# MaxCompute

## 使用指南

为了无法计算的价值 | 阿里云

# 使用指南

# DataHub实时数据通道

重要提示：

- 目前 老版本 DataHub 已处于维护状态，不再接入新用户。使用老版本的用户可以参考之前的使用文档
- 自2016年11月21日起，新版本DataHub正式公测上线。

# 产品简介

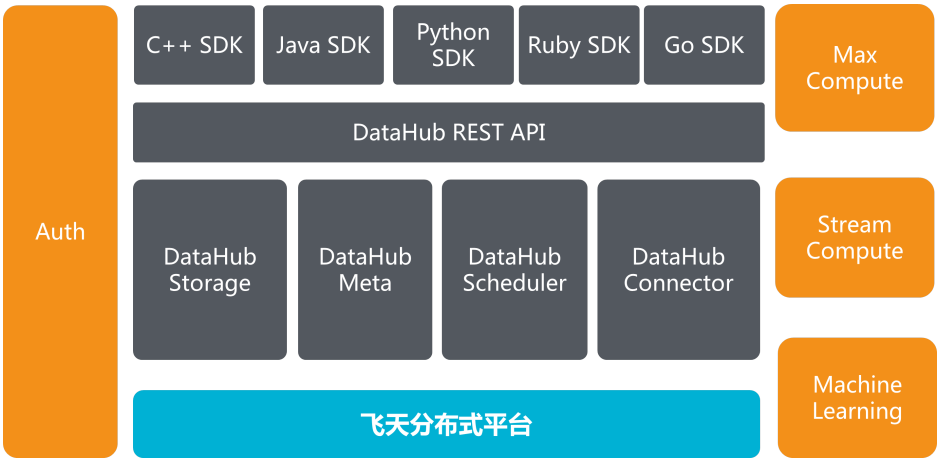
## DataHub基本介绍

DataHub是 MaxCompute 提供的流式数据处理(Streaming Data)服务，它提供流式数据的发布 (Publish)和订阅 (Subscribe)的功能，让您可以轻松构建基于流式数据的分析和应用。DataHub 可以对各种移动设备，应用软件，网站服务，传感器等产生的大量流式数据进行持续不断的采集，存储和处理。用户可以编写应用程序或者使用流计算引擎来处理写入到 DataHub 的流式数据比如实时web访问日志、应用日志、各种事件等，并产出各种实时的数据处理结果比如实时图表、报警信息、实时统计等。

DataHub基于阿里云自研的飞天平台，具有高可用，低延迟，高可扩展，高吞吐的特点。DataHub 与阿里云流计算引擎 StreamCompute 无缝连接，用户可以轻松使用SQL进行流数据分析。

DataHub 也提供流式数据归档的功能，支持流式数据归档进入MaxCompute(原ODPS)。

系统整体功能图




# 产品优势

## 高吞吐

最高支持单主题(Topic)每日T级别的数据量写入，每个分片(Shard)支持最高每日8000万Record级别的写入量。

## 实时性

通过 DataHub ，您可以实时的收集各种方式生成的数据并进行实时的处理，对您的业务产生快速的响应。

## 易用性

DataHub 提供丰富的SDK包，包括C++, JAVA, Pyhon, Ruby, Go等语言。DataHub服务也提供Restful API规范，您可以用自己的方式实现访问接口。除了SDK以外，DataHub 还提供一些常用的客户端插件，包括：Fluentd，LogStash，Flume等。您可以使用这些客户端工具往 DataHub 里面写入流式数据。DataHub 同时支持强Schema的结构化数据和无类型的非结构化数据，您可以自由选择。

## 高可用

服务可用性不低于99.999%。规模自动扩展，不影响对外服务；数据持久性不低于99.999%。数据自动多重冗余备份。

## 动态伸缩

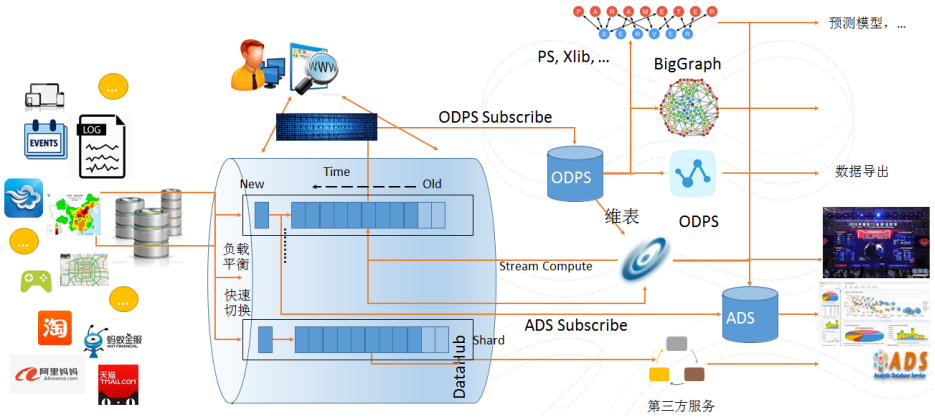
每个主题(Topic)的数据流吞吐能力可以动态扩展和减少，最高可达到每主题256000 Records/s的吞吐量。

## 高安全性

提供企业级多层次安全防护，多用户资源隔离机制；提供多种鉴权和授权机制及白名单、主子账号功能。

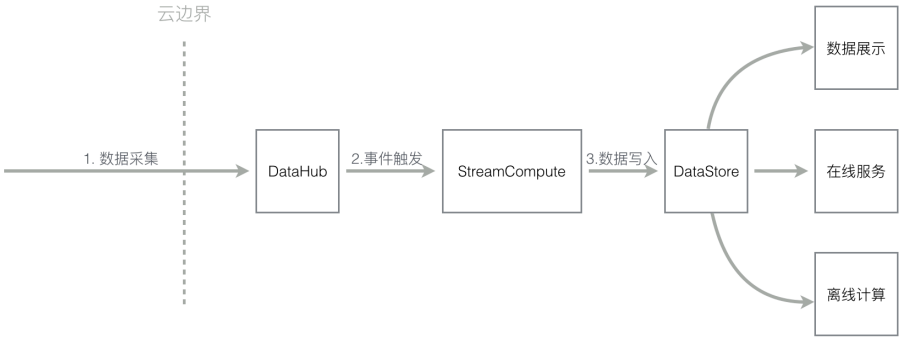
# 使用场景

DataHubR作为一个流式数据处理服务，结合阿里云众多云产品，可以构建一站式的数据处理服务。




## 流计算StreamCompute

StreamCompute是阿里云提供的流计算引擎，提供使用类SQL的语言来进行流式计算。DataHub 和StreamCompute无缝结合，可以作为StreamCompute的数据源和输出源。




## 流处理应用

用户可以编写应用订阅DataHub中的数据，并进行实时的加工，把加工后的结果输出。用户可以把应用计算产生的结果输出到DataHub中，并使用另外一个应用来处理上一个应用生成的流式数据，来构建数据处理流程的DAG。

## 流式数据归档

用户的流式数据可以归档到 MaxCompute（原ODPS）中。用户通过创建DataHub Connector，指定相关配

置，即可创建将Datahub中流式数据定期归档的同步任务。

# 名词解释

| 名词 | 解释 |
| --- | --- |
| Project | 项目（Project）是DataHub数据的基本组织单元,下面包含多个Topic。值得注意的是，DataHub的项目空间与MaxCompute的项目空间是相互独立的。用户在MaxCompute中创建的项目不能复用于DataHub，需要独立创建。 |
| Topic | Topic是 DataHub 订阅和发布的最小单位，用户可以用Topic来表示一类或者一种流数据。 更多详情请参考：Project及Topic数量限制。 |
| Topic Lifecycle | 表示一个Topic中写入数据在系统中可以保存的最长时间，以天为单位 |
| Shard | Shard表示对一个Topic进行数据传输的并发通道，每个Shard会有对应的ID。每个Shard会有多种状态：Opening - 启动中，Active - 启动完成可服务。每个Shard启用以后会占用一定的服务端资源，建议按需申请Shard数量。 |
| Shard Hash Key Range | 每个Shard都有的属性，包括开始和结束的Key范围，写入数据的时候具有相同Key的数据会落到同一个Shard上。对一个Shard的Key范围是左闭右开。更多详情请参考：根据HashKey写入数据。 |
| Shard Merge | Shard合并，可以把相邻的Key Range连接的Shard merge成一个Shard。更多详情请参考：Shard扩容缩容。 |
| Shard Split | Shard分裂，可以把一个Shard分裂成Shard Key Range相连接的两个Shard |
| Record | 用户数据和 DataHub 端交互的基本单位 |
| RecordType | Topic的数据类型，目前支持Tuple与Blob两种类型。Tuple类型的Topic支持类似于数据库的记录的数据，每条记录包含多个列。Blob类型的Topic仅支持写入一块二进制数据。 |

# 数据类型介绍

- Tuple类型下只支持写入数据是有格式的数据，支持以下几种数据类型

| 类型 | 含义 | 值域 |
| --- | --- | --- |
| Bigint | 8字节有符号整型。请不要使用整型的最小值 (-9223372036854775808)，这是系统保留值。 | -9223372036854775807 ~ 9223372036854775807 |

| | | |
|---|---|---|
| String | 字符串，只支持UTF-8编码。 | 单个String列最长允许1MB。 |
| Boolean | 布尔型。 | 可以表示为<br>True/False，true/false, 0/1 |
| Double | 8字节双精度浮点数。 | -1.0 *10308 ~ 1.0* 10308 |
| TimeStamp | 时间戳类型 | 表示到**微秒**的时间戳类型 |

- Blob模式下支持写入一块二进制数据作为一个Record，数据将会以BASE64编码传输。

# Shard状态说明

| 状态 | 说明 |
|---|---|
| Opening | Topic刚创建，所有shard会处于Opening状态直至准备完成。不可读写。 |
| Active | Shard通道打开后，状态会置为Active，此时表示Shard正常可读写。 |
| Closing | Shard进行了Split/Merge操作，后台正在关闭该通道。该状态Shard不可读写。 |
| Closed | Shard在Split/Merge完成后，会变为Closed态，此时Shard为只读状态。 |

# 异常描述

| ErrorCode | HttpCode | 含义 |
|---|---|---|
| InvalidUriSpec | 400 | 请求的Uri非法 |
| InvalidParameter | 400 | 参数错误，详细内容请看返回的ErrorMessga |
| Unauthorized | 401 | 签名错误 |
| NoPermission | 403 | 账号权限不足 |
| InvalidSchema | 400 | Schema格式错误 |
| InvalidCursor | 400 | 无效或过期的cursor |
| NoSuchProject | 404 | 请求的Project不存在 |
| NoSuchTopic | 404 | 请求的Topic不存在 |
| NoSuchShard | 404 | 请求的ShardID不存在 |
| ProjectAlreadyExist | 400 | Project已存在 |
| TopicAlreadyExist | 400 | Topic已存在 |
| InvalidShardOperation | 405 | 非法Shard操作，如Shard已经Closed后继续写入. |

| | | |
|---|---|---|
| LimitExceeded | 400 | 请求参数超出限制，如Shard总数超过512个. |
| InternalServerError | 500 | 未知错误或内部服务异常或系统处于升级中. |

DataHub采用阿里云RAM进行访问控制。用户对DataHub资源的访问，通过RAM进行鉴权。阿里云主账号拥有所属资源的所有权限，子用户在创建时并没有任何权限，不能访问任何资源，用户需要在RAM中对该子用户进行授权操作。关于如何创建RAM子用户与创建授权策略并进行授权可参见RAM使用文档。以下将介绍DataHub在RAM下的访问控制体系。

# DataHub访问域名

对DataHub资源的访问请求，需根据资源所属服务，选择正确的域名。

## DataHub域名列表

| 地区 | Region | 外网Endpoint | 经典网络ECS Endpoint | VPC ECS Endpoint |
|---|---|---|---|---|
| 华东1(杭州) | dh-cn-hangzhou | http://dh-cn-hangzhou.aliyuncs.com | http://dh-cn-hangzhou.aliyun-inc.com | http://dh-cn-hangzhou-vpc.aliyuncs.com |

特别提醒：目前，DataHub部署于华东1(杭州)可用区，MaxCompute服务本身在华东2(上海)可用区。DataHub自身会完成到MaxCompute离线集群的数据同步，此行为对用户透明，无需用户关心。

# DataHub RAM权限控制

## DataHub资源

DataHub在RAM的访问控制中的资源体系只包含Project和Topic。目前只支持Project和Topic级别的鉴权，并不支持Shard的访问控制。

| 资源 | RAM中的资源描述 |
|---|---|
| Project | acs:dhs:$region:$accountid:projects/$projectName |
| Topic | acs:dhs:$region:$accountid:projects/$projectName/topics/$topicName |

# DataHub API及对应在RAM中的授权策略

## Project

| API | Action | Resource |
| --- | --- | --- |
| CreateProject | dhs:CreateProject | acs:dhs:$region:$accountid:projects/* |
| ListProject | dhs:ListProject | acs:dhs:$region:$accountid:projects/* |
| DeleteProject | dhs:DeleteProject | acs:dhs:$region:$accountid:projects/$projectName |
| GetProject | dhs:GetProject | acs:dhs:$region:$accountid:projects/$projectName |

## Topic

| API | Action | Resource |
| --- | --- | --- |
| CreateTopic | dhs:CreateTopic | acs:dhs:$region:$accountid:projects/$projectName/topics/* |
| ListTopic | dhs:ListTopic | acs:dhs:$region:$accountid:projects/$projectName/topics/* |
| DeleteTopic | dhs:DeleteTopic | acs:dhs:$region:$accountid:projects/$projectName/topics/$topicName |
| GetTopic | dhs:GetTopic | acs:dhs:$region:$accountid:projects/$projectName/topics/$topicName |
| UpdateTopic | dhs:UpdateTopic | acs:dhs:$region:$accountid:projects/$projectName/topics/$topicName |

## Shard

| API | Action | Resource |
| --- | --- | --- |
| ListShard | dhs:ListShard | acs:dhs:$region:$accountid:projects/$projectName/topics/$topicName |
| MergeShard | dhs:MergeShard | acs:dhs:$region:$accountid:projects/$projectName/topics/$topicName |
| SplitShard | dhs:SplitShard | acs:dhs:$region:$accountid:projects/$projectName/topics/$topicName |

### PubSub

| API | Action | Resource |
| --- | --- | --- |
| PutRecords | dhs:PutRecords | acs:dhs:$region:$accountid:projects/$projectName/topics/$topicName |
| GetRecords | dhs:GetRecords | acs:dhs:$region:$accountid:projects/$projectName/topics/$topicName |
| GetCursor | dhs:GetRecords | acs:dhs:$region:$accountid:projects/$projectName/topics/$topicName |

## DataHub支持的Condition

| Condition | 功能 | 合法取值 |
| --- | --- | --- |
| acs:SourceIp | 指定ip网段 | 普通ip, 支持*通配 |
| acs:SecureTransport | 是否是https协议 | true/false |
| acs:MFAPresent | 是否多设备认证 | true/false |
| acs:CurrentTime | 指定访问时间 | ISO8601格式 |

## DataHub服务角色

```
{
"Version": "1",
"Statement": [
{
"Action": "sts:AssumeRole",
"Effect": "Allow",
"Principal": {
"Service": [
"dhs.aliyuncs.com"
]
}
}
]
}
```

## DataHub系统授权策略

### AliyunDataHubFullAccess

```
{
```

```
"Version": "1",
"Statement": [
{
"Action": "dhs:*",
"Resource": "*",
"Effect": "Allow"
}
]
}
```

## AliyunDataHubReadOnlyAccess

```
{
"Version": "1",
"Statement": [
{
"Action": ["dhs:List*", "dhs:Get*"],
"Resource": "*",
"Effect": "Allow"
}
]
}
```

## DataHub自定义授权策略示例

```
//只允许用户获取指定Project下topic的信息
{
"Version": "1",
"Statement": [
{
"Action": ["dhs:ListTopic", "dhs:GetTopic"]
"Resource": "acs:dhs:cn-hangzhou:12121312:projects/foo/topics/*",
"Effect": "Allow"
}
]
}


/* PubSub
* 进行发布订阅，除了需要PutRecords，GetRecords权限外
* 往往用户需要知道topic的schema和该topic的shard状态
* 所以最好同时授予用户GetTopic和ListShard权限
*/
{
"Version": "1",
"Statement": [
{
"Action": ["dhs:*Records", "dhs:GetTopic", dhs:ListShard"],
"Resource": "acs:dhs:cn-hangzhou:12121312:projects/foo/topics/bar",
"Effect": "Allow"
}
}
```

```
//对所有topic进行PubSub操作
{
"Version": "1",
"Statement": [
{
"Action": ["dhs:*Records", "dhs:GetTopic", dhs:ListShard"],
"Resource": "acs:dhs:cn-hangzhou:12121312:*",
"Effect": "Allow"
}
}

// 对指定Topic进行 Split/Merge shard, 包括ListShard, SplitShard, MergeShard
{
"Version": "1",
"Statement": [
{
"Action": ["dhs:*Shard"],
"Resource": "acs:dhs:cn-hangzhou:12121312:projects/foo/topics/bar",
"Effect": "Allow"
}
}
```

# 限制描述

| 限制项 | 描述 | 值域范围 |
| --- | --- | --- |
| 活跃shard数 | 每个topic中活跃shard数量限制 | (0,256] |
| 总shard数 | 每个topic中总shard数量限制 | (0,512] |
| Http BodySize | http请求中body大小限制 | 4MB |
| 单个String长度 | 数据中单个String字段长度限制 | 1MB |
| Merge/Split频率限制 | 每个新产生的shard在一定时间内不允许进行Merge/Split操作 | 5s |
| QPS限制 | 每个Shard写入QPS限制(非Record/s，Batch写入同一Shard仅计算为1次) | 1000 |
| Throughput限制 | 每个Shard写入每秒吞吐限制 | 1MB |
| Project限制 | 每个云账号能够创建的Project上限 | 5 |
| Topic限制 | 每个Project内能创建的Topic数量限制，如有特殊请求请联系管理员 | 20 |

# 命名规范

| 名词 | 描述 | 长度限制 | 值 |
| --- | --- | --- | --- |

| | | | |
|---|---|---|---|
| Project | 项目名称 | [3,32] | 英文字母开头，仅允许英文字母、数字及“_”，大小写不敏感。 |
| Topic | 主题名称 | [1,128] | 英文字母开头，仅允许英文字母、数字及“_”，大小写不敏感。 |

# 快速指引

本节主要描述使用DataHub Java SDK进行数据的读写。

## 准备工作

- 使用DataHub服务之前,需要注册阿里云云账号,利用阿里云账号的AccessId与AccessKey接入DataHub服务。

## 创建Project/Topic

- 登录DataHub Web Console页面，创建Topic。

## 发布数据/订阅数据

发布/订阅数据需要使用sdk完成， DataHub Java SDK使用说明

```
// 新建client
Account account = new AliyunAccount("your access id", "your access key");
DatahubConfiguration conf = new DatahubConfiguration(account, "datahub endpoint");
DatahubClient client = new DatahubClient(conf);
// 可通过listShard接口获取shard列表，所有ACTIVE的shard均可使用，本利使用"0"
String shardId = "0";
// 构造需要上传的records
RecordSchema schema = client.getTopic("test_project", "test_topic").getRecordSchema();
List<RecordEntry> recordEntries = new ArrayList<~>();
RecordEntry entry = new RecordEntry(schema);
for (int i=0; i<entry.getFieldCount(); i++) {
entry.setBigint(i, 1); //set your fields' value according to the field's type
}
entry.setShardId(shardId);
recordEntries.add(entry);

// 数据写入
```

```
PutRecordsResult result = client.putRecords("test_project", "test_topic", recordEntries);
if (result.getFailedRecordCount() != 0) {
List<ErrorEntry> errors = result.getFailedRecordError();
List<RecordEntry> records = result.getFailedRecords();
// 存在写入失败的记录，建议日志记录错误原因并重试写入
}

// 根据时间获取游标
GetCursorResult cursorRs = client.getCursor("test_project", "test_topic", shardId, 1455869335000 /*ms*/);

// 数据读取
//每次限读100条，最大不可超过1000
int limit = 100;
String cursor = cursorRs.getCursor();
while (true) {
try {
GetRecordsResult recordRs = client.getRecords("test_project", "test_topic", shardId, cursorRs.getCursor(), limit,
schema);
List<RecordEntry> recordEntries = recordRs.getRecords();
if (recordEntries.size() == 0) {
// 无最新数据，请稍等重试
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
// 拿到下一个游标
cursor = recordRs.getNextCursor();
} catch (InvalidCursorException ex) {
// 非法游标或游标已过期，建议重新定位后开始消费
cursorRs = client.getCursor(projectName, topicName, shardId, GetCursorRequest.CursorType.OLDEST);
cursor = cursorRs.getCursor();
}
}
```

# 使用指南

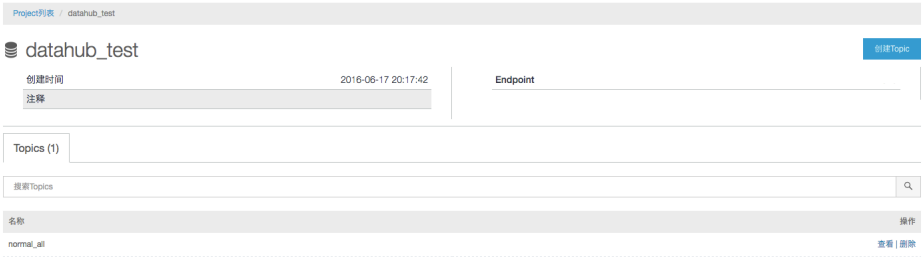
控制台地址 DataHub WebConsole

## 创建Project

在WebConsole中直接点击创建Project后填写相关信息

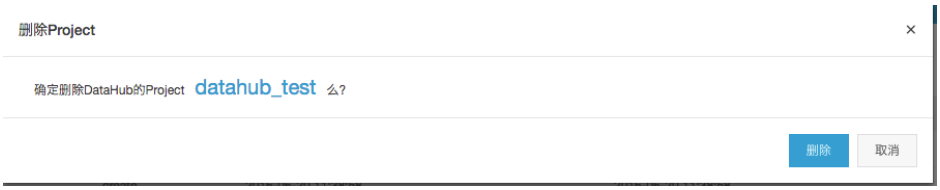

# 查看Project

用户点击Project列表中的查看按钮，可查看该Project的详细信息以及该Project下的Topic列表。




# 删除Project

用户点击Project列表中的删除按钮进行Project的删除。需要注意的是，一旦删除Project，该Project下所有Topic，数据，及资源均被删除，无法恢复，请谨慎操作。




# 创建Topic

用户点击Project详情页面中的创建Topic按钮，进行Topic的创建。

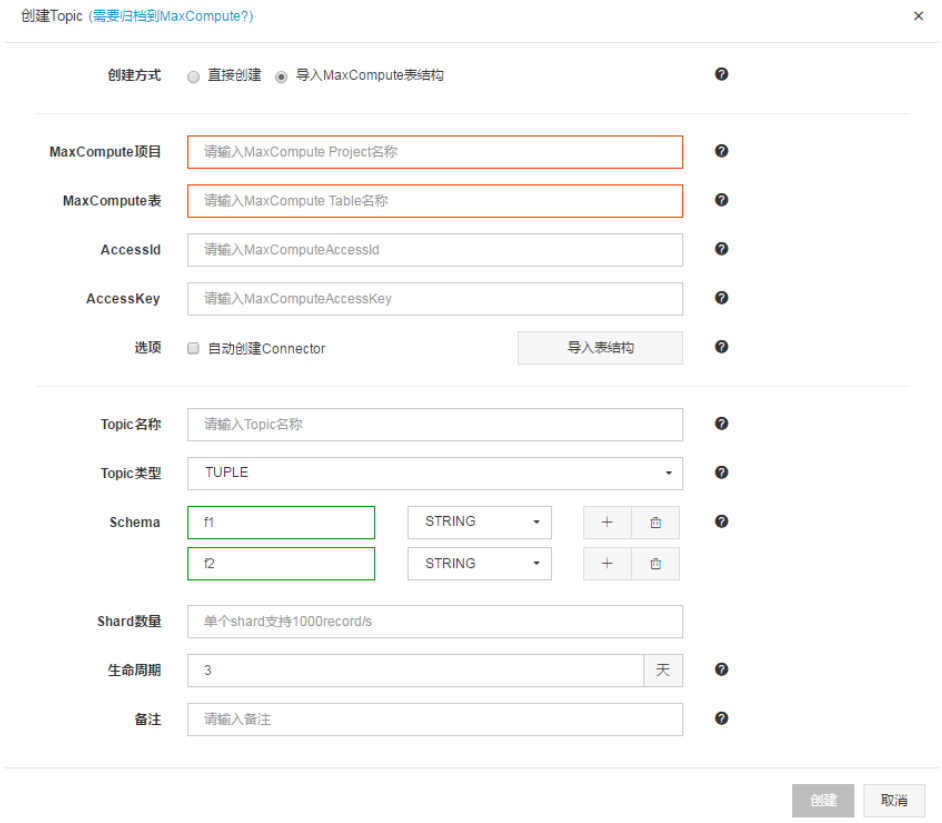
可以通过两种方式创建Topic

## 自定义格式直接创建

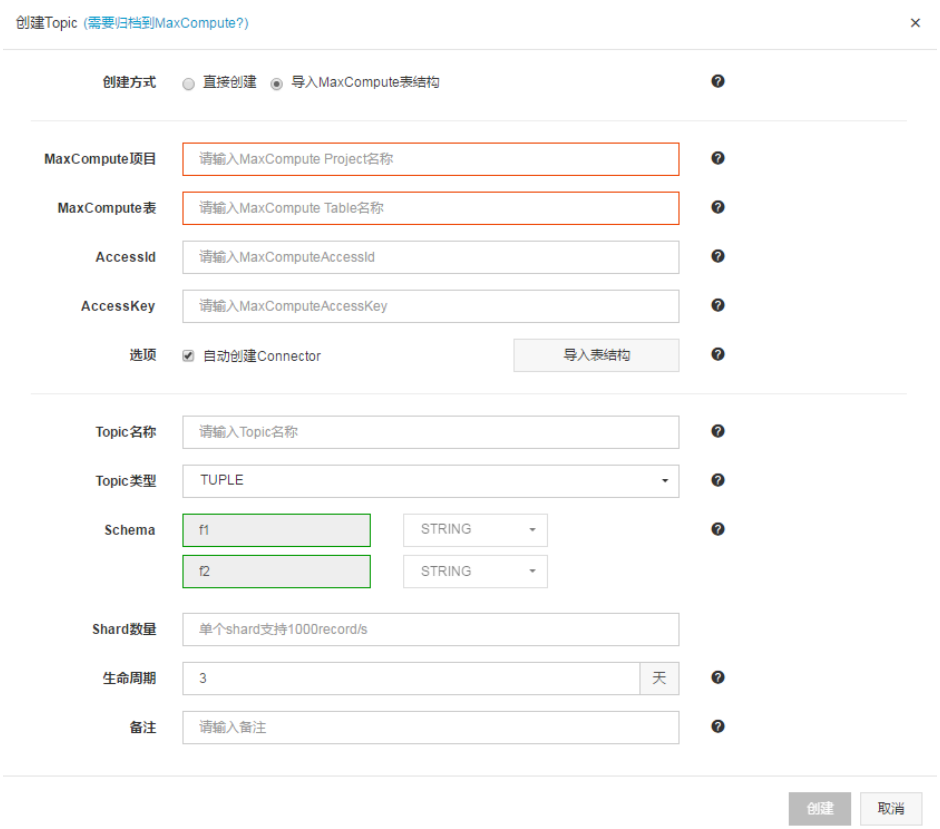

用户可根据业务需求进行Topic的创建，各选项的意义可查看帮助文档中的基本概念。

## 通过导入MaxCompute表结构进行创建

需要指定访问MaxCompute的相关参数：




该种方式创建Topic可以自动创建导入数据到MaxCompute的Connector，勾选如下图所示选项即可，选中后将不可修改格式：



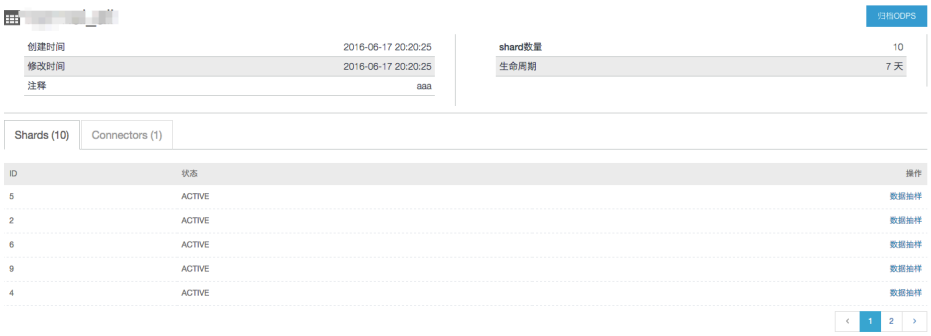

## 删除Topic

用户点击Topic列表中的删除按钮可进行Topic的删除。需要注意的是，一旦删除Topic，该Topic下的数据，资源(Shard, Connector)均被删除，无法恢复，请谨慎操作。

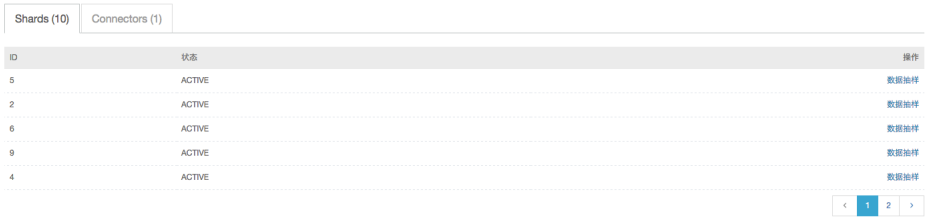

## 查看Topic

用户点击Topic列表中的查看按钮，可查看该Topic详细信息以及该Topic下的资源列表，包括Shard和Connector

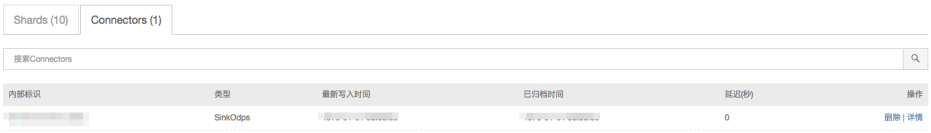
Topic详情展示了Topic的创建/修改时间，shard数量，Topic的生命周期以及注释等信息；




Shard列表中包括ShardId，各个Shard状态；其中ShardId是该Shard在Topic中的唯一标识; Shard状态反应该Shard是否可进行数据发布与订阅；

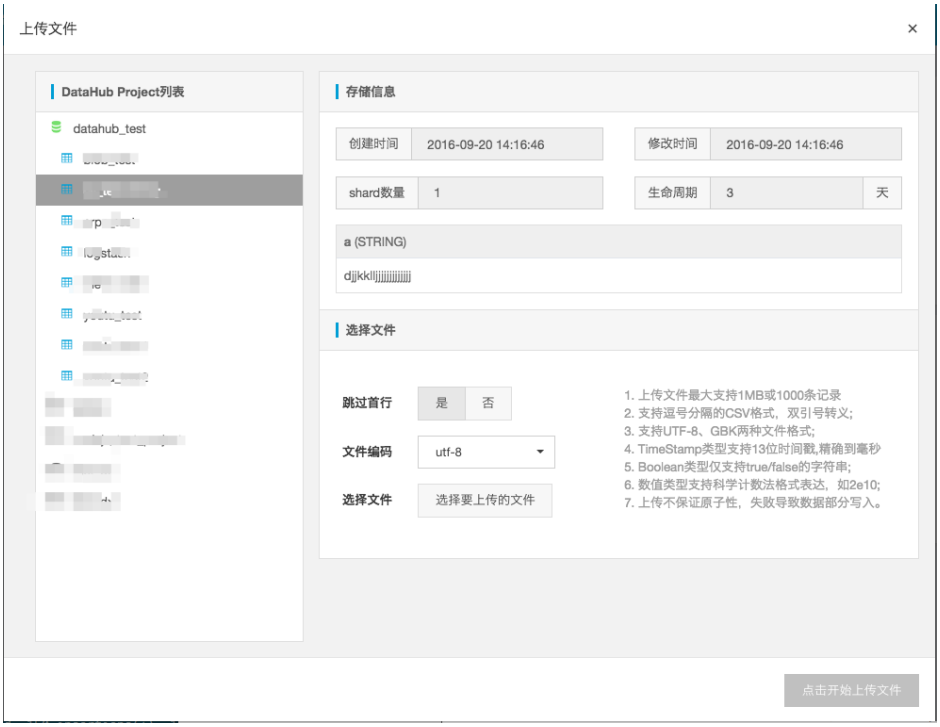



Connector列表展示该Topic下存在的Connector信息: 内部标识唯一确定该Topic下的Connector; 类型表示该Connector所指向的目的端，目前只有Odps;目的项目名称以及目的表名称指定需要归档到的ODPS项目和表名称;




# 数据采集

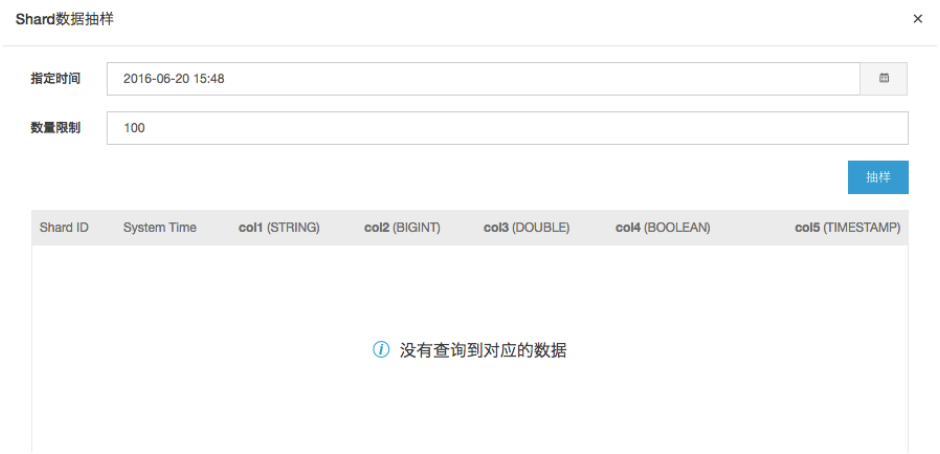
Web Console集成了上传文件到DataHub的功能。




选择Topic之后，指定与Topic对应格式的文件，并点击 “开始上传文件按钮 “，文件中的数据就按照指定要求上传到对应的Topic中。

## 数据抽样

Web Console提供对Shard的基于时间的数据抽样功能。用户可在Shard列表中选择Shard点击列表尾端的数据抽样按钮，进入数据抽样页面。




设置指定时间与数量限制，展示在指定时间之后写入的指定数量的Record，当Record数不足指定数量时，展示仅有的Records。当指定时间大于等于当前时间，仅展示最后一条记录。

## 归档MaxCompute

目前，DataHub仅提供创建归档到MaxCompute的Connector的功能。在Topic详情页中，点击归档

MaxCompute按钮，进入创建Connector页面

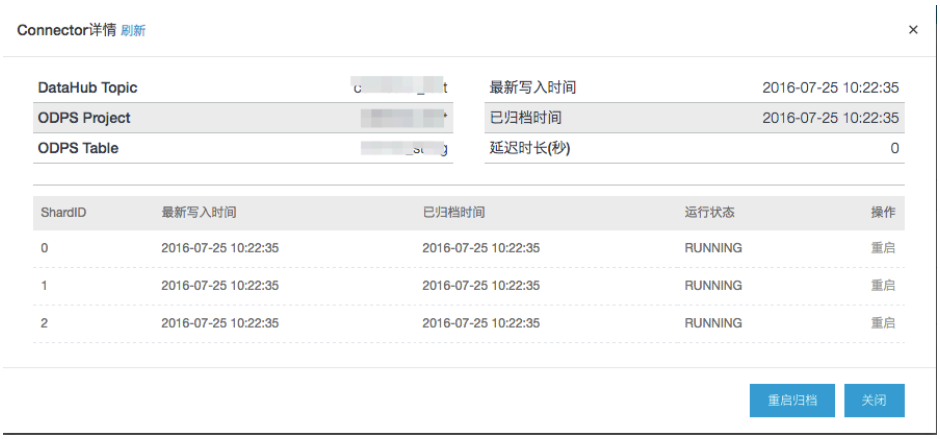

MaxCompute Project: 指定Topic需要同步到的MaxCompute project名称;

MaxCompute Table: 指定Topic需要同步到的MaxCompute table名称;

AccessId/AccessKey: 访问MaxCompute所需要的AK对，该AK必须是子用户AK，并且具有指定的MaxCompute table下CreateInstance、Desc、Alter权限;

## 归档详情

归档详情提供归档任务的基本信息，包括DataHub的Topic以及需要归档到的MaxCompute项目和表；以及整体归档进度，包括最新写入时间、已归档时间和归档延迟。最新写入时间表示指定Topic中最新的写入记录的系统时间; 已归档时间表示在该时间之前的记录已全部归档; 延迟时长为最新写入时间和已归档时间的差值，单位秒。

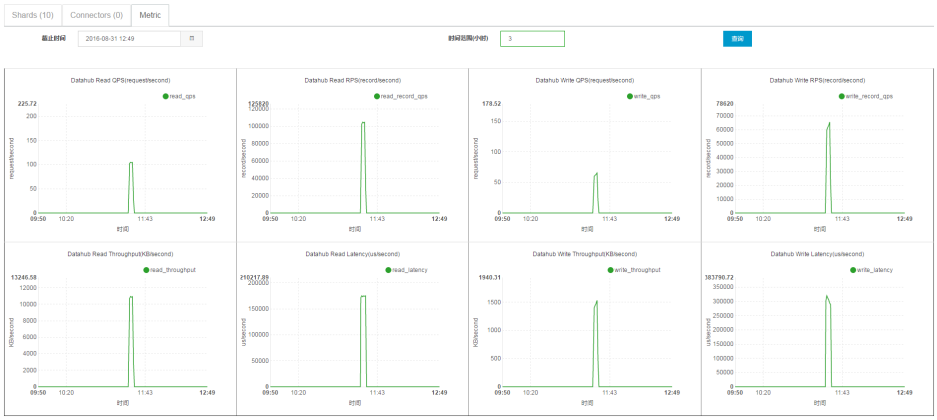



归档详情中可以查看该Connector归档任务中每个shard的归档状况。当运行状态为”ERROR”时，鼠标移动到帮助图标会显示该Shard归档任务终止的详细报错信息，并且该Shard的重启功能可用。用户可根据报错信息进行修复(如，因为删除MaxCompute Partition导致的任务终止，可以通过MaxCompute控制台,SDK等创建Partition，修复完成后，点击该Shard的”重启”按钮，该Shard的归档任务将重新被调度，并且从上次终止的记录开始进行归档。若所有Shard或者大部分Shard处于非”RUNNING”状态，可点击”重启归档”按钮，重启所有Shard的归档任务。

