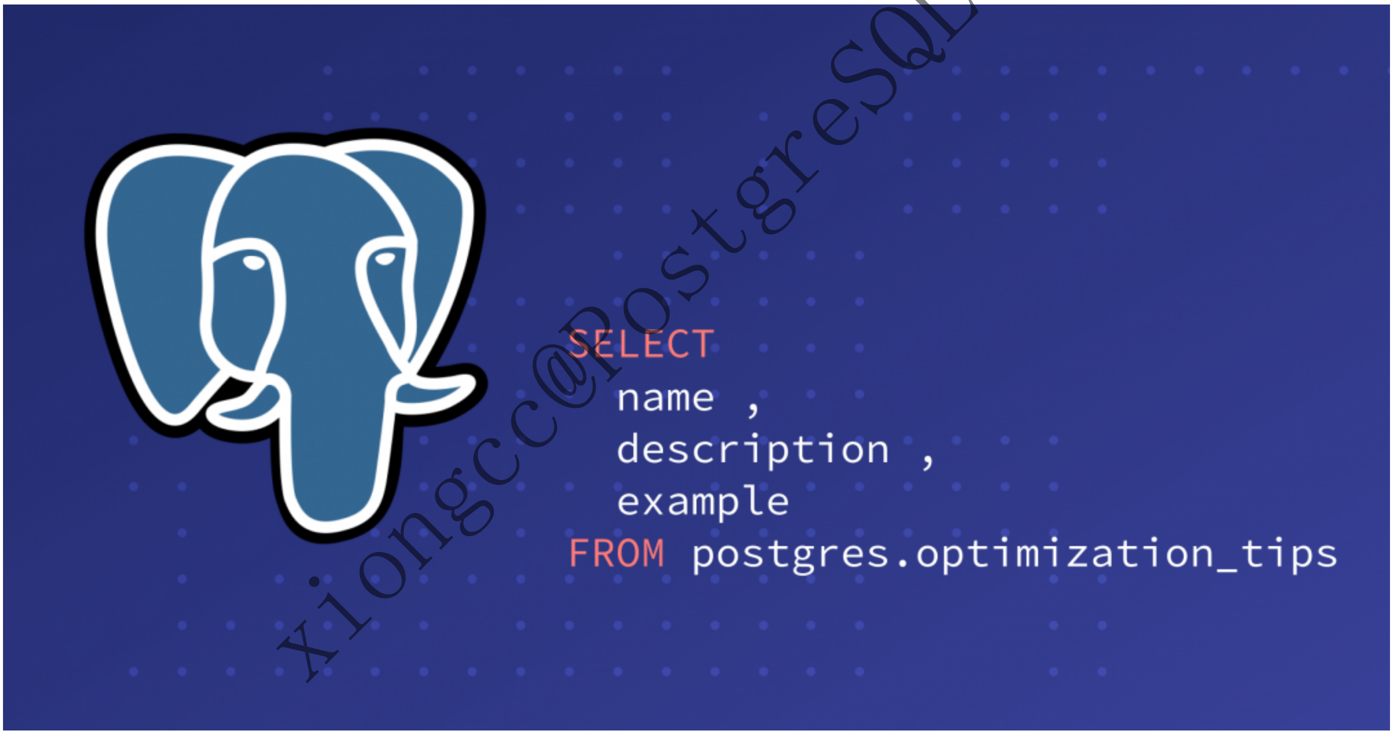
前言

SQL优化是一个任重而道远的活，有时线上可能仅仅是一条小小的不起眼SQL突然跑慢了，便可能引发业务系统雪崩，所以SQL优化的重要性不言而喻，但是，不少DBA或者开发都认为SQL优化很简单，加索引嘛，有啥难的，看老夫万能Btree打天下。遗憾的是，SQL优化是一件十分复杂的技术栈，不仅仅是：

1. analyze简单地收集一下统计信息
2. 根据where列、order by列等创建索引
3. 使用pg_hint_plan或者pg_plan_advsr建议一下优化器
4. 使用prepareStatements绑定一下执行计划减少解析时间
5. 无脑让开发降低并发，无脑scale out或者scale up，减少连接表的个数等

核心还要学会SQL的等价改写，但是这需要经历一个漫长的经验累积的过程，以我目前的功力还远远达不到去对SQL等价改写这个复杂的技术栈去评头论足 (此处推荐阅读罗炳森的《SQL等价改写核心思想》)，不过既然高级一点的不会，那我们就旁路，迂回战术，使用一些现有的工具，如pgMustard — review Postgres query plans quickly，pganalyze自动化性能诊断和优化产品等，助力我们优化SQL，达到事半功倍的效果。

今天分享的是PostgreSQL中，协助我们分析SQL的工具，让你假装是一个SQL优化大佬，带妹一绝。



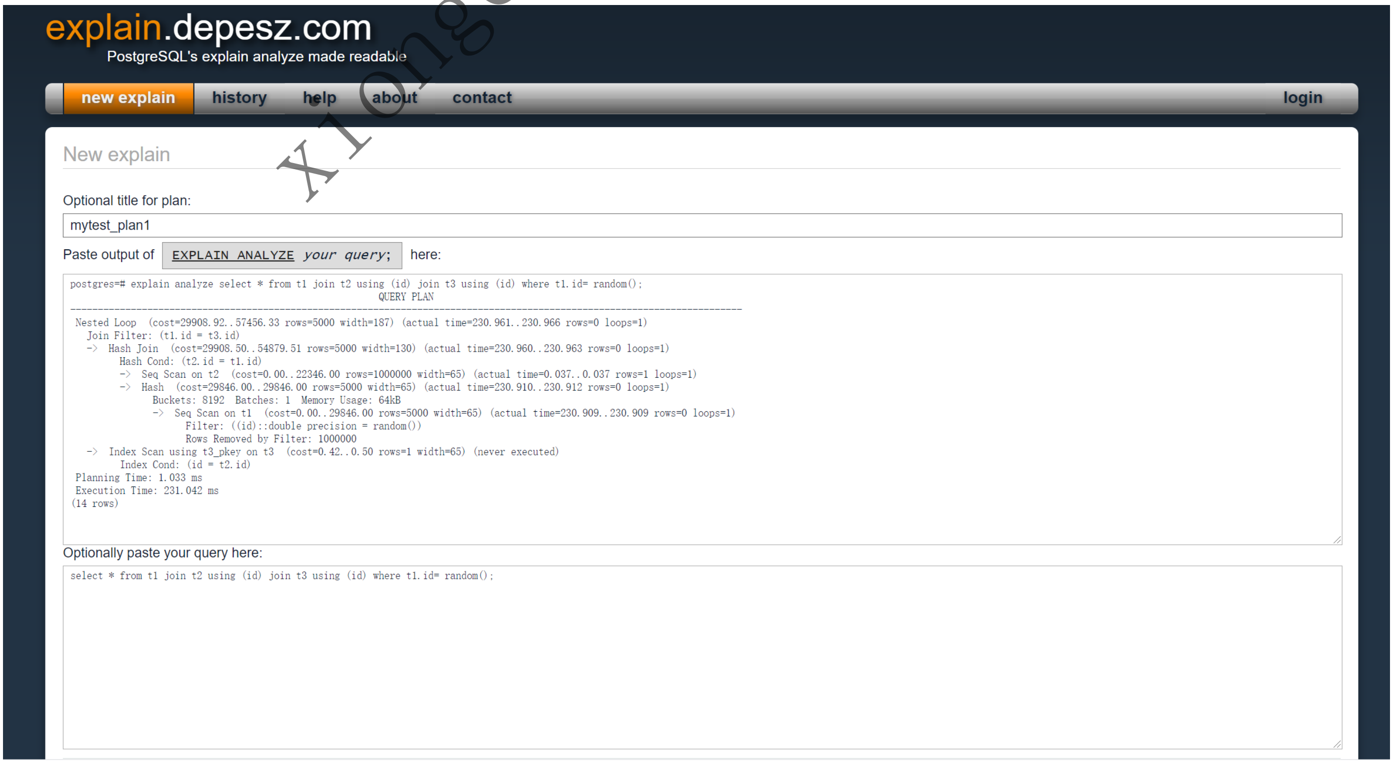
Depesz’ EXPLAIN ANALYZE visualizer

这个工具是Hubert Lubaczewski大师的杰作，地址在：https://explain.depesz.com/

使用很简单，先造个数据，模拟一下多表join

```
postgres=# create table t1(
postgres(#   id int primary key,
postgres(#   info text default 'tesssssssssssssssssssssssssssssssssssst',
postgres(#   state int default 0,
postgres(#   crt_time timestamp default now(),
postgres(#   mod_time timestamp default now()
postgres(# );
CREATE TABLE
postgres=# create table t2 (like t1 including all);
CREATE TABLE
postgres=# create table t3 (like t1 including all);
CREATE TABLE
postgres=# insert into t1 select generate_series(1,1000000);
INSERT 0 1000000
postgres=# insert into t2 select * from t1;
INSERT 0 1000000
postgres=# insert into t3 select * from t1;
INSERT 0 1000000
postgres=# explain select * from t1 join t2 using (id) join t3 using (id) where t1.id=
random();
                                 QUERY PLAN
----------------------------------------------------------------------------
 Nested Loop  (cost=29908.93..57468.83 rows=5000 width=178)
   Join Filter: (t1.id = t3.id)
   ->  Hash Join  (cost=29908.50..54879.51 rows=5000 width=130)
         Hash Cond: (t2.id = t1.id)
         ->  Seq Scan on t2  (cost=0.00..22346.00 rows=1000000 width=65)
         ->  Hash  (cost=29846.00..29846.00 rows=5000 width=65)
               ->  Seq Scan on t1  (cost=0.00..29846.00 rows=5000 width=65)
                     Filter: ((id)::double precision = random())
   ->  Index Scan using t3_pkey on t3  (cost=0.43..0.51 rows=1 width=56)
         Index Cond: (id = t2.id)
(10 rows)
```

使用很简单，分别上传explain analyze + SQL 的真实执行计划情况和对应的SQL



Optional title for plan:

mytest_plan1

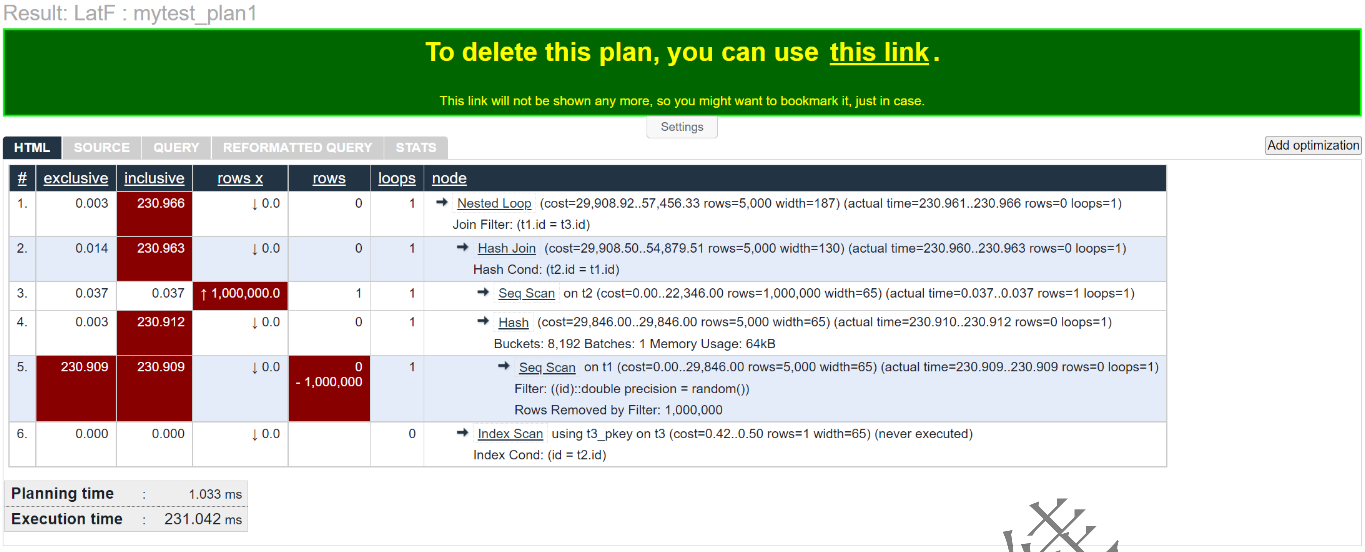
Paste output of EXPLAIN ANALYZE *your query*; here:

```
postgres=# explain analyze select * from t1 join t2 using (id) join t3 using (id) where t1.id= random();
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=29908.92..57456.33 rows=5000 width=187) (actual time=230.961..230.966 rows=0 loops=1)
   Join Filter: (t1.id = t3.id)
   ->  Hash Join  (cost=29908.50..54879.51 rows=5000 width=130) (actual time=230.960..230.963 rows=0 loops=1)
         Hash Cond: (t2.id = t1.id)
         ->  Seq Scan on t2  (cost=0.00..22346.00 rows=1000000 width=65) (actual time=0.037..0.037 rows=1 loops=1)
         ->  Hash  (cost=29846.00..29846.00 rows=5000 width=65) (actual time=230.910..230.912 rows=0 loops=1)
               Buckets: 8192  Batches: 1  Memory Usage: 64kB
               ->  Seq Scan on t1  (cost=0.00..29846.00 rows=5000 width=65) (actual time=230.909..230.909 rows=0 loops=1)
                     Filter: ((id)::double precision = random())
                     Rows Removed by Filter: 1000000
   ->  Index Scan using t3_pkey on t3  (cost=0.42..0.50 rows=1 width=65) (never executed)
         Index Cond: (id = t2.id)
 Planning Time: 1.033 ms
 Execution Time: 231.042 ms
(14 rows)
```

