表达式（/）套聚合表达式（做为其子表达式）

0.8版本中不支持但是0.9版本的Full MPP Mode支持（详见4.4节）：

- SELECT a + COUNT(*) FROM A：普通表达式（+）不能同时套普通表达式（a）和聚合表达式（COUNT(...)）；
- SELECT SUM(COUNT(*))：聚合表达式不能套聚合表达式做为其子表达式；

FROM/JOIN子句（表扫描、连接与子查询）

用法：

```
FROM table_references
```

或

```
FROM table_referencesA as table_aliasA JOIN table_referencesB as table_aliasB on join_conditions
```

或

```
FROM (SELECT ...) as aliasA JOIN table_referencesB as table_aliasB on join_conditions
```

或

```

```

```
FROM (SELECT ...) as aliasA JOIN (SELECT ...)as aliasB on join_conditions
```

例如：

- FROM A：指定表
- FROM (SELECT ...)：子查询
- FROM A JOIN B ON A.a = B.b JOIN C ON A.a = C.c：单表或多表连接，默认是符合条件记录的笛卡尔集。需要注意的是，目前分析型数据库中使用JOIN时若一方是物理表，那么物理表参与JOIN的列必须是分区列并且已建立HASH索引
- FROM (SELECT ...) AS X JOIN A on X.x = A.a：表、子查询和JOIN结合使用
- SELECT ... FROM A JOIN (SELECT ...)：在V0.6.x及后续版本支持
- SELECT ... FROM (SELECT ...) JOIN (SELECT ...)：两个子查询之间的Join，在V0.6.x及后续版本支持

不支持或语义错误：

- SELECT ... FROM A FULL JOIN B：不支持全连接；
- SELECT ... FROM A RIGHT JOIN B：不支持右连接，需要转换为左连接；
- SELECT ... FROM A SEMI JON B：不支持半连接；
- SELECT ... FROM A, B：单表或多表连接，但没有ON条件
- SELECT ... FROM A, B WHERE A.a = B.b：单表或多表连接在WHERE子句中有隐含ON条件，但是没有on子句的，暂不支持

0.8版本中两个表关联可运行的充要条件（四原则）：

（1）两个表均为事实表且在同一个表组，或两个表中有一个是维度表。

（2）两个表均为事实表且拥有相同的一级分区列，或两个表中有一个是维度表。

（3）两个表均为事实表且关联条件(on)中至少含有一个条件是两个表各自的分区列的等值关联条件，或两个表中有一个是维度表。

（4）关联条件（on）中的条件两端有有效的HashMap索引

0.9版本中两个表关联可加速运行的条件：

（1）两个表均为事实表且在同一个表组，或两个表中有一个是维度表。

（2）两个表均为事实表且拥有相同的一级分区列，或两个表中有一个是维度表。

（3）两个表均为事实表且关联条件(on)中至少含有一个条件是两个表各自的分区列的等值关联条件，或两个表中有一个是维度表。

0.9版本若使用Full MPP Mode支持任意形态的表关联（详见4.4节），该功能目前邀请少数客户测试，后续全面开放。

WHERE子句（过滤表达式）

在过滤表达式中，单个条件之间可以用AND和OR进行连接，支持括号决定条件表达式的优先级，亦支持表达式嵌套。

像普通的SQL一样，单个条件的组成是一个布尔判断。暨无论如何组合，单个条件最终是一个返回True/False的表达式。在最简单的情况下，单个条件由左侧的主体、中间的比较操作符和右侧的断言值组成。

例如：column_name IS NOT NULL 中， column_name 为布尔主体，IS NOT为比较操作符，NULL为断言值。

分析型数据库中比较操作符支持 = 、 >= 、 > 、 <= 、 < 、 <> 、 != 等基本操作符，也支持IS、IN、CONTAINS、BETWEEN...AND... 、LIKE这样的关系操作符，并且关系操作符均支持其反义的版本（IS NOT 、 NOT IN 、 NOT LIKE等）。

布尔主体可以是一个列，也可以是表达式的嵌套。

Note

- IN()语句中值的个数默认不能超过400个，如需要在IN()语句中使用超过400个值，请联系技术支持。

例如：

- WHERE a > 0： 关系表达式（ > ，>= ，< ，<= ，= ，<> ，!= ）
- WHERE a IS [NOT] NULL：关系表达式，判断是否为空值
- WHERE a BETWEEN 1 AND 10：对于表达式（a）是数字类型（long，int，short，double，boolean）和时间类型（date，time，timestamp）支持
- WHERE s LIKE '%sample'：对于表达式（s）是字符串（string）类型支持
- WHERE a IN (1,2,3)：对于表达式（a）是多值列类型，或普通类型支持
- WHERE a IN (select a from dim_table where c=1)：对于对于表dim_table是维度表，支持IN后带子查询（V0.6.2之后的版本支持）
- WHERE a CONTAINS (1,2,3)：对于表达式（a）是多值列类型支持
- WHERE a > 0 AND b > 0：逻辑表达式
- WHERE (a >0 OR b > 0) AND c > 0：支持括号决定条件表达式的优先级
- WHERE (CASE WHEN (a + b) > 0 THEN 0 ELSE 1 END) != 0 AND UDF_EXAMPLE(a, b) > 0：表达式嵌套。

不支持或语义错误：

- WHERE EXISTS (SELECT...)：不支持
- WHERE a = ANY (SELECT ...)：不支持
- WHERE a = ALL (SELECT ...)：不支持

0.8版本中不支持但是0.9版本的Full MPP Mode支持（详见4.3节）：

- WHERE a IN (select a from fact_table where c=1): fact_table不是维度表的情况下0.8版本不支持

GROUP BY/HAVING子句（分组和分组后过滤）

GROUP BY子句的用法：

```
GROUP BY {col_expr}
```

col_expr为列（表中的列或子查询、别名产生的列）。

例如：

- `... FROM A GROUP BY a：仅指定列
- `... FROM A GROUP BY A.a：同时指定表名和列
- GROUP BY s, a IS NULL, b + c, UDF_EXAMPLE(d)：一个或多个列表达式
- SELECT a, SUM(a) FROM A GROUP BY a, s：可以包含不属于列透视的列表达式（s`）
- `... FROM (SELECT A.a AS x, b FROM A JOIN B ...) AS X JOIN C ... GROUP BY b, c, X.x：包含在子查询或多表连接的一个或多个列表达式
- `SELECT CASE WHEN a > 1 THEN ‘Y’ ELSE ‘N’ END AS x ... GROUP BY x ：别名引用产生的列表达式
- SELECT a, b, c ... GROUP BY 3, 1：按位置引用列投射的列表达式

0.8版本中不支持但是0.9版本的Full MPP Mode支持（详见4.4节）：

- ... FROM A GROUP BY s ORDER BY COUNT(a) LIMIT 100：分区表达式（s）不是分区列（a）同时又有TOP-N（ORDER BY COUNT(a) LIMIT 100）0.8版本下计算的结果可能是不精确的（详见4.3节）

HAVING子句的用法：

HAVING子句用于在GROUP BY子句执行后对分组后的结果进行过滤。支持使用列投射和聚合表达式后的结果进行过滤。

```
HAVING having_condition
```

例如：

- SELECT a, sum(length(s)) AS x FROM A JOIN B ON A.a = B.b GROUP BY a HAVING (CASE WHEN (x / a > 1) THEN a ELSE 0 END) != 0：分组表达式（a）是分区列，HAVING表达式是基于列表达式（a，sum(length(s))，s）的行过滤表达式

0.8版本中不支持但是0.9版本的Full MPP Mode支持（详见4.4节）：

- SELECT s, count(a) AS x FROM A GROUP BY s HAVING x > 1：分组表达式（s）不是分区列（a）

ORDER BY子句（排序）

例如：

- SELECT a FROM A ORDER BY a DESC LIMIT 100：按照分区列（a）排序并返回TOP-N（100）
- SELECT a FROM A ORDER BY (a + b) DESC LIMIT 100：按照非分区列（a）排序并返回TOP-N（100）
- SELECT a, COUNT(a) FROM A GROUP BY a ORDER BY x：按照分区列（a）分组聚合再排序

0.8版本中不支持但是0.9版本的Full MPP Mode支持（详见4.4节）：

- SELECT a + b, COUNT(1) AS x FROM A GROUP BY a + b ORDER BY x：0.8版本下按照非分区列表达式（a+b）分组聚合再取TOP-N是非精确的（详见4.3）

UNION [ALL]/INTERSECT/MINUS子句

UNION/INTERSECT/MINUS 0.8版本下仅支持操作字段为分区列，但是0.9版本的Full MPP Mode无限制（详见4.4节）。

UNION ALL无任何限制。

LIMIT子句（返回行数限制）

示例：

- SELECT ... LIMIT 100：返回行限制为100行

不支持：

- SELECT ... LIMIT 100, 50：不支持带偏移量的返回行限制

需要注意的是，分析型数据库会对通过SELECT语句进行查询的返回行数做全局的最大限制，公共云上目前为10000行，若不加LIMIT或LIMIT行数超过10000，则只会返回10000行。

基本语法

分析型数据库支持通过查询方式进行表数据复制，目标表必须是实时写入表：

- 提供列名的方式，请确保SELECT的列表与提供的INSERT目标表的列匹配（顺序、数据类型）：

```
INSERT INTO db_name.target_table_name (col1, col2, col3)
SELECT col1, col2, col3 FROM db_name.source_table_name
WHERE col4 = 'xxx';
```

- 不提供列名的方式，请确保SELECT的列表与提供的INSERT目标表的列匹配（顺序、数据类型）：

```
INSERT INTO db_name.target_table_name
SELECT col1, col2, col3, col4 FROM db_name.source_table_name
WHERE col4 = 'xxx';
```

可通过如下语句查询表的列定义的顺序:

```
SHOW CREATE TABLE db_name.target_table_name;
```

多种引擎模式下的执行

目前，INSERT from SELECT支持多种执行模式，在LM模式下：

- SELECT部分的查询走FRONTNODE + COMPUTENODE的两阶段模式，该模式下具有最好的执行性能，但是SELECT的查询部分不会做最终的数据聚合，所以，查询要考虑是否符合按HASH分区分片计算ONLY的模式，否则写入的数据不保证整体语义的正确性、完整性，默认执行引擎为COMPUTENODE 时，无需加 /*+engine=COMPUTENODE*/ hint：

```
/*+engine=COMPUTENODE*/INSERT INTO db_name.target_table_name (col1, col2, col3)
SELECT col1, col2, col3 FROM db_name.source_table_name
WHERE col4 = 'xxx';
```

分析型数据库的查询模式适合在海量数据中进行分析计算后输出适量数据，若需要输出的数据量达到一定规模，分析型数据库提供数据导出（DUMP）的方式。注意目前DUMP方式中不能使用针对非分区列的聚合函数。

DUMP TO ODPS

通过类DML语句导出到MaxCompute(ODPS)

前提须知:

```
--注意正确输入需要授权的表命名、project和正确的云账号
USE prj_name; --表所属ODPS project
ADD USER ALIYUN$xxxx@aliyun.com;
GRANT createInstance ON project prj_name TO USER ALIYUN$xxxx@aliyun.com;
GRANT Describe,Select,alter,update,drop ON TABLE table_name
TO USER ALIYUN$xxxx@aliyun.com;
```

导出命令

类似于普通的SQL查询语句，用户也可通过类似于DML语句进行数据导出。

语法格式:

```
DUMP DATA [OVERWRITE] INTO 'odps://project_name/table_name'
SELECT C1, C2 FROM DB1.TABLE1 WHERE C1 = 'xxxx' LIMIT N;

DUMP DATA [OVERWRITE] INTO 'odps://project_name/table_name/ds=xxxxxxxx/xxx=xxx'
SELECT C1, C2 FROM DB1.TABLE1 WHERE C1 = 'xxxx' LIMIT N
```

注意：

- 目标ODPS表必须包含和SELECT查询选择的列名（别名，如果有别名的话）相同的列，先后顺序必须一致；

- 当指定 OVERWRITE 时，会先DROP目标ODPS表，重新创建后再导出数据；
- 当不指定 OVERWRITE 时，会append追加到目标ODPS表中。

DUMP TO OSS

通过类DML语句导出到表格存储(OSS)

前提须知:

```
--注意请申请和您所使用Analyticdb对应集群下的OSS实例，否则可能会由于网络原因导致失败
```

导出命令

类似于普通的SQL查询语句，通过JDBC接口访问，如果成功则会返回一行记录，其中包含了您指定dump的OSS路径。

语法格式:

```
/*+ dump-col-del=[,], dump-row-del=[\n], dump-oss-accesskey-id=oss的ACCESS_KEY_ID,
dump-oss-accesskey-secret=oss的ACCESS_KEY_SECRET*/
DUMP DATA [OVERWRITE] INTO
'oss://endpoint_domain/bulket_name/filename'
SELECT C1, C2 FROM DB1.TABLE1 WHERE C1 = 'xxxx' LIMIT N
```

注意：

- 其中必须在Hint中指定有目标OSS路径写入权限的AK
- 由于OSS中的文件时非结构化存储，为了方便您的使用，我们提供了可以任意指定行列分隔符的配置，通过在Hint中指定 dump-col-del / dump-row-del 即可生效，注意’[]’内为具体的分隔符
- 目标路径最多只支持一级文件夹，目前暂不支持多级目录指定
- 目标路径的文件夹需已经创建成功，目前暂不支持动态创建目录
- 当指定 OVERWRITE 时，会对原有文件进行覆盖写
- 当不指定 OVERWRITE 时，会对原有文件进行追加写

关于返回数据行数(适用于DUMP TO ODPS 和 DUMP TO OSS)

导出方式对海量数据的计算输出具有良好的性能（百万行数据导出在数百毫秒数据级），但是， 对于数据精确度有一定牺牲，即实际返回的数据行数，可能是不完全精确。以限制导出行数为1000为例（LIMIT 1000）:

- 实际数据行数可能稍大于1000，例如此时有120个数据分片，则等同于每个分区明确指定”LIMIT 9”，最多可能返回1080；
- 实际数据行数可能稍小于1000，如果符合条件的行数的总数小于1000；

- 实际数据行数可能稍小于1000，如果数据分片很均匀，例如此时有120个数据分片，如果某些分片返回数据行小于9的话，则等同于每个分区明确指定”LIMIT 9”。
- 可以通过hint指定显示返回行数（默认情况下不显示数据行数），可以通过return-dump-record-count=TRUE来指定返回时显示数据行数。

CTAS（Create Table As Select）支持通过查询方式进行建表和表数据复制，目标表必须是实时写入表。

CTAS在系统内部处理时，分为如下几个步骤：

- Step 1: 执行CREATE TABLE建表，和其他实时表建表步骤一致
- Step 2: 执行实时表的LOAD DATA表上线操作
- Step 3: 执行INSERT from SELECT操作，关于INSERT from SELECT介绍，请参考 :ref:insert-from-select
- Step 4: 执行实时表的Merge Baseline，对刚刚INSERT的实时数据进行基线合并

Note

- 目前AnalyticDB并不保证CTAS整体操作流程的原子性，其中，建表（Step 1）、表上线（Step 2）都是原子的，如果失败，表会回滚
- 然而INSERT from SELECT实时数据写入（Step 3并不保证成功，可能出现数据部分插入成功的状态，业务使用CTAS时，需要从业务应用角度考虑-在ETL流程中使用CTAS（包括INSERT from SELECT），一种保证业务完整性的做法是通过向目标表插入标记数据的方法:
    - INSERT from SELECT时，向目标表插入一条标记数据
    - 执行实时表的FLUSH操作
    - 查询标记数据，保证标记数据入库成功，如果未成功，可结合业务情况，考虑删表和重试操作

CTAS中完全定义列的情况下，语法示例:

```
CREATE TABLE IF NOT EXISTS weitao.table1 (
user_id long COMMENT 'user ID',
seller_id long COMMENT 'seller ID',
follow_id long COMMENT 'user_id follower',
score double COMMENT 'user_id score',
interest_flag multivalue COMMENT 'multiple value column' delimiter ',',
primary Key (user_id, seller_id)
)
PARTITION BY HASH KEY(user_id) PARTITION NUM 128
CLUSTERED BY (follow_id,score)
TABLEGROUP weitao_group12
SELECT * FROM db1.table1 WHERE col1 = 3;
```

其中，指定 IF NOT EXISTS 时，当目标表已经存在时，不报错。

CTAS中不定义列的情况下，语法示例:

```
CREATE TABLE bar (PRIMARY KEY (n)) SELECT n FROM foo;
```

其中，目标表不指定列定义，则列定义采用SELECT子句中返回的列名和列定义，但是主键必须定义。

在ADS的FRONTNODE节点构建本地local的内置DB引擎，能够存储一定量的本地数据缓存表（Cache Table），进行快速本地单表查询。基于Cache table，可进行一定范围内的高效的分页数据查询。

实现方案介绍

- 存储：Cache Table存储在FRONTNODE本地节点上，每个节点上有多个Cache Table存储分桶，桶内采取换入换出机制，每个FRONTNODE上最大能容纳的Cache Table数据量为 256MB * FRONTNODE所在物理机的总核数 ；从业务角度来说，Cache Table为临时存储，不能作为永久性存储，主要用来加速局部数据复用查询和分页查询。
- 查询：Cache Table只支持单表查询。
- 管理：业务要通过捕获相应SQLException异常错误码，来感知Cache Table的失效，并决定是否进行重建。后续ADS会涉及Cache Table的自管理方案。目前业务方需要从业务侧维护Cache ID与Cache Table之间的映射。

使用介绍

1）创建Cache Table：

创建Cache Table的Sample语句:

```
CREATE TABLE cache.table_name OPTIONS(cache=true) AS SELECT * FROM yt_vip_user_tag LIMIT 200;
```

其中table_name为业务定义的Cache Table名, AS后面为结果需要存入Cache Table的查询语句。 成功执行后，会返回单列一行数据： cache_id ，用来唯一标识这张位于某台特定FRONTNODE节点上Cache Table:

```
CREATE TABLE cache.test_cache_table_1 OPTIONS(cache=true) AS SELECT * FROM yt_vip_user_tag LIMIT 200;
```

```
+---------------------------------------------+
| cache_id |
+---------------------------------------------+
| 1683065103.38806.6.0.082539 |
+---------------------------------------------+
```

后续查询这张表，都需要带上cache_id的HINT进行查询。

2）单个Cache Table行数限制：

Cache Table单表存储的最大行数限制在**300000**行。 假设select * from yt_vip_user_tag查询会返回999999行数据，若针对该查询结果创建Cache Table，会报错，错误码为 18066 ，错误消息为：”

EXECUTE_CREATE_CACHE_TABLE_ERROR message=Exceed the max cache table row limitation: 300001”。:

```
mysql> CREATE TABLE cache.test_cache_table_1 OPTIONS(cache=true) AS select * from yt_vip_user_tag;
ERROR 18066 (HY000): Error code: 18066, message: EXECUTE_CREATE_CACHE_TABLE_ERROR message=Exceed the max cache table row limitation: 300001
```

注意：创建Cache Table查询的SELECT语句不要加LIMIT子句，为保证创建Cache Table的查询结果集行数在限制（ 300000 ）以内，可以预先执行:

```
SELECT count(*) FROM (SELECT * FROM yt_vip_user_tag WHERE col1 = 2);
```

来确保创建该Cache Table的查询结果集在限制以内。

3）查询Cache Table：

每张Cache Table对应一个自己唯一的 Cache ID （创建时产生），每次查询Cache Table时，都需要带上该Cache ID 的hint，例如:

```
/*+cache_id=1683065103.38806.6.0.082539*/select * from cache.test_cache_table_1;
```

Cache Table支持分页查询，例如:

```
/*+cache_id=1683065103.38806.6.0.082539*/select * from cache.test_cache_table_1 limit 10, 5;
```

> **Note**
>
> 注意：在使用Cache Table实现分页方案时，生成的Cache Table仅仅为单表，最佳实践是仅仅对生成的Cache Table进行分页明细数据查询，请勿针对Cache Table进行复杂聚合计算和窗口函数分析查询。

4）查询Cache Table时的异常处理：

查询Cache Table异常分两种类型，查询本身的用户异常，即用户查询写错了，该类 SQLException 异常的错误码为 30018 ，用户需要检查查询是否正确； 另外一类，是Cache Table系统异常，该类 SQLException 异常的错误码为 30016 ，遇到 30016 ，用户需要考虑重建Cache Table。该类异常发生原因有：

- FRONTNODE节点重启、发布升级导致位于该FRONTNODE节点上的Cache Table失效；
- FRONTNODE节点上用来容纳Cache Table的bucket满，触发了换出的淘汰机制，导致目标Cache Table失效。

5）删除Cache Table：

业务可以删除指定的Cache Table，和查询一样，需要指定对应的Cache ID:

```
/*+cache_id=1683065103.38806.6.0.082539*/drop table cache.test_cache_table_1;
```

分析型数据库在v2.0(0.9.45)后提供了两种hint用于优化查询。相关hint的使用建议咨询专业人员。

no-index

用于指定某列不使用索引。一般用于在多个where过滤条件中，存在1-2个条件筛选率非常低，导致索引装载时间消耗很长，反而不如不用索引时。

示例：

```
select a1, a2, count(distinct a3) rs from  a join  b on a.a1 = b.a1 and b.a4 in ('110', '120') and a.aa3 = 1003 and b.aa2
<= 201503 group by a1, a2;
```

该sql中，aa2筛选率很差，所以应该增加no-index hint

```
/*+no-index=[b.aa2]*/
select a1, a2, count(distinct a3) rs from a join b on a.a1 = b.a1 and b.a4 in ('110', '120') and a.aa3 = 1003 and b.aa2
<= 201503 group by a1, a2;
```

nocache

不常用，用于某些列在查询中不适合进入缓存时指定其不进入缓存。

用法：

```
/*+no-cache=[colname]*/
select ...
```

4.3 MPP计算引擎

MPP计算引擎模式下，INSERT/DELETE的基本语法与LM计算引擎模式一致，请参见“4.2.1 INSERT/DELETE语法”章节。

语法描述

```
[ WITH with_query [, ...] ]
SELECT [ ALL | DISTINCT ] select_expr [, ...]
[ FROM from_item [, ...] ]
```

```
[ WHERE condition ]
[ GROUP BY [ ALL | DISTINCT ] grouping_element [, ...] ]
[ HAVING condition]
[ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
[ ORDER BY expression [ ASC | DESC ] [, ...] ]
[ LIMIT [ count | ALL ] ]
```

其中 from_item 为以下之一

```
table_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]

from_item join_type from_item [ ON join_condition | USING ( join_column [, ...] ) ]
```

并且 join_type 为以下之一

```
[ INNER ] JOIN
LEFT [ OUTER ] JOIN
RIGHT [ OUTER ] JOIN
FULL [ OUTER ] JOIN
CROSS JOIN
```

并且 grouping_element 为以下之一

```
()
expression
GROUPING SETS ( ( column [, ...] ) [, ...] )
CUBE ( column [, ...] )
ROLLUP ( column [, ...] )
```

WITH 从句

这个 WITH 从句定义了一个命名好的关系以供在一个查询里面使用。 它可以扁平化嵌套查询或者简化子查询。例如，下面的查询是等价的:

```
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;
```

也同意适用于多子查询的情况:

```
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;
```

另外, 在 WITH 分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;
```

GROUP BY从句

这个 GROUP BY 分句对 SELECT 语句的输出进行分组，分组中是匹配值的数据行。一个简单的 GROUP BY 分句可以包含 由输入列组成的任何表达式或者或列序号（从1开始）.

以下查询是等价的。 他们都对 nationkey 列进行分组， 第一个查询使用列序号， 第二个查询使用列名:

```
SELECT count(*), nationkey FROM customer GROUP BY 2;

SELECT count(*), nationkey FROM customer GROUP BY nationkey;
```

在查询语句中没有指定列名的情况下， GROUP BY 子句也可以将输出进行分组。 例如，以下查询使用列 mktsegment 进行分组， 统计出 customer 表的行数。:

```
SELECT count(*) FROM customer GROUP BY mktsegment;
```

```
_col0
-------
29968
30142
30189
29949
29752
(5 rows)
```

在 SELECT 语句中使用 GROUP BY 子句时， 所有输出的列要么是聚会函数， 要么是 GROUP BY 子句中的列。

Complex Grouping Operations

AnalyticDB MPP 同样也支持复杂的聚合函数使用 GROUPING SETS, CUBE 和 ROLLUP 语法。 这个语法允许用户执行那些需要在一个单一查询上在多个列上聚合的分析查询。复杂的分组运算不支持在由输入列组成的表达式上做分组。只有列名和序列号是被允许的。

复杂的分组运算符经常等价于 UNION ALL 和 GROUP BY 的组合表达式, 正如下面的例子所示。 然而当参与聚合的数据源不是确定性的时候，这个等价就不存在了。

GROUPING SETS

允许用户去指定多个包含列的列表去做分组。那些不属于分组列集合中的子列表中的列被设置为 NULL。

```
SELECT * FROM shipping;


origin_state | origin_zip | destination_state | destination_zip | package_weight
--------------+------------+-------------------+-----------------+----------------
California | 94131 | New Jersey | 8648 | 13
California | 94131 | New Jersey | 8540 | 42
New Jersey | 7081 | Connecticut | 6708 | 225
California | 90210 | Connecticut | 6927 | 1337
California | 94131 | Colorado | 80302 | 5
New York | 10002 | New Jersey | 8540 | 3
(6 rows)
```

GROUPING SETS 语法上可以参考下面的示范:

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


 origin_state | origin_zip | destination_state | _col0
--------------+------------+-------------------+-------
New Jersey | NULL | NULL | 225
California | NULL | NULL | 1397
New York | NULL | NULL | 3
California | 90210 | NULL | 1337
California | 94131 | NULL | 60
New Jersey | 7081 | NULL | 225
New York | 10002 | NULL | 3
NULL | NULL | Colorado | 5
NULL | NULL | New Jersey | 58
NULL | NULL | Connecticut | 1562
(10 rows)
```

前一个查询可以被认为和一个包含多个 GROUP BY 的 UNION ALL 在逻辑上是等价的:

```
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;
```

然而，带有复杂分组语法的查询(GROUPING SETS, CUBE 或者 ROLLUP) 将只从潜在的数据源读一次，然而带有UNION ALL的查询读了潜在的数据源3次。 这就是为什么当数据源不确定的时候，带着UNION ALL的查询有可能会产生不一致的结果。

CUBE

CUBE 运算符为一个指定的列的集合生成所有可能的分组集合。例如, 下面的查询:

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

和下面的查询等价:

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

```
 origin_state | destination_state | _col0
--------------+-------------------+-------
California | New Jersey | 55
California | Colorado | 5
New York | New Jersey | 3
New Jersey | Connecticut | 225
California | Connecticut | 1337
California | NULL | 1397
New York | NULL | 3
New Jersey | NULL | 225
NULL | New Jersey | 58
NULL | Connecticut | 1562
NULL | Colorado | 5
NULL | NULL | 1625
(12 rows)
```

ROLLUP

这个 ROLLUP 运算符为一个指定列的集合生成所有可能的subtotals。例如下面的查询:

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

```
origin_state | origin_zip | _col2
--------------+------------+-------
California | 94131 | 60
California | 90210 | 1337
New Jersey | 7081 | 225
New York | 10002 | 3
California | NULL | 1397
New York | NULL | 3
New Jersey | NULL | 225
NULL | NULL | 1625
(8 rows)
```

和下面的查询是等价的:

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

合并多个分组表达式

多个分组表达式在同一个查询被翻译成跨产品语义。例如, 下面的查询:

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

可以被重写为:

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

在逻辑上和下面的查询等价:

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

```
 origin_state | destination_state | origin_zip | _col3
--------------+-------------------+------------+-------
New York | New Jersey | 10002 | 3
California | New Jersey | 94131 | 55
New Jersey | Connecticut | 7081 | 225
California | Connecticut | 90210 | 1337
California | Colorado | 94131 | 5
New York | New Jersey | NULL | 3
New Jersey | Connecticut | NULL | 225
California | Colorado | NULL | 5
California | Connecticut | NULL | 1337
California | New Jersey | NULL | 55
(10 rows)
```

ALL 和 DISTINCT 这两个量词决定了是否每一个重复的分组集合都产生出唯一的输出行。 这个尤其在多个复杂分组集合被组合在一个查询里的时候有用。 例如，下面的查询:

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

和下面的查询等价:

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

然后, 如果查询使用 DISTINCT 量词在 GROUP BY 的时候

```
SELECT origin_state, destination_state, origin_zip, sum(package_weight)
FROM shipping
GROUP BY DISTINCT
CUBE (origin_state, destination_state),
ROLLUP (origin_state, origin_zip);
```

只有唯一的分组集合将被生成出来:

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

默认的量词是 ALL.

GROUPING Operation

grouping(col1, ..., colN) -> bigint

这个分组操作符返回一个由bit set转换成的decimal, 指定了哪些列在一个分组中出现。 这个必须要和 GROUPING SETS, ROLLUP, CUBE 或者 GROUP BY 连接使用。 并且他的参数必须严格和在 GROUPING SETS, ROLLUP, CUBE 或者 GROUP BY 分句中引用的值相匹配。为了计算特定行的结果位集，将位分配给参数列，最右边的列是最低有效位。 对于一个给定分组，一个bit被设置成0如果相应的列包含在分组中，否则设置成1。例如, 下面的分组:

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


origin_state | origin_zip | destination_state | _col3 | _col4
--------------+------------+-------------------+-------+-------
California | NULL | NULL | 1397 | 3
New Jersey | NULL | NULL | 225 | 3
New York | NULL | NULL | 3 | 3
California | 94131 | NULL | 60 | 1
New Jersey | 7081 | NULL | 225 | 1
California | 90210 | NULL | 1337 | 1
New York | 10002 | NULL | 3 | 1
NULL | NULL | New Jersey | 58 | 6
NULL | NULL | Connecticut | 1562 | 6
NULL | NULL | Colorado | 5 | 6
(10 rows)
```

上面的第一个分组的结果只包含 origin_state 列并且排除了 origin_zip 和 destination_state 这两列。这个bit set在这种情况被构建成 011 这里最重要的位代表了 origin_state.

HAVING 从句

HAVING 子句与聚合函数以及 GROUP BY 子句共同使用， 用来控制选择分组。 HAVING 子句去掉不满足条件的分组。 在分组和聚合计算完成后，HAVING 对分组进行过滤。 以下示例查询 customer 表，并进行分组，查出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
(7 rows)
```

UNION | INTERSECT | EXCEPT 从句

UNION INTERSECT 和 EXCEPT 都是全集合操作符。这些分句被用来组合多于一个查询语句的结果最终形成一个结果:

query UNION [ALL | DISTINCT] query

query INTERSECT [DISTINCT] query

query EXCEPT [DISTINCT] query

参数 ALL 或 DISTINCT 控制最终结果集包含哪些行。如果指定参数 ALL ，则包含全部行，即使行完全相同。如果指定参数 DISTINCT ， 则合并结果集，结果集只有唯一不重复的行。如果不指定参数，执行时默认使用 DISTINCT。参数 ALL 不支持使用 INTERSECT 或者 EXCEPT

多个集合操作符会从做到有的被处理，除非顺序通过括弧被显示指定。另外，INTERSECT 比 EXCEPT and UNION 有更高的优先级，这意味着 A UNION B INTERSECT C EXCEPT D 和这个表达式是相同的 A UNION (B INTERSECT C) EXCEPT D.

UNION

UNION 把所有结果集两个结果集合并起来。下面是一个最简单的可能使用 UNION 分句的例子。它选择了值 13 并且合并了第二个选择的值 42，把他们结合起来:

```
SELECT 13
UNION
SELECT 42;
```

```
 _col0
-------
13
42
(2 rows)
```

下面的查询演示了 UNION 和 UNION ALL 之间的不同。 它选择了值 13 并把它和第二次查询钻展的值 42 和 13 做合并:

```
SELECT 13
UNION
SELECT * FROM (VALUES 42, 13);


 _col0
-------
13
42
(2 rows)


SELECT 13
UNION ALL
SELECT * FROM (VALUES 42, 13);


 _col0
-------
13
42
13
(2 rows)
```

INTERSECT

INTERSECT 只返回那些在同时在第一个和第二个查询里面都出现的行的结合。下面的例子是一个最简单的可能使用 INTERSECT 分句的例子。 它选择了值 13 和 42 并把他们和第二个查询选择的值 13 做合并。 既然 42 值在第一个查询的结果集中, 它并不会被包含在最终的结果集里面:

```
SELECT * FROM (VALUES 13, 42)
INTERSECT
SELECT 13;


 _col0
-------
13
(2 rows)
```

EXCEPT

EXCEPT 返回那些行仅存在于第一个查询结果集不在第二个查询结果集中。下面是最简单的使用 EXCEPT 分句的例子。 它选择了值 13 和 42 并把他们和第二个查询选择的值 13 做合并。 既然 13 也同时存在在第二个查询结果集中, 它不会被包含在最终的结果集中:

```
SELECT * FROM (VALUES 13, 42)
EXCEPT
SELECT 13;


 _col0
-------
42
(2 rows)
```

ORDER BY Clause

ORDER BY 分句被用来排序一个结果集通过一个或者多个输出表达式:

ORDER BY expression [ASC | DESC] [NULLS { FIRST | LAST }] [, ...]

每个表达式由列名或列序号（从1开始）组成。 ORDER BY 子句作为查询的最后一步， 在 GROUP BY 和 HAVING 子句之后。

LIMIT Clause

LIMIT 分句限制了最终结果集的行数。 LIMIT ALL 和略去 LIMIT 分句的结果一样。 以下示例为查询一个大表 ， LIMIT 子句限制它只输出5行（因为查询没有 ORDER BY， 所以随意返回几行）:

SELECT orderdate FROM orders LIMIT 5;

```
 o_orderdate
-------------
1996-04-14
1992-01-15
1995-02-01
1995-11-12
1992-04-26
(5 rows)
```

TABLESAMPLE

有多种抽样方法:

BERNOULLI

查出的每行记录都源于表样本，使用样本百分比概率。 当使用 BERNOULLI 方法对表进行抽样时，会扫描表的所有物理块， 并跳过某些行。 （基于样本百分比与运行时随机计算之间的比较）

- 结果中每行记录的概率都是独立的。 这不会减少从磁盘读取抽样表所需要的时间。 如果对抽样输出做处理， 它可能对整体查询时间有影响。

SYSTEM

这种抽样方法将表划分为逻辑数据段， 并按此粒度进行抽样。 这种抽样方法要么从特定数据段查询全部行， 要么跳过它。 （基于样本百分比与运行时随机计算之间的比较）

- 系统抽样选取哪些行，取决于使用哪种连接器。 例如，使用Hive， 它取决于HDFS上的数据是怎样存储的。 这种方法无法保证独立抽样概率。

> **Note**
>
> 这两种方法都不能确定返回行数的范围。

示例:

```
SELECT *
FROM users TABLESAMPLE BERNOULLI (50);

SELECT *
FROM users TABLESAMPLE SYSTEM (75);
```

通过join进行抽样:

```
SELECT o.*, i.*
FROM orders o TABLESAMPLE SYSTEM (10)
JOIN lineitem i TABLESAMPLE BERNOULLI (40)
ON o.orderkey = i.orderkey;
```

UNNEST

UNNEST 用于展开 数组类型 或 map类型 的子查询。数组展开为单列，map展开为双列（键，值）。 UNNEST 以使用多个参数，它们展开为多列， 行数与最大的基础参数一样（其他列填空）。 UNNEST 可以带一个可选择的 WITH ORDINALITY 分句, 在这个情况下一个额外的序号列将被在最后添加上。 UNNEST 通常与 JOIN 一起使用 也可以引用 JOIN 左侧的关系列。

使用单独的列:

```
SELECT student, score
FROM tests
CROSS JOIN UNNEST(scores) AS t (score);
```

使用多列:

```
SELECT numbers, animals, n, a
FROM (
VALUES
(ARRAY[2, 5], ARRAY['dog', 'cat', 'bird']),
(ARRAY[7, 8, 9], ARRAY['cow', 'pig'])
) AS x (numbers, animals)
CROSS JOIN UNNEST(numbers, animals) AS t (n, a);
```

```
  numbers |     animals      | n   | a
-----------+------------------+------+------
[2, 5] | [dog, cat, bird] | 2 | dog
[2, 5] | [dog, cat, bird] | 5 | cat
[2, 5] | [dog, cat, bird] | NULL | bird
[7, 8, 9] | [cow, pig] | 7 | cow
[7, 8, 9] | [cow, pig] | 8 | pig
[7, 8, 9] | [cow, pig] | 9 | NULL
(6 rows)
```

WITH ORDINALITY 分句:

```
SELECT numbers, n, a
FROM (
VALUES
(ARRAY[2, 5]),
(ARRAY[7, 8, 9])
) AS x (numbers)
CROSS JOIN UNNEST(numbers) WITH ORDINALITY AS t (n, a);
```

```
  numbers  | n | a
-----------+---+---
[2, 5] | 2 | 1
[2, 5] | 5 | 2
[7, 8, 9] | 7 | 1
[7, 8, 9] | 8 | 2
[7, 8, 9] | 9 | 3
(5 rows)
```

Joins

它允许你去合并来自多个关联的数据。

交叉连接

CROSS JOIN 反回关联两边的笛卡尔积。CROSS JOIN 要么被通过 CROSS JOIN 语法要么通过FROM 分句中通过关联指定。

下面两个查询是等价的:

```
SELECT *
FROM nation
CROSS JOIN region;

SELECT *
FROM nation, region;
```

nation 表包含了25行 region 表包含了 5 行, 所以结果两个表最终产生了 125 行:

```
SELECT n.name AS nation, r.name AS region
FROM nation AS n
CROSS JOIN region AS r
ORDER BY 1, 2;

    nation     |   region
---------------+-------------
ALGERIA | AFRICA
ALGERIA | AMERICA
ALGERIA | ASIA
ALGERIA | EUROPE
ALGERIA | MIDDLE EAST
ARGENTINA | AFRICA
ARGENTINA | AMERICA
...
(125 rows)
```

限定列名

当列名在两个关联的表中有相同的名称，它们的引用必须通过关联的别名引用（如果关联有别名），或者通过关联的表名引用:

```
SELECT nation.name, region.name
FROM nation
CROSS JOIN region;

SELECT n.name, r.name
FROM nation AS n
CROSS JOIN region AS r;

SELECT n.name, r.name
FROM nation n
CROSS JOIN region r;
```

下面的查询将会返回错误Column ‘name’ is ambiguous:

```
```

```
SELECT name
FROM nation
CROSS JOIN region;
```

子查询

一个子查询是一个包含了查询的表达式。子查询当它引用子查询之外的列时是相关的。 逻辑上来说, 子查询会被被它的外围查询逐行评估。被引用的列将因此是固定的在子查询的评估过程中。

> Note
>
> 对于向关联的子查询是受限的，并不是每一个形式都是支持的。

EXISTS

EXISTS 断言决定是否一个子查询可以返回任何行:

```
SELECT name
FROM nation
WHERE EXISTS (SELECT * FROM region WHERE region.regionkey = nation.regionkey)
```

IN

IN 断言决定一个子查询返回的值是否在一个被给定的结果集中。IN 的结果依照对 nulls 的标准结果。子查询必须产生仅仅一列:

```
SELECT name
FROM nation
WHERE regionkey IN (SELECT regionkey FROM region)
```

标量子查询

标量子查询是一个非关联的子查询，他会返回零或者1行数据。如果这个子查询返回了多以一行的数据，那将是个错误。如果子查询没有返回任何行，则返回的结果是NULL:

```
SELECT name
FROM nation
WHERE regionkey = (SELECT max(regionkey) FROM region)
```

> Note
>
> 当前仅仅单列可以被用在标量子查询里。

基本语法

MPP计算引擎模式下，INSERT FROM SELECT的基本语法与LM计算引擎模式一致，请参考“4.2.3 INSERT FROM SELECT语法”章节。

多种引擎模式下的执行

在MPP模式下，INSERT FROM SELECT支持两种执行方式：

- SELECT部分的查询走MPP模式，数据批量返回FRONTNODE节点，以批量的方式发起实时数据INSERT，默认一批的记录数为100条：

```
/*+engine=MPP*/INSERT INTO db_name.target_table_name (col1, col2, col3)
SELECT col1, col2, col3 FROM db_name.source_table_name
WHERE col4 = 'xxx';
```

- 上述模式中，所有数据由FRONTNODE节点单点写入，并发度受限，为提高并发度，还有一种模式为Native MPP INSERT from SELECT，数据写入节点直接由多个worker节点并发完成，每次batch一批的记录数还是100条, 通过指定 mppNativeInsertFromSelect=true 的hint走这种模式：

```
/*+engine=MPP, mppNativeInsertFromSelect=true*/INSERT INTO db_name.target_table_name (col1, col2, col3)
SELECT col1, col2, col3 FROM db_name.source_table_name
WHERE col4 = 'xxx';
```

Full MPP Mode下使用Dump to ODPS

通过 engine=MPP 的hint和 dump-header 的hint执行MPP Dump to ODPS，示例:

```
/*+ engine=MPP, dump-header=[DUMP DATA [OVERWRITE] INTO 'odps://project_name/table_name']*/ SELECT ...
/*+ engine=MPP, return-dump-record-count=TRUE, dump-header=[DUMP DATA [OVERWRITE] INTO
'odps://project_name/table_name']*/ SELECT ...
```

Full MPP Mode下使用Dump to OSS

通过 engine=MPP 的hint和 dump-header 的hint执行MPP Dump to OSS，示例:

```
/*+ engine=MPP,
dump-col-del=[,],
dump-row-del=[\n],
dump-oss-accesskey-id=xxxx,
dump-oss-accesskey-secret=xxxx,
dump-header=[DUMP DATA OVERWRITE INTO 'oss://xxx/xxx/xxx']*/ SELECT ...
```

```
/*+ engine=MPP,
return-dump-record-count=TRUE,
dump-col-del=[,],
dump-row-del=[\n],
dump-oss-accesskey-id=xxxx,
dump-oss-accesskey-secret=xxxx,
dump-header=[DUMP DATA OVERWRITE INTO 'oss://xxx/xxx/xxx']*/ SELECT ...
```

除hint外，MPP计算引擎模式下的DUMP导出命令与LM计算引擎模式一致，请参见“4.2.4 DUMP语法”章节。

MPP计算引擎模式下，CTAS的基本语法与LM计算引擎模式一致，请参见“4.2.5 CTAS”章节。

在MPP计算引擎模式下，Cache Table的使用方式与LM计算引擎模式一致，请参见“4.2.6 Cache Table”章节。

VALUES语法目前只在MPP计算引擎模式下支持。

语法描述

INSERT INTO table_name [(column [, ...])] query

其中 row 是一个单独的表达式或者:

(column_expression [, ...])

