Efficient Massively Parallel Join Optimization for Large Queries

Riccardo Mancini,Riccardo Mancini,Bikash Chandra,Vasilis Mageirakos,Anastasia Ailamaki

摘要

本文提出了一个大规模平行 JOIN 算法，可以在利用现代硬件并行计算的同时修剪大规模搜索空间，目的在于减少执行多表 JOIN 的查询优化树生成时间。

关键词：Parallel Query Optimization；Parallel Query Optimization

1 引言

现代数据分析工作经常涉及到大数量表的联合查询。最优化查询计划对于在可接受的范围内实现查询工作是极为重要的。然而，随着查询涉及的表的数量的增加，由于查询计划的优化随表数量呈指数增长，的找出该查询的最优查询计划成为了查询性能的瓶颈。现存的最优化查询计划算法在并行性和有效计算率上无法兼顾；因此，作者提出了一种新的大规模并行化查询计划生成算法——MPDP，能在有效的减少搜索空间的同时利用现代硬件系统进行大规模并行计算。相比于目前最好的优化算法，MPDP 算法有着几个数量级的速度提升。因而可以将启发算法的使用极限从 12 张表提升到 25 张表。与此同时，对于必须使用启发式算法的情况，MPDP 也进行了优化。最终的结果表明，MPDP 算法可以提升约 7 倍以上的性能。

本次课程的复现工作将实现论文中的 MPDP 算法，并将其整合到 postgreSQL 中。再选择一个新的数据集进行性能测试，通过与原有的性能进行对比来验证 MPDP 算法的有效性。

2 相关工作

JOIN 优化问题自诞生之日起已历经 40 余年。到目前为止，已有文章讨论了处理 1000 个关系的大型查询的重要性，并发现现有优化器无法处理这些查询。我们将先前的工作分为最优算法和启发式解决方案。

2.1 最优算法

第一个 JOIN 计划最优算法是 DPSIZE，它被广泛用于众多大型的开源和商业数据库中，比如 PostgresSQL 和 IBM DB2。DPSUB 使用子集驱动的枚举方式。尽管 DPSIZE 和 DPSUB 的性能取决于查询的拓扑结构，由此会计算一些不必要的对，但是它们是可并行的。Moerkette 提出了 DPCCP 算法，使用基于枚举的 JOIN 图并只计算 CCP-Pair。在单机运行的情况下，DPCCP 比 DPSIZE 和 DPSUB 表现都好。然而，由于枚举 JOIN-pair 时，DPCCP 算法存在依赖，DPCCP 算法并不能并行执行。基于以上的情况，这篇文章中提出了新的算法：MPDP。可以在并行化计算的同时最小化冗余的 JOIN-pair 计算。

2.2 启发式算法

由于 JOIN 顺序优化是 NP 难问题。因此存在很多使用启发式算法的解决方案。一些技术着眼于在优先搜索空间内生成查询计划。IKKBZ 限制搜索空间到左深树。类似的，Trummer 和 Koch 将 JOIN 顺序优化问题表述为整数线性优化问题 GOO 技术和 min-sel 技术会利用贪心算法找到最优计划。

3 本文方法

3.1 本文方法概述

1. 拟使用 c/c++ 实现如图 1伪代码所示的 MPDP 生成算法：

Algorithm 3 : MPDP generalization (with cycles)

```
1:  for i := 2 to QI.querySize do
2:      S_i = {S | S ⊆ R and |S| = i and S is connected}
3:      for all S ∈ S_i do
4:          BLOCKS ← Find-Blocks(S, QI)
5:          for all block ∈ BLOCKS do
6:              for all lb ⊂ block, lb ≠ ∅ do
7:                  EvaluatedCounter++
8:                  rb ← block \ lb
9:                  /*Begin CCP Block */
10:                 if rb == ∅ or lb == ∅ continue
11:                 if not lb is connected  continue
12:                 if not rb is connected  continue
13:                 if not rb ∩ lb == ∅ continue
14:                 if not rb is connected to lb  continue
15:                 /* End CCP Block */
16:                 CCP-Counter ++
17:                 S_left ← grow(lb, S \ rb)
18:                 S_right ← S \ S_left
19:                 CurrPlan = CreatePlan(S_left, S_right)
20:                 if CurrPlan < BestPlan(S) then
21:                     BestPlan(S) = CurrPlan
22:                 end if
23:             end for
24:         end for
25:     end for
26: end for
```

