阿里云
MaxCompute

最佳实践

文档版本：20180803

法律声明

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

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。
2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。
3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。
4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的”现状“、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。
5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含”阿里云”、Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。
6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	 **禁止：** 重置操作将丢失用户配置数据。
	该类警示信息可能导致系统重大变更甚至故障，或者导致人身伤害等结果。	 **警告：** 重启操作将导致业务中断，恢复业务所需时间约10分钟。
	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	 **说明：** 您也可以通过按**Ctrl** + **A**选中全部文件。
>	多级菜单递进。	设置 > 网络 > 设置网络类型
粗体	表示按键、菜单、页面名称等UI元素。	单击 **确定**。
`courier` 字体	命令。	执行 `cd /d C:/windows` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *`Instance_ID`*
[]或者[a|b]	表示可选项，至多选择一个。	`ipconfig` *`[-all|-t]`*
{}或者{a|b}	表示必选项，至多选择一个。	`swich` *`{stand | slave}`*

目录

1 SQL实现多行数据转一条

本文将为您介绍，如何使用 SQL 实现多条数据压缩为一条。

场景示例

以下表数据为例：

class	gender	name
1	M	LiLei
1	F	HanMM
1	M	Jim
2	F	Kate
2	M	Peter

场景一

根据需求，常见场景如下：

class	names
1	LiLei,HanMM,Jim
2	Kate,Peter

类似这样使用某个分隔符做字符串拼接，可以使用如下语句：

```
SELECT class, wm_concat(',', name) FROM students GROUP BY class;
```

场景二

另外一种常见需求，如下所示：

class	cnt_m	cnt_f
1	2	1
2	1	1

类似这样转多列的需求，可以使用如下语句：

```
SELECT
class
,SUM(CASE WHEN gender = 'M' THEN 1 ELSE 0 END) AS cnt_m
,SUM(CASE WHEN gender = 'F' THEN 1 ELSE 0 END) AS cnt_f
```

```
FROM students
GROUP BY class;
```

2 分组取出每组数据的前N条

本文将为您介绍如何对数据进行分组，取出每组数据的前 N 条数据。

示例数据

目前的数据，如下表所示：

empno	ename	job	sal
7369	SMITH	CLERK	800.0
7876	SMITH	CLERK	1100.0
7900	JAMES	CLERK	950.0
7934	MILLER	CLERK	1300.0
7499	ALLEN	SALESMAN	1600.0
7654	MARTIN	SALESMAN	1250.0
7844	TURNER	SALESMAN	1500.0
7521	WARD	SALESMAN	1250.0

实现方法

您可以通过以下两种方法实现：

- 取出每条数据的行号，再用 where 语句进行过滤。

```
SELECT * FROM (
  SELECT empno
  , ename
  , sal
  , job
  , ROW_NUMBER() OVER (PARTITION BY job ORDER BY sal) AS rn
  FROM emp
) tmp
WHERE rn < 10;
```

- 使用 UDTF 实现 Split 函数。

  详情请参见 此文 中最后的示例。这个例子可以更迅速地判断当前的序号，如果是已经超过预定的条数（比如 10 条），便不做处理了，从而提高计算效率。

3 导出SQL的运行结果

本文将通过示例，为您介绍几种下载 MaxCompute SQL 计算结果的方法。

说明：

本文中所有的 SDK 部分仅举例介绍 Java 的例子。

您可以通过以下几种方法导出SQL的运行结果：

- 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。
- *DataWorks* 可以方便地帮您运行 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** 可帮助您方便地把 MaxCompute 中的数据导出到目标数据源，详情请参见 *DataX*概述。

SQLTask 方式导出

SQL Task 是 SDK 直接调用 MaxCompute SQL 的接口，能很方便地运行 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。不过此方法有个缺陷，详情请参见 其他操作中提到的`SetProject READ_TABLE_MAX_ROW`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

如果您需要导出的查询结果是某张表的全部内容（或者是具体的某个分区的全部内容），可以通过 Tunnel 来实现，详情请参见 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示例，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，详情请参见 批量数据通道概述。

```
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 方式导出的介绍可以看到，SQL Task 不能处理超过 1 万条记录，而 Tunnel 可以，两者可以互补。所以可以基于两者实现数据的导出。

代码实现的示例如下：

```
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

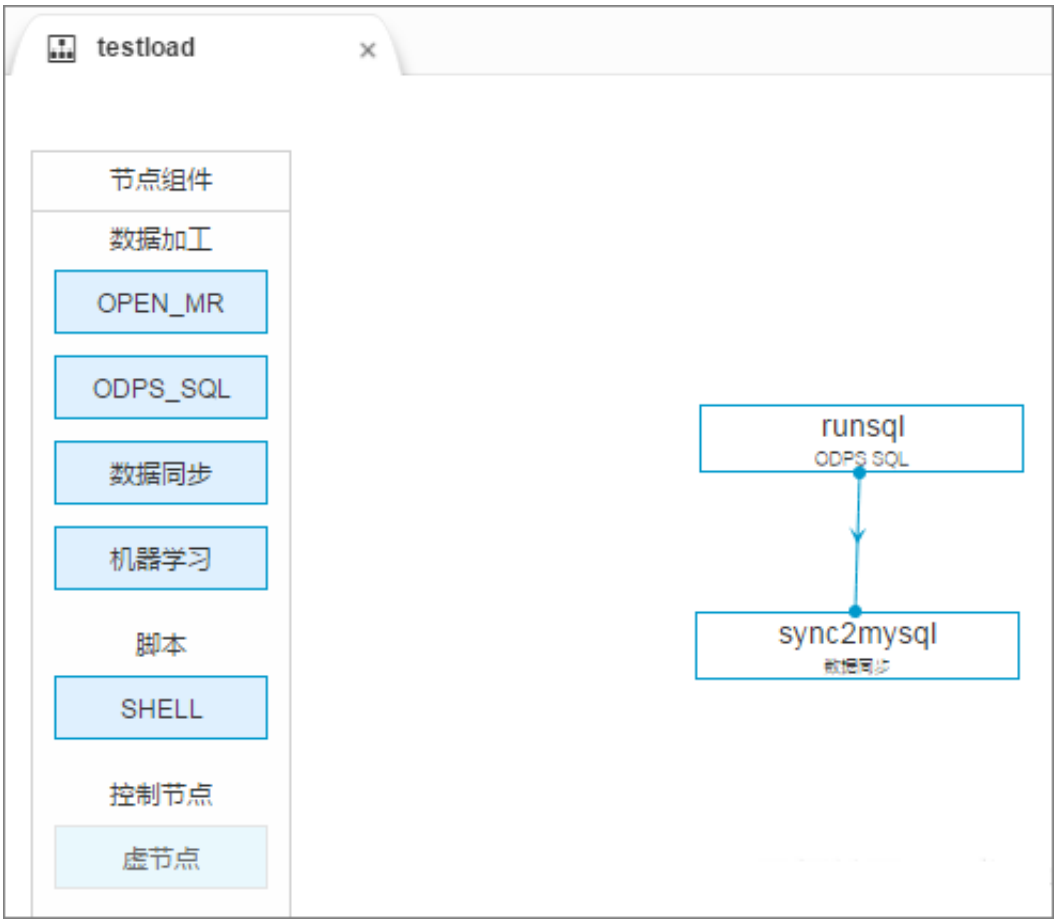
前面介绍的方式解决了数据下载后保存的问题，但是没解决数据的生成以及两个步骤之间的调度依赖的问题。