说明

定义一个常数内联表

VALUES 可以在任何允许使用查询语句的地方使用（例如，SELECT， INSERT语句中的 FROM 子句，甚至在语句的第一层级）。VALUES 创建了一个没有列名的匿名表，但是该表的表名和列名可以通过使用带有列别名的 AS 子句定义。

举例

返回一个表，包括一列三行数据:

VALUES 1, 2, 3

返回一个表，包含两列三行数据:

```
VALUES
(1, 'a'),
(2, 'b'),
(3, 'c')
```

返回一个表，包含 id 和 name 两个列:

```
SELECT * FROM (
VALUES
(1, 'a'),
(2, 'b'),
(3, 'c')
) AS t (id, name)
```

第五章 Data Pipeline

分析型数据库支持多种数据入库方式，包括但不限于：

（1）内置将MaxCompute中的海量数据快速批量导入；

（2）支持标准的insert/delete语法，可使用用户程序、Kettle等第三方工具写入实时写入表；

（3）支持阿里云数据集成(CDP)，将各类数据源导入批量导入表或实时写入表。阿里云大数据开发平台；

（4）支持阿里云数据传输(DTS)，从阿里云RDS实时同步数据变更到分析型数据库。

本章节主要描述（1）和（3）两种数据导入方式，其余导入方式详见本手册第八章。

SQL方式将MaxCompute数据导入

分析型数据库目前内置支持从阿里云外售的MaxCompute（原ODPS）中导入数据。

将MaxCompute数据导入分析型数据库有几种方法：直接使用SQL命令进行导入，或通过MDS for AnalyticDB界面进行导入、通过数据集成（原CDP）配置job进行导入。

通过DMS for AnalyticDB界面导入数据已在《快速入门》进行了介绍；通过数据集成导入数据可以参考下一小章节。下面主要详细介绍通过SQL命令将MaxCompute数据导入AnalyticDB。

步骤一： 准备MaxCompute表或分区。

首先我们需要创建好数据源头的MaxCompute表（可以是分区表或非分区表，目前要求AnalyticDB中的字段名称和MaxCompute表中的字段名称一致），并在表中准备好要导入的数据。

如在MaxCompute中创建一个表：

```
use projecta；--在MaxCompute的某个project中创建表
CREATE TABLE
odps2ads_test (
user_id bigint ,
amt bigint ,
num bigint ,
```

```
cat_id bigint ,
thedate bigint
)
PARTITIONED BY(dt STRING);
```

往表中导入数据并创建分区，如：dt=’ 20160808’ 。

步骤二：账号授权。

首次导入一个新的MaxCompute表时，需要在MaxCompute中将表Describe和Select权限授权给AnalyticDB的导入账号。公共云导入账号为garuda_build@aliyun.com以及garuda_data@aliyun.com（两个都需要授权）。各个专有云的导入账号名参照专有云的相关配置文档，一般为test1000000009@aliyun.com。

授权命令：

```
USE projecta；--表所属ODPS project
ADD USER ALIYUN$xxxx@aliyun.com;--输入正确的云账号
GRANT Describe,Select ON TABLE table_name TO USER ALIYUN$xxxx@aliyun.com;--输入需要赋权的表和正确的云账号
```

另外为了保护用户的数据安全，AnalyticDB目前仅允许导入操作者为Proejct Owner的ODPS Project的数据，或者操作者为MaxCompute表的owner（大部分专有云无此限制）。

步骤三：准备AnalyticDB表，注意创建时更新方式属性为“批量更新”，同时把表的Load Data权限授权给导入操作者（一把表创建者都默认有该权限）。

如表：

```
CREATE TABLE db_name.odps2ads_test（
user_id bigint,
amt int,
num int,
cat_id int,
thedate int,
primary key (user_id)
)
PARTITION BY HASH KEY(user_id)
PARTITION NUM 40
TABLEGROUP group_name
options (updateType='batch');
--注意指定好数据库名和表组名
```

步骤四:在AnalyticDB中通过SQL命令导入数据。

语法格式：

```
LOAD DATA
FROM 'sourcepath'
[OVERWRITE] INTO TABLE tablename [PARTITION (partition_name,...)]
```

说明：

如果使用MaxCompute（原ODPS）数据源，则sourcepath为：

```
odps://<project>/<table>/[<partition-1=v1>/.../<partition-n=vn>]
```

- ODPS项目名称 project 。
- ODPS表名 table 。
- ODPS分区 partition-n=vn ，可以是多级任意类型分区。
- 如果不指定则取当前时间（秒），格式是 yyyyMMddHHmss ，如 20140812080000。

- 覆盖导入选项（ OVERWRITE ），导入时如果指定数据日期的表在分析型数据库线上已存在，则返回异常，除非显示指定覆盖。
- 分析型数据库表名（ tablename ），格式 table_schema.table_name ，其中 table_name 是分析型数据库表名， table_schema 是表所属DB名，表名不区分大小写。

PARTITION，分析型数据库表分区，分区格式 partition_column=partition_value ，其中 partition_column 是分区列名， partition_value 是分区值

- 分区值必须是 long 型；
- 分区值不存在时表示动态分区，例如第一级hash分区；
- 不区分大小写；
- 目前最多支持二级分区。

执行返回值： <jobId>（任务ID），**用于后续查询导入状态**.任务ID是唯一标识该导入任务，返回的是字符串，最长256字节。

示例1：

```
LOAD DATA
FROM 'odps://<project>/odps2ads_test/dt=20160808'
INTO TABLE db_name.odps2ads_test
--注意<project> 填写MaxCompute表所属的项目名称
```

示例2：

如果分析型数据库表有二级分区，也可指定导入到相应二级分区：

```
LOAD DATA
FROM 'odps://<project>/odps2ads_test/dt=20160808'
INTO TABLE db_name.odps2ads_test PARTITION(user_id, dt=20160808)
```

步骤五：查看导入状态(参见5.2章节)和成功导入的数据。

通过数据集成将RDS等其他数据源的数据导入

当表的数据源是RDS、OSS等其它的云系统，我们可以通过阿里云的数据集成产品进行数据同步。

批量更新表导入

专有云上批量导入

专有云上可以通过**大数据开发套件**的数据集成任务里的数据同步进行操作，数据同步任务即通过封装数据集成实现数据导入。具体步骤请看“大数据开发套件”的用户指南相关数据源配置和数据同步任务配置章节。

前提条件：

- 分析型数据库目标表更新方式是“批量更新”。
- 在分析型数据库中给MaxCompute的base_meta这个project的owner账号至少授予表的Load Data权限，base_meta的owner账号信息可以在CMDB中查到。

> 注意“大数据开发套件”中数据集成同步任务目标为ads数据源的“导入模式”配置项需要选择“批量导入”。

公共云上批量导入

公共云上可在 http://www.aliyun.com/product/cdp/ 上开通数据集成（可能需要申请公测），

前提条件：

- 分析型数据库目标表更新方式是“批量更新”。
- 在分析型数据库中给cloud-data-pipeline@aliyun-inner.com这个账号至少授予表的Load Data权限。

配置下面的数据同步任务，准备工作要添加相应的数据源，添加各种数据源详细信息可以参考下面的文档：数据源配置

向导模式配置同任务

下面以mysql同步到ads为例：

1. 新建同步任务，如下图所示：

2. 选择来源： 选择mysql数据源及源头表emp，数据浏览默认是收起的，选择后点击下一步，如下图

所示：

3. 选择目标：选择ads数据源及目标表emp，选择后点击下一步，如下图所示：

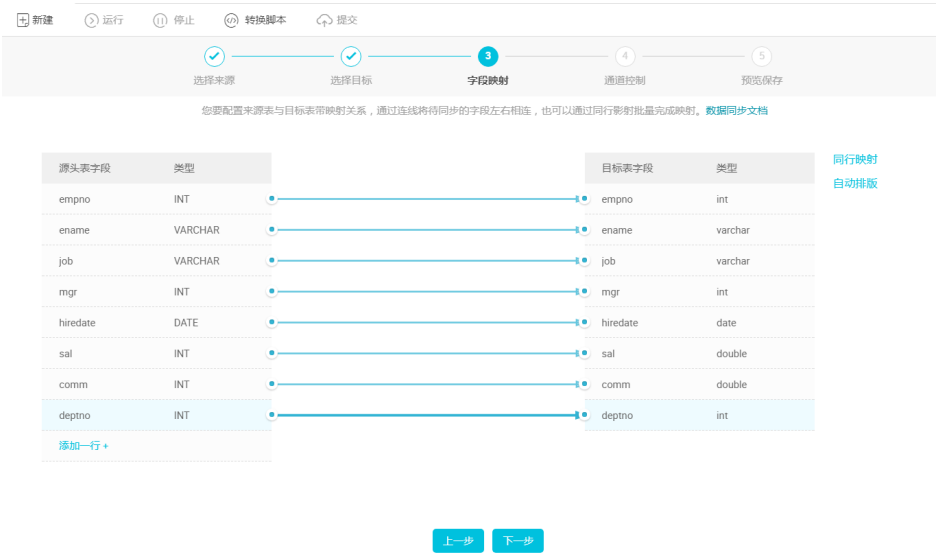
- **导入模式**：即上述参数说明中的“writeMode”，支持Load Data（批量导入）和Insert Ignore（实时插入）两种模式。

清理规则：

1）写入前清理已有数据：导数据之前，清空表或者分区的所有数据，相当于 insert overwrite 。

2）写入前保留已有数据：导数据之前不清理任何数据，每次运行数据都是追加进去的，相当于 insert into 。

4.映射字段：点击下一步，选择字段的映射关系。需对字段映射关系进行配置，左侧“源头表字段”和右侧“目标表字段”为一一对应的关系 ，如下图所示。



1. 通道控制点击下一步，配置作业速率上限和脏数据检查规则，如下图所示：

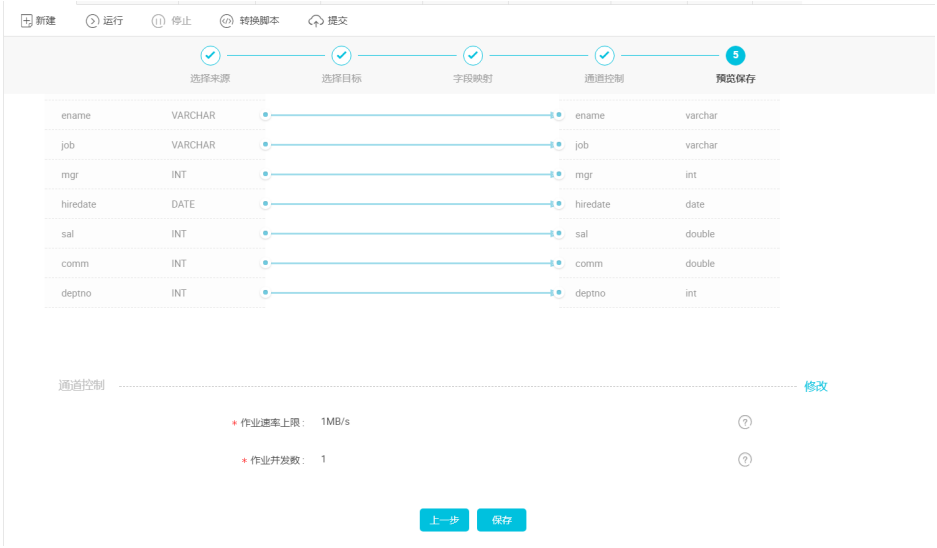
作业速率上限：是指数据同步作业可能达到的最高速率，其最终实际速率受网络环境、数据库配置等的影响。

作业并发数：

从单同步作业来看：作业并发数*单并发的传输速率=作业传输总速率；

当作业速率上限已选定的情况下，应该如何选择作业并发数？
① 如果你的数据源是线上的业务库，建议您不要将并发数设置过大，以防对线上库造成影响；
② 如果您对数据同步速率特别在意，建议您选择最大作业速率上限和较大的作业并发数

1. 预览保存：完成以上配置后，上下滚动鼠标可查看任务配置，如若无误，点击保存，如下图所示：



脚本模式配置同步任务

mysql同步到ads任务配置示例：

```
{
"type": "job",
"version": "1.0",
"configuration": {
"setting": {
"errorLimit": {
"record": "0"
},
"speed": {
"mbps": "1",//一个并发的速率上线是1MB/S
"concurrent": "1"//并发的数目
}
}
},
"reader": {
"plugin": "mysql",
"parameter": {
"datasource": "xxx",//数据源名，建议数据源都添加数据源后进行同步任务
"table": "k",//mysql源端表名
"splitPk": "id",//切分键：源数据表中某一列作为切分键，切分之后可进行并发数据同步，目前仅支持整型字段；建议使用主键或有索引的列作为切分键
"column": [
"int"
],//列名
"where": "id>100"请参考相应的SQL语法填写where过滤语句（不需要填写where关键字）该过滤 语句通常作为增量同步
}
},
"writer": {
"plugin": "ads",
"parameter": {
"datasource": "yyy",//数据源名，建议数据源都添加数据源后进行同步任务
"table": "dd1",//ads目标表名
"partition": "id",//分区
```

```
"overWrite": "true",//写入规则
"batchSize": "256",//同步的批量大小
"writeMode": "load",//写入模式load
"column": [
"*"
]
}
}
}
}
```

其他的数据源跟ads之间的同步，可以参考脚本模式各数据源的reader配置里的文档。

其他数据导入方式

用户亦可利用阿里云数据传输（DTS)进行RDS到分析型数据库的实时数据同步（请参照手册第八章的相关内容）。

分析型数据库亦兼容通过kettle等第三方工具，或用户自行编写的程序将数据导入/写入实时写入表。

另外，分析型数据库进行实时插入和删除时，不支持事务，并且仅遵循最终一致性的设计，所以分析型数据库并不能作为OLTP系统使用。

5.2 数据导入状态查询

数据导入命令发送后，数据并不会立刻导入到分析型数据库中，而是会在后台进行数据的导入工作。用户可以通过使用SQL命令或在DMS中进行查询数据导入状态。高级用户也可以直接在information_schema中查询全部数据导入的信息（具体见附录）。

查询数据的导入状态有多种方法：

（1）通过SQL语句查询：

通过SQL命令查询适合提交导入没有完成或完成不久的情况。

SQL语法：

```
select state from information_schema.current_job where job_id = '<jobid>'
```

返回值：

```
'<jobState>'
```

返回值表示该任务状态（ jobState ）：

- NEW 初始状态
- INITED 排队中

- RUNNING 正在导入
- SUCCEEDED 导入成功
- FAILED 导入失败
- ERROR 导入出错（系统内部错误）
- 任务不存在，则返回空
- 已完成的任务会被会被定时清理转为历史任务

（2）通过DMS查询数据导入状态:

在DMS中，点击菜单上的导入导出->导入状态，即可查询每天的导入任务情况，并且通过多个维度进行筛选浏览。

提高导入效率

数据导入运行时会运行Map Reduce任务并行的构建索引和数据，为避免长尾任务，需要确保建表时分区列选择正确，没有明显分区数据倾斜。

另外导入运行时的并行任务数和资源类型及实例个数有关，如果每日需要运行的导入任务较多，可选择配置较高的资源类型及较多的实例个数（具体参见“数据导入相关限制”）。 若提交的导入任务数超过当前资源类型及实例个数允许的并发度，该任务会进入排队状态。

提高数据导入后的磁盘利用率

导入后的数据会按分区个数均匀地存储在实例节点中，为确保磁盘使用最大化，建表时表的一级分区数应和实例个数成比例关系(0.5的整数倍)，且一级分区数大于实例个数。如实例个数为8，表的分区数可为8（最少允许分区数），12，16，20，...256（最多允许分区数）. 考虑到以后潜在的扩容，可适当设置较大的分区数。

取消运行中的导入任务

当某个导入任务正在运行但由于各种原因需要取消时，可执行以下命令取消导入。（<DB ID>可参见“附录一元数据库数据字典”查询SCHEMATA表获得，<Job ID>在执行导入任务时已返回）

```
ALTER DATABASE <DB Name> properties(databaseId='<DB ID>') KILL LOADDATA <Job ID>
```

重试失败的导入任务

当导入任务失败时（失败场景包括提交导入任务失败和任务执行失败，提交导入任务失败即执行LOAD DATA命令错误，任务执行失败可通过“5.2 数据导入状态查询”查看），可再次发起LOAD DATA请求（程序触发的话建议加上重试间隔），如果多次重试依然失败，联系技术支持解决。

注：运行中的导入任务无法再次发送LOAD DATA请求。

数据导入相关限制

并发导入的任务数和每日导入数据量都会根据资源类型和实例数量有相关quota限制，具体参见产品和业务限制。

分析型数据库的查询模式适合在海量数据中进行分析计算后输出适量数据，若需要输出的数据量达到一定规模，分析型数据库提供数据导出（DUMP）的方式。注意目前DUMP方式中不能使用针对非分区列的聚合函数。

通过类DML语句导出到MaxCompute

前提须知

> 分析型数据库通过一个固定云账号进行数据导出到MaxCompute（与数据从MaxCompute导入到AnalyticDB情况类似）。各个专有云的导出账号名参照专有云的相关配置文档，一般为test1000000009@aliyun.com（与导入账号一致），公共云导出账号为garuda_data@aliyun.com。
>
> 需要给导入账号授予目标MaxCompute项目的createInstance权限，以及目标表的describe、select、alter、update、drop权限。

授权命令：

```
--注意正确输入需要授权的表命名、project和正确的云账号

USE prj_name；--表所属ODPS project
ADD USER ALIYUN$xxxx@aliyun.com;
GRANT createInstance ON project prj_name TO USER ALIYUN$xxxx@aliyun.com;
GRANT Describe,Select,alter,update,drop ON TABLE table_name TO USER ALIYUN$xxxx@aliyun.com;
```

导出命令

类似于普通的SQL查询语句，用户也可通过类似于DML语句进行数据导出。

语法格式：

```
DUMP DATA
[OVERWRITE] INTO 'odps://project_name/table_name'
SELECT C1, C2 FROM DB1.TABLE1 WHERE C1 = 'xxxx' LIMIT N
```

通过类DML语句导出到OSS（当前为公测功能，非商业化使用）

导出到OSS时，需要持有对该oss bucket有写权限的AK（为安全起见，必须使用子账号的AK）。

语法格式：

```
/*+ dump-col-del=[,],dump-row-del=[\n],dump-oss-accesskey-id=oss的ACCESS_KEY_ID,
dump-oss-accesskey-secret=oss的ACCESS_KEY_SECRET*/ DUMP DATA
[OVERWRITE] INTO 'oss://endpoint_domain/bulket_name/filename'
SELECT C1, C2 FROM DB1.TABLE1 WHERE C1 = 'xxxx' LIMIT N
```

说明：

- endpoint_domain是与ads同一个region的oss的内网endpoint，跨region访问时需要填写oss的公网endpoint（部分region之间可能无法跨region访问oss）。
- 部分情况下，目前可能会dump oss失败（dump下的sql中有分区倾斜时）
- dump-col-del=[,],dump-row-del=[\n] 用于规定导出文件的行列分隔符，可不填。

关于返回数据行数

导出方式对海量数据的计算输出具有良好的性能（百万行数据导出在数百毫秒数据级），但是，对于数据精确度有一定牺牲，即实际返回的数据行数，可能是不完全精确。以限制导出行数为1000为例（LIMIT 1000）：

- 实际数据行数**可能稍大于1000**，例如此时有120个数据分片，则等同于每个分区明确指定”LIMIT 9”，最多肯能返回1080
- 实际数据行数**可能稍小于1000**，如果符合条件的行数的总数小于1000
- 实际数据行数**可能稍小于1000**，如果数据分片很均匀，例如此时有120个数据分片，如果某些分片返回数据行小于9的话，则等同于每个分区明确指定”LIMIT 9”

第六章 权限与安全

分析型数据库用户基于阿里云帐号进行认证，用户建立的数据库属于该用户，用户也可以授权给其他用户访问其数据库下的表。

用户帐号与认证

阿里云帐号

- 帐号格式： ALIYUN$user_account@aliyun.com，其中ALIYUN$为账号前缀，标识该账号类型为阿里云账号，未来分析型数据库会推出更多的账号类型支持。
- 通过阿里云官方网站注册

认证

- Access Key： 通过阿里云官网的阿里云控制台生成和管理，使用分析型数据库的用户必须在 https://i.aliyun.com/access_key/ 拥有至少一个有效的access key。
- 访问分析型数据库服务时使用 Access Key。

用户类型

- 数据库拥有者：必须开通了分析型数据库服务，同时是数据库的创建者
- 用户：被授权的数据库用户，由数据库拥有者授权时添加，无须开通分析型数据库服务

添加用户

SQL语法

```
ADD USER user ON db_name.*
```

返回值

```
空
```

添加用户

- 创建数据库时，数据库的创建者即做为拥有者添加
- 用户在第一次被授权一个数据库中的任何权限时，分析型数据库会自动添加为该数据库的用户
- 主动添加到数据库的用户初始时没有任何权限

查看用户列表

SQL语法

```
LIST USERS ON db_name.*
```

返回值

```
\<user_id\> \<user_id\> ...
```

查看用户列表

- 只有数据库的拥有者可以查看该数据库的所有用户

分析型数据库支持基于数据库表的层级权限管理模型，提供类似MySQL的ACL授权模式。一个ACL授权由被授权的用户、授权对象和授予的对象权限组成。

分析型数据库中的用户

如6.1所述，任何分析型数据库支持的账号类型均可视为一个用户。和MySQL略有不同的是，分析型数据库目前不支持针对用户在host上授权。

分析型数据库的权限对象和各对象权限

- Database，即 db_name.* 或 *（默认数据库），指定数据库或数据库上所有表/表组
- Table，即 db_name.table_name 或 table_name ，指特定表
- TableGroup，即 db_name.table_group_name 或 table_group_name ，指特定表组
- Column，语法上由 column_list 和 Table 组成，指定表的特定列
- 聚合： Database -> Table[Group] -> Column，即每个权限级别能聚合其下面级别的所有权限
- 在 Database 级别上，某个权限实际可能包含多个权限，例如 GRANT CREATE ON *.* 同时包含创建数据库和创建表的权限

-	Database	Table 或 TableGroup	Column	说明
SELECT	+	+	+	查询数据
LOAD DATA	+	+	-	导入表（分区）数据
DUMP DATA	+	+	-	导出表（分区）数据
DESCRIBE	+	+	-	查看数据库、表/表组信息（Global、Database）、查看表/表组信息（Table[Group]）
SHOW	+	+	-	列出数据库、表/表组内部对象（Global、Database）、列出表内部对象（Table[Group]）
ALTER	+	+	-	修改表/表组定义
DROP	+	+	-	删除数据库、表/表组或分区（Global、Database）、删除表/表组或分区（Table[Group]）
CREATE	+	-	-	创建数据库、表/表组或分区（Global）、创建表/表组

				（Database）
INSERT	+	+	-	执行Insert的权限
DELETE	+	+	-	执行Delete的权限
ALL [PRIVILEGES]	+	+	+	以上所有权限

查询数据的权限

- 查询表数据需要 SELECT 权限，最小级别是列
- 并非所有查询都需要该权限，例如 SELECT now()

导出数据的权限

- 导出数据同时需要 DUMP DATA 和 SELECT 权限
- 同时需要数据导出目的地的数据写入相关权限

前文提到，类似MySQL，分析型数据库中的数据库创建者可以使用GRANT/REVOKE语句进行授权和权限回收。

进行授权

SQL语法

```
GRANT privilege_type [(column_list)] [, privilege_type [(column_list)]] ... ON [object_type] privilege_level TO user [, user] ...
```

返回值

空

被授权用户

```
user: 'user_name'[@'host_name']
```

- user_name 和 host_name 必须使用单引号或双引号，如 'ALIYUN$test-user@aliyun.com'@'%'
- 目前，host_name 只支持 % ，即不支持指定Host
- 可以写成 user_name ，等价于 'user_name'@'%'

权限

privilege_type 为具体的权限类型，每一级对象拥有的权限类型参见6.2节

[(column_list)] 为可选，当对象级别为表时，这里可以填写列的列表，进行针对具体列的授权

[object_type] 为可选，标明权限对象的类型，如Database、Table、TableGroup等，建议填写

privilege_level 为被授权对象的表达式，填写方法见6.2节

例子

```
GRANT describe, select ON tablegroup db_name.table_group_name TO user 'ALIYUN$test_user@aliyun.com'@'%';
GRANT describe, select (col1, col2) ON table db_name.table__name TO user 'ALIYUN$test_user@aliyun.com';
```

权限回收

SQL语法

REVOKE privilege_type [(column_list)] [, privilege_type [(column_list)]] ... ON [object_type] privilege_level FROM user [, user] ...

返回值

空

- 在语法上与授权语句基本一致

查看用户权限

SQL语法

SHOW GRANTS [FOR user] ON [object_type] privilege_level

返回值

'GRANT ALL ON db_name.* TO user' 'GRANT SELECT(column_name) ON db_name.table_name TO user'

查看用户权限

- 不指定用户或是指定用户即当前用户，则列出当前用户在指定数据库上被授予的权限

- 否则，列出当前用户在指定数据库上授予指定用户的权限

其它相关内容

权限的默认约定

- 数据库创建者拥有在该数据库上的所有权限
- 表（组）创建者拥有在该表（组）上的所有权限
- 其他数据库用户拥有被各数据库拥有者授予的权限
- SHOW DATABASES 语句不进行权限检查，可在不指定数据库的前提下之星，会返回用户所有拥有权限的数据库列表
- 创建数据库时不检查权限，但用户必须通过阿里云官网开通了分析型数据库服务并处于服务正常运行的状态

授权/回收权限的权限

- 数据库拥有者（暨创建者）在该数据库上执行 Database 及其级别以下的授权/回收权限，如 SELECT ON db_name.*
- 其他数据库用户目前无法执行授权/回收权限操作

设置ip白名单

目前分析型数据库支持设置DB粒度的ip访问白名单，该功能目前处于内测，有需要的客户可以提交工单索取设置方法和注意事项。

AnalyticDB VPC功能介绍

AnalyticDB从2.3版本开始支持VPC功能，默认使用single tunnel的方式，也可以通过配置切换到any tunnel。VPC的概念及相关架构可以参考文档链接(https://help.aliyun.com/document_detail/34217.html)。目前由于售卖端还没有针对VPC特性进行改造，所以公共云用户如果想要使用VPC功能，可以提交工单单独开启。目前专有云V3中已经全面开启VPC，DTCenter也针对这一特性进行了改造及支持创建VPC库。需要注意的是目前暂不支持反向访问。

AnalyticDB VPC实现原理

- single tunnel模式:

该模式是目前默认的VPC方式，通过从用户VPC网段内获取一个IP（如192.158.1.2）作为当前库的访问VIP，在用户VPC网段内访问AnalyticDB。实现的原理是先调vpc-nner-api创建云服务实例，该云服务实例数据中包含一个用户VPC网段内的一个IP。再通过云服务实例ID及相关参数调vpc-inner-api，查询该云服务实例的tunnel id。用返回的网段内的IP及tunnel id作为参数，调创建SLB实例接口创建VPC SLB实例。后续步骤和经典网络一样创建RS_POOL及添加REAL_SERVER到RS_POOL内，完成整个创建VPC实例的过程。需要说明的是这个过程中会同时创建一个经典网络和VPC的域名及端口，用户可以根据实际需求选择连接串。该模式可以实现不同VPC内实现网络隔离的目标，同时支持无缝切换VPC网络的命令。

- any tunnel模式:

该模式可以通过配置来切换，修改配置后下一次创建库时生效或者通过修改元数据及重启frontnode生效。实现的过程和上述类似，不同的是创建云服务实例的参数不同以及返回的VIP在所有VPC内网络都通畅，理论上满足不了不同VPC之间网络隔离的需求。

VPC相关操作

创建共享经典网络VIP方式的single tunnel类型DB:

```
确认/global/config/master/useSingleTunnel=true（默认）；
确认global/config/master/withVPCEnv=true；
确认global/config/master/sharedVip=true（默认）；
连接sysdb执行 create database vpc_test options(resource_type='ecu' ecu_type='c1' ecu_count=2 zone_id='xxxx'
vpc_id='xxxx' vswitch_id='xxxx')
```

创建共享经典网络VIP及共享any tunnel方式的VPC类型DB:

```
查询元数据slb_instance表，确认有没有any_tunnel类型的共享SLB实例，select * from slb_instance where
source_type='any_tunnel' and shared=1;
如果没有any_tunnel类型的共享SLB实例，登录sysdb创建any_tunnel类型的SLB实例，在sysdb上执行alb
createloadbalancer internet/intranet any_tunnel（internet/intranet二选一）；
确认/global/config/master/useSingleTunnel=false；
确认global/config/master/withVPCEnv=true；
确认global/config/master/sharedVip=true（默认）；
连接sysdb执行 create database vpc_test options(resource_type='ecu' ecu_type='c1' ecu_count=2 zone_id='xxxx'
vpc_id='xxxx' vswitch_id='xxxx')
```

创建VIP独占方式的single tunnel类型DB:

```
确认/global/config/master/useSingleTunnel=true（默认）；
确认global/config/master/withVPCEnv=true；
确认global/config/master/sharedVip=false；
连接sysdb执行 create database vpc_test options(resource_type='ecu' ecu_type='c1' ecu_count=2 zone_id='xxxx'
vpc_id='xxxx' vswitch_id='xxxx')
```

创建VIP独占方式的any tunnel类型DB:

```
确认/global/config/master/useSingleTunnel=true（默认）；
确认global/config/master/withVPCEnv=true；
确认global/config/master/sharedVip=false；
连接sysdb执行 create database vpc_test options(resource_type='ecu' ecu_type='c1' ecu_count=2 zone_id='xxxx'
vpc_id='xxxx' vswitch_id='xxxx')
```

创建读写分离且共享经典网络VIP方式的single tunnel类型DB:

```
确认/global/config/master/useSingleTunnel=true（默认）；
确认global/config/master/withVPCEnv=true；
确认global/config/master/sharedVip=true（默认）；
连接sysdb执行 create database vpc_test options(resource_type='ecu' ecu_type='c1' ecu_count=2 zone_id='xxxx'
```

vpc_id='xxxx' vswitch_id='xxxx' worker_labels=read:write worker_ratios=2:1)，（具体读写比例根据实际需求填写）

创建读写分离且共享经典网络VIP及共享any tunnel方式的VPC类型DB:

查询元数据slb_instance表，确认有没有any_tunnel类型的共享SLB实例，select * from slb_instance where source_type='any_tunnel' and shared=1;
如果没有any_tunnel类型的共享SLB实例，登录sysdb创建any_tunnel类型的SLB实例，在sysdb上执行alb createloadbalancer internet/intranet any_tunnel（internet/intranet二选一）；
确认/global/config/master/useSingleTunnel=false；
确认global/config/master/withVPCEnv=true；
确认global/config/master/sharedVip=true（默认）；
连接sysdb执行 create database vpc_test options(resource_type='ecu' ecu_type='c1' ecu_count=2 zone_id='xxxx' vpc_id='xxxx' vswitch_id='xxxx' worker_labels=read:write worker_ratios=2:1)

创建读写分离且VIP独占方式的single tunnel类型DB:

确认/global/config/master/useSingleTunnel=true（默认）；
确认global/config/master/withVPCEnv=true；
确认global/config/master/sharedVip=false；
连接sysdb执行 create database vpc_test options(resource_type='ecu' ecu_type='c1' ecu_count=2 zone_id='xxxx' vpc_id='xxxx' vswitch_id='xxxx' worker_labels=read:write worker_ratios=2:1)

创建读写分离且VIP独占方式的any tunnel类型DB:

确认/global/config/master/useSingleTunnel=true（默认）；
确认global/config/master/withVPCEnv=true；
确认global/config/master/sharedVip=false；
连接sysdb执行 create database vpc_test options(resource_type='ecu' ecu_type='c1' ecu_count=2 zone_id='xxxx' vpc_id='xxxx' vswitch_id='xxxx' worker_labels=read:write worker_ratios=2:1)

已存在DB开启共享经典网络VIP方式的single tunnel类型:

确认/global/config/master/useSingleTunnel=true（默认）；
确认global/config/master/withVPCEnv=true；
确认global/config/master/sharedVip=true（默认）；
设置schemata表对应DB的记录domain_url、vip、port、vpc_domain_url、vpc_vip、vpc_port值为NULL，update schemata set domain_url=NULL, vip=NULL, port=NULL, vpc_domain_url=NULL, vpc_vip=NULL, vpc_port=NULL where cluster_name='xxxx' and schema_name='xxxx';
重启当前DB所有frontnode；
查询schemata表对应DB的记录domain_url、vip、port、vpc_domain_url、vpc_vip、vpc_port值是否生成。

已存在DB开启共享经典网络VIP及共享any tunnel方式的VPC类型:

查询元数据slb_instance表，确认有没有any_tunnel类型的共享SLB实例，select * from slb_instance where source_type='any_tunnel' and shared=1;
如果没有any_tunnel类型的共享SLB实例，登录sysdb创建any_tunnel类型的SLB实例，在sysdb上执行alb createloadbalancer internet/intranet any_tunnel（internet/intranet二选一）；
确认/global/config/master/useSingleTunnel=false；
确认global/config/master/withVPCEnv=true；
确认global/config/master/sharedVip=true（默认）；

设置schemata表对应DB的记录domain_url、vip、port、vpc_domain_url、vpc_vip、vpc_port值为NULL，update schemata set domain_url=NULL, vip=NULL, port=NULL, vpc_domain_url=NULL, vpc_vip=NULL, vpc_port=NULL where cluster_name='xxxx' and schema_name='xxxx';
重启当前DB所有frontnode；
查询schemata表对应DB的记录domain_url、vip、port、vpc_domain_url、vpc_vip、vpc_port值是否生成。

已存在DB开启VIP独占方式的single tunnel类型:

确认/global/config/master/useSingleTunnel=true（默认）；
确认global/config/master/withVPCEnv=true；
确认global/config/master/sharedVip=false；
设置schemata表对应DB的记录domain_url、vip、port、vpc_domain_url、vpc_vip、vpc_port值为NULL，update schemata set domain_url=NULL, vip=NULL, port=NULL, vpc_domain_url=NULL, vpc_vip=NULL, vpc_port=NULL where cluster_name='xxxx' and schema_name='xxxx';
重启当前DB所有frontnode；
查询schemata表对应DB的记录domain_url、vip、port、vpc_domain_url、vpc_vip、vpc_port值是否生成。

已存在DB开启VIP独占方式的any tunnel类型:

确认/global/config/master/useSingleTunnel=true（默认）；
确认global/config/master/withVPCEnv=true；
确认global/config/master/sharedVip=false；
设置schemata表对应DB的记录domain_url、vip、port、vpc_domain_url、vpc_vip、vpc_port值为NULL，update schemata set domain_url=NULL, vip=NULL, port=NULL, vpc_domain_url=NULL, vpc_vip=NULL, vpc_port=NULL where cluster_name='xxxx' and schema_name='xxxx';
重启当前DB所有frontnode；
查询schemata表对应DB的记录domain_url、vip、port、vpc_domain_url、vpc_vip、vpc_port值是否生成。

已存在DB开启读写分离且共享经典网络VIP方式的single tunnel类型:

确认/global/config/master/useSingleTunnel=true（默认）；
确认global/config/master/withVPCEnv=true；
确认global/config/master/sharedVip=true（默认）；
设置schemata表对应DB的记录domain_url、vip、port、vpc_domain_url、vpc_vip、vpc_port、vpc_read_domain_url、vpc_read_vip、vpc_read_port、vpc_write_domain_url、vpc_write_vip、vpc_write_port值为NULL，update schemata set domain_url=NULL, vip=NULL, port=NULL, vpc_domain_url=NULL, vpc_vip=NULL, vpc_port=NULL, vpc_read_domain_url=NULL, vpc_read_vip=NULL, vpc_read_port=NULL, vpc_write_domain_url=NULL, vpc_write_vip=NULL, vpc_write_port=NULL where cluster_name='xxxx' and schema_name='xxxx';
重启当前DB所有frontnode；
查询schemata表对应DB的记录domain_url、vip、port、vpc_domain_url、vpc_vip、vpc_port、vpc_read_domain_url、vpc_read_vip、vpc_read_port、vpc_write_domain_url、vpc_write_vip、vpc_write_port值是否生成。

已存在DB开启读写分离且共享经典网络VIP及共享any tunnel方式的VPC类型:

查询元数据slb_instance表，确认有没有any_tunnel类型的共享SLB实例，select * from slb_instance where source_type='any_tunnel' and shared=1;
如果没有any_tunnel类型的共享SLB实例，登录sysdb创建any_tunnel类型的SLB实例，在sysdb上执行alb createloadbalancer internet/intranet any_tunnel（internet/intranet二选一）；
确认/global/config/master/useSingleTunnel=false；
确认global/config/master/withVPCEnv=true；
确认global/config/master/sharedVip=true（默认）；

设置schemata表对应DB的记录domain_url、vip、port、vpc_domain_url、vpc_vip、vpc_port、vpc_read_domain_url、vpc_read_vip、vpc_read_port、vpc_write_domain_url、vpc_write_vip、vpc_write_port值为NULL，update schemata set domain_url=NULL, vip=NULL, port=NULL, vpc_domain_url=NULL, vpc_vip=NULL, vpc_port=NULL, vpc_read_domain_url=NULL, vpc_read_vip=NULL, vpc_read_port=NULL, vpc_write_domain_url=NULL, vpc_write_vip=NULL, vpc_write_port=NULL where cluster_name='xxxx' and schema_name='xxxx';
重启当前DB所有frontnode；
查询schemata表对应DB的记录domain_url、vip、port、vpc_domain_url、vpc_vip、vpc_port、vpc_read_domain_url、vpc_read_vip、vpc_read_port、vpc_write_domain_url、vpc_write_vip、vpc_write_port值是否生成。

已存在DB开启读写分离且VIP独占方式的single tunnel类型：

确认/global/config/master/useSingleTunnel=true（默认）；
确认global/config/master/withVPCEnv=true；
确认global/config/master/sharedVip=false；
设置schemata表对应DB的记录domain_url、vip、port、vpc_domain_url、vpc_vip、vpc_port、vpc_read_domain_url、vpc_read_vip、vpc_read_port、vpc_write_domain_url、vpc_write_vip、vpc_write_port值为NULL，update schemata set domain_url=NULL, vip=NULL, port=NULL, vpc_domain_url=NULL, vpc_vip=NULL, vpc_port=NULL, vpc_read_domain_url=NULL, vpc_read_vip=NULL, vpc_read_port=NULL, vpc_write_domain_url=NULL, vpc_write_vip=NULL, vpc_write_port=NULL where cluster_name='xxxx' and schema_name='xxxx';
重启当前DB所有frontnode；
查询schemata表对应DB的记录domain_url、vip、port、vpc_domain_url、vpc_vip、vpc_port、vpc_read_domain_url、vpc_read_vip、vpc_read_port、vpc_write_domain_url、vpc_write_vip、vpc_write_port值是否生成。