图 1: MPDP 生成算法

3.2 算法正确性证明

（1）算法生成的 Sl 和 Sr 一定是 CCP-Pair：先证明 S 中不属于 lb 和 rb 的一点要么和 lb 连通，要么和 rb 连通，之后将 S 分为四块，观察 grow 的输出，可以证明 Sl 和 Sr 一定是 CCP-Pair

（2）所有的 CCP-Pair 都可以被此算法生成：等价于证明对于任意的 CCP-Pair，跨边都属于一个 block

3.3 时间复杂度分析

（1）原算法瓶颈：第八行 for 循环的时间复杂度是 2 的 |S| 次方，每一次循环都需要判断当前 Pair 是否为一个 CCP-Pair 由此，如果存在某种算法能够减少 CCP-Pair 的判断次数，必定能提升算法的运

行速度

（2）改进后算法：不再对 S 中的所有顶点进行穷举来得到 CCP-Pair，而是找到 S 中所有的 block，再对 block 穷举, 这样可以大大降低枚举次数

4 复现细节

4.1 与已有开源代码对比

根据 MPDP 生成算法伪代码，利用 c++ 语言复现出 MPDP 算法如下所示

```
#include <list>
#include <vector>
#include <stack>
#include <iostream>
#include <cmath>
#include <cstdint>

#include "optimizer/gpuqo_common.h"

#include "gpuqo.cuh"
#include "gpuqo_timing.cuh"
#include "gpuqo_debug.cuh"
#include "gpuqo_cost.cuh"
#include "gpuqo_filter.cuh"
#include "gpuqo_cpu_dpsub.cuh"
#include "gpuqo_dpsub.cuh"
#include "gpuqo_binomial.cuh"

using namespace std;

template<typename BitmapsetN>
struct call_stack_el_t {
    unsigned int u;
    BitmapsetN edges;
    bool remain;
};
```

图 2: 代码截图 1

```
template<typename BitmapsetN, typename memo_t, bool manage_best>
class DPsubBiCCCPUAlgorithm : public DPsubGenericCPUAlgorithm<BitmapsetN, memo_t, manage_best> {
private:

    BitmapsetN output_block(unsigned int u, unsigned int v,
        stack<edge_stack_el_t>& edge_stack) {
        BitmapsetN block(0);

        edge_stack_el_t el;

        do {
            el = edge_stack.top();
            edge_stack.pop();

            block.set(el.u);
            block.set(el.v);
        } while (el.u != u || el.v != v);

        LOG_DEBUG("output_comp(%d, %d): %u\n", u, v, block.toUint());

        return block;
    }
```

图 3: 代码截图 2

```
void find_blocks(BitmapsetN set, list<BitmapsetN>& blocks) {
    stack<call_stack_el_t<BitmapsetN>> call_stack;
    stack<edge_stack_el_t> edge_stack;
    vector<bool> visited(BitmapsetN::SIZE, false);
    vector<int> depth(BitmapsetN::SIZE, -1);
    vector<unsigned int> parent(BitmapsetN::SIZE, 0);
    vector<int> low(BitmapsetN::SIZE, numeric_limits<int>::max());

    auto info = CPUAlgorithm<BitmapsetN, memo_t>::info;

    LOG_DEBUG("find_blocks(%u)\n", set.toUint());

    call_stack.push((call_stack_el_t<BitmapsetN>) {
        .u = set.lowestPos(),
            .edges = BitmapsetN(0),
            .remain = false
    });
    int curr_depth = 1;
    while (!call_stack.empty()) {
        call_stack_el_t<BitmapsetN> el = call_stack.top();
        call_stack.pop();
        if (el.edges.empty()) {
            visited[el.u] = true;
            depth[el.u] = curr_depth;
            LOG_DEBUG("visited %d at depth %d\n", el.u, curr_depth);
            curr_depth++;
```

