分布与并行数据库实验报告

林艺辉　陈明　王佳悦　曾翔

完成时间：2019 年 1 月 3 日

分布与并行数据库实验报告 1
任务描述 3
1.2 系统需求 3
1.3 运行环境 3
1.4 开发环境 3
3 模块实现 3
3.1 执行流程 3
3.1.1 DEFINE SITE 3
3.1.2 FRAG 3
3.1.3 CREATE/DELETE/INSERT 4
CREATE 4
DELETE 4
INSERT 4
3.1.4 LOAD 4
3.2 节点通信 4
3.3 SQL 解析 5
3.3.1.结构体定义： 5
3.3.2.正则表达式解析： 8
3.4 元数据设计 10
3.4.1.ETCD 介绍： 10
3.4.2.GDD 结构设计： 10
3.5 查询树及查询计划 11
3.5.1.查询树的生成及优化： 11
3.5.2.生成查询计划： 20
4 任务分工及小结 20
4.1 任务分工 20
4.2 王佳悦小结 20
4.3 林艺辉小结 21
4.4 陈明小结 22
4.5 曾翔的小结 22

任务描述

本次项目使用 mysql 作为底层数据库, 将数据按照规则分布在不同的站点上, 通过网络将不同的数据库连接起来，同时向用户隐藏复杂的底层细节，而向外提供一个类似 mysql 的简单的接口, 从而达到提高数据库存储容量和提高查询速度等目标。

1.2 系统需求

1. 支持站点的定义和展示
2. 支持数据分片规则的划分和分布，包括垂直分片和水平分片
3. 支持元数据的输入和修改以及展示以及元数据在不同机器上的同步
4. 支持表的创建
5. 支持数据的批量导入，并且根据分片规则，将数据分布在不同的站点上
6. 支持数据的插入和删除
7. 支持查询语句和表的 join 操作等。

1.3 运行环境

我们的分布式数据库采用 C++语言开发，运行在 Ubuntu 16.04 上，底层的数据库使用 MySQL5.7，并且使用了 Etcd 管理元数据信息。采用了 P2P 的架构，每个 Server 都可以部署在任意的机器上，并且每一个节点都可以作为客户端程序进行访问。我们的实验环境使用了三台机器，在 PC1，PC2 上分别部署了一个站点，在 PC3 上使用不同的端口部署了多个站点。

1.4 开发环境

在开发的过程中我们使用了 C++11, 同时 cmake 作为整个项目的管理工具, 通过编写 CMakelist 运行生成的 Makefile 文件编译部署。

3 模块实现

3.1 执行流程

3.1.1 define site

define site 语句可以控制站点的划分，包括设置每个站点的 IP，PORT，以及索要操作的数据库的名称, 并将站点的信息通过 ectd 的接口上传至 etcd 中保存。

3.1.2 frag

frag 语句用于数据库的分片，所要提供的信息包括水平划分还是垂直划分，

划分的表名、属性名、划分条件以及所要分布的站点的名称。Frag Parser 将分片的语句解析之后，调用 etcd 的接口，同样也将分片的信息上传至 etcd 中。

3.1.3 create/delete/insert

create

获取 SQL Parser 解析完了的 Create 结构体，将表的信息通过 etcd 的接口上传至 etcd 服务中保存和同步，并且根据表名获取对应的分片信息的集合，重新组合之后生成新的建表语句向对应的站点发送。

Delete

同样也是获取解析过后的结构体，由于一条数据只在一个站点上存放，因此直接将对应的 SQL 语句直接向所有的站点发送并执行。但是当所要删除的条件不是表的主键的时候，会出现多删的情况，因此在后续的版本中需要改进。对于垂直分片的数据，需要根据 where 后面的条件查询到主键的信息，重新生成 delete 语句向垂直分片所在的所有站点发送。

Insert

在获取解析后的节后，根据表的信息和分片的信息组合生成新的 SQL 语句向对应站的发送，执行插入的操作。而垂直分片需要生成一组 SQL 语句，向一组站点发送执行。

3.1.4 load

load 语句有两个步骤，首先需要创建一个对应的临时表，将本地的 tsv 格式的数据文件导入到临时表中，之后根据数据的分片信息筛选出不同站点上所要执行的数据，使用批量 insert 语句将数据库插入到不同的站点所对应的数据库中。

3.2 节点通信

节点之间使用 RPC（远程过程调用）进行通信，对于 insert，create 等语句在远程程序执行完成之后，可以直接返回一个 bool 值作为执行结果是否成功的响应，而 select 语句需要返回查询的结果。

首先，需要将所要执行的操作绑定在 RPC 的服务中，其中包括 execute，query 等，代码详见下图：

```
bool RPCExecute(string ip,int port,string db_name,string stmt){
    rpc::client client(ip,port);
    bool ok = client.call("localExecute",db_name,stmt).as<bool>();
    // cout <<  "localExecute\t" << stmt << "[" << ok << "]" << endl;
    return ok;
}

int RPCExecuteUpdate(string ip,int port,string db_name,string stmt){
    rpc::client client(ip,port);
    int cnt = client.call("localExecuteUpdate",db_name,stmt).as<int>();
    cout <<  "localExecuteUpdate\t" << stmt << "[" << cnt << "]" << endl;
    return cnt;
}
string RPCExecuteQuery(string ip,int port,string db_name,string stmt){
    rpc::client client(ip,port);
    string res = client.call("localExecuteQuery",db_name,stmt).as<string>();
    cout <<  "localExecuteQuery\t" << stmt << "\n";
    //cout << res << endl;
    return res;
}
bool RPCExecuteIndsertTable(string ip,int port,string db_name,string stmt){
    rpc::client client(ip,port);
    bool ok = client.call("localInsertTable",db_name,stmt).as<bool>();
    cout <<  "localExecuteTable\t" << stmt << "[" << ok << "]" << endl;
    return ok;
}
```

PRC 服务绑定

在每个客户端上需要启动 RPC Server 的服务，监听在指定的端口上，等待其他节点的调用，代码如下所示：

```
void startServer(int port){
    cout << "StartListening" << endl;
    rpc::server srv(port);
    srv.bind("localExecute",&localExecute);
    srv.bind("localExecuteUpdate",&localExecuteUpdate);
    srv.bind("localExecuteQuery",&localExecuteQuery);
    srv.bind("localInsertTable",&localInsertTable);
    srv.suppress_exceptions(true);
    srv.run();
    cout << "srv.run()" << endl;
}
```

PRC 服务监听

3.3 SQL 解析

3.3.1.结构体定义：

Select 语句： selectQuery 是记录查询语句的结构体，其中 sel_count, cond_count, from_count, join_count 为投影的个数，选择操作的的个数，查