已存在DB开启读写分离且VIP独占方式的any tunnel类型:

确认/global/config/master/useSingleTunnel=true（默认）；
确认global/config/master/withVPCEnv=true；
确认global/config/master/sharedVip=false；
设置schemata表对应DB的记录domain_url、vip、port、vpc_domain_url、vpc_vip、vpc_port、vpc_read_domain_url、vpc_read_vip、vpc_read_port、vpc_write_domain_url、vpc_write_vip、vpc_write_port值为NULL，update schemata set domain_url=NULL, vip=NULL, port=NULL, vpc_domain_url=NULL, vpc_vip=NULL, vpc_port=NULL, vpc_read_domain_url=NULL, vpc_read_vip=NULL, vpc_read_port=NULL, vpc_write_domain_url=NULL, vpc_write_vip=NULL, vpc_write_port=NULL where cluster_name='xxxx' and schema_name='xxxx';
重启当前DB所有frontnode；
查询schemata表对应DB的记录domain_url、vip、port、vpc_domain_url、vpc_vip、vpc_port、vpc_read_domain_url、vpc_read_vip、vpc_read_port、vpc_write_domain_url、vpc_write_vip、vpc_write_port值是否生成。

切换DB所在VPC网络，注意不能在同一个VPC网络内切换:

```
alter database vpc_test set zone_id='xxxx' vpc_id='xxxx' vswitch_id='xxxx';
```

VPC相关运维命令:

vpc createvpc，创建VPC网络，主要用来创建测试环境中的VPC网络，在测试时使用；
vpc createvswitch $vpc_id $zone_id $cidr_block，在测试VPC内创建虚拟交换机划分子网；
vpc queryvpc $vpc_id，查询VPC；
vpc queryvswitch $vpc_id $zone_id $v_switch_id，查询虚拟交换机；
vpc deleteinstance instance_id uid bid，删除VPC实例，这个命令主要在用户切换VPC网络异常或者删除DB时异常，人工手动删除云服务实例，实例的ID可以再slb_instance中查到，对应cloud_instance_id字段。

VPC相关参数说明:

zone_id，可用区ID。实际上代表机房，公共云上需要找VPC或者相关团队查询一下具体值，专有云在DTCenter中可以通过下拉菜单选择；
vpc_id，VPC的ID，专有云和公共云都可以再对应的控制台上看到；
vswitch_id，虚拟交换机ID，专有云和公共云都可以再对应的控制台上看到。

VPC相关配置

VPC的开启的前提条件是SLB配置要配置正确，SLB相关配置可以参考SLB配置文档，下面是VPC相关的配置:

anyTunnelResourceUid，资源any tunnel资源所属的用户uid，目前这个字段没有严格校验，默认1234567890。
vpcEndpointName
vpcEndpointName
vpcEndpointRegionId
vpcEndpointProduct
vpcEndpointDomain，VPC服务endpoint；
vpcLocationRegionId
vpcLocationProduct
vpcLocationEndpoint
vpcRegion，当前集群所在区域;
useSingleTunnel，是否使用single tunnel方式。默认为true，如果为false则为any_tunnel方式；
sharedVip，是否共享VIP。默认为true，如果为false则为独占VIP的方式。

上述参数具体可以参考(http://gitlab.alibaba-inc.com/vpc-yaochi-wiki/yaochi-document/blob/master/singleTunnel/%E4%BA%91%E4%BA%A7%E5%93%81%E6%8E%A5%E5%85%A5%E8%AF%B4%E6%98%8E.md)

在0.8.43或0.9.7以后的版本的阿里云分析型数据库中，支持通过阿里云访问控制(https://ram.console.aliyun.com/ ）创建的子账号登录分析型数据库，并管理子账号在不同条件下是否有使用分析型数据库的权限。

主账号在阿里云访问控制的控制台中，可以新建多个子账号，通过授予对应的授权策略，使子账号在一定条件下可以访问分析型数据库。子账号访问分析型数据库的MySQL协议端时需要使用其的Access Key ID/Secret作为用户名和密码。若在访问控制中允许子账号登录阿里云控制台，子账号亦可登录分析型数据库的控制台DMS。

在阿里云访问控制中，授权一个子账号可以访问分析型数据库，可以使用系统内置的授权策略 AliyunAnalyticDBFullAccess 授予子账号权限。在需要添加更多条件，或所在环境中找不到系统内置的授权策略的，可以按照如下示例的方式制定访问策略(限制访问来源IP必须是)：

```
{
"Version": "1",
"Statement": [
{
"Action": "ads:*",
"Resource": "*",
"Effect": "Allow"
```

```
}
],
"Condition":{
"IpAddress": {
"acs:SourceIp": ["42.120.66.0/24"]
}
}
}
```

此策略限制该用户只能在42.120.66.0/24网段访问分析型数据库。详细的策略编写方法详见 https://help.aliyun.com/document_detail/28663.html 。

注意一旦授予子账号相关访问策略，子账号将继承主账号在分析型数据库的全部权限（除ACL授权相关权限），包括主账号作为owner的数据库的权限，以及主账号被授权的其他数据库的相关被授予的权限。另外需要注意，目前暂不支持STS Token访问分析型数据库。

第七章 性能优化和诊断

7.1 使用执行计划

在进行一个业务SQL的性能调优，或是查看一个业务SQL的计算成本消耗时，使用explain命令来进行查看是一个很好的选择。iDB Cloud for 分析型数据库中提供图形化的执行计划命令，同时，用户也可以手动使用explain指令进行查看。

explain 语句

分析型数据库支持通过explain语句来查看逻辑计划和物理执行计划，当用户发起一个explain查询到分析型数据库系统后，分析型数据库会抽样一个数据分区来分析执行计划，并以图形方式展现给用户。

explain语句的格式为， explain + select语句, 例如:

简单sql的explain

```
explain select count(*) from test4dmp.test where id > 0
```

复杂sql的explain

```
 explain
select student.id, count(*)
from test4dmp.student
inner join test4dmp.grade on student.id =grade.sid
```

```
inner join test4dmp.elective on elective.id=student.id
group by student.id
having count(*) > 2
order by student.id
limit 10
```

返回文本格式

当用户通过查询的方式，想要获取文本格式的explain语句后，将会得到如下的json串：

- 返回的json格式

```
  | EXPLAIN       |
| logical json |
| physical json |
```

返回格式说明一共会返回2行1列并且列名为EXPLAIN的ResultSet记录。其中第一行为逻辑计划，第二行为物理计划。

JSON格式说明

- Node 代表着唯一的子节点
- LeftNode 代表左子树
- RightNode 代表右子树
- MiddleNode 代表所有的中间子树（多叉执行树），可以有多个MiddleNode
- 其余key-value对统一在同层级的Node和*Node节点显示

Explain 逻辑计划详细语义

样例sql

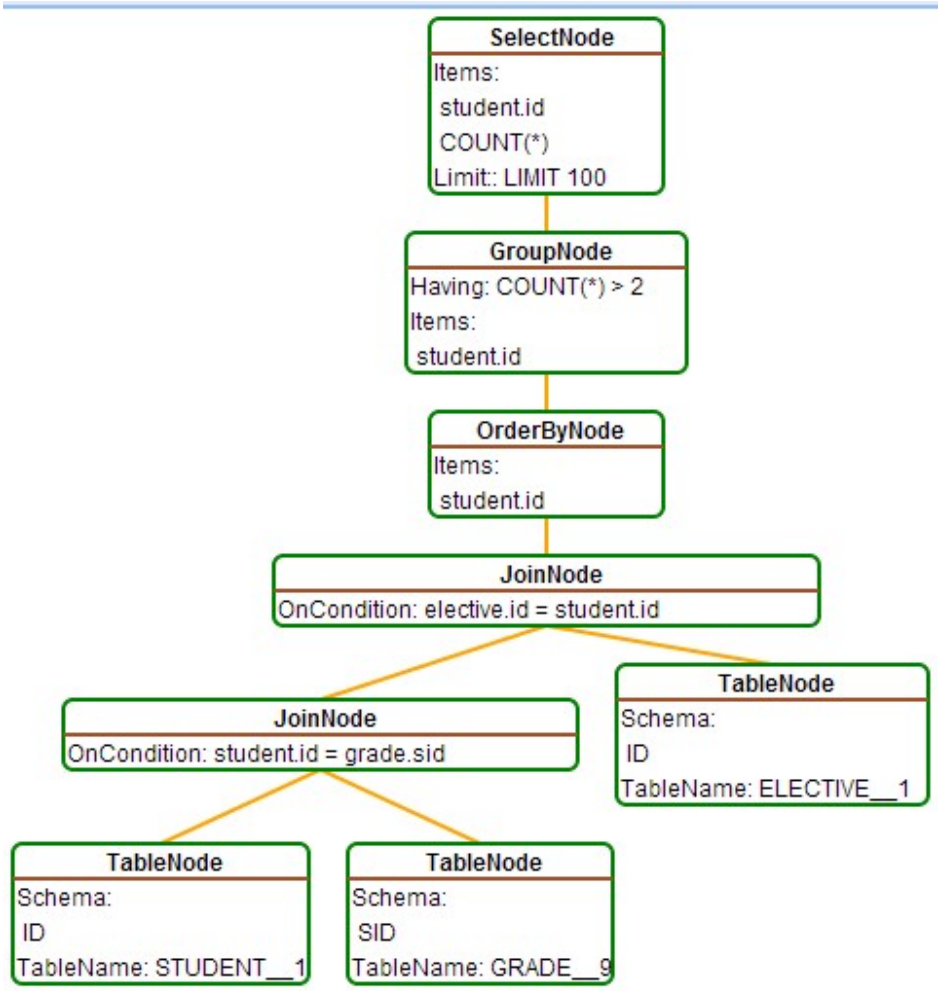
```
 explain
select student.id, count(*)
from test4dmp.student
inner join test4dmp.grade on student.id =grade.sid
inner join test4dmp.elective on elective.id=student.id
group by student.id
having count(*) > 2
order by student.id
limit 10
```

逻辑计划的explain string

```
 {
```

```
"Items": ["student.id", "COUNT(*)"],
"Name": "SelectNode",
"Node": {
"Having": "COUNT(*) > 2",
"Items": ["student.id"],
"Name": "GroupNode",
"Node": {
"Items": ["student.id"],
"Name": "OrderByNode",
"Node": {
"Name": "JoinNode",
"LeftNode": {
"Name": "JoinNode",
"LeftNode": {
"Name": "TableNode",
"Schema": ["ID"],
"TableName": "STUDENT__1"
},
"OnCondition": "student.id = grade.sid",
"RightNode": {
"Name": "TableNode",
"Schema": ["SID"],
"TableName": "GRADE__9"
}
},
"OnCondition": "elective.id = student.id",
"RightNode": {
"Name": "TableNode",
"Schema": ["ID"],
"TableName": "ELECTIVE__1"
}
}
}
},
"Limit:": "LIMIT 100"
}
```


逻辑计划explain string的图形化展示效果：



- 逻辑计划各个节点说明：
  - SelectNode表示这select中最终输出表达式的相关信息， 例如select要输出的表达式集合
  - GroupNode表示GroupBy语句的相关信息，例如groupby的列，having的表达式等
  - OrderByNode表示OrderBy的列信息，例如列名，顺序等。
  - JoinNode表示逻辑Join树的信息，例如join的on条件
  - TableNode表示分区表的信息，例如参与计算的列，表名等。

Explain 物理执行计划详细语义

样例sql

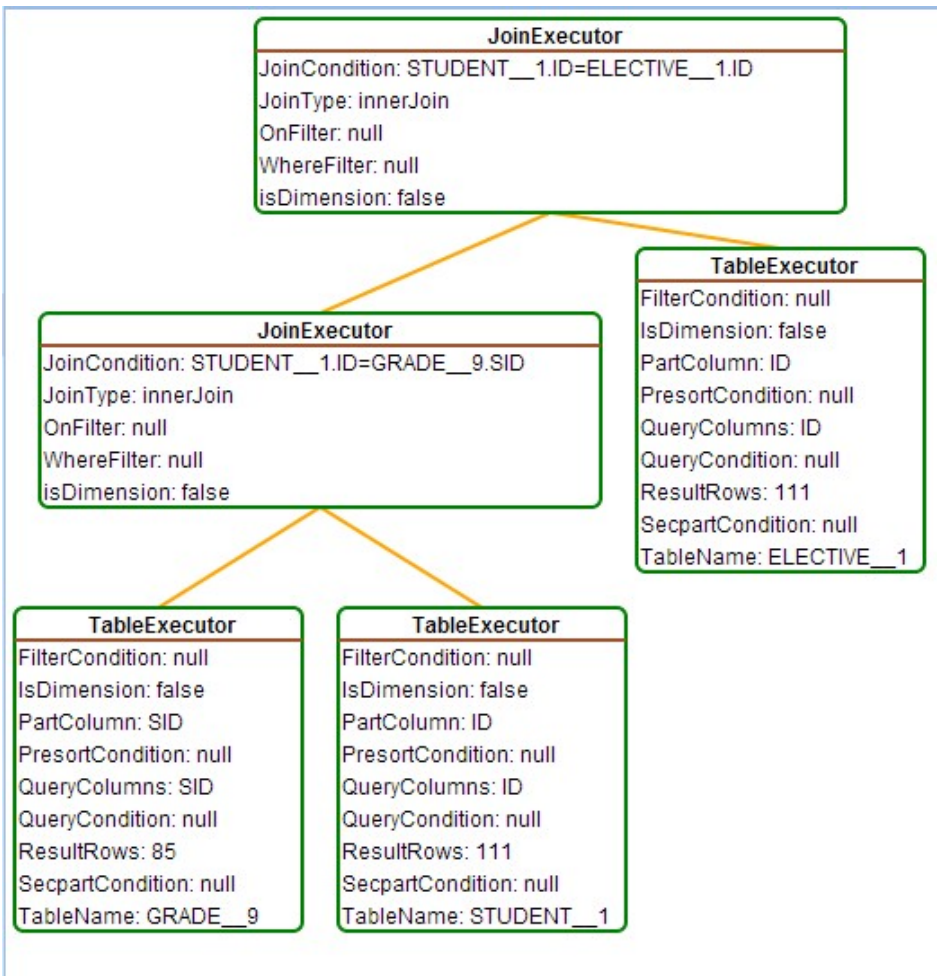
```
 explain
select student.id, count(*)
from test4dmp.student
inner join test4dmp.grade on student.id =grade.sid
inner join test4dmp.elective on elective.id=student.id
group by student.id
having count(*) > 2
order by student.id
```

```
limit 10
```

物理计划的explain string

```
 {
"Name": "JoinExecutor",
"isDimension": "false",
"LeftNode": {
"Name": "TableExecutor",
"PresortCondition": "null",
"SecpartCondition": "null",
"QueryColumns": "ID, BOOLEAN_TEST, LONG_TEST",
"FilterCondition": "null",
"ResultRows": 0,
"PartColumn": "ID",
"IsDimension": "false",
"TableName": "TEST__2",
"QueryCondition": "(ID = 0)"
},
"JoinCondition": "TEST__2.ID=TEST__2.ID",
"RightNode": {
"Name": "TableExecutor",
"PresortCondition": "null",
"SecpartCondition": "null",
"QueryColumns": "ID, BOOLEAN_TEST",
"FilterCondition": "null",
"ResultRows": 0,
"PartColumn": "ID",
"IsDimension": "false",
"TableName": "TEST__2",
"QueryCondition": "((INT_TEST < 0) AND (ID = 0))"
},
"JoinType": "innerJoin",
"WhereFilter": "(BOOLEAN_TEST = BOOLEAN_TEST)",
"OnFilter": "null"
}
```

物理计划explain string的图形化展示效果：



- 物理计划各个节点说明：
    - JoinExecutor表示Join的节点
        - JoinCondition：join的条件
        - JoinType：join的类型，innerJoin, leftJoin, rightJoin, simiJoin等
        - OnFilter: on上面的过滤条件
        - WhereFilter： where里面的过滤条件（只有leftjoin中on和where的条件才有差异）
        - IsDimension：该Join子树中是否含有维度表
    - TableExecutor表示参与计算的表信息
        - FilterCondition: 该表参与过滤计算的表达上（不能下沉索引的表达式）
        - IsDimension：该表是否是维度表
        - PartColumn：一级分区列
        - SecPartColumn：二级分区列
        - QueryColumns: 参与计算的列数（下沉索引计算的列不包含在此之内）
        - QueryCondition： 索引条件
        - ResultRows： 该节点预估cost
        - PresortCondition： 预排序条件，可以用做优先索引条件下沉
        - SecpartCondition：二级分区筛选条件，可以用做二级分区筛选。
        - TableName: 分区表名

语法

```
/*+engine=mpp*/ EXPLAIN [ ( option [, ...] ) ] statement

其中 option 可以是以下之一:

FORMAT { TEXT | GRAPHVIZ }
TYPE { LOGICAL | DISTRIBUTED | VALIDATE }
```

描述

通过EXPLAIN可以查看一个语句的逻辑或分布式执行计划，或者对一个语句进行逻辑验证。 通过 TYPE DISTRIBUTED 选项可以查看分布式的分片计划（fragmented plan）。每个分片计划由一个或者多个COMPUTENODE Worker节点执行。 多个分片在COMPUTENODE Worker节点之间通过Data Exchange交换数据。分片类型指定了分片如何在COMPUTENODE Worker节点之间交换数据，以及数据在多个分片中如何分布:

SINGLE

- 分片在单个COMPUTENODE Worker节点执行。

HASH

- 分片在固定个数的COMPUTENODE Worker节点执行，每个节点的输入数据通过Hash函数进行分桶输入。

ROUND_ROBIN

- 分片在固定个数的COMPUTENODE Worker节点执行，每个节点的输入数据通过轮询的方式进行输入。

BROADCAST

- 分片在固定个数的COMPUTENODE Worker节点执行，输入数据通过广播的方式广播到所有节点。

SOURCE

- 分片在输入数据源的访问节点上执行。

例子

逻辑执行计划:

```
tiny> /*+engine=mpp*/ EXPLAIN SELECT regionkey, count(*) FROM nation GROUP BY 1;
Query Plan
-----------------------------------------------------------------------------------------------------
- Output[regionkey, _col1] => [regionkey:bigint, count:bigint]
```

```
_col1 := count
- RemoteExchange[GATHER] => regionkey:bigint, count:bigint
- Aggregate(FINAL)[regionkey] => [regionkey:bigint, count:bigint]
count := "count"("count_8")
- LocalExchange[HASH][$hashvalue] ("regionkey") => regionkey:bigint, count_8:bigint, $hashvalue:bigint
- RemoteExchange[REPARTITION][$hashvalue_9] => regionkey:bigint, count_8:bigint, $hashvalue_9:bigint
- Project[] => [regionkey:bigint, count_8:bigint, $hashvalue_10:bigint]
$hashvalue_10 := "combine_hash"(BIGINT '0', COALESCE("$operator$hash_code"("regionkey"), 0))
- Aggregate(PARTIAL)[regionkey] => [regionkey:bigint, count_8:bigint]
count_8 := "count"(*)
- TableScan[tpch:tpch:nation:sf0.1, originalConstraint = true] => [regionkey:bigint]
regionkey := tpch:regionkey
```

分布式执行计划:

```
tiny> /*+engine=mpp*/ EXPLAIN (TYPE DISTRIBUTED) SELECT regionkey, count(*) FROM nation GROUP BY 1;
Query Plan
-----------------------------------------------------------------------------------------------
Fragment 0 [SINGLE]
Output layout: [regionkey, count]
Output partitioning: SINGLE []
- Output[regionkey, _col1] => [regionkey:bigint, count:bigint]
_col1 := count
- RemoteSource[1] => [regionkey:bigint, count:bigint]

Fragment 1 [HASH]
Output layout: [regionkey, count]
Output partitioning: SINGLE []
- Aggregate(FINAL)[regionkey] => [regionkey:bigint, count:bigint]
count := "count"("count_8")
- LocalExchange[HASH][$hashvalue] ("regionkey") => regionkey:bigint, count_8:bigint, $hashvalue:bigint
- RemoteSource[2] => [regionkey:bigint, count_8:bigint, $hashvalue_9:bigint]

Fragment 2 [SOURCE]
Output layout: [regionkey, count_8, $hashvalue_10]
Output partitioning: HASH [regionkey][$hashvalue_10]
- Project[] => [regionkey:bigint, count_8:bigint, $hashvalue_10:bigint]
$hashvalue_10 := "combine_hash"(BIGINT '0', COALESCE("$operator$hash_code"("regionkey"), 0))
- Aggregate(PARTIAL)[regionkey] => [regionkey:bigint, count_8:bigint]
count_8 := "count"(*)
- TableScan[tpch:tpch:nation:sf0.1, originalConstraint = true] => [regionkey:bigint]
regionkey := tpch:regionkey
```

验证查询语句:

```
tiny> /*+engine=mpp*/ EXPLAIN (TYPE VALIDATE) SELECT regionkey, count(*) FROM nation GROUP BY 1;
Valid
-------
true
```

MPP引擎执行计划运行时分析

语法

/*+engine=mpp*/ EXPLAIN ANALYZE statement

描述

执行语句，并展示语句对应的分布式执行计划和每个操作的详细开销信息。

> **Note**
>
> 相关开销的统计信息可能不是完全准确，特别是对于执行非常快的查询。

例子

在以下例子中，你可以查看每个stage的CPU开销，以及在stage中每个Plan Node的相对开销。注意，每个Plan Node的相对开销是基于Wall Time的，Wall Time与CPU Time可能相关，也可能不相关。 针对每个Plan Node，你还能看到一些额外的统计信息内容（比如：每个节点实例的平均输入，相关Plan Node的平均Hash冲突数）。 这些统计信息有助于用户发现查询的执行时的数据异常情况（数据倾斜、异常Hash冲突等）。

```
sf1> /*+engine=mpp*/ EXPLAIN ANALYZE SELECT count(*), clerk FROM orders WHERE orderdate > date '1995-01-01' GROUP BY clerk;

Query Plan
-----------------------------------------------------------------------------------------------
Fragment 1 [HASH]
Cost: CPU 88.57ms, Input: 4000 rows (148.44kB), Output: 1000 rows (28.32kB)
Output layout: [count, clerk]
Output partitioning: SINGLE []
- Project[] => [count:bigint, clerk:varchar(15)]
Cost: 26.24%, Input: 1000 rows (37.11kB), Output: 1000 rows (28.32kB), Filtered: 0.00%
Input avg.: 62.50 lines, Input std.dev.: 14.77%
- Aggregate(FINAL)[clerk][$hashvalue] => [clerk:varchar(15), $hashvalue:bigint, count:bigint]
Cost: 16.83%, Output: 1000 rows (37.11kB)
Input avg.: 250.00 lines, Input std.dev.: 14.77%
count := "count"("count_8")
- LocalExchange[HASH][$hashvalue] ("clerk") => clerk:varchar(15), count_8:bigint, $hashvalue:bigint
Cost: 47.28%, Output: 4000 rows (148.44kB)
Input avg.: 4000.00 lines, Input std.dev.: 0.00%
- RemoteSource[2] => [clerk:varchar(15), count_8:bigint, $hashvalue_9:bigint]
Cost: 9.65%, Output: 4000 rows (148.44kB)
Input avg.: 4000.00 lines, Input std.dev.: 0.00%

Fragment 2 [tpch:orders:1500000]
Cost: CPU 14.00s, Input: 818058 rows (22.62MB), Output: 4000 rows (148.44kB)
Output layout: [clerk, count_8, $hashvalue_10]
Output partitioning: HASH [clerk][$hashvalue_10]
- Aggregate(PARTIAL)[clerk][$hashvalue_10] => [clerk:varchar(15), $hashvalue_10:bigint, count_8:bigint]
Cost: 4.47%, Output: 4000 rows (148.44kB)
Input avg.: 204514.50 lines, Input std.dev.: 0.05%
Collisions avg.: 5701.28 (17569.93% est.), Collisions std.dev.: 1.12%
```

```
count_8 := "count"(*)
- ScanFilterProject[table = tpch:tpch:orders:sf1.0, originalConstraint = ("orderdate" > "$literal$date"(BIGINT '9131')),
filterPredicate = ("orderdate" > "$literal$date"(BIGINT '9131'))] => [cler
Cost: 95.53%, Input: 1500000 rows (0B), Output: 818058 rows (22.62MB), Filtered: 45.46%
Input avg.: 375000.00 lines, Input std.dev.: 0.00%
$hashvalue_10 := "combine_hash"(BIGINT '0', COALESCE("$operator$hash_code"("clerk"), 0))
orderdate := tpch:orderdate
clerk := tpch:clerk
```

7.3 表结构优化

分区列选择

基本原理分析型数据库的表一级分区采用hash分区，可指定任意一列（不支持多列）作为分区列，然后采用以下标准CRC算法，计算出CRC值，并将CRC值模分区数，得出每条记录的分区号。空值的hash值与字符串”-1”相同。

```
private static long getCRC32(String value) {
Checksum checksum = new CRC32();
byte[] bytes = value.getBytes();
checksum.update(bytes, 0, bytes.length);
return checksum.getValue();
}
```

分析型数据库的调度模块会将同一个表组下所有表的相同分区分配在同一个计算节点上。好处在于，当出现多表使用分区列join时，单计算节点内部直接计算，避免跨机计算。

分区列选择依据（按优先级高低排序）：

1. 如果有多个事实表（不包括维度表）进行join，选择参与join的列作为分区列。

如果有多列join怎么办？可根据查询重要程度或者查询性能要求（例如某SQL的查询频率特别高），选择某列作为分区列，这样可以保证基于分区列join的查询性能具有较好的性能。

选择group by 或distinct 包含的列作为分区列。

选择值分布均匀的列，不要选择分区倾斜的列作为分区列。

常用SQL包含某列的等值或in查询条件，选择该列作为分区列。

例如：

select * from table where id=123 and ….;

或

select * from table where user in(1, 2,3);

一级分区个数

基本原理

分析型数据库的COMPUTENODE Local/Merge计算引擎（大部分查询主要的计算引擎），会在每个分区并行计算，每个分区计算使用一个线程，分区计算结果汇总到FRONTNODE。因此分区数过小，会导致并发低，单查询RT时间长。而如果分区数过多，会导致计算结果数过多，增加FRONTNODE压力；同时由于分区数过大，更容易产生长尾效应。因此需要根据资源配置和查询特点，选择合适的分区数。

一级分区个数选择依据

注意:一级分区数不可修改。如需修改，必须删表重建。

- 参快速join的多个事实表分区数必须相同。
- 单分区的数据记录数建议为300万条到2000万之间。如果为二级分区，保证每个一级分区下的二级分区的记录数为300万条到2000万条之间。
- 分区数应该大于ECU数量 X 6，同时需要考虑到将来扩容。例如：某DB为8个C1，则分区数需要大于8*6=48。
- 单表一级分区数最大值为256。在某些极其特殊的环境中，可能最大值为512。
- 单计算节点的分区数（包括二级分区）不能超过1万。

二级分区表适用场景以及二级分区保留个数设置

基本原理

二级分区主要是解决数据按固定时间（例如，天，周，月）增量导入，同时需要保留历史数据设计的。每个一级分区下会包含多个二级分区，其中每个二级分区的分区列值（通常为日期格式的bigint数值型，如2015091210）相同，并作为一个完整的索引构建单元（每次导入产生一个二级分区）。

在执行查询过程，计算引擎能够自动根据查询条件，筛选出满足的二级分级，然后对每个符合条件的二级分区执行计算。

如果二级分区过多，由于每个二级分区作为独立查询单元，导致多次索引查询，性能下降；同时由于每个二级分区有独立的meta，因此会占用更大内存。如果二级分区较少的话，用户导入频率会降低，会影响数据的实时性。因此用户需要根据实际情况综合评估合理的二级分区更新间隔，以及保留个数。

最佳实践

单表二级分区数小于等于90，同时每个计算节点上总二级分区个数不超过1万个。每个一级分区下的二级分区包含的数据条数在300万到2000万之间。

聚集列的选择和设置

基本原理

分析型数据库数据存储支持按一列或多列排序（先按第一列排序，第一列相同情况下，使用第二列排序），保证该列值相同或相近的数据在磁盘同一位置。它的好处是，当以聚集列为查询条件时，查询结果保存在磁盘相

同位置，可以减少IO次数。由于主聚集列只有一列，因此需要最合适的列作为主聚集列。

聚集列选择依据

- 主要或大多数查询条件包括这一列，且该条件具有较高的筛选率。
- Join 等值条件列（通常为一级分区列）作为聚集列。

列类型选择

基本原理AnalyticDB处理数值类型的性能远好于处理字符串类型。原因在于：

- 值类型定长，占用内存少，存储空间小；
- 数值类型计算更快，尤其是join时；

因此，强烈建议用户尽可能使用数值类型，减少使用字符串类型。

常见将字符串转换为数值类型方法：

- 包含字符前缀或后缀，例如E12345，E12346等。可以直接去掉前缀或者将前缀映射为数字。
- 该列只有少数几个值，例如国家名。可以对每个国家编码，每个国家对应一个唯一数字。

单表查询

索引和扫描选择

分析型数据库默认为全索引，对于查询的多个条件分别检索索引，得出多个结果集（行集合），然后采用流式归并算法得出满足组合条件的最终结果集。索引的性能主要受key的分布影响，包括：cardinality(散列程度)，范围查询的记录数/表记录数。

但是在以下四种情况下，索引性能较差。

- 范围查询（或等值查询）筛选能力差，即满足条件的记录数/表总记录超过10%。
- 不等于条件查询（不包括not null）。
- 中缀或后缀查询，例如 like ‘%abc’ 或like ‘%abc%’ 。
- AND 条件中某一条件具有高筛选能力，其他条件走索引性能比扫描性能差。

对于以上四种情况，扫描性能反而比索引的性能好。用户通过加hint的方法强制查询不走索引。

Hint格式如下：

```
/*+ no-index=[table1.x;table2.x;y]*/。
```

例如：

```
/*+ no-index=[table1.time]*/ select * from table1 where x= 3 and time between 0 and 10000
```

表示强制条件time between 0 and 10000走扫描。计算引擎首先检索列x的索引，得出满足条件x=3的行集合，然后读取每行所对应的time列数据，如果满足time between 0 and 10000，则将该行数据加入返回结果。

二级分区查询优化

一级分区包含多个二级分区；计算时，每个二级分区依次执行条件查询，并将所有二级分区的结果进行汇总。由于每个二级分区都要参与所有条件筛选（索引查询），当二级分区较多时，查询性能较差。如果能够预知数据的分布，确定二级分区的范围，可以在查询条件中增加二级分区列条件，这样可以快速过滤无效的二级分区，减少搜索范围。

例子：

```
select * from table where id = 3 and time between ‘2016-04-01 00:00:00’ and ‘2016-04-01 12:00:00’;
```

如果根据业务场景确认满足time between ‘2016-04-01 00:00:00’ and ‘2016-04-01 12:00:00’ 的二级分区列为20160401，则可以将该SQL改写为：

```
select * from table where id = 3 and time between ‘2016-04-01 00:00:00’ and ‘2016-04-01 12:00:00’ and pid = 20160401;
```

pid为二级分区列名。

条件改写

当SQL中条件为函数时，无法走索引过滤，自动走扫描。在大多数情况下，性能会比较差，因此尽量改写条件去除函数。例如以下SQL：

```
Select * from table where year（date_test） > 1990
```

应该改为

```
Select * from table where date_test > ‘1990-00-00’
```

多表查询

首先我们来复习一下可加速join的条件：

对于COMPUTENODE Local/Merge计算模式的表Join查询，需要满足以下几个条件：

- 连接计算的所有事实表必须在同一个表组。
- 所有事实表的一级分区数相同。
- 事实表Join时， on条件必须包含分区列且必须是等值查询，其他列无限制。
- Left join时，右表是事实表时，左表不能是维度表。

子查询使用

对于子查询，分析型数据库会首先执行子查询，并将子查询的结果保存在内存中，然后将该子查询作为一个逻辑表，执行条件筛选。由于子查询没有索引，所有条件筛选走扫描。因此如果子查询结果较大时，性能比较差；反之当子查询结果集较小时，扫描性能反而超过索引查询。

对于join查询，由于分析型数据库默认采用hash join算法，如果其中一张表结果集（条件筛选后）较大时，扫描性能会比索引差很多，因此尽量不要采用子查询。例如以下SQL：

```
Select A.id from table1 A join (select table2.id from table2 where table2.y = 6) B on A.id= B.id where A.x=5
```

当满足条件x=5 和y=6的条数较多时，应改成：

```
Select A.id from table1 A join table2 B on A.id = B.id where B.y = 6 and A.x=5
```

一个例外情况是，当结果集较大的表是实时表，应尽量采用子查询。原因在于，实时表的最新数据（基线合并后写入的数据），索引能力很弱，查询性能非常差。子查询可以减少搜索范围，性能反而更好。

小表广播/Full MPP Mode查询优化

Full MPP Mode 与COMPUTENODE Local/Merge模式选择

COMPUTENODE Local/Merge模式是分析型数据库支持的一种极速查询模式，通过对用户SQL的有效甄别，快速判断是否和数据分布相匹配，从而达到优化数据shuffle逻辑，下沉大量计算操作的目的。

使用Full MPP Mode一般需用户使用Hint显示指定。

用户也可通过在查询hint中指定来强制查询使用MPP或COMPUTENODE模式。典型使用COMPUTENODE Local/Merge模式的场景有：

- 不包含表连接或者子查询的任意查询。
- 表连接条件为等值条件且连接列均为一级分区列的任意查询。
- 子查询包含了分区列的的groupby操作的任意查询。

小表广播

小表广播是基于COMPUTENODE Local/Merge模式扩展出的一种支持子查询内部不包含分区列的join操作的查询模式。典型的使用场景有：

- 非分区列的连接。如：

```
select name from student1 a join student2 b on a.name = b.name
```

两个表分区列都为id，由于表连接为非分区列，该查询不能使用模式。若student2表数据量不超过3万条，使用小表广播后，查询可以改写为：

```
select name from student1 a join (select /*+broadcast=true*/ name from student2) b on a.id = b.id
```

即可通过COMPUTENODE模式进行快速查询。

- 子查询内不包含分区列。如：

```
select name from student1 where name in (select name from student2 where id = 10)
```

该查询无法使用COMPUTENODE模式计算出正确结果，若id=10的name记录数不大于3万条，可修改为小表广播后：

```
select name from student1 where name in (select /*+broadcast=true*/ name from student2 where id = 10)
```

Full MPP Mode最优写法

当查询无法使用COMPUTENODE模式进行查询时，对使用MPP查询的SQL进行必要的修改，可以很大程度的提高查询的速度。以TPC-H测试中第2个查询为例，原查询为：

```
select
s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address, s_phone, s_comment
from
(select p_partkey, p_mfgr from part p where p_size = 15 and p_type like '%BRASS') inner join partsupp ps on
p_partkey = ps_partkey
inner join supplier s on s_suppkey = ps_suppkey
inner join (
select p_partkey as min_p_partkey, min(ps_supplycost) as min_ps_supplycost from part p
inner join partsupp ps on p_partkey = ps_partkey
inner join supplier s on s_suppkey = ps_suppkey
inner join nation n on s_nationkey = n_nationkey
inner join region r on n_regionkey = r_regionkey and r_name = 'EUROPE'
group by p_partkey
)A on ps_supplycost = A.min_ps_supplycost and p_partkey =A.min_p_partkey
inner join nation n on s_nationkey = n_nationkey
inner join region r on n_regionkey = r_regionkey and r_name = 'EUROPE'
order by
s_acctbal desc,
n_name,
s_name,
p_partkey
limit 100
```

将小表结果进行前置，进行join的reordering后，查询性能提高了近1倍

```
/*+engine=MPP*/select
s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address, s_phone, s_comment
from
region r
inner join nation n on n_regionkey = r_regionkey and r_name = 'EUROPE'
inner join supplier s on s_nationkey = n_nationkey
inner join partsupp ps on s_suppkey = ps_suppkey
inner join (select p_partkey, p_mfgr from part p where p_size = 15 and p_type like '%BRASS') p on p_partkey =
```

```
ps_partkey
inner join (
select p_partkey as min_p_partkey, min(ps_supplycost) as min_ps_supplycost from region r
inner join nation n on n_regionkey = r_regionkey and r_name = 'EUROPE'
inner join supplier s on s_nationkey = n_nationkey
inner join partsupp ps on s_suppkey = ps_suppkey
inner join part p on p_partkey = ps_partkey
group by p_partkey
)A on ps_supplycost = A.min_ps_supplycost and p_partkey =A.min_p_partkey
order by
s_acctbal desc,
n_name,
s_name,
p_partkey
limit 100
```

Insert 语句最优写法

Insert语句标准语法如下：

```
INSERT [IGNORE]
[INTO] tbl_name
[(col_name,...)]
{VALUES | VALUE}
```

为了利用分析型数据库的高性能写入能力，建议写入时进行如下改造：

- 采用批量写入（batch insert）模式，即每次在VALUES部分添加多行数据，一般建议每次批量写入数据量大约为16KB，以提高网络和磁盘吞吐。
- 如果对一行的所有列都进行插入，则去除col_name并保证values顺序与表结构中的col_name顺序一致，以降低网络带宽耗用。
- 保持主键相对有序。AnalyticDB的insert语句要求必须提供主键，且主键可以为复合主键。当确定复合主键时，根据业务含义调整复合主键中各个列的次序，从业务层面保证插入时主键是严格递增或近似递增的，也可以提升实时写入速度。
- 增加ignore关键字。执行不带ignore关键字的insert sql，当主键冲突时，后续数据会覆盖之前插入的数据；带上ignore关键字，则主键冲突时，会保留之前插入的数据而自动忽略新数据。如果业务层没有数据覆盖的语义要求，则建议所有insert sql都加上ignore关键字，以减小覆盖数据带来的性能开销。

按hash分区列聚合写入

分析型数据库需要对数据进行分区存储，当一次Batch insert中含有属于不同分区的多行数据时，将会耗费大量CPU资源进行分区号计算。因此建议在写入程序中提前计算好每行数据的分区号，并且将属于同一分区的多行数据组成一个批次，一次性插入。

实现聚合写入目前主要有两种途径：

其一，用户自行实现该聚合方法，对分区号的计算规则为

partition_num = CRC32(hash_partition_column_value) mod m

其中hash_partition_column_value是分区列的值，m是分区总数。

示例代码如下：

```
import java.util.ArrayList;
import java.util.List;
import java.util.zip.CRC32;
import java.util.zip.Checksum;

public class BatchInsertExample {
// 一级分区个数
private int partitionCount;
// 分区列id，分区列在建表语句中序号
private int partitionColumnIndex;
// 总列数
private int columnCount;
// batch insert 条数
private int batchSize;
private List<List<Object[]>> partitionGroup;
private String tableName;
private boolean forTest = true;

public BatchInsertExample(String tableName, int partitionCount, int columnCount, int partitionColumnIndex, int
batchSize) {
this.columnCount = columnCount;
this.tableName = tableName;
this.partitionColumnIndex = partitionColumnIndex;
this.batchSize = batchSize;
this.partitionCount = partitionCount;
partitionGroup = new ArrayList<List<Object[]>>(partitionCount);
for(int i = 0; i < partitionCount; i++) {
List<Object[]> listi = new ArrayList<Object[]>(batchSize);
partitionGroup.add(listi);
}
}

/**
* Function role, batch insert recordCount records, all records are strings except null.
* @param records 的length 建议为 columnCount X batchSize。
*/
public void insert(Object[][] records, int recordCount) {
for(int i = 0; i < recordCount; i++) {
int partitionNum = getPartitionNum(records[i]);
List<Object[]> batchRecord = partitionGroup.get(partitionNum);
batchRecord.add(records[i]);
if (batchRecord.size() >= batchSize) {
String sql = generateInsertSql(batchRecord);
executeInsert(sql);
batchRecord.clear();
}
}

for(int pn = 0; pn < partitionCount; pn++) {
```

```
List<Object[]> batchRecord = partitionGroup.get(pn);
if (batchRecord.size() > 0) {
String sql = generateInsertSql(batchRecord);
executeInsert(sql);
batchRecord.clear();
}
}
}

/**
*
* @param batchRecord
* @return insert sql with multiple values,
* like insert into zs_test
values(NULL,NULL,NULL),(7,'7',7),(20,'20',20),(48,'48',48),(57,'57',57),(60,'60',60),(95,'95',95),(97,'97',97),(100,'100',100),
(102,'102',102)
*/
private String generateInsertSql(List<Object[]> batchRecord) {
StringBuffer sb = new StringBuffer("insert into ").append(tableName).append(" values");
for(int i = 0; i < batchRecord.size(); i++) {
if (i > 0) {
sb.append(",");
}
sb.append("(");
Object[] objs = batchRecord.get(i);
assert(objs.length == columnCount);
for(int j = 0; j < objs.length; j++) {
if (j > 0) {
sb.append(",");
}
if (objs[j] == null) {
sb.append("NULL");
} else {
sb.append(objs[j].toString());
}
}
sb.append(")");
}
return sb.toString();
}

/**
*
* @param record
* @return calculte partition num basing on partition column value
*/
private int getPartitionNum(Object[] record) {
Object value = record[partitionColumnIndex];
long crc32 = (value == null ? getCRC32("-1") : getCRC32(value.toString()));
return (int) (crc32 % this.partitionCount);
}

private final long getCRC32(String value) {
Checksum checksum = new CRC32();
byte[] bytes = value.getBytes();
checksum.update(bytes, 0, bytes.length);
```

```
return checksum.getValue();
}

/**
* user should implement it with mysql protocol.
* @param insertSql
*/
public void executeInsert(String insertSql) {
if (forTest) {
System.out.println(insertSql);
} else {
throw new RuntimeException("user should implement it!!!!!");
}
}

public static void main(String args[]) {
String tableName="zs_test";

int partitionCount = 12;
int batchSize = 10;
int columnCount = 3;
int partitionColunIndex = 0;

BatchInsertExample insert = new BatchInsertExample(tableName, partitionCount, columnCount,
partitionColunIndex, batchSize);

// buffersize should always be same with batchsize * partition-count
int bufferSize = partitionCount * batchSize;
int recordCount = 0;


Object objs[][] = new Object[bufferSize][];
int totalRecord = 1000;
for(int i = 0; i < totalRecord; i++) {
objs[recordCount] = i == 0 ? new Object[3]: new Object[]{i, "'" + i + "'", i};
recordCount++;
if (recordCount == bufferSize || (i == totalRecord - 1)) {
insert.insert(objs, recordCount);
recordCount = 0;
}
}
}
}
```

