分布与并行数据库实验报告

刘同禹 陈思蓓 时青 郝新丽

完成时间 2020.1.14

目 录

第1章 元数据设计

1.1 etcd 介绍：

我们使用 etcd 来存储全局信息，及数据库的表信息，分片信息及站点信息等元数据。etcd 以 Key-value 形式存储数据，且能够做到同步一个集群中不同站点上的元数据信息，且具有良好的故障恢复性，可以通过 curl 指令以 json 格式返回要查询的信息。

1.2 GDD 结构设计：

1.2.1 站点信息结构：

```
struct site_info
{
        int site_id;
        vector<int> fragment_ids;
        vector<int> temp_ids;
    string user;
    string password;
        string ip;
        string port;
};
```

1.2.2 表与属性的信息结构

```
//表的属性信息数据结构
struct attr_info
{
        string attr_name;
        string type;
        bool is_key;
};

//表相关信息的数据结构
struct table_info
{
        string table_name;
        string key;
        bool is_temp;
        vector<string> attr_names;
        map<string, attr_info> attributes;
        vector<int> h_frags;
        vector<int> v_frags;
};
```

1.2.3 分片信息结构

```
//分片信息的数据结构
struct frag_info
{
        int frag_id;
        FRAGTYPE frag_type;
        string table_name;
        int site_id;
    bool is_temp;
    int size;
        vector<predicateS> preds;
        vector<predicateV> predv;
        vector<string> attr_names;
        map<int, attr_info> attr_infos;
};
```

1.3 etcd 结构设计：

我们分别以/siteinfo, /tableinfo 和/fraginfo 这三个目录存放站点信息、表信息和分片信息，对于站点信息，我们以站点 id 作为其目录，以“/siteinfo/[site_id]“访问对应站点，对于表信息，我们以表名作为其目录，以“/tableinfo/[table_name]“访问对应表信息，对于分片信息，我们以分片 id 作为其目录，以“/fraginfo/[frag_id]“访问对应分片信息。

第2章　查询树构建及优化

2.1 相关结构体定义：

predicateV: 数值型谓词结构体，负责存储一条 table.attribute rel value(数值型 value)谓词，包含的成员变量： string 型变量 table_name，存储谓词对应的表名； string 型变量 attr_name，存储谓词对应的属性名；枚举型变量 rel，存储谓词中的关系(>,<,>=,<=,!=,=)；成员函数 get_str()，获得谓词的字符串表示形式，用于输出展示使用。

```
struct predicateV{
        string table_name;
        string attr_name;
        RELATION rel;
        int value;
        string get_str(){
                return table_name+"."+attr_name+rela_to_token(rel)+to_string(value);
        }
};     // INT 型谓词 e.g Table.attribute = 12
```

predicateS: 字符串型谓词结构体，负责存储一条 table.attribute rel value(字符串型 value)谓词，包含的成员变量：同 predicateV.

```
struct predicateS{
        string table_name;
        string attr_name;
        RELATION rel;
        string value;
        string get_str(){
                return table_name+"."+attr_name+rela_to_token(rel)+value;
        }
};      // CHAR 型谓词  e.g Table.attribute = 'string'
```

predicateT: 表型谓词结构体，负责存储两表之间的关系谓词:table1.attr1 rel table2.attr2，包含的成员变量：string 型变量 left_table，存储谓词关系左侧表名；string 型变量 left_attr，存储谓词关系左侧属性名；枚举型变量 rel，存储谓词中的关系(>,<,>=,<=,!=,=)；string 型变量 right_table，存储谓词关系右侧表名； string 型变量 right_attr，存储谓词关系右侧属性名；成员函数 get_str()，同 predicateV.

```
struct predicateT{
        string left_table;
        string left_attr;
        RELATION rel;
        string right_attr;
        string right_table;
        string get_str(){
                return left_table+"."+left_attr+rela_to_token(rel)+right_table+"."+right_attr;
        }
};     // TAB 型谓词 e.g Table1.attribute = Table2.attribute
```

query_tree_node: query_tree_node 是查询树中树节点节点的结构体，包含的成员变量有：枚举型变量 node_type，存储节点类型（JOIN，SELECT，FRAGMENT，PROJECT，UNION，TABLE）；字符串向量 table_names，attr_names，存储节点涉及到的表名与属性名，整数型与整数向量 frag_id 和 frag_ids，存储节点涉及到的分片 id；存储节点涉及到的谓词类型 pred_type 和三种类型谓词向量 predv，preds，predt；树节点指针向量 child，存储该节点的孩子节点的地址；树节点指针 parent，存储该节点的父亲节点地址；成员函数 get_str()，负责打印节点；成员函数 remove_null_child()，负责删除 child 中的空指针；成员函数 get_child_index()，获得某孩子节点在其 child 向量中的索引；成员函数 copy()，递归地复制该节点及该节点的所有孩子和后代。

```
struct query_tree_node{
        NODETYPE node_type;

        vector<string> table_names;
        vector<string> attr_names;
        int frag_id;
        vector<int> frag_ids;

        PREDTYPE pred_type;
        vector<predicateV> predv;
        vector<predicateS> preds;
        vector<predicateT> predt;

        vector<query_tree_node*> child;
        query_tree_node* parent;
        string get_str(void);
        void remove_null_child(void);
        int get_child_index(query_tree_node*);
        query_tree_node* copy();
};
```

2.2 查询树构建：

2.2.1 SQL 语句分割:

负责将输入的一条 sql 语句 select [a] from [b] where [c]中的三个主要部分 a : select items、b : from items、c : where items 抽取出来。

```
void query_tree::split_sql() {
    int sel_index = this->sql.find("select");
    int from_index = this->sql.find("from");
    int where_index = this->sql.find("where");

    this->sel_items = this->sql.substr(sel_index+6, from_index-sel_index-6);
    this->from_items = this->sql.substr(from_index+4, where_index-from_index-4);
    if (where_index > 0)
        this->where_items = this->sql.substr(where_index+5);
}
```

2.2.2 SQL 语句解析:

对抽取出来的 sql 语句的各个部分使用正则表达式进行解析，对 select items 部分，解析出要 select 的表名和列名，对 from items，解析出要进行查询操作的表名，对 where items，识别出本次查询的谓词，并谓词通过 extract_pred()函数抽取出来，存为相应的谓词结构体。

```
void query_tree::parser_sql() {
    vector<string> results;
    //cout << "--------split_sql---------" << endl;
    split_sql();

    //cout << this->sel_items <<"|"<< endl;
    //cout << this->from_items <<"|"<< endl;
    //cout << this->where_items <<"|"<< endl;

    //cout << "---------parser_sel---------" << endl;
    results = parser_regex(sel_items, "[\\w]+\\.[\\w]+");
    for (auto res:results)
        projects.push_back(res);
    //cout << "---------parser_from--------" << endl;
    trim(from_items);
    table_names = split(from_items, ",");
    //cout << "---------parser_where--------" << endl;
    if(where_items.size() != 0){
        results = parser_regex(where_items, "[\\w]+\\.[\\w]+\\=[0-9]+");
        for (auto res:results)
            extract_pred(res, "=", EQ, INT);
        results = parser_regex(where_items, "[\\w]+\\.[\\w]+\\>[0-9]+");
        for (auto res:results)
            extract_pred(res, ">",G, INT);
```