询的表的个数以及 join 的个数。SelItem 是记录投影属性的结构体，包括属性的表名 table_name 和列名 col_name；FromItem 记录表的结构体，包括查询语句所涉及的所有表名 tb_name；Join 是记录 join 操作的结构体，包含 join 的两条属性的属性名 col_name1，col_name2 和属性属于哪个表 tb_name1，tb_name2；Condition 是选择条件的结构体，包括操作符 op，属性的列名表名 col_name，tb_name 以及与之比较的 value。

```
enum OP{
    E=1,GE,G,LE,L,NE
};

struct SelItem{
    std::string table_name;
    std::string col_name;
};
struct FromItem{
    std::string tb_name;
};
struct Join{
    std::string tb_name1;
    std::string tb_name2;
    std::string col_name1;
    std::string col_name2;
};
struct Condition{
    OP op;
    std::string tb_name;
    std::string col_name;
    std::string value;
};

struct SelectQuery{
    int sel_count=0;
    int cond_count=0;
    int from_count=0;
    int join_count=0;
    SelItem  SelList[MAX_SEL_NUM];
    FromItem  FromList[MAX_FROM_NUM];
    Join   JoinList[MAX_JOIN_NUM];
    Condition  CondList[MAX_COND_NUM];
};
```

Delete 语句：Delete 是记录删除语句的结构体，其中 table_name 为删除语句所涉及的表名，condition_count 为删除条件个数，delCondition 是记录删除条件的结构体，包括属性名 column_name，操作符 op，以及与之比较的 value。

```
struct delCondition{
    std::string column_name;
    std::string op;
    std::string value;
};

struct Delete{
    std::string table_name="";
    int condition_count = 0;
    delCondition conditionList[MAX_DEL_NUM];
};
```

Fragment 语句：Fragment 是记录分片语句的结构体，其中 frag_count 为分

片的个数，frag 中的 DBnum 记录了该分片属于哪个 DB，以及该条分片中包含的垂直分片个数 condition_v_count 和水平分片个数 condition_h_count，condition_v 记录了垂直分片后包含哪些列，condition_h 则记录了水平分片的条件，和上述的 condition 同理。

```
struct ConditionH{
    std::string attr_name="";
    std::string operation="";
    std::string attr_value="";
};

struct frag{
    int DBnum=0;
    int condition_h_count = 0;
    ConditionH condition_h[MAX_FRAG_H];
    int condition_v_count = 0;
    std::string condition_v[MAX_FRAG_V];
};

struct Fragment{
    std::string tb_name="";
    int frag_count=0;
    frag fragment[MAX_FRAG_COUNT];
};
```

Insert 语句：Insert 是记录 insert 语句的结构体，其中 tb_name 为插入的表，valuesList 存储插入的属性，values_count 记录了插入的属性数量。

```
struct Insert{
    std::string tb_name="";
    std::string valuesList[MAX_INSERT_COUNT];
    int values_count=0;
};
```

Create 语句：Table 是记录 create 语句的结构体，其中 tb_name 为创建的表名，table_att 存储表中各列的属性，att_name 为列名，type 为属性类型，size 为类型的长度，当类型为 int 时，长度为-1，iskey 为是否主键的标志符。

```
struct table_att{
    std::string att_name;
    std::string type;
    int size;
    bool iskey = 0;
};

struct Table{
    std::string tb_name="";
    table_att att_list[MAX_ATT_COUNT];
    int att_count=0;
};
```

Load 语句：Load 是记录 load 语句的结构体，其中 tb_name 为载入的表名，filepath 为数据文件的路径。

```
struct Load{
    std::string tb_name;
    std::string filepath;
};
```

DefineSite 语句：SiteInfo 是记录定义站点语句的结构体，其中 siteId 为定义的站点 id，db_name 为站点名称，ip 和 port 分别为站点的 ip 地址和端口号。

```
struct SiteInfo{
    int siteID;
    std::string db_name;
    std::string ip;
    std::string port;
};
```

3.3.2.正则表达式解析：

正则表达式：通过 regex_match 函数将输入语句与正则表达式匹配，正则表达式为 select\\s+(.*)\\s+from\\s+(.*)\\s+where\\s+(.*)；将 select 语句拆分为三段，然后调用 sel_parser 将第一段解析为一段段的属性名并存入结构体，其中 regex_search 为搜索函数，将搜索字符串中所有与正则表达式匹配的子串，本例中[\\w]+\\.[\\w]+即搜索所有符合“表名.属性”的子串，然后再进行接下来的解析操作。

```
void sel_parser(std::string sel_item ,SelectQuery& sql){
    std::regex sel_item_Pattern("[\\w]+\\.[\\w]+");
    std::smatch sel_item_Results;
    std::string::const_iterator start = sel_item.begin();
    std::string::const_iterator end = sel_item.end();
    std::string selectTempo[MAX_SEL_NUM];
    int i = 0 ;
    while(regex_search(start, end, sel_item_Results, sel_item_Pattern))
    {
        std::string msg(sel_item_Results[0].first, sel_item_Results[0].second);
        selectTempo[i] = msg;
        i++;
        start = sel_item_Results[0].second;
    }
    int c=0;
    while(selectTempo[c] != "\0"){c++;}
    sql.sel_count = c;
    for(int i = 0 ; i < c ; i++ ){
        int index = selectTempo[i].find_first_of(".");
        sql.SelList[i].table_name = selectTempo[i].substr(0,index);
        sql.SelList[i].col_name = selectTempo[i].substr(index+1);
    }

}
```

由于空间限制，下面举一个 fragment 语句解析的例子，其余语句的解析不再详细说明。首先通过 Frag\\s+(\\w+)(.*\\s+db1)?(\\s+.*\\s+db 2)?(\\s+.*\

