TDSQL PostgreSQL版

实践教程

腾讯云

版权所有：腾讯云计算（北京）有限责任公司

【版权声明】

©2013-2025 腾讯云版权所有

本文档（含所有文字、数据、图片等内容）完整的著作权归腾讯云计算（北京）有限责任公司单独所有，未经腾讯云事先明确书面许可，任何主体不得以任何形式复制、修改、使用、抄袭、传播本文档全部或部分内容。前述行为构成对腾讯云著作权的侵犯，腾讯云将依法采取措施追究法律责任。

【商标声明】

腾讯云

及其它腾讯云服务相关的商标均为腾讯云计算（北京）有限责任公司及其关联公司所有。本文档涉及的第三方主体的商标，依法由权利人所有。未经腾讯云及有关权利人书面许可，任何主体不得以任何方式对前述商标进行使用、复制、修改、传播、抄录等行为，否则将构成对腾讯云及有关权利人商标权的侵犯，腾讯云将依法采取措施追究法律责任。

【服务声明】

本文档意在向您介绍腾讯云全部或部分产品、服务的当时的相关概况，部分产品、服务的内容可能不时有所调整。您所购买的腾讯云产品、服务的种类、服务标准等应由您与腾讯云之间的商业合同约定，除非双方另有约定，否则，腾讯云对本文档内容不做任何明示或默示的承诺或保证。

【联系我们】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100或95716。

文档目录

实践教程
客户端工具下载及使用 pg_dump 导出数据
优化 SQL 语句
优化实例

版权所有：腾讯云计算（北京）有限责任公司

实践教程
客户端工具下载及使用 pg_dump 导出数据

最近更新时间：2025-07-29 17:19:22

驱动说明

TDSQL PostgreSQL 是腾讯自主研发的分布式数据库系统，您可以使用我们更新后的驱动来使用客户端工具。

- 数据库大版本 V2 相关客户端下载地址 。
- 数据库大版本 V5.06 相关客户端下载地址 。
- 数据库大版本 V5.21 相关客户端下载地址 。

将文件上传至服务器目录，解压后，设置环境变量，即可执行 pg_dump：

```
unzip -d ~ tbase_pgxz_v5.zip
export PATH=~/tbase_pgxz_v5/bin:$PATH;export
LD_LIBRARY_PATH=~/tbase_pgxz_v5/lib
```

pg_dump 工具说明

pg_dump 是一个用于备份 PostgreSQL 数据库的工具。它甚至可以在数据库正在使用的时候进行完整一致的备份。pg_dump 并不阻塞其它用户对数据库的访问（读或者写）。建议以 schema 为单位进行导入导出。下面列举几种 pg_dump 的常用使用场景供参考。

- 导出 postgres 库的所有模式下的所有数据。

```
pg_dump postgres -h 9.101.17.6 -p 5432 -U dbadmin -f /data/xxx.sql
```

- 只导出 postgres 库的所有模式下的表结构。

```
pg_dump postgres -h 9.101.17.6 -p 5432 -U dbadmin -s -f /data/xxx.sql
```

- 导出 postgres 库的 scname 模式下的数据。

```
pg_dump postgres -h 9.101.17.6 -p 5432 -U dbadmin -n scname -f
/data/xxx.sql
```

- 导出 postgres 库的 scname1 和 scname2 模式下的数据。

版权所有：腾讯云计算（北京）有限责任公司

```
pg_dump postgres -h 9.101.17.6 -p 5432 -U dbadmin -n scname1 -n
scname2 -f /data/xxx.sql
```

- 只导出 postgres 库的 scname 模式下的表结构。

```
  pg_dump postgres -h 9.101.17.6 -p 5432 -U dbadmin -s -n scname -f
/data/xxx.sql
```

更多的 pg_dump 使用请参考 工具文档 。

优化 SQL 语句

最近更新时间：2025-03-07 15:00:42

查看是否为分布键查询

```
postgres=# explain select * from tbase_1 where f1=1;
                                    QUERY PLAN
\-----------------------------------------------------------------------
---------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Gather  (cost=1000.00..7827.20 rows=1 width=14)
         Workers Planned: 2
         ->  Parallel Seq Scan on tbase_1  (cost=0.00..6827.10 rows=1
width=14)
               Filter: (f1 = 1)
(6 rows)

postgres=# explain select * from tbase_1 where f2=1;
                                    QUERY PLAN
\-----------------------------------------------------------------------
---------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001
   ->  Gather  (cost=1000.00..7827.20 rows=1 width=14)
         Workers Planned: 2
         ->  Parallel Seq Scan on tbase_1  (cost=0.00..6827.10 rows=1
width=14)
               Filter: (f2 = 1)
(6 rows)
```

如上，第一个查询为非分布键查询，需要发往所有节点，这样最慢的节点决定了整个业务的速度，需要保持所有节点的响应性能一致，如第二个查询所示，业务设计查询时尽可能带上分布键。

查看是否使用索引

```
postgres=# create index tbase_2_f2_idx on tbase_2(f2);
CREATE INDEX
postgres=# explain select * from tbase_2 where f2=1;
                                    QUERY PLAN
```

```
\-----------------------------------------------------------------------
--------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Index Scan using tbase_2_f2_idx on tbase_2  (cost=0.42..4.44
rows=1 width=14)
         Index Cond: (f2 = 1)
(4 rows)

postgres=# explain select * from tbase_2 where f3='1';
                                    QUERY PLAN
\-----------------------------------------------------------------------
---------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Gather  (cost=1000.00..7827.20 rows=1 width=14)
         Workers Planned: 2
         ->  Parallel Seq Scan on tbase_2  (cost=0.00..6827.10 rows=1
width=14)
               Filter: (f3 = '1'::text)
(6 rows)

postgres=#
```