2.2.3 构建初始查询树:

根据对 sql 语句解析后的结果，建立初始查询树。先建立 join 节点，在建立 join 节点时自底向下构建，先根据参与查询的表名构建出 TABLE 型节点作为叶子节点，再根据谓词信息建立对应的 join 节点，使其左右孩子为 join 要在一起的两个表节点，或是含有要 join 的表的 join 节点。之后再将对应的 select 节点及 project 节点插入即可，初始的查询树为一棵二叉树。如，对查询：

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’

得到的初始查询树如图 1：

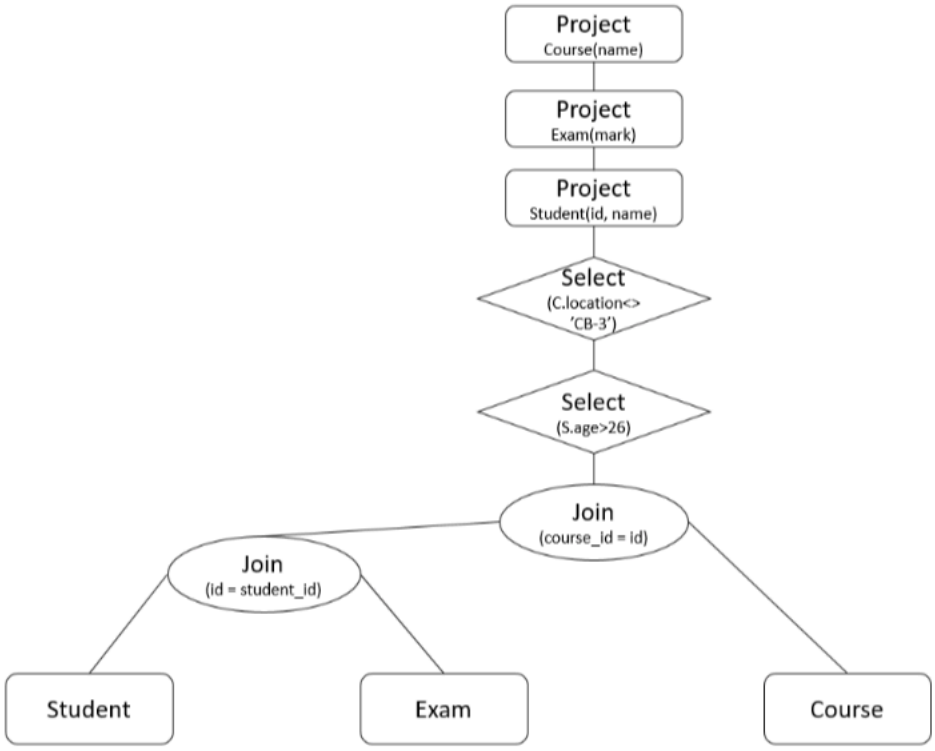


图 1

2.3 查询树优化：

2.3.1 化 TABLE 为 FRAGMENT：

根据 gdd 中存储的元数据信息，若一个 TABEL 节点的分片方式为水平分片，则将该节点转化为一个 UNION 节点，且该 UNION 节点的儿子节点为该表所对

应的分片节点，若分片方式为垂直划分，则该节点转化为一个根据主键进行 join 的 JOIN 节点，左右孩子几位垂直划分出的两个节点。其余节点保持不变，此时的查询树转化为图 2 所示。

2.3.2 SELECT 节点下推并剪枝：

对每一个 SELECT 节点，让其顺着孩子节点下推，直到碰到一个 FRAGMENT 节点，判断若选择的表和分片所属的表不同，则不进行任何操作，否则判断选择条件是否与该分片的分片条件冲突，若不冲突，则成为该分片节点的新父亲，使该选择操作进行的时间提前，若冲突，则删除该分片节点对应的全部分支，使其不会再出现在之后的优化操作中。

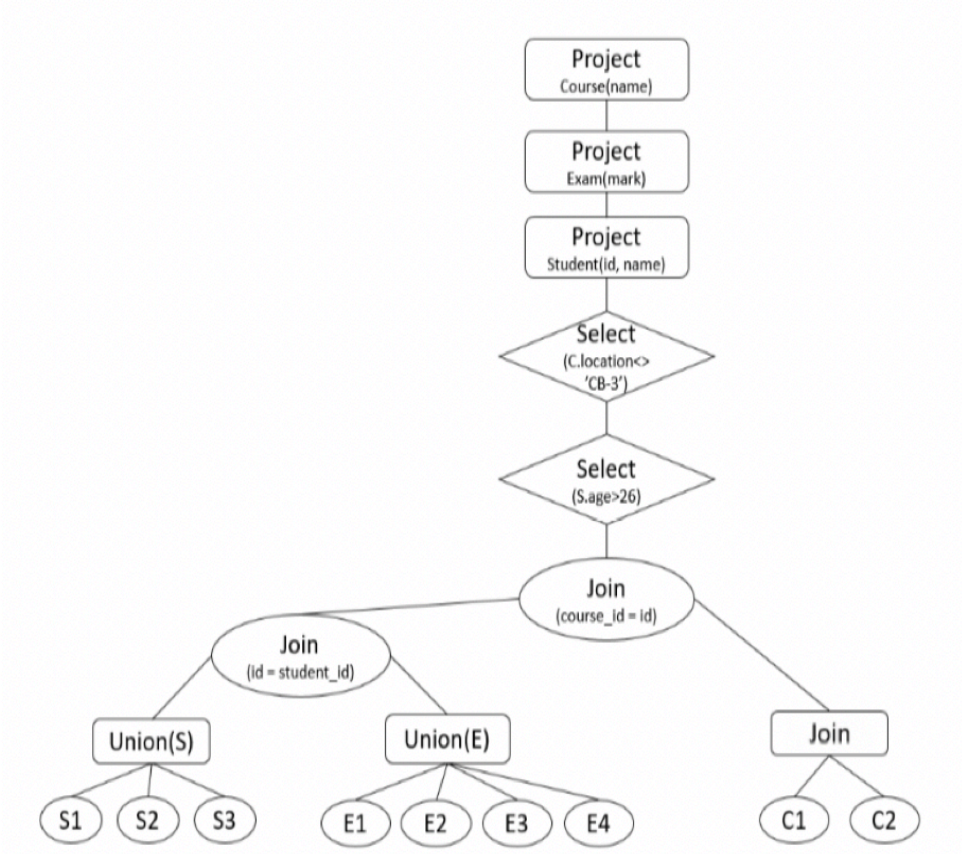
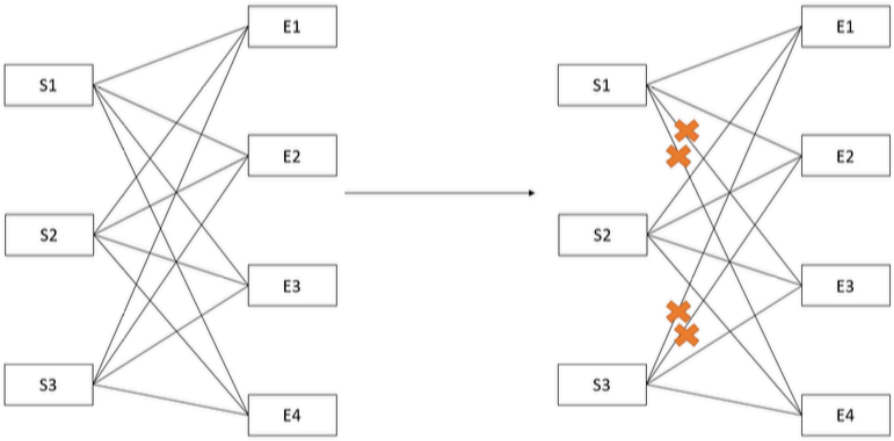