\ s + db3)? (\\s+.*\\s+db4)?; 将 fragment 语句切分成每个 DB 的子句

```
bool FragmentParser(std::string fragmentSql,Fragment& sql){
    std::regex fragPattern("frag\\s+(\\w+)(.*\\s+db1)?(\\s+.*\\s+db2)?(\\s+.*\\s+db3)?(\\s+.*\\s+db4)?;");
    std::smatch fragResults;
    std::string fragTemp[4];
    int loc_fag = 0;
    if (regex_match(fragmentSql, fragResults, fragPattern)){
        sql.tb_name = fragResults[1];
        if(fragResults[2]!="\0"){
            sql.fragment[loc_fag].DBnum=1;
            sql.frag_count++;
            fragTemp[loc_fag] = fragResults[2];
            fragTemp[loc_fag] = fragTemp[loc_fag].substr(0, fragTemp[loc_fag].length() - 4);
            loc_fag++;
        }
        if(fragResults[3]!="\0"){
            sql.fragment[loc_fag].DBnum=2;
            sql.frag_count++;
            fragTemp[loc_fag] = fragResults[3];
            fragTemp[loc_fag] = fragTemp[loc_fag].substr(0, fragTemp[loc_fag].length() - 4);
            loc_fag++;
        }
        if(fragResults[4]!="\0"){
            sql.fragment[loc_fag].DBnum=3;
            sql.frag_count++;
            fragTemp[loc_fag] = fragResults[4];
            fragTemp[loc_fag] = fragTemp[loc_fag].substr(0, fragTemp[loc_fag].length() - 4);
            loc_fag++;
        }
        if(fragResults[5]!="\0"){
            sql.fragment[loc_fag].DBnum=4;
            sql.frag_count++;
            fragTemp[loc_fag] = fragResults[5];
            fragTemp[loc_fag] = fragTemp[loc_fag].substr(0, fragTemp[loc_fag].length() - 4);
        }
    }else{
        return false;
    }
```

然后用^\\((.*)\\)匹配来判断垂直分片，用(.*)\\((.*)\\)匹配来判断水平垂直混合分片，最后将匹配到的子段输入到提前写好的 parserAtt 和 parserCon 函数来抽取具体的属性和选择条件子段并存入结构体。

```
    std::regex vPattern("^\\((.*)\\)");
    std::smatch vResults;
    std::regex hvPattern("(.*)\\((.*)\\)");
    std::smatch hvResults;
    for( int i = 0 ; i < sql.frag_count ;  i++ ){
        fragTemp[i] = fragTemp[i].substr(1);
        //std::cout<<fragTemp[i]<<std::endl;
        if(regex_match(fragTemp[i], vResults, vPattern)){
            parserAtt(vResults[1],sql.fragment[i].condition_v);
            sql.fragment[i].condition_v_count = lenn(sql.fragment[i].condition_v);
            continue;
        }
        if(regex_match(fragTemp[i], hvResults, hvPattern)){
            parserAtt(hvResults[2],sql.fragment[i].condition_v);
            sql.fragment[i].condition_v_count = lenn(sql.fragment[i].condition_v);
            std::string split1[15];
            parserCon(hvResults[1],split1);
            for(int j = 0 ,jj = 0 ; j < lenn(split1) ; jj++ ){
                sql.fragment[i].condition_h[jj].attr_name = split1[j++];
                sql.fragment[i].condition_h[jj].operation = split1[j++];
                sql.fragment[i].condition_h[jj].attr_value = split1[j++];
                sql.fragment[i].condition_h_count++;
            }
            continue;
        }
        std::string split2[15];
        parserCon(fragTemp[i],split2);
        for(int j = 0 ,jj = 0 ; j < lenn(split2) ; jj++ ){
            sql.fragment[i].condition_h[jj].attr_name = split2[j++];
            sql.fragment[i].condition_h[jj].operation = split2[j++];
            sql.fragment[i].condition_h[jj].attr_value = split2[j++];
            sql.fragment[i].condition_h_count++;
        }
    }
    return true;
}
```

3.4 元数据设计

3.4.1.etcd 介绍：

我们数据库选择用 Etcd，以 Key-value 来存储 GDD。Etcd 的特点具有很好的故障恢复性，在每一个节点都维护全部数据，避免了数据的缺失，符合 GDD 的要求。我们使用 3 个站点来维护 GDD。

3.4.2.GDD 结构设计：

Table 数据结构

```
struct AttrInfo {
    // string tb_name;
    string attr_name;
    string type;
    bool is_key = 0;
    int size;
};

struct Table{
    string tb_name;
    AttrInfo attrs[MAX_ATTR_COUNT];
    int attr_count = 0;
};
```

Site 数据结构

```
struct SiteInfo{
    int siteID;
    std::string db_name;
    std::string ip;
    std::string port;
};

struct SiteInfos{
    int sitenum;
```

```
    SiteInfo site[MAX_SITE_COUNT];
};
```

Fragment 数据结构

```
struct ConditionH{
    string attr_name="";
    string operation="";
    string attr_value="";
};

struct frag{
    int DBnum;
    int condition_h_count = 0;
    ConditionH condition_h[MAX_FRAG_H];
    int condition_v_count = 0;
    string condition_v[MAX_FRAG_V];
};

struct Fragment{
    string tb_name;
    int frag_count=0;
    frag fragment[MAX_FRAG_COUNT1];
};
```

3.5 查询树及查询计划

3.5.1.查询树的生成及优化：

输入 1：当前查询通过 parser 解析后的结构体 SelectQuery

```
struct SelectQuery{
//  int distinct;
    int sel_count;
    int cond_count;
    int from_count;
    int join_count;
    SelItem     SelList[MAX_SELITEM_NUM];
    FromItem    FromList[MAX_FROM_NUM];
    Join        JoinList[MAX_JOIN_NUM];
    Condition   CondList[MAX_COND_NUM];
    //SelectQuery* next;
};
```

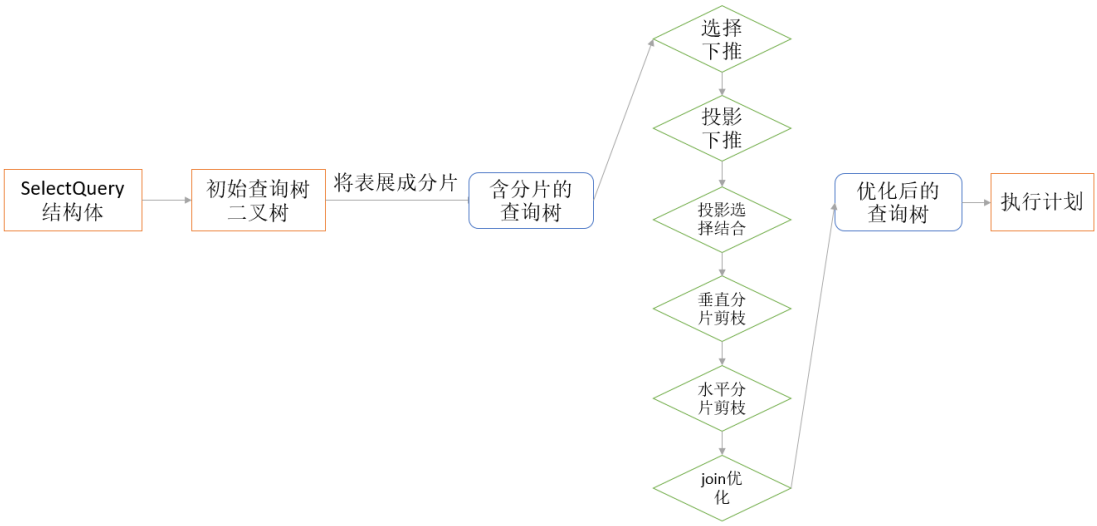
输入 2：表的结构和站点、DB、端口信息

```
struct Select_Table{
    string tb_name;
    AttrInfo attrs[MAX_ATTR_COUNT];
    int attr_count;
    int is_need_frag;
    int frag_count;
    Select_Fragment fraginfo[MAX_FRAG_COUNT];
    int frag_type; //0 h 1 v 2 mix
    int db_id_needed[MAX_DB_COUNT];
};

struct TableList{
    int table_num;
    Select_Table* tbl[MAX_TABLE_COUNT];
};
```

