**Klustron 执行计划说明和解读**

**注意：**

 如无特别说明，文中的版本号可以使用任何已发布版本的版本号代替。所有已发布版本详见：<http://doc.klustron.com/zh/Release_notes.html>。

**本文内容：**

在Klustron中，执行计划是指优化器生成的一份详细的SQL执行的步骤。执行计划可以帮助我们了解SQL的执行过程，包括SQL的执行顺序、使用的索引、使用的连接方式等等。通过分析执行计划，我们可以找到查询性能瓶颈，并进行优化。在本文中，我们通过具体的案例来解读在Klustron中运行SQL的执行计划。

* 1. **执行计划的生成**

Klustron执行计划的生成过程可以分为以下几个步骤：

1. 语法分析：将查询语句解析成一棵语法树。
2. 语义分析：对语法树进行语义分析，包括表名解析、列名解析、类型检查等等。
3. 优化器：对查询语句进行分布式查询优化，生成最优的执行计划。
4. 执行：将优化后的执行计划转换成可执行的代码，然后通过与后端存储 shard 做交互来完成分布式查询执行。交互的方法就是根据 SQL 语句的需要和数据表分区在后端 shard 的分布信息，为相关的后端存储 shard 生成 SQL 语句。

在这个过程中，优化器是最关键的一步。优化器会根据查询语句的特点和相关表的统计信息选择最优的执行计划。执行计划的优化过程非常复杂，包括选择最优的连接方式、选择最优的索引、选择最优的排序方式等等。在优化器的帮助下，Klustron可以处理非常复杂的SQL语句，并且保证SQL的性能。

* 1. **Explain语法**

在Klustron 中，EXPLAIN 命令可以输出SQL 语句的查询计划，具体语法如下：

EXPLAIN [ ( option [, ...] ) ] statement

EXPLAIN [ ANALYZE ] [ VERBOSE ] statement

where option can be one of:

ANALYZE [ boolean ]

VERBOSE [ boolean ]

COSTS [ boolean ]

BUFFERS [ boolean ]

TIMING [ boolean ]

SUMMARY [ boolean ]

FORMAT { TEXT | XML | JSON | YAML }

其中：

ANALYZE 选项为TRUE 会实际执行SQL，并获得相应的查询计划，默认为FALSE。如果优化一些修改数据的SQL 需要真实的执行但是不能影响现有的数据，可以放在一个事务中，分析完成后可以直接回滚。

VERBOSE 选项为TRUE 会显示查询计划的附加信息，默认为FALSE。附加信息包括查询计划中每个节点（后面具体解释节点的含义）输出的列（Output），表的SCHEMA 信息，函数的SCHEMA 信息，表达式中列所属表的别名，被触发的触发器名称等。

COSTS 选项为TRUE 会显示每个计划节点的预估启动代价（找到第一个符合条件的结果的代价）和总代价，以及预估行数和每行宽度，默认为TRUE。

TIMING 选项为TRUE 会显示每个计划节点的实际启动时间和总的执行时间，默认为TRUE。该参数只能与ANALYZE 参数一起使用。因为对于一些系统来说，获取系统时间需要比较大的代价，如果只需要准确的返回行数，而不需要准确的时间，可以把该参数关闭。

SUMMARY 选项为TRUE 会在查询计划后面输出总结信息，例如查询计划生成的时间和查询计划执行的时间。当ANALYZE 选项打开时，它默认为TRUE。

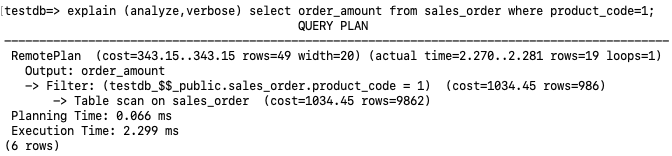
FORMAT 指定输出格式，默认为TEXT。各个格式输出的内容都是相同的，其中XML | JSON | YAML 更有利于我们通过程序解析SQL 语句的查询计划，为了更有利于阅读，我们下文的例子都是使用TEXT 格式的输出结果。

一般采用下面的方式执行SQL并生成详细的SQL执行信息

explain analyze verbose select … ;

