

大规模信息系统构建技术导论

分布式数据库系统设计报告

2023学年 春学期

|  |  |
| --- | --- |
| 学号 | 姓名 |
| 3210104116 | 郭一铭 |

2024年 5月 27日

目录

[大规模信息系统构建技术导论 1](#_Toc14548)

[分布式数据库系统设计报告 1](#_Toc27193)

[1 引言 4](#_Toc28493)

[1.1 系统目标 4](#_Toc2612)

[1.2 设计说明和任务分工 5](#_Toc6474)

[2 总体设计 5](#_Toc29208)

[2.1 总体架构设计 5](#_Toc9517)

[2.2 Client模块 6](#_Toc9311)

[2.2.1 架构设计 6](#_Toc10919)

[2.2.2 工作流程 8](#_Toc25773)

[2.2.3 个人主要工作 8](#_Toc1712)

[2.3 Master模块 8](#_Toc19539)

[2.3.1 架构设计 8](#_Toc25747)

[2.3.2 工作流程 11](#_Toc23498)

[2.3.3 个人主要工作 12](#_Toc1007)

[2.4 Region模块 12](#_Toc14612)

[2.4.1 架构设计 12](#_Toc19478)

[2.4.2 工作流程 13](#_Toc2605)

[2.4.3 个人主要工作流程 13](#_Toc32578)

[2.5 util文件 14](#_Toc5099)

[3 个人实现功能介绍 14](#_Toc13415)

[3.1 通信架构和通信协议 14](#_Toc18690)

[3.1.1 通信架构 14](#_Toc13491)

[3.1.2 通信协议 14](#_Toc4616)

[3.2 数据分布 15](#_Toc510)

[3.3 复杂sql语句的查询 15](#_Toc7849)

[4 系统测试 16](#_Toc13746)

[4.1 大规模测试 17](#_Toc20844)

[4.2 分布式查询 21](#_Toc8890)

[4.3 复杂查询 21](#_Toc19722)

[5 心得体会 24](#_Toc32006)

# 1 引言

本项目是《大规模信息系统构建技术导论》的课程项目，基于《数据库系统》课程的基本知识和本课程所学的大规模信息技术系统构建的相关知识进行开发，完成一个分布式大规模数据库系统。

本系统通过设计 Client、Master、Region模块，使用etcd作为集群管理工具，使用sqlite作为数据库，实现了一个在多台主机上共享数据资源访问的分布式数据库系统。

本系统使用go语言开发，使用GitHub进行版本管理和协作开发，由小组内的三名成员共同完成。

* 1. 系统目标

本次实验项目是一个分布式大规模数据库系统。

项目实现了以下具体功能：

1. 数据库查询：支持基本的 SQL 语句查询，包括数据库数据的插入、查询、修改、删除等数据库的基本功能。同时可以支持复杂的查询，如join。
2. 数据分布：将完整数据表存储在服务器上，分割成若干个数据子集。一个Region管理若干table,Master记录Region与其管理的table的对应关系。
3. 集群管理：使用etcd作为集群管理工具。etcd是采用了 raft 分布式一致性算法的分布式键值对存储系统（key-value store）
4. 分布式查询：Master维护缓存表以供查询对应table和index所在的region，存储所有table和index与Region的关系，Region处理增删改查操作。
5. 副本管理：采用主从复制策略，两个region为一组，其中一个region作为主region，另一个为备份region。
6. 负载均衡：Master 可以对其他 Region进行管理。Master 内存中存储了分布式数据库所有表的信息（名称）及它们的位置，以及所有用户创建的索引信息（名称）；数据表与索引的数据存储于 Region 服务器中。用户有建表需求时，Master 进程将选择存储的表数量最少的 Region 主服务器作为新表的存储地。当某一Region出现繁忙时，Master通过设计好的负载均衡算法将某些 Region重新分配到其他的 Region上。同时各Region可以实现数据传输，等待负载均衡之后修改 table的定位信息。
7. 容错容灾：主要处理各个服务器意外下线情况。通过主从服务器结构，来处理主服务器、副本服务器和Master服务器意外断联等情况，保证用户数据不丢失。

## 1.2 设计说明和任务分工

本项目由小组3位成员合作编写完成，采用 go程序设计语言，使用 etcd作为集群管理工具，支持在不同的操作系统与语言环境下跨平台使用。

我使用 IDEA作为主要开发工具，我的分工如下：

|  |  |  |
| --- | --- | --- |
| 学号 | 姓名 | 分工职责 |
| 3210104116 | 郭一铭 | 分布式查询，数据库查询以及复杂查询 |

