PostgreSQL中文社区

2021 PostgreSQL China Conference

主办：PostgreSQL 中文社区

第 11 届 PostgreSQL 中国技术大会

开源论道 × 数据驱动 × 共建数字化未来

POSTGRESQL执行计划详解
分析SQL的基础

何敏，曾就职于成都文武、人大金仓，参与开发PG高可用系统、RDS、数据库接口。精通数据库迁移、高可用、系统方案设计，有丰富的开发和运维经验。

PG中文社区首届 PG MVP
中国首期PG ACE 伙伴
中国POSTGRESQL分会官方认证讲师
盘古云课堂高级讲师

初识执行计划

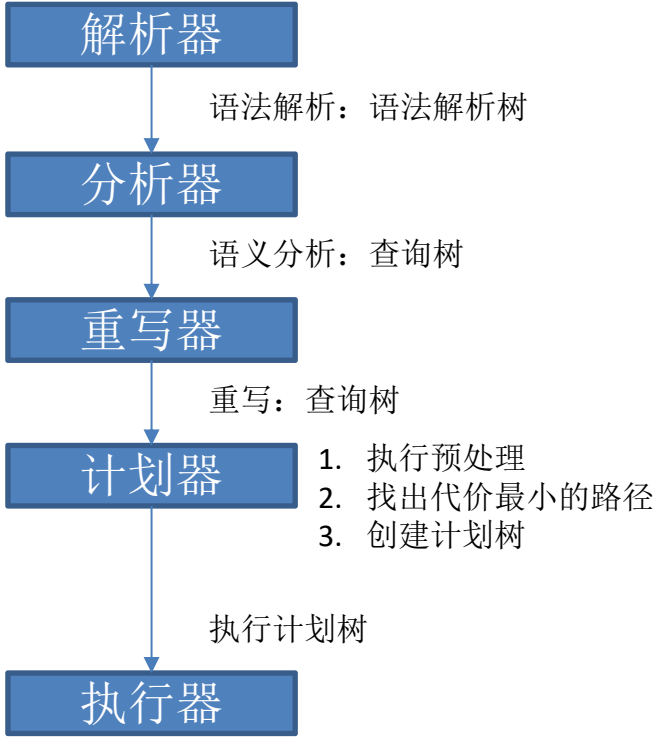
PostgreSQL在执行任何查询时，都会先经过语法、语义解析，生成查询表达式树；

然后根据规则系统对查询树进行转换，视图也在这里展开，最终生成逻辑查询树；

最后根据系统统计信息、物理存储的情况对查询进行优化，生成物理查询计划；

最后根据查询计划进行执行，最终得到结果。因此，选择正确的查询结构和数据属性的规划，对执行效率至关重要。

我们可以通过explain命令来查看执行计划，进而对不合理的地方进行调整，提高SQL的执行效率。



初识执行计划

通过在SQL语句前面加上explain操作，就可以获取到该SQL的执行计划，该SQL并没有实际执行。输出的执行计划有如下特点：

1. 查询规划是以规划为节点的树形结构。树的最底节点是扫描节点：他返回表中的原数据行。
2. 不同的表有不同的扫描节点类型：顺序扫描，索引扫描和位图索引扫描。
3. 也有非表列源，如VALUES子句并设置FROM返回，他们有自己的扫描类型。
4. 如果查询需要关联，聚合，排序或其他操作，会在扫描节点之上增加节点执行这些操作。
5. EXPLAIN的输出是每个树节点显示一行，内容是基本节点类型和执行节点的消耗评估。可能会出现同级别的节点，从汇总行节点缩进显示节点的其他属性。第一行（最上节点的汇总行）是评估执行计划的总消耗，这个值越小越好。

初识执行计划

```
pithe=# create table test(id int, info text, crt_time timestamp);
CREATE TABLE
pithe=# insert into test select generate_series(1, 10000), md5(cast(random() as text)), now();
INSERT 0 10000
pithe=# analyze test;
ANALYZE
pithe=# explain select * from test;
                       QUERY PLAN
-------------------------------------------------------
 Seq Scan on test  (cost=0.00..194.00 rows=10000 width=45)
(1 row)
```

