第三篇 Part 3

提 高 篇

- 第 9 章　PostgreSQL 中执行计划
- 第 10 章　PostgreSQL 中的技术内幕
- 第 11 章　PostgreSQL 的特色功能
- 第 12 章　数据库优化
- 第 13 章　Standby 数据库的搭建

Chapter 9 第 9 章

PostgreSQL 中执行计划

9.1 执行计划的解释

9.1.1 EXPLAIN 命令

在关系型数据库中，一般使用 explain 命令来显示 SQL 的执行计划，只是不同的数据库中，这个命令的具体格式会有一些差别。

注意 在此先介绍 explain 的语法，如果对语法的一些解释看不明白，可以看后面的示例，通过示例会更明白 explain 命令的使用方法。

PostgreSQL 中 explain 命令的格式如下：

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

命令的可选选项“options”为：

```
ANALYZE [ boolean ]
VERBOSE [ boolean ]
COSTS [ boolean ]
BUFFERS [ boolean ]
FORMAT { TEXT | XML | JSON | YAML }
```

ANALYZE 选项通过实际执行的 SQL 来获得相应的执行计划。因为它真正被执行，所以可以看到执行计划每一步花掉了多少时间，以及它实际返回的行数。

注意 加上 ANALYZE 选项后，会真正执行实际的 SQL，如果 SQL 语句是一个插入、删除、更新或 CREATE TABLE AS 语句，这些语句会修改数据库。为了不想影响实际

的数据，可以把 EXPLAIN ANALYZE 放到一个事务中，执行完后回滚事务，如下：

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

VERBOSE 选项用于显示计划的附加信息。这些附加信息有：计划树中每个节点输出的各个列，如果触发器被触发，还会输出触发器的名称。该选项值默认为 FALSE。

COSTS 选项显示每个计划节点的启动成本和总成本，以及估计行数和每行宽度。该选项值默认为 TRUE。

BUFFERS 选项显示关于缓冲区使用的信息。该参数只能与 ANALYZE 参数一起使用。显示的缓冲区信息包括共享块、本地块和临时块读和写的块数。共享块、本地块和临时块分别包含表和索引、临时表和临时索引，以及在排序和物化计划中使用的磁盘块。上层节点显示出来的块数包括其所有子节点使用的块数。该选项值默认为 FALSE。

FORMAT 选项指定输出格式，输出格式可以是 TEXT、XML、JSON 或 YAML。非文本输出包含与文本输出格式相同的信息，但其他程序更容易解析。该参数默认为 TEXT。

9.1.2　EXPLAIN 输出结果解释

下面以一个最简单的 EXPLAIN 的输出结果做解释：

```
osdba=# explain select * from testtab01;
                          QUERY PLAN
---------------------------------------------------------------
Seq Scan on testtab01  (cost=0.00..184.00 rows=10000 width=36)
(1 row)
```

结果中“Seq Scan on testtab01”表示顺序扫描表“testtab01”，顺序扫描也就是全表扫描，即从头到尾地扫描表。后面的内容“(cost=0.00..184.00 rows=10000 width=36)”可以分为三部分。

- cost=0.00..184.00：“cost=”后面有两个数字，中间是由“..”分隔，第一个数字“0.00”表示启动的成本，也就是说返回第一行需要多少 cost 值；第二个数字表示返回所有的数据的成本，成本“cost”是什么后面会解释。
- rows=10000：表示会返回 10000 行。
- width=36：表示每行平均宽度为 36 字节。

下面解释什么是成本“cost”。成本“cost”描述一个 SQL 执行的代价是多少，默认情况下不同的操作其“cost”的值如下：

- 顺序扫描一个数据块，cost 值定为 1。
- 随机扫描一个数据块，cost 值定为 4。
- 处理一个数据行的 CPU，cost 为 0.01。
- 处理一个索引行的 CPU，cost 为 0.005。

❑ 每个操作符的 CPU 代价为 0.0025。

PostgreSQL 可以根据上面的操作类型，智能地计算出一个 SQL 的执行代价，虽然计算结果不是很精确，但多数情况下够用了。

更复杂的执行计划如下：

```
osdba=# explain select a.id,b.note from testtab01 a,testtab02 b where a.id=b.id;
                                QUERY PLAN
-------------------------------------------------------------------------------
Hash Join  (cost=309.00..701.57 rows=9102 width=36)
  Hash Cond: (b.id = a.id)
  ->  Seq Scan on testtab02 b  (cost=0.00..165.02 rows=9102 width=36)
  ->  Hash  (cost=184.00..184.00 rows=10000 width=4)
        ->  Seq Scan on testtab01 a  (cost=0.00..184.00 rows=10000 width=4)
(5 rows)
```