图 4: find-block1

```
            BitmapsetN edges = info->edge_table[el.u - 1] & set;
            if (!edges.empty()) {
                call_stack.push((call_stack_el_t<BitmapsetN>) {
                    .u = el.u,
                        .edges = edges,
                        .remain = false
                });
            }
        }
        else {
            unsigned int v = el.edges.lowestPos();
            LOG_DEBUG("edge %d %d: ", el.u, v);
            if (el.remain) {
                if (low[v] >= depth[el.u]) {
                    blocks.push_back(output_block(el.u, v, edge_stack));
                }
                low[el.u] = min(low[el.u], low[v]);
                LOG_DEBUG("remain, update low of %d: %d\n", el.u, low[el.u]);
                el.edges.unset(v);
                if (!el.edges.empty()) {
                    call_stack.push((call_stack_el_t<BitmapsetN>) {
                        .u = el.u,
                            .edges = el.edges,
                            .remain = false
                    });
                }
            }
```

图 5: find-block2

```
            else {
                if (!visited[v]) {
                    edge_stack.push((edge_stack_el_t) {
                        .u = el.u,
                            .v = v
                    });
                    parent[v] = el.u;
                    call_stack.push((call_stack_el_t<BitmapsetN>) {
                        .u = el.u,
                            .edges = el.edges,
                            .remain = true
                    });
                    LOG_DEBUG("queue visit to %d\n", v);
                    call_stack.push((call_stack_el_t<BitmapsetN>) {
                        .u = v,
                            .edges = BitmapsetN(0),
                            .remain = false
                    });
                }
                else if (parent[el.u] != v && depth[v] < depth[el.u]) {
                    edge_stack.push((edge_stack_el_t) {
                        .u = el.u,
                            .v = v
                    });
                    low[el.u] = min(low[el.u], depth[v]);
                    LOG_DEBUG("update low of %d: %d", el.u, low[el.u]);
                    el.edges.unset(v);
```

图 6: find-block3

```
                    if (!el.edges.empty()) {
                        call_stack.push((call_stack_el_t<BitmapsetN>) {
                            .u = el.u,
                                .edges = el.edges,
                                .remain = false
                        });
                    }
                    else {
                        LOG_DEBUG(" (end %d)", el.u);
                    }
                    LOG_DEBUG("\n");
                }
                else {
                    LOG_DEBUG("skip");
                    el.edges.unset(v);
                    if (!el.edges.empty()) {
                        call_stack.push((call_stack_el_t<BitmapsetN>) {
                            .u = el.u,
                                .edges = el.edges,
                                .remain = false
                        });
                    }
                    else {
                        LOG_DEBUG(" (end %d)", el.u);
                    }
                    LOG_DEBUG("\n");
                }
```

图 7: find-block4

4.2 实验环境搭建

使用具有 Intel Xeon Gold 6326 @ 32x 3.5GHz 的服务器 CPU 具有 4 个内核和 12 个线程，具有 128GB 的 RAM，GPU 为 NVIDIA T400 4GB