其二，采用阿里云数据集成产品进行实时数据实时同步。（专有云中“大数据开发套件”的“数据同步”任务即为采用“数据集成”工具实现）

提前进行optimize table

分析型数据库的optimize table是指对实时写入的数据进行索引构建并生成高度优化的文件结构的过程。

分析型数据库为实时写入的数据只建立了简单的索引，在进行optimize table之后则会建立相对复杂但是功能更强、性能更佳的索引；在适当时候进行optimize table是提升查询性能的好方法。

目前有两种方法进行基线合并：

其一，自动optimize table。目前系统每天会自动进行一次optimize table（一般在每晚20:00开始）。

其二，手动进行optimize table。AnalyticDB提供了optimize命令，用户可以手动发送该命令，强制系统立刻进行基线合并。命令格式如下：

```
optimize table <table_name>
```

第八章 在生产中使用分析型数据库

8.1 业务系统连接分析型数据库并进行查询

使用jdbc odbc php python R-Mysql等连接分析型数据库的例子和注意事项介绍，以及流控、retry链接、异常处理等方法。

分析型数据库集群系统部署在阿里云环境中，用户在部署业务应用系统时，尽可能保证业务系统与阿里云环境的网络联通性，如购买阿里云ECS主机（ https://www.aliyun.com/product/ecs/ ）作为业务系统的服务器。

JDBC直接连接分析型数据库

最简单直接的连接并访问分析型数据库的方式是通过JDBC，分析型数据库支持MySQL自带的客户端以及大部分版本的mysql-jdbc驱动。

支持的mysql jdbc驱动版本号

- **5.0系列**： 5.0.2，5.0.3，5.0.4，5.0.5，5.0.7，5.0.8
- **5.1系列**：
  5.1.1，5.1.2，5.1.3，5.1.4，5.1.5，5.1.6，5.1.7，5.1.8，5.1.11，5.1.12，5.1.13，5.1.14，5.1.15，5.1.16，5.1.17，5.1.18，5.1.19，5.1.20，5.1.21，5.1.22，5.1.23，5.1.24，5.1.25，5.1.26，5.1.27，5.1.28，5.1.29，5.1.31，5.1.32，5.1.33，5.1.34
- **5.4系列**
- **5.5系列**

Java程序中，将合适的mysql-jdbc驱动包（mysql-connector-java-x.x.x.jar）加入CLASSPATH中，通过以下示例程序就能连接并访问分析型数据库。通过该JDBC方式直连分析型数据库时，和直连MySQL类似，需注意在使用完连接并不准备进行复用的情况下，需要释放连接资源。用户根据具体情况，设置${user_db}，${url}，${my_access_key_id}，${my_access_key_secret}和${query}值。

不带重试的JDBC样例程序片段

```
Connection connection = null;
Statement statement = null;
ResultSet rs = null;
try {
Class.forName("com.mysql.jdbc.Driver");
String url = "jdbc:mysql://mydbname.ads-
hz.aliyuncs.com:5544/my_ads_db?useUnicode=true&characterEncoding=UTF-8";

Properties connectionProps = new Properties();
connectionProps.put("user", "my_access_key_id");
connectionProps.put("password", "my_access_key_secret");

connection = DriverManager.getConnection(url, connectionProps);
statement = connection.createStatement();

String query = "select count(*) from information_schema.tables";
rs = statement.executeQuery(query);

while (rs.next()) {
System.out.println(rs.getObject(1));
}

} catch (ClassNotFoundException e) {
e.printStackTrace();
} catch (SQLException e) {
e.printStackTrace();
} catch (Exception e) {
e.printStackTrace();
} finally {
if (rs != null) {
try {
rs.close();
} catch (SQLException e) {
e.printStackTrace();
}
}
if (statement != null) {
try {
statement.close();
} catch (SQLException e) {
e.printStackTrace();
}
}
if (connection != null) {
try {
connection.close();
} catch (SQLException e) {
e.printStackTrace();
}
}
}
```

带重试的JDBC样例程序片段

```
public static final int MAX_QUERY_RETRY_TIMES = 3;
```

```
public static Connection conn = null;
public static Statement statement = null;
public static ResultSet rs = null;

public static void main(String[] args) throws ClassNotFoundException {
String yourDB = "user_db";
String username = "my_access_key_id";
String password = "my_access_key_secret";

Class.forName("com.mysql.jdbc.Driver");
String url = "jdbc:mysql://mydbname.ads-hz.aliyuncs.com:5544/" + yourDB +
"?useUnicode=true&characterEncoding=UTF-8";

Properties connectionProps = new Properties();
connectionProps.put("user", username);
connectionProps.put("password", password);

String query = "select id from test4dmp.test limit 10";

int retryTimes = 0;
while (retryTimes < MAX_QUERY_RETRY_TIMES) {
try {
getConn(url, connectionProps);
execQuery(query);
break; // Query execution successfully, break out.

} catch (SQLException e) {
System.out.println("Met SQL exception: " + e.getMessage() + ", then go to retry task ...");
try {
if (conn == null || conn.isClosed()) {
retryTimes++;
}
} catch (SQLException e1) {
if (conn != null) {
try {
conn.close();
} catch (SQLException e2) {
e.printStackTrace();
}
}
}
}
}

// Clear connection resource.
closeResource();
}

/**
* Get connection.
*
* @param url
* @param connectionProps
* @throws SQLException
*/
```

```
public static void getConn(String url, Properties connectionProps) throws SQLException {
conn = DriverManager.getConnection(url, connectionProps);
}

/**
* Query task execution logic.
*
* @param sql
* @throws SQLException
*/
public static void execQuery(String sql) throws SQLException {
Statement statement = null;
ResultSet rs = null;
statement = conn.createStatement();

for (int i = 0; i < 10; i++) {
long startTs = System.currentTimeMillis();
rs = statement.executeQuery(sql);
int cnt = 0;
while (rs.next()) {
cnt++;
System.out.println(rs.getObject(1) + " ");
}
long endTs = System.currentTimeMillis();
System.out.println("Elapse Time: " + (endTs - startTs));
System.out.println("Row count: " + cnt);
try {
Thread.sleep(160000);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}

/**
* Close connection resource.
*/
public static void closeResource() {
if (rs != null) {
try {
rs.close();
} catch (SQLException e) {
e.printStackTrace();
}
}
if (statement != null) {
try {
statement.close();
} catch (SQLException e) {
e.printStackTrace();
}
}
if (conn != null) {
try {
conn.close();
} catch (SQLException e) {
```

```
e.printStackTrace();
}
}
}
```

通过JDBC连接池连接分析型数据库

通过JDBC连接池来连接分析型数据库是不错的选择，在1.5章节中，已经演示了通过连接池Druid连接分析型数据库的方式，请参考。

通过PHP连接分析型数据库

在1.5章节中，已经演示了在PHP应用中连接分析型数据库的方式，请参考。

关于负载均衡

用户在创建数据库（章节3.1）时，分析型数据库会根据用户指定的服务参数，为用户在分析型数据库集群中分配特定数量的前端服务机，这些机器对用户透明，用户得到的是访问该数据库的URL（例如：mydbname-xxxx.ads-hz.aliyuncs.com:5544，在控制台中的连接信息中获取），分析型数据库集群自己使用阿里云负载均衡产品SLB（ http://www.aliyun.com/product/slb ）来对访问请求进行负载均衡，用户无需关心访问分析型数据库时的负载均衡策略。若用户的业务应用自己想采用一定的负载均衡方案，可以参考 http://www.aliyun.com/product/slb 。

关于重试以及异常处理

网络环境下连接和使用分析型数据库，从业务应用的角度来看，使用适当的重试和异常处理机制是保证业务应用高可用性的手段之一。无论是通过JDBC直连、JDBC连接池、PHP，还是Python等方式连接分析型数据库，出现连接连接异常时，应用均可采用适当的重试机制来保证连接可用性。在海量数据并发处理任务量大的情况下，偶尔在查询过程中出现异常时，也可采用适当的重试机制来重发查询。

在使用分析型数据库时，稳定的数据导入是非常重要的生产要素。一般新用户经常在进行首次的数据导入时因为操作不当无法成功，或成功后无法稳定运行。这里我们来看一下建立一个生产化的数据导入任务的注意事项。

数据的准备方面

想要稳定的导入数据，首先要在数据的源头稳定的产出数据。一份对于分析型数据库来说稳定的数据至少要满足：

数据所在的项目名（对应源头为ODPS）/文件访问路径（对应源头为OSS，暂不支持）/服务器连接串（对应源头为RDS，暂不支持）和表名与LOAD DATA命令的中的源头一致并保持稳定。

数据表的字段名，在源头上与在分析型数据库上的配置一致，源头表可以比分析型数据库有更多的字段，但是不能比分析型数据库表缺少字段。

源头表进行导入的分区的数据不能为空，进行导入的数据主键不能有NULL值，HASH分区键不能存在大量NULL值或同样的HASH分区键的数据条数过多，例如超过了每个分区的平均数据量的三倍。否则不仅会对查询性能造成影响，在极端情况下也会导致数据导入时间过长或者失败。

调用导入命令

在数据产出后，可以通过MySQL连接的方式或者HTTP Rest-API的方式调用数据导入命令，这时应该注意：

调用命令时，命令所引用的源头表/分区的数据已经完整的产出完毕，并且若源头是ODPS/OSS，应该不在有任何在源头的写入操作。所以通常需要一个良好的离线任务调度系统（例如阿里云DPC中的数据开发平台）来进行相关的任务运行和调度。

调用命令时，要确保命令所引用的源头表/分区已经对ALIYUN$garuda_build@aliyun.com授予足够的权限并未开启保护模式等阻止数据流出的安全策略。

调用命令时分析型数据库中该表没有正在运行的导入任务，否则会返回失败。

查询数据导入状态和解决导入中的问题

在生产系统中查询数据导入状态，通常更多的是通过HTTP API进行的查询的，这里如果有一个较好的离线任务调度系统，那么实现难度并不大。

在数据导入的过程中，经常会因为出现各种错误而导致导入中断，具体的错误处理可以见附录一：错误码中。

目前，阿里云分析型数据库能够支持在部分BI工具中以MySQL连接的方式进行连接和使用。

前提条件

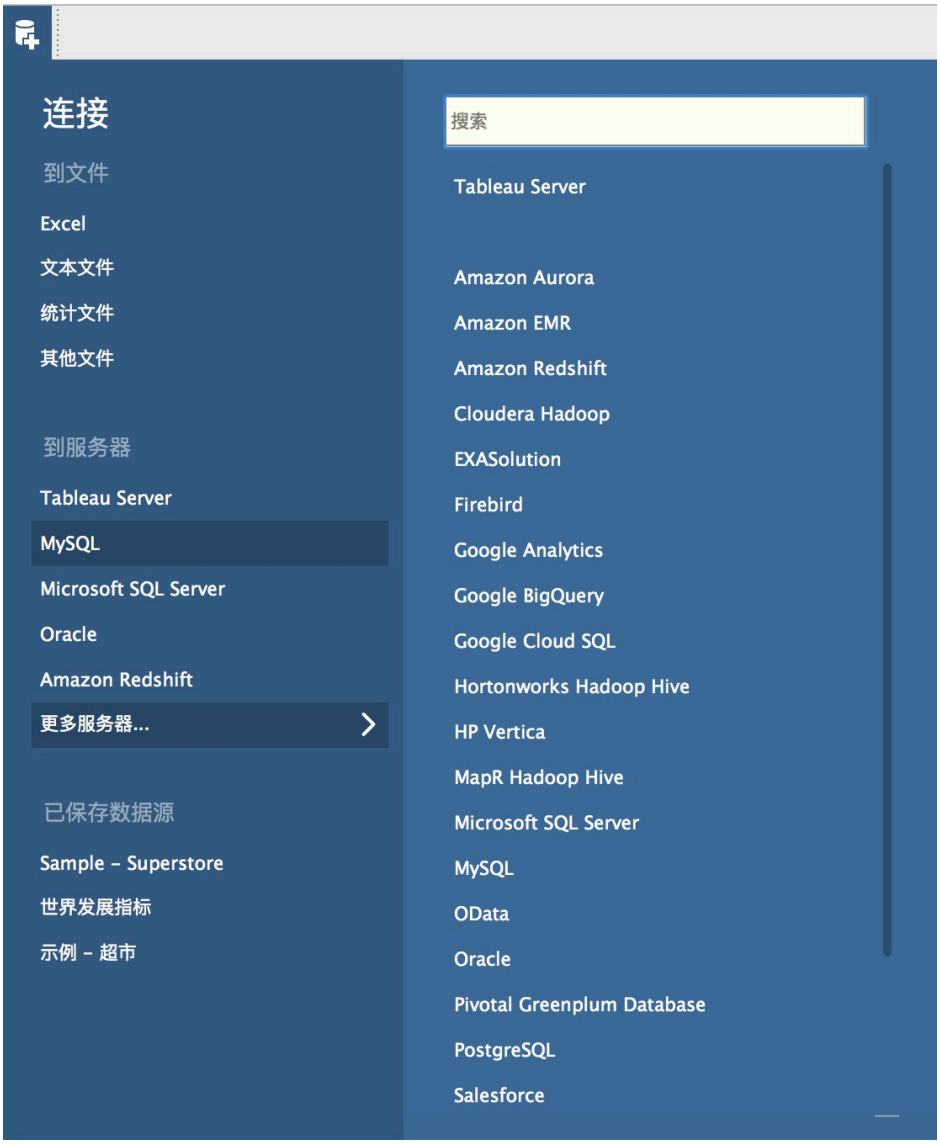
在使用这些BI工具连接分析型数据库时，通常需要先安装好OBDC MySQL Driver，这里推荐使用MySQL Connector/ODBC 3.51 版本或5.3版本（下载地址 http://dev.mysql.com/downloads/connector/odbc/3.51.html 或 https://dev.mysql.com/downloads/connector/odbc/5.3.html）。

在Tableau Desktop中使用分析型数据库

现在可以在Tableau Desktop 9.0及以上版本，通过MySQL连接的方式连接到阿里云分析型数据库读取和分析数据，产生可视化报表，大部分Tableau Desktop 9.x的功能可以在阿里云分析型数据库的连接中使用（如需最大限度的兼容如兼容全部函数等，需开启BI 工具兼容模式，V2.0.2版本后可以提交工单申请开通）。

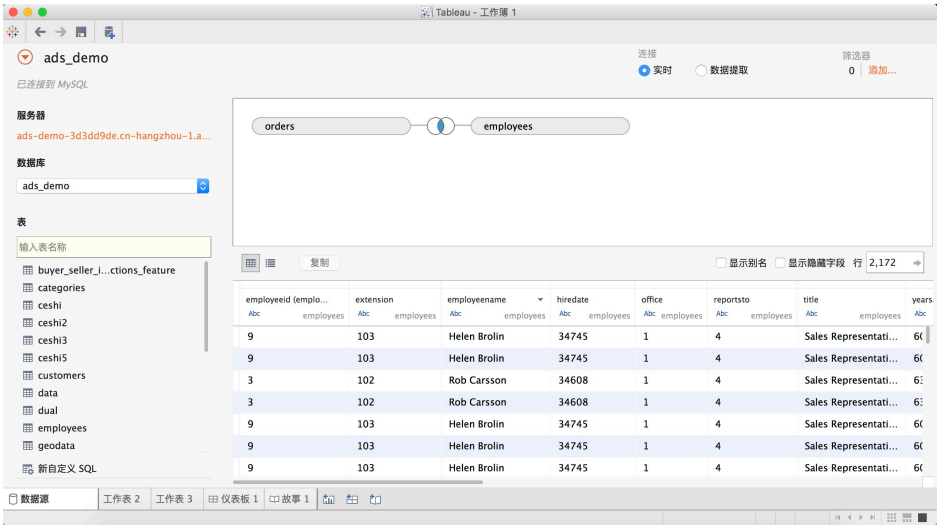
操作步骤

步骤一：打开Tableau Desktop，点击“添加新的数据源”按钮 并选择数据源类型为MySQL。

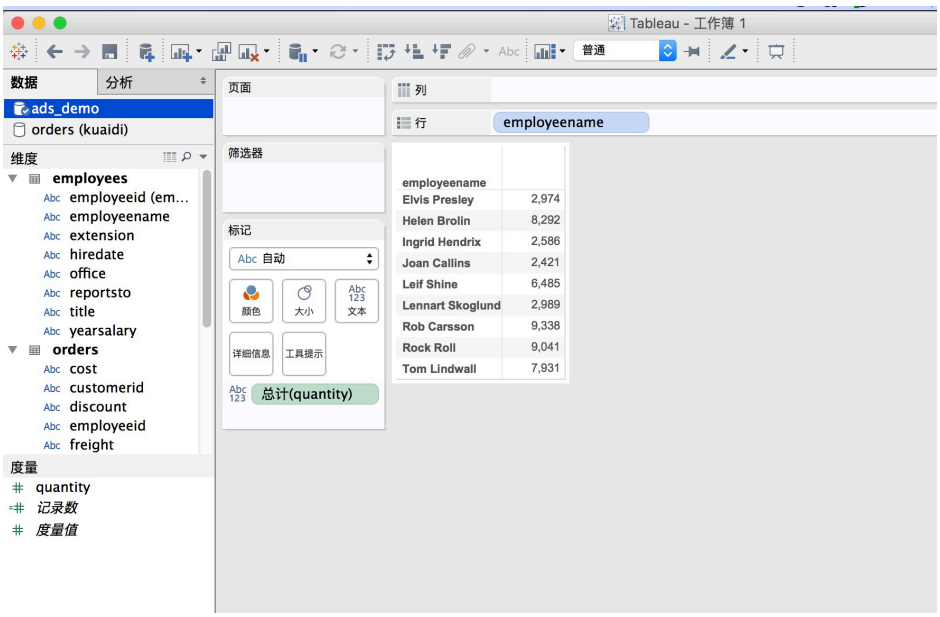


步骤二：在弹出的对话框中，输入您要访问的数据库的域名和端口号（在DMS的数据库管理页面获取），以及将您的账号在阿里云的Access Key ID / Access Key Secret输入到用户名/密码框中，点击确定。

步骤三：在Tableau的数据源页面上，可以看到您有权限的数据库，注意，您这里只能选择您连接到的域名端口所对应的数据库。选择后可以进行数据表选取，以及数据预览。



步骤四：后续便可以进行工作表的建立以及更多其他功能。

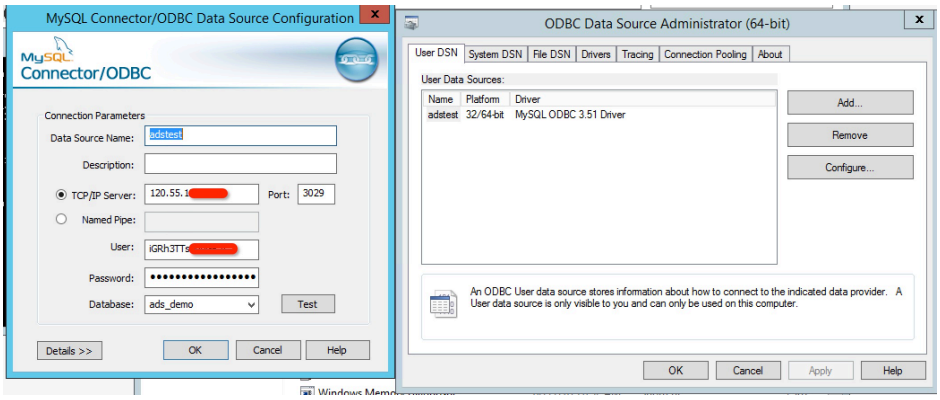


在QlikView中使用分析型数据库

操作步骤

步骤一：创建数据源。

要使用QlikView连接分析型数据库，首先需要配置操作系统的ODBC数据源。安装OBDC MySQL Driver后，进入 控制面板->管理工具->ODBC数据源（不同操作系统此步骤可能不同），新建一个用户DSN，选择“MySQL ODBC 5.xx Driver”一项。

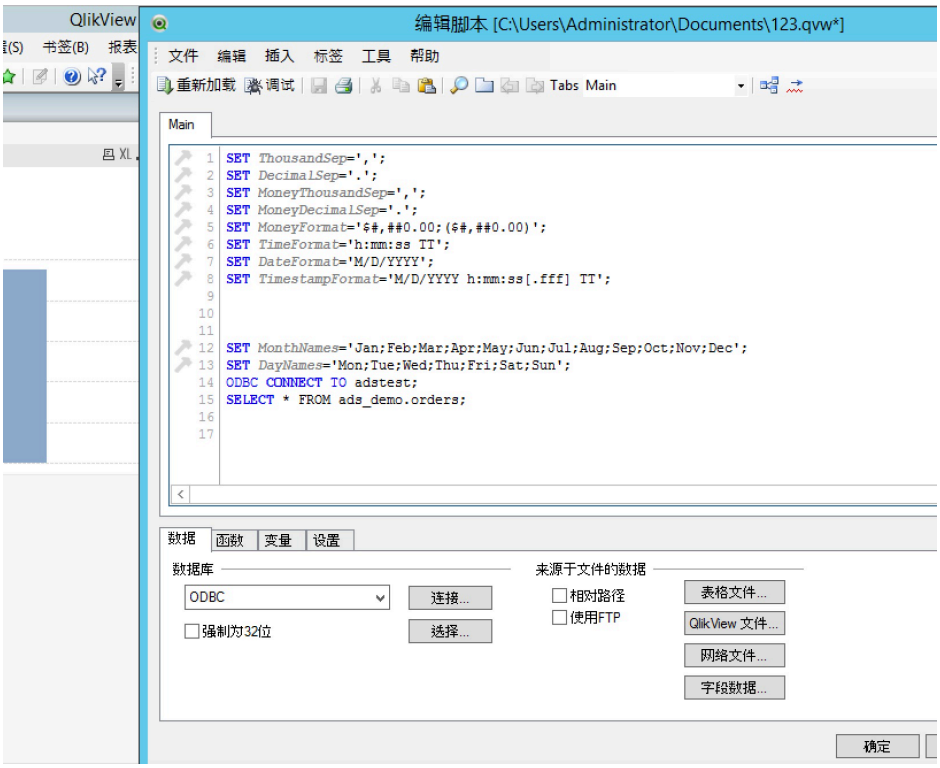


步骤二：填写数据源信息。

- data source name，输入数据名称。
- ODBC TCP/IP server，填入连接分析型数据库的域名和端口号（在DMS的数据库管理页面获取），部分情况下，可能需要您将域名解析为IP地址填入。
- user/password,将您的账号在阿里云的Access Key ID / Access Key Secret输入。
- database，选择您要连接的数据库，
- 上述配置项都填写完成后，点击“test”按钮测试连接通，若连通测试通过，则点击“OK”按钮确认添加数据源。

步骤三：通过QlikView读取AnalyticDB数据。打开QlikView（建议版本：11.20.x），打开“编辑脚本”，在末端键入：

```
ODBC CONNECT TO adstest;
select * from ads_demo.example_table limit 100;
```



其中adstest为之前在ODBC中建立的数据源名称。之后运行脚本，即可在QlikView中读取和使用分析型数据库中的数据。

在Microsoft Excel中读取分析型数据库中的数据

若是要在Excel中读取分析型数据库中的数据，建议安装微软的免费插件：Microsoft Power Query for Excel。需要按照Qlikview一节中相同的方式配置好ODBC数据源之后，参照MySQL相关教程进行。

注意：本章节目前只针对公共云，专有云该功能尚未可用。

通过阿里云数据传输（ https://www.aliyun.com/product/dts/ ），可以将您在阿里云RDS中的数据表的变更实时同步到分析型数据库中对应的实时写入表中（RDS端目前暂时仅支持MySQL引擎）。

详见数据传输的相关文档 https://help.aliyun.com/document_detail/49082.html 。

之前使用数据订阅 + dts-ads-writer的用户，建议尽快切换到当前数据传输提供的标准模式。

若用户希望在本地PC上连接和管理分析型数据库，则可以使用MySQL官方的命令行工具，或建议使用SQL WorkBench/J工具进行连接和管理。

使用MySQL官方命令行工具时，支持5.1.x/5.5.x/5.6.x版本的MySQL CLI，连接范例如下：

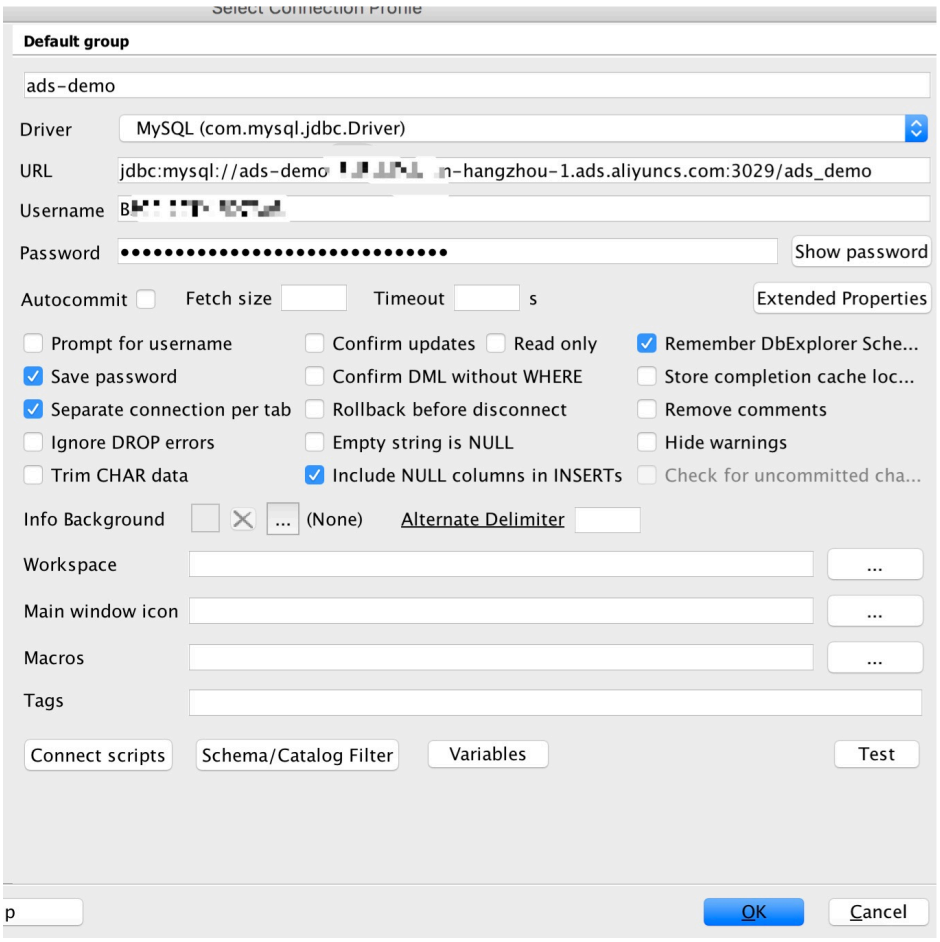
```
[cloud@iZ28unhwyqvZ ~]$ mysql -h<DOMAIN_URL> -P<PORT> –u<ACCESS_KEY_ID> -p<ACCESS_KEY_SECRET> -D<DB_NAME>
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 968288632
Server version: 5.1.31-mysql-ads

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

需要注意的是，连接时建议使用 -D 提前指定数据库名，或是在连接成功后立刻使用use语句切换数据库，否则执行其他命令时会报”only <create database> or <show database> are allowed from admin database connection”错误。

更多的，我们会使用一些本地GUI工具在PC机上进行开发调试，很多工具均能部分上兼容分析型数据库。综合来看兼容性最好的跨平台GUI工具是SQL WorkBench/J，目前可以在 http://www.sql-workbench.net/ 上免费下载。

下载好SQL Workbench/J程序后，还需要下载MySQL JDBC 驱动（如：http://dev.mysql.com/downloads/connector/j/ 中下载）。下载驱动后，打开SQL WorkBench/J程序，选择 File -> Manage Drivers，选择MySQL后填写刚刚下载的驱动jar包路径。

一切配置完毕后，点击File -> Connect Window，Driver选择MySQL，按照 jdbc:mysql://<Domain_url>:\<Port>/<db_name> 格式填写URL，并在用户名/密码中填写合法账号的ACCESS_KEY_ID和ACCESS_KEY_SECRET，如下图所示：



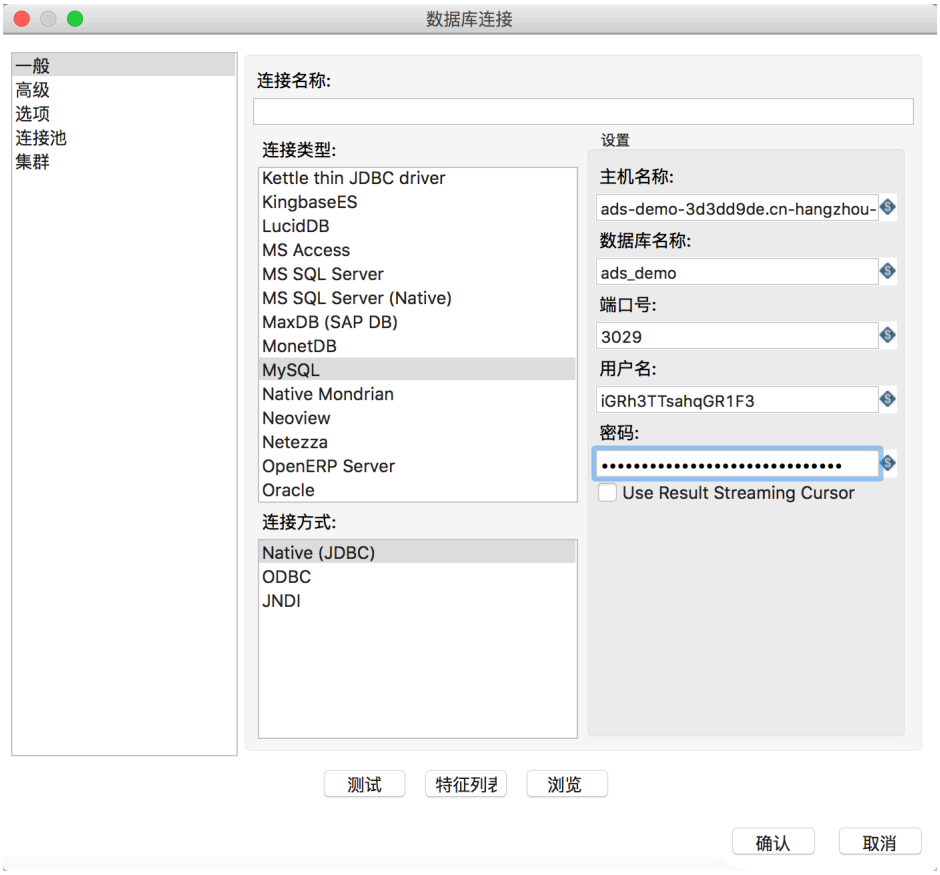
配置完毕后点击ok，就可以连接到分析型数据库，执行DDL或DML等。

Pentaho Data Integration(又称Kettle)是一款非常受欢迎的开源ETL工具软件。分析型数据库支持用户利用Kettle将外部数据源写入实时写入表中。

Kettle的数据输出程序并未为分析型数据库进行过优化，因此写入分析型数据库的速度并不是很快（通常不超过700 rec/s），不是特别适合大批量数据的写入，但是对于本地文件上传、小数据表等的写入等场景是非常合适的。

我们以导入本地excel文件为例，首先在分析型数据库中建立对应结构的实时写入表。然后用户可在 http://community.pentaho.com/projects/data-integration/ 上下载kettle软件，安装运行后，新建一个转换。

在该转换的DB连接中新建一项，连接类型选择MySQL，连接方式使用Native(JDBC)。主机名填写分析型数据库的连接域名，端口号填写链接端口号，用户名和密码填写access key信息，并去掉”Use Result Streaming Cursor”选项，如下图所示：

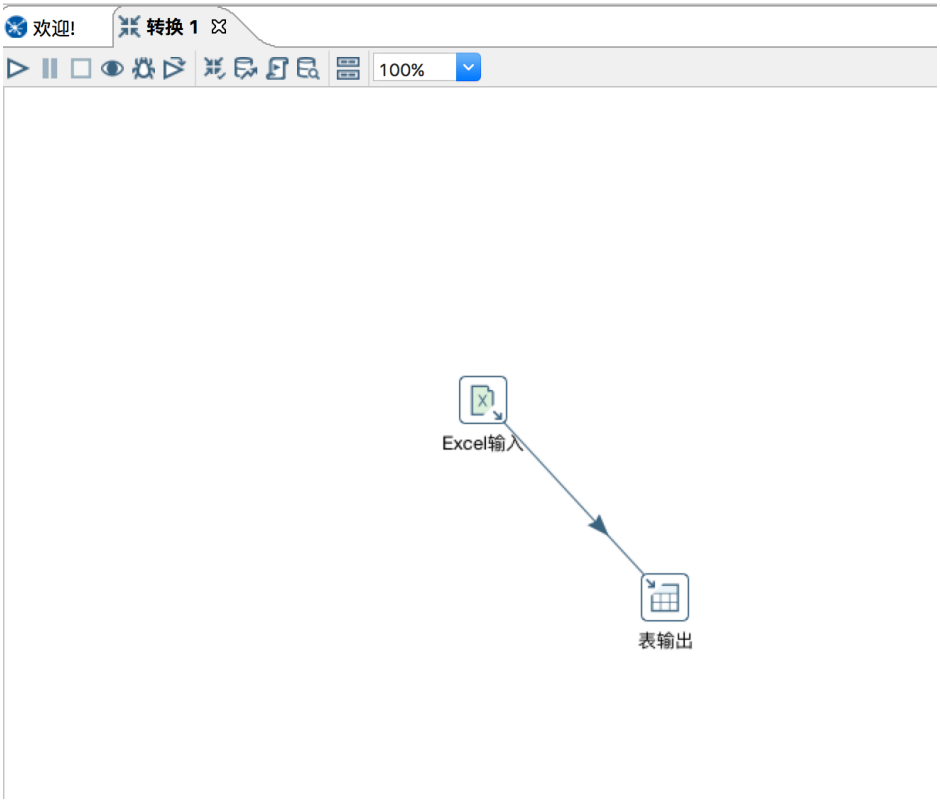


然后在kettle中，核心对象的“输入”中找到Excel输入拖拽到工作区，浏览并增加需要导入的Excel文件，根据实际需要设置工作表、内容、字段等选项卡，之后点击预览记录来查看输入的数据是否符合要求。

之后在核心对象的输出中找到表输出拖入工作区。新建一个从Excel输入指向表输出的连线。然后在表输出的属性中，手工填写目标模式（数据库名）、目标表名，暂不支持浏览功能。提交记录数量建议设置在30左右。选择“指定数据库字段”和“使用批量插入”，在数据库字段选项卡中点击获取字段和输入字段映射，映射excel文件的列与ads表的列名的映射关系，全部配置结束后如下：


表输出
步骤名称
表输出
数据库连接
ads_demo
编辑...
新建...
Wizard...
目标模式
ads_demo
浏览(B)...
目标表
student
浏览(B)...
提交记录数量
30
裁剪表
忽略插入错误
指定数据库字段
主选项
数据库字段
插入的字段:
表字段 流字段
1 yx 院系
2 bj 班级
3 xh 学号
4 xm 姓名
获取字段
输入字段映射
Help
确定(O)
取消(C)
SQL


表输出
步骤名称
表输出
数据库连接
ads_demo
编辑...
新建...
Wizard...
目标模式
ads_demo
浏览(B)...
目标表
student
浏览(B)...
提交记录数量
30
裁剪表
忽略插入错误
指定数据库字段
主选项
数据库字段
表分区数据
分区字段
每个月分区数据
每天分区数据
使用批量插入
表名定义在一个字段里?
包含表名的字段:
存储表名字段
返回一个自动产生的关键字
自动产生的关键字的字段名称
Help
确定(O)
取消(C)
SQL



之后便可单击白色三角箭头运行这个转换，观察运行日志和运行状态即可。

Kettle拥有非常强大的过滤、数据格式转换、清洗、抽取等功能，更多的使用详情请参考Kettle官方文档。

附录

分析型数据库的元数据库分为记载性能相关信息的performance_schema和记载元数据的information_schema，并和MySQL的元数据库有一定的兼容性，但是不是100%一致。

查询元数据库可以直接在JDBC连接中使用SQL语句进行查询，如：

```
SELECT state
FROM information_schema.current_job
WHERE table_schema='db_name'
AND table_name='table_name'
ORDER BY start_time DESC
LIMIT 10
```

即可查询分析型数据库某表近期的10次数据导入的状态。

information_schema 库

SCHEMATA

- SCHEMATA表提供了关于数据库的信息

FIELD	TYPE	ALLOW_NULL	PK	DEFAULT_VALUE	COMMENT
CLUSTER_NAME	varchar(16)	NO		NULL	集群名称
CATALOG_NAME	varchar(16)	YES		NULL	CATALOG名称
SCHEMA_NAME	varchar(64)	NO		NULL	SCHEMA名称
DEFAULT_CHARACTER_SET_NAME	varchar(64)	YES		UTF-8	默认字符集
DEFAULT_COLLATION_NAME	varchar(64)	YES		OFF	默认排序规则
id	int(11)	NO	PRI	NULL	ID
CREATOR_ID	varchar(64)	YES		NULL	创建者ID
CREATOR_NAME	varchar(64)	YES		NULL	创建者名称
SQL_PATH	varchar(255)	YES		NULL	SQL路径
DOMAIN_URL	varchar(255)	YES		NULL	域URL
QUERY_TYPE	varchar(64)	YES		NULL	查询类型
DISABLED	tinyint(1)	YES		0	禁用标记
LOAD_DISABLED	tinyint(1)	YES		0	禁止装载标记
TABLE_COUNT	int(11)	YES		NULL	表个数
TABLE_GROUP_COUNT	int(11)	YES		NULL	表组个数
MAX_TABLE_COUNT_LIMIT	int(11)	YES		256	最大表个数限制
MAX_TABLE_GROUP_COUNT_LIMIT	int(11)	YES		256	最大表组个数限制
QPS	int(11)	YES		-1	每秒查询数
LOAD_FACT	double	YES		-1	装载因子

OR					
INITIAL_CAPACITY	int(11)	YES		-1	初始化大小
INCREMENTAL_CAPACITY	int(11)	YES		-1	增量大小
DB_ASSIGN_STRATEGY	varchar(255)	YES		NULL	
SHARED	tinyint(1)	YES		0	是否是共享实例
CREATE_TIME	timestamp	NO		CURRENT_TIMESTAMP	创建时间
UPDATE_TIME	timestamp	NO		0000-00-0000:00:00	修改时间

- 如果用户需要了解当前数据库的基本信息包括当前表的个数，表组的个数，最大表的限制，最大表组的限制，每秒查询数等信息的时候可以查询这张表。例如：