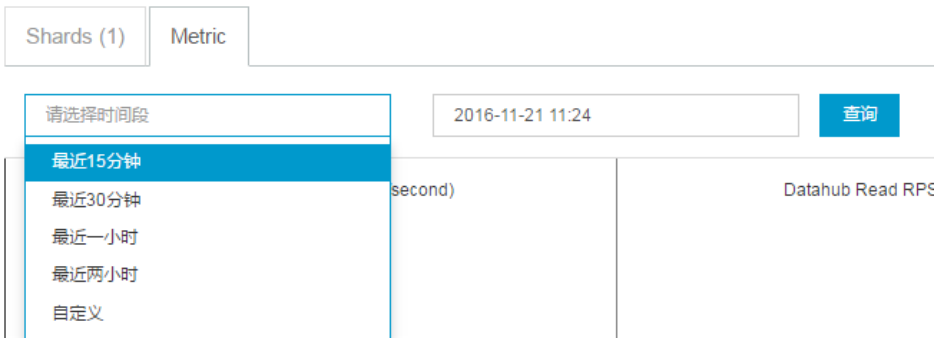

## Metric查看

Web Console目前提供Metric功能，用户可以通过Metric界面查看准实时的Topic级别流量等信息，目前提供的指标有：

- 读写Request/Second
- 读写Record/Second
- 读写Throughput/Second (单位KB)
- 读写请求Latency/Request （单位微秒）



可以指定历史时间段，目前仅支持一次最多查询4个小时内的统计信息，暂不支持一次查询中时间跨天。如图，选定一个时间范围将会显示对应期间的Metric信息。



# 第三方工具插件

# Fluentd采集工具

## 产品介绍

该插件是基于Fluentd开发的输出插件，主要是将采集到的数据写入DataHub。该插件遵守Fluentd输出插件开发规范，安装方便，可以很方便地将采集得到的数据写到DataHub。

## 产品安装

### 通过Ruby gem安装

注意：RubyGem源建议更改为https://gems.ruby-china.org/

```
gem install fluent-plugin-datahub
```

### 本地安装

> 当前Fluentd仅支持Linux环境, 同时要求用户安装Ruby。
>
> 目前支持两种安装模式，对于没有安装过fluentd的客户我们提供了一键fluent+datahub全部安装模式，对于曾经安装了fluentd的客户我们提供datahub写入插件单独安装模式

1) **一键安装**: 如果之前没安装过fluentd，请**点击下载Fluentd完全安装包。**注意，**完全安装包提供的fluentd是fluentd-0.12.23.gem的版本。**

```
    $ tar -xzvf fluentd-with-datahub-0.12.23.tar.gz
$ cd fluentd-with-dataHub
$ sudo sh install.sh
```

2) **单独安装** 如果之前安装过fluentd，请**点此下载fluentd datahub插件包**, 使用gem命令安装datahub插件。

```

```

```
$ sudo gem install --local fluent-plugin-dataHub-0.0.2.gem
```

# 使用案例

## 案例一: CSV文件上传

下面以增量的CSV文件为例，说明下如何使用Fluentd将增量的CSV文件准实时上传到DataHub数据。CSV文件的格式如下图所示:

```
0,qe614c760fuk8judu01tn5x055rpt1,true,100.1,14321111111
1,znv1py74o8ynn87k66o32ao4x875wi,true,100.1,14321111111
2,7nm0mtpgo1q0ubuljjjx9b000ybltl,true,100.1,14321111111
3,10t0n6pvonnan16279w848ukko5f6l,true,100.1,14321111111
4,0ub584kw88s6dczd0mta7itmta10jo,true,100.1,14321111111
5,1ltfpf0jt7fhvf0oy4lo8m3z62c940,true,100.1,14321111111
6,zpqsfxqy9379lmcehd7q8kftntrozb,true,100.1,14321111111
7,ce1ga9aln346xcj761c3iytshyzuxg,true,100.1,14321111111
8,k5j2id9a0ko90cykl40s6ojq6gruyi,true,100.1,14321111111
9,ns2zcx9bdip5y0aqd1tdicf7bkdmsm,true,100.1,14321111111
10,54rs9cm1xau2fk66pzyz62tf9tsse4,true,100.1,14321111111
```

上述CSV文件中每行一条Record，按照(,)区分字段。保存在本地路径/temp/test.csv中。DataHub Topic格式如下:

| 字段名称 | 字段类型 |
| --- | --- |
| id | BIGINT |
| name | STRING |
| gender | BOOLEAN |
| salary | DOUBLE |
| my_time | TIMESTAMP |

使用如下Fluentd的配置，配置文件地址在 ${CONFIG_HOME}/fluentd_test.conf:

```
<source>
@type tail
path 你的文件路径
tag test1
format csv
keys id,name,gender,salary,my_time
</source>

<match test1>
@type dataHub
access_id your_app_id
access_key your_app_key
```

```
endpoint http://ip:port
project_name test_project
topic_name fluentd_performance_test_1
column_names ["id", "name", "gender", "salary", "my_time"]
flush_interval 1s
buffer_chunk_limit 3m
buffer_queue_limit 128
dirty_data_continue true
dirty_data_file 脏数据记录文件路径
retry_times 3
put_data_batch_size 1000
</match>
```

使用如下命令启动Fluentd，即可完成CSV文件数据采集进入DataHub:

```
${FLUENTD_HOME}/fluentd-with-dataHub/bin/fluentd -c ${CONFIG_HOME}/fluentd_test.conf
```

# 案例二: Log4J日志采集

Log4j的日志格式如下:

```
11:48:43.439 [qtp1847995714-17] INFO  AuditInterceptor - [c2un5sh7cu52ek6am1ui1m5h] end
/web/v1/project/tefe4mfurtix9kwwyrvfqd0m/node/0m0169kapshvgc3ujskwkk8g/health GET, 4061 ms
```

使用如下Fluentd配置:

```
<source>
@type tail
path bayes.log
tag test
format /(?<request_time>\d\d:\d\d:\d\d.\d+)\s+\[(?<thread_id>[\w\-
]+)\]\s+(?<log_level>\w+)\s+(?<class>\w+)\s+-\s+\[(?<request_id>\w+)\]\s+(?<detail>.+)/
</source>

<match test>
@type dataHub
access_id your_access_id
access_key your_access_key
endpoint http://ip:port
project_name test_project
topic_name dataHub_fluentd_out_1
column_names ["thread_id", "log_level", "class"]
</match>
```

使用该配置启动即可完成log4j日志采集进入DataHub的功能。

# 配置参数

读插件配置
tag test1 : 指定路由, 和<match>会进行路由正则匹配
format csv : 数据按照csv方式采集
keys id,name,gender,salary,my_time : 指定需要采集的列名，必须和目的DataHub表的列名一致

写插件配置
shard_id 0 : 指定shard_id写入，默认round-robin方式写入
shard_keys ["id"] : 指定用作分区key，用key值hash后作为写入shard的索引
flush_interval 1 : fluentd 每一秒钟至少写一次, 默认60s
buffer_chunk_limit 3m : 块大小，支持“k”(KB),“m”(MB)单位，建议值3m
buffer_queue_limit 128 : 块队列大小，此值与buffer_chunk_limit共同决定整个缓冲区大小
put_data_batch_size 1000 : 每1000条record写一次DataHub
retry_times 3 : 重试次数
retry_interval 3 : 重试间隔(单位:s)
dirty_data_continue true : 遇到增数据是否继续，若为true 遇到脏数据会重试，重试次数用完，会将脏数据写入脏数据文件
dirty_data_file /xxx/yyy : 指定脏数据文件的位置

column_names ["id"] : 指定需要采集的列

# 性能测试

性能测试环境: Fluentd运行环境为2核4G，操作系统为Linux；性能结果如下:

| 单条record大小 | put_data_batch_size(record) | | |
|---|---|---|---|
| | 100 | 500 | 1000 |
| 100B | 3030r/s | 3226r/s | 3226r/s |
| 512B | 2777r/s | 2857r/s | 2857r/s |
| 1K | 2380r/s | 2439r/s | 2564r/s |
| 100K | 50r/s | 不可用 | 不可用 |

从本次DataHub插件性能测试数据中可以看到:

针对单条512B的数据，写入速度极本保持在 2800record/s 左右

随着put_data_batch_size的增加速度略有提升，但效果不大。

对于单条100K的数据, put_data_batch_size只有100的时候可以正常work，500和1000都不可用:
因为一次写入数据过大，已经大于50m

总的平均写入速度(MB/S)保持在 3MB/S

# FAQ

Q: 关于Fluentd，如何编写Format的正则表达式？

A: 如何写 format正则

# Logstash采集工具

Logstash是一种分布式日志收集框架，非常简洁强大，经常与ElasticSearch，Kibana配置，组成著名的ELK技术栈，非常适合用来做日志数据的分析。 阿里云流计算为了方便用户将更多数据采集进入DataHub，提供了针对Logstash的DataHub Output/Input插件。使用Logstash，您可以轻松享受到Logstash开源社区多达30+种数据源支持(file，syslog，redis，log4j，apache log或nginx log)，同时Logstash还支持filter对传输字段自定义加工等功能。下面我们就以Logstash日志采集为例介绍如何使用Logstash快速将日志数据采集进入DataHub。

注: Logstash DataHub Output/Input插件遵循Apache License 2.0开源协议。

## 安装

### 安装限制

Logstash安装要求JRE 7版本及以上，否则部分功能无法使用。

### 如何安装

阿里云提供两种LogStash的安装方式：

**一键安装**

一键安装包，支持将Logstash和DataHub插件一键安装，点此下载

当前我们提供了免安装的版本，解压即可使用:

```
    $ tar -xzvf logstash-with-datahub-2.3.0.tar.gz
$ cd logstash-with-datahub-2.3.0
```

命令执行完成后，logstash即安装成功。

**单独安装**

安装Logstash: 参看Logstash官网提供的[安装教程]进行安装工作。特别注意的是，最新的Logstash需要Java 7及以上版本。

安装DataHub插件: 下载所需要的插件。

上传至DataHub请使用： DataHub Logstash Output插件

下载DataHub中数据请使用： DataHub Logstash Input插件

分别使用如下命令进行安装:

```
$ {LOG_STASH_HOME}/bin/plugin install --local logstash-output-datahub-1.0.0.gem
$ {LOG_STASH_HOME}/bin/plugin install --local logstash-input-datahub-1.0.0.gem
```

# 使用DataHub Logstash Output插件上传数据

## 上传Log4j日志到DataHub

下面以Log4j日志产出为例，讲解下如何使用Logstash采集并结构化日志数据。Log4j的日志样例如下：

```
20:04:30.359 [qtp1453606810-20] INFO  AuditInterceptor - [13pn9kdr5tl84stzkmaa8vmg] end
/web/v1/project/fhp4clxfbu0w3ym2n7ee6ynh/statistics?executionName=bayes_poc_test GET, 187 ms
```

针对上述Log4j文件，我们希望将数据结构化并采集进入DataHub，其中DataHub的Topic格式如下:

| 字段名称 | 字段类型 |
| --- | --- |
| request_time | STRING |
| thread_id | STRING |
| log_level | STRING |
| class_name | STRING |
| request_id | STRING |
| detail | STRING |

Logstash任务配置如下：

```
input {
file {
path => "${APP_HOME}/log/bayes.log"
start_position => "beginning"
}
}

filter{
grok {
match => {
"message" => "(?<request_time>\d\d:\d\d:\d\d\.\d+)\s+\[(?<thread_id>[\w\-
]+)\]\s+(?<log_level>\w+)\s+(?<class_name>\w+)\s+\-\s+\[(?<request_id>\w+)\]\s+(?<detail>.+)"
}
```

```
}
}

output {
datahub {
access_id => "Your accessId"
access_key => "Your accessKey"
endpoint => "Endpoint"
project_name => "project"
topic_name => "topic"
#shard_id => "0"
#shard_keys => ["thread_id"]
dirty_data_continue => true
dirty_data_file => "/Users/ph0ly/trash/dirty.data"
dirty_data_file_max_size => 1000
}
}
```

使用如下命令启动Logstash:

```
logstash -f <上述配置文件地址>
```

batch_size是每次向DataHub发送的记录条数，默认为125，指定batch_size启动Logstash:

```
logstash -f <上述配置文件地址> -b 256
```

## 参数

参数说明说明如下：

```
    access_id(Required): 阿里云access id
access_key(Required): 阿里云access key
endpoint(Required): 阿里云datahub的服务地址
project_name(Required): datahub项目名称
topic_name(Required): datahub topic名称
retry_times(Optional): 重试次数，-1为无限重试、0为不重试、>0表示需要有限次数, 默认值为-1
retry_interval(Optional): 下一次重试的间隔，单位为秒，默认值为5
shard_keys(Optional)：数组类型，数据落shard的字段名称，插件会根据这些字段的值计算hash将每条数据落某个shard, 注意shard_keys和shard_id都未指定，默认轮询落shard
shard_id(Optional): 所有数据落指定的shard，注意shard_keys和shard_id都未指定，默认轮询落shard
dirty_data_continue(Optional): 脏数据是否继续运行，默认为false，如果指定true，则遇到脏数据直接无视，继续处理数据。当开启该开关，必须指定@dirty_data_file文件
dirty_data_file(Optional): 脏数据文件名称，当数据文件名称，在@dirty_data_continue开启的情况下，需要指定该值。特别注意：脏数据文件将被分割成两个部分.part1和.part2，part1作为更早的脏数据，part2作为更新的数据
dirty_data_file_max_size(Optional): 脏数据文件的最大大小，该值保证脏数据文件最大大小不超过这个值，目前该值仅是一个参考值
```

# 使用DataHub Logstash Input插件读取DataHub中数据

## 消费DataHub数据写入文件

logstash的配置如下：

```
input {
datahub {
access_id => "Your accessId"
access_key => "Your accessKey"
endpoint => "Endpoint"
project_name => "test_project"
topic_name => "test_topic"
interval=> 5
#cursor => {
# "0"=>"20000000000000000000000003110091"
# "2"=>"20000000000000000000000003110091"
# "1"=>"20000000000000000000000003110091"
# "4"=>"20000000000000000000000003110091"
# "3"=>"20000000000000000000000003110000"
#}
shard_ids => []
pos_file => "/home/admin/logstash/logstash-2.3.0/pos_file"
}
}

output {
file {
path => "/home/admin/logstash/logstash-2.3.0/output"
}
}
```

## 参数介绍

```
    access_id(Required): 阿里云access id
access_key(Required): 阿里云access key
endpoint(Required): 阿里云datahub的服务地址
project_name(Required): datahub项目名称
topic_name(Required): datahub topic名称
retry_times(Optional): 重试次数，-1为无限重试、0为不重试、>0表示需要有限次数
retry_interval(Optional): 下一次重试的间隔，单位为秒
shard_ids(Optional)：数组类型，需要消费的shard列表，空列表默认全部消费
cursor(Optional)：消费起点，默认为空，表示从头开始消费
pos_file(Required)：checkpoint记录文件，必须配置，优先使用checkpoint恢复消费offset
```

# 更多

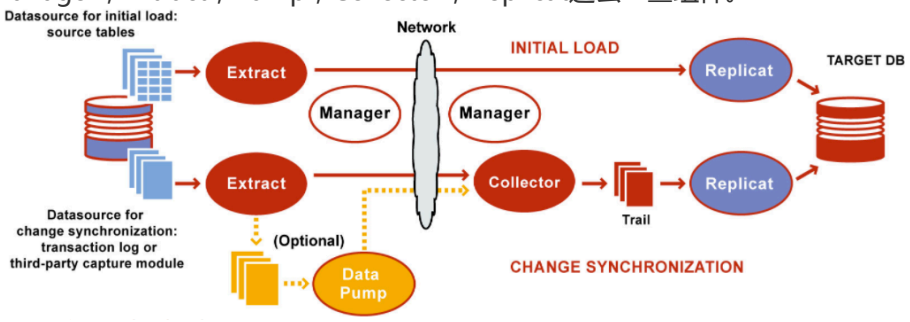
更多的配置参数请参考logstash官方网站，以及ELK stack中文指南

# OGG采集工具

## 一、背景介绍

随着数据规模的不断扩大，传统的RDBMS难以满足OLAP的需求，本文将介绍如何将Oracle的数据实时同步到阿里云的大数据处理平台当中，并利用大数据工具对数据进行分析。

OGG（Oracle GoldenGate）是一个基于日志的结构化数据备份工具，一般用于Oracle数据库之间的主从备份以及Oracle数据库到其他数据库（DB2, MySQL等）的同步。下面是Oracle官方提供的一个OGG的整体架构图，从图中可以看出OGG的部署分为源端和目标端两部分组成，主要有
Manager，Extract，Pump，Collector，Replicat这么一些组件。




- Manager：在源端和目标端都会有且只有一个Manager进程存在，负责管理其他进程的启停和监控等；
- Extract：负责从源端数据库表或者事务日志中捕获数据，有初始加载和增量同步两种模式可以配置，初始加载模式是直接将源表数据同步到目标端，而增量同步就是分析源端数据库的日志，将变动的记录传到目标端，本文介绍的是增量同步的模式；
- Pump：Extract从源端抽取的数据会先写到本地磁盘的Trail文件，Pump进程会负责将Trail文件的数据投递到目标端；
- Collector：目标端负责接收来自源端的数据，生成Trail文件
- Replicat：负责读取目标端的Trail文件，转化为相应的DDL和DML语句作用到目标数据库，实现数据同步。

本文介绍的Oracle数据同步是通过OGG的Datahub插件实现的，该Datahub插件在架构图中处于Replicat的位置，会分析Trail文件，将数据的变化记录写入Datahub中，可以使用流计算对datahub中的数据进行实时分析，也可以将数据归档到MaxCompute中进行离线处理。

## 二、安装步骤

### 0. 环境要求

- 源端已安装好Oracle
- 源端已安装好OGG（建议版本Oracle GoldenGate V12.1.2.1）

- 目标端已安装好OGG Adapters（建议版本Oracle GoldenGate Application Adapters 12.1.2.1）
- java 7

**（下面将介绍Oracle/OGG相关安装和配置过程，Oracle的安装将不做介绍，另外需要注意的是：Oracle/OGG相关参数配置以熟悉Oracle/OGG的运维人员配置为准，本示例只是提供一个可运行的样本，Oracle所使用的版本为ORA11g）**

# 1. 源端OGG安装

下载OGG安装包解压后有如下目录：

```
drwxr-xr-x install
drwxrwxr-x response
-rwxr-xr-x runInstaller
drwxr-xr-x stage
```

目前oracle一般采取response安装的方式，在response/oggcore.rsp中配置安装依赖，具体如下：

```
oracle.install.responseFileVersion=/oracle/install/rspfmt_ogginstall_response_schema_v12_1_2
# 需要目前与oracle版本对应
INSTALL_OPTION=ORA11g
# goldegate主目录
SOFTWARE_LOCATION=/home/oracle/u01/ggate
# 初始不启动manager
START_MANAGER=false
# manger端口
MANAGER_PORT=7839
# 对应oracle的主目录
DATABASE_LOCATION=/home/oracle/u01/app/oracle/product/11.2.0/dbhome_1
# 暂可不配置
INVENTORY_LOCATION=
# 分组（目前暂时将oracle和ogg用同一个账号ogg_test，实际可以给ogg单独账号）
UNIX_GROUP_NAME=oinstall
```

执行命令：

```
runInstaller -silent -responseFile {YOUR_OGG_INSTALL_FILE_PATH}/response/oggcore.rsp
```

本示例中，安装后OGG的目录在/home/oracle/u01/ggate，安装日志在/home/oracle/u01/ggate/cfgtoollogs/oui目录下，当silentInstall{时间}.log文件里出现如下提示，表明安装成功：

```
The installation of Oracle GoldenGate Core was successful.
```

执行/home/oracle/u01/ggate/ggsci命令，并在提示符下键入命令：CREATE SUBDIRS，从而生成ogg需要的各种目录（dir打头的那些）。至此，源端OGG安装完成。

## 2. 源端Oracle配置

以dba分身进入sqlplus：sqlplus / as sysdba

```
 # 创建独立的表空间
create tablespace ATMV datafile '/home/oracle/u01/app/oracle/oradata/uprr/ATMV.dbf' size 100m autoextend on
next 50m maxsize unlimited;

# 创建ogg_test用户，密码也为ogg_test
create user ogg_test identified by ogg_test default tablespace ATMV;

# 给ogg_test赋予充分的权限
grant connect,resource,dba to ogg_test;

# 检查附加日志情况
Select SUPPLEMENTAL_LOG_DATA_MIN, SUPPLEMENTAL_LOG_DATA_PK, SUPPLEMENTAL_LOG_DATA_UI,
SUPPLEMENTAL_LOG_DATA_FK, SUPPLEMENTAL_LOG_DATA_ALL from v$database;

# 增加数据库附加日志及回退
alter database add supplemental log data;
alter database add supplemental log data (primary key, unique,foreign key) columns;
# rollback
alter database drop supplemental log data (primary key, unique,foreign key) columns;
alter database drop supplemental log data;

# 全字段模式，注意：在该模式下的delete操作也只有主键值
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
# 开启数据库强制日志模式
alter database force logging;
# 执行marker_setup.sql 脚本
@marker_setup.sql
# 执行@ddl_setup.sql
@ddl_setup.sql
# 执行role_setup.sql
@role_setup.sql
# 给ogg用户赋权
grant GGS_GGSUSER_ROLE to ogg_test;
# 执行@ddl_enable.sql，开启DDL trigger
@ddl_enable.sql
# 执行优化脚本
@ddl_pin ogg_test
# 安装sequence support
@sequence.sql
#
alter table sys.seq$ add supplemental log data (primary key) columns;
```

## 3. OGG源端mgr配置

以下是通过ggsci对ogg进行配置

配置mgredit params mgr

```
PORT 7839
DYNAMICPORTLIST 7840-7849
USERID ogg_test, PASSWORD ogg_test
PURGEOLDEXTRACTS ./dirdat/*, USECHECKPOINTS, MINKEEPDAYS 7
LAGREPORTHOURS 1
LAGINFOMINUTES 30
LAGCRITICALMINUTES 45
PURGEDDLHISTORY MINKEEPDAYS 3, MAXKEEPDAYS 7
PURGEMARKERHISTORY MINKEEPDAYS 3, MAXKEEPDAYS 7
```

启动mgr（运行日志在ggate/dirrpt中）

start mgr

查看mgr状态

info mgr

查看mgr配置

view params mgr

## 4. OGG源端extract配置

以下是通过ggsci对ogg进行配置

配置extract（名字可任取，extract是组名）edit params extract

```
EXTRACT extract
SETENV (NLS_LANG="AMERICAN_AMERICA.AL32UTF8")
DBOPTIONS ALLOWUNUSEDCOLUMN
USERID ogg_test, PASSWORD ogg_test
REPORTCOUNT EVERY 1 MINUTES, RATE
NUMFILES 5000
DISCARDFILE ./dirrpt/ext_test.dsc, APPEND, MEGABYTES 100
DISCARDROLLOVER AT 2:00
WARNLONGTRANS 2h, CHECKINTERVAL 3m
EXTTRAIL ./dirdat/st, MEGABYTES 200
DYNAMICRESOLUTION
TRANLOGOPTIONS CONVERTUCS2CLOBS
TRANLOGOPTIONS RAWDEVICEOFFSET 0
DDL &
INCLUDE MAPPED OBJTYPE 'table' &
INCLUDE MAPPED OBJTYPE 'index' &
INCLUDE MAPPED OBJTYPE 'SEQUENCE' &
EXCLUDE OPTYPE COMMENT
DDLOPTIONS NOCROSSRENAME REPORT
TABLE OGG_TEST.*;
SEQUENCE OGG_TEST.*;

GETUPDATEBEFORES
```

增加extract进程（ext后的名字要跟上面extract对应，本例中extract是组名）add ext extract,tranlog, begin

now

删除某废弃进程DP_TESTdelete ext DP_TEST

添加抽取进程，每个队列文件大小为200madd exttrail ./dirdat/st,ext extract, megabytes 200

启动抽取进程（运行日志在ggate/dirrpt中）start extract extract至此，extract配置完成，数据库的一条变更可以在ggate/dirdat目录下的文件中看到

## 5. 生成def文件

源端ggsci起来后执行如下命令，生成defgen文件,并且拷贝到目标端dirdef下edit params defgen

```
DEFSFILE ./dirdef/ogg_test.def
USERID ogg_test, PASSWORD ogg_test
table OGG_TEST.*;
```

在shell中执行如下命令，生成ogg_test.def./defgen paramfile ./dirprm/defgen.prm

## 6. 目标端OGG安装和配置

解压adapter包将源端中dirdef/ogg_test.def文件拷贝到adapter的dirdef下

执行ggsci起来后执行如下命令，创建必须目录create subdirs

编辑mgr配置edit params mgr

```
PORT 7839
DYNAMICPORTLIST 7840-7849
PURGEOLDEXTRACTS ./dirdat/*, USECHECKPOINTS, MINKEEPDAYS 7
LAGREPORTHOURS 1
LAGINFOMINUTES 30
LAGCRITICALMINUTES 45
PURGEDDLHISTORY MINKEEPDAYS 3, MAXKEEPDAYS 7
PURGEMARKERHISTORY MINKEEPDAYS 3, MAXKEEPDAYS 7
```

启动mgrstart mgr

## 7. 源端ogg pump配置

启动ggsci后执行如下操作：

编辑pump配置edit params pump

```
EXTRACT pump
RMTHOST xx.xx.xx.xx, MGRPORT 7839, COMPRESS
PASSTHRU
NUMFILES 5000
RMTTRAIL ./dirdat/st
```

```
DYNAMICRESOLUTION
TABLE OGG_TEST.*;
SEQUENCE OGG_TEST.*;
```

添加投递进程，从某一个队列开始投add ext pump,exttrailsource ./dirdat/st

备注：投递进程，每个队文件大小为200madd rmttrail ./dirdat/st,ext pump,megabytes 200

启动pumpstart pump启动后，结合上面adapter的配置，可以在目标端的dirdat目录下看到过来的trailfile

# 8. Datahub插件安装和配置

依赖环境：jdk1.7。配置好JAVA_HOME, LD_LIBRARY_PATH，可以将环境变量配置到~/.bash_profile中，例如

```
export JAVA_HOME=/xxx/xxx/jrexx
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:$JAVA_HOME/lib/amd64:$JAVA_HOME/lib/amd64/server
```

修改环境变量后，请重启adapter的mgr进程下载datahub-ogg-plugin.tar.gz并解压:

修改conf路径下的javaue.properties文件，将{YOUR_HOME}替换为解压后的路径

```
gg.handlerlist=ggdatahub

gg.handler.ggdatahub.type=com.aliyun.odps.ogg.handler.datahub.DatahubHandler
gg.handler.ggdatahub.configureFileName={YOUR_HOME}/datahub-ogg-plugin/conf/configure.xml

goldengate.userexit.nochkpt=false
goldengate.userexit.timestamp=utc

gg.classpath={YOUR_HOME}/datahub-ogg-plugin/lib/*
gg.log.level=debug

jvm.bootoptions=-Xmx512m -Dlog4j.configuration=file:{YOUR_HOME}/datahub-ogg-plugin/conf/log4j.properties -
Djava.class.path=ggjava/ggjava.jar
```

修改conf路径下的log4j.properties文件，将{YOUR_HOME}替换为解压后的路径

修改conf路径下的configure.xml文件，修改方式见文件中的注释

```
<?xml version="1.0" encoding="UTF-8"?>
<configue>

<defaultOracleConfigure>
<!-- oracle sid, 必选-->
<sid>100</sid>
<!-- oracle schema, 可以被mapping中的oracleSchema覆盖, 两者必须有一个非空-->
<schema>ogg_test</schema>
</defaultOracleConfigure>

<defalutDatahubConfigure>
```

```
<!-- datahub endpoint, 必填-->
<endPoint>YOUR_DATAHUB_ENDPOINT</endPoint>
<!-- datahub project, 可以被mapping中的datahubProject, 两者必须有一个非空-->
<project>YOUR_DATAHUB_PROJECT</project>
<!-- datahub accessId, 可以被mapping中的datahubAccessId覆盖, 两者必须有一个非空-->
<accessId>YOUR_DATAHUB_ACCESS_ID</accessId>
<!-- datahub accessKey, 可以被mapping中的datahubAccessKey覆盖, 两者必须有一个非空-->
<accessKey>YOUR_DATAHUB_ACCESS_KEY</accessKey>
<!-- 数据变更类型同步到datahub对应的字段，可以被columnMapping中的ctypeColumn覆盖 -->
<ctypeColumn>optype</ctypeColumn>
<!-- 数据变更时间同步到datahub对应的字段，可以被columnMapping中的ctimeColumn覆盖 -->
<ctimeColumn>readtime</ctimeColumn>
<!-- 数据变更序号同步到datahub对应的字段, 按数据变更先后递增, 不保证连续, 可以被columnMapping中的cidColumn覆
盖 -->
<cidColumn>record_id</cidColumn>
</defalutDatahubConfigure>

<!-- 默认最严格，不落文件 直接退出 无限重试-->

<!-- 运行每批上次的最多纪录数, 可选, 默认1000-->
<batchSize>1000</batchSize>

<!-- 默认时间字段转换格式, 可选, 默认yyyy-MM-dd HH:mm:ss-->
<defaultDateFormat>yyyy-MM-dd HH:mm:ss</defaultDateFormat>

<!-- 脏数据是否继续, 可选, 默认false-->
<dirtyDataContinue>true</dirtyDataContinue>

<!-- 脏数据文件, 可选, 默认datahub_ogg_plugin.dirty-->
<dirtyDataFile>datahub_ogg_plugin.dirty</dirtyDataFile>

<!-- 脏数据文件最大size, 单位M, 可选, 默认500-->
<dirtyDataFileMaxSize>200</dirtyDataFileMaxSize>

<!-- 重试次数, -1:无限重试 0:不重试 n:重试次数, 可选, 默认-1-->
<retryTimes>0</retryTimes>

<!-- 重试间隔, 单位毫秒, 可选, 默认3000-->
<retryInterval>4000</retryInterval>

<!-- 点位文件, 可选, 默认datahub_ogg_plugin.chk-->
<checkPointFileName>datahub_ogg_plugin.chk</checkPointFileName>

<mappings>
<mapping>
<!-- oracle schema, 见上描述-->
<oracleSchema></oracleSchema>
<!-- oracle table, 必选-->
<oracleTable>t_person</oracleTable>
<!-- datahub project, 见上描述-->
<datahubProject></datahubProject>
<!-- datahub AccessId, 见上描述-->
<datahubAccessId></datahubAccessId>
<!-- datahub AccessKey, 见上描述-->
<datahubAccessKey></datahubAccessKey>
<!-- datahub topic, 必选-->
```

```
<datahubTopic>t_person</datahubTopic>
<ctypeColumn></ctypeColumn>
<ctimeColumn></ctimeColumn>
<cidColumn></cidColumn>
<columnMapping>
<!--
src:oracle字段名称, 必须;
dest:datahub field, 必须;
destOld:变更前数据落到datahub的field, 可选;
isShardColumn: 是否作为shard的hashkey, 可选, 默认为false, 可以被shardId覆盖
isDateFormat: timestamp字段是否采用DateFormat格式转换, 默认true. 如果是false, 源端数据必须是long
dateFormat: timestamp字段的转换格式, 不填就用默认值
-->
<column src="id" dest="id" isShardColumn="true" isDateFormat="false" dateFormat="yyyy-MM-dd HH:mm:ss"/>
<column src="name" dest="name" isShardColumn="true"/>
<column src="age" dest="age"/>
<column src="address" dest="address"/>
<column src="comments" dest="comments"/>
<column src="sex" dest="sex"/>
<column src="temp" dest="temp" destOld="temp1"/>
</columnMapping>

<!--指定shard id, 优先生效, 可选-->
<shardId>1</shardId>
</mapping>
</mappings>
</configue>
```

在ggsci下启动datahub writer

edit params dhwriter

```
extract dhwriter
getEnv (JAVA_HOME)
getEnv (LD_LIBRARY_PATH)
getEnv (PATH)
CUSEREXIT ./libggjava_ue.so CUSEREXIT PASSTHRU INCLUDEUPDATEBEFORES, PARAMS "{YOUR_HOME}/datahub-ogg-plugin/conf/javaue.properties"
sourcedefs ./dirdef/ogg_test.def
table OGG_TEST.*;
```

添加dhwriteradd extract dhwriter, exttrailsource ./dirdat/st

启动dhwriterstart dhwriter

# 三、使用场景

这里会用一个简单的示例来说明数据的使用方法，例如我们在Oracle数据库有一张商品订单表orders（oid int, pid int, num int），该表有三列，分别为订单ID, 商品ID和商品数量。将这个表通过OGG Datahub进行增量数据同步之前，我们需要先将源表已有的数据通过DataX同步到MaxCompute中。增量同步的关键步骤如下：（1）在Datahub上创建相应的Topic，Topic的schema为(string record_id, string optype, string

readtime, bigint oid_before, bigint oid_after, bigint pid_before, bigint pid_after, bigint num_before, bigint num_after)；（2）OGG Datahub的插件按照上述的安装流程部署配置，其中列的Mapping配置如下：

```
        <ctypeColumn>optype</ctypeColumn>
<ctimeColumn>readtime</ctimeColumn>
<cidColumn>record_id</cidColumn>
<columnMapping>
<column src="oid" dest="oid_after" destOld="oid_before" isShardColumn="true"/>
<column src="pid" dest="pid_after" destOld="pid_before"/>
<column src="num" dest="num_after" destOld="num_before"/>
</columnMapping>
```

其中optype和readtime字段是记录数据库的数据变更类型和时间，optype有”I”，“D”，“U”三种取值，分别对应为“增”，“删”，“改”三种数据变更操作。（3）OGG Datahub插件部署好成功运行后，插件会源源不断的将源表的数据变更记录输送至datahub中，例如我们在源订单表中新增一条记录（1，2，1），datahub里收到的记录如下：

```
+--------+------------+------------+------------+------------+------------+------------+------------+------------+
| record_id | optype | readtime | oid_before | oid_after | pid_before | pid_after | num_before | num_after |
+-------+------------+------------+------------+------------+------------+------------+------------+------------+
| 14810373343020000 | I | 2016-12-06 15:15:28.000141 | NULL | 1 | NULL | 2 | NULL | 1 |
```

修改这条数据，比如把num改为20，datahub则会收到的一条变更数据记录，如下：

```
+-------+------------+------------+------------+------------+------------+------------+------------+------------+
| record_id | optype | readtime | oid_before | oid_after | pid_before | pid_after | num_before | num_after |
+--------+------------+------------+------------+------------+------------+------------+------------+------------+
| 14810373343080000 | U | 2016-12-06 15:15:58.000253 | 1 | 1 | 2 | 2 | 1 | 20 |
```

## 实时计算

在前一天的离线计算的基础数据上，我们可以写一个StreamCompute流计算的分析程序，很容易地对数据进行实时汇总，例如实时统计当前总的订单数，每种商品的销售量等。处理思路就是对于每一条到来的变更数据，可以拿到变化的数值，实时更新统计变量即可。

## 离线处理

为了便于后续的离线分析，我们也可以将Datahub里的数据归档到MaxCompute中，在MaxCompute中创建相应Schema的表：

```
create table orders_log(record_id string, optype string, readtime string, oid_before bigint, oid_after bigint,
pid_before bigint, pid_after bigint, num_before bigint, num_after bigint);
```

在Datahub上创建MaxCompute的数据归档，上述流入Datahub里的数据将自动同步到MaxCompute当中。

建议将同步到MaxCompute中的数据按照时间段进行划分，比如每一天的增量数据都对应一个独立分区。这样当天的数据同步完成后，我们可以处理对应的分区，拿到当天所有的数据变更，而与和前一天的全量数据进行合并后，即可得到当天的全量数据。为了简单起见，先不考虑分区表的情况，以2016-12-06这天的增量数据为例，假设前一天的全量数据在表orders_base里面，datahub同步过来的增量数据在orders_log表中，将orders_base与orders_log做合并操作，可以得到2016-12-06这天的最终全量数据写入表orders_result中。这个过程可以在MaxCompute上用如下这样一条SQL完成。

```
INSERT OVERWRITE TABLE orders_result
SELECT t.oid, t.pid, t.num
FROM
(
SELECT oid, pid, num, '0' x_record_id, 1 AS x_optype
FROM
orders_base
UNION ALL
SELECT decode(optype,'D',oid_before,oid_after) AS oid
, decode(optype,'D', pid_before,pid_after) AS pid
, num_after AS num
, record_id x_record_id
, decode(optype, 'D', 0, 1) AS x_optype
FROM
orders_log
) t
JOIN
(
SELECT
oid
, pid
, max(record_id) x_max_modified
FROM
(
SELECT
oid
, pid
, '0' record_id
FROM
orders_base UNION ALL SELECT
decode(optype,'D',oid_before,oid_after) AS oid
, decode(optype,'D', pid_before,pid_after) AS pid
, record_id
FROM
orders_log ) g
GROUP BY oid , pid
) s
ON
t.oid = s.oid AND t.pid = s.pid AND t.x_record_id = s.x_max_modified AND t.x_optype <> 0;
```

# 四、常见问题

Q：目标端报错 OGG-06551 Oracle GoldenGate Collector: Could not translate host name localhost into an Internet address.

A：目标端机器hostname在/etc/hosts里面重新设置localhost对应的ip

Q：找不到jvm相关的so包

A：将jvm的so路径添加到LD_LIBRARY_PATH后，重启mgr

```
例如：export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:$JAVA_HOME/lib/amd64:$JAVA_HOME/lib/amd64/server
```

Q：有了DDL语句，比如增加一列，源端ogg没有问题，但是adapter端的ffwriter和jmswriter进程退出，且报错： 2015-06-11 14:01:10 ERROR OGG-01161 Bad column index (5) specified for table OGG_TEST.T_PERSON, max columns = 5.

