分布与并行数据库
实验报告

gDB 小组

2018104077	柴茗珂	Select 语句解析、查询计划生成与优化、查询执行
2018104183	沈运龙	除 select 以外语句的执行以及元数据的存取
2018000835	洪殷昊	框架搭建，元数据存储，RPC 通讯

目录

1 功能实现 3

2 总体设计 4

3 模块详细设计 5

3.1 SELECT 执行模块 5

3.1.1 SELECT 语句解析与优化 5

3.1.2 查询计划与优化 6

3.1.3 查询执行 7

3.2 DataManger 模块 7

3.2.1 SQL 语句的解析 8

3.2.2 元数据的设计 9

3.2.3 SQL 语句的执行 10

3.3 其它模块 11

4 个人总结 12

4.1 柴茗珂 12

4.2 沈运龙 12

4.3 洪殷昊 12

1 功能实现

功能	关键字
定义站点	ADD
创建表	CREATE TABLE
定义分片 创建分片	FRAGMENT ALLOCATE
批量导入	LOAD
插入记录	INSERT
删除记录	DELETE
查询	SELECT

表 1 功能概览

2 总体设计

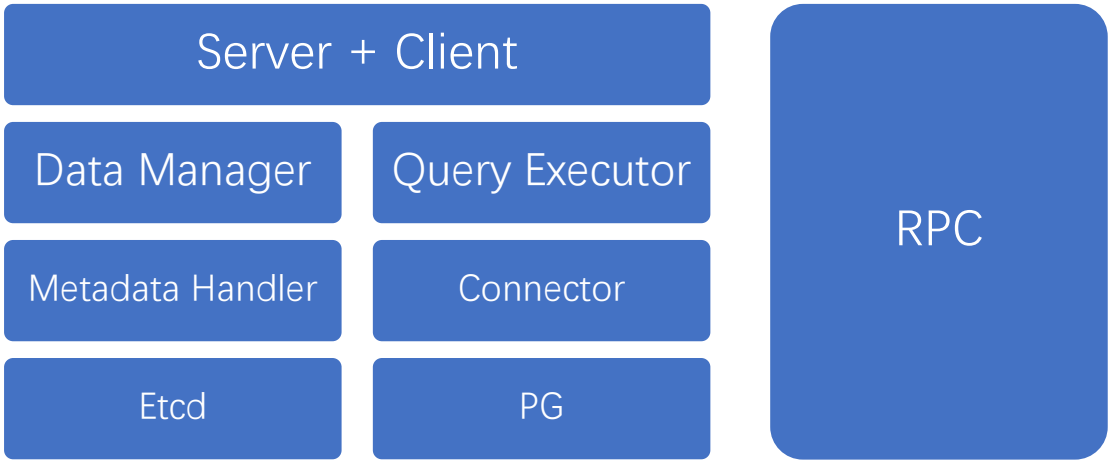

图 2 系统架构

系统基于 Go 语言编写，底层数据存储使用了 PostgreSQL，元数据存储使用了 etcd，通讯基于 gRPC。

如图 2 所示，gDB 是一个客户端和服务端不分离的设计，顶层直接负责和用户进行交互。顶层判断用户的输入类型后，会传入执行层。Query Executor 和 Data Manager 是执行层的两个模块，分别用于执行 Select 和其它类型的数据库语句。执行层的下方是数据访问层，该层屏蔽了 etcd 和 PG 的具体细节，提供了存储的 API。RPC 是执行层用于和其它实例通讯的模块。

3 模块详细设计

3.1 SELECT 执行模块

3.1.1 SELECT 语句解析与优化

1.语句解析：利用正则表达式对 SQL 语句进行解析，对于每一个 SQL 语句都有以下几种结构：

（1）Table：每一张表都会有自己的分片信息，包括分片的大小、模式（水平或垂直）等。

```
type Table struct {
    tab  []Distributed_Table
    size int
}
```

```
type Distributed_Table struct {
    tab     string
    segment []string
    size    int
    HV      int
}
```

（2）Join：每一个连接操作包含连接表的表名、连接属性。这里还指明了连接策略，即哪些分片之间要执行 join。

```
type Join struct {
    jo   []Join_each
    size int
}
```

```
type Join_each struct {
    tab1      string
    tab2      string
    join_attr string
    Strategy  Join_Stra
}
```

（3）Select：每一个选择操作包含选择的属性、操作符和值。

```
type Select struct {
    selT []Select_eachT
    size int
}
```

```
type Select_Each struct {
    attribute string
    operator  string
    value     string
}
```

（4）Projection：存了具体投影的属性，刚开始投影数组只有一个值，随着投影下推，对每一个表都有投影。

```
type Projection struct {
    pro  []string
    size int
}
```