# 2 总体设计

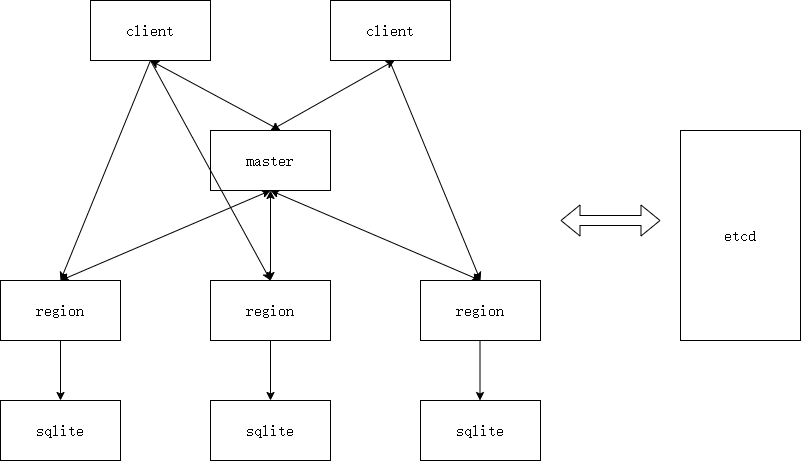
## 2.1 总体架构设计

系统将整个项目划分成三个模块，分别Client，Master 和 Region。

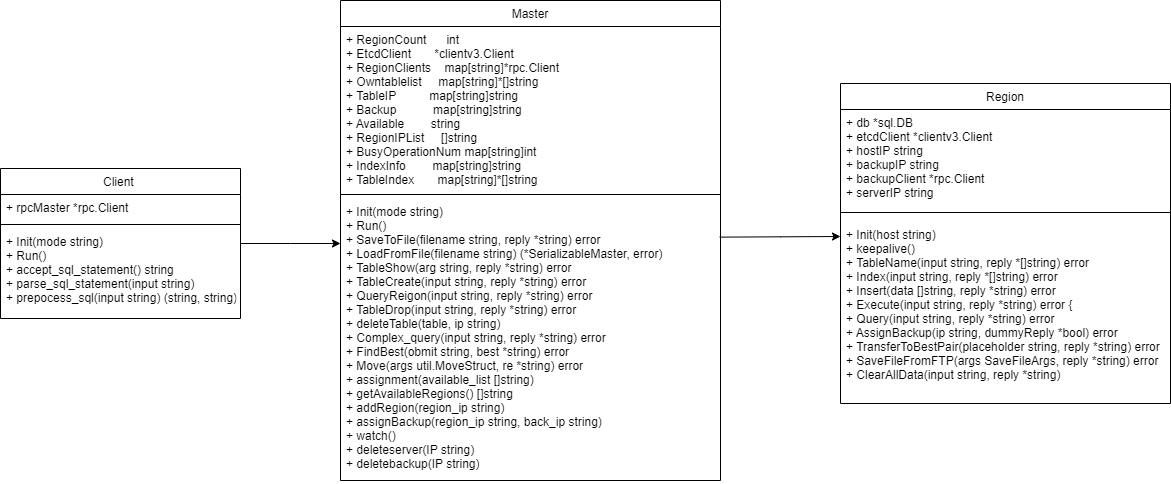
Client模块是用户与数据库交互的接口；Master模块管理Region模块，实现系统的集群管理、负载均衡、副本管理、容错容灾、数据分布、分布式查询等功能；Region负责数据库引擎的运行并且服从master的调度安排。

三者都可以在一定的通信框架下进行通信，同时通过etcd集群。

系统总体架构示意图如下图所示：



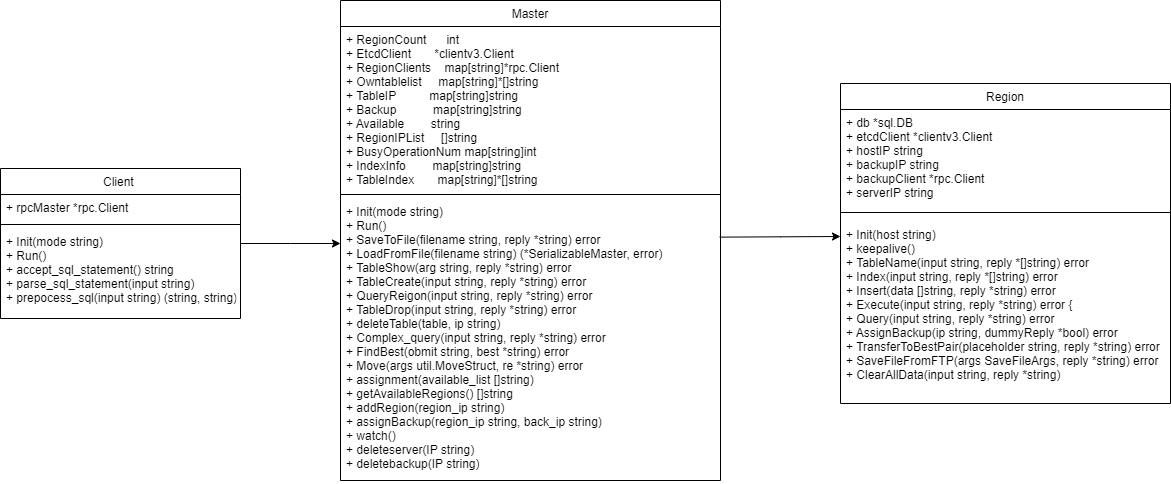
总体的类图如下：



## 2.2 Client模块

### 2.2.1 架构设计

Client的类图如下：



client向上负责与用户进行直接交互，向下与region和master进行连接。其主要的功能包括：预处理输入的sql语句，根据解析调用master或者region中接口进行执行语句。

* 预处理输入：

对用户输入的sql语句进行解析，获取需要操作的table和index。

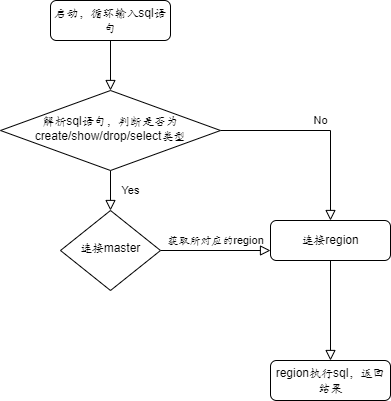
函数签名为func (client \*Client) parse\_sql\_statement(input string)

输入：sql语句

* 调用接口：

如果语句为create、show、drop、select调用master接口，由master调用region接口；其他情况调用master得到table所在的region，然后调用对应的region接口。