因为这个查询没有WHERE子句，所以必须扫描表中的所有行，所以规划器选择使用简单的顺序扫描规划。括号中的数字从左到右依次是：

1. 评估开始消耗：这是可以开始输出前的时间，比如排序节点的排序的时间。
2. 评估总消耗：假设查询从执行到结束的时间。有时父节点可能停止这个过程，比如LIMIT子句。
3. 评估查询节点的输出行数，假设该节点执行结束。
4. 评估查询节点的输出行的平均字节数。

初识执行计划

需要知道的是：上级节点的消耗包括其子节点的消耗。这个消耗值只反映规划器关心的内容，一般这个消耗不包括将数据传输到客户端的时间。

评估的行数不是执行和扫描查询节点的数量，而是节点返回的数量。它通常会少于扫描数量，因为有WHERE条件会过滤掉一些数据。理想情况顶级行数评估近似于实际返回的数量。

这个消耗的计算依赖于规划器的设置参数，这里的例子都是在默认参数下运行。

cost描述一个SQL执行的代价是多少，而不是具体的时间。下面是默认情况下，对数据操作的消耗评估基础：

初识执行计划

```
pithe=# select name, setting from pg_settings where name like '%cost%' and name not like '%vacuum%';
          name           | setting
-------------------------+---------
 cpu_index_tuple_cost    | 0.005
 cpu_operator_cost       | 0.0025
 cpu_tuple_cost          | 0.01
 jit_above_cost          | 100000
 jit_inline_above_cost   | 500000
 jit_optimize_above_cost | 500000
 parallel_setup_cost     | 1000
 parallel_tuple_cost     | 0.1
 random_page_cost        | 4
 seq_page_cost           | 1
(10 rows)
```

```
pithe=# select relpages, reltuples from pg_class where relname = 'test';
 relpages | reltuples
----------+-----------
       94 |     10000
(1 row)
```

回到刚才的例子，表test有10000条数据分布在94个磁盘页，评估时间是（磁盘页*seq_page_cost）+（扫描行*cpu_tuple_cost）。默认seq_page_cost是1.0，cpu_tuple_cost是0.01，所以评估值是(94 * 1.0) + (10000 * 0.01) = 194。

什么时候更新的pg_class、pg_stat_user_tables等统计信息

初识执行计划

```
pithe=# explain select * from test where id < 1000;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on test  (cost=0.00..219.00 rows=999 width=45)
   Filter: (id < 1000)
(2 rows)
```

查询节点增加了“filter”条件。这意味着查询节点为扫描的每一行数据增加条件检查，只输入符合条件数据。评估的输出记录数因为where子句变少了，但是扫描的数据还是10000条，所以消耗没有减少，反而增加了一点CPU的计算时间。

这个查询实际输出的记录数是1000，但是评估是个近似值，多次运行可能略有差别，这中情况可以通过ANALYZE命令改善。

几种扫描方式

全表扫描：Seq Scan

全表扫描，当数据表中没有索引，或者满足条件的数据集较大，索引扫描的成本高于全表扫描，这时规划器会选择使用全表扫描。

至于全表扫描的过程和索引扫描的过程在这里不详细说，后续可以再开一个主题。

几种扫描方式

索引扫描：index scan

如果查询的列创建有索引，则直接扫描索引，不再进行全表扫描，耗费时间小于顺序扫描。

```
pithe=# create index idx_test_id on test(id);
CREATE INDEX
pithe=# analyze test;
ANALYZE
pithe=# explain select * from test where id < 1000;
                              QUERY PLAN
------------------------------------------------------------------------
 Index Scan using idx_test_id on test  (cost=0.29..42.77 rows=999 width=45)
   Index Cond: (id < 1000)
(2 rows)

pithe=#
```

