中国人民大学信息学院

SCHOOL OF INFORMATION RENMIN UNIVERSITY OF CHINA

分布与并行数据库

期末课程报告

RPQL: a Robust Performant SQL

RPQL 小组

刘佳伟 王芃 薛钦亮(组长)

2022-01-11

目录

一、项目介绍

本项目为 RPQL 小组设计并完成的分布与并行数据库期末作业，RPQL 的取名寓意为：a Robust Performant SQL，也即高鲁棒高性能关系型分布式数据。我们的系统完成了表的创建与删除、分片的创建、数据的增（insert、load）、删（delete）、改（update）、查（select），并且实现了查询与传输优化、集群站点的加入与退出、站点重命名、数据库的创建、切换与删除、p2p 测试等功能。我们设计了比较好用的前端命令行，可以识别错误语法，并且提供运行时的错误信息，是一个功能较为完善的作业。

项目完整代码见：https://github.com/XueQinliang/DDB_RPQL。

1.1 项目环境

操作系统：三台 centos7
数据库：mysql 5.7.35
编程语言：python 3.9
网络传输：grpcio==1.43.0
元信息存储：etcd3
查询树可视化：graphviz

1.2 项目架构

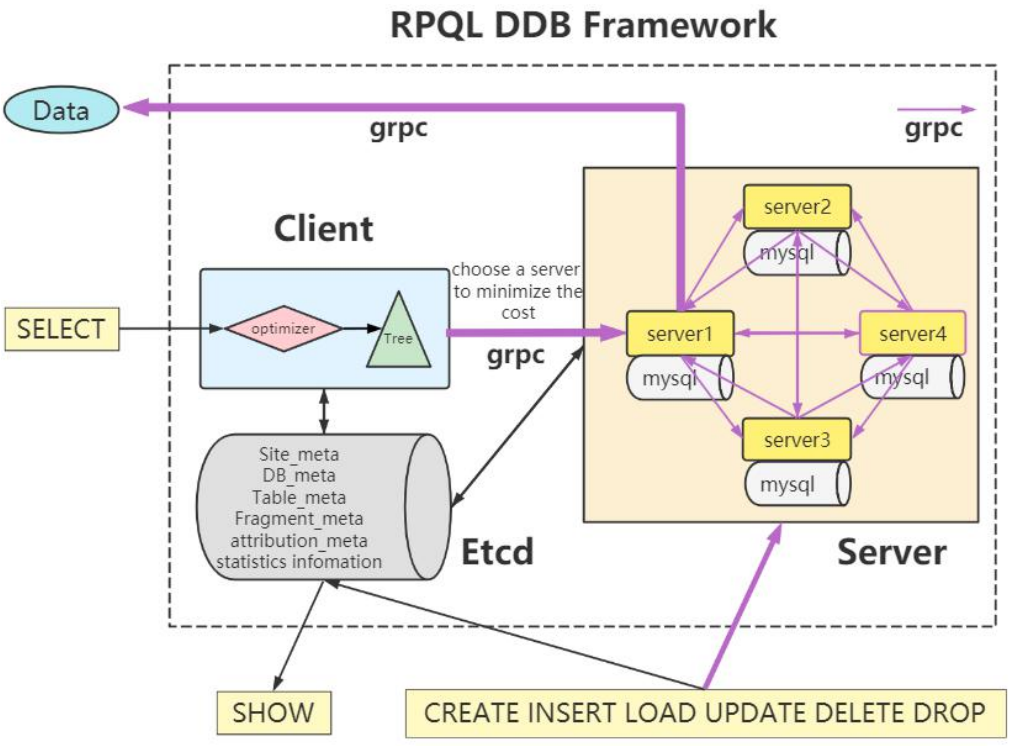

图 1 项目总体架构

如图 1 所示，项目架构沿用期中展示的基本架构，在期中的基础上进行了细化。如上图

所示，系统主要分为 server、client、etcd 三部分。

Server 端的代码通过 pymysql 连接 mysql 数据库，通过 grpc 与 client 端通信以及相互之间发送数据（图中的紫色箭头），同时可以读写 etcd 中的数据。

Client 端输入查询语句，如果是 CREATE、INSERT、LOAD、UPDATE、DELETE、DROP 开头的命令，直接处理之后发送给 etcd 或者 server，不需要进行优化，而 SHOW 相关的命令仅仅展示使用，读取存在 etcd 中的元信息即可。

如果 Client 端是输入 select 语句，则调用优化器生成优化后的查询树，并且选择查询树的最后一个节点所在站点，发送请求，并由这个站点转发到其他节点，并发地使用 DFS 算法去执行，最终把数据传送回来。

1.3 项目分工及时间安排

如图 2 所示，薛钦亮主要负责服务器的环境配置、系统架构设计、grpc 和 etcd 使用调研、DDL 以及除 select 语句以外的简单 DML 语言的实现，王芃主要负责构建查询树、查询优化、传输优化、基数估计、查询树可视化等工作，刘佳伟主要负责对查询树进行递归执行、利用 grpc 在各服务站点通信、多线程实现、利用索引的连接优化等功能。最终完整系统的实现，离不开每一个人的熬夜通宵爆肝，以及来不及吃饭时核桃派的帮助。

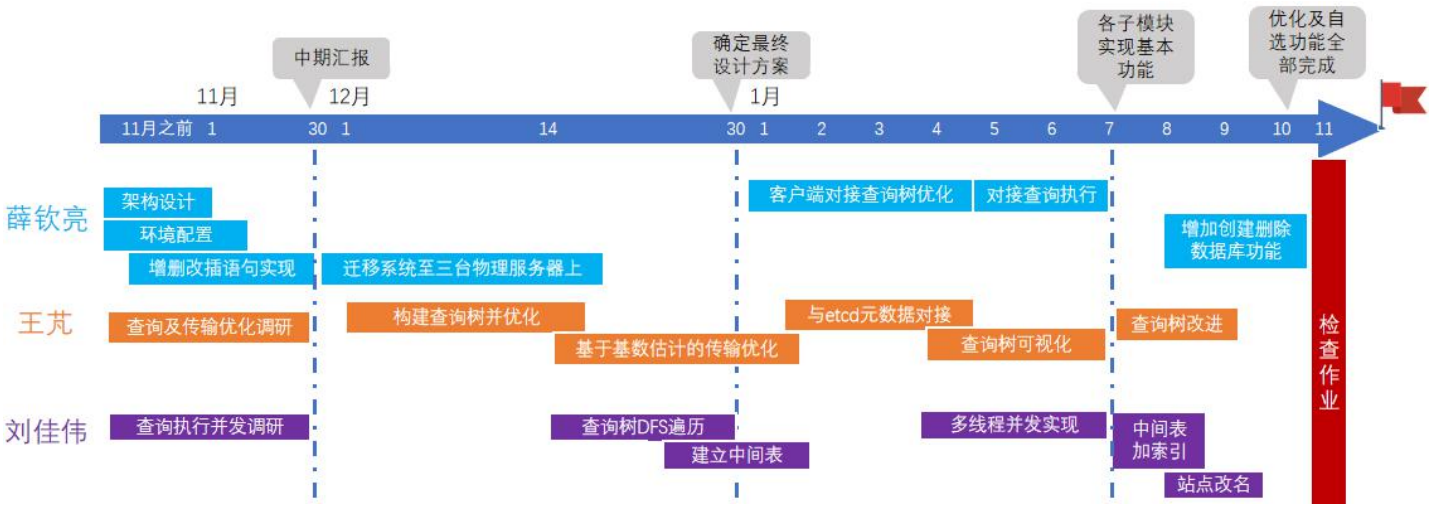

图 2 rpql 小组工作时间轴

1.4 项目全部功能一览

1. Database

支持 **create database、drop database、use database、show databases** 四种语句，会将分布式数据库的逻辑信息写入 etcd，而在具体的站点上建立逻辑名称+端口号以示区分的物理数据库。

etcd 中一次只存储一个数据库的元信息，其他数据库的元信息保存至磁盘中，如果需要切换数据库，则写回当前数据库的元信息，加载要使用的数据库的元信息到 etcd 中。

2. Table

支持 **create table、drop table、show tables、show tables with fragment** 四种语句，这里创建表只会把 table 的 metadata 写入 etcd，并不会实际在 mysql 中创建表，drop table 时会把 table 的 metadata 从 etcd 删除，同时删除包含的分片信息和数据，show tables 会展示当前数据库中已经创建的表，show tables with fragment 会展示当前数据库中已经创建的表及其包含的分片信息。

3. Fragment

支持 **create fragment、show fragment、show fragment with sites** 三种语句。创建 fragment 会实际地在 mysql 中建立一个表，且把 fragment 包含的列以及水平分片的条件写入 etcd 中。注意到，Fragment 没有标记到底是垂直分片还是水平分片，我们记录包含的列以及水平分片的条件具有更强的一般性，包含了垂直分片、水平分片、混合分片等各种情况，统一与这样的设计之下。

相比 table，fragment 不支持单独删除分片，如果需要删除，则需要删除整张表，这是为了保证数据的一致性。

show fragment 会展示所有分片信息，show fragment with sites 则会展示站点及其上的分片信息。

4. Site

Site 是通过 etcd 的 watch 机制动态发现与删除的，这样的好处是可以维护数量确定的 grpc 连接，并且最大程度复用已经建立的 grpc 连接，销毁已经不再有效的 grpc 连接，提高程序的安全性和性能。