图 2

2.3.3 JOIN 节点下推并剪枝：

对于 JOIN 节点，若其左右孩子中包含一个 UNION 节点，则将自己变成一个新的 UNION 节点，并将孩子 UNION 节点中的每一个节点与另一个孩子节点进行 join，生成一系列新的 sub join 节点作为新的 UNION 节点的子节点，在生

成 sub join 的过程中，若进行 sub join 操作的两个节点在谓词条件上产生冲突，则删除该 sub join 节点。如下图所示。

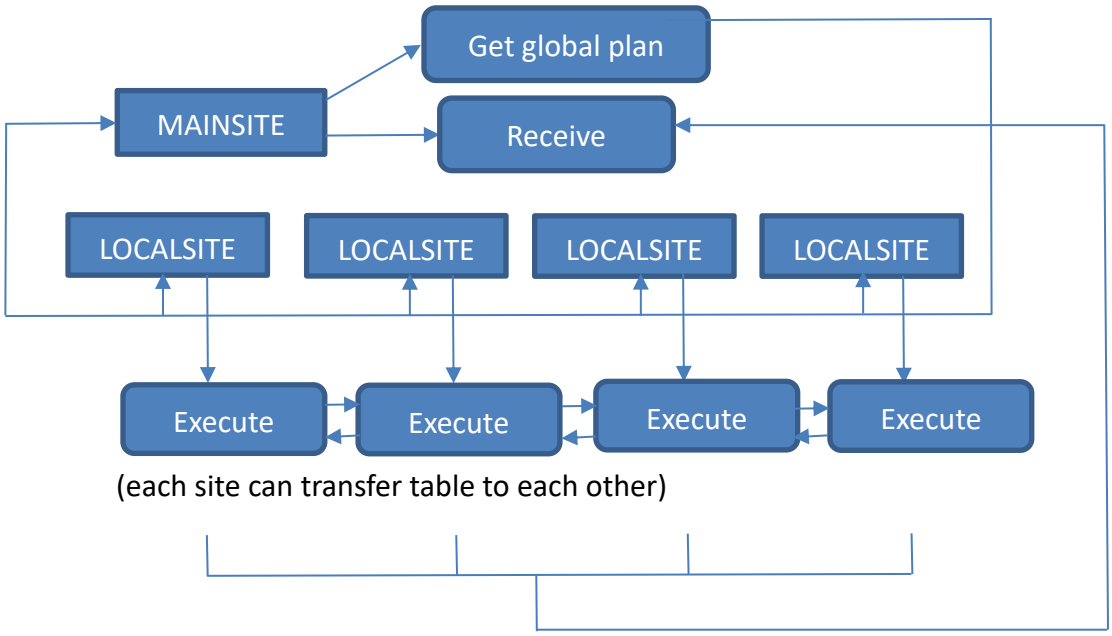


第3章　执行模块

3.1 模块功能

该模块的输入是一棵已经优化好的查询树，执行模块通过查询树得到执行计划，然后各个站点执行计划，将用户查询语句的最终执行结果返回给用户。

3.2 整体架构



（1） 主站点生成执行计划

操作类型：

ESS(sql, temp_table_name)：执行sql语句，新建一个$temp_table_name$，将执行结果存入该表。

SAT(table_name, target_site_id)：得到名为$table_name$的表的所有内容。并将改内容传输给$target_site_id$。

CAR(content, temp_table_name)：接受其他站点发的内容，然后将其存入一个名为$temp_table_name$的分片。

在主站点部署***plan***类和***receive***类，两者分别创建两个线程同时进行，***receive***类负责接受最终结果反馈给用户，***plan***类包含全局计划数组和解析查询树函数，全局计划数组给每一个 local 站点分配一个计划队列。解析查询树函数通过后序顺序，遍历每一个树节点，对于每一个树节点：

A. 如果该节点是**FRAGMENT**属性，跳过。

B. 如果该节点是**SELECT**属性，

那么他的孩子仅有一个分片节点。该分片节点带有$frag_id$属性，通过gdd模块和id可以得到分片的名字$frag_name$和它所在的站点$frag_site_id$。**生成*sql_select*语句：**

$$\boldsymbol{SELECT * FROM\ frag\_name;}$$

通过gdd模块分配一个新的临时表$temp_id$和名字"$temp\_table\_id$"。然后向$\boldsymbol{target_site_id}$的站点队列插入$\mathbf{ESS}(\boldsymbol{sql_select}, "\boldsymbol{temp_table_id}")$。

C. 如果该节点是**PROJECT**属性，

那么他的孩子仅有一个分片节点。该分片节点带有$frag_id$属性和需要投影的属性数组$Column$。通过gdd模块和id可以得到分片的名字$frag_name$和它所在的站点$frag_site_id$。**生成$\boldsymbol{sql_project}$语句：**

$$\boldsymbol{SELECT\ col1, col2, \dots, coln\ FROM\ frag_{name};\ where\ col1, col2, \dots, coln \in Column;}$$

通过gdd模块分配一个新的临时表$temp_id$和名字"$temp\_table\_id$"。然后向$\boldsymbol{target_site_id}$的站点队列插入

$\mathbf{ESS}(\boldsymbol{sql\_project}, "\boldsymbol{temp\_table\_id}")$

如果该节点是**JOIN**属性，

它本身带有 join 列的属性$attr_name$，并且他的孩子有两个分片节点，即两个将要 join 的分片。两个节点带有$frag_id$属性，通过gdd模块和id可以得到分片的名字$frag_name1, frag_name2$和它们所在的站点$frag_site_id1, frag_site_id2$。如果$frag_site_id1 != frag_site_id2$，则通过$gdd$判断$frag_name1$和$frag_name2$两个分片的大小，小的表存放的站点叫$source_site_id$，大的表存放的站点叫$target_site_id$（由于该计划是在执行之前确定的，临时分片无分片大小信息，若其中有分片无大小信息，默认$frag_site_id1$为$source_site_id$，$frag_site_id2$为$target_site_id$），向$\boldsymbol{source_site_id}$站点队列插入：