Optionally paste your query here:

```
select * from t1 join t2 using (id) join t3 using (id) where t1.id= random();
```

然后submit提交即可，就会基于当前的执行计划生成一份HTML的报告

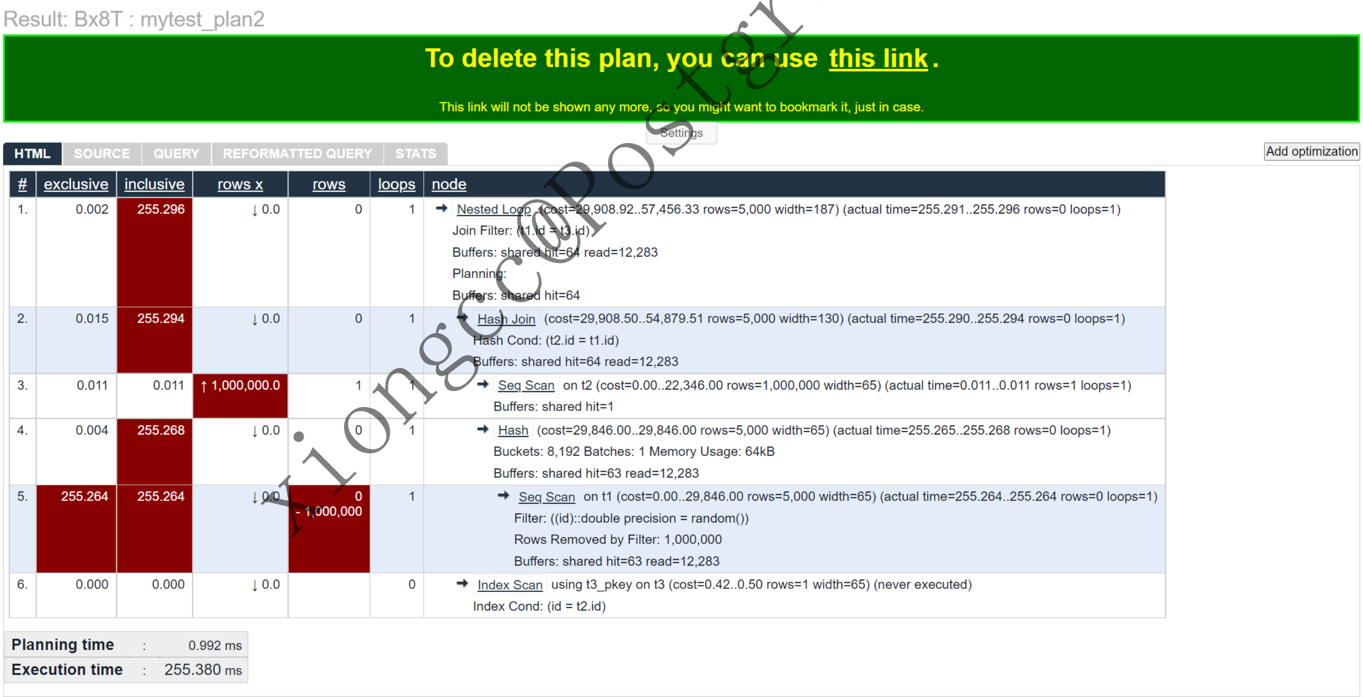

Result: LatF : mytest_plan1
To delete this plan, you can use this link.
This link will not be shown any more, so you might want to bookmark it, just in case.
Settings
HTML
SOURCE
QUERY
REFORMATTED QUERY
STATS
Add optimization

#	exclusive	inclusive	rows x	rows	loops	node
1.	0.003	230.966	↓ 0.0	0	1	→ Nested Loop (cost=29,908.92..57,456.33 rows=5,000 width=187) (actual time=230.961..230.966 rows=0 loops=1) Join Filter: (t1.id = t3.id)
2.	0.014	230.963	↓ 0.0	0	1	→ Hash Join (cost=29,908.50..54,879.51 rows=5,000 width=130) (actual time=230.960..230.963 rows=0 loops=1) Hash Cond: (t2.id = t1.id)
3.	0.037	0.037	↑ 1,000,000.0	1	1	→ Seq Scan on t2 (cost=0.00..22,346.00 rows=1,000,000 width=65) (actual time=0.037..0.037 rows=1 loops=1)
4.	0.003	230.912	↓ 0.0	0	1	→ Hash (cost=29,846.00..29,846.00 rows=5,000 width=65) (actual time=230.910..230.912 rows=0 loops=1) Buckets: 8,192 Batches: 1 Memory Usage: 64kB
5.	230.909	230.909	↓ 0.0	0 - 1,000,000	1	→ Seq Scan on t1 (cost=0.00..29,846.00 rows=5,000 width=65) (actual time=230.909..230.909 rows=0 loops=1) Filter: ((id)::double precision = random()) Rows Removed by Filter: 1,000,000
6.	0.000	0.000	↓ 0.0		0	→ Index Scan using t3_pkey on t3 (cost=0.42..0.50 rows=1 width=65) (never executed) Index Cond: (id = t2.id)

Planning time	:	1.033 ms
Execution time	:	231.042 ms

当然，还可以加上一些附加的explain选项，如buffers查看缓冲区的使用情况，这样会更加方便我们分析

```
postgres=# explain (analyze,buffers) select * from t1 join t2 using (id) join t3 using
(id) where t1.id= random();
```


Result: Bx8T : mytest_plan2
To delete this plan, you can use this link.
This link will not be shown any more, so you might want to bookmark it, just in case.
Settings
HTML
SOURCE
QUERY
REFORMATTED QUERY
STATS
Add optimization

#	exclusive	inclusive	rows x	rows	loops	node
1.	0.002	255.296	↓ 0.0	0	1	→ Nested Loop (cost=29,908.92..57,456.33 rows=5,000 width=187) (actual time=255.291..255.296 rows=0 loops=1) Join Filter: (t1.id = t3.id) Buffers: shared hit=64 read=12,283 Planning: Buffers: shared hit=64
2.	0.015	255.294	↓ 0.0	0	1	→ Hash Join (cost=29,908.50..54,879.51 rows=5,000 width=130) (actual time=255.290..255.294 rows=0 loops=1) Hash Cond: (t2.id = t1.id) Buffers: shared hit=64 read=12,283
3.	0.011	0.011	↑ 1,000,000.0	1	1	→ Seq Scan on t2 (cost=0.00..22,346.00 rows=1,000,000 width=65) (actual time=0.011..0.011 rows=1 loops=1) Buffers: shared hit=1
4.	0.004	255.268	↓ 0.0	0	1	→ Hash (cost=29,846.00..29,846.00 rows=5,000 width=65) (actual time=255.265..255.268 rows=0 loops=1) Buckets: 8,192 Batches: 1 Memory Usage: 64kB Buffers: shared hit=63 read=12,283
5.	255.264	255.264	↓ 0.0	0 - 1,000,000	1	→ Seq Scan on t1 (cost=0.00..29,846.00 rows=5,000 width=65) (actual time=255.264..255.264 rows=0 loops=1) Filter: ((id)::double precision = random()) Rows Removed by Filter: 1,000,000 Buffers: shared hit=63 read=12,283
6.	0.000	0.000	↓ 0.0		0	→ Index Scan using t3_pkey on t3 (cost=0.42..0.50 rows=1 width=65) (never executed) Index Cond: (id = t2.id)

Planning time	:	0.992 ms
Execution time	:	255.380 ms

这里简单回顾一下执行计划：

1. 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。
2. 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
3. rows 指的是该节点实际的返回行数

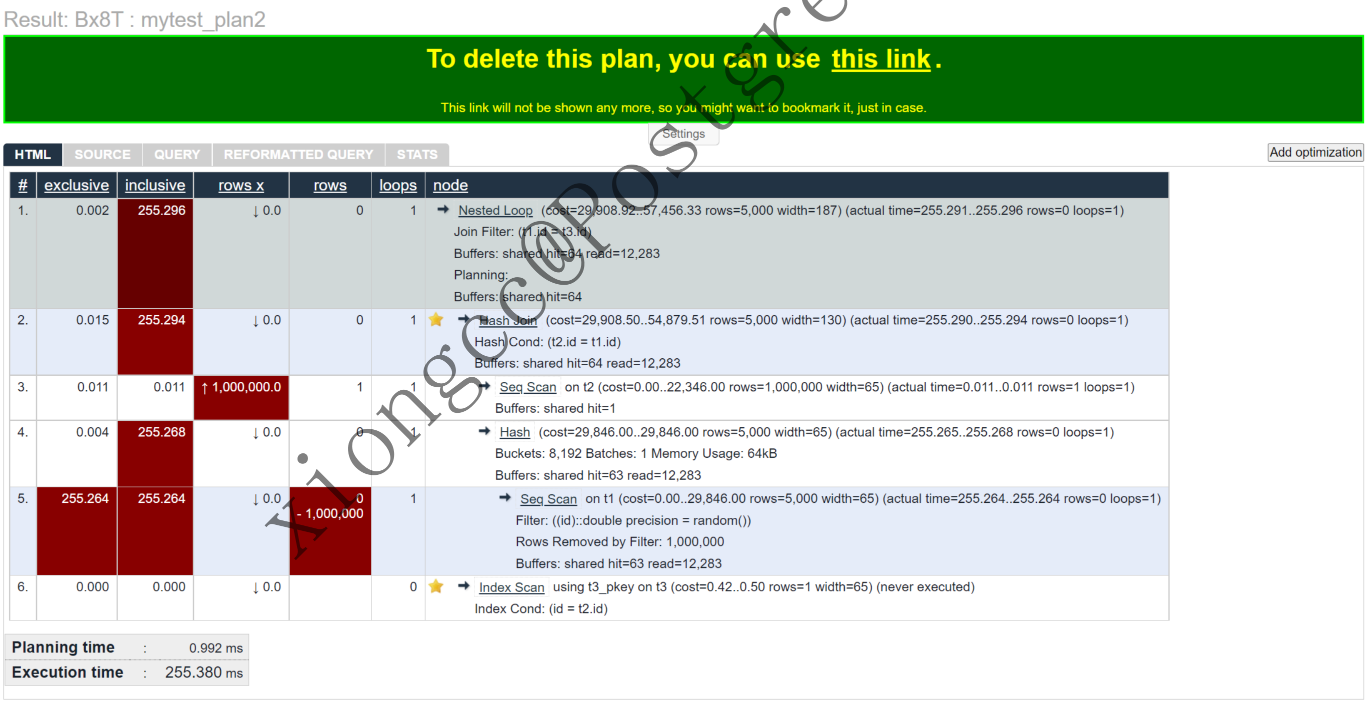
4. loops 指的是该节点实际的重启次数。如果一个计划节点在运行过程中，它的相关参数值（如绑定变量）发生了变化，就需要重新运行这个计划节点。

执行计划看起来与原有的执行计划有点相似，但从外观上看起来更加清晰。还有一些有用的附加特性：详细参考 https://explain.depesz.com/help#col-inclusive

1. 每个节点的总执行时间和净执行时间，耗时最高的节点用红色背景高亮显示，重点观察对象
2. inclusive：包含启动成本的总执行时间，会以由浅到深的颜色标识，对于特别大的会高亮显示，提醒我们这一块是优化重点
3. exclusive：较下层Node增长了多少时间，会以由浅到深的颜色标识，对于增长特别多的会高亮显示，提醒我们这一块是优化重点
4. rows x：这一列可以帮我们协助分析是什么因素让PostgreSQL错误地高估或低估了返回的行数。错误的估算会用红色背景高亮，此例中我们可以看到实际返回了1行，但是预估的行数是1000000，100万，可以看到，此时优化器严重估算错误 (实际原因是因为等于条件为random())
5. rows：就是实际的返回行数，以及通过过滤条件过滤了多少行，比如第5行，过滤了- 1000000行，实际返回了0行，说明选择率极高，可以考虑建合适的索引，比如万能Btree，适用于数组、多列的倒排Gin、时序数据Brin等
6. loops：就是实际的循环重启次数

当然还有一些其他有趣的特性：

我将鼠标放至顶层Node节点，会将子节点用⭐标记出来，这会很方便我们看下层Node节点



Result: Bx8T : mytest_plan2

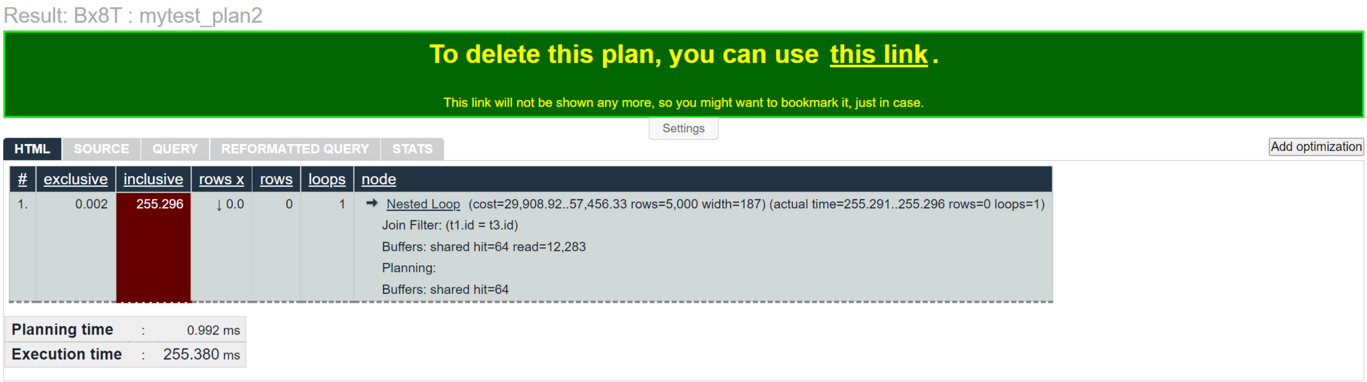
To delete this plan, you can use this link.