数加·*DataWorks* 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 和 多任务之间依赖，彻底解决了前面的烦恼。

接下来将用一个简单示例，为您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，以完成数据生成和导出需求。

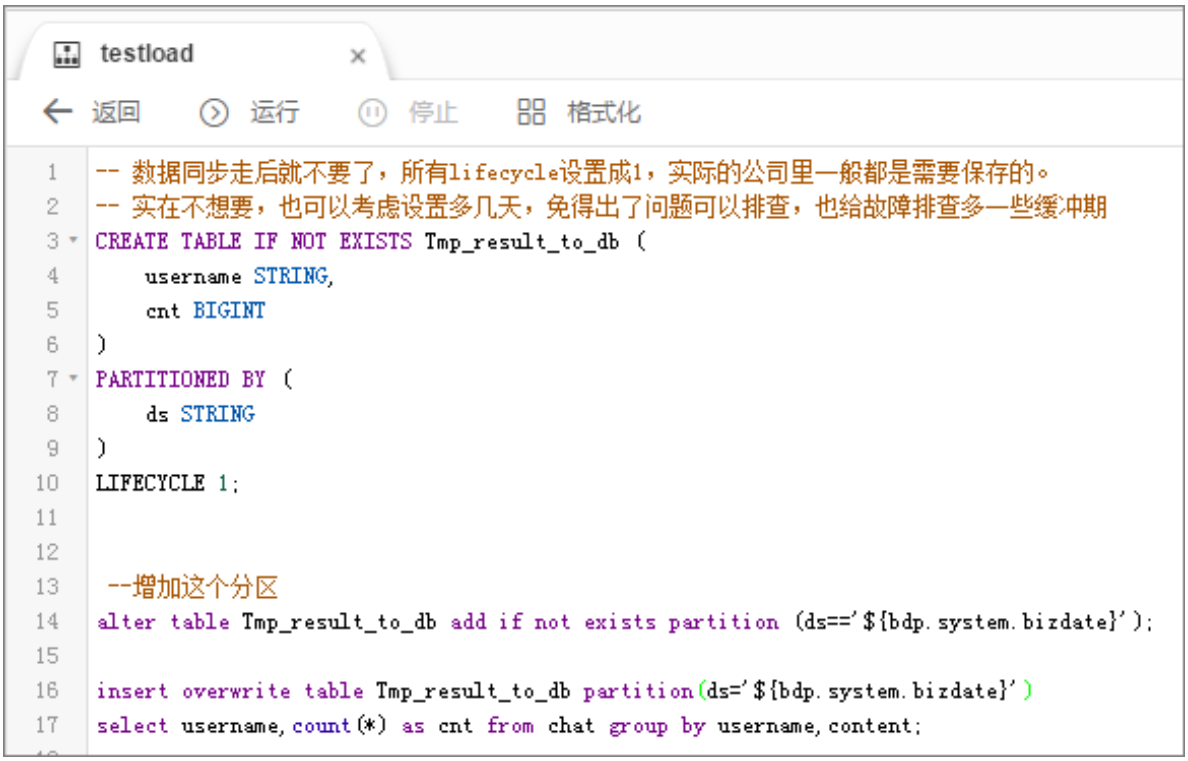
操作步骤

1. 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。



2. 配置 SQL 节点。

> **说明：**
> 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）。



3. 配置数据同步任务。

   a. 选择来源。

   b. 选择目标。

c. 字段映射。

d. 通道控制。

e. 预览保存。

4. 工作流调度配置完成后（可以直接使用默认配置），保存并提交工作流，然后单击 **测试运行**。

   查看数据同步的运行日志，如下所示：

```
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
```

5. 输入 SQL 语句查看数据同步的结果，如下图所示：

4 修改不兼容SQL实战

MaxCompute 开发团队近期已经完成了 MaxCompute2.0 灰度升级。新升级的版本完全拥抱开源生态，支持更多的语言功能，带来更快的运行速度，同时新版本会执行更严格的语法检测，以致于一些在老编译器下正常执行的不严谨的语法 case 在 MaxCompute2.0 下会报错。

为了使 MaxCompute2.0 灰度升级更加平滑，MaxCompute 框架支持回退机制，如果 MaxCompute2.0 任务失败，会回退到 MaxCompute1.0 执行。回退本身会增加任务 E2E 时延。鼓励大家提交作业之前，手动关闭回退`set odps.sql.planner.mode=lot;`以避免 MaxCompute 框架回退策略修改对大家造成影响。

MaxCompute 团队会根据线上回退情况，邮件或者钉钉等通知有问题任务的 Owner，请大家尽快完成 SQL 任务修改，否则会导致任务失败。烦请大家仔细 check 以下报错情况，进行自检，以免通知遗漏造成任务失败。

下面列举常见的一些会报错的语法：

group.by.with.star

SELECT * …GROUP BY… 的问题。

旧版 MaxCompute 中，即使 * 中覆盖的列不在 group by key 内，也支持 `select * from group by key` 的语法，但 MaxCompute2.0 和 Hive 兼容，并不允许这种写法，除非 group by 列表是所有源表中的列。示例如下：

场景一：**group by key** 不包含所有列

错误写法：

```
SELECT * FROM t GROUP BY key;
```

报错信息：

```
FAILED: ODPS-0130071:[1,8] Semantic analysis exception - column
reference t.value should appear in GROUP BY key
```

正确改法：

```
SELECT DISTINCT key FROM t;
```

场景二：**group by key** 包含所有列

不推荐写法：

```
SELECT * FROM t GROUP BY key, value; -- t has columns key and value
```

虽然 MaxCompute2.0 不会报错，但推荐改为：

```
SELECT DISTINCT key, value FROM t;
```

bad.escape

错误的 escape 序列问题。

按照 MaxCompute 文档的规定，在 string literal 中应该用反斜线加三位8进制数字表示从 0 到 127 的 ASCII 字符，例如：使用 \001， \002 表示 0，1 等。但目前\01，\0001 也被当作 \001 处理了。

