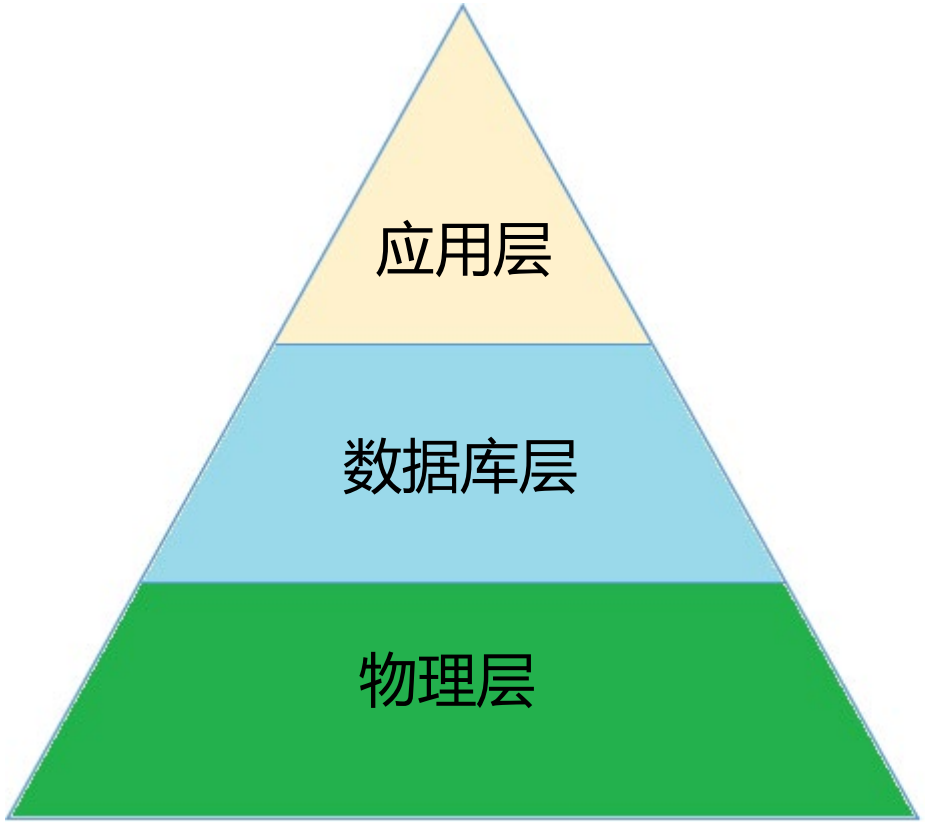
PostgreSQL 性能调优

中兴通讯 王文娟

影响数据库性能的三层架构

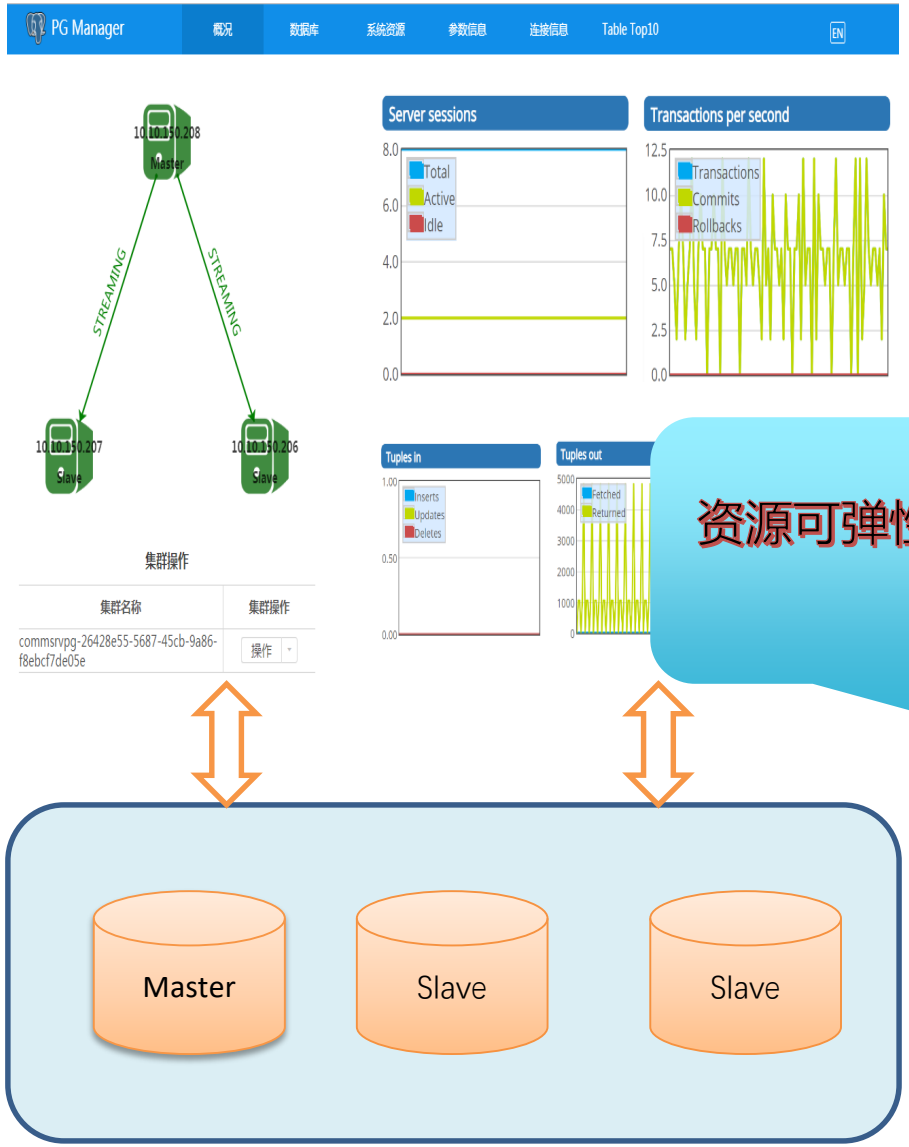


目录

影响性能的硬件因素

硬件因素

容器资源弹性伸缩，动态调节PG处理能力



实时监控容器资源使用情况

- CPU、IO、内存、网络等物理资源
- 数据库连接数、读写请求数、表空间等
- 监控包括网管系统和数据库系统2个层面

根据监控结果产生预警信息

- CPU占用达70%预警，提示增加CPU核数
- 内存、磁盘占用达到阈值时提示增加配置

资源可弹性伸缩是PG云化部署的最大收益之一！

资源配置

内存、磁盘等

提高实际使用率

不建议做

目录

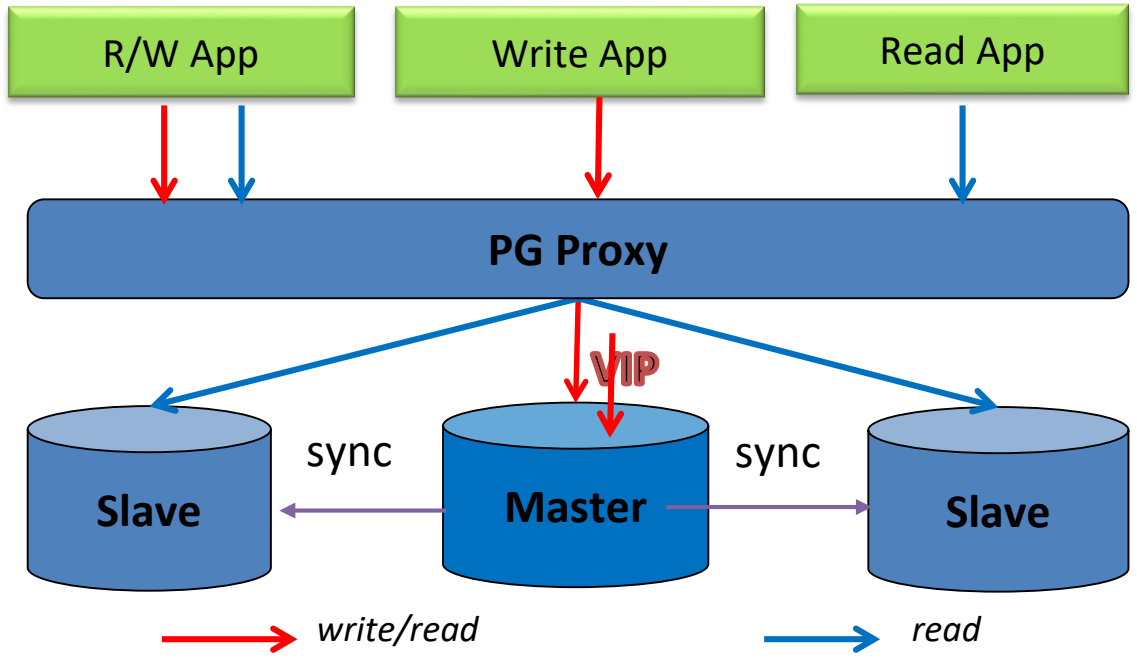
影响性能的数据库因素

读写分离，充分发挥Slave节点能力

- synchronous_commit参数影响主备节点的数据一致性状态
- 要求高一致性读的场景必须把该参数配置成remote_replay，比如涉及到金额的在线交易事务
- 对数据实时性要求不高的场景，可以把该参数配置成默认的on级别，比如报表统计
- 该参数配置等级越高，Master节点写延迟越大，性能影响越大

PG参数synchronous_commit指定事务提交所要求的WAL记录同步等级，可以取5个有效值：

① off：事务提交不需要等待WAL日志刷写入（flush）本地磁盘
② local：事务等待WAL日志刷写入本地磁盘后才提交
③ remote write：事务等待同步备节点接收到WAL日志并写入操作系统才能提交
④ on：事务等待同步备节点接收到WAL日志并刷写到（flush）磁盘才能提交
⑤ remote_replay：事务等待同步备节点接收到WAL日志并回放完毕才能提交