我们支持 **define site、show sites** 两种操作。define site 允许对未命名的站点命名，也允许对已经命名的站点重命名，重命名的时候需要对 etcd 中保存的 table 和 fragment 元信息进行修改，用修改之后的站点名替换原有的站点名。这样设计是考虑到实际场景中，应该允许用户对包含数据的站点名进行修改，否则改名相当于迁移数据，是效率十分低下的做法。

5. Add info

Add info 是对数据库中表含有的列添加统计信息，添加的统计信息用于基数估计的传输优化，统计信息支持**正态分布、均匀分布、离散分布**三种类型，分别用 N、U、P 加上分布的参数来表示。之后的查询优化和传输优化会根据这里记录的数据分布，以及表的 metadata 中对每一列数据大小的统计，去估计返回结果的大小，从而估计查询代价和传输代价。

6. Insert

如图 3 所示，当用户执行 insert 时，客户端程序会把用户输入的 tuple 进行分组和切割，组织成要发送往每个站点的 tuple list，然后拼接成一个 sql 语句发送到各个服务器执行插入，这一过程是并发执行的，不保证每个站点插入完成的先后顺序，但程序会阻塞直到每一个站点都发送插入完成的信息给客户端。

7. Load

如图 3 所示，当用户执行 load 时，客户端程序会从文件中按行读取数据，把每一行数

据组织成一个 tuple，然后根据分片是垂直分片还是水平分片，把 tuple 分组或者切割，组织成要发送往每个站点的 tuple list，然后拼接成一个 sql 语句发送到各个服务器执行插入，也即 load 的底层依然是用 insert 来插入的。这一过程是并发执行的，由于数据较多，体现出了并发的好处，这里载入最大的表也只需要 10s 左右，远快于其他小组。

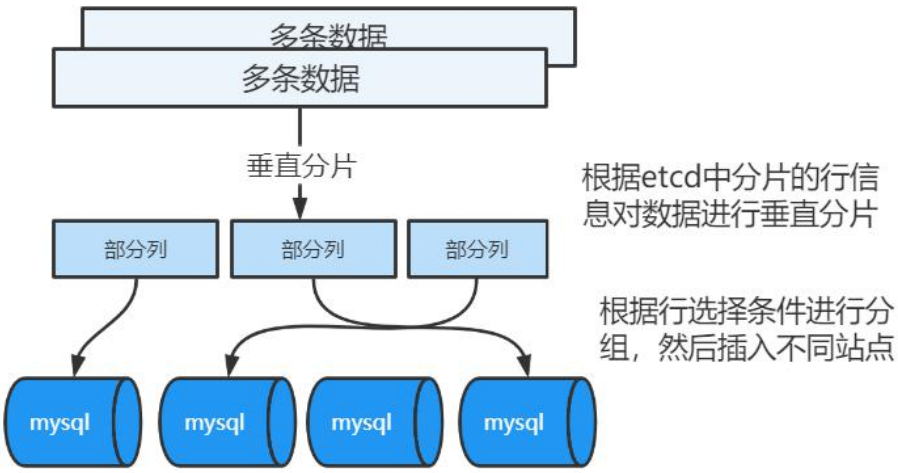

图 3 插入数据过程示意图

8. Delete

通用的 delete 语句过程如图 4 所示，分布式系统中的删除大体将分为查找、集合运算、删除三个过程：

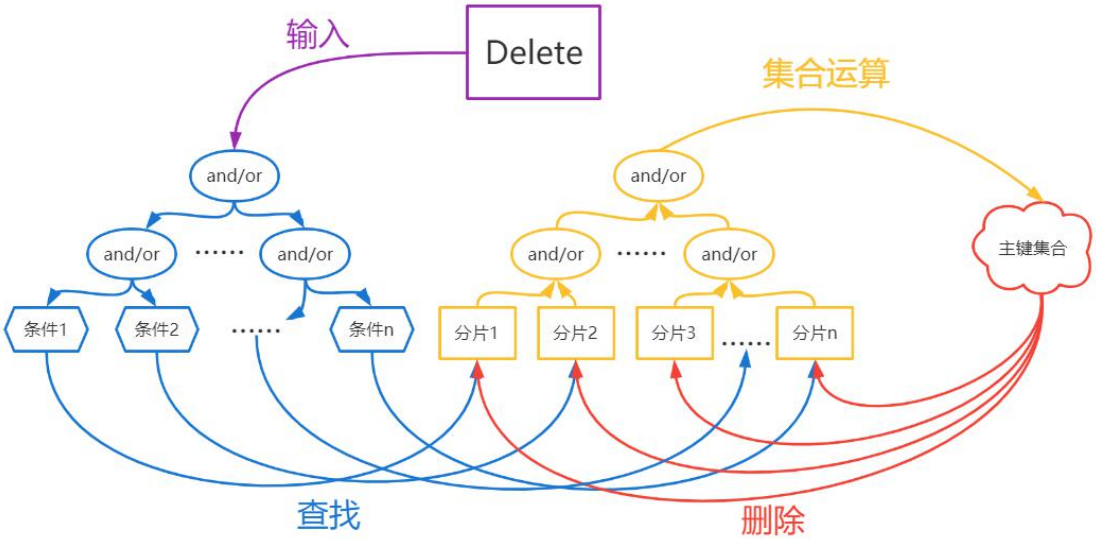

图 4 删除数据过程示意图

要删除数据时，客户端程序会根据 etcd 中存储的 meta 信息，重新组织删除语句发给对应的站点。如果对应的站点上没有要删除的数据，直接不去发送请求。如果对应的站点上数据完全满足 delete 语句附带的条件，则去除条件直接执行删除语句，这样效率更高。

如果对应的站点是垂直分片，情况更为复杂，简单地执行删除语句会造成同一个表的不同列主键不一致的情况。因此我们需要先在各个站点上执行一次查询语句，得到满足删除条件的主键，根据删除附带的条件的并和交的关系，去对主键的值执行集合运算，可以得到最

终要删除的行的主键的集合，最后并发地删除所有站点上满足这一要求的数据，可以保证同一个表在删除数据时各列保持一致。

9. Update

通用的 update 过程其实与 delete 较为相似，也是先查询，再集合运算找出满足条件的行，然后对每个站点进行更新，此处不详细讲解。

10. Select

当一条 Select 语句输入之后，首先生成一棵查询树，并根据查询的表信息和基数估计，进行查询优化和传输优化，生成一棵优化后的查询树，如图 5 所示。查询树的最后一个节点即为数据汇总的节点，因此 client 端要选择连接这个节点，可以最小化传输的数据量。当一个节点收到其他地方（client 或者其他 server）发送的请求时，首先检查数据是否在本站上，如果在则直接执行查询，如果不在，则新建一个线程向包含这种数据的其他站点发送请求并等待接收，当收到数据之后，再执行计算，发送给向它请求数据的站点（client 或者其他 server）。

整个执行过程是一棵树的 DFS 遍历过程，我们用了递归的方法，在执行中使用了多线程和消息队列的技术，来处理多个请求。具体执行过程请见第六章。

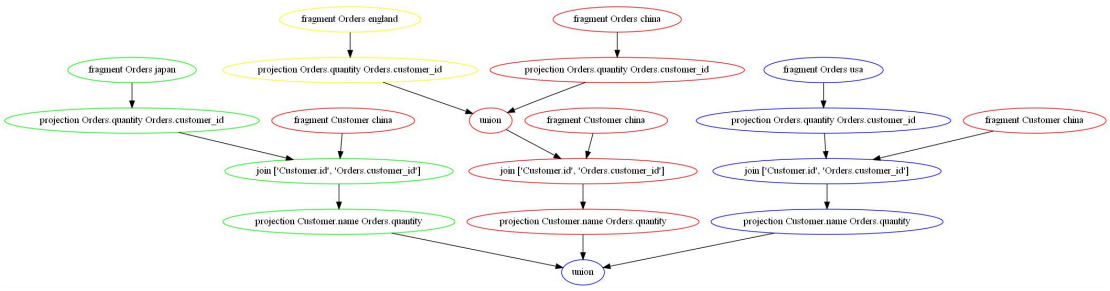

图 5 一棵查询树的可视化，不同颜色代表不同站点

二、配置过程

2.1 服务器的 vscode 配置方式

我们是用 vscode 去连接老师给的几台服务器的，助教给的是 ppk 文件，这个文件可以直接当做私钥文件，用于 putty 或者 mobaxterm 的连接，但不能直接用于 vscode 的连接。

我把这里的配置过程整理为 csdn 博客：

https://blog.csdn.net/weixin_43997331/article/details/121484353。

简单叙述一下过程：

1. 到 putty 官网下载了 PuTTY 全套的工具，我们会用到里面有一个工具叫做 PuTTYgen。

2. 双击启动软件，点 Load 把 ppk 私钥文件加载进来，然后点击 Conversions，最后点击 Export OpenSSH key，会生成一个文件，这里我们可以随意命名，然后把文件放入 C 盘的用户目录下。

