# 分布式与并行数据库实验报告

组长：江流洋 组员：孙伟浩，李婧瑶

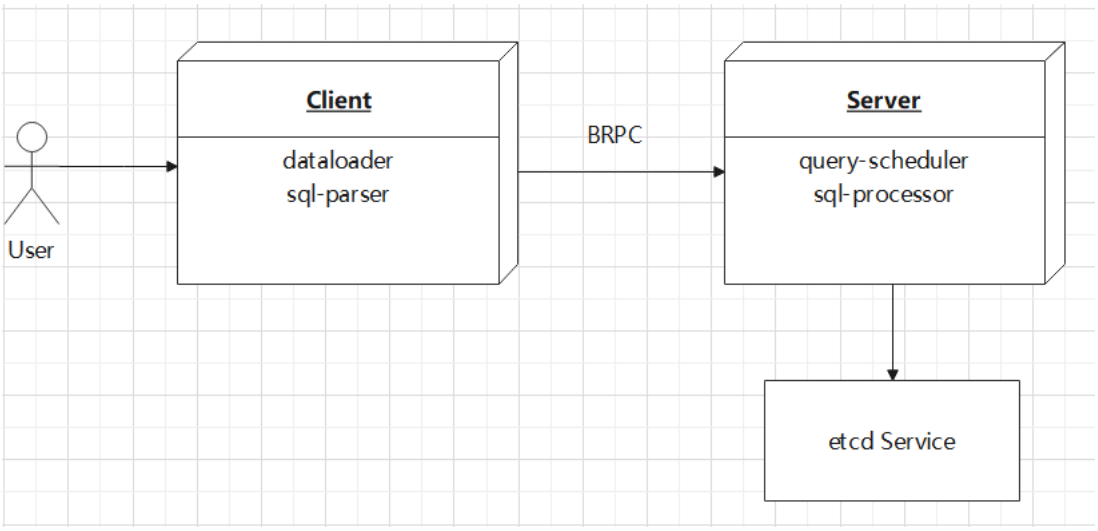
2021 年 1 月 15 日

## 一、项目说明

**操作系统：CentOS Linux release 7.4.1708 (Core)**
**GCC 版本：gcc version 7.3.1 20180303 (Red Hat 7.3.1-5) (GCC)**
**Cmake 版本：cmake version 3.14.5**
**Mysql 版本：mysql Ver 14.14 Distrib 5.7.29, for Linux (x86_64) using EditLine wrapper**
**Etcd 版本：3.4**

**项目地址：**https://github.com/jaingmengmeng/DDBS

整体项目架构图如下：

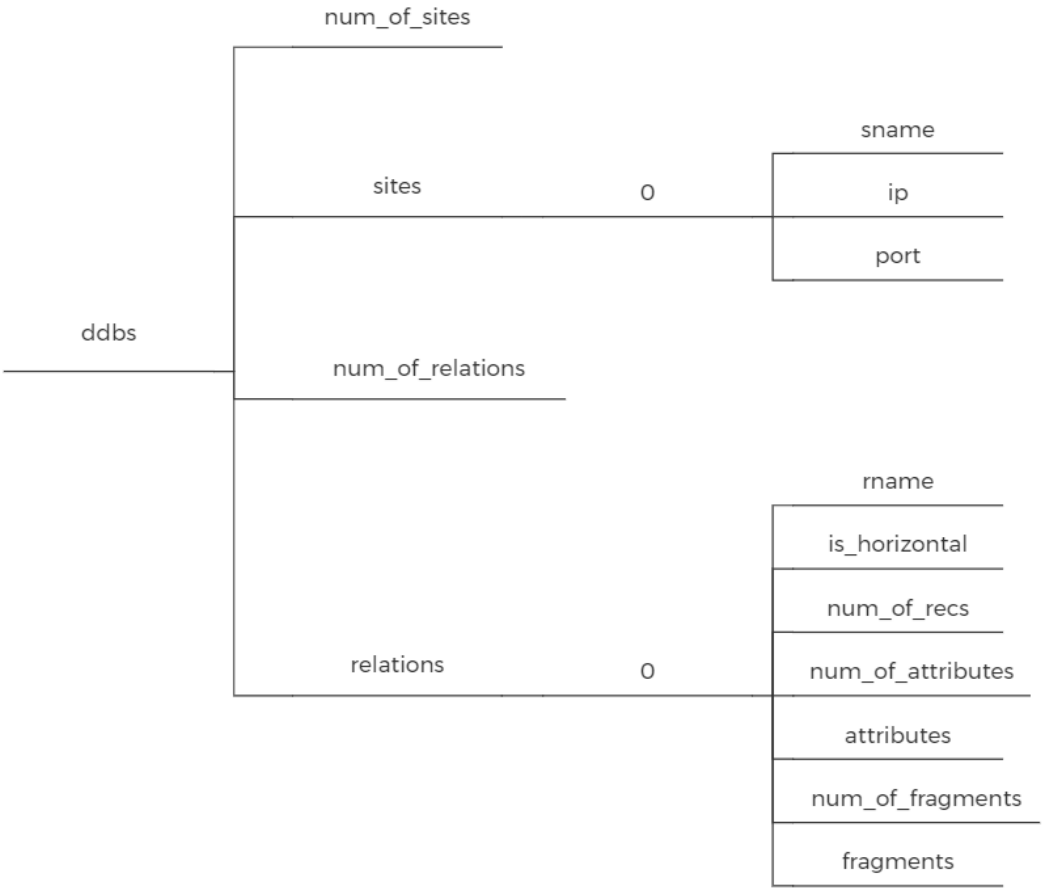



整个系统的架构如图所示，主要分为 Client、Sercer 两部分。Client 获取到用户在命令行输入的 sql 语句，解释（sql-parser）之后并判断是否有效（dataloader），将操作指令发送给 Server，随后 Server 根据数据库的列表属性信息对进行查询优化（sql-processor），生成查询树，之后对分布式数据库进行增删改查（query-scheduler）等操作。其中，各站点之间的数据库的元信息以及查询过程中生成的临时数据结构都是通过 etcd 服务来同步的。数据库选择的是 mysql。