除了“Seq Scan”全表扫描外，还有一些其他的操作，如“Hash”、“Hash Join”等，这些内容后面会详细讲解。

9.1.3 EXPLAIN 使用示例

默认情况下，输出的执行计划是文本格式，但也可以输出 Json 格式，例如：

```
osdba=# explain (format json) select * from testtab01;
            QUERY PLAN
-------------------------------------
[                                    +
  {                                  +
    "Plan": {                        +
      "Node Type": "Seq Scan",       +
      "Relation Name": "testtab01",+
      "Alias": "testtab01",          +
      "Startup Cost": 0.00,          +
      "Total Cost": 184.00,          +
      "Plan Rows": 10000,            +
      "Plan Width": 36               +
    }                                +
  }                                  +
]
(1 row)
```

也能输出 xml 格式，如下：

```
osdba=# explain (format xml) select * from testtab01;
                       QUERY PLAN
-------------------------------------------------------------
<explain xmlns="http://www.postgresql.org/2009/explain">+
  <Query>                                               +
    <Plan>                                              +
      <Node-Type>Seq Scan</Node-Type>                   +
      <Relation-Name>testtab01</Relation-Name>          +
      <Alias>testtab01</Alias>                          +
      <Startup-Cost>0.00</Startup-Cost>                 +
      <Total-Cost>184.00</Total-Cost>                   +
```

```
      <Plan-Rows>10000</Plan-Rows>                    +
      <Plan-Width>36</Plan-Width>                     +
    </Plan>                                           +
  </Query>                                            +
</explain>
(1 row)
```

还可以输出 YAML 格式，如下：

```
osdba=# explain (format YAML ) select * from testtab01;
          QUERY PLAN
--------------------------------
- Plan:                         +
    Node Type: "Seq Scan"       +
    Relation Name: "testtab01"+
    Alias: "testtab01"          +
    Startup Cost: 0.00          +
    Total Cost: 184.00          +
    Plan Rows: 10000            +
    Plan Width: 36
(1 row)
```

加上“analyze”参数后，可通过实际执行来获得更精确的执行计划，命令如下：

```
osdba=# explain analyze select * from testtab01;
                                  QUERY PLAN
-------------------------------------------------------------------------------
Seq Scan on testtab01  (cost=0.00..184.00 rows=10000 width=36) (actual
   time=0.493..4.320 rows=10000 loops=1)
Total runtime: 5.653 ms
(2 rows)
```

从上面可以看出，加了“analyze”参数后，可以看到实际的启动时间（第一行返回的时间）、执行时间、实际的扫描的行数：(actual time=0.493..4.320 rows=10000 loops=1)，其中启动时间为 0.493 毫秒，返回所有行的时间为 4.320 毫秒，返回的行数是 10000。

analyze 选项也可以使用另一种语法，即放在小括号内，得到的结果与上面的完全一致，如下：

```
osdba=# explain (analyze true) select * from testtab01;
                                  QUERY PLAN
-------------------------------------------------------------------------------
Seq Scan on testtab01  (cost=0.00..184.00 rows=10000 width=36) (actual
   time=0.019..2.650 rows=10000 loops=1)
Total runtime: 4.004 ms
(2 rows)
```

如果只想查看执行的路径情况，不看 cost 值，则可以加“(cost false)”选项，命令如下：

```
osdba=# explain (costs false) select * from testtab01;
     QUERY PLAN
----------------------
Seq Scan on testtab01
(1 row)
```

联合使用 analyze 选项和 buffers 选项，可通过实际执行来查看实际的代价和缓冲区命中的情况，命令如下：

```
osdba=# explain (analyze true,buffers true ) select * from testtab03;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on testtab03  (cost=0.00..474468.18 rows=26170218 width=36) (actual
    time=0.498..8543.701 rows=10000000 loops=1)
   Buffers: shared hit=16284 read=196482 written=196450
 Total runtime: 9444.707 ms
(3 rows)
```

因为加了 buffers 选项，执行计划的结果中就会出现一行“Buffers: shared hit=16284 read=196482 written=196450”，这行中“shared hit=16284”表示在共享内存中直接读到 16284 个块，整行表示从磁盘中读了 16482 块，写磁盘共 196450 块。有人问 SELECT 为什么会写？这是因为共享内存中有脏块，从磁盘中读出的块必须把内存中的脏块挤出去，所以产生了很多的写。