优化数据库配置参数

优化内存资源类参数

控制系统在构建索引时将使用的最大内存量。
为了构建一个B树索引，必须对输入的数据进行排序，如果要排序的数据在maintenance_work_mem设定的内存中放置不下，它将会溢出到磁盘中。

① **maintenance_work_mem**

② autovacuum

autovacuum_work_mem
(autovacuum_*等系列参数)

扫描索引的代价

vacuum扫描索引并删除这些TID对应的所有索引项。如果它在扫描完整个表之前耗尽了存放无效元组TID所用的内存，它将停止表扫描，转而扫描索引，以丢弃堆积的TID列表，之后从它中断的位置继续扫描表。对于一个大表，多次扫描索引的代价是非常昂贵的，特别是在表中有很多索引的情况下。如果maintenance_work_mem设置太低，甚至可能需要两次以上的索引扫描。

内存耗尽

当使用缺省配置autovacuum_max_workers = 3，并且假设设置maintenance_work_mem = 10GB，你将会经常消耗30GB的内存专门用于自动空间清理，这还不包括你可能从前台发起的VACUUM或 CREATE INDEX操作所需的内存。这样，你会很容易把一个小系统的内存耗尽，即便是一个大系统，也可能存在诸多性能问题。

估算

如何估算autovacuum_work_mem?
建议：maintenenance_work_mem内存大小大约是最大表的1/100

shared_buffers | **work_mem** | **dynamic_shared_memory_type** | **huge_page**

优化数据库配置参数

优化脏页刷写类参数

bgwriter_delay | bgwriter_lru_maxpages | bgwriter_lru_multiplier | bgwriter_flush_after