### 2.2.2 工作流程



### 2.2.3 个人主要工作

在Client中，我主要做的工作如下：

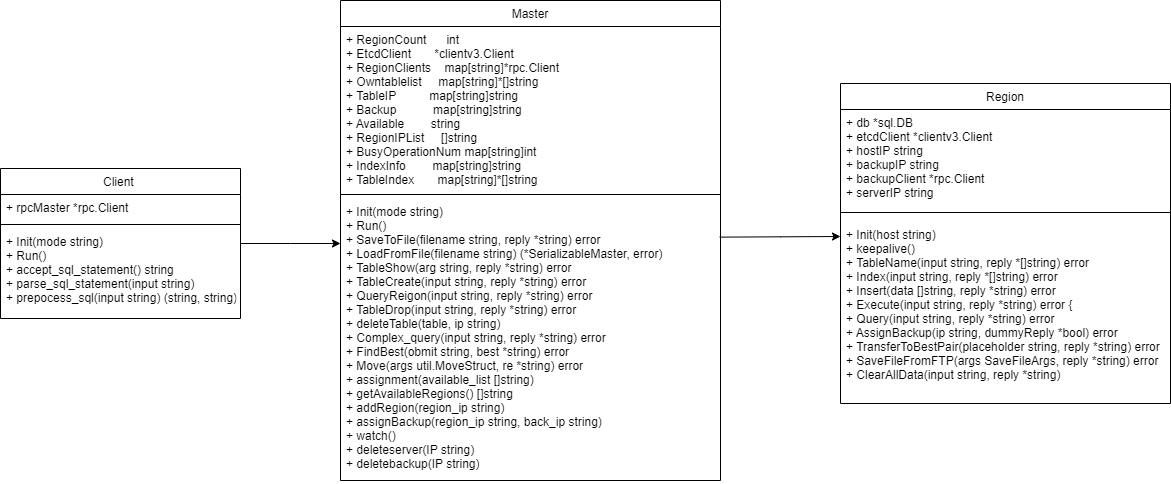
在create、show、drop类型中添加index相关的逻辑；

添加对select类型的判断和后续操作。

## 2.3 Master模块

### 2.3.1 架构设计

Master的设计如下图所示:



1. master的主要变量

type Master struct中定义了如下变量和缓存表：

* RegionCount int：记录了region主服务器的数量；
* EtcdClient \*clientv3.Client：为 master 进程与 master 进程所在的服务器（即 master 服务器）上的 etcd 进程通信的媒介；
* RegionClients map[string]\*rpc.Client：为 master 进程与所有在线的 region 服务器上 region 进程通信的媒介。
* Owntablelist map[string]\*[]string和TableIP map[string]string：记录了table与其所在region的ip 的对应关系；
* IndexInfo map[string]\*[]string和TableIndex map[string]string：记录了index与其所在table 的对应关系；
* BusyOperationNum map[string]int ：记录了region的忙碌状态；
* RegionIPList []string：记录了所有连接过的region的 ip；
* Backup map[string]string：记录了region和其备份region的对应关系；
* Available string：记录当前活跃的region的 ip。
* Tablecnt map[string]int：记录缓存中table理论上应该被复制的次数。

1. master的主要函数
2. Master.go中函数：

* func (master \*Master) Init(mode string) ：master系统的初始化，包括对数据库文件的读取，对变量和缓存表的初始化和数据载入；
* func (master \*Master) Run() ：初始化etcd集群并开启rpc服务进程；
* func (master \*Master) GetTableIP(table string, reply \*string) error：用于查询table和region的对应关系；
* func (master \*Master) InitTableIP() ：同步本地数据库文件中的table信息；
* func (master \*Master) InitIndex(table string) ：同步本地数据库文件中的index信息；

1. Func.go中函数：

* func (m \*Master) TableShow(arg string, reply \*string) error：直接查看owntablelsit查询所有region的table，以表格形式返回所有table以及其所属regionip；
* func (master \*Master) TableCreate(input string, reply \*string) error ：在数据库中创建table；
* func (master \*Master) TableDrop(input string, reply \*string) error：在数据库中删除table；
* func (master \*Master) deleteTable(table, ip string) ：删除缓存表中的table信息；
* func (master \*Master) check\_and\_reset\_Regions() error ：检查和重置region；
* func extractTable(s string) string ：提取table名称；
* func (master \*Master) IndexShow(arg string, reply \*string) error：查询所有table的index，以表格形式返回所有index以及其所属table；
* func (master \*Master) IndexCreate(input string, reply \*string) error ：在数据库中创建index；
* func (master \*Master) IndexDrop(input string, reply \*string) error：在数据库中删除index；
* func (master \*Master) deleteIndex(index string, table string) ：删除缓存表中的index相关信息；
* func (master \*Master) Move(args util.MoveStruct, re \*string) error：将table移动到region；
* func (master \*Master) Complex\_query\_master(input string, reply \*string) error：对复杂查询的处理；
* func (master \*Master) Copy(table string, ip string) error：将table复制到本地缓存数据库；

1. Etcd.go中函数：

* func (master \*Master) assignment(available\_list []string)：记录了etcd的相关代码，监视在util中的region\_ip的etcd变化；
* func (master \*Master) getAvailableRegions()[]string：获取当前待用region；
* func (master \*Master) addRegion (region\_ip string) ：用于初始化和后续加入region的连接；
* func (master \*Master) watch() ：进行对 etcd 键值对存储系统的各种更新（键值对的增添与删除）的监听，并进行相应的操作：
* func (master \*Master) deleteserver(IP string) ：处理某主服务器下线。如果有待用region，启动副本region作为新的主服务器，待用节点作为其副本服务器；否则把副本region中的内容都转存到某个region对中，将副本region变成待用region。
* func (master \*Master) deletebackup(IP string) ：处理某副本服务器下线。如果有待用region，将其设为副本region；否则把主服务器region中的内容都转存到其他region对中，将主region转换为待用region。