3. 最后，把 Vscode 中 ssh 的 config 文件中的路径改为新生成的私钥文件即可正常连接。具体为，在自己电脑的用户目录下（我的电脑是 C:/Users/HP）打开.ssh/config 文件，加入如

下内容，IdentityFile 就是私钥文件的路径。

```
Host 10.77.70.61
  HostName 10.77.70.61
  LogLevel verbose
  User centos
  IdentityFile "c:/users/hp/ddbppk"
```

使用 vscode 最大的好处是有很多插件可以安装，但由于服务器无法联网，我们需要手动下载插件的 vsix 文件，可以去插件官网（https://marketplace.visualstudio.com/vscode）下载，然后在 vscode 插件页面右上方的“…”按钮处，点击 install from vsix，就可以安装了。

2.2 etcd 配置过程

1. 服务器成功配置

etcd 部署在三台服务器上，分别为 10.77.70.61、10.77.70.62、10.77.70.63，按照晓桐师姐给出的指导，修改/home/centos/.bashrc 文件，把三台服务器组成一个集群，分别启动之后，三个 etcd 共用同一份数据。

以 10.77.70.61 为例，加入的内容为：

```
# for all machines
TOKEN=token-01
CLUSTER_STATE=new
NAME_1=machine-1
NAME_2=machine-2
NAME_3=machine-3
HOST_1=10.77.70.61
HOST_2=10.77.70.62
HOST_3=10.77.70.63
CLUSTER=${NAME_1}=http://${HOST_1}:2380,${NAME_2}=http://${HOST_2}:2380,${NAME_3}=http://${HOST_3}:2380
THIS_NAME=${NAME_1}
THIS_IP=${HOST_1}
ETCDCTL_API=3
ENDPOINTS=$HOST_1:2379,$HOST_2:2379,$HOST_3:2379
```

此外，还建议把 etcd 加入环境变量，例如 etcd 安装在~/etcd 目录下，则在~/.bashrc 文件中添加：export PATH="~/etcd:$PATH"。这样，就可以不需要一定要到 etcd 目录下去启动了。

启动的命令为，当测试成功之后，应该把这几个进程后台启动，以免阻塞窗口，这几个启动的进程在之后的编程过程中需要始终保持启动状态。

```
etcd --data-dir=data.etcd --name ${THIS_NAME} \
--initial-advertise-peer-urls http://${THIS_IP}:2380 --listen-peer-urls \
http://${THIS_IP}:2380 \
--advertise-client-urls http://${THIS_IP}:2379 --listen-client-urls
http://${THIS_IP}:2379 \
--initial-cluster ${CLUSTER} \
```

```
--initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKEN}
```

2. etcd 配置失败经验

1. 在 docker 中配置可以，但不能在同一台服务器的三个容器内，也不能使用三台虚拟机，这样不符合本课程真正的网络传输与分布式系统的要求，可以考虑在三台服务器上创建同一个 docker 镜像创建出来的容器，并且做端口映射，来进行服务器间的通信。Docker 的镜像不能多次打包，不然大小会变得很大，移动和下载都很不方便。

2. 在 Windows 笔记本的 wsl 中配置 etcd，会出现很多额外的困难，首先要解决 wsl 每次启动 ip 随机的问题，还需要把 wsl 的端口和笔记本的端口做映射，此外需要关掉所有防火墙，较为麻烦，而且如果小组成员不经常在一起，这种方法非常不方便。

2.3 mysql 离线安装

离线安装参看这篇文章：https://zhuanlan.zhihu.com/p/387253758。

首先需要卸载掉旧的版本，主要是卸载 rpm 安装包和删除安装目录下所有文件。

接下来，需要手动把 mysql 的安装包上传到服务器，解压，移动到一个安装目录，之后设置 mysql 的开机自启动服务。这里注意，如果是课程给的 centos 服务器并没有 service 命令，而是用 systemctl 命令代替。

建议把三台机器上的 mysql 用户名、密码设置为完全相同，这样方便之后去连接。

我们的实现中，在如果一个站点启动两个 server，会连接一个 mysql 中的两个 database，如果想要实现启动两个 mysql，需要先停止当前的 mysql 数据库，然后按照这篇文章步骤操作一下：https://blog.51cto.com/jack88/2051320。

2.4 python 离线安装

离线安装过程比较简单，我这里使用了 Anaconda，把安装的脚本 sh 上传到服务器，执行一下即可，记得中间所有过程都选择 y，并把环境变量添加到.bashrc 中。如果 anaconda 安装在~/anaconda 下，则添加的环境变量是：export PATH="~/anaconda3/bin:$PATH"，且添加之后记得要 source .bashrc。

这个时候，应该就可以看到 base 的虚拟环境了，当然自己也可以额外新建一个虚拟环境。在当前的虚拟环境，我们需要自己离线安装 python 包。

我的方法有两种：

1. 在 python 包的官网上找到符合自己当前操作系统的 whl 文件，进行下载，然后上传到服务器，依次手动安装。

2. （更为推荐）通过一个可以联网的机器 python 虚拟环境来安装

   (1) 找一个和服务器操作系统相同的虚拟机

   (2) 新建一个 python 虚拟环境（主要是为了保证之前没有安装过任何其他的包，以免干扰服务器环境）

   (3) 新建一个文件夹，不妨命名为 package

   (4) cd package，以安装 grpcio 为例，运行

```
pip install grpcio
pip download grpcio
```

(5) 当安装好之后，运行

```
pip freeze > requirements.txt
```

就可以自动生成需要的包的列表。此时，package 文件夹下面应该有一些 whl 文件，以及一个 requirements.txt 文件。

(6) 把这个文件夹打包上传到服务器，解压之后进入这个目录下，运行

```
pip install --no-index --find-links=./ -r requirements.txt
```

就可以自动安装所有的 python 包了。

(7) 之后有包要更新或者新增，可以手动把单个包的 whl 传上去，也可以如上步骤再来一遍。

(8) 可以用 pip list 来检查一下自己的包是否成功安装。

三、Metadata

我们分布式数据库的 metadata 写在 etcd 中，并且每个数据库对应的 metadata 在磁盘上有一份备份文件，用于数据库的切换和恢复。下面表 1 中我们将具体展示 etcd 中存储的 metadata 内容。

表 1 etcd 中存储的 metadata 说明

key	value	example
SITES	a list, all active sites' ip and port, and the db name they use	['10.77.70.61:8883:rpql1_8883', '10.77.70.61:8885:rpql1_8885', '10.77.70.62:8883:rpql1_8883', '10.77.70.63:8883:rpql1_8883']
sitenames	a dict that key is site name, value is site info in SITES	{'china': '10.77.70.61:8883:rpql1_8883', 'japan': '10.77.70.61:8885:rpql1_8885', 'britain': '10.77.70.62:8883:rpql1_8883', 'usa': '10.77.70.63:8883:rpql1_8883'}
DB	a list, all created database names	['rpql1', 'rpql2', 'rpql3']
/table	a list, all created table name in using database	['Publisher', 'Book', 'Customer', 'Orders']
/table/<table_name>/columns	a list of list, one list item is one column, the first is column name, the second is type, the third is primary key	[['id', 'int', 'key'], ['name', 'char(25)'], ['rank', 'int']]

/table/<table_name>/columnsize/<column_name>	a number, the avg size of this column, used for CE	9.0168
/table/<table_name>/fragment/<site_name>	a dict contains columns、conditions、sitename、siteinfo, supporting vertical and horizontal fragment, but size is discard, has moved to /table/<table_name>/lenfragment/<site_name>	{'columns': ['customer_id', 'book_id', 'quantity'], 'conditions': 'customer_id >= 307000 and book_id < 215000', 'sitename': 'britain', 'site': '10.77.70.62:8883:rpql1_8883', 'size': 0}
/table/<table_name>/lenfragment/<site_name>	a integer, the row number of fragment	5000
/attrinfo/<table_name>.<column_name>	statistic information for a column	{'attr': 'Book.copies', 'type': 'U', 'numbers': ['0', '10000', '50000']}
/site/<site_name>/fragment/<table_name>	the same as /table/<table_name>/fragment/<site_name>	{'columns': ['id', 'title', 'authors', 'publisher_id', 'copies'], 'conditions': 'id < 205000', 'sitename': 'china', 'site': '10.77.70.61:8883:rpql1_8883', 'size': 0}
/tree	the query tree after encoding	—