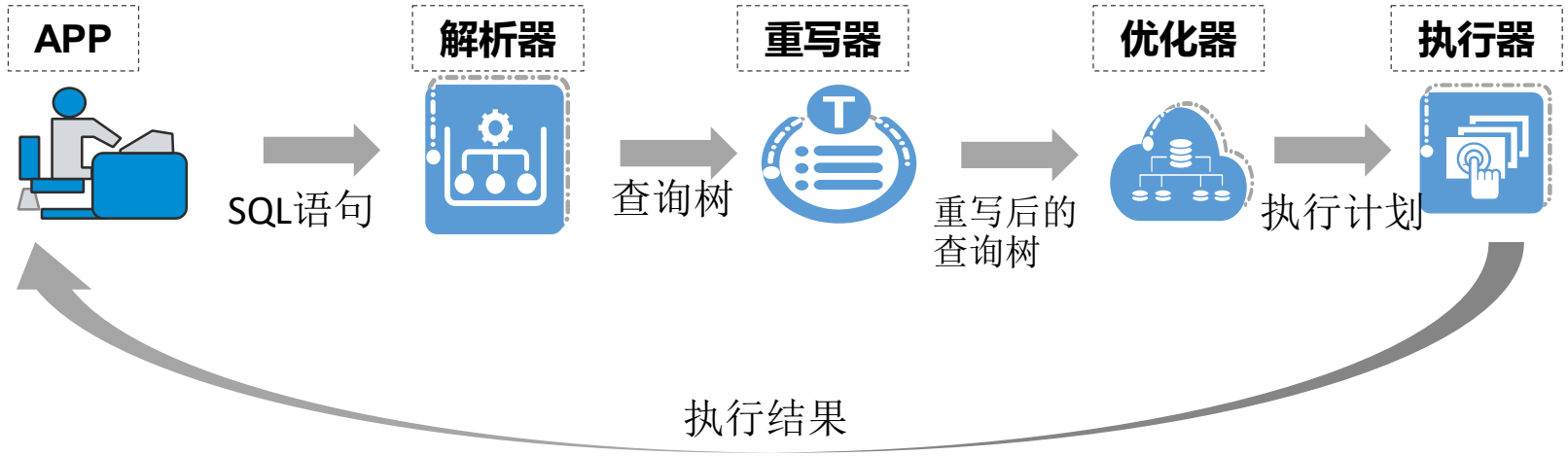
优化WAL相关参数

fsync | synchronous_commit | wal_sync_method | full_page_writes

commit_delay | commit_siblings | max_wal_size | min_wal_size

wal_*类名称参数 | chechpoint_*类名称参数

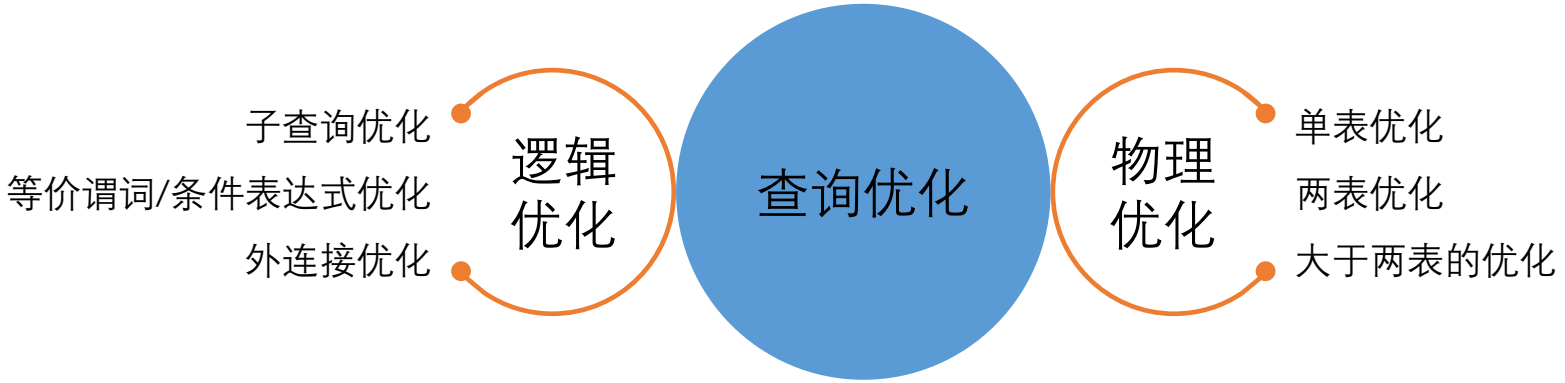
PG的查询优化

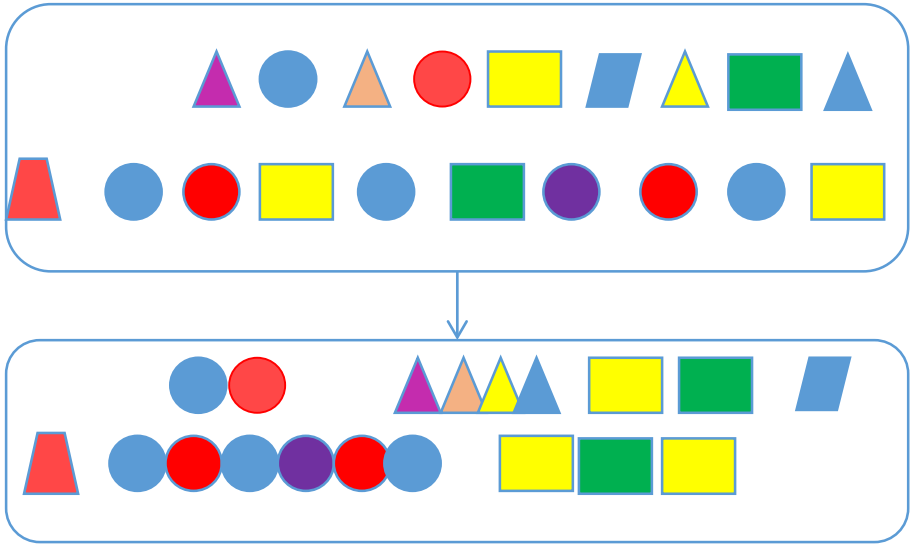


debug_print_parse（重写前的查询树）

debug_print_rewritten（重写后的查询树）

重写前查询树	重写后查询树
LOG: parse tree: DETAIL: {QUERY .....//省略 {ALIAS :aliasname myview :colnames ("id1" "id2") } .....//省略 }	LOG: rewritten parse tree: DETAIL: ({QUERY .....//省略 {ALIAS :aliasname department :colnames ("id1" "id2") } .....//省略 })





PG的查询优化

并行查询

```
postgres=# EXPLAIN SELECT * FROM employee WHERE
empid<1000;
                        QUERY PLAN
-------------------------------------------------------------------
Gather  (cost=1000.00..11713.53 rows=992 width=15)
  Workers Planned: 2
  ->  Parallel Seq Scan on employee  (cost=0.00..10614.33
rows=413 width=15)
        Filter: (empid < 1000)
(4 rows)
```

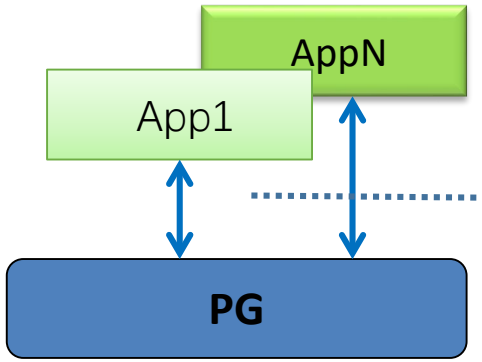
```
postgres=# set  max_parallel_workers_per_gather=0; //关掉并
    行顺序扫描
SET
postgres=#  EXPLAIN SELECT * FROM employee WHERE
     empid<1000;
                        QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on employee  (cost=0.00..17906.00 rows=992
     width=15)
   Filter: (empid < 1000)
(2 rows)
```

虽然并行查询可以提高查询性能，但是当存在复杂多事务时，要谨慎使用，尤其不要在数据库层强制使用并行查询。

目录

影响性能的应用层因素

数据库SQL性能问题排查



测试中发现问题：

CPU占用率高

IO占用高

内存耗尽

网络流量占满

.....

