DDBMS 期末报告

小组：DDBMS200

组长：涂荐泓　组员：李晓桐、罗尹清

日期：2020 年 12 月

一、项目要求

本项目以 MySQL 为本地数据库，利用网络传输技术将不同站点的数据库连接起来，实现构建分布式数据库，最终提供便捷的用户接口。用户只需输入类似 SQL 的语句，即可对该分布式数据库进行操作。下面是项目要求的详细描述：

1、功能要求

实现分布式数据库系统的基本功能，包括但不限于：

（1）初始化数据库：维护元数据。

（2）新建表格/删除表格：维护元数据、根据分片规则在不同站点建立对应表格。

（3）Load 数据：向各站点传输数据并存储到相应表格内。

（4）元数据（metadata）的管理

（5）基础查询、插入、删除数据操作

（6）查询优化

2、配置要求

站点设置：3 台机器，4 个站点，P2P

二、项目环境

1、实验环境

系统：3 台 WSL，Ubuntu18

语言：C++

编码：VSCode

网络传输：gRPC

元数据管理：ETCD

本地数据库：MySQL

2、项目地址

https://github.com/tuhahaha/ddbms200.git

三、项目分工

按照整个工程的功能需求，我们将整个任务划分为 4 个板块，分别由三位同学负责：

（1）Parser 模块 ：李晓桐

（2）Metadata 模块 ：涂荐泓

（3）执行模块：罗尹清

（4）传输模块：涂荐泓

下图较为清晰的展示了各个模块的链接关系。

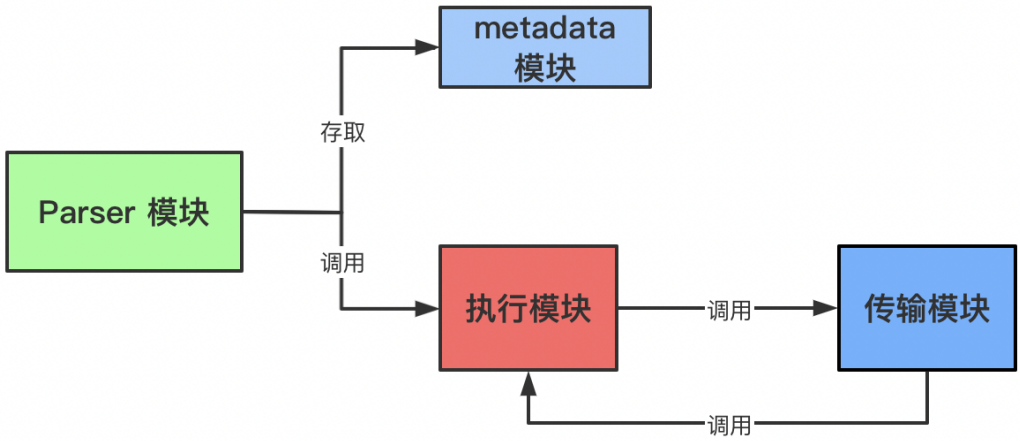

图 1：模块调用关系

四、模块设计

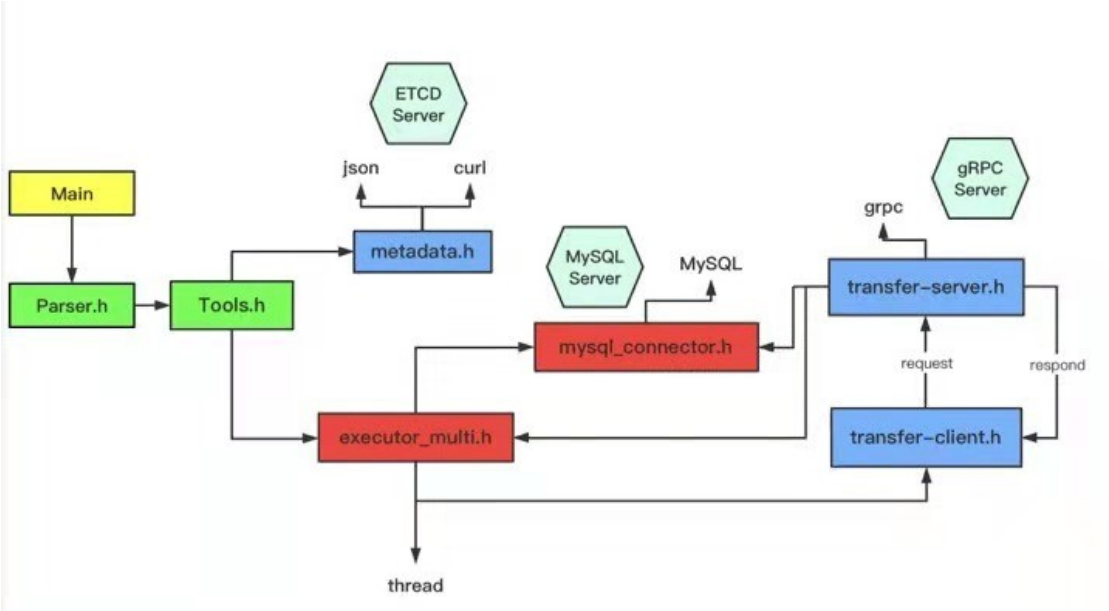

图 2：具体模块调用关系

模块之间具体的调用关系见图 2，从主函数（对于整个 DDBMS 系统而言的客户端）输入 SQL 语句，该语句经过绿色的 Parser 模块进行解析，而 Parser 会调用 metadata 帮助生成具体的执行计划。成型的执行计划会被传给多线程执行模块 executor_multi 进行具体执行，而 Executor_multi 为了处理底层数据存取与关系表运算，会调用 mysql_connector 来使用 MySQL 服务，如果涉及不同站点的数据存取和传输，Executor_multi 会调用 transfer-client，从而让 transfer-server 在另外的站点继续执行 executor_multi 中的某个函数并返回结果到本站点。如此 executor_multi 最后把结果返回给主函数（客户端），完成所输入的 SQL 语句。