1. db.go中函数：

与region模块的同名函数功能相似，故不在此赘述。

### 2.3.2 工作流程

该项目中 Master 层的功能由四个子文件实现：

1. master.go文件。

存储了所有节点和表格的信息。定义了缓存表并进行了系统的初始化，同步了数据库中的信息。

1. etcd.go文件。

建立与 etcd的连接，实现数据集群管理，对节点的状态改变进行监听，处理新增节点、节点出错等情况，基于此实现了容错容灾等功能。

1. func.go文件。

建立与 Client 和Region的通信，通过 Client 的请求查找相关 Region 信息，并将结果返回至 Client。处理了增加/删除table/index请求时对master缓存信息的更新。

1. db.go 文件。

存放master本地缓存数据库（用于复杂查询）所需要的函数。

### 2.3.3 个人主要工作

添加了index相关的函数。

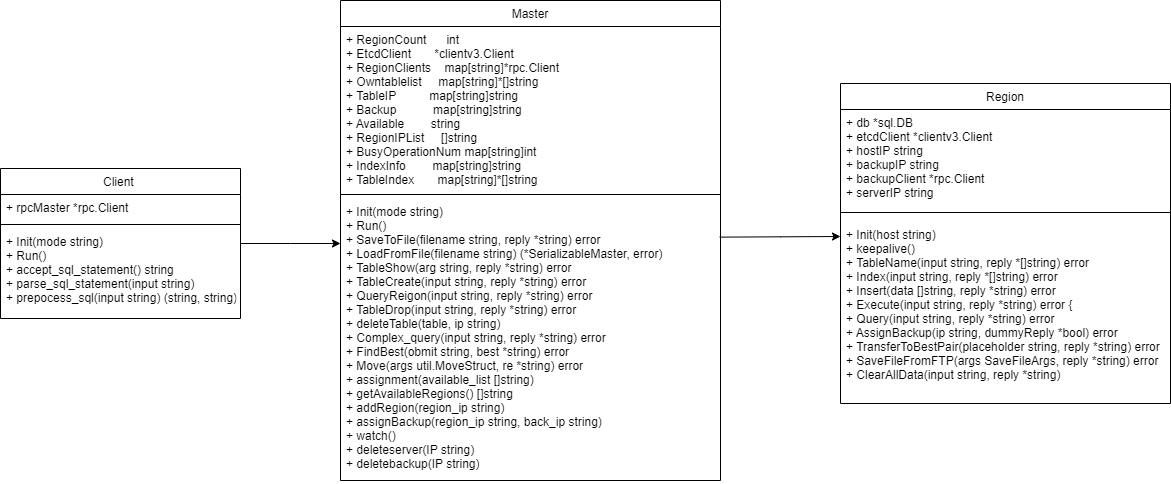
编写了移动table到region操作相关的函数，如move。

完成了复杂查询的编写以及将table复制到master本地数据库中相关的工作。

## 2.4 Region模块

### 2.4.1 架构设计

Region的类图如下：



相关的主要函数如下：

1. Region.go中函数：

* func (region \*Region) Init()：实现region的初始化，包括连接数据库文件和注册rpc服务；
* func (region \*Region) keepalive()：在etcd中保持活跃；
* func (region \*Region)foundhostIP()string：获取本机IP地址。

1. Func.go中函数：

* func (region \*Region) TableName(input string, reply \*[]string) error：返回当前region管理的table的名字；
* func (region \*Region) Index(input string, reply \*[]string) error：返回当前region管理的index的名字；
* func (region \*Region) Execute(input string, reply \*string) error：处理非查询类的sql语句；
* func (region \*Region) Query(input string, reply \*string) error：处理查询类的sql语句；
* func (region \*Region) AssignBackup(ip string, dummyReply \*bool) error ：给主region分配副本region。
* func (region \*Region) Insert(data []string, reply \*string) error：向table中批量插入sql语句。
* func (region \*Region) Get(input string, reply \*[]string) error：sql查询后以字符串组形式返回获取的所有数据，每个字符串代表一个返回值。

### 2.4.2 工作流程

本项目中Region模块主要实现了以下功能：

1. 加入etcd的集群管理
2. 处理Client端和master端发送的sql语句对数据库进行增删改查

### 2.4.3 个人主要工作流程

完成了index相关的函数，如index。

完成了table迁移相关的函数，如insert、get。

## 2.5 util文件

该模块主要包含的函数如下：

* func FindElement(pSlice \*[]string, str string) int ：查找元素；
* func TimeoutRPC(call \*rpc.Call, ms int) (\*rpc.Call, error)：反馈rpc调用是否超时；
* func DeleteFromSlice(pSlice \*[]string, str string) bool ：删除字符串组中字符串；
* func AddToSlice(ptr \*[]string, newString string) & func AddToSliceIndex(ptr \*[]string, newString string) ：在字符串组中加入字符串；
* func CleanDir(localDir string) ：删除指定目录中的所有文件。

# 3 个人实现功能介绍

## 3.1 通信架构和通信协议

### 3.1.1 通信架构

远程过程调用（Remote Procedure Call，RPC）是一个计算机通信协议，该协议允许运行于一台计算机的程序调用另一台计算机的子程序。

在架构中，client与Master进行多对一的连接，Region与Master进行多对一的连接，而client与Region的连接不定（正常操作下随着SQL语句的执行应该会逐步变成一对多）。