这种行为会给新用户带来困扰，比如需要用“\0001”表示“\000”+“1”，便没有办法实现。同时对于从其他系统迁移而来的用户而言，会导致正确性错误。

 **说明：**

`\000`后面在加数字，如`\0001 - \0009`或`\00001`的写法可能会返回错误。

MaxCompute2.0 会解决此问题，需要 script 作者将错误的序列进行修改，示例如下：

错误写法：

```
SELECT split(key, "\01"), value like "\0001" FROM t;
```

报错信息：

```
FAILED: ODPS-0130161:[1,19] Parse exception - unexpected escape
sequence: 01
ODPS-0130161:[1,38] Parse exception - unexpected escape sequence: 0001
```

正确改法：

```
SELECT split(key, "\001"), value like "\001" FROM t;
```

column.repeated.in.creation

create table 时列名重复的问题。

如果 create table 时列名重复，MaxCompute2.0 将会报错，示例如下：

错误写法：

```
CREATE TABLE t (a BIGINT, b BIGINT, a BIGINT);
```

报错信息：

```
FAILED: ODPS-0130071:[1,37] Semantic analysis exception - column
repeated in creation: a
```

正确改法：

```
CREATE TABLE t (a BIGINT, b BIGINT);
```

string.join.double

写 **JOIN** 条件时，等号的左右两边分别是 **String** 和 **Double** 类型。

出现上述情况，旧版 MaxCompute 会把两边都转成 Bigint，但会导致严重的精度损失问题，例如：1.1 = “1” 在连接条件中会被认为是相等的。但 MaxCompute2.0 会与 Hive 兼容转为 Double。

不推荐写法：

```
SELECT * FROM t1 JOIN t2 ON t1.double_value = t2.string_value;
```

warning 信息：

```
WARNING:[1,48]  implicit conversion from STRING to DOUBLE, potential
data loss, use CAST function to suppress
```

推荐改法：

```
select * from t1 join t2 on t.double_value = cast(t2.string_value as
double);
```

除以上改法外，也可使用用户期望的其他转换方式。

window.ref.prev.window.alias

Window Function 引用同级 **Select List** 中的其他 **Window Function Alias** 的问题。

示例如下：

如果 rn 在 t1 中不存在，错误写法如下：

```

SELECT row_number() OVER (PARTITION BY c1 ORDER BY c1) rn,
row_number() OVER (PARTITION by c1 ORDER BY rn) rn2
```

```
FROM t1;
```

报错信息：

```
FAILED: ODPS-0130071:[2,45] Semantic analysis exception - column rn
cannot be resolved
```

正确改法：

```
SELECT row_number() OVER (PARTITION BY c1 ORDER BY rn) rn2
FROM
(
SELECT c1, row_number() OVER (PARTITION BY c1 ORDER BY c1) rn
FROM t1
) tmp;
```

select.invalid.token.after.star

select * 后面接 **alias** 的问题。

Select 列表里面允许用户使用 * 代表选择某张表的全部列，但 * 后面不允许加 alias（即使 * 展开之后只有一列也不允许），新一代编译器将会对类似语法进行报错，示例如下：

错误写法：

```
select * as alias from dual;
```

报错信息：

```
FAILED: ODPS-0130161:[1,10] Parse exception - invalid token 'as'
```

正确改法：

```
select * from dual;
```

agg.having.ref.prev.agg.alias

有 **Having** 的情况下，**Select List** 可以出现前面 **Aggregate Function Alias** 的问题。示例如下：

错误写法：

```
SELECT count(c1) cnt,
sum(c1) / cnt avg
FROM t1
GROUP BY c2
```

```
HAVING cnt > 1;
```

报错信息：

```
FAILED: ODPS-0130071:[2,11] Semantic analysis exception - column cnt
cannot be resolved
ODPS-0130071:[2,11] Semantic analysis exception - column reference cnt
 should appear in GROUP BY key
```

其中 s、cnt 在源表 t1 中都不存在，但因为有 HAVING，旧版 MaxCompute 并未报错，

MaxCompute2.0 则会提示 column cannot be resolve，并报错。

正确改法：

```
SELECT cnt, s, s/cnt avg
FROM
(
SELECT count(c1) cnt,
sum(c1) s
FROM t1
GROUP BY c2
HAVING count(c1) > 1
) tmp;
```

order.by.no.limit

ORDER BY 后没有 LIMIT 语句的问题。

MaxCompute 默认 order by 后需要增加 limit 限制数量，因为 order by 是全量排序，没有 limit 时执

行性能较低。示例如下：

错误写法：

```
select * from (select *
from (select cast(login_user_cnt as int) as uv, '3' as shuzi
from test_login_cnt where type = 'device' and type_name = 'mobile') v
order by v.uv desc) v
order by v.shuzi limit 20;
```

报错信息：

```
FAILED: ODPS-0130071:[4,1] Semantic analysis exception - ORDER BY must
 be used with a LIMIT clause
```

正确改法：

在子查询 order by v.uv desc 中增加 limit。

另外，MaxCompute1.0 对于 view 的检查不够严格。比如在一个不需要检查 LIMIT 的 Projec (odps.sql.validate.orderby.limit=false) 中，创建了一个 View：

```
CREATE VIEW dual_view AS SELECT id FROM dual ORDER BY id;
```

若访问此 View：

```
SELECT * FROM dual_view;
```

MaxCompute1.0 不会报错，而 MaxCompute2.0 会报如下错误信息：

```
FAILED: ODPS-0130071:[1,15] Semantic analysis exception - while
resolving view xdj.xdj_view_limit - ORDER BY must be used with a LIMIT
 clause
```

generated.column.name.multi.window

使用自动生成的 **alias** 的问题。

旧版 MaxCompute 会为 Select 语句中的每个表达式自动生成一个 alias，这个 alias 会最后显示在 console 上。但是，它并不承诺这个 alias 的生成规则，也不承诺这个 alias 的生成规则会保持不变，所以不建议用户使用自动生成的 alias。

MaxCompute2.0 会对使用自动生成 alias 的情况给予警告，由于牵涉面较广，暂时无法直接给予禁止。

对于某些情况，MaxCompute 的不同版本间生成的 alias 规则存在已知的变动，但因为已有一些线上作业依赖于此类 alias，这些查询在 MaxCompute 版本升级或者回滚时可能会失败，存在此问题的用户，请修改您的查询，对于感兴趣的列，显式地指定列的别名。示例如下：

不推荐写法：

```
SELECT _c0 FROM (SELECT count(*) FROM dual) t;
```

建议改法：

```
SELECT c FROM (SELECT count(*) c FROM dual) t;
```

non.boolean.filter

使用了非 **boolean** 过滤条件的问题。

MaxCompute 不允许布尔类型与其他类型之间的隐式转换，但旧版 MaxCompute 会允许用户在某些情况下使用 Bigint 作为过滤条件。MaxCompute2.0 将不再允许，如果您的脚本中存在这样的过滤条件，请及时修改。示例如下：