四、grpc 及网络传输

4.1 grpc proto

Proto 中定义了 server 所包含的方法以及这些方法的接收信息格式和返回信息格式。

```
syntax = "proto3";
package net;
service NetService {
  //test method
  rpc Test(Data) returns (Data1){}
```

```
  rpc Createtable(SQL) returns (Status){}
  rpc Droptable(SQL) returns (Status){}
  rpc Loaddata(LoadParams) returns (Status){}
  rpc Insertdata(LoadParams) returns (DataReturn){}
  rpc Deletedata(SQL) returns (DataReturn){}
  rpc SimpleSelect(SQL) returns (SimpleSelectReturn){}
  rpc Excute(SQLTree) returns (TableData) {}
  rpc jr_grpc_test(para_jr_grpc_test) returns (ret_jr_grpc_test) {}
  rpc grpc_dfs(para_grpc_dfs) returns (ret_grpc_dfs) {}
  rpc start_jr(para_start_jr) returns (ret_start_jr) {}
  rpc temp_GC(para_temp_GC) returns (ret_temp_GC) {}
  rpc createdatabase(para_dbname) returns (dbres) {}
  rpc dropdatabase1(para_dbname) returns (dbres) {}
  rpc dropdatabase2(para_dbname) returns (dbres) {}
  rpc dropdatabase3(para_dbname) returns (usedbres) {}
  rpc usedatabase1(para_dbname) returns (dbres) {}
  rpc usedatabase2(para_dbname) returns (dbres) {}
  rpc usedatabase3(para_dbname) returns (usedbres) {}
  rpc jr_exit(para_jr_exit) returns (ret_jr_exit) {}
}
// 输入输出的参数格式详情略
```

写完之后，执行：

```
    python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. ./net/net.proto
```

可以自动更新 net_pb2.py 和 net_pb2_grpc.py，net_pb2.py 会自动封装一个方法接受和返回的数据格式，net_pb2_grpc.py 会调用那些格式组成 server 的方法，之后我自己可以写一个类继承 net_pb2_grpc.py 中自动生成的类，在我们自定义的类中具体去实现 server 的方法。

4.2 grpc client 发送数据

client 的使用方法是首先构建一个 channel，需要的信息是 ip 和 port，然后通过 client 调用 server 的方法，传入的参数在 proto 中需要定义好，这里是定义了一个 para_dbname 类型，包含一个参数是 dbname，并且得到 response。Response 中包含的就是在 proto 定义的信息。

```
    conn = grpc.insecure_channel(ip + ":" + port)
    client = net_pb2_grpc.NetServiceStub(channel=conn)
    # print("start")
    response = client.createdatabase(net_pb2.para_dbname(dbname=dbname))
```

4.3 grpc server 接收数据

以上面用到的 createdatabase 方法为例，首先接收到 request 参数，从中得到 dbname，

然后执行中间的逻辑，最后返回的时候，返回的类型也需要在 proto 中定义好，和 client 端接收的 response 相同。这里是返回了一个 dbres，内含一个变量，是 res，这里给 res 赋值为“yes”。

```
def createdatabase(self, request, context):
        dbname = request.dbname + '_' + _PORT
        conn =
pymysql.connect(host='localhost',user=_MYSQL_USER,password=_MYSQL_PASSWORD,charset='utf8mb4')
        # 创建游标
        cursor = conn.cursor()
        # 创建数据库的 sql(如果数据库存在就不创建，防止异常)
        sql = "CREATE DATABASE IF NOT EXISTS " + dbname + ";"
        # 执行创建数据库的 sql
        cursor.execute(sql)
        cursor.close()
        # 关闭数据库连接
        conn.close()
        return net_pb2.dbres(res="yes")
```

4.4 多线程

多线程用于并发地去执行一些操作，比如同时向多个站点的 mysql 插入数据。这里需要定义好一个函数，然后再创建一个线程池，之后多次提交任务，把要调用的函数和参数传进去。之后应该有一个等待操作，等候所有任务执行完成，再进行下一步操作。

有时候多线程不能一步到位，需要拆成多个阶段，每个阶段要等待所有线程执行到这个阶段的结束才能执行下一个阶段。

```
def tmp(i, ip, port, table_name, fragment_columns, bytedata):
    conn = grpc.insecure_channel(ip + ":" + port)
    client = net_pb2_grpc.NetServiceStub(channel=conn)
    try:
        response = client.Loaddata(
            net_pb2.LoadParams(
                table_name=(table_name),
                fragment_columns=(str(fragment_columns[i])),
                data=bytedata,
            )
        )
        return (response.status, ip, port)
    except:
        print(f"there is no table {table_name} on {ip}:{port} or {ip}:{port} cannot be connected",
time.asctime())
        return ("something is wrong", ip, port)
```

```
tasks = []
with ThreadPoolExecutor(max_workers=5) as t:
    for i in range(len(fragment)):
        subdata = data[i]
        len_of_fragment = len(subdata)
        # reverse_sitenames = {v:k for k,v in sitenames.items()}

        etcd.put("/table/{}/lenfragment/{}".format(table_name, fragment[i]["sitename"]),
str(len_of_fragment))
        bytedata = pickle.dumps(subdata)
        ip, port, db = fragment[i]["site"].split(":")
        print("start", i)
        # tasks.append(tmp(i, ip, port, table_name, fragment_columns, bytedata))
        task = t.submit(tmp, i, ip, port, table_name, fragment_columns, bytedata)
        tasks.append(task)

    for future in as_completed(tasks):
        message, ip, port = future.result()
        if ip+":"+port in namesites:
            sitename = namesites[ip+":"+port]
            print(f"message: {message} on {sitename}", time.asctime())
        else:
            print(f"message: {message} on {ip}:{port}", time.asctime())
```

五、查询树

5.1 结构定义

1. 树节点结构

```
class Node:
    def __init__(self, index: int, type:str, parent:int, children:List[int], tables:List[str],
                 site:str, size:int, columns:List[str], if_valid:bool=True, f_id:int=None,
                 f_name:str=None, select_condi: List[List[str]]=None, projection:List[str]=None,
                 join:List[str]=None, top:int=None):
        self.id = index                    #node id, start from 0
        self.type = type                   #node type: fragment, select, projection, join, union
        self.parent = parent               #the parent node id, default -1
        self.children = children           #the children node ids, defalut []
        self.tables = tables               #names of all tables included in this node
        self.site = site                   #which site this node is in
        self.size = size                   #total bytes of data in this node
        self.if_valid = if_valid           #if this node has been pruned
        self.columns = columns             #names of all attributes included in this node
        self.f_id = f_id                   #if type is 'fragment', the fragment id
        self.f_name = f_name               #if type is 'fragment', the table name
        self.select_condi = select_condi       #if type is 'select', the select condition
        self.projection = projection       #if type is 'projection', the project attributes
        self.join = join                   #if type is join, the two join keys
        self.top = top                  #if type is 'fragment', the node id after select and project
```

查询树中一个节点的结构如上图所示。其中比较重要的是，id 存储节点序号，type 存储节点类型，有 fragment, select, project, join, union 五种，涵盖了查询树中所有节点。parent 存储父节点的序号，children 存储所有子节点的序号，site 存储此节点所在站点，size 存储此节点估计的数据量大小，用于传输优化。如果节点类型是 fragment，f_id 和 f_name 分别存储分片序号和表名；如果节点类型是 select，select_condi 把条件存成一个长度为 3 的 list；如果节点类型是 projection，projection 把投影后的属性存在一个 list；如果节点类型是 join，join 把连接的 key 存成一个长度为 2 的 list。最后的树就是一个由多个 Node 组成的 list。

2. 查询树生成过程中用到的其他结构

Table 结构，Fragment 结构，select 条件结构和 attribute 结构如下图所示。

```
class Attribute:
    def __init__(self, table: str, attr: str, if_join_attr: bool=False, f_id: int=None):
        self.table = table       #table name
        self.attr = attr        #attribute name
        self.f_id = f_id        #fragment id if this attribute is a condition for vertical fragment
        self.if_join_attr = if_join_attr #if this attribute is included in a join condition
        #self.s_id = s_id
```

```
class Selection:
    def __init__(self, attr: Attribute, operator: str, f_id: int, s_id:int,
                 num_value: float=None, str_value: str=None):
        self.attr = attr       #attribute name
        self.operator = operator       #operator > = < >= <=
        self.f_id = f_id             #fragment id if this select is a condition for
                                     #horizontal fragment, start from 0
        self.s_id = s_id           #site id if this select is a condition for horizontal fragment,
                                    #start from 1, 1,2,3,4
        self.num_value = num_value    #if the value is a number
        self.str_value = str_value    #if the value is a string

class Fragment:
    def __init__(self, id:int, table: str, s_id: int, size: int, hor_or_ver: int,
                 hori_condi: List[Selection]=None, verti_condi: List[Attribute]=None):
        self.table = table       #the table name this fragment is in
        self.id = id            #the fragment id, start from 0
        self.s_id = s_id        #the site id, start from 1, 1,2,3,4
        self.size = size        #total bytes of data in this fragment
        self.hor_or_ver = hor_or_ver       #horizontal of vertical fragment
        self.hori_condi = hori_condi       #horizontal fragment conditions
        self.verti_condi = verti_condi     #vertical fragment attributes

class Table:
    def __init__(self, table:str, frag_list:List[Fragment], attributes:List[str]):
        self.table=table                #table name
        self.frag_list = frag_list      #the fragments of this table
        self.attributes = attributes    #attributes of this table
```

5.2 查询树生成过程

1. select 语句解析

解析 select 语句我们主要利用 python 的 sqlparse 库，他可以识别出来语句中哪些 token 是要选择的属性，哪些 token 是 where 后面的连接、选择条件。遍历这些 token，进行进一步的判断和整合，我们得到了这些在生成查询树时会用到的信息：