多了筛选条件后，会打开每条记录，进行筛选记录，花费时间变多了。但是，将筛选条件放到扫描里面是有好处的，尤其是在多表join时，构造自然选择的块有很大的区别

```
pithe=# explain select * from test where id < 1000 and info = '3756da242bee7967edcd3041769f8f96';
                              QUERY PLAN
------------------------------------------------------------------------
 Index Scan using idx_test_id on test  (cost=0.29..45.27 rows=1 width=45)
   Index Cond: (id < 1000)
   Filter: (info = '3756da242bee7967edcd3041769f8f96'::text)
(3 rows)
```

几种扫描方式

全索引扫描：index only scan

当查询的条件都在索引中，也会走该扫描方式，不会读取表文件。

```
pithe=# explain select id  from test where id < 1000 ;
                                QUERY PLAN
---------------------------------------------------------------------------
 Index Only Scan using test_pk on test  (cost=0.29..29.77 rows=999 width=4)
   Index Cond: (id < 1000)
(2 rows)

pithe=# explain select id  from test where id = 1000 ;
                               QUERY PLAN
-------------------------------------------------------------------------
 Index Only Scan using test_pk on test  (cost=0.29..4.30 rows=1 width=4)
   Index Cond: (id = 1000)
(2 rows)
```

几种扫描方式

位图扫描： Bitmap Index Scan

位图扫描也是一种走索引的方式，方法是扫描索引，把满足条件的行或者块在内存中建一个位图，扫描完索引后，再跟进位图中记录的指针到表的数据文件读取相应的数据。

在or、and、in子句和有多个条件都可以同时走不同的索引时，都可能走Bitmap Index Scan ：

```
pithe=# explain select * from test where id < 1000 or id > 9000 ;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=40.51..164.50 rows=1899 width=45)
   Recheck Cond: ((id < 1000) OR (id > 9000))
   ->  BitmapOr  (cost=40.51..40.51 rows=1999 width=0)
         ->  Bitmap Index Scan on test_pk  (cost=0.00..19.78 rows=999 width=0)
               Index Cond: (id < 1000)
         ->  Bitmap Index Scan on test_pk  (cost=0.00..19.79 rows=1000 width=0)
               Index Cond: (id > 9000)
(7 rows)
```

几种扫描方式

位图扫描： Bitmap Index Scan

Bitmap Heap Scan的启动时间就是两个子Bitmap Index Scan的总和，可以看出在进行组合时花费了大量时间。

Bitmap Index Scan和Index Scan扫描的区别很明显：
Index scan: 输出的是tuple，它先扫描索引块，然后得到rowid扫描数据块得到目标记录。一次只读一条索引项，那么一个 PAGE面有可能被多次访问.

Bitmap index scan; 输出的是索引条目，并不是行的数据，输出索引条目后，交给上一个节点 bitmap heap scan(之间可能将索引条目根据物理排列顺序进行排序)。一次性将满足条件的索引项全部取出，然后交给bitmap heap scan节点，并在内存中进行排序, 根据取出的索引项访问表数据。

几种扫描方式

limit对执行计划的影响

这条查询的where条件和上面的一样，只是增加了limit，所以不是所有数据都需要返回，规划器在发现limit后改变了规划。虽然在索引扫描节点总消耗和返回记录数是244，但查询是从上往下拉取数据的，当扫描到满足要求的2行结果后，达到了Limit 2 取两条的要求，就直接返回了，预期时间消耗是26，所以总时间消耗是26。

```
pithe=#  explain select * from test where id < 1000 or id > 9000 limit 2;
                          QUERY PLAN
---------------------------------------------------------------
 Limit  (cost=0.00..0.26 rows=2 width=45)
   ->  Seq Scan on test  (cost=0.00..244.00 rows=1899 width=45)
         Filter: ((id < 1000) OR (id > 9000))
(3 rows)
```