错误写法：

```
select id, count(*) from dual group by id having id;
```

报错信息：

```
FAILED: ODPS-0130071:[1,50] Semantic analysis exception - expect a
BOOLEAN expression
```

正确改法：

```
select id, count(*) from dual group by id having id <> 0;
```

post.select.ambiguous

在 **order by、 cluster by、 distribute by、sort by** 等语句中，引用了名字冲突的列的问题。

旧版 MaxCompute 中，系统会默认选取 Select 列表中的后一列作为操作对象，MaxCompute2.0 将会进行报错，请及时修改。示例如下：

错误写法：

```
select a, b as a from t order by a limit 10;
```

报错信息：

```
FAILED: ODPS-0130071:[1,34] Semantic analysis exception - a is
ambiguous, can be both t.a or null.a
```

正确改法：

```
select a as c, b as a from t order by a limit 10;
```

本次推送修改会包括名字虽然冲突但语义一样的情况，虽然不会出现歧义，但是考虑到这种情况容易导致错误，作为一个警告，希望用户进行修改。

duplicated.partition.column

在 **query** 中指定了同名的 **partition** 的问题。

旧版 MaxCompute 在用户指定同名 partition key 时并未报错， 而是后一个的值直接覆盖了前一个，容易产生混乱。MaxCompute2.0 将会对此情况进行报错，示例如下：

错误写法一：

```
insert overwrite table partition (ds = '1', ds = '2')select ... ;
```

实际上，在运行时 ds = ‘1’ 被忽略。

正确改法：

```
insert overwrite table partition (ds = '2')select ... ;
```

错误写法二：

```
create table t (a bigint, ds string) partitioned by (ds string);
```

正确改法：

```
create table t (a bigint) partitioned by (ds string);
```

order.by.col.ambiguous

Select list 中 **alias** 重复，之后的 **Order by** 子句引用到重复的 **alias** 的问题。

错误写法：

```
SELECT id, id
FROM dual
ORDER BY id;
```

正确改法：

```
SELECT id, id id2
FROM dual
ORDER BY id;
```

需要去掉重复的 alias，Order by 子句再进行引用。

in.subquery.without.result

colx in subquery 没有返回任何结果，则 **colx** 在源表中不存在的问题。

错误写法：

```
SELECT * FROM dual
WHERE not_exist_col IN (SELECT id FROM dual LIMIT 0);
```

报错信息：

```
FAILED: ODPS-0130071:[2,7] Semantic analysis exception - column
not_exist_col cannot be resolved
```

ctas.if.not.exists

目标表语法错误问题。

如果目标表已经存在，旧版 MaxCompute 不会做任何语法检查，MaxCompute2.0 则会做正常的语法检查，这种情况会出现很多错误信息，示例如下：

错误写法：

```
CREATE TABLE IF NOT EXISTS dual
AS
SELECT * FROM not_exist_table;
```

报错信息：

```
FAILED: ODPS-0130131:[1,50] Table not found - table meta_dev.
not_exist_table cannot be resolved
```

worker.restart.instance.timeout

旧版 MaxCompute UDF 每输出一条记录，便会触发一次对分布式文件系统的写操作，同时会向 Fuxi 发送心跳，如果 UDF 10 分钟没有输出任何结果，会得到如下错误提示：

```
FAILED: ODPS-0123144: Fuxi job failed - WorkerRestart errCode:252,
errMsg:kInstanceMonitorTimeout, usually caused by bad udf performance.
```

MaxCompute2.0 的 Runtime 框架支持向量化，一次会处理某一列的多行来提升执行效率。但向量化可能导致原来不会报错的语句（2 条记录的输出时间间隔不超过 10 分钟），因为一次处理多行，没有及时向 Fuxi 发送心跳而导致 timeout。

遇到这个错误，建议首先检查 UDF 是否有性能问题，每条记录需要数秒的处理时间。如果无法优化 UDF 性能，可以尝试手动设置 batch row 大小来绕开（默认为1024）：

```
set odps.sql.executionengine.batch.rowcount=16;
```

divide.nan.or.overflow

旧版 **MaxCompute** 不会做除法常量折叠的问题。

比如如下语句，旧版 MaxCompute 对应的物理执行计划如下：

```
EXPLAIN
SELECT IF(FALSE, 0/0, 1.0)
FROM dual;
In Task M1_Stg1:
    Data source: meta_dev.dual
    TS: alias: dual
      SEL: If(False, Divide(UDFToDouble(0), UDFToDouble(0)), 1.0)
```

```
        FS: output: None
```

由此可以看出，IF 和 Divide 函数仍然被保留，运行时因为 IF 第一个参数为 false，第二个参数 Divide 的表达式不需要求值，所以不会出现除零异常。

而 MaxCompute2.0 则支持除法常量折叠，所以会报错。如下所示：

错误写法：

```
SELECT IF(FALSE, 0/0, 1.0)
FROM dual;
```

报错信息：

```
FAILED: ODPS-0130071:[1,19] Semantic analysis exception - encounter
 runtime exception while evaluating function /, detailed message:
DIVIDE func result NaN, two params are 0.000000 and 0.000000
```

除了上述的 nan，还可能遇到 overflow 错误，比如：

错误写法：

```
SELECT IF(FALSE, 1/0, 1.0)
FROM dual;
```

报错信息：

```
FAILED: ODPS-0130071:[1,19] Semantic analysis exception - encounter
 runtime exception while evaluating function /, detailed message:
DIVIDE func result overflow, two params are 1.000000 and 0.000000
```

正确改法：

建议去掉 /0 的用法，换成合法常量。

CASE WHEN 常量折叠也有类似问题，比如：CASE WHEN TRUE THEN 0 ELSE 0/0，MaxCompute2.0 常量折叠时所有子表达式都会求值，导致除0错误。

CASE WHEN 可能涉及更复杂的优化场景，比如：

```
SELECT CASE WHEN key = 0 THEN 0 ELSE 1/key END
FROM (
SELECT 0 AS key FROM src
UNION ALL
```

```
SELECT key FROM src) r;
```

优化器会将除法下推到子查询中，转换类似于：

```
M (
SELECT CASE WHEN 0 = 0 THEN 0 ELSE 1/0 END c1 FROM src
UNION ALL
SELECT CASE WHEN key = 0 THEN 0 ELSE 1/key END c1 FROM src) r;
```

报错信息：

```
FAILED: ODPS-0130071:[0,0] Semantic analysis exception - physical plan
 generation failed: java.lang.ArithmeticException: DIVIDE func result
overflow, two params are 1.000000 and 0.000000
```