A：由于表结构改变，需要重做def文件，将重做的def文件放入dirdef后重启即可

# SDK介绍

## Java SDK介绍

### 概要

Java SDK通过Maven管理,pom信息如下

```
<dependency>
<groupId>com.aliyun.datahub</groupId>
<artifactId>aliyun-sdk-datahub</artifactId>
<version>2.3.0-public</version>
</dependency>
```

## Java SDK Example

### 准备工作

访问DataHub服务需要使用阿里云认证账号，需要提供阿里云accessId及accessKey。 同时需要提供访问的服务地址，公网服务地址为：

```
http://dh-cn-hangzhou.aliyuncs.com
```

初始化AliyunAccount与DatahubConfiguration

```
String accessId = "Your AccessId";
```

```
String accessKey = "Your AccessKey";
String endpoint = "http://dh-cn-hangzhou.aliyuncs.com";
AliyunAccount account = new AliyunAccount(accessId, accessKey);
DatahubConfiguration conf = new DatahubConfiguration(account, endpoint);
```

初始化DataHubClient, DataHub服务所有操作均可用该client进行

```
DatahubClient client = new DatahubClient(conf);
```

## TupleTopic使用样例

### 创建Tuple Topic

```
RecordSchema schema = new RecordSchema();
schema.addField(new Field("a", FieldType.STRING));
schema.addField(new Field("b", FieldType.BIGINT));
int shardCount = 5;
int lifeCycle = 3;
String topicName = "topic_example";
String topicDesc = "topic_example_desc";

client.createTopic(projectName, topicName, shardCount, lifeCycle, RecordType.TUPLE, schema, topicDesc);
//等待服务端通道打开
client.waitForShardReady(projectName, topicName);
```

### 获取Shard列表

```
ListShardResult listShardResult = client.listShard(projectName, topicName);
```

### 写入Tuple数据

准备数据：

```
List<RecordEntry> recordEntries = new ArrayList<RecordEntry>();
// 此处可选用listShardResult任意status为ACTIVE的shard进行写入，本例取第一个shard写入
String shardId = listShardResult.getShards().get(0).getShardId();

RecordEntry entry = new RecordEntry(schema);
entry.setString(0, "Test");
entry.setBigint(1, 5L);
entry.setShardId(shardId);
recordEntries.add(entry);

PutRecordsResult result = client.putRecords(projectName, topicName, recordEntries);
if (result.getFailedRecordCount() != 0) {
List<ErrorEntry> errors = result.getFailedRecordError();
// deal with result.getFailedRecords()
}
```

更多写入方式请参考多方式数据写入

### 消费Tuple数据

获取Cursor，可以通过三种方式获取：” OLDEST”，“LATEST”，“SYSTEM_TIME”

```
GetCursorResult cursorRs = client.getCursor(projectName, topicName, shardId,
GetCursorRequest.CursorType.OLDEST);
//GetCursorResult cursorRs = client.getCursor(projectName, topicName, shardId, System.currentTimeMillis() - 24 *
3600 * 1000 /* ms */); 可以获取到24小时内的第一条数据Cursor
```

读取数据:

```
int limit = 100;
String cursor = cursorRs.getCursor();
while (true) {
try {
GetRecordsResult recordRs = client.getRecords(projectName, topicName, shardId, cursor, limit, schema);
List<RecordEntry> recordEntries = recordRs.getRecords();
if (recordEntries.size() == 0) {
// 无最新数据，请稍等重试
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
// 拿到下一个游标
cursor = recordRs.getNextCursor();
} catch (InvalidCursorException ex) {
// 非法游标或游标已过期，建议重新定位后开始消费
cursorRs = client.getCursor(projectName, topicName, shardId, GetCursorRequest.CursorType.OLDEST);
cursor = cursorRs.getCursor();
} catch (DatahubClientException ex) {
// 发生异常，需要重试
System.out.printf(ex.getMessage());
ex.printStackTrace();
}
}
```

## BlobTopic写入二进制数据使用样例

### 创建Blob Topic

```
int shardCount = 5;
int lifeCycle = 3;
String topicName = "topic_example";
String topicDesc = "topic_example_desc";

client.createTopic(projectName, topicName, shardCount, lifeCycle, RecordType.BLOB, topicDesc);
//等待服务端通道打开
```

```
client.waitForShardReady(projectName, topicName);
```

**获取Shard列表**

同TupleTopic

**写入Blob二进制数据**

准备数据：

```
List<BlobRecordEntry> recordEntries = new ArrayList<BlobRecordEntry>();
// 此处可选用listShardResult任意status为ACTIVE的shard进行写入，本例取第一个shard写入
String shardId = listShardResult.getShards().get(0).getShardId();
String data = String.valueOf(System.currentTimeMillis());

BlobRecordEntry entry = new BlobRecordEntry();
entry.setData(data.getBytes());

recordEntries.add(entry);

PutBlobRecordsResult result = client.putBlobRecords(projectName, topicName, recordEntries);
if (result.getFailedRecordCount() != 0) {
List<ErrorEntry> errors = result.getFailedRecordError();
// deal with result.getFailedRecords()
}
```

**消费Blob数据**

获取Cursor，同TupleTopic.

读取数据:

```
int limit = 100;
String cursor = cursorRs.getCursor();
while (true) {
try {
GetBlobRecordsResult recordRs = client.getBlobRecords(projectName, topicName, shardId, cursor, limit);
List<BlobRecordEntry> recordEntries = recordRs.getRecords();

if (recordEntries.size() == 0) {
// 无最新数据，请稍等重试
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
// 拿到下一个游标
cursor = recordRs.getNextCursor();
} catch (InvalidCursorException ex) {
// 非法游标或游标已过期，建议重新定位后开始消费
cursorRs = client.getCursor(projectName, topicName, shardId, GetCursorRequest.CursorType.OLDEST);
cursor = cursorRs.getCursor();
```

```
} catch (DatahubClientException ex) {
// 发生异常，需要重试
System.out.printf(ex.getMessage());
ex.printStackTrace();
}
}
```

# 安装

## 快速安装

```
$ sudo pip install pydatahub
```

## 源码安装

```
$ git clone https://github.com/aliyun/aliyun-datahub-sdk-python.git
$ cd aliyun-datahub-sdk-python
$ sudo python setup.py install
```

## 安装验证

```
$ python -c "from datahub import DataHub"
```

如果上述命令执行成功，恭喜你安装Datahub Python版本SDK成功！

# 基本概念

详见: https://help.aliyun.com/document_detail/47440.html?spm=5176.product27797.3.2.VGxgya

# 准备工作

- 访问DataHub服务需要使用阿里云认证账号，需要提供阿里云accessId及accessKey。 同时需要提供访问的服务地址。
- 创建Project
    - 登陆Datahub WebConsole页面，创建Project

- 初始化Datahub

```
import sys
import traceback

from datahub import DataHub
from datahub.utils import Configer
from datahub.models import Topic, RecordType, FieldType, RecordSchema, BlobRecord, TupleRecord, CursorType
from datahub.errors import DatahubException, ObjectAlreadyExistException

access_id = ***your access id***
access_key = ***your access key***
endpoint = ***your datahub server endpoint***
dh = DataHub(access_id, access_key, endpoint)
```

# Topic操作

## Tuple Topic

- Tuple类型Topic写入的数据是有格式的，需要指定Record Schema，目前支持以下几种数据类型:

| 类型 | 含义 | 值域 |
| --- | --- | --- |
| Bigint | 8字节有符号整型。请不要使用整型的最小值 (-9223372036854775808)，这是系统保留值。 | -9223372036854775807 ~ 9223372036854775807 |
| String | 字符串，只支持UTF-8编码。 | 单个String列最长允许1MB。 |
| Boolean | 布尔型。 | 可以表示为True/False，true/false, 0/1 |
| Double | 8字节双精度浮点数。 | -1.0 10308 ~ 1.0 10308 |
| TimeStamp | 时间戳类型 | 表示到微秒的时间戳类型 |

- 创建示例

```
topic = Topic(name=topic_name)
topic.project_name = project_name
topic.shard_count = 3
topic.life_cycle = 7
topic.record_type = RecordType.TUPLE
topic.record_schema = RecordSchema.from_lists(['bigint_field', 'string_field', 'double_field', 'bool_field', 'time_field'],
[Fie
ldType.BIGINT, FieldType.STRING, FieldType.DOUBLE, FieldType.BOOLEAN, FieldType.TIMESTAMP])

try:
dh.create_topic(topic)
```

```
print "create topic success!"
print "========================================\n\n"
except ObjectAlreadyExistException, e:
print "topic already exist!"
print "========================================\n\n"
except Exception, e:
print traceback.format_exc()
sys.exit(-1)
```

## Blob Topic

- Blob类型Topic支持写入一块二进制数据作为一个Record，数据将会以BASE64编码传输。

```
topic = Topic(name=topic_name)
topic.project_name = project_name
topic.shard_count = 3
topic.life_cycle = 7
topic.record_type = RecordType.BLOB

try:
dh.create_topic(topic)
print "create topic success!"
print "========================================\n\n"
except ObjectAlreadyExistException, e:
print "topic already exist!"
print "========================================\n\n"
except Exception, e:
print traceback.format_exc()
sys.exit(-1)
```

# 数据发布/订阅

## 获取Shard列表

- list_shards接口获取topic下的所有shard

```
shards = dh.list_shards(project_name, topic_name)
```

返回结果是一个List对象，每个元素是一个shard，可以获取shard_id，state状态
，begin_hash_key，end_hash_key等信息

## 发布数据

- put_records接口向一个topic发布数据

```
failed_indexs = dh.put_records(project_name, topic_name, records)
```

其中传入参数records是一个List对象，每个元素为一个record，但是必须为相同类型的record，即Tuple类型或者Blob类型，返回结果为写入失败记录的数组下标

- 写入Tuple类型Record示例

```
try:
# block等待所有shard状态ready
dh.wait_shards_ready(project_name, topic_name)
print "shards all ready!!!"
print "========================================\n\n"

topic = dh.get_topic(topic_name, project_name)
print "get topic suc! topic=%s" % str(topic)
if topic.record_type != RecordType.TUPLE:
print "topic type illegal!"
sys.exit(-1)
print "========================================\n\n"

shards = dh.list_shards(project_name, topic_name)
for shard in shards:
print shard
print "========================================\n\n"

records = []

record0 = TupleRecord(schema=topic.record_schema, values=[1, 'yc1', 10.01, True, 1455869335000000])
record0.shard_id = shards[0].shard_id
record0.put_attribute('AK', '47')
records.append(record0)

record1 = TupleRecord(schema=topic.record_schema)
record1['bigint_field'] = 2
record1['string_field'] = 'yc2'
record1['double_field'] = 10.02
record1['bool_field'] = False
record1['time_field'] = 1455869335000011
record1.shard_id = shards[1].shard_id
records.append(record1)

record2 = TupleRecord(schema=topic.record_schema)
record2['bigint_field'] = 3
record2['string_field'] = 'yc3'
record2['double_field'] = 10.03
record2['bool_field'] = False
record2['time_field'] = 1455869335000013
record2.shard_id = shards[2].shard_id
records.append(record2)

failed_indexs = dh.put_records(project_name, topic_name, records)
print "put tuple %d records, failed list: %s" %(len(records), failed_indexs)
# failed_indexs如果非空最好对failed record再进行重试
```

```
print "========================================\n\n"
except DatahubException, e:
print traceback.format_exc()
sys.exit(-1)
else:
sys.exit(-1)
```

# 获取cursor

获取Cursor，可以通过三种方式获取：OLDEST, LATEST, SYSTEM_TIME

OLDEST: 表示获取的cursor指向当前有效数据中时间最久远的record

LATEST: 表示获取的cursor指向当前最新的record

SYSTEM_TIME: 表示获取的cursor指向该时间之后接收到的第一条record

```
cursor = dh.get_cursor(project_name, topic_name, CursorType.OLDEST, shard_id)
```

通过get_cursor接口获取用于读取指定位置之后数据的cursor

# 订阅数据

- 从指定shard读取数据，需要指定从哪个Cursor开始读，并指定读取的上限数据条数，如果从Cursor到shard结尾少于Limit条数的数据，则返回实际的条数的数据。

```
dh.get_records(topic, shard_id, cursor, 10)
```

- 消费Tuple类型Record示例

```
try:
# block等待所有shard状态ready
dh.wait_shards_ready(project_name, topic_name)
print "shards all ready!!!"
print "========================================\n\n"

topic = dh.get_topic(topic_name, project_name)
print "get topic suc! topic=%s" % str(topic)
if topic.record_type != RecordType.TUPLE:
print "topic type illegal!"
sys.exit(-1)
print "========================================\n\n"
```

```
cursor = dh.get_cursor(project_name, topic_name, CursorType.OLDEST, '0')
while True:
(record_list, record_num, next_cursor) = dh.get_records(topic, '0', cursor, 10)
for record in record_list:
print record
if 0 == record_num:
time.sleep(1)
cursor = next_cursor

except DatahubException, e:
print traceback.format_exc()
sys.exit(-1)
else:
sys.exit(-1)
```

# 结尾

Python API Doc

Python Package Index

Github地址

# 高级特性

## 数据归档MaxCompute Connector

DataHub Connector是把 DataHub 中的实时数据归档到其他存储系统的功能，目前只支持将Topic中的数据归档到MaxCompute(ODPS)中。数据归档到MaxCompute支持at least once语义，在网络服务异常等场景下可能会导致导入到MaxCompute中的数据产生重复。

### 如何创建

创建Connector主要需要如下前置条件：

准备对应的MaxCompute表，该表**字段类型、名称、顺序**必须与DataHub Topic字段完全一致，如果三个条件中的任意一个不满足，则归档Connector无法创建。字段类型对应表见后表。

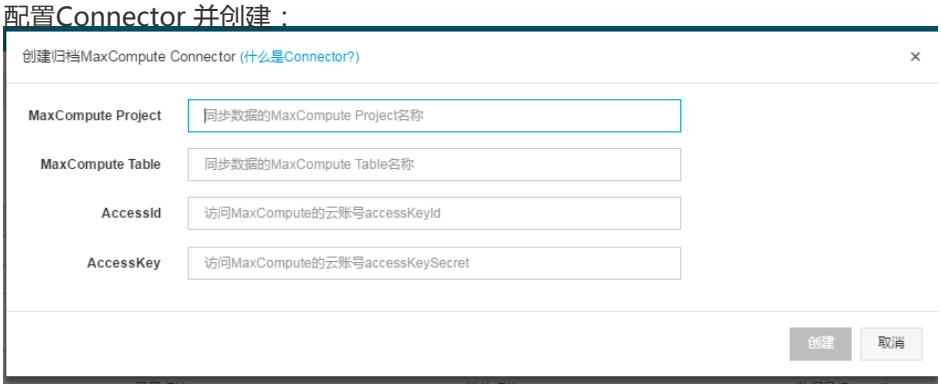
访问MaxCompute账号的设置，该账号必须具备该MaxCompute的Project的CreateInstance权限和归档MaxCompute表的Desc、Alter、Update权限，建议使用一个特殊最小权限的账号(如何配置访问MaxCompute账号权限？)。建议使用RAM用户账号(如何创建RAM用户账号？)。

DataHub Topic的Owner/Creator账号, 才有相应的权限操作Connector，包括创建，删除等。

操作流程： Project列表->Project查看->Topic查看->点击归档MaxCompute->填写配置，点击创建

进入Topic的详情页面：

配置Connector 并创建：




在Connector页面查看数据归档状态：


## 配置说明

| 名称 | 是否必须 | 描述 |
| --- | --- | --- |
| MaxCompute Project | yes | MaxCompute Project名称 |
| MaxCompute Table | yes | MaxCompute表名称 |
| AccessId | yes | 访问MaxCompute的阿里云账号AccessID |
| AccessKey | yes | 访问MaxCompute的阿里云账号AccessKey |

## 注意

- 支持MaxCompute分区表，例如：

MaxCompute表：

```
table_test(f1 string, f2 string, f3 double) partitioned by (pt string)
```

对应Topic应为如下:

```
topic_test(f1 string, f2 string, f3 double, pt string)
```

> MaxCompute分区字段必须为STRING类型。
>
> 数据归档的频率为每个Shard每5分钟或者Shard中新写入的数据量达到64MB，Connector服务会批量进行一次数据归档进入MaxCompute表的操作。所以数据写入DataHub Topic后至多5分钟后在MaxCompute可以被查询到。

## DataHub与MaxCompute字段类型对应表

| MaxCompute表中的类型 | DataHub Topic中的类型 |
| --- | --- |
| STRING | STRING |
| DOUBLE | DOUBLE |
| BIGINT | BIGINT |
| DATETIME | TIMESTAMP |
| BOOLEAN | BOOLEAN |
| DECIMAL | 不支持 |
| MAP | 不支持 |
| ARRAY | 不支持 |

DataHub 支持为Topic动态扩容/缩容,通过SplitShard/MergeShard来实现。

## 使用场景

Datahub具有服务弹性伸缩功能，用户可根据实时的流量调整Shard数量，来应对突发性的流量增长或达到节约资源的目的。

例如在双11大促期间，大部分Topic数据流量会激增，平时的Shard数量可能完全无法满足这样的流量增长，此时可以对其中一些Shard进行Split操作，一变二，二变四，最大可扩容至256个Shard，按目前的流控限制足以达到256MB/s的流量。

在双11大促后，流量下降，多余的Shard会占用没有必要的quota，因此可以进行Merge操作，每两个Shard合

并为一个，直到合适为止。

## Shard属性

可以通过ListShard接口获取所有Shard的信息，每个Shard拥有如下属性, 样例：

```
{
"ShardId": "string",
"State": "string",
"ClosedTime": uint64,
"BeginHashKey": "string",
"EndHashKey": "string",
"ParentShardIds": [string,string,],
"LeftShardId": "string",
"RightShardId": "string"
}
```

## SplitShard

指定一个128 bit的HashKey以及一个ShardID，通过SDK或者Console进行操作.

SplitShard操作会将指定的Shard分裂为两个ChildShard，并且返回ChildShard的Id以及Key信息，同时Parent Shard会被置为CLOSED状态。

例如，Split之前存在如下一个Shard：

```
ShardId:0 Status:ACTIVE BeginHashKey:00000000000000000000000000000000
EndHashKey:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

通过SDK进行Split操作：

```
String shardId = "0";
SplitShardRequest req = new SplitShardRequest(projectName, topicName, shardId,
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
SplitShardResult resp = client.splitShard(req);
```

最终将会变成如下3个Shard：

```
ShardId:0 Status:CLOSED BeginHashKey:00000000000000000000000000000000
EndHashKey:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
ShardId:1 Status:ACTIVE BeginHashKey:00000000000000000000000000000000
EndHashKey:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ShardId:2 Status:ACTIVE BeginHashKey:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EndHashKey:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

## MergeShard

指定两个相邻的ShardID，通过SDK或者Console进行操作.

MergeShard操作会将指定的两个Shard合并为一个新的Shard，并且返回新Shard的ID以及Key信息，同时两个ParentShard会被置为为CLOSED状态。

例如，Merge之前存在如下两个Shard：

```
ShardId:0 Status:ACTIVE BeginHashKey:00000000000000000000000000000000
EndHashKey:7FFFFFFFFFFFFFFF7FFFFFFFFFFFFFFF
ShardId:1 Status:ACTIVE BeginHashKey:7FFFFFFFFFFFFFFF7FFFFFFFFFFFFFFF
EndHashKey:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

通过SDK进行Merge操作：

```
String shardId = "0";
String adjacentShardId = "1";
MergeShardRequest req = new MergeShardRequest(projectName, topicName, shardId, adjacentShardId);
MergeShardResult resp = client.mergeShard(req);
```

最终将会变成如下3个Shard：

```
ShardId:0 Status:CLOSED BeginHashKey:00000000000000000000000000000000
EndHashKey:7FFFFFFFFFFFFFFF7FFFFFFFFFFFFFFF
ShardId:1 Status:CLOSED BeginHashKey:7FFFFFFFFFFFFFFF7FFFFFFFFFFFFFFF
EndHashKey:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
ShardId:2 Status:ACTIVE BeginHashKey:00000000000000000000000000000000
EndHashKey:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

## 注意事项

当Shard进行Merge/Split后会被置为CLOSED状态，该状态可以继续消费读取数据，但是不可写入，也不可再次进行Merge/Split操作，当到达Topic的lifecycle后该Shard会被回收。

如果配置了Connector，对应任务会在复制完该Shard数据后自动挂起，待该Shard回收后会自动删除任务。

Topic在进行Merge/Split后新的Shard需要等待变为ACTIVE状态后方可正常使用，通常不会超过5秒。

**多方式数据写入（Hash/PartitionKey）**

DataHub服务支持三种写入方式：

## 按ShardID写入

指定写入某个Shard，该场景主要用于用户需要保证每个通道中数据有序，因此需要将部分数据指定写入到某个Shard中。样例代码：

```
// 新建client
```

```
Account account = new AliyunAccount("your access id", "your access key");
DatahubConfiguration conf = new DatahubConfiguration(account, "datahub endpoint");
DatahubClient client = new DatahubClient(conf);

// 构造需要上传的records
RecordSchema schema = client.getTopic("projectName", "topicName").getRecordSchema();
List<RecordEntry> recordEntries = new ArrayList<~>();
RecordEntry entry = new RecordEntry(schema);
for (int i=0; i<entry.getFieldCount(); i++) {
entry.setBigint(i, 1);
}
entry.setShardId("shardId");
recordEntries.add(entry);

// 数据写入
client.putRecords("projectName", "topicName", recordEntries);
```

## 按HashKey写入

指定一个128 bit的MD5值。 按照HashKey写入，根据Shard的beginHashKey与endHashKey决定数据写入的Shard。

该种方式的写入场景主要用于用户不关心数据的写入顺序，根据某个字段值或用户维护的key来进行写入。

```
// 新建client
Account account = new AliyunAccount("your access id", "your access key");
DatahubConfiguration conf = new DatahubConfiguration(account, "datahub endpoint");
DatahubClient client = new DatahubClient(conf);

// 构造需要上传的records
RecordSchema schema = client.getTopic("projectName", "topicName").getRecordSchema();
List<RecordEntry> recordEntries = new ArrayList<~>();
RecordEntry entry = new RecordEntry(schema);
for (int i=0; i<entry.getFieldCount(); i++) {
entry.setBigint(i, 1);
}
entry.setHashKey("7FFFFFFFFFFFFFFD7FFFFFFFFFFFFFFD");
recordEntries.add(entry);

// 数据写入
client.putRecords("projectName", "topicName", recordEntries);
```

## 按PartitionKey写入

指定一个String类型参数作为PartitionKey,系统根据该String的MD5值以及Shard的beginHashKey与endHashKey决定写入的Shard。

该种方式的应用场景与按HashKey写入方式类似，区别在于用户不需要提供固定范围的HashKey，而是通过一个字符串Key，系统会计算出其对应的HashKey进行写入。

```
// 新建client
```

```
Account account = new AliyunAccount("your access id", "your access key");
DatahubConfiguration conf = new DatahubConfiguration(account, "datahub endpoint");
DatahubClient client = new DatahubClient(conf);

// 构造需要上传的records
RecordSchema schema = client.getTopic("projectName", "topicName").getRecordSchema();
List<RecordEntry> recordEntries = new ArrayList<~>();
RecordEntry entry = new RecordEntry(schema);
for (int i=0; i<entry.getFieldCount(); i++) {
entry.setBigint(i, 1);
}
entry.setPartitionKey("TestPartitionKey");
recordEntries.add(entry);

// 数据写入
client.putRecords("projectName", "topicName", recordEntries);
```

# FAQ常见问题

**1. 访问DataHub服务的域名是什么**

| 地区 | Region | 外网Endpoint | 经典网络ECS Endpoint（金融云） | VPC ECS Endpoint |
| --- | --- | --- | --- | --- |
| 华东1(杭州) | dh-cn-hangzhou | http://dh-cn-hangzhou.aliyuncs.com | http://dh-cn-hangzhou.aliyun-inc.com | http://dh-cn-hangzhou-vpc.aliyuncs.com |

**2. DataHub服务如何收费**

目前还处于公测阶段，暂时不会向用户收取任何费用。

**3. 如何申请云账号的RAM用户账号**

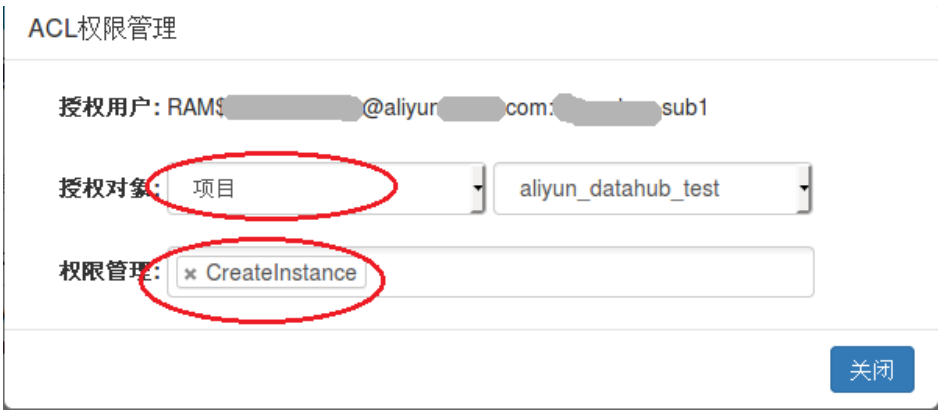
首先需要开通阿里云访问控制功能，此功能免费。创建RAM用户账号的操作步骤：登入RAM管理控制台，选择用户管理 -> 新建用户，进入创建用户页面，并勾选上“为该用户自动生成AccessKey选项”；具体操作请见链接。

**4. MaxCompute中如何配置Connector访问的账号权限**

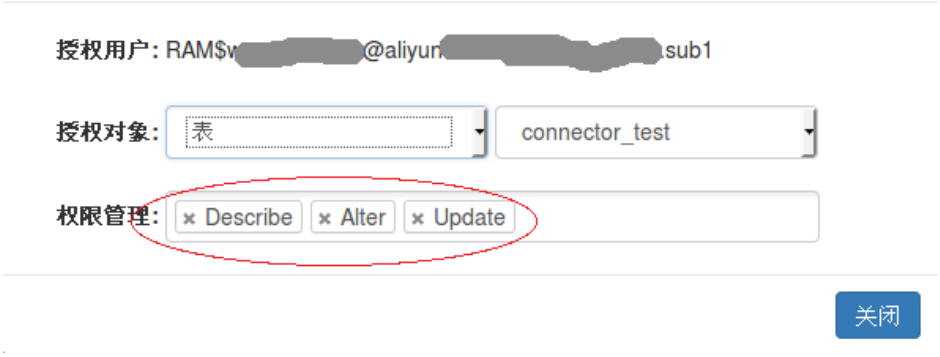
1. 确保MaxCompute Project打开RAM子账号功能


2. 添加用户到MaxCompute Project中


给用户赋MaxCompute项目的CreateInstance权限




给用户赋MaxCompute归档表的Alter, Describe, Updte权限




### 5. 读写DataHub服务性能问题排查

如果在使用期间遇到了性能瓶颈，可以按照以下步骤排查性能问题：

首先确定您的网络环境，我们提供公网/经典ECS/VPC三个不同域名，您可以选择最合适的域名使用。

DataHub是一个高性能的流式数据处理服务，对网络延时比较敏感，较高的网络延时会直接影响吞吐性能，简单的排查网络延时的办法可以通过ping命令查看。

```
$ping dh-cn-hangzhou.aliyuncs.com
PING dh-cn-hangzhou.aliyuncs.com (121.40.13.9) 56(84) bytes of data.
64 bytes from 121.40.13.9: icmp_seq=1 ttl=38 time=5.15 ms
64 bytes from 121.40.13.9: icmp_seq=2 ttl=38 time=5.16 ms
64 bytes from 121.40.13.9: icmp_seq=3 ttl=38 time=5.14 ms
64 bytes from 121.40.13.9: icmp_seq=4 ttl=38 time=5.13 ms
64 bytes from 121.40.13.9: icmp_seq=5 ttl=38 time=5.21 ms
64 bytes from 121.40.13.9: icmp_seq=6 ttl=38 time=5.16 ms
```

如果延时较低的情况吞吐量仍然不高，并且确认与DataHub性能指标差距较大，请检查本地机器的带宽情况（ECS用户建议使用经典网络或VPC域名，否则会占用出口带宽）是否较为拥堵。

### 6. 为什么Connector归档到MaxCompute系统中时间字段都变成了 1970.xx.xx

DataHub中的TimeStamp字段表示的是精度到微秒的时间戳，所以在写入DataHub系统请确保设置的是微秒的时间戳值。

# 数据上传下载

# 简介

本节是对MaxCompute系统的数据上传下载的一个综述，包含了服务连接，SDK，工具，数据上云场景几个模块。

总的来说，进出MaxCompute系统的途径可以分为两类，分别是DataHub实时数据通道和Tunnel批量数据通道。DataHub和Tunnel各自都提供了SDK，而基于这些SDK又衍生了许多用于数据上传下载的工具，方便用户各种场景下的数据上传下载需求。

数据上传下载的工具主要包括：大数据开发套件, DTS, OGG插件, Sqoop, Flume插件, LogStash插件，Flunted插件， Kettle插件以及MaxCompute客户端等。从这些工具使用的底层数据通道分类如下：

DataHub通道系列

- OGG插件
- Flume插件
- LogStash插件
- Flunted插件

Tunnel通道系列

- 大数据开发套件
- DTS
- Sqoop
- Kettle插件
- MaxCompute客户端

基于上述丰富的数据上传下载工具，大部分常见的数据上云场景都可以得到满足，后续的章节会对工具本身以及Hadoop数据迁移，数据库数据同步，日志采集等数据上云的场景进行介绍，为用户在做技术方案选型时提供参考。

Datahub和Tunnel在不同网络环境场景下，所使用的EndPoint会有区别，用户在不同网络环境下，需要

选择不同的服务地址 (Endpoint) 来连接服务，否则将无法向服务发起请求。同时，不同的网络连接也会对用户的 计费 产生影响。

下面将分别对公网、ECS经典网络以及ECS专有网络三种环境下所使用的EndPoint进行介绍。

## 公网

DataHub公网EndPoint为: **http://dh-cn-hangzhou.aliyuncs.com**

Tunnel公网EndPoint为：**http://dt.odps.aliyun.com**(使用MaxCompute的EndPoint相应为：**http://service.odps.aliyun.com/api**，可以自动路由到tunnel，无须单独配置tunnel的endpoint.)

## ECS经典网络

在ECS的经典网络环境下，DataHub的EndPoint为: **http://dh-cn-hangzhou.aliyun-inc.com**

Tunnel的EndPoint为：**http://dt-ext.odps.aliyun-inc.com**(使用MaxCompute的EndPoint相应为：**http://odps-ext.aliyun-inc.com/api**，可以自动路由到tunnel，无须单独配置tunnel的endpoint.)

## ECS专有网络（VPC）

在ECS专有网络（VPC）环境下，Tunnel和DataHub的VPC EndPoint配置如下表所示。

| 区域 | Tunnel | DataHub |
| --- | --- | --- |
| 华北2 | http://dt-ext.nu16.odps.aliyun-inc.com | 不支持，需要用户走公网访问 |
| 华东2 | http://dt-ext.eu13.odps.aliyun-inc.com | 不支持，需要用户走公网访问 |
| 华东1 | http://dt-ext.odps.aliyun-inc.com | http://dh-cn-hangzhou-vpc.aliyuncs.com |
| 华东2 | http://dt-ext.eu13.odps.aliyun-inc.com | 不支持，需要用户走公网访问 |
| 其他 | 不支持，需要用户走公网访问 | 不支持，需要用户走公网访问 |

**特别注意：华东 1 区域 Tunnel 服务，VPC 网络：http://dt-ext.odps.aliyun-inc.com （跨 Region 访问上海 Tunnel 连接地址，收费）**

# SDK

DataHub是 MaxCompute 提供的流式数据处理(Streaming Data)服务，它提供流式数据的发布 (Publish)和订阅 (Subscribe)的功能，让您可以轻松构建基于流式数据的分析和应用。DataHub 也提供流式数据归档的功能，支持流式数据归档进入MaxCompute。

DataHub提供了Java和Python两种语言的SDK可供使用：

- DataHub Java SDK介绍
- DataHub Python SDK介绍

Tunnel是MaxCompute平台的批量数据通道，MaxCompute客户端的数据上传下载工具就是基于Tunnel的SDK编写的。

Tunnel提供了Java SDK，相关介绍参考：Tunnel SDK介绍。

MaxCompute 平台的数据上传和下载目前有着丰富的工具（其中大部分已经在github 上开源，走开源社区的维护方式）可以使用，各自有不同的应用场景，具体分为阿里云数加产品和开源产品两大类，下面将对这些工具分别进行介绍：

## 阿里云数加产品

**1.大数据开发套件之数据集成**

大数据开发套件之数据集成（也叫数据同步），是阿里集团对外提供的稳定高效、弹性伸缩的数据同步平台，致力于为阿里云上各类异构数据存储系统提供离线全量和实时增量的数据同步、集成、交换服务 。其中数据同步任务支持的数据源类型包括：MaxCompute、RDS（MySQL、SQL Server、PostgreSQL）、Oracle、FTP、ADS、OSS、OCS、DRDS，详细介绍请参见：数据同步简介，具体使用方法请参见：创建数据同步任务 。

**2. MaxCompute 客户端**

- 客户端的安装和基本使用请参见：https://help.aliyun.com/document_detail/27971.html ；
- 数据上传下载客户端基于 批量数据通道 的 SDK 实现了内置的 tunnel 命令，使用方法请参见：tunnel命令的基本使用介绍 。

备注：该项目已经开源，Github 项目地址为：https://github.com/aliyun/aliyun-odps-console 。

**3. DTS**

数据传输 (Data Transmission) 服务 DTS 是阿里云提供的一种支持 RDBMS (关系型数据库)、NoSQL、OLAP

等多种数据源之间数据交互的数据服务 。它提供了数据迁移、实时数据订阅及数据实时同步等多种数据传输功能 。

DTS 可以支持 RDS、MySQL 实例的数据实时同步到 MaxCompute 表中，暂不支持其他数据源类型。具体使用方法请参见：创建 RDS 到 MaxCompute 数据实时同步作业 。

## 开源产品：

### 1. Sqoop

Sqoop 基于社区 sqoop 1.4.6 版本开发，增强了对 MaxCompute 的支持，可以将数据从 Mysql 等关系数据库导入/导出到 MaxCompute 表中，也可以从 Hdfs/Hive 导入数据到 MaxCompute 表中。基本使用方法请参见：https://github.com/aliyun/aliyun-maxcompute-data-collectors/wiki/odps-sqoop 。

> 备注：该项目已经开源，Github项目地址为：https://github.com/aliyun/aliyun-maxcompute-data-collectors

### 2. Kettle

Kettle 是一款开源的 ETL 工具，纯 java 实现，可以在 Windows、Unix 和 Linux 上运行，提供图形化的操作界面，可以通过拖拽控件的方式，方便地定义数据传输的拓扑 。该工具的详细介绍和使用请参见：《基于 Kettle 的 MaxCompute 插件实现数据上云》。

> 备注：该项目已经开源，Github项目地址为：https://github.com/aliyun/aliyun-maxcompute-data-collectors

### 3. Flume

Apache Flume 是一个分布式的、可靠的、可用的系统，可高效地从不同的数据源中收集、聚合和移动海量日志数据到集中式数据存储系统，支持多种 Source 和 Sink 插件 。

Apache Flume 的 DataHub Sink 插件可以将日志数据实时上传到 Datahub，并归档到 MaxCompute 表中 。具体使用方法请参见： https://github.com/aliyun/aliyun-maxcompute-data-collectors/wiki/flume_plugin 。

> 备注：该项目已经开源，Github 项目地址为：https://github.com/aliyun/aliyun-maxcompute-data-collectors

### 4. Flunted

Fluentd 是一个开源的软件，用来收集各种源头日志（包括 Application Log、Sys Log 及 Access Log），允许用户选择插件对日志数据进行过滤,并存储到不同的数据处理端（包括 MySQL、Oracle、MongoDB、Hadoop、Treasure Data 等）。

Fluentd 的 DataHub 插件可以将日志数据实时上传到 DataHub，并归档到 MaxCompute 表中 。具体使用方法请参见：Flume 插件介绍 。

#### 5. LogStash

Logstash 是一款开源日志收集处理框架，logstash-output-datahub 插件实现了将数据导入 DataHub 的功能 。通过简单的配置即可完成数据的采集和传输，结合 MaxCompute/StreamCompute 可以轻松构建流式数据从采集到分析的一站式解决方案 。

LogStash 的 DataHub 插件可以将日志数据实时上传到 DataHub，并归档到 MaxCompute 表中 。具体使用案例请参见：Logstash + DataHub + MaxCompute/StreamCompute 进行实时数据分析 。

#### 6. OGG

OGG 的 DataHub 插件可以支持将 Oracle 数据库的数据实时地以增量方式同步到 DataHub 中，并最终归档到 MaxCompute 表中 。具体原理和使用方法请参见：《基于OGG Datahub插件将Oracle数据同步上云》 。

> 备注：该项目已经开源，Github 项目地址为：https://github.com/aliyun/aliyun-maxcompute-data-collectors

> 利用 MaxCompute 平台的 数据上传下载工具，可以广泛用于各种数据上云的应用场景，下面将对将对比较常用的几种经典场景进行介绍 。

## Hadoop 数据迁移

Hadoop 数据迁移有两种可选的工具，分别是 Sqoop 和大数据开发套件 。

- Sqoop 执行时会在原来的 Hadoop 集群上执行 MR 作业，可以分布式地将数据传输到 MaxCompute 上，效率会比较高，具体使用可以参考 Sqoop 工具的介绍 。
- 使用大数据开发套件结合 DataX 进行 Hadoop 数据迁移的示例可以参考云栖文章：《Hadoop 数据迁移新手教程》。

## 数据库数据同步

数据库数据同步到 MaxCompute 需要根据数据库的类型和同步策略来选择相应的工具：

- 离线批量的数据库数据同步，可以选择大数据开发套件，支持的数据库种类比较丰富，有 MySQL、SQL Server、PostgreSQL 等，详情请参见：数据同步简介，您也可以参考 创建数据同步任务 进行实例操作；
- Oracle 数据库数据实时同步，可以选择 OGG 插件工具；
- RDS 数据库数据实时同步，可以选择 DTS 同步 。

## 日志采集

日志采集可以选用 Flume、Flunted、LogStash 等工具 。具体场景示例可以参考云栖社区文章：《Flume 收集网站日志数据到 MaxCompute》 和 《海量日志数据分析与应用》。

# SQL

# 概要介绍

MaxCompute SQL适用于海量数据(TB级别)，实时性要求不高的场合，它的每个作业的准备，提交等阶段要花费较长时间，因此要求每秒处理几千至数万笔事务的业务是不能用 MaxCompute 完成的。