要了解“create table as”的执行计划，命令如下：

```
osdba=# explain  create table testtab04 as select * from testtab03 limit 100000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.00..3127.66 rows=100000 width=142)
   ->  Seq Scan on testtab03  (cost=0.00..312766.02 rows=10000002 width=142)
(2 rows)
```

要了解 Insert 语句的执行计划，命令如下：

```
osdba=# explain insert into testtab04 select * from testtab03 limit 100000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Insert on testtab04  (cost=0.00..4127.66 rows=100000 width=142)
   ->  Limit  (cost=0.00..3127.66 rows=100000 width=142)
         ->  Seq Scan on testtab03  (cost=0.00..312766.02 rows=10000002
             width=142)
(3 rows)
```

要了解删除语句的执行计划，命令如下：

```
osdba=# explain delete from  testtab04;
                            QUERY PLAN
-------------------------------------------------------------------
 Delete on testtab04  (cost=0.00..22.30 rows=1230 width=6)
   ->  Seq Scan on testtab04  (cost=0.00..22.30 rows=1230 width=6)
(2 rows)
```

要了解更新语句的执行计划，命令如下：

```
osdba=# explain update testtab04 set note='bbbbbbbbbbbbbbb';
                            QUERY PLAN
-------------------------------------------------------------------
 Update on testtab04  (cost=0.00..22.30 rows=1230 width=10)
   ->  Seq Scan on testtab04  (cost=0.00..22.30 rows=1230 width=10)
```

```
(2 rows)
```

9.1.4　全表扫描

全表扫描在 PostgreSQL 也称为顺序扫描（seq scan），全表扫描就是把表的所有数据块从头到尾读一遍，然后从数据块中找到符合条件的数据块。

全表扫描在 EXPLAIN 命令输出的结果中用“Seq Scan”表示，如下：

```
osdba=# EXPLAIN SELECT * FROM testtab01;
                            QUERY PLAN
-------------------------------------------------------------
 Seq Scan on testtab01  (cost=0.00..2754.05 rows=151905 width=36)
(1 row)
```

9.1.5　索引扫描

索引通常是为了加快查询数据的速度而增加的。索引扫描，就是在索引中找出需要的数据行的物理位置，然后再到表的数据块中把相应的数据读出来的过程。

索引扫描在 EXPLAIN 命令输出的结果中用“Index Scan”表示，如下：

```
osdba=# EXPLAIN SELECT * FROM testtab01 where id=1000;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using idx_testtab01_id on testtab01  (cost=0.29..8.31 rows=1
    width=70)
   Index Cond: (id = 1000)
(2 rows)
```

9.1.6　位图扫描

位图扫描也是走索引的一种方式。方法是扫描索引，把满足条件的行或块在内存中建一个位图，扫描完索引后，再根据位图到表的数据文件中把相应的数据读出来。如果走了两个索引，可以把两个索引形成的位图进行“and”或“or”计算，合并成一个位图，再到表的数据文件中把数据读出来。

当执行计划的结果行数很多时会进行这种扫描，如非等值查询、IN 子句或有多个条件都可以走不同的索引时。

下面是非等值的一个示例：

```
osdba=# explain select * from testtab02 where id2 >10000;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on testtab02  (cost=18708.13..36596.06 rows=998155 width=16)
   Recheck Cond: (id2 > 10000)
   ->  Bitmap Index Scan on idx_testtab02_id2  (cost=0.00..18458.59 rows=998155
       width=0)
         Index Cond: (id2 > 10000)
(4 rows)
```

在位图扫描中可以看到，“Bitmap Index Scan”先在索引中找到符合条件的行，然后在内存中建立位图，之后再到表中扫描，也就是看到的“Bitmap Heap Scan”。

大家还会看到“Recheck Cond: (id2 > 10000)”，这是因为多版本的原因，当从索引找出的行从表中读出后，还需要再检查一下条件。

下面是一个因为“in”子句走到位图索引的示例：

```
osdba=# explain select * from testtab02 where id1 in (2,4,6,8);
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on testtab02  (cost=17.73..33.47 rows=4 width=16)
   Recheck Cond: (id1 = ANY ('{2,4,6,8}'::integer[]))
   ->  Bitmap Index Scan on idx_testtab02_id1  (cost=0.00..17.73 rows=4 width=0)
         Index Cond: (id1 = ANY ('{2,4,6,8}'::integer[]))
(4 rows)
```