变量名	变量类型	含义
if_project_all	bool	是否投影所有属性（是否是 select * from）
selects	List	所有选择条件，每个条件是长度为 3 的元组
joins	List	所有连接条件，每个条件是长度为 3 的元组
all_need_attributes	List	语句涉及到的所有属性，包括选择、连接属性
tables	List	涉及到的所有表
final_attributes	List	最后一步投影出的属性
exclude_select_attributes	List	除去选择条件以外所有涉及到的属性
join_extra_selects	List	选择条件涉及 T1.a，连接条件有 T1.a=T2.b，则 T2.b 与 T1.a 条件相同加入选择条件，用于选择剪枝

2. 基数估计的方法

在之前 load data 的过程中，就在 etcd 里存储了每个表每个属性的大小（单位：bytes），基本上是按照字段长度统计，例如 Book.title 是长度为 12 的字符串，大小就记 12。若有的

属性每个记录大小不同，就取平均值。由此，可以得到每个分片的初始大小，就是用行数×所有属性大小总和。

所有的表的统计信息也通过 load data 后的 add info 语句加入到了 etcd 中。关于选择操作，如果选择的属性与分片条件无关，默认每个分片的该属性分布相同，结合已给的统计信息，算出进行选择操作后剩下多大部分的数据。例如 Q10 中的选择条件 Book.copies>100，统计信息是 copies 在 0 到 10000 中均匀分布，就可以算出 B1, B2, B3 这三个分片剩下的数据大约均是原来的 99/100。如果选择的属性与分片条件无关，那么如果分片条件是某属性在 [a, b]之间，选择条件是该属性在[c, d]之间（因为这个场景下不存在无限大或无限小的属性值，所以 a, b, c, d 都会是确切值），也比较容易能通过 a, b, c, d 的大小关系分类讨论得出，选择后的 new_size 占原来 old_size 的几分之几，代码里比较详细，这里不多赘述了。

关于投影操作，因为已知了每个属性的大小，所以很容易通过：

new_size =（投影后属性大小总和/投影前属性大小总和）× old_size

算出新的该分片的大小。

关于 join 操作和 union 操作，因为这两个操作的基数估计只会用在最后一步决定汇总到哪个 site 上，不像选择操作和投影操作会影响传输优化，所以精确度没有要求很高，只要能反应大小关系就行，而且往往不同 sites 大小的差距已经很明显，我们就用的把两个分片或多个分片的大小加和来估计。

3. 查询树生成步骤

以 Q10 为例说明查询树的生成过程：

```
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'
```

① **得到 query 语句涉及到的所有表的所有分片，作为查询树的叶子节点**

在 Q10 中四个表都有涉及，所以可能用到的分片就是 P1, P2, P3, P4, B1, B2, B3, C1, C2, O1, O2, O3, O4。

② **对于所有的垂直分片，检查其属性是否全是不需要的属性，若是，剪掉该垂直分片，删除该叶子节点**

垂直分片 C1 和 C2，其中 C2 的属性除了用于垂直分片连接的主键 Customer.id 以外只有 Customer.rank，后面不需要，所以需要剪掉 C2。

③ **对于剩余的垂直分片，看每个分片中涉及的属性是否有在选择条件中的，若有，为分片设置 select 类型的父节点。**

垂直分片还剩余 C1，Customer.id 在选择条件中，所以为 C1 设置条件为 Customer.id>308000 的 select 类型父节点。

④ **对于 query 所有的选择条件，检查是否有分片条件和它冲突的水平分片，若冲突，剪掉该水平分片，删除该叶子节点**

在 Q10 中，有条件 Customer.id>308000，根据上述生成的变量 join_extra_selects 可知，也同样有条件 Orders.customer_id>308000，这和 O1 的分片条件 customer_id < 307000 and book_id < 215000，O2 的分片条件 customer_id < 307000 and book_id >= 215000 冲突，应把 O1 和 O2 这两个分片剪掉。接着 Book.copies 和 Orders.quantity 不在分片条件中。条件 Publisher.nation='PRC'，和 P2 的分片条件 id < 104000 and nation='USA'，P4 的分片条件 id >= 104000 and nation='USA'冲突，所以要剪掉 P2 和 P4。

⑤ **对于每个剩余的水平分片，若有，设置关于该表的选择条件的 select 类型节点**

剩余的水平分片还有 P1, P3, B1, B2, B3, O3, O4，要为 B1, B2, B3 分别设置条件为 Book.copies>100 的 select 类型父节点，为 O3, O4 分别设置条件为 Orders.quantity>1 的 select 类型父节点。

⑥ **对于现在的所有分片（包括水平分片和垂直分片），若分片中含有之后的连接操作和最后的投影操作不需要的属性，为这些分片设置 projection 类型父节点**

Publisher 表在后面只需要 Publisher.id 和 Publisher.name，P1 和 P3 还包含一个 Publisher.nation，所以需要给 P1、P3 分别设置一个投影父节点。Book 表到后面只需要 Book.id, Book.title, Book.publisher_id，B1, B2, B3 中还包含 authors 和 copies，所以也给 B1, B2, B3 分别设置一个投影父节点。C1 中的 Customer.id 和 Customer.name 后面都需要，所以不做投影操作。（在其他 query 比如 Q8 中，会有因为 Customer.rank 在选择操作后不再需要，要把 C2 投影出 Customer.id 的情况。）O3, O4 中的 Orders.quantity, Orders.book_id, Orders.customer_id 后面都需要，不做投影操作。

⑦ **连接所有垂直分片，把所有分片都发往 size 最大的分片所在站点**

Q10 中不涉及此步骤。

⑧ **考察每个连接条件关联的两个表的所有分片，如果它们的分片条件冲突，剪掉这两个分片之间的连接**

Q10 中 Book.id=Orders.book_id 这个条件，Book 表的分片（B1, B2, B2）和 Orders 表的分片（O3, O4）的所有组合中，因为 B1 的条件是 id < 205000，O4 的条件是 customer_id >= 307000 and book_id >= 215000，连接条件是冲突的，所以剪掉。同理，B2 的条件是 id >= 205000 AND id < 210000，也和 O4 冲突了，剪掉。

⑨ **传输优化**

我们主要采用的是 Distributed INGRES QOA 算法决定连接操作在哪几个站点上执行。经过基数估计，现在得到的每个分片的大小如下表：

	site3	site4	site1	site2
Publisher	12190		46621	
Book	950423		118776	118800
Customer			105000	
Orders	184796	435846		

按照算法应该选择 Book 表为 Rp，上表已按站点各分片大小总和，从大到小排列。

j=1 时，46621+105000+435846 < 950423，因此选择 site3；

j=2 时，12190+46621+105000+184796 > 0，因此包括此之后的 sites 都不选择。

⑩ 判断哪些组合在哪些站点上执行，以及哪些分片都需要发往哪些站点

在第⑧步进行 join 剪枝之后，得到了所有应该被 join 的分片组合，如下（table 的顺序按照 query 里 join 条件的顺序）：

C2 O3 B1 P1
C2 O3 B2 P1
C2 O3 B3 P1
C2 O4 B3 P1
C2 O3 B1 P3
C2 O3 B2 P3
C2 O3 B3 P3
C2 O4 B3 P3

对于每一个组合，若其中有一个分片所在的站点属于连接操作执行站点，则把这个组合所有其余分片都发往这个站点；若有多个分片所在站点都属于连接操作执行站点，则发往 size 最大的分片所在的站点；若没有分片所在站点属于连接操作执行站点，则把组合里所有分片发往 size 总和最大的执行站点。

由于此例中只有 site3 作为执行站点，因此所有分片都发往 site3，需要发往 site3 的分片有 P1, B1, B2, C2, O4。

⑪ 在每个站点上对该站点所有组合进行合并（判断哪些 table 的分片可以先 union 再做 join）

例如 site3 上的上一步的那些分片组合，我们通过算法（具体在代码中体现），把这 8 个组合合并成了两个 pattern：

C2 O3（B1|B2|B3）（P1|P3）

C2 O4 B3（P1|P3）

这些 pattern 中用括号括起来的部分是要 union 起来的分片。对于这两个 pattern 中相同的部分，即 Customer 表和 Publisher 表上下都是 C2 和（P1|P3），可以直接保留，作为等待 join 的节点（P1 和 P3 要先 union 起来）。对于这两个 pattern 不同的部分，就是 Orders 和 Book 表，在此例中，对于第 1 个 pattern，需要先为 B1, B2, B3 设置 union 父节点 U1，然后再为 O3 和 U1 设置 join 父节点 J1，对于第 2 个 pattern，由于没有需要先 union 的分片，所以直接为 O4 和 B3 设置 join 父节点 J2，最后再为 J1 和 J2 设置 union 父节点 U2，作为 Orders 表和 Book 表的 join 结果。

现在有 3 个待 join 的节点，分别是 Customer 表的，Orders 和 Book 表已经 join 起来的，和 Publisher 表的，把这三个节点依次 join 即可，得到了一个站点上的 join 结果。

⑫ 如果需要，对每个站点上最后 join 的结果做投影（去掉那些用于连接的键值）

为了减小传输代价，需要先把结果进行投影，再发往最终的站点汇总。投影的结果就是 query 中 select 后面需要的那几个属性。此例中，设置 projection 父节点投影 Customer.name, Book.title, Publisher.name, Orders.quantity 这几个属性。