1、Parser 模块

实现功能层面上，Parser 模块实现 8 个功能：CREATE TABLE, CREATE FRAGMENTATION,

LOCAL LOAD, LOCAL，CREATE EMPTY TABLE, SELECT, INSERT, DELETE, DROP。其中 CREATE TABLE 可以在 ETCD 中建立数据信息的元数据，CREATE FRAGMENTATION 在 ETCD 中建立数据分片信息的元数据，LOCAL LOAD 将本地数据按照分片方法分别存储在不同站点上的数据库中，LOCAL CREATE EMPTY TABLE 在分站点建立空数据分片，SELECT 执行选择语句，INSERT 执行插入指令，DELETE 执行删除指令和 DROP 执行删除数据分片指令。

实现这些功能依赖于 parser 文件夹中的 Parse_INIT.cpp, Parse_LOAD.cpp, Parse_DELETE.cpp, Parse_INSERT.cpp, Parse_SELECT.cpp,Parse.cpp 则调用了前面 5 个 c++文件中定义的一系列功能实现函数。特别的，一些通用的函数如：遍历一个 vector<string>类型的数据结构，在多个功能的实现中都用使用，因此额外创建了 Tools.cpp 代码文件用于实现常见功能。

文件	函数

Parse_INIT.cpp	`// Create table`
	`string InitGetTableCreateTable(string sql_statement);`
	`vector<ColumnDef> InitGetColumnsCreateTable(string sql_statement);`
	`GDD InitGetGDDCreateTable(string sql_statement);`
	`// Create Fragment`
	`string InitGetTableCreateFragmentation(string sql_statement);`
	`string InitGetFragmentTypeCreateFragmentation(string sql_statement);`
	`vector<FragDef> InitGetFragDefCreateFragmentation(string sql_statement);`
	`int InitGetFragNumCreateFragmentation(string sql_statement);`
	`Fragment InitGetFragmentCreateFragment(string sql_statement);`
	`string InitGetFragsColumn(string condition);`
	`string InitGetColumnFromCondition(string condition);`

Parse_LOAD.cpp	`// local create`
	`int GetSite(string sql_statement);`
	`string GetLocalCreate(string sql_statement);`
	
	`string GetSitename(string sql_statement);`
	`// local load ALLDATA`
	`string GetPathFromLocalLoad(string sql_statement);`
	`string GetTableFromLocalLoad(string sql_statement);`
	`string GetFragType(string sql_statement);`
	
	`string GetLoadSql(string sql_statement);`
	`// local select and load`
	`string GetSelectSqlFromLoadLocalData(string sql_statement);`
	`string GetToTable(string sql_statement);`

	vector<string> GetSiteNames(vector<string> sql_statements);
	vector<string> GetSqls(vector<string> sql_statements);
	vector<string> GetTableNames(vector<string> sql_statements);

Parse_DELETE.cpp	string GetSelectFromDelete(string sql_statement);
	vector<string> GetDeleteSqlH(string sql_statement, Fragment fragment);
	vector<string> GetDeleteSiteH(string sql_statement, Fragment fragment);
	vector<string> GetsqlDelete(string sql_statement);
	vector<string> GetsitenameDelete(string sql_statement);
	vector<string> GetColumnListFromDelete(string sql_statement);
	string GetTableListFromDelete(string sql_statement);
	vector<string> GetValueListFromDelete(string sql_statement);
	bool JudgeDelete(vector<string> condition_list, vector<string> condition_column_list, vector<string> insert_column_list, vector<string> insert_value_list);
	string GetSelectSql(string sql_statement);

Parse_INSERT.cpp	vector<string> GetsqlInsert(string sql_statement);
	vector<string> GetsitenameInsert(string sql_statement);
	vector<string> GetPure(vector<string> column_list_get);

Parse_SELECT.cpp	vector<string> GetCollumnsSelect(vector<string> table_list,vector<TCC> TCCList);
	TCC GetTCC(string table_name, vector<string> column_list, vector<string> select_list);
	vector<TCC> GetTCCList(vector<string> table_list, vector<string> column_list, vector<string> select_list);
	vector<string> GetSelectConditionList(vector<string> condition_list, vector<string> table_list);
	vector<string> GetJoinConditionList(vector<string> condition_list, vector<string> table_list);
	bool JudgeNotin(vector<string> BigList, string item);
	bool JudgeNotinInt(vector<int> BigList, int item);
	void TraverseTCCList(vector<TCC> TccList);
	string GetTableFromSelectCondition(string select_condition);
	vector<string> GetTableFromJoinCondition(string join_condition);
	vector<int> GetNodeListFromTableList(vector<string> table_in_join_condition, vector<TABLE> TableList);

	vector<TCC> GetTCCListTest(string sql_statement);
	// vector<NODE> GetData(string sql_statement, int treeid);
	TREE SELECT(string sql_statement, int treeid);
	void TraverseTree(vector<NODE> tree);
	void TraverseInt(vector<int> intlist);
	void TraverseTableList(vector<TABLE> TableList);
	void TraverseTableMap(map<string,int> TableMap);
	vector<string> GetKeyofTable(string table_name);
	vector<string> GetFullKeyofTable(string table_name);
	string Get_join_condition(int treeid, int a, int b, string key);
	TREE SELECTLocal(string sql_statement,int treeid,map<string,Fragment> FragmentMap,map<string,string> TableKeyMap);
	vector<string> GetLeafConditionList(vector<string> condition_list);
	vector<string> GetInerColumnList(vector<string> column_list, int treeid, int nodeid);
	vector<string> GetCollumnFromSql(string sql_statement);
	vector<string> GetTableCollumnFromSql(string sql_statement);
	void DrawTree(vector<NODE> Nodes);