第一个查询使用了索引，第二个没有使用索引，通常情况下，使用索引可以加速查询速度，但索引也会增加更新的开销。

查看是否为分布 key join

```
postgres=# explain  select tbase_1.* from tbase_1,tbase_2 where
tbase_1.f1=tbase_2.f1 ;
                                         QUERY PLAN
\-----------------------------------------------------------------------
-------------------------
 Remote Subquery Scan on all (dn001,dn002)  (cost=29.80..186.32
rows=3872 width=40)
   ->  Hash Join  (cost=29.80..186.32 rows=3872 width=40)
         Hash Cond: (tbase_1.f1 = tbase_2.f1)
         ->  Remote Subquery Scan on all (dn001,dn002)
(cost=100.00..158.40 rows=880 width=40)
               Distribute results by S: f1
               ->  Seq Scan on tbase_1  (cost=0.00..18.80 rows=880
width=40)
```

```
          ->  Hash  (cost=18.80..18.80 rows=880 width=4)
                ->  Seq Scan on tbase_2  (cost=0.00..18.80 rows=880
width=4)
(8 rows)

postgres=# explain  select tbase_1.* from tbase_1,tbase_2 where
tbase_1.f2=tbase_2.f1 ;
                                    QUERY PLAN
\----------------------------------------------------------------------
----------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Hash Join  (cost=18904.69..46257.08 rows=500564 width=14)
         Hash Cond: (tbase_1.f2 = tbase_2.f1)
         ->  Seq Scan on tbase_1  (cost=0.00..9225.64 rows=500564
width=14)
         ->  Hash  (cost=9225.64..9225.64 rows=500564 width=4)
               ->  Seq Scan on tbase_2  (cost=0.00..9225.64 rows=500564
width=4)
(7 rows)
```

第一个查询需要数据重分布，而第二个不需要，分布键 join 查询性能会更高。

查看 join 发生的节点

```
postgres=#  explain  select tbase_1.* from tbase_1,tbase_2 where
tbase_1.f1=tbase_2.f1 ;
                                          QUERY PLAN
\----------------------------------------------------------------------
------------------------
 Hash Join  (cost=29.80..186.32 rows=3872 width=40)
   Hash Cond: (tbase_1.f1 = tbase_2.f1)
   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.00..158.40
rows=880 width=40)
         ->  Seq Scan on tbase_1  (cost=0.00..18.80 rows=880 width=40)
   ->  Hash  (cost=126.72..126.72 rows=880 width=4)
         ->  Remote Subquery Scan on all (dn001,dn002)
(cost=100.00..126.72 rows=880 width=4)
               ->  Seq Scan on tbase_2  (cost=0.00..18.80 rows=880
width=4)
(7 rows)

postgres=# set prefer_olap to on;
```

```
SET
postgres=# explain  select tbase_1.* from tbase_1,tbase_2 where
tbase_1.f1=tbase_2.f1 ;
                                          QUERY PLAN
\-----------------------------------------------------------------------
-------------------------
 Remote Subquery Scan on all (dn001,dn002)  (cost=29.80..186.32
rows=3872 width=40)
   ->  Hash Join  (cost=29.80..186.32 rows=3872 width=40)
         Hash Cond: (tbase_1.f1 = tbase_2.f1)
         ->  Remote Subquery Scan on all (dn001,dn002)
(cost=100.00..158.40 rows=880 width=40)
               Distribute results by S: f1
               ->  Seq Scan on tbase_1  (cost=0.00..18.80 rows=880
width=40)
         ->  Hash  (cost=18.80..18.80 rows=880 width=4)
               ->  Seq Scan on tbase_2  (cost=0.00..18.80 rows=880
width=4)
(8 rows)
```

第一个 join 在 cn 节点执行，第二个在 dn 上重分布后再 join。业务设计上，一般 OLTP 类业务在 cn 上进行少数据量 join ，性能会更好。

查看并行的 worker 数

```
postgres=# explain select count(1) from tbase_1;
                                     QUERY PLAN
\-----------------------------------------------------------------------
----------------
 Finalize Aggregate  (cost=118.81..118.83 rows=1 width=8)
   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=118.80..118.81
rows=1 width=0)
         ->  Partial Aggregate  (cost=18.80..18.81 rows=1 width=8)
               ->  Seq Scan on tbase_1  (cost=0.00..18.80 rows=880
width=0)
(4 rows)

postgres=# analyze tbase_1;
ANALYZE
postgres=# explain select count(1) from tbase_1;
                                          QUERY PLAN
\-----------------------------------------------------------------------
-----------------------------
```

```
 Parallel Finalize Aggregate  (cost=14728.45..14728.46 rows=1 width=8)
   ->  Parallel Remote Subquery Scan on all (dn001,dn002)
(cost=14728.33..14728.45 rows=1 width=0)
         ->  Gather  (cost=14628.33..14628.44 rows=1 width=8)
               Workers Planned: 2
               ->  Partial Aggregate  (cost=13628.33..13628.34 rows=1
width=8)
                     ->  Parallel Seq Scan on tbase_1
(cost=0.00..12586.67 rows=416667 width=0)
(6 rows)
```

上面第一个查询没走并行，第二个查询 analyze 后走并行才是正确的，建议大数据量更新再执行 analyze。

执行计划

查看执行计划

EXPLAIN 语法如下：

```
EXPLAIN [ ( 参数 [, ...] ) ]  SQL_clause
or
EXPLAIN [ ANALYZE ] [ VERBOSE ]  SQL_clause
```

参数说明：

- ANALYZE: 执行命令并且显示实际的运行时间和其他统计信息。这个参数默认被设置为 FALSE。
- VERBOSE: 显示关于执行计划的附加信息。
  具体包括计划树中每个结点的输出列列表、模式限定的表和函数名称、在表达式中使用范围表别名标记变量，并输出显示统计信息的每个触发器的名称。这个参数默认被设置为 FALSE。
- COSTS: 每一个计划结点的估计启动和总代价，以及估计的行数和每行的宽度。这个参数默认被设置为 TRUE。
- SETTINGS: 有关配置参数的信息。具体包括影响查询计划的选项，其值与内置默认值不同。此参数默认为 FALSE。
- BUFFERS: 缓冲区使用的信息。具体包括共享块命中、读取、标记为脏和写入的次数、本地块命中、读取、标记为脏和写入的次数、以及临时块读取和写入的次数。一次命中表示避免了一次读取，因为需要的块已经在缓存中找到了。共享块包含着来自于常规表和索引的数据，本地块包含着来自于临时表和索引的数据，而临时块包含着在排序、哈希等情况中使用的短期工作数据。脏块的数量表示被这个查询改变的之前未被修改块的数量，而写入块的数量表示这个后台在查询处理期间从缓存中替换出去的脏块的数量。为一个较高层结点显示的块数包括它的所有子结点所用到的块数。在文本格式中，只会打印非零值。只有当 ANALYZE 也被启用时，这个参数才能使用。它的默认被设置为 ALSE。
- WAL: 有关 WAL 记录生成的信息。 具体包括记录数、整页图像数（fpi）和生成的 WAL 字节数量。 在文本格式中，仅打印非零值。 此参数只能在同时启用ANALYZE 时使用。 它默认为 FALSE。