⑬ 把每个站点的结果 union 起来

最后一步很简单，就是设置 union 父节点，把每个站点上 11~12 步处理后的结果 union 到 size 最大的那个站点上。此例中由于只在 1 个 site 上所以不需要 union。

最后生成的查询树就是：

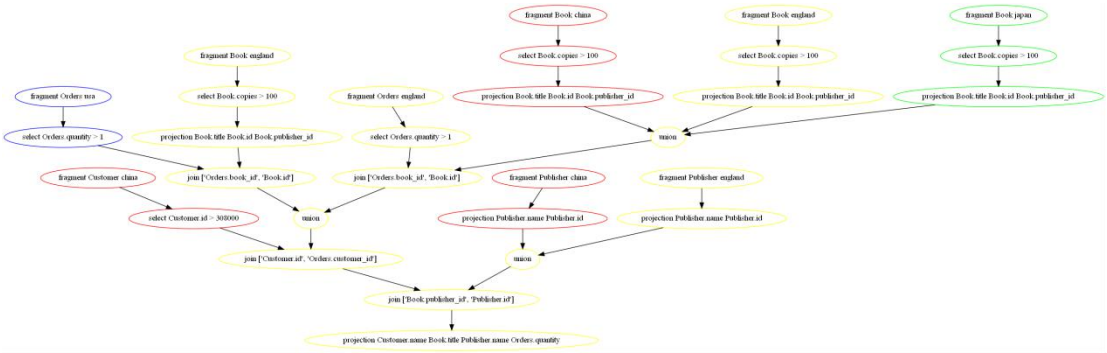

图 6 Q10 查询树

六、查询执行

6.1 总体框架

对查询树做并行的深度优先搜索（DFS），对于不同类型的结点，对应有不同的操作：

（1）fragment 结点：该结点返回一个分片的所有数据，是查询树的叶子结点。处理流程如下：①读取查询树中该结点对应的分片的标识符，在 etcd 中读取该分片的元数据；②执行 SELECT 列 1,列 2,…,列 m FROM 表；③返回列的元数据和查询结果。

（2）select 结点：此类结点有一个孩子，对孩子结点的中间结果表做条件选择操作。处理流程如下：①dfs 唯一的孩子；②把孩子结点的结果写入中间结果表；③执行 SELECT * FROM 中间结果表；④返回列的元数据和查询结果。

（3）projection 结点：此类结点有一个孩子，对孩子结点的中间结果表做投影。处理流程如下：①dfs 唯一的孩子；②把孩子结点的结果写入中间结果表；③执行 SELECT 投影列 1,投影列 2,…,投影列 m FROM 中间结果表；④返回列的元数据和查询结果。

（4）join 结点：此类结点有两个孩子（关系树里的单个结点只存在两表连接的情况，多表连接会被拆成多结点的树结构），对两个孩子结点的中间结果表做连接。处理流程如下：①dfs 两个孩子；②把孩子结点的结果分别写入中间结果表 1 和中间结果表 2；③执行 SELECT * FROM 中间结果表 1,中间结果表 2 WHERE 中间结果表 1.连接列 1 = 中间结果表 2.连接列 2；④返回列的元数据和查询结果。

（5）union 结点：此类结点有若干孩子，对多个孩子结点的中间结果表做连接。处理流程如下：①dfs 每个孩子；②把孩子结点的结果分别写入中间结果表 1,中间结果表 2,中间结果表 k；③执行 SELECT * FROM 中间结果表 1 UNION ALL SELECT * FROM 中间结果表 2 UNION ALL … UNION ALL SELECT * FROM 中间结果表 1,中间结果表 2,...,中间结果表 k；④返回列的元数据和查询结果。**值得注意的点：UNION 关键词会过滤两表相同元组，而 UNION ALL 不会。**

6.2 基于 gRPC 的远程过程调用

假设 DFS 现在正处于查询树中的任意一个结点 x0，如果 x0 的某个孩子 x1’就在本 site，

则直接对 x1'进行 DFS，如果 x0 的另一个孩子 x1''不在本 site，则需要对远程调用 x1''所在 site 上的 DFS 函数。相当于，这个 DFS 递归函数，是本地与远程结合的。上文已经提到 gRPC，我们利用此进行 DFS 的远程过程调用。gRPC 可以调用的方法需要在 net.proto 中注册，且传递的参数和返回值需要包装为数值或字符串（像 list、tuple、dict 等可迭代对象，应当转化为字符串传递，使用时再转回可迭代对象）。

为了提高代码的可读性和可维护性，我们并不会把 DFS 重写一个 gRPC 版本（从 6.1 可以看出，这个 DFS 代码量较大），而是新增了一个简短的函数 grpc_dfs，其任务仅仅是把传递的参数转回可迭代对象调用本地的 DFS，再把本地 DFS 的返回值转成数值和字符串返回给远程调用端。也即孩子与家长在同一 site，直接调用 DFS，孩子与家长在不同 site，调用 grpc_dfs。

直接这样这样做，DFS 还需判断递归孩子需要远程调用还是本地调用，会进一步加大 DFS 的代码量，不便维护。因此，由单独编写一个 dfs_execute 函数，DFS 对于每个孩子调用 dfs_execute，传入结点及其想要递归的孩子的信息，流程为：①如果孩子和家长在同一个 site，调用 DFS；如果孩子和家长不在同一个 site，调用 grpc_dfs；②**家长结点本地**新建临时表（因为中间结果要被家长查询），插入孩子返回的结果；结果把孩子结点的结果分别写入中间结果表 1,中间结果表 2,中间结果表 k；③返回中间结果的元数据。

6.3 基于线程池的多线程并行

查询树是一棵树，由于树的无环性，在 DFS 的过程中，孩子之间不会交叉调用，而是独立成两条支路不再会面。因此，递归孩子的过程可以并行。我们使用线程池技术，资源池化的好处是，节省每次新建对象的开销，同时有效控制内存的使用。每个结点 DFS 时：①先建立一个包含若干线程启动者（比如 5 个）的线程池，把对每个孩子执行 dfs_execute 的动作加入线程池的队列（启动者有空闲会立即并发执行，否则在队列中等待分配启动者）；②等待所有孩子的 dfs_execute 线程执行完毕；③进行下一步操作，查询孩子的中间过程得到家长的中间过程，并返回给家长的家长。

6.4 执行模块的方法层次

底层工具：（1）sql_gen，根据查询树结点的信息，返回本结点调用 MySQL 的 SQL 查询指令；（2）sql_execute，连接 MySQL，执行 SQL 查询指令，返回结果（根据执行的需要进行的包装）。

执行部分：（1）dfs_execute，每次 DFS 调用的起点；（2）grpc_dfs，远程过程调用，负责参数转换和转回；（3）DFS，递归查询树。

各函数/方法之间的调用关系如下：

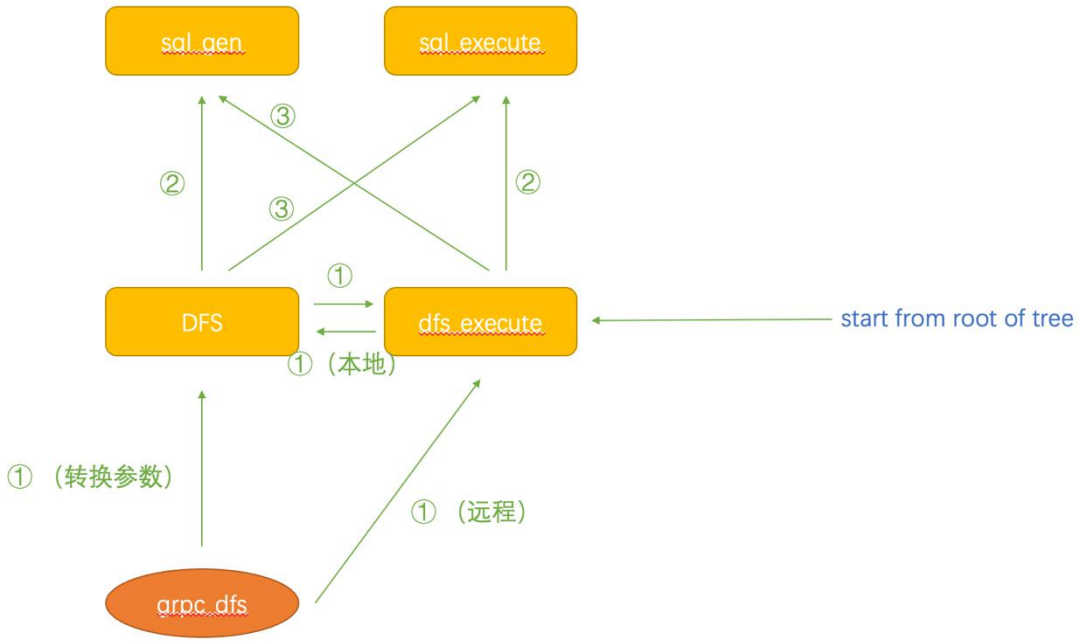

图 7 查询执行模块的方法层次

七、测试结果

7.1 插入数据测试的分块结果和时间

表 2 插入数据测试结果

表名	分块	行数	时间
Publisher	china	2027	1.15
	japan	1972	
	england	530	
	usa	471	
Book	china	4999	16.94
	japan	5000	
	england	40001	
Customer	china	15000	6.64
	japan	15000	
Orders	china	13896	16.12
	japan	32606	
	england	15929	
	usa	37569	

