数据库内核月报 － 2018 / 11

mysql.taobao.org/monthly/2018/11/06

背景

在使用数据库时，经常会有开发者有这样的疑问：“我的表对应字段已经创建了索引，为什么这个SQL 语句执行还是这么慢？”虽然数据库SQL 执行慢有很多原因，但是对于PostgreSQL DBA 来说，好像有个共识，遇到用户慢SQL优化的问题，先拿EXPLAIN 命令查看下对应的查询计划，从而可以快速定位慢在哪里。这就引出了本文的主角－PostgreSQL 的EXPLAIN 命令。

EXPLAIN 语法

在PostgreSQL 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。
- BUFFERS 选项为TRUE 会显示关于缓存的使用信息，默认为FALSE。该参数只能与ANALYZE 参数一起使用。缓冲区信息包括共享块（常规表或者索引块）、本地块（临时表或者索引块）和临时块（排序或者哈希等涉及到的短期存在的数据块）的命中块数，更新块数，挤出块数。
- 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。

EXPLAIN 的输出结构

阅读到这里，我们已经知道了如何使用EXPLAIN 命令。接下来，我们将学习如何理解EXPLAIN 的输出，从而快速地定位问题。

以下面的输出为例（该例子选自PostgreSQL官方文档），我们来分析下EXPLAIN 命令输出的结构：

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;

                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------
-------------------------------------------------------------
 Sort  (cost=717.34..717.59 rows=101 width=488) (actual time=7.761..7.774 rows=100
loops=1)
   Sort Key: t1.fivethous
   Sort Method: quicksort  Memory: 77kB
   ->  Hash Join  (cost=230.47..713.98 rows=101 width=488) (actual
time=0.711..7.427 rows=100 loops=1)
         Hash Cond: (t2.unique2 = t1.unique2)
         ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244)
(actual time=0.007..2.583 rows=10000 loops=1)
         ->  Hash  (cost=229.20..229.20 rows=101 width=244) (actual
time=0.659..0.659 rows=100 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 28kB
               ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101
width=244) (actual time=0.080..0.526 rows=100 loops=1)
                     Recheck Cond: (unique1 < 100)
                     ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04
rows=101 width=0) (actual time=0.049..0.049 rows=100 loops=1)
                           Index Cond: (unique1 < 100)
 Planning time: 0.194 ms
 Execution time: 8.008 ms
```

从上面的例子来看，EXPLAIN 命令的输出可以看做是一个树形结构，我们称之为查询计划树，树的每个节点包括对应的节点类型，作用对象以及其他属性例如cost、rows、width等。如果只显示节点类型，上面的例子可以简化为如下结构：

```
Sort
└── Hash Join
    ├── Seq Scan
    └── Hash
        └── Bitmap Heap Scan
            └── Bitmap Index Scan
```

为了帮助大家更好地理解，我们简要介绍下PostgreSQL中SQL 执行的一些特点：

- 按照查询计划树从底往上执行
- 基于火山模型（参考文档火山模型介绍）执行，即可以简单理解为每个节点执行返回一行记录给父节点（Bitmap Index Scan 除外）

在了解了这些特点之后，我们会对EXPLAIN 命令的输出有一个整体结构的概念。其实EXPLAIN 输出的就是一个用户可视化的查询计划树，可以告诉我们执行了哪些节点（操作），并且每个节点（操作）的代价预估是什么样的。接下来，我们将详细去阐述每个节点的类型和具体干了哪些事情。

节点类型

在EXPLAIN 命令的输出结果中可能包含多种类型的执行节点，我们可以大体分为几大类（参考彭智勇、彭煜玮编著的《PostgreSQL 数据库内核分析》）：

- 控制节点（Control Node)
- 扫描节点（ScanNode)
- 物化节点（Materialization Node)
- 连接节点（Join Node)

为了更有针对性，本文只介绍扫描节点。

扫描节点

扫描节点，简单来说就是为了扫描表的元组，每次获取一条元组（Bitmap Index Scan除外）作为上层节点的输入。当然严格的说，不光可以扫描表，还可以扫描函数的结果集、链表结构、子查询结果集等。

目前在PostgreSQL 中支持：

- Seq Scan，顺序扫描
- Index Scan，基于索引扫描，但不只是返回索引列的值
- IndexOnly Scan，基于索引扫描，并且只返回索引列的值，简称为覆盖索引
- BitmapIndex Scan，利用Bitmap 结构扫描
- BitmapHeap Scan，把BitmapIndex Scan 返回的Bitmap 结构转换为元组结构
- Tid Scan，用于扫描一个元组TID 数组
- Subquery Scan，扫描一个子查询
- Function Scan，处理含有函数的扫描
- TableFunc Scan，处理tablefunc 相关的扫描
- Values Scan，用于扫描Values 链表的扫描
- Cte Scan，用于扫描WITH 字句的结果集
- NamedTuplestore Scan，用于某些命名的结果集的扫描
- WorkTable Scan，用于扫描Recursive Union 的中间数据
- Foreign Scan，用于外键扫描
- Custom Scan，用于用户自定义的扫描

其中，我们将重点介绍最常用的几个：Seq Scan、Index Scan、IndexOnly Scan、BitmapIndex Scan、BitmapHeap Scan。

Seq Scan

Seq Scan 是全表顺序扫描，一般查询没有索引的表需要全表顺序扫描，例如下面的 EXPLAIN 输出：