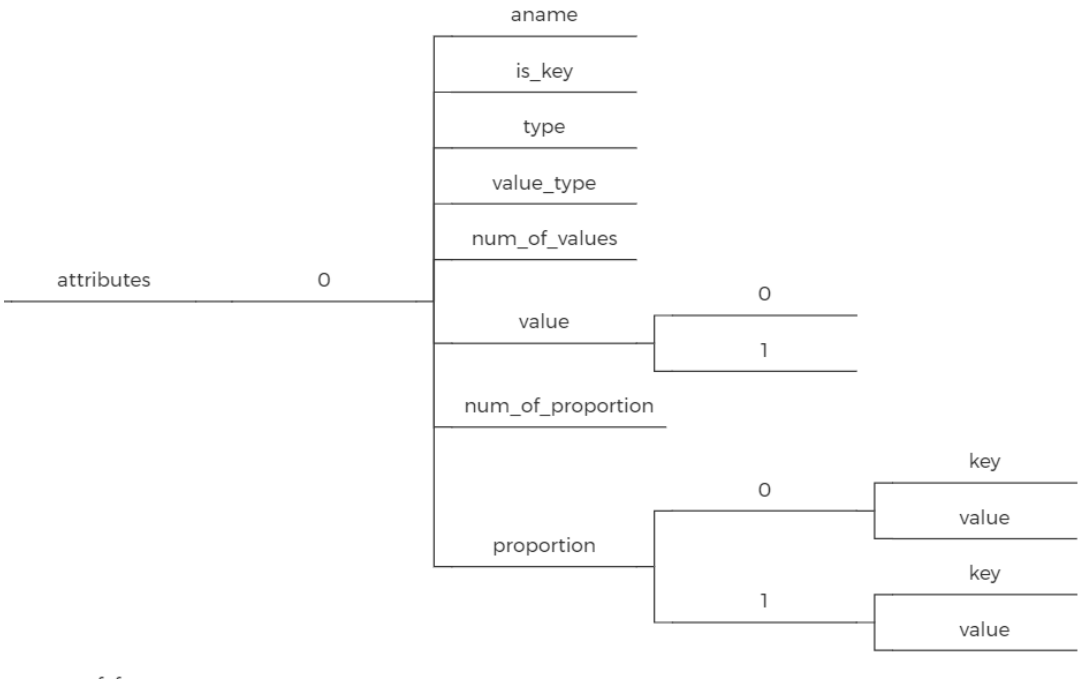
# 二、模块

## 1、dataloader

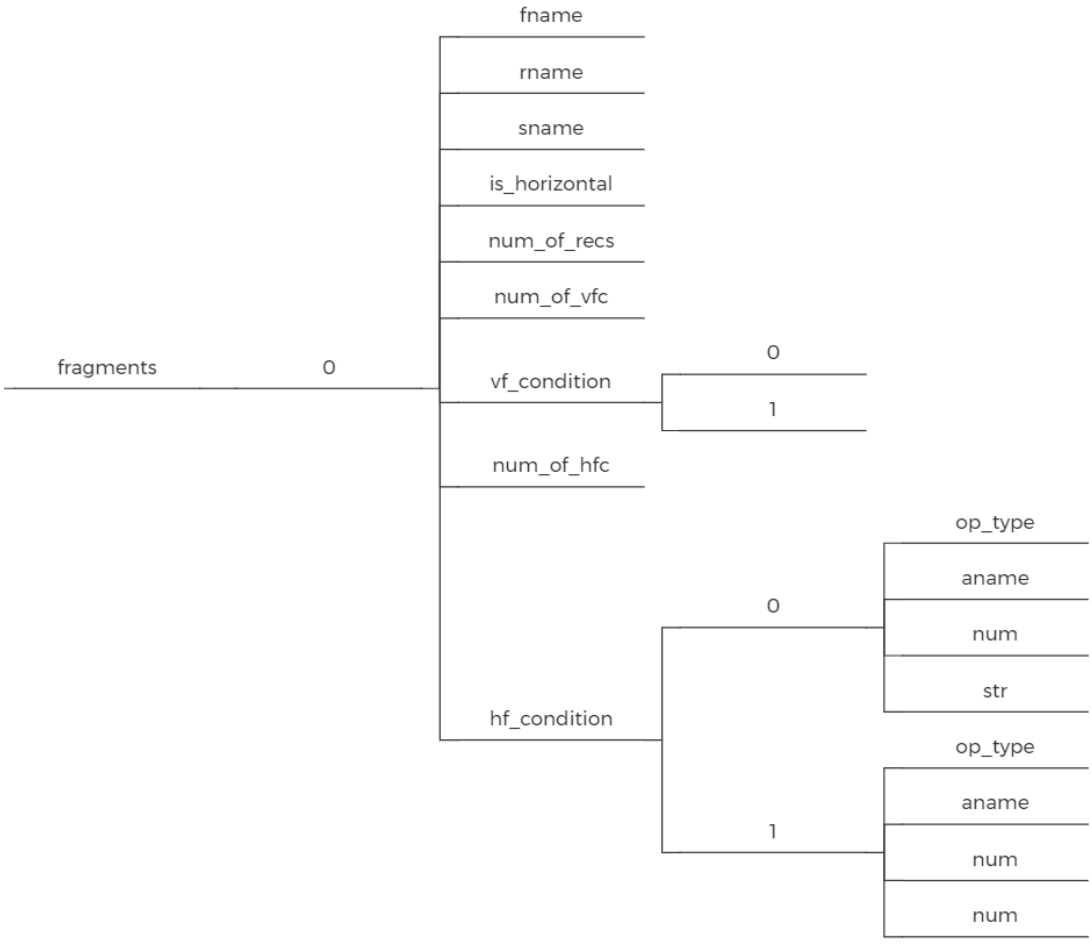
Dataloader 的主要作用是通过服务端接口从 etcd 服务获取到数据库表的元信息。关于 etcd 中元信息的结构在设计时也是根据基础的数据结构来确定的，具体结构图如下：




