MaxCompute

最佳实践

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;
}
```

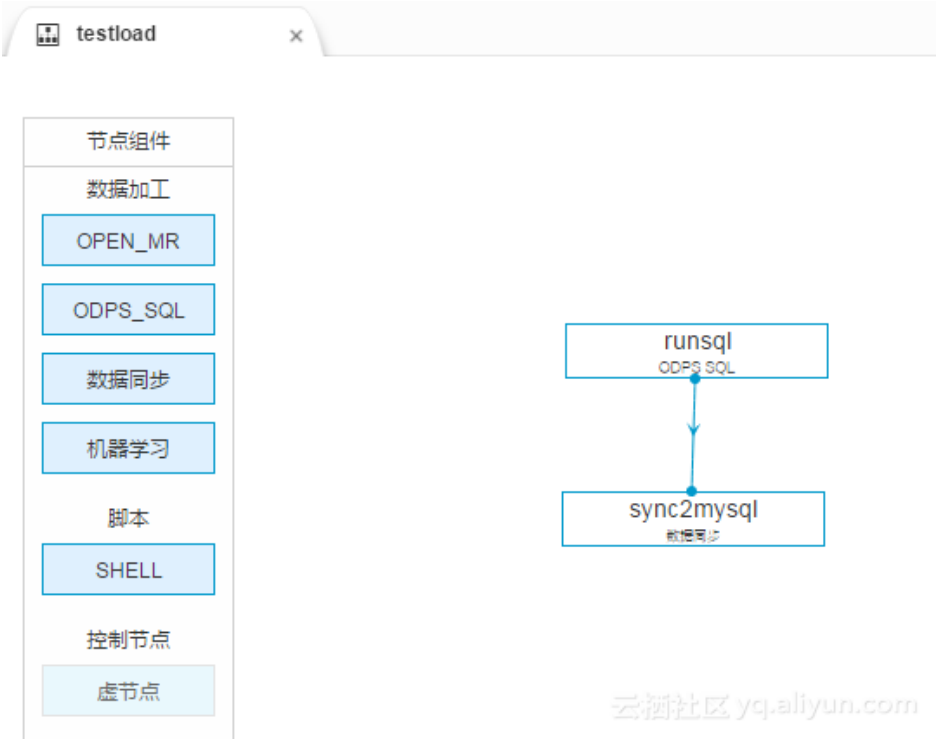
大数据开发套件的数据同步方式导出

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。

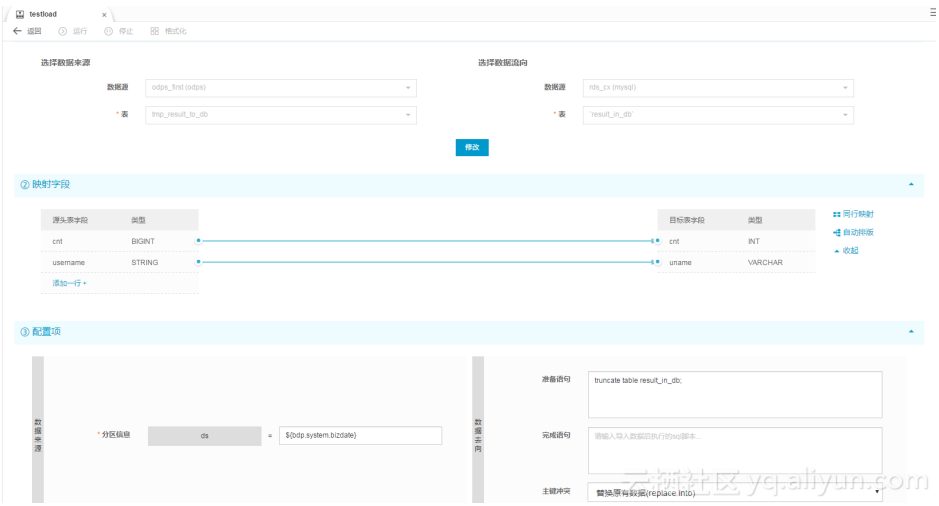
我们将用一个简单例子介绍如何通过大数据开发套件运行SQL并配置数据同步任务完成数据生成和导出需求。

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。

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。

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 不会抱错，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 MaxCompiute 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：

```
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 MaxCompute2.0 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(out of 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 resolved
ODPS-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（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)
```

文章来自云栖社区。

背景

在电子商务公司（如淘宝），对用户的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里，经常需要计算最近N天访客数、最近N天的购买用户数、老客数等等类似的指标。

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，如下代码计算商品的最近30天访客数：

```
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;
```

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？原因：每天的日志数据很大，30天的数据量更是惊人。这时候Select 操作需要大量的Map Instance，结果查过了Instance的上限，代码无法运行

目的

如何计算长周期的指标，又不影响性能？

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，就可以解决这个问题了。

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，这样就能去掉很多重复数据，减少数据量。

方案

构建中间表，每天汇总一次，比如对于上面的例子，构建一个item_id+visitor_id粒度的中间表。

计算多天的数据，依赖中间表进行汇总。

例子如下：

- 构建item_id+visitior_id粒度的日汇总表，记作A

```
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;
```

- 对A进行30天汇总**

```
  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，都要N个分区的数据。那么能不能有一种方式，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。

业务上是有类似场景的，有方式如下：**增量累计方式计算长周期指标。**

例子：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：

```
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。

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！