01 找到性能瓶颈

02 找到性能的提升点

03 使用Explain

04 SQL优化

常见的SQL优化方法：使用索引避免表扫描

- 在等值或范围查询、排序及分组查询时使用索引字段，以尽量避免表扫描
- 减少使用导致索引扫描失效的SQL语句书写方式（函数，表达式等）

实例：索引失效
场景：查询年工资大于15万元的员工

```
postgres=# EXPLAIN ANALYZE SELECT empname FROM employee WHERE salary*12>150000;
                                                        QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
 Seq Scan on employee  (cost=0.00..20406.00 rows=333333 width=3) (actual time=0.027..159.189 rows=166141 loops=1)
   Filter: ((salary * 12) > 150000)
   Rows Removed by Filter: 833859
 Planning time: 0.096 ms
 Execution time: 169.870 ms
(5 rows)
```

```
postgres=# EXPLAIN ANALYZE SELECT empname FROM employee WHERE salary>150000/12;
                                                        QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on employee  (cost=3123.47..10611.79 rows=166586 width=3) (actual time=24.631..78.022 rows=166141 loops=1)
  Recheck Cond: (salary > 12500)
  Heap Blocks: exact=5406
  ->  Bitmap Index Scan on idx_salary  (cost=0.00..3081.82 rows=166586 width=0) (actual time=23.610..23.610 rows=166141 loops=1)
        Index Cond: (salary > 12500)
 Planning time: 0.147 ms
 Execution time: 88.302 ms
(7 rows)
```

常见的SQL优化方法：子查询优化

◆ 对性能低的子查询进行Explain分析并改写语句

实例：IN子查询优化
分析：原始SQL语句中的外层父查询不需要获取emp_order_insurance表中的任何字段值，因此该表只需要出现在子查询的FROM子句中

```
postgres=# EXPLAIN ANALYZE SELECT DISTINCT deptname FROM
emp_order_insurance,department WHERE department.deptid IN (SELECT deptid FROM
employee WHERE  employee.empid=emp_order_insurance.empid) ;
                                  QUERY PLAN
------------------------------------------------------------------------------
 HashAggregate  (cost=44512682.50..44512782.50 rows=10000 width=9) (actual
time=31441.556..31441.808 rows=952 loops=1)
   Group Key: department.deptname
   ->  Nested Loop  (cost=0.00..44500182.50 rows=5000000 width=9) (actual
time=23.688..31440.051 rows=1000 loops=1)
         Join Filter: (SubPlan 1)
         Rows Removed by Join Filter: 9999000
         ->  Seq Scan on department  (cost=0.00..164.00 rows=10000 width=13) (actual
time=0.022..2.608 rows=10000 loops=1)
         ->  Materialize  (cost=0.00..21.00 rows=1000 width=4) (actual time=0.000..0.102
rows=1000 loops=10000)
               ->  Seq Scan on emp_order_insurance  (cost=0.00..16.00 rows=1000 width=4) (actual
time=0.014..0.366 rows=1000 loops=1)
         SubPlan 1
          ->  Index Scan using pk_employee on employee  (cost=0.42..8.44 rows=1 width=4)
(actual time=0.002..0.002 rows=1 loops=10000000)
                Index Cond: (empid = emp_order_insurance.empid)
 Planning time: 0.253 ms
 Execution time: 31442.049 ms
(13 rows)
```

```
postgres=#  EXPLAIN ANALYZE SELECT DISTINCT deptname FROM department WHERE department.deptid
IN (SELECT deptid FROM employee,emp_order_insurance WHERE
employee.empid=emp_order_insurance.empid) ;
                                  QUERY PLAN
------------------------------------------------------------------------------
 HashAggregate  (cost=310.54..320.54 rows=1000 width=9) (actual time=8.452..8.753 rows=952 loops=1)
   Group Key: department.deptname
   ->  Hash Semi Join  (cost=102.91..308.04 rows=1000 width=9) (actual time=2.529..7.893 rows=952
loops=1)
         Hash Cond: (department.deptid = employee.deptid)
         ->  Seq Scan on department  (cost=0.00..164.00 rows=10000 width=13) (actual time=0.017..2.316
rows=10000 loops=1)
         ->  Hash  (cost=90.41..90.41 rows=1000 width=4) (actual time=2.496..2.496 rows=1000 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 44kB
               ->  Merge Join  (cost=2.89..90.41 rows=1000 width=4) (actual time=0.045..2.109 rows=1000
loops=1)
                     Merge Cond: (employee.empid = emp_order_insurance.empid)
                     ->  Index Scan using pk_employee on employee  (cost=0.42..31389.42 rows=1000000 width=8)
(actual time=0.017..0.503 rows=1001 loops
=1)
                     ->  Index Only Scan using idx_emp_order_insurance on emp_order_insurance  (cost=0.28..44.27
rows=1000 width=4) (actual time=0.023.
.0.557 rows=1000 loops=1)
                           Heap Fetches: 1000
 Planning time: 0.769 ms
 Execution time: 8.943 ms
(14 rows)
```