输出是经过优化后的查询树以及查询计划。

构建查询树和查询计划的流程可见下图：



1.1 初始化的查询树构建：

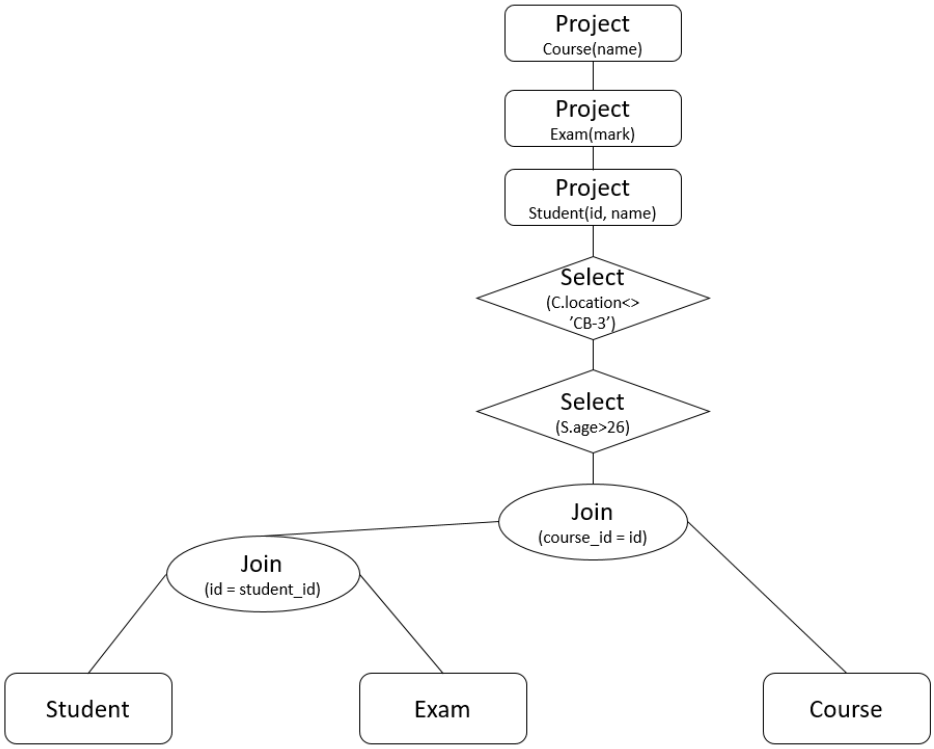
初始化的查询树是一颗二叉树，叶子结点是表，非叶子结点均是操作，

构建顺序为先构建 join 结点，join 结点的孩子结点有可能是两个表，也有可能一个表，一个叶子结点，而后构建 select 结点，select 结点只有一个孩子，孩子结点为表结点或者 join 结点，每一个选择条件构建了一个节点，最后构建投影节点，每一个表一个 project 节点。

举例说明：查询

Select Student.id, Student.name, Exam.mark, Course.name from Student, Exam, Course where Student.id = Exam.student_id and Exam.course_id = Course.id and Student.age>26 and Course.location<>’ CB-3’ ;

生成的初始查询树为：



1.2 将表展开成分片信息：

由 etcd 获取了表的信息，下一步将表节点（叶子结点）转换成分片的形式。此时树不再是二叉树，所以用的顺序存储方式，树的修改如下：

1） select 节点和 project 节点不变。

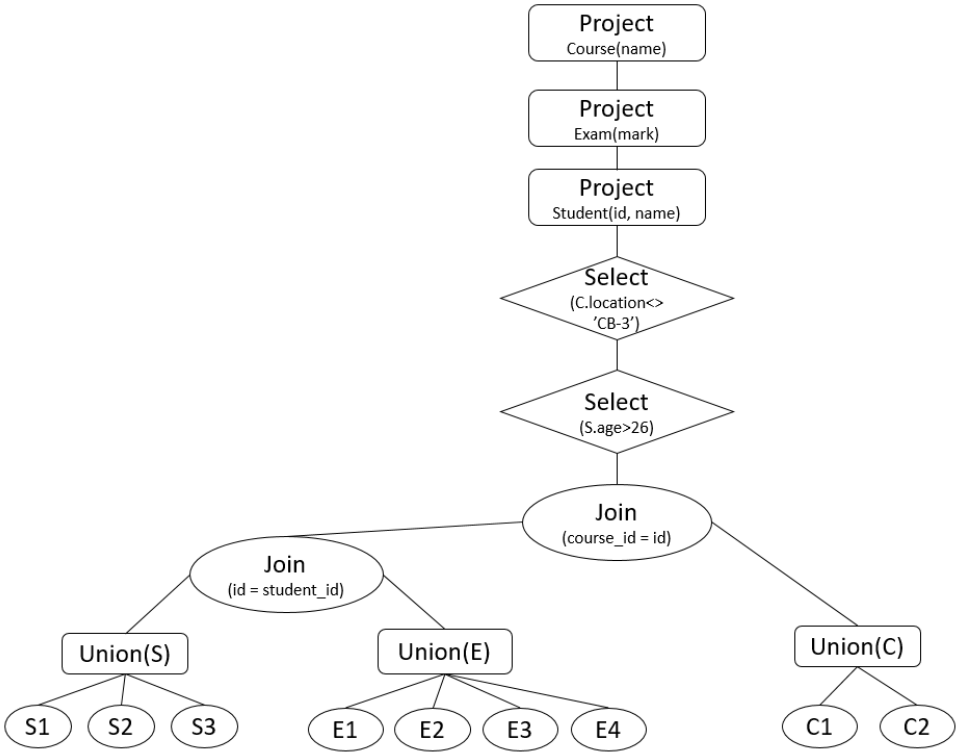
2） 如果 join 的节点是一个表，那么加上一个虚拟的 union 节点，（并不是后期真正的 union，是一个虚拟节点），union 节点的父节点就是对应的 join 节点，子节点是分片。