下面是走了两个索引后，把位图进行了“BitmapOr”运算的示例：

```
osdba=# explain select * from testtab02 where id2 >10000 or  id1  <200000;
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on testtab02  (cost=20854.46..41280.46 rows=998446 width=16)
   Recheck Cond: ((id2 > 10000) OR (id1 < 200000))
   ->  BitmapOr  (cost=20854.46..20854.46 rows=1001000 width=0)
         ->  Bitmap Index Scan on idx_testtab02_id2  (cost=0.00..18458.59
              rows=998155 width=0)
               Index Cond: (id2 > 10000)
         ->  Bitmap Index Scan on idx_testtab02_id1  (cost=0.00..1896.65
              rows=102430 width=0)
               Index Cond: (id1 < 200000)
(7 rows)
```

在上面的执行计划中，可以看到“BitmapOr”的操作，这就表示使用“or”合并了两个位图。

9.1.7 条件过滤

条件过滤，一般就是在 where 条件上加的过滤条件，当扫描数据行时，会找出满足过滤条件的行。条件过滤在执行计划中显示为“Filter”，示例如下：

```
osdba=# EXPLAIN SELECT * FROM testtab01 where id<1000 and note like 'asdk%';
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Index Scan using idx_testtab01_id on testtab01  (cost=0.29..48.11 rows=1
width=70)
   Index Cond: (id < 1000)
   Filter: (note ~~ 'asdk%'::text)
```

如果条件的列上有索引，可能会走索引，不走过滤，如下：

```
osdba=# EXPLAIN SELECT * FROM testtab01 where  id<1000;
                                    QUERY PLAN
-------------------------------------------------------------------------------------
```

```
 Index Scan using idx_testtab01_id on testtab01   (cost=0.29..45.63 rows=991
    width=70)
   Index Cond: (id < 1000)
(2 rows)

osdba=# EXPLAIN SELECT * FROM testtab01 where  id>1000;
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on testtab01  (cost=0.00..2485.00 rows=99009 width=70)
   Filter: (id > 1000)
(2 rows)
```

9.1.8　Nestloop Join

嵌套循环连接（Nestloop Join）是在两个表做连接时最朴素的一种连接方式。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（> 10000 不适合），要把返回子集较小的表作为外表，而且在内表的连接字段上要有索引，否则会很慢。

执行的过程为：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。适用于驱动表的记录集比较小（<10000）而且 inner 表有有效的访问方法（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。

9.1.9　Hash Join

优化器使用两个表中较小的表，并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可以完全放于内存中的情况，这样总成本就是访问两个表的成本之和。但是如果表很大，不能完全放入内存，优化器会将它分割成若干不同的分区，把不能放入内存的部分写入磁盘的临时段，此时要有较大的临时段以便尽量提高 I/O 的性能。

下面就是一个 Hash Join 的示例：

```
osdba=# explain select a.id,b.id,a.note from testtab01 a, testtab02 b where a.id=b.id
    and b.id<=1000000;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Hash Join  (cost=20000041250.75..20000676975.71 rows=999900 width=93)
   Hash Cond: (a.id = b.id)
   ->  Seq Scan on testtab01 a   (cost=10000000000.00..10000253847.55
       rows=10000055 width=89)
   ->  Hash  (cost=10000024846.00..10000024846.00 rows=999900 width=4)
         ->  Seq Scan on testtab02 b   (cost=10000000000.00..10000024846.00
             rows=999900 width=4)
               Filter: (id <= 1000000)
(6 rows)
```

然后看表大小，如下：

```
osdba=# select pg_relation_size('testtab01');
 pg_relation_size
------------------
       1260314624
(1 row)

osdba=# select pg_relation_size('testtab02');
 pg_relation_size
------------------
        101138432
(1 row)
```

因为表“'testtab01”大于“'testtab02”，所以 HashJoin 是先在较小的表“testtab02”上建立散列表，然后扫描较大的表“testtab01”并探测散列表，找出与散列表匹配的行。