站点的信息保存在 sites 目录下，包括站点名称 sname，站点 ip 地址，站点端口 port。数据表的信息保存在 relations 目录下，包含了表名 rname，水平划分标志位 is_horizontal，数据的数目 num_of_recs，属性的数量 num_of_attributes，分片的数量 num_of_fragments，各个属性的信息记录在 attributes 目录下，其结构如下（与数据结构相对应）：




各个分片的信息记录在 fragments 目录下，其结构图如下（与数据结构相对应）：



# 2、sql-parser

Sql-parser 的主要作用是在获取到用户的命令行输入之后，判断 sql 语句的类型，并根据 dataloader 记录的数据库表的元信息来判断该 sql 语句是否有效。目前项目支持的 sql 语句包括:

- ✓ **show tables——打印所有表的列信息及分配信息**
- ✓ **show sites——打印所有的站点信息**
- ✓ **define site——新建一个站点**
- ✓ **create table——新建一个关系表**
- ✓ **fragment——新建一个分片**
- ✓ **allocate——将分片分配到某个站点**
- ✓ **load data from file——从文件中导入数据到某一关系表**
- ✓ **select——查询关系表**
- ✓ **delete——从关系表中删除某一条数据**
- ✓ **insert——向关系表中插入某一条数据**

这里 sql-parser 的实现，对于 select、delete、insert 等语句，是调用了一个外部 cpp 库，sql-parser（https://github.com/hyrise/sql-parser），能够根据字符串，生成实例化的 sql-statement 对象。对于其他语句，都是通过正则表达式匹配来解析的，使用的是 boost 的 regex 库函数。用户输入时不区分大小写，且如果有输入错误，会有错误提示。对于输入的格式，可以参考项目目录下 files/test.sql。

# 3、sql-processor

Sql-processor 的主要作用是对于 select 的 sql 查询语句，根据 etcd 中记录的数据库表的元信息，进行查询优化，生成查询计划。

**1.数据结构**

主要输入:

```
SelectStatement sql //sql 语句结构体
```

```
vector<Relation> relations //表结构
```

主要输出:

```
map<string,string>& output_for_etcd1 //最终生成的查询树
```

其中涉及到的类主体结构设计如下:

(1)sql 语句结构:

```
class SelectStatement {

public:

    std::vector<std::string> from;

    std::vector<std::string> select;

    std::vector<Predicate> where;

    ...
```

```
}
```

(2)表结构:

```
class Relation {  //表结构，包含表名、属性列信息、分片信息等
public:
    std::string rname;
    bool is_horizontal;
    int num_of_recs;
    std::vector<Attribute> attributes;
    std::vector<Fragment> frags;
    ...
}
class Attribute { //属性列信息，包含属性的分布信息
public:
    std::string aname; // Name of attribute e.g.customer_rank
    bool is_key;
    int type; //1:Integer 2:String
    int value_type;
    //1:ID:no duplicate e.g.1-100
    //2:U:Uniform Distribution e.g.U[1 100]
    //3:PN:Positive Gaussian Distribution e.g.X~N(3,2)and X>0
    //4:Discrete:give corresponding key and proportion.
    std::vector<double> value; //Value for ValueType 1,2,3
    std::map<std::string, double> proportion; //Value for ValueType 4.
    ...
}
class Fragment {  //分片信息
public:
    std::string rname; // relation name
    std::string fname; // e.g.cus1 cus2 ord1 ord2 ord3 ord4 pub1 ...
    std::string sname; // site name
    bool is_horizontal;
    int num_of_recs;
    std::vector<std::string> vf_condition; //e.g. customer_id customer_name
    std::vector<Predicate> hf_condition;
    ...
}
```

(3)树结构：生成树过程中采用的结构如下，最终输出给 etcd 时按照一定格式输出这颗树为

map<string,string>类型

```
class Tree{
public:
    vector<TreeNode> tn;
    int n; //Total Number of Nodes in Tree
};
class TreeNode{
public:
    int node_type; //1:Fragments 2:Join 3:Union
    string rname;
    string fname; //if node_type=1
    vector<string> join;  //if node_type=2 e.g.customer_id orders_cid
    bool has_selection;
    vector<Predicate> selection; //if has_selection=true
    bool has_projection;
    vector<string> projection; //if has_projection=true e.g. book_copies book_id
    string sname; //Name of executed site
    int num_of_recs;
    vector<TNAttribute> tn_attributes;
    int tn_size;
    string layer; // e.g._1_1_2, _1_2_4
    int  parent;  //-1 if no parent
    vector<int> child; //child[0]=-1 if no child
    ...
};
class TNAttribute{
public:
    string aname;
    int type; //1:Integer 2:String
    int value_type;
    //1:ID:no duplicate e.g.1-100
    //2:U:Uniform Distribution e.g.U[1 100]
    //3:PN:Positive Gaussian Distribution e.g.X~N(3,2)and X>0
    //4:Discrete:give corresponding key and proportion.
    vector<double> value; //Value for ValueType 1,2,3
    map<string, double> proportion; //Value for ValueType 4.
    bool has_restriction;
```