3） 表的结点删除（代码中的做法是让其父节点为-1）。

4） 无论是垂直分片还是水平分片现阶段都用 union 节点表示，如果是

垂直分片后期将 union 节点转换成 join 节点，如果是水平分片，后期将 union 节点保留。

上面的查询展开成分片信息的结果如下图所示：



1.3 选择下推

分布式在进行 join 时可能会经常将表从一个站点发到另外一个站点，因此为了减少 IO 操作，将每一个 select 操作下推。具体做法为：

1） 查看每一个 select 操作，以 Student.id 为例，扫描每一个 union 节点，如果 union 节点的表名和 Select 操作对应的表名相同的话，为 union 对应的每一个 fragment 创建一个父亲节点，该节点是一个 select 节点，孩子节点即为对应的 fragment 节点，父节点就是 fragment 之前的父节点 union 节点。

2） 删除上面的 select 节点。

特别注意的点：

对于垂直分片的表来说，要找到选择条件针对的列，对于水平分列的表来说，就是每一个分片上面都创建这样一个 select 节点，但是对于垂直分片来说，是包含这个列的分片上面创建这样的 select 节点。

1.4 投影下推

同样是为了减少 IO 操作，将投影节点下推，但是需要注意以下两点：

1） 投影节点的下推要注意保留该表在整个查询中用到的全部的列，例如对于 Exam 表，虽然上面的 project 节点只投影了它的 Exam 列，但是在 join 过程中用到了它的 student_id, course_id 列, 因此也要保留。

2） 正是因为上面的原因，不应该删除上面的 project 的节点，只是在下面加上一个 project 节点。

因此具体过程为：

1） 对每一个表都找到在整个查询中所有需要的列（扫面和他相关的 select 节点、join 节点、project 节点）。

2） 在每一个 union 下创建一个 project 节点, project 出该表所有需要的列，该节点的父节点是 union 节点，子节点是 fragment 节点或者 1.3 步骤生成的 select 节点。

3） 原来的查询树上面的 project 操作不做任何操作。

特别需要注意的点：

同选择下推相同, 如果是水平分片, 即每一个分片创建这样一个父节点，但是对于垂直分片，要看当前的 project 对应的列对应的是该表的哪一个分片。

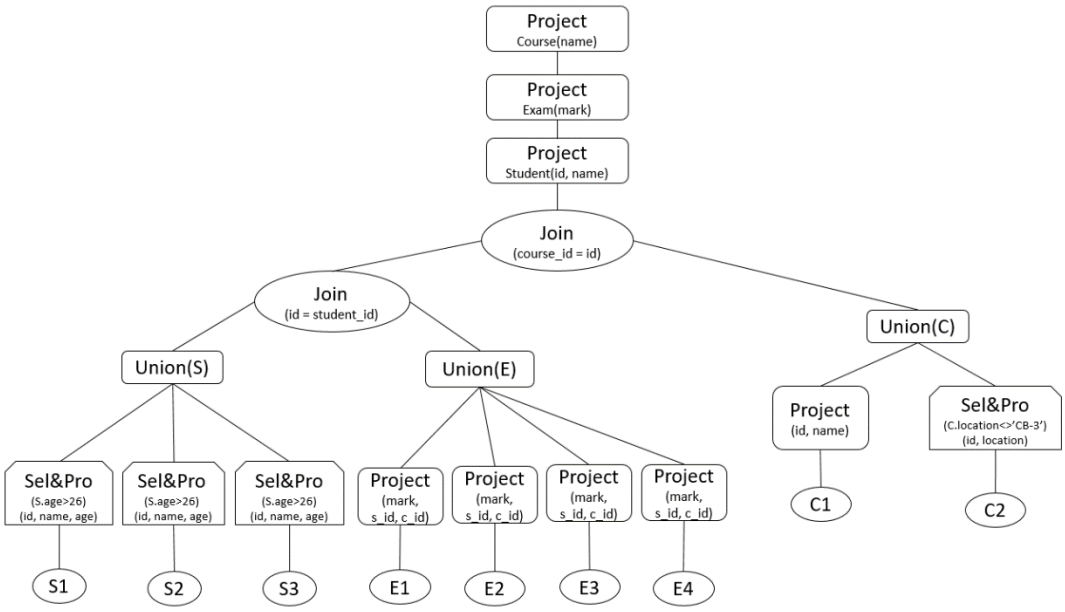
1.5 选择投影结合

经过 1.3,1.4 步骤，将选择和投影条件进行了下推，因为选择和投影可以在一个 sql 操作中实现，因此将两个操作结合在一起，具体操作如下：

1） 如果一个 union 节点子树下面既有 select 操作节点又有 project 节点，那么构建一个 selpro 节点，该节点记录了 select 信息和 project 出来的列信息, 该节点的父节点是对应的 union 节点, 子节点是 union 子树下对应的 fragment 节点。

2） 删除 union 节点下面的 select 节点和 project 节点。

经过步骤 1.3,1.4,1.5 选择投影下推和结合后，得到的查询树如下图所示：



1.6 垂直分片剪枝

当一个表是垂直分片时，有可能遇到剪枝的情况，具体描述如下：

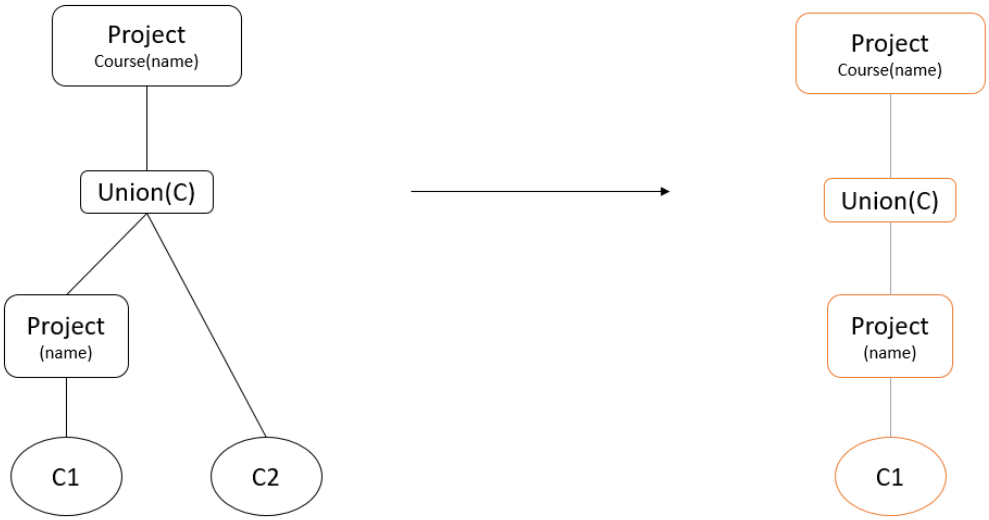
1） 对于每一个垂直分片的表，找到所有它在这个查询中会用到的列（和之前的方法相同，遍历和他相关的 select 节点、join 节点、project 节点）。