建立rpcClient时，只需要指定IP地址和端口号即可通过DialHttp函数获取到rpcClient，随后便可使用rpcClient对region/master的函数进行调用。

### 3.1.2 通信协议

由2.5可以看到，系统封装了非阻塞的RPC调用函数TimeoutRPC，master/func.go和region/func.go中的大部分函数采用了基于传入两个指针参数，一个作为封装好的函数固有参数，另一个作为返回值的策略。

## 3.2 数据分布

出于最小化系统复杂度的考虑，我们选择将完整的数据表存储在服务器中，而不是将数据表的所有记录分散存储在不同服务器上。

master 进程的内存中存储了分布式数据库所有表的信息（名称）及它们与region的关系，以及所有索引信息（名称）及它们与table的关系；数据表与索引的数据存储于 region 服务器中。

Master中的存储信息如下：

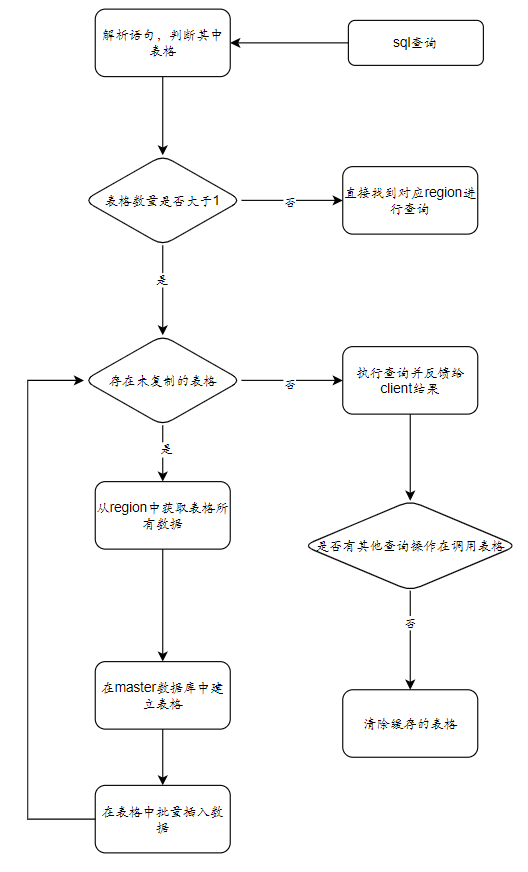
type Master struct {  
 RegionCount      int  
 EtcdClient       \*clientv3.Client  
 db               \*sql.DB  
 RegionClients    map[string]\*rpc.Client  
 Owntablelist     map[string]\*[]string // ip -> tables  
 TableIP          map[string]string    // table -> ip  
 Backup           map[string]string    // region server ip -> Backup server ip  
 Available        string               // available regions  
 RegionIPList     []string  
 BusyOperationNum map[string]int       // operations for each region in 1 minute, > BUSY\_THRESHOLD deemed as busy  
 IndexInfo        map[string]string    // index->table  
 TableIndex       map[string]\*[]string // table->indexs  
 TableCnt         map[string]int       // table->count 存储本地数据库中table应该copy了多少次  
}