其中 UNION ALL 第一个子句常量折叠报错，建议将 SQL 中的 CASE WHEN 挪到子查询中，并去掉无用的 CASE WHEN 和去掉/0用法：

```
SELECT c1 END
FROM (
SELECT 0 c1 END FROM src
UNION ALL
SELECT CASE WHEN key = 0 THEN 0 ELSE 1/key END) r;
```

small.table.exceeds.mem.limit

旧版 MaxCompute 支持 Multi-way Join 优化，多个 Join 如果有相同 Join Key，会合并到一个 Fuxi Task 中执行，比如下面例子中的 J4_1_2_3_Stg1：

```
EXPLAIN
SELECT t1.*
FROM t1 JOIN t2 ON t1.c1 = t2.c1
JOIN t3 ON t1.c1 = t3.c1;
```

旧版 MaxCompute 物理执行计划：

```
In Job job0:
root Tasks: M1_Stg1, M2_Stg1, M3_Stg1
J4_1_2_3_Stg1 depends on: M1_Stg1, M2_Stg1, M3_Stg1

In Task M1_Stg1:
    Data source: meta_dev.t1

In Task M2_Stg1:
    Data source: meta_dev.t2

In Task M3_Stg1:
    Data source: meta_dev.t3

In Task J4_1_2_3_Stg1:
    JOIN: t1 INNER JOIN unknown INNER JOIN unknown
```

```
        SEL: t1._col0, t1._col1, t1._col2
            FS: output: None
```

如果增加 MapJoin hint，旧版 MaxCompute 物理执行计划不会改变。也就是说对于旧版 MaxCompute 优先应用 Multi-way Join 优化，并且可以忽略用户指定 MapJoin hint。

```
EXPLAIN
SELECT /*+mapjoin(t1)*/ t1.*
FROM t1 JOIN t2 ON t1.c1 = t2.c1
JOIN t3 ON t1.c1 = t3.c1;
```

旧版 MaxCompute 物理执行计划同上。

MaxCompute2.0 Optimizer 会优先使用用户指定的 MapJoin hint，对于上述例子，如果 t1 比较大的话，会遇到类似错误：

```
FAILED: ODPS-0010000:System internal error - SQL Runtime Internal
 Error: Hash Join Cursor HashJoin_REL··· small table exceeds, memory
 limit(MB) 640, fixed memory used ···, variable memory used ···
```

对于这种情况，如果 MapJoin 不是期望行为，建议去掉 MapJoin hint。

sigkill.oom

同 small.table.exceeds.mem.limit，如果用户指定了 MapJoin hint，并且用户本身所指定的小表比较大。在旧版 MaxCompute 下有可能被优化成 Multi-way Join 从而成功。但在 MaxCompute 2.0 下，用户可能通过设定 odps.sql.mapjoin.memory.max 来避免小表超限的错误，但每个 MaxCompute worker 有固定的内存限制，如果小表本身过大，则 MaxCompute worker 会由于内存超限而被杀掉，错误类似于：

```
Fuxi job failed - WorkerRestart errCode:9,errMsg:SigKill(OOM), usually
 caused by OOM(outof memory).
```

这里建议您去掉 MapJoin hint，使用 Multi-way Join。

wm_concat.first.argument.const

聚合函数 中关于 WM_CONCAT 的说明，一直要求 WM_CONCAT 第一个参数为常量，旧版 MaxCompute 检查不严格，比如源表没有数据，就算 WM_CONCAT 第一个参数为 ColumnReference，也不会报错。

```
函数声明：
string wm_concat(string separator, string str)
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。
```

MaxCompute2.0，会在 plan 阶段便检查参数的合法性，假如 WM_CONCAT 的第一个参数不是常量，会立即报错。示例如下：

错误写法：

```
SELECT wm_concat(value, ',') FROM src GROUP BY value;
```

报错信息：

```
FAILED: ODPS-0130071:[0,0] Semantic analysis
 exception - physical plan generation failed:
 com.aliyun.odps.lot.cbo.validator.AggregateCallValidator
$AggregateCallValidationException: Invalid argument type - The first
 argument of WM_CONCAT must be constant string.
```

pt.implicit.convertion.failed

srcpt 是一个分区表，并有两个分区：

```
CREATE TABLE srcpt(key STRING, value STRING) PARTITIONED BY (pt STRING
);
ALTER TABLE srcpt ADD PARTITION (pt='pt1');
ALTER TABLE srcpt ADD PARTITION (pt='pt2');
```

对于以上 SQL，String 类型 pt 列 IN INT 类型常量，都会转为 Double 进行比较。即使 Project 设置了 odps.sql.udf.strict.mode=true，旧版 MaxCompute 不会报错，所有 pt 都会过滤掉，而 MaxCompute2.0 会直接报错。示例如下：

错误写法：

```
SELECT key FROM srcpt WHERE pt IN (1, 2);
```

报错信息：

```
FAILED: ODPS-0130071:[0,0] Semantic analysis exception - physical
plan generation failed: java.lang.NumberFormatException: ODPS-0123091
:Illegal type cast - In function cast, value 'pt1' cannot be casted
from String to Double.
```

建议避免 String 分区列和 INT 类型常量比较，将 INT 类型常量改成 String 类型。

having.use.select.alias

SQL 规范定义 Group by + Having 子句是 Select 子句之前阶段，所以 Having 中不应该使用 Select 子句生成的 Column alias，示例如下：

错误写法：

```
SELECT id id2 FROM DUAL GROUP BY id HAVING id2 > 0;
```

报错信息：

```
FAILED: ODPS-0130071:[1,44] Semantic analysis exception - column id2
 cannot be resolvedODPS-0130071:[1,44] Semantic analysis exception -
column reference id2 should appear in GROUP BY key
```

其中 id2 为 Select 子句中新生成的 Column alias，不应该在 Having 子句中使用。

dynamic.pt.to.static

MaxCompute2.0 动态分区某些情况会被优化器转换成静态分区处理，示例如下：

```
INSERT OVERWRITE TABLE srcpt PARTITION(pt) SELECT id, 'pt1' FROM dual;
```

会被转化成

```
INSERT OVERWRITE TABLE srcpt PARTITION(pt='pt1') SELECT id FROM dual;
```

如果用户指定的分区值不合法，比如错误的使用了'${bizdate}'，MaxCompute2.0 语法检查阶段便会报错。详情请参见 *MaxCompute* 分区值定义说明。

错误写法：

```
INSERT OVERWRITE TABLE srcpt PARTITION(pt) SELECT id, '${bizdate}'
FROM dual LIMIT 0;
```

报错信息：

```
FAILED: ODPS-0130071:[1,24] Semantic analysis exception - wrong
columns count 2 in data source, requires 3 columns (includes dynamic
partitions if any)
```