9.1.10 Merge Join

通常情况下散列连接的效果比合并连接好，然而如果源数据上有索引，或者结果已经被排过序，在执行排序合并连接时就不需要排序了，这时合并连接的性能会优于散列连接。

下面的示例中，表 testtab01 和表 testtab02 的 id 字段上都有索引，且从索引扫描的数据已经排好序了，可以直接走 Merge Join 了：

```
osdba=# explain select a.id,b.id,a.note from testtab01 a, testtab02 b where a.id=b.
    id and b.id<=100000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Merge Join  (cost=1.47..47922.57 rows=99040 width=93)
   Merge Cond: (a.id = b.id)
   ->  Index Scan using idx_testtab01_id on testtab01 a  (cost=0.43..413538.43
        rows=10000000 width=89)
   ->  Index Only Scan using idx_testtab02_id on testtab02 b  (cost=0.42..4047.63
        rows=99040 width=4)
         Index Cond: (id <= 100000)
(5 rows)
```

现在把表 testtab02 上的索引删除，下面示例中的执行计划是对 testtab02 排序后再走 Merge Join：

```
osdba=# drop index idx_testtab02_id;
DROP INDEX
osdba=# explain select a.id,b.id,a.note from testtab01 a, testtab02 b where a.id=b.
    id and b.id<=100000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Merge Join  (cost=34419.21..78788.84 rows=99040 width=93)
   Merge Cond: (a.id = b.id)
   ->  Index Scan using idx_testtab01_id on testtab01 a  (cost=0.43..413538.43
        rows=10000000 width=89)
   ->  Materialize  (cost=34418.70..34913.90 rows=99040 width=4)
         ->  Sort  (cost=34418.70..34666.30 rows=99040 width=4)
```

```
                Sort Key: b.id
                ->  Seq Scan on testtab02 b  (cost=0.00..24846.00 rows=99040
                     width=4)
                      Filter: (id <= 100000)
(8 rows)
```

从上面的执行计划可以看到“Sort Key：b.id”，这就是对表 testtab02 的 id 字段排序。

9.2　与执行计划相关的配置项

9.2.1　ENABLE_* 参数

在 PostgreSQL 中有一些以“ENABLE_”开头的参数，这些参数提供了影响查询优化器选择不同执行计划的方法。有时，如果优化器为特定查询选择的执行计划并不是最优的，可以设置这些参数强制优化器选择一个更好的执行计划来临时解决这个问题，但一般不会在 postgresql 中改变这些参数值的默认值。因为在通常情况下，PostgreSQL 都不会走错执行计划。PostgreSQL 走错执行计划是统计信息收集得不及时导致的，可通过更频繁地运行 ANALYZE 来解决这个问题，使用“ENABLE_”只是一个临时的方法。这些参数的详细说明见表9-1。

表 9-1　ENABLE_* 参数

参数名称	类　型	说　明
enable_seqscan	boolean	是否选择全表顺序扫描。实际上并不能完全禁止全表扫描，但是把这个变量关闭会让优化器在存在其他方法时优先选择其他方法
enable_indexscan	boolean	是否选择索引扫描
enable_bitmapscan	boolean	是否选择位图扫描
enable_tidscan	boolean	是否选择位图扫描
enable_nestloop	boolean	多表连接时，是否选择嵌套循环连接。如果设置为“off”，执行计划只有走嵌套循环连接一条路时，优化器也只能选择这条路，但如果有其他连接方法可以走，优化器会优先选择其他方法
enable_hashjoin	boolean	多表连接时，是否选择 hash 连接
enable_mergejoin	boolean	多表连接时，是否选择 merge 连接
enable_hashagg	boolean	是否使用 hash 聚合
enable_sort	boolean	是否使用明确的排序，如果设置为“off”，执行计划只有排序一条路时，优化器也只能选择这条路，但如果有其他方法可以走，优化器会优先选择其了方法

9.2.2　COST 基准值参数

执行计划在选择最优路径时，不同路径的 cost 值只有相对意义，同时缩放它们将不会对不同路径的选择产生任何影响。默认情况下，它们以顺序扫描一个数据块的开销作为基准单位，也就是说将顺序扫描的基准参数 seq_page_cost 默认设为 1.0，其他开销的基准参数都对