2） 如果所有需要的列都在一个分片上，那么可以将另外的分片删除，或者说，一个分片在这个查询中没有需要得列（主值除外），那么将该分片删除。

上述的例子没有涉及到垂直分片，用下例说明垂直分片：

Select Course.name from Course.

查询树的变化如图所示：



1.7 水平分片剪枝

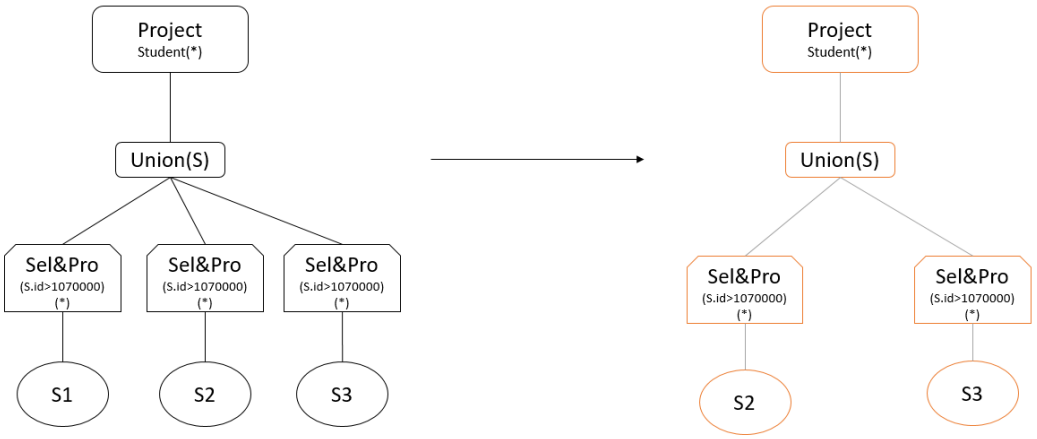
如果一个表是水平分片的，且他还有 select 操作，那么 select 操作对应的条件和该分片的分片条件相违背的话，那么应该删除该分片进行简化。举例说明：

Select * from Student where Student.id > 1070000.

S1 的分片依据是 Student.id < 1050000，而该选择条件是其大于 1070000，两个条件没有交集，因此应该删除这个分片。

需要注意的是，删除该 fragment 的时候应该也删除对应的 select 节点或者 select&project 节点，同时更新当前的 union 节点的孩子个数。

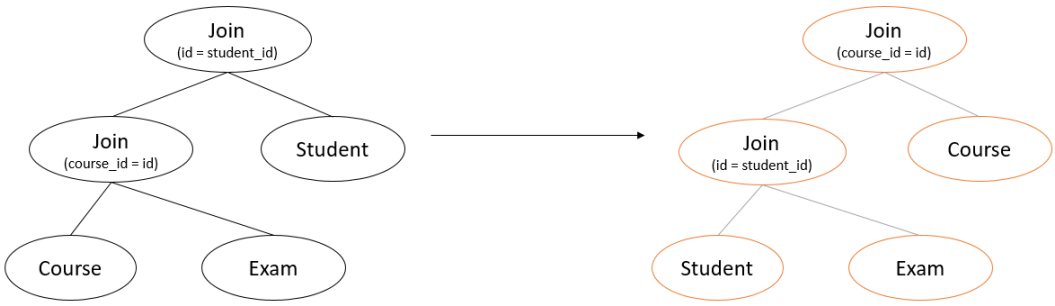
上个例子中删除水平分片后的结果如下图所示：



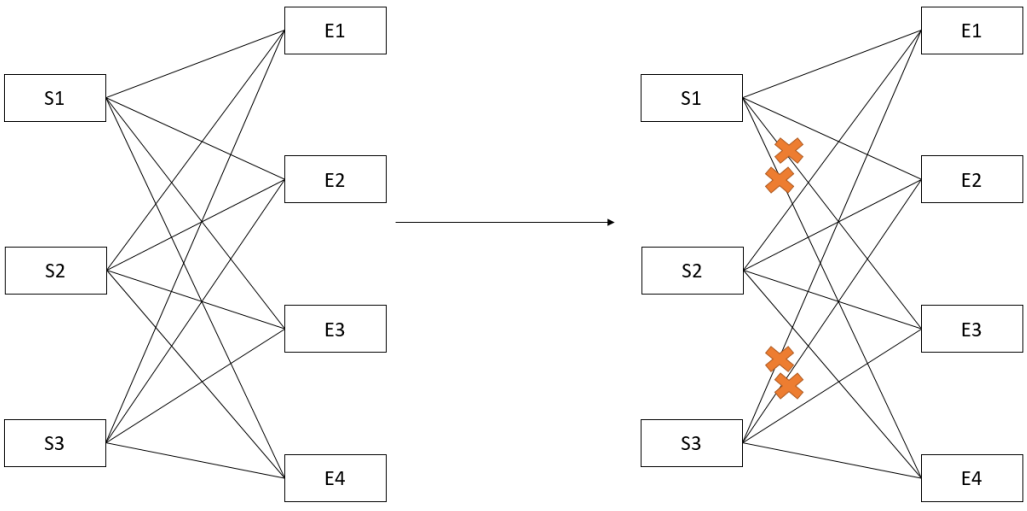
1.8 join 的优化

Join 的优化主要考虑以下几点：

1） 该查询的总 join 顺序，主要针对三表及以上连接，如果两个表都是水平分片且依据的是相同的列（或者互为外码的关系），那么应该先 join 这两个表。具体表示如图所示：