- TIMING: 在输出中包括实际启动时间以及在每个结点中花掉的时间。反复读取系统时钟的负荷在某些系统上会显著地拖慢查询，因此在只需要实际的行计数而不是实际时间时，把这个参数设置为 FALSE 可能会有用。即便用这个选项关闭结点层的计时，整个语句的运行时间也总是会被度量。只有当 ANALYZE 也被启用时，这个参数才能使用。它的默认被设置为 TRUE。
- SUMMARY: 在查询计划之后包含摘要信息（例如，总计的时间信息）。当使用 ANALYZE 时默认包含摘要信息。不使用 ANALYZE 时可以使用此选项仅启用摘要信息。 使用 EXPLAIN EXECUTE 中的计划时间包括从缓存中获取计划所需的时间 以及重新计划所需的时间（如有必要）。
- FORMAT: 指定输出格式，可以是 TEXT、XML、JSON 或者 YAML。非文本输出包含和文本输出格式相同的信息。这个参数默认被设置为 TEXT。

解读执行计划

使用 EXPLAIN 命令可以查看 TDSQL PG 优化器为每个查询生成的具体执行计划。EXPLAIN 的输出是一个节点树。 其中每一行对应一个数据库执行算子， 显示算子的节点类型和优化器为执行这个节点预估的开销。
通过以下示例来解读执行计划：

```
create table t3(f1 int, f2 int) distribute by hash(f1);
create table t4(f1 int, f2 int) distrinute by hash(f1);
```

> **说明：**
> distribute by hash 表示对 t3，t4数据按 f1 哈希值进行分布到不同节点 dn 上。

查看 SQL 的执行计划 `explain select * from t3, t4 where t3.f1=t4.f2;` 可以得到如下输出结果：

```
                                QUERY PLAN
-----------------------------------------------------------------------
--------
Remote Subquery Scan on all (datanode_1,datanode_2) (cost=120.19..222.56
rows=4556  width=16)
  -> Hash Join (cost=120.19..222.56 rows=4556 width=16)
      Hash Cond:(t4.f2 = t3.f1)
      -> Remote Subquery Scan on all (datanode_1,datanode_2)
(cost=100.00..120.53  rows=675 width=8)
          Distribute results by H: f2
          -> Seq Scan on t4 (cost=0.00..11.75 rows=675 width=8)
      -> Hash (cost=11.75..11.75 rows=675 width=8)
          -> Seq Scan on t3 (cost=0.00..11.75 rows=675 width=8)
(8rows)
```

执行计划树里的每个节点代指一个执行算子。每个节点的 EXPLAIN 输出（如下所示）格式为算子名称（算子开销预估）。

```
-> Hash Join(cost=120.19..222.56 rows=4556 width=16)
    Hash Cond:(t4.f2 = t3.f1)
```

括号中的数字从左到右依次是：

- 启动代价: 这是该算子在读取第一条元组前的开销预估。比方说索引扫描算子的启动代价就是读取目标表的索引页，读取到第一个元组的开销。以上示例中的哈希关联(Hash Join)的启动代价是120.19。
- 算子总代价：这是该算子从执行到结束的总代价。以上示例中的哈希关联(Hash Join)的总代价是222.56。
- 算子输出的总行数。 以上示例中的哈希关联(Hash Join)的预估输出行数是4556。
  算子输出行的行宽（字节数）。 以上示例中的哈希关联(Hash Join)的预估行宽是16字节。

有些算子在 EXPLAIN 的输出里除了上述的基本通用信息外， 还有算子的一些特定信息。 比方说上面的哈希关联算子的 EXAPLIN 输出还打印了关联条件。
在 SQL 执行计划输出结果中，通常最底层节点是表扫描节点，扫描节点返回表中的原数据行。 不同的表有不同的扫描节点类型，如顺序扫描，索引扫描和位图索引扫描。最底层的扫描节点也可能是也有非表列源，如 VALUES 子句并设置 FROM 返回，它们有自己的扫描类型。
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。这些操作通常都有多种方法，因此在这些位置也有可能出现不同的执行节点类型。
分布式计划中通常会有个特殊的算子（“Remote Subquery Scan”）。这个算子实现了分布式架构的核心数据 shuffle 的功能。 这个算子有几种不同的形态， 分别对应分布式架构下不同的数据 shuffle 功能：
（1）数据收集形态 – 作用是 CN 从 DN 收集数据。
（2）数据重分布形态 – 作用是 CN 根据选定的列按照特定的规则把数据重分布到所有的 DN。
数据重分布的规则有：

- Redistribute by Hash: 把指定的列按照 hash redistribute。 这种情况下， 执行计划的输出里会在“Remote Subquery Scan”的那一行的下面一行打印 – “Distribute results by H：列名”。 如下所示。 其中"Remote Subquery Scan on all"后面括号里显示的是该算子在哪些数据节点上执行。

  ```
  -> Remote Subquery Scan on all (datanode_1,datanode_2)
  (cost=100.00..120.53  rows=675 width=8)
       Distribute results by H: f2
       -> Seq Scan on t4 (cost=0.00..11.75 rows=675 width=8)
  ```

- Redistribute by Shard: 把指定的列按照 shard redistribute。 这种情况下， 执行计划的输出里会在“Remote Subquery Scan”的那一行的下面一行打印 – “Distribute results by S：列名”
- Broadcast – 把指定的表广播到所有的数据节点上。 这种情况下， 执行计划输出里在“Remote Subquery Scan”的那一行下面没有额外的重分布信息。