几种连接方式

Hash Join

这个计划是使用hash join的方式来进行表连接的，首先确定两个表的大小，使用小表建立hash map，然后扫描大表，比较hash值，最终获取查询结果。

```
pithe=# create table test1 as select * from test;
SELECT 10000
pithe=# create unique index idx_test1_id on test1(id);
CREATE INDEX
pithe=# explain select * from test t, test1 t1 where t.id < 10 and t.info = t1.info;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=8.55..240.15 rows=9 width=89)
   Hash Cond: (t1.info = t.info)
   ->  Seq Scan on test1 t1  (cost=0.00..194.00 rows=10000 width=44)
   ->  Hash  (cost=8.44..8.44 rows=9 width=45)
         ->  Index Scan using test_pk on test t  (cost=0.29..8.44 rows=9 width=45)
               Index Cond: (id < 10)
(6 rows)

pithe=# delete from test1 where id > 101;
DELETE 9899
pithe=# analyze test1;
ANALYZE
pithe=# explain select * from test t, test1 t1 where t.id < 1000 and t.info = t1.info;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Hash Join  (cost=96.56..142.89 rows=10 width=90)
   Hash Cond: (t.info = t1.info)
   ->  Index Scan using test_pk on test t  (cost=0.29..42.77 rows=999 width=45)
         Index Cond: (id < 1000)
   ->  Hash  (cost=95.01..95.01 rows=101 width=45)
         ->  Seq Scan on test1 t1  (cost=0.00..95.01 rows=101 width=45)
(6 rows)
```

几种连接方式

连接方式	说明	执行步骤	适用场景	时间消耗
hash join	散列连接是做大数据集连接时的常用方式，优化器使用两个表中较小的表（或数据源）利用连接键在内存中建立散列表，然后扫描较大的表并探测散列表，找出与散列表匹配的行。 这种方式适用于较小的表完全可以放于内存中的情况，这样总成本就是访问两个表的成本之和。但是在表很大的情况下并不能完全放入内存，这时优化器会将它分割成若干不同的分区，不能放入内存的部分就把该分区写入磁盘的临时段，此时要有较大的临时段从而尽量提高I/O 的性能。	将两个表中较小的一个在内存中构造一个HASH表（对JOIN KEY），扫描另一个表，同样对JOIN KEY进行HASH后探测是否可以JOIN。适用于记录集比较大的情况。需要注意的是：如果HASH表太大，无法一次构造在内存中，则分成若干个partition，写入磁盘的temporary segment，则会多一个写的代价，会降低效率	两个大表关联查询	cost = (outer access cost * # of hash partitions) + inner access cost

几种连接方式

Nested Loop Join

这个规划中有一个内连接的节点，它有两个子节点。节点摘要行的缩进反映了规划树的结构。最外层是一个连接节点，子节点是一个Seq Scan扫描。外部节点为t1.info = '3756da242bee7967edcd3041769f8f96'的结果。接下来为每一个从外部节点得到的记录运行内部查询节点（t.id < 10）。

外部节点的消耗加上循环内部节点的消耗（2.26 + 1*8.44）再加一点CPU时间就得到规划的总消耗10.82。

```
pithe=# explain select * from test t, test1 t1 where t.id < 10 and t1.info = '3756da242bee7967
edcd3041769f8f96' and t.crt_time < t1.crt_time;
                                QUERY PLAN
---------------------------------------------------------------------------
 Nested Loop  (cost=0.29..103.82 rows=3 width=90)
   Join Filter: (t.crt_time < t1.crt_time)
   ->  Seq Scan on test1 t1  (cost=0.00..95.26 rows=1 width=45)
         Filter: (info = '3756da242bee7967edcd3041769f8f96'::text)
   ->  Index Scan using test_pk on test t  (cost=0.29..8.44 rows=9 width=45)
         Index Cond: (id < 10)
(6 rows)
```