照它来设置。从理论上来说也可以使用其他基准方法，如用以毫秒计的实际执行时间做基准，但这些基准方法可能会更复杂一些。这些 COST 基准值参数如表 9-2 所示。

表 9-2 COST 基准值参数

参数名称	类 型	说 明
seq_page_cost	float	执行计划中一次顺序访问一个数据块页面的开销。默认值是 1.0
random_page_cost	float	执行计划中计算随机访问一个数据块页面的开销。默认值是 4.0，也就是说随机访问一个数据块页的开销是顺序访问的 4 倍
cpu_tuple_cost	float	执行计划中计算处理一条数据行的开销。默认值为 0.01
cpu_index_tuple_cost	float	执行计划中计算处理一条索引行的开销。默认为 0.005
cpu_operator_cost	float	执行计划中执行一个操作符或函数的开销。默认为 0.0025
effective_cache_size	int	执行计划中在一次索引扫描中可用的磁盘缓冲区的有效大小。在计算一个索引的预计开销值时会对这个参数加以考虑。更高的数值会导致更可能使用索引扫描，更低的数值会导致更有可能选择顺序全表扫描。这个参数对 PostgreSQL 分配的共享内存大小没有任何影响，它只用于执行计划中代价的估算。数值是用数据页来计算的，通常每个页面是 8KB 大小。默认是 16384 个数据块大小，即 128MB

在上面的配置项中，“seq_page_cost”一般作为基准，不用改变。可能需要改变的是“random_page_cost”，如果在读数据时，数据基本都命中在内存中，这时随机读和顺序读的差异不大，可能需要把“random_page_cost”的值调得小一些。如果想让优化器偏向走索引，而不走全表扫描，可以把“random_page_cost”的值调得高一些。

9.2.3 基因查询优化的参数

基因查询优化（GEQO）是一个使用探索式搜索来执行查询规划的算法，它可以降低负载查询的规划时间。GEQO 的检索是随机的，因此它生成的执行计划可能会有不确定性。

基因查询优化器相关的配置参数如表 9-3 所示。

表 9-3 基因查询优化器的相关参数

参数名称	类 型	说 明
geqo	boolean	允许或禁止基因查询优化，在生产系统中最好把此参数打开，默认是打开的。geqo_threshold 参数提供了一种是否使用基因查询优化方法的更精细的控制方法
geqo_threshold	integer	只有当涉及的 FROM 关系数量至少有这么多个时，才使用基因查询优化。对于数量小于此值的查询，也许使用判定性的穷举搜索更有效。但是对于有许多表的查询，规划器做判断要花很多时间。默认是 12。请注意，一个 FULL OUTER JOIN 只算一个 FROM 项
geqo_effort	integer	控制 GEQO 里规划时间和查询规划有效性之间的平衡。这个变量必须是一个从 1 到 10 的整数。默认值是 5。大的数值增加花在进行查询规划上面的时间，但是也很可能提高选中更有效的查询规划的几率 geqo_effort 实际上并没有直接干什么事情，只是用于计算其他那些影响 GEQO 行为变量的默认值（在下面描述）。如果愿意，可以手工设置其他参数

（续）

参数名称	类　型	说　明
geqo_pool_size	integer	控制 GEQO 使用的池大小。池大小是基因全体中的个体数量，它必须至少是 2，有用的数值通常在 100 到 1000 之间。如果把它设置为 0（默认值），那么就会基于 geqo_effort 和查询中表的数量选取一个合适的值
geqo_generations	integer	控制 GEQO 使用的子代数目。子代的意思是算法的迭代次数。它必须至少是 1，有用值的范围和池大小相同。如果设置为 0（默认值），那么将基于 geqo_pool_size 选取合适的值
geqo_selection_bias	float	控制 GEQO 使用的选择性偏好。选择性偏好是指在一个种群中的选择性压力。数值可以是 1.5 到 2.0 之间，默认值是 2.0
geqo_seed	float	控制 GEQO 使用的随机数产生器的初始值，用以选择随机路径。 这个值可以从 0（默认）到 1。修改这个值会改变连接路径搜索的设置，同时会找到最优或最差路径

当没有很多表做关联查询时，并不需要关注这些基因查询优化器的参数，因为这时基本不会走基因查询，只有当关联查询表的数目超过“ geqo_threshold”配置项时，才会走基因查询优化算法。如果不清楚基因查询的原理，不能理解以上这些参数，保持它们的默认值就可以了。