Tools.cpp	map<string,Fragment> GetETCDFragment();
	map<string,string> GetETCDTableKey();
	string GetBetween(string sql_statement, string start, string end);
	string GetAfter(string sql_statement, string start);
	string GetBefore(string sql_statement, string end);
	string GetExactAfter(string sql_statement, string start);
	vector<string> GetList(string line, string split, string stop);
	void Traverse(vector<string> input);
	string Link(vector<string> input, string devide);
	string GetTableName(string sql_statement);
	vector<string> GetTableList(string TableName);
	string GetSelectColumnName(string sql_statement);
	vector<string> GetSelectColumnList(string sql_statement);
	string GetCondition(string sql_statement);
	vector<string> GetConditionList(string sql_statement);
	string GetTableFromColumn(string column, vector<string> TableList);

```
vector<string> GetColumnFromCondition(string condtion, v
ector<string> TableList);
vector<string> GetColumnListFromConditionList(vector<str
ing> ConditionList, vector<string> TableList);
vector<string> GetAllColumnList(string sql_statement);
vector<string> GetAllDifferentColumnList(string sql_stat
ement);
vector<string> GetAllDifferentTreeNodeColumnList(string
sql_statement, string table_1, string treenode1, string
table_2, string treenode2);
vector<string> GetAllTreeNodeCollumnList(string sql_stat
ement, string table_1, string treenode1, string table2,
string treenode2);
vector<string> GetAllData(string sql_statement);
int GetTCLoc(string table, string column);
string GetPureColumnFromColumn(string column);
string Link(vector<string> input, string devide);
void Traversefrags(vector<FragDef> frags);
void TraverseFragment(Fragment frags);
string GetTableListFromInsert(string sql_statement);
vector<string> GetColumnListFromInsert(string sql_statem
ent,string table_name);
vector<string> GetValueListFromInsert(string sql_stateme
nt);
bool JudgeHit(vector<string> BigList, string oneitem);
bool JudgeHit2 (vector<string> FirstList, vector<string>
 SecondList);
int GetLocateHit(vector<string> BigList, string oneitem)
;
bool Judge(vector<string> condition_list, vector<string>
 condition_column_list, vector<string> insert_column_lis
t, vector<string> insert_value_list);
string GetValue(vector<string> column_in_frag_list, vect
or<string> value_list, vector<string> column_list);
string GetInsetItem(vector<string> column_list, vector<s
tring> value_list);
void trim(string &s);
void TraverseGDD(GDD gdd);
void TraverseGDDCol(vector<ColumnDef> cols);

vector<string> GetCollumnListOfTable(string table);
string Getall_collumn(string table_1,string treenode1,st
ring table_2,string treenode2);
string GetTreeNode(int treeid,int nodeid);
```

	vector<string> GetAllDifferentCollumnListOfTable(string sql_statement, string table_name);
	vector<string> GetCleanList(vector<string> input);
	map<string,GDD> GetETCDGDD();

2、Metadata 模块

为了保证分布式数据库系统的正常运转，需要有一个稳定的系统来维护数据库的基本信息（例如数据库名称，表名，表格的存储位置等元数据），这些信息需要广播到所有站点，以保证用户对数据库的正常操作。我们选择 ETCD 存储元数据，并将所有站点加入 ETCD 集群，保证所有站点同步元数据信息。

（1）ETCD 官方文档地址

https://etcd.io/docs/v3.4.0/

（2）Metadata 结构设计

我们需要存储的元数据包括表格信息和表格的分块信息，主要的数据结构定义为下面 4 个结构体：

```
// column
struct ColumnDef {
    string name;
    string type;
    bool null=false;
    bool key=false;
    string desc;
};
// table
struct GDD {
    string name;
    vector<ColumnDef> cols;
};
// fragment
struct FragDef {
    int id;
    int site;
    string column; //该分片涉及到的 column，如果是多个，则用逗号分隔
    string condition; //分片条件
    int size; //记录该分片的大小，以 byte 为单位
};
// table-fragment
struct Fragment {
    string name;
    string fragtype; // H/V
    int fragnum;
    vector<FragDef> frags;
};
```

针对上面的结构体，我们在 ETCD 中设计了层级存储结构，以 publisher 表为例，有下列元信息：

表 1：ETCD 存储结构

Key	Value	Example
/gdd_table	table-name	</gdd_table,”publisher, customer, book , order“>
/gdd_table/publisher	attri-name	</gdd_table/publisher,”id,name,nation”>
/gdd_table/publisher/id	attri-type	</gdd_tablepublisher/id,int>
/gdd_table/publisher/name	attri-type	</gdd_table/publisher/name,string>
/gdd_table/publisher/nation	attri-type	</gdd_table/publisher/nation,string>
/gdd_table/publisher/desc/id	description	</gdd_table/publisher/desc/id,”The ID of oublisher”>
/gdd_table/publisher/desc/name	description	</gdd_table/publisher/desc/name,”…”>
/gdd_table/publisher/desc/nation	description	</gdd_table/publisher/desc/nation,”…”>
/gdd_key/publisher	primary-key	</gdd_key/publisher,”id”>
/part_schema/publisher	H/V	</part_schema/publisher ,”H”>
/part_schema/publisher/H	associated attri	</part_schema/publisher/H,”id,nation”>
/part_info/publisher	number of fragments	</part_info/publisher,4>
/part_info/publisher/publisher.1	condition	</part_info/publisher/p.1,”id<104000 and nation=’PRC’”>
/part_info/publisher/publisher.2	condition	</part_info/publisher/p.2,”id<104000 and nation=’USA’”>
/part_info/publisher/publisher.3	condition	</part_info/publisher/p.3,”id>=104000 and nation=’PRC’”>
/part_info/publisher/publisher.4	condition	</part_info/publisher/p.4,”id>=104000 and nation=’USA’”>
/size/publisher.1	size	</part_info/publisher/p.1,int >
/size/publisher.2	size	</part_info/publisher/p.2,int >
/size/publisher.3	size	</part_info/publisher/p.3 ,int >
/size/publisher.4	size	</part_info/publisher/p.4,int >
/part_site/ publisher.1	position	</part_site/ publisher.1,s1>
/part_site/ publisher.2	position	</part_site/ publisher.2,s2>
/part_site/ publisher.3	position	</part_site/ publisher.3,s3>
/part_site/ publisher.4	position	</part_site/ publisher.4,s4>