* 1. **执行计划的解读**

Klustron是存算分离的架构，SQL是在计算节点解析生成执行计划，然后在存储节点访问数据。



执行计划输出主要包含下面的几个部分：

1. 访问方式。RemotePlan是Klustron新增的查询计划节点，用于获取存储节点上的用户数据。从上面执行计划RemotePlan可以看出这条SQL完全是在存储节点执行的，在存储节点的数据访问方式是Table scan。在存储节点的过滤条件为testdb\_$$\_public.sales\_order.product\_code = 1。
2. 访问成本。cost=343.15..343.15执行查询的成本估算为343.15。
3. 返回行数。查询估计返回的行数为rows=49, 实际返回的行数为rows=19。而下面的Table scan估计sales\_order全表的行数为rows=9862, 在执行过滤之后(product\_code=1)估计返回的行数为rows=986。这里可以看到RemotePlan的估算值rows=49和执行过滤之后预估的行数rows=986不太匹配，这是因为RemotePlan的估计行数是由计算节点估算的，而RemotePlan下面的算子和代价的估算则是由存储节点MySQL完成的。并且在analyze情况下展示的MySQL的执行计划步骤不包含实际的执行时间。
4. 执行时间。Planning Time部分表示生成执行计划的时间为0.066ms，Execution部分表示真实的执行时间为2.299ms。

**3.1 全表扫描**

文本

中度可信度描述已自动生成

再分析之前的执行计划，其中在存储节点执行了Table scan on sales\_order，在全表扫描的同时进行Filter(product\_code=1)。实际执行时间为2.299ms，返回了19行记录。

**3.2 索引扫描**

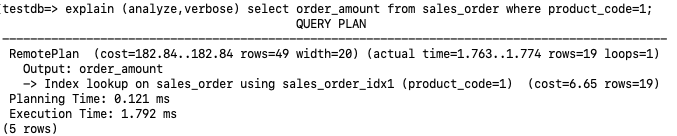
**3.2.1 主键索引/唯一索引的单行扫描**

文本

描述已自动生成

通过主键索引或者唯一索引访问单行记录时，在执行计划中会显示Rows fetched before execution。

**3.2.2 Index lookup（等值条件）**



从执行计划看，在存储节点执行Index lookup on sales\_order using sales\_order\_idx1。这次查询优化器估算值比较准(cost=6.65 rows=19), 最后执行时间为1.792ms。

**3.2.3 Index range scan（范围条件）**

图形用户界面

中度可信度描述已自动生成

在存储节点的sales\_order\_idx2上执行了索引的范围扫描。

**3.2.4 Index scan**

文本

低可信度描述已自动生成

由于返回的字段都包含在索引中，此时在存储节点执行了Index scan扫描了整个索引，省去了回表的步骤。

**3.3 Nested Loop Join**

文本

描述已自动生成

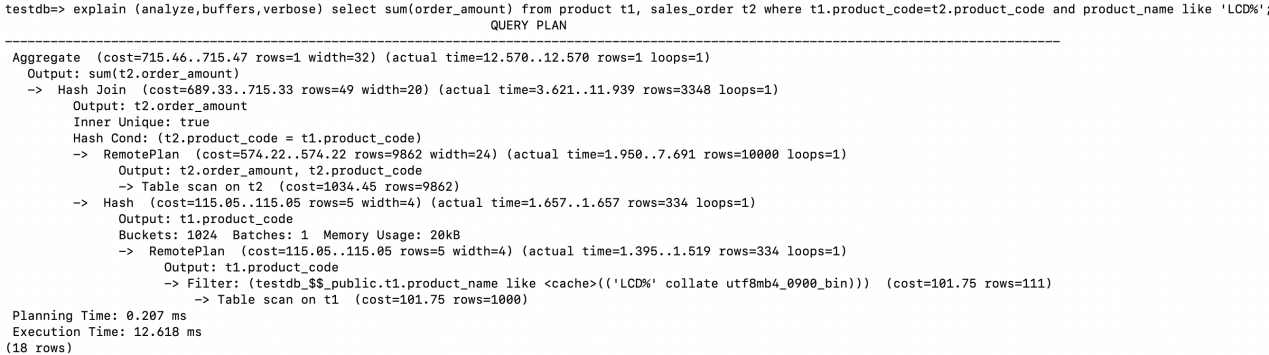
具体执行步骤如下：

1 Nested Loop的驱动表为product，其访问的方式为RemotePlan，在存储节点的主键索引上执行了Index scan，通过字段t1.product\_name like ‘LCD%’进行过滤，最后返回满足条件的4条记录(rows=4)给计算节点。

2 在计算节点拿着步骤1返回的每条记录的product\_code发给存储节点执行Remote SQL Remote SQL: SELECT `t2`.`order\_amount`,`t2`.`product\_code` FROM `testdb\_$$\_public`.`sales\_order` as t2 WHERE (? = `t2`.`product\_code`), 循环执行次数为4次(loops=4)，执行结果集同样也送回到计算节点。