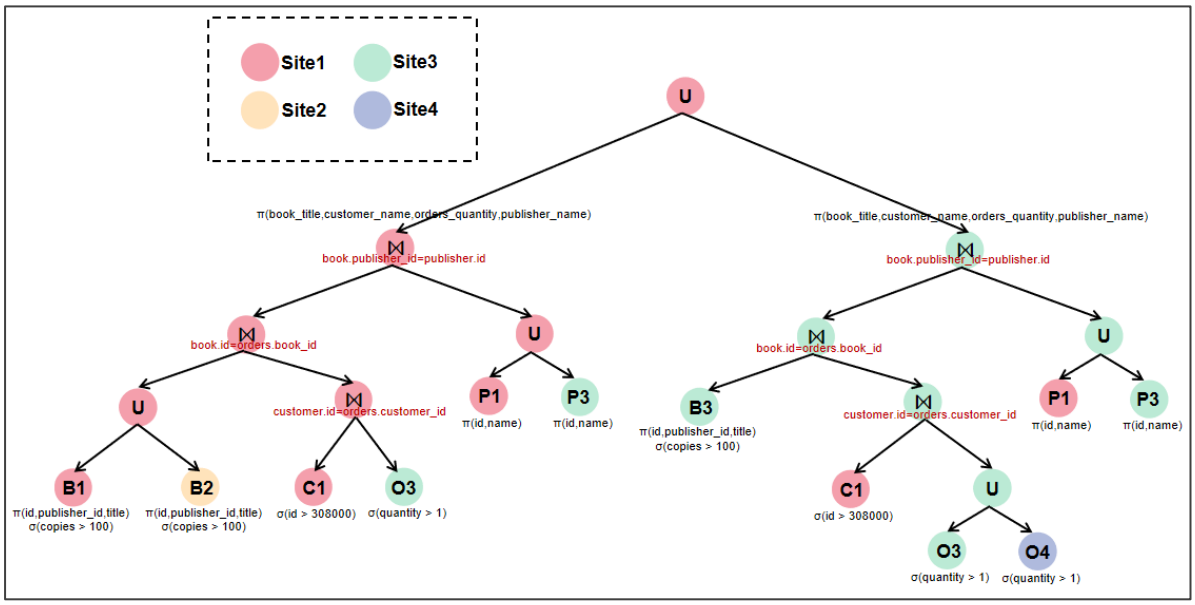
```
    vector<double> restriction_range; //restriction for ValueType 1,2,3
    string restriction;  //restriction for ValueType 4
};
```

**2.算法思路**

下面以一个列子为例，详述查询树的生成算法：设在站点 1 上发出如下查询请求

| |
|---|
| select Customer.name, Book.title, Publisher.name, Orders.quantity |
| from Customer, Book, Publisher, Orders |
| where Customer.id=Orders.customer_id |
| and Book.id=Orders.book_id |
| and Book.publisher_id=Publisher.id |
| and Customer.id>308000 |
| and Book.copies>100 |
| and Orders.quantity>1 |
| and Publisher.nation='PRC' |

生成的查询树如下：注意其中的节点类型只有 3 类：Union, Join, Fragment.而选择操作和投影操作由于不能改变所处站点位置，不单列为节点，而是作为节点的属性，在下图中以黑色字段写出。连接的条件以红色字段写出。Fragment 只能作为叶子节点，如 B2 代表 Book 表在 Site2 上的分片。以不同的节点颜色代表不同的执行站点（或初始分片所处站点）。下面从这个例子入手描述查询树的生成过程。




**Step1. 根据 sql 语句 from 中的内容找到所有参与的表**

e.g. 在此例中，为 Customer, Book, Publisher, Orders

**Step2.根据 sql 语句找到所有参与的属性，包括 select 和 where 中涉及到的属性**

e.g.在此例中，为 Customer.name, Customer.id, Book.id, Book.copies, Book.title, Book.publisher_id, Publisher.name, Publisher.id, Publisher.nation, Orders.quantity, Orders.customer_id, Orders.book_id

**Step3.根据第 1 步中得到的参与表和第 2 步中得到的参与属性，剪枝垂直分片的参与表：（1）如果只有 key 属性参与，则保留任一垂直分片即可；（2）如果有非 key 属性参与，则保留所有含有参与非 key 属性的表**

e.g.此例中垂直分片的参与表为 customer,参与的属性为 customer.id(key)和 customer.name(非 key)，故属于情况（2），仅保留含有 customer.name 的表，即 C1。C2 被剪掉。

**Step4.生成叶子节点并完成选择和投影。注意仅在投影属性少于现有分片上属性时触发投影操作(置 has_projection=true)：（1）水平分片：检测水平分片条件是否与 where 条件冲突，在不冲突的情况下生成叶子节点（否则剪枝），并根据 where 条件完成选择，根据 select 中涉及的属性和 where 非选择条件中涉及的属性完成投影；（2）垂直分片：为第 3 步剪枝后剩余的垂直分片生成叶子节点，并根据 where 条件完成选择，根据 select 中涉及的属性和 where 非选择条件中涉及的属性完成投影。注意如果有大于等于 2 个剩余垂直分片，投影除了上述属性外，还应保留 key 属性以便后续垂直分片的连接。**

e.g.本例中，根据 where 中的条件 Publisher.nation='PRC'剪枝 P2,P4。生成叶子节点并加上选择、投影条件如下:

P1(π(id,name)), P3(π(id,name)),

B1(σ(copies>100),π(id,publisher_id,title)),

B2(σ(copies>100),π(id,publisher_id,title)),

B3(σ(copies>100),π(id,publisher_id,title)),

O1(σ(quantity>1)),O2(σ(quantity>1)),O3(σ(quantity>1)),O4(σ(quantity>1)),

C1(σ(id>308000))。

其中，Orders 和 Customer 表，由于投影属性与现有属性相同，故不触发投影操作(置 has_projection=false)

注意：在生成的树节点中还需要维护所处站点 sname，记录行数 num_of_recs，属性列及其上约束（由 where 条件和水平分片条件决定）tn_attributes，节点大小（=记录行数*一行大小）tn_size，以及父母孩子等信息。其中记录行数又由所属表总行数和属性列上的约束及属性分布特征估算。详细描述见 TreeNode 数据结构。

**Step5.根据 where 中的 join 条件剪枝第 4 步中生成的叶子节点：对于 where 中的每一个 join 条件，考察涉及到的两个表各分片间在连接属性上是否存在冲突，完成连接的剪枝。**

e.g.本例中，首先考虑 Customer(C1)与 Orders(O1,O2,O3,O4)的连接，由于 C1 在连接属性 customer.id 上具有约束 customer_id>308000,而 O1,O2,O3,O4 分别在连接属性上具有约束 customer_id < 307000,customer_id < 307000,customer_id >= 307000,customer_id >= 307000，故剪枝 O1,O2；类似的，剪枝 B1*O4 和 B2*O4，最终得到剩余节点为:

C1,O3,O4,B1,B2,B3,P1,P3 它们间的连接关系描述如下：（1 代表 join,0 为不 join）

C*O

cus1_ord3 ： 1

cus1_ord4 ： 1

O*B

boo1_ord3 ： 1

boo2_ord3 ： 1

boo3_ord3 ： 1

boo1_ord4 ： 0

boo2_ord4 ： 0

boo3_ord4 ： 1

B*P

boo1_pub1 ： 1

boo1_pub3 ： 1

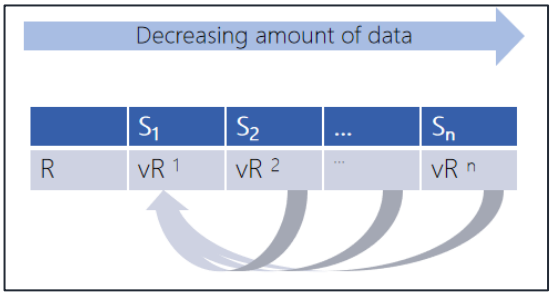
boo2_pub1 ： 1

boo2_pub3 ： 1

boo3_pub1 ： 1

boo3_pub3 ： 1

**Step6.连接垂直分片。**若剩余垂直分片数量大于等于 2，生成连接节点，连接垂直分片。传输优化：选定 tn_size 最大的分片，其它分片均传到它所在的站点上，以节省传输开销




e.g.此例中，由于剩余垂直分片仅 C1,故跳过此步骤

**Step7.完成传输。**参考下图考虑传输优化

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $R_1$ | $R_1^1$ | $R_1^2$ | $R_1^3$ | $R_1^4$ |
| $R_2$ | $R_2^1$ | $R_2^2$ | $R_2^3$ | $R_2^4$ |
| $R_3$ | $R_3^1$ | $R_3^2$ | $R_3^3$ | $R_3^4$ |
| $R_4$ | $R_4^1$ | $R_4^2$ | $R_4^3$ | $R_4^4$ |

Decreasing amount of data

e.g. p=k=3
Notice that when k=1, it is similar to vertical situation in broadcasting network.

Step1. Find Rp as the largest Ri

Step2.Decide the number of processing sites: k
(A site is selected as a processing site only if the amount of data it must receive is smaller than it would have to send if it were not a processing site.)

If $\sum_{i \neq p}(size(R_i) - size(R_i^1)) > size(R_p^1)$
then
k=1 (that's the best we can achieve)
else
k is the largest j, such that
$\sum_{i \neq p}(size(R_i) - size(R_i^j)) \leq size(R_p^j)$

e.g.本例中，首先选出 tn_size 和最大的关系，为 Book(即 B1,B2,B3 的 tn_size 和最大)。然后对于 Book 分片所在的每一个站点 S1,S2,S3，决定它是否为处理站点：

（1）若 Si 作为处理站点的开销<Si 不作处理站点的开销，则设 Si 为处理站点

（2）若 Si 作为处理站点的开销>Si 不作处理站点的开销，则 Si 不作处理站点

若 Si 均不作处理站点，则取|Si 作为处理站点的开销-Si 不作处理站点的开销|最小的站点为唯一处理站点。

其中不作处理站点的开销为 Size(Bi)，即将 Bi 发到其它站点的传输开销

作处理站点的开销为所有需要发往 Bi 的分片 Size 和，注意哪些分片需要发往 Bi 由第 5 步中得到的连接关系确定，借助递归算法找出

本例中，由于：

Size(P3)+Size(O3)<Size(B1)

Size(P1)+Size(P3)+Size(O3)+Size(C1)>Size(B2)

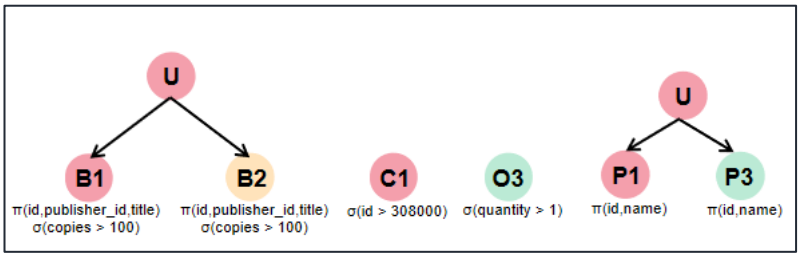
Size(P1)+Size(O4)+Size(C1)<Size(B3)

故选择 S1,S3 作为处理站点

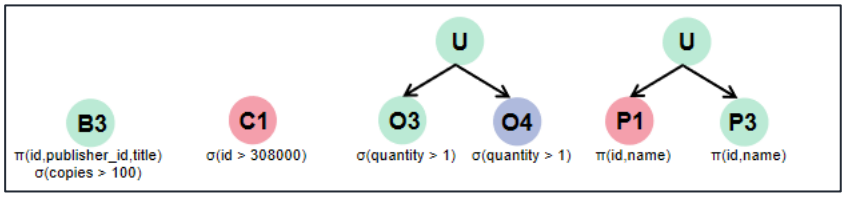
将最大表 Book 在非处理站点上的部分(B2)均发至第一个处理站点(S1)，并向处理站点上传输与处理站点上 Book 分片有 join 关系的各分片。本例中向 S1 传输 B2,P3,O3 向 S3 传输 P1,O4,C1。最终 S1 上有 B1,B2,P1,P3,O3,C1，S3 上有 B3,P1,P3,O3,O4,C1。