$\mathbf{SAT}(\mathbf{small\_frag\_content}, \mathbf{target\_site\_id})$。通过$gdd$模块分配一个新的临时表$temp_id$和名字"$temp\_table\_id$"，然后生成$\boldsymbol{sql_join}$语句：

$$\boldsymbol{SELECT * FROM\ frag_{name1}, frag_{name2}\ where\ frag_{name1}.attr = frag_{name2}.attr;}$$

通过gdd模块分配一个新的临时表$temp_id$和名字"$temp\_table\_id$"。然后向$\boldsymbol{target_site_id}$的站点队列插入

$\mathbf{ESS}(\boldsymbol{sql\_join}, "\boldsymbol{temp\_table\_id}")$。

D. 如果该节点是**UNION**属性，

他的孩子有n个分片节点，它们都是需要 union 的分片。每个节点带有$frag_id$属性，通过gdd模块和id可以得到分片的名字$frag_name1, frag_name2, \dots, frag_namen$，和它们在的站点$frag_site_id1, frag_site_id2, \dots, frag_site_idn$。如果这些$id$不全相等，判断每个站点存在表的大小，确定最大的站点为$target_site_id$，然后**对其他站点插入$\mathbf{SAT}(\boldsymbol{frag_i_content}, \boldsymbol{target_site_id})$**，通过$gdd$模块分配一个新的临时表$temp_id$和名字"$temp\_table\_id$"，该操作确保将需要 union 的分片统一传输到一个站点（由于该计划是在执行之前确定的，临时分片无分片大小信息，若其中有分片无大小信息，则存在分片最多的站点为$target_site_id$）。然后**生成$\boldsymbol{sql_union}$语句：**

$$\boldsymbol{SELECT * FROM\ frag_{name1}, frag_{name2}, \dots, frag_{namen};}$$

通过gdd模块分配一个新的临时表$temp_id$和名字"$temp\_table\_id$"。然后**向$\boldsymbol{target_site_id}$的站点队列插入**

ESS(***sql_union***,"***temp_table_id***")。

当树遍历到根节点时，执行发送计划函数，分别把 local 站点的计划队列放松给 local 站点。

（2） Local 站点执行计划

Local 站点创建一个***receive***类，负责接受计划和分片，其中包含了一个***site_execution*** 类，***site_execution*** 类负责解析计划并执行，***site_execution***类包含***MySql***类，***MySql***类通过 c++ mysql connector与mysql数据库连接进行操作。

A. ***receive***类：

该类接受四种类型的信息，通过将第一个字符设置为标识符的方法区分这四类消息。

第一类消息：接受计划，标识符为‘0’，将接收到的信息存储为 Operator 的数组，Operator 是上一节提到的 ESS 和 SAT 操作类型。将这个数组存储为***site_execution***类的执行计划队列。

第二类消息：接受分片，标识符为‘1’，收到这样的消息过后，执行 **CAR**(**content**, **temp_table_name**)操作，其中 content 是接收到的表的内容，*temp_table_name*是*gdd*新分配的临时分配 id 生成的分片名。该操作的内容是新建一个临时分片**temp_table_name**，将接收到的分片存储进该分片中。将临时分片名插入***site_execution***类的分片队列。

第三类信息：接受结果，标识符为‘1’，该消息仅限于 local 站点发送给主站点，其功能是将最终结果的分片发送给主站点，主站点将其打印出来反馈给用户。主站点收到这样的消息过后，执行 **CAR**(**content**, **result_table_name**)操作。

第四类消息：接受删除临时分片命令，标识符为‘2’，收到这样的消息过后，执行*MySql*类的*release_all_temp_table*函数，删除本地数据库中所有的临时分配。

该类每收到一个消息创建一个新线程，该类只负责接受消息并分类，分别调用功能封装好的***site_execuion***类函数。

B. ***site_execution***类:

该类维护两个个队列：计划队列，其中包括待执行的操作，为 Operator 类型；分片队列，其中包括该站点已有的分片。

该类包括*check_plan*函数，该函数扫描一次计划队列，并核对分片队列，返回可执行 Operator 的数组。

该类包括*execute*函数，该函数的输入是*check_plan*执行后的结果，得到可执行的计划数组，如果数组长度大于 0，则一一执行，执行结束过后，再次调用*check_plan*。

C. ***MySql***类

该类包括三类函数 execute_select_sql, select_all_table, create_and_receive，分别是三种 Operator 的底层执行。另外，该类包括release_all_temp_table来删除本地所有的临时分片。

如果 Operator 是 ESS，则执行execute_select_sql，如果是 SAT，则执行

select_all_table，如果是 CAR，则执行create_and_receive

3.3 代码结构

plan类

QueryOptimise.h

QueryOptimise.cpp

receive类

rpc_receive.h

rpc_receive.cpp

site_execution类

site_execution.h

site_execution.cpp

MySql类

local_sql_execution.h

local_sql_execution.cpp

结构体:

```
enum OPERATORTYPE {
    ESS, SAT, CAR,
};

struct Operator{
    int id;
    string content;
    OPERATORTYPE ope;
    int result_frag_id;
    int target_site_id;
    int is_end ;
    vector<string> table_names;

    // select * from table where...-> ESS;
    // table_name, target_frag_id -> SAT;
};
```

第4章　通信模块

分布式数据库系统与单机数据库系统的一个重大的不同在于通信，各个节点之间需要合理的通信机制传输执行计划、查询所需数据以及执行结果。在这一模块中，我们先后尝试了使用 RPC 和 socket 两种通信手段。

4.1 RPC

我们首先尝试了使用 RPC 进行分布式系统的通信。RPC（ Remote Procedure Call ）是分布式系统通信处理的事实标准,实现消息传输的透明性。指用户可以像调用本地过程一样调用不同地域的不同计算机上的过程，从而使得应用程序设计人员不必设计和开发有关发送和接收信息的实现细节。

以下展示 RPC 的使用实例：

在每个客户端上需要启动 RPC Server 的服务，持续监听指定的端口，等待其他节点的调用，当其他节点通过“ReceiveTable”调用 RPC Server 服务时，服务端执行 rpc_receive 类的 ReceiveTable 成员函数，并按照 ReceiveTable 的返回值类型将结果返回给远程的调用者。

```
void startListening(int port){
    std::cout << "StartListening" << std::endl;
    std::cout << port << std::endl;
    rpc::server srv(port);
    //rpc_receive s(1);
    srv.bind( name: "ReceivePlan",&rpc_receive::ReceivePlan);
    srv.bind( name: "ReceiveTable",&rpc_receive::ReceiveTable);

    srv.suppress_exceptions( suppress: true);
    std::cout << "srv.run()" << std::endl;
    srv.run();
}
```

发起远程调用的节点通过 client(target_site_ip,target_port)与指定 ip 与端口的远程 RPC Server 服务建立联系；通过 client.call 调用远程 RPC Server 的某一个函数，并将所需参数一并传递过去，使用 as< >()指定返回值的类型，并接收返回值。

```
bool SendTable(int frag_id, string frag_content, int target_site_id)
{
    cout <<  "start SendTable \t" << endl;
    string target_site_ip = mapIdtoIp(target_site_id);
    int target_port=8080;
    rpc::client client(target_site_ip,target_port);
    bool ok = client.call("ReceiveTable",frag_id, frag_content).as<bool>();
    cout <<  "return ok\t" << ok  << endl;
    return ok;
}
```