This link will not be shown any more, so you might want to bookmark it, just in case.

#	exclusive	inclusive	rows x	rows	loops	node
1.	0.002	255.296	↓ 0.0	0	1	→ Nested Loop (cost=29,908.92..57,456.33 rows=5,000 width=187) (actual time=255.291..255.296 rows=0 loops=1) Join Filter: (t1.id = t3.id) Buffers: shared hit=64 read=12,283 Planning: Buffers: shared hit=64
2.	0.015	255.294	↓ 0.0	0	1	⭐ → Hash Join (cost=29,908.50..54,879.51 rows=5,000 width=130) (actual time=255.290..255.294 rows=0 loops=1) Hash Cond: (t2.id = t1.id) Buffers: shared hit=64 read=12,283
3.	0.011	0.011	↑ 1,000,000.0	1	1	→ Seq Scan on t2 (cost=0.00..22,346.00 rows=1,000,000 width=65) (actual time=0.011..0.011 rows=1 loops=1) Buffers: shared hit=1
4.	0.004	255.268	↓ 0.0	0	1	→ Hash (cost=29,846.00..29,846.00 rows=5,000 width=65) (actual time=255.265..255.268 rows=0 loops=1) Buckets: 8,192 Batches: 1 Memory Usage: 64kB Buffers: shared hit=63 read=12,283
5.	255.264	255.264	↓ 0.0	0 - 1,000,000	1	→ Seq Scan on t1 (cost=0.00..29,846.00 rows=5,000 width=65) (actual time=255.264..255.264 rows=0 loops=1) Filter: ((id)::double precision = random()) Rows Removed by Filter: 1,000,000 Buffers: shared hit=63 read=12,283
6.	0.000	0.000	↓ 0.0		0	⭐ → Index Scan using t3_pkey on t3 (cost=0.42..0.50 rows=1 width=65) (never executed) Index Cond: (id = t2.id)

Planning time	:	0.992 ms
Execution time	:	255.380 ms

点击一下上层Node节点，会将子节点隐藏起来，对于特别长特别复杂的执行计划，会很方便我们分析SQL


Result: Bx8T : mytest_plan2

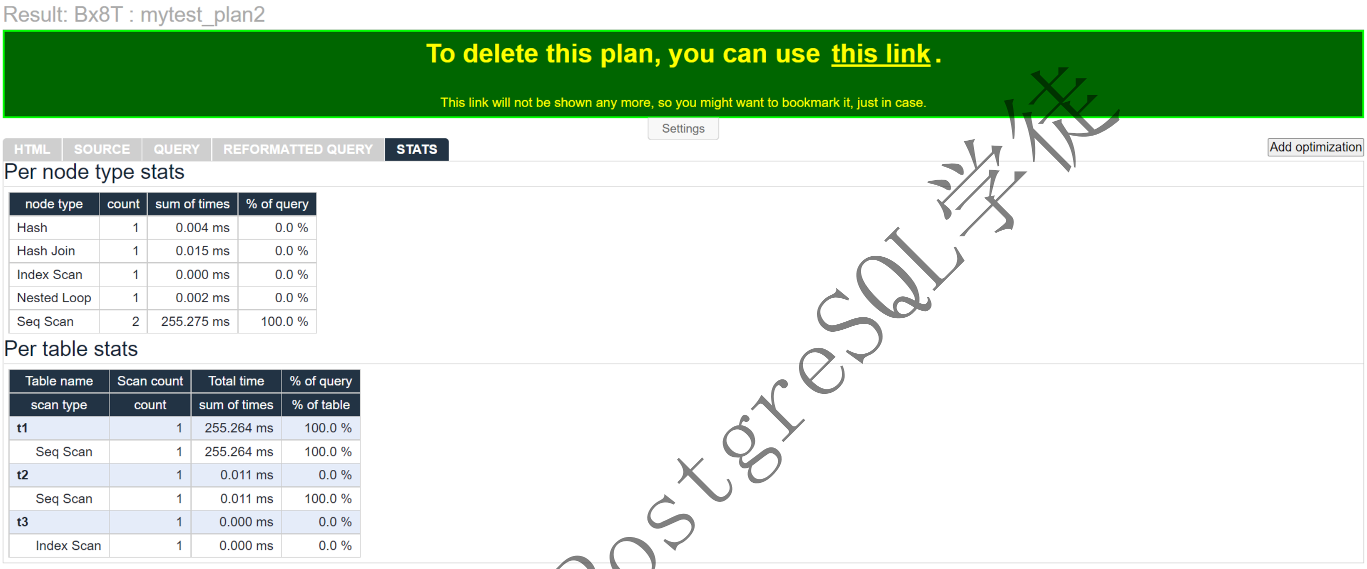
To delete this plan, you can use this link.

This link will not be shown any more, so you might want to bookmark it, just in case.

#	exclusive	inclusive	rows x	rows	loops	node
1.	0.002	255.296	↓ 0.0	0	1	→ Nested Loop (cost=29,908.92..57,456.33 rows=5,000 width=187) (actual time=255.291..255.296 rows=0 loops=1) Join Filter: (t1.id = t3.id) Buffers: shared hit=64 read=12,283 Planning: Buffers: shared hit=64

Planning time	:	0.992 ms
Execution time	:	255.380 ms

最后一个stats列则是SQL的总览信息


Result: Bx8T : mytest_plan2

To delete this plan, you can use this link.

This link will not be shown any more, so you might want to bookmark it, just in case.

Per node type stats

node type	count	sum of times	% of query
Hash	1	0.004 ms	0.0 %
Hash Join	1	0.015 ms	0.0 %
Index Scan	1	0.000 ms	0.0 %
Nested Loop	1	0.002 ms	0.0 %
Seq Scan	2	255.275 ms	100.0 %

Per table stats

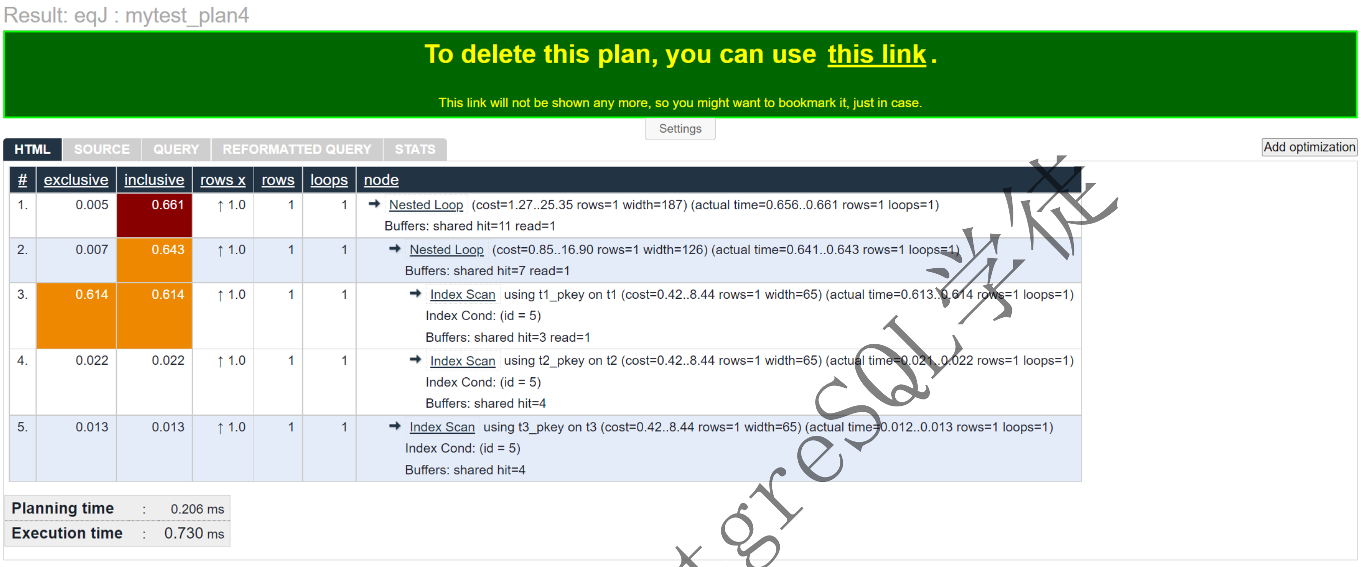
Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
t1	1	255.264 ms	100.0 %
Seq Scan	1	255.264 ms	100.0 %
t2	1	0.011 ms	0.0 %
Seq Scan	1	0.011 ms	100.0 %
t3	1	0.000 ms	0.0 %
Index Scan	1	0.000 ms	0.0 %

这里我们可以看到，我们的SQL实际跑起来，和统计信息的差距相差很多，因为我们的条件是等于random()，换成一个具体的值再看一下：

```
postgres=# explain (analyze,buffers) select * from t1 join t2 using (id) join t3 using (id) where t1.id= 5;
                                                                  QUERY PLAN

-----------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=1.27..25.35 rows=1 width=187) (actual time=0.656..0.661 rows=1 loops=1)
   Buffers: shared hit=11 read=1
   ->  Nested Loop  (cost=0.85..16.90 rows=1 width=126) (actual time=0.641..0.643 rows=1 loops=1)
         Buffers: shared hit=7 read=1
         ->  Index Scan using t1_pkey on t1  (cost=0.42..8.44 rows=1 width=65) (actual time=0.613..0.614 rows=1 loops=1)
               Index Cond: (id = 5)
               Buffers: shared hit=3 read=1
         ->  Index Scan using t2_pkey on t2  (cost=0.42..8.44 rows=1 width=65) (actual time=0.021..0.022 rows=1 loops=1)
```

```
        Index Cond: (id = 5)
        Buffers: shared hit=4
  ->  Index Scan using t3_pkey on t3  (cost=0.42..8.44 rows=1 width=65) (actual
time=0.012..0.013 rows=1 loops=1)
     Index Cond: (id = 5)
     Buffers: shared hit=4
 Planning Time: 0.206 ms
 Execution Time: 0.730 ms
(15 rows)
```



Result: eqJ : mytest_plan4

To delete this plan, you can use this link.

This link will not be shown any more, so you might want to bookmark it, just in case.

#	exclusive	inclusive	rows x	rows	loops	node
1.	0.005	0.661	↑ 1.0	1	1	→ Nested Loop (cost=1.27..25.35 rows=1 width=187) (actual time=0.656..0.661 rows=1 loops=1) Buffers: shared hit=11 read=1
2.	0.007	0.643	↑ 1.0	1	1	→ Nested Loop (cost=0.85..16.90 rows=1 width=126) (actual time=0.641..0.643 rows=1 loops=1) Buffers: shared hit=7 read=1
3.	0.614	0.614	↑ 1.0	1	1	→ Index Scan using t1_pkey on t1 (cost=0.42..8.44 rows=1 width=65) (actual time=0.613..0.614 rows=1 loops=1) Index Cond: (id = 5) Buffers: shared hit=3 read=1
4.	0.022	0.022	↑ 1.0	1	1	→ Index Scan using t2_pkey on t2 (cost=0.42..8.44 rows=1 width=65) (actual time=0.021..0.022 rows=1 loops=1) Index Cond: (id = 5) Buffers: shared hit=4
5.	0.013	0.013	↑ 1.0	1	1	→ Index Scan using t3_pkey on t3 (cost=0.42..8.44 rows=1 width=65) (actual time=0.012..0.013 rows=1 loops=1) Index Cond: (id = 5) Buffers: shared hit=4

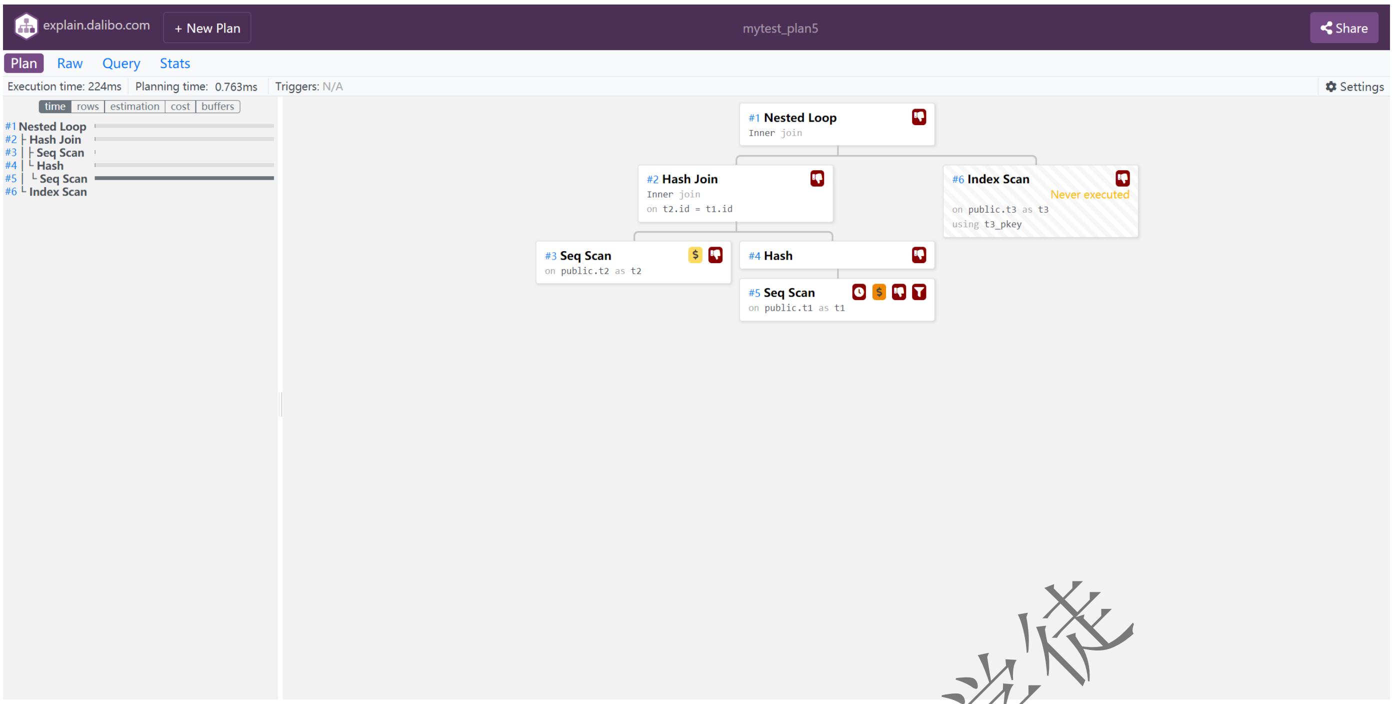
Planning time	:	0.206 ms
Execution time	:	0.730 ms