几种连接方式

连接方式	说明	执行步骤	适用场景	时间消耗
nested loop join	对于被连接的数据子集较小的情况，嵌套循环连接是个较好的选择。在嵌套循环中，内表被外表驱动，外表返回的每一行都要在内表中检索找到与它匹配的行，因此整个查询返回的结果集不能太大（大于1 万不适合），要把返回子集较小表的作为外表（默认外表是驱动表），而且在内表的连接字段上一定要有索引。	确定一个驱动表(outer table)，另一个表为inner table，驱动表中的每一行与inner表中的相应记录JOIN。类似一个嵌套的循环。	Nested loop一般用在连接的表中有索引，并且索引选择性较好的时候。 适用于驱动表的记录集比较小（<10000）而且inner表需要有有效的访问方法（Index）。需要注意的是：JOIN的顺序很重要，驱动表的记录集一定要小，返回结果集的响应时间是最快的。	cost = outer access cost + (inner access cost * outer cardinality)

几种连接方式

Materialize应用

上面Nestted Loop每次都会执行一次内部节点的查询，虽然内部节点经过一次查询后，就放到了内存，但是也很费时间，计划生成器会使用另外一种技术来规避这个问题，但需要内层的结果是不变的：

```
pithe=# explain select * from test t, test1 t1 where t.id < 10 and t.id > t1.id;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=0.29..117.11 rows=303 width=90)
   Join Filter: (t.id > t1.id)
   ->  Seq Scan on test1 t1  (cost=0.00..95.01 rows=101 width=45)
   ->  Materialize  (cost=0.29..8.49 rows=9 width=45)
         ->  Index Scan using test_pk on test t  (cost=0.29..8.44 rows=9 width=45)
               Index Cond: (id < 10)
(6 rows)
```

规划器选择使用Meaterialize节点，这意味着内部节点的索引扫描只做一次，即使嵌套循环需要读取这些数据101次，Meterialize节点将数据保存在work_mem或临时文件中。尤其是物化的数据能全部放到work_mem的情况，每次循环都从work_mem中读取数据，时间消耗较小。时间消耗是2.01+8.44+cup消耗 = 24.11。

几种连接方式

Merge Join

Merge Join需要对输入数据已经做好了排序，通常情况下，散列连接效果比合并连接好。然而如果两个表上都有相应的索引，或者数据表已经顺序排列的了，那么在执行合并连接时就不需要排序，这时合并连接的性能就可能优于Hash Join。

```
pithe=# explain select * from test t, test1 t1 where t.id < 10 and t.id = t1.id;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=0.55..79.16 rows=1 width=90)
   ->  Index Scan using test_pk on test t  (cost=0.29..8.44 rows=9 width=45)
         Index Cond: (id < 10)
   ->  Index Scan using idx_test1_id on test1 t1  (cost=0.27..7.84 rows=1 width=45)
         Index Cond: (id = t.id)
(5 rows)

pithe=# drop index idx_test1_id ;
DROP INDEX
pithe=# explain select * from test t, test1 t1 where t.id < 10 and t.id = t1.id;
                                QUERY PLAN
---------------------------------------------------------------------------
 Merge Join  (cost=98.66..100.08 rows=1 width=90)
   Merge Cond: (t.id = t1.id)
   ->  Index Scan using test_pk on test t  (cost=0.29..8.44 rows=9 width=45)
         Index Cond: (id < 10)
   ->  Sort  (cost=98.37..98.62 rows=101 width=45)
         Sort Key: t1.id
         ->  Seq Scan on test1 t1  (cost=0.00..95.01 rows=101 width=45)
(7 rows)
```

几种连接方式

连接方式	说明	执行步骤	适用场景	时间消耗
merge join	通常情况下散列连接的效果都比排序合并连接要好，然而如果行源已经被排过序，在执行排序合并连接时不需要再排序了，这时排序合并连接的性能可能会优于散列连接。	将两个表排序，然后将两个表合并。	Sort Merge join 用在数据已经排序，且数据量不大的情况。	cost = (outer access cost * # of hash partitions) + inner access cost