以上展示的是 RPC 的基本原理与使用方法，在实际使用过程中，需要将数据转化为字符串进行传递，以下展示如何将一个 Operator 类型的 vector 转化为 string 进而通过 RPC 发送给 RPC Server 端。

```
bool SendPlan(vector<Operator> plan, int target_site_id)
{
    string results = "";
    for(int i=0; i<plan.size();i++){
        //Operator -> string
        string plans = "";
        plans.append(plan[i].content);
        plans.append( s: "#");
        string ope=EnumToString(plan[i].ope);
        plans.append(ope);
        plans.append( s: "#");
        plans.append(to_string(plan[i].result_frag_id));
        plans.append( s: "#");
        plans.append(to_string(plan[i].target_site_id));
        plans.append( s: "#");
        plans.append(to_string(plan[i].is_end));
        plans.append( s: "#");
        for(int j=0; j<plan[i].table_names.size(); j++){
            plans.append(plan[i].table_names[j]);
            if(j != plan[i].table_names.size()-1)
                plans.append( s: "#");
        }
        results.append(plans);
        cout <<  "SendResults :\t" << results <<endl;
        if(i != plan.size()-1)
            results.append( s: "$");
    }
    //call
    string target_site_ip=mapIdtoIp(target_site_id);
    int target_port=8080;
    rpc::client client(target_site_ip,target_port);
    bool ok = client.call("ReceivePlan",results).as<bool>();
    cout <<  "return ok\t" << ok  << endl;
```

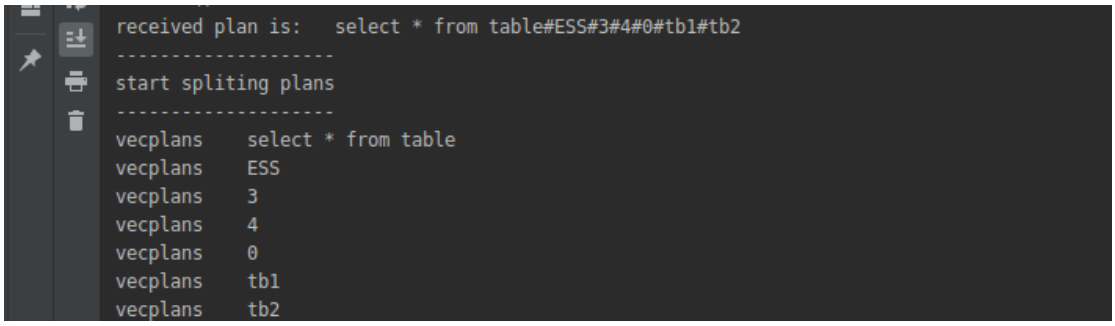
RPC Server 端开始监听指定端口 8080

```
/home/dyj/CLionProjects/DDBclass3/cmake-build-debug/DDBclass3
Hello, World!
StartListening
8080
srv.run()
```

调用节点将拼接好的字符串传递给远程函数

```
/home/dyj/CLionProjects/DDBclass2/cmake-build-debug/DDBclass2
start SendPlan
SendPlanResults :   select * from table#ESS#3#4#0#tb1#tb2
```

远程 RPC Server 将接收到的 plan 重新按照规定好的格式重新解析，并存储在相应的结构体里，方便后续算法的使用。



4.2 socket

但是在深入了解后，并在大量实验验证的基础上，我们认为 RPC 不适用于我们的设计理念，理由如下：RPC 的设计是将执行结果返回给进程的远程调用者，即远程函数的返回值是需要传递给调用者的。但是面对不需要将执行结果返回给调用者，而是发送给另一个 slave 站点的情况，RPC 使用起来就会使得步骤冗余繁杂。并且，RPC 的返回值被调用者占用，使得此函数接收并重新拼接形成的执行计划结构体无法传递给别的函数进一步使用。我们起初使用了 C++类函数来解决这一问题，但是 RPC 在调用远程 C++类函数的语法上又出现了问题，经过一定时间的研究，发现解决起来有困难，并且由于此问题发现的时机很晚了，出于时间关系来不及解决。相比之下，传统的 socket 使用更加方便，且实现原理更加符合我们的设计理念。

因此经过我们的讨论，我们一致决定，使用传统的基于 TCP 协议的 socket 进行通信。网络上的两个程序通过一个双向的通信连接实现数据的交换，这个连接的一端称为一个 socket。建立网络通信连接至少要一对端口号(socket)。socket 本质是编程接口(API)，对 TCP/IP 的封装，TCP/IP 也要提供可供程序员做网络开发所用的接口，这就是 Socket 编程接口；HTTP 是轿车，提供了封装或者显示数据的具体形式；Socket 是发动机，提供了网络通信的能力。

根据连接启动的方式以及本地套接字要连接的目标，套接字之间的连接过程可以分为三个步骤：服务器监听，客户端请求，连接确认。

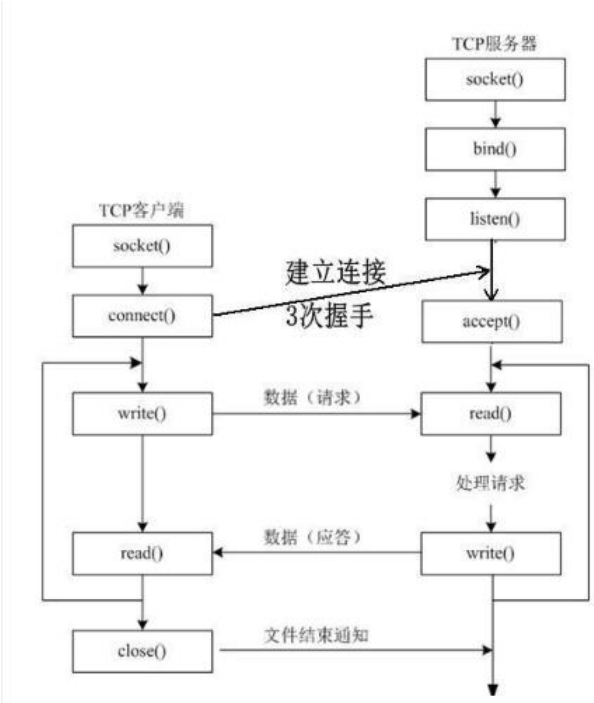
（1）服务器监听：是服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态。

（2）客户端请求：是指由客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套

接字，指出服务器端套接字的地址和端口号，然后就向服务器端套接字提出连接请求。

（3）连接确认：是指当服务器端套接字监听到或者说接收到客户端套接字的连接请求，它就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，连接就建立好了。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

TCP（Transmission Control Protocol，传输控制协议）是一种面向连接的、可靠的运输层协议，为用户提供可靠的、全双工的字节流服务。TCP 是可靠的，并且使用确认机制。UDP（User DataGram Protocl，用户数据报协议）是一种简单的面向数据报的传输层协议。是一种无连接协议，不保证数据报一定能到达目的地，不具有可靠性，不支持确认和重发。我们的系统要求通信是可靠的，因此我们使用基于 TCP 的 socket。