Dalibo’s EXPLAIN ANALYZE visualizer

又名 大力波！网址：https://explain.dalibo.com/，大力波官方建议我们使用

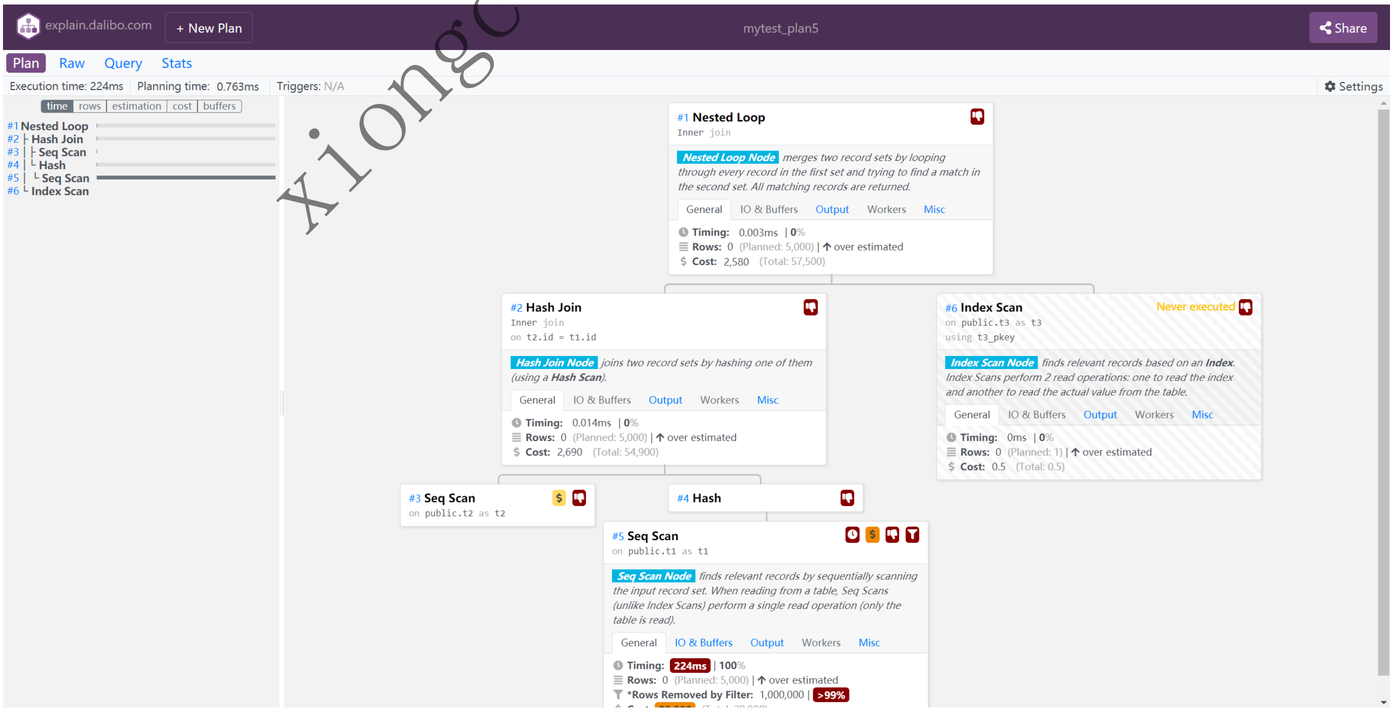
> For best results, use `EXPLAIN (ANALYZE, COSTS, VERBOSE, BUFFERS, FORMAT JSON)`

我们继续用原来的SQL看一下效果，和Depesz的是两种不同风格，左边是一个总览Overview，右边是具体的细节Details



点开即可看到Details，有一个很贴心的地方在于，大力波会把Scan Node的意思告诉你，所以对于看不懂Node的，通过这里也可以看出一二

> *Seq Scan Node finds relevant records by sequentially scanning the input record set. When reading from a table, Seq Scans (unlike Index Scans) perform a single read operation (only the table is read).*
>
> *Index Scan Node finds relevant records based on an* Index*. Index Scans perform 2 read operations: one to read the index and another to read the actual value from the table.*
>
> *Nested Loop Node merges two record sets by looping through every record in the first set and trying to find a match in the second set. All matching records are returned.*
>
> *Hash Join Node joins two record sets by hashing one of them (using a* Hash Scan*).*



Execution time: 224ms | Planning time: 0.763ms | Triggers: N/A

#1 Nested Loop
Inner join

Nested Loop Node merges two record sets by looping through every record in the first set and trying to find a match in the second set. All matching records are returned.

Timing: 0.003ms | 0%
Rows: 0 (Planned: 5,000) | ↑ over estimated
$ Cost: 2,580 (Total: 57,500)

#2 Hash Join
Inner join
on t2.id = t1.id

*Hash Join Node joins two record sets by hashing one of them (using a **Hash Scan**).*

Timing: 0.014ms | 0%
Rows: 0 (Planned: 5,000) | ↑ over estimated
$ Cost: 2,690 (Total: 54,900)

#6 Index Scan
Never executed
on public.t3 as t3
using t3_pkey

*Index Scan Node finds relevant records based on an **Index**. Index Scans perform 2 read operations: one to read the index and another to read the actual value from the table.*

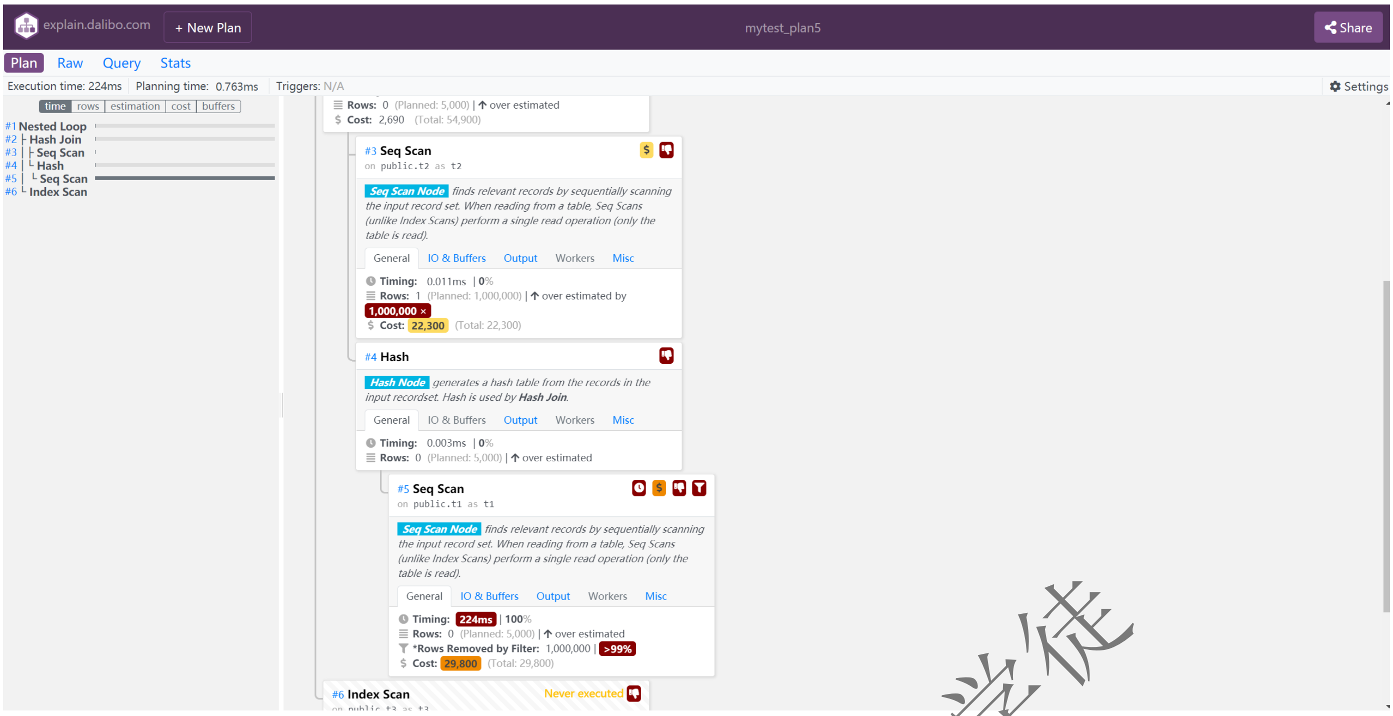
Timing: 0ms | 0%
Rows: 0 (Planned: 1) | ↑ over estimated
$ Cost: 0.5 (Total: 0.5)

#5 Seq Scan
on public.t1 as t1

Seq Scan Node finds relevant records by sequentially scanning the input record set. When reading from a table, Seq Scans (unlike Index Scans) perform a single read operation (only the table is read).

Timing: 224ms | 100%
Rows: 0 (Planned: 5,000) | ↑ over estimated
*Rows Removed by Filter: 1,000,000 | >99%

这种方式假如觉得抽象不易阅读的话，可以转化为传统风格，通过右上角的settings设置classic：



Execution time: 224ms | Planning time: 0.763ms | Triggers: N/A

```
#1 Nested Loop
#2 ├ Hash Join
#3 │ ├ Seq Scan
#4 │ └ Hash
#5 │   └ Seq Scan
#6 └ Index Scan
```

Rows: 0 (Planned: 5,000) | ↑ over estimated
$ Cost: 2,690 (Total: 54,900)

#3 **Seq Scan**
on public.t2 as t2

Seq Scan Node finds relevant records by sequentially scanning the input record set. When reading from a table, Seq Scans (unlike Index Scans) perform a single read operation (only the table is read).

General | IO & Buffers | Output | Workers | Misc

Timing: 0.011ms | 0%
Rows: 1 (Planned: 1,000,000) | ↑ over estimated by 1,000,000 ×
$ Cost: 22,300 (Total: 22,300)

#4 **Hash**

*Hash Node generates a hash table from the records in the input recordset. Hash is used by **Hash Join**.*

General | IO & Buffers | Output | Workers | Misc

Timing: 0.003ms | 0%
Rows: 0 (Planned: 5,000) | ↑ over estimated

#5 **Seq Scan**
on public.t1 as t1

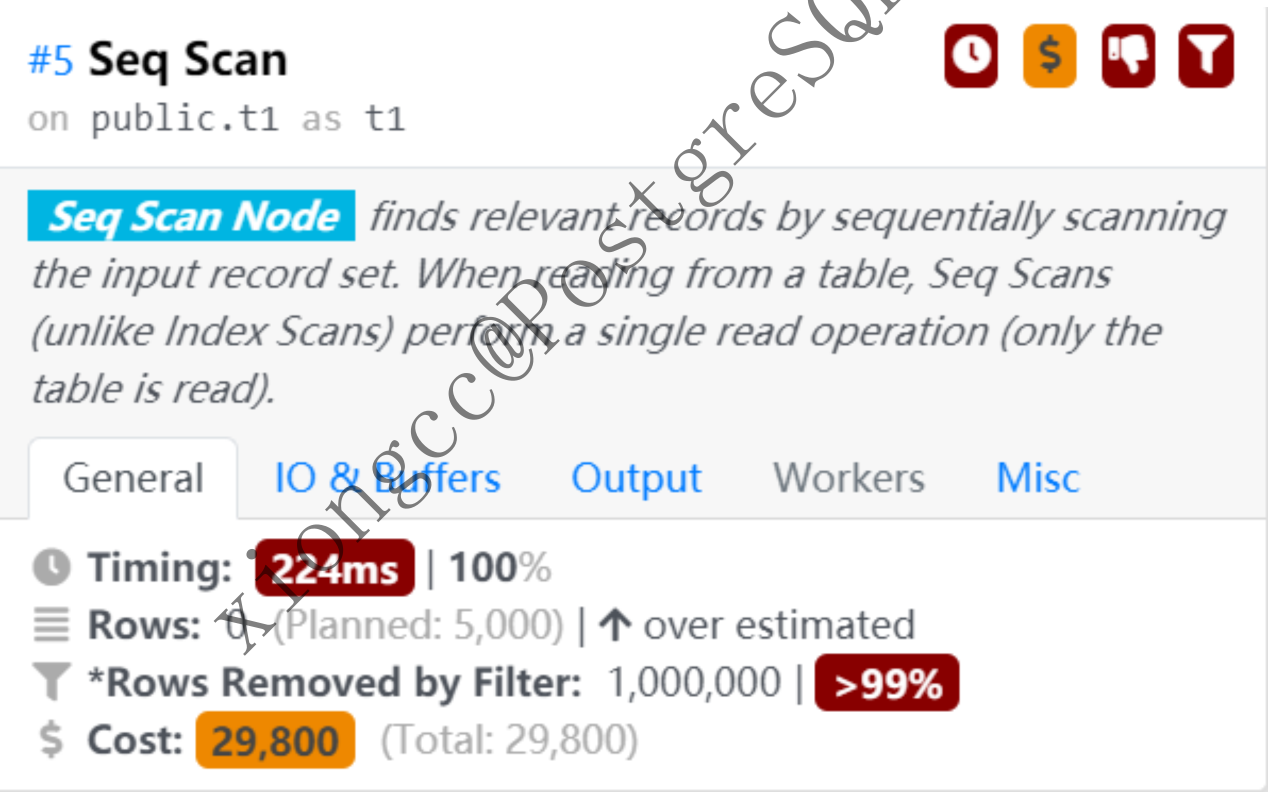
Seq Scan Node finds relevant records by sequentially scanning the input record set. When reading from a table, Seq Scans (unlike Index Scans) perform a single read operation (only the table is read).