（3）Metadata 主要存取接口

针对上面的元信息定义，我们主要提供如下四个元信息存储接口，Parser 模块可以直接调用，用以存取 ETCD 内的元信息。

```
// 将表格信息存入 ETCD
bool saveTableToEtcd(GDD table);
```

```
// 从 ETCD 中读取表格的信息
GDD getTableFromEtcd(string tablename);
// 将表格分块信息存入 ETCD
bool saveFragToEtcd(Fragment frag);
// 从 ETCD 中读取表格的分块信息
Fragment getFragFromEtcd(string tablename);
```

3、执行模块

3.1 mysql_connector

首先，由于已经确定底层数据存取与关系运算均直接采用 MySQL 数据库，所以需要先配置环境，然后构建专用的模块来方便地操作 MySQL，取名为 mysql_connector。

配置环境具体方法请见代码库中的说明文件 readme.md。由于有现成的 C++接口 libmysqlclient-dev 来操作 MySQL，所以我选择直接利用该接口进行简单的封装。由于后续需要调用传输模块在不同站点之间传输数据，所以综合考虑代码量和可用时间，我们决定将数据以 sql 文件的形式进行传输和临时表存储，因此定义了全局的临时文件存储目录"/mnt/d/ddbms200tmp/"。

用法参考链接 https://blog.csdn.net/android_lover2014/article/details/53125105

具体封装的函数与功能如下图所示，每个函数都有的 site 参数其实是因为在机器 3 上有两个站点 site，所以是为了在 MySQL 连接的时候进行区分而设；其中 local_Insert_Delete 其实除了插入删除语句，还可以对 MySQL 执行任何只需要返回成功与否的 SQL 语句；local_Load 之所以需要创建表的 SQL 语句，是因为 MySQL 在导入非 sql 文件的时候必须指定导入到哪一张表，而本实验中提供的数据文件是 tsv 格式，故需要创建表的 SQL 语句 sql_create；Local_Select 虽然只返回查询结果的名称，但其实在本地临时文件存储目录下会已经存有以该名称命名的 sql 文件；随后的 Local_Tmp_Load 函数便是为了从该存储目录下将临时表存入 MySQL 以便后续操作；my_mysql_res_print 函数可以直接打印出任何临时文件存储目录下的表（sql 文件）：

```
101  /* 本地执行插入和删除函数，输入SQL语句和站点名称（s1, s2, s3, s4），返回执行结果(OK or FAILED) */
102  string local_Insert_Delete(string sql, string site);
103
104  /* 本地执行文件导入函数，输入创建表的SQL语句和导入文件的SQL语句，和站点名称（s1, s2, s3, s4）,返回执行结果(OK or FAIL)
105     例子：以Book表为例
106     local_Load(
107        "create table book(id int(6), title char(100), authors char(200), publisher_id int(6), copies int(5), key(id) )",
108        "load data local infile '/home/roy/ddbms/rawdata/book.tsv' into table book"); */
109  string local_Load(string sql_create, string sql_load, string site);
110
111  /* 本地执行查询函数，输入SQL语句，和指定返回结果集名称（唯一），和站点名称（s1, s2, s3, s4），
112     返回执行结果表名(全局唯一)，或者"FAIL"(暂时没有了) */
113  string Local_Select(string sql, string res_name, string site);
114
115  /* 本地执行表存储函数，输入待存的数据表名，和站点名称（s1, s2, s3, s4）,返回执行结果(OK or FAIL) */
116  string Local_Tmp_Load(string tmp_data, string site);
117
118  /* 相当于打印sql文件里的数据
119     输出样例为：
120  查询到 2 行
121  id      title   authors publisher_id    copies
122  200001  Book #200001    H. Johnston     100366  7231
123  200002  Book #200002    L. Houghton     101543  694 */
124  void my_mysql_res_print(string my_res);
```

除此之外，为了方便客户端的展示、操作和结果返回，我还另外封装了三个函数，让 parser 模块彻底与底层数据存取结构解耦。如下图所示，my_mysql_res_output 是考虑到有时查询结果集过于庞大，直接输出在终端时不方便查看，便可以选择输出到文件中，方便查

看；my_mysql_res_get_rows 是为了单独查看结果集的行数，主要用于 load，insert，delete 等一切需要返回数据改动行数的命令；my_mysql_res_get_column1 其实是为了 delete 命令打的补丁，由于按列分块的数据集在 delete 的时候有两步操作，第一步是通过 where 子句的条件获取需要删除的数据 id，第二步是通过 id 去准确删除数据，在这两个步骤衔接处，对于 parser 而言需要在内部获得 vector 形式的数据，因此我专门提供了这一函数。

```
126  /* 相当于输出sql文件里的数据到某个文件，输入sql文件名和输出文件名（带后缀），例如"output.txt"
127     输出样例为：
128  id      title          authors         publisher_id    copies
129  200001  Book#200001    H. Johnston     100366          7231
130  200002  Book#200002    L. Houghton     101543          694 */
131  void my_mysql_res_output(string my_res, string filename);
132
133  /* 获得SQL文件里数据的行数，要在load之前（之后再也不需要数据库里有这个数据）  */
134  int my_mysql_res_get_rows(string my_res);
135
136  /* 获得SQL文件里第一列数据(此处其实特意为id而设)，要在load之前（之后再也不需要数据库里有这个数据）  */
137  vector<int> my_mysql_res_get_column1(string my_res);
```

3.2 executor_multi

在解决与 MySQL 相关的所有问题之后，便可以专心于执行所有 parser 传来的执行计划。执行计划按照命令可分为：load 类，insert_delete 类，select 类。接下来分别介绍每一类执行计划的实现。