MaxCompute SQL采用的是类似于SQL的语法，可以看作是标准SQL的子集，但不能因此简单的把MaxCompute 等价成一个数据库，它在很多方面并不具备数据库的特征，如事务、主键约束、索引等。目前在 MaxCompute 中允许的最大SQL长度是2MB。

# 关键字

MaxCompute 将SQL语句的关键字作为保留字。在对表、列或是分区命名时如若使用关键字，需给关键字加``符号进行转义，否则会报错。保留字不区分大小写。下面只给出常用的保留字列表，完整的保留字列表请参阅MaxCompute SQL保留字。

```
  %   &   &&   (   )   *   +
- . / ; < <= <>
= > >= ? ADD ALL ALTER
AND AS ASC BETWEEN BIGINT BOOLEAN BY
CASE CAST COLUMN COMMENT CREATE DESC DISTINCT
DISTRIBUTE DOUBLE DROP ELSE FALSE FROM FULL
GROUP IF IN INSERT INTO IS JOIN
LEFT LIFECYCLE LIKE LIMIT MAPJOIN NOT NULL
ON OR ORDER OUTER OVERWRITE PARTITION RENAME
REPLACE RIGHT RLIKE SELECT SORT STRING TABLE
THEN TOUCH TRUE UNION VIEW WHEN WHERE
```

MaxCompute SQL允许数据类型之间的转换，类型转换方式包括：显式类型转换及隐式类型转换。

# 类型转换说明

## 显式类型转换

显式类型转换是用cast将一种数据类型的值转换为另一种类型的值的行为，在MaxCompute SQL中支持的显式类型转换如下：

| From/To | Bigint | Double | String | Datetime | Boolean | Decimal |
|---|---|---|---|---|---|---|
| Bigint | – | Y | Y | N | N | Y |
| Double | Y | – | Y | N | N | Y |
| String | Y | Y | – | Y | N | Y |
| Datetime | N | N | Y | – | N | N |
| Boolean | N | N | N | N | – | N |
| Decimal | Y | Y | Y | N | N | - |

其中，’ Y’ 表示可以转换，’ N’ 表示不可以转换，’ –’ 表示不需要转换。

比如：

```
select cast(user_id as double) as new_id from user;

select cast('2015-10-01 00:00:00' as datetime) as new_date from user;
```

备注:

- 将double类型转为bigint类型时，小数部分会被截断，例如：cast(1.6 as bigint) = 1；
- 满足double格式的string类型转换为bigint时，会先将string转换为double，再将double转换为bigint，因此，小数部分会被截断，例如cast(“1.6” as bigint) = 1；
- 满足bigint格式的string类型可以被转换为double类型，小数点后保留一位，例如：cast(“1” as double) = 1.0；
- 不支持的显式类型转换会导致异常；
- 如果在执行时转换失败，报错退出；
- 日期类型转换时采用默认格式yyyy-mm-dd hh:mi:ss，详细说明信息请参考String类型与Datetime类型之间的转换；
- 部分类型之间不可以通过显式的类型转换，但可以通过SQL内建函数进行转换，例如：从boolean类型转换到string类型，可使用函数to_char，详细介绍请参考 TO_CHAR ，而to_date函数同样支持从string类型到datetime类型的转换，详细介绍请参考 TO_DATE ；
- 关于cast的介绍请参阅 CAST ；
- DECIMAL超出值域，CAST STRING TO DECIMAL可能会出现最高位溢出报错，最低位溢出截断等情况。

# 隐式类型转换及其作用域

隐式类型转换是指在运行时，由 MaxCompute 依据上下文使用环境及类型转换规则自动进行的类型转换。MaxCompute 支持的隐式类型转换规则与显式转换相同：

| From/To | Bigint | Double | String | Datetime | Boolean | Decimal |
|---|---|---|---|---|---|---|
| Bigint | – | Y | Y | N | N | Y |

| | | | | | | |
|---|---|---|---|---|---|---|
| Double | Y | – | Y | N | N | Y |
| String | Y | Y | – | Y | N | Y |
| Datetime | N | N | Y | – | N | N |
| Boolean | N | N | N | N | – | N |
| Decimal | Y | Y | Y | N | N | - |

其中，’ Y’ 表示可以转换，’ N’ 表示不可以转换，’ –’ 表示不需要转换。

常见用法如下：

```
select user_id+age+'12345',
concat(user_name,user_id,age)
from user;
```

备注:

- 不支持的隐式类型转换会导致异常；
- 如果在执行时转换失败，也会导致异常；
- 由于隐式类型转换是 MaxCompute 依据上下文使用环境自动进行的类型转换，因此，我们推荐在类型不匹配时显式的用cast进行转换;
- 隐式类型转换规则是有发生作用域的。在某些作用域中，只有一部分规则可以生效。详细信息请参考隐式类型转换的作用域；

## 关系运算符

关系运算符包括：=, <>, <, <=, >, >=, IS NULL, IS NOT NULL, LIKE, RLIKE和IN。由于LIKE, RLIKE和IN的隐式类型转换规则不同于其他关系运算符，将单独拿出章节对这三种关系运算符做出说明。本小节的说明不包含这三种特殊的关系运算符。当不同类型的数据共同参与关系运算时，按照下述原则进行隐式类型转换。

| From/To | Bigint | Double | String | Datetime | Boolean | Decimal |
|---|---|---|---|---|---|---|
| Bigint | – | Double | Double | N | N | Decimal |
| Double | Double | – | Double | N | N | Decimal |
| String | Double | Double | – | Datetime | N | Decimal |
| Datetime | N | N | Datetime | – | N | N |
| Boolean | N | N | N | N | – | N |
| Decimal | Decimal | Decimal | Decimal | N | N | - |

备注:

- 如果待比较的两个类型间不能进行隐式类型转换，则该关系运算不能完成，报错退出；

- 关系运算符介绍，请参阅 关系操作符 ；

## 特殊的关系运算符(LIKE, RLIKE, IN)

LIKE及RLIKE的使用方式形如：

```
    source like pattern;
source rlike pattern;
```

此二者在隐式类型转换中的注意事项：

- LIKE和RLIKE的source和pattern参数均仅接受string类型；
- 其他类型不允许参与运算，也不能进行到string类型的隐式类型转换；IN的使用方式形如：

```
key in (value1, value2, ...)
```

In的隐式转换规则：

- In右侧的value值列表中的数据类型必须一致；
- 当key与values之间比较时，若bigint, double, string之间比较，统一转double，若datetime和string之间比较，统一转datetime。除此之外不允许其它类型之间的转换。

## 算术运算符

算术运算符包括：+, -, * , /, %, +, - ，其隐式转换规则：

只有string、bigint、double和Decimal才能参与算术运算。String在参与运算前会进行隐式类型转换到double。Bigint和double共同参与计算时，会将bigint隐式转换为double。日期型和布尔型不允许参与算数运算。

备注：

- 算术运算符的相关章节 算术操作符。

## 逻辑运算符

逻辑运算符包括：and, or和not，其隐式转换规则:

- 只有boolean才能参与逻辑运算。
- 其他类型不允许参与逻辑运算，也不允许其他类型的隐式类型转换。

备注：

- 逻辑运算符的相关章节 逻辑操作符。

# MaxCompute SQL内建函数

MaxCompute SQL提供了大量的系统函数，方便用户对任意行的一列或多列进行计算，输出任意种的数据类型。其隐式转换规则:

- 在调用函数时，如果输入参数的数据类型与函数定义的参数数据类型不一致，把输入参数的数据类型转换为函数定义的数据类型。
- 每个ODPS SQL内建函数的参数对于允许的隐式类型转换的要求不同，详见 内建函数 部分的说明。

# CASE WHEN

Case when的隐式转换规则:

- 如果返回类型只有bigint,double，统一转double；
- 如果返回类型中有string类型，统一转string，如果不能转则报错(如boolean类型)；
- 除此之外不允许其它类型之间的转换；

备注：

- Case when的详细介绍请参阅 [CASE WHEN表达式]

# 分区表

MaxCompute SQL支持分区表。指定分区表会对用户带来诸多便利，例如：提高SQL运行效率，减少计费等。在如下场景下使用分区表将会带来较大的收益：

- 在Select语句的Where条件过滤中使用分区列作为过滤条件；

```
create table src (key string, value bigint) partitioned by (pt string);  -- 目前，MaxCompute 仅承诺String类型分区
select * from src where pt='20151201'; -- 正确使用方式。MaxCompute 在生成查询计划时只会将'20151201'分区的数据纳入输入中
select * from src where pt = 20151201; -- 错误的使用方式。在这样的使用方式下，MaxCompute并不能保障分区过滤机制的有效性。pt是String类型，当String类型与Bigint(20151201)比较时，MaxCompute会将二者转换为Double类型，此时有可能会有精度损失。
```

与此同时，部分对分区操作的SQL的运行效率则较低，给您带来较高的计费，例如：

- 使用动态分区

对于部分 MaxCompute 操作命令，处理分区表和非分区表时的语法有差别，详细情况请参考DDL语句 及 DML语句 部分的说明 。目前，ODPS分区仅支持string类型，不承诺其他分区类型的正确性，不支持其他任意类型的隐式类型转换。

# UNION ALL

参与 UNION ALL 运算的所有列的数据类型、列个数、列名称必须完全一致，否则抛异常。

# String类型与Datetime类型之间的转换

MaxCompute 支持string类型和datetime类型之间的相互转换。转换时使用的格式为yyyy-mm-dd hh:mi:ss，其中：

| 单位 | 字符串(忽略大小写) | 有效值域 |
| --- | --- | --- |
| 年 | yyyy | 0001 ~ 9999 |
| 月 | mm | 01 ~ 12 |
| 日 | dd | 01 ~ 28,29,30,31 |
| 时 | hh | 00 ~ 23 |
| 分 | mi | 00 ~ 59 |
| 秒 | ss | 00 ~ 59 |

备注：

- 各个单位的值域中，如果首位为0，不可省略，例如：”2014-1-9 12:12:12”就是非法的datetime格式，无法从这个string类型数据转换为datetime类型，必须写为”2014-01-09 12:12:12”。
- 只有符合上述格式描述的string类型才能够转换为datetime类型，例如：cast(“2013-12-31 02:34:34” as datetime)，将会把string类型”2013-12-31 02:34:34”转换为datetime类型。同理，datetime转换为string时，默认转换为yyyy-mm-dd hh:mi:ss的格式。

类似于下面的转换尝试，将会失败导致异常，例如：

```
cast("2013/12/31 02/34/34" as datetime)
cast("20131231023434" as datetime)
cast("2013-12-31 2:34:34" as datetime)
```

值得注意的是，” dd”部分的阈值上限取决于月份实际拥有的天数，如果超出对应月份实际拥有的天数，将会导致异常退出，例如：

```
cast("2013-02-29 12:12:12" as datetime)     -- 异常返回，2013年2月没有29日
cast("2013-11-31 12:12:12" as datetime) -- 异常返回，2013年11月没有31日
```

MaxCompute 提供了to_date函数，用以将不满足日期格式的string类型数据转换为datetime类型。详细信息请参阅 TO_DATE 。

| 操作符 | 说明 |
| --- | --- |
| A=B | 如果A或B为NULL，返回NULL；如果A等于B，返回TRUE，否则返回FALSE |
| A<>B | 如果A或B为NULL，返回NULL；如果A不等于B，返回TRUE，否则返回FALSE |
| A<B | 如果A或B为NULL，返回NULL；如果A小于B，返回TRUE，否则返回FALSE |
| A<=B | 如果A或B为NULL，返回NULL；如果A小于等于B，返回TRUE，否则返回FALSE |
| A>B | 如果A或B为NULL，返回NULL；如果A大于B，返回TRUE，否则返回FALSE |
| A>=B | 如果A或B为NULL，返回NULL；如果A大于等于B，返回TRUE，否则返回FALSE |
| A IS NULL | 如果A为NULL，返回TRUE，否则返回FALSE |
| A IS NOT NULL | 如果A不为NULL，返回TRUE，否则返回FALSE |
| A LIKE B | 如果A或B为NULL，返回NULL，A为字符串，B为要匹配的模式， 如果匹配，返回TRUE，否则返回FALSE。'%'匹配任意多个字符，'_'匹配单个字符。要匹配'%'或’ _’ 需要用转义符表示’ \%’ ，’ _’ 。<br>- ‘aaa’ like ‘a__’ = TRUE ‘aaa’ like ‘a%’ = TRUE ‘aaa’ like ‘aab’ = FALSE ‘a%b’ like ‘a\%b’ = TRUE ‘axb’ like ‘a\%b’ = FALSE |
| A RLIKE B | A是字符串，B是字符串常量正则表达式； 如果匹配成功，返回TRUE，否则返回FALSE； 如果B为空串会报错退出；如果A或B为NULL，返回NULL； |
| A IN B | B是一个集合，如果A为NULL，返回NULL，如A在B中则返回TRUE，否则返回FALSE 若B仅有一个元素NULL，即A IN (NULL)，则返回NULL。 若B含有NULL元素，将NULL视为B集合中其他元素的类型。 B必须是常数并且至少有一项，所有类型要一致 |

常见用法如下：

```
select * from user where user_id = '0001';
select * from user where user_name <> 'maggie';
select * from user where age > ‘50’;
select * from user where birth_day >= '1980-01-01 00:00:00';
```

```
select * from user where is_female is null;
select * from user where is_female is not null;
select * from user where user_id in (0001,0010);
select * from user where user_name like 'M%';
```

由于double值存在一定的精度差，因此，我们不建议直接使用等号=对两个double类型数据进行比较。用户可以使用两个double类型相减，而后取绝对值的方式判断。当绝对值足够小时，认为两个double数值相等，例如:

```
abs(0.9999999999 - 1.0000000000) < 0.000000001
-- 0.9999999999和1.0000000000为10位精度，而0.000000001为9位精度。
-- 此时可以认为0.9999999999和1.0000000000相等。
```

备注：

- Abs是ODPS提供的内建函数，意为取绝对值，详细可参考 ABS 。
- 通常情况下，ODPS的double类型能够保障14位有效数字。

# 算术操作符

| 操作符 | 说明 |
| --- | --- |
| A + B | 如果A或B为NULL，返回NULL；否则返回A + B的结果。 |
| A – B | 如果A或B为NULL，返回NULL；否则返回A – B的结果。 |
| A * B | 如果A或B为NULL，返回NULL；否则返回A * B的结果。 |
| A / B | 如果A或B为NULL，返回NULL；否则返回A / B的结果。注：如果A和B为bigint类型，返回结果为double类型。 |
| A % B | 如果A或B为NULL，返回NULL；否则返回A模B的结果。 |
| +A | 仍然返回A。 |
| -A | 如果A为NULL，返回NULL，否则返回-A。 |

常见用法如下：

```
select age+10, age-10, age%10, -age, age*age, age/10
 from user;
```

备注

- 只有string，bigint，double才能参与算术运算，日期型和布尔型不允许参与运算。
- String类型在参与运算前会进行隐式类型转换到double类型。
- bigint和double共同参与计算时，会将bigint隐式转换为double再进行计算，返回结果为double类型。
- A和B都是bigint类型，进行A/B运算,返回结果为double类型；进行上述其他运算仍然返回bigint类型。

# 位操作符

| 操作符 | 说明 |
| --- | --- |
| A & B | 返回A与B进行按位与的结果。例如：1&2返回0，1&3返回1，NULL与任何值按位与都为NULL。 A和B必须为Bigint类型。 |
| A \| B | 返回A与B进行按位或的结果。例如：1 \|2返回3，1 \|3返回3，NULL与任何值按位或都为NULL。A和B 必须为Bigint类型。 |

备注：

- 位运算符不支持隐式转换，只允许bigint类型。

# 逻辑操作符

```
操作符          说明
A and B        TRUE and TRUE=TRUE
               TRUE and FALSE=FALSE
               FALSE and TRUE=FALSE
               FALSE and NULL=FALSE
               NULL and FALSE=FALSE
               TRUE and NULL=NULL
               NULL and TRUE=NULL
               NULL and NULL=NULL

A or B         TRUE or TRUE=TRUE
               TRUE or FALSE=TRUE
               FALSE or TRUE=TRUE
               FALSE or NULL=NULL
               NULL or FALSE=NULL
               TRUE or NULL=TRUE
               NULL or TRUE=TRUE
               NULL or NULL=NULL

NOT A          如果A是NULL，返回NULL
```

```
如果A是TRUE，返回FALSE
如果A是FALSE，返回TRUE
```

备注

- 逻辑操作符只允许boolean类型参与运算，不支持隐式类型转换。

# 表操作

## 创建表

语法格式

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] table_name
[(col_name data_type [COMMENT col_comment], ...)]
[COMMENT table_comment]
[PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
[LIFECYCLE days]
[AS select_statement]
[STORED BY StorageHandler] -- 仅限外部表
[WITH SERDEPROPERTIES (Options)] -- 仅限外部表
[LOCATION OSSLocation];-- 仅限外部表


CREATE TABLE [IF NOT EXISTS] table_name
LIKE existing_table_name
```

说明：

- 表名与列名均对大小写不敏感；
- 在创建表时，如果不指定 if not exists 选项而存在同名表，则返回出错；若指定此选项，则无论是否存在同名表，即使原表结构与要创建的目标表结构不一致，均返回成功 。已存在的同名表的元信息不会被改动；
- 数据类型只能是：bigint，double，boolean，datetime，decimal及string；
- 表名，列名中不能有特殊字符，只能用英文的 a-z，A-Z 及数字和下划线_，且以字母开头，名称的长度不超过 128 字节；
- Partitioned by 指定表的分区字段，目前仅支持 string 类型 。分区值不允许有双字节字符(如中文)，必须是以英文字母 a-z，A-Z 开始后可跟字母数字，名称的长度不超过 128 字节 。 允许的字符包括：空格 ‘ ’，冒号’:’ ，下划线’ _’ ，美元符’ $’ ，井号’ #’ ，点’ .’ ，感叹号’ !’ 和’ @’ ，出现其他字符行为未定义，例如：” \t” ，” \n” ，” /” 等 。当利用分区字段对表进行分区时，新增分区、更新分区内数据和读取分区数据均不需要做全表扫描，可以提高处理效率；
- 注释内容是长度不超过 1024 字节的有效字符串；
- lifecycle 指明此表的生命周期，create table like 语句不会复制源表的生命周期属性；

- 目前，在表中建的分区层次不能超过 6 级；
- 更多有关外部表的说明请参考云栖社区文章之 MaxCompute上如何处理非结构化数据 。

在下面的例子中，创建表 sale_detail 保存销售记录，该表使用销售时间 (sale_date) 和销售区域 (region) 作为分区列：

```
create table if not exists sale_detail(
shop_name string,
customer_id string,
total_price double)
partitioned by (sale_date string,region string);
-- 创建一张分区表 sale_detail
```

也可以通过 create table ... as select ..语句创建表，并在建表的同时将数据复制到新表中，如：

```
create table sale_detail_ctas1 as
select * from sale_detail;
```

此时，如果 sale_detail 中存在数据，上面的示例会将 sale_detail 的数据全部复制到 sale_detail_ctas1 表中。但请注意，此处 sale_detail 是一张分区表，而通过 create table ... as select ... 语句创建的表不会复制分区属性，只会把源表的分区列作为目标表的一般列处理，即 sale_detail_ctas1 是一个含有 5 列的非分区表 。

在 create table ... as select ...语句中，如果在 select 子句中使用常量作为列的值，建议指定列的名字，例如：

```
create table sale_detail_ctas2 as
select shop_name,
customer_id,
total_price,
'2013' as sale_date,
'China' as region
from sale_detail;
```

如果不加列的别名，如：

```
create table sale_detail_ctas3 as
select shop_name,
customer_id,
total_price,
'2013',
'China'
from sale_detail;
```

则创建的表 sale_detail_ctas3 的第四、五列会是类似_c5，_c6。 如果希望源表和目标表具有相同的表结构，可以尝试使用create table ... like操作，如：

```
create table sale_detail_like like sale_detail;
```

此时，sale_detail_like的表结构与sale_detail完全相同 。除生命周期属性外，列名、列注释以及表注释等均相同 。但sale_detail中的数据不会被复制到sale_detail_like表中 。

# 删除表

语法格式

```
DROP TABLE [IF EXISTS] table_name;
```

说明：

- 如果不指定 if exists 选项而表不存在，则返回异常；若指定此选项，无论表是否存在，皆返回成功；
- 删除外部表时，OSS 上的数据不会被删除 。

示例：

```
create table sale_detail_drop like sale_detail;
drop table sale_detail_drop;
--若表存在，成功返回；若不存在，异常返回；
drop table if exists sale_detail_drop2;
--无论是否存在 sale_detail_drop2 表，均成功返回 。
```

# 重命名表

语法格式

```
ALTER TABLE table_name RENAME TO new_table_name;
```

说明：

- rename 操作仅修改表的名字，不改动表中的数据；
- 如果已存在与 new_table_name 同名表，报错；
- 如果 table_name 不存在，报错 。

示例：

```
create table sale_detail_rename1 like sale_detail;
alter table sale_detail_rename1 rename to sale_detail_rename2;
```

# 修改表的注释

命令格式：

```
ALTER TABLE table_name SET COMMENT 'tbl comment';
```

说明：

- table_name 必须是已存在的表；comment 最长 1024 字节；

示例：

```
alter table sale_detail set comment 'new coments for table sale_detail';
```

通过 MaxCompute 的desc命令可以查看表中 comment 的修改，请参阅 查看表信息 。

# 修改表的生命周期属性

MaxCompute 提供数据生命周期管理功能，方便用户释放存储空间，简化回收数据的流程 。

语法格式：

```
ALTER TABLE table_name SET lifecycle days;
```

说明：

- days 参数为生命周期时间，只接受正整数。单位：天；

如果表 table_name 是非分区表，自最后一次数据被修改开始计算，经过days天后数据仍未被改动，则此表无需用户干预，将会被 MaxCompute 自动回收(类似drop table操作)。

在 MaxCompute 中，每当表的数据被修改后，表的 LastDataModifiedTime 将会被更新，因此，MaxCompute 会根据每张表的 LastDataModifiedTime 以及 lifecycle 的设置来判断是否要回收此表。

如果 table_name 是分区表，则根据各分区的 LastDataModifiedTime 判断该分区是否该被回收 。关于 LastDataModifiedTime 的介绍请参考 查看表信息 。

不同于非分区表，分区表的最后一个分区被回收后，该表不会被删除。生命周期只能设定到表级别，不能再分区级设置生命周期。创建表时即可指定生命周期。

示例：

```
create table test_lifecycle(key string) lifecycle 100;
-- 新建test_lifecycle表，生命周期为100天。

alter table test_lifecycle set lifecycle 50;
-- 修改test_lifecycle表，将生命周期设为50天。
```

# 修改表的修改时间

MaxCompute SQL 提供touch操作用来修改表的LastDataModifiedTime。效果会将表的LastDataModifiedTime修改为当前时间 。

语法格式：

```
ALTER TABLE table_name TOUCH;
```

说明：

- table_name 不存在，则报错返回；；
- 此操作会改变表的”LastDataModifiedTime”的值，此时，MaxCompute 会认为表的数据有变动，生命周期的计算会重新开始 。

## 清空非分区表里的数据

将指定的非分区表中的数据清空， 该命令不支持分区表。对于分区表，可以用ALTER TABLE table_name DROP PARTITION的方式将分区里的数据清除 。

语法格式:

```
TRUNCATE TABLE table_name;
```

# 视图操作

## 创建视图

语法格式

```
CREATE [OR REPLACE] VIEW [IF NOT EXISTS] view_name
[(col_name [COMMENT col_comment], ...)]
[COMMENT view_comment]
[AS select_statement]
```

说明：

- 创建视图时，必须有对视图所引用表的读权限；
- 视图只能包含一个有效的 select 语句；
- 视图可以引用其它视图，但不能引用自己，也不能循环引用；
- 不允许向视图写入数据，例如使用 insert into 或者 insert overwrite 操作视图；
- 当视图建好以后，如果视图的引用表发生了变更，有可能导致视图无法访问，例如：删除被引用表。用户需要自己维护引用表及视图之间的对应关系；
- 如果没有指定 if not exists，在视图已经存在时用create view会导致异常。这种情况可以用 create

or replace view 来重建视图，重建后视图本身的权限保持不变 。

示例：

```
create view if not exists sale_detail_view
(store_name, customer_id, price, sale_date, region)
comment 'a view for table sale_detail'
as select * from sale_detail;
```

# 删除视图

语法格式

```
DROP VIEW [IF EXISTS] view_name;
```

说明：

- 如果视图不存在且没有指定 if exists，报错 。

示例：

```
DROP VIEW IF EXISTS sale_detail_view;
```

# 重命名视图

语法格式

```
ALTER VIEW view_name RENAME TO new_view_name;
```

说明：

- 如果已存在同名视图，报错 。

示例：

```
create view if not exists sale_detail_view
(store_name, customer_id, price, sale_date, region)
comment 'a view for table sale_detail'
as select * from sale_detail;

alter view sale_detail_view rename to market;
```

# 分区/列操作

# 添加分区

语法格式:

```
ALTER TABLE TABLE_NAME ADD [IF NOT EXISTS] PARTITION partition_spec

partition_spec:
: (partition_col1 = partition_col_value1, partition_col2 = partiton_col_value2, ...)
```

说明：

- 仅支持新增分区，不支持新增分区字段；
- 如果未指定 if not exists 而同名的分区已存在，则出错返回；
- 目前 MaxCompute 单表支持的分区数量上限为 6 万；
- 对于多级分区的表，如果想添加新的分区，必须指明全部的分区值 。

为 sale_detail 表添加一个新的分区示例：

```
alter table sale_detail add if not exists partition (sale_date='201312', region='hangzhou');
-- 成功添加分区，用来存储2013年12月杭州地区的销售记录。

alter table sale_detail add if not exists partition (sale_date='201312', region='shanghai');
-- 成功添加分区，用来存储2013年12月上海地区的销售记录。

alter table sale_detail add if not exists partition(sale_date='20111011');
-- 仅指定一个分区sale_date，出错返回

alter table sale_detail add if not exists partition(region='shanghai');
-- 仅指定一个分区region，出错返回
```

# 删除分区

```
ALTER TABLE TABLE_NAME DROP [IF EXISTS] PARTITION partition_spec;

partition_spec:
: (partition_col1 = partition_col_value1, partition_col2 = partiton_col_value2, ...)
```

说明：

- 如果分区不存在且未指定 if exists，则出错返回。

从表 sale_detail 中删除一个分区示例：

```
alter table sale_detail drop if exists partition(sale_date='201312',region='hangzhou');
-- 成功删除 2013 年 12 月杭州分区的销售 。
```

# 添加列

命令格式：

```
ALTER TABLE table_name ADD COLUMNS (col_name1 type1, col_name2 type2...)
```

说明：

- 列的数据类型只能是：bigint，double，boolean，datetime，decimal及string 类型 。

# 修改列名

命令格式：

```
ALTER TABLE table_name CHANGE COLUMN old_col_name RENAME TO new_col_name;
```

说明：

- old_col_name 必须是已存在的列；
- 表中不能有名为 new_col_name 的列 。

# 修改列、分区注释

命令格式：

```
ALTER TABLE table_name CHANGE COLUMN col_name COMMENT comment_string;
```

说明：

- COMMENT 内容最长 1024 字节；

# 同时修改列名及列注释

命令格式：

```
ALTER TABLE table_name CHANGE COLUMN old_col_name new_col_name column_type COMMENT
column_comment;
```

说明：

- old_col_name 必须是已存在的列；
- 表中不能有名为 new_col_name 的列；
- COMMENT 内容最长 1024 字节 。

# 修改表、分区的修改时间

MaxCompute SQL 提供touch操作用来修改分区的LastDataModifiedTime。效果会将分区的LastDataModifiedTime修改为当前时间 。

语法格式：

```
ALTER TABLE table_name TOUCH PARTITION(partition_col='partition_col_value', ...);
```

说明：

- table_name 或 partition_col 不存在，则报错返回；
- 指定的 partition_col_value 不存在，则报错返回；
- 此操作会改变表的”LastDataModifiedTime”的值，此时， MaxCompute 会认为表或分区的数据有变动，生命周期的计算会重新开始 。

# 修改分区值

MaxCompute SQL 支持通过 rename 操作更改对应表的分区值 。

语法格式：

```
ALTER TABLE table_name PARTITION (partition_col1 = partition_col_value1, partition_col2 = partiton_col_value2, ...)
RENAME TO PARTITION (partition_col1 = partition_col_newvalue1, partition_col2 = partiton_col_newvalue2, ...);
```

说明：

- 不支持修改分区列列名,只能修改分区列对应的值；
- 修改多级分区的一个或者多个分区值，多级分区的每一级的分区值都必须写上 。

# 更新表中的数据(INSERT OVERWRITE/INTO)

语法格式：

```
INSERT OVERWRITE|INTO TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)]
select_statement
FROM from_statement;
```

备注:

- MaxCompute 的Insert语法与通常使用的MySQL或Oracle的Insert语法有差别，在insert

overwrite|into 后需要加入table关键字，后不是直接使用tablename。

在MaxCompute SQL处理数据的过程中，insert overwrite/into用于将计算的结果保存目标表中。insert into与insert overwrite的区别是，insert into会向表或表的分区中追加数据，而insert overwrite则会在向表或分区中插入数据前清空表中的原有数据。在使用MaxCompute 处理数据的过程中，insert overwrite/into是最常用到的语句，它们会将计算的结果保存一个表中，可以供下一步计算使用。如，可以用如下操作计算sale_detail表中不同地区的销售额

```
    create table sale_detail_insert like sale_detail;
alter table sale_detail_insert add partition(sale_date='2013', region='china');
insert overwrite table sale_detail_insert partition (sale_date='2013', region='china')
select shop_name, customer_id, total_price from sale_detail;
```

需要注意的是，在进行insert更新数据操作时，源表与目标表的对应关系依赖于在select子句中列的顺序，而不是表与表之间列名的对应关系，下面的SQL语句仍然是合法的：

```
    insert overwrite table sale_detail_insert partition (sale_date='2013', region='china')
select customer_id, shop_name, total_price from sale_detail;
-- 在创建sale_detail_insert表时，列的顺序为：
-- shop_name string, customer_id string, total_price bigint
-- 而从sale_detail向sale_detail_insert插入数据是，sale_detail的插入顺序为：
-- customer_id, shop_name, total_price
-- 此时，会将sale_detail.customer_id的数据插入sale_detail_insert.shop_name
-- 将sale_detail.shop_name的数据插入sale_detail_insert.customer_id
```

向某个分区插入数据时，分区列不允许出现在select列表中：

```
    insert overwrite table sale_detail_insert partition (sale_date='2013', region='china')
select shop_name, customer_id, total_price, sale_date, region from sale_detail;
-- 报错返回，sale_date, region为分区列，不允许出现在静态分区的 insert 语句中。
```

同时，partition的值只能是常量，不可以出现表达式。以下用法是非法的：

```
    insert overwrite table sale_detail_insert partition (sale_date=datepart('2016-09-18 01:10:00', 'yyyy') ,
region='china')
select shop_name, customer_id, total_price from sale_detail;
```

# 多路输出(MULTI INSERT)

MaxCompute SQL支持在一个语句中插入不同的结果表或者分区

语法格式：

```
    FROM from_statement
INSERT OVERWRITE | INTO TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)]
select_statement1 [FROM from_statement]
```

```
[INSERT OVERWRITE | INTO TABLE tablename2 [PARTITION (partcol1=val3, partcol2=val4 ...)]
select_statement2 [FROM from_statement]]
```

说明：

- 一般情况下，单个SQL里最多可以写128路输出，超过128路报语法错误。
- 在一个multi insert中，对于分区表，同一个目标分区不允许出现多次; 对于未分区表，该表不能出现多次。
- 对于同一张分区表的不同分区，不能同时有insert overwrite和insert into操作，否则报错返回。

如，

```
create table sale_detail_multi like sale_detail;

from sale_detail
insert overwrite table sale_detail_multi partition (sale_date='2010', region='china' )
select shop_name, customer_id, total_price where .....
insert overwrite table sale_detail_multi partition (sale_date='2011', region='china' )
select shop_name, customer_id, total_price where .....;
-- 成功返回，将sale_detail的数据插入到sales里的2010年及2011年中国大区的销售记录中

from sale_detail
insert overwrite table sale_detail_multi partition (sale_date='2010', region='china' )
select shop_name, customer_id, total_price
insert overwrite table sale_detail_multi partition (sale_date='2010', region='china' )
select shop_name, customer_id, total_price;
-- 出错返回，同一分区出现多次

from sale_detail
insert overwrite table sale_detail_multi partition (sale_date='2010', region='china' )
select shop_name, customer_id, total_price
insert into table sale_detail_multi partition (sale_date='2011', region='china' )
select shop_name, customer_id, total_price;
-- 出错返回，同一张表的不同分区，不能同时有insert overwrite和insert into操作
```

# 输出到动态分区(DYNAMIC PARTITION)

在insert overwrite到一张分区表时，可以在语句中指定分区的值。也可以用另外一种更加灵活的方式，在分区中指定一个分区列名，但不给出值。相应的，在select子句中的对应列来提供分区的值。

语法格式：

```
insert overwrite table tablename partition (partcol1, partcol2 ...) select_statement from from_statement;
```

**说明：**

- select_statement字段中，后面的字段将提供目标表动态分区值。如目标表就一级动态分区，则select_statement最后一个字段值即为目标表的动态分区值。
- 目前，在使用动态分区功能的SQL中，在分布式环境下，单个进程最多只能输出512个动态分区，否

则引发运行时异常；
- 在现阶段，任意动态分区SQL不允许生成超过2000个动态分区，否则引发运行时异常；
- 动态生成的分区值不允许为NULL，也不支持含有特殊字符和中文，否则会引发异常,如：“FAILED: ODPS-0123031:Partition exception - invalid dynamic partition value: province=xxx”；
- 如果目标表有多级分区，在运行insert语句时允许指定部分分区为静态，但是静态分区必须是高级分区；

下面，我们使用一个简单的例子来说明动态分区：

```
    create table total_revenues (revenue bigint) partitioned by (region string);
insert overwrite table total_revenues partition(region)
select total_price as revenue, region
from sale_detail;
```

按照这种写法，在SQL运行之前，是不知道会产生哪些分区的，只有在select运行结束后，才能由region字段产生的值确定会产生哪些分区，这也是为什么叫做”动态分区”的原因。

其他示例：

```
create table sale_detail_dypart like sale_detail;--创建示例目标表


--示例一：
insert overwrite table sale_detail_dypart partition (sale_date, region)
select shop_name,customer_id,total_price,sale_date,region from sale_detail;
-- 成功返回;
```

- 此时sale_detail表中，sale_date的值决定目标表的sale_date分区值，region的值决定目标表的region分区值；
- **动态分区中，select_statement字段和目标表动态分区的对应是按字段顺序决定。**如该示例中，select语句若写成select shop_name,customer_id,total_price,region,sale_date from sale_detail;则，sale_detail表中，region值决定决定目标表的sale_date分区值；sale_date的值决定目标表的region分区值。

```
--示例二：
insert overwrite table sale_detail_dypart partition (sale_date='2013', region)
select shop_name,customer_id,total_price,region from sale_detail;
-- 成功返回，多级分区，指定一级分区


--示例三：
insert overwrite table sale_detail_dypart partition (sale_date='2013', region)
select shop_name,customer_id,total_price from sale_detail;
-- 失败返回，动态分区插入时，动态分区列必须在select列表中


--示例四：
```

```
insert overwrite table sales partition (region='china', sale_date)
select shop_name,customer_id,total_price,region from sale_detail;
-- 失败返回，不能仅指定低级子分区，而动态插入高级分区
```

# SELECT语法介绍

语法格式：

```
    SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[ORDER BY order_condition]
[DISTRIBUTE BY distribute_condition [SORT BY sort_condition] ]
[LIMIT number]
```

在使用select语句时需要注意如下几点：

- select操作从表中读取数据，要读的列可以用列名指定，或者用*代表所有的列，一个简单的select如下:

```
    select * from sale_detail;
```

或者只读取sale_detail的一列shop_name

```
    select shop_name from sale_detail;
```

在where中可以指定过滤的条件，如

```
    select * from sale_detail where shop_name like 'hang%';
```

备注:

- 请注意，当使用Select语句屏显时，目前最多只能显示1000行结果。当select作为子句时，无此限制，select子句会将全部结果返回给上层查询。

- where子句：支持的过滤条件包括：

| 过滤条件 | 描述 |
| --- | --- |
| >, < , =, >=, <=, <> | 关系操作符 |
| like, rlike | like和rlike的source和pattern参数均仅接受string类型。 |
| in, not in | 如果在in/not in条件后加子查询，子查询只能返回 |

| | |
|---|---|
| | 一列值，且返回值的数量不能超过1000。 |

在select语句的where子句中可以指定分区范围，这样可以仅仅扫描表的指定部分，避免全表扫描。如下所示：

```
    select sale_detail.*
from sale_detail
where sale_detail.sale_date >= '2008' and sale_detail.sale_date <= '2014';
```

MaxCompute SQL的where子句不支持between条件查询。

- 在table_reference中支持使用嵌套子查询，如：

```
    select * from (select region from sale_detail) t where region = 'shanghai';
```

- distinct：如果有重复数据行时，在字段前使用distinct，会将重复字段去重，只返回一个值，而使用all将返回字段中所有重复的值，不指定此选项时默认效果和all相同。使用distinct只返回一行记录，如

```
    select distinct region from sale_detail;
select distinct region, sale_date from sale_detail;
-- distinct多列，distinct的作用域是select的列集合，不是单个列。
```

- group by：分组查询, 一般group by是和聚合函数配合使用。在select中包含聚合函数时：

1. 用group by的key可以是输入表的列名;
2. 也可以是由输入表的列构成的表达式，不允许是select语句的输出列的别名;
3. 规则1的优先级高于规则2。当规则1和规则2发生冲突时，即group by的key即是输入表的列或表达式，又是select的输出列，以规则1为准。

如果如

```
    select region from sale_detail group by region;
-- 直接使用输入表列名作为group by的列，可以运行

select sum(total_price) from sale_detail group by region;
-- 以region值分组，返回每一组的销售额总量，可以运行

select region, sum(total_price) from sale_detail group by region;
-- 以region值分组，返回每一组的region值(组内唯一)及销售额总量，可以运行



select region as r from sale_detail group by r;
-- 使用select列的别名运行，报错返回

select 'China-' + region as r from sale_detail group by 'China-' + region;
-- 必须使用列的完整表达式
```