2.查询树的生成与优化：采用左深树的策略，从根节点到叶子节点依次是全局投影、连接、表的投影、选择、表。则得到的图类似于下图所示：

```
-----------------step1: Create SQL parse tree-----------------
..............................................................................
--------------------------------P : course.course_name, teacher.teacher_name, student.student_name, exam.mark
----------------J : teacher course : teacher_id                              #!
--------J : course exam : course_id              P : teacher_name, teacher_id              #!              #!
----J : exam student : student_id        P : course_name, course_id, teacher_id        S : teacher.teacher_id>2014000,        #!        #!        #!        #!        #!
--P : mark, student_id, course_id  P : student_name, student_id  S : course.credit_hour<=2,  #!  T : teacher  #!  #!  #!  #!  #!  #!  #!  #!  #!  #!  #!
-S : exam.mark<60, #!S : student.student_id>1040000;, #!T : course#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!
T : exam#!#!#!T : student#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!
```

为了减少站点之间数据的传输，我们尽可能让每个分片先进行选择和投影操作，即：投影下推和选择下移。值得注意的是，投影条件不光是 SQL 解析出来的投影条件，还要考虑 Join 的属性和选择操作里的属性。

查询树的优化主要体现在表的顺序的重排序和分片的剪枝。

（1） 表的重排序：由于我们的策略所限，所有表都要对最左边的表发数据，所以我们需要保证最左边的表是元组数目最多的。所以我们利用 GDD 的元组数信息，对 join 顺序进行了重排序，具体是保证所有 join 都可以正常执行的情况下，表从左到右的顺序按照表元组数从多到少排序。

（2） 分片剪枝：分片包括垂直分片和水平分片。水平分片主要根据对应表的选择操作进行剪枝；垂直分片的剪枝需要访问所有投影和 join 的属性，如果该垂直分片没有这些属性中的任何一个就剪掉。此外，若该分片没有数据也可以剪枝。

3.1.2 查询计划与优化

查询计划采用的策略是层层递进式。每次取两个表汇总（join 或者 union）成中间表，当访问到下一个表时，下一个表和上一步形成的中间表再汇总。如下图所示，Student1、Student2 都发到 Exam 所在站点，即 Exam1 和 Exam3 所在站点。Exam1、Student1、Student2 通过一系列 join 或 union 形成 TEMP0_0。同理，Exam3、Student1、Student2 汇总成 TEMP1_2。第一轮 join 结束，此时开始第二轮，Course1 和 Course2 发到 Exam 所在站点，Course 分片在 Exam 两个站点分别和上一步形成的 Temp 表 Join，最后形成的两个表再做 Union 形成最后结果。这里每一步具体是 Union 还是 Join 由分片的模式决定。如果水平划分则 Union，否则 Join。值得注意的是，这里的 Union 不去重，即为 Union all。

```
-----------------step1: Create SQL parse tree-----------------
..............................................................................
--------------------------------P : exam.mark, student.student_id, student.student_name, course.course_name
----------------J : exam course : course_id                              #!
--------J : exam student : student_id              P : course_name, course_id              #!              #!
----P : mark, student_id, course_id        P : student_id, student_name        S : course.location<>'CB-3',        #!        #!        #!        #!        #!
--S :    #!  S : student.age>26,   #!  T : course  #!  #!  #!  #!  #!  #!  #!  #!  #!  #!  #!  #!  #!
-T : exam#!#!#!T : student#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!#!
..............................................................................
-----------------step2: Fragments after H/V pruning-----------------
水平： [exam.1.0 exam.3.2]
水平： 水平剪枝
time="2019-01-11T15:07:16+08:00" level=debug msg="原student.1.0"
time="2019-01-11T15:07:16+08:00" level=debug msg="原student.2.1"
水平剪枝
time="2019-01-11T15:07:16+08:00" level=debug msg="原student.3.2"
水平剪枝
[student.1.0 student.2.1 student.3.2]
垂直： [course.1.0 course.2.1]
-----------------step3: Join strategy-----------------
剪枝：(exam.1.0t, student_id, student.3.2t)
剪枝：(exam.3.2t, student_id, student.1.0t)
UU: {[(exam.1.0t, student_id, student.1.0t)U(exam.1.0t, student_id, student.2.1t) (exam.3.2t, student_id, student.2.1t)U(exam.3.2t, student_id, student.3.2t)] [TEMP0_0 TEMP1_2] 2 0 }
UJ: {[(TEMP0_0, course_id, course.1.0t), course_id, (TEMP0_0, course_id, course.2.1t) (TEMP1_2, course_id, course.1.0t), course_id, (TEMP1_2, course_id, course.2.1t)] [TEMP2_0 TEMP3_2] 2 0 }
```

可以看出来我们的策略需要大量中间表。为了减少磁盘 IO，我们决定使用视图。所有中间表都以视图的形式存取。若该中间表需要向其他站点传输时，也和普通的 table 一样，先 select * from Temp，再把结果以字符串的形式传到指定站点。但是删除视图的时候要考虑视图之间的依赖，可以以栈的方式删除。