旧版 MaxCompute 因为 LIMIT 0，SQL 最终没有输出任何数据，动态分区不会创建，所以最终不报错。

lot.not.in.subquery

In subquery 中 null 值的处理问题。

在标准 SQL 的 IN 运算中，如果后面的值列表中出现 null，则返回值不会出现 false，只可能是 null 或者 true。如 1 in (null, 1, 2, 3) 为 true，而 1 in (null, 2, 3) 为 null，null in (null, 1, 2, 3) 为 null。同理 not in 操作在列表中有 null 的情况下，只会返回 false 或者 null，不会出现 true。

MaxCompute2.0 会用标准的行为进行处理，收到此提醒的用户请注意检查您的查询，IN 操作中的子查询中是否会出现空值，出现空值时行为是否与您预期相符，如果不符合预期请做相应的修改。示例如下：

```
select * from t where c not in (select accepted from c_list);
```

若 accepted 中不会出现 null 值，则此问题可忽略。若出现空值，则 c not in (select accepted from c_list) 原先返回 true，则新版本返回 null。

正确改法：

```
select * from t where c not in (select accepted from c_list where
accepted is not null)
```

5 长周期指标的计算优化方案

实验背景

电子商务公司（如淘宝）对用户数据分析的角度和思路可谓是应有尽有、层出不穷，所以在电商数据仓库和商业分析场景中，经常需要计算最近 N 天的访客数、购买用户数、老客数等类似的指标。

这些指标有一个共同点：都需要根据用户在电商平台上（或网上店铺）一段时间积累的数据进行计算（这里讨论的前提是数据都存储在 MaxCompute 上）。

一般情况下，这些指标的计算方式就是从日志明细表中计算就行了，如下代码计算商品最近 30 天的访客数：

```
select item_id --商品id
  ,count(distinct visitor_id) as ipv_uv_1d_001
from 用户访问商品日志明细表
where ds <= ${bdp.system.bizdate}
and ds >=to_char(dateadd(to_date(${bdp.system.bizdate},'yyyymmdd'),-29
,'dd'),'yyyymmdd')
 group by item_id;
```

说明：

代码中的变量都是DataWorks的调度变量，仅适用于DataWorks的调度任务。为了方便后文不再提醒。

当每天的日志量很大时，上面代码存在一个严重的问题，需要的 **Map Instance** 个数太多，甚至会超过 **99999** 个 Instance 个数的限制，Map Task 就没有办法顺利执行，更别说后续的操作了。

为什么 Instance 个数需要那么多呢？是因为每天的日志数据很大，30 天的数据量更是惊人。此时 Select 操作需要大量的 Map Instance，结果超过了 Instance 的上限，导致代码无法运行。

实验目的

如何计算长周期的指标，又不影响性能呢？通常有以下两种思路：

- 多天汇总的问题根源是数据量的问题，如果把数据量给降低了，便可解决此问题。
- 减少数据量最直接的办法是把每天的数据量都给减少，因此需要构建临时表，对 1d 的数据进行轻度汇总，这样便可去掉很多重复数据，减少数据量。

实验方案

操作步骤

1. 构建中间表，每天汇总一次。

   比如对于上面的例子，可以构建一个 item_id+visitor_id 粒度的中间表。即构建item_id+visitior_id 粒度的日汇总表，记作 A。如下所示：

```
insert overwrite table mds_itm_vsr_xx(ds='${bdp.system.bizdate} ')
select item_id,visitor_id,count(1) as pv
  from
  (
  select item_id,visitor_id
  from 用户访问商品日志明细表
  where ds =${bdp.system.bizdate}
  group by item_id,visitor_id
  )a;
```

2. 计算多天的数据，依赖中间表进行汇总。

   对 A 进行 30 天的汇总，如下所示：

```
select item_id
       ,count(distinct visitor_id) as uv
       ,sum(pv) as pv
  from mds_itm_vsr_xx
  where ds <= '${bdp.system.bizdate} '
  and ds >= to_char(dateadd(to_date('${bdp.system.bizdate} ','
yyyymmdd'),-29,'dd'),'yyyymmdd')
  group by item_id;
```

影响及思考

上面讲述的方法，对每天的访问日志明细数据进行单天去重，从而减少了数据量，提高了性能。缺点是每次计算多天的数据的时候，都需要读取 N 个分区的数据。

那么是否有一种方式，不需要读取 N 个分区的数据，而是把 N 个分区的数据压缩合并成一个分区的数据，让一个分区的数据包含历史数据的信息呢？

业务上是有类似场景的，可以通过 **增量累计方式计算长周期指标**。

场景示例

求最近 1 天店铺商品的老买家数。老买家数的算法定义为：过去一段时间有购买的买家（比如过去 30 天）。

一般情况下，老买家数计算方式如下所示：

```
select item_id --商品id
       ,buyer_id as old_buyer_id
from 用户购买商品明细表
```

```
where ds < ${bdp.system.bizdate}
and ds >=to_char(dateadd(to_date(${bdp.system.bizdate},'yyyymmdd'),-29
,'dd'),'yyyymmdd')
group by item_id
        ,buyer_id;
```

改进思路：

- 维护一张店铺商品和买家购买关系的维表记作表 A，记录买家和店铺的购买关系，以及第一次购买时间，最近一次购买时间，累计购买件数，累计购买金额等信息。
- 每天使用最近 1 天的支付明细日志更新表 A 的相关数据。
- 计算老买家时，最需要判断最近一次购买时间是否是 30天 之内就行了，从而做到最大程度上的数据关系对去重，减少了计算输入数据量。

6 计算长尾调优

长尾问题是分布式计算中最常见的问题之一，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，整个任务需要等最慢的节点完成才能结束。

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 单独运行最重的那份工作。本文将为您介绍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

问题原因：

Join 出现长尾，是因为 Join 时出现某个 Key 里的数据特别多的情况。

解决办法：

您可以从以下三方面进行考虑：

- 排除两张表都是小表的情况，若两张表里有一张大表和一张小表，可以考虑使用 Mapjoin，对小表进行缓存，具体的语法和说明请参见 *Select* 操作。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

问题原因：

Group By Key 出现长尾，是因为某个 Key 内的计算量特别大。

解决办法：

您可以通过以下两种方法解决：

- 可对 SQL 进行改写，添加随机数，把长 Key 进行拆分。如下所示：

```
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，就变成如下语句：

```
-- 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;
```

由上可见，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 2 个步骤的处理，整体的时间消耗可能反而有所减少。

 说明：

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。

- 使用通用的优化策略 — 系统参数，设置如下：

```
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。您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

对于 Distinct，上述 Group By 长尾时，把长 Key 进行拆分的策略已经不生效了。对这种场景，您可以考虑通过其他方式解决。

解决办法：

```
--原始SQL,不考虑Uid为空
SELECT COUNT(uid) AS Pv
    , COUNT(DISTINCT uid) AS Uv
FROM UserLog;
```