```
select region, total_price from sale_detail group by region;
-- 报错返回，select的所有列中，没有使用聚合函数的列，必须出现在group by中

select region, total_price from sale_detail group by region, total_price;
-- 可以运行
```

有这样的限制是因为，在SQL解析中，group by操作通常是先于select操作的，因此group by只能接受输入表的列或表达式为key。

备注:

- 关于聚合函数的介绍请参考 **聚合函数**

- order by：对所有数据按照某几列进行全局排序。如果您希望按照降序对记录进行排序，可以使用DESC关键字。由于是全局排序，**order by必须与limit共同使用。**对在使用order by排序时，NULL会被认为比任何值都小，这个行为与Mysql一致，但是与Oracle不一致。与group by不同，order by后面必须加select列的别名，当select某列时，如果没有指定列的别名，将列名作为列的别名。

```
select * from sale_detail order by region;
-- 报错返回，order by没有与limit共同使用

select * from sale_detail order by region limit 100;

select region as r from sale_detail order by region limit 100;
-- 报错返回，order by后面必须加列的别名。

select region as r from sale_detail order by r limit 100;
```

[limit number]的number是常数，限制输出行数。当使用无limit的select语句直接从屏幕输出查看结果时，最多只输出5000行。每个项目空间的这个屏显最大限制限制可能不同，可以通过控制台面板控制。

distribute by：对数据按照某几列的值做hash分片，必须使用select的输出列别名。

```
select region from sale_detail distribute by region;
-- 列名即是别名，可以运行

select region as r from sale_detail distribute by region;
-- 报错返回，后面必须加列的别名。

select region as r from sale_detail distribute by r;
```

- sort by：局部排序，语句前必须加distribute by。实际上sort by是对distribute by的结果进行局部排序。必须使用select的输出列别名。

```
    select region from sale_detail distribute by region sort by region;
select region as r from sale_detail sort by region;
-- 没有distribute by，报错退出。
```

- order by不和distribute by/sort by共用，同时group by也不和distribute by/sort by共用，必须使用select的输出列别名。

备注:

- order by/sort by/distribute by的key必须是select语句的输出列，即列的别名。在MaxCompute SQL解析中， order by/sort by/distribute by是后于select操作的，因此它们只能接受select语句的输出列为key。

# 子查询

普通的select是从几张表中读数据，如select column_1, column_2 ... from table_name，但查询的对象也可以是另外一个select操作，如：

```
    select * from (select shop_name from sale_detail) a;
```

备注：

- 子查询必须要有别名。

在from子句中，子查询可以当作一张表来使用，与其它的表或子查询进行join操作，如

```
    create table shop as select * from sale_detail;

select a.shop_name, a.customer_id, a.total_price from
(select * from shop) a join sale_detail on a.shop_name = sale_detail.shop_name;
```

# UNION ALL

语法格式：

```
    select_statement UNION ALL select_statement
```

将两个或多个select操作返回的数据集联合成一个数据集，如果结果有重复行时，会返回所有符合条件的行，不进行重复行的去重处理。需要注意的是：MaxCompute SQL不支持顶级的两个查询结果合并，要改写为一个子查询的形式．如

```
    select * from sale_detail where region = 'hangzhou'
union all
select * from sale_detail where region = 'shanghai';
```

需要改成：

```
    select * from (
select * from sale_detail where region = 'hangzhou'
union all
select * from sale_detail where region = 'shanghai')
t;
```

备注:

- union all操作对应的各个子查询的列个数、名称和类型必须一致。如果列名不一致时，可以使用列的别名加以解决。
- 一般情况下，MaxCompute 最多允许128路union all，超过此限制报语法错误。

# JOIN操作

MaxCompute 的JOIN支持多路间接，但不支持笛卡尔积，即无on条件的链接。语法定义：

```
    join_table:
table_reference join table_factor [join_condition]
| table_reference {left outer|right outer|full outer|inner} join table_reference join_condition

table_reference:
table_factor
| join_table

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

join_condition:
on equality_expression ( and equality_expression )*
```

备注:

- equality_expression是一个等式表达式

left join 会从左表(shop)那里返回所有的记录，即使在右表(sale_detail)中没有匹配的行。

```
    select a.shop_name as ashop, b.shop_name as bshop from shop a
```

```
left outer join sale_detail b on a.shop_name=b.shop_name;
-- 由于表shop及sale_detail中都有shop_name列，因此需要在select子句中使用别名进行区分。
```

right outer join 右连接，返回右表中的所有记录，即使在左表中没有记录与它匹配，例如：

```
    select a.shop_name as ashop, b.shop_name as bshop from shop a
right outer join sale_detail b on a.shop_name=b.shop_name;
```

full outer join 全连接，返回左右表中的所有记录，例如：

```
    select a.shop_name as ashop, b.shop_name as bshop from shop a
full outer join sale_detail b on a.shop_name=b.shop_name;
```

在表中存在至少一个匹配时，inner join 返回行。 关键字inner可省略。

```
    select a.shop_name from shop a inner join sale_detail b on a.shop_name=b.shop_name;

select a.shop_name from shop a join sale_detail b on a.shop_name=b.shop_name;
```

连接条件，只允许and连接的等值条件，并且最多支持16路join操作。只有在MAPJOIN中，可以使用不等值连接或者使用or连接多个条件。

```
    select a.* from shop a full outer join sale_detail b on a.shop_name=b.shop_name
full outer join sale_detail c on a.shop_name=c.shop_name;
-- 支持多路join链接示例，最多支持16路join

select a.* from shop a join sale_detail b on a.shop_name != b.shop_name;
-- 不支持不等值Join链接条件，报错返回。
```

# MAPJOIN HINT

当一个大表和一个或多个小表做join时，可以使用mapjoin，性能比普通的join要快很多。mapjoin的基本原理是：在小数据量情况下，SQL会将用户指定的小表全部加载到执行join操作的程序的内存中，从而加快join的执行速度。需要注意，使用mapjoin时：

- left outer join的左表必须是大表；
- right outer join的右表必须是大表；
- inner join左表或右表均可以作为大表；
- full outer join不能使用mapjoin；
- mapjoin支持小表为子查询；
- 使用mapjoin时需要引用小表或是子查询时，需要引用别名；
- 在mapjoin中，可以使用不等值连接或者使用or连接多个条件；
- 目前MaxCompute 在mapjoin中最多支持指定6张小表，否则报语法错误；
- 如果使用mapjoin，则所有小表占用的内存总和不得超过512MB。请注意由于MaxCompute 是压缩存储，因此小表在被加载到内存后，数据大小会急剧膨胀。此处的512MB限制是加载到内存后的空间

大小；
- 多个表join时，最左边的两个表不能同时是mapjoin的表。

下面是一个简单的示例：

```
    select /* + mapjoin(a) */
a.shop_name,
b.customer_id,
b.total_price
from shop a join sale_detail b
on a.shop_name = b.shop_name;
```

MaxCompute SQL不支持支持在普通join的on条件中使用不等值表达式、or 逻辑等复杂的join条件，但是在mapjoin中可以进行如上操作，例如：

```
    select /*+ mapjoin(a) */
a.total_price,
b.total_price
from shop a join sale_detail b
on a.total_price < b.total_price or a.total_price + b.total_price < 500;
```

# HAVING子句

由于MaxCompute SQL的WHERE关键字无法与合计函数一起使用，可以采用having字句。

语法格式：

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
HAVING aggregate_function(column_name) operator value
```

使用场景举例：比如有一张订单表Orders，包括客户名称（Customer,），订单金额（OrderPrice），订单日期（Order_date），订单号（Order_id）四个字段。现在希望查找订单总额少于2000的客户。现在我们可以写如下语句：

```
SELECT Customer,SUM(OrderPrice) FROM Orders
GROUP BY Customer
HAVING SUM(OrderPrice)<2000
```

# 内建函数

MaxCompute SQL提供了针对datetime类型的操作函数。

# DATEADD

函数声明：

```
datetime dateadd(datetime date, bigint delta, string datepart)
```

用途：按照指定的单位datepart和幅度delta修改date的值。

参数说明：

- date：Datetime类型，日期值。若输入为string类型会隐式转换为datetime类型后参与运算，其它类型抛异常。。
- delta：Bigint类型，修改幅度。若输入为string类型或double型会隐式转换到bigint类型后参与运算，其他类型会引发异常。若delta大于0，加；否则减。
- datepart：String类型常量。此字段的取值遵循string与datetime类型转换的约定，即”yyyy”表示年，” mm”表示月....关于类型转换的规则请参考 String类型与Datetime类型之间的转换 。此外也支持扩展的日期格式：年-“year” ，月-“month”或”mon” ，日-“day” ，小时-“hour”。非常量、不支持的格式会或其它类型抛异常。

返回值：Datetime类型。若任一输入参数为NULL，返回NULL。

备注:

- 按照指定的单位增减delta时导致的对更高单位的进位或退位，年、月、时、分、秒分别按照10进制、12进制、24进制、60进制、60进制计算。当delta的单位是月时，计算规则如下：若datetime的月部分在增加delta值之后不造成day溢出，则保持day值不变，否则把day值设置为结果月份的最后一天。
- datepart的取值遵循string与datetime类型转换的约定，即”yyyy”表示年，” mm”表示月....datetime相关的内建函数如无特殊说明均遵守此约定。同时如果没有特殊说明，所有datetime相关的内建函数的part部分也同样支持扩展的日期格式：年-“year” ，月-“month”或”mon” ，日-“day” ，小时-“hour”。

示例：

```
   若trans_date = 2005-02-28 00:00:00：
dateadd(trans_date, 1, 'dd') = 2005-03-01 00:00:00
-- 加一天，结果超出当年2月份的最后一天，实际值为下个月的第一天
dateadd(trans_date, -1, 'dd') = 2005-02-27 00:00:00
-- 减一天
dateadd(trans_date, 20, 'mm') = 2006-10-28 00:00:00
-- 加20个月，月份溢出，年份加1
```

```
若trans_date = 2005-02-28 00:00:00, dateadd(transdate, 1, 'mm') = 2005-03-28 00:00:00
若trans_date = 2005-01-29 00:00:00, dateadd(transdate, 1, 'mm') = 2005-02-28 00:00:00
-- 2005年2月没有29日，日期截取至当月最后一天
若trans_date = 2005-03-30 00:00:00, dateadd(transdate, -1, 'mm') = 2005-02-28 00:00:00
```

此处对trans_date的数值表示仅作示例使用，在文档中有关datetime介绍会经常使用到这种简易的表达方式。
在MaxCompute SQL中，datetime类型没有直接的常数表示方式，如下使用方式是错误的：

```
    select dateadd(2005-03-30 00:00:00, -1, 'mm') from tbl1;
```

如果一定要描述datetime类型常量，请尝试如下方法：

```
    select dateadd(cast("2005-03-30 00:00:00" as datetime), -1, 'mm') from tbl1;
-- 将String类型常量显式转换为Datetime类型
```

# DATEDIFF

命令格式：

```
    bigint datediff(datetime date1, datetime date2, string datepart)
```

用途：计算两个时间date1，date2在指定时间单位datepart的差值。

参数说明：

- datet1，date2：Datetime类型，被减数和减数，若输入为string类型会隐式转换为datetime类型后参与运算，其它类型抛异常。
- datepart：String类型常量。支持扩展的日期格式。若datepart不符合指定格式或者其它类型则会发生异常。

返回值：Bigint类型。任一输入参数是NULL，返回NULL。如果date1小于date2，返回值可以为负数。

备注：

- 计算时会按照datepart切掉低单位部分，然后再计算结果。

示例：

```
    若start = 2005-12-31 23:59:59，end = 2006-01-01 00:00:00:
datediff(end, start, 'dd') = 1
datediff(end, start, 'mm') = 1
datediff(end, start, 'yyyy') = 1
datediff(end, start, 'hh') = 1
datediff(end, start, 'mi') = 1
datediff(end, start, 'ss') = 1
```

```
datediff(2013-05-31 13:00:00, 2013-05-31 12:30:00, 'ss') = 1800
datediff(2013-05-31 13:00:00, 2013-05-31 12:30:00, 'mi') = 30
```

# DATEPART

函数声明：

```
bigint datepart(datetime date, string datepart)
```

用途：提取日期date中指定的时间单位datepart的值。

参数说明：

- date：Datetime类型，若输入为string类型会隐式转换为datetime类型后参与运算，其它类型抛异常。
- datepart：String类型常量。支持扩展的日期格式。若datepart不符合指定格式或者其它类型则会发生异常。

返回值：Bigint类型。若任一输入参数为NULL，返回NULL。

示例：

```
datepart('2013-06-08 01:10:00', 'yyyy') = 2013
datepart('2013-06-08 01:10:00', 'mm') = 6
```

# DATETRUNC

函数声明：

```
datetime datetrunc (datetime date, string datepart)
```

用途：返回日期date被截取指定时间单位datepart后的日期值。

参数说明：

- date：Datetime类型，若输入为string类型会隐式转换为datetime类型后参与运算，其它类型抛异常。
- datepart：String类型常量。支持扩展的日期格式。若datepartt不符合指定格式或者其它类型则会发生异常。

返回值：Datetime类型。任意一个参数为NULL的时候返回NULL。

示例：

```
datetrunc(2011-12-07 16:28:46, 'yyyy') = 2011-01-01 00:00:00
datetrunc(2011-12-07 16:28:46, 'month') = 2011-12-01 00:00:00
```

```
datetrunc(2011-12-07 16:28:46, 'DD') = 2011-12-07 00:00:00
```

# FROM_UNIXTIME

函数声明：

```
datetime from_unixtime(bigint unixtime)
```

用途：将数字型的unix时间日期值unixtime转为日期值。

参数说明：

- unixtime：Bigint类型，秒数，unix格式的日期时间值，若输入为string，double，decimal类型会隐式转换为bigint后参与运算。

返回值：Datetime类型的日期值，unixtime为NULL时返回NULL。

示例：

```
from_unixtime(123456789) = 2009-01-20 21:06:29
```

# GETDATE

函数声明：

```
datetime getdate()
```

用途：获取当前系统时间。使用东八区时间作为MaxCompute标准时间。

返回值：返回当前日期和时间，datetime类型。

备注：

- 在一个MaxCompute SQL任务中(以分布式方式执行)，getdate总是返回一个固定的值。返回结果会是MaxCompute SQL执行期间的任意时间，时间精度精确到秒。

# ISDATE

函数声明：

```
boolean isdate(string date, string format)
```

用途：判断一个日期字符串能否根据对应的格式串转换为一个日期值，如果转换成功返回TRUE，否则返回

FALSE。

参数说明：

- date：String格式的日期值，若输入为bigint，double，decimal或者datetime类型会隐式转换为string类型后参与运算，其它类型报异常。
- format：String类型常量，不支持日期扩展格式。其它类型或不支持的格式会抛异常。如果format中出现多余的格式串，则只取第一个格式串对应的日期数值，其余的会被视为分隔符。如isdate(“1234-yyyy “, “yyyy-yyyy “)，会返回TRUE。

返回值：Boolean类型，如任意参数为NULL，返回NULL。

# LASTDAY

函数声明：

```
datetime lastday(datetime date)
```

用途：取date当月的最后一天，截取到天，时分秒部分为00:00:00。

参数说明：

- date：Datetime类型，若输入为string类型会隐式转换为datetime类型后参与运算，其它类型报异常。

返回值：Datetime类型，如输入为NULL，返回NULL

# TO_DATE

函数声明：

```
datetime to_date(string date, string format)
```

用途：将一个字符串date按照format指定的格式转成日期值。

参数说明：

- date：String类型，要转换的字符串格式的日期值，若输入为bigint，double，decimal或者datetime类型会隐式转换为String类型后参与运算，为其它类型抛异常，为空串时抛异常。
- format：String类型常量，日期格式。非常量或其他类型会引发异常。format不支持日期扩展格式，其他字符作为无用字符在解析时忽略。format参数至少包含”yyyy”，否则引发异常，如果format中出现多余的格式串，则只取第一个格式串对应的日期数值，其余的会被视为分隔符。如to_date(“1234-2234 “, “yyyy-yyyy “)会返回1234-01-01 00:00:00。

返回值：Datetime类型。若任一输入为NULL，返回NULL值。

示例：

```
    to_date('阿里巴巴2010-12*03', '阿里巴巴yyyy-mm*dd') = 2010-12-03 00:00:00
to_date('20080718', 'yyyymmdd') = 2008-07-18 00:00:00

to_date('2008718', 'yyyymmdd')
-- 格式不符合，引发异常
to_date('阿里巴巴2010-12*3', '阿里巴巴yyyy-mm*dd')
-- 格式不符合，引发异常
to_date('2010-24-01', 'yyyy')
-- 格式不符合，引发异常
```

# TO_CHAR

函数声明：

```
    string to_char(datetime date, string format)
```

用途：将日期类型date按照format指定的格式转成字符串

参数类型：

- date：Datetime类型，要转换的日期值，若输入为string类型会隐式转换为datetime类型后参与运算，其它类型抛异常。
- format：String类型常量。非常量或其他类型会引发异常。format中的日期格式部分会被替换成相应的数据，其它字符直接输出。

返回值：String类型。任一输入参数为NULL，返回NULL。

示例：

```
to_char('2010-12-03 00:00:00', '阿里金融yyyy-mm*dd') = '阿里金融2010-12*03'
to_char('2008-07-18 00:00:00', 'yyyymmdd') = '20080718'
to_char('阿里巴巴2010-12*3', '阿里巴巴yyyy-mm*dd') -- 引发异常
to_char('2010-24-01', 'yyyy') -- 会引发异常
to_char('2008718', 'yyyymmdd') -- 会引发异常
```

备注:

- 关于其他类型向string类型转换请参考 字符串函数 TO_CHAR 。

# UNIX_TIMESTAMP

函数声明：

```
bigint unix_timestamp(datetime date)
```

用途：将日期date转化为整型的unix格式的日期时间值。

参数说明：

- date：Datetime类型日期值，若输入为string类型会隐式转换为datetime类型后参与运算，其它类型抛异常。

返回值：Bigint类型，表示unix格式日期值，date为NULL时返回NULL。

# WEEKDAY

函数声明：

```
bigint weekday (datetime date)
```

用途：返回date日期当前周的第几天。

参数说明：

- date：Datetime类型，若输入为string类型会隐式转换为datetime类型后参与运算，其它类型抛异常。

返回值：Bigint类型，若输入参数为NULL，返回NULL。周一作为一周的第一天，返回值为0。其他日期依次递增，周日返回6。

# WEEKOFYEAR

函数声明：

```
bigint weekofyear(datetime date)
```

用途：返回日期date位于那一年的第几周。周一作为一周的第一天。需要注意的是，关于这一周算上一年， 还是下一年，主要是看这一周大多数日期（4天以上）在哪一年多。 算在前一年，就是前一年的最后一周。算在后一年就是后一年的第一周。

参数说明:

- date：Datetime类型日期值，若输入为string类型会隐式转换为datetime类型后参与运算，其它类型抛异常。

返回值：Bigint类型。若输入为NULL，返回NULL。

示例说明：

```
select weekofyear(to_date("20141229", "yyyymmdd")) from dual;
返回结果：
+------------+
| _c0 |
+------------+
| 1 |
+------------+
-虽然20141229属于2014年，但是这一周的大多数日期是在2015年，因此返回结果为1，表示是2015年的第一周。

select weekofyear(to_date("20141231", "yyyymmdd")) from dual；--返回结果为1。
select weekofyear(to_date("20151229", "yyyymmdd")) from dual；--返回结果为53。
```

# ABS

函数定义：

```
    double abs(double number)
bigint abs(bigint number)
decimal abs(decimal number)
```

用途：返回绝对值。

参数说明：

- number：Double或bigint类型或Decimal类型，输入为bigint时返回bigint，输入为double时返回double类型。输入decimal类型时返回decimal类型。若输入为string类型会隐式转换到double类型后参与运算，其它类型抛异常。

返回值：Double或者bigint类型或者decimal类型，取决于输入参数的类型。若输入为null，返回null。

备注：

- 当输入bigint类型的值超过bigint的最大表示范围时，会返回double类型，这种情况下可能会损失精度。

示例：

```
    abs(null) =  null
abs(-1) = 1
abs(-1.2) = 1.2
abs("-2") = 2.0
abs(12232083745629837659238745692374 8) = 1.2232083745629837e32
```

下面是一个完整的abs函数在SQL中使用的例子，其他内建函数(除窗口函数、聚合函数外)的使用方式与其类似，不再一一举例：

```
    select abs(id) from tbl1;
-- 取tbl1表内id字段的绝对值
```

# ACOS

函数定义：

```
    double acos(double number)
decimal acos(decimal number)
```

用途：计算number的反余弦函数。

参数说明：

- number：Double类型或Decimal类型，-1≤number≤1。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。

返回值：Double类型或Decimal类型，值域在0 ~ π 之间。若number为NULL，返回NULL。

示例：

```
    acos("0.87") = 0.5155940062460905
acos(0) = 1.5707963267948966
```

# ASIN

函数定义：

```
    double asin(double number)
decimal asin(DECIMAL number)
```

用途：反正弦函数。

参数说明：

- number：Double类型或Decimal类型，-1≤number≤1。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。

返回值：Double类型或Decimal类型，值域在-π/2 ~π/2之间。若number为NULL，返回NULL。

示例：

```
    asin(1) = 1.5707963267948966
asin(-1) = -1.5707963267948966
```

# ATAN

函数定义：

```
double atan(double number)
```

用途：反正切函数。

参数说明：

- number：Double类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。

返回值：Double类型，值域在-π/2 ~π/2之间。若number为NULL，返回NULL。

示例：

```
atan(1) = 0.7853981633974483
atan(-1) = -0.7853981633974483
```

# CEIL

函数定义：

```
bigint ceil(double value)
bigint ceil(decimal value)
```

用途：返回不小于输入值value的最小整数

参数说明：

- value：Double类型或Decimal类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。

返回值：Bigint类型。任一输入为NULL，返回NULL。

示例：

```
ceil(1.1) = 2
ceil(-1.1) = -1
```

# CONV

函数定义：

```
string conv(string input, bigint from_base, bigint to_base)
```

用途：进制转换函数

参数说明：

- input：以string表示的要转换的整数值，接受bigint，double的隐式转换。
- from_base，to_base：以十进制表示的进制的值，可接受的的值为2，8，10，16。接受string及double的隐式转换。

返回值：String类型。任一输入为NULL，返回NULL。转换过程以64位精度工作，溢出时报异常。输入如果是负值，即以”-“开头，报异常。如果输入的是小数，则会转为整数值后进行进制转换，小数部分会被舍弃。

示例

```
conv('1100', 2, 10) = '12'
conv('1100', 2, 16) = 'c'
conv('ab', 16, 10) = '171'
conv('ab', 16, 16) = 'ab'
```

# COS

函数定义：

```
double cos(double number)
decimal cos(decimal number)
```

用途：余弦函数，输入为弧度值。

参数说明：

- number：Double类型或Decimal类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。

返回值：Double类型或Decimal类型。若number为NULL，返回NULL。

示例：

```
cos(3.1415926/2)=2.6794896585028633e-8
cos(3.1415926)=0.9999999999999986
```

# COSH

函数定义：

```
double cosh(double number)
```

```
decimal cosh(decimal number)
```

用途：双曲余弦函数。

参数说明：

- number：Double类型或Decimal类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型或Decimal类型。若number为NULL，返回NULL。

# COT

函数定义：

```
double cot(double number)
decimal cot(decimal number)
```

用途：余切函数，输入为弧度值。

参数说明：

- number：Double类型或Decimal类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型或Decimal类型。若number为NULL，返回NULL。

# EXP

函数定义：

```
double exp(double number)
decimal exp(decimal number)
```

用途：指数函数。返回number的指数值。

参数说明：

- number：Double类型或Decimal类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型或Decimal类型。若number为NULL，返回NULL。

# FLOOR

函数定义：

```
bigint floor(double number)
```

```
bigint floor(decimal number)
```

用途：向下取整，返回比number小的整数值。

参数说明：

- number：Double类型或Decimal类型，若输入为string类型或bigint型会隐式转换到double类型后参与运算，其他类型抛异常返回值：返回Bigint类型。若number为NULL，返回NULL。

示例

```
    floor(1.2)=1
floor(1.9)=1
floor(0.1)=0
floor(-1.2)=-2
floor(-0.1)=-1
floor(0.0)=0
floor(-0.0)=0
```

# LN

函数定义：

```
    double ln(double number)
decimal ln(decimal number)
```

用途：返回number的自然对数。

参数说明：

- number：Double类型或Decimal类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。若number为NULL返回NULL,若number为负数或零，则抛异常。返回值：Double类型或Decimal类型。

# LOG

函数定义：

```
    double log(double base, double x)
decimal log(decimal base, DECIMAL x)
```

用途：返回以base为底的x的对数。

参数说明：

- base：Double类型或Decimal类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。

- x：Double类型或Decimal类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型或Decimal类型的对数值，若base和x中存在NULL，则返回NULL；若base和x中某一个值为负数或0，会引发异常；若base为1(会引发一个除零行为)也会引发异常。

# POW

函数定义：

```
    double pow(double x, double y)
decimal pow(decimal x, DECIMAL y)
```

用途：返回x的y次方，即x^y。

参数说明：

- X：Double类型或Decimal类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。
- Y：Double类型或Decimal类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型或Decimal类型。若x或y为NULL，则返回NULL

# RAND

函数定义：

```
    double rand(bigint seed)
```

用途：以seed为种子返回double类型的随机数，返回值区间是的0～1。

参数说明：

- seed：可选参数，Bigint类型，随机数种子，决定随机数序列的起始值。

返回值：Double类型。

示例：

```
select rand() from dual;
select rand(1) from dual;
```

# ROUND

函数定义：

```
   double round(double number, [bigint decimal_places])
decimal round(decimal number, [bigint decimal_places])
```

用途：四舍五入到指定小数点位置。

参数说明：

- number：Double类型或Decimal类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。
- decimal_place：Bigint类型常量，四舍五入计算到小数点后的位置，其他类型参数会引发异常. 如果省略表示四舍五入到个位数。默认值为0。返回值：返回Double类型或Decimal类型。若number或decimal_places为NULL，返回NULL。

备注：

- decimal_places可以是负数。负数会从小数点向左开始计数，并且不保留小数部分；如果decimal_places超过了整数部分长度，返回0.示例：

```
   round(125.315) = 125.0
round(125.315, 0) = 125.0
round(125.315, 1) = 125.3
round(125.315, 2) = 125.32
round(125.315, 3) = 125.315
round(-125.315, 2) = -125.32
round(123.345, -2) = 100.0
round(null) = null
round(123.345, 4) = 123.345
round(123.345, -4) = 0.0
```

# SIGN

函数声明：

```
sign(x)
```

用途：判断x是否为正值或者是否为负值。参数说明：x:Double类型或者Decimal类型，可以为常量、函数或者表达式。返回值：

- 当x为正值时，返回1.0。
- 当x为负值时，返回-1.0。
- 当x为0时，返回0.0。
- 当x为空时，抛异常。

示例：

```
select sign(5-13) from dual;
```

```
返回：
+------------+
| _c0 |
+------------+
| -1.0 |
+------------+
```

# SIN

函数定义：

```
    double sin(double number)
decimal sin(decimal number)
```

用途：正弦函数，输入为弧度值。

参数说明：

- number：Double类型或Decimal类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型或Decimal类型。若number为NULL，返回NULL。

# SINH

函数定义：

```
    double sinh(double number)
decimal sinh(decimal number)
```

用途：双曲正弦函数。

参数说明：

- number：Double类型或Decimal类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型或Decimal类型。若number为NULL，返回NULL。

# SQRT

函数定义：

```
    double sqrt(double number)
decimal sqrt(decimal number)
```

用途：计算平方根。

参数说明：

- number：Double类型或Decimal类型，必须大于0。小于0时引发异常。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：返回double类型或Decimal类型。若number为NULL，返回NULL。

# TAN

函数声明：

```
    double tan(double number)
decimal tan(decimal number)
```

用途：正切函数，输入为弧度值。

参数说明：

- number：Double类型或Decimal类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型或Decimal类型。若number为NULL，返回NULL。

# TANH

函数声明：

```
    double tanh(double number)
decimal tanh(decimal number)
```

用途：双曲正切函数。

参数说明：

- number：Double类型或Decimal类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型或Decimal类型。若number为NULL，返回NULL。

# TRUNC

函数声明：

```
    double trunc(double number[, bigint decimal_places])
decimal trunc(decimal number[, bigint decimal_places])
```

用途：将输入值number截取到指定小数点位置。

参数说明：

- number：Double类型或Decimal类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。
- decimal_places：Bigint类型常量，要截取到的小数点位置，其他类型参数会隐式转为bigint，省略此参数时默认到截取到个位数。返回值：返回值类型为Double或Decimal类型。若number或decimal_places为NULL，返回NULL。

备注:

- 截取掉的部分补0。
- decimal_places可以是负数，负数会从小数点向左开始截取，并且不保留小数部分；如果decimal_places超过了整数部分长度，返回0。

示例：

```
   trunc(125.815) = 125.0
trunc(125.815, 0) =125.0
trunc(125.815, 1) = 125.8
trunc(125.815, 2) = 125.81
trunc(125.815, 3) = 125.815
trunc(-125.815, 2) = -125.81
trunc(125.815, -1) = 120.0
trunc(125.815, -2) = 100.0
trunc(125.815, -3) = 0.0
trunc(123.345, 4) = 123.345
trunc(123.345, -4) = 0.0
```

MaxCompute SQL中可以使用窗口函数进行灵活的分析处理工作，窗口函数只能出现在select子句中，窗口函数中不要嵌套使用窗口函数和聚合函数，窗口函数不可以和同级别的聚合函数一起使用。目前在一个MaxCompute SQL语句中，可以使用至多5个窗口函数。

窗口函数的语法

```
   window_func() over (partition by col1, [col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]] windowing_clause)
```

- partition by部分用来指定开窗的列。分区列的值相同的行被视为在同一个窗口内。现阶段，同一窗口内最多包含1亿行数据，否则运行时报错。
- order by用来指定数据在一个窗口内如何排序
- windowing_clause部分可以用rows指定开窗方式，有两种方式： **rows between x preceding|following and y preceding|following表示窗口范围是从前或后x行到前或后y行。** rows x preceding|following窗口范围是从前或后第x行到当前行。** x，y必须为大于等于0的整数常量，限定范围0 ~ 10000，值为0时表示当前行。必须指定order by才可以用rows方式指定窗口范围。

备注:

- 并非所有的窗口函数都可以用rows指定开窗方式，支持这种用法的窗口函数有avg、count、max、min、stddev和sum。

# COUNT

函数声明：

```
bigint count([distinct] expr) over(partition by col1[, col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]] [windowing_clause])
```

用途：计算计数值。

参数说明：

- expr：任意类型，当值为NULL时，该行不参与计算。当指定distinct关键字时表示取唯一值的计数值。
- partition by col1[, col2...]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：不指定order by时，返回当前窗口内expr的计数值，指定order by时返回结果以指定的顺序排序，并且值为当前窗口内从开始行到当前行的累计计数值。

返回值：Bigint类型。

备注:

- 当指定distinct关键字时不能写order by。

示例：假设存在表test_src，表中存在bigint类型的列user_id，

```
select user_id,
count(user_id) over (partition by user_id) as count
from test_src;

+---------+------------+
| user_id | count |
+---------+------------+
| 1 | 3 |
| 1 | 3 |
| 1 | 3 |
| 2 | 1 |
| 3 | 1 |
+---------+------------+

-- 不指定order by时，返回当前窗口内user_id的计数值

select user_id,
count(user_id) over (partition by user_id order by user_id) as count
```

```
from test_src;

+---------+------------+
| user_id | count |
+---------+------------+
| 1 | 1 | -- 窗口起始
| 1 | 2 | -- 到当前行共计两条记录，返回2
| 1 | 3 |
| 2 | 1 |
| 3 | 1 |
+---------+------------+

-- 指定order by时，返回当前窗口内从开始行到当前行的累计计数值。
```

# AVG

函数声明：

```
    avg([distinct] expr) over(partition by col1[, col2...]
[order by col1 [asc|desc] [, col2[asc|desc]...]] [windowing_clause])
```

用途：计算平均值。

参数说明：

- distinct：当指定distinct关键字时表示取唯一值的平均值。
- expr：Double类型，decimal类型,若输入为string，bigint会隐式转换到double类型后参与运算，其它类型抛异常。当值为NULL时，该行不参与计算。Boolean类型不允许参与计算。
- partition by col1[, col2]...：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：不指定order by时返回当前窗口内所有值的平均值，指定order by时返回结果以指定的方式排序，并且返回窗口内从开始行到当前行的累计平均值。

返回值：Double类型。

备注:

- 指明distinct关键字时不能写order by。

# MAX

函数声明：

```
    max([distinct] expr) over(partition by col1[, col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]] [windowing_clause])
```

用途：计算最大值。

参数说明：

- expr：除Boolean以外的任意类型，当值为NULL时，该行不参与计算。当指定distinct关键字时表示取唯一值的最大值(指定该参数与否对结果没有影响)。
- partition by col1[, col2...]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：不指定order by时，返回当前窗口内的最大值。指定order by时，返回结果以指定的方式排序，并且值为当前窗口内从开始行到当前行的最大值。

返回值：同expr类型。

备注：

- 当指定distinct关键字时不能写order by。

# MIN

函数声明：

```
min([distinct] expr) over(partition by col1[, col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]] [windowing_clause])
```

用途：计算最小值。

参数说明：

- expr：除boolean以外的任意类型，当值为NULL时，该行不参与计算。当指定distinct关键字时表示取唯一值的最小值(指定该参数与否对结果没有影响)。
- partition by col1[, col2..]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：不指定order by时，返回当前窗口内的最小值。指定order by时，返回结果以指定的方式排序，并且值为当前窗口内从开始行到当前行的最小值。

返回值：同expr类型。

备注：

- 当指定distinct关键字时不能写order by。

# MEDIAN

函数声明：

```
double median(double number1,number2...) over(partition by col1[, col2...])

decimal median(decimal number1,number2...) over(partition by col1[,col2...])
```

用途：计算中位数。

参数说明：

number1,number1...：Double类型或decimal类型的1到255个数字。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。当输入值为null时忽略。如果传入的参数是一个Double类型的数，会默认转成一个double 的array。

partition by col1[, col2...]：指定开窗口的列。

返回值：Double类型。

# STDDEV

函数声明：

```
double stddev([distinct] expr) over(partition by col1[, col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]] [windowing_clause])
decimal stddev([distinct] expr) over(partition by col1[,col2...] [order by col1 [asc|desc][, col2[asc|desc]...]]
[windowing_clause])
```

用途：总体标准差。

参数说明：

expr：Double类型或decimal类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。当输入值为NULL时忽略该行。当指定distinct关键字时表示计算唯一值的总体标准差。

partition by col1[, col2..]：指定开窗口的列。

order by col1 [asc|desc], col2[asc|desc]：不指定order by时，返回当前窗口内的总体标准差。指定order by时，返回结果以指定的方式排序，并且值为当前窗口内从开始行到当前行的总体标准差。

返回值：输入为Decimal类型时返回Decimal类型，否则返回Double类型。

示例：

```
select window, seq, stddev_pop('1\01') over (partition by window order by seq) from dual;
```

备注：

- 当指定distinct关键字时不能写order by。
- stddev还有一个别名函数stddev_pop，用法跟stddev一样。

# STDDEV_SAMP

函数声明：

```
double stddev_samp([distinct] expr) over(partition by col1[, col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]] [windowing_clause])
DECIMAL STDDEV_SAMP([DISTINCT] expr) OVER(PARTITION BY col1[,col2...] [ORDER BY col1 [ASC|DESC][,
col2[ASC|DESC]...]] [windowing_clause])
```

用途：样本标准差。

参数说明：

- expr：Double类型或decimal类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。当输入值为NULL时忽略该行。当指定distinct关键字时表示计算唯一值的样本标准差。
- partition by col1[, col2..]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：不指定order by时，返回当前窗口内的样本标准差。指定order by时，返回结果以指定的方式排序，并且值为当前窗口内从开始行到当前行的样本标准差。

返回值：输入为Decimal类型时返回Decimal类型，否则返回Double类型。

备注：

- 当指定distinct关键字时不能写order by。

# SUM

函数声明：

```
sum([distinct] expr) over(partition by col1[, col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]] [windowing_clause])
```

用途：计算汇总值。

参数说明：

- expr：Double类型或Decimal类型或Bigint类型，当输入为string时隐式转换为double参与运算，其它类型报异常。当值为NULL时，该行不参与计算。指定distinct关键字时表示计算唯一值的汇总值。
- partition by col1[, col2..]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：不指定order by时，返回当前窗口内expr的汇总值。指定

order by时，返回结果以指定的方式排序，并且返回当前窗口从首行至当前行的累计汇总值。返回值：输入参数是bigint返回bigint,输入参数为decimal类型时返回decimal类型， 输入参数为double或string时，返回double类型。

备注：

- 当指定distinct时不能用order by。

# DENSE_RANK

命令格式：

```
bigint dense_rank() over(partition by col1[, col2...]
order by col1 [asc|desc][, col2[asc|desc]...])
```

用途：计算连续排名。col2相同的行数据获得的排名相同。

参数说明：

- partition by col1[, col2..]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：指定排名依据的值。

返回值：Bigint类型。

# RANK

命令格式：

```
bigint rank() over(partition by col1[, col2...]
order by col1 [asc|desc][, col2[asc|desc]...])
```

用途：计算排名。col2相同的行数据获得排名顺序下降。

参数说明：

- partition by col2[, col2..]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：指定排名依据的值。

返回值：Bigint类型。

# LAG

函数声明：

```
    lag(expr，bigint offset, default) over(partition by col1[, col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]])
```

用途：按偏移量取当前行之前第几行的值，如当前行号为rn，则取行号为rn-offset的值。

参数说明：

- expr：任意类型。
- offset：Bigint类型常量，输入为string，double到bigint的隐式转换，offset > 0。
- default：当offset指定的范围越界时的缺省值，常量，默认值为NULL。
- partition by col1[, col2..]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：指定返回结果的排序方式。

返回值：同expr类型。

# LEAD

函数声明：

```
lead(expr, bigint offset, default) over(partition by col1[, col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]])
```

用途：按偏移量取当前行之后第几行的值，如当前行号为rn则取行号为rn+offset的值。

参数说明：

expr：任意类型。

offset：可选，Bigint类型常量，输入为string，decimal，double到bigint的隐式转换，offset > 0。

default：可选，当offset指定的范围越界时的缺省值，常量。

partition by col1[, col2..]：指定开窗口的列。

order by col1 [asc|desc], col2[asc|desc]：指定返回结果的排序方式。

返回值：同expr类型。

示例：

```
select c_double_a,c_string_b,c_int_a,lead(c_int_a,1) over(partition by c_double_a order by c_string_b) from dual;
```

```
select c_string_a,c_time_b,c_double_a,lead(c_double_a,1) over(partition by c_string_a order by c_time_b) from dual;

select c_string_in_fact_num,c_string_a,c_int_a,lead(c_int_a) over(partition by c_string_in_fact_num order by c_string_a)
from dual;
```