```
postgres=> explain(ANALYZE,VERBOSE,BUFFERS) select * from class where st_no=2;
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------
----------------------
 Seq Scan on public.class  (cost=0.00..26.00 rows=1 width=35) (actual
time=0.136..0.141 rows=1 loops=1)
   Output: st_no, name
   Filter: (class.st_no = 2)
   Rows Removed by Filter: 1199
   Buffers: shared hit=11
 Planning time: 0.066 ms
 Execution time: 0.160 ms
```

其中：

- Seq Scan on public.class 表明了这个节点的类型和作用对象，即在class 表上进行了全表扫描
- (cost=0.00..26.00 rows=1 width=35) 表明了这个节点的代价估计，这部分我们将在下文节点代价估计信息中详细介绍
- (actual time=0.136..0.141 rows=1 loops=1) 表明了这个节点的真实执行信息，当EXPLAIN 命令中的ANALYZE选项为on时，会输出该项内容，具体的含义我们将在下文节点执行信息中详细介绍
- Output: st_no, name 表明了SQL 的输出结果集的各个列，当EXPLAIN 命令中的选项VERBOSE 为on时才会显示
- Filter: (class.st_no = 2) 表明了Seq Scan 节点之上的Filter 操作，即全表扫描时对每行记录进行过滤操作，过滤条件为class.st_no = 2
- Rows Removed by Filter: 1199 表明了过滤操作过滤了多少行记录，属于Seq Scan 节点的VERBOSE 信息，只有EXPLAIN 命令中的VERBOSE 选项为on 时才会显示
- Buffers: shared hit=11 表明了从共享缓存中命中了11 个BLOCK，属于Seq Scan 节点的BUFFERS 信息，只有EXPLAIN 命令中的BUFFERS 选项为on 时才会显示
- Planning time: 0.066 ms 表明了生成查询计划的时间
- Execution time: 0.160 ms 表明了实际的SQL 执行时间，其中不包括查询计划的生成时间

Index Scan

Index Scan 是索引扫描，主要用来在WHERE 条件中存在索引列时的扫描，如上面Seq Scan 中的查询如果在st_no 上创建索引，则EXPLAIN 输出如下：

```
postgres=> explain(ANALYZE,VERBOSE,BUFFERS) select * from class where st_no=2;
                                                      QUERY PLAN
------------------------------------------------------------------------------------------------
-------------------------------------
 Index Scan using no_index on public.class  (cost=0.28..8.29 rows=1 width=35)
(actual time=0.022..0.023 rows=1 loops=1)
   Output: st_no, name
   Index Cond: (class.st_no = 2)
   Buffers: shared hit=3
 Planning time: 0.119 ms
 Execution time: 0.060 ms
(6 rows)
```

其中：

- Index Scan using no_index on public.class 表明是使用的public.class 表的no_index 索引对表进行索引扫描的
- Index Cond: (class.st_no = 2) 表明索引扫描的条件是class.st_no = 2

可以看出，使用了索引之后，对相同表的相同条件的扫描速度变快了。这是因为从全表扫描变为索引扫描，通过Buffers: shared hit=3 可以看出，需要扫描的BLOCK（或者说元组）少了，所以需要的代价也就小了，速度也就快了。

IndexOnly Scan

IndexOnly Scan 是覆盖索引扫描，所需的返回结果能被所扫描的索引全部覆盖，例如上面Index Scan中的SQL 把“select * ” 修改为“select st_no”，其EXPLAIN 结果输出如下：