3.2.1 Data_Load_Execute

由于其实不管是按行分块，按列分块，还是混合分块，要获得往某个站点存的具体分块数据，都可以利用 SQL 查询语句，例如按行分块：要分成 id<10005 与 id>=10005 两块，就可以构造 SQL 语句：select * from book where id<10005; select * from book where id>=10005，得到的两个数据块就是我们想要的，只需要分别存到对应站点即可。这种方法可以处理任何复杂的分块。因此如下图所示，Data_Load_Thread 函数的功能是：输入站点 site、分块 SQL 语句 frag_sql 和分片名称 frag_name（如 s1, select * from book where id<10005, book_1），将正确的数据按照指定的名称存入对应站点，返回执行结果（成功与否）。

而为了加快数据导入速度，我特意支持了一体化的并行数据导入。Data_Load_Execute 函数的功能是：输入 create_sql，load_sql，main_name 来从客户端正确导入 tsv 文件到本地 MySQL，表名为 main_name，然后输入一一对应的站点、分块 SQL 语句和分片名称列表参数 sitenames、sqls 和 table_names，是为了并行执行多个分块并存储到对应站点，在存储结束后所有中间文件都会被自动删除，以免增加内存负担。这一函数会返回每个站点上数据存储量（如果存储失败会显示 FAIL）以及总共用时。

```
58  /* for循环内原先的内容被封装为另外一个函数，
59  输入sitenames, sqls, table_names, 输出String - "xx rows imported on site x.\n" 或者 "FAIL on site x.\n"
60  最后一个参数是为了传递主函数给每个线程的对应变量预留的空间，而不是传递值 */
61  void Data_Load_Thread(string site, string frag_sql, string frag_name, std::promise<string> &resultObj);
62
63  /* 本函数供parser调用
64  本函数用于执行整个load流程，输入本地创建和导入表的SQL语句，本地表的表名，站点列表，
65  分片select语句列表和分片表名列表，返回
66  "x1 rows imported on site 1.
67   x2 rows imported on site2.
68   y seconds used."
69   或者"FAIL" */
70  string Data_Load_Execute(string create_sql, string load_sql, string main_name, \
71                           vector<string> sitenames, vector<string> sqls, vector<string> table_names);
```

3.2.2 Data_Insert_Delete_Execute

其实单纯的插入删除都是每个站点上最多一句 SQL 便可以执行完毕的，所以采取了与 Load 类似的思路，如下图所示，先写了 Data_Insert_Delete_Thread 函数来实现在对应站点上执行输入的 SQL 语句的功能，然后为了加速运行支持了多线程，因此

Data_Insert_Delete_Execute 可以并行执行一整组插入删除的 SQL 语句，返回每个站点上的执行结果（成功与否）与花费的总时间。

```
/* for循环内原先的内容被封装为另外一个函数，
输入sitenames, sqls, 输出String - "OK on site x.\n" 或者 "FAIL on site x.\n"
最后三个参数是为了传递主函数给每个线程的对应变量预留的空间，而不是传递值 */
void Data_Insert_Delete_Thread(string site, string frag_sql, std::promise<string> &resultObj);

/* 本函数供parser调用
本函数用于执行整个Insert或Delete流程，输入站点列表，分片sql语句列表，返回
"OK/FAIL on site s1.
 OK/FAIL on site s2.
 y seconds used." */
string Data_Insert_Delete_Execute(vector<string> sitenames, vector<string> sqls);
```

3.2.3 Data_Select_Execute

这一类执行计划是最为复杂的，也是本项目的重点。查询类的执行计划是树的形式，因此很自然地可以想到采用递归的方式执行，每次执行查询树的一棵子树，直到叶子节点才直接从 MySQL 中拿表，而拿到返回结果的父亲节点根据查询树指导的操作进行数据整合（join 或者 union），再继续返回给上一层函数。

而执行之后需要返回的信息量又包括：站点之间传输的总数据量，执行时间，因此我针对 parser 传入的执行计划树，设计了对应的执行结果树，结构如下图所示。如此一来每个节点所花的时间和传输的数据量以及执行成功与否都一目了然，既方便了 parser 模块的调试，让 parser 负责人能清楚地看到是哪个节点出了问题，完全解耦了执行模块和 parser 模块的调试工作；又为客户端提供了充分的信息，不论最后需要返回执行时间还是响应时间，总的数据传输量还是分站点的数据传输量，都可以通过简单的计算得到结果，不需要执行模块反复修改打补丁；最关键的是还提供了直观的瓶颈分析途径，可以通过每个节点所花时间来分析拖慢整个执行过程的关键点在哪里，从而为查询计划的优化提供宝贵的参考。

```
// 现在定义一下时间和数据传输量记录的结构
// 也用树结构记录每个节点花费的时间和数据传输量
struct exec_node{
    int node_id; // 计划树中该节点对应的ID
    double time_spend; // 执行对应节点所花的时间，单位为秒
    size_t volume; // 该节点上结果的数据量，单位为比特
    string res; //该节点上结果的执行情况，OK或者FAIL
    vector<int> child; //该节点的孩子，与计划树一致
    int parent; //该节点的父亲，与计划树一致
    int site; //该节点的site，与计划树一致
};
struct exec_tree{
    int tree_id; // 应当与它执行的树的ID一致
    int root; // 为了与原来的树保持一致
    vector<exec_node> Nodes; // 应当与它计划树的node数量一致
};
```

在这里涉及到很多对树的操作，首先我封装了四个函数用于从树中获取想要的信息，如下图所示，TREE 指的是执行计划树结构，exec_tree 指的是执行结果树结构，通过 get_root、get_root、get_node、get_sub_tree，我便可以方便地获得任何一棵树的子树、根节点或其他节点的信息。

然后根据递归的思路，整个执行只需要一个函数 Data_Select_Execute，它会不断地调用自身。但是为了提升执行速度，让任何节点的所有子节点都能并行执行，就必须另外定义一