# PERCENT_RANK

函数声明：

```
    percent_rank() over(partition by col1[, col2...]
order by col1 [asc|desc][, col2[asc|desc]...])
```

用途：计算一组数据中某行的相对排名。

参数说明：

- partition by col1[, col2..]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：指定排名依据的值。

返回值：Double类型，值域为[0, 1]，相对排名的计算方式为为：(rank-1)/(number of rows -1)。

备注:

- 目前限制单个窗口内的行数不超过10,000,000条。

# ROW_NUMBER

函数声明：

```
    row_number() over(partition by col1[, col2...]
order by col1 [asc|desc][, col2[asc|desc]...])
```

用途：计算行号，从1开始。

参数说明：

- partition by col1[, col2..]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：指定结果返回时的排序的值。

返回值：Bigint类型。

示例，如表emp中数据如下：

```
| empno | ename | job | mgr | hiredate| sal| comm | deptno |
7369,SMITH,CLERK,7902,1980-12-17 00:00:00,800,,20
7499,ALLEN,SALESMAN,7698,1981-02-20 00:00:00,1600,300,30
7521,WARD,SALESMAN,7698,1981-02-22 00:00:00,1250,500,30
```

```
7566,JONES,MANAGER,7839,1981-04-02 00:00:00,2975,,20
7654,MARTIN,SALESMAN,7698,1981-09-28 00:00:00,1250,1400,30
7698,BLAKE,MANAGER,7839,1981-05-01 00:00:00,2850,,30
7782,CLARK,MANAGER,7839,1981-06-09 00:00:00,2450,,10
7788,SCOTT,ANALYST,7566,1987-04-19 00:00:00,3000,,20
7839,KING,PRESIDENT,,1981-11-17 00:00:00,5000,,10
7844,TURNER,SALESMAN,7698,1981-09-08 00:00:00,1500,0,30
7876,ADAMS,CLERK,7788,1987-05-23 00:00:00,1100,,20
7900,JAMES,CLERK,7698,1981-12-03 00:00:00,950,,30
7902,FORD,ANALYST,7566,1981-12-03 00:00:00,3000,,20
7934,MILLER,CLERK,7782,1982-01-23 00:00:00,1300,,10
```

现在需要列出每个部门的薪水前3名的人员的姓名以及他们的名次

```
SELECT *
FROM (
SELECT deptno
, ename
, sal
, ROW_NUMBER() OVER (PARTITION BY deptno ORDER BY sal DESC) AS nums
FROM emp
) emp1
WHERE emp1.nums < 4;
--执行结果如下：
+------------+-------+------------+------------+
| deptno | ename | sal | nums |
+------------+-------+------------+------------+
| 10 | KING | 5000.0 | 1 |
| 10 | CLARK | 2450.0 | 2 |
| 10 | MILLER | 1300.0 | 3 |
| 20 | SCOTT | 3000.0 | 1 |
| 20 | FORD | 3000.0 | 2 |
| 20 | JONES | 2975.0 | 3 |
| 30 | BLAKE | 2850.0 | 1 |
| 30 | ALLEN | 1600.0 | 2 |
| 30 | TURNER | 1500.0 | 3 |
+------------+-------+------------+------------+
```

# CLUSTER_SAMPLE

函数声明：

```
    boolean cluster_sample(bigint x[, bigint y])
over(partition by col1[, col2..])
```

用途：分组抽样。

参数说明：

- x：Bigint类型常量，x>=1。若指定参数y，x表示将一个窗口分为x份；否则，x表示在一个窗口中抽取x行记录(即有x行返回值为true)。x为NULL时，返回值为NULL。

- y：Bigint类型常量，y>=1，y<=x。表示从一个窗口分的x份中抽取y份记录(即y份记录返回值为true)。y为NULL时，返回值为NULL。
- partition by col1[, col2]：指定开窗口的列。

返回值：Boolean类型。

示例，如表test_tbl中有key，value两列，key为分组字段，值有groupa，groupb两组，value为值，如下

```
+------------+--------------------+
| key | value |
+------------+--------------------+
| groupa | -1.34764165478145 |
| groupa | 0.740212609046718 |
| groupa | 0.167537127858695 |
| groupa | 0.630314566185241 |
| groupa | 0.0112401388646925 |
| groupa | 0.199165745875297 |
| groupa | -0.320543343353587 |
| groupa | -0.273930924365012 |
| groupa | 0.386177958942063 |
| groupa | -1.09209976687047 |
| groupb | -1.10847690938643 |
| groupb | -0.725703978381499 |
| groupb | 1.05064697475759 |
| groupb | 0.135751224393789 |
| groupb | 2.13313102040396 |
| groupb | -1.11828960785008 |
| groupb | -0.849235511508911 |
| groupb | 1.27913806620453 |
| groupb | -0.330817716670401 |
| groupb | -0.300156896191195 |
| groupb | 2.4704244205196 |
| groupb | -1.28051882084434 |
+------------+--------------------+
```

想要从每组中抽取约10%的值，可以用以下MaxCompute SQL完成：

```
select key, value
from (
select key, value, cluster_sample(10, 1) over(partition by key) as flag
from tbl
) sub
where flag = true;

+--------+--------------------+
| key | value |
+--------+--------------------+
| groupa | -1.34764165478145 |
| groupb | -0.725703978381499 |
| groupb | 2.4704244205196 |
+-----+----------------------+
```

# CHAR_MATCHCOUNT

函数声明：

```
bigint char_matchcount(string str1, string str2)
```

用途：用于计算str1中有多少个字符出现在str2中。

参数说明：

- str1，str2：String类型，必须为有效的UTF-8字符串，如果对比中发现有无效字符则函数返回负值。

返回值：Bigint类型。任一输入为NULL返回NULL。

示例：

```
char_matchcount('abd', 'aabc') = 2
-- str1中得两个字符串'a', 'b'在str2中出现过
```

# CHR

函数声明：

```
string chr(bigint ascii)
```

用途：将给定ASCII码ascii转换成字符。

参数说明：

- ascii：Bigint类型ASCII值，若输入为string类型或double类型或decimal类型会隐式转换到bigint类型后参与运算，其它类型抛异常。

返回值：String类型。参数范围是0~255，超过此范围会引发异常。输入值为NULL返回NULL。

# CONCAT

函数声明：

```
string concat(string a, string b...)
```

用途：返回值是将参数中的所有字符串连接在一起的结果。

参数说明：

- a，b等为String类型，若输入为bigint，decimal，double或datetime类型会隐式转换为string后参与运算，其它类型报异常。

返回值：String类型。如果没有参数或者某个参数为NULL，结果均返回NULL。

示例：

```
    concat('ab', 'c') = 'abc'
concat() = NULL
concat('a', null, 'b') = NULL
```

# GET_JSON_OBJECT

函数声明:

```
    STRING GET_JSON_OBJECT(STRING json,STRING path)
```

用途：在一个标准json字符串中，按照path抽取指定的字符串.

参数说明：

- json:String类型，标准的json格式字符串。
- path: **String类型，用于描述在json中的path，以$开头。** 关于新实现中json path的说明：参考 http://goessner.net/articles/JsonPath/index.html#e2***$表示根节点*** “.”表示child ***“[number]”表示数组下标*** 对于数组，格式为 key[sub1][sub2][sub3]……***[]返回整个数组**** 不支持转义

返回值：String类型

注解：

- 如果json为空或者非法的json格式，返回NULL
- 如果path为空或者不合法（json中不存在）返回NULL
- 如果json合法，path也存在则返回对应字符串

示例1

```
+----+
json
+----+
{"store":
{"fruit":\[{"weight":8,"type":"apple"},{"weight":9,"type":"pear"}],
"bicycle":{"price":19.95,"color":"red"}
},
"email":"amy@only_for_json_udf_test.net",
"owner":"amy"
```

```
}
```

通过以下查询，可以提取json对象中的信息：

```
 odps> SELECT get_json_object(src_json.json, '$.owner') FROM src_json;
amy
odps> SELECT get_json_object(src_json.json, '$.store.fruit\[0]') FROM src_json;
{"weight":8,"type":"apple"}
odps> SELECT get_json_object(src_json.json, '$.non_exist_key') FROM src_json;
NULL
```

示例2

```
get_json_object('{"array":[[aaaa,1111],[bbbb,2222],[cccc,3333]]}','$.array[1].[1]') = "2222"

get_json_object('{"aaa":"bbb","ccc":{"ddd":"eee","fff":"ggg","hhh":["h0","h1","h2"]},"iii":"jjj"}','$.ccc.hhh[*]') =
"["h0","h1","h2"]"

get_json_object('{"aaa":"bbb","ccc":{"ddd":"eee","fff":"ggg","hhh":["h0","h1","h2"]},"iii":"jjj"}','$.ccc.hhh[1]') = "h1"
```

# INSTR

函数声明：

```
bigint instr(string str1, string str2[, bigint start_position[, bigint nth_appearance]])
```

用途：计算一个子串str2在字符串str1中的位置。

参数说明：

- str1：String类型，搜索的字符串，若输入为bigint，decimal，double或datetime类型会隐式转换为string后参与运算，其它类型报异常。
- str2：String类型，要搜索的子串，若输入为bigint，decimal，double或datetime类型会隐式转换为string后参与运算，其它类型报异常。
- start_position：Bigint类型，其它类型会抛异常，表示从str1的第几个字符开始搜索，默认起始位置是第一个字符位置1。
- nth_appearance：Bigint类型，大于0，表示子串在字符串中的第nth_appearance次匹配的位置，如果nth_appearance为其它类型或小于等于0会抛异常。

返回值：Bigint类型。

备注:

- 如果在str1中未找到str2，返回0。
- 任一输入参数为NULL返回NULL
- 如果str2为空串时总是能匹配成功，因此instr（‘abc’，‘’）会返回1。

示例：

```
    instr('Tech on the net', 'e') = 2
instr('Tech on the net', 'e', 1, 1) = 2
instr('Tech on the net', 'e', 1, 2) = 11
instr('Tech on the net', 'e', 1, 3) = 14
```

# IS_ENCODING

函数声明：

```
    boolean is_encoding(string str, string from_encoding, string to_encoding)
```

用途：判断输入字符串str是否可以从指定的一个字符集from_encoding转为另一个字符集to_encoding。可用于判断输入是否为”乱码”，通常的用法是将from_encoding设为”utf-8”，to_encoding设为”gbk”。

参数说明：

- str：String类型，输入为NULL返回NULL。空字符串则可以被认为属于任何字符集。
- from_encoding，to_encoding：String类型，源及目标字符集。输入为NULL返回NULL。

返回值：Boolean类型，如果str能够成功转换，则返回true，否则返回false

示例：

```
    is_encoding('测试', 'utf-8', 'gbk') = true
is_encoding('測試', 'utf-8', 'gbk') = true
-- gbk字库中有这两个繁体字
is_encoding('測試', 'utf-8', 'gb2312') = false
-- gb2312库中不包括这两个字
```

# KEYVALUE

函数声明:

```
KEYVALUE(STRING srcStr，STRING split1，STRING split2， STRING key)
KEYVALUE(STRING srcStr， STRING key) //split1 = ";"，split2 = ":"
```

用途: 将srcStr（源字符串）按split1分成“key-value”对，按split2将key-value对分开，返回“key”所对应的value。

参数说明：

- srcStr 输入待拆分的字符串。

- key：string类型。源字符串按照split1和split2拆分后，根据该key值的指定，返回其对应的value。
- split1, split2：用来作为分隔符的字符串，按照指定的这两个分隔符拆分源字符串。如果表达式中没有指定这两项，默认split1为’;’，split2为’:’。当某个被split1拆分后的字符串中有多个split2时，返回结果未定义；

返回值:

- String类型。
- Split1或split2为NULL时，返回NULL。
- srcStr,key为NULL或者没有匹配的key时，返回NULL。
- 如果有多个key-value匹配，返回第一个匹配上的key对应的value。

示例：

```
keyvalue('0:1\;1:2', 1) = '2'
-源字符串为“0:1\;1:2”，因为没有指定split1和split2，默认split1为";"，split2为“:”。经过split1拆分后，key-value对为：
0:1\,1:2
经过split2拆分后变成：
0 1/
1 2
返回key为1所对应的value值，为2。

keyvalue("\;decreaseStore:1\;xcard:1\;isB2C:1\;tf:21910\;cart:1\;shipping:2\;pf:0\;market:shoes\;instPayAmount:0\;","\;",":","tf") = "21910"
-源字符串为
“\;decreaseStore:1\;xcard:1\;isB2C:1\;tf:21910\;cart:1\;shipping:2\;pf:0\;market:shoes\;instPayAmount:0\;”，按照split1“\;”拆分后，得出的key-value对为：
decreaseStore:1，xcard:1，isB2C:1，tf:21910，cart:1，shipping:2，pf:0，market:shoes，instPayAmount:0
按照split2":"拆分后变成：
decreaseStore 1
xcard 1
isB2C 1
tf 21910
cart 1
shipping 2
pf 0
market shoes
instPayAmount 0
key值为“tf”,返回其对应的value:21910。

keyvalue("阿里云=飞天=2;飞天=数据平台", ";","=", "阿里云") 返回NULL，请用户避免这种用法。
```

# LENGTH

命令格式：

```
    bigint length(string str)
```

用途：返回字符串str的长度。

参数说明：

- str：String类型，若输入为bigint，double，decimal或datetime类型会隐式转换为string后参与运算，其它类型报异常。

返回值：Bigint类型。若str是NULL返回NULL。如果str非UTF-8编码格式，返回-1。

示例

```
length('hi! 中国') = 6
```

# LENGTHB

函数声明：

```
bigint lengthb(string str)
```

用途：返回字符串str的以字节为单位的长度。

参数说明：

- str：String类型，若输入为bigint，double，decimal或者datetime类型会隐式转换为string后参与运算，其它类型报异常。

返回值：Bigint类型。若str是NULL返回NULL。

示例

```
lengthb('hi! 中国') = 10
```

# MD5

函数声明：

```
string md5(string value)
```

用途：计算输入字符串value的md5值

参数说明：

- value：String类型，如果输入类型是bigint，double，decimal或者datetime会隐式转换成string类型参与运算，其它类型报异常。输入为NULL，返回NULL。

返回值：String类型。

# REGEXP_EXTRACT

函数声明：

```
string regexp_extract(string source, string pattern[, bigint occurrence])
```

用途：将字符串source按照pattern正则表达式的规则拆分，返回第occurrence个group的字符。

参数说明：

- source：String类型，待搜索的字符串。
- pattern：String类型常量，pattern为空串时抛异常，pattern中如果没有指定group，抛异常。
- occurrence：Bigint类型常量，必须>=0，其它类型或小于0时抛异常，不指定时默认为1，表示返回第一个group。若occurrence = 0，返回满足整个pattern的子串。

返回值：String类型，任一输入为NULL返回NULL。

示例：

```
regexp_extract('foothebar', 'foo(.*?)(bar)', 1) = the
regexp_extract('foothebar', 'foo(.*?)(bar)', 2) = bar
regexp_extract('foothebar', 'foo(.*?)(bar)', 0) = foothebar
regext_extract('8d99d8', '8d(\\d+)d8') = 99
-- 如果是在MaxCompute客户端上提交正则计算的SQL，需要使用两个"\"作为转移字符

regexp_extract('foothebar', 'foothebar')
-- 异常返回，pattern中没有指定group
```

# REGEXP_INSTR

命令格式：

```
bigint regexp_instr(string source, string pattern[,
bigint start_position[, bigint nth_occurrence[, bigint return_option]])
```

用途：返回字符串source从start_position开始，和pattern第n次(nth_occurrence)匹配的子串的起始/结束位置。任一输入参数为NULL时返回NULL。

参数说明：

- source：String类型，待搜索的字符串。
- pattern：String类型常量，pattern为空串时抛异常。
- start_position：Bigint类型常量，搜索的开始位置。不指定时默认值为1，其它类型或小于等于0的值会抛异常。

- nth_occurrence：Bigint类型常量，不指定时默认值为1，表示搜索第一次出现的位置。小于等于0或者其它类型抛异常。
- return_option：Bigint类型常量，值为0或1，其它类型或不允许的值会抛异常。0表示返回匹配的开始位置，1表示返回匹配的结束位置。

返回值：Bigint类型。视return_option指定的类型返回匹配的子串在source中的开始或结束位置。

示例：

```
regexp_instr("i love www.taobao.com", "o[[:alpha:]]{1}", 3, 2) = 14
```

# REGEXP_REPLACE

函数声明：

```
string regexp_replace(string source, string pattern, string replace_string[, bigint occurrence])
```

用途：将source字符串中第occurrence次匹配pattern的子串替换成指定字符串replace_string后返回。

参数说明：

- source：String类型，要替换的字符串。
- pattern：String类型常量，要匹配的模式，pattern为空串时抛异常。
- replace_string：String类型，将匹配的pattern替换成的字符串。
- occurrence：Bigint类型常量，必须大于等于0，表示将第几次匹配替换成replace_string，为0时表示替换掉所有的匹配子串。其它类型或小于0抛异常。可缺省，默认值为0。

返回值：String类型，当引用不存在的组时，不进行替换。当输入source，pattern，occurrence参数为NULL时返回NULL，若replace_string为NULL且pattern有匹配，返回NULL，replace_string为NULL但pattern不匹配，则返回原串 。

备注：

- 当引用不存在的组时，行为未定义。

示例：

```
regexp_replace("123.456.7890", "([[:digit:]]{3})\\.([[:digit:]]{3})\\.([[:digit:]]{4})",
"(\\1)\\2-\\3", 0) = "(123)456-7890"
regexp_replace("abcd", "(.)", "\\1 ", 0) = "a b c d "
regexp_replace("abcd", "(.)", "\\1 ", 1) = "a bcd"
regexp_replace("abcd", "(.)", "\\2", 1) = "abcd"
-- 因为pattern中只定义了一个组，引用的第二个组不存在，
-- 请避免这样使用，引用不存在的组的结果未定义。
regexp_replace("abcd", "(.*)(.)$", "\\2", 0) = "d"
regexp_replace("abcd", "a", "\\1", 0) = "bcd"
```

```
-- 因为在pattern中没有组的定义，所以\1引用了不存在的组，
-- 请避免这样使用，引用不存在的组的结果未定义。
```

# REGEXP_SUBSTR

函数声明：

```
string regexp_substr(string source, string pattern[, bigint start_position[, bigint nth_occurrence]])
```

用途：从start_position位置开始，source中第nth_occurrence次匹配指定模式pattern的子串。

参数说明：

- source：String类型，搜索的字符串。
- pattern：String类型常量，要匹配的模型，pattern为空串时抛异常。
- start_position：Bigint常量，必须大于0。其它类型或小于等于0时抛异常，不指定时默认为1，表示从source的第一个字符开始匹配。不指定时默认为1，表示从source的第一个字符开始匹配。
- nth_occurrence：Bigint常量，必须大于0，其它类型或小于等于0时抛异常。不指定时默认为1，表示返回第一次匹配的子串。不指定时默认为1，表示返回第一次匹配的子串。

返回值：String类型。任一输入参数为NULL返回NULL。没有匹配时返回NULL。

示例：

```
regexp_substr ("I love aliyun very much", "a[[:alpha:]]{5}") = "aliyun"
regexp_substr('I have 2 apples and 100 bucks!', '[[:blank:]][[:alnum:]]*', 1, 1) = " have"
regexp_substr('I have 2 apples and 100 bucks!', '[[:blank:]][[:alnum:]]*', 1, 2) = " 2"
```

# REGEXP_COUNT

函数声明：

```
bigint regexp_count(string source, string pattern[, bigint start_position])
```

用途：计算source中从start_position开始，匹配指定模式pattern的子串的次数。

参数说明：

- source：String类型，搜索的字符串，其它类型报异常。
- pattern：String类型常量，要匹配的模型，pattern为空串时抛异常，其它类型报异常。
- start_position：Bigint类型常量，必须大于0。其它类型或小于等于0时抛异常，不指定时默认为1，表示从source的第一个字符开始匹配。

返回值：Bigint类型。没有匹配时返回0。任一输入参数为NULL返回NULL。

示例：

```
    regexp_count('abababc', 'a.c') = 1
regexp_count('abcde', '[[:alpha:]]{2}', 3) = 1
```

# SPLIT_PART

函数声明：

```
    string split_part(string str, string separator, bigint start[, bigint end])
```

用途：依照分隔符separator拆分字符串str，返回从第start部分到第end部分的子串(闭区间)。

参数说明：

- Str：String类型，要拆分的字符串。如果是bigint，double，decimal或者datetime类型会隐式转换到string类型后参加运算，其它类型报异常。
- Separator：String类型常量，拆分用的分隔符，可以是一个字符，也可以是一个字符串，其它类型会引发异常。
- start：Bigint类型常量，必须大于0。非常量或其它类型抛异常。返回段的开始编号(从1开始)，如果没有指定end，则返回start指定的段。
- end：Bigint类型常量，大于等于start，否则抛异常。返回段的截止编号，非常量或其他类型会引发异常。可省略，缺省时表示最后一部分。

返回值：String类型。若任意参数为NULL，返回NULL；若separator为空串，返回原字符串str。

备注:

- 如果separator不存在于str中，且start指定为1，返回整个str。若输入为空串，输出为空串。
- 如果start的值大于切分后实际的分段数，例如：字符串拆分完有6个片段，但start大于6，返回空串”“。
- 若end大于片段个数，按片段个数处理。

示例：

```
    split_part('a,b,c,d', ',', 1) = 'a'
split_part('a,b,c,d', ',', 1, 2) = 'a,b'
split_part('a,b,c,d', ',', 10) = ''
```

# SUBSTR

函数声明：

```
    string substr(string str, bigint start_position[, bigint length])
```

用途：返回字符串str从start_position开始往后数，长度为length的子串。

参数说明：

- str：String类型，若输入为bigint，decimal，double或者datetime类型会隐式转换为string后参与运算，其它类型报异常。
- start_position：Bigint类型，起始位置为1。当start_position为负数时表示开始位置是从字符串的结尾往前倒数，最后一个字符是-1，往前数依次就是-2，-3...，其它类型抛异常。
- length：Bigint类型，大于0，其它类型或小于等于0抛异常。子串的长度。

返回值：String类型。若任一输入为NULL，返回NULL。

备注：

- 当length被省略时，返回到str结尾的子串。

示例

```
    substr("abc", 2) = "bc"
substr("abc", 2, 1) = "b"
substr("abc",-2,2")="bc"
substr("abc",-3)="abc"
```

# TOLOWER

函数声明：

```
    string tolower(string source)
```

用途：输出英文字符串source对应的小写字符串。

参数说明：

- source：String类型，若输入为bigint，double，decimal或者datetime类型会隐式转换为string后参与运算，其它类型报异常。

返回值：String类型。输入为NULL时返回NULL。

示例：

```
    tolower("aBcd") = "abcd"
tolower("哈哈Cd") = "哈哈cd"
```

# TOUPPER

函数声明：

```
string toupper(string source)
```

用途：输出英文字符source串对应的大写字符串。

参数说明：

- source：String类型，若输入为bigint，double，decimal或者datetime类型会隐式转换为string后参与运算，其它类型报异常。

返回值：String类型。输入为NULL时返回NULL。

示例：

```
toupper("aBcd") = "ABCD"
toupper("哈哈Cd") = "哈哈CD"
```

# TO_CHAR

函数声明：

```
string to_char(boolean value)
string to_char(bigint value)
string to_char(double value)
string to_char(decimal value)
```

用途：将Boolean类型、bigint类型、decimal类型或者double类型转为对应的string类型表示

参数说明：

- value：可以接受boolean类型、bigint类型、decimal类型或者double类型输入，其它类型抛异常。对datetime类型的格式化输出请参考另一同名函数 TO_CHAR 。

返回值：String类型。如果输入为NULL，返回NULL。

示例：

```
to_char(123) = '123'
to_char(true) = 'TRUE'
to_char(1.23) = '1.23'
to_char(null) = NULL
```

# TRIM

函数声明：

```
string trim(string str)
```

用途：将输入字符串str去除左右空格。

参数说明：

- str：String类型，若输入为bigint，decimal，double或者datetime类型会隐式转换为string后参与运算，其它类型报异常。

返回值：String类型。输入为NULL时返回NULL。

# LTRIM

函数声明：

```
string ltrim(string str)
```

用途：将输入的字符串str去除左边空格。

参数说明：

- str：String类型，若输入为bigint，decimal，double或者datetime类型会隐式转换为string后参与运算，其它类型报异常。返回值：String类型。输入为NULL时返回NULL。

示例:

```
select ltrim(' abc ') from dual;
返回：
+-----+
| _c0 |
+-----+
| abc |
+-----+
```

# RTRIM

函数声明：

```
string rtrim(string str)
```

用途：将输入的字符串str去除右边空格。

参数说明：

- str：String类型，若输入为bigint，decimal，double或者datetime类型会隐式转换为string后参与运算，其它类型报异常。返回值：String类型。输入为NULL时返回NULL。

示例:

```
select rtrim('a abc ') from dual;
返回：
+-----+
| _c0 |
+-----+
| a abc |
+-----+
```

# REVERSE

函数申明：

```
 STRING REVERSE(string str)
```

用途：返回倒序字符串。参数说明：

- str：String类型，若输入为bigint，double，decimal或datetime类型会隐式转换为string后参与运算，其它类型报异常。返回值：String类型。输入为NULL时返回NULL。

示例：

```
select reverse('abcedfg') from dual;

返回：
+-----+
| _c0 |
+-----+
| gfdecba |
+-----+
```

# SPACE

函数声明：

```
STRING SPACE(bigint n)
```

用途：空格字符串函数，返回长度为n的字符串。参数说明：

- n: bigint类型。长度不超过2M。如果为空，则抛异常。返回值：String类型。

示例：

```
select length(space(10)) from dual;    ----返回10。
```

```
select space(400000000000) from dual； ----报错，长度超过2M。
```

# REPEAT

函数声明：

```
STRING REPEAT(string str, bigint n)
```

用途：返回重复n次后的str字符串。参数说明：

- str：String类型，若输入为bigint，double，decimal或datetime类型会隐式转换为string后参与运算，其它类型报异常。
- n: Bigint类型。长度不超过2M。如果为空，则抛异常。返回值：String类型。

示例：

```
 select repeat('abc',5) from lxw_dual;
返回：abcabcabcabcabc
```

# ASCII

函数声明：

```
Bigint ASCII(string str)
```

用途：返回字符串str第一个字符的ascii码。参数说明：

- str：String类型，若输入为bigint，double，decimal或datetime类型会隐式转换为string后参与运算，其它类型报异常。返回值：Bigint类型。

示例：

```
select ascii('abcde') from dual;
返回值：97
```

聚合函数，其输入与输出是多对一的关系，即将多条输入记录聚合成一条输出值。可以与SQL中的group by语句联用。

# COUNT

函数声明：

```
```

```
bigint count([distinct|all] value)
```

用途：计算记录数。

参数说明：

- distinct|all：指明在计数时是否去除重复记录，默认是all，即计算全部记录，如果指定distinct，则可以只计算唯一值数量。
- value：可以为任意类型，当value值为NULL时，该行不参与计算，value可以为，*当count()*时，返回所有行数。

返回值：Bigint类型。

示例：

```
-- 如表tbla有列col1类型为bigint
+------+
| COL1 |
+------+
| 1 |
+------+
| 2 |
+------+
| NULL |
+------+

select count(*) from tbla; -- 值为3,
select count(col1) from tbla; -- 值为2
```

聚合函数可以和group by一同使用，例如：假设存在表test_src，存在如下两列：key string类型，value double类型，

```
-- test_src的数据为

+-----+-------+
| key | value |
+-----+-------+
| a | 2.0 |
+-----+-------+
| a | 4.0 |
+-----+-------+
| b | 1.0 |
+-----+-------+
| b | 3.0 |
+-----+-------+

-- 此时执行如下语句，结果为：

select key, count(value) as count from test_src group by key;


+-----+-------+
```

```
| key | count |
+-----+-------+
| a | 2 |
+-----+-------+
| b | 2 |
+-----+-------+

-- 聚合函数将对相同key值得value值做聚合计算。下面介绍的其他聚合函数使用方法均与此例相同，不一一举例。
```

# AVG

函数声明：

```
    double avg(double value)
decimal avg(decimal value)
```

用途：计算平均值。

参数说明：

- value：Double类型或Decimal类型，若输入为string或bigint会隐式转换到double类型后参与运算，其它类型抛异常。当value值为NULL时，该行不参与计算。Boolean类型不允许参与计算。

返回值：若输入Decimal类型，返回Decimal类型，其他合法输入类型返回Double类型。

示例：

```
    -- 如表tbla有一列value，类型为bigint

+-------+
| value |
+-------+
| 1 |
| 2 |
| NULL |
+-------+

-- 则对该列计算avg结果为(1+2)/2=1.5

select avg(value) as avg from tbla;

+------+
| avg |
+------+
| 1.5 |
+------+
```

# MAX

函数声明：

```
max(value)
```

用途：计算最大值。

参数说明：

- value：可以为任意类型，当列中的值为NULL时，该行不参与计算。Boolean类型不允许参与运算。

返回值：与value类型相同。

示例：

```
-- 如表tbla有一列col1，类型为bigint
+------+
| col1 |
+------+
| 1 |
+------+
| 2 |
+------+
| NULL |
+------+

select max(value) from tbla; -- 返回值为2
```

# MIN

命令格式：

```
MIN(value)
```

用途：计算最小值。

参数说明：

- value：可以为任意类型，当列中的值为NULL时，该行不参与计算。Boolean类型不允许参与计算。

示例：

```
-- 如表tbla有一列value，类型为bigint
+------+
| value|
+------+
| 1 |
+------+
| 2 |
+------+
```

```
| NULL |
+------+
select min(value) from tbla; -- 返回值为1
```

# MEDIAN

函数声明：

```
    double median(double number)
decimal median(decimal number)
```

用途：计算中位数。

参数说明：

- number：Double类型或者Decimal类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。当输入值为NULL时忽略。

返回值：Double类型或者Decimal类型。

# STDDEV

函数声明：

```
    double stddev(double number)
decimal stddev(decimal number)
```

用途：计算总体标准差。

参数说明：

- number：Double类型或者Decimal类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。当输入值为NULL时忽略。

返回值：Double类型或者Decimal类型。

# STDDEV_SAMP

命令格式：

```
    double stddev_samp(double number)
decimal stddev_samp(decimal number)
```

用途：计算样本标准差。

参数说明：

- number：Double类型或者Decimal类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。当输入值为NULL时忽略。

返回值：Double类型或者Decimal类型。

# SUM

函数声明：

```
sum(value)
```

用途：计算汇总值。

参数说明：

- value：Double、decimal或bigint类型，若输入为string会隐式转换到double类型后参与运算，当列中的值为NULL时，该行不参与计算。Boolean类型不允许参与计算。

返回值：输入为bigint时返回bigint，输入为double或string时返回double类型。

示例：

```
-- 如表tbla有一列value，类型为bigint
+------+
| value|
+------+
| 1 |
+------+
| 2 |
+------+
| NULL |
+------+
select sum(value) from tbla; -- 返回值为3
```

# WM_CONCAT

函数声明：

```
string wm_concat(string separator, string str)
```

用途：用指定的spearator做分隔符，链接str中的值。

参数说明：

- separator：String类型常量，分隔符。其他类型或非常量将引发异常。
- str：String类型，若输入为bigint，double或者datetime类型会隐式转换为string后参与运算，其它类型报异常。

返回值：String类型。

备注:

- 对语句”select wm_concat( ‘,’ , name) from test_src;” ，若test_src为空集合，这MaxCompute SQL条语句返回NULL值。聚合函数，其输入与输出是多对一的关系，即将多条输入记录聚合成一条输出值。可以与SQL中的group by语句联用。

# CAST

函数声明：

```
cast(expr as <type>)
```

用途：将表达式的结果转换成目标类型，如cast( ‘1’ as bigint)将字符串”1”转为整数类型的1，如果转换不成功或不支持的类型转换会引发异常。

备注:

- cast(double as bigint)，将double值转换成bigint。
- cast(string as bigint) 在将字符串转为bigint时，如果字符串中是以整型表达的数字，会直接转为bigint类型。如果字符串中是以浮点数或指数形式表达的数字，则会先转为double类型，再转为bigint类型。
- cast(string as datetime) 或 cast(datetime as string)时，会采用默认的日期格式yyyy-mm-dd hh:mi:ss。

# COALESCE

函数声明：

```
coalesce(expr1, expr2, ...)
```

用途：返回列表中第一个非NULL的值，如果列表中所有的值都是NULL则返回NULL。

参数说明：

- expri是要测试的值。所有这些值类型必须相同或为NULL，否则会引发异常。

返回值：返回值类型和参数类型相同。

备注:

- 参数至少要有一个，否则引发异常。

# DECODE

函数声明：

```
decode(expression, search, result[, search, result]...[, default])
```

用途：实现if-then-else分支选择的功能。

参数说明：

- expression：要比较的表达式。
- search：和expression进行比较的搜索项。
- result：search和expression的值匹配时的返回值。
- default：可选项，如果所有的搜索项都不匹配，则返回此default值，如果未指定，则会返回NULL。

返回值：返回匹配的search；如果没有匹配，返回default；如果没有指定default，返回NULL。

备注:

- 至少要指定三个参数。
- 所有的result类型必须一致，或为NULL。不一致的数据类型会引发异常。所有的search和expression类型必须一致，否则报异常。
- 如果decode中的search选项有重复时且匹配时，会返回第一个值。

示例：

```
select
decode(customer_id,
1, 'Taobao',
2, 'Alipay',
3, 'Aliyun',
NULL, 'N/A',
'Others') as result
from sale_detail;
```

上面的decode函数实现了下面if-then-else语句中的功能：

```
if customer_id = 1 then
result := 'Taobao';
elsif customer_id = 2 then
result := 'Alipay';
elsif customer_id = 3 then
result := 'Aliyun';
...
else
```

```
result := 'Others';
end if;
```

但需要用户注意的是，通常情况下MaxCompute SQL在计算NULL = NULL时返回NULL，但在decode函数中，NULL与NULL的值是相等的。在上述事例中，当customer_id的值为NULL时，decode函数返回”N/A”。

# GREATEST

函数声明：

```
greatest(var1, var2, ...)
```

用途：返回输入参数中最大的一个。

参数说明：

- var1，var2可以为bigint，double，decimal，datetime或者string。若所有值都为NULL则返回NULL。

返回值:

- 输入参数中的最大值，当不存在隐式转换时返回同输入参数类型。
- NULL为最小值。
- 当输入参数类型不同时，double，bigint，decimal，string之间的比较转为double；string，datetime的比较转为datetime。不允许其它的隐式转换。

# ORDINAL

函数声明：

```
ordinal(bigint nth, var1, var2, ...)
```

用途：将输入变量按从小到大排序后，返回nth指定的位置的值。

参数说明：

- nth：Bigint类型，指定要返回的位置，为NULL时返回NULL。
- var1，var2：类型可以为bigint，double，datetime或者string。

返回值：

- 排在第nth位的值，当不存在隐式转换时返回同输入参数类型。
- 有类型转换时，double，bigint，string之间的转换返回double。string，datetime之间的转换返回datetime。不允许其它的隐式转换。
- NULL为最小。

示例：

```
ordinal(3, 1, 3, 2, 5, 2, 4, 6) = 2
```

# LEAST

函数声明：

```
least(var1, var2, ...)
```

用途：返回输入参数中最小的一个。

参数说明：

- var1，var2可以为bigint，double，decimal，datetime或者string。若所有值都为NULL则返回NULL。

返回值：

- 输入参数中的最小值，当不存在隐式转换时返回同输入参数类型。
- NULL为最小。
- 有类型转换时，double, bigint, string之间的转换返回double。string, datetime之间的转换返回datetime; decimal和double,bigint,string之间比较时转为decimal。不允许其它的隐式类型转换。

# UUID

命令格式：

```
string uuid()
```

用途：返回一个随机ID，形式示例：” 29347a88-1e57-41ae-bb68-a9edbdd94212” 。

# SAMPLE

函数声明：

```
boolean sample(x, y, column_name)
```

用途：对所有读入的column_name的值，sample根据x，y的设置做采样，并过滤掉不满足采样条件的行。

参数说明：

- x，y：Bigint类型，表示哈希为x份，取第y份。y可省略，省略时取第一份，如果省略参数中的y，则

必须同时省略column_name。x，y为整型常量，大于0，其它类型或小于等于0时抛异常，若y>x也抛异常。x，y任一输入为NULL时返回NULL。

- column_name是采样的目标列。column_name可以省略，省略时根据x，y的值随机采样。任意类型，列的值可以为NULL。不做隐式类型转换。如果column_name为常量NULL会报异常。

返回值：Boolean类型。

备注:

- 为了避免NULL值带来的数据倾斜，因此对于column_name中为NULL的值，会在x份中进行均匀哈希。如果不加column_name，则数据量比较少时输出不一定均匀，在这种情况下建议加上column_name，以获得比较好的输出结果。

示例：假定存在表tbla，表内有列名为cola的列，

```
    select * from tbla where sample (4, 1 , cola) = true;
-- 表示数值会根据cola hash为4份，取第1份
select * from tbla where sample (4, 2) = true;
-- 表示数值会对每行数据做随机哈希分配为4份，取第2份
```

# CASE WHEN表达式

MaxCompute提供两种case when语法格式，如下所述：

```
    case value
when (_condition1) then result1
when (_condition2) then result2
...
else resultn
end

case
when (_condition1) then result1
when (_condition2) then result2
when (_condition3) then result3
...
else resultn
end
```

case when表达式可以根据表达式value的计算结果灵活返回不同的值， 如以下语句根据shop_name的不同情况得出所属区域

```
    select
case
when shop_name is null then 'default_region'
when shop_name like 'hang%' then 'zj_region'
end as region
```

```
from sale_detail;
```

说明：

- 如果result类型只有bigint，double，统一转double再返回；
- 如果result类型中有string类型，统一转string再返回，如果不能转则报错(如boolean型)；
- 除此之外不允许其它类型之间的转换；

## IF

函数声明：

```
if(testCondition, valueTrue, valueFalseOrNull)
```

用途：判断testCondition是否为真，如果为真,返回valueTrue，如果不满足则返回另一个值(valueFalse或者Null)。

参数说明：

- testCondition:要判断的表达式， boolean类型；
- valueTrue: 表达式testCondition为True的时候，返回的值。
- valueFalseOrNull：不满足表达式testCondition时2，返回的值，可以设为Null.返回值：返回值类型和参数valueTrue或者valueFalseOrNul的类型一致。示例：

```
select if(1=2,100,200) from dual;
返回值：
+------------+
| _c0 |
+------------+
| 200 |
+------------+
```