General | IO & Buffers | Output | Workers | Misc

Timing: 224ms | 100%
Rows: 0 (Planned: 5,000) | ↑ over estimated
*Rows Removed by Filter: 1,000,000 | >99%
$ Cost: 29,800 (Total: 29,800)

#6 **Index Scan** Never executed
on public.t3 as t3

通过Node的Details，我们可以获取到如下信息，以Seq Scan为例：

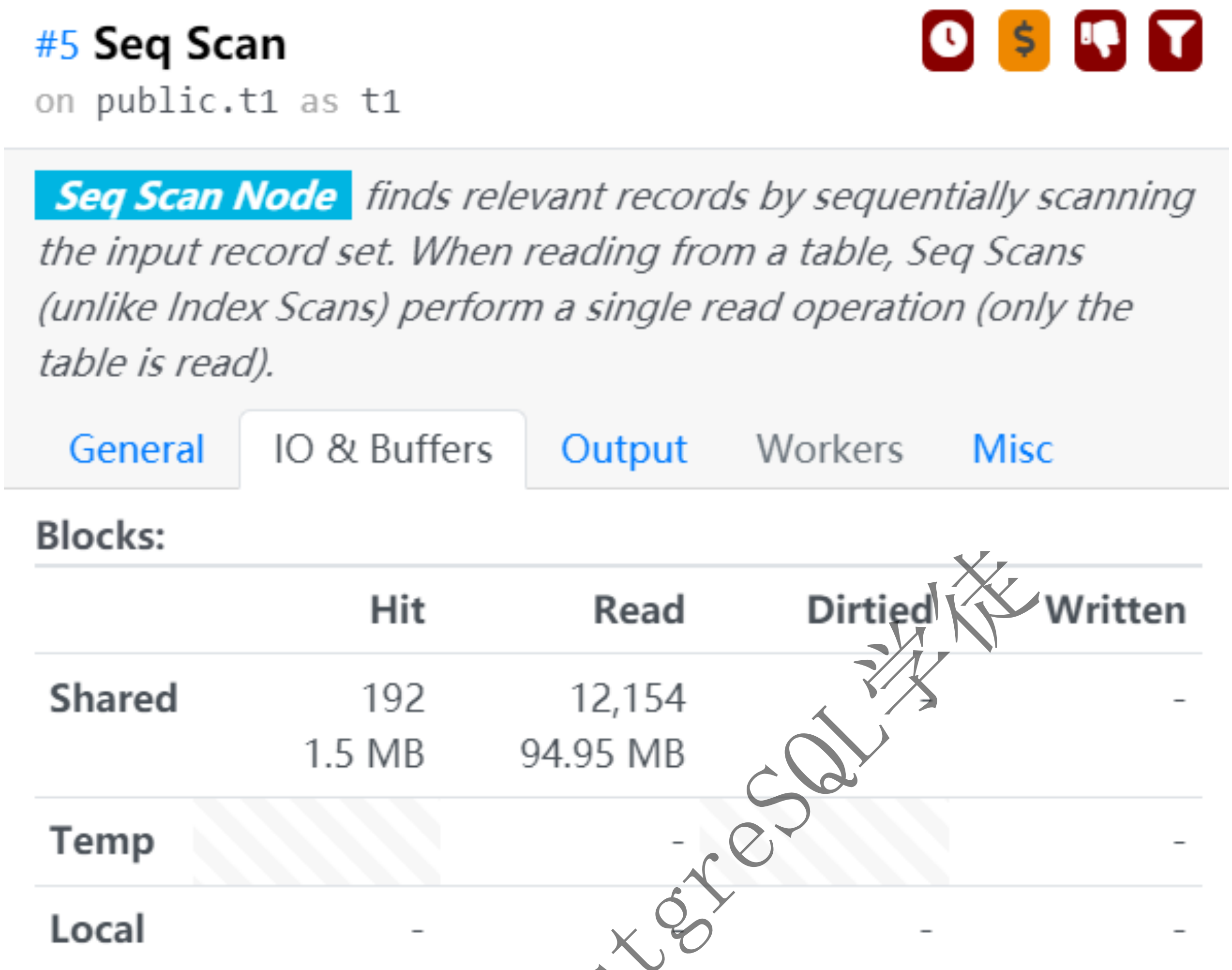


#5 **Seq Scan**
on public.t1 as t1

Seq Scan Node finds relevant records by sequentially scanning the input record set. When reading from a table, Seq Scans (unlike Index Scans) perform a single read operation (only the table is read).

General | IO & Buffers | Output | Workers | Misc

Timing: 224ms | 100%
Rows: 0 (Planned: 5,000) | ↑ over estimated
*Rows Removed by Filter: 1,000,000 | >99%
$ Cost: 29,800 (Total: 29,800)



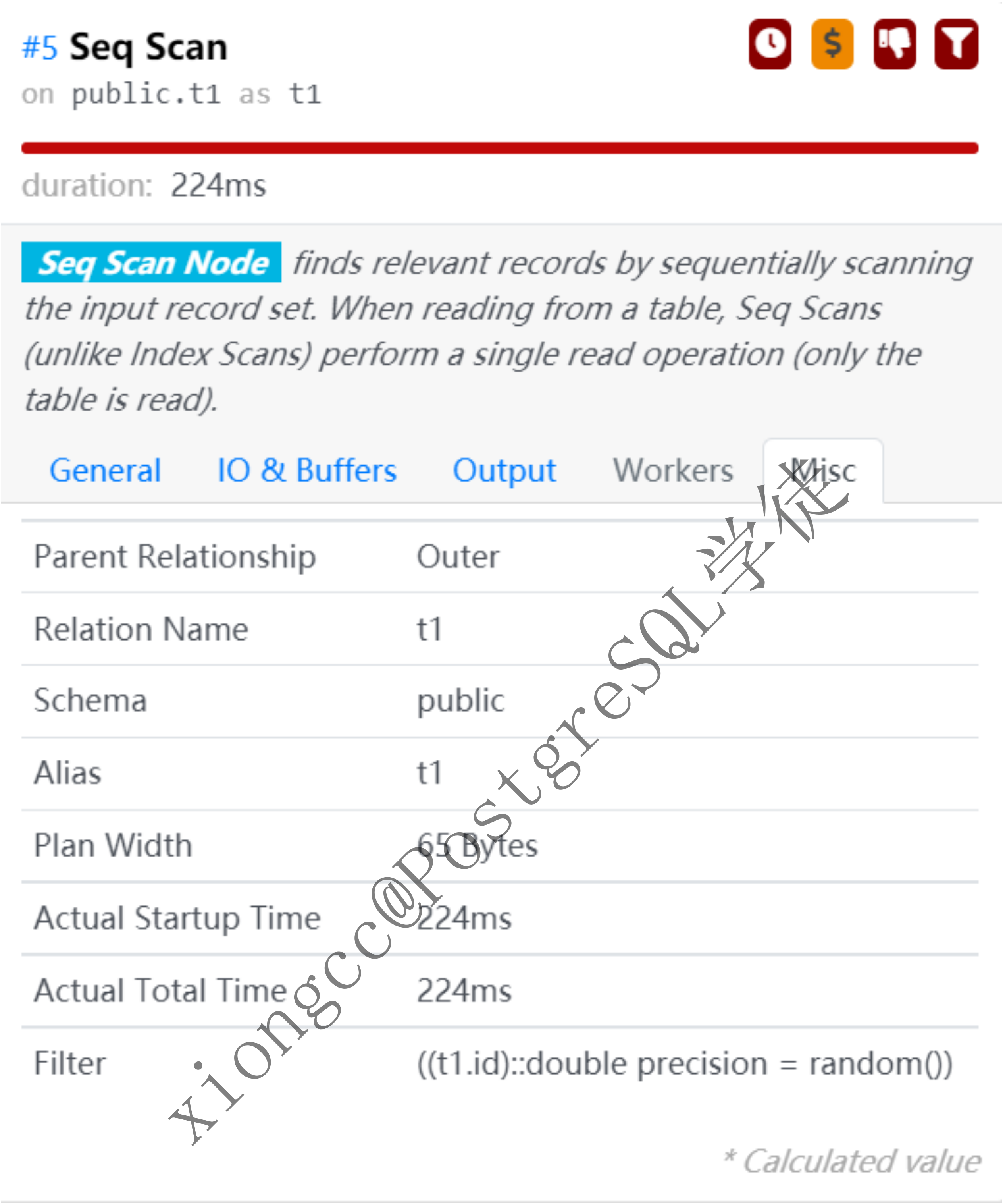
#5 **Seq Scan**

on `public.t1` as `t1`

Seq Scan Node finds relevant records by sequentially scanning the input record set. When reading from a table, Seq Scans (unlike Index Scans) perform a single read operation (only the table is read).

Blocks:

	Hit	Read	Dirtied	Written
Shared	192 1.5 MB	12,154 94.95 MB	-	-
Temp		-		-
Local	-		-	-



#5 **Seq Scan**

on public.t1 as t1

duration: 224ms

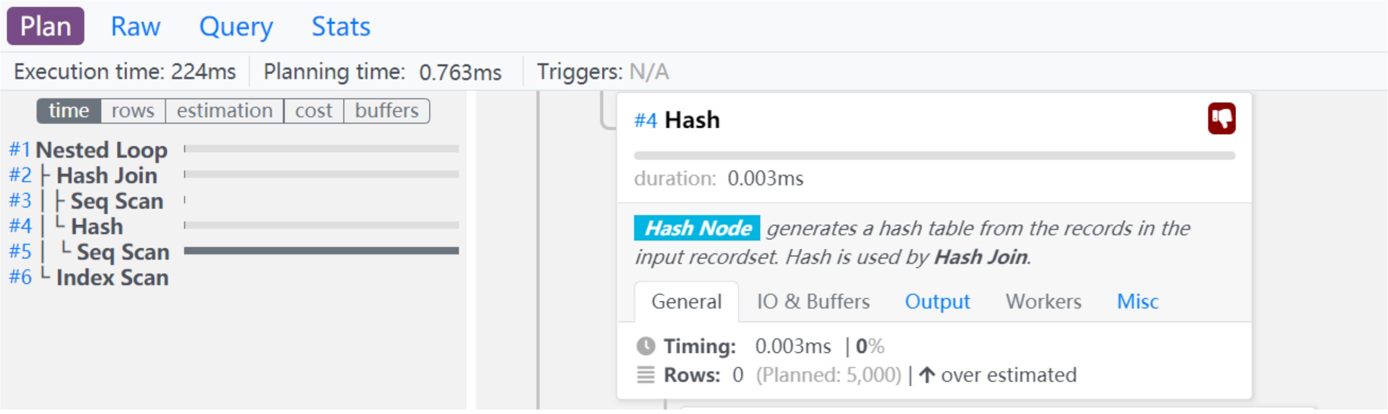
Seq Scan Node *finds relevant records by sequentially scanning the input record set. When reading from a table, Seq Scans (unlike Index Scans) perform a single read operation (only the table is read).*

Parent Relationship	Outer
Relation Name	t1
Schema	public
Alias	t1
Plan Width	65 Bytes
Actual Startup Time	224ms
Actual Total Time	224ms
Filter	((t1.id)::double precision = random())

** Calculated value*

1. 预估严重偏差，优化评估结果较真实结果多了5000行
2. 过滤了99%的行，意味着选择率极高，可以考虑建索引
3. 时间占用总SQL 100%的时间，也就是单条SQL基本全部耗时花在了这里，那么是我们重点观察的对象
4. 成本较高29800，颜色是较高级别橙色
5. buffers命中了1.5MB，读取了95MB，另外还会显示出Dirtyed，比如shared_buffer太小了，bgwriter进程没有维护好足够多的buffer，那么backend process得自己去找合适的block，假如更不巧还是脏块的话，还得做刷脏的累活

左侧是可选的输出什么，比如输出时间还是输出cost



Execution time: 224ms Planning time: 0.763ms Triggers: N/A

#1 Nested Loop
#2 ├ Hash Join
#3 │ ├ Seq Scan
#4 │ └ Hash
#5 │ └ Seq Scan
#6 └ Index Scan