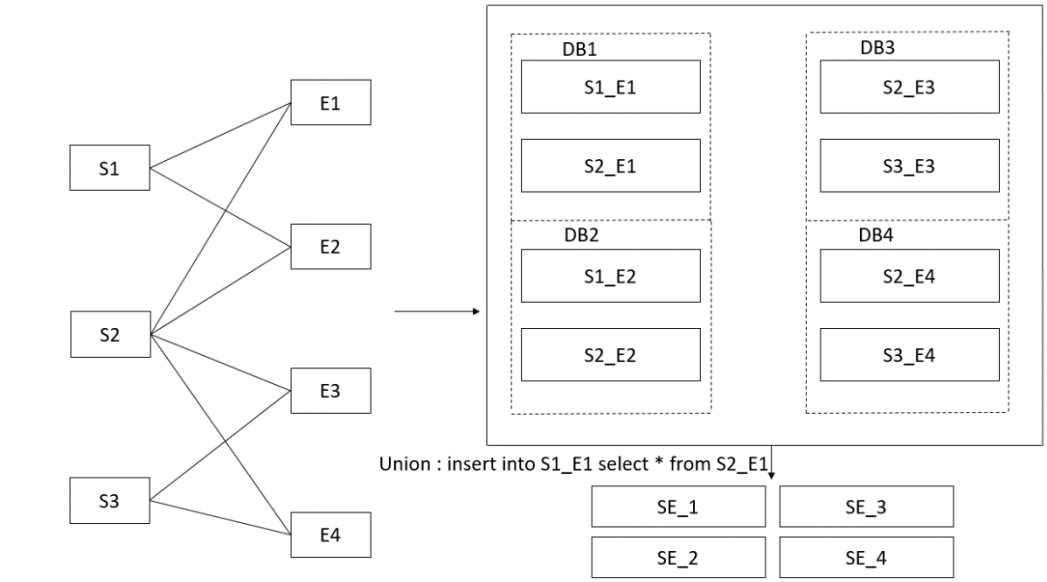
2） 对于每一个 join 进行剪枝，首先生成 joinnlist，比如 Student 表和 Exam 表的 join，Student 表有 3 个分片，S1, S2, S3，Exam 表有 4 个分片，E1, E2, E3, E4，不剪枝之前，应该是 3*4=12 次 join。但是如果两个表都是水平分片且水平分片对应的列相同（或者互为外码），那么如果两个分片的信息相违背的话，不应该进行 join，例如 S1 的依据是 Student.id<1050000, 而 E3 的依据是 student_id>=1070000 and course_id<301200, Student 的 id 是 Exam 表的外码, 而小于 1050000 和大于等于 1070000 中间没有交集，所以不需要进行 join。Join 的剪枝如下图所示, 生成的 joinlist 为 S1&E1, S1&E2, S2&E1, S2&E2, S2&E3, S2&E4, S3&E3, S3&E4.

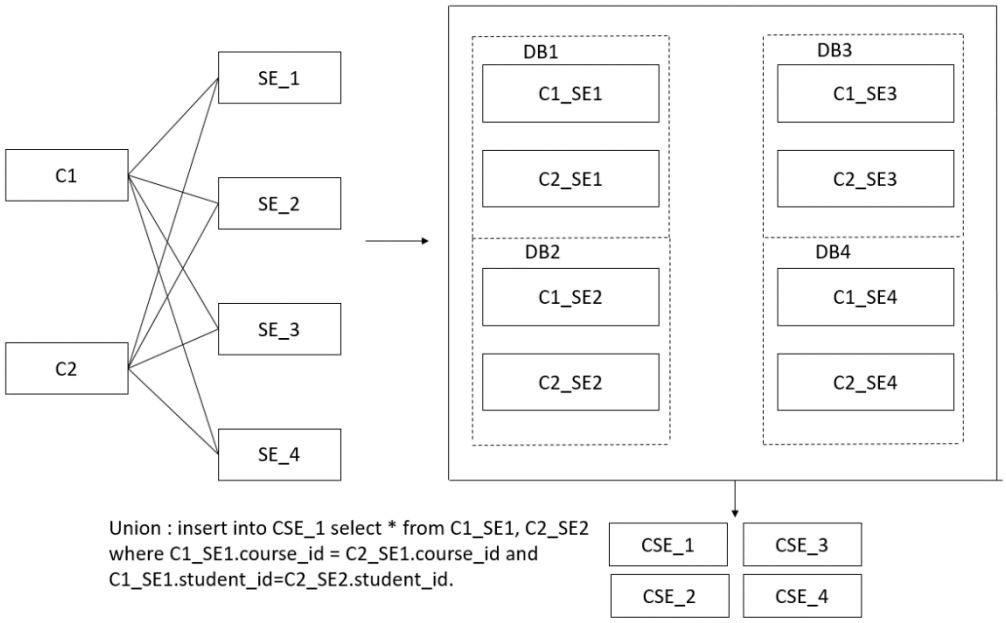


3） 对于 joinlist 中的每一次 subjoin，首先确定应该先发哪个表。（这里

利用的是哪个表的列少，后面需要改进），大的表不动，小表进行传输。例如上例中发的表就是 Student。

4） 当每一个 subjoin 完成后，在每一个 db 上生成的 join 表进行 union，如果两个表都是水平分片，那么进行 union，如果有垂直分片的话，那么进行 join 操作。举例说明 Student，Exam，Course 之间进行 join 的结果。Join 的顺序是 Student 先与 Exam 进行 join，join 的结果与 Course 进行 join。具体表示如下图所示：





雷区：在三表或多表连接时，如果在第三个表或以上遇到了垂直分片的表

时，在 union 的时候，条件不只是垂直分片表的主值，还包括之前 join 的条件，如上例中不仅仅是 C1.course_id 的 join 还包括 SE 表的 Student 的 join。

3.5.2.生成查询计划：

得到最后优化的查询树时，下一步依据该查询树和相应结构体中的信息生成相应的查询计划，以便调用 RPC 进行执行。生成查询计划的步骤是：

1） 扫描 fragment 列表，对于每一个分片的表，先在当前站点进行 select 和 project 操作。例如，select Student.name，Student.id from Student where Student.age > 26;
2） 接下来是 join 的操作，对于 joinlist 中的每一个 subjoin，如果两个表在一个站点，那么直接 join，即生成 join 后的表，然后依据 join 的条件进行 join，例如：create table Student_frag1_Exam_frag1 (…); insert into Student_frag1_Exam_frag1 select (…) from Student_frag1，Exam_frag1 where Student_frag1.id = Exam_frag1.student_id;
3） 如果两个表不在一个站点上，首先要进行发表，发表的过程是先在目标站点上创建一模一样的表结构，然后将数据从源站点发到目标站点，在进行 insert，（此过程有相应的 RPC 接口）。在进行步骤 2.
4） 当 join 结束后，在一个站点上的进行 union，union 可能是两个表简单的并在一起，也有可能是两个表进行 join。
5） 最后进行 project 操作，project 操作直接对每一个站点进行，即最后一步的 join 得到的站点上的表不在进行汇总，而是直接将结果存在提前设好的 vector 中（此步也有相应的接口）。