在分布式架构中， 当 SQL 可以完全下推到各个 DN 上计算，中间计算结果不需要重新分布到其他 DN 上的时候，执行计划中会有个分布式快速执行算子 – “Remote Fast Query Execution”， 如下示例。在下面这个例子

里，两表关联的列都是分布健。这种情况下，各个 DN 上各自独立执行 SQL，CN 只需要把 DN 返回的结果直接返回即可，无需额外的计算。

```
explain select * from t3, t4 where t3.f1=t4.f1;
```

```
 QUERY PLAN
-----------------------------------------------------------------
Remote Fast Query Execution (cost=0.00..0.00 rows=0 width=0)
 Node/s:datanode_1, datanode_2
 ->Merge Join (cost=187.38..330.81 rows=9112 width=16)
   Merge Cond:(t3.f1=t4.f1)
   -> Sort (cost=93.69..97.07 rows=1350 width=8)
      Sort Key: t3.f1
      -> Seq Scan on t3 (cost=0.00..23.50 rows=1350 width=8)
   -> Sort (cost=93.69..97.07 rows=1350 width=8)
      Sort Key: t4.f1
      -> Seq Scan on t4 (cost=0.00..23.50 rows=1350 width=8)
(10rows)
```

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，指的是一个 SQL执行的代价是多少，而不是具体的时间。
评估的行数不是执行和扫描查询节点的数量，是节点返回的数量。一般会少于扫描数量，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。
代价预估计算是依赖于规划器的设置参数。需要知道的是：上级节点的消耗代价包括其子节点的消耗代价。下面是默认情况下，对数据操作的消耗评估基础：

```
seq_page_cost=1.0                    #measured on an arbitrary scale
random_page_cost=4.0                 #same scale as above，如果是SSD这个代价
因子可以酌情调整
cpu_tuple_cost=0.01                  #same scale as above
cpu_index_tuple_cost=0.005            #same scale as above
```

举例说明如下：

```
tdsql=# create table test(id int, info text, crt_time timestamp);
CREATE TABLE
tdsql=# insert into test select generate_series(1, 10000),
md5(cast(random() as text)), now();
INSERT 0 10000
tdsql=# analyze test;
```

```
ANALYZE
tdsql=# explain select * from test;
                           QUERY PLAN
-----------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Seq Scan on test  (cost=0.00..98.39 rows=5039 width=19)
(3 rows)

---test表做全表扫描的消耗评估情况
--test表的数据分布情况:
tdsql=# select xc_node_id, count(*) from test group by 1;
 xc_node_id | count
------------+-------
  -17499968 |  4961
 2142761564 |  5039
(2 rows)

--test表的总数据块级行数统计:
tdsql=# select relpages, reltuples from pg_class where relname = 'test';
 relpages | reltuples
----------+-----------
       95 |     10000
(1 row)

--test表在DN001和DN002上的数据块和行数统计:
tdsql=# execute direct on (dn001)  'select relpages, reltuples from
pg_class where relname = ''test''';
relpages | reltuples
----------+-----------
       48 |      5039
(1 row)

tdsql=# execute direct on (dn002)  'select relpages, reltuples from
pg_class where relname = ''test''';
 relpages | reltuples
----------+-----------
       47 |      4961
(1 row)
tdsql=# explain select * from test;
                           QUERY PLAN
-----------------------------------------------------------------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
```

```
    Node/s: dn001, dn002
    ->  Seq Scan on test  (cost=0.00..98.39 rows=5039 width=19)
 (3 rows)
```

表 test 有10000条数据分布在95个块，其中 DN001 节点上有48块，评估消耗是（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：(48 * 1.0) + (5039 * 0.01) = 98.39，和执行计划吻合。

执行计划一般都是以下几种的组合：

（1）表扫描：分为全表扫描、索引扫描、索引扫描还分为 bitmap 扫描。

（2）表连接：nested loop、hash join、merge join。

（3）聚合运算：count、min、max 等聚合操作。

（4）排序、分组：order by、group by。

因此我们需要关注这几类节点的执行消耗统计情况，一般分析原则如下：

- 从里到外：
  外层的消耗是在里层消耗的基础上进行评估的，越在里层的执行计划树节点，越早执行，外层的节点都依赖里层的结果。因此我们需要从里层入手去消除cost 较高的节点。里层节点很容易被循环调用，里层的节点一般都是被驱动对象，如果表连接方式不正确，很容易产生循环调用，例如应该走 hash loop 的走了 nested loop。
- 从大到小：从 cost 较大的节点入手。
- 消除重分布：当两个表进行连接时，连接键不都是分片键时，会走重分布。在当前的 V5 版本中，DN 重分布会需要将本节点数据发送到其他所有 DN 节点，代价非常高。重分布关键词：Distribute results by。
- 避免分布式事务：如果 SQL 不带分片键，那么就会被发送到所有 DN 去执行。select 语句则就涉及 CN 收取多个 DN 的结果，insert、update、delete 则会涉及分布式事务，代价比带分片键高很多。

优化实例

最近更新时间：2023-11-13 17:49:44

count(distinct xx) 优化

```
postgres=# CREATE TABLE t1(f1 serial not null unique,f2 text,f3 text,f4
text,f5 text,f6 text,f7 text,f8 text,f9 text,f10 text,f11 text,f12 text)
distribute by shard(f1);
NOTICE:  Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE
Time: 89.938 ms

postgres=# insert into t1 select
t,md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t
::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text)
from generate_series(1,1000000) as t;
INSERT 0 1000000
Time: 14849.045 ms (00:14.849)

postgres=# analyze t1;
ANALYZE
Time: 1340.387 ms (00:01.340)

postgres=# explain (verbose)  select count(distinct f2) from t1;
                                                                QUERY PLAN
\-----------------------------------------------------------------------
-------------------------------------------------------------
 Aggregate  (cost=103320.00..103320.01 rows=1 width=8)
   Output: count(DISTINCT f2)
   ->  Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=100.00..100820.00 rows=1000000 width=33)
         Output: f2
         ->  Seq Scan on public.t1  (cost=0.00..62720.00 rows=1000000
width=33)
               Output: f2
(6 rows)

Time: 0.748 ms
postgres=# select count(distinct f2) from t1;
```

```
  count
\---------
 1000000
(1 row)

Time: 6274.684 ms (00:06.275)

postgres=# select count(distinct f2) from t1 where f1 <10;
 count
\-------
     9
(1 row)

Time: 19.261 ms
```