常见的SQL优化方法：利用LEFT JOIN的连接消除优化功能

◆ 对性能低的子查询进行Explain分析并改写语句

实例：利用LEFT JOIN的连接消除优化功能

分析：PostgreSQL优化器支持消除某些LEFT JOIN语句中的无用的连接，当LEFT JOIN输出的仅仅是左表中的字段，且连接条件中的右表的字段具有唯一性，那么就会消除外连接。因此在类似场景下，当认为可能会消除外连接时，可以考虑是否可以使用LEFT JOIN的连接消除功能。在本例中，如果确定parentid原本就是唯一的，那么可以增加DISTINCT字段进行SQL优化。

```
postgres=# EXPLAIN ANALYZE SELECT employee.empname FROM
employee LEFT JOIN  department ON employee.deptid =
department.parentid;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Hash Left Join  (cost=289.00..41929.38 rows=1000000 width=3)
(actual time=6.809..563.189 rows=1565506 loops=1)
   Hash Cond: (employee.deptid = department.parentid)
   ->  Seq Scan on employee  (cost=0.00..15406.00 rows=1000000
width=7) (actual time=0.019..141.899 rows=1000000 loops=1)
   ->  Hash  (cost=164.00..164.00 rows=10000 width=4) (actual
time=6.756..6.756 rows=10000 loops=1)
         Buckets: 16384  Batches: 1  Memory Usage: 480kB
         ->  Seq Scan on department  (cost=0.00..164.00 rows=10000
width=4) (actual time=0.012..3.420 rows=10000 loops=1)
 Planning time: 0.800 ms
 Execution time: 663.777 ms
(8 rows)
```

```
postgres=# EXPLAIN ANALYZE SELECT employee.empname FROM employee
LEFT JOIN (SELECT DISTINCT parentid FROM department)dept ON
employee.deptid = dept.parentid;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on employee  (cost=0.00..15406.00 rows=1000000 width=3)
(actual time=0.020..163.427 rows=1000000 loops=1)
 Planning time: 0.115 ms
 Execution time: 225.985 ms
(3 rows)
```

常见的SQL优化方法：使用物化视图

◆ 使用物化视图提高性能

实例：获取历史销售数据
分析：由于不关心当前的最新数据，所以可以使用物化视图，使用后，性能约是原来使用视图的4倍。

```
postgres=# EXPLAIN ANALYZE SELECT * FROM sales_summary WHERE
seller_no=10001;
                                QUERY PLAN
-------------------------------------------------------------------------
 GroupAggregate  (cost=2291.01..2291.03 rows=1 width=16) (actual
time=32.495..32.496 rows=1 loops=1)
   Group Key: sales.seller_no, sales.sales_date
   ->  Sort  (cost=2291.01..2291.02 rows=1 width=12) (actual
time=32.487..32.488 rows=1 loops=1)
         Sort Key: sales.sales_date
         Sort Method: quicksort  Memory: 25kB
         ->  Seq Scan on sales  (cost=0.00..2291.00 rows=1 width=12) (actual
time=3.297..32.456 rows=1 loops=1)
               Filter: ((seller_no = 10001) AND (sales_date < now()))
               Rows Removed by Filter: 99999
 Planning time: 0.394 ms
 Execution time: 32.568 ms
```

```
postgres=# EXPLAIN ANALYZE SELECT * FROM sales_summary_m
WHERE seller_no=10001;
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on sales_summary_m  (cost=0.00..538.00 rows=1
width=16) (actual time=0.022..8.494 rows=1 loops=1)
   Filter: (seller_no = 10001)
   Rows Removed by Filter: 29999
 Planning time: 0.096 ms
 Execution time: 8.528 ms
```

其他常见的SQL优化方法

- **使用UNION ALL代替UNION**
- **使用PG的几种特色索引：GIN索引、GiST索引、SP-GiST索引、BRIN索引**
- **尽量减少使用SELECT *：**

1、SELECT列是关系运算中的投影操作，它并不影响结果集的条数，但是会影响到结果集数据包的大小。数据包大，会影响到结果集返回的速度。
2、此外，如果SQL语句不是简单查询，而是进一步涉及到表连接操作、排序操作、子查询操作等时，会较大程度影响到实际使用的工作内存。
3、当仅需要查询单列且该列字段是索引字段时，那么把SQL语句写为只查询该列，则可能会使用Index Only Scan，该种扫描类型的查询性能较高。
4、从应用角度看，如果使用SELECT *语句，当数据库表字段变更后，获得的字段值及顺序会随之变更，导致应用程序某些代码异常。

数据库的逻辑设计

- **关系数据库的逻辑设计非常重要，必须满足一定的范式规范，最低要求是满足第一范式（1NF），此外还有2NF、3NF、BCNF、4NF、5NF**
- **需要兼顾性能与范式规范要求，满足越高的范式则数据冗余和操作异常越少，但是性能也越低**
- **谨慎使用PG为支持NoSQL功能而引入的数据类型，比如JSON和JSONB**