# UDF

UDF 全称 User Defined Function，即用户自定义函数 。MaxCompute 提供了很多内建函数来满足用户的计算需求，同时用户还可以通过创建自定义函数来满足不同的计算需求 。UDF 在使用上与普通的 内建函数 类似，Java 和 MaxCompute 的数据类型对应关系，请参见：参数与返回值类型 。

使用 Maven 的用户可以从 Maven 库 中搜索”odps-sdk-udf”获取不同版本的 Java SDK，相关配置信息如下：

```
<dependency>
```

```
<groupId>com.aliyun.odps</groupId>
<artifactId>odps-sdk-udf</artifactId>
<version>0.20.7-public</version>
</dependency>
```

在 MaxCompute 中，用户可以扩展的 UDF 有两种：

| UDF 分类 | 描述 |
| --- | --- |
| User Defined Scalar Function，通常也称之为 UDF | 用户自定义标量值函数(User Defined Scalar Function)通常也称之为 UDF 。其输入与输出是一对一的关系，即读入一行数据，写出一条输出值 。 |
| UDTF(User Defined Table Valued Function) | 自定义表值函数，是用来解决一次函数调用输出多行数据场景的，也是唯一能返回多个字段的自定义函数。而 UDF 只能一次计算输出一条返回值。 |
| UDAF(User Defined Aggregation Function) | 自定义聚合函数，其输入与输出是多对一的关系，即将多条输入记录聚合成一条输出值。可以与 SQL 中的 Group By 语句联用。具体语法请参见： 聚合函数 。 |

备注:

- UDF 广义的说法代表了自定义标量函数，自定义聚合函数及自定义表函数三种类型的自定义函数的集合。狭义来说，仅代表用户自定义标量函数。文档会经常使用这一名词，请读者根据文档上下文判断具体含义 。
- SQL 语句中有使用自定义的函数，提示内存不够。请配置 set odps.sql.udf.joiner.jvm.memory=xxxx；原因是数据量太大并且有倾斜，任务超出默认设置的内存 。

# UDF示例

请参见快速入门分册 UDF 示例 。

MaxCompute 的 UDF 包括：UDF，UDAF，UDTF 三种函数，本文将重点介绍如何通过 JAVA 实现这 3 种函数 。

首先总结 Java 和 MaxCompute 数据类型对应关系，即参数与返回值类型表格所示：

## 参数与返回值类型

UDF 支持 MaxCompute SQL 的数据类型有：Bigint, String, Double, Boolean 类型 。MaxCompute 数据类型与 Java 类型的对应关系如下：

| MaxComput | Bigint | String | Double | Boolean | Decimal |
| --- | --- | --- | --- | --- | --- |

| e SQL Type | | | | | |
|---|---|---|---|---|---|
| Java Type | Long | String | Double | Boolean | BigDecimal |

**注意：**

java 中对应的数据类型以及返回值数据类型是对象，首字母请务必大写；

目前暂不支持 datetime 数据类型，建议可以转换成 String 类型传入处理 。

SQL 中的 NULL 值通过 Java 中的 NULL 引用表示，因此 Java primitive type 是不允许使用的，因为无法表示 SQL 中的 NULL 值 。

# UDF

实现 UDF 需要继承 com.aliyun.odps.udf.UDF 类，并实现 evaluate 方法。evaluate 方法必须是非 static的 public 方法 。Evaluate 方法的参数和返回值类型将作为 SQL 中 UDF 的函数签名 。这意味着用户可以在 UDF 中实现多个 evaluate 方法，在调用 UDF 时，框架会依据 UDF 调用的参数类型匹配正确的 evaluate 方法 。

下面是一个 UDF 的例子:

```
    package org.alidata.odps.udf.examples;
import com.aliyun.odps.udf.UDF;

public final class Lower extends UDF {
public String evaluate(String s) {
if (s == null) { return null; }
return s.toLowerCase();
}
}
```

可以通过实现void setup(ExecutionContext ctx)和void close()来分别实现 UDF 的初始化和结束代码 。

UDF的使用方式于 ODPS SQL 中普通的内建函数相同，详情请参见： 内建函数 。

# UDAF

实现 Java UDAF 类需要继承 com.aliyun.odps.udf.Aggregator，并实现如下几个接口：

```
public abstract class Aggregator implements ContextFunction {

@Override
public void setup(ExecutionContext ctx) throws UDFException {
}

@Override
```

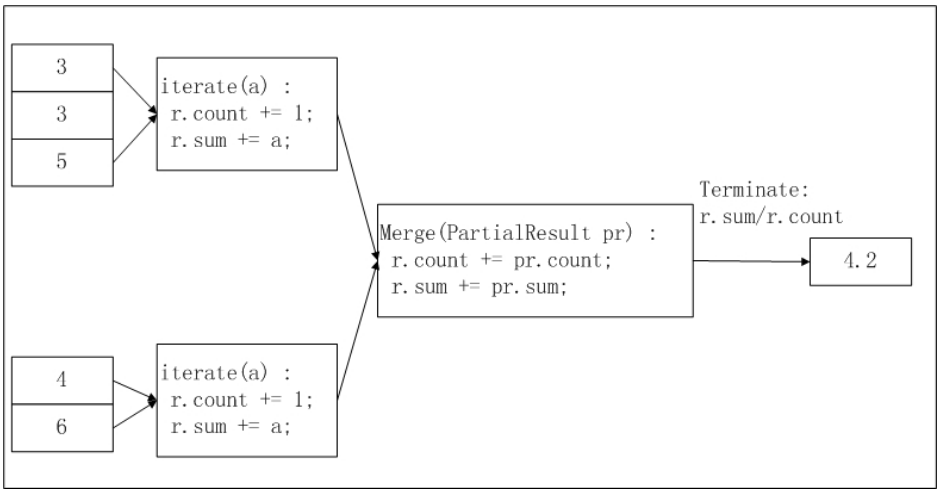
```
public void close() throws UDFException {
}

/**
* 创建聚合Buffer
* @return Writable 聚合buffer
*/
abstract public Writable newBuffer();
/**
* @param buffer 聚合buffer
* @param args SQL中调用UDAF时指定的参数
* @throws UDFException
*/
abstract public void iterate(Writable buffer, Writable[] args) throws UDFException;

/**
* 生成最终结果
* @param buffer
* @return Object UDAF的最终结果
* @throws UDFException
*/
abstract public Writable terminate(Writable buffer) throws UDFException;

abstract public void merge(Writable buffer, Writable partial) throws UDFException;
}
```

其中最重要的是 iterate，merge 和 terminate 三个接口，UDAF 的主要逻辑依赖于这三个接口的实现 。此外，还需要用户实现自定义的 Writable buffer 。以实现求平均值 avg 为例，下图简要说明了在MaxCompute UDAF 中这一函数的实现逻辑及计算流程：




在上图中，输入数据被按照一定的大小进行分片(有关分片的描述请参见 MapReduce )，每片的大小适合一个 worker 在适当的时间内完成 。这个分片大小的设置需要用户手动配置完成 。UDAF 的计算过程分为两阶段：

- 在第一阶段，每个 worker 统计分片内数据的个数及汇总值，我们可以将每个分片内的数据个数及汇总值视为一个中间结果；
- 在第二阶段，worker 汇总上一个阶段中每个分片内的信息 。在最终输出时，r.sum / r.count 即是所有输入数据的平均值；

下面是一个计算平均值的 UDAF 的代码示例:

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import com.aliyun.odps.io.DoubleWritable;
import com.aliyun.odps.io.Writable;
import com.aliyun.odps.udf.Aggregator;
import com.aliyun.odps.udf.UDFException;
import com.aliyun.odps.udf.annotation.Resolve;

@Resolve({"double->double"})
public class AggrAvg extends Aggregator {

private static class AvgBuffer implements Writable {

private double sum = 0;
private long count = 0;

@Override
public void write(DataOutput out) throws IOException {
out.writeDouble(sum);
out.writeLong(count);
}
@Override
public void readFields(DataInput in) throws IOException {
sum = in.readDouble();
count = in.readLong();
}
}

private DoubleWritable ret = new DoubleWritable();

@Override
public Writable newBuffer() {
return new AvgBuffer();
}

@Override
public void iterate(Writable buffer, Writable[] args) throws UDFException {
DoubleWritable arg = (DoubleWritable) args[0];
AvgBuffer buf = (AvgBuffer) buffer;
if (arg != null) {
buf.count += 1;
buf.sum += arg.get();
}
}

@Override
public Writable terminate(Writable buffer) throws UDFException {
AvgBuffer buf = (AvgBuffer) buffer;
if (buf.count == 0) {
ret.set(0);
```

```
} else {
ret.set(buf.sum / buf.count);
}
return ret;
}

@Override
public void merge(Writable buffer, Writable partial) throws UDFException {
AvgBuffer buf = (AvgBuffer) buffer;
AvgBuffer p = (AvgBuffer) partial;
buf.sum += p.sum;
buf.count += p.count;
}

}
```

注意：

- UDAF 在 SQL 中的使用语法与普通的内建聚合函数相同，详情请参见 聚合函数 。
- 关于如何运行 UDTF,方法与 UDF 类似，具体步骤请参见 运行 UDF 。

# UDTF

Java UDTF 需要继承 com.aliyun.odps.udf.UDTF 类 。这个类需要实现 4 个接口 ,如下表所示：

| 接口定义 | 描述 |
| --- | --- |
| public void setup(ExecutionContext ctx) throws UDFException | 初始化方法，在UDTF处理输入数据前，调用用户自定义的初始化行为。在每个Worker内setup会被先调用一次。 |
| public void process(Object[] args) throws UDFException | 这个方法由框架调用，SQL中每一条记录都会对应调用一次process，process的参数为SQL语句中指定的UDTF输入参数。输入参数以Object[]的形式传入，输出结果通过forward函数输出。用户需要在process函数内自行调用forward，以决定输出数据。 |
| public void close() throws UDFException | UDTF的结束方法，此方法由框架调用，并且只会被调用一次，即在处理完最后一条记录之后。 |
| public void forward(Object ...o) throws UDFException | 用户调用forward方法输出数据，每次forward代表输出一条记录。对应SQL语句UDTF的as子句指定的列。 |

下面将给出一个 UDTF 程序示例：

```
package org.alidata.odps.udtf.examples;

import com.aliyun.odps.udf.UDTF;
import com.aliyun.odps.udf.UDTFCollector;
import com.aliyun.odps.udf.annotation.Resolve;
```

```
import com.aliyun.odps.udf.UDFException;

// TODO define input and output types, e.g., "string,string->string,bigint".
@Resolve({"string,bigint->string,bigint"})
public class MyUDTF extends UDTF {

@Override
public void process(Object[] args) throws UDFException {
String a = (String) args[0];
Long b = (Long) args[1];

for (String t: a.split("\\s+")) {
forward(t, b);
}
}
}
```

注：

- 以上只是程序示例，关于如何在 MaxCompute 中运行 UDTF,方法与 UDF 类似，具体实现步骤请参见：运行 UDF 。

在 SQL 中可以这样使用这个 UDTF，假设在 MaxCompute 上创建 UDTF 时注册函数名为 user_udtf：

```
select user_udtf(col0, col1) as (c0, c1) from my_table;
```

假设 my_table 的 col0， col1 的值为：

```
+------+------+
| col0 | col1 |
+------+------+
| A B | 1 |
| C D | 2 |
+------+------+
```

则 select 出的结果为：

```
+----+----+
| c0 | c1 |
+----+----+
| A | 1 |
| B | 1 |
| C | 2 |
| D | 2 |
+----+----+
```

## 使用说明

UDTF 在 SQL 中的常用方式如下：

```
select user_udtf(col0, col1, col2) as (c0, c1) from my_table;
select user_udtf(col0, col1, col2) as (c0, c1) from
(select * from my_table distribute by key sort by key) t;
select reduce_udtf(col0, col1, col2) as (c0, c1) from
(select col0, col1, col2 from
(select map_udtf(a0, a1, a2, a3) as (col0, col1, col2) from my_table) t1
distribute by col0 sort by col0, col1) t2;
```

但使用 UDTF 有如下使用限制：

- 同一个 SELECT 子句中不允许有其他表达式

```
select value, user_udtf(key) as mycol ...
```

- UDTF 不能嵌套使用

```
select user_udtf1(user_udtf2(key)) as mycol...
```

- 不支持在同一个 select 子句中与 group by / distribute by / sort by 联用

```
select user_udtf(key) as mycol ... group by mycol
```

## 其他 UDTF 示例

在 UDTF 中，用户可以读取 MaxCompute 的 资源 。下面将介绍利用 UDTF 读取 MaxCompute 资源的示例：

编写 UDTF 程序，编译成功后导出 jar 包(udtfexample1.jar)。

```
package com.aliyun.odps.examples.udf;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Iterator;

import com.aliyun.odps.udf.ExecutionContext;
import com.aliyun.odps.udf.UDFException;
import com.aliyun.odps.udf.UDTF;
import com.aliyun.odps.udf.annotation.Resolve;

/**
* project: example_project
* table: wc_in2
* partitions: p2=1,p1=2
* columns: colc,colb
```

```
*/
@Resolve({ "string,string->string,bigint,string" })
public class UDTFResource extends UDTF {
ExecutionContext ctx;
long fileResourceLineCount;
long tableResource1RecordCount;
long tableResource2RecordCount;

@Override
public void setup(ExecutionContext ctx) throws UDFException {
this.ctx = ctx;
try {
InputStream in = ctx.readResourceFileAsStream("file_resource.txt");
BufferedReader br = new BufferedReader(new InputStreamReader(in));
String line;
fileResourceLineCount = 0;
while ((line = br.readLine()) != null) {
fileResourceLineCount++;
}
br.close();

Iterator<Object[]> iterator = ctx.readResourceTable("table_resource1").iterator();
tableResource1RecordCount = 0;
while (iterator.hasNext()) {
tableResource1RecordCount++;
iterator.next();
}

iterator = ctx.readResourceTable("table_resource2").iterator();
tableResource2RecordCount = 0;
while (iterator.hasNext()) {
tableResource2RecordCount++;
iterator.next();
}

} catch (IOException e) {
throw new UDFException(e);
}
}

@Override
public void process(Object[] args) throws UDFException {
String a = (String) args[0];
long b = args[1] == null ? 0 : ((String) args[1]).length();

forward(a, b, "fileResourceLineCount=" + fileResourceLineCount + "|tableResource1RecordCount="
+ tableResource1RecordCount + "|tableResource2RecordCount=" + tableResource2RecordCount);

}
}
```

添加资源到 MaxCompute：

```
Add file file_resource.txt;
Add jar udtfexample1.jar;
```

```
Add table table_resource1 as table_resource1;
Add table table_resource2 as table_resource2;
```

在 MaxCompute 中创建 UDTF 函数(my_udtf)：

```
create function mp_udtf as com.aliyun.odps.examples.udf.UDTFResource using 'udtfexample1.jar,
file_resource.txt, table_resource1, table_resource2';
```

在 MaxCompute 创建资源表 table_resource1、table_resource2, 物理表 tmp1，并插入相应的数据。

运行该 UDTF：

```
select mp_udtf("10","20") as (a, b, fileResourceLineCount) from table_resource1;

返回：
+-------+------------+-------+
| a | b | fileResourceLineCount |
+-------+------------+-------+
| 10 | 2 | fileResourceLineCount=3|tableResource1RecordCount=0|tableResource2RecordCount=0 |
| 10 | 2 | fileResourceLineCount=3|tableResource1RecordCount=0|tableResource2RecordCount=0 |
+-------+------------+-------+
```

# 附录

在ODPS SQL中的字符串常量可以用单引号或双引号表示，可以在单引号括起的字符串中包含双引号，或在双引号括起的字符串中包含单引号， 否则要用转义符来表达，如以下表达方式都是可以的

```
"I'm a happy manong!"
'I\'m a happy manong!'
```

在ODPS SQL中反斜线”\”是转义符，用来表达字符串中的特殊字符，或将其后跟的字符解释为其本身。当读入字符串常量时， 如果反斜线后跟三位有效的8进制数字，范围在001 ~177之间，系统会根据ASCII值转为相应的字符。对于以下情况，则会将其解释为特殊字符：

| 转义 | 字符 |
| --- | --- |
| \b | backspace |
| \t | tab |
| \n | newline |

| | |
|---|---|
| \r | carriage-return |
| \’ | 单引号 |
| \” | 双引号 |
| \ \ | 反斜线 |
| \; | 分号 |
| \Z | control-Z |
| \0或\00 | 结束符 |

```
select length('a\tb') from dual;
```

结果是3，表示字符串里实际有三个字符,”\t”被视为一个字符。在转义符后的其它字符被解释为其本身。

```
select 'a\ab',length('a\ab') from dual;
```

结果是’ aab’，3。” \a”被解释成了普通的”a”。

在LIKE匹配时，” %”表示匹配任意多个字符，” _”表示匹配单个字符，如果要匹配” %”或” _”本身，则要对其进行转义, “\%”匹配字符” %” ，” _”匹配字符” _”。

```
'abcd' like 'ab%' -- true
'abcd' like 'ab\%' -- false
'ab%cd' like 'ab\\%%' -- true
```

备注：

- 关于字符串的字符集，目前ODPS SQL支持UTF-8的字符集，如果数据是以其它格式编码，可能计算出的结果不正确。

ODPS SQL中的正则表达式采用的是PCRE的规范，匹配时是按字符进行，支持的元字符如下：

```
元字符        说明
^             行首
$             行尾
.             任意字符
*             匹配零次或多次
+             匹配1次或多次
?             匹配零次或1次
?             匹配修饰符，当该字符紧跟在任何一个其他限制符 (* ,+, ?, {n}, {n,}, {n,m})后面时，匹配模式是非贪 婪的。
             非贪婪模式尽可能少的匹配所搜索的字符串， 而默认的贪婪模式则尽可能多的匹配所搜索的字符串。
A|B             A或B
(abc)*          匹配abc序列零次或多次
```

```
{n}或{m,n}       匹配的次数
[ab]           匹配括号中的任一字符,例中模式匹配a或b
[a-d]          匹配a,b,c,d任一字符
[^ab]          ^表示非，匹配任一非a非 b的字符
[::]           见下表POSIX字符组
\              转义符
\n             n为数字1-9，后向引用
\d             数字
\D             非数字
```

POSIX字符组

```
POSIX字符组       说明              范围
[[:alnum:]]      字母字符和数字字符    [a-zA-Z0-9]
[[:alpha:]]      字母              [a-zA-Z]
[[:ascii:]]      ASCII字符          [\x00-\x7F]
[[:blank:]]      空格字符和制表符    [ \t]
[[:cntrl:]]      控制字符           [\x00-\x1F\x7F]
[[:digit:]]      数字字符           [0-9]
[[:graph:]]      空白字符之外的字符    [\x21-\x7E]
[[:lower:]]      小写字母字符        [a-z]
[[:print:]]      [:graph:]和空白字符    [\x20-\x7E]
[[:punct:]]      标点符号           [][!"#$%&'()*+,./:;<=>? @\^_`{|}~-]
[[:space:]]      空白字符           [ \t\r\n\v\f]
[[:upper:]]      大写字母字符        [A-Z]
[[:xdigit:]]    十六进制字符        [A-Fa-f0-9]
```

由于系统采用反斜线"\"作为转义符，因此正则表达式的模式中出现的"\"都要进行二次转义。如正则表达式要匹配字符串"a+b"，其中"+"是正则中的一个特殊字符，因此要用转义的方式表达，在正则引擎中的表达方式是"a\+b"，由于系统还要解释一层转义，因此能够匹配该字符串的表达式是"a\\+b"。例如，假设存在表test_dual，

```
select 'a+b' rlike 'a\\\+b' from test_dual;

+------+
| _c1  |
+------+
| true |
+------+
```

极端的情况，如果在要匹配字符"\"，由于在正则引擎中"\"是一个特殊字符，因此要表示为"\"，而系统还要对表达式进行一次转义，因此写成"\\"

```
select 'a\\b', 'a\\b' rlike 'a\\\\b' from test_dual;

+-----+------+
| _c0 | _c1  |
+-----+------+
| a\b | true |
+-----+------+
```

备注:

- 在ODPS SQL中写”a\b” ， 而在输出结果中显示’ a\b’ ，同样是因为ODPS会对表达式进行转义。

如果字符串中有制表符TAB, 系统在读入’ \t’ 这两个字符的时候，即已经将其存为一个字符，因此在正则的模式中也是一个普通的字符。

```
select 'a\tb', 'a\tb' rlike 'a\tb' from test_dual;

+---------+------+
| _c0     | _c1  |
+---------+------+
| a     b | true |
+---------+------+
```

以下ODPS SQL的全部保留字，在对表、列或是分区命名时请不要使用，否则会报错。保留字不区分大小写。

```
  %   &   &&   (   )   *   +
- . / ; < <= <>
= > >= ? ADD AFTER ALL
ALTER ANALYZE AND ARCHIVE ARRAY AS ASC
BEFORE BETWEEN BIGINT BINARY BLOB BOOLEAN BOTH DECIMAL
BUCKET BUCKETS BY CASCADE CASE CAST CFILE
CHANGE CLUSTER CLUSTERED CLUSTERSTATUS COLLECTION COLUMN COLUMNS
COMMENT COMPUTE CONCATENATE CONTINUE CREATE CROSS CURRENT
CURSOR DATA DATABASE DATABASES DATE DATETIME DBPROPERTIES
DEFERRED DELETE DELIMITED DESC DESCRIBE DIRECTORY DISABLE
DISTINCT DISTRIBUTE DOUBLE DROP ELSE ENABLE END
ESCAPED EXCLUSIVE EXISTS EXPLAIN EXPORT EXTENDED EXTERNAL
FALSE FETCH FIELDS FILEFORMAT FIRST FLOAT FOLLOWING
FORMAT FORMATTED FROM FULL FUNCTION FUNCTIONS GRANT
GROUP HAVING HOLD_DDLTIME IDXPROPERTIES IF IMPORT IN
INDEX INDEXES INPATH INPUTDRIVER INPUTFORMAT INSERT INT
INTERSECT INTO IS ITEMS JOIN KEYS LATERAL
LEFT LIFECYCLE LIKE LIMIT LINES LOAD LOCAL
LOCATION LOCK LOCKS LONG MAP MAPJOIN MATERIALIZED
MINUS MSCK NOT NO_DROP NULL OF OFFLINE
ON OPTION OR ORDER OUT OUTER OUTPUTDRIVER
OUTPUTFORMAT OVER OVERWRITE PARTITION PARTITIONED PARTITIONPROPERTIES PARTITIONS
PERCENT PLUS PRECEDING PRESERVE PROCEDURE PURGE RANGE
RCFILE READ READONLY READS REBUILD RECORDREADER RECORDWRITER
REDUCE REGEXP RENAME REPAIR REPLACE RESTRICT REVOKE
RIGHT RLIKE ROW ROWS SCHEMA SCHEMAS SELECT
SEMI SEQUENCEFILE SERDE SERDEPROPERTIES SET SHARED SHOW
SHOW_DATABASE SMALLINT SORT SORTED SSL STATISTICS STORED
STREAMTABLE STRING STRUCT TABLE TABLES TABLESAMPLE TBLPROPERTIES
TEMPORARY TERMINATED TEXTFILE THEN TIMESTAMP TINYINT TO
TOUCH TRANSFORM TRIGGER TRUE UNARCHIVE UNBOUNDED UNDO
UNION UNIONTYPE UNIQUEJOIN UNLOCK UNSIGNED UPDATE USE
USING UTC UTC_TMESTAMP VIEW WHEN WHERE WHILE
```

一些用户因没注意限制条件，业务启动后才发现限制条件，导致业务停止。为避免此类现象发生，方便用户查看，本文将对 MaxCompute SQL 限制项做以下汇总：

| 边界名 | 最大值/限制条件 | 分类 | 配置项名称 | 说明 |
| --- | --- | --- | --- | --- |
| 表名长度 | 128字节 | 长度限制 | | 表名，列名中不能有特殊字符，只能用英文的a-z,A-Z及数字和下划线_，且以字母开头 |
| 注释长度 | 1024字节 | 长度限制 | | 注释内容是长度不超过1024字节的有效字符串 |
| 表的列定义 | 10000个 | 数量限制 | apsara.odps.meta.column.max | 单表的列定义个数最多10000个 |
| 单表分区数 | 60000 | 数量限制 | apsara.odps.metastore.ots.MaxPartitions | 一张表最多允许60000个分区 |
| 表的分区层级 | 6级 | 数量限制 | apsara.odps.meta.part.level.max | 在表中建的分区层次不能超过6级 |
| 表统计定义个数 | 100个 | 数量限制 | | 表统计定义个数 |
| 表统计定义长度 | 64000 | 长度限制 | apsara.odps.sql.stat.maxlen | 表统计项定义长度 |
| 屏显 | 10000行 | 数量限制 | | SELECT语句屏显默认最多输出10000行 |
| insert目标个数 | 256个 | 数量限制 | apsara.odps.sql.inserts.max | multiins同时insert的数据表数量 |
| UNION ALL | 256个表 | 数量限制 | apsara.odps.sql.unioners.max | 最多允许256个表的UNION ALL |
| join源 | 16个 | 数量限制 | apsara.odps.sql.maximum.join | join的源表个数最多运行16个 |
| MAPJOIN | 8个小表 | 数量限制 | | MAPJOIN最多允许8张小表 |
| MAPJOIN内存限制 | 512M | 数量限制 | apsara.odps.analytic.function.maxexprs | MAPJOIN所有小表的内存限制不能超过512M |
| 窗口函数 | 5个 | 数量限制 | | 一个SELECT中最多允许5个窗口函数 |
| ptinsubq | 1000行 | 数量限制 | | pt in subquery返回的结果不可超过 |

| | | | | |
|---|---|---|---|---|
| | | | | 1000行 |
| sql语句长度 | 2M | 长度限制 | | 允许的sql语句的最大长度 |
| wherer子句条件个数 | 256个 | 数量限制 | | where子句中可使用条件个数 |
| 列记录长度 | 8M | 数量限制 | | 表中一条记录的最大长度 |
| in的参数个数 | 1024 | 数量限制 | | in的最大参数限制，如in (1,2,3....,10240)。in(...)如果参数过多，会造成编译时的压力；1024是建议值、不是限制值 |
| jobconf.json | 1M | 长度限制 | | jobconf.json的大小为1M。当表包含的Partition数量太多时，可能超过jobconf.json超过1M。 |
| 视图 | 不可写 | 操作限制 | | 视图不可以写，不可用insert操作 |
| 列的数据类 | 不允许 | 操作限制 | | 不允许修改列的数据类型，列位置 |
| java udf函数 | 不能是abstract或者static | 操作限制 | | java udf函数不能是 abstract 或者 static |
| 最多查询分区个数 | 10000 | 数量限制 | | 最多查询分区个数不能超过10000 |

备注：以上MaxCompute SQL限制项均不可以被人为修改配置。

# 最佳实践

本文通过几个例子，介绍了几种下载MaxCompute SQL计算结果的方法。为了减少篇幅，所有的SDK部分都只举例介绍Java的例子。

**导出方式有：**

- 如果数据比较少，可以直接用**SQL Task**得到全部的查询结果。
- 如果只是想导出某个表或者分区，可以用**Tunnel**直接导出数据。
- 如果SQL比较复杂，需要Tunnel和SQL相互配合才行。
- **大数据开发套件**可以方便地帮我们运行SQL，**同步数据**，并有定时调度，配置任务依赖的功能。
- 开源工具**DataX**能帮助我们很方便把MaxCompute里的数据导出到目标数据源，具体请看**该文档**，本文不做具体介绍。

# SQLTask方式导出

SQL Task是SDK直接调用MaxCompute SQL的接口，能很方便得运行SQL并获得其返回结果。从文档可以看到，SQLTask.getResult(i); 返回的是一个List，可以循环迭代这个List，获得完整的SQL计算返回结果。不过这个方法有个缺陷，可以参考**这里**提到的SetProject READ_TABLE_MAX_ROW的功能。目前Select语句返回给客户端的数据条数最大可以调整到**1万**。也就是说如果在客户端上（包括SQLTask）直接Select，那相当于查询结果上最后加了个Limit N（如果是CREATE TABLE XX AS SELECT或者用INSERT INTO/OVERWRITE TABLE把结果固化到具体的表里就没关系）。

# Tunnel方式导出

如果需要导出的查询结果就是某张表的全部内容（或者是具体的某个分区的全部内容），可以用Tunnel来做。官网提供了**命令行工具**和基于SDK编写的**Tunnel SDK**。

在此提供一个Tunnel命令行导出数据的简单例子，Tunnel SDK的编写是在有一些命令行没办法支持的情况下才需要考虑，具体使用请**阅读文档**。

```
tunnel d wc_out c:\wc_out.dat;
2016-12-16 19:32:08 - new session: 201612161932082d3c9b0a012f68e7 total lines: 3
2016-12-16 19:32:08 - file [0]: [0, 3), c:\wc_out.dat
downloading 3 records into 1 file
2016-12-16 19:32:08 - file [0] start
2016-12-16 19:32:08 - file [0] OK. total: 21 bytes
download OK
```

# SQLTask+Tunnel方式导出

从前面SQL Task方式导出介绍可以看到，SQLTask不能处理超过1万条记录，而Tunnel可以，两者存在互补。所以可以基于两者实现数据的导出。以下用一个代码的例子来实现:

```
    private static final String accessId = "userAccessId";
private static final String accessKey = "userAccessKey";
private static final String endPoint = "http://service.odps.aliyun.com/api";
private static final String project = "userProject";
private static final String sql = "userSQL";
private static final String table = "Tmp_" + UUID.randomUUID().toString().replace("-", "_");//其实也就是随便找了个随
```

```
机字符串作为临时表的名字
private static final Odps odps = getOdps();

public static void main(String[] args) {
System.out.println(table);
runSql();
tunnel();
}

/*
* 把SQLTask的结果下载过来
* */
private static void tunnel() {
TableTunnel tunnel = new TableTunnel(odps);
try {
DownloadSession downloadSession = tunnel.createDownloadSession(
project, table);
System.out.println("Session Status is : "
+ downloadSession.getStatus().toString());
long count = downloadSession.getRecordCount();
System.out.println("RecordCount is: " + count);
RecordReader recordReader = downloadSession.openRecordReader(0,
count);
Record record;
while ((record = recordReader.read()) != null) {
consumeRecord(record, downloadSession.getSchema());
}
recordReader.close();
} catch (TunnelException e) {
e.printStackTrace();
} catch (IOException e1) {
e1.printStackTrace();
}
}

/*
* 保存这条数据
* 数据量少的话直接打印后拷贝走也是一种取巧的方法。实际场景可以用Java.io写到本地文件，或者写到远端数据等各种目标保存起来。
* */
private static void consumeRecord(Record record, TableSchema schema) {
System.out.println(record.getString("username")+","+record.getBigint("cnt"));
}

/*
* 运行SQL，把查询结果保存成临时表，方便后面用Tunnel下载
* 这里保存数据的lifecycle为1天，所以哪怕删除步骤出了问题，也不会太浪费存储空间
* */
private static void runSql() {
Instance i;
StringBuilder sb = new StringBuilder("Create Table ").append(table)
.append(" lifecycle 1 as ").append(sql);
try {
System.out.println(sb.toString());
i = SQLTask.run(getOdps(), sb.toString());
i.waitForSuccess();
```

```
} catch (OdpsException e) {
e.printStackTrace();
}
}

/*
* 初始化MaxCompute(原ODPS)的连接信息
* */
private static Odps getOdps() {
Account account = new AliyunAccount(accessId, accessKey);
Odps odps = new Odps(account);
odps.setEndpoint(endPoint);
odps.setDefaultProject(project);
return odps;
}
```

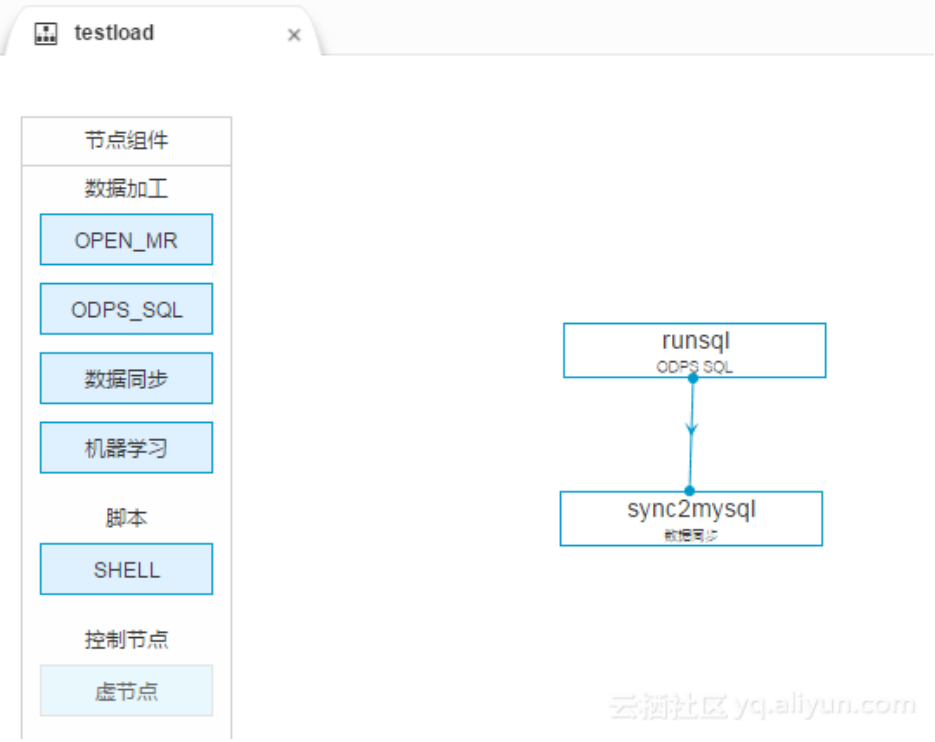
# 大数据开发套件的数据同步方式导出

前面介绍的方式解决了数据下载后保存的问题，但是没解决数据的生成以及两个步骤之间的调度依赖的问题。

本小节介绍的**数加·大数据开发套件**这个产品，可以运行SQL、**配置数据同步任务**，还可以设置自动**周期性运行**还有**多任务之间的依赖**，彻底解决了前面的烦恼。

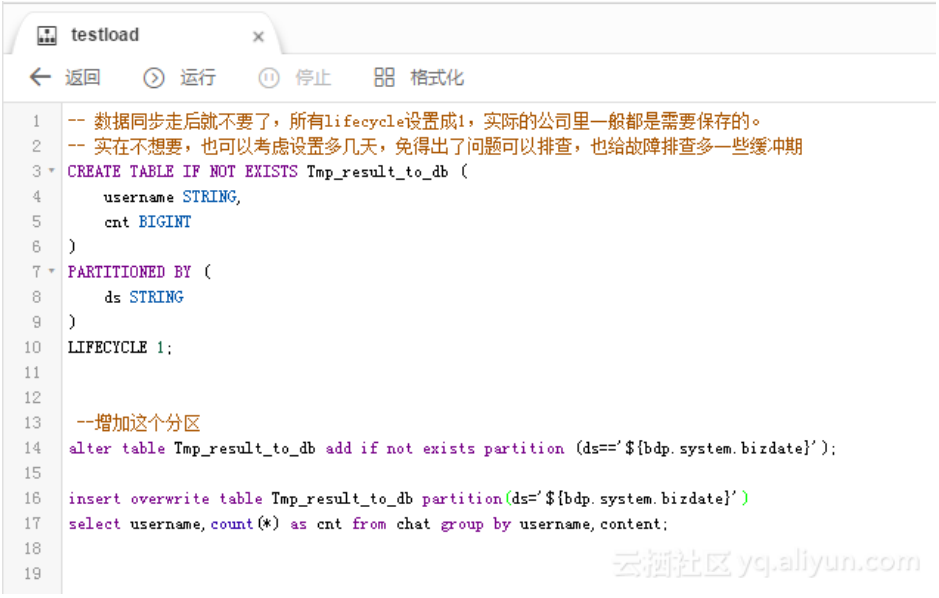
我们将用一个简单例子介绍如何通过大数据开发套件运行SQL并配置数据同步任务完成数据生成和导出需求。

步骤一：创建一个工作流，工作流里创建一个SQL节点和一个数据同步节点，并将两个节点连线配置成依赖关系，SQL节点作为数据产出的节点，数据同步节点作为数据导出节点。

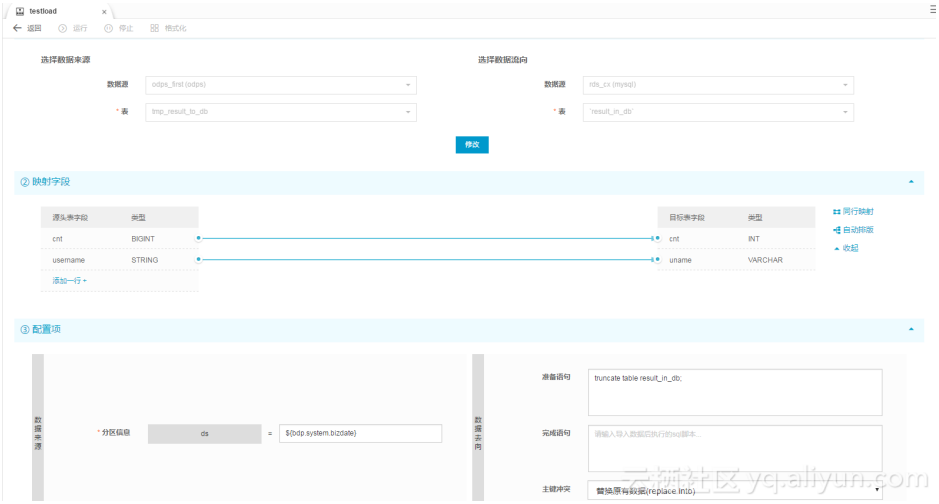



步骤二：配置SQL节点。注意，SQL这里的创建表要先执行一次再去配置同步（否则表都没有，同步任务没办

法配置）




步骤三：配置数据同步任务。




步骤四：工作流调度配置好后（可以直接使用默认配置），保存提交工作流，然后操作测试运行。对数据同步任务查看运行日志可看到