几种连接方式

三种表连接方式比较

连接方式	说明
hash join	Hash join的工作方式是将一个表（通常是小一点的那个表）做hash运算，将列数据存储到hash列表中，从另一个表中抽取记录，做hash运算，到hash 列表中找到相应的值，做匹配。 小表做hash。
nested loop join	工作方式是从一张表中读取数据，访问另一张表（通常是索引）来做匹配，nested loops适用的场合是当一个关联表比较小的时候，效率会更高。 小表做驱动。
merge join	是先将关联表的关联列各自做排序，然后从各自的排序表中抽取数据，到另一个排序表中做匹配，因为merge join需要做更多的排序，所以消耗的资源更多。 一般情况下，能够使用merge join的地方，hash join都可以发挥更好的性能。 小表做驱动。

与执行计划相关的参数

在数据库中有一些以enable_开头的参数可以控制执行器选择不同的执行计划，尤其是有时候某个SQL的执行计划不是最优的时候，可以人为的修改这些参数来强制优化器选择一个更好的执行计划来临时解决问题。另外一种情况是做测试的时候，希望测试不同的执行计划的效果差距，可以调整这些参数。

通常情况下，数据库不会走错执行计划，除非Analyze的频率太低，导致数据库的统计信息不正确，进而进行了错误的评估。

```
pithe=# select name, setting, short_desc from pg_settings where name like '%enable_%';
              name               | setting |                        short_desc
---------------------------------+---------+-----------------------------------------------------
 enable_async_append             | on      | Enables the planner's use of async append plans.
 enable_bitmapscan               | on      | Enables the planner's use of bitmap-scan plans.
 enable_gathermerge              | on      | Enables the planner's use of gather merge plans.
 enable_hashagg                  | on      | Enables the planner's use of hashed aggregation plans.
 enable_hashjoin                 | on      | Enables the planner's use of hash join plans.
 enable_incremental_sort         | on      | Enables the planner's use of incremental sort steps.
 enable_indexonlyscan            | on      | Enables the planner's use of index-only-scan plans.
 enable_indexscan                | on      | Enables the planner's use of index-scan plans.
 enable_material                 | on      | Enables the planner's use of materialization.
 enable_memoize                  | on      | Enables the planner's use of memoization.
 enable_mergejoin                | on      | Enables the planner's use of merge join plans.
 enable_nestloop                 | on      | Enables the planner's use of nested-loop join plans.
 enable_parallel_append          | on      | Enables the planner's use of parallel append plans.
 enable_parallel_hash            | on      | Enables the planner's use of parallel hash plans.
 enable_partition_pruning        | on      | Enables plan-time and execution-time partition pruning.
 enable_partitionwise_aggregate  | off     | Enables partitionwise aggregation and grouping.
 enable_partitionwise_join       | off     | Enables partitionwise join.
 enable_seqscan                  | on      | Enables the planner's use of sequential-scan plans.
 enable_sort                     | on      | Enables the planner's use of explicit sort steps.
 enable_tidscan                  | on      | Enables the planner's use of TID scan plans.
(20 rows)
```

使用Analyze查看真实执行情况

使用explain analyze会使SQL真正执行，显示真实的返回记录数和运行每个规划节点的时间，进而评估查询器的准确性。

```
pithe=# explain analyze select * from test t, test1 t1 where t.id < 10 and t1.info = '3756da242bee7967edcd3041769f8f96' and t.crt_time = t1.crt_time;
                                                  QUERY PLAN
---------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.29..103.82 rows=9 width=90) (actual time=0.531..0.531 rows=0 loops=1)
   Join Filter: (t.crt_time = t1.crt_time)
   ->  Seq Scan on test1 t1  (cost=0.00..95.26 rows=1 width=45) (actual time=0.531..0.531 rows=0 loops=1)
         Filter: (info = '3756da242bee7967edcd3041769f8f96'::text)
         Rows Removed by Filter: 101
   ->  Index Scan using idx_test_id on test t  (cost=0.29..8.44 rows=9 width=45) (never executed)
         Index Cond: (id < 10)
 Planning Time: 2.044 ms
 Execution Time: 0.546 ms
(9 rows)
```