如上 count(distinct f2) 发生在 cn 节点，对于 TP 类业务，需要操作的数据量少的情况下，性能开销没有问题，且比下推执行的性能开销还要小。
但对于一次要操作的数据量比较大的 AP 类业务，网络传输就会成为瓶颈，如下是改写后的执行计划：

```
postgres=# explain (verbose)  select count(1) from (select f2 from t1
group by f2) as t ;

QUERY PLAN
\-----------------------------------------------------------------------
-----------------------------------------------------------------------
------------------
 Finalize Aggregate  (cost=355600.70..355600.71 rows=1 width=8)
   Output: count(1)
   ->  Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=355600.69..355600.70 rows=1 width=0)
         Output: PARTIAL count(1)
         ->  Partial Aggregate  (cost=355500.69..355500.70 rows=1
width=8)
               Output: PARTIAL count(1)
               ->  Group  (cost=340500.69..345500.69 rows=1000000
width=33)
                     Output: t1.f2
                     Group Key: t1.f2
                     ->  Sort  (cost=340500.69..343000.69 rows=1000000
width=0)
                           Output: t1.f2
                           Sort Key: t1.f2
```

```
                              ->  Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=216192.84..226192.84 rows=1000000 width=0)
                                    Output: t1.f2
                                    Distribute results by S: f2
                                    ->  Group  (cost=216092.84..221092.84
rows=1000000 width=33)
                                          Output: t1.f2
                                          Group Key: t1.f2
                                          ->  Sort
(cost=216092.84..218592.84 rows=1000000 width=33)
                                                Output: t1.f2
                                                Sort Key: t1.f2
                                                ->  Seq Scan on public.t1
(cost=0.00..62720.00 rows=1000000 width=33)
                                                      Output: t1.f2
(23 rows)
```

改写后，并行推到 dn 执行，此时查看执行的效果，可以看到对于大量数据计算的 AP 类业务，性能提高了5倍。

```
postgres=# select count(1) from (select f2 from t1 group by f2) as t ;
  count
\---------
 1000000
(1 row)
Time: 1328.431 ms (00:01.328)
postgres=# select count(1) from (select f2 from t1 where f1<10 group by
f2) as t ;
 count
\-------
     9
(1 row)

Time: 24.991 ms
postgres=#
```

增大 work_mem 减少 io 访问

增大 work_mem 后，性能提高了40倍，因为 work_mem 足够放下 filter 的数据，不需要再做 Materialize 物化，filter 由原来的 subplan 变成了 hash subplan，直接在内存 hash 表中 filter，性能提升。

> **注意：**

work_mem 默认不宜过大，建议在某个具体的查询语句中再根据需要进行调整即可。

```
postgres=# CREATE TABLE t1(f1 serial not null unique,f2 text,f3 text,f4
text,f5 text,f6 text,f7 text,f8 text,f9 text,f10 text,f11 text,f12 text)
distribute by shard(f1);
NOTICE:  Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE
Time: 70.545 ms

postgres=# CREATE TABLE t2(f1 serial not null unique,f2 text,f3 text,f4
text,f5 text,f6 text,f7 text,f8 text,f9 text,f10 text,f11 text,f12 text)
distribute by shard(f1);
NOTICE:  Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE
Time: 61.913 ms

postgres=# insert into t1 select
t,md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t
::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text)
from generate_series(1,1000) as t;
INSERT 0 1000
Time: 48.866 ms

postgres=# insert into t2 select
t,md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t
::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text)
from generate_series(1,50000) as t;
INSERT 0 50000
Time: 792.858 ms

postgres=# analyze t1;
ANALYZE
Time: 175.946 ms

postgres=# analyze t2;
ANALYZE
Time: 318.802 ms
postgres=#
```

```
postgres=# explain  select * from t1 where f2 not in (select f2 from
t2);
                                                                  QUERY
PLAN
\-----------------------------------------------------------------------
-----------------------------------------------------------------------
 Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=0.00..2076712.50 rows=500 width=367)
   ->  Seq Scan on t1  (cost=0.00..2076712.50 rows=500 width=367)
         Filter: (NOT (SubPlan 1))
         SubPlan 1
           ->  Materialize  (cost=0.00..4028.00 rows=50000 width=33)
                 ->  Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cost=0.00..3240.00
rows=50000 width=33)
                       ->  Seq Scan on t2  (cost=0.00..3240.00
rows=50000 width=33)
(7 rows)

Time: 0.916 ms
postgres=# select * from t1 where f2 not in (select f2 from t2);
 f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 | f9 | f10 | f11 | f12
----+----+----+----+----+----+----+----+----+-----+-----+-----
(0 rows)

Time: 4226.825 ms (00:04.227)

postgres=# set work_mem to '8MB';
SET
Time: 0.289 ms
postgres=# explain  select * from t1 where f2 not in (select f2 from
t2);
                                                                    QUERY
PLAN
\-----------------------------------------------------------------------
-----------------------------------------------------------------
 Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=3365.00..3577.50 rows=500 width=367)
   ->  Seq Scan on t1  (cost=3365.00..3577.50 rows=500 width=367)
         Filter: (NOT (hashed SubPlan 1))
         SubPlan 1
```

```
            ->  Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)  (cost=0.00..3240.00
rows=50000 width=33)
                  ->  Seq Scan on t2  (cost=0.00..3240.00 rows=50000
width=33)
(6 rows)

Time: 0.890 ms
postgres=# select * from t1 where f2 not in (select f2 from t2);
 f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 | f9 | f10 | f11 | f12
----+----+----+----+----+----+----+----+----+-----+-----+-----
(0 rows)

Time: 105.249 ms
postgres=#
```