```
postgres=> explain(ANALYZE,VERBOSE,BUFFERS) select st_no from class where st_no=2;
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------
-----------------------------------------
 Index Only Scan using no_index on public.class  (cost=0.28..4.29 rows=1 width=4)
(actual time=0.015..0.016 rows=1 loops=1)
   Output: st_no
   Index Cond: (class.st_no = 2)
   Heap Fetches: 0
   Buffers: shared hit=3
 Planning time: 0.058 ms
 Execution time: 0.036 ms
(7 rows)
```

其中：

- Index Only Scan using no_index on public.class 表明使用public.class 表的no_index 索引对表进行覆盖索引扫描
- Heap Fetches 表明需要扫描数据块的个数。

虽然Index Only Scan 可以从索引直接输出结果。但是因为PostgreSQL MVCC 机制的实现，需要对扫描的元组进行可见性判断，即检查visibility MAP 文件。当新建表之后，如果没有进行过vacuum和autovacuum操作，这时还没有VM文件，而索引并没有保存记录的版

本信息，索引Index Only Scan 还是需要扫描数据块（Heap Fetches 代表需要扫描的数据块个数）来获取版本信息，这个时候可能会比Index Scan 慢。

BitmapIndex Scan 与BitmapHeap Scan

BitmapIndex Scan 与Index Scan 很相似，都是基于索引的扫描，但是BitmapIndex Scan 节点每次执行返回的是一个位图而不是一个元组，其中位图中每位代表了一个扫描到的数据块。而BitmapHeap Scan一般会作为BitmapIndex Scan 的父节点，将BitmapIndex Scan 返回的位图转换为对应的元组。这样做最大的好处就是把Index Scan 的随机读转换成了按照数据块的物理顺序读取，在数据量比较大的时候，这会大大提升扫描的性能。

我们可以运行set enable_indexscan =off; 来指定关闭Index Scan ，上文中Index Scan 中SQL 的EXPLAIN 输出结果则变为：