1. 满足Join条件的结果集有81行(rows=81)
2. 最后在计算节点完成最后的SUM。

**3.4 Hash Join**



执行计划中驱动表的访问在Hash Join的下半部分，这个和之前的Nested Loop Join驱动表的访问有所不同。

1 先在存储节点执行Table scan on t1, 并且进行过滤（product\_name like ‘LCD%’），之后将结果集返回给计算节点。

2 在计算节点上根据Join的字段创建Hash table。Hash table的情况为：

Buckets: 1024 Batches: 1 Memory Usage: 20kB

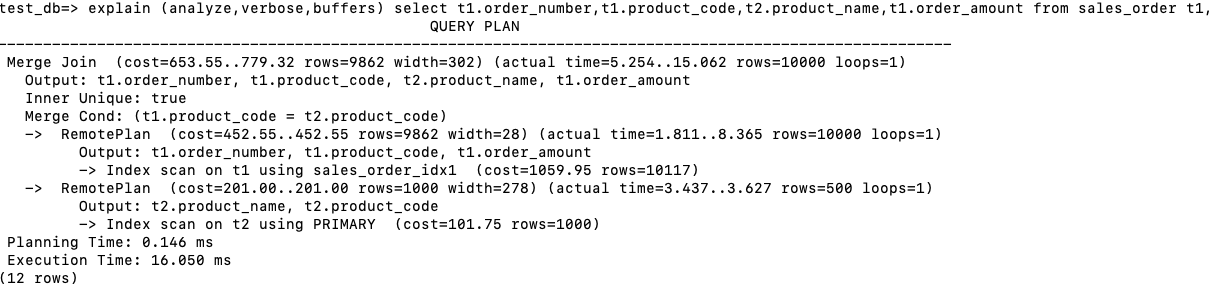
3 在存储节点执行Table scan to t2, 返回字段order\_amount和product\_code给计算节点。

4 在计算节点进行Hash Join，通过步骤2中创建的Hash table去probe步骤3中返回的结果集。Hash Cond: (t2.product\_code = t1.product\_code)，返回满足join条件记录的order\_amount值。

5 最后执行sum(order\_amount)的聚集操作。

**3.5 Sort Merge Join**

通常情况下Hash Join的连接效率会比Sort Merge Join好，然而如果行源已经被排过序，在执行Sort Merge Join时不需要再排序了，这时Sort Merge Join的性能会优于Hash Join。下面两张表Join的字段都创建了索引，所以通过Index scan扫描的结果都是已经排好序的，直接可以用作Sort Merge Join的输入。

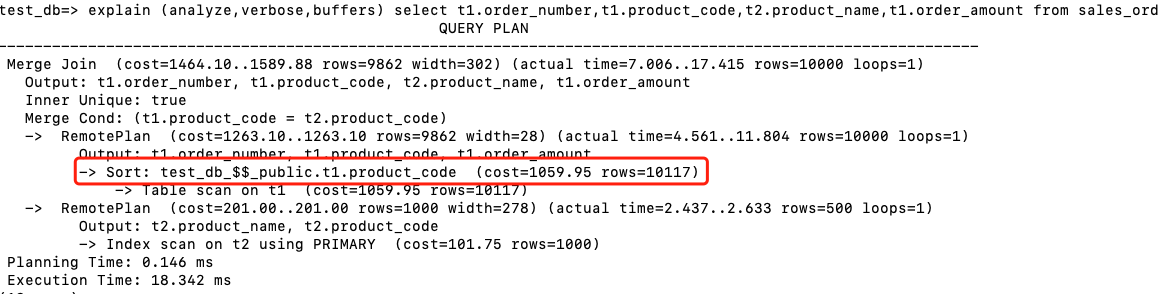


1 在存储节点执行Index scan on t1, 回表后返回product\_code,order\_amount和order\_number。

2 在存储节点执行 Index scan on t2, 回表后返回product\_code和product\_name。

3 在计算节点进行Merge Join。注意由于Index返回的结果都是根据product\_code排序的，所以省略了Sort步骤。

如果删除了其中一个表的product\_code索引，则出现了下面的执行计划，会多出了Sort排序的步骤。



**3.6 并行执行计划**

在Klustron中打开enable\_parallel\_remotescan之后后，计算节点的优化器不仅可以把扫描不同表分区的RemotePlan分配给多个子任务，从而让多个worker进程并行执行，还可以依据统计信息把扫描同一个表分区或者未分区的单表的RemotePlan按照需要扫描的行的范围拆分为多个子任务，这样即使RemotePlan 需要扫描的是非分区表也可以并行执行。