```
SELECT CLUSTER_NAME, SCHEMA_NAME, DEFAULT_CHARACTER_SET_NAME, DEFAULT_COLLATION_NAME, TABLE_COUNT, TABLE_GROUP_COUNT, MAX_TABLE_COUNT_LIMIT, MAX_TABLE_GROUP_COUNT_LIMIT, QPS FROM information_schema.schemata WHERE SCHEMA_NAME = 'xxx';
```

TABLES

- TABLES表提供数据库表信息。该部分数据包括表的元数据与部分表对应数据的元数据，如分区信息等。

FIELD	TYPE	ALLOW_NULL	PK	DEFAULT_VALUE	COMMENT
CLUSTER_NAME	varchar(16)	NO	PRI	NULL	集群名称
TABLE_SCHEMA	varchar(128)	NO	PRI	NULL	所属SCHEMA的名称
TABLE_GROUP	varchar(128)	YES		NULL	所属表组的名称
TABLE_NAME	varchar(128)	NO	PRI	NULL	表名称
TABLE_SCHEMA_ID	varchar(128)	YES		NULL	所属SCHEMA的ID
TABLE_GROUP_ID	varchar(128)	YES		NULL	所属表组的ID
TABLE_ID	varchar(128)	YES		NULL	表的ID

TABLE_TYPE	varchar(128)	YES		NULL	标记：分区表PARTITION_TABLE、维度表DIMENSION_TABLE
UPDATE_TYPE	varchar(128)	YES		NULL	标记：批量表batch、实时表realtime
ONLINE_GROUP	varchar(128)	YES		NULL	
PARTITION_TYPE	varchar(128)	YES		NULL	分区类型：DIM、HASH
PARTITION_COLUMN	varchar(128)	YES		NULL	分区列名称
PARTITION_COUNT	int(11)	YES		NULL	分区数量
IS_SUB_PARTITION	tinyint(1)	YES		NULL	标记：是否是二级分区
SUB_PARTITION_TYPE	varchar(128)	YES		NULL	二级分区类型
SUB_PARTITION_COLUMN	varchar(128)	YES		NULL	二级分区列名称
SUB_PARTITION_COUNT	int(11)	YES		NULL	二级分区数量
CREATE_TIME	timestamp	NO		CURRENT_TIMESTAMP	创建时间
UPDATE_TIME	timestamp	NO		0000-00-0000:00:00	更新时间
CREATOR_ID	varchar(128)	YES		NULL	创建者ID
CREATOR_NAME	varchar(128)	YES		NULL	创建者名称
CURRENT_VERSION	bigint(20)	YES		NULL	当前版本
PREVIOUS_VERSION	bigint(20)	YES		NULL	上一版本
MIN_REDUNDANCY	int(11)	YES		2	最小副本数
COMMENTS	varchar(255)	YES		NULL	说明

CLUSTER_BY_COLUMNS	varchar(255)	YES		NULL	聚簇列名称
PRIMARY_KEY_COLUMNS	varchar(255)	YES		NULL	主键列名称
MAX_COLUMN_COUNT_LIMIT	int(11)	YES		990	列最多个数限制
EXECUTE_TIMEOUT	int(11)	YES		30000	执行超时时间
from_ctas	tinyint(1)	YES		NULL	是否创建于CTAS

- 如果用户需要了解某一个表的基本信息包括表的类型，一级分区键，二级分区键，分区类型，批量表或是实时表等信息的时候可以查询这张表。例如：

```
SELECT CLUSTER_NAME, TABLE_SCHEMA, TABLE_GROUP, TABLE_NAME, TABLE_TYPE, ONLINE_GROUP, PARTITION_TYPE, PARTITION_COLUMN, PARTITION_COUNT, SUB_PARTITION_TYPE, SUB_PARTITION_COLUMN, SUB_PARTITION_COUNT FROM information_schema.tables WHERE TABLE_SCHEMA = 'xxx' AND TABLE_GROUP = 'xxx' AND TABLE_NAME = 'xxx';
```

- 如果不知道具体的表名，只是希望了解某一个SCHEMA下有多少张不同类型的表。例如：

```
SELECT DISTINCT TABLE_GROUP, TABLE_NAME FROM information_schema.tables WHERE TABLE_SCHEMA = 'xxx';
```

- 如果希望了解某一个SCHEMA下面哪些表是维度表，哪些表是分区表。例如：

```
SELECT TABLE_GROUP, TABLE_NAME FROM information_schema.tables WHERE TABLE_SCHEMA = 'xxx' AND TABLE_TYPE = 'DIMENSION_TABLE';
```

- 如果希望了解某一个SCHEMA下面哪些表是批量表，哪些表是实时表。例如：

```
SELECT TABLE_GROUP, TABLE_NAME FROM information_schema.tables WHERE TABLE_SCHEMA = 'xxx' AND UPDATE_TYPE = 'realtime';
```

TABLE_GROUPS

- 该表存储了所有的表组的详细信息。

FIELD	TYPE	ALLOW_NULL	PK	DEFAULT_VALUE	COMMENT
CLUSTER_NAME	varchar(16)	NO	PRI	NULL	集群名称
TABLE_SCH	varchar(64)	NO	PRI	NULL	所属

EMA					SCHEMA
TABLE_GROUP	varchar(64)	NO	PRI	NULL	表组名称
TABLE_SCHEMA_ID	varchar(64)	YES		NULL	所属SCHEMA的ID
TABLE_GROUP_ID	varchar(64)	YES		NULL	所属表组的ID
CREATOR_ID	varchar(64)	YES		NULL	创建者ID
CREATOR_NAME	varchar(64)	YES		NULL	创建者名称
CREATE_TIME	timestamp	NO		CURRENT_TIMESTAMP	创建时间
UPDATE_TIME	timestamp	NO		0000-00-0000:00:00	更新时间
TABLE_COUNT	int(11)	YES		NULL	包含表的数量
MAX_TABLE_COUNT_LIMIT	int(11)	YES		256	最大表数量限制
MIN_REDUNDANCY	int(11)	YES		2	最小副本数
EXECUTE_TIMEOUT	bigint(20)	YES		30000	执行超时时间

- 如果希望了解某一个SCHEMA下面有多少个不同的表组，可以根据 TABLE_SCHEMA 查询 TABLE_GROUP 列。
- 如果希望了解某一个表组下包含了多少张表，可以根据 TABLE_SCHEMA 查询 TABLE_COUNT 列。

COLUMNS

- 该表存储了所有的表中字段的详细信息。

FIELD	TYPE	ALLOW_NULL	PK	DEFAULT_VALUE	COMMENT
CLUSTER_NAME	varchar(16)	NO	PRI	NULL	集群名称
TABLE_CATALOG	varchar(8)	YES		NULL	CATALOG名称
TABLE_SCHEMA	varchar(64)	NO	PRI	NULL	所属SCHEMA
TABLE_GROUP	varchar(64)	YES		NULL	所属表组

TABLE_NAME	varchar(64)	NO	PRI	NULL	所属表名
COLUMN_NAME	varchar(64)	NO	PRI	NULL	列名
ORDINAL_POSITION	int(11)	YES		NULL	在表中的位置
COLUMN_DEFAULT	varchar(255)	YES		NULL	
IS_NULLABLE	tinyint(1)	YES		1	是否允许空
DATA_TYPE	int(11)	YES		NULL	数据类型名称
TYPE_NAME	varchar(64)	YES		NULL	列类型名称
CHARACTER_MAXIMUM_LENGTH	int(11)	YES		NULL	字符最大长度
CHARACTER_OCTET_LENGTH	int(11)	YES		NULL	字符八进制长度
NUMERIC_PRECISION	int(11)	YES		NULL	数值精度
NUMERIC_SCALE	int(11)	YES		NULL	数值范围
COLUMN_TYPE	varchar(64)	YES		NULL	
COLUMN_COMMENT	varchar(255)	YES		NULL	列说明
DISABLE_INDEX	tinyint(1)	YES		0	是否禁用索引
DISABLE_DETAIL	tinyint(1)	YES		0	
IS_PRIMARY KEY	tinyint(1)	YES		0	是否是主键
IS_PRESORT	tinyint(1)	YES		NULL	
PRESORT_ORDER	smallint(6)	YES		NULL	
IS_VIRTUAL	tinyint(1)	YES		0	
IS_MULTIVALUED	tinyint(1)	YES		NULL	是否是多值列
DELIMITER	varchar(64)	YES		NULL	分隔符
IS_DELETED	tinyint(1)	YES		0	是否被删除

INDEXES	varchar(255)	YES		NULL	索引
CREATE_TIME	timestamp	NO		CURRENT_TIMESTAMP	创建时间
UPDATE_TIME	timestamp	NO		0000-00-0000:00:00	更新时间

- 如果希望了解某一个表包含的所有列信息，可以根据 TABLE_CATALOG , TABLE_GROUP 和 TABLE_SCHEMA 查询需要的列信息。

1 逻辑运算符

运算符	描述	例子
AND	a和b两者都为true则返回True	a AND b
OR	a或b有一个为true就返回True	a OR b
NOT	a为false则返回True	NOT a

Null值对逻辑运算符的影响

当运算符 AND 的输入中包含 FALSE 时必然返回 FALSE， 否则如果输入中包含了 NULL ，则会返回 NULL:

```
SELECT CAST(null AS boolean) AND true; -- null

SELECT CAST(null AS boolean) AND false; -- false

SELECT CAST(null AS boolean) AND CAST(null AS boolean); -- null
```

当运算符 OR 的输入中包含 TRUE 时必然返回 TRUE， 否则如果输入中包含了 NULL，则会返回 NULL:

```
SELECT CAST(null AS boolean) OR CAST(null AS boolean); -- null

SELECT CAST(null AS boolean) OR false; -- null

SELECT CAST(null AS boolean) OR true; -- true
```

下表详细列举了 NULL 参与 AND 和 OR 逻辑运算的结果：

a	b	a AND b	a OR b
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE

FALSE	NULL	FALSE	NULL
NULL	TRUE	NULL	TRUE
NULL	FALSE	FALSE	NULL
NULL	NULL	NULL	NULL

NOT 运算符在 NULL 上的操作实例如下:

```
SELECT NOT CAST(null AS boolean); -- null
```

下表详细列举了 NULL 参与 NOT 逻辑运算的结果：

a	NOT a
TRUE	FALSE
FALSE	TRUE
NULL	NULL

2 比较运算符

运算符	描述
<	小于
>	大于
<=	小于等于
>=	大于等于
=	等于
<>	不等于
!=	不等于 (非标准语法)

区间运算符: BETWEEN

BETWEEN 运算符用来判断值是否在区间里， 使用语法固定为 value BETWEEN min AND max:

```
SELECT 3 BETWEEN 2 AND 6;
```

上例中的查询语句等同于:

```
SELECT 3 >= 2 AND 3 <= 6;
```

判断值是否在区间外可以使用 NOT BETWEEN:

```
SELECT 3 NOT BETWEEN 2 AND 6;
```

上例中的查询语句等同于:

```
SELECT 3 < 2 OR 3 > 6;
```

在 BETWEEN or NOT BETWEEN 运算符中出现 NULL 都会使结果为 NULL:

```
SELECT NULL BETWEEN 2 AND 4; -- null

SELECT 2 BETWEEN NULL AND 6; -- null
```

BETWEEN and NOT BETWEEN 运算符同样可以作用于字符串型数据判断:

```
SELECT 'Paul' BETWEEN 'John' AND 'Ringo'; -- true
```

必须注意的是，BETWEEN and NOT BETWEEN 运算符的输入数据类型必须相同。例如，判断 ‘John’ is between 2.3 and 35.2 会使数据库报错。

IS NULL 和 IS NOT NULL

IS NULL and IS NOT NULL 运算符用于测试值是否为 NULL (未定义)。这两个运算符对所有数据类型有效。

用 IS NULL 判断 NULL 值会返回 TRUE:

```
select NULL IS NULL; -- true
```

但是其他任务非 NULL 值都会返回 FALSE:

```
SELECT 3.0 IS NULL; -- false
```

IS DISTINCT FROM and IS NOT DISTINCT FROM

在SQL语义中 NULL 代表一个未知的值，所以任何比较运算符在接收 NULL 输入时都会返回 NULL。 IS DISTINCT FROM 和 IS NOT DISTINCT FROM 运算符会把 NULL 当成特殊的值， 当这两个运算符的输入中包含 NULL 时，它们仍然会返回True或者False:

```
SELECT NULL IS DISTINCT FROM NULL; -- false

SELECT NULL IS NOT DISTINCT FROM NULL; -- true
```

在上面的例子中， 两个 NULL 值被认为是重复的。当你的比较运算数据中可能包含 NULL 值时，你可以使用这两个运算符来确保获得 TRUE 或者 FALSE 结果。

下表详细列举了 NULL 参与比较运算符时产生的结果：

a	b	a = b	a <> b	a DISTINCT b	a NOT DISTINCT b
1	1	TRUE	FALSE	FALSE	TRUE
1	2	FALSE	TRUE	TRUE	FALSE
1	NULL	NULL	NULL	TRUE	FALSE
NULL	NULL	NULL	NULL	FALSE	TRUE

GREATEST and LEAST

这两个函数不在标准的SQL语法中，但都是很常见的扩展。 和其他比较运算符一样，当函数的输入中出现 NULL 时函数返回结果为 NULL。 注意！在一些其他数据库，例如 PostgreSQL 中，只有当输入全部为 NULL 时才返回 NULL。

比较函数支持以下数据类型: DOUBLE, BIGINT, VARCHAR, TIMESTAMP, TIMESTAMP WITH TIME ZONE, DATE

greatest(value1, value2) → [same as input]

返回参数中的最大值。

least(value1, value2) → [same as input]

返回参数中的最小值。

比较运算符支持的量词: ALL, ANY and SOME

ALL, ANY and SOME 量词可以和比较运算符结合使用:

```
expression operator quantifier ( subquery )
```

例如:

```
SELECT 'hello' = ANY (VALUES 'hello', 'world'); -- true

SELECT 21 < ALL (VALUES 19, 20, 21); -- false

SELECT 42 >= SOME (SELECT 41 UNION ALL SELECT 42 UNION ALL SELECT 43); -- true
```

下表列举了比较运算符和量词组合的一些含义:

表达式	含义
A = ALL (...)	返回 true 当 A 等于所有匹配数据时.
A <> ALL (...)	返回 true 当 A 不等于所有匹配数据时.
A < ALL (...)	返回 true 当 A 小于所有匹配数据时.

A = ANY (...)	返回 true 当 A 等于任意某个匹配数据. 语义和 A IN (...) 相同.
A <> ANY (...)	返回 true 当 A 不等于任意某个匹配数据.
A < ANY (...)	返回 true 当 A 小于任意某个匹配数据.

ANY 和 SOME 含义相同，可以互换使用。

3 字符串运算符

运算符 || 完成字符串连接操作.

4 数学运算符

运算符	描述
+	加
-	减
*	乘
/	除 (整形除法会截断)
%	模数 (余数)

5 日期时间运算符

运算符	示例	结果
+	date '2012-08-08' + interval '2' day	2012-08-10
+	time '01:00' + interval '3' hour	04:00:00.000
+	timestamp '2012-08-08 01:00' + interval '29' hour	2012-08-09 06:00:00.000
+	timestamp '2012-10-31 01:00' + interval '1' month	2012-11-30 01:00:00.000
+	interval '2' day + interval '3' hour	2 03:00:00.000
+	interval '3' year + interval '5' month	3-5
-	date '2012-08-08' - interval '2' day	2012-08-06
-	time '01:00' - interval '3' hour	22:00:00.000

-	timestamp '2012-08-08 01:00' - interval '29' hour	2012-08-06 20:00:00.000
-	timestamp '2012-10-31 01:00' - interval '1' month	2012-09-30 01:00:00.000
-	interval '2' day - interval '3' hour	1 21:00:00.000
-	interval '3' year - interval '5' month	2-7

6 数组运算符

Subscript Operator: []

[] 操作符用来获取数组的某个元素:

```
SELECT my_array[1] AS first_element
```

Concatenation Operator: ||

|| 操作符可以将相同类型的数组或元素连接:

```
SELECT ARRAY [1] || ARRAY [2]; -- [1, 2]
SELECT ARRAY [1] || 2; -- [1, 2]
SELECT 2 || ARRAY [1]; -- [2, 1]
```

7 MAP运算符

Subscript Operator: []

[] 运算符用于取出map中指定键的值:

```
SELECT name_to_age_map['Bob'] AS bob_age;
```

附录三 系统函数

CASE

标准SQL中 CASE 表达式有两种结构。 简单的 CASE 表达式会从左到右依次查找 value 直到找到和 expression 相等的:

```
CASE expression
WHEN value THEN result
[ WHEN ... ]
[ ELSE result ]
END
```

找到相等的 value 后会返回对应的 result 结果。 如果没有找到相等的 value，则会返回 ELSE 语句的 result 结果。 例如:

```
SELECT a,
CASE a
WHEN 1 THEN 'one'
WHEN 2 THEN 'two'
ELSE 'many'
END
```

高级的 CASE 表达式会从左到右依次计算 condition 直到第一个为 TRUE 的 condition，并返回对应的 result 结果。

```
CASE
WHEN condition THEN result
[ WHEN ... ]
[ ELSE result ]
END
```

如果没有为 True 的 condition，则 ELSE 子句的 result 会返回。 例如:

```
SELECT a, b,
CASE
WHEN a = 1 THEN 'aaa'
WHEN b = 2 THEN 'bbb'
ELSE 'ccc'
END
```

IF

IF 函数是和以下 CASE 表达式效果相同的语言结构:

```
CASE
WHEN condition THEN true_value
[ ELSE false_value ]
END
```

if(condition, true_value)

计算返回 true_value 如果 condition 为 TRUE， 否则返回 NULL。

if(condition, true_value, false_value)计算返回 true_value 如果 condition 为 TRUE， 否则计算返回 false_value。

COALESCE

coalesce(value[, ...])

返回第一个非 NULL 的 value。 和 CASE 表达式类似, 参数仅在需要的时候计算。

IFNULL

If expr1 is not NULL, IFNULL() returns expr1; otherwise it returns expr2.

支持的语法：

IFNULL(expr1,expr2)

例子：

```
SELECT IFNULL(1,0);

SELECT IFNULL(NULL,10);
```

NULLIF

nullif(value1, value2)返回 NULL 如果 value1 等于 value2， 否则返回 value1。

支持的语法：

NULLIF(expr1,expr2)

例子：

```
SELECT NULLIF(1,1);

SELECT NULLIF(1,2);
```

TRY

try(expression)

计算返回表达式 expression 如果表达式计算遇到错误则返回 NULL。

当你不希望查询抛出异常的时候，可以使用 TRY 函数屏蔽异常。TRY 函数在遇到异常的时候会返回 NULL， 如果希望在查询出错的时候返回默认值，可以 使用 COALESCE 函数指定默认值。

TRY 函数可以处理以下错误:

- 除0
- Invalid cast or function argument
- 数值大小超出范围

例子

源数据表包含非法数据:

```
SELECT * FROM shipping;


 origin_state | origin_zip | packages | total_cost
--------------+------------+----------+------------
California | 94131 | 25 | 100
California | P332a | 5 | 72
California | 94025 | 0 | 155
New Jersey | 08544 | 225 | 490
(4 rows)
```

不用 TRY 函数导致查询失败:

```
SELECT CAST(origin_zip AS BIGINT) FROM shipping;


Query failed: Can not cast 'P332a' to BIGINT
```

TRY 函数返回 NULL:

```
SELECT TRY(CAST(origin_zip AS BIGINT)) FROM shipping;


origin_zip
------------
94131
NULL
94025
08544
(4 rows)
```

不用 TRY 函数导致查询失败:

```
SELECT total_cost / packages AS per_package FROM shipping;


Query failed: / by zero
```

COALESCE 嵌套 TRY 函数返回默认值:

```
SELECT COALESCE(TRY(total_cost / packages), 0) AS per_package FROM shipping;


 per_package
-------------
4
14
0
19
(4 rows)
```

Control Flow 函数MySQL兼容性(ADS 已经支持的MySQL函数)

COALESCE

Returns the first non-NULL value in the list, or NULL if there are no non-NULL values.

支持的语法：

COALESCE(value,...)

例子：

```
SELECT COALESCE(NULL,1);

SELECT COALESCE(NULL,NULL,NULL);
```

GREATEST

With two or more arguments, returns the largest (maximum-valued) argument.

支持的语法：

GREATEST(value1,value2,...)

例子：

```
SELECT GREATEST(2,0);

SELECT GREATEST(34.0,3.0,5.0,767.0);

SELECT GREATEST('B','A','C');
```

LEAST

With two or more arguments, returns the smallest (minimum-valued) argument.

支持的语法：

LEAST(value1,value2,...)

例子：

```
SELECT LEAST(2,0);

SELECT LEAST(34.0,3.0,5.0,767.0);

SELECT LEAST('B','A','C');
```

Control Flow 函数Oracle兼容性(ADS 已经支持的Oracle函数)

NVL2

NVL2 lets you determine the value returned by a query based on whether a specified expression is null or not null. If expr1 is not null, then NVL2 returns expr2. If expr1 is null, then NVL2 returns expr3.

支持的语法：

NVL2(expr1, expr2, expr3)

例子：

```
SELECT NVL2(1, 2, 3);

SELECT NVL2(NULL, 2, 3);
```

DECODE

DECODE compares expr to each search value one by one. If expr is equal to a search, then AnalyticDB returns the corresponding result.

支持的语法：

DECODE(expr, search, result, default)

例子：

```
SELECT DECODE(1, 1, '1A', 2, '2A', '3A');

SELECT DECODE(2, 1, '1A', 2, '2A', '3A');
```

Note:

这些函数只处理UTF-8编码的可见字符，对输入字符串不作编码格式检查，非法的字符或字符串会导致查询结果异常。

lower() 和 upper() 函数只处理英文字母的大小写转换，这对中文等非拉丁语言会返回错误结果

。

chr(n) → varchar

返回下标为 n 位置的字符。

concat(string1, ..., stringN) → varchar

字符串连接操作，返回 string1 , string2 , ... , stringN 字符串连接。 此功能与标准SQL的连接运算符 (||)功能相同。

length(string) → bigint

返回字符串 string 长度。

lower(string) → varchar

转换字符串 string 为小写。

lpad(string, size, padstring) → varchar

将字符串 string 左边拼接 padstring 直到长度达到 size 并返回填充后的字符串。如果 size 比 string 长度小，则截断。 size 不能为负数， padstring 必须非空。

ltrim(string) → varchar

删除字符串所有前导空格。

replace(string, search) → varchar

删除字符串 string 中的所有子串 search 。

replace(string, search, replace) → varchar

将字符串 string 中所有子串 search 替换为 replace。

Left/Right(str,len)

返回从字符串str 开始的len 最左/右字符

reverse(string) → varchar

将字符串 string 逆序后返回。

rpad(string, size, padstring) → varchar

将字符串 string 右边拼接 padstring 直到长度达到 size ,返回填充后的字符串。如果 size 比 string 长度小，则截断。 size 不能为负数， padstring 必须非空。

rtrim(string) → varchar

删除字符串 string 右边所有空格。

split(string, delimiter) → array<varchar>

将字符串 string 按分隔符 delimiter 进行分隔，并返回数组。

split(string, delimiter, limit) → array<varchar>

将字符串 string 按分隔符 delimiter 分隔，并返回按 limit 大小限制的数组。数组中的最后一个元素包含字符串中的所有剩余内容。 limit 必须是正数。

split_part(string, delimiter, index) → varchar

将字符串 string 按分隔符 delimiter 分隔，并返回分隔后数组下标为 index 的子串。 index 以 1 开头，如果大于字段数则返回null。

split_to_map(string, entryDelimiter, keyValueDelimiter) → map<varchar, varchar>

通过 entryDelimiter 和 keyValueDelimiter 拆分字符串并返回map。 entryDelimiter 将字符串分解为key-value对， keyValueDelimiter 将每对分隔成key、value。

strpos(string, substring) → bigint

返回字符串中子字符串的第一次出现的起始位置。位置以 1 开始 ，如果未找到则返回 0 。

position(substring IN string) → bigint

返回字符串中子字符串的第一次出现的起始位置。位置以 1 开始，如果未找到则返回 0 。

substr(string, start) → varchar

返回 start 位置开始到字符串结束。位置从 1 开始。如果 start 为负数，则起始位置代表从字符串的末尾开始倒数。

substr(string, start, length) → varchar

返回 start 位置开始长度为 length 的子串，位置从 1 开始。如果 start 为负数，则起始位置代表从字符串的末尾开始倒数。

trim(string) → varchar

删除字符串 string 前后的空格。

upper(string) → varchar

转换字符串为大写

uuid()

返回一个字符串，在当前集群内保证唯一，算法参考mongodb的objectid实现

Unicode 函数

normalize(string) → varchar

用NFC规范化形式转换字符串。

normalize(string, form) → varchar

使用指定的规范化形式转换字符串。 form 必须是以下关键字之一:

Form	解释
NFD	规范分解
NFC	规范分解，其次是规范组合
NFKD	兼容性分解
NFKC	兼容性分解，其次是规范组合

Note:

- SQL标准函数有特殊的语法，需要将 form 指定为关键字，而不是字符串。

to_utf8(string) → varbinary

将字符串编码为UTF-8格式。

from_utf8(binary) → varchar

将二进制 binary 解码为UTF-8编码的字符串,无效的UTF-8字符被替换为Unicode字符 U+FFFD 。

from_utf8(binary, replace) → varchar

将二进制 binary 解码为UTF-8编码的字符串。无效的UTF-8字符替换为 replace 。替换字符串必须是单个字符或空格（无效字符会被删除）。

char2hexint(string) → varchar

返回字符串的UTF-16BE编码的十六进制表示形式.

index(string, substring) → bigint

同 strpos() function.

substring(string, start) → varchar

同 substr() function.

substring(string, start, length) → varchar

同 substr() function.

String 函数MySQL兼容性(ADS 已经支持的MySQL函数)

ASCII

Returns the numeric value of the leftmost character of the string str.

支持的语法：

ASCII(str)

例子：

```
SELECT ASCII('2');
SELECT ASCII(2);
SELECT ASCII('dx');
```

BIN

Returns a string representation of the binary value of N, where N is a longlong (BIGINT) number.

支持的语法：

BIN(N)

例子：

```
SELECT BIN(12);
```

CHAR

CHAR() interprets each argument N as an integer and returns a string consisting of the characters given by the code values of those integers. NULL values are skipped.

支持的语法：

CHAR(N,...)

例子：

```
SELECT CHAR(77,121,83,81,76);
```

CHAR_LENGTH

Returns the length of the string str, measured in characters. A multibyte character counts as a single character.

支持的语法：

CHAR_LENGTH(str)

例子：

```
SELECT CHAR_LENGTH('111');
```

CHARACTER_LENGTH

Returns the length of the string str, measured in characters. A multibyte character counts as a single

character.

支持的语法：

CHARACTER_LENGTH(str)

例子：

```
SELECT CHARACTER_LENGTH('111');
```

EXPORT_SET

Returns a string such that for every bit set in the value bits, you get an on string and for every bit not set in the value, you get an off string.

支持的语法：

EXPORT_SET(bits,on,off[,separator[,number_of_bits]])

例子：

```
SELECT EXPORT_SET(5,'Y','N',',',4);
SELECT EXPORT_SET(6,'1','0',',',10);
```

FIND_IN_SET

Returns a value in the range of 1 to N if the string str is in the string list strlist consisting of N substrings.

支持的语法：

FIND_IN_SET(str,strlist)

例子：

```
SELECT FIND_IN_SET('b','a,b,c,d');
```

FORMAT

Formats the number X to a format like ‘#,###,###.##’, rounded to D decimal places, and returns the result as a string.

支持的语法：

FORMAT(X,D[,locale])

例子：

```
SELECT FORMAT(12332.123456, 4);
```

```
SELECT FORMAT(12332.1, 4);
SELECT FORMAT(12332.2,0);
SELECT FORMAT(12332.2,2,'de_DE');
```

FROM_BASE64

Takes a string encoded with the base-64 encoded rules used by TO_BASE64() and returns the decoded result as a binary string.

支持的语法：

FROM_BASE64(str)

例子：

```
SELECT TO_BASE64('abc');
SELECT FROM_BASE64_MYSQL(TO_BASE64('abc'));
```

HEX

For a string argument str, HEX() returns a hexadecimal string representation of str where each byte of each character in str is converted to two hexadecimal digits.

支持的语法：

HEX(str) , HEX(N)

例子：

```
SELECT HEX('abc');
SELECT HEX(255);
SELECT UNHEX_MYSQL(HEX('abc'));
```

INSERT

Returns the string str, with the substring beginning at position pos and len characters long replaced by the string newstr.

支持的语法：

INSERT(str,pos,len,newstr)

例子：

```
SELECT PRESTO_INSERT('Quadratic', 3, 4, 'What');
SELECT PRESTO_INSERT('Quadratic', -1, 4, 'What');
SELECT PRESTO_INSERT('Quadratic', 3, 100, 'What');
```

INSTR

Returns the position of the first occurrence of substring substr in string str.

支持的语法：

INSTR(str,substr)

例子：

SELECT INSTR('foobarbar', 'bar'); SELECT INSTR('xbar', 'foobar');

LCASE

LCASE() is a synonym for LOWER().

支持的语法：

LCASE(str)

例子：

```
SELECT LCASE('FoOBAr');
SELECT Lower('FoOBAr');
```

Ucase

返回字符串的全大写

UPPER

UPPER() is a synonym for UCASE().

支持的语法：

UPPER(str)

例子：

```
SELECT UPPER('FoOBAr');
SELECT UCASE('FoOBAr');
```

LEFT

Returns the leftmost len characters from the string str, or NULL if any argument is NULL

支持的语法：

LEFT(str,len)

例子：

```
SELECT PRESTO_LEFT('foobarbar', 5);
```

LENGTH

Returns the length of the string str, measured in bytes. A multibyte character counts as multiple bytes.

支持的语法：

LENGTH(str)

例子：

```
SELECT LENGTH('text');
```

LOCATE

The first syntax returns the position of the first occurrence of substring substr in string str. The second syntax returns the position of the first occurrence of substring substr in string str, starting at position pos.

支持的语法：

LOCATE(substr,str) , LOCATE(substr,str,pos)

例子：

```
SELECT LOCATE('bar', 'foobarbar');
SELECT LOCATE('xbar', 'foobar');
SELECT LOCATE('xbar', 'foobar');
```

LPAD

Returns the string str, left-padded with the string padstr to a length of len characters. If str is longer than len, the return value is shortened to len characters.

支持的语法：

LPAD(str,len,padstr)

例子：

```
SELECT LPAD('hi',4,'??');
SELECT LPAD('hi',1,'??');
```

LTRIM

Returns the string str with leading space characters removed.

支持的语法：

LTRIM(str)

例子：

```
SELECT LTRIM('     foOBarBaR');
```

SUBSTRING

The forms without a len argument return a substring from string str starting at position pos. The forms with a len argument return a substring len characters long from string str, starting at position pos. The forms that use FROM are standard SQL syntax. It is also possible to use a negative value for pos. In this case, the beginning of the substring is pos characters from the end of the string, rather than the beginning. A negative value may be used for pos in any of the forms of this function. SUBSTR() is a synonym for SUBSTRING().

支持的语法：

SUBSTRING(str,pos) , SUBSTRING(str FROM pos) , SUBSTRING(str,pos,len) , SUBSTRING(str FROM pos FOR len)

例子：

```
SELECT SUBSTRING('Quadratically',5);
SELECT SUBSTRING('foobarbar' FROM 4);
SELECT SUBSTRING('Quadratically',5,6);
SELECT SUBSTRING('Sakila', -3);
SELECT SUBSTRING('Sakila', -5, 3);
SELECT SUBSTRING('Sakila' FROM -4 FOR 2);
SELECT SUBSTR('Quadratically',5);
SELECT SUBSTR('Quadratically',5,6);
SELECT SUBSTR('Sakila', -3);
SELECT SUBSTR('Sakila', -5, 3)
```

MID

MID(str,pos,len) is a synonym for SUBSTRING(str,pos,len).

支持的语法：

MID(str,pos,len)

例子：

```
SELECT mid('Quadratically',5,6);
```

OCT

Returns a string representation of the octal value of N, where N is a longlong (BIGINT) number.

支持的语法：

OCT(N)

例子：

```
SELECT OCT(12);
```

ORD

If the leftmost character of the string str is a multibyte character, returns the code for that character.

支持的语法：

ORD(str)

例子：

```
SELECT ORD('2');
```

REPEAT

Returns a string consisting of the string str repeated count times.

支持的语法：

REPEAT(str,count)

例子：

```
SELECT REPEAT('MySQL', 3);
```

REPLACE

Returns the string str with all occurrences of the string from_str replaced by the string to_str.

支持的语法：

REPLACE(str,from_str,to_str)

例子：

```
SELECT REPLACE('www.mysql.com', 'w', 'Ww');
```

REVERSE

Returns the string str with the order of the characters reversed.

支持的语法：

REVERSE(str)

例子：

```
SELECT REVERSE('abc');
```

RIGHT

Returns the rightmost len characters from the string str, or NULL if any argument is NULL.

支持的语法：

RIGHT(str,len)

例子：

```
SELECT PRESTO_RIGHT('foobarbar', 4);
```

RPAD

Returns the string str, right-padded with the string padstr to a length of len characters. If str is longer than len, the return value is shortened to len characters.

支持的语法：

RPAD(str,len,padstr)

例子：

```
SELECT RPAD('hi',5,'?');
SELECT RPAD('hi',1,'?');
```

RTRIM

Returns the string str with trailing space characters removed.

支持的语法：

RTRIM(str)

例子：

```
SELECT RTRIM('barbar   ');
```

SOUNDEX

Returns a soundex string from str.

支持的语法：

SOUNDEX(str)

例子：

```
SELECT SOUNDEX('Hello');
SELECT SOUNDEX('Quadratically');
```

SPACE

Returns a string consisting of N space characters.

支持的语法：

SPACE(N)

例子：

```
SELECT SPACE(6);
```

SUBSTRING_INDEX

Returns the substring from string str before count occurrences of the delimiter delim. If count is positive, everything to the left of the final delimiter (counting from the left) is returned. If count is negative, everything to the right of the final delimiter (counting from the right) is returned. SUBSTRING_INDEX() performs a case-sensitive match when searching for delim.

支持的语法：

SUBSTRING_INDEX(str,delim,count)

例子：

```
SELECT SUBSTRING_INDEX('www.mysql.com', '.', 2);
SELECT SUBSTRING_INDEX('www.mysql.com', '.', -2);
```

TRIM

Returns the string str with all remstr prefixes or suffixes removed. If none of the specifiers BOTH, LEADING, or TRAILING is given, BOTH is assumed. remstr is optional and, if not specified, spaces are removed.

支持的语法：

TRIM(str)

例子：

```
SELECT TRIM('  bar   ');
```

ip2region

Returns the region of the given ip address. You can specify the level as ‘COUNTRY’, ‘AREA’, ‘PROVINCE’, ‘CITY’, ‘ISP’.

支持的语法：

ip2region(ip_address, level)

例子：

```
SELECT ip2region('101.105.35.57', 'COUNTRY');
SELECT ip2region('101.105.35.57', 'Province');
SELECT ip2region('101.105.35.57', 'CITY');
```

String函数Oracle兼容性(ADS 已经支持的Oracle函数)

INITCAP

INITCAP returns char, with the first letter of each word in uppercase, all other letters in lowercase.

支持的语法：

INITCAP(char)

例子：

```
SELECT INITCAP('the soap')
SELECT INITCAP('the,soap')
```

INSTR

The INSTR functions search string for substring. The function returns an integer indicating the position of the character in string that is the first character of this occurrence.

支持的语法：

INSTR(string, substring) | INSTR(string, substring, position) | INSTR(string, substring, position, occurence)

例子：

```
SELECT INSTR('CORPORATE FLOOR','OR');
SELECT INSTR('CORPORATE FLOOR','OR', 3);
```

```
SELECT INSTR('CORPORATE FLOOR','OR', 3, 2);
```

LPAD

The LPAD function returns an expression, left-padded to a specified length with the specified characters; or, when the expression to be padded is longer than the length specified after padding, only that portion of the expression that fits into the specified length.

支持的语法：

LPAD (text-exp , length) | LPAD (text-exp , length, pad-exp)

例子：

```
SELECT LPAD('HELLO', 5);
SELECT LPAD('HELLO', 5, 'A');
```

LTRIM

LTRIM removes from the left end of char all of the characters contained in set.

支持的语法：

LTRIM(char) | LTRIM(char, set)

例子：

```
SELECT LTRIM('    WWW.TTTT');
SELECT LTRIM('WWW.TTTT','W');
```

REGEXP_COUNT

REGEXP_COUNT complements the functionality of the REGEXP_INSTR function by returning the number of times a pattern occurs in a source string.

支持的语法：

REGEXP_COUNT(source_char, pattern)

例子：

```
SELECT REGEXP_COUNT('rat cat\nbat dog', '.at');
```

REGEXP_SUBSTR

REGEXP_SUBSTR extends the functionality of the SUBSTR function by letting you search a string for a regular expression pattern.

支持的语法：

REGEXP_SUBSTR(source_char, pattern) | REGEXP_SUBSTR(source_char, pattern, position)

例子：

```
REGEXP_SUBSTR('Hello world bye', '\\b[a-z]([a-z]*)');
REGEXP_SUBSTR('Hello world bye', '\\b[a-z]([a-z]*)', 1);
```

REPLACE

REPLACE returns char with every occurrence of search_string replaced with replacement_string.

支持的语法：

REPLACE(char, search_string, replacement_string) | REPLACE(char, search_string)

例子：

```
SELECT REPLACE('JACK and JUE','J','BL');
SELECT REPLACE('JACK and JUE','J');
```

RPAD

RPAD returns expr1, right-padded to length n characters with expr2, replicated as many times as necessary.

支持的语法：

RPAD (expr1, n, expr2) | RPAD (expr1, n)

例子：

```
SELECT RPAD('HELLO', 5);
SELECT RPAD('HELLO', 5, 'A');
```

RTRIM

RTRIM removes from the right end of char all of the characters that appear in set.

支持的语法：

RTRIM(char) | RTRIM(char, set)

例子：

```
SELECT RTRIM('WWW.TTTT     ');
SELECT RTRIM('WWW.TTTT', 'T');
```

TRANSLATE

TRANSLATE returns expr with all occurrences of each character in from_string replaced by its corresponding character in to_string.

支持的语法：

TRANSLATE(expr, from_string, to_string)

例子：

```
SELECT TRANSLATE('acbd','ab','AB');
SELECT TRANSLATE('acbd','abc','A');
SELECT TRANSLATE('acbd','abc','');
```

abs(x) → [same as input]

返回 x 的绝对值.

cbrt(x) → double

返回 x 的立方根.

ceil(x) → [same as input]

ceiling() 的同名方法.

ceiling(x) → [same as input]

返回 x 的向上取整的数值.

cosine_similarity(x, y) → double

返回稀疏向量 x 和 y 之间的余弦相似度:

```
SELECT cosine_similarity(MAP(ARRAY['a'], ARRAY[1.0]), MAP(ARRAY['a'], ARRAY[2.0])); -- 1.0
```

degrees(x) → double

将角度 x 以弧度转换为度.

e() → double

返回欧拉常量.

exp(x) → double

返回 x 的欧拉常量次幂.

floor(x) → [same as input]

返回 x 向下取整的最近整数值.

from_base(string, radix) → bigint

返回 radix 进制的字符串 string 代表的数值:

```
SELECT from_base('0110', 2); -- 2
SELECT from_base('0110', 8); -- 456
SELECT from_base('00a0', 16); -- 160
```

ln(x) → double

返回 x 的自然对数.

log2(x) → double

返回 x 以2为底的对数.

log10(x) → double

返回 x 以10为底的对数.

log(x, b) → double

返回 x 以 b 为底的对数.

mod(n, m) → [same as input]

返回 n 除 m 的模数(余数).

pi() → double

返回常量Pi.

pow(x, p) → double

power() 的同名方法.

power(x, p) → double

返回 x 的 p 次幂.

radians(x) → double

将角度 x 以度为单位转换为弧度.

rand() → double

random() 的同名方法.

random() → double

返回 0.0 <= x < 1.0 范围内的伪随机数.

random(n) → [same as input]

返回 0 <= x < n 范围内的伪随机数.

round(x) → [same as input]

返回 x 四舍五入后的最近的整数值.

round(x, d) → [same as input]

返回 x 四舍五入到 d 位小数位的值.

sign(x) → [same as input]

x 的正负号函数, 即:

x 为0, 返回0,

x 为正, 返回1,

x 为负, 返回-1.

对于double类型参数, 则:

x 为NaN, 返回NaN,

x 为正无穷, 返回1,

x 为负无穷, 返回-1.

sqrt(x) → double

返回 x 的平方根.

to_base(x, radix) → varchar

返回 x 的 radix 进制表示的字符串.

truncate(x) → double

舍弃 x 的小数位,返回整数值.

width_bucket(x, bound1, bound2, n) → bigint

bound1 到 bound2 范围等长划分成n个桶, 返回x在其中的桶号.

width_bucket(x, bins) → bigint

返回 x 在数组 bins 描述的分桶中的桶号. 参数 bins 必须是一个double类型的数组, 并且嘉定是按照升序排序的
.

三角函数

所有三角函数都是以弧度表示. 单位转换请参考 degrees() 和 radians().

acos(x) → double

返回 x 的反余弦.

asin(x) → double

返回 x 的反正弦.

atan(x) → double

返回 x 的反正切.

atan2(y, x) → double

返回 y / x 的反正切.

cos(x) → double

返回 x 的余弦值.

cosh(x) → double

返回 x 的双曲余弦值.

sin(x) → double

返回 x 的正弦值.

tan(x) → double

返回 x 的正切值.

tanh(x) → double

返回 x 的双曲正切.

浮点函数

infinity() → double

返回表示正无穷大的常量.

is_finite(x) → boolean

判定 x 是否有限.

is_infinite(x) → boolean

判定 x 是否无限.

is_nan(x) → boolean

判定 x 是非法数值.

nan() → double

返回代表非数值的常量值.

Math 函数MySQL兼容性(ADS 已经支持的MySQL函数)

ABS

Returns the absolute value of X.

支持的语法：

ABS(X)

例子：

```
SELECT ABS(2);
SELECT ABS(-32);
```

ACOS

Returns the arc cosine of X, that is, the value whose cosine is X. Returns NULL if X is not in the range -1 to 1.

支持的语法：

ACOS(X)

例子：

```
SELECT ACOS(1);
SELECT ACOS(1.0001);
SELECT ACOS(0);
```

ASIN

Returns the arc sine of X, that is, the value whose sine is X. Returns NULL if X is not in the range -1 to 1.

支持的语法：

ASIN(X)

例子：

```
SELECT ASIN(0.2);
```

ATAN

Returns the arc tangent of X, that is, the value whose tangent is X.

支持的语法：

ATAN(X) , ATAN(Y,X)

例子：

```
SELECT ATAN(2);
SELECT ATAN(-2);
SELECT ATAN2(-2,2);
SELECT ATAN2(PI(),0);
```

ATAN2

Returns the arc tangent of the two variables X and Y. It is similar to calculating the arc tangent of Y / X, except that the signs of both arguments are used to determine the quadrant of the result.

支持的语法：

ATAN2(Y,X)

例子：

```
SELECT ATAN(2);
SELECT ATAN(-2);
SELECT ATAN2(-2,2);
SELECT ATAN2(PI(),0);
```

CEILING

Returns the smallest integer value not less than X.

支持的语法：

CEILING(X)

例子：

```
SELECT CEILING(1.23);
SELECT CEILING(-1.23);
```

CEIL

Returns the smallest integer value not less than X.

支持的语法：

CEIL(X)

例子：

```
SELECT CEIL(1.23);
```

```
SELECT CEIL(-1.23);
```

COS

Returns the cosine of X, where X is given in radians.

支持的语法：

COS(X)

例子：

```
SELECT COS(PI());
```

COT

Returns the cotangent of X.

支持的语法：

COT(X)

例子：

```
SELECT COT(12);
SELECT COT(0);
```

CRC32

Computes a cyclic redundancy check value and returns a 32-bit unsigned value. The result is NULL if the argument is NULL.

支持的语法：

CRC32(expr)

例子：

```
SELECT CRC32('MySQL');
SELECT CRC32('mysql');
```

DEGREES

Returns the argument X, converted from radians to degrees.

支持的语法：

DEGREES(X)

例子：

```
SELECT DEGREES(PI());
SELECT DEGREES(PI() / 2);
```

EXP

Returns the value of e (the base of natural logarithms) raised to the power of X. The inverse of this function is LOG()

支持的语法：

EXP(X)

例子：

```
SELECT EXP(2);
SELECT EXP(-2);
SELECT EXP(0);
```

FLOOR

Returns the largest integer value not greater than X.

支持的语法：

FLOOR(X)

例子：

```
SELECT FLOOR(1.23);
SELECT FLOOR(-1.23);
```

LN

Returns the natural logarithm of X; that is, the base-e logarithm of X.

支持的语法：

LN(X)

例子：

```
SELECT LN(2);
SELECT LN(-2);
```

LOG, LOG2, LOG10

If called with one parameter, this function returns the natural logarithm of X.

支持的语法：

LOG(X) , LOG2(X) , LOG10(X)

例子：

```
SELECT LOG(2);
SELECT LOG(-2);
SELECT LOG(2,65536);
SELECT LOG(10,100);
SELECT LOG(1,100);
SELECT LOG2(65536);
SELECT LOG2(-100);
SELECT LOG10(2);
SELECT LOG10(100);
SELECT LOG10(-100);
```

MOD

Modulo operation. Returns the remainder of N divided by M.

支持的语法：

MOD(N,M)

例子：

```
SELECT MOD(234, 10);
SELECT 253 % 7;
SELECT MOD(34.5,3);
```

POW

Returns the value of X raised to the power of Y.

支持的语法：

POW(X,Y)

例子：

```
SELECT POW(2,2);
SELECT POW(2,-2);
```

POWER

Returns the value of X raised to the power of Y.

支持的语法：

POWER(X,Y)

例子：

```
SELECT POWER(2,2);
SELECT POWER(2,-2);
```

RADIANS

Returns the argument X, converted from degrees to radians.

支持的语法：

RADIANS(X)

例子：

```
SELECT RADIANS(90);
```

RAND

Returns a random floating-point value v in the range 0 <= v < 1.0.

支持的语法：

RAND([N])

例子：

```
SELECT RAND();
SELECT RAND(2);
```

ROUND

Rounds the argument X to D decimal places. The rounding algorithm depends on the data type of X. D defaults to 0 if not specified.

支持的语法：

ROUND(X) , ROUND(X,D)

例子：

```
SELECT ROUND(-1.23);
SELECT ROUND(23.298, -1);
SELECT ROUND(1.298, 0);
SELECT ROUND(1.298, 1);
```

SIGN

Returns the sign of the argument as -1, 0, or 1, depending on whether X is negative, zero, or positive.

支持的语法：

SIGN(X)

例子：

```
SELECT SIGN(-32);
SELECT SIGN(0);
SELECT SIGN(234);
```

SIN

Returns the sine of X, where X is given in radians.

支持的语法：

SIN(X)

例子：

```
SELECT SIN(PI());
SELECT ROUND(SIN(PI()));
```

SQRT

Returns the square root of a nonnegative number X.

支持的语法：

SQRT(X)

例子：

```
SELECT SQRT(4);
SELECT SQRT(20);
SELECT SQRT(-16);
```

TAN

Returns the tangent of X, where X is given in radians.

支持的语法：

TAN(X)

例子：

```
SELECT TAN(PI());
SELECT TAN(PI()+1);
```

TRUNCATE

Returns the number X, truncated to D decimal places. If D is 0, the result has no decimal point or fractional part.

支持的语法：

TRUNCATE(X,D)

例子：

```
SELECT TRUNCATE(1.223,1);
SELECT TRUNCATE(1.999,1);
SELECT TRUNCATE(1.999,0);
SELECT TRUNCATE(-1.999,1);
SELECT TRUNCATE(122,2);
```

Math函数Oracle兼容性(ADS 已经支持的Oracle函数)

REMAINDER

REMAINDER returns the remainder of n2 divided by n1.

支持的语法：

REMAINDER(n1, n2)

例子：

```
SELECT REMAINDER(3.5, 2);
SELECT REMAINDER(11, 4);
```

TANH

TANH returns the hyperbolic tangent of n.

支持的语法：

TANH(n)

例子：

```
SELECT TANH(0.5);
```

BITAND

BITAND computes an AND operation on the bits of expr1 and expr2, both of which must resolve to nonnegative integers, and returns an integer.

支持的语法：

BITAND(expr1, expr2)

例子：

```
```

```
SELECT BITAND(5, 6);
```

WIDTH_BUCKET

Returns the bin number of x in an equi-width histogram with the specified bound1 and bound2 bounds and n number of buckets.

支持的语法：

WIDTH_BUCKET(x, bound1, bound2, n)

例子：

```
SELECT  WIDTH_BUCKET(5,3,4,5);
```

时区转换

运算符：AT TIME ZONE，用于设置一个时间戳的时区:

```
SELECT timestamp ‘2012-10-31 01:00 UTC’; 2012-10-31 01:00:00.000 UTC

SELECT timestamp ‘2012-10-31 01:00 UTC’ AT TIME ZONE ‘America/Los_Angeles’; 2012-10-30 18:00:00.000 America/Los_Angeles
```

日期时间函数

current_date -> date

返回查询开始时的当前日期.

current_time -> time with time zone

返回查询开始时的当前时间.

current_timestamp -> timestamp with time zone

返回查询开始时的当前时间戳.

current_timezone() → varchar

以IANA（例如，America / Los_Angeles）定义的格式返回当前时区，或以UTC的固定偏移量（例如+08：35）返回当前时区

from_iso8601_timestamp(string) → timestamp with time zone

将ISO 8601格式化的字符串解析为具有时区的时间戳

from_iso8601_date(string) → date

将ISO 8601格式的字符串解析为日期

from_unixtime(unixtime) → timestamp

返回unixtime时间戳

from_unixtime(unixtime, string) → timestamp with time zone

返回指定时区的unixtime时间戳

from_unixtime(unixtime, hours, minutes) → timestamp with time zone

返回为hours和minutes对应时区的unixtime时间戳

localtime -> time

返回查询开始时的当前时间

localtimestamp -> timestamp

返回查询开始时的当前时间戳

now() → timestamp with time zone

这是current_timestam的另一种表达

to_iso8601(x) → varchar

将x格式化为ISO 8601字符串。 x可以是date, timestamp,或带时区的timestamp

to_unixtime(timestamp) → double

转换为unix时间戳

Note

下列SQL标准的函数不使用括号：

- current_date
- current_time
- current_timestamp
- localtime
- localtimestamp

截取函数

函数date_trunc支持如下单位：

单位	示例结果
second	2001-08-22 03:04:05.000
minute	2001-08-22 03:04:00.000
hour	2001-08-22 03:00:00.000
day	2001-08-22 00:00:00.000

week	2001-08-20 00:00:00.000
month	2001-08-01 00:00:00.000
quarter	2001-07-01 00:00:00.000
year	2001-01-01 00:00:00.000

上面的例子使用时间戳： 2001-08-22 03:04:05.321 作为输入。

date_trunc(unit, x) → [same as input]返回x截取到单位unit之后的值

间隔函数

本章中的函数支持如下所列的间隔单位：

单位	描述
millisecond	Milliseconds
second	Seconds
minute	Minutes
hour	Hours
day	Days
week	Weeks
month	Months
quarter	Quarters of a year
year	Years

date_add(unit, value, timestamp) → [same as input]

在timestamp的基础上加上value个unit。如果想要执行相减的操作，可以通过将value赋值为负数来完成

date_diff(unit, timestamp1, timestamp2) → bigint

返回 timestamp2 - timestamp1之后的值，该值的表示单位是unit

MySQL日期函数

在这一章节使用与MySQLdate_parse和str_to_date方法兼容的格式化字符串。下面的表格是基于MySQL手册列出的，描述了各种格式化描述符：

分类符	说明
%a	Abbreviated weekday name (Sun .. Sat)
%b	Abbreviated month name (Jan .. Dec)
%c	Month, numeric (0 .. 12)

%D	Day of the month with English suffix (0th, 1st, 2nd, 3rd, ...)
%d	Day of the month, numeric (00 .. 31)
%e	Day of the month, numeric (0 .. 31)
%f	Fraction of second (6 digits for printing: 000000 .. 999000; 1 - 9 digits for parsing: 0 .. 999999999)
%H	Hour (00 .. 23)
%h	Hour (01 .. 12)
%I	Hour (01 .. 12)
%i	Minutes, numeric (00 .. 59)
%j	Day of year (001 .. 366)
%k	Hour (0 .. 23)
%l	Hour (1 .. 12)
%M	Month name (January .. December)
%m	Month, numeric (00 .. 12)
%p	AM or PM
%r	Time, 12-hour (hh:mm:ss followed by AM or PM)
%S	Seconds (00 .. 59)
%s	Seconds (00 .. 59)
%T	Time, 24-hour (hh:mm:ss)
%U	Week (00 .. 53), where Sunday is the first day of the week
%u	Week (00 .. 53), where Monday is the first day of the week
%V	Week (01 .. 53), where Sunday is the first day of the week; used with %X
%v	Week (01 .. 53), where Monday is the first day of the week; used with %x
%W	Weekday name (Sunday .. Saturday)
%w	Day of the week (0 .. 6), where Sunday is the first day of the week
%X	Year for the week where Sunday is the first day of the week, numeric, four digits; used with %V
%x	Year for the week, where Monday is the first day of the week, numeric, four digits; used with %v

%Y	Year, numeric, four digits
%y	Year, numeric (two digits)
%%	A literal % character
%x	x, for any x not listed above

[1] Timestamp被截断为毫秒。

[2] 解析时，两位数的年份格式假定为1970.2069，因此1970年将会产生“70”，而“69”将产生2069年。

[3] 下列说明符尚不支持: %D %U %u %V %w %X

date_format(timestamp, format) → varchar使用format指定的格式，将timestamp格式化成字符串。

date_parse(string, format) → timestamp按照format指定的格式，将字符串string解析成timestamp。

Java日期函数

在这一章节中使用的格式化字符串都是与Java的SimpleDateFormat样式兼容的。

format_datetime(timestamp, format) → varchar

使用format指定的格式，将timestamp格式化成字符串。

parse_datetime(string, format) → timestamp with time zone

按照format指定的格式，将字符串string解析成带时间戳的timestamp。

抽取函数

可以使用抽取函数来抽取如下域：

域	描述
YEAR	year()
QUARTER	quarter()
MONTH	month()
WEEK	week()
DAY	day()
DAY_OF_MONTH	day()
DAY_OF_WEEK	day_of_week()
DOW	day_of_week()

DAY_OF_YEAR	day_of_year()
DOY	day_of_year()
YEAR_OF_WEEK	year_of_week()
YOW	year_of_week()
HOUR	hour()
MINUTE	minute()
SECOND	second()
TIMEZONE_HOUR	timezone_hour()
TIMEZONE_MINUTE	timezone_minute()

抽取函数支持的数据类型取决于需要抽取的域。大多数域都支持日期和时间类型。

extract(field FROM x) → bigint

从x中返回域

Note

- SQL标准的函数一般都会使用特定的语法来指定参数。

便利的抽取函数

day(x) → bigint

返回指定日期在当月的天数

day_of_month(x) → bigint

day(x)的另一种表述

day_of_week(x) → bigint

返回指定日期对应的星期值，值范围从1 (星期一) 到 7 (星期天).

day_of_year(x) → bigint

返回指定日期对应一年中的第几天，值范围从1到 366.

dow(x) → bigint

day_of_week()的另一种表达

doy(x) → bigint

day_of_year()的另一种表达

hour(x) → bigint

返回指定日期对应的小时，值范围从1到 23

minute(x) → bigint

返回指定日期对应的分钟

month(x) → bigint

返回指定日期对应的月份

quarter(x) → bigint

返回指定日期对应的季度，值范围从1到 4

second(x) → bigint

返回指定日期对应的秒

timezone_hour(timestamp) → bigint

返回从指定时间戳对应时区偏移的小时数

timezone_minute(timestamp) → bigint

返回从指定时间戳对应时区偏移的分钟数

week(x) → bigint

返回指定日期对应一年中的ISO week，值范围从1到 53

week_of_year(x) → bigint

week的另一种表述

year(x) → bigint

返回指定日期对应的年份

year_of_week(x) → bigint

返回指定日期对应的ISO week的年份

yow(x) → bigint

year_of_week()的另一种表达

这一部分使用了和Teradata SQL 的datetime函数兼容的字符串格式. 下表基于Teradata的使用手册, , 介绍了支持的格式:

格式	描述
- / , . ; :	标点符号被忽略
dd	天 (1-31)
hh	12小时制 (1-12)
hh24	24小时制 (0-23)
mi	分钟 (0-59)
mm	月 (01-12)

ss	秒 (0-59)
yyyy	4位年
yy	2位年

Warning

- 目前不支持大小写区分，所有说明符必须为小写

to_char(timestamp, format) → varchar

timestamp 转化为 format 格式的时期字符串.

to_timestamp(string, format) → timestamp

将 string 用 format 解析为 TIMESTAMP .

to_date(string, format) → date

将 string 用 format 解析为 DATE .

DateTime 函数MySQL兼容性(ADS 已经支持的MySQL函数)

ADDDATE

Add time values (intervals) to a date value. When invoked with the INTERVAL form of the second argument, ADDDATE() is a synonym for DATE_ADD().

支持的语法：

ADDDATE(date,INTERVAL expr unit), ADDDATE(expr,days)

例子：

```
SELECT DATE_ADD('2008-01-02', INTERVAL 31 DAY);
SELECT ADDDATE('2008-01-02', INTERVAL 31 DAY);
SELECT ADDDATE('2008-01-02', 31);
```

ADDTIME

ADDTIME() adds expr2 to expr1 and returns the result. expr1 is a time or datetime expression, and expr2 is a time expression.

支持的语法：

ADDTIME(expr1,expr2)

例子：

```
SELECT ADDTIME('2007-12-31 23:59:59.999999', '1 1:1:1.000002');
SELECT ADDTIME('01:00:00.999999', '02:00:00.999998');
```

CURDATE

Returns the current date as a value in ‘YYYY-MM-DD’. CURRENT_DATE and CURRENT_DATE() are synonyms for CURDATE().

支持的语法：

CURDATE()

例子：

```
SELECT CURDATE();
SELECT CURRENT_DATE();
SELECT CURRENT_DATE;
```

SYSDATE

Returns the current date and time as a value in ‘YYYY-MM-DD HH:MM:SS’ format.

支持的语法：

SYSDATE()

例子：

```
SELECT SYSDATE();
```

CURRENT_DATE

Synonyms for CURDATE()

支持的语法：

CURRENT_DATE ， CURRENT_DATE()

例子：

```
SELECT CURRENT_DATE;
SELECT CURRENT_DATE();
```

CURRENT_TIME

Synonyms for CURTIME()

支持的语法：

CURRENT_TIME ， CURRENT_TIME()

例子：

```
SELECT CURRENT_TIME;
SELECT CURRENT_TIME();
```

CURTIME

Returns the current date as a value in ‘HH.MM.SS’. CURRENT_TIME and CURRENT_TIME() are synonyms for CURTIME().

支持的语法：

CURTIME()

例子：

```
SELECT CURTIME();
SELECT CURRENT_TIME();
SELECT CURRENT_TIME;
```

Yearmonth

查询指定列的日和月，例如YEARMONTH(‘20140602’)=201406;

DATE

Extracts the date part of the date or datetime expression expr.

支持的语法：

DATE(expr)

例子：

```
SELECT DATE('2003-12-31 01:02:03')
```

DATEDIFF

DATEDIFF() returns expr1 − expr2 expressed as a value in days from one date to the other. expr1 and expr2 are date or date-and-time expressions. Only the date parts of the values are used in the calculation.

支持的语法：

DATEDIFF(expr1,expr2)

例子：

```
SELECT DATEDIFF('2007-12-31 23:59:59','2007-12-30');
SELECT DATEDIFF('2010-11-30 23:59:59','2010-12-31');
```

DATE_FORMAT

Formats the date value according to the format string.

支持的语法：

DATE_FORMAT(expr1,expr2)

例子：

```
SELECT DATE_FORMAT('2009-10-04 22:23:00', '%W %M %Y');
SELECT DATE_FORMAT('2007-10-04 22:23:00', '%H:%i:%s');
SELECT DATE_FORMAT('1997-10-04 22:23:00', '%d');
SELECT DATE_FORMAT('2009-10-04 22:23:00', '%Y-%m-%d');
SELECT DATE_FORMAT('2009-10-04 22:23:00', '%y-%m-%d');
SELECT DATE_FORMAT('2009-10-04 22:23:00', '%Y-%m-%d %T');
SELECT DATE_FORMAT('2009-10-04 22:23:00', '%Y-%m-%d %r');
```

DAY

DAY() is a synonym for DAYOFMONTH().

支持的语法：

DAY(date)

例子：

```
SELECT DAY('2007-02-03');
SELECT DAYOFMONTH('2007-02-03');
```

DAYNAME

Returns the name of the weekday for date.

支持的语法：

DAYNAME(date)

例子：

```
SELECT DAYNAME('2007-02-03');
```

DAYOFWEEK

Returns the weekday index for date.

支持的语法：

DAYOFWEEK(date)

例子：

```
SELECT DAYOFWEEK('2007-02-03');
```

DAYOFYEAR

Returns the day of the year for date, in the range 1 to 366.

支持的语法：

DAYOFYEAR(date)

例子：

```
SELECT DAYOFYEAR('2007-02-03');
```

EXTRACT

The EXTRACT() function uses the same kinds of unit specifiers as DATE_ADD() or DATE_SUB(), but extracts parts from the date rather than performing date arithmetic.

支持的语法：

EXTRACT(unit FROM date)

例子：

```
SELECT EXTRACT(YEAR FROM '2009-07-02');
SELECT EXTRACT(MONTH FROM '2009-07-02');
SELECT EXTRACT(DAY FROM '2009-07-02');
SELECT EXTRACT(HOUR FROM '2003-01-02 10:30:00');
SELECT EXTRACT(MINUTE FROM '2003-01-02 10:30:00');
SELECT EXTRACT(SECOND FROM '2003-01-02 10:30:00');
```

FROM_DAYS

Given a day number N, returns a DATE value.

支持的语法：

FROM_DAYS(N)

例子：

```
```

```
SELECT FROM_DAYS(730669);
```

FROM_UNIXTIME

Returns a representation of the unix_timestamp argument as a value in 'YYYY-MM-DD HH:MM:SS'.

支持的语法：

FROM_UNIXTIME(unix_timestamp) , FROM_UNIXTIME(unix_timestamp,format)

例子：

```
SELECT FROM_UNIXTIME(1447430881);
SELECT FROM_UNIXTIME(UNIX_TIMESTAMP(),'yyyy-MM-dd');
```

HOUR

Returns the hour for time. The range of the return value is 0 to 23 for time-of-day values.

支持的语法：

HOUR(time)

例子：

```
SELECT HOUR('10:05:03');
SELECT HOUR('272:59:59');
```

LAST_DAY

Takes a date or datetime value and returns the corresponding value for the last day of the month.

支持的语法：

LAST_DAY(date)

例子：

```
SELECT LAST_DAY('2003-02-05');
SELECT LAST_DAY('2004-01-01 01:01:01');
```

LOCALTIME

LOCALTIME and LOCALTIME() are synonyms for NOW()

支持的语法：

LOCALTIME , LOCALTIME()

例子：

```
SELECT LOCALTIME;
SELECT LOCALTIME();
```

NOW

NOW() is synonyms for LOCALTIME and LOCALTIME()

支持的语法：

NOW()

例子：

```
SELECT NOW();
```

LOCALTIME

LOCALTIMESTAMP and LOCALTIMESTAMP() are synonyms for NOW()

支持的语法：

LOCALTIME , LOCALTIME()

例子：

```
SELECT LOCALTIMESTAMP;
SELECT LOCALTIMESTAMP();
```

MAKETIME

Returns a time value calculated from the hour, minute, and second arguments.

支持的语法：

MAKETIME(hour,minute,second)

例子：

```
SELECT MAKETIME(12,15,30);
```

MINUTE

Returns the minute for time, in the range 0 to 59.

支持的语法：

MINUTE(time)

例子：

```
SELECT MINUTE('2008-02-03 10:05:03');
```

MONTH

Returns the month for date, in the range 1 to 12 for January to December.

支持的语法：

MONTH(date)

例子：

```
SELECT MONTH('2008-02-03');
```

MONTHNAME

Returns the full name of the month for date.

支持的语法：

MONTHNAME(date)

例子：

```
SELECT MONTHNAME('2008-02-03');
```

NOW

Returns the current date and time as a value in ‘YYYY-MM-DD HH:MM:SS’.

支持的语法：

NOW()

例子：

```
SELECT NOW();
```

PERIOD_ADD

Adds N months to period P (in the format YYMM or YYYYMM). Returns a value in the format YYYYMM.

支持的语法：

PERIOD_ADD(P,N)

例子：

```
SELECT PERIOD_ADD(200801,2);
```

PERIOD_DIFF

Returns the number of months between periods P1 and P2.

支持的语法：

PERIOD_DIFF(P1,P2)

例子：

```
SELECT PERIOD_DIFF(200802,200703);
```

QUARTER

Returns the quarter of the year for date, in the range 1 to 4.

支持的语法：

QUARTER(date)

例子：

```
SELECT QUARTER('2008-04-01');
```

SECOND

Returns the second for time, in the range 0 to 59.

支持的语法：

SECOND(time)

例子：

```
SELECT SECOND('10:05:03');
```

SEC_TO_TIME

Returns the seconds argument, converted to hours, minutes, and seconds, as a TIME value.

支持的语法：

SEC_TO_TIME(seconds)

例子：

```
```

```
SELECT SEC_TO_TIME(2378);
```

STR_TO_DATE

This is the inverse of the DATE_FORMAT() function. It takes a string str and a format string format.

支持的语法：

STR_TO_DATE(str,format)

例子：

```
SELECT STR_TO_DATE('01,5,2013','%d,%m,%Y');
```

SUBDATE

When invoked with the INTERVAL form of the second argument, SUBDATE() is a synonym for DATE_SUB().

支持的语法：

SUBDATE(date,INTERVAL expr unit), SUBDATE(expr,days)

例子：

```
SELECT DATE_SUB('2008-01-02', INTERVAL 31 DAY);
SELECT SUBDATE('2008-01-02', INTERVAL 31 DAY);
SELECT SUBDATE('2008-01-02 12:00:00', 31);
```

SUBTIME

SUBTIME() returns expr1 − expr2 expressed as a value in the same format as expr1

支持的语法：

SUBTIME(expr1,expr2)

例子：

```
SELECT SUBTIME('2007-12-31 23:59:59.999999','1 1:1:1.000002');
SELECT SUBTIME('01:00:00.999999', '02:00:00.999998');
```

STR_TO_DATE

This is the inverse of the DATE_FORMAT() function. It takes a string str and a format string format.

支持的语法：

STR_TO_DATE(str,format)

例子：

```
SELECT STR_TO_DATE('01,5,2013','%d,%m,%Y');
```

TIME

Extracts the time part of the time or datetime expression expr and returns it as a string.

支持的语法：

TIME(expr)

例子：

```
SELECT TIME('2003-12-31 01:02:03');
```

TIMESTAMP

With a single argument, this function returns the date or datetime expression expr as a datetime value. With two arguments, it adds the time expression expr2 to the date or datetime expression expr1 and returns the result as a datetime value.

支持的语法：

TIMESTAMP(expr) , TIMESTAMP(expr1,expr2)

例子：

```
SELECT TIMESTAMP('2003-12-31');
SELECT TIMESTAMP('2003-12-31 12:00:00','12:00:00');
```

TIMESTAMPADD

Adds the integer expression interval to the date or datetime expression datetime_expr.

支持的语法：

TIMESTAMPADD(unit,interval,datetime_expr)

例子：

```
SELECT TIMESTAMPADD(WEEK,1,'2003-01-02');
SELECT TIMESTAMPADD(YEAR,1,'2003-01-02');
SELECT TIMESTAMPADD(MONTH,1,'2003-01-02');
SELECT TIMESTAMPADD(DAY,1,'2003-01-02');
```

TIMESTAMPDIFF

Returns datetime_expr2 − datetime_expr1, where datetime_expr1 and datetime_expr2 are date or datetime expressions.

支持的语法：

TIMESTAMPDIFF(unit,datetime_expr1,datetime_expr2)

例子：

```
SELECT TIMESTAMPDIFF(MONTH,'2003-02-01','2003-05-01');
SELECT TIMESTAMPDIFF(YEAR,'2002-05-01','2001-01-01');
SELECT TIMESTAMPDIFF(MINUTE,'2003-02-01','2003-05-01 12:05:55');
```

TIME_TO_SEC

Returns the time argument, converted to seconds.

支持的语法：

TIME_TO_SEC(time)

例子：

```
SELECT TIME_TO_SEC('22:23:00');
SELECT TIME_TO_SEC('00:39:38');
```

TO_DAYS

Given a date date, returns a day number (the number of days since year 0).

支持的语法：

TO_DAYS(date)

例子：

```
SELECT TO_DAYS('2007-10-07');
SELECT TO_DAYS('2008-10-07');
SELECT TO_DAYS('0000-00-00');
```

TO_SECONDS

Given a date or datetime expr, returns the number of seconds since the year 0. If expr is not a valid date or datetime value, returns NULL.

支持的语法：

TO_SECONDS(date)

例子：

```
SELECT TO_SECONDS('2009-11-29');
SELECT TO_SECONDS('2009-11-29 13:43:32');
SELECT TO_SECONDS('0000-00-00');
```

TO_SECONDS

Given a date or datetime expr, returns the number of seconds since the year 0. If expr is not a valid date or datetime value, returns NULL.

支持的语法：

TO_SECONDS(date)

例子：

```
SELECT TO_SECONDS('2009-11-29');
SELECT TO_SECONDS('2009-11-29 13:43:32');
SELECT TO_SECONDS('0000-00-00');
```

UNIX_TIMESTAMP

If called with no argument, returns a Unix timestamp (seconds since ‘1970-01-01 00:00:00’ UTC).

支持的语法：

UNIX_TIMESTAMP() , UNIX_TIMESTAMP(date)

例子：

```
SELECT UNIX_TIMESTAMP();
SELECT UNIX_TIMESTAMP('2015-11-13 10:20:19');
SELECT UNIX_TIMESTAMP('2015-11-13 10:20:19.012');
SELECT UNIX_TIMESTAMP('2005-03-27 03:00:00');
```

UTC_DATE

Returns the current UTC date as a value in ‘YYYY-MM-DD’ .

支持的语法：

UTC_DATE()

例子：

```
SELECT UTC_DATE()
```

UTC_TIME

Returns the current UTC time as a value in ‘HH:MM:SS’.

支持的语法：

UTC_TIME()

例子：

```
SELECT UTC_TIME()
```

UTC_TIMESTAMP

Returns the current UTC date and time as a value in ‘YYYY-MM-DD HH:MM:SS’.

支持的语法：

UTC_TIMESTAMP()

例子：

```
SELECT UTC_TIMESTAMP()
```

WEEK

This function returns the week number for date. The two-argument form of WEEK() enables you to specify whether the week starts on Sunday or Monday and whether the return value should be in the range from 0 to 53 or from 1 to 53.

支持的语法：

WEEK(date[,mode])

例子：

```
SELECT WEEK('2008-02-20');
SELECT WEEK('2008-02-20',0);
SELECT WEEK('2008-02-20',1);
```

WEEKDAY

Returns the weekday index for date (0 = Monday, 1 = Tuesday, … 6 = Sunday).

支持的语法：

WEEKDAY(date)

例子：

```
SELECT WEEKDAY('2008-02-03 22:23:00');
```

```
SELECT WEEKDAY('2007-11-06');
```

WEEKOFYEAR

Returns the calendar week of the date as a number in the range from 1 to 53.

支持的语法：

WEEKOFYEAR(date)

例子：

```
SELECT WEEKOFYEAR('2008-02-20');
```

YEAR

Returns the year for date, in the range 1000 to 9999, or 0 for the “zero” date.

支持的语法：

YEAR(date)

例子：

```
SELECT YEAR('1987-01-01');
```

YEARWEEK

Returns year and week for a date. The year in the result may be different from the year in the date argument for the first and the last week of the year.

支持的语法：

YEARWEEK(date) , YEARWEEK(date,mode)

例子：

```
SELECT YEARWEEK('1987-01-01');
SELECT YEARWEEK('1987-01-01',1);
```

TIMEDIFF

TIMEDIFF() returns expr1 − expr2 expressed as a time value. expr1 and expr2 are date-and-time expressions, but both must be of the same type.

支持的语法：

TIMEDIFF(expr1,expr2)

例子：

```
SELECT TIMEDIFF('2008-12-31 23:59:50', '2008-12-31 23:59:59');
SELECT TIMEDIFF('2008-12-30 23:59:59', '2008-12-31 23:59:59');
SELECT TIMEDIFF('2008-12-30 11:59:59', '2008-12-31 23:59:59');
SELECT TIMEDIFF('2008-12-30 11:50:59', '2008-12-31 23:59:59');
SELECT TIMEDIFF('2008-12-30 11:50:50', '2008-12-31 23:59:59');
```

CONVERT_TZ

CONVERT_TZ() converts a datetime value dt from the time zone given by from_tz to the time zone given by to_tz and returns the resulting value.

支持的语法：

CONVERT_TZ(dt,from_tz,to_tz)

例子：

```
SELECT CONVERT_TZ('2004-01-01 12:00:00','GMT','MET');
```

TIME_FORMAT

This is used like the DATE_FORMAT() function, but the format string may contain format specifiers only for hours, minutes, seconds, and microseconds. Other specifiers produce a NULL value or 0.

支持的语法：

TIME_FORMAT(time,format)

例子：

```
SELECT TIME_FORMAT('10:00:00', '%H %k %h %I %l');
```

Datetime函数Oracle兼容性(ADS 已经支持的Oracle函数)

ADD_MONTHS

ADD_MONTHS returns the date date plus integer months.

支持的语法：

ADD_MONTHS(date, integer)

例子：

```
ADD_MONTHS('2010-10-10',1)
```

CURRENT_DATE

CURRENT_DATE returns the current date in the session time zone, in a value in the Gregorian calendar of datatype DATE.

支持的语法：

CURRENT_DATE

例子：

```
SELECT CURRENT_DATE;
```

CURRENT_TIMESTAMP

CURRENT_TIMESTAMP returns the current date and time in the session time zone, in a value of datatype TIMESTAMP WITH TIME ZONE.

支持的语法：

CURRENT_TIMESTAMP

例子：

```
SELECT CURRENT_TIMESTAMP;
```

DBTIMEZONE

DBTIMEZONE returns the value of the database time zone.

支持的语法：

DBTIMEZONE()

例子：

```
SELECT DBTIMEZONE();
```

EXTRACT

EXTRACT extracts and returns the value of a specified datetime field from a datetime or interval value expression.

支持的语法：

EXTRACT(unit FROM datetime)

例子：

```
SELECT EXTRACT(YEAR FROM '2001-01-01');
SELECT EXTRACT(MONTH FROM '2001-01-01');
SELECT EXTRACT(DAY FROM '2001-01-01');
SELECT EXTRACT(HOUR FROM '2001-01-01 19:10:11');
SELECT EXTRACT(MINUTE FROM '2001-01-01 19:10:11');
SELECT EXTRACT(SECOND FROM '2001-01-01 19:10:11');
```

LAST_DAY

LAST_DAY returns the date of the last day of the month that contains date.

支持的语法：

LAST_DAY

例子：

```
LAST_DAY('2001-01-01');
```

LOCALTIMESTAMP

LOCALTIMESTAMP returns the current date and time in the session time zone in a value of datatype TIMESTAMP.

支持的语法：

LOCALTIMESTAMP()

例子：

```
SELECT LOCALTIMESTAMP();
```

MONTH_BETWEEN

MONTHS_BETWEEN returns number of months between dates date1 and date2.

支持的语法：

MONTH_BETWEEN(date1, date2)

例子：

```
SELECT MONTH_BETWEEN('2017-03-03', '2017-07-07');
SELECT MONTH_BETWEEN('2017-04-03', '2017-07-07');
```

NEXT_DAY

NEXT_DAY returns the date of the first weekday named by char that is later than the date date.

支持的语法：

NEXT_DAY(date, char)

例子：

```
SELECT NEXT_DAY('2010-10-10','TUESDAY');
SELECT NEXT_DAY('2010-10-10','TUE');
```

ROUND

ROUND returns date rounded to the unit specified by the format model fmt. The value returned is always of datatype DATE, even if you specify a different datetime datatype for date.

支持的语法：

ROUND(date, fmt)

例子：

```
SELECT ROUND(TIMESTAMP '2010-08-21', 'YY');
SELECT ROUND(TIMESTAMP '2010-08-21', 'MM');
SELECT ROUND(TIMESTAMP '2010-08-21', 'q');
SELECT ROUND(TIMESTAMP '2010-08-21', 'D');
SELECT ROUND(TIMESTAMP '2010-08-21 19:00:00', 'DD');
```

SESSIONTIMEZONE

SESSIONTIMEZONE returns the time zone of the current session.

支持的语法：

SESSIONTIMEZONE()

例子：

```
SELECT SESSIONTIMEZONE();
```

SYSDATE

SYSDATE returns the current date and time set for the operating system on which the database resides.

支持的语法：

SYSDATE()

例子：

```
SELECT SYSDATE();
```

TO_CHAR

TO_CHAR (datetime) converts a datetime or interval value of DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, or TIMESTAMP WITH LOCAL TIME ZONE datatype to a value of VARCHAR2 datatype in the format specified by the date format fmt.

支持的语法：

TO_CHAR(datetime, fmt)

例子：

```
SELECT TO_CHAR('2013-05-17 23:35:10', '%Y-%m-%d %H:%i:%s');
SELECT TO_CHAR('2013-05-17 00:35:10', '%Y-%m-%d %H:%i:%s');
SELECT TO_CHAR('2013-05-17 12:35:10', '%Y-%m-%d %h:%i:%s %p');
```

TRUNC

The TRUNC (date) function returns date with the time portion of the day truncated to the unit specified by the format model fmt.

支持的语法：

TRUNC(date, fmt)

例子：

```
SELECT TRUNC(TIMESTAMP '2010-08-21', 'YY');
SELECT TRUNC(TIMESTAMP '2010-08-21', 'MM');
SELECT TRUNC(TIMESTAMP '2010-08-21', 'q');
SELECT TRUNC(TIMESTAMP '2010-08-21 19:00:00', 'DD');
```

TO_DATE

TO_DATE converts char of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 datatype to a value of DATE datatype.

支持的语法：

TO_DATE(char, fmt)

例子：

```
SELECT TO_DATE('2013-05', '%Y-%m');
SELECT TO_DATE('2013-05-17 12:35:10', '%Y-%m-%d %h:%i:%s');
```

NEW_TIME

NEW_TIME returns the date and time in time zone timezone2 when date and time in time zone timezone1 are date.

支持的语法：

NEW_TIME(date, timezone1, timezone2)

例子：

```
SELECT NEW_TIME('2004-01-01 12:00:00','GMT','MET');
SELECT NEW_TIME('2004-01-01 12:00:00.123','GMT','MET');
```

AnalyticDB 会尝试隐式转换数值类型和字符类型值到正确的类型，但是不会在数值类型和字符类型之间自动转换。 例如，查询返回的长整型数据不会自动转变为你想要的varchar类型。

转换函数

cast(value AS type) → type

显式把value转换到type类型。可用于把字符类型值转变为数值类型，反之亦然。

try_cast(value AS type) → type

类似 cast(), 但是会在转型失败的时候返回 NULL

Miscellaneous

typeof(expr) → varchar

返回 expr 表达式的结果类型:

```
SELECT typeof(123); -- integer

SELECT typeof('cat'); -- varchar(3)

SELECT typeof(cos(2) + 1.5); -- double
```

bit_count(x, bits) → bigint

返回 x (视为 bits 位的有符号整形) 的 bits 位补码表示中, 为1的位的个数:

```
SELECT bit_count(9, 64); -- 2
SELECT bit_count(9, 8); -- 2
SELECT bit_count(-7, 64); -- 62
SELECT bit_count(-7, 8); -- 6
```

bitwise_and(x, y) → bigint

返回 x 和 y 按位与的补码表示.

bitwise_not(x) → bigint

返回 x 取反的补码表示.

bitwise_or(x, y) → bigint

返回 x 和 y 按位或的补码表示.

bitwise_xor(x, y) → bigint

返回 x 和 y 按位异或的补码表示.

参见 bitwise_and_agg() and bitwise_or_agg().

json_array_contains(json, value) → boolean

确定json中是否存在值（包含JSON数组的字符串）

```
SELECT json_array_contains(‘[1, 2, 3]’, 2);
```

json_array_get(json_array, index) → varchar

将指定index处的元素返回到json数组中,index从0开始计数