Socket 服务器端监听客户端的连接。

```
void startListening(int port){
    struct sockaddr_in server_addr;
    bzero(&server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htons(INADDR_ANY);
    server_addr.sin_port = htons(port);

    int server_socket = socket(PF_INET, type: SOCK_STREAM, protocol: 0);
    if (server_socket < 0)
    {
        printf("Create Socket Failed!\n");
        exit( status: 1);
    }
    if (bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)))
    {
        printf("Server Bind Port: %d Failed!\n", port);
        exit( status: 1);
    }
    // server_socket用于监听
    if (listen(server_socket, n: 20))
    {
        printf("Server Listen Failed!\n");
        exit( status: 1);
    }
```

当有新的请求连接时，服务器开启新的线程。

```
while(1)
{
    printf("waiting for new connection...\n");
    pthread_t thread_id;
    int new_server_socket = accept(server_socket, (struct sockaddr*)&client_addr, &length);
    if (new_server_socket < 0)
    {
        printf("Server Accept Failed!\n");
        continue;
    }
    printf("A new connection occurs!\n");
    if (pthread_create(&thread_id, attr: NULL,&Data_handle,(void *)(&new_server_socket)) == -1)  //创建一个线程
    {
        fprintf(stderr, format: "pthread_create error!\n");
        break;                                  //结束循环
    }
}
```

客户端请求服务器端，向服务器端发送数据

```
void socket_client(string target_site_ip,string results)
{
    char **argv;
    argv = new char*[1];
    argv[0] = new char[255];
    strcpy( dest: argv[0],target_site_ip.c_str());
    cout << "target_site_ip.c_str()"<< argv[0]<<endl;
    struct sockaddr_in client_addr;
    bzero(&client_addr, sizeof(client_addr));
    client_addr.sin_family = AF_INET; // internet协议族
    client_addr.sin_addr.s_addr = htons(INADDR_ANY); // INADDR_ANY表示自动获取本机地址
    client_addr.sin_port = htons( hostshort: 0);
    int client_socket = socket(AF_INET,  type: SOCK_STREAM,  protocol: 0);
    if (client_socket < 0)
    {
        printf("Create Socket Failed!\n");
        exit( status: 1);
    }
    if (bind(client_socket,(struct sockaddr*)&client_addr, sizeof(client_addr)))
    {
        printf("Client Bind Port Failed!\n");
        exit( status: 1);
    }
    struct sockaddr_in  server_addr;
    bzero(&server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    if (inet_aton( cp: argv[0], &server_addr.sin_addr) == 0)
    {
        printf("Server IP Address Error!\n");
        exit( status: 1);
    }
    server_addr.sin_port = htons(SERVER_PORT);
    socklen_t server_addr_length = sizeof(server_addr);
    if (connect(client_socket, (struct sockaddr*)&server_addr, server_addr_length) < 0)
    {
        printf("Can Not Connect To %s!\n", argv[0]);
        exit( status: 1);
    }
    char buffer[BUFFER_SIZE];
    bzero(buffer, sizeof(buffer));
    strncpy(buffer, results.c_str(),  n: results.length()  > BUFFER_SIZE ? BUFFER_SIZE : results.length());
    send(client_socket, buffer, BUFFER_SIZE,  flags: 0);
    close(client_socket);
}
```

但是在真正合并到系统中使用的时候，我们发现 socket 还有着缺陷，传输的数据量有限，并且速度很慢，这也导致了整个系统的功能以及性能。

第5章　环境部署

详细总结记录下在系统环境配置的过程。

5.1 gcc 安装

直接通过 sudo yum install gcc, 发现其版本过低，后进行升级。CentOS 6.5 升级 GCC G++ (当前最新 GCC/G++版本为 4.4.7)没有便捷方式，yum update.... yum install 或者 添加 yum 的 repo 文件 也不行, 只能更新到 4.4.7，所以只能下载源代码手动编译安装，并且由于对其版本没有深究，在后续的使用过程中还是出现版本不合适的问题，多次进行了升级操作。

```
线程模型： posix
gcc 版本 4.4.7 20120313 (Red Hat 4.4.7-23) (GCC)
[DDB_homework@fj-chensibei-5 gcc-4.9.4-build-temp]$ gcc --version
gcc (GCC) 4.4.7 20120313 (Red Hat 4.4.7-23)
Copyright © 2010 Free Software Foundation, Inc.
```

gcc 升级到 4.9.4

```
[DDB_homework@fj-chensibei-5 ~]$ gcc -v
使用内建 specs。
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/local/libexec/gcc/x86_64-unknown-linux-gnu/4.9.4/lto-wrapper
目标： x86_64-unknown-linux-gnu
配置为： ../gcc-4.9.4/configure --enable-checking=release --enable-languages=c,c++ --disable-multilib
线程模型： posix
gcc 版本 4.9.4 (GCC)
[DDB_homework@fj-chensibei-5 ~]$
```

后面将 gcc 升级到 6.1.0，项目进行的比较顺利，现将过程详细记录如下

1. 获取安装包并解压

wget http://ftp.gnu.org/gnu/gcc/gcc-6.1.0/gcc-6.1.0.tar.bz2

tar -jxvf gcc-6.1.0.tar.bz2

2. 下载供编译需求的依赖项、配置安装依赖库

cd gcc-6.1.0

./contrib/download_prerequisites

3. 建立一个目录供编译出的文件存放

mkdir gcc-build-6.1.0

cd gcc-build-6.1.0

4. 生成 Makefile 文件

../configure -enable-checking=release -enable-languages=c,c++ -disable-multilib

5. 编译

make -j4

（-j4 选项是 make 对多核处理器的优化，plus 此步骤非常耗时）

6. 安装并查看 gcc 是否安装成功及其版本

sudo make install

ls /usr/local/bin | grep gcc

gcc -v

升级 gcc，生成的动态库没有替换老版本 gcc 的动态库

源码编译升级安装了 gcc 后，编译程序或运行其它程序时，有时会出现类似 /usr/lib64/libstdc++.so.6: version `GLIBCXX_3.4.21' not found 的问题。这是因为升级 gcc 时，生成的动态库没有替换老版本 gcc 的动态库导致的，将 gcc 最新版本的动态库替换系统中老版本的动态库即可解决。

// 运行以下命令检查动态库：

strings /usr/lib64/libstdc++.so.6 | grep GLIBC

从输出可以看出，gcc 的动态库还是旧版本的。说明出现这些问题，是因为升级 gcc 时，生成的动态库没有替换老版本 gcc 的动态库。

// 执行以下命令，查找编译 gcc 时生成的最新动态库：

find / -name "libstdc++.so*"

将上面的最新动态库 libstdc++.so.6.0.22 复制到/usr/lib64 目录下

cd /usr/lib64

cp

/root/Downloads/gcc-6.1.0/gcc-build-6.1.0/stage1-x86_64-pc-linux-gnu/libstdc++-v3/src/.libs/libstdc++.so.6.0.22 ./