说明：JSON/JSONB是PG为了提供文档数据库能力而引入的数据类型，具有NoSQL的Schema-free和Document Based的特征，支持动态增删字段以及对象嵌套的数据结构。文档数据库的典型代表是MongoDB。

案例　把JSONB表转换成普通表提高性能

场景：
①有些业务原先使用MongoDB，迁移到PG之后直接使用JSONB类型替代
②不同模块需要动态地添加各自的业务字段
③复杂查询：GROUP BY，ORDER BY，输出结果集50万~200万

```
SELECT value FROM t1 WHERE (( value -> 'a' IN ('1','2') AND value -> 'v' = '1' AND (( value
-> 'm' IS NULL OR value -> 'm' -> 'value' IN ('0') ) ) AND value -> 'p' IN ('1','2','3','4') ) )
ORDER BY value -> 's' DESC, value -> 'id' DESC LIMIT 10 OFFSET 40 ;
```

问题：复杂查询执行时间太长，不能满足业务需求。表现为：虽然JSONB灵活，表达力强大，但是所有元素（子字段）都存放在一个JSONB字段中（注，整个表就一个value字段），在查询处理过程中，从单个value字段解析出各个子字段耗费了大量CPU，在并发查询场景下，CPU几乎100%。

解决：(1)把JSONB表转换成普通表，把公共字段提取出来，创建一个父表tb，然后利用PG的表继承特性,各业务模块继承父表tb创建自己的子表tXXX，在子表中添加各自独有的业务字段。或者，(2)公共字段改造成普通字段，预留一个JSONB字段让各个业务模块添加自己的独有字段。

效果：查询性能提升10**倍**左右，满足业务需求。

数据库的逻辑设计

- **索引对于提高SQL查询性能至关重要，原则上WHERE查询条件中经常使用的过滤字段一定要创建索引**
- **PG索引功能非常强大，提供了B-tree、Hash、GiST、SP-GiST、GIN、BRIN等多种索引类型，每种索引类型适合不同的查询场景**
- **注意：索引不是越多越好，够用就好。索引太多会影响表的DML语句的性能**

案例　为JSONB字段创建合适的索引提高性能

场景： 还是上述案例中的场景，由于把JSONB表彻底改造成普通表对业务现有的代码逻辑冲击较大，因此在保持当前表定义不变的前提下如何提升查询性能成为必须要解决的问题。

问题： 起初为JSONB字段创建GIN索引，查询性能有不小的提升，基本能满足业务需求，但入库性能下降较多，不满足要求。于是业务尝试改用B-tree索引，但是有些查询场景性能很差，不能接受。

```
SELECT to_jsonb(__t__) AS value FROM (SELECT value -> 'p' AS p,value -> 'a' AS a , COUNT(value) AS count FROM t1 WHERE ( value -> 'v' = '1' ) GROUP BY value -> 'p',value -> 'a' ORDER BY value -> 'p' DESC) __t__;
```

定位：

(1)使用explain命令分析它的执行计划，发现优化器估算GROUP BY后的记录数高达173383 行，所以它选择了GroupAggregate算法。

(2)进一步分析发现，用于GROUP BY的p和a子字段的不同取值（Distinct Values）的个数都很少，分别只有4个和1个，因此优化器的估算严重不准确。这归咎于没有对p和a子字段分别创建索引，致使优化器使用错误的统计信息。

(3)分别p和a子字段创建索引后，再使用explain命令查看执行计划，发现优化器估算GROUP BY后的记录数只有3 行，改成使用HashAggregate算法进行分组。

效果： 上述优化之后，该SQL查询性能提升3倍多，满足业务需求。

数据库的逻辑设计

mongoing 中文社区 PostgreSQL 中文社区

```
 Subquery Scan on __t__  (cost=274591.10..281092.97 rows=173383
width=32) (actual time=15004.376..17488.863 rows=4 loops=1)
   Output: to_jsonb(__t__.*)
   Buffers: shared hit=34395, temp read=30752 written=30764
   -> GroupAggregate  (cost=274591.10..278925.68
rows=173383 width=72) (actual time=15004.299..17488.691
rows=4 loops=1)
         Output: ...
         Group Key: ...
         Buffers: shared hit=34395, temp read=30752 written=30764
         -> Sort  (cost=274591.10..275024.56 rows=173383 width=1467)
(actual time=15004.264..16796.003 rows=173379 loops=1)
               Output:...
               Sort Key:...
               Sort Method: external merge  Disk: 246016kB
               Buffers: shared hit=34395, temp read=30752 written=30764
               -> Seq Scan on ...
                     Output: ...
                     Filter:...
                     Buffers: shared hit=34395
 Planning time: 2.098 ms
 Execution time: 17800.406 ms
(18 rows)
```

```
Subquery Scan on __t__  (cost=39162.99..39163.04 rows=3
width=32) (actual time=5314.232..5314.244 rows=4 loops=1)
   Output: to_jsonb(__t__.*)
   Buffers: shared hit=34395
   -> Sort  (cost=39162.99..39163.00 rows=3 width=72) (actual
time=5314.187..5314.188 rows=4 loops=1)
         Output: ...
         Sort Key: ...
         Sort Method: quicksort  Memory: 25kB
         Buffers: shared hit=34395
         -> HashAggregate  (cost=39162.92..39162.97
rows=3 width=72) (actual time=5314.167..5314.169
rows=4 loops=1)
               Output:...
               Group Key:...
               Buffers: shared hit=34395
               -> Seq Scan on ...
                     Output:...
                     Filter: ...
                     Buffers: shared hit=34395
 Planning time: 0.515 ms
 Execution time: 5314.341 ms
(18 rows)
```