9.2.4　其他执行计划配置项

其他一些与执行计划相关的配置项如表 9-4 所示。

表 9-4　其他执行计划配置项

参数名称	类　型	说　明
default_statistics_target	integer	此参数设置表字段的默认直方图统计目标值，如果表字段的直方图统计目标值没有用 ALTER TABLE SET STATISTICS 明确设置过，则使用此参数指定的值。此值越大，ANALYZE 需要花费越多的时间，同时统计出的直方图信息越详细，这样生成的执行计划也越准确。默认值是 100，最大值是 10000
constraint_exclusion	enum	指定在执行计划中是否使用约束排除。可以取三个值：partition、on、off。默认值为 partition。约束排除就是指优化器分析 where 中的过滤条件与表上的 check 约束，当从语义上就能分析出不需要访问这张表时，执行计划直接跳过这张表。如表上的一个字段有约束 check col1>10000，当查询表“ select * from t where col1<900;”时，优化器对比约束条件，知道根本没有 col1<900 的记录，跳过对表的扫描直接返回 0 条记录 当优化器使用约束排除时，需要花更多的时间去对比约束条件和 where 中的过滤条件，在多数情况下，对无继承的表打开约束排除意义不大，所以 PostgreSQL 把此值默认设置为 partition。当对一张表做查询时，如果这张表有很多继承的子表，通常也需要扫描这些子表，设置为“ partition”，优化器就会对这些子表做约束排除分析
cursor_tuple_fraction	float	游标在选择执行计算时有两种策略：第一种是选择总体执行代价最低的，第二种是返回第一条记录时代价最低的。有时总体执行代价最低，但返回第一条记录的代价不是最低，这时返回给用户的第一条记录的时间比较长，这会让用户觉得等待较长的时间，系统才有响应，导致用户体验不太好。为了让用户体验比较好，可以选择返回第一条记录最快的执行计划，这时用户可以比较快地看到第一条记录。

（续）

参数名称	类 型	说 明
cursor_tuple_fraction	float	设置游标，在选择总体代价最低的执行计划和返回第一条记录代价最低的执行计划两者之间，比较倾向性的大小。默认值是 0.1。最大值是 1.0，此时游标会选择总体代价最低的执行计划，而不考虑多久才会输出第一个行
from_collapse_limit	integer	默认值是 8。如果查询重写生成的 FROM 后的项目数不超过这个限制数目，优化器将把子查询融合到上层查询。小的数值可缩短规划的时间，但是可能会生成差一些的查询计划。将这个值设置得与配置项 geqo_threshold 的数值相同或更大，可能触发使用 GEQO 规划器，从而产生不确定的执行计划
join_collapse_limit	integer	如果查询重写生成的 FROM 后的项目数不超过这个限制数目，优化器把显式使用 JOIN 子句（不包括 FULL JOIN）的连接也重写到 FROM 后的列表中。小的数值可缩短规划的时间，但是可能会生成差一些的查询计划值。默认值与 from_collapse_limit 一样。将这个值设置得与配置项 geqo_threshold 的数值相同或更大，可能触发使用 GEQO 规划器，从而产生不确定的执行计划

9.3 统计信息的收集

PgStat 子进程是 PostgreSQL 中专门的统计信息收集器进程。收集的统计信息主要用于查询优化时的代价估算，当然这些统计数据对数据库活动的监控及性能分析也能有很大的帮助。表和索引的行数、块数等统计信息记录在系统表 pg_class 中，其他的统计信息主要收集在系统表 pg_statistic 中。

9.3.1 统计信息收集器的配置项

统计信息收集器的配置项如表 9-5 所示。

表 9-5 统计信息收集器的配置项

参数名称	类 型	说 明
track_counts	boolean	控制是否收集表和索引上访问的统计信息。默认是打开的
track_functions	enum	是否收集函数调用次数和时间的统计信息。可以取“none”、“pl”、“all”三个值。“none”表示不收集；“pl”表示只收集过程语言函数；“all”表示收集所有的函数，包括 SQL 和 C 语言函数。默认为“none”
track_activities	boolean	是否允许跟踪每个 session 正在执行的 SQL 命令的信息和命令开始的时间。这些信息可以在视图 pg_stat_activity 中看到。此参数默认为打开的
track_activity_query_size	integer	在 pg_stat_activity 视图中的 query 字段最多显示多少字节，默认值是 1024，超过此设置的内容被截断
track_io_timing	boolean	是否允许统计 IO 调用的时间，默认为关掉。如果打开了此选项，在带“BUFFERS”选项的 EXPLAIN 命令中将显示 IO 调用的时间。这些 IO 统计信息也可以在 pg_stat_database 和 pg_stat_statements 中看到，这是 PostgreSQL9.2 之后才新增加的参数