**Step8.各处理站点上完成同一表水平分片的 Union**

e.g.本例中，S1 上 B1 与 B2 完成 Union，P1 与 P3 完成 Union：



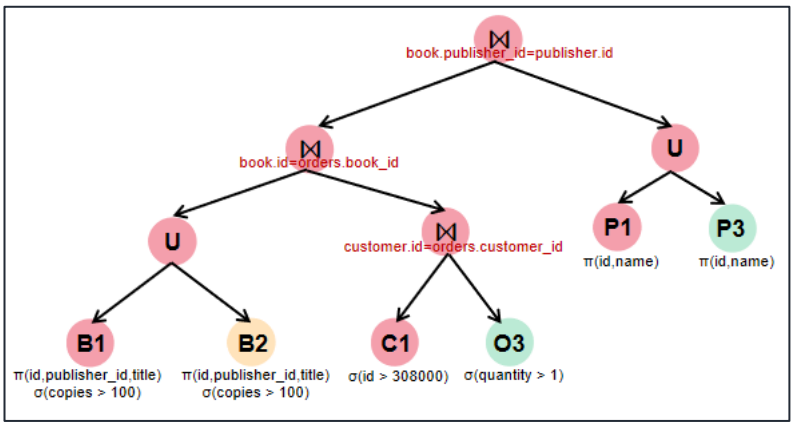

S3 上 P1 与 P3 完成 Union,O3 与 O4 完成 Union：



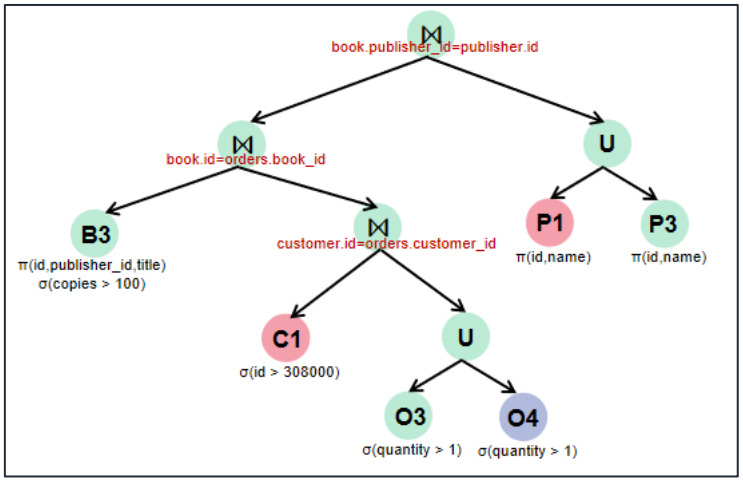

**Step9. 按照 where 中 join predicate 的顺序在各处理站点上两两 join 各表**

e. g. 本例中 join 顺序是 C*O, O*B, B*P

S1 上:




S3 上:




**Step10. 各站点子树根节点投影到 select 最终输出的属性上, 并传输所得数据到发出请求的站点上，作 Union。**

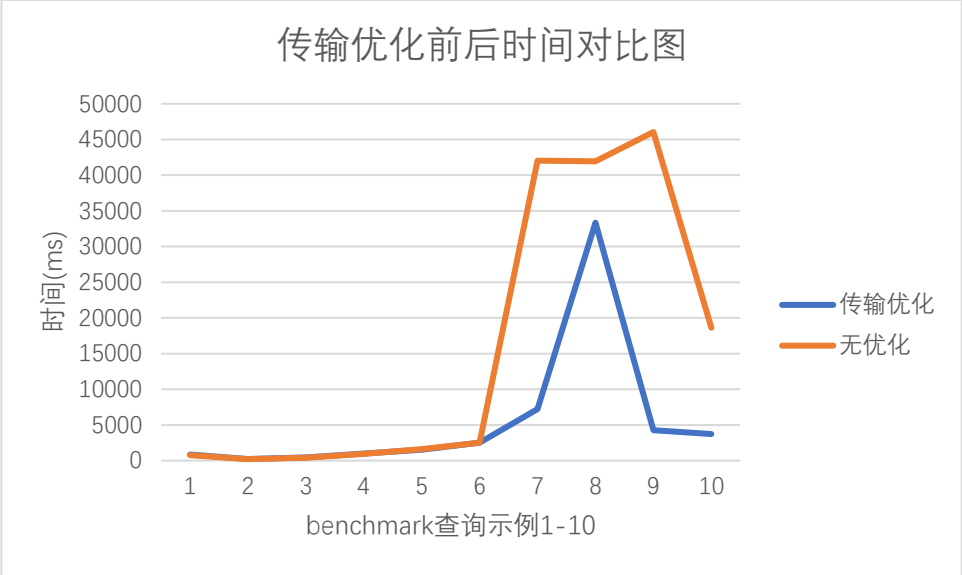
e. g. 本例中 S1 及 S3 上的子树根节点均投影到 Customer.name, Book.title, Publisher.name, Orders.quantity 上，并最终 Union 到发出请求的站点即 S1 上。得到最终的查询树。

**3.传输优化效果展示**

注意：对于无传输优化的对比组，置

```
#define use_transport_optimization 0 //0: not use, 1:use
```

则取第一个分片所属的表为 max_r，所处的站点为唯一处理站点，其它所有分片都传输到这一站点进行处理。




# 4、sql-scheduler

Sql-scheduler 的主要作用是生成查询计划之后，将查询计划同步到 etcd，之后在每个站点执行查询操作。对于一些连接操作，需要传输的数据会保存在站点的临时表上，连接操作完成之后，临时表将会被删除。

**4.1 依赖**

4.1.1 brpc + protobuf

采用 brpc+ protobuf 来进行网络通信. 相比于常用的 gRPC 来说, brpc 优势主要在于以下 3 个方面:

(1) 性能好. 来源于 brpc 文档中的测试结果.

(2) 兼容性好. brpc 的 client 可以和包括 brpc server 在内的多种 rpc server 通信，比如 grpc server, thrift server, http server 等，本实验中需要通过 http 和 etcd server 通信，只需使用 brpc client，而无需使用原生 socket.

(3) ParallelChannel. 实际业务场景经常存在复杂的 rpc 请求/响应组合需求，虽然可以通过多线程+全局计数来实现一定的组合，但不够灵活并且无法适用于更深更复杂的组合场景. brpc 提供了 ParallelChannel，可以实现复杂的 Channel 组合，用户只需定义好组合逻辑即可. 具体来说，通过 RequestCallMapper 将父 channel 的请求映射为各个子 rpc 请求，通过 ResponseMerger 将子节点相应合并起来作为父 rpc 请求的响应，其中具体的映射和合并逻辑可自己灵活定义，从而实现复杂的 rpc 组合逻辑. 本实验中存在组合 rpc 的需求，因此 brpc 是很好的选择.