4 任务分工及小结

4.1 任务分工

组员分工如下：

1. 王佳悦：查询计划，查询优化
2. 林艺辉：元数据管理，etcd 设计，系统部署
3. 陈明：SQL 解析，系统部署
4. 曾翔：系统设计，节点通信，Local SQL Executor，查询以外的操作执行

4.2 王佳悦小结

虽然本科也是计算机专业，但是这个课真的是我上过最硬核的课（核爆炸），

学期开始，由于对分(lao)布(shi)式(hao)数(shuai)据(ying)库(yu)十(hao)分(hao)好(ting)奇,开始旁听这门课，学到了很多知识，所以当老师讲述大作业的要求时，跃跃欲试加入了这一组，想将学到的知识真正的实现一下，事实证明虽然代码量空前，但是在这个过程中真的锻炼了自己的编程能力。

首先，我负责的部分是查询树的生成、优化、和查询计划的生成部分。这一部分利用将查询解析成的结构体以及 etcd 提供的表和站点信息，生成查询计划，写代码的过程中遇到了很多问题，比如树结构的选择，优化过程中垂直分片的复杂，因为先进行了投影操作的下推，所以后面 join 过程中建立的表都有哪些列等等问题，当自己三步一 bug，每天 debug 想要放弃的时候，发现已经写了这么多，征服它的愿望越来越强烈，还好最后坚持下来了。

完成的过程虽然艰辛，但是也积累的很多经验，比如结构体的设计（类的设计）是基础，一定要设计好结构体后在进行函数的编写，才能避免代码经常性的重构，此外，函数还是应该写的活一点，例如之前没有太意识到垂直分片和水平分片本质上的不同，很多函数的设计都比较针对水平分片，后期考虑垂直分片的剪枝时，函数修改的太多，还有就是代码的安全性问题，经常写完 if 语句后，没有意识把 else 报错信息写出来，所以后期代码量较多，debug 时，经常不清楚是哪一步的问题，如果写好报错信息，那么 debug 过程要容易的多。

团队的其他成员都特别的靠谱，在合代码的时候，基本上都确定自己的接口没有问题，这使得我们合代码的速度比想象中快很多。

最后想说，这门课真的满满干货，上的很值得，大作业做的也很值得，原来我真的可以学计算机耶！

4.3 林艺辉小结

本学期选了分布式与并行数据库，刚开始是因为这是研究生第一堂课，不想去放弃。接着就是因为与其他课程相比，这个课程明显感觉到能收获很多。所以就硬着头皮选下去了。我们组从 10 月开始讨论怎么做项目，我分配到做 GDD。其实，GDD 任务并不重，编写代码一周就可以完成，但是难的是部署环境。由于我们开发环境是 C++，需要许多包，比如 libjson（解析 json 数据）、libcurl（请求 etcd）,安装弄懂部署这些环境花费了很多时间。再者，etcd 返回的数据是无序的，刚开始没注意到这一点。后面就是因为这点，使得许多操作都是错误的。这也花费了时间。最后是小组成员在数据结构设计上没有商量好，3 人有 3 套标准，这就意味着我要写 3 个接口来转换数据。这无疑是浪费了时间。我负责了搭建集群环境（天呐，千万别入这个部署环境的坑，不然你脑袋会爆炸的），许多

环境标准不一，找问题一定要沟通，只有沟通才能更快解决问题，并且在部署环境中，看博客一定要仔仔细细看完，不要囫囵吞枣就去部署，会吃亏的。在这个项目中，我可能做的最杂的部分，所以要保持一个冷静的心态去学习吧。非常感谢曾翔、王佳悦和陈明同学。也感谢范老师，他讲课超棒！这门课很硬，收获也很大。我会记住 12.30 午夜阳光地带编程的。

4.4 陈明小结

通过与小组成员近两个月合作，终于圆满地完成了分布式并行数据库系统，这是我第一次从头参与开发一个相对完整的系统。开发过程中的问题层出不穷，最开始每个人的接口定义都不相同，浪费了很多时间来修改代码，后来代码整合到一起也会经常出现莫名其妙的 bug，所以以后在写代码时应该首先设计好代码的层次结构，增加代码复用性，在修改时也会方便很多；除了代码上的问题我们在配置环境时也不顺利，在临近作业检查的时候发现在虚拟机上配置的三台机器并不能实现通信，于是又找了 3 台物理机重新配置环境，这才可以在多机环境下运行程序，然后还增加了多线程并行，由于查询一开始很慢所以调试程序也很麻烦。和其他课程相比，这门课确实让我和分布式数据库这个深坑有了一次亲密接触，在做大作业的最后阶段也曾多次想过放弃，不过在团队成员的相互帮助下终于让我们在 ddl 前完成了作业，最后，感谢曾翔、王佳悦和林艺辉同学，他们每个人都非常厉害，也感谢范老师的认真备课与教学，让我学到了很多关于分布式的知识。

建议这门课正式更名为：分布式数据库从入门到放弃。

4.5 曾翔的小结

本学期的分布式与并行数据库对我来说是一门非常重要的课，尽管上过课的师兄师姐说它一点都不“水”，我还是选了，事实也证明这门课一点也不容易，最后验收完甚至有一种毕业了的感觉，也深深地感受到“talk is easy, show me the code.”。在写大作业的过程中确实有许多的优化的方案和想法，有的优化甚至可以用显然来描述，但是真正的用代码去实现的时候就会遇到非常多的困难，包括数据结构的设计，代码的逻辑等，让我深深的体会到了自己工程能力的不足。在本次大作业中有许多收获：明白了数据结构对于整个系统实现的重要性，一个良好的数据结构能够省下很多不必要的精力和时间；其次，任务的切割也非常重要，通过预先定义好的接口能让多个任务并行的完成，减少组员之间相互等待的时间，但是在接口的定义和结构体的设计沟通上我们小组出现了一点偏差，三个人甚至使用了三种 Fragment 结构体，浪费了许多时间在结构的转化上；最后，

非常感谢王佳悦同学的帮助，没有她的帮助，我们不可能在规定的时间完成任务。

关于这门课一个小小的建议：可以找一找国外相关课程的实验，定义代码的框架，大家只需要把最核心部分的算法实现了，相比从头实现一个分布式数据库更能抓住和理解这门课核心的思想。