注意，实际时间(actual time)的值是已毫秒为单位的实际时间，cost是评估的消耗，是个虚拟单位时间，所以他们看起来不匹配。通常最重要的是看评估的记录数是否和实际得到的记录数接近。在这个例子里评估数完全和实际一样，但这种情况很少出现。

某些查询规划可能执行多次子规划。比如之前提过的内循环规划（nested-loop），内部索引扫描的次数是外部数据的数量。在这种情况下，报告显示循环执行的总次数、平均实际执行时间和数据条数。这样做是为了和评估值表示方式一至。由循环次数和平均值相乘得到总消耗时间。

使用其他参数

其他参数:

```
pithe=# \h explain
Command:     EXPLAIN
Description: show the execution plan of a statement
Syntax:
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement

where option can be one of:

    ANALYZE [ boolean ]
    VERBOSE [ boolean ]
    COSTS [ boolean ]
    SETTINGS [ boolean ]
    BUFFERS [ boolean ]
    WAL [ boolean ]
    TIMING [ boolean ]
    SUMMARY [ boolean ]
    FORMAT { TEXT | XML | JSON | YAML }

URL: https://www.postgresql.org/docs/14/sql-explain.html
```

```
pithe=# explain (ANALYZE,VERBOSE,COSTS,SETTINGS,BUFFERS,WAL,TIMING,SUMMARY) select * from test t, test1 t1 where t.id <
 10 and t1.info = '3756da242bee7967edcd3041769f8f96' and t.crt_time = t1.crt_time;
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.29..103.82 rows=9 width=90) (actual time=0.716..0.719 rows=0 loops=1)
   Output: t.id, t.info, t.crt_time, t1.id, t1.info, t1.crt_time
   Join Filter: (t.crt_time = t1.crt_time)
   Buffers: shared read=94
   ->  Seq Scan on public.test1 t1  (cost=0.00..95.26 rows=1 width=45) (actual time=0.715..0.716 rows=0 loops=1)
         Output: t1.id, t1.info, t1.crt_time
         Filter: (t1.info = '3756da242bee7967edcd3041769f8f96'::text)
         Rows Removed by Filter: 101
         Buffers: shared read=94
   ->  Index Scan using idx_test_id on public.test t  (cost=0.29..8.44 rows=9 width=45) (never executed)
         Output: t.id, t.info, t.crt_time
         Index Cond: (t.id < 10)
 Planning:
   Buffers: shared hit=184 read=41
 Planning Time: 9.456 ms
 Execution Time: 0.788 ms
(16 rows)

pithe=# explain (ANALYZE,VERBOSE,COSTS,SETTINGS,BUFFERS,WAL,TIMING,SUMMARY) select * from test t, test1 t1 where t.id <
 10 and t1.info = '3756da242bee7967edcd3041769f8f96' and t.crt_time = t1.crt_time;
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.29..103.82 rows=9 width=90) (actual time=0.082..0.083 rows=0 loops=1)
   Output: t.id, t.info, t.crt_time, t1.id, t1.info, t1.crt_time
   Join Filter: (t.crt_time = t1.crt_time)
   Buffers: shared hit=94
   ->  Seq Scan on public.test1 t1  (cost=0.00..95.26 rows=1 width=45) (actual time=0.082..0.082 rows=0 loops=1)
         Output: t1.id, t1.info, t1.crt_time
         Filter: (t1.info = '3756da242bee7967edcd3041769f8f96'::text)
         Rows Removed by Filter: 101
         Buffers: shared hit=94
   ->  Index Scan using idx_test_id on public.test t  (cost=0.29..8.44 rows=9 width=45) (never executed)
         Output: t.id, t.info, t.crt_time
         Index Cond: (t.id < 10)
 Planning:
   Buffers: shared hit=3
 Planning Time: 0.160 ms
 Execution Time: 0.110 ms
(16 rows)
```