个函数 Data_Select_Execute_Thread 来执行子节点上的操作。这带来的问题是：Data_Select_Execute 要并行执行子节点，便需要调用 Data_Select_Execute_Thread，返回执行结果树，这会涉及到子进程与主进程通信所用的公共空间（std::promise 类）， 而如果该子节点在另外的站点，两个站点之间并不能直接共享某个内存空间，强行设定会给执行模块和传输模块都带来不小的工作量。因此简单的本地实现在分布式场景下并不现实。

为此，我特意询问了传输模块 GRPC 的运行逻辑，发现 GRPC 接到父亲节点站点调用指令的是 client，而执行的是 server，server 再调用本站点的 executor_multi 中指定函数。因此我决定对要使用 GRPC 进行异站点执行的线程加一层封装，于是有了 RPC_Data_Select_Execute_Thread，这个函数的功能是在指定站点 site 执行 Data_Select_Execute（而不是 Data_Select_Execute_Thread，从而避免 std::promise 类传输的问题），将返回的查询执行树和结果 sql 文件传回本站点，然后把查询执行树通过 std::promise 类标识的本地共用内存空间返回给对应主进程，从而顺利实现多线程并行。

```
/* 通过系统stat结构体获取文件大小，单位bytes，size_t为长整型，若要打印占位符为%ld */
size_t get_filebytes(const char *filename);

/* 输入一个计划树结构，返回这棵树的根节点 */
NODE get_root(TREE tree);

/* 输入一个执行树结构，返回这棵树的根节点 */
exec_node get_root(exec_tree tree);

/* 输入一个树结构和节点id，返回对应节点 */
NODE get_node(TREE tree, int node_id);

/* 输入一个树结构和节点id，返回以对应节点为根节点的子树 */
TREE get_sub_tree(TREE tree, int node_id);

/* RPC_Data_Select_Execute的可并行化版本，合并前先注释掉方便调试 */
void RPC_Data_Select_Execute_Thread(TREE tree, string site, std::promise<exec_tree> &resultObj);

/* Data_Select_Execute的可并行化版本 */
void Data_Select_Execute_Thread(TREE tree, std::promise<exec_tree> &resultObj);

/* 本函数供parser调用
用于递归执行整个select流程，输入一棵查询计划树，返回对应的执行结果树
至于查询到的数据结果，从tree与node的id便可以推测得到结果表的名字，
mysql_connector.h里面也提供了根据结果表名字打印结果的函数void my_mysql_res_print(string my_res);
注意此版本非并行化版本，仅供RPC_Data_Select_Execute或者parser调用（递归的开始），内部调用的均为可并行化版本
exec_tree Data_Select_Execute(TREE tree);
```

4、传输模块

传输模块的主要功能有两方面：数据传输，远程函数调用。执行模块在各个站点执行时，需要调用其他站点的函数并同时传输数据，因此，传输模块的成功执行时整个系统正常运行的基础。

我们使用 gRPC 传输框架，gRPC 默认使用 protoc buffers，protoc buffers 是谷歌成熟的开源的用于结构化数据序列化的机制。由于 protoc buffers 只能传输 message 格式的数据，我们需要将所有需要传的数据结构编码为 message 格式，当传输到目标站点时，再解码为原本的数据结构。该方法较为复杂，且每次传输量有限制，因此只能用来传输一些参数，而不适用于数据表的传输。针对数据表，我们采用流式传输，保证传输数据的高效性和完整性。

（1）gRPC 参考地址

a) 官方文档：https://grpc.io/docs/languages/cpp/quickstart/

b) 安装笔记：https://www.jianshu.com/p/73ec0014ac0f

（2）gRPC 函数接口

```
// 在目标 site 实现 insert/delete 操作
string RPC_local_Insert_Delete(string sql, string site);
// 在目标 site 实现 Load 操作
string RPC_local_Load(string sql_create, string sql_load, string site);
// 在目标 site 实现 select 操作，并传回结果
string RPC_Local_Select(string sql, string res_name, string site);
// 向目标 site 传输数据，并传回结果
string RPC_Local_Tmp_Load(string tmp_data, string site);
// 传输执行树并传回结果
exec_tree RPC_Data_Select_Execute(TREE tree, string site);
// 获取目标 site 的文件
string RPC_GET_FILE(string filename, string site);
```

五、测试结果

这里介绍 SELECT 功能的实现结果，本次实验测试了 11 个 SELECT 语句，全部测试都得到正确的查询结果。尹清实现了一个打印执行树执行结果的接口，调用这个接口就可以看到每个节点的执行时间以及传输数据量的大小。

以 SELECT * FROM customer;语句为例，这个语句的实现依赖于一棵含有三个节点的执行树的构建。三个节点的实现信息列举如下表。NODE id 为 1 的节点所用时间为 0，传输量为 316728，其没有孩子节点，其父亲节点为 3 号节点，操作位于 2 号站点。3 号站点的父亲节点为 0，则意味着 3 号节点是执行树中最后一个实现的节点，其对应的时间即为完成整个树的执行所需要的时间。

表 2：以第一个 SELECT 语句为例介绍实验结果

NODE	TIME	VOLUME	RES	CHILD	PARENT	SITE
			1			
2	2	196476	OK	3	3	2
1	0	316728	OK	NULL	3	1
3	27	376761	OK	2,1	0	1

全部指令的运行所传输的数据量大小，以及运行总时间如下表所示，用时最长的指令是 6 号指令：SELECT customer.name,orders.quantity FROM customer,orders WHERE customer.id=orders.customer_id;这个指令最终返回的结果涉及到 100000 条结果或影响了语句的执行时间。

表 3：实验结果

指令编号	传输大小/字节	时间/秒
1	196476	27
2	70692	2
3	541548	5
4	946472	6
5	1428531	68
6	1149040	909

7	1149040	86
8	2450439	255
9	2355913	449
10	3075747	326
11	2612566	174

有意思的是，6 号指令并非是传输数据量最大的，8 号指令是传输数据量大小最大的指令，也并非执行节点最多的，10 号与 11 号指令一共包含 22 个执行节点，但却是执行时间最多的，这个现象十分反常，有待后续进一步的研究。更完整的实验结果在附录中展示。