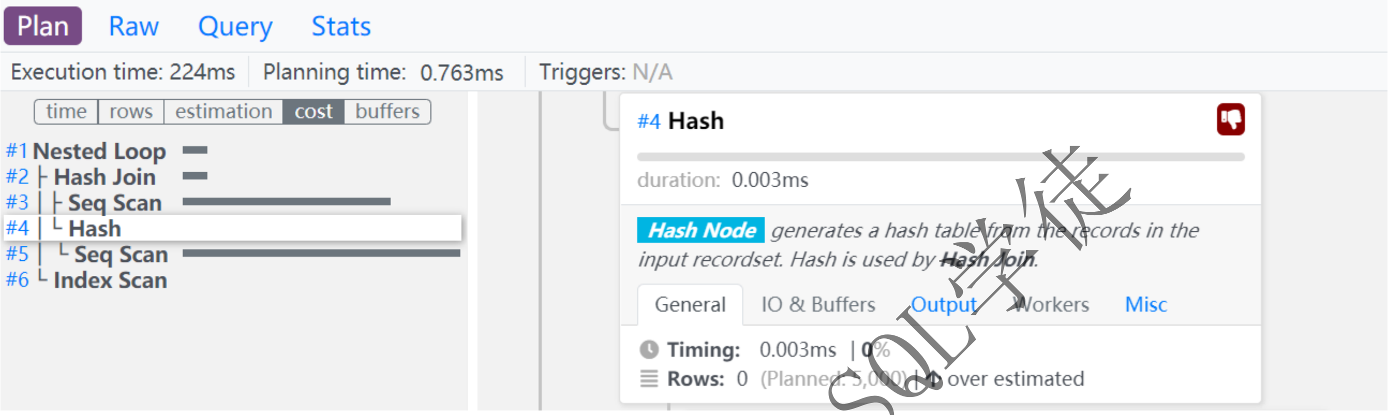
#4 Hash

duration: 0.003ms

*Hash Node generates a hash table from the records in the input recordset. Hash is used by **Hash Join**.*

Timing: 0.003ms | 0%

Rows: 0 (Planned: 5,000) | ↑ over estimated



Execution time: 224ms Planning time: 0.763ms Triggers: N/A

#1 Nested Loop
#2 ├ Hash Join
#3 │ ├ Seq Scan
#4 │ └ Hash
#5 │ └ Seq Scan
#6 └ Index Scan

#4 Hash

duration: 0.003ms

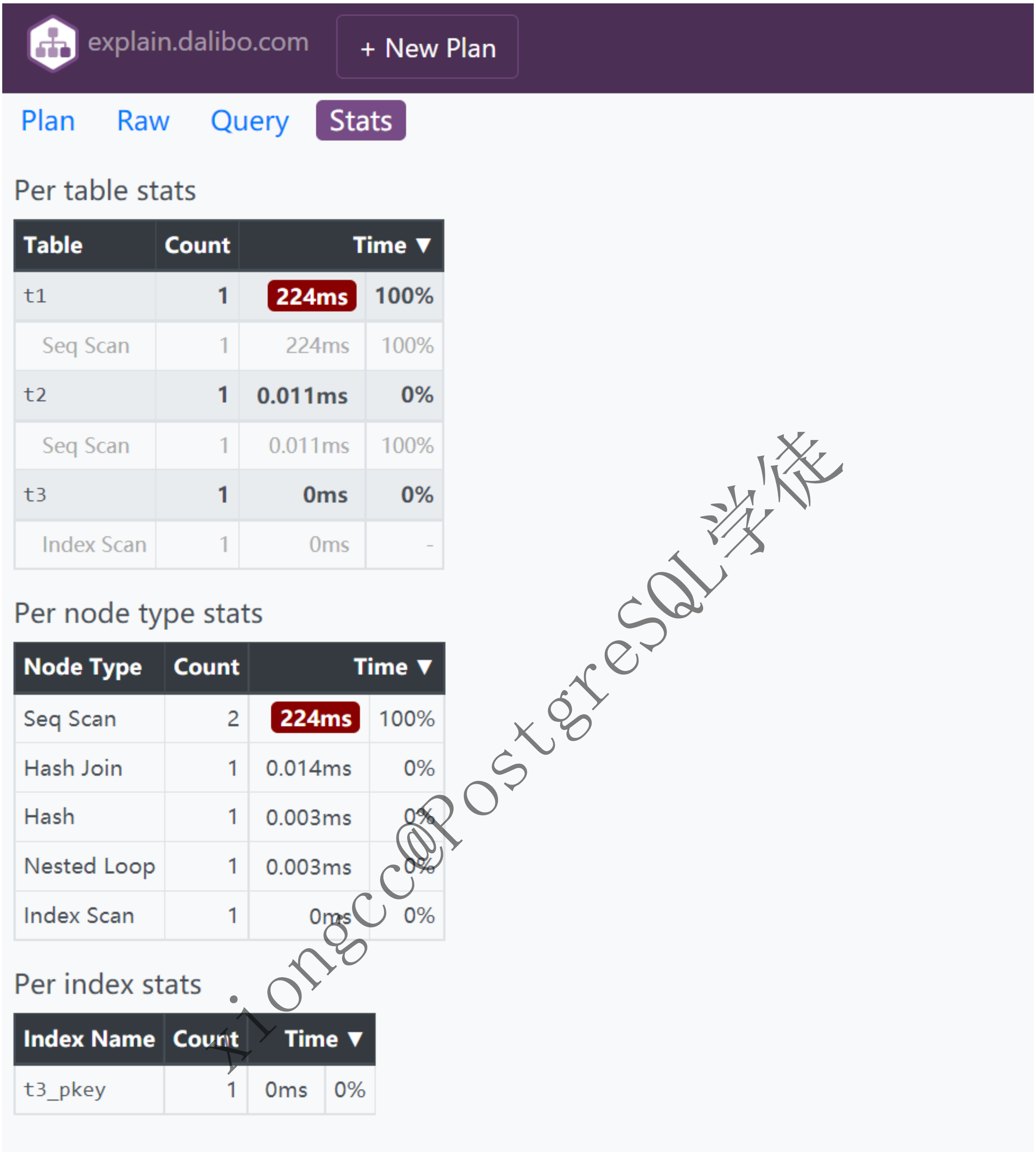
*Hash Node generates a hash table from the records in the input recordset. Hash is used by **Hash Join**.*

Timing: 0.003ms | 0%

Rows: 0 (Planned: 5,000) | ↑ over estimated

同理，还有一个总览stats

这里我要吐槽一下，我复制了半天始终提示我不是JSON格式，所以用一下样例吧。



Per table stats

Table	Count	Time ▼	
t1	1	224ms	100%
Seq Scan	1	224ms	100%
t2	1	0.011ms	0%
Seq Scan	1	0.011ms	100%
t3	1	0ms	0%
Index Scan	1	0ms	-

Per node type stats

Node Type	Count	Time ▼	
Seq Scan	2	224ms	100%
Hash Join	1	0.014ms	0%
Hash	1	0.003ms	0%
Nested Loop	1	0.003ms	0%
Index Scan	1	0ms	0%

Per index stats

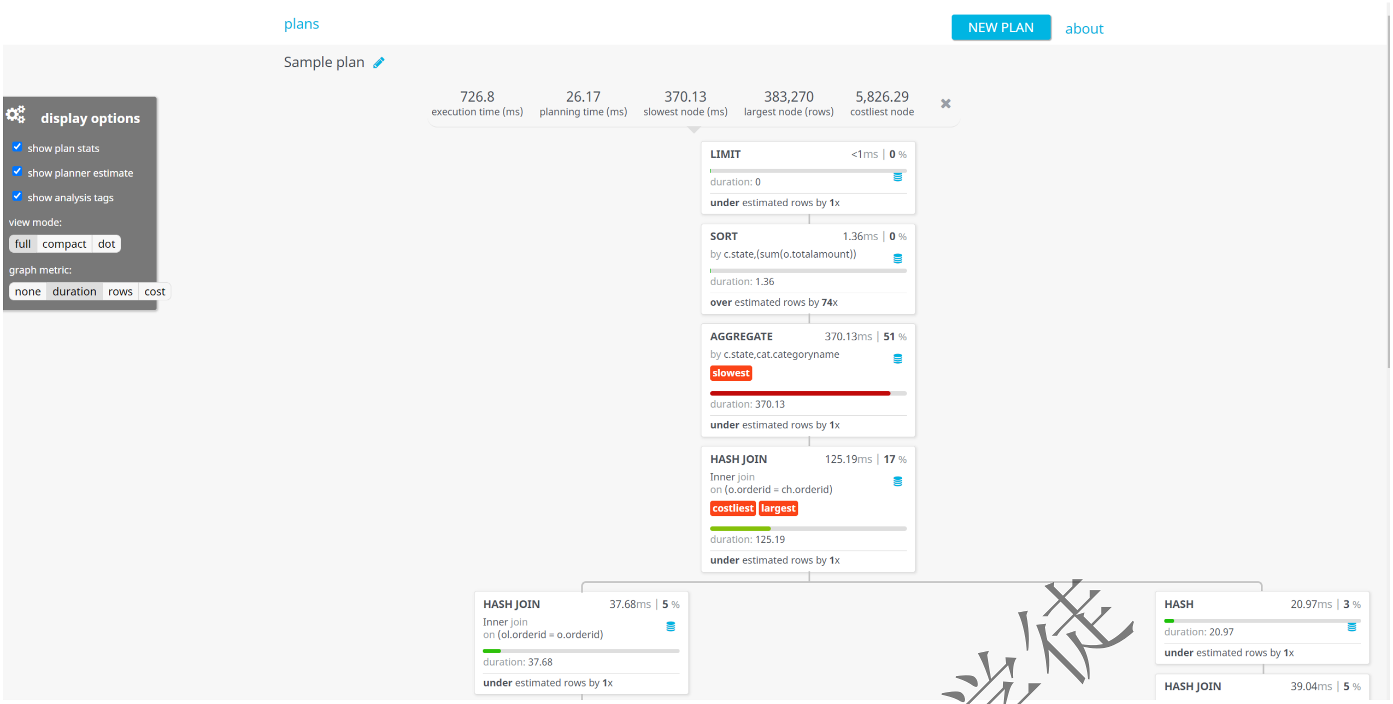
Index Name	Count	Time ▼	
t3_pkey	1	0ms	0%

Tatiyants’s EXPLAIN ANALYZE visualizer

https://tatiyants.com/pev/#/plans/new

这里我要吐槽一下，我复制了半天始终提示我不是JSON格式，所以用一下样例吧。

这个和大力波十分类似，可以看到slowest最慢的，costliest最昂贵的，返回结果集最大的largest的Node节点



Sample plan

726.8 execution time (ms) | 26.17 planning time (ms) | 370.13 slowest node (ms) | 383,270 largest node (rows) | 5,826.29 costliest node

LIMIT <1ms | 0 %
duration: 0
under estimated rows by 1x

SORT 1.36ms | 0 %
by c.state,(sum(o.totalamount))
duration: 1.36
over estimated rows by 74x

AGGREGATE 370.13ms | 51 %
by c.state,cat.categoryname
slowest
duration: 370.13
under estimated rows by 1x

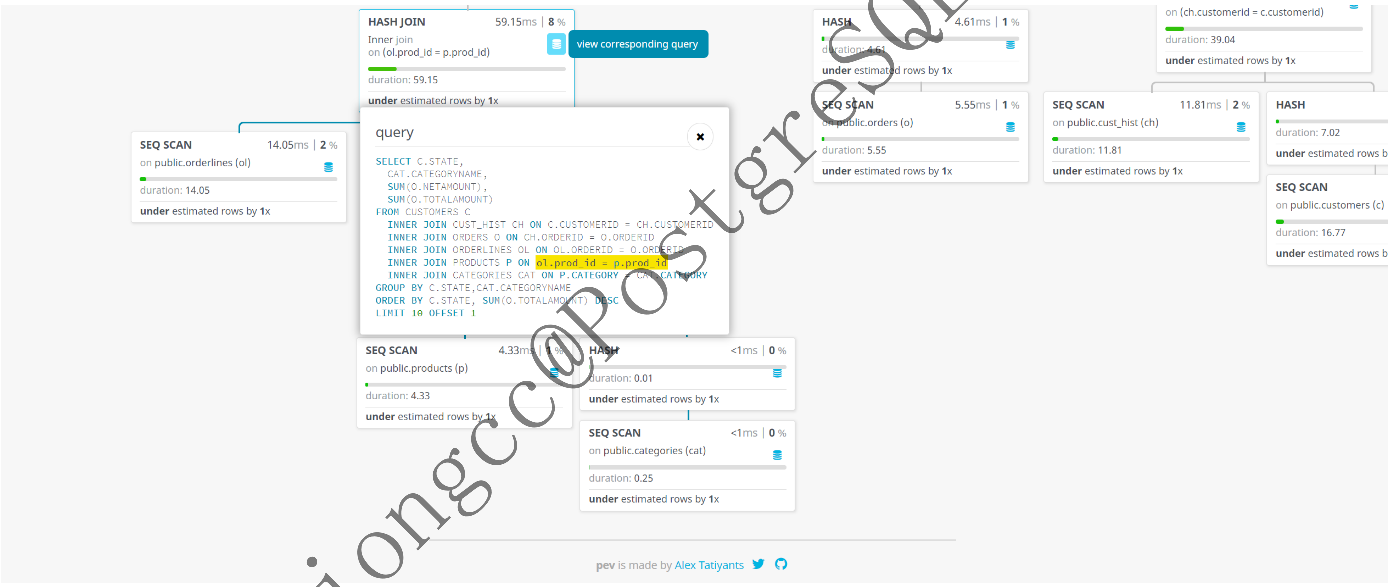
HASH JOIN 125.19ms | 17 %
Inner join
on (o.orderid = ch.orderid)
costliest largest
duration: 125.19
under estimated rows by 1x