（续）

参数名称	类　型	说　明
update_process_title	boolean	当后台服务进程正在执行命令（如一条 SQL）时，是否更新其 title 信息。在 Linux 下这个参数默认是打开的，所以在 Linux 下，默认可以使用 ps 命令查看一个后台服务进程是否正在执行命令
stats_temp_directory	string	设置存储临时统计数据的路径，可以是一个相对于数据目录的相对路径，也可以是一个绝对路径。默认值是 pg_stat_tmp

9.3.2　SQL 执行的统计信息输出

可以使用以下 4 个 boolean 类型的参数来控制是否输出 SQL 执行过程的统计信息到日志中：

- log_statement_stats
- log_parser_stats
- log_planner_stats
- log_executor_stats

参数 log_statement_stats 控制是否输出所有 SQL 语句的统计信息，其他的参数控制每个 SQL 是否输出不同执行模块中的统计信息。

9.3.3　手工收集统计信息

手工收集统计信息的命令是 analyze 命令，此命令收集表的统计信息，然后把结果保存在系统表 pg_statistic 里。优化器可以使用这些统计信息来确定最优的执行计划。

在默认的 PostgreSQL 配置中，autovacuum 守护进程是打开的，它能自动地分析表，并收集表的统计信息。当 autovacuum 关闭时，需要周期性地，或者在表的大部分内容变更后运行 ANALYZE 命令。准确的统计信息将帮助优化器生成最优的执行计划，从而改善查询的性能。一种比较常用的策略是每天在数据库比较空闲的时候运行一次 VACUUM 和 ANALYZE。

ANALYZE 命令的格式如下：

```
ANALYZE [ VERBOSE ] [ table [ ( column [, ...] ) ] ]
```

命令中的一些选项说明如下。

- VERBOSE：增加此选项将显示处理的进度，以及表的一些统计信息 。
- table：要分析的表名，如果不指定，则对整个数据库中的所有表作分析。
- column：要分析的特定字段的名字。默认是分析所有字段。

ANALYZE 命令的示例如下。

只分析表 test01 的 id2 列，命令如下：

```
osdba=# ANALYZE test01(id2);
ANALYZE
```

分析表 test01 的 id1 和 id2 两个列，命令如下：

```
osdba=# ANALYZE test01(id1,id2);
```

```
ANALYZE
```

分析表 test01 的所有列，命令如下：

```
osdba=# ANALYZE test01;
ANALYZE
```

ANALYZE 命令会在表上加一个读锁，因此它可以和表上的其他 SQL 并发地执行。ANALYZE 会收集表中每个字段的直方图和最常用数值的列表。

对于大表，ANALYZE 只读取表的部分内容做一个随机抽样，不读取表的所有内容，这样就保证了即使是在很大的表上也只需要很少时间就可以完成统计信息的收集。统计信息只是近似的结果，即使表内容实际上没有改变，运行 ANALYZE 后 EXPLAIN 显示的执行计划中的 COST 值都会有一些小变化。为了调整所收集的统计信息的准确度，可以增大随机抽样比例，这可以通过调整参数“default_statistics_target”来实现，这个参数可在 session 级别设置，比如，在分析不同的表时设置不同的值。在下面的示例中，假设表 test01 的行数较少，设置“default_statistics_target”为 500，然后进行分析，表 test02 的行数较多，设置“default_statistics_target”为 10，再分析 test02 表，命令如下：

```
osdba=# set default_statistics_target to 500;
SET
osdba=# analyze test01;
ANALYZE
osdba=# set default_statistics_target to 10;
SET
osdba=# analyze test02;
ANALYZE
```

也可以直接设置表中每个列的统计 target 值，如下：

```
osdba=# ALTER TABLE test01 ALTER COLUMN id2 SET STATISTICS 200;
ALTER TABLE
```

ANALYZE 有一个统计项是估计出现在每列的不同值的数目。但因为仅仅抽样部分行，所以这个统计项的估计值有时会很不准确，为了避免因这个错误导致差的查询计划，可以手工指定这个列有多少个唯一值，其命令是“ALTER TABLE ... ALTER COLUMN ... SET (n_distinct = ...)”。示例如下：

```
osdba=# ALTER TABLE test01 ALTER COLUMN id2 SET (n_distinct=2000);
ALTER TABLE
```

另外，如果表是有继承关系的其他子表的父表，还可以设置“n_distinct_inherited”，这样子表会继续使用这个父表的设置值，示例如下：

```
osdba=# ALTER TABLE test01 ALTER COLUMN id2 SET (n_distinct_inherited=2000);
ALTER TABLE
```