可以改写成如下语句：

```
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

问题原因：

- 动态分区功能为了整理小文件，会在最后启一个 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。
- 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。

解决办法：

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 的示例中，已提到这种做法。通过 Combiner，减少 Mapper 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。

说明：

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时，便不能直接在 Combiner 里把 (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，日志里突然打出如下的内容（+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，但仍然不能解决所有问题。有时碰到的长尾问题，还需要从业务角度上去考虑是否有更好的解决方法，示例如下：

- 实际数据可能包含非常多的噪音。比如：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。比如：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。

7 分区剪裁合理性评估

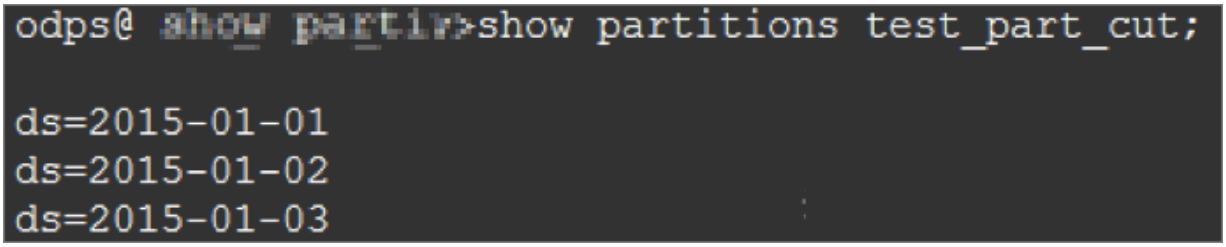
背景及目的

MaxCompute的 分区表 是指在创建表时指定分区空间，即指定表内的某几个字段作为分区列。使用数据时，如果指定了需要访问的分区名称，则只会读取相应的分区，避免全表扫描，提高处理效率，降低费用。

分区剪裁是指对分区列指定过滤条件，使得 SQL 执行时只用读取表的部分分区数据，避免全表扫描引起的数据错误及资源浪费。看起来非常简单，但是实际上经常会出现分区失效的情况，本文将通过示例为您介绍一些常见问题的解决方案。

问题示例

测试表 test_part_cut 的分区，如下图所示：

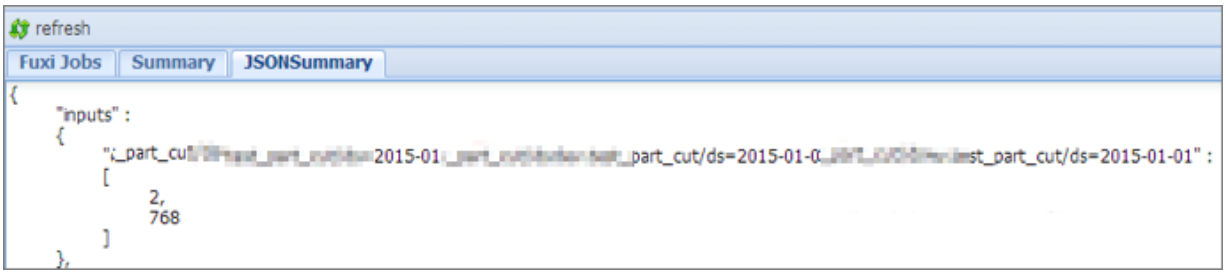


执行以下 SQL 代码：

```
select count(*)
from test_part_cut
where ds= bi_week_dim('20150102');

--其中为bi_week_dim自定义函数:返回格式为 （年,第几周）：
--如果是正常日期，判断日期是所传入参数中年份所属周，以周四为一周的起始日期，如果碰到20140101因为属于周三所以算在2013年最后一周返回2013,52。而20150101则返回是2015,1。
--如果是类似20151231是周四又恰逢与20160101在同一周，则返回2016,1。
```

bi_week_dim('20150102')的返回结果是 2015,1，不符合表 test_part_cut 的分区值，通常我们会认为上面的 SQL 不会读任何分区，而实际情况却是 该 **SQL** 读了表 **test_part_cut** 的所有分区，LogView 如下图所示：



从上图可以看出，该 SQL 在执行的时候读取了表 test_part_cut 的所有分区。

由上述示例可见，分区剪裁使用尽管简单，但也容易出错。因此，本文将从以下两方面进行介绍：

- 判断 SQL 中分区剪裁是否生效。
- 了解常见的导致分区剪裁失效的场景。

判断分区剪裁是否生效

通过 explain 命令查看 SQL 的执行计划，用于发现 SQL 中的分区剪裁是否生效。

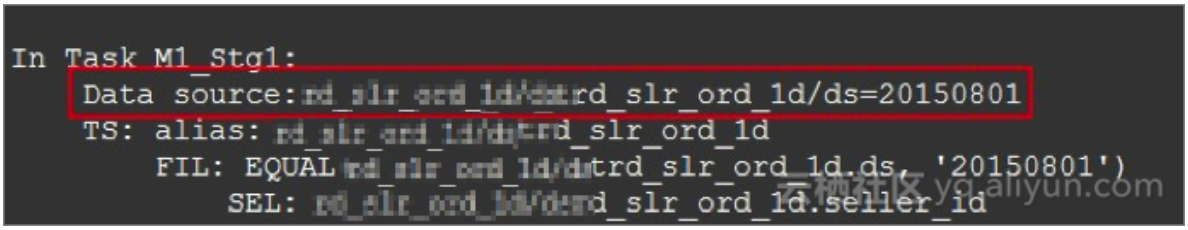
- 分区剪裁未生效的效果。

```
explain
select seller_id
from xxxxx_trd_slr_ord_1d
where ds=rand();
```

看上图中红框的内容，表示读取了表 xxxxx_trd_slr_ord_1d 的 1344 个分区，即该表的所有分区。

- 分区剪裁生效的效果。

```
explain
select seller_id
from xxxxx_trd_slr_ord_1d
where ds='20150801';
```



看上图中红框的内容，表示只读取了表 xxxxx_trd_slr_ord_1d 的 20150801 的分区。

分区剪裁失效的场景分析

分区剪裁在使用自定义函数或者部分系统函数的时候会失效，在 Join 关联时的 Where 条件中也有可能会失效。下面针对这两种场景分别举例说明。

自定义函数导致分区剪裁失效

当分区剪裁的条件中使用了用户自定义函数，则分区剪裁会失效，即使是使用系统函数也可能会导致分区剪裁失效。所以，对于分区值的限定，如果使用了非常规函数需要用 explain 命令通过查看执行计划，确定分区剪裁是否已经生效。