HASH JOIN 37.68ms | 5 %
Inner join
on (ol.orderid = o.orderid)
duration: 37.68
under estimated rows by 1x

HASH 20.97ms | 3 %
duration: 20.97
under estimated rows by 1x

HASH JOIN 39.04ms | 5 %

不过有一点很棒，可以看到具体Node的SQL，如下，其余就不再赘述，和大力波类似。



HASH JOIN 59.15ms | 8 %
Inner join
on (ol.prod_id = p.prod_id)
duration: 59.15
under estimated rows by 1x

HASH 4.61ms | 1 %
duration: 4.61
under estimated rows by 1x

on (ch.customerid = c.customerid)
duration: 39.04
under estimated rows by 1x

SEQ SCAN 14.05ms | 2 %
on public.orderlines (ol)
duration: 14.05
under estimated rows by 1x

SEQ SCAN 5.55ms | 1 %
on public.orders (o)
duration: 5.55
under estimated rows by 1x

SEQ SCAN 11.81ms | 2 %
on public.cust_hist (ch)
duration: 11.81
under estimated rows by 1x

HASH
duration: 7.02
under estimated rows b

SEQ SCAN
on public.customers (c)
duration: 16.77
under estimated rows b

query

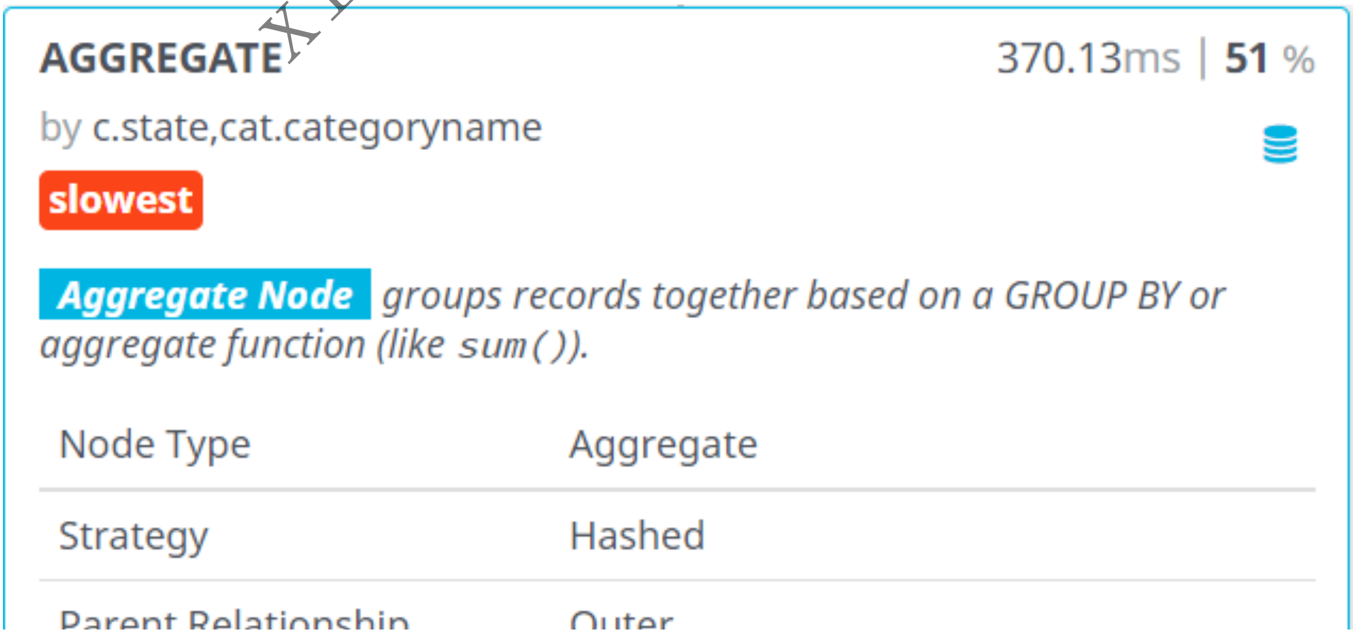
```
SELECT C.STATE,
  CAT.CATEGORYNAME,
  SUM(O.NETAMOUNT),
  SUM(O.TOTALAMOUNT)
FROM CUSTOMERS C
  INNER JOIN CUST_HIST CH ON C.CUSTOMERID = CH.CUSTOMERID
  INNER JOIN ORDERS O ON CH.ORDERID = O.ORDERID
  INNER JOIN ORDERLINES OL ON OL.ORDERID = O.ORDERID
  INNER JOIN PRODUCTS P ON ol.prod_id = p.prod_id
  INNER JOIN CATEGORIES CAT ON P.CATEGORY = CAT.CATEGORY
GROUP BY C.STATE,CAT.CATEGORYNAME
ORDER BY C.STATE, SUM(O.TOTALAMOUNT) DESC
LIMIT 10 OFFSET 1
```

SEQ SCAN 4.33ms | 1 %
on public.products (p)
duration: 4.33
under estimated rows by 1x

HASH <1ms | 0 %
duration: 0.01
under estimated rows by 1x

SEQ SCAN <1ms | 0 %
on public.categories (cat)
duration: 0.25
under estimated rows by 1x

pev is made by Alex Tatiyants



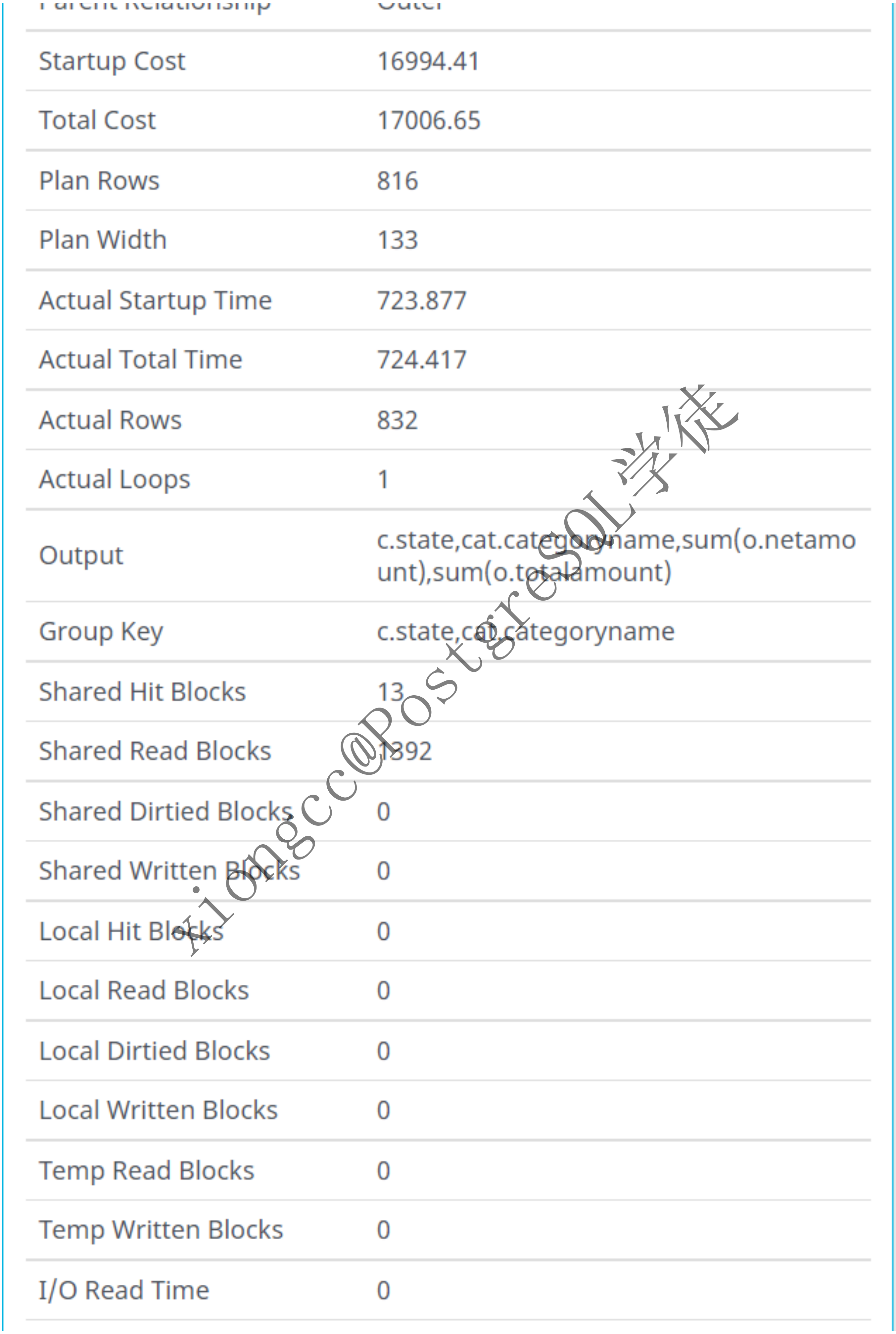
AGGREGATE 370.13ms | 51 %

by c.state,cat.categoryname

slowest

Aggregate Node groups records together based on a GROUP BY or aggregate function (like sum()).

Node Type	Aggregate
Strategy	Hashed
Parent Relationship	Outer

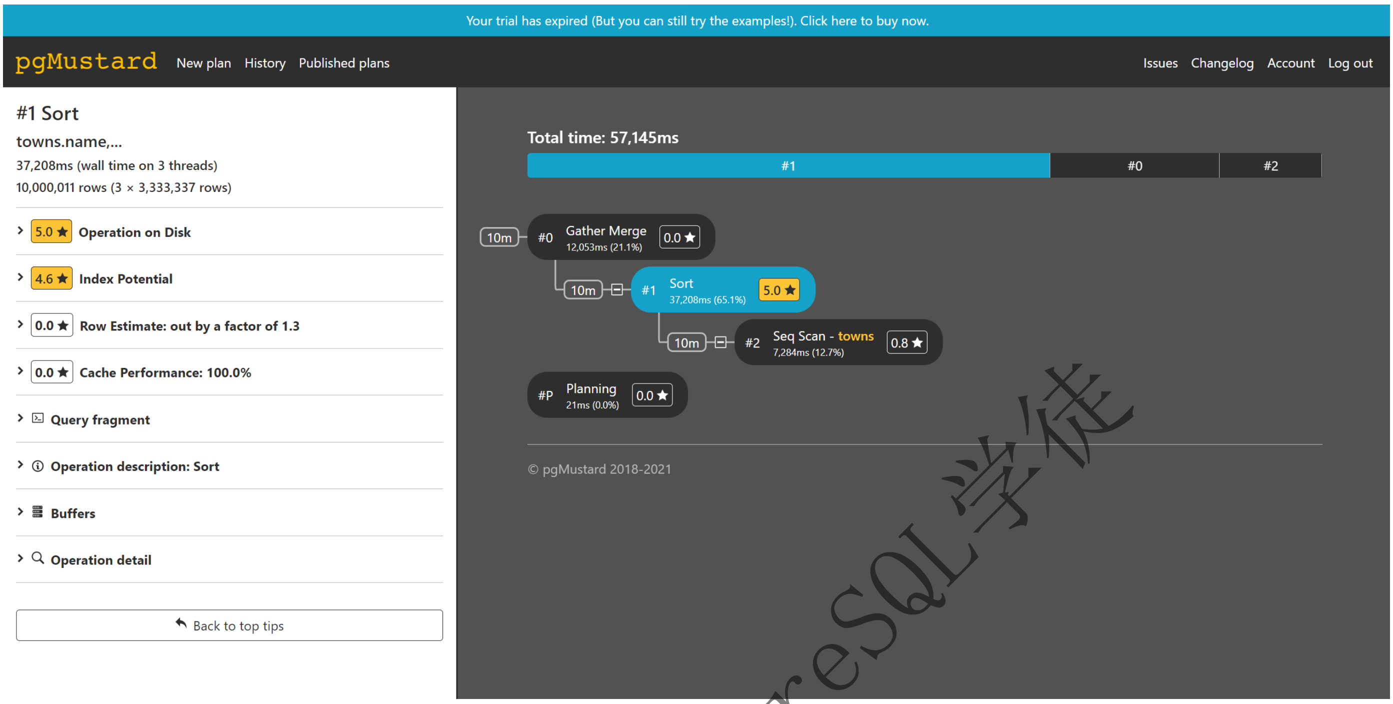


Parent Relationship	Outer
Startup Cost	16994.41
Total Cost	17006.65
Plan Rows	816
Plan Width	133
Actual Startup Time	723.877
Actual Total Time	724.417
Actual Rows	832
Actual Loops	1
Output	c.state,cat.categoryname,sum(o.netamount),sum(o.totalamount)
Group Key	c.state,cat.categoryname
Shared Hit Blocks	13
Shared Read Blocks	1392
Shared Dirtied Blocks	0
Shared Written Blocks	0
Local Hit Blocks	0
Local Read Blocks	0
Local Dirtied Blocks	0
Local Written Blocks	0
Temp Read Blocks	0
Temp Written Blocks	0
I/O Read Time	0

I/O Write Time	0

pgMustard

这个是一项收费的产品，https://www.pgmustard.com/，不过可以免费试用5次


Your trial has expired (But you can still try the examples!). Click here to buy now.
pgMustard
New plan
History
Published plans
Issues
Changelog
Account
Log out

#1 Sort

towns.name,...