六、心得体会

1、李晓桐

我负责写 parser 部分代码，在中期报告中我曾经描述了一个我打算在这个学期实现的 SELECT 解析算法，并未完全实现，在执行树剪枝这块并没有实现最开始的想法，有些遗憾。在功能实现方面我需要调用尹清和涂涂的代码，因此前期做了很多的沟通，包括她们的代码在哪里定义，需要怎样的输入等等。由于交流作得很足，并没有出现写完代码后发现驴唇不对马嘴的情况，没有浪费时间做无用功。

这门课收获颇丰，之前没写过这么多 c++代码，这一次得到了充分的锻炼。实现代码的过程并没有我想的那么容易，总会出现一些奇怪的问题。这次我的工作主要就是字符串处理，将用户输入的字符串转成尹清执行层面可以作为输入的字符串，有时一个空格就会引起执行的失败，因此需要极大的细心。

在写代码方面，我总结了新的写代码思路，或许可以帮助后来者更好完成代码。我将代码分成三类：主要代码（用户所面对的），功能代码（主函数所调用的），与工具代码（功能函数所调用的，实现功能的代码）。以本次作业为例，Parse.cpp 就是主要代码，Parse_INIT.cpp, Parse_LOAD.cpp, Parse_DELETE.cpp, Parse_INSERT.cpp, Parse_SELECT.cpp 就是功能代码，Tools.cpp 就是工具代码。在完成大作业的过程中，我发现在实现大型工程时，编译容易出错的一个原因是引用关系混乱，在写代码的时候最好就先定好大框架，确定每个模块的输入输出（这部分要确认输入输出的变量名称、类型与含义），确定好框架也就同时确定好应用关系，为 MAKE 做准备。

2、罗尹清

我负责执行 parser 递过来的所有计划，目标是让 parser 不需要接触任何与 MySQL 有关的 C++代码，也不需要操心执行过程并行还是串行，只需要调用我提供的函数就能执行所有的计划，所以接口的沟通非常重要。好在我和晓桐宿舍很近，有充分的时间进行沟通，确保了衔接的顺畅。

考虑到 parser 灵活性更强，而执行部分限制更多，所以接口主要是我进行设计，然后给晓桐讲解使用方法和含义。这也意味着我其实要先想好从客户端到各个站点的 MySQL 再回到客户端的整个功能的实现逻辑，并在这个过程中和荐泓反复商讨确认什么是可以实现的，不能实现的设想再做调整。比如一开始我想直接利用 MySQL 的 C++接口中定义的数据结构来存储数据，后来发现对 GRPC 来说传输这样一个结构代码量巨大，因此我们商量之后才改成传输 sql 文件，虽然磁盘读写代价比内存直接处理更高，但是有效降低了荐泓的工作量。再比如一开始我想把 select 的并行化执行直接利用 GRPC 传输，但是里面涉及的 std::promise

等结构传输也比较复杂，因此后来特意加了一层封装，把涉及与不涉及 GRPC 的部分分开处理。

在一开始老师便提醒我们不要小看这个大作业，优先保证完整做出来而不是抠细节，所以我的设计思路是在保证功能齐全的基础上优先选择工作量小的方案(而不是性能最优的方案)，如果后面有时间再优化（后面证明确实没多少时间优化）。也是这个原因让我直接选择了逻辑上简洁明了的递归执行方案。不过最后测试的时候执行速度看起来还有提升空间，可以根据返回的执行结果树再进行瓶颈分析，继续优化。

这个大作业促使我学习 C++，研究多线程，锻炼了我的编程能力。而且因为我们一开始制定了时间表，我每周都会留充足的时间按计划实现，撰写日志和文档，查缺补漏，并细致地考虑代码的美观性和逻辑缺陷，写上尽量完备的注释，因此我这部分提前一点完成了。因为是认真一点点打磨过的，所以每周都心中有数，完成之后成就感比较高。

在这个过程中我的两位队友也付出良多。一开始传输环境的配置荐泓是从 0 开始学习，非常不容易；后期 parser 部分细碎但庞大的工作量也让考试月的晓桐颇费心力。但好在最终功能都顺利实现了，也算我们的心血没有白费。感谢老师细致的指导与及时的督促，感谢助教的帮助与鼓励，也感谢其他组同学热心的建议，这些都共同营造了非常良好的学习氛围。

3、涂荐泓

在完成这个分布式数据库系统的过程中，我遇到过许多问题，也收获颇丰，下面主要以问题驱动思考的方式记录。

问题 1：学习 RPC 和 ETCD 时，由于完全不了解他们是什么东西，所以刚开始的时候踩坑不断。（此处省略一万字），总结一句：珍爱光阴，远离 CSDN，好的官方文档是人间正道。对于搭建环境，确实万事开头难，沉下心来好好看官方文档肯定会解决的，问题来了不要怕，一个一个解决总会暂时解决完的。环境问题解决后，学学新语法，很快就完成了基础代码编写，之前的坑一切都是值得的。

问题 2：团队合作中不可避免的交流问题。我们小组在本次合作中，工作划分算比较清晰的，接口也比较清晰。但还是出现了偶尔的不 match，好在都及时沟通解决了。我也更加体会到在团队合作写代码中，一定要非常细心，避免马虎小错误，并严格按照计划完成任务。此外，大家一定要提前统一进度安排，这是团队高效进行合作的基础。

问题 3：莫名其妙的问题……是的，我甚至在某次部署 ETCD 时遇到了机器时钟不同步的问题……针对此类问题，我的收获是，遇事不要慌，离开座位，去喝点热水比什么都管用。

【附录】

1、SELECT 语句实验结果展示

表 4：完整实验结果展示

NODE	TIME	VOLUME	RES	CHILD	PARENT	SITE
			1			
2	2	196476	OK	3	3	2
1	0	316728	OK	NULL	3	1
3	27	376761	OK	2,1	0	1
			2			
4	1	12279	OK	NULL	5	4