(4) Native Latency/Communication-Cost Measurement Support. brpc 提供原生的 rpc 延迟和传输成本的 metric 数据，无需自己手动测量.

4.1.2 mysql

采用 mysql-connector-c/c++-1.3 和 mysql5.6 交互

4.1.3 etcd

采用 etcd-3.4，etcd v3 版本相比之前版本有了巨大变动，其存储模型变为完全的扁

平模型，而不是之前的树形结构，因此已经基本变成一个完全的分布式 kv 存储.

其 API 上变动主要有两点:

(1) c++无原生 client，需要使用 etcd v3 的 grpc server 暴露的 http json service. 请求的 key 需要经过 base64 编码，响应的 value 也是 base64 编码的，每次请求都要进行编码/解码过程，比较麻烦.

(2) 目录逻辑实现. 由于 v3 并不存在目录逻辑，目录逻辑的实现需要转化为前缀匹配查询，但 etcd 并未提供原生的前缀查询接口，而只提供了 range 查询，因此对于前缀为 prefix 的查询，需要以 prefix 的 base64 编码为 range 的 begin，并且找出 prefix 经过 base64 编码之后的字节序列的后一个相邻的字节序列作为 range 的 end 进行实现.

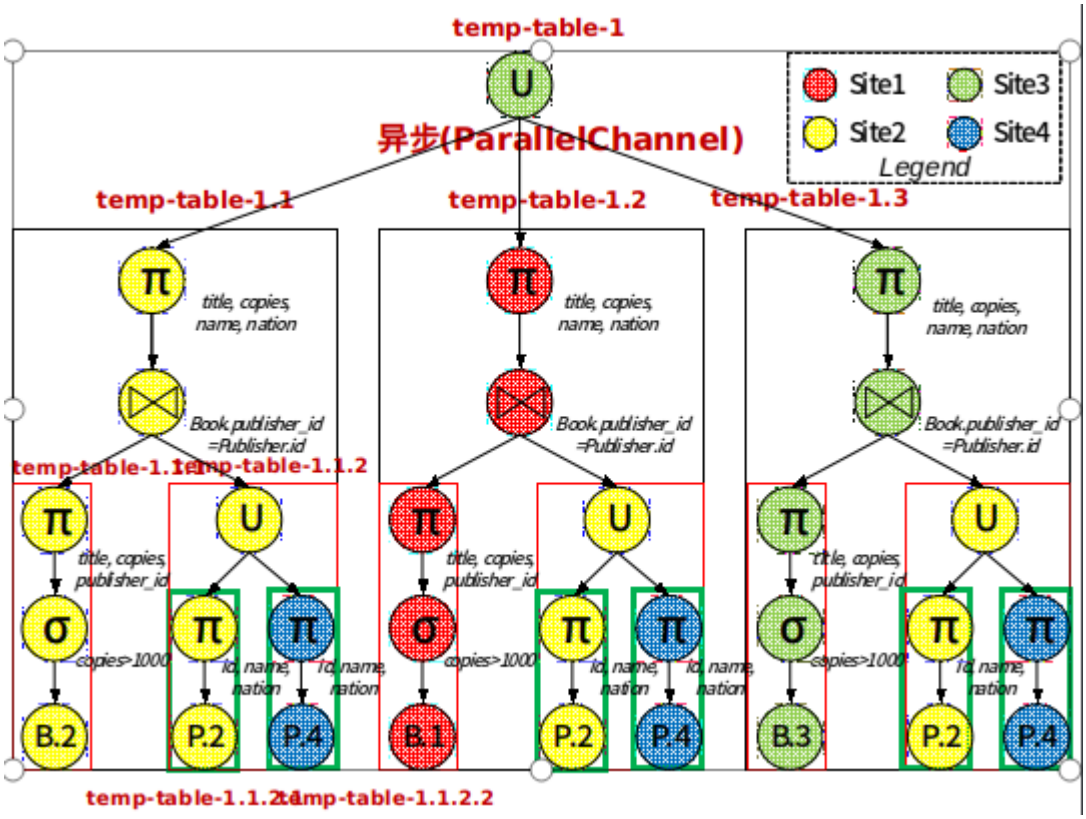
### 4.2 查询树实现思路

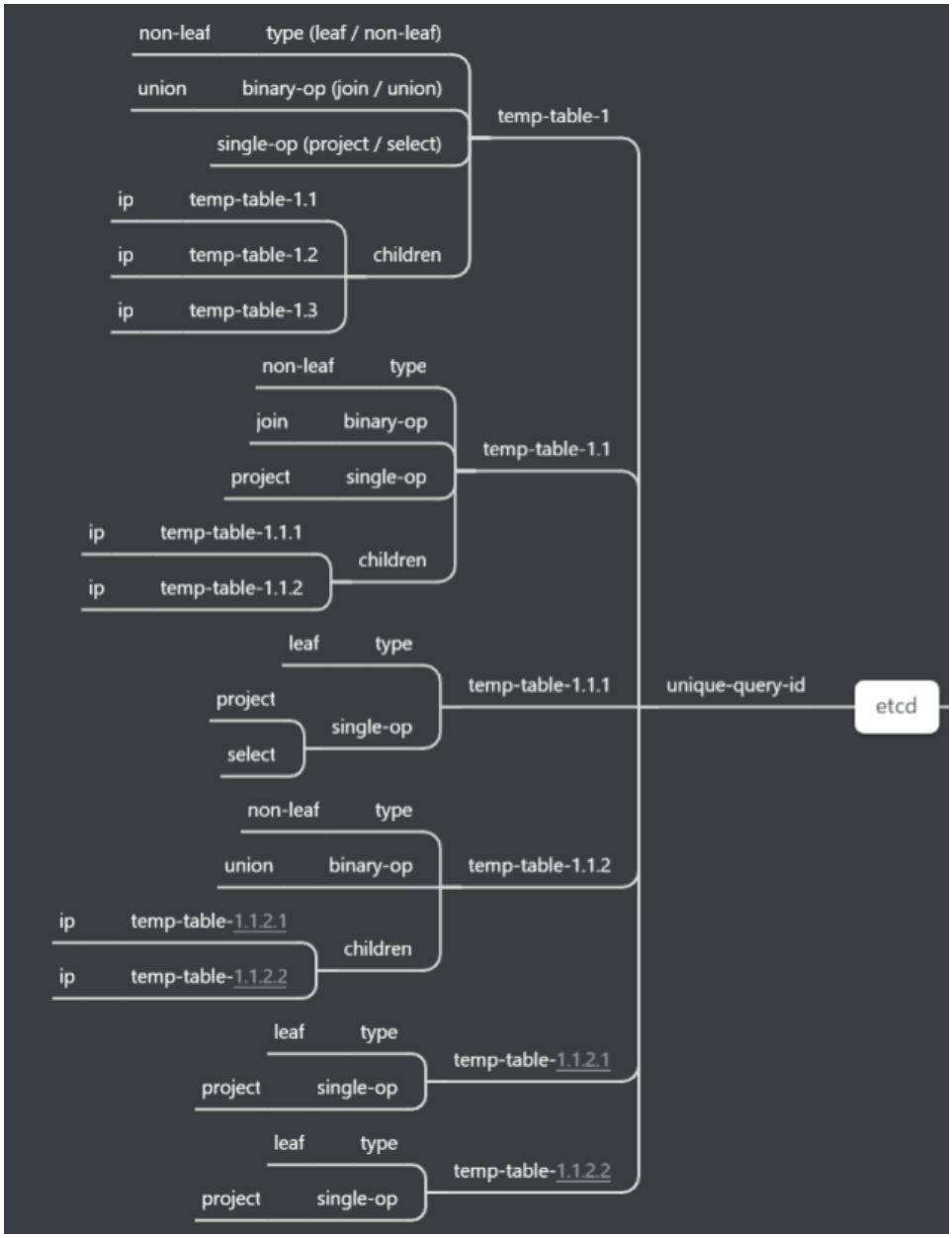
整体上采用 pull 模型实现.

具体来说，如图所示，查询树本身的执行可以看做一个递归的 rpc 调用过程(pull 模型)，但如果用纯粹的 pull 模型实现，则每次递归调用，都要传递子树作为参数，这会导致查询树生成和执行两部分内容的数据结构耦合在一起. 因此采用将查询树的逻辑写入 etcd，形成一个扁平结构，每次递归调用时，只需传递需要的子节点名称即可，而 server 端只需要通过 etcd 查询这个节点对应的信息即可.

具体来说，共存在非叶节点和叶子节点两种节点，前者的主要逻辑是组合(join/union)多个子节点，后者则是物理存储的表，两种节点都会有额外的 project 和 select 属性(非叶节点可能不应该有).

由于逻辑上存在 rpc 组合需求，采用上述的 brpc 的 ParallelChannel 实现即可.






### 4.3 具体实现

4.3.1 定义 Service

brpc 采用 proto 文件来定义服务器端对外暴露的 service，本实验中共定义了如下 5 个 service:

(1) RequestTable. 获取某个临时表的数据

(2) LoadTable. 导入表，供初始数据导入使用.

(3) DeleteTable. 删除表，主要用于删除临时表.

(4) ExecuteNonQuerySQL. 对某个站点的 mysql 执行非查询的 sql 语句. 用于数据插入等

(5) ExecuteQuerySQL. 对某个站点的 mysql 执行查询 sql 语句.

4.3.2 实现 service

4.3.2.1 RequestTable 实现.

(1) 获取 client 请求的临时表名，并从 etcd 读取如下信息.

```
struct temp_table {
    int ret_code{};
    std::string type;
    std::string project_expr;
    std::string select_expr;
    bool is_union{};
    std::string join_expr;
    std::unordered_map<std::string, std::string> children;