// 删除原来软连接：

rm -rf libstdc++.so.6

// 将默认库的软连接指向最新动态库：

ln -s libstdc++.so.6.0.22 libstdc++.so.6

// 默认动态库升级完成。重新运行以下命令检查动态库：

strings /usr/lib64/libstdc++.so.6 | grep GLIBC

可以看到 输出有"GLIBCXX_3.4.21" 了

gcc 升级到 6.1.0 基本可以满足项目对 gcc 的要求，但后面在运行 jdbc 时还是出现问题，无奈还需升级，由于下载源码编译时间太长了，最后选择了 scl 这个软件集合。

```
$ sudo yum install centos-release-scl

$ sudo yum install devtoolset-7

$ scl enable devtoolset-7 bash
```

第三步就是使新的工具集生效，这时用 gcc --version 查询，可以看到版本已经是 7.3.1 了

```
[DDB_homework@fj-chensibei-7 ~]$ gcc --version
gcc (GCC) 7.3.1 20180303 (Red Hat 7.3.1-5)
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

[DDB_homework@fj-chensibei-7 ~]$ which gcc
/opt/rh/devtoolset-7/root/usr/bin/gcc
[DDB_homework@fj-chensibei-7 ~]$ whereis gcc
gcc: /usr/bin/gcc /usr/lib/gcc /usr/local/bin/gcc /usr/local/lib/gcc /usr/libexec/gcc /usr/share/man/man1/gcc.1.gz
[DDB_homework@fj-chensibei-7 ~]$
```

5.2 部署 ETCD

ETCD　被设计为高可用强一致的 Key-Value 小型数据存储。etcd 是一个分布式的可靠的键值存储系统，它是用于存储分布式中关键数据的。通常 etcd 是由三个或五个节点组成，多个节点直接通过 Raft 一致性算法的方式来完成分布式数据一致性协同。多个节点之间，算法会选举出一个主节点作为 leader，由 leader 负责数据的同步与分发。若 leader 发生故障，系统会自动选取另一个成为 leader，并重新完成数据的同步与分发。

// 下载并解压

wget

https://github.com/coreos/etcd/releases/download/v3.2.6/etcd-v3.2.6-linux-amd64.tar.gz

tar -zxvf etcd-v3.2.6-linux-amd64.tar.gz -C /opt/

// 编辑环境变量

sudo vi /etc/profile

export ETCDCTL_API=3

source /etc/profile

// 创建配置文件目录

sudo mkdir /etc/etcd

// 创建 etcd 配置文件

sudo vi /etc/etcd/conf.yml

// 删除 data 中的内容

sudo rm -rf data

--添加如下内容

name: etcd-1

data-dir: /opt/etcd-v3.2.6-linux-amd64/data

listen-client-urls: http://10.77.70.127:2379,http://127.0.0.1:2379

advertise-client-urls: http://10.77.70.127:2379,http://127.0.0.1:2379

listen-peer-urls: http://10.77.70.127:2380

initial-advertise-peer-urls: http://10.77.70.127:2380

initial-cluster: etcd-1=http://10.77.70.127:2380,etcd-2=http://10.77.70.126:2380

initial-cluster-token: etcd-cluster-token

initial-cluster-state: new

name: etcd2

data-dir: /opt/etcd/data

listen-client-urls: http://10.77.70.126:2379,http://127.0.0.1:2379

advertise-client-urls: http://10.77.70.126:2379,http://127.0.0.1:2379

listen-peer-urls: http://10.77.70.126:2380

initial-advertise-peer-urls: http://10.77.70.126:2380

initial-cluster: 'etcd1=http://10.77.70.127:2380, etcd2=http://10.77.70.126:2380'

initial-cluster-token: 'etcd-cluster-1'

initial-cluster-state: 'existing'

//启动

/~/opt/etcd-v3.2.6-linux-amd64/ sudo .etcd --config-file=/etc/etcd/conf.yml

sudo ./etcd --config-file=/etc/etcd/conf.yml

5.3 安装 mysql

本系统使用 mysql 作为数据库，通过 yum 安装 mysql

//查看 yum 服务器上 mysql 数据库的可下载版本信息：

yum list | grep mysql

//安装

yum install -y mysql-server mysql mysql-devel

```
Installed:
  mysql.x86_64 0:5.1.73-8.el6_8                 mysql-server.x86_64 0:5.1.73-8.el6_8

Dependency Installed:
  mysql-libs.x86_64 0:5.1.73-8.el6_8              perl-DBD-MySQL.x86_64 0:4.013-3.el6
  perl-DBI.x86_64 0:1.609-4.el6

Complete!
```

//查看刚安装好的 mysql-server 的版本

rpm -qi mysql-server

```
[DDB_homework@fj-chensibei-5 ~]$ rpm -qi mysql-server
Name        : mysql-server                 Relocations: (not relocatable)
Version     : 5.1.73                            Vendor: CentOS
```

mysql 数据库的初始化及相关配置

//启动 mysql 服务。

service mysqld start

```
Please report any problems with the /usr/bin/mysqlbug script!

                                                           [  OK  ]
Starting mysqld:                                           [  OK  ]
```

// 为 root 账号设置密码

mysqladmin -u root password 'ddb09890'

//通过 mysql -u root -p 命令来登录我们的 mysql 数据库

```
[DDB_homework@fj-chensibei-5 ~]$ mysqladmin -u root password 'ddb09890'
[DDB_homework@fj-chensibei-5 ~]$ mysql -u root -p
Enter password:
```

5.4 安装 Jsoncpp

安装部署 json 工具时也绕了弯，几次都不成功，无法解析 json 格式的数据。最后成功过程记录如下

1. 安装 scons

下载地址：

http://sourceforge.net/projects/scons/files/scons/2.1.0/scons-2.1.0.tar.gz/download

解压：tar -zvxf scons-2.1.0.tar.gz

进入到解压目录 scons-2.1.0，执行命令：sudo python setup.py install

2. 安装 Jsoncpp

下载地址：http://sourceforge.net/projects/jsoncpp/

解压：tar -zvxf jsoncpp-src-0.5.0.tar.gz

进入到 jsoncpp 解压目录下，执行命令：sudo scons platform=linux-gcc

```
128重新开始
还要以下两步操作:

将/jsoncpp-src-0.5.0/include/目录下的json文件夹拷贝到 /usr/local/include/下

👉👉 sudo cp -r /home/DDB_homework/jsoncpp-src-0.5.0/include/json /usr/local/
include/

将jsoncpp-src-0.5.0/libs/
linux-gcc-4.9.1/目录下的libjson_linux-gcc-4.9.1_libmt.a拷贝到 /usr/local/lib/
下，并为了方便使用，将其重命名为libjsoncpp.a

👉👉 sudo cp /home/DDB_homework/jsoncpp-src-0.5.0/libs/linux-gcc-6.1.0/
libjson_linux-gcc-6.1.0_libmt.a /usr/local/lib/

为了方便使用，将其重命名为libjsoncpp.a
👉👉 sudo mv libjson_linux-gcc-6.1.0_libmt.a libjsoncpp.a


到此，配置已经完成，只需要在代码中添加头文件：#include <json/json.h>即可。
```

另外需要安装部署的工具 curl, boost，可直接通过 yum 来安装.