3	1	13577	OK	NULL	5	3
2	1	44836	OK	NULL	5	2
1	0	46046	OK	NULL	5	1
5	2	111452	OK	4,3,2,1	6	1
6	2	111452	OK	5	0	1
3						
3	0	1908	OK	NULL	4	3
2	3	539640	OK	NULL	4	2
1	1	61080	OK	NULL	4	1
4	4	599232	OK	3,2,1	5	1
5	5	424855	OK	4	0	1
4						
4	0	412530	OK	NULL	5	4
3	0	176173	OK	NULL	5	3
2	2	357769	OK	NULL	5	2
1	1	153224	OK	NULL	5	1
5	4	1094343	OK	4,3,2,1	6	1
6	6	1094343	OK	5	0	1
5						
3	0	1949	OK	NULL	4	3
2	4	1335665	OK	NULL	4	2
1	1	151019	OK	NULL	4	1
4	6	1485155	OK	3,2,1	10	1
8	0	18459	OK	NULL	9	4
7	0	1929	OK	NULL	9	3
6	2	70529	OK	NULL	9	2
5	0	1464	OK	NULL	9	1
9	2	87027	OK	8,7,6,5,	10	1
10	67	1448604	OK	4,9	10	1
11	68	1097758	OK	10	0	1
6						
2	3	196476	OK	NULL	3	2
1	0	316728	OK	NULL	3	1
3	758	376761	OK	2,1	9	1
7	0	415223	OK	NULL	8	4
6	0	177175	OK	NULL	8	3
5	0	360166	OK	NULL	8	2
4	0	154350	OK	NULL	8	1
8	6	1101561	OK	7,6,5,4	9	1
9	907	3002877	OK	3,8	10	1
10	909	1602779	OK	9	0	1
7						
2	4	196476	OK	NULL	3	2
1	1	316728	OK	NULL	3	1

3	28	376761	OK	2,1	4	1
4	28	154986	OK	3	10	1
8	0	415223	OK	NULL	9	4
7	0	177175	OK	NULL	9	3
6	4	360166	OK	NULL	9	2
5	1	154350	OK	NULL	9	1
9	6	1101561	OK	8,7,6,5	10	1
10	85	1400700	OK	4,9	11	1
11	86	825220	OK	10	0	1
			8			
12	1	1939	OK	NULL	13	3
11	6	696633	OK	NULL	13	2
10	0	78493	OK	NULL	13	1
13	7	773619	OK	12,11,10	15	1
2	6	196476	OK	NULL	3	2
1	1	316728	OK	NULL	3	1
3	33	376761	OK	2,1	4	1
4	34	154986	OK	3	14	1
8	1	678239	OK	NULL	9	4
7	1	288711	OK	NULL	9	3
6	5	588441	OK	NULL	9	2
5	1	251655	OK	NULL	9	1
9	8	1801595	OK	8,7,6,5	14	1
14	100	1688420	OK	4,9	15	1
15	254	1156944	OK	13,14	16	1
16	255	641458	OK	15	0	1
			9			
6	0	1938	OK	NULL	7	3
5	7	991518	OK	NULL	7	2
4	1	1473	OK	NULL	7	1
7	8	991531	OK	6,5,4	19	1
2	6	196476	OK	NULL	3	2
1	1	316728	OK	NULL	3	1
3	31	376761	OK	2,1	18	1
16	0	469429	OK	NULL	17	4
15	0	198863	OK	NULL	17	3
14	6	406757	OK	NULL	17	2
13	1	176207	OK	NULL	17	1
17	8	1245782	OK	16,15,14,13	18	1
18	137	2559970	OK	3,17	19	1
19	396	2245431	OK	7,18	20	1
11	0	18467	OK	NULL	12	4
10	0	1936	OK	NULL	12	3
9	3	70529	OK	NULL	12	2

8	1	1464	OK	NULL	12	1
12	4	87035	OK	11,10,9,8	20	1
20	448	1113224	OK	19,12	21	1
21	449	385471	OK	20	0	1
10						
7	0	1973	OK	NULL	8	3
6	8	1778310	OK	NULL	8	2
5	1	198860	OK	NULL	8	1
8	10	1975630	OK	7,6,5	20	1
2	6	196476	OK	NULL	3	2
1	1	316728	OK	NULL	3	1
3	31	376761	OK	2,1	4	1
4	32	176475	OK	3	19	1
17	0	469429	OK	NULL	18	4
16	0	198863	OK	NULL	18	3
15	6	406757	OK	NULL	18	2
14	1	176207	OK	NULL	18	1
18	8	1245782	OK	17,16,15,14	19	1
19	83	1194348	OK	4,18	20	1
20	283	1942567	OK	8,19	21	1
12	0	1936	OK	NULL	13	4
11	0	20532	OK	NULL	13	3
10	2	1471	OK	NULL	13	2
9	1	72454	OK	NULL	13	1
13	2	91025	OK	12,11,10,9	21	1
21	325	1013636	OK	20,13	22	1
22	326	312058	OK	21	0	1
11						
7	0	1973	OK	NULL	8	3
6	7	1304732	OK	NULL	8	2
5	1	1508	OK	NULL	8	1
8	9	1304745	OK	7,6,5	20	1
2	5	196476	OK	NULL	3	2
1	0	316728	OK	NULL	3	1
3	31	376761	OK	2,1	4	1
4	32	126642	OK	3	19	1
17	0	469429	OK	NULL	18	4
16	0	198863	OK	NULL	18	3
15	5	406757		NULL	18	2
14	0	176207		NULL	18	1
18	7	1245782		17,16,15,14	19	1
19	70	860495		4,18	20	1
20	165	922066		8,19	21	1
12	0	15608		NULL	13	4

11	0	17290	NULL	13	3
10	2	1438	NULL	13	2
9	1	1431	NULL	13	1
13	2	30497	12,11,10,9	21	1
21	174	175515	20,13	22	1
22	174	60567	21	0	1