};
```

(2) 根据 type 判断节点类型，如果是叶子节点，则执行本地执行查询 sql 语句并返回结果数据

(3) 如果是非叶节点，则利用 ParallelChannel 组合对多个子节点的请求. 如果是对子节点进行 union，则在 ResponseMerger 直接采用 Protobuf 默认的 merge 组合成一个更大的 response，这样的优势是无需创建临时表再执行 union 操作，问题是无法自动去重，但由于本场景下各子节点的数据不会存在冗余，因此是可行的;

如果是需要进行 join 操作，则需要在 ResponseMerger 中将每个子节点的 response 存为临时表，并对需要 join 的属性建立索引，提高 join 速度.

在当前 rpc 请求完成之后，将每个 channel 的 latency 和 communication-cost metric 信息存入 etcd，供最后统计使用；此外，复用上述的 ParallelChannel 删除所有创建的子临时表.

4.3.2.2 其余 Service 实现.

其余 4 个 service 实现较为简单，执行相应的 sql 语句并返回结果即可，不再赘述.

## 4.4 client 和 server 的交互实现

为减少网络部分和其余部分的耦合，将所有需要暴露的网络通信接口创建为一个共享库，以供其余部分使用. 所有暴露的接口如下:

```
// etcd
int write_kv_to_etcd(const std::string& key, const std::string& value);
int write_map_to_etcd(const std::map<std::string, std::string>& mp);
std::string read_from_etcd_by_key(const std::string& key);
std::unordered_map<std::string, std::string> read_from_etcd_by_prefix(const std::string& prefix);
int delete_from_etcd_by_key(const std::string& key);
int delete_from_etcd_by_prefix(const std::string& prefix);
std::map<std::string, std::string> read_map_from_etcd(const std::vector<std::string>& keys);

// rpc
int load_table(const std::string& host, const std::string& table_name, const std::string& attr_meta, const std::vector<std::string>& attr_values);
std::vector<std::string> request_table(const std::string& temp_table_name);
std::map<std::string, std::string> get_request_statistics(const std::vector<std::string>& temp_table_names);
std::string execute_non_query_sql(const std::string& ip, const std::string& sql);
std::vector<std::string> execute_query_sql(const std::string& ip, const std::string& sql);
```