not in 改写为 anti join

上文通过增大计算内存提高性能，但内存不可能无限扩大，如下通过改写语句来提高查询的性能。

```
postgres=#  explain select * from t1 left outer join t2 on t1.f2 = t2.f2
where t2.f2 is null;
                                                                  QUERY
PLAN
\-----------------------------------------------------------------------
-----------------------------------------------------------------------
--
 Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=6405.00..9260.75 rows=1 width=734)
   ->  Hash Anti Join  (cost=6405.00..9260.75 rows=1 width=734)
         Hash Cond: (t1.f2 = t2.f2)
         ->  Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=100.00..682.00 rows=1000 width=367)
               Distribute results by S: f2
               ->  Seq Scan on t1  (cost=0.00..210.00 rows=1000
width=367)
         ->  Hash  (cost=21940.00..21940.00 rows=50000 width=367)
               ->  Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=100.00..21940.00 rows=50000 width=367)
                     Distribute results by S: f2
```

```
                      ->  Seq Scan on t2  (cost=0.00..3240.00 rows=50000
width=367)
(10 rows)

Time: 1.047 ms
postgres=# select * from t1 left outer join t2 on t1.f2 = t2.f2 where
t2.f2 is null;
 f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 | f9 | f10 | f11 | f12 | f1 | f2
| f3 | f4 | f5 | f6 | f7 | f8 | f9 | f10 | f11 | f12
----+----+----+----+----+----+----+----+----+-----+-----+-----+----+----
+----+----+----+----+----+----+----+-----+-----+-----
(0 rows)

Time: 107.233 ms
postgres=#
```

也可以修改 not exists：

```
postgres=# explain select * from t1 where not exists( select 1 from t2
where t1.f2=t2.f2);
                                                                  QUERY
PLAN
\-----------------------------------------------------------------------
-----------------------------------------------------------------------
 Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=3865.00..4078.75 rows=1 width=367)
   ->  Hash Anti Join  (cost=3865.00..4078.75 rows=1 width=367)
         Hash Cond: (t1.f2 = t2.f2)
         ->  Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=100.00..682.00 rows=1000 width=367)
               Distribute results by S: f2
               ->  Seq Scan on t1  (cost=0.00..210.00 rows=1000
width=367)
         ->  Hash  (cost=5240.00..5240.00 rows=50000 width=33)
               ->  Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=100.00..5240.00 rows=50000 width=33)
                     Distribute results by S: f2
                     ->  Seq Scan on t2  (cost=0.00..3240.00 rows=50000
width=33)
```

```
(10 rows)

Time: 0.974 ms
postgres=# select * from t1 where not exists( select 1 from t2 where
t1.f2=t2.f2);
 f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 | f9 | f10 | f11 | f12
----+----+----+----+----+----+----+----+----+-----+-----+-----
(0 rows)

Time: 42.944 ms
postgres=#
```

分布 key join+limit 优化

数据准备：

```
postgres=# CREATE TABLE t1(f1 serial not null unique,f2 text,f3 text,f4
text,f5 text,f6 text,f7 text,f8 text,f9 text,f10 text,f11 text,f12 text)
distribute by shard(f1);
NOTICE:  Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE
postgres=# CREATE TABLE t2(f1 serial not null unique,f2 text,f3 text,f4
text,f5 text,f6 text,f7 text,f8 text,f9 text,f10 text,f11 text,f12 text)
distribute by shard(f1);
NOTICE:  Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE
postgres=# insert into t1 select
t,md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t
::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text)
from generate_series(1,1000000) as t;
INSERT 0 1000000
postgres=# insert into t2 select
t,md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t
::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text)
from generate_series(1,1000000) as t;
INSERT 0 1000000
postgres=# analyze t1;
ANALYZE
postgres=# analyze t2;
ANALYZE
postgres=#
```

```
postgres=# \timing
Timing is on.
postgres=# explain  select t1.* from t1,t2 where t1.f1=t2.f1 limit 10;
                                                                  QUERY
PLAN
\-----------------------------------------------------------------------
-----------------------------------------------------------------------
 Limit  (cost=0.25..1.65 rows=10 width=367)
   ->  Merge Join  (cost=0.25..140446.26 rows=1000000 width=367)
         Merge Cond: (t1.f1 = t2.f1)
         ->  Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=100.12..434823.13 rows=1000000 width=367)
               ->  Index Scan using t1_f1_key on t1
(cost=0.12..62723.13 rows=1000000 width=367)
         ->  Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=100.12..71823.13 rows=1000000 width=4)
               ->  Index Only Scan using t2_f1_key on t2
(cost=0.12..62723.13 rows=1000000 width=4)
(7 rows)

Time: 1.372 ms
postgres=# explain analyze  select t1.* from t1,t2 where t1.f1=t2.f1
limit 10;

QUERY PLAN
\-----------------------------------------------------------------------
-----------------------------------------------------------------------
-------------------------------------------
 Limit  (cost=0.25..1.65 rows=10 width=367) (actual
time=2675.437..2948.199 rows=10 loops=1)
   ->  Merge Join  (cost=0.25..140446.26 rows=1000000 width=367) (actual
time=2675.431..2675.508 rows=10 loops=1)
         Merge Cond: (t1.f1 = t2.f1)
         ->  Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=100.12..434823.13 rows=1000000 width=367) (actual
time=1.661..1.704 rows=10 loops=1)
         ->  Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
```

```
(cost=100.12..71823.13 rows=1000000 width=4) (actual
time=2673.761..2673.783 rows=10 loops=1)
 Planning time: 0.358 ms
 Execution time: 2973.948 ms
(7 rows)

Time: 2976.008 ms (00:02.976)
postgres=#
```