## 3.3 复杂sql语句的查询

本系统实现了复杂sql语句的查询。

当查询中涉及不止一个table的时候，将所有的table缓存到master中，在master中进行查询后，输出结果并清空缓存。

流程图如下：

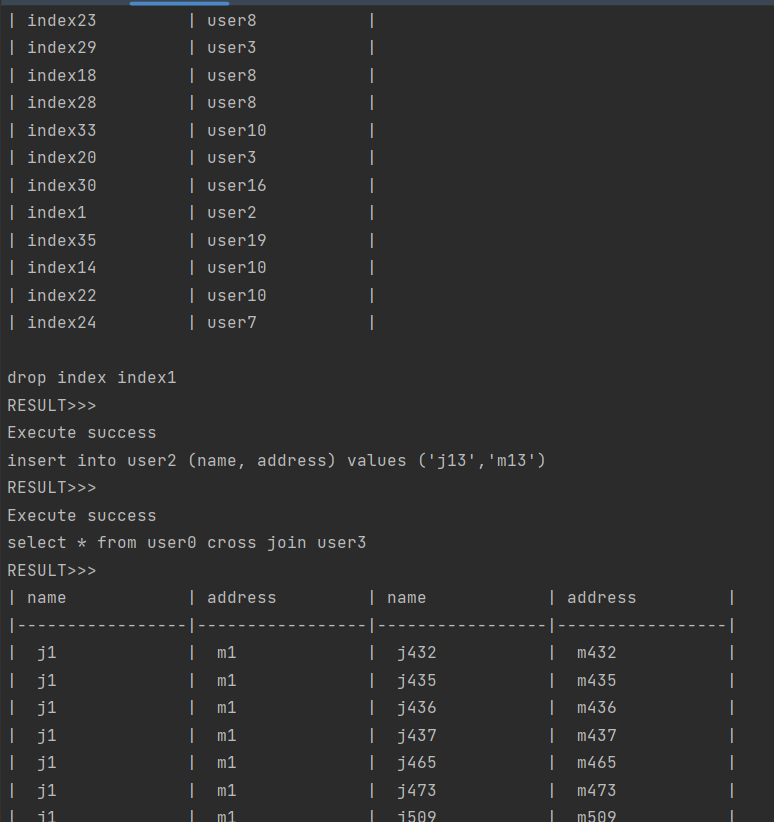


# 4 系统测试

# 4.1 大规模测试

测试100000条代码成功。

部分截图如下：



# 

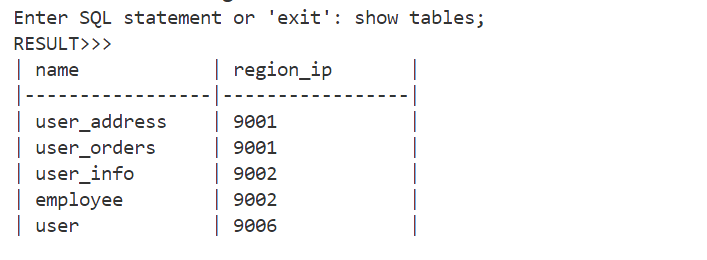
测试用例构造代码如下：

*#include<bits/stdc++.h>*  
using namespace std;  
int size;  
int cnt;  
int index;  
void s1(){  
 string s="create table user";  
 s=s+to\_string(size)+"(name TEXT,address TEXT);";  
 size++;  
 cout<<s<<endl;  
}  
void s2(){  
 string s="show tables;";  
 cout<<s<<endl;  
}  
void s3(){  
 string s="drop table user";  
 size--;  
 s=s+to\_string(size)+";";  
 cout<<s<<endl;  
}  
void s4(){  
 string s="insert into user";  
 int x=rand()%size;  
 s=s+to\_string(x)+" (name, address) VALUES ('J";  
 cnt++;  
 s=s+to\_string(cnt);  
 s=s+"','M"+to\_string(cnt)+"');";  
 cout<<s<<endl;  
}  
void s5(){  
 string s="create index index";  
 index++;  
 s=s+to\_string(index)+" on user";  
 int x=rand()%size;  
 s=s+to\_string(x)+"(name);";  
 cout<<s<<endl;  
}  
void s6(){  
 string s="show indexes;";  
 cout<<s<<endl;  
}  
void s7(){  
 string s="drop index index";  
 s=s+to\_string(index)+";";  
 index--;  
 cout<<s<<endl;  
}  
void s8(){  
 string s="select name from user";  
 int x=rand()%size;  
 s=s+to\_string(x)+";";  
 cout<<s<<endl;  
}  
  
void s9(){  
 string s="select \* from user";  
 int x=rand()%size;  
 s=s+to\_string(x)+" cross join user";  
 x=rand()%size;  
 s=s+to\_string(x)+";";  
 cout<<s<<endl;  
}  
int main(){  
 freopen("testbig.txt","w",stdout);  
 srand(time(NULL));  
 int T=100000;   
 size=1;  
 cout<<"create table user0(name TEXT,address TEXT);"<<endl;  
 for(int i=0;i<T;i++){  
  if(size==0) {  
   size=1;  
   cout<<"create table user0(name TEXT,address TEXT);"<<endl;  
  }  
  int k=rand()%9;  
  switch (k){  
   case 0:  
    s1();  
    break;  
   case 1:  
    s2();  
    break;  
   case 2:  
    s3();  
    break;  
   case 3:  
    s4();  
    break;  
   case 4:  
    s5();  
    break;  
   case 5:  
    s6();  
    break;  
   case 6:  
    if(index>0)s7();  
    break;  
   case 7:  
    s8();  
    break;  
   case 8:  
    if(size>2)s9();  
    break;  
   }  
 }  
 return 0;  
}

# 4.2 分布式查询

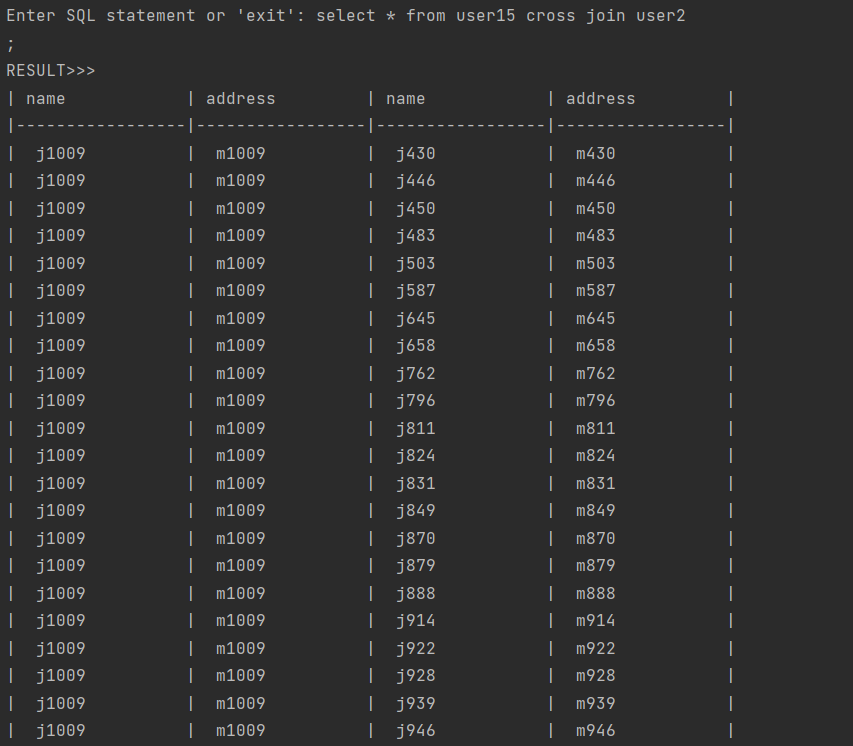
master中的TableIP变量和Owntablelist变量记录了各个表格存在的节点；；Backup变量存储了每个主服务器对应的副本服务器。具体增删改查操作在节点服务器中进行。