```
SELECT json_array_get(‘[“a”, “b”, “c”]’, 0); – ‘a’ SELECT json_array_get(‘[“a”, “b”, “c”]’, 1); – ‘b’
```

此函数还支持通过负的index从数组末尾读取元素索引

```
SELECT json_array_get(‘[“c”, “b”, “a”]’, -1); – ‘a’ SELECT json_array_get(‘[“c”, “b”, “a”]’, -2); – ‘b’
```

如果指定索引处的元素不存在，则函数返回null：

```
SELECT json_array_get(‘[]’, 0); – null SELECT json_array_get(‘[“a”, “b”, “c”]’, 10); – null SELECT json_array_get(‘[“c”, “b”, “a”]’, -10); – null
```

json_array_length(json) → bigint

返回json的数组长度（包含JSON数组的字符串）

```
SELECT json_array_length(‘[1, 2, 3]’);
```

json_extract(json, json_path) → json

评估json上的JSONPath表达式json_path（包含JSON的字符串），并将结果作为JSON字符串返回：

```
SELECT json_extract(json, '$.store.book');
```

json_extract_scalar(json, json_path) → varchar

和json_extract（）类似，但返回结果值作为一个字符串（而不是编码为JSON）。 json_path引用的值必须是scalar（boolean, number or string）：

```
SELECT json_extract_scalar('[1, 2, 3]', '$[2]'); SELECT json_extract_scalar(json, '$.store.book[0].author');
```

json_format(json) → varchar

将json作为字符串返回

```
SELECT json_format(JSON '[1, 2, 3]'); – '[1,2,3]' SELECT json_format(JSON '"a"'); – '"a"'
```

json_parse(string) → json

解析字符串作为json

```
SELECT json_parse('[1, 2, 3]'); – JSON '[1,2,3]' SELECT json_parse('"a"'); – JSON '"a"'
```

json_size(json, json_path) → bigint

跟json_extract（）一样，但返回size的大小。对于对象或数组，大小是成员数，scalar 的大小为零：

```
SELECT json_size('{"x":{"a":1,"b":2}}', '$.x'); – 2 SELECT json_size('{"x":[1, 2, 3]}', '$.x'); – 3
SELECT json_size('{"x":{"a":1,"b":2}}', '$.x.a'); – 0
```

聚合函数主要应用于一组数据计算出一个结果。

除了 count(), count_if(), max_by(), min_by() 和 approx_distinct() 之外, 所有这些聚合函数均会忽略null值并且在没有输入数据或者所有数据 均是null值的情况下返回空结果。比如，sum() 会返回null而不是0, 以及 avg() 不会包含null值计数。 coalesce 函数会把null值转换成0。

常规聚合函数

arbitrary(x) → [类型与输入参数相同]

返回 x 的任意非null值。

array_agg(x) → array<[类型和输入参数相同]>

返回以输入参数 x 为元素的数组。

avg(x) → double

返回所有输入值的平均数（算数平均数）。

bool_and(boolean) → boolean

只有所有参数均为 TRUE 则返回 TRUE ，否则返回 FALSE 。

bool_or(boolean) → boolean

任何一个参数为 TRUE ，则返回 TRUE ， 否则返回 FALSE 。

checksum(x) → varbinary

返回不受给定参数值顺序影响的校验值。

count(*) → bigint

返回输入数据行的统计个数。

count(x) → bigint

返回非null值的输入参数个数。

count_if(x) → bigint

返回输入参数中 TRUE 的个数。 这个函数和 count(CASE WHEN x THEN 1 END) 等同。

every(boolean) → boolean

这个函数是 bool_and() 的别名。

geometric_mean(x) → double

返回所有输入参数的几何平均值。

max_by(x, y) → [与x类型相同]

返回 x 与 y 的全部关联中，y 最大值所关联的第一个 x 值。

max_by(x, y, n) → array<[与x类型相同]>

x 与 y 的全部关联中， 以 y 降序排列前 n 个最大值所关联的 x 值中， 返回前 n 个值

min_by(x, y) → [与x类型相同]

返回 x 与 y 的全部关联中，y 最小值所关联的第一个 x 值。

min_by(x, y, n) → array<[与x类型相同]>

x 与 y 的全部关联中， 以 y 升序排列前 n 个值所关联的 x 值中， 返回前 n 个值

max(x) → [与输入类型相同]

返回输入参数中最大的值。

max(x, n) → array<[与x类型相同]>

返回所有参数 x 中前 n 大的值.

min(x) → [与输入类型相同]

返回所有输入参数中最小的值。

min(x, n) → array<[与x类型相同]>

返回所有输入参数 x 中，前 n 小的值。

sum(x) → [和输入类型相同]

返回所有输入参数的和。

位聚合函数

bitwise_and_agg(x) → bigint

返回所有输入值二进制与的结果，以二进制补码表示。

bitwise_or_agg(x) → bigint

返回所有输入值二进制或的结果，以二进制补码表示。

映射表聚合函数

histogram(x) → map<K,bigint>

返回一个包含输入参数出现次数的映射表。

map_agg(key, value) → map<K,V>

返回一个根据输入 key 和 value 对构造的映射表。

map_union(x<K, V>) → map<K,V>

返回所有输入映射表的联合。如果一个键同时出现在多个输入映射表中，结果中对应键值会取任意输入映射表中的一个。

multimap_agg(key, value) → map<K,array<V>>

返回一个由输入参数 key 和 value 对组成的映射表。每个键可以对应多个值。

近似计算聚合函数

approx_distinct(x) → bigint

返回输入参数中不重复值的近似个数。 这个函数提供 count(DISTINCT x) 这个方法的近似计算。 如果所有输入参数均为null，则返回0。

这个函数会产生2.3%的标准误差。这个误差是基于所有可能集合上的标准误差的正常分布。 它并不能保证在特定集合上的误差上限。

approx_distinct(x, e) → bigint

返回输入参数中不重复值的近似个数。 这个函数提供 count(DISTINCT x) 这个方法的近似计算。 如果所有输入参数均为null，则返回0。

这个函数会产生一个标准误差不会超过 e 的结果。这个误差是基于所有可能集合上的标准误差的正常分布。 它并不能保证在特定集合上的误差上限。当前这个函数的实现需要 e 值在范围[0.01150, 0.26000]之间。

approx_percentile(x, percentage) → [与x类型相同]

返回输入值 x 中处于 percentage 占比的近似百分位数值。 percentage 的数值范围必须在0到1之间且 对每个输入行均是常量。如近似计算中位数为 approx_percentile(x, 0.5) 。

approx_percentile(x, percentages) → array<[与x类型相同]>

返回输入值 x 中处于每个特定百分占比的近似百分位值。其中 percentages 数组中每个元素值范围必须 在0到1且数组对每个输入行必须是常数。

approx_percentile(x, w, percentage) → [与x类型相同]

按照百分比 percentage 的每项权重 w 返回 x 的所有输入值的近似百分位数。权重必须至少有一个 整型数值。 它实际上是百分位数集合中的值 x 的复制计数。 percentage 的值必须在0和1之间且对所有 输入行必须是常数。

approx_percentile(x, w, percentage, accuracy) → [与x类型相同]

按照百分比 percentage 的每项权重 w 计算所有 x 的输入值的近似百分位数，以精度为 accuracy 。 权重必须至少有一个整型数值。它实际上是百分位数集合中的值 x 的复制计数。 percentage 的值必须在0和1之 间且对所有输入行必须是常数。 accuracy 必须要大于0并且小于1并且对于所有输入行为常量。

approx_percentile(x, w, percentages) → array<[与x类型相同]>

按照百分比 percentage 的每个项目权重 w 返回 x 的所有输入值的近似称重百分位数。 权重必须至少有一个整型数值。它实际上是百分位数集合中的值 x 的复制计数。 percentages 的每个值 值必须在0和1之间且对所有输入行必须是常数。

numeric_histogram(buckets, value, weight) → map<double, double>

按照 buckets 桶的数量,为所有的 value 计算近似直方图, 每一项的权重使用 weight 。本算法大体基于:

Yael Ben-Haim and Elad Tom-Tov, "A streaming parallel decision tree algorithm", J. Machine Learning Research 11 (2010), pp. 849--872.

buckets 必须是 bigint 类型. value 和 weight 必须是数值类型。

numeric_histogram(buckets, value) → map<double, double>按照 buckets 桶的数量,为所有的 value 计算近似直方图,本函数与 numeric_histogram() 相同，只是 weight 为1.

统计聚合函数

corr(y, x) → double

返回输入值的相关系数。

covar_pop(y, x) → double

返回输入值的总体协方差。

covar_samp(y, x) → double

返回输入值的样本协方差。

regr_intercept(y, x) → double

返回输入值的线性回归截距。 y 是因变量。 x 是自变量。

regr_slope(y, x) → double

返回输入值的线性回归斜率。 y 是因变量。 x 是自变量。

stddev(x) → double

这个函数是 stddev_samp() 的别名函数。

stddev_pop(x) → double

返回所有输入值的总体标准偏差。

stddev_samp(x) → double

返回所有输入值的样本标准偏差。

variance(x) → double

这个函数是 var_samp() 的别名函数

var_pop(x) → double

返回所有输入值的总体方差。

var_samp(x) → double

返回所有输入值的样本方差。

特色聚合函数

UDF_SYS_COUNT_COLUMN

作用：用于做多group by的聚合， 可以将多个group by的语句合并成多个UDF，写到一条sql中

格式：UDF_SYS_COUNT_COLUMN(columnName, columnName2...), 参数必须是列名

返回：一个json的字符串列，类似：{“0”:3331656,”2”:3338142,”1”:3330202}

实例：

a. select UDF_SYS_COUNT_COLUMN(c1) from table 等价于select count(*) from table group by c1

b. select UDF_SYS_COUNT_COLUMN(c1,c2) from table 等价于select count(*) from table group by c1,c2

c. select UDF_SYS_COUNT_COLUMN(c1), UDF_SYS_COUNT_COLUMN(c2),UDF_SYS_COUNT_COLUMN(c3), UDF_SYS_COUNT_COLUMN(c4)

等价于如下四条sql语句：select count() *from table group by c1;select count()* from table group by c2;select count() *from table group by c3;select count()* from table group by c4;

d. 一个真实的实例：select UDF_SYS_COUNT_COLUMN(user_gender), UDF_SYS_COUNT_COLUMN(user_level) from db_name.userbase返回1行，2列：{“0”:3331656,”2”:3338142,”1”:3330202}，{“0”:4668150,”2”:1891176,”1”:1984606，”6”:5818}

UDF_SYS_RANGECOUNT_COLUMN

作用：用于做静态样本分段(老的函数UDF_SYS_SEGCOUNT_COLUMN已经弃用，请使用该函数)

格式：UDF_SYS_RANGECOUNT_COLUMN(columnName, count, min, max)

- 第一个参数是列名
- 第二个参数分段数目
- 第三个参数是参与分段的该列在全表的最小值
- 第四个参数是参与分段的该列在全表的最大值

返回：一个json的字符串列，类似：{“ranges”:[{“start”:0,”end”:599}, {“start”:600,”end”:1899}, {“start”:1900,”end”:65326003}]}

使用说明：使用该函数进行动态分段统计，需要三个步骤：

- 第一步，通过min, max求出符合条件的最小值和最大值
- 第二步，通过UDF_SYS_RANGECOUNT_COLUMN获取各个分段。
- 第三步，通过case when+group by获取每个分段中真实聚合数据。

特别说明UDF_SYS_RAGNECOUNT_COLUMN和通过UDF_SYS_RANGECOUNT_SAMPLING_COLUMN的区别在于，前者是静态分段，也就是根据(max-min+1)/segcount进行分段，而后在是动态分段，可以保证每个分段区间内的数目是大致均衡的。

group_concat

字符串聚合函数。

目前仅支持在聚合时Group By中包括所有参与计算的表分分区列（维度表）时可以使用。

语法：GROUP_CONCAT([DISTINCT] expr [,expr ...] [ORDER BY col_name [ASC | DESC] [,col_name ...]] [SEPARATOR str_val])

distinct用于将组内多个相同的字符串仅输出一个。若含有ORDER BY结构，则组内字符串会根据col_name列表排序输出。使用SEPARATOR可以指定聚合后的字符串分隔符，默认是逗号。

length(binary) → bigint

返回 binary 字节长度。

to_base64(binary) → varchar

将 binary 编码为base64字符串。

from_base64(string) → varbinary

将base64编码的 string 解码为二进制数据。

to_base64url(binary) → varchar

使用URL安全字符将 binary 编码为base64字符串。

from_base64url(string) → varbinary

使用URL安全字符，将base64编码的 string 解码为二进制数据。

to_hex(binary) → varchar

将 binary 编码为16进制字符串。

from_hex(string) → varbinary

将16进制编码的字符串 string 解码为二进制数据

to_big_endian_64(bigint) → varbinary

将bigint编码为大字节序的二进制格式。

from_big_endian_64(binary) → bigint

将一个大字节序的 binary 解码为bigint。

md5(binary) → varbinary

返回 binary md5哈希值。

sha1(binary) → varbinary

返回 binary sha1哈希值。

sha256(binary) → varbinary

返回 binary sha256哈希值。

sha512(binary) → varbinary

返回 binary sha512哈希值。

xxhash64(binary) → varbinary

返回 binary 的xxhash64哈希值

所有的正则表达式函数都使用Java样式的语法。

- 当使用多行模式（通过（ ?m）标志启用）时，只有n被识别为行终止符。 （ ?d）标志不受支持，不能使用。
- 大小写相关以Unicode方式执行（通过（ ?i）标志启用）。 不支持上下文相关和本地匹配。 （ ?u）标志不支持。
- 不支持代理对。 例如，uD800 uDC00不被视为U + 10000，必须指定为x {10000}
- 对于没有基本字符的非间距标记，（b）会被错误地处理。
- 字符集（例如[A-Z123]）不支持Q和E，将会被认为是文字。
- Unicode字符类（p {prop}）支持以下场景：
    - 姓名中的所有下划线都必须删除。例如，使用OldItalic而不是Old_Italic
    - 必须直接指定脚本，而不需要使用Is，script =或sc = prefixes。 示例：p {Hiragana}
    - 必须使用In前缀指定blcoks。 不支持block =和blk =前缀。 示例：p {Mongolian}
    - 必须直接指定类别，不带Is，general_category =或gc =前缀。 示例：p {L}
    - 二进制属性必须直接指定，不需要Is。 示例：p {NoncharacterCodePoint}

regexp_extract_all(string, pattern) → array<varchar>

返回字符串中pattern模式匹配的substring(s) ：

```
SELECT regexp_extract_all(‘1a 2b 14m’, ‘d+’); – [1, 2, 14]
```

regexp_extract_all(string, pattern, group) → array<varchar>

查找字符串中所有出现的pattern模式，并返回group编号组

```
SELECT regexp_extract_all(‘1a 2b 14m’, ‘(d+)([a-z]+)’, 2); – [‘a’, ‘b’, ‘m’]
```

regexp_extract(string, pattern) → varchar

返回字符串中匹配pattern的第一个子字符串

```
SELECT regexp_extract(‘1a 2b 14m’, ‘d+’); – 1
```

regexp_extract(string, pattern, group) → varchar

查找字符串中第一个出现的pattern模式，并返回group编号组

```
SELECT regexp_extract(‘1a 2b 14m’, ‘(d+)([a-z]+)’, 2); – ‘a’
```

regexp_like(string, pattern) → boolean

评估pattern并确定它是否包含在字符串中

此函数类似于LIKE运算符，期望pattern仅需要包含在字符串中，而不需要匹配所有的字符串。执行一个包含操作而不是匹配操作。可以通过使用^和$来匹配整个字符串

```
SELECT regexp_like(‘1a 2b 14m’, ‘d+b’); – true
```

regexp_replace(string, pattern) → varchar

从字符串中删除与pattern匹配的子字符串的每个实例

```
SELECT regexp_replace(‘1a 2b 14m’, ‘d+[ab] ’); – ‘14m’
```

regexp_replace(string, pattern, replacement) → varchar

替换字符串中pattern匹配的子字符串的每个实例。 可以使用$ g作为替代使用的引用组，也可以使用${name}来引用。 替换中的美元符号（$）可能会需要用（$）进行转义

```
SELECT regexp_replace(‘1a 2b 14m’, ‘(d+)([ab]) ’, ‘3c$2 ’); – ‘3ca 3cb 14m’
```

regexp_split(string, pattern) → array<varchar>

使用pattern模式拆分字符串并返回数组。结尾的空字符串被保留

```
SELECT regexp_split(‘1a 2b 14m’, ‘s*[a-z]+s*’); – [1, 2, 14, ]
```

提取函数

URL方法用于从HTTP URLs（或者是任何满足RFC 2396标准的有效URIs）中提取相应的信息。URL方法支持如下的语法：

[protocol:][//host[:port]][path][?query][#fragment]

被从URLs中提取出来的部分，不会包括URI的语法分隔符（如:或者?）

url_extract_fragment(url) → varchar

从URL返回fragment标识符

url_extract_host(url) → varchar

从url返回host

url_extract_parameter(url, name) → varchar

从url返回名为name的第一个查询字符串参数的值。 参数提取按照RFC 1866＃section-8.2.1规定的典型方式

处理

url_extract_path(url) → varchar

从url返回path

url_extract_port(url) → bigint

从url返回port值

url_extract_protocol(url) → varchar

从url返回protocol

url_extract_query(url) → varchar

从url返回extract_query

Encoding函数

url_encode(value) → varchar

通过编码来转义值，以便它可以安全地包含在URL查询参数名称和值中

- Alphanumeric字符不进行编码
- 字符.,-, * and _ 不进行编码
- ASCII空格字符编码为+
- 所有其他字符都转换为UTF-8，字节编码为字符串%XX，其中XX是UTF-8字节的大写十六进制值

url_decode(value) → varchar

解除URL编码的值。 这个函数是url_encode（ ）的反向。

array_distinct(x) → array

从数组 x 删除相同的值.

array_intersect(x, y) → array

返回 x 与 y 的交集,没有重复.

array_union(x, y) → array

返回 x 与 y 的并集,没有重复.

array_join(x, delimiter, null_replacement) → varchar

使用delimiter来连接数组的元素组成字符串,并且用可选的null_replacement来代替空值.

array_max(x) → x

返回数组的最大值.

array_min(x) → x

返回数组的最小值.

array_position(x, element) → bigint

返回 x 中第一个匹配 element 的位置,如果没有找个匹配的元素,则返回0.

array_remove(x, element) → array

移除 x 中所有等与 element 的元素.

array_sort(x) → array

对 x 进行排序. x 的元素必须是可比较的. 空值将会被放置在结果数组的末尾.

cardinality(x) → bigint

返回 x 的size.

concat(x, y) → array

连接 x 和 y. 这个函数的功能与操作符 || 功能相同.

contains(x, element) → boolean

如果 x 包含 element 则返回true.

element_at(array<E>, index) → E

返回 array 中给定 index 位置的元素. 如果 index >= 0, 这个函数和操作符 []的功能相同. 如果 index 小于 0，element_at 就从最后一个元素开始倒序获取第 index 绝对值位置的元素.

filter(array, function) → array

参见 filter().

flatten(x) → array

通过连接将 array(array(T)) 映射到 array(T) .

reduce(array, initialState, inputFunction, outputFunction) → x

参见 reduce().

reverse(x) → array

返回一个与 x 反序的数组.

sequence(start, stop) → array<bigint>

返回 start 到 stop 之间的证书序列. 如果 start 小于或等于 stop 则按 1 递增, 否则按 -1 递减 .

sequence(start, stop, step) → array<bigint>

返回一个从 start to stop 之间按step 递增的序列.

sequence(start, stop, step) → array<timestamp>

返回一个从 start to stop 之间按step 递增的序列.

step 的类型可以是 INTERVAL DAY TO SECOND 或者 INTERVAL YEAR TO MONTH .

shuffle(x) → array

生成数组 x 的随机排列.

slice(x, start, length) → array

从 x 生成一个 start 位置开始length结束的数组,如果 start 是负数，则从数组末尾开始.

transform(array, function) → array

参见 transform().

zip(array1, array2[, ...]) → array<row>

将给定数组合并为一个单行数组. 原多行数组中的第M行的第N个元素将会被放置在第N个输出元素的第M位置. 如果多行的长度不一致，那么会将缺省值补为NULL:

```
SELECT zip(ARRAY[1, 2], ARRAY['1b', null, '3b']); -- [ROW(1, '1b'), ROW(2, null), ROW(null, '3b')]
```

cardinality(x) → bigint

返回map x的Size（大小）

element_at(map<K, V>, key) → V

返回一个key的value，如果map不包含当前的key则返回NULL.

map() → map<unknown, unknown>

返回空的map

```
SELECT map(); -- {}
```

map(array<K>, array<V>) → map<K,V>

返回使用给定key/value数组创建的map

```
SELECT map(ARRAY[1,3], ARRAY[2,4]); -- {1 -> 2, 3 -> 4}
```

See also map_agg() and multimap_agg() for creating a map as an aggregation.

map_concat(x<K, V>, y<K, V>) → map<K,V>

返回两个map的并集. 如果一个key同时包含在 x and y, 那么这个key的value返回包含在 y 中的value.

map_filter(map<K, V>, function) → map<K,V>

参见 map_filter().

map_keys(x<K, V>) → array<K>

返回 map x 的所有的key.

map_values(x<K, V>) → array<V>

返回 map x 所有的value.

Lambda Expression

Lambda表达式写为 ->:

```
x -> x + 1
(x, y) -> x + y
x -> regexp_like(x, 'a+')
x -> x[1] / x[2]
x -> IF(x > 0, x, -x)
x -> COALESCE(x, 0)
x -> CAST(x AS JSON)
```

大多数SQL表达式可以在lambda体中使用，少数的需要注意: *不支持子查询.* TRY 函数不支持. (try_cast() 支持.)
* Capture is not supported yet:

- Columns or relations cannot be referenced.
- Only lambda variables from the inner-most lambda expression can be referenced.

Lambda Functions

filter(array<T>, function<T, boolean>) → ARRAY<T>

从 array 的元素中获取可以使 function 返回true的元素构成新的数组:

```
SELECT filter(ARRAY [], x -> true); -- []
SELECT filter(ARRAY [5, -6, NULL, 7], x -> x > 0); -- [5, 7]
SELECT filter(ARRAY [5, NULL, 7, NULL], x -> x IS NOT NULL); -- [5, 7]
```

map_filter(map<K, V>, function<K, V, boolean>) → MAP<K,V>

从 map 的元素中获取使 function 返回true的元素构成新的数组:

```
SELECT map_filter(MAP(ARRAY[], ARRAY[]), (k, v) -> true); -- {}
SELECT map_filter(MAP(ARRAY[10, 20, 30], ARRAY['a', NULL, 'c']), (k, v) -> v IS NOT NULL); -- {10 -> a, 30 -> c}
SELECT map_filter(MAP(ARRAY['k1', 'k2', 'k3'], ARRAY[20, 3, 15]), (k, v) -> v > 10); -- {k1 -> 20, k3 -> 15}
```

transform(array<T>, function<T, U>) → ARRAY<U>

收集 array 的每个元素作为 function 的输入得到的结果构建新的数组:

```
SELECT transform(ARRAY [], x -> x + 1); -- []
```

```
SELECT transform(ARRAY [5, 6], x -> x + 1); -- [6, 7]
SELECT transform(ARRAY [5, NULL, 6], x -> COALESCE(x, 0) + 1); -- [6, 1, 7]
SELECT transform(ARRAY ['x', 'abc', 'z'], x -> x || '0'); -- ['x0', 'abc0', 'z0']
SELECT transform(ARRAY [ARRAY [1, NULL, 2], ARRAY[3, NULL]], a -> filter(a, x -> x IS NOT NULL)); -- [[1, 2], [3]]
```

reduce(array<T>, initialState S, inputFunction<S, T, S>, outputFunction<S, R>) → R

reduce array 的结果是单个值. inputFunction 将会按顺序读取 array 的元素. 在获取元素的同时, inputFunction 获取当前的状态, 初始化 initialState, 并返回一个新的值. outputFunction 将会被唤醒将结果置为最终状态. 它可能是 (i -> i) 这样的函数. 例如:

```
SELECT reduce(ARRAY [], 0, (s, x) -> s + x, s -> s); -- 0
SELECT reduce(ARRAY [5, 20, 50], 0, (s, x) -> s + x, s -> s); -- 75
SELECT reduce(ARRAY [5, 20, NULL, 50], 0, (s, x) -> s + x, s -> s); -- NULL
SELECT reduce(ARRAY [5, 20, NULL, 50], 0, (s, x) -> s + COALESCE(x, 0), s -> s); -- 75
SELECT reduce(ARRAY [5, 20, NULL, 50], 0, (s, x) -> IF(x IS NULL, s, s + x), s -> s); -- 75
SELECT reduce(ARRAY [2147483647, 1], CAST (0 AS BIGINT), (s, x) -> s + x, s -> s); -- 2147483648
SELECT reduce(ARRAY [5, 6, 10, 20], -- calculates arithmetic average: 10.25
CAST(ROW(0.0, 0) AS ROW(sum DOUBLE, count INTEGER)),
(s, x) -> CAST(ROW(x + s.sum, s.count + 1) AS ROW(sum DOUBLE, count INTEGER)),
s -> IF(s.count = 0, NULL, s.sum / s.count));
```

UDF_SYS_GEO_IN_CYCLE

作用：用于做基于地理位置的经纬度画圈

格式：UDF_SYS_GEO_IN_CYCLE(longitude, latitude, point, radius)

- 第一个参数为经度列名称， 类型float
- 第二个参数为纬度列名称， 类型float
- 第三个参数为圆圈中心点的位置，格式=>”经度，维度”，=>”120.85979,30.011984”
- 第四个参数为圆圈的半径，单位米

返回：返回一个boolean值

使用说明：

```
select count(*) from db_name.usertag where udf_sys_geo_in_cycle(longitude,latitude, "120.85979,30.011984",
5000)=true
```

求以”120.85979,30.011984”为中心点，半径为5km的圆圈内的人数

```
select longitude,latitude from db_name.usertag where udf_sys_geo_in_cycle(longitude,latitude,
"120.85979,30.011984", 5000)=true order by longitude
```

UDF_SYS_GEO_IN_RECTANGLE

作用：用于做基于地理位置的经纬度画矩形

格式：UDF_SYS_GEO_IN_RECTANGLE(longitude, latitude, pointA, pointB)

- 第一个参数为经度列名称, 类型float
- 第二个参数为纬度列名称, 类型float
- 第三个参数为矩形的左下角坐标，格式=>”经度，维度”，=>”120.85979,30.011984”
- 第四个参数为矩形的右上角坐标，格式=>”经度，维度”，=>”120.88450,31.21011”

返回：返回一个boolean值

使用说明：

```
select count(*) from db_name.usertag where udf_sys_geo_in_rectangle(longitude,latitude, "120.85979,30.011984",
"120.88450,31.21011")=true
```

求以”120.85979,30.011984”和”“120.88450,31.21011””为2个斜角构成的矩形圈内的人数

UDF_SYS_GEO_DISTANCE

作用：用作一个经纬度列和一个固定的坐标点的距离计算

格式：UDF_SYS_GEO_DISTANCE(longitude, latitude, pointA)

- 第一个参数为经度列名称, 类型float
- 第二个参数为纬度列名称, 类型float
- 第三个参数为固定坐标点的经纬度，格式=>”经度，维度”，
  =>”120.85979,30.011984”

返回：返回一个int值，单位为米(M)

使用说明：

```
select count(*) from db_name.usertag where udf_sys_geo_in_rectangle(longitude,latitude, "120.85979,30.011984",
"120.88450,31.21011")=true
```

求以”120.85979,30.011984”和”“120.88450,31.21011””为2个斜角构成的矩形圈内的人数

窗口函数基于查询结果的行数据进行计算。窗口函数运行在 HAVING 子句之后，但是在 ORDER BY 子句之前

。触发一个窗口函数需要特殊的关键字 OVER 子句来指定窗口。一个窗口包含三个组成部分：

- 分区规范，用于将输入行分裂到不同的分区中。这个过程和 GROUP BY 子句的分裂过程相似。
- 排序规范，用于决定输入数据行在窗口函数中执行的顺序。
- 窗口框架，用于指定一个滑动窗口的数据给窗口函数处理给定的行数据。如果这个框架没有指定，它默认的方式是 RANGE UNBOUNDED PRECEDING ，与 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW 相同。这个框架包含当前分区中所有从开始到目前行所有数据。

例如，下面的查询对每一个店员进行订单价钱的大小排序：

```
SELECT orderkey, clerk, totalprice,
rank() OVER (PARTITION BY clerk ORDER BY totalprice DESC) AS rnk
FROM orders ORDER BY clerk, rnk
```

聚合函数

所有 聚合函数 可以通过添加 OVER 子句来作为窗口函数使用。这些聚合函数会基于当前 滑动窗口内的数据行计算每一行数据。

例如，下面的查询语句为每个店员计算每天的滚动订单价格总和：