```
postgres=> explain(ANALYZE,VERBOSE,BUFFERS) select * from class where st_no=2;
                                                      QUERY PLAN
------------------------------------------------------------------------------------------------
-------------------------------
 Bitmap Heap Scan on public.class  (cost=4.29..8.30 rows=1 width=35) (actual
time=0.025..0.025 rows=1 loops=1)
   Output: st_no, name
   Recheck Cond: (class.st_no = 2)
   Heap Blocks: exact=1
   Buffers: shared hit=3
   ->  Bitmap Index Scan on no_index  (cost=0.00..4.29 rows=1 width=0) (actual
time=0.019..0.019 rows=1 loops=1)
         Index Cond: (class.st_no = 2)
         Buffers: shared hit=2
 Planning time: 0.088 ms
 Execution time: 0.063 ms
(10 rows)
```

其中：

- Bitmap Index Scan on no_index 表明使用no_index 索引进行位图索引扫描
- Index Cond: (class.st_no = 2) 表明位图索引的条件为class.st_no = 2
- Bitmap Heap Scan on public.class 表明对public.class 表进行Bitmap Heap 扫描
- Recheck Cond: (class.st_no = 2) 表明Bitmap Heap Scan 的Recheck操作 的条件是class.st_no = 2，这是因为Bitmap Index Scan 节点返回的是位图，位图中每位代表了一个扫描到的数据块，通过位图可以定位到一些符合条件的数据块（这里是3，Buffers: shared hit=3），而Bitmap Heap Scan 则需要对每个数据块的元组进行Recheck
- Heap Blocks: exact=1 表明准确扫描到数据块的个数是1

至此，我们对这几种主要的扫描节点有了一些认识。一般来说：

- 大多数情况下，Index Scan 要比 Seq Scan 快。但是如果获取的结果集占所有数据的比重很大时，这时Index Scan 因为要先扫描索引再读表数据反而不如直接全表扫描来的快。

- 如果获取的结果集的占比比较小，但是元组数很多时，可能Bitmap Index Scan 的性能要比Index Scan 好。
- 如果获取的结果集能够被索引覆盖，则Index Only Scan 因为不用去读数据，只扫描索引，性能一般最好。但是如果VM 文件未生成，可能性能就会比Index Scan 要差。

上面的结论都是基于理论分析得到的结果，但是其实PostgreSQL 的EXPLAIN 命令中输出的cost，rows，width 等代价估计信息中已经展示了这些扫描节点或者其他节点的预估代价，通过对预估代价的比较，可以选择出最小代价的查询计划树。

代价估计信息

从上文可知，EXPLAIN 命令会在每个节点后面显示代价估计信息，包括cost、rows、width，这里将一一介绍。

在PostgreSQL 中，执行优化器会基于代价估计自动选择代价最小的查询计划树。而在EXPLAIN 命令的输出结果中每个cost 就是该执行节点的代价估计。它的格式是xxx..xxx，在.. 之前的是预估的启动代价，即找到符合该节点条件的第一个结果预估所需要的代价，在..之后的是预估的总代价。而父节点的启动代价包含子节点的总代价。

而在本文开头讲述PostgreSQL DBA 对慢SQL 的常见诊断方法就是使用EXPLAIN 命令，分析其中哪个节点cost （或者下文的 actual time ）最大，通过快速优化它达到优化慢SQL 的目的。

那cost 是怎么计算而来的呢？简单来说，是PostgreSQL 根据周期性收集到的统计信息（参考PostgreSQL · 特性分析 · 统计信息计算方法），按照一个代价估计模型计算而来的。其中会根据以下几个参数来作为代价估计的单位（详见PostgreSQL 官方文档）：

- seq_page_cost
- random_page_cost
- cpu_tuple_cost
- cpu_index_tuple_cost
- cpu_operator_cost
- parallel_setup_cost
- parallel_tuple_cost

其中，seq_page_cost 和random_page_cost 可以使用ALTER TABLESPACE 对每个TABLESPACE 进行修改。

代价估计信息中的其他两个，rows 代表预估的行数，width 代表预估的结果宽度，单位为字节。两者都是根据表的统计信息预估而来的。

真实执行信息

当EXPLAIN 命令中ANALYZE 选项为on时，会在代价估计信息之后输出真实执行信息，包括：

- actual time 执行时间，格式为xxx..xxx，在.. 之前的是该节点实际的启动时间，即找到符合该节点条件的第一个结果实际需要的时间，在..之后的是该节点实际的执行时间
- rows 指的是该节点实际的返回行数
- loops 指的是该节点实际的重启次数。如果一个计划节点在运行过程中，它的相关参数值（如绑定变量）发生了变化，就需要重新运行这个计划节点。

这里需要注意的是，代价估计信息一般是和真实执行信息比较相近的，即预估代价和实际时间成正比且返回结果集的行数相近。但是由于统计信息的时效性，有可能找到的预估代价最小的性能却很差，这就需要开发者调整参数或者主动执行vacuum analyze 命令对表的统计信息进行及时更新，保证PostgreSQL 的执行优化器能够找到相对较优的查询计划树。

总结

至此，我们分析了EXPLAIN 命令的语法和输出的整个结构，可以发现它的输出实际上是PostgreSQL 查询计划树的一个可视化的摘要。在执行优化器内部，就是依靠这些信息找到最优的查询计划树并执行。但是执行优化器不是万能的，由于统计信息不准确或者代价估计模型的问题，PostgreSQL 有可能得到一个性能比较差的查询计划树并执行。这需要做一些case by case 的优化，感兴趣的同学可以参考《WHY IS MY INDEX NOT BEING USED》 和《RDS for PG加入Plan Hint功能》 等相关文章。

参考资料