优化应用使用数据库方式

- 同样的查询请求尽量只发起一次，可以在应用端进行数据缓存并完成后续工作，不要在同一流程中多次发起同一查询。
- 默认的查询网页不显示全量数据的查询，可以默认只显示部分数据（条件查询），尤其涉及到两个或多个大表的查询（避免大表间的笛卡尔积的运算），例如只显示当天的数据。
- 在网页的排序功能中，去掉排序慢的不必要的字段的排序，过滤性非常差的字段进行全表排序时由于磁盘IO耗时非常严重（例如：同一字段取值相同时为了保证顺序还要使用主键id字段进行二次排序），无法保证效率。
- 减少SQL语句的硬解析，尽可能利用软解析。

实例: 某应用从Oracle数据库迁移到PG，迁移后，测试性能比Oracle数据库低，且高并发时无法正常运行业务。

分析：应用程序中大量使用到了SELECT ... WHERE 字段a='1'（数值会变）语句，在做每个查询语句时，PG都需要为其重新生成一次执行计划（即硬解析），会严重影响性能。

修改方法：考虑对应用程序代码改动最小，使用函数的方法，每次把查询值作为参数传入。这样PG函数内部会做绑定变量。

效果：修改前：900并发时，资源消耗为: 29.68%（均值）/55.10%（峰值）；
1700并发时，应用消息积压出现重启 。
修改后：1700 TPS并发，资源消耗为15.76%（平均值）/23.42%（峰值），与Oracle时资源消耗基本一致。

```
select p.productcode, NULL, (p.funcswitch & 2)
        from zx1.s_user_subs zx2, zx1.s_product p
        where zx2.payuserid = i_payuserid and p.producttype <> 2
        and zx2.productid = p.productid and (p.terminaltypes &
i_termnaltype <> 0 or p.terminaltypes = 0)
        and zx2.begintime <= to_char(current_timestamp,
'yyyymmddhh24miss')
        and zx2.endtime >= to_char(current_timestamp,
'yyyymmddhh24miss') and zx2.status = 1
union all
select NULL, m.mixnum, c.mediaservices
  from zx1.s_user_order a, zx1.s_product p, zx1.s_channel c,
zx1.s_mix_channel m
  where a.payuserid = i_payuserid
  and a.productid = p.productid
  and a.contenttype = 2
  and a.status = 1
  and (p.terminaltypes & i_termnaltype <> 0 or p.terminaltypes = 0)
  and c.mediaservices & i_mediaservice <> 0
  and a.begintime <= to_char(current_timestamp, 'yyyymmddhh24miss')
  and a.endtime >= to_char(current_timestamp, 'yyyymmddhh24miss')
  and a.contentcode=c.channelcode
  and m.mapcode = c.channelcode
  and m.teamid = -1
```

```
\echo sp_cponly_getproductlist
create or replace function sp_cponly_getproductlist(
  i_payuserid int,
  i_termnaltype int,
  i_mediaservice int
)
returns setof zx1.type_cp_productlist
as
$
declare
rec zx1.type_cp_productlist%rowtype;
begin
  for rec in (select p.productcode, NULL, (p.funcswitch & 2)
        from zx1.s_user_subs zx2, zx1.s_product p
        where zx2.payuserid = i_payuserid and p.producttype <> 2
        and zx2.productid = p.productid and (p.terminaltypes & i_termnaltype <> 0 or p.terminaltypes = 0)
        and zx2.begintime <= to_char(current_timestamp, 'yyyymmddhh24miss')
        and zx2.endtime >= to_char(current_timestamp, 'yyyymmddhh24miss') and zx2.status = 1) loop
    return next rec;
  end loop;
  for rec in (select NULL, m.mixnum, c.mediaservices
  from zx1.s_user_order a, zx1.s_product p, zx1.s_channel c, zx1.s_mix_channel m
  where a.payuserid = i_payuserid
  and a.productid = p.productid
  and a.contenttype = 2
  and a.status = 1
  and (p.terminaltypes & i_termnaltype <> 0 or p.terminaltypes = 0)
  and c.mediaservices & i_mediaservice <> 0
  and a.begintime <= to_char(current_timestamp, 'yyyymmddhh24miss')
  and a.endtime >= to_char(current_timestamp, 'yyyymmddhh24miss')
  and a.contentcode=c.channelcode
  and m.mapcode = c.channelcode
  and m.teamid = -1) loop
    return next rec;
  end loop;
  return;
end;
$
language 'plpgsql';
```