```
SELECT clerk, orderdate, orderkey, totalprice,
sum(totalprice) OVER (PARTITION BY clerk ORDER BY orderdate) AS rolling_sum
FROM orders ORDER BY clerk, orderdate, orderkey
```

排序函数

cume_dist() → bigint

返回一组数值中每个值的累计分布。 结果返回的是按照窗口分区下窗口排序后的数据集下，当前行前面包括当前行数据的行数。因此，排序中任何关联 值均会计算成相同的分布值。

dense_rank() → bigint

返回一组数值中每个数值的排名。这个函数与 rank() ，相似，除了关联值不会产生顺序上的空隙。

ntile(n) → bigint

为每个窗口分区的数据分裂到桶号从 1 到最大 n 的 n 个桶中。桶号值最多间隔是 1 。 如果窗口分区中的数据行数不能均匀的分到每一个桶中，则剩余值将每一个桶分一个，从第一个桶开始。

比如，6 行数据和 4 个桶， 最后桶的值如下所示： 1 1 2 2 3 4

percent_rank() → bigint

返回数据集中每个数据的排名百分比。结果是根据 (r - 1) / (n - 1) 其中 r 是由 rank() 计算 的当前行排名， n 是当前窗口分区内总的行数。

rank() → bigint

返回数据集中每个值的排名。排名值是根据当前行之前的行数加1，不包含当前行。因此，排序的关联值可能产生顺序上的空隙。 这个排名会对每个窗口分区进行计算。

row_number() → bigint

为每行数据返回一个唯一的顺序的行号，从1开始，根据行在窗口分区内的顺序。

值函数

first_value(x) → [与输入类型相同]

返回窗口内的第一个值。

last_value(x) → [与输入类型相同]

返回窗口内的最后一个值。

nth_value(x, offset) → [与输入类型相同]

返回窗口内指定偏移的值。偏移量从 1 开始。如果偏移量是null或者大于窗口内值的个数，返回null。 如果偏移量为0或者负数，则会报错。

lead(x[, offset[, default_value]]) → [与输入类型相同]

返回窗口内，距当前行后偏移 offset 的值。偏移量起始值是 0, 就是指当前数据行。偏移量可以是 标量表达式 。默认 offset 是 1 。如果偏移量的值是null或者大于窗口长度， default_value 会被返回， 如果没有指定则会返回 null 。

lag(x[, offset[, default_value]]) → [与输入类型相同]

返回窗口内，距当前行前偏移 offset 的值。偏移量起始值是 0, 就是指当前数据行。偏移量可以是 标量表达式 。默认 offset 是 1 。如果偏移量的值是null或者大于窗口长度， default_value 会被返回， 如果没有指定则会返回 null 。

DDL相关错误码

范围	说明
18000 ~ 18599	DDL CREATE语句用户错误
18600 ~ 18799	DDL ALTER语句用户错误
18800 ~ 18899	DDL DROP语句用户错误
19000 ~ 19599	DDL CREATE语句系统错误
19600 ~ 19799	DDL ALTER语句系统错误
19800 ~ 19899	DDL DROP语句系统错误

DDL CREATE语句用户错误

错误码	错误信息	解决办法

18000	DenyAccessException:You do not have[xxx] access to resource[xxx]	无对特定数据库资源的特定操作权限，请确认，或按需要进行赋权限操作。
18001	Not support CREATE DATABASE in db: schema	无法在该数据连接上进行CREATE DATABASE操作，请检查所用数据库连接是否正确。
18002	非ADS user导致的失败信息。	操作必须由ADS user进行，请确认当前使用的UMM账号是ADS用户账号
18003	NA	NA
18004	Illegal options in CREATE DATABASE:	非法的CREATE DATABASE命令选项参数，请参考建库文档进行修改。
18005	NA	NA
18006	xxx database already exists.	目标数据库已经存在，请确认数据库是否重名。
18007	Target database does not exist.	目标数据库不存在，请确认数据库名是否正确。
18008	Table group ‘schema.tablegroup’ already exists.	目标表组不存在，请确认表组名是否正确。
18009	IllegalParameterException: parameterName: xxx, message: xxx	DDL语句参数不正确，请参考DDL文档，或进一步联系技术支持。
18010	参数值非法的详细错误信息。	DDL语句参数值非法，请按详细提示信息修改，或进一步联系技术支持。
18011	SUBPARTITION is not supported by DIMENSION table.	维度表不支持二级分区，请修改。
18012	TABLEGROUP must not be specified for DIMENSION table.	维度表建表语句不能指定表组，维度表均归属于系统默认维度表组，请修改。
18013	The minimum PARTITION NUM allowed for fact table is xxx, but xxx was defined.	不满足分区表的最小分区数定义，请修改。
18014	Table ‘table’ already exists.	目标表已经存在，请确认表是否重名。
18015	xxx is the dimension table group, could not be used.	自定义表组不能使用系统默认维度表组名，请修改。
18016	NA	NA
18017	Exceed the tables limitation (xxx) of database ‘xxx’	目标数据库下的表数量已经到上限，不可继续建表，请联系技术支持。

18018	Exceed the tables limitation (xxx) of table group ‘xxx’	目标数据库表组下的表数量已经到上限，不可继续建表，请联系技术支持。
18019	NA	NA
18020	Duplicated column definition	DDL中有重复列定义，请修改。
18021	There are xxx columns, which is not in the valid range: 1 to xxx	建表语句列数超限，请联系技术支持。
18022	Partition column does not exist	DDL中定义的分区列名不在定义的列中，请检查并修改。
18023	Column type is invalid	列数据类型非法，请参考DDL文档中关于支持的数据类型进行修改。
18024	DDL语句语法错误的详细信息。	DDL语句语法错误，请参考DDL文档进行修改，或进一步联系技术支持。
18025	No database selected.	相关操作未找到目标数据库，请检查数据库连接是否正确包含目标数据库信息，或者操作是否指定目标数据库。
18026	Table group should be created first.	建分区表时指定的表组不存在，需先建表组。
18027	相关命名的详细错误信息。	数据库对象命名错误，请按照提示进行修改，或进一步联系技术支持。
18028	Target sub-partition column does not exist.	指定的二级分区列列名不在定义的列中，请检查并修改。
18029	Only LONG/BIGINT is allowed for subpartition column data type.	二级分区列数据类型只能为LONG/BIGINT，请修改。
18030	Sub-partition number is illegal.	二级分区数非法，请修改。
18031	Exceed maximum user database limitation: xxx, user: xxx	用户下数据库总数已经达到上限，不可继续建库，请联系技术支持。
18032	No partition information provided for none dimension table.	建分区表时缺少分区信息子句，请参考DDL文档进行修改。
18033	Not support create table without tablegroup	建分区表时缺少表组子句，请参考DDL文档进行修改。
18034	Wrong table options	建表指定的options子句中参数错误，请参考DDL文档进行修改。
18035	PRIMARY KEY does not exist for real time table.	建实时表时缺少主建定义，请参考DDL文档进行修改。

18036	PRIMARY KEY does not contain partition column “xxx” for real time partition table.	建实时表时，主建定义中未包含分区列，请修改。
18037	Subpartition must not exist for real time table.	实时表暂不支持二级分区，请联系技术支持。
18038	NA	NA
18039	Table option LIFECYCLE is not allowed for non real time table.	建非实时表时不能指定LIFECYCLE参数，请修改。
18040	Old version of CREATE TABLE statement is blocked, please use the new version.	老版本建表语句被禁止，请参考DDL文档，使用当前的建表语句语法。
18041	CLUSTER BY column does not exist	建表语句中，CLUSTER BY指定的列不在定义的列中，请检查并修改。
18042	It is not allowed to create real time table on database: xxx	禁止在目标数据库中建实时表，请联系技术支持。
18043	Index column was not in the column definition list	索引列不在定义的列中，请检查并修改。
18044	Duplicated index name	DDL中有重复索引定义，请修改。
18045	The maximum PARTITION NUM allowed for fact table is xxx, but xxx was defined.	建分区表的分区数超过上限，请修改，或进一步联系技术支持。
18046	There are xxx sub partition number in definition, which is not in the valid range: x to x	二级分区数超过上限，请修改，或进一步联系技术支持。
18047	Invalid default value for xxx	列定义中的默认值表达式非法，请修改。
18048	MULTIVALUE/TIME/DATE is not allowed for partition column data type.	多值列、TIME、DATE类型的列不能作为分区列，请修改。
18049	Subpartition column must be one of the primary key columns.	建实时表时，二级分区列必须是主建列之一，请修改。
18050	相关功能还不支持的详细信息。	请参考提示的详细信息，或进一步联系技术支持。
18051	CTAS_LOAD_DATA_TIMEOUT schema=xxx table=xxx	CTAS执行的LOAD DATA阶段超时，请重试，或进一步联系技术支持。
18052	CTAS_META_CHECK_THREAD_ERROR message=xxx	CTAS执行的元数据校验阶段超时，请重试，或进一步联系技术支持。

18053	CTAS_LOAD_DATA_FAILED schema=xxx table=xxx jobState=xxx	CTAS执行的LOAD DATA阶段失败，请重试，或进一步联系技术支持。
18054	CTAS_SELECT_SQL_ANALYZE_ERROR schema=xxx table=xxx message=xxx	CTAS语句的SELECT子句语法分析失败，请检查语法，或进一步联系技术支持。
18055	CTAS_INSERT_THREAD_ERROR message=xxx	CTAS执行的INSERT阶段失败，请重试，或进一步联系技术支持。
18056	CTAS_INSERT_TIMEOUT schema=xxx table=xxx timeoutDuration=xxx	CTAS_INSERT_THREAD_ERROR message=xxx
18057	SUBPARTITION column conflicts with PARTITION column	二级分区列与一级分区列冲突，请修改。
18058	No column definition.	DDL语句无列定义，请参考DDL文档修改。
18059	Illegal table partition type: xxx, only HASH is allowed.	建分区表时，错误的分区类型，目前仅支持HASH分区类型，请修改。
18060	Illegal table sub partition type: xxx, only LIST is allowed.	建二级分区表时，错误的二级分区类型，目前仅支持LIST分区类型，请修改。
18061	Invalid index type: xxx	不支持的索引类型，请参考DDL文档修改。
18062	CTAS_RETRIEVE_COL_DEF_FAIL message=xxx	CTAS执行的列定义获取阶段失败，请检查SELECT部分的语法，或重试，或进一步联系技术支持。
18063	Illegal worker label value: xxx, value pattern should be “read:write”; Illegal worker ratio value: xxx, value pattern should be “number:number”	读写分离读写比例参数错误，请参考DDL文档修改。
18064	EXECUTE_CREATE_CACHE_TABLE_ERROR message=xxx query=xxx	建cache表失败，请参考DDL文档检查语法，或进行重试，或进一步联系技术支持。
18065	Lack options in CREATE DATABASE: xxx, options should be contained both ‘worker_labels’ and ‘worker_ratios’ or not.	建读写分离库时，读写分离参数错误，请根据提示和DDL文档修改。
18067	NA	NA
18068	Wrong format for external catalog properties.	外部数据源catalog创建语句的properties子句格式错误，请参考DDL文档修改。

18069	xxx external catalog already exists.	外部数据源catalog已经存在，请确认catalog是否重名。
18070	NA	NA
18071	Could not use the existing schema name	外部数据源catalog名字不能和已存在的数据库重名，请修改。
18072	Table option UPDATETYPE is unknown. Only ‘realtime’ and ‘batch’ are supported.	表选项UPDATETYPE只支持realtime和batch，请修改。
18073	Invalid FULLTEXT index column data type xxx of column xxx, VARCHAR was expected.	FULLTEXT全文索引列只支持VARCHAR类型，请修改。
18074	Database name length exceeds the limitation of ‘xxx’，current length is ‘xxx’.	CREATE DATABASE指定的目标数据库名长度超限，请修改。
18075	Illegal ‘dbNameBytesMaxLength’ zk value: xxx, value type should be numeric.	CREATE DATABASE的目标数据库名长度限制ZK配置数据类型错误，请提工单。
18076	Target view ‘xxx’ already exist.	CREATE VIEW的目标视图已经存在，请确认或修改。
18077	Table with the same name ‘xxx.xxx’ already exists.	CREATE VIEW的目标视图名和已经存在表名冲突，请确认或修改。
18100	Realtime table count exceeds limit in this database: xxx, ecu type: xxx	目标数据库实时表数量超过了ecu的上限，无法继续建实时表，请联系技术支持。

DDL ALTER语句用户错误

错误码	错误信息	解决办法
18600	DenyAccessException:You do not have[xxx] access to resource[xxx]	无对特定数据库资源的特定操作权限，请确认，或按需要进行赋权限操作。
18601	NA	NA
18602	DDL语句语法错误的详细信息。	DDL语句语法错误，请参考DDL文档进行修改，或进一步联系技术支持。
18603	No database selected.	相关操作未找到目标数据库，请检查数据库连接是否正确包含目标数据库信息，或者操作是否指定目标数据库。
18604	目标数据库对象不存在的详细信息。	目标数据库对象不存在，请检查。

18605	参数值非法的详细错误信息。	DDL语句参数值非法，请按详细提示信息修改，或进一步联系技术支持。
18606	参数值非法的详细错误信息。	DDL语句参数值非法，请按详细提示信息修改，或进一步联系技术支持。
18607	AlterRepeatException: columnName: xxx repeat.	列名重复，请确认目标表无该名字的列。
18608	AlterRepeatException: indexName: xxx repeated.	索引目标列已经定义过索引，不能再添加其他索引。
18609	NA	NA
18610	Column type is invalid:	列数据类型非法，请参考DDL文档中关于支持的数据类型进行修改。
18611	DROP COLUMN is not supported yet.	暂不支持ALTER TABLE DROP COLUMN。
18612	Illegal index type:	不支持的索引类型，请参考DDL文档修改。
18613	Do not allow to add column to real time table:	暂不支持对实时表进行加列，请联系技术支持。
18614	Invalid default value for xxx	列定义中的默认值表达式非法，请修改。
18615	Add PK column is not allowed for real time table.	实时表不允许通过ALTER加主建列，请修改。
18616	Add virtual column is not allowed for real time table.	实时表不允许通过ALTER加虚拟列，请修改。
18617	非法的ALTER语句参数的详细信息。	ALTER语句参数非法，请根据提示参考DDL文档修改。
18618	Index “xxx” already exist on xxx.xxx	索引名重复，请确认目标表上无该名字的
18619	The operation is not supported yet.	不支持的ALTER操作，请联系技术支持。
18620	Resize Operation is not allowed in this database.	当前用户不允许对目标数据库进行变更资源操作，请联系技术支持。
18621	There are xxx sub partition number in definition, which is not in the valid range: x to x	二级分区数超过上限，请修改，或进一步联系技术支持。
18622	Target is not sub partition table.	目标表不是二级分区表，无法进行二级分区数调整操作。
18623	Illegal frontnode_rw_instance_ratio value: xxx, value pattern	读写分离读写比例参数错误，请参考DDL文档修改。

	should be “number:number”	

DDL DROP语句用户错误

错误码	错误信息	解决办法
18800	No database selected.	相关操作未找到目标数据库，请检查数据库连接是否正确包含目标数据库信息，或者操作是否指定目标数据库。
18801	Database ‘xxx’ was not found.	目标数据库不存在，请确认。
18802	Can not drop non-empty database.	不能DROP非空的数据库，先DROP完库中的表，之后再DROP该库。
18803	Could not explicitly drop dimension group ‘xxx’.	不能DROP系统默认的维度表组。
18804	Target table group does not exist:	目标表组不存在，请确认。
18805	Can not drop non-empty tablegroup.	不能DROP非空的表组，先DROP完表组中的表，之后再DROP该表组。
18806	Illegal drop target type, only DATABASE/TABLEGROUP/TABLE/EXTERNAL CATALOG is allowed.	DROP的目标数据库对象类型非法，只允许DROP DATABASE/TABLEGROUP/TABLE/EXTERNAL CATALOG。
18807	Target table does not exist:	目标表不存在，请确认。
18808	Drop failed. xxx message=xxx	删除cache表失败，请重试，或进一步联系技术支持。
18809	Drop external catalog failure due to:	删除外部数据源catalog失败，请重试，或进一步联系技术支持。

DDL CREATE语句系统错误

错误码	错误信息	解决办法
19000	LOCK_SERVICE_ERROR message=Acquire lock failed.	DDL操作获取ZK锁失败，请稍后重试，或进一步联系技术支持。
19001	NA	NA
19002	NO_ALB_INSTANCE message=No available ALB load balancer instance found.	创建目标库时，未找到集群可用的load balancer实例，无法创建，请联系技术支持。
19003	NA	NA

19004	Illegal SLB VIP front end port: xxx, the valid port range is xxx-xxx	在元数据中指定VIP/PORT创建目标库时，指定的PORT不在合理范围内，请联系技术支持。
19005	ALB_OPERATION_FAIL message=xxx	创建目标库时，创建VIP步骤失败，请联系技术支持。
19006	Add DNS resolve record failed.	创建目标库时，绑定DNS域名步骤失败，请联系技术支持。
19007	NA	NA
19008	NA	NA
19009	NA	NA
19010	分配数据库对象ID失败的详细信息。	为创建目标数据库对象分配ID失败，请稍后重试，或进一步联系技术支持。
19011	CREATE statement is blocked.	DDL CREATE语句被禁止，请联系技术支持。
19012	CREATE DATABASE statement is disabled.	DDL CREATE DATABASE语句被禁止，请提交工单。
19013	CREATE TABLEGROUP statement is disabled for database ‘xxx’.	目标数据库的DDL CREATE TABLEGROUP语句被禁止，请提交工单。
19014	CREATE TABLE statement is disabled for database ‘xxx’.	目标数据库的DDL CREATE TABLE语句被禁止，请提交工单。
19015	CREATE EXTERNAL CATALOG statement is disabled for database ‘xxx’.	目标数据库的DDL CREATE EXTERNAL CATALOG语句被禁止，请提交工单。
19599	CREATE操作的其他类型失败的详细信息。	CREATE操作遇到其他类型失败，请联系技术支持。

DDL ALTER语句系统错误

错误码	错误信息	解决办法
19600	NA	NA
19601	ALTER statement is blocked.	DDL ALTER语句被禁止，请联系技术支持。
19602	ALTER DATABASE statement is disabled.	DDL ALTER DATABASE语句被禁止，请提交工单。
19603	ALTER TABLEGROUP statement is disabled for database ‘xxx’.	目标数据库的DDL ALTER TABLEGROUP语句被禁止，请提交工单。
19604	ALTER TABLE statement is disabled for database ‘xxx’.	目标数据库的DDL ALTER TABLE语句被禁止，请提交工单。

19605	Invalid FULLTEXT index column data type xxx of column xxx, VARCHAR was expected.	FULLTEXT全文索引列只支持VARCHAR类型，请修改。
19699	ALTER操作的其他类型失败的详细信息。	ALTER操作遇到其他类型失败，请联系技术支持。

DDL DROP语句系统错误

错误码	错误信息	解决办法
19800	NA	NA
19801	DROP statement is blocked.	DDL DROP语句被禁止，请联系技术支持。
19802	DROP DATABASE statement is disabled.	DDL DROP DATABASE语句被禁止，请提交工单。
19803	DROP TABLEGROUP statement is disabled for database ‘xxx’.	目标数据库的DDL DROP TABLEGROUP语句被禁止，请提交工单。
19804	DROP TABLE statement is disabled for database ‘xxx’.	目标数据库的DDL DROP TABLE语句被禁止，请提交工单。
19805	DROP EXTERNAL CATALOG statement is disabled for database ‘xxx’.	目标数据库的DDL DROP EXTERNAL CATALOG语句被禁止，请提交工单。
19899	DROP操作的其他类型失败的详细信息。	DROP操作遇到其他类型失败，请联系技术支持。

DML相关错误码

范围	说明
0 ~ 20999	DML用户错误
30000 ~ 60000	DML系统错误

DML用户错误

错误码	错误信息	解决办法
1044	Can not use this command here !	不支持的MySQL协议命令，请确认。
1045	其他语句执行异常	请提交工单。
1046	No database specified in FROM table	查询未能找到目标表的归属DB，请检查数据库连接使用的DB，或者直接为目标表前面加上DB前缀。

1143	Access deny for accessing database ‘schema’ from this node.	请联系技术支持，确认是否进行了DB节点绑定操作。
1146	Table xxx doesn’t exist.	请确认目标表xxx在当前连接和访问的DB下确实存在。
1147	No COLUMN:column found in TABLE table	请确认目标表table确实包含列column。
1148	NA（查询语句中Column分析的相关错误）	请联系技术支持查看执行的SQL。
1235	SQL/dumping result rows limit exceed : limit_max; SQL where expression contains/in items count exceed limit: multi_value_limit	1.查询语句中LIMIT子句值超过配置允许的最大值，请修改；2.查询语句中CONTAINS/IN子句中的项目数超过配置允许的最大值，请修改减少项目数。
1236	SQL feature NOT supported yet: xxx	查询语句中xxx相关语法功能不支持。
20000	[USER ERROR] Invalid SQL.	SQL语法异常，请检查并修改，或进一步联系技术支持。
20001	[USER ERROR] Invalid data value type.	SQL中有数据的值和其对应的数据列类型不匹配，请根据提示修改。
20002	[USER ERROR] Invalid hint.	下发到COMPUTENODE的查询HINT非法，请检查SQL，或进一步联系技术支持。
20003	[USER ERROR] Invalid UDF.	COMPUTENODE不支持的UDF，请确认，或进一步联系技术支持。
20004	[USER ERROR] Query items exceed limitation.	COMPUTENODE计算超限，请优化SQL，根据业务场景提高SQL的筛选率，或进一步联系技术支持。
20005	[USER ERROR] Query must GROUP BY column.	SQL语句错误，按照提示，必须GROUP BY提示的列。
20006	[USER ERROR] Query misses join condition.	SQL语句错误，join子句缺少ON条件。
20007	[USER ERROR] Query misses join index.	已经不再出现该错误，如出现，请联系技术支持。
20008	NA	NA
20009	[USER ERROR] Unsupported query syntax.	SQL语法错误，请参考SQL语法文档，进行修改。
20010	NA	NA
20011	[USER ERROR] fact table partition column was not provided for INSERT statement.	目标表为分区表时，INSERT语句的列集合中，必须包含分区列。

20012	[USER ERROR] invalid column value:	非法的列值，列值和列类型不匹配，请修改。
20013	INSERT语句语法错误的详细信息。	INSERT语句语法错误，请按照提示修改。
20014	INSERT statement is blocked.	INSERT语句被禁止，请联系技术支持。
20015	INSERT执行失败的详细信息。	请重试，或进一步联系技术支持。
20016	[USER ERROR] Target table was not found at this time.	已经不再出现该错误，如出现，请联系技术支持。
20017	Delete statement is blocked.	DELETE语句被禁止，请联系技术支持。
20018	DELETE语句语法错误的详细信息。	DELETE语句语法错误，请按照提示修改。
20019	DELETE执行失败的详细信息。	请重试，或进一步联系技术支持。
20020	[USER ERROR] Target table was not found at this time.	已经不再出现该错误，如出现，请联系技术支持。
20021	[USER ERROR] Target table has no primary key.	目标实时表没有主建列，请联系技术支持。
20022	[USER ERROR] Must and only contains AND joined EQUAL-TO predicates of all columns within the primary key:	严格的DELETE where条件检查，请联系技术支持更改系统配置。
20023	[USER ERROR] column was not found: column=	INSERT语句中，目标列不存在，请修改。
20024	[USER ERROR] column was not found: column=	严格的DELETE where条件检查时，DELETE语句中，目标列不存在，请修改，并联系技术支持更改系统配置。
20025	[USER ERROR] column value is invalid: column=xxx, type=xxx, value=xxx	严格的DELETE where条件检查时，DELETE语句中，目标列的值和类型不匹配，请修改，并联系技术支持更改系统配置。
20026	Real time data FLUSH is blocked.	FLUSH语句被禁止，请联系技术支持。
20027	Real time data MERGE is blocked.	MERGE语句被禁止，请联系技术支持。
20028	INSERT is not allowed when the real time table is not ready.	已经不再出现该错误，如出现，请联系技术支持。
20029	DELETE is not allowed when the real time table is not ready.	已经不再出现该错误，如出现，请联系技术支持。

20030	[USER ERROR] Target table is not real time table.	目标表不是实时表，无法INSERT。
20031	[USER ERROR] Target table is not real time table.	目标表不是实时表，无法INSERT。
20032	[USER ERROR] column value lists do not match.	INSERT语句中列集合的列表和VALUES子句中值的个数不匹配，请确认修改。
20033	Insert failed due to encoding exception:	INSERT语句中数据编码有问题，请确认，或进一步联系技术支持。
20034	[USER ERROR] Target table has no primary key.	目标实时表没有主建列，请联系技术支持。
20035	[USER ERROR] miss primary key column:	INSERT语句的列集合未包含所有的主建列，请修改。
20036	RT_DATA_WRITE_TIMEOUT schema=xxx table=xxx	INSERT执行超时，请重试，或进一步联系技术支持。
20037	[USER ERROR] null value is not allowed for NOT NULL column: column=xxx, value=xxx	INSERT语句中，NOT NULL的列插入了NULL值，请修改。
20038	[USER ERROR] Illegal argument.	COMPUTENODE执行下发SQL时出现参数错误，请参考SQL语法文档，或进一步联系技术支持。
20039	[USER ERROR] Unsupported query operation.	COMPUTENODE执行下发SQL时出现语法错误，请参考SQL语法文档，或进一步联系技术支持。
20040	No database selected.	请检查是否在数据库连接中指定了目标数据库。
20041	No database selected.	请检查是否在数据库连接中指定了目标数据库。
20042	COUNT DISTINCT on non-partitioning column exceeds the partition data limitation: xxx, part is xxx	进行非分区列COUNT DISTINCT操作时，单分区返回的明细数据超过了限制，请优化SQL，根据业务场景提高SQL的筛选率，或进一步联系技术支持。
20043	UDF_SYS_ROWCOUNT exceeds the partition data limitation: xxx, part is xxx	SQL中包含UDF_SYS_ROWCOUNT函数时，单分区返回的明细数据超过了限制，请优化SQL，根据业务场景提高SQL的筛选率，或进一步联系技术支持。
20044	[USER ERROR] Must contains WHERE expression.	DELETE语句中未写WHERE子句，请修改。
20045	[USER ERROR] sub select	已经不再出现该错误，如出现

	condition was not support for delete statement.	，请联系技术支持。
20046	SQL feature NOT supported yet: GROUP_CONCAT without grouping tables' partition columns.	使用GROUP_CONCAT函数时，SQL语句必须GROUP BY所有目标表的分区列。
20047	SQL feature NOT supported yet: COUNT(DISTINCT x1, x2, ...) against mulitple non-partitioning columns	不支持对多个非分区列的COUNT DISTINCT操作。
20048	HAVING子句表达式非法的详细信息。	请参考SQL文档进行修改。
20049	SQL dump data selecting columns count exceed limit:	DUMP DATA语句中的SELECT列数超过上限，请限制并修改，或进一步联系技术支持。
20050	SQL selecting columns count exceed limit:	查询语句中的SELECT列数超过上限，请限制并修改，或进一步联系技术支持。
20051	NA	NA
20052	UDF_SYS_SUM exceeds the partition data limitation: xxx, part is xxx	SQL中包含UDF_SYS_SUM函数时，单分区返回的明细数据超过了限制，请优化SQL，根据业务场景提高SQL的筛选率，或进一步联系技术支持。
20053	INSERT is not allowed since the compute node disk is almost full: worker=xxx disk_used=xxx disk_size=xxx usage_ratio=xxx	COMPUTENODE实例节点存储空间到上限，无法继续INSERT数据，请联系技术支持。
20054	Unknown column xxx in 'ORDER BY clause'	ORDER BY子句中的列非法，请检查SQL并修改。
20055	SQL feature NOT supported yet: SELECT ... ORDER BY ... UNION/UNION ALL/INTERSECT/MINUS SELECT ... ORDER BY ...	不支持的UNION语法：UNION子查询中带ORDER BY子句。
20056	Unknown column xxx in 'GROUP BY clause'	GROUP BY子句中的列非法，请检查SQL并修改。
20057	NA	NA
20058	RT_UPN_ALTERTABLE_ERROR db=xxx table=xxx part=xxx message=xxx	向BUFFERNODE发ALTER TABLE语句时，遇到encoding异常，请联系技术支持。
20059	[USER ERROR] null value is not allowed for sub partitioning column: column=xxx, value=xxx	INSERT实时数据时，二级分区列对应的值不能为NULL，请修改。

20060	[USER ERROR] sub partitioning column value is out of range: column=xxx, type=xxx, value=xxx	INSERT实时数据时，二级分区列对应的值不在合法的范围，请修改。
20061	[USER ERROR] sub partitioning column value is invalid: column=xxx, type=xxx, value=xxx	INSERT实时数据时，二级分区列对应的值不合法，请修改。
20062	[USER ERROR] sub partition column was not provided for INSERT statement.	INSERT实时数据时，如果目标表是二级分区表，插入的列和值的集合必须包含二级分区列，请修改。
20063	[USER ERROR] sub partitioning column value is out of range: column=xxx, type=LONG/BIGINT, value=xxx	已经不再出现该错误，如出现，请联系技术支持。
20064	[USER ERROR] sub partitioning column value is invalid: column=xxx, type=LONG/BIGINT, value=xxx	已经不再出现该错误，如出现，请联系技术支持。
20065	[USER ERROR] sub partitioning column EQUAL-TO predicate was not provided.	已经不再出现该错误，如出现，请联系技术支持。
20066	ORDER BY item ‘xxx’ was not found in GROUP BY clause.	查询语句包含GROUP BY和ORDER BY子句时，ORDER BY中的列必须出现在GROUP BY子句中。
20067	SQL GROUP-BY column expected:	SQL语句错误，按照提示，必须GROUP BY提示的列。
20068	INSERT is not allowed since the compute node stops insert: worker=	COMPUTENODE设置了停止实时数据写入的标志，导致INSERT数据失败，请联系技术支持。
20069	[USER ERROR] Row expected exceeds limit.	查询语句的LIMIT子句（若不写，系统自动补上LIMIT子句，LIMIT默认值为10000）的值超过配置的上限，请限制并修改，或进一步联系技术支持。
20070	Only support single INSERT statement.	一次只能执行一个INSERT语句。
20071	执行INSERT FROM SELECT报错的详细信息。	请根据报错的详细信息，联系技术支持。
20072	Only TOP priority query is allowed.	只允许TOP优先级的查询，请联系技术支持。
20073	Exceeded max AND/OR	WHERE子句中AND/OR连接的

	combined predicate number:	谓词过滤条件数超过了允许的上限，请做限制。
20074	Reject execution of sql with key, statement: sql	包含特定关键字的查询被拒绝执行，请确认并联系技术支持，确认这些关键已经被配置用来过滤SQL。
20075	[USER ERROR] all primary key columns value are NULL: xxx	INSERT的数据记录中，不允许所有主键列全为NULL，请修改。
20076	[USER ERROR] sub partition keys count exceed the table limit for a duration (second): limit=xxx duration=xxx	在一个特定的周期内（默认为一天），目标表的INSERT数据的目标二级分区总数超限，默认为10个，请确认，或进一步提工单。
20077	[USER ERROR] delete statement count exceeds the table limit for a duration (second): limit=xxx duration=xxx	在一个特定的周期内（默认为一天），目标表的DELETE语句总数超限，默认为10000000，请确认，或进一步提工单。
20078	20078 No replica reported from COMPUTENODE.	MPP查询中COMPUTENODE节点未汇报路由信息，请提工单。
20079	[USER ERROR] Total row expected exceeds limit + offset. SQL (dumping) result rows limit offset (xxx) exceed : xxx	LIMIT m OFFSET n或者LIMIT n, m中，m + n超过了limitMax的查询限制，请确认修改查询LIMIT OFFSET的限制，或进一步提工单。
20080	Restart command is illegal. Please check its syntax.	Restart命令不合法，请检查该命令的语法。
20081	[USER ERROR] Insert record volume has exceeded the database limit for a duration (second): schema=xxx totalRecordCount=xxx recordlimit=xxx duration=xxx	在一个特定的周期内（默认为一天），目标DB的INSERT数据的总记录数超限，默认为2亿条 * COMPUTENODE节点数，请确认，或进一步提工单。
20082	[USER ERROR] Insert data size volume has exceeded the database limit for a duration (second): schema=xxx totaDataSize=xxx dataSizelimit=xxx duration=xxx	在一个特定的周期内（默认为一天），目标DB的INSERT数据（整个INSERT语句的字节数）的总大小超限，默认为100GB * COMPUTENODE节点数，请确认，或进一步提工单。
20500	MPP_QUERY_CANCELED message=Query has been canceled!	MPP查询被CANCEL，请重试。

DML系统错误

错误码	错误信息	解决办法

30000	[SYSTEM ERROR] Execution timeout.	查询执行超时，请稍后重试，或联系技术支持。
30001	[SYSTEM ERROR] Dump service initialization error.	COMPUTENODE初始化TFS DUMP操作失败，请重试，或联系技术支持。
30002	[SYSTEM ERROR] Dump service write error.	COMPUTENODE执行TFS DUMP操作失败，请重试，或联系技术支持。
30003	[SYSTEM ERROR] Table hash partition count is invalid:	目标分区表的分区数非法，请联系技术支持。
30004	[SYSTEM ERROR] Target hash partition number is invalid:	INSERT语句计算分区列hash值时失败，请联系技术支持。
30005	[SYSTEM ERROR] sub partitioning column is invalid in the meta.	二级分区列在元数据中不存在，请检查列名是否正确，或进一步联系技术支持。
30006	Invalid column value:	列对应的值非法，请检查列值是否符合列的数据类型。
30007	RT_ROUTER_ERROR table=xxx DB=xxx message=real time table is not ready.	COMPUTENODE还未汇报目标实时表的版本心跳，请稍后重试，或进一步联系技术支持。
30008	实时数据相关功能调用BUFFERNODE API报的详细错误信息。	根据详细错误信息查看BUFFERNODE的错误码表，进一步联系技术支持。
30009	NA	NA
30010	[SYSTEM ERROR] Column data was not found.	COMPUTENODE处理下发查询时找不到目标表的某个列的元数据，请稍后重试，或进一步联系技术支持。
30011	[SYSTEM ERROR] Table hash partition count is invalid:	目标分区表的分区数非法，请联系技术支持。
30012	[SYSTEM ERROR] Target hash partition number is invalid:	已经不再出现该错误，如出现，请联系技术支持。
30013	Cluster nodes route were not established completely: A. system is starting, please wait; B. COMPUTENODEs were GCing or have already crashed.	集群服务还未启动完成，请稍等；或有部分COMPUTENODE实例GC，请联系技术支持。
30014	BROADCAST_SUBQUERY_TIMEOUT schema=xxx process=xxx query_block=xxx	小表广播模式查询的子查询超时，请优化SQL，根据业务场景提高SQL的筛选率，或进一步联系技术支持。
30015	BROADCAST_SUBQUERY_ERROR schema=xxx process=xxx query_block=xxx	小表广播模式查询的子查询失败，请检查SQL，或进一步联系技术支持。

	message=xxx	
30016	CACHE_QUERY_FAILED message=xxx	Cache表查询失败，请检查SQL，或进一步联系技术支持。
30017	COMPUTENODE执行下发查询失败的详细信息。	请检查SQL，或进一步联系技术支持。
30018	CACHE_QUERY_FAILED message=xxx	Cache表查询失败，请检查SQL，或进一步联系技术支持。
30021	Query canceled in COMPUTENODE	查询被中断，请重试。
30022	Query exceeded scan limitation	查询超过扫描行数限制，请增加过滤条件缩小扫描范围，或进一步联系技术支持。
30023	Insert into buffernode failed	InsertIntoSelect插入到buffernode时报错，请重试，或进一步联系技术支持。
30100	HTTP client to MPP engine has gone.	MPP查询时，MPP client的HTTP连接断开，请重试，或进一步联系技术支持。
30101	MPP查询失败的详细信息。	请检查SQL，或进一步联系技术支持。
30102	MPP执行INSERT FROM SELECT失败的详细信息。	请检查INSERT FROM SELECT语句，或进一步联系技术支持。
30999	[SYSTEM ERROR] Other error.	其他查询错误，请联系技术支持。
31000	SQL planning partition routing error.	路由系统错误，联系技术支持。
32000	DATA_NOT_FOUND table_id=table_id db_id=db_id part_id=part_id	请检查目标表db_id.table_id是否上线完成。
50000	QUERY_WARNING, server is not ready now.	FRONTNODE节点尚未启动完成，请稍后再发起查询。
55000	Partition merging error: xxx	FRONTNODE节点进行分区结果合并操作异常，请联系技术支持。
60001	Must provide dump-header hint for cache table dump.	对CACHE表进行dump data操作时，必须通过/*+dump-header=[DUMP DATA ...]*/ SELECT ...的方式，请修改。
60002	Failed of getting upload id for xxx dump.	Dump data在获取upload id阶段失败，请参考详细信息，或进一步提工单。
60003	Must provide OSS dump parameters hints.	OSS dump必须在hint中提供OSS服务参数，请修改。

60004	No block ID available.	ODPS dump在get upload id阶段未获得block id，请提工单。
60005	ODPS dump error. Message=xxx	OSS dump异常，查看详细异常消息，或进一步提工单。

ACL相关错误码

范围	说明
18900 ~ 18999	ACL操作相关用户错误

ACL操作相关用户错误

错误码	错误信息	解决办法
18900	System privilege must equal with “.”	当ACL对象为System类型的resource，对象必须为.。
18901	Invalid resource type.	非法的ACL对象类型，必须为System或Table。
18902	xxx is not allowed, should be “/db_name/db_name.table_name/db_name.table_group_name/table_name/table_group_name”	非法的ACL对象名，请按照提示修改。
18903	ACL对象不存在的详细提示信息。	ACL对象不存在，请确认ACL对象名是否正确。
18904	ACL鉴权失败的详细信息。	ACL鉴权失败，请确认操作对象相关的ACL权限。
18905	Only support operation on connectted db.	只允许在该数据库对应的数据库连接上操作，请检查数据库连接，或进一步联系技术支持。
18906	非法账号相关的详细信息。	账号非法，请确认是否为合法的阿里云ADS账号。
18907	Creator must be the DB Admin when creating database for delegated user!	通过委托模式创建目标库，创建账号必须是DB Admin账号，请确认并联系技术支持。
18908	Do not support specific ACL user @ target host:	GRANT不支持特定user@特定host的模式。
18909	ALB ACL was only supported under SYSDB connection.	ALB相关的黑白名单GRANT操作只能在SYSDB连接下进行。
18910	VIP/port was not found for this database: xxx; ALB load balancer instance was not found for VIP: xxx	ALB相关的黑白名单GRANT操作时，VIP/PORT或相关load balancer对象不存在，请确认并修改。

18911	‘ALL’ privilege is invalid on column, only ‘SELECT’ is allowed.	列上不支持ALL权限。
18912	RAMException: Deny for having no privilege.	RAM子账号不是ADS账号，请确认，或提交工单。
18913	RAMException: Unknown RAM code [xxx]	未识别的RAM操作返回码，请提交工单。
18914	RAMException: Invalid sub user, should be like ‘RAM$parent_user_account:sub_user_account’ or ‘RAM$sub_user_account’	RAM子账号格式错误，请按照提示修改。
18915	NA	NA
18916	RAMException: Sub user account does not match with parent.	RAM子账号与父账号不匹配，请确认。
18917	RAMException: Sub user account does not have xxx privilege.	用户无RAM子账号权限，请确认，或进行子账号赋权。
18918	RAMException: Sub user account does not allow xxx	RAM子账号ACL功能未开启，请提交工单。
18919	RAMException: Sub user account name is invalid.	RAM子账号在元数据中的值非法，请提交工单。
18920	The account of database owner is not allowed to be removed.	创建数据库的账号不能被删除。
18921	AclException:GRANT_ERROR	授权错误，请提交工单。
18922	AclException:REVOKE_ERROR	回收权限错误，请提交工单。

系统相关错误码

范围	说明
39900 ~ 39949	系统操作相关用户错误
39950 ~ 39999	系统操作相关系统错误

系统操作相关用户错误

错误码	错误信息	解决办法
39900	Wrong parameter for query, only SYNCCACHE [size=new_cache_size_in_MB] is allowed.	SYNCCACHE语句语法错误，请修改。
39901	Wrong parameter for query,	CLEARCACHE语句语法错误

	CLEARCACHE command format is “CLEARCACHE db=schema tablegroup=table_group” or “CLEARCACHE db=schema table=table”	，请修改。
39902	Wrong parameter for query, FLUSH command format is “FLUSH db=schema table=table timeout=timeout_duration_ms”	FLUSH语句语法错误，请修改。
39903	Wrong parameter for query, MERGE command format is “MERGE db=schema table=table”	MERGE语句语法错误，请修改。
39904	No target table for OPTIMIZE TABLE command/Wrong OPTIMIZE TABLE command, syntax should be “OPTIMIZE TABLE [dbname.]table_name1 [, [dbname.]table_name2]”	OPTIMIZE TABLE语句语法错误，请修改。
39905	MPP引擎执行SHOW CATALOGS相关的详细错误信息	请联系技术支持。
39906	Resource Manager返回的详细错误信息，包含Resource Manager自身的错误码和详细描述。	查看Resource Manager错误码表，或进一步联系技术支持。
39907	TABLE_NOT_FOUND schema=xxx table=xxx	相关操作的目标表不存在，请检查。
39908	Target table is not realtime table.	相关实时表操作的目标表不是实时表，请检查。
39910	Cannot find worker db information.	无法找到系统后台WorkerDB，请联系技术支持。
39911	Show tables command is only allowed to show tables under current database.	该命令只允许显示当前DB下的表。

系统操作相关系统错误

错误码	错误信息	解决办法
39950	SYNCCACHE操作详细的错误信息。	请联系技术支持。
39951	CLEARCACHE操作详细的错误信息。	请联系技术支持。
39952		请联系技术支持。

39953		请联系技术支持。
39954	RT_UPN_FLUSH_ERROR message=target table does not exist./result=xxx DBID=xxx TableID=xxx PartitionCount=xxx	实时表FLUSH操作失败，请联系技术支持。
39955	FLUSH_TIMEOUT message=Flush version check timeout.	实时表FLUSH操作超时，实时数据强制版本同步慢，导致超时，请联系技术支持。
39956	Flush version check error:	实时表FLUSH操作版本确认失败，请联系技术支持。
39957	RT_UPN_FLUSH_ERROR db=schema table=table part=part_num message=Flush failed due to exception:	实时表FLUSH操作中，向BUFFERNODE写入目标版本信息失败，请联系技术支持。
39958	RT_UPN_ALTERTABLE_ERROR db=schema table=table part=part_num message=Alter table failed due to exception:	实时表FLUSH操作中，向BUFFERNODE写入ALTER指令失败，请联系技术支持。
39959	SLB操作详细错误信息。	查看SLB错误码表，或进一步联系技术支持。
39960	SLB操作详细错误信息。	查看SLB错误码表，或进一步联系技术支持。
39961	SLB操作详细错误信息。	查看SLB错误码表，或进一步联系技术支持。
39962	DNS_CMD_SYNTAX_ERRROR message=Incorrect DNS command syntax, the correct syntax should be “DNS [ADD | DELETE] domain ip”.	请联系技术支持。
39963	Delete DNS resolve record failed.	DNS命令执行失败，请联系技术支持。
39964	DNS_PARM_ERROR message=xxx	DNS命令执行失败，请联系技术支持。
39965	ALB_OPERATION_FAIL message=All load balancers are fully loaded. Please extend with new load balancers.	SLB的负载均衡实例全部满载（每个实例的VIP到上限），请提交工单进行负载均衡实例扩容。
39966	INSTANCE_VPC_DNS_PROCESS_ERROR	vpc vip申请错误，请联系技术支持。
39967	SHOW PLANCACHE STATUS执行失败的详细原因信息。	SHOW PLANCACHE STATUS执行失败，请联系技术支持。
39968	SHOW PLANCACHE PLAN执	SHOW PLANCACHE PLAN执

	行失败的详细原因信息。	行失败，请联系技术支持。
39999		请联系技术支持。

DDL建库、建表对象保留字

数据库、表组、索引不能包含：

```
DEBUG,WARN,ERROR,__
```

数据库、表组、表、列名均不能等于：

保留字	
BEFORE	reserved
BINARY	reserved
BLOB	reserved
BOTH	reserved
CALL	reserved
CASCADE	reserved
CHANGE	reserved
CHAR	reserved
CHARACTER	reserved
CHECK	reserved
COLLATE	reserved
CONDITION	reserved
CONNECTION	reserved
CONSTRAINT	reserved
CONTINUE	reserved
CONVERT	reserved
CROSS	reserved
CURRENT_DATE	reserved
CURRENT_TIME	reserved
CURRENT_TIMESTAMP	reserved
CURRENT_USER	reserved
CURSOR	reserved
DATABASES	reserved

STATUS	reserved
DAY_HOUR	reserved
DAY_MICROSECOND	reserved
DAY_MINUTE	reserved
DAY_SECOND	reserved
DEC	reserved
DECIMAL	reserved
DECLARE	reserved
DEFAULT	reserved
DELAYED	reserved
DETERMINISTIC	reserved
DISTINCTROW	reserved
DIV	reserved
DOUBLE	reserved
DUAL	reserved
EACH	reserved
ELSEIF	reserved
ENCLOSED	reserved
ESCAPED	reserved
EXIT	reserved
FETCH	reserved
FLOAT	reserved
FOR	reserved
FORCE	reserved
FOREIGN	reserved
FUNCTION	reserved
GOTO	reserved
HOUR_MICROSECOND	reserved
HOUR_MINUTE	reserved
HOUR_SECOND	reserved
IF	reserved
INFILE	reserved
INT	reserved

INTEGER	reserved
ITERATE	reserved
KILL	reserved
LABEL	reserved
LEADING	reserved
LEAVE	reserved
LOAD	reserved
LOCALTIME	reserved
LOCALTIMESTAMP	reserved
LOCK	reserved
LONG	reserved
LOOP	reserved
MATCH	reserved
MINUTE_MICROSECOND	reserved
MINUTE_SECOND	reserved
MOD	reserved
MODIFIES	reserved
OPTIMIZE	reserved
OPTION	reserved
OUT	reserved
OUTFILE	reserved
PRECISION	reserved
RANGE	reserved
READ	reserved
REAL	reserved
REFERENCES	reserved
REGEXP	reserved
RELEASE	reserved
RENAME	reserved
REPEAT	reserved
REQUIRE	reserved
RESTRICT	reserved
RETURN	reserved

RLIKE	reserved
SCHEMA	reserved
SECOND+MICROSECOND	reserved
SEPARATOR	reserved
SMALLINT	reserved
SPATIAL	reserved
SPECIFIC	reserved
SQL	reserved
TINYINT	reserved
TINYBLOB	reserved
UNLOCK	reserved
UNSIGNED	reserved
USE	reserved
VARCHAR	reserved
WHILE	reserved
WRITE	reserved
XOR	reserved
YEAR_MONTH	reserved
AS	reserved
BY	reserved
DO	reserved
IS	reserved
IN	reserved
OR	reserved
ON	reserved
TO	reserved
ALL	reserved
AND	reserved
ANY	reserved
ADD	reserved
KEY	reserved
NOT	reserved
SET	reserved

ASC	reserved
TOP	reserved
END	reserved
DESC	reserved
DESCRIBE	reserved
LOAD DATA	reserved
DUMP DATA	reserved
ALL PRIVILEGES	reserved
CREATE DATABASE	reserved
CREATE SCHEMA	reserved
SHOW DATABASES	reserved
CREATE USER	reserved
GRANT OPTION	reserved
SYSTEM	reserved
INTO	reserved
IGNORE	reserved
LEFT	reserved
NULL	reserved
TRUE	reserved
LIKE	reserved
DROP	reserved
JOIN	reserved
FROM	reserved
OPEN	reserved
CASE	reserved
WHEN	reserved
THEN	reserved
ELSE	reserved
SOME	reserved
FULL	reserved
WITH	reserved
FALSE	reserved
ALTER	reserved

SHOW	reserved
RIGHT	reserved
TABLE	reserved
TABLES	reserved
WHERE	reserved
USING	reserved
UNION	reserved
GROUP	reserved
GRANT	reserved
BEGIN	reserved
INDEX	reserved
INDEXES	reserved
KEYS	reserved
INNER	reserved
LIMIT	reserved
OUTER	reserved
ORDER	reserved
REVOKE	reserved
DELETE	reserved
CREATE	reserved
UNIQUE	reserved
SELECT	reserved
OFFSET	reserved
EXISTS	reserved
HAVING	reserved
INSERT	reserved
UPDATE	reserved
VALUES	reserved
ESCAPE	reserved
WITHIN	reserved
COLUMN	reserved
COLUMNS	reserved
NULLS	reserved

FIRST	reserved
LAST	reserved
ROWS	reserved
RANGE	reserved
UNBOUNDED	reserved
PRECEDING	reserved
FOLLOWING	reserved
CURRENT	reserved
ROW	reserved
KEEP	reserved
EXPLAIN	reserved
PRIMARY	reserved
NATURAL	reserved
REPLACE	reserved
BETWEEN	reserved
INTERVAL	reserved
CONTAINS	reserved
TRUNCATE	reserved
DISTINCT	reserved
DATABASE	reserved
TABLEGROUP	reserved
IF EXISTS	reserved
INTERSECT	reserved
MINUS	reserved
OVER	reserved
PROCEDURE	reserved
ADDONINDEX	reserved
PARTITION BY	reserved
IF NOT EXISTS	reserved
PARTITION NUM	reserved
SUBPARTITION BY	reserved
SUBPARTITION NUM	reserved
PARTITION OPTIONS	reserved

SUBPARTITION OPTIONS	reserved
ENGINE	reserved
OPTIONS	reserved
OVERWRITE	reserved
PROPERTIES	reserved
DIMENSION	reserved
CLUSTERED BY	reserved
MODIFY COLUMN	reserved
SET CONFIG	reserved
GET CONFIG	reserved
SEPARATOR	reserved
KILL	reserved
CACHED	reserved

MPP查询语句保留字

The following table lists all of the keywords that are reserved in AnalyticDB MPP, along with their status in the SQL standard. These reserved keywords must be quoted (using double quotes) in order to be used as an identifier.

保留字	SQL:2016	SQL-92
ALTER	reserved	reserved
AND	reserved	reserved
AS	reserved	reserved
BETWEEN	reserved	reserved
BY	reserved	reserved
CASE	reserved	reserved
CAST	reserved	reserved
CONSTRAINT	reserved	reserved
CREATE	reserved	reserved
CROSS	reserved	reserved
CUBE	reserved	
CURRENT_DATE	reserved	reserved
CURRENT_TIME	reserved	reserved
CURRENT_TIMESTAMP	reserved	reserved

DEALLOCATE	reserved	reserved
DELETE	reserved	reserved
DESCRIBE	reserved	reserved
DISTINCT	reserved	reserved
DROP	reserved	reserved
ELSE	reserved	reserved
END	reserved	reserved
ESCAPE	reserved	reserved
EXCEPT	reserved	reserved
EXECUTE	reserved	reserved
EXISTS	reserved	reserved
EXTRACT	reserved	reserved
FALSE	reserved	reserved
FOR	reserved	reserved
FROM	reserved	reserved
FULL	reserved	reserved
GROUP	reserved	reserved
GROUPING	reserved	
HAVING	reserved	reserved
IN	reserved	reserved
INNER	reserved	reserved
INSERT	reserved	reserved
INTERSECT	reserved	reserved
INTO	reserved	reserved
IS	reserved	reserved
JOIN	reserved	reserved
LEFT	reserved	reserved
LIKE	reserved	reserved
LOCALTIME	reserved	
LOCALTIMESTAMP	reserved	
NATURAL	reserved	reserved
NORMALIZE	reserved	
NOT	reserved	reserved

NULL	reserved	reserved
ON	reserved	reserved
OR	reserved	reserved
ORDER	reserved	reserved
OUTER	reserved	reserved
PREPARE	reserved	reserved
RECURSIVE	reserved	
RIGHT	reserved	reserved
ROLLUP	reserved	
SELECT	reserved	reserved
TABLE	reserved	reserved
THEN	reserved	reserved
TRUE	reserved	reserved
UESCAPE	reserved	
UNION	reserved	reserved
UNNEST	reserved	
USING	reserved	reserved
VALUES	reserved	reserved
WHEN	reserved	reserved
WHERE	reserved	reserved
WITH	reserved	reserved