```
2016-12-17 23:43:46.394 [job-15598025] INFO  JobContainer -
任务启动时刻 : 2016-12-17 23:43:34
任务结束时刻 : 2016-12-17 23:43:46
任务总计耗时 : 11s
任务平均流量 : 31.36KB/s
记录写入速度 : 1668rec/s
读出记录总数 : 16689
读写失败总数 : 0
```

到mysql里查看数据同步结果。


长尾问题是分布式计算里最常见的问题之一，也是典型的疑难杂症。究其原因，是因为数据分布不均，导致各个节点的工作量不同，整个任务就需要等最慢的节点完成才能完成。

处理这类问题的思路就是把工作分给多个Worker去执行，而不是一个Worker单独抗下最重的那份工作。本文分享平时工作中遇到的一些典型的长尾问题的场景及其解决方案。

# Join长尾

Join能出现长尾，是因为Join时出现某个Key里的数据特别多的情况。

解法：

> 排除两张表都是小表的情况，若两张表里有一张大一张小，可以考虑使用Mapjoin，对小表进行缓存。具体语法和说明可以参考此文档。如果是MapReduce作业，可以使用资源表的功能，对小表进行缓存。
>
> 但是如果两张表都比较大，就需要先尽量去重。
>
> 若还是不能解决，就需要从业务上考虑，为什么会有这样的两个大数据量的Key要做笛卡尔积，直接考虑从业务上进行优化。

# Group By长尾

Group By Key 出现长尾的原因是因为某个Key内的计算量特别大。

解法 一：可对SQL进行改写，添加随机数，把长Key进行拆分。如SQL：

```
Select  Key,Count(*) As Cnt From TableName Group By Key;
```

不考虑Combiner，M节点会Shuffle到R上，然后R再做Count操作。对应的执行计划是M->R

但是如果对长尾的Key再做一次工作再分配，就变成：

```
-- 假设长尾的Key已经找到是KEY001
SELECT a.Key
, SUM(a.Cnt) AS Cnt
FROM (
SELECT Key
, COUNT(*) AS Cnt
FROM TableName
GROUP BY Key,
CASE
WHEN Key = 'KEY001' THEN Hash(Random()) % 50
ELSE 0
END
) a
GROUP BY a.Key;
```

可以看到，这次的执行计划变成了M->R->R。虽然执行的步骤变长了，但是长尾的Key经过了2个步骤的处理，整体的时间消耗可能反而有所减少。**注意，若数据的长尾并不严重，用这种方法人为地增加一次R的过程，最终的时间消耗可能反而更大。**

解法二：使用通用的优化策略——系统参数，设置

```
set  odps.sql.groupby.skewindata=true。
```

但是通用性的优化策略无法针对具体的业务进行分析，得出的结果不总是最优的。开发人员可以根据实际的数据情况，用更加高效的方法来改写SQL。

# Distinct长尾

可以看到，对于Distinct，上述Group By长尾“把长Key进行拆分”的策略已经不生效了。对这种场景，我们可以考虑其他方式解决。

解法：

```
--原始SQL,不考虑Uid为空
SELECT COUNT(uid) AS Pv
, COUNT(DISTINCT uid) AS Uv
FROM UserLog;
```

可以改写成

```
SELECT SUM(PV) AS Pv
, COUNT(*) AS UV
```

```
FROM (
SELECT COUNT(*) AS Pv
, uid
FROM UserLog
GROUP BY uid
) a;
```

该解法是把Distinct改成了普通的Count，这样的计算压力不会落到同一个Reducer上。而且这样改写后，既能支持前面提到的Group By优化，系统又能做Combiner，性能会有较大的提升。

# 动态分区长尾

动态分区功能为了整理小文件，会在最后起一个Reduce，对数据进行整理，所以如果使用动态分区写入数据时若有倾斜，就会发生长尾。另外一般情况下滥用动态分区的功能也是产生这类长尾的一个常见原因。

解法：若写入的数据已经确定需要把数据写入某个具体分区，那可以在Insert的时候指定需要写入的分区，而不是使用动态分区。

# 通过Combiner解决长尾

对于MapRedcuce作业，使用Combiner是一种常见的长尾优化策略。在WordCount的例子里，就已经有提到这种做法。通过Combiner，减少Maper Shuffle往Reducer的数据，可以大大减少网络传输的开销。对于MaxCompute SQL，这种优化会由系统自动完成。

需要注意的是，Combiner只是Map端的优化，需要保证是否执行Combiner的结果是一样的。以WordCount为例，传2个(KEY,1)和传1个(KEY,2)的结果是一样的。但是比如在做平均值的时候，就不能在Combiner里就把(KEY,1)和(KEY,2)合并成(KEY,1.5)。

# 通过系统优化解决长尾

针对长尾这种场景，除了前面提到的Local Combiner，MaxCompute系统本身还做了一些优化。比如在跑任务的时候，日志里突然打出这样的内容(+N backups部分)：

```
M1_Stg1_job0:0/521/521[100%]    M2_Stg1_job0:0/1/1[100%]    J9_1_2_Stg5_job0:0/523/523[100%]
J3_1_2_Stg1_job0:0/523/523[100%]    R6_3_9_Stg2_job0:1/1046/1047[100%]
M1_Stg1_job0:0/521/521[100%] M2_Stg1_job0:0/1/1[100%] J9_1_2_Stg5_job0:0/523/523[100%]
J3_1_2_Stg1_job0:0/523/523[100%] R6_3_9_Stg2_job0:1/1046/1047[100%]
M1_Stg1_job0:0/521/521[100%] M2_Stg1_job0:0/1/1[100%] J9_1_2_Stg5_job0:0/523/523[100%]
J3_1_2_Stg1_job0:0/523/523[100%] R6_3_9_Stg2_job0:1/1046/1047(+1 backups)[100%]
M1_Stg1_job0:0/521/521[100%] M2_Stg1_job0:0/1/1[100%] J9_1_2_Stg5_job0:0/523/523[100%]
J3_1_2_Stg1_job0:0/523/523[100%] R6_3_9_Stg2_job0:1/1046/1047(+1 backups)[100%]
```

可以看到1047个Reducer，有1046个已经完成了，但是最后一个一直没完成。系统识别出这种情况后，自动启动了一个新的Reducer，跑一样的数据，然后看两个哪个快，取快的数据归并到最后的结果集里。

## 通过业务优化解决长尾

虽然前面的优化策略有很多，但是实际上还是有限。有时候碰到长尾问题，还需要从业务角度上想想是否有更好的解决方法，比如：

- 实际数据可能包含非常多的噪音。如，需要根据访问者的ID进行计算，看每个用户的访问记录的行为。需要先去掉爬虫的数据（现在的爬虫已越来越难识别），否则爬虫数据很容易长尾计算的长尾。类似的情况还有根据xxid进行关联的时候，需要考虑这个关联字段是否存在为空的情况。
- 一些业务特殊情况，如，ISV的操作记录，在数据量、行为方式上都会和普通的个人会有很大的区别。那么可以考虑针对大客户，使用特殊的分析方式进行单独处理。
- 数据分布不均匀的情况下，不要使用常量字段做Distribute by字段来实现全排序。

本文通过课程实践方式介绍MaxCompute SQL，快速掌握SQL的写法，并清楚MaxCompute SQL和标准SQL的区别。请结合MaxCompute SQL基础文档阅读。

# 数据集准备

这里选择大家比较熟悉的Emp/Dept表做为数据集。为了方便大家操作，给出了相关的MaxCompute 建表语句和数据文件（emp表数据文件，dept表数据文件），可自行在MaxCompute项目上创建表并上传数据。

```
--创建emp表DDL语句
CREATE TABLE IF NOT EXISTS emp (
EMPNO string ,
ENAME string ,
JOB string ,
MGR bigint ,
HIREDATE datetime ,
SAL double ,
COMM double ,
DEPTNO bigint );

--创建dept表DDL语句
CREATE TABLE IF NOT EXISTS dept (
DEPTNO bigint ,
DNAME string ,
LOC string);
```

# SQL操作

## 初学SQL常遇到的问题点

1. 使用Group by，那Select的部分要么是分组项，要么就得是聚合函数。

2. Order by后面必须加Limit n。
3. Select表达式里不能用子查询，可以用Join改写。
4. Join不支持笛卡尔积，以及MapJoin的用法和使用场景。
5. Union all需要改成子查询的格式。
6. In/Not in语句对应的子查询只能有一列，而且返回的行数不能超过1000。否则也需要改成Join。

## 编写SQL进行解题

**题目一：列出至少有一个员工的所有部门。**

为了避免数据量太大的情况下导致“常遇问题点”中的第6点，我们需要使用Join进行改写:

```
SELECT d.*
FROM dept d
JOIN (
SELECT DISTINCT deptno AS no
FROM emp
) e
ON d.deptno = e.no;
```

**题目二：列出薪金比“SMITH”多的所有员工。**

MapJoin的典型场景：

```
SELECT /*+ MapJoin(a) */ e.empno
, e.ename
, e.sal
FROM emp e
JOIN (
SELECT MAX(sal) AS sal
FROM `emp`
WHERE `ENAME` = 'SMITH'
) a
ON e.sal > a.sal;
```

**题目三：列出所有员工的姓名及其直接上级的姓名。**

非等值连接：

```
SELECT a.ename
, b.ename
FROM emp a
LEFT OUTER JOIN emp b
ON b.empno = a.mgr;
```

**题目四：列出最低薪金大于1500的各种工作。**

Having的用法：

```
```

```
SELECT emp.`JOB`
, MIN(emp.sal) AS sal
FROM `emp`
GROUP BY emp.`JOB`
HAVING MIN(emp.sal) > 1500;
```

**题目五：列出在每个部门工作的员工数量、平均工资和平均服务期限。**

时间处理上有很多好用的内建函数：

```
SELECT COUNT(empno) AS cnt_emp
, ROUND(AVG(sal), 2) AS avg_sal
, ROUND(AVG(datediff(getdate(), hiredate, 'dd')), 2) AS avg_hire
FROM `emp`
GROUP BY `DEPTNO`;
```

**题目六： 列出每个部门的薪水前3名的人员的姓名以及他们的名次(Top n的需求非常常见)。**

```
SELECT *
FROM (
SELECT deptno
, ename
, sal
, ROW_NUMBER() OVER (PARTITION BY deptno ORDER BY sal DESC) AS nums
FROM emp
) emp1
WHERE emp1.nums < 4;
```

**题目七： 用一个SQL写出每个部门的人数、“CLERK”（办事员）的人数占该部门总人数占比。**

```
SELECT deptno
, COUNT(empno) AS cnt
, ROUND(SUM(CASE
WHEN job = 'CLERK' THEN 1
ELSE 0
END) / COUNT(empno), 2) AS rate
FROM `EMP`
GROUP BY deptno;
```

本文档将从习惯使用关系型数据库SQL的用户实践角度出发，列举用户在使用MaxCompute SQL时比较容易遇见的问题。具体的MaxCompute SQL语法建议考对应的文档，本文配合文档使用可以快速上手MaxCompute SQL。

# MaxCompute SQL基本区别

## 场景

- 不支持事物（没有commit和rollback，建议代码具有等幂性支持重跑，不推荐使用Insert Into，推荐Insert Overwrite写入数据）。
- 不支持索引和主外键约束。
- 不支持自增字段和默认值。如果有默认值，请在数据写入时自行赋值。

## 表分区

- 单表支持6万个分区。
- 一次查询输入的分区数不能大于1万，否则执行会报错 。另外如果是2级分区且查询时只根据2级分区进行过滤，总的分区数大于1万也可能导致报错。
- 一次查询输出的分区数不能大于2048 。

## 精度

- Double类型因为存在精度问题，不建议在关联时候进行直接等号关联两个Double字段。一个比较推荐的做法是把两个数做下减法，如果差距小于一个预设的值就认为是相同，比如abs(a1- a2) < 0.000000001。
- 目前产品上已经支持高精度的类型Decimal。但如果有更高精度要求的，可以先把数据存成String类型，然后使用UDF来实现对应的计算。

## 数据类型转换

- 为了防止出现各种预期外的错误，建议如果有2个不同的字段类型需要做Join，还是自己先把类型转好了后再Join，同时还能让代码更容易维护。
- 关于日期型和字符串的隐式转换。在需要传入日期型的函数里如果传入一个字符串，字符串和日期类型的转换根据yyyy-mm-dd hh:mi:ss格式进行转换。如果是其他格式请参考内建函数TO_DATE部分。

# DDL区别及解法

## 表结构

- 不能修改分区列列名，只能修改分区列对应的值。具体分区列和分区的区别可以参考此文档。
- 支持增加列，但是不支持删除列以及修改列的数据类型，请参考此处说明 。如果有需要一定要操作，最安全的方法参考此文档 。

# DML区别及解法

## INSERT

- 语法上最直观的区别是Insert into/overwrite后面有个关键字’Table’ 。

- 目前只支持Insert Into/Overwrite Table TableName Select的语法批量插入数据，还不支持Insert Into/Overwrite Table TableName Values(xxx)的语法。但是如果确实有需要写入单条数据，可以参考此文档。
- 数据插入表的字段映射不是根据Select的别名做的，而是根据Select的字段的顺序和表里的字段的顺序。

## UPDATE/DELETE

- 目前不支持Update/Delete语句，如果有需要可以参考此文档。另外对于删除，还可以参考此文档。

## SELECT

- 输入表的数量不能超过16张。
- Group by查询里的Select字段，要么是Group By的分组字段，要么需要使用聚合函数。从逻辑角度理解，如发现一个非分组列同一个Group By Key里的数据有多条，不使用聚合函数的话就没办法展示。

## 子查询

- 子查询必须要有别名。建议查询都带别名。

## IN/NOT IN

- 关于In/Not In,Exist/Not Exist，后面的子查询数据量不能超过1000条，解决办法参考此文档。如果业务上已经保证了子查询返回结果的唯一性，可以考虑去掉Distinct增加查询性能。

## SQL返回10000条

- MaxCompute限制了单独执行select语句时返回的数据条数，用户可以参考此文档进行配置，设置上限为1万。如果需要查询的结果数据条数很多，可以参考此文档配和Tunnel获取全部数据。

## MAPJOIN

- Join不支持笛卡尔积，也就是Join必须要用On设置关联条件。如果有一些小表需要做广播表，需要用Mapjoin Hint。具体可以参考此文档。

## ORDER BY

- **Order By 后面需要配合Limit n使用**。如果希望做很大的数据量的排序，甚至需要做全表排序，可以把这个N设置的很大。不过请谨慎使用，因为无法使用到分布式系统的优势，可能会有性能问题。可以参考此文档。

### UNION ALL

- 参与UNION ALL运算的所有列的数据类型、列个数、列名称必须完全一致，否则抛异常。
- UNION ALL查询外面需要再嵌套一层子查询。

针对MaxCompute SQL与标准SQL的区别和解法，欢迎到云栖社区此帖一起讨论！

# 批量数据通道

# SDK介绍

MaxCompute Tunnel是 MaxCompute 的数据通道，用户可以通过Tunnel向 MaxCompute 中上传或者下载数据。目前Tunnel仅支持表（不包括视图View）数据的上传下载。

MaxCompute 提供的 数据上传下载工具 即是基于Tunnel SDK编写的。

使用Maven的用户可以从Maven库中搜索”odps-sdk-core”获取不同版本的Java SDK，相关配置信息：

```
<dependency>
<groupId>com.aliyun.odps</groupId>
<artifactId>odps-sdk-core</artifactId>
<version>0.24.0-public</version>
</dependency>
```

这篇教程从用户的角度出发，介绍Tunnel SDK的主要接口，不同版本的SDK在使用上有差别，准确信息以SDK Java Doc为准。

| 主要接口 | 描述 |
| --- | --- |
| TableTunnel | 访问 MaxCompute Tunnel服务的入口类。用户可以通过公网或者阿里云内网环境对MaxCompute 及其Tunnel进行访问。当用户在阿里云内网环境中，使用Tunnel内网连接下载数据时，MaxCompute 不会将该操作产生的流量计入计费。此外内网地址仅对杭州域的云产品有效。 |
| TableTunnel.UploadSession | 表示一个向 MaxCompute 表中上传数据的会话。 |
| TableTunnel.DownloadSession | 表示一个向 MaxCompute 表中下载数据的会话。 |

备注：

- 关于SDK的更多详细信息请参阅 SDK Java Doc ；
- 有关服务连接的说明请参考 服务连接；

接口定义：

```
public class TableTunnel {
public DownloadSession createDownloadSession(String projectName, String tableName);
public DownloadSession createDownloadSession(String projectName, String tableName, PartitionSpec
partitionSpec);
public UploadSession createUploadSession(String projectName, String tableName);
public UploadSession createUploadSession(String projectName, String tableName, PartitionSpec partitionSpec);
public DownloadSession getDownloadSession(String projectName, String tableName, PartitionSpec partitionSpec,
String id);
public DownloadSession getDownloadSession(String projectName, String tableName, String id);
public UploadSession getUploadSession(String projectName, String tableName, PartitionSpec partitionSpec, String
id);
public UploadSession getUploadSession(String projectName, String tableName, String id);
}
```

TableTunnel：

- 生命周期: 从TableTunnel实例被创建开始，一直到程序结束。
- 提供创建Upload对象和Download对象的方法

接口定义：

```
public class UploadSession {

UploadSession(Configuration conf, String projectName, String tableName,

String partitionSpec) throws TunnelException;

UploadSession(Configuration conf, String projectName, String tableName,

String partitionSpec, String uploadId) throws TunnelException;

public void commit(Long[] blocks);

public Long[] getBlockList();

public String getId();

public TableSchema getSchema();

public UploadSession.Status getStatus();

public Record newRecord();
```

```
public RecordWriter openRecordWriter(long blockId);

public RecordWriter openRecordWriter(long blockId, boolean compress);

public RecordWriter openBufferedWriter();

public RecordWriter openBufferedWriter(boolean compress);
}
```

Upload对象：

生命周期：从创建Upload实例到结束上传

创建Upload实例，可以通过调用构造方法创建，也可以通过TableTunnel创建；

请求方式：同步

Server端会为该Upload创建一个session， 生成唯一uploadId标识该Upload，客户端可以通过getId获取

上传数据：

请求方式：同步

调用openRecordWriter方法，生成RecordWriter实例，其中参数blockId用于标识此次上传的数据，也描述了数据在整个表中的位置，取值范围：[0,20000]，当数据上传失败，可以根据blockId重新上传。

查看上传：

请求方式：同步

调用getStatus可以获取当前Upload状态

调用getBlockList可以获取成功上传的blockid list，可以和上传的blockid list对比，对失败的blockId重新上传

结束上传：

请求方式：同步

调用commit(Long[] blocks)方法，参数blocks列表表示已经成功上传的block列表，server端会对该列表进行验证

该功能是加强对数据正确性的校验，如果提供的block列表与server端存在的block列表不一致抛出异常

Commit失败可以进行重试

7种状态说明：

UNKNOWN, server端刚创建一个session时设置的初始值

NORMAL, 创建upload对象成功

CLOSING, 当调用complete方法(结束上传)时，服务端会先把状态置为CLOSING。

CLOSED, 完成结束上传(即把数据移动到结果表所在目录)后

EXPIRED, 上传超时

CRITICAL, 服务出错

注意：

同一个UploadSession里的blockId不能重复。也就是说，对于同一个UploadSession，用一个blockId打开RecordWriter，写入一批数据后，调用close，然后再commit完成后，不可以重新再用该blockId打开另一个RecordWriter写入数据。

一个block大小上限 100GB，建议 大于 64M的数据。

每个Session在服务端的生命周期为24小时。

上传数据时，Writer每写入8KB数据会触发一次网络动作，如果120秒内没有网络动作，服务端将主动关闭连接，届时Writer将不可用，请重新打开一个新的Writer写入。

建议使用openBufferedWriter接口上传数据，该接口对用户屏蔽了blockId的细节，并且内部带有数据缓存区，会自动进行失败重试，具体使用方法参见TunnelBufferedWriter的介绍和示例。

接口定义：

```
public class DownloadSession {
    DownloadSession(Configuration conf, String projectName, String tableName,
        String partitionSpec) throws TunnelException
    DownloadSession(Configuration conf, String projectName, String tableName,
        String partitionSpec, String downloadId) throws TunnelException
    public String getId()
    public long getRecordCount()
    public TableSchema getSchema()
    public DownloadSession.Status getStatus()
    public RecordReader openRecordReader(long start, long count)
    public RecordReader openRecordReader(long start, long count, boolean compress)
}
```

Download对象：

- 生命周期：从创建Download实例到下载结束
- 创建Download实例，可以通过调用构造方法创建，也可以通过TableTunnel创建；
    - 请求方式：同步
    - Server端会为该Download创建一个session，生成唯一downloadId标识该Download，客户端可以通过getId获取
    - 该操作开销较大，server端会对数据文件创建索引，当文件数很多时，该时间会比较长；
    - 同时server端会返回总Record数，可以根据总Record数启动多个并发同时下载
- 下载数据：
    - 请求方式：异步
    - 调用openRecordReader方法，生成RecordReader实例，其中参数start标识本次下载的record的起始位置，从0开始，取值范围是 >= 0, count标识本次下载的记录数，取值范围是>0。
- 查看下载：
    - 请求方式：同步
    - 调用getStatus可以获取当前Download状态
- 4种状态说明：
    - UNKNOWN, server端刚创建一个session时设置的初始值
    - NORMAL, 创建Download对象成功
    - CLOSED, 下载结束后
    - EXPIRED, 下载超时

一次完整的上传流程通常包括以下步骤：

先对数据进行划分;

为每个数据块指定 block id，即调用 openRecordWriter(id);

然后用一个或多个线程分别将这些 block 上传上去, 并在某个 block 上传失败以后，需要对整个 block 进行重传;

在所有 block 都上传以后，向服务端提供上传成功的 blockid list 进行校验，即调用 session.commit([1,2,3,...])

而由于服务端对block管理，连接超时等的一些限制，上传过程逻辑变得比较复杂，为了简化上传过程，SDK提供了更高级的一种RecordWriter——TunnelBufferWriter。

接口定义：

```
    public class TunnelBufferedWriter implements RecordWriter {
public TunnelBufferedWriter(TableTunnel.UploadSession session, CompressOption option) throws IOException;

public long getTotalBytes();

public void setBufferSize(long bufferSize);

public void setRetryStrategy(RetryStrategy strategy);

public void write(Record r) throws IOException;

public void close() throws IOException;
}
```

TunnelBufferedWriter对象：

生命周期：从创建RecordWriter到数据上传结束；

创建TunnelBufferedWriter实例：通过调用UploadSession的openBufferedWriter接口创建；

数据上传：调用write接口，数据会先写入本地缓存区，缓存区满后会批量提交到服务端，避免了连接超时，同时，如果上传失败会自动进行重试；

结束上传: 调用close接口，最后再调用UploadSession的commit接口，即可完成上传；

缓冲区控制: 可以通过setBufferSize这个接口修改缓冲区占内存的字节数（bytes），建议设置64M以上的大小，避免服务端产生过多小文件，影响性能，一般无须设置，维持默认值即可；

重试策略设置：用户可以选择三种重试回避策略：指数回避（EXPONENTIAL_BACKOFF）、线性时间回避（LINEAR_BACKOFF）、常数时间回避（CONSTANT_BACKOFF）。例如下面这段代码可以将，write 的重试次数调整为 6，每一次重试之前先分别回避 4s、8s、16s、32s、64s 和 128s（从 4 开始的指数递增的序列），这个也是默认的行为，一般情况不建议调整。

```
RetryStrategy retry
= new RetryStrategy(6, 4, RetryStrategy.BackoffStrategy.EXPONENTIAL_BACKOFF)

writer = (TunnelBufferedWriter) uploadSession.openBufferedWriter();
writer.setRetryStrategy(retry);
```

# SDK示例

- ODPS提供了两个服务地址供用户选择。Tunnel服务地址的选择会直接影响用户上传数据的效率及计量计费。详细说明请参考 Tunnel SDK简介
- 数据上传建议使用TunnelBufferedWriter，使用方式可以参考BufferedWriter相关的示例代码。
- 不同版本SDK会有不同，本示例仅供参考。请用户注意版本变更。

```
 import java.io.IOException;
import java.util.Date;

import com.aliyun.odps.Column;
import com.aliyun.odps.Odps;
import com.aliyun.odps.PartitionSpec;
import com.aliyun.odps.TableSchema;
import com.aliyun.odps.account.Account;
import com.aliyun.odps.account.AliyunAccount;
import com.aliyun.odps.data.Record;
import com.aliyun.odps.data.RecordWriter;
import com.aliyun.odps.tunnel.TableTunnel;
import com.aliyun.odps.tunnel.TunnelException;
import com.aliyun.odps.tunnel.TableTunnel.UploadSession;

public class UploadSample {
private static String accessId = "<your access id>";
private static String accessKey = "<your access Key>";
private static String odpsUrl = "http://service.odps.aliyun.com/api";

private static String project = "<your project>";
private static String table = "<your table name>";
private static String partition = "<your partition spec>";

public static void main(String args[]) {
Account account = new AliyunAccount(accessId, accessKey);
Odps odps = new Odps(account);
odps.setEndpoint(odpsUrl);
odps.setDefaultProject(project);
try {
TableTunnel tunnel = new TableTunnel(odps);
```

```
PartitionSpec partitionSpec = new PartitionSpec(partition);
UploadSession uploadSession = tunnel.createUploadSession(project,
table, partitionSpec);

System.out.println("Session Status is : "
+ uploadSession.getStatus().toString());

TableSchema schema = uploadSession.getSchema();
// 准备数据后打开Writer开始写入数据，准备数据后写入一个Block
// 单个Block内写入数据过少会产生大量小文件 严重影响计算性能， 强烈建议每次写入64MB以上数据(100GB以内数据均可写入同一Block)
// 可通过数据的平均大小与记录数量大致计算总量即 64MB < 平均记录大小*记录数 < 100GB

RecordWriter recordWriter = uploadSession.openRecordWriter(0);
Record record = uploadSession.newRecord();
for (int i = 0; i < schema.getColumns().size(); i++) {
Column column = schema.getColumn(i);
switch (column.getType()) {
case BIGINT:
record.setBigint(i, 1L);
break;
case BOOLEAN:
record.setBoolean(i, true);
break;
case DATETIME:
record.setDatetime(i, new Date());
break;
case DOUBLE:
record.setDouble(i, 0.0);
break;
case STRING:
record.setString(i, "sample");
break;
default:
throw new RuntimeException("Unknown column type: "
+ column.getType());
}
}
for (int i = 0; i < 10; i++) {
// Write数据至服务端，每写入8KB数据会进行一次网络传输
// 若120s没有网络传输服务端将会关闭连接，届时该Writer将不可用，需要重新写入

recordWriter.write(record);
}
recordWriter.close();
uploadSession.commit(new Long[]{0L});
System.out.println("upload success!");

} catch (TunnelException e) {
e.printStackTrace();
} catch (IOException e) {
e.printStackTrace();
}
}
}
```

构造器举例说明：PartitionSpec(String spec)：通过字符串构造此类对象。参数:spec: 分区定义字符串，比如: pt=’1’,ds=’2’。因此程序中应该这样配置：private static String partition = “pt=’XXX’,ds=’XXX’”；

```
 import java.io.IOException;
import java.util.Date;

import com.aliyun.odps.Column;
import com.aliyun.odps.Odps;
import com.aliyun.odps.PartitionSpec;
import com.aliyun.odps.TableSchema;
import com.aliyun.odps.account.Account;
import com.aliyun.odps.account.AliyunAccount;
import com.aliyun.odps.data.Record;
import com.aliyun.odps.data.RecordReader;
import com.aliyun.odps.tunnel.TableTunnel;
import com.aliyun.odps.tunnel.TableTunnel.DownloadSession;
import com.aliyun.odps.tunnel.TunnelException;

public class DownloadSample {
private static String accessId = "<your access id>";
private static String accessKey = "<your access Key>";
private static String odpsUrl = "http://service.odps.aliyun.com/api";


private static String project = "<your project>";
private static String table = "<your table name>";
private static String partition = "<your partition spec>";

public static void main(String args[]) {
Account account = new AliyunAccount(accessId, accessKey);
Odps odps = new Odps(account);
odps.setEndpoint(odpsUrl);
odps.setDefaultProject(project);
TableTunnel tunnel = new TableTunnel(odps);
PartitionSpec partitionSpec = new PartitionSpec(partition);
try {
DownloadSession downloadSession = tunnel.createDownloadSession(project, table,
partitionSpec);
System.out.println("Session Status is : "
+ downloadSession.getStatus().toString());

long count = downloadSession.getRecordCount();
System.out.println("RecordCount is: " + count);

RecordReader recordReader = downloadSession.openRecordReader(0,
count);
Record record;
while ((record = recordReader.read()) != null) {
consumeRecord(record, downloadSession.getSchema());
}
recordReader.close();
```

```
} catch (TunnelException e) {
e.printStackTrace();
} catch (IOException e1) {
e1.printStackTrace();
}
}

private static void consumeRecord(Record record, TableSchema schema) {
for (int i = 0; i < schema.getColumns().size(); i++) {
Column column = schema.getColumn(i);
String colValue = null;
switch (column.getType()) {
case BIGINT: {
Long v = record.getBigint(i);
colValue = v == null ? null : v.toString();
break;
}
case BOOLEAN: {
Boolean v = record.getBoolean(i);
colValue = v == null ? null : v.toString();
break;
}
case DATETIME: {
Date v = record.getDatetime(i);
colValue = v == null ? null : v.toString();
break;
}
case DOUBLE: {
Double v = record.getDouble(i);
colValue = v == null ? null : v.toString();
break;
}
case STRING: {
String v = record.getString(i);
colValue = v == null ? null : v.toString();
break;
}
default:
throw new RuntimeException("Unknown column type: "
+ column.getType());
}
System.out.print(colValue == null ? "null" : colValue);
if (i != schema.getColumns().size())
System.out.print("\t");
}
System.out.println();
}

}
```

本示例中，为了方便测试，数据是通过System.out.println打出来，在实际使用时，可改写成直接输出到文本文件。

```
 import java.io.IOException;
import java.util.ArrayList;
import java.util.Date;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import com.aliyun.odps.Column;
import com.aliyun.odps.Odps;
import com.aliyun.odps.PartitionSpec;
import com.aliyun.odps.TableSchema;
import com.aliyun.odps.account.Account;
import com.aliyun.odps.account.AliyunAccount;
import com.aliyun.odps.data.Record;
import com.aliyun.odps.data.RecordWriter;
import com.aliyun.odps.tunnel.TableTunnel;
import com.aliyun.odps.tunnel.TunnelException;
import com.aliyun.odps.tunnel.TableTunnel.UploadSession;

class UploadThread implements Callable<Boolean> {
private long id;
private RecordWriter recordWriter;
private Record record;
private TableSchema tableSchema;

public UploadThread(long id, RecordWriter recordWriter, Record record,
TableSchema tableSchema) {
this.id = id;
this.recordWriter = recordWriter;
this.record = record;
this.tableSchema = tableSchema;
}

@Override
public Boolean call() {

for (int i = 0; i < tableSchema.getColumns().size(); i++) {
Column column = tableSchema.getColumn(i);
switch (column.getType()) {
case BIGINT:
record.setBigint(i, 1L);
break;
case BOOLEAN:
record.setBoolean(i, true);
break;
case DATETIME:
record.setDatetime(i, new Date());
break;
case DOUBLE:
record.setDouble(i, 0.0);
break;
case STRING:
record.setString(i, "sample");
break;
default:
```

```
throw new RuntimeException("Unknown column type: "
+ column.getType());
}
}

for (int i = 0; i < 10; i++) {
try {
recordWriter.write(record);
} catch (IOException e) {
recordWriter.close();
e.printStackTrace();
return false;
}
}
recordWriter.close();
return true;
}

}

public class UploadThreadSample {
private static String accessId = "<your access id>";
private static String accessKey = "<your access Key>";

private static String odpsUrl = "<http://service.odps.aliyun.com/api>";


private static String project = "<your project>";
private static String table = "<your table name>";
private static String partition = "<your partition spec>";

private static int threadNum = 10;

public static void main(String args[]) {
Account account = new AliyunAccount(accessId, accessKey);
Odps odps = new Odps(account);
odps.setEndpoint(odpsUrl);
odps.setDefaultProject(project);
try {
TableTunnel tunnel = new TableTunnel(odps);
PartitionSpec partitionSpec = new PartitionSpec(partition);
UploadSession uploadSession = tunnel.createUploadSession(project,
table, partitionSpec);

System.out.println("Session Status is : "
+ uploadSession.getStatus().toString());

ExecutorService pool = Executors.newFixedThreadPool(threadNum);
ArrayList<Callable<Boolean>> callers = new ArrayList<Callable<Boolean>>();
for (int i = 0; i < threadNum; i++) {
RecordWriter recordWriter = uploadSession.openRecordWriter(i);
Record record = uploadSession.newRecord();
callers.add(new UploadThread(i, recordWriter, record,
uploadSession.getSchema()));
}
```

```
pool.invokeAll(callers);
pool.shutdown();

Long[] blockList = new Long[threadNum];
for (int i = 0; i < threadNum; i++)
blockList[i] = Long.valueOf(i);
uploadSession.commit(blockList);
System.out.println("upload success!");

} catch (TunnelException e) {
e.printStackTrace();
} catch (IOException e) {
e.printStackTrace();
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}
```

注意：对于tunnel endpoint，支持指定或者不指定。如果指定，按照指定的endpoint路由。如果不指定，支持自动路由。

```
 import java.io.IOException;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

import com.aliyun.odps.Column;
import com.aliyun.odps.Odps;
import com.aliyun.odps.PartitionSpec;
import com.aliyun.odps.TableSchema;
import com.aliyun.odps.account.Account;
import com.aliyun.odps.account.AliyunAccount;
import com.aliyun.odps.data.Record;
import com.aliyun.odps.data.RecordReader;
import com.aliyun.odps.tunnel.TableTunnel;
import com.aliyun.odps.tunnel.TableTunnel.DownloadSession;
import com.aliyun.odps.tunnel.TunnelException;

class DownloadThread implements Callable<Long> {
private long id;
private RecordReader recordReader;
private TableSchema tableSchema;

public DownloadThread(int id,
```

```
RecordReader recordReader, TableSchema tableSchema) {
this.id = id;
this.recordReader = recordReader;
this.tableSchema = tableSchema;
}

@Override
public Long call() {
Long recordNum = 0L;
try {
Record record;
while ((record = recordReader.read()) != null) {
recordNum++;
System.out.print("Thread " + id + "\t");
consumeRecord(record, tableSchema);
}
recordReader.close();
} catch (IOException e) {
e.printStackTrace();
}
return recordNum;
}

private static void consumeRecord(Record record, TableSchema schema) {
for (int i = 0; i < schema.getColumns().size(); i++) {
Column column = schema.getColumn(i);
String colValue = null;
switch (column.getType()) {
case BIGINT: {
Long v = record.getBigint(i);
colValue = v == null ? null : v.toString();
break;
}
case BOOLEAN: {
Boolean v = record.getBoolean(i);
colValue = v == null ? null : v.toString();
break;
}
case DATETIME: {
Date v = record.getDatetime(i);
colValue = v == null ? null : v.toString();
break;
}
case DOUBLE: {
Double v = record.getDouble(i);
colValue = v == null ? null : v.toString();
break;
}
case STRING: {
String v = record.getString(i);
colValue = v == null ? null : v.toString();
break;
}
default:
throw new RuntimeException("Unknown column type: "
+ column.getType());
```

```
}
System.out.print(colValue == null ? "null" : colValue);
if (i != schema.getColumns().size())
System.out.print("\t");
}
System.out.println();
}

}

public class DownloadThreadSample {
private static String accessId = "<your access id>";
private static String accessKey = "<your access Key>";

private static String odpsUrl = "http://service.odps.aliyun.com/api";


private static String project = "<your project>";
private static String table = "<your table name>";
private static String partition = "<your partition spec>";

private static int threadNum = 10;

public static void main(String args[]) {
Account account = new AliyunAccount(accessId, accessKey);
Odps odps = new Odps(account);
odps.setEndpoint(odpsUrl);
odps.setDefaultProject(project);
TableTunnel tunnel = new TableTunnel(odps);
PartitionSpec partitionSpec = new PartitionSpec(partition);
DownloadSession downloadSession;
try {
downloadSession = tunnel.createDownloadSession(project, table,
partitionSpec);

System.out.println("Session Status is : "
+ downloadSession.getStatus().toString());

long count = downloadSession.getRecordCount();
System.out.println("RecordCount is: " + count);

ExecutorService pool = Executors.newFixedThreadPool(threadNum);
ArrayList<Callable<Long>> callers = new ArrayList<Callable<Long>>();

long start = 0;
long step = count / threadNum;
for (int i = 0; i < threadNum - 1; i++) {
RecordReader recordReader = downloadSession.openRecordReader(
step * i, step);
callers.add(new DownloadThread( i, recordReader, downloadSession.getSchema()));
}
RecordReader recordReader = downloadSession.openRecordReader(step * (threadNum - 1), count
- ((threadNum - 1) * step));
callers.add(new DownloadThread( threadNum - 1, recordReader, downloadSession.getSchema()));

Long downloadNum = 0L;
```

```
List<Future<Long>> recordNum = pool.invokeAll(callers);
for (Future<Long> num : recordNum)
downloadNum += num.get();
System.out.println("Record Count is: " + downloadNum);
pool.shutdown();

} catch (TunnelException e) {
e.printStackTrace();
} catch (IOException e) {
e.printStackTrace();
} catch (InterruptedException e) {
e.printStackTrace();
} catch (ExecutionException e) {
e.printStackTrace();
}
}
}
```

注意：对于tunnel endpoint，支持指定或者不指定。如果指定，按照指定的下载。如果不指定，按照我们的自动路由下载。

代码片段：

```
class UploadThread extends Thread {
private UploadSession session;
private static int RECORD_COUNT = 1200;

public UploadThread(UploadSession session) {
this.session = session;
}

@Override
public void run() {
RecordWriter writer = up.openBufferedWriter();
Record r = up.newRecord();
for (int i = 0; i < RECORD_COUNT; i++) {
r.setBigint(0, i);
writer.write(r);
}
writer.close();
}
};

public class Example {
public static void main(String args[]) {

// 初始化 MaxCompute 和 tunnel 的代码

TableTunnel.UploadSession uploadSession = tunnel.createUploadSession(projectName, tableName);
UploadThread t1 = new UploadThread(up);
UploadThread t2 = new UploadThread(up);
```

```
t1.start();
t2.start();
t1.join();
t2.join();

uploadSession.commit();
}
```

代码片段：

```
// 初始化 MaxCompute 和 tunnel 的代码

RecordWriter writer = null;
TableTunnel.UploadSession uploadSession = tunnel.createUploadSession(projectName, tableName);

try {
int i = 0;
// 生成TunnelBufferedWriter的实例
writer = uploadSession.openBufferedWriter();
Record product = uploadSession.newRecord();

for (String item : items) {
product.setString("name", item);
product.setBigint("id", i);
// 调用write接口写入数据
writer.write(product);
i += 1;
}
} finally {
if (writer != null) {
// 关闭TunnelBufferedWriter
writer.close();
}
}
// uploadSession提交，结束上传
uploadSession.commit();
```