日志参数

在数据库日志中，默认是不会输出执行计划相关的日志的，因为这会极大的影响数据库性能。但是在开发和测试阶段，可以将这些参数打印处理，便于分析不合理的SQL。

```
pithe=# select name, setting from pg_settings where name like '%log_%stats%';
        name         | setting
---------------------+---------
 log_executor_stats  | off
 log_parser_stats    | off
 log_planner_stats   | off
 log_statement_stats | off
(4 rows)
```

log_statement_stats：报告整个语句的统计数据。
log_parser_stats：记载数据库解析器的统计数据。
log_planner_stats：报告数据库查询优化器的统计数据。
log_executor_stats：报告数据库执行器的统计数据。

```
pithe=# set log_planner_stats = on;
SET
pithe=# explain (analyze,verbose,timing,costs,
2022-01-05 11:10:49.585 CST [94593] LOG:  PLAN
2022-01-05 11:10:49.585 CST [94593] DETAIL:  !
        !       0.000051 s user, 0.000014 s sy
        !       [0.007382 s user, 0.004364 s s
        !       6288 kB max resident size
```

```
pithe=# set log_executor_stats = on;
SET
pithe=# explain (analyze,verbose,timing,costs,buffers,SETTINGS,summary) select * from test_v where id < 1000;
2022-01-05 11:15:01.986 CST [95548] LOG:  EXECUTOR STATISTICS
2022-01-05 11:15:01.986 CST [95548] DETAIL:  ! system usage stats:
        !       0.001700 s user, 0.000371 s system, 0.002071 s elapsed
        !       [0.003485 s user, 0.001769 s system total]
        !       5636 kB max resident size
        !       0/0 [0/0] filesystem blocks in/out
```

```
pithe=# set log_statement_stats = 1;
SET
pithe=# explain (analyze,verbose,timing,costs,buffers,SETTINGS,summary) select * from test_v where id < 1000;
2022-01-05 11:18:50.948 CST [96132] LOG:  QUERY STATISTICS
2022-01-05 11:18:50.948 CST [96132] DETAIL:  ! system usage stats:
        !       0.001977 s user, 0.003413 s system, 0.016170 s elapsed
        !       [0.005592 s user, 0.007794 s system total]
        !       6360 kB max resident size
        !       0/0 [0/0] filesystem blocks in/out
        !       0/553 [0/1655] page faults/reclaims, 0 [0] swaps
        !       0 [0] signals rcvd, 0/0 [3/2] messages rcvd/sent
        !       46/3 [78/18] voluntary/involuntary context switches
2022-01-05 11:18:50.948 CST [96132] STATEMENT:  explain (analyze,verbose,timing,costs,buffers,SETTINGS,summary) select * f
rom test_v where id < 1000;
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Seq Scan on public.test  (cost=0.00..244.00 rows=999 width=45) (actual time=0.015..1.233 rows=999 loops=1)
   Output: test.id, test.info, test.crt_time
   Filter: ((test.id < 1000) AND (test.id < 1000))
   Rows Removed by Filter: 9001
   Buffers: shared read=94
 Planning:
   Buffers: shared hit=52 read=14
 Planning Time: 4.650 ms
 Execution Time: 1.671 ms
(9 rows)
```

```
pithe=# set log_parser_stats = on;
SET
```

SQL优化的方向

从最底层的扫描入手：

尽量走索引
选择合理的索引类型
减少不不必要的索引
insert: 单条->批量 copy

从SQL入手：

CTE减少嵌套
减少子查询
物化视图、临时表
Exits使用
拆分SQL
改写SQL
类型转换失真

从数据库参数入手:

精确统计信息
干涉执行计划
调整性能参数
pg_hint_plan

其他:

连接池
操作系统参数
硬件性能

THANKS

谢谢观看