7.2 查询语句测试的返回结果和时间

表 3 查询语句测试时间

	语句	时间	结果行数	传输数据量
Q1	select * from Customer;	1.39	15000	195049
Q2	select Publisher.name from Publisher;	0.58	5000	71499
Q3	select Book.title from Book where copies>5000;	3.7	24906	94034
Q4	select customer_id, quantity from Orders where quantity < 8;	6.18	99346	806420
Q5	select Book.title,Book.copies, Publisher.name,Publisher.nation from Book,Publisher where Book.publisher_id=Publisher.id and Publisher.nation='USA' and Book.copies > 1000;	7.01	21923	433073
Q6	select Customer.name,Orders.quantity from Customer,Orders where Customer.id = Orders.customer_id;	7.6	100000	2022547
Q7	select Customer.name,Customer.rank, Orders.quantity from Customer,Orders where Customer.id=Orders.customer_id and Customer.rank=1;	6.35	41098	1331344
Q8	select Customer.name ,Orders.quantity, Book.title from Customer,Orders,Book where Customer.id=Orders.customer_id and Book.id=Orders.book_id and Customer.rank=1 and Book.copies>5000;	16.8	20612	2967419
Q9	select Customer.name, Book.title, Publisher.name, Orders.quantity from Customer, Book, Publisher, Orders where	9.59	20209	3321130

	Customer.id=Orders.customer_id and Book.id=Orders.book_id and Book.publisher_id=Publisher.id and Book.id>220000 and Publisher.nation='USA' and Orders.quantity>1;			
Q10	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';	17.14	16345	1105543
Q11（现场测试）	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<5 and Publisher.nation='USA';	18.83	18987	1238032