以上执行计划是在 cn 上执行，merge join 需要把要 join 的数据拉回 cn 再排序，然后再 join，这里主切的开销在于网络，优化方法是让语句推下去计算，如下所示，两者相差150倍的性能，一般情况下，如果需要拉取大量的数据回 cn 计算，则下推执行的效率会更好。

```
postgres=# set prefer_olap to on;
SET
Time: 0.291 ms
postgres=#  explain  select t1.* from t1,t2 where t1.f1=t2.f1 limit 10;
                                                          QUERY PLAN
\-----------------------------------------------------------------------
-------------------------------------------------------
 Limit  (cost=100.25..101.70 rows=10 width=367)
   ->  Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=100.25..101.70 rows=10 width=367)
         ->  Limit  (cost=0.25..1.65 rows=10 width=367)
               ->  Merge Join  (cost=0.25..140446.26 rows=1000000
width=367)
                     Merge Cond: (t1.f1 = t2.f1)
                     ->  Index Scan using t1_f1_key on t1
(cost=0.12..62723.13 rows=1000000 width=367)
                     ->  Index Only Scan using t2_f1_key on t2
(cost=0.12..62723.13 rows=1000000 width=4)
(7 rows)

Time: 1.061 ms
postgres=# explain analyze  select t1.* from t1,t2 where t1.f1=t2.f1
limit 10;

QUERY PLAN
\-----------------------------------------------------------------------
-----------------------------------------------------------------------
---------------------------
```

```
 Limit  (cost=100.25..101.70 rows=10 width=367) (actual
time=1.527..3.899 rows=10 loops=1)
   ->  Remote Subquery Scan on all
(dn01,dn02,dn03,dn04,dn05,dn06,dn07,dn08,dn09,dn10)
(cost=100.25..101.70 rows=10 width=367) (actual time=1.525..1.529
rows=10 loops=1)
 Planning time: 0.360 ms
 Execution time: 18.193 ms
(4 rows)

Time: 19.921 ms
```

非分布 key join 使用 hash join 性能一般更好

为提高 TP 类业务查询的性能，经常需要对一些字段建立索引，使用有索引字段 join 时，系统往往也会使用 Merge Cond 和 nestloop。

```
mydb=# CREATE TABLE t1(f1 serial not null,f2 text,f3 text,f4 text,f5
text,f6 text,f7 text,f8 text,f9 text,f10 text,f11 text,f12 text)
distribute by shard(f1);
NOTICE:  Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE
Time: 481.042 ms

mydb=# create index t1_f1_idx on t1(f2);
CREATE INDEX
Time: 85.521 ms

mydb=# CREATE TABLE t2(f1 serial not null,f2 text,f3 text,f4 text,f5
text,f6 text,f7 text,f8 text,f9 text,f10 text,f11 text,f12 text)
distribute by shard(f1);
NOTICE:  Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE
Time: 75.973 ms

mydb=# create index t2_f1_idx on t2(f2);
CREATE INDEX
Time: 29.890 ms

mydb=# insert into t1 select
t,md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t
```

```
::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text)
from generate_series(1,1000000) as t;
INSERT 0 1000000
Time: 16450.623 ms (00:16.451)

mydb=# insert into t2 select
t,md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t
::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text),md5(t::text)
from generate_series(1,1000000) as t;
INSERT 0 1000000
Time: 17218.738 ms (00:17.219)

mydb=# analyze t1;
ANALYZE

Time: 2219.341 ms (00:02.219)
mydb=# analyze t2;
ANALYZE

Time: 1649.506 ms (00:01.650)
mydb=#

--merge join

mydb=# explain select t1.* from t1,t2 where t1.f2=t2.f2 limit 10;
                                                        QUERY PLAN
\-----------------------------------------------------------------------
-------------------------------------------------
 Limit  (cost=100.25..102.78 rows=10 width=367)
   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.25..102.78
rows=10 width=367)
         ->  Limit  (cost=0.25..2.73 rows=10 width=367)
               ->  Merge Join  (cost=0.25..248056.80 rows=1000000
width=367)
                     Merge Cond: (t1.f2 = t2.f2)
                     ->  Remote Subquery Scan on all (dn001,dn002)
(cost=100.12..487380.85 rows=1000000 width=367)
                           Distribute results by S: f2
                           ->  Index Scan using t1_f1_idx on t1
(cost=0.12..115280.85 rows=1000000 width=367)
                     ->  Materialize  (cost=100.12..155875.95
rows=1000000 width=33)
```

```
                              ->  Remote Subquery Scan on all (dn001,dn002)
(cost=100.12..153375.95 rows=1000000 width=33)
                                    Distribute results by S: f2
                                    ->  Index Only Scan using t2_f1_idx on
t2  (cost=0.12..115275.95 rows=1000000 width=33)
(12 rows)

Time: 4.183 ms
mydb=# explain analyze select t1.* from t1,t2 where t1.f2=t2.f2 limit
10;
                                                                  QUERY
PLAN
\-----------------------------------------------------------------------
-------------------------------------------------------------------
 Limit  (cost=100.25..102.78 rows=10 width=367) (actual
time=6555.346..6556.296 rows=10 loops=1)
   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.25..102.78
rows=10 width=367) (actual time=6555.343..6555.349 rows=10 loops=1)
 Planning time: 0.473 ms
 Execution time: 6569.828 ms
(4 rows)

Time: 6614.439 ms (00:06.614)

--nested loop

mydb=# set enable_mergejoin to off;
SET
Time: 0.422 ms
mydb=# explain select t1.* from t1,t2 where t1.f2=t2.f2 limit 10;
                                                     QUERY PLAN
\-----------------------------------------------------------------------
---------------------------------------------
 Limit  (cost=100.12..103.57 rows=10 width=367)
   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.12..103.57
rows=10 width=367)
         ->  Limit  (cost=0.12..3.52 rows=10 width=367)
               ->  Nested Loop  (cost=0.12..339232.00 rows=1000000
width=367)
                     ->  Remote Subquery Scan on all (dn001,dn002)
(cost=100.00..434740.00 rows=1000000 width=367)
                           Distribute results by S: f2
```

```
                          ->  Seq Scan on t1  (cost=0.00..62640.00
rows=1000000 width=367)
                     ->  Materialize  (cost=100.12..100.31 rows=1
width=33)
                          ->  Remote Subquery Scan on all (dn001,dn002)
(cost=100.12..100.30 rows=1 width=33)
                                Distribute results by S: f2
                                ->  Index Only Scan using t2_f1_idx on
t2  (cost=0.12..0.27 rows=1 width=33)
                                      Index Cond: (f2 = t1.f2)
(12 rows)

Time: 1.033 ms
mydb=# explain analyze select t1.* from t1,t2 where t1.f2=t2.f2 limit
10;
                                                                QUERY
PLAN
\-----------------------------------------------------------------------
-----------------------------------------------------------------
 Limit  (cost=100.12..103.57 rows=10 width=367) (actual
time=5608.326..5609.571 rows=10 loops=1)
   ->  Remote Subquery Scan on all (dn001,dn002)  (cost=100.12..103.57
rows=10 width=367) (actual time=5608.323..5608.349 rows=10 loops=1)
 Planning time: 0.347 ms
 Execution time: 5669.901 ms
(4 rows)

Time: 5672.584 ms (00:05.673)

mydb=# set enable_nestloop to off;
SET
Time: 0.436 ms
mydb=#  explain select t1.* from t1,t2 where t1.f2=t2.f2 limit 10;
                                                         QUERY PLAN
\-----------------------------------------------------------------------
-------------------------------------------------
 Limit  (cost=85983.00..85984.94 rows=10 width=367)
   ->  Remote Subquery Scan on all (dn001,dn002)
(cost=85983.00..85984.94 rows=10 width=367)
         ->  Limit  (cost=85883.00..85884.89 rows=10 width=367)
               ->  Hash Join  (cost=85883.00..274580.00 rows=1000000
width=367)
```

```
                      Hash Cond: (t1.f2 = t2.f2)
                      ->  Remote Subquery Scan on all (dn001,dn002)
(cost=100.00..434740.00 rows=1000000 width=367)
                            Distribute results by S: f2
                            ->  Seq Scan on t1  (cost=0.00..62640.00
rows=1000000 width=367)
                      ->  Hash  (cost=100740.00..100740.00 rows=1000000
width=33)
                            ->  Remote Subquery Scan on all (dn001,dn002)
(cost=100.00..100740.00 rows=1000000 width=33)
                                  Distribute results by S: f2
                                  ->  Seq Scan on t2
(cost=0.00..62640.00 rows=1000000 width=33)
(12 rows)

Time: 1.141 ms
mydb=#  explain analyze select t1.* from t1,t2 where t1.f2=t2.f2 limit
10;
                                                                  QUERY
PLAN
\-----------------------------------------------------------------------
-----------------------------------------------------------------------
 Limit  (cost=85983.00..85984.94 rows=10 width=367) (actual
time=1083.691..1085.962 rows=10 loops=1)
   ->  Remote Subquery Scan on all (dn001,dn002)
(cost=85983.00..85984.94 rows=10 width=367) (actual
time=1083.688..1083.699 rows=10 loops=1)
 Planning time: 0.530 ms
 Execution time: 1108.830 ms
(4 rows)

Time: 1117.713 ms (00:01.118)
mydb=#
```