```
explain
select ...
from xxxxx_base2_brd_ind_cw
where ds = concat(SPLIT_PART(bi_week_dim(' ${bdp.system.bizdate}'),
 ',', 1), SPLIT_PART(bi_week_dim(' ${bdp.system.bizdate}'), ',', 2))
```

可以看出上面的 SQL 因为分区剪裁使用了用户自定义的函数导致全表扫描。

Join 使用时分区剪裁失效

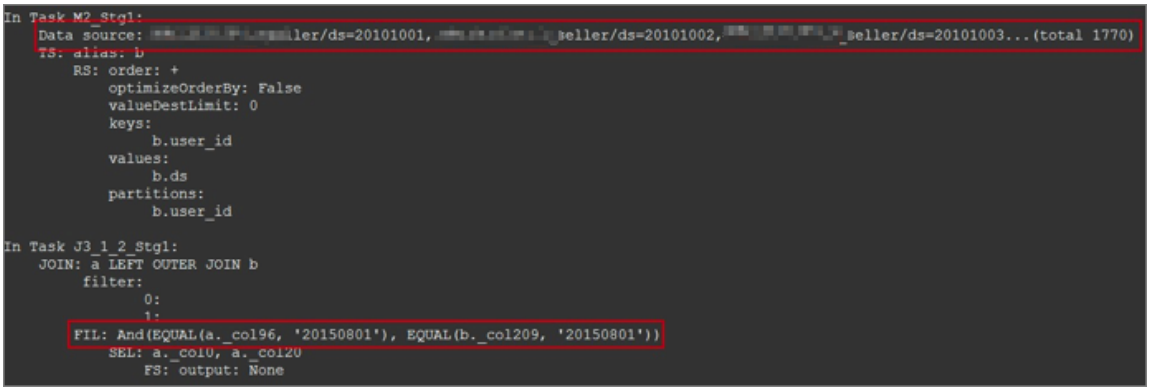
在 SQL 语句中，使用 Join 进行关联时，如果分区剪裁条件放在 where 中，则分区剪裁会生效，如果放在 on 条件中，从表的分区剪裁会生效，主表则不会生效。下面针对三种 Join 具体说明。

- **Left Outer Join**

  — 分区剪裁条件均放在 on 中

```
explain
select a.seller_id
    ,a.pay_ord_pbt_1d_001
from xxxxx_trd_slr_ord_1d a
left outer join
     xxxxx_seller b
on a.seller_id=b.user_id
and a.ds='20150801'
```

```
and b.ds='20150801';
```



由上图可见，左表进行全表扫描，只有右表的分区裁剪有效果。

- 分区剪裁条件均放在 where 中

```
explain
select a.seller_id
    ,a.pay_ord_pbt_1d_001
from xxxxx_trd_slr_ord_1d a
left outer join
    xxxxx_seller b
on a.seller_id=b.user_id
where a.ds='20150801'
```

```
and b.ds='20150801';
```

```
In Task M2_Stg1:
    Data source:[illegible]seller/ds=20150801
    TS: alias: b
        FIL: EQUAL(b.ds, '20150801')
            RS: order: +
                optimizeOrderBy: False
                valueDestLimit: 0
                keys:
                     b.user_id
                values:
                partitions:
                     b.user_id

In Task J3_1_2_Stg1:
    JOIN: a LEFT OUTER JOIN unknown
         filter:
                0:
                1:
        SEL: a._col0, a._col20
            FS: output: None

In Task M1_Stg1:
    Data source:[illegible]trd_slr_ord_1d/ds=20150801
    TS: alias: a
```

由上图可见，两张表的分区裁剪都有效果。

- **Right Outer Join**

  与 Left Outer Join 类似，分区剪裁条件如果放在 on 中则只有 Right Outer Join的左表生效，如果放在 where 中，则两张表都会生效。

- **Full Outer Join**

  分区剪裁条件只有都放在 where 中才会生效，放在 on 中则都不会生效。

影响及思考

- 分区剪裁如果失效会影响比较大，且用户不容易发现。因此，分区剪裁失效最好在代码提交的时候发现比较合适。
- 对于用户自定义函数不能用于分区剪裁的问题，需要平台再深入思考解决方法。

8 快速掌握SQL写法

本文通过课程实践的方式，为您介绍 MaxCompute SQL，让您快速掌握 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 进行阅读。

数据集准备

这里选择大家比较熟悉的 Emp/Dept 表做为数据集。为方便大家操作，特提供相关的 MaxCompute 建表语句和数据文件（*emp* 表数据文件，*dept* 表数据文件），您可自行在 MaxCompute 项目上创建表并上传数据。

创建 emp 表的 DDL 语句，如下所示：

```
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 表的 DDL 语句，如下所示：

```
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

- 使用 Group by，那么 Select 的部分要么是分组项，要么就得是聚合函数。
- Order by 后面必须加 Limit n。
- Select 表达式中不能用子查询，可以改写为 Join。
- Join 不支持笛卡尔积，以及 MapJoin 的用法和使用场景。
- Union all 需要改成子查询的格式。
- 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，您需要使用 Join 进行改写。如下所示：

```
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，如下所示：

```
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，如下所示：

```
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，如下所示：

```
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，如下所示：

```
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）。

SQL 语句如下所示：

```
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。

SQL 语句如下所示：

```
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;
```

9 与标准SQL的主要区别及解决方法

本文将从习惯使用关系型数据库 SQL 用户的实践角度出发，列举用户在使用 MaxCompute SQL 时比较容易遇见的问题。具体的 MaxCompute SQL 语法请参见 *SQL* 概述。

MaxCompute SQL 基本区别

应用场景

- 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，可以先把数据存为 String 类型，然后使用 UDF 来实现对应的计算。

数据类型转换

- 为防止出现各种预期外的错误，如果有 2 个不同的字段类型需要做 Join，建议您先把类型转好后再 Join，同时更容易维护代码。
- 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见 日期函数 > *TO_DATE*。

DDL 的区别及解法

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请参见 常见问题。
- 支持增加列，但是不支持删除列以及修改列的数据类型，请参见 *SQL* 常见问题。

DML 的区别及解法

INSERT

- 语法上最直观的区别是：Insert into/overwrite 后面有个关键字 **Table**。
- 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，如果有需要请参见 更新和删除数据。

SELECT

- 输入表的数量不能超过 *16* 张。
- Group by 查询中的 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 中的数据有多条，不使用聚合函数的话就没办法展示。
- MaxCompute不支持Group by cube（group by with rollup），可以通过union来模拟，比如 `select k1, null, sum(v) from t group by k1 union all select k1, k2, sum(v) from t group by k1, k2;`。

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，解决办法请参见 如何使用*Not In*。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，从而提升查询性能。

SQL 返回 10000 条

- MaxCompute 限制了单独执行 Select 语句时返回的数据条数，具体配置请参见 其他操作，设置上限为 1 万。如果需要查询的结果数据条数很多，请参见 如何获取所有数据，配合 Tunnel 命令获取全部数据。

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。详情请参见 如何解决*Join*报错。

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。详情请参见 *MaxCompute* 查询数据的排序。

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。