查看计算节点并行相关的参数：

表格

描述已自动生成

下面我们看看分布式并行查询的执行计划。

**3.6.1 非分区表并行扫描**

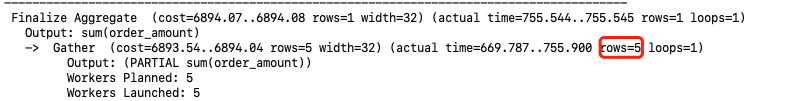
图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

1 我们看到Workers Planned:5 Workers Launched:5 , 其实总共分配了6个并行进程：Leader，Worker 0-4。通过Parallel RemotePlan最后的loops=6也可以看出6个并行。

2 每个并行进程在存储节点都执行了Index range scan on sales\_order using sales\_order\_idx1, 各自进行sum(order\_amount)，输出PARTIAL sum(order\_amount)的结果给计算节点。由于是部分结果集的汇总，所以这里多了”PARTIAL”的字样。

3 Leader进程Gather 其它5个sum(order\_amount)的结果，然后进行最后的汇总。



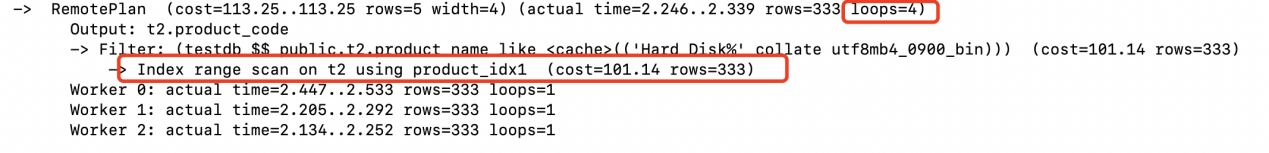
**3.6.2 分区表并行扫描**

文本

描述已自动生成

1 同样我们看到Workers Planned:3, Workers Launched:3，但是总共并行工作的进程有4个：Leader, Worker 0, Worker 1, Worker 2。另外从loops=4可以窥到一些端倪。

2 两张表Join的方式为Hash Join, 驱动表product是一个非分区表，在存储节点上4个并行的进程采用了Index range scan分别扫描了product表一次。满足条件的结果集有333行，返回给计算节点创建Hash Table。



3 Parallel Append说明采用并行的方式Index Scan了sales\_order的4个索引分区。将获取到满足条件的记录返回给计算节点。（2562+2506+2497+2435）

图片包含 表格

描述已自动生成

4 在计算节点并行完成Hash Join

图片包含 文本

描述已自动生成

5 Partial Aggregate 对每个并行worker的Hash Join返回的行做了聚集运算。

6 Gather收集每个worker进程返回的结果，leader进程执行Gather以及Gather之上的部分。Gather需要把结果集送给Finalize Aggregate从而用每个worker的Partial Aggregate 结果形成最终的输出。

文本

中度可信度描述已自动生成

并行执行的RemotePlan在worker进程中会各自连接到存储节点执行查询任务，为了得到一致的查询结果，这些worker进程在各自的连接中必须使用相同的快照，因此Klustron在存储节点层中增加了连接快照共享能力来配合计算节点执行并行的分布式查询计划。

**3.7 算子下推在执行计划中的展现**

在Klustron中支持将多种类型的算子下推到存储节点执行，参考上一节大表的并行执行的案例，算子下推可以充分利用底层多个存储节点的运算能力。下面通过具体执行计划来展现下推的功能。

**3.7.1 聚合函数下推**

聚合函数下推到存储节点执行。

图形用户界面, 文本, 应用程序

描述已自动生成

**3.7.2 Distinct算子下推**

下面执行计划将count(distinct customer\_number)都下推到存储节点执行

图形用户界面, 文本, 应用程序

描述已自动生成

**3.7.3 Limit和Order by下推**

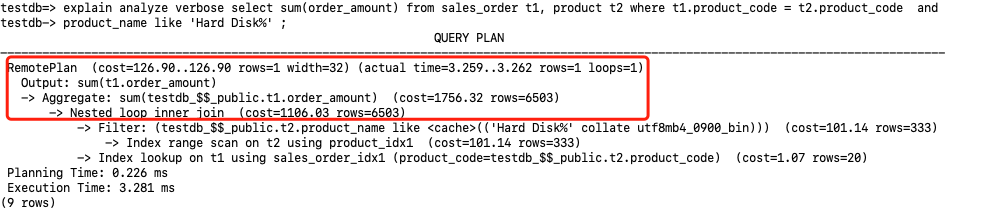
下面的执行计划中体现了order by order\_amount desc limit 10都下推到了存储节点执行

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

**3.7.4 Join下推**

下面执行计划中展示了将sales\_order和product两张表的Nested Loop Join的运算下推到存储节点执行



【END】