7.3 查询语句生成的查询树展示

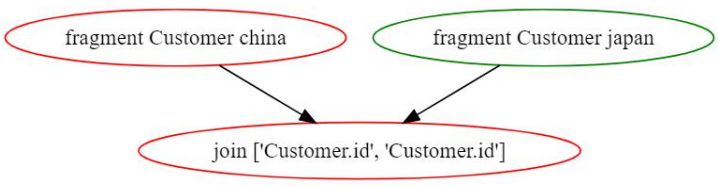

图 8 Q1 查询树

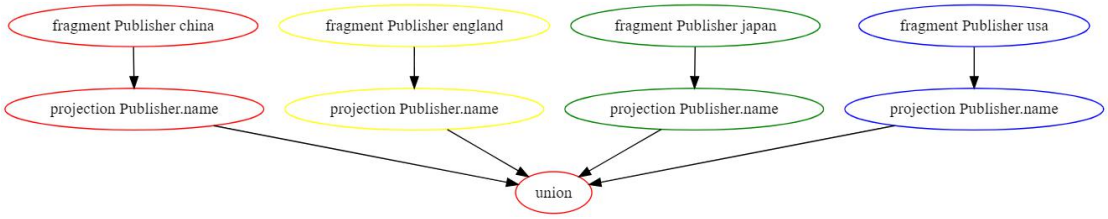

图 9 Q2 查询树

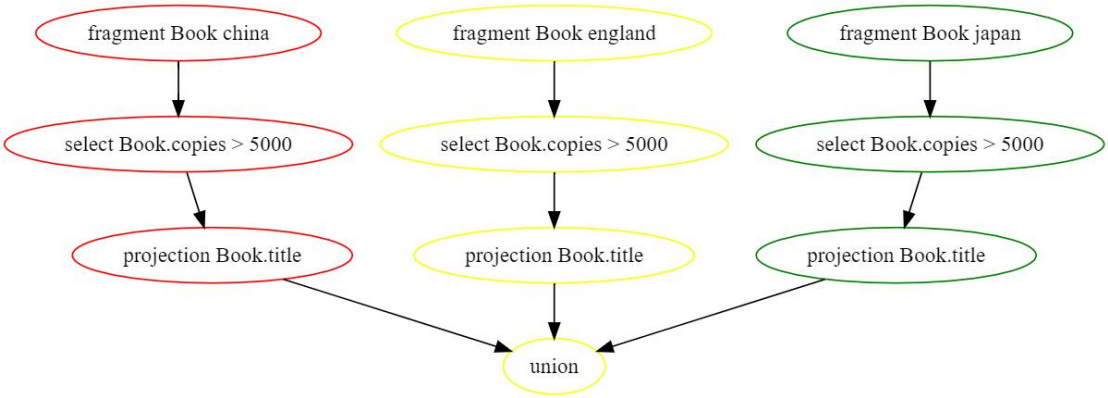

图 10 Q3 查询树

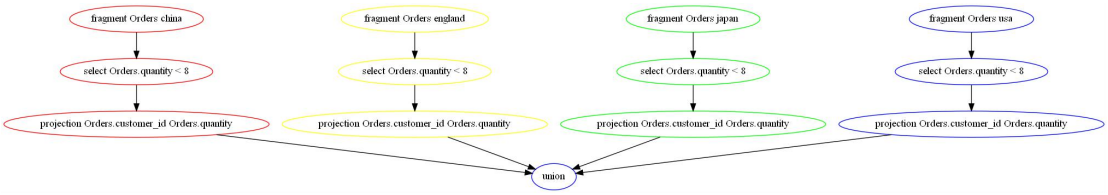

图 11 Q4 查询树

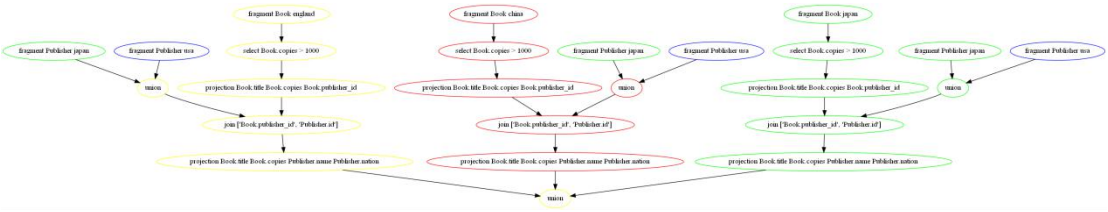

图 12 Q5 查询树

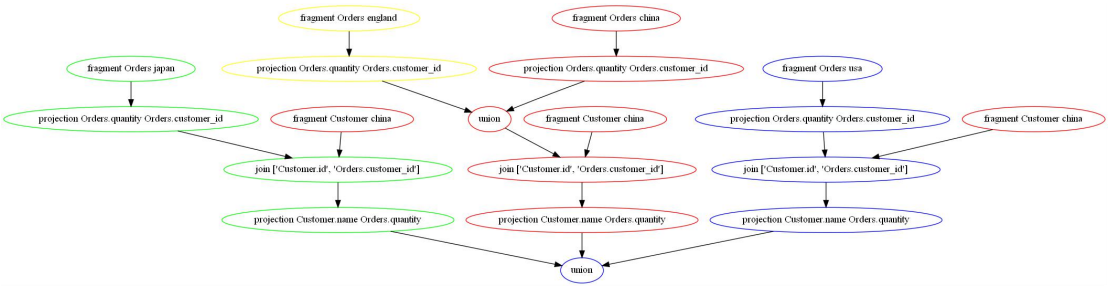

图 13 Q6 查询树

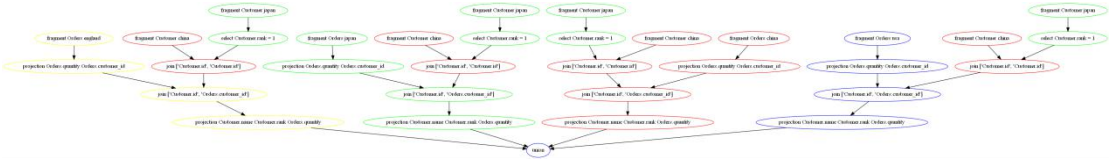

图 14 Q7 查询树

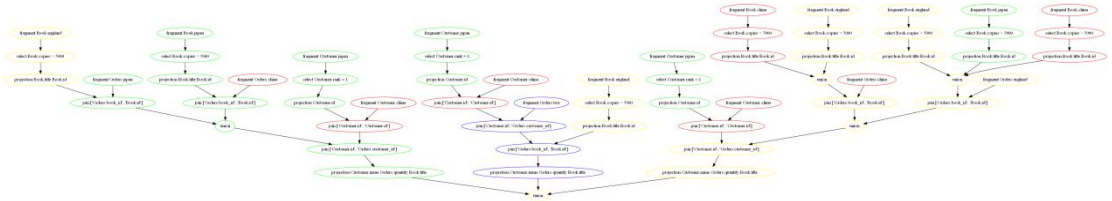

图 15 Q8 查询树

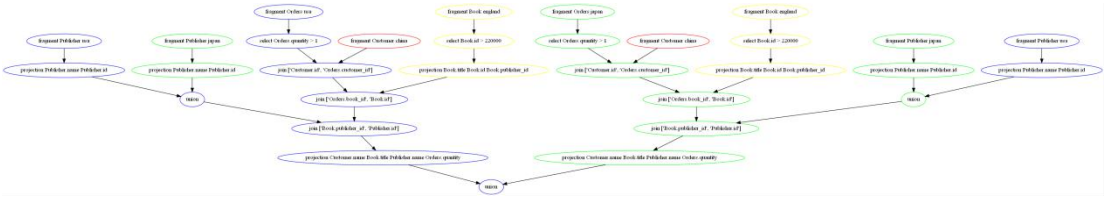

图 16 Q9 查询树

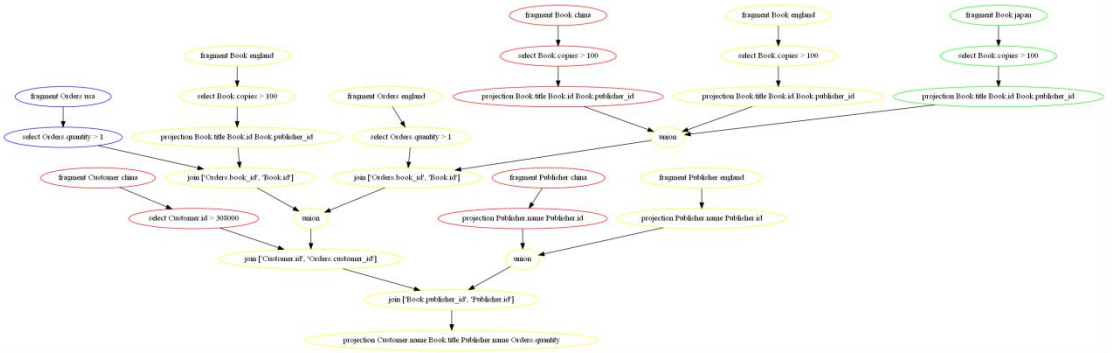

图 17 Q10 查询树

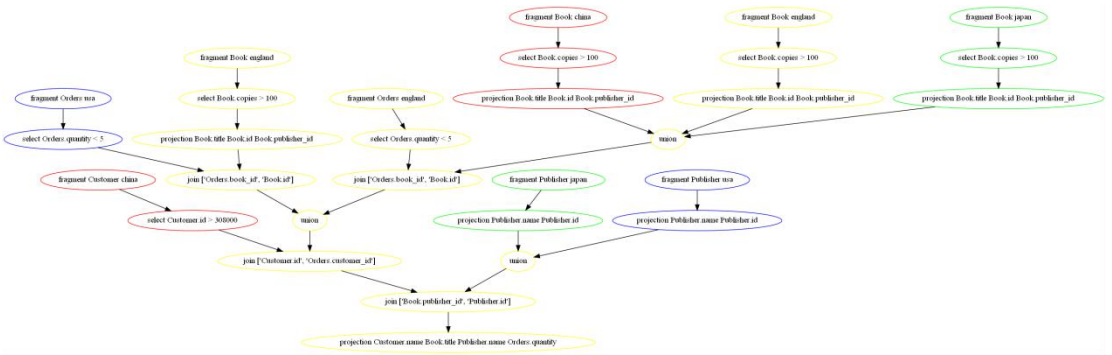

图 18 Q11 查询树

八、总结感想

8.1 薛钦亮

这次开发真的是本学期最艰难的一段时间，也是一次非常难忘的经历，更是我这学期收获最大的一个作业。这次作业的迟交，我没有规划好最后几天的开发进度，有很大的责任。这次作业的完成，最大的因素是队友的靠谱以及共同的目标，感谢芃芃和佳伟这一段时间的共同付出。下面是我的个人收获：

首先，提升了自己的调研能力。我们组开始较晚，刚开始的时候完全不懂 etcd、 grpc 怎么用，也没有往届小组使用 python，没有经验可以参考，我用一天时间快速调研 python 的可行性以及 python 中 etcd 和 grpc 的用法，实现了第一条简单的“select * from Publisher;”语句。

其次，提升了自己对于架构和模块的设计能力。对于整体架构的设计，我在第一个粗糙的版本上逐步修改了很多次，从用户的角度出发，思考什么样的架构是合理的架构，什么样的用法是好用的用法，确定了面向用户的接口之后，再去优化服务端的设计，比如一开始我曾考虑用户只与一个主 server 连接，这个主 server 与其他几个 server 连接，但考虑到 client 可以启动在任意一个地方，应该允许 client 选择当下最优的节点去连接。几个 server 之间我曾考虑把连接时创建的 channel 写在具体的函数中，需要创建时创建，后来考虑到这样会涉及反复的创建销毁，应该在 server 启动时就建立固定数量的连接，来节省开销。考虑到分布式的集群应该是动态的，所以摒弃了一开始想的把节点信息写死在配置文件中的做法，而改为通过 etcd 动态发现与更新节点信息。考虑到 pymysql 和 etcd 的设计缺点，自行封装了两个新的类来执行这一部分的功能，在开发上更加简洁和方便。在考虑清楚这些内容之后，剩下的就是在 client 和 server 上填写具体的逻辑了。

再次，提升了自己配置环境，尤其是离线配置环境的能力。这是一个考验心态的过程，仅仅是选用的环境就一共有三版考虑。第一次考虑在一台可以联网的阿里云启动三个 docker 容器，但不符合课程要求被否定，第二次考虑三台笔记本，但迟迟没有与王芃成功连通，最终又放弃，第三次已经到 12 月中旬，开始配置服务器上的环境，抽丝剥茧一般地配置所需要的环境和解决一些依赖问题，一直临近检查的前一两天，我们的环境还出现过重大问题，只得耐心地一点一点去检查报错，去网络上搜索需要手动安装的包。以及在配置环境的时候也需要思考，哪些是必需的，哪些可以有，配置成什么样的环境最方便使用，我配置了三台服务器的免密码登录，改了三台服务器的 hostname，用 python 创建了虚拟环境，把 mysql 和 etcd 加入到了开机自启动中，摸索出来一套成熟的 python 包离线安装方法，在其他队友加入开发之前把 vscode 上需要的插件全部离线安装完成，这些对于之后的开发起了很大的帮助。

最后，提升了自己的团队合作能力。在这个过程中，我意识到了明确的分工、清晰的接口、详尽的文档都是很重要的。由于我们团队三人考试集中的时间并不一致，所以一些工作并不是同步在开展，而是有先后的，这个时候我才意识到明确的分工以及清晰的文档的重要性，很多架构方面的细节我一开始并没有留好文档，当后期队友需要在这上面开发的时候，我才发现应该提供一个详尽的文档来节约时间。幸好我们初期调研比较充分扎实，补充文档的工作我很快就完成了，以及我们的分工十分明确，架构设计层次分明比较合理，大家互相留下的接口也都很清晰，所以一旦某一个模块完成之后，可以很快地与框架对接上并开展测

试，并且第一次测试就顺利通过了绝大部分的测试点，也让我们感到有些惊奇。

8.2 王芃

这次作业中我负责的主要是生成 select 语句的查询树。一开始构思的时候觉得先建立最原始的查询树，再优化改变树的结构比较麻烦，时间可能也比较长，所以打算从叶子节点从底向上直接建立优化后的查询树。前面结构定义，选择和投影的剪枝相对来说比较容易上手开始写，虽然要考虑的东西挺多，但是都是比较固定的。但是从传输优化、连接那部分开始我推进得很慢，在纸上根据估算的分片大小计算传输优化的方法就算了快两天，一开始甚至以为把 join 条件的表两两配对就可以，后来才反应过来可以 union 的结果应该都是所有表的分片 join 的结果，并且要把所有分片的组合都考虑到。一开始想的是用贪心顺序按照连接条件一步步选择哪个站点，很明显很多情况都不是最优。在终于选择好了传输优化的方法之后，又遇到一个问题，因为看 benchmark 中示例的查询树是会先把同一个表的分片 union 起来再 join，但又不能简单地直接 union，因为这样剪枝掉的连接就不会反映出来，又想了一个下午加一个晚上，才想出了把所有的组合分情况合并的算法，对于涉及到连接剪枝的表，进行先 union 再 join 再 union。这部分的代码逻辑也是想了很久才想通顺。传输优化和 join 这一块是我主要遇到的困难。

到后面和 etcd 对接读取分片信息算是比较顺利，稍微修改一下就能得到行数、属性大小这些信息。

我收获也是非常大的，第一次写查询树，要考虑的步骤和情况很多，从零开始想优化方法，是代码量最大的大作业了。我也非常佩服钦亮和佳伟，他们的工作都完成得高效、正确且清晰，还主导做了好几个优化和选做，和他们对接的过程都很顺利，基本上没有什么大的修改。总的来说这个过程是很珍贵的。

8.3 刘佳伟

本次大作业我们组投注了较多精力。最后，做出了一个具有一定程度实用性的，较为完整的分布式并行数据库。经过这次大作业的历练，我大大提高了复杂系统开发、调试的能力，同时提高了团队合作能力，特别是对松耦合的理解和应用，还从代码层面理解通信、远程调用和并发。最重要的一点，进一步打磨面对挫折时的心态提升了自己的团队合作能力。我们会永远记得明德地下机房通宵的夜晚和连吃五天的核桃派。