exists 优化

exists 在数据量比较大情况下，一般使用的是 Semi Join ，在 work_mem 足够大的情况下使用的是 hash join，性能会更好。如下，性能提升大约一倍。

```
postgres=# show work_mem;
 work_mem
\----------
 4MB
```

```
(1 row)

Time: 0.298 ms
postgres=# explain  select count(1) from t1 where exists(select 1 from
t2 where t2.t1_f1=t1.f1);
                                                    QUERY PLAN
\-----------------------------------------------------------------------
-----------------------------------------------
 Finalize Aggregate  (cost=242218.32..242218.33 rows=1 width=8)
   ->  Remote Subquery Scan on all (dn001,dn002)
(cost=242218.30..242218.32 rows=1 width=0)
         ->  Partial Aggregate  (cost=242118.30..242118.31 rows=1
width=8)
               ->  Hash Semi Join  (cost=110248.00..242118.30
rows=505421 width=0)
                     Hash Cond: (t1.f1 = t2.t1_f1)
                     ->  Seq Scan on t1  (cost=0.00..17420.00
rows=1000000 width=4)
                     ->  Hash  (cost=79340.00..79340.00 rows=3000000
width=4)
                           ->  Remote Subquery Scan on all (dn001,dn002)
(cost=100.00..79340.00 rows=3000000 width=4)
                                 Distribute results by S: t1_f1
                                 ->  Seq Scan on t2
(cost=0.00..52240.00 rows=3000000 width=4)
(10 rows)

Time: 1.091 ms
postgres=# select count(1) from t1 where exists(select 1 from t2 where
t2.t1_f1=t1.f1);
 count
\--------
 500000
(1 row)

Time: 3779.401 ms (00:03.779)
postgres=# set work_mem to '128MB';
SET
Time: 0.368 ms
postgres=# explain  select count(1) from t1 where exists(select 1 from
t2 where t2.t1_f1=t1.f1);
                                                    QUERY PLAN
```

```
\-----------------------------------------------------------------------
-----------------------------------------
 Finalize Aggregate  (cost=101763.76..101763.77 rows=1 width=8)
   ->  Remote Subquery Scan on all (dn001,dn002)
(cost=101763.75..101763.76 rows=1 width=0)
         ->  Partial Aggregate  (cost=101663.75..101663.76 rows=1
width=8)
               ->  Hash Join  (cost=89660.00..101663.75 rows=505421
width=0)
                     Hash Cond: (t2.t1_f1 = t1.f1)
                     ->  Remote Subquery Scan on all (dn001,dn002)
(cost=59840.00..69443.00 rows=505421 width=4)
                           Distribute results by S: t1_f1
                           ->  HashAggregate  (cost=59740.00..64794.21
rows=505421 width=4)
                                 Group Key: t2.t1_f1
                                 ->  Seq Scan on t2
(cost=0.00..52240.00 rows=3000000 width=4)
                     ->  Hash  (cost=17420.00..17420.00 rows=1000000
width=4)
                           ->  Seq Scan on t1  (cost=0.00..17420.00
rows=1000000 width=4)
(12 rows)

Time: 4.739 ms
postgres=# select count(1) from t1 where exists(select 1 from t2 where
t2.t1_f1=t1.f1);
 count
\--------
 500000
(1 row)

Time: 1942.037 ms (00:01.942)
postgres=#
```