5.5 安装 curl

sudo yum install curl-dev
sudo yum install curl-devel

5.6 安装 boost

sudo yum install boost
sudo yum install boost-devel　boost-doc

第6章 任务分工

成员	分工
刘同禹	ETCD 设计、查询树的构建及优化
陈思蓓	执行模块
郝新丽	通信模块
时青	环境部署

第7章 总结

刘同禹：

对于这次大作业的完成结果，我是非常不满意的，因为我们没有很好地把整个分布式数据库的查询流程给跑通，甚至连一些基础的 SQL 输入界面都没来得及完成，这也从侧面反映了我在系统设计和实现方面还有很多不足，得在今后的学习中重视这一方面的学习和锻炼。

但总体来看，我还是从这次大作业的任务中收获到了很多东西。首先是捡起了荒废很久的 C++，对这一偏底层的编程语言（与 python 相比）重新熟悉了一些，并且由于我要完成查询树的建立和优化工作，因此加深了许多对与数据结构相关内容的学习，尤其是对树的相关操作和递归遍历的相关思想。

除此之外，我还负责了在 etcd 中读取数据的接口的实现，因此，熟悉了之前从未接触过的 etcd 这一集群数据管理工具，算是学习到了一个新的东西。而更重要的，则是为了能够使我们的代码能够编译通过，我详细去了解了一些 C++ 编译时链接库与头文件的相关知识，并学会了该怎么写 makefile 来引入外部的 package，这一整个过程让我对系统部署更加熟悉，下次再碰到类似的任务应该不会再因为从未接触过而感到茫然无措了吧。

但是由于没有把老师布置的任务完整地完成，还是有不少的遗憾，在之后的合作任务中，我会吸取这次大作业的教训，避免同样的情况再次发生。

陈思蓓：

(1) 编码能力方面:

在编写程序时,我有三方面的不成熟，希望在后续实践中提升能力。

第一点是 Linux 编写 c++环境配置，在 Linux 服务器上我不了解 c++方便使用的 ide。于是使用的是最原始的 makefile 和 print 方法 debug，这样大大的减小了我们的编程效率。在韩寒师哥的帮助下，我了解了远程桌面的功能，并且可以使用 CLION，但是由于了解的时间比较晚，我们换不过来了，下次进行开发的时候可以使用这种方法来改进我们的原始方法。第二点是我们对第三方包并不了解，在编程使用它们之前，应该详细的阅读文档，发现他们都优缺点和功能。例如使用 c++ mysql connector 插入功能较慢，没有批量插入的功能，这影响了我们代码的速度，这应该在完成编写代码之前了解清楚。第三方面是我们对第三方包的编译不了解，第一个原因是第一点提到的我们没有用上好的 ide 工具，第二个原因是我们编译环境的能力不足，应该在这方面着重锻炼，而不是唯一注重编写代码的能力。

(2) 团队计划方面

我们的团队没有中心领导者，这是我们在最开始的时候的决策失误。由于没有领导者监督，我们没有定下一个完整的时期计划，导致某一项工作拖延了很长时间，甚至导致拖慢了整个进度，而后续工作草草完成，这是导致现状的最重要因素之一。我们应该对这个大型工程的时期划分明确，并定下若干个 DDL，保证在每个 DDL 之前完成任务。我们这次由于编译第三方库，执行模块与通信模块对接工作拖延了很长时间，导致临近检查日系统才完成通信，执行模块才能完成编译并 debug，这导致后续问题出现没有时间解决，这个问题就是通信使用 socket 信息截断的问题。

(3) 团队协作方面

由于我们对团队成员各自的能力了解不足，导致工作分配不合理，这个问题在最后阶段我们才有所察觉，这导致了我们的整体进度不能如其运作。我们应该在开始工作之前了解团队成员的能力，并且划分出适合自己的工作计划。在接口对接方面，应该互相明确代码分工，而不能随意修改，导致最后出现混乱的情景。

郝新丽：

这门课真的使我受益匪浅。

上第一节课的时候，我发现选课人数不多，经验告诉我，这门课可能真的很硬核，我也有一丝动摇。但是在我听完一节课之后，我决定一定要选范老师的课。事实证明我的选择是对的。那些临阵脱逃的同学，你们真的是损失了。

我超级喜欢范老师的讲课风格，老师讲课深入浅出，生动有趣，每节课讲的内容我都可以当堂理解以及吸收，在老师的带领下，对分布式数据库有了一些基础且必要的认识。除此以外，老师还有一些前沿讲座课，比如众包数据库等，同样让我受益匪浅。

并且在做大作业的过程中结识了一些优秀的伙伴，我们共同努力，相互沟通，相互合作，共同完成了大作业，使我在编程、工程以及团队合作方面都有所收获。与其他老师不同，范老师会留出很多时间在课上和大家讨论大作业，我们可以及时向老师汇报这一周的进度，老师根据目前的情况，对下一步的方向进行指导，在这个过程中，由于能够和老师进行充分的交流以及请教，效率有所提高，能力也有所提升。

但在这个过程中，在系统编码以及多人协作方面我深刻认识到自己的不足。首先是自己解决的问题能力不足，动手能力不够，其次，在团队合作过程中，缺乏一个有经验的全局把控者，并且在合作后期，存在着互相沟通不顺畅的情况，我没有收到关于 socket 通信应用到整个系统中出现问题的相关反馈。

总而言之，这次大作业让我有所反思，对我有所激励，以后一定要吸取本次在团队合作以及工程编码方面的教训，向更优秀的同学学习并不断进步。

时青：

有机会能上分布式数据库这门课，感谢范老师一学期认真备课和辛苦教学，老师总是以非常好的方式对同学们进行指导和鼓励；同时十分很珍惜和其他同学一起上课，共同完成大作业的过程，感恩老师和小组其他同学在学习过程中给予的鼓励和帮助。首先对于课堂学习部分，虽尽量做到课前预习准备，课后再回顾看看，感觉老师课上讲的东西也听的挺明白，但通过课堂小测发觉自己做的最慢，也没有答出对问题有效的解决方案。回顾来看，可能自己感觉的“听到明白”是老师讲的好，但并不代表自己已经把知识理解消化了，还需要自己多思考以及不懂的地方课下多向其他同学请教。

对于大作业我做的部分主要是环境部署，之前对 Linux 系统非常不熟悉，整个过程中熟悉了 Linux 系统和如何在 Linux 上安装软件以及命令行的使用，过程中也花时间看完了一本 The Linux command line。在安装过程中走了不少弯路，开始时真的弄不清楚自己要做什么，也不好意思多请教。在做自己的部分任务时，

盲目找教程，经常弄很久依然漏洞百出。想想其实这些试错过程本就该经历，以前实操的机会太少了，这些也是宝贵的经验，但进一步想了一下，以后应该先补一下相关内容的基本知识和原理，遇到问题多多总结，这样的话收获应该更多。