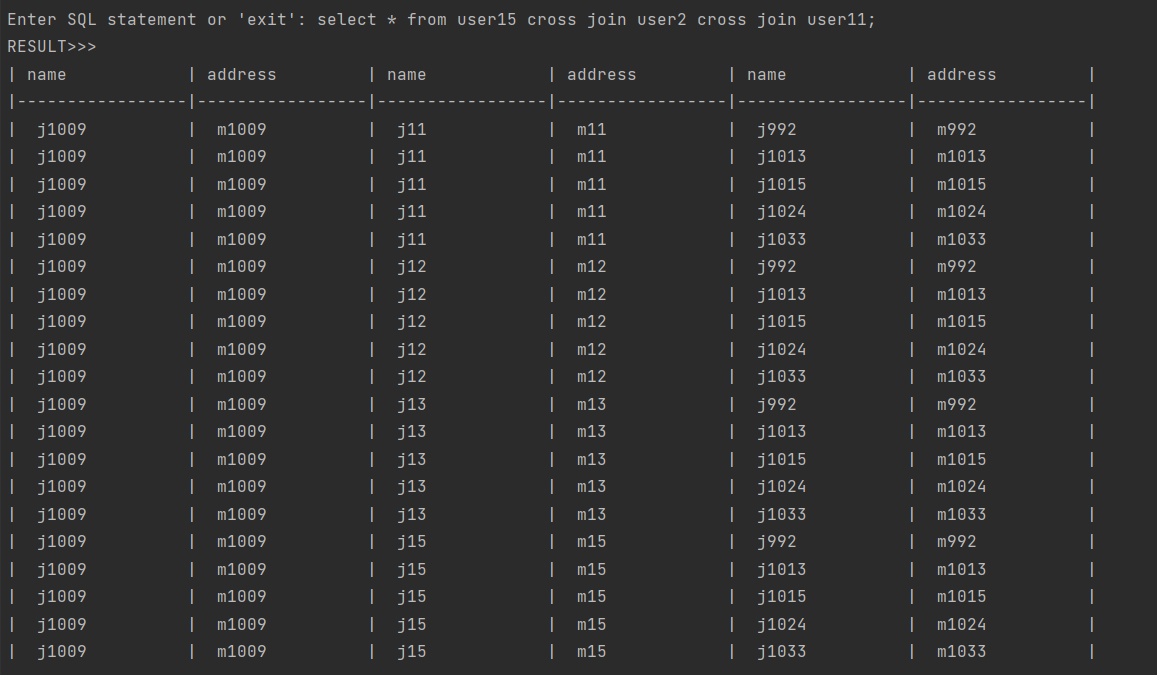
输入show tables指令，返回各个表和存在的节点位置。可见各个表存在不同的服务器上。