37,208ms (wall time on 3 threads)

10,000,011 rows (3 × 3,333,337 rows)

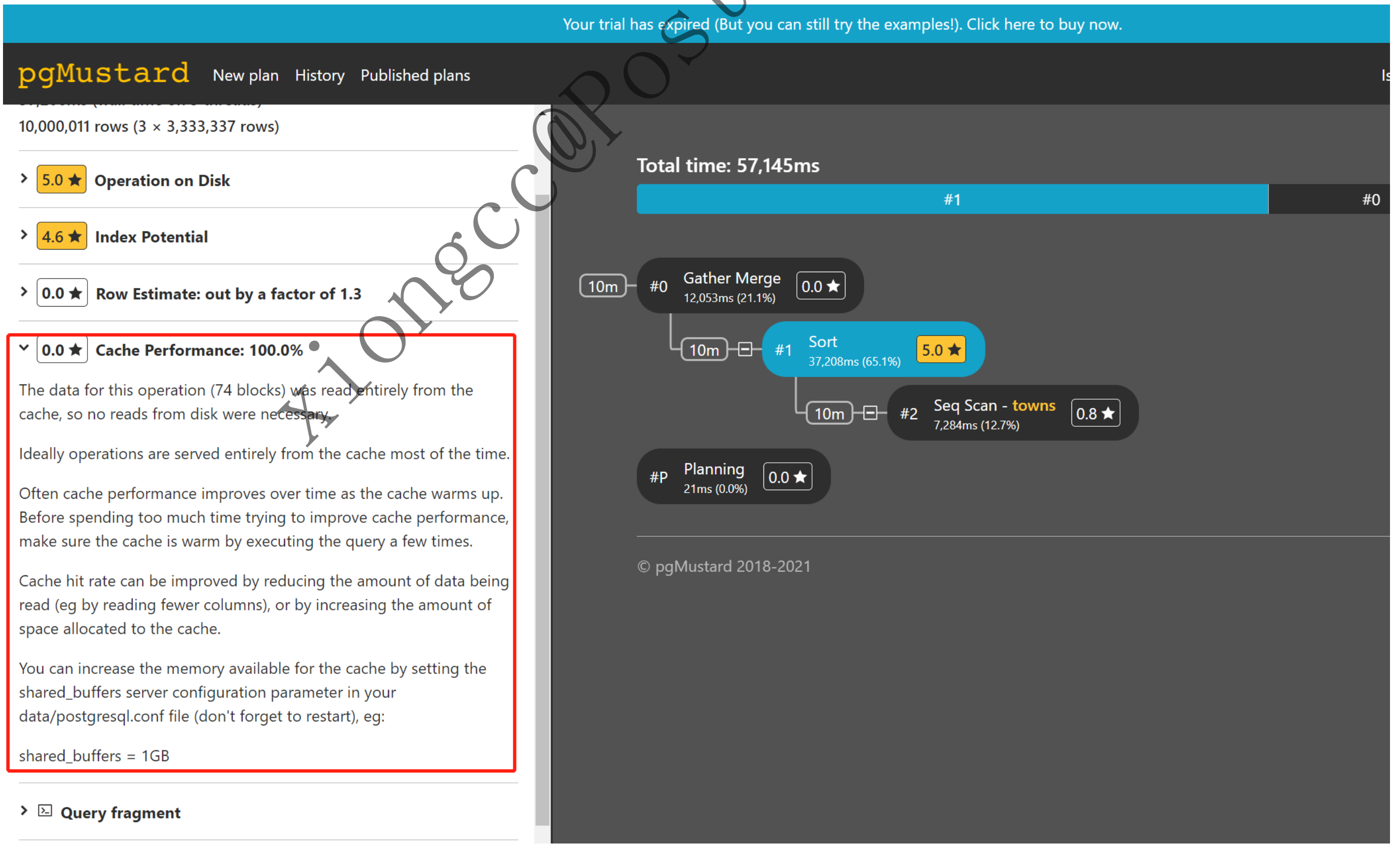
- 5.0 ★ Operation on Disk
- 4.6 ★ Index Potential
- 0.0 ★ Row Estimate: out by a factor of 1.3
- 0.0 ★ Cache Performance: 100.0%
- Query fragment
- Operation description: Sort
- Buffers
- Operation detail

Total time: 57,145ms

#1
#0
#2
10m
#0 Gather Merge 12,053ms (21.1%) 0.0 ★
10m
#1 Sort 37,208ms (65.1%) 5.0 ★
10m
#2 Seq Scan - towns 7,284ms (12.7%) 0.8 ★
#P Planning 21ms (0.0%) 0.0 ★

© pgMustard 2018-2021

毕竟收费嘛，功能更加高级一点，会输出相应的建议，对于新手来说，聊胜于无


Your trial has expired (But you can still try the examples!). Click here to buy now.
pgMustard
New plan
History
Published plans

10,000,011 rows (3 × 3,333,337 rows)

- 5.0 ★ Operation on Disk
- 4.6 ★ Index Potential
- 0.0 ★ Row Estimate: out by a factor of 1.3
- 0.0 ★ Cache Performance: 100.0%

The data for this operation (74 blocks) was read entirely from the cache, so no reads from disk were necessary.

Ideally operations are served entirely from the cache most of the time.

Often cache performance improves over time as the cache warms up. Before spending too much time trying to improve cache performance, make sure the cache is warm by executing the query a few times.

Cache hit rate can be improved by reducing the amount of data being read (eg by reading fewer columns), or by increasing the amount of space allocated to the cache.

You can increase the memory available for the cache by setting the shared_buffers server configuration parameter in your data/postgresql.conf file (don't forget to restart), eg:

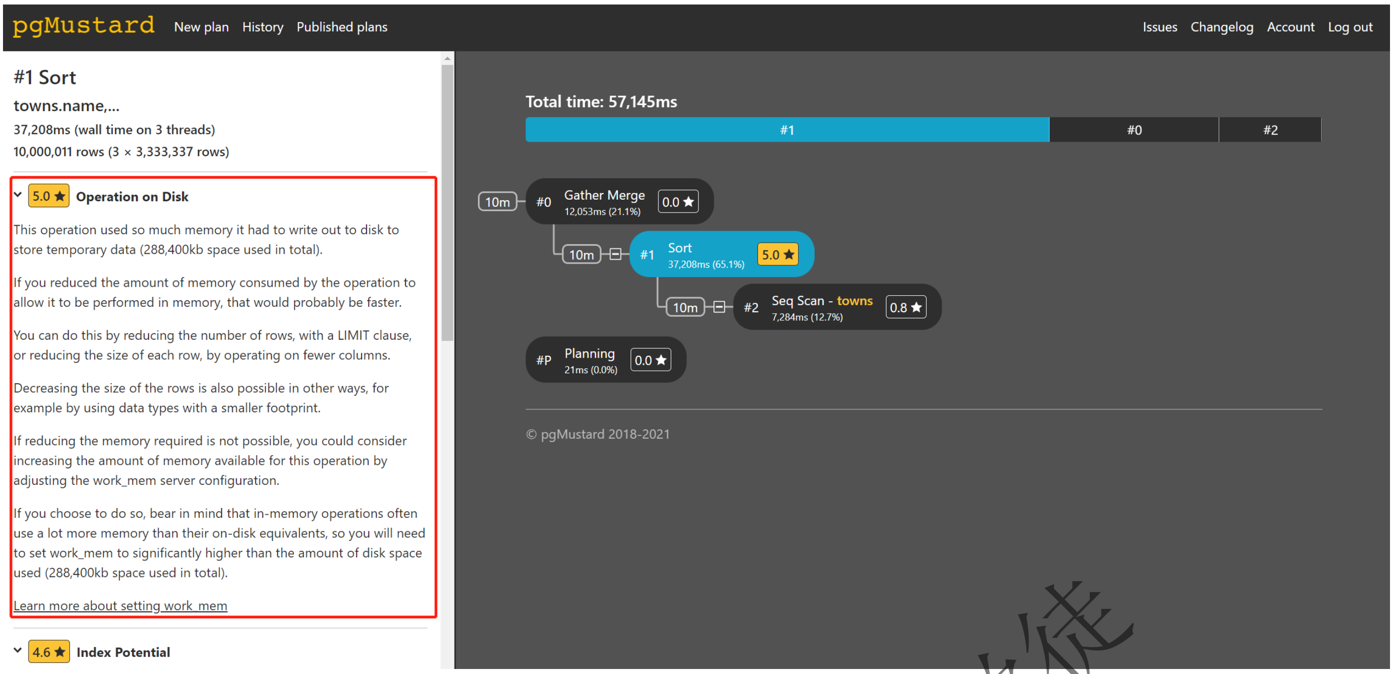
```
shared_buffers = 1GB
```

- Query fragment

Total time: 57,145ms

#1
#0
10m
#0 Gather Merge 12,053ms (21.1%) 0.0 ★
10m
#1 Sort 37,208ms (65.1%) 5.0 ★
10m
#2 Seq Scan - towns 7,284ms (12.7%) 0.8 ★
#P Planning 21ms (0.0%) 0.0 ★

© pgMustard 2018-2021



#1 Sort
towns.name,...
37,208ms (wall time on 3 threads)
10,000,011 rows (3 × 3,333,337 rows)

5.0 ★ Operation on Disk

This operation used so much memory it had to write out to disk to store temporary data (288,400kb space used in total).

If you reduced the amount of memory consumed by the operation to allow it to be performed in memory, that would probably be faster.

You can do this by reducing the number of rows, with a LIMIT clause, or reducing the size of each row, by operating on fewer columns.

Decreasing the size of the rows is also possible in other ways, for example by using data types with a smaller footprint.

If reducing the memory required is not possible, you could consider increasing the amount of memory available for this operation by adjusting the work_mem server configuration.

If you choose to do so, bear in mind that in-memory operations often use a lot more memory than their on-disk equivalents, so you will need to set work_mem to significantly higher than the amount of disk space used (288,400kb space used in total).

Learn more about setting work_mem

4.6 ★ Index Potential

Total time: 57,145ms

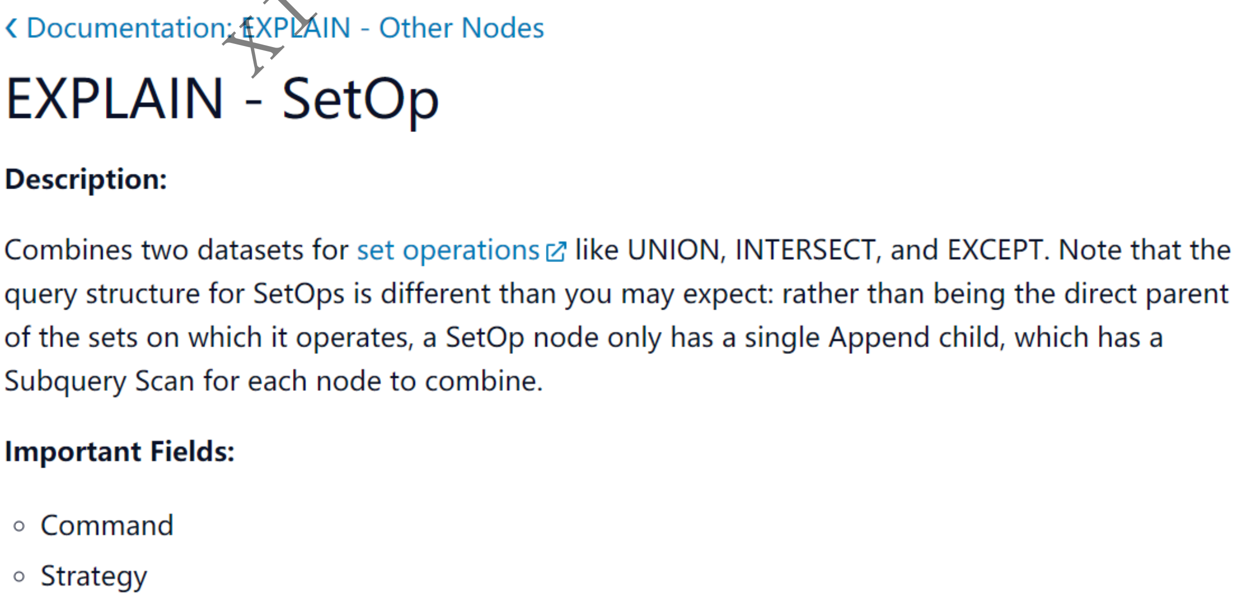
#1
#0
#2
10m
#0 Gather Merge 12,053ms (21.1%) 0.0 ★
10m
#1 Sort 37,208ms (65.1%) 5.0 ★
10m
#2 Seq Scan - towns 7,284ms (12.7%) 0.8 ★
#P Planning 21ms (0.0%) 0.0 ★

© pgMustard 2018-2021

pganalyze

pganalyze应该称作是一项解决方案，监控、调优、告警等，都囊括其中了。在此就不做演示了，感兴趣的可以自行尝试，有15天的试用期，虽然要收费，不过难不倒我们白嫖党，

我们可以当作是一个很好的学习案例，每个Node都会有解释：https://pganalyze.com/docs/explain/other-nodes/set-op


‹ Documentation: EXPLAIN - Other Nodes

EXPLAIN - SetOp

Description:

Combines two datasets for set operations like UNION, INTERSECT, and EXCEPT. Note that the query structure for SetOps is different than you may expect: rather than being the direct parent of the sets on which it operates, a SetOp node only has a single Append child, which has a Subquery Scan for each node to combine.

Important Fields:

- Command
- Strategy

除此之外，pganalyze比较体系化，可以通过阅读文档，了解该产品的思想，以及对各种瓶颈的分析和建议。

> Postgres Query Analysis & Postgres Explain Plans
>
> Discover the root cause of critical issues, optimize slow queries, and find missing indices.

小结

利用好工具，避免重复造轮子，最大化利用身边的资源，可以起到事半功倍的效果，待到神功练成之时，也就是各位出师之时。