4.3 界面分析与使用说明

将算法整合到 postgreSQL 中，利用 pg 进行性能测试

```
psql (14.0)
Type "help" for help.

postgres=# alter user postgres with password 'postgres'
postgres-#
postgres-# exit
Use \q to quit.
postgres-# \q
bash-4.2$ psql
could not change directory to "/root": Permission denied
psql (14.0)
Type "help" for help.

postgres=# exit
bash-4.2$ pwd
/root
bash-4.2$ exit
exit
[root@localhost ~]# sudo firewall-cmd --add-port=5432/tcp --permanent
FirewallD is not running
[root@localhost ~]# vim /var/lib/pgsql/14/data/postgresql.conf
[root@localhost ~]# vim /var/lib/pgsql/14/data/postgresql.conf
[root@localhost ~]# vim /var/lib/pgsql/14/data/pg_hba.conf
[root@localhost ~]# sudo systemctl restart postgresql-14
[root@localhost ~]# su - postgres
Last login: Fri Oct 29 15:24:17 CST 2021 on pts/0
-bash-4.2$ psql
psql (14.0)
Type "help" for help.

postgres=# alter user postgres with password 'postgres';
ALTER ROLE
postgres=# exit
-bash-4.2$ exit
logout
[root@localhost ~]# sudo su - postgresql
su: user postgresql does not exist
[root@localhost ~]# sudo su - postgres
Last login: Fri Oct 29 15:30:57 CST 2021 on pts/0
-bash-4.2$ pwd
/var/lib/pgsql
-bash-4.2$
```

图 8: pg-run

5 实验结果分析

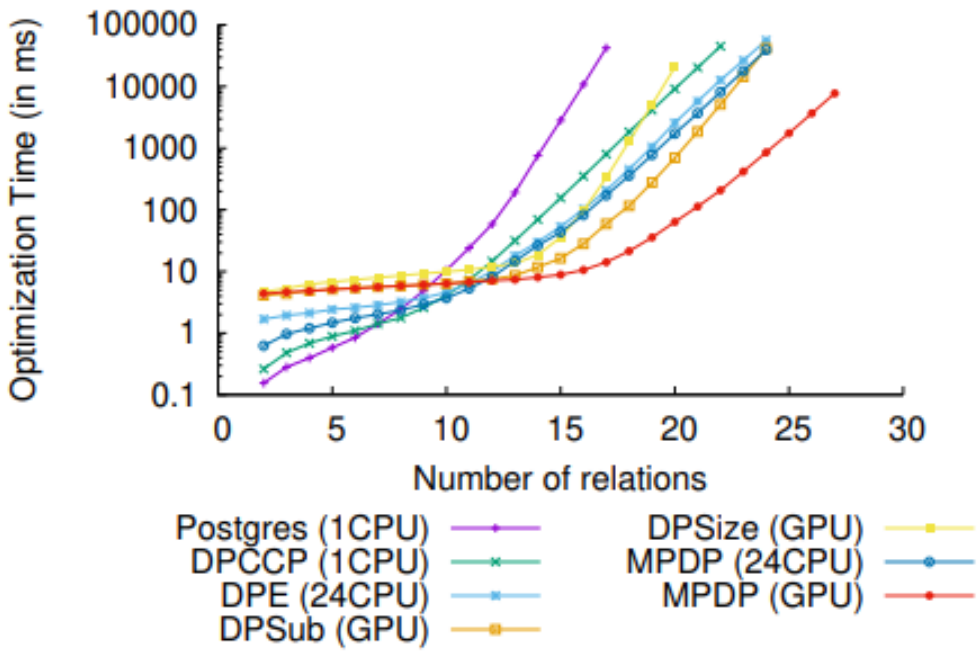

图 9: results

从图九中可以看到 MPDP 算法在 1s 内可以处理的关系数多达 25，因此将使用启发式算法的表界限提升到 25 以上

6 总结与展望

本次复现完成了对 MPDP 算法的 C++ 语言的复现，并将其整合到 postgreSQL 之中，该算法是一种对于具有大量连接的查询进行连接顺序优化的技术。MPDP 算法能够并行运行，同时显著减少了搜索空间，也可以在 GPU 上高效实现。复现实验表明，在精确场景的情况下，MPDP 算法在查询优化时间方面显著优于其他最先进的技术。进一步的工作可以着眼于将 MPDP 算法应用于云计算领域，利用更加广泛的资源来进行查询优化。比如图分析和大数据分析等。