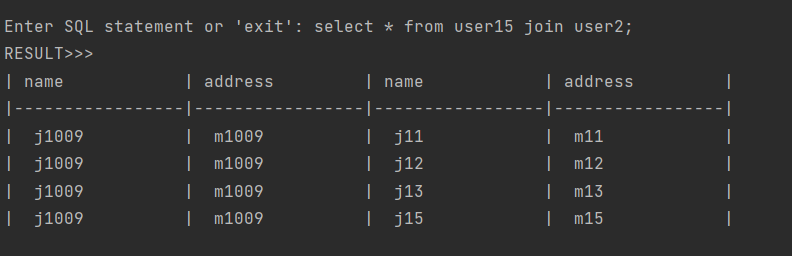
# 4.3 复杂查询

下图展示了cross join，能够输出结果。

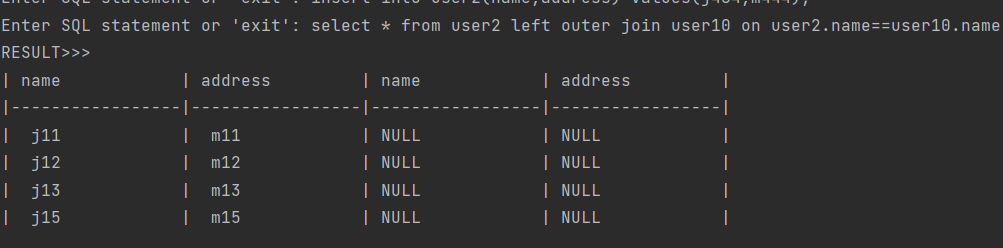


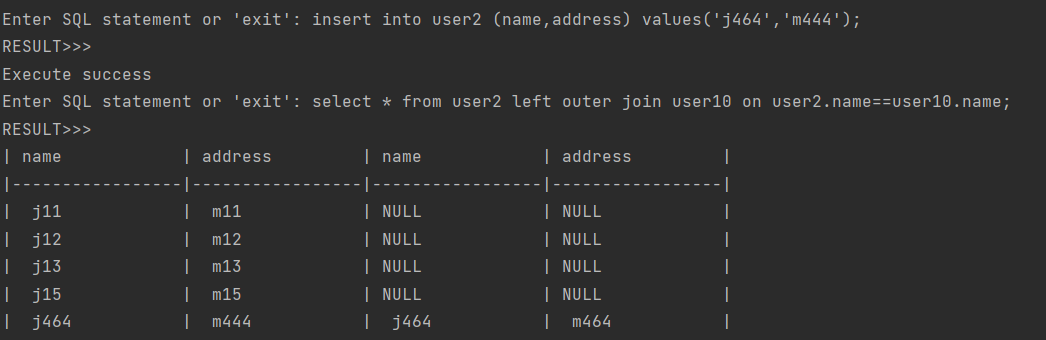


（inner）join：



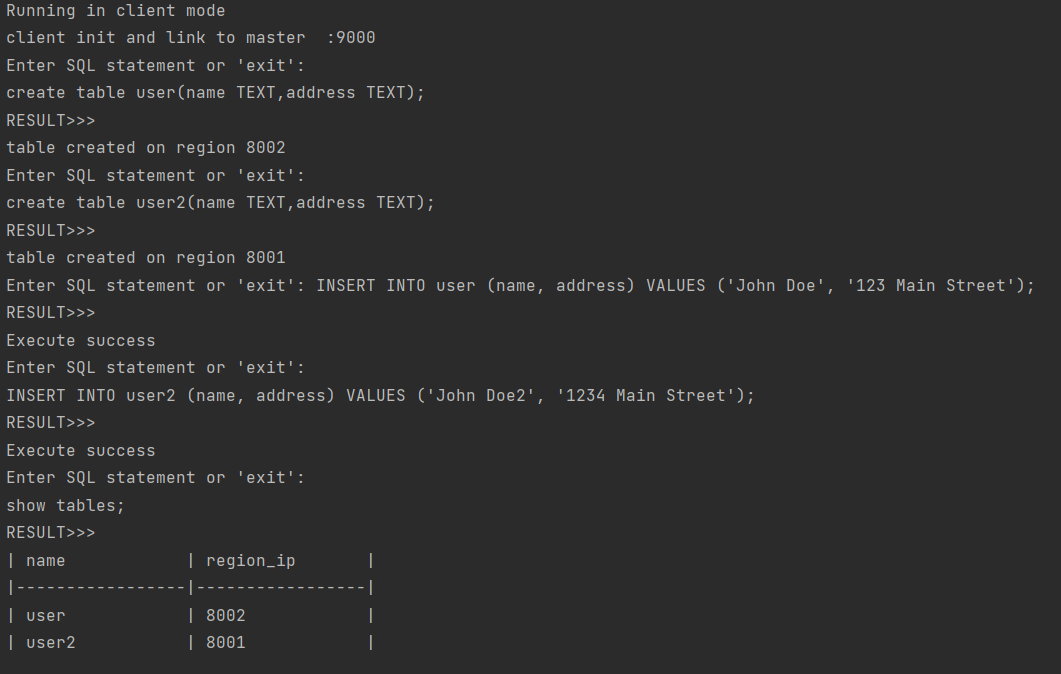
Left out join：

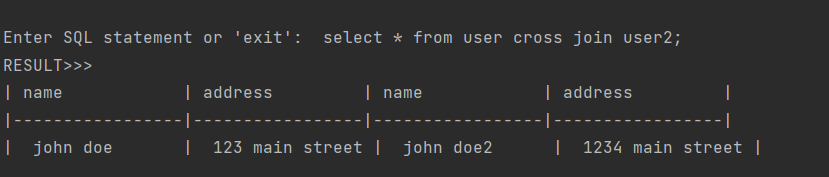




支持对数据库的简单操作。

以下是来自不同region的table进行join查询的过程：





# 5 心得体会

这次项目中我收获颇丰，获得了很多宝贵的经验。

本次我们小组搭建了一个分布式数据库系统，从理论知识的学习、系统架构的搭建到代码的实现、测试、总结和报告，我们三名成员紧密合作，共同完成了整个项目。

这个项目让我有机会将课堂上学习到的理论知识，包括数据库的知识、远程通信的知识、分布式大规模数据库的知识，运用到实际的开发中。在开发的过程中我也不断的深化对理论知识的理解。我也感受到了实际开发和理论的巨大差异。在系统架构过程中，我们一度以为这个项目会很容易实现。但实际开发过程中发现总是会出现各种各样意料不到的问题，导致我们的开发过程比预想中长。

同时我也学习了新的知识，比如我第一次使用go语言进行开发，也是第一次接触sqlite数据库。在开发的过程中，我感受到了go语言和我惯用语言的差别，发现了它便捷和不便之处；同样，使用sqlite数据库后，它的轻量、不需要服务器和配置带给我很多方便，同时它的存储方式和不支持数据拆分合并的特性也让我们的实现走了一些弯路。总体来说，这些新的知识为我以后项目的技术选型拓展了一条新的道路。

当然，我的代码能力也得到了锻炼。尤其是在实现复杂查询的时候，我不断的遇到问题，也不断的寻找解决问题的方案。一开始设想的是构建语法树来解决问题，后来发现这个做法过于复杂，我们的项目时间与之不匹配，于是选择简单的迁移数据表到一个数据库中。但是在实现过程中发现sqlite不支持数据表分割，于是选择从旧region中取出所有数据再插入到一个region中。在这个过程中，我想办法提高效率，如选择批量插入；也遇到新的问题，比如不知道表的数据类型，为此进行了搜索和学习，更深入的了解了sqlite数据库的运作原理。最后想到多个client同时查询的问题，我决定将table复制到master服务器中来避免冲突。在这个过程中我的解决问题的能力得到了提升。

此外，我的团队协作能力也得到了锻炼。三个人一起实现项目会出现很多问题。我们制定了详细的计划，也明确了彼此的分工，通过尽量给每个人分配不冲突的部分、定期汇报和沟通等等方式来减少问题的出现。当然不可避免的，过程中遇到一些较简单的问题，比如个人系统环境不同、在github上pull代码时merge出错等等；也有比较复杂的问题，比如组员要调用我写的一个函数，虽然函数的逻辑不复杂，但是因为对里面的细节理解出现偏差，导致我们调了好几个小时的bug。

总体来说，这次项目对我来说是一次很宝贵的经历。我不仅学习、实践了很多知识，也锻炼了自己的能力。这次经历也为我以后的职业发展打下了坚实的基础。