此外，有些情况的 Join 是可以剪枝的。对于两个表的相同 Join 属性，访问 GDD 得到该属性在这个表的的取值范围，如果取值范围无交集则可剪枝。

3.1.3 查询执行

虽然各个站点是并行的，但对于具体某一个站点，它所执行的策略是串行的，所以我们这里使用线程。一开始就为每一个站点开启线程而且将执行计划写入 plan，站点就可以访问它对应的 plan 得到。Plan 里面主要分以下几类计划：

（1）[EXIST]

若要在本地执行 join 或 union 语句，首先要保证对应的表存在，所以这里有[EXIST]操作。以前我们直接使用数据库是否有这个表名来判断该表是否存在。但就有可能存在其他站点向该站点发表的信息，该站点执行 create 和 insert 操作。在 insert 中途，甚至是 create 一结束，程序就检测到该表存在，而实际上这个表还没有稳定。所以我们这里设置了一个全局变量，如果 insert 结束这个值赋为 1，检测到全局变量有更新才可说明插入了一个新表。当然这个更新次数由 Exist plan 中具体有几个是非本站点直接形成的表数决定。

（2）本地执行

如 Create View 语句就可在本地执行无需消息的传送。就直接调用 pg 的 Excute 接口函数即可。

（3）[Send]

当向其他站点发送某张表时，首先用 select *语句访问该表，得到的结果封装成一个字符串，再加上表名、属性名一个发送到对应站点。再在对应站点 Create 和 insert。这块有些遗憾，由于封装时间过长，我们想如果改成流处理会更快。此外，为提高 insert 速度，我们这里使用了 prepare。

（4）[Final]

执行最后一个 plan 时只需 select *，无需发送到其他站点。我们这里会打印行数与前五行。

3.2 DataManger 模块

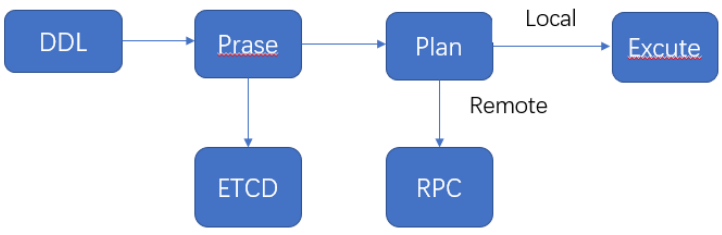

图 3-1-1　dataManager 的基本设计思路

3.2.1 SQL 语句的解析

词法解析器用于将原始的 SQL 语句字符串解析成一个一个 Token 供 yacc 解析，在我们的代码中，由 Lexer.go 与 scanner.go 共同构成词法解析器，

Parser.y 作为语法解析器的 yacc 文件，利用 goyacc -l -o parser.go parser.y 命令生成可执行的 parser.go 文件作为语法解析器

```
type lex struct {
    input     *Scanner
    statement Statement
    err       error
}
```

图 3-1-2　lexer.go 中的 lex 结构体

Lex 结构体中的 input 存储输入的 SQL 输入字符流, statement 存储 SQL 解析出来的最终结果，err 存储返回的错误。

在我们写好一个 yacc 规则后，生成的 go 文件。其中包含两个重要的对象

```
type yyLexer interface {
        Lex(lval *yySymType) int
        Error(e string)
}
```

```
type yyParser interface {
        Parse(yyLex) int
        Lookahead() int
}
```

图 3-1-3　parser.go 中的 yylexer 和 yyParser

yyparser 是 yacc 自动实现的，不需要我们操作，parser 是入口函数，会不停的调用 Lex 函数来获取 TOKEN 进行文法分析。我们至需要自己实现 yyLexer 接口:

```
//  implement the lex and error for goyacc
//  the lex stop when return 0
func (l *lex) Lex(lval *yySymType) int {
    // this is like a state-transfer machin

func (l *lex) Error(s string) {
```

图 3-1-4　lex 中用于实现 yyparser 接口的函数

在 Lex 函数中，我们调用 scan 函数来获取一个字段的 type 也就是 TOKEN 和该字符串 val。虽然返回值只有 token，但是我们将 val 的值根据其不同类型也传给了 yacc。需要注意两点，（1）如果读取到文件末尾，需要返回 0，则 parser 就会知道已完结。（2）文法分析中不包含空格，所以读取到空格时忽略知道读取到非空格返回 TOKEN。Scanner 的实现也是一种有限状态机。

在 yacc 文件中，我们利用预先写好的语法规则进行规约，到最后将结果通过 setStmtResult 函数将最终的结果返回给 Lex

```
// 这个函数用来将最终获得的statement赋值给lex
func setStmtResult(l yyLexer, v Statement) {
  l.(*lex).statement = v
}
```

图 3-1-4　setStmtResult 函数

最后我们将整个语句的解析过程封装在函数 Parse 中:

```
// the func is the whole point we want
func Parse(r io.Reader) (Statement, error) {
    l := newLex(r)
    _ = yyParse(l)
    return l.statement, l.err
}
```

图 3-1-5　Parse 函数

3.2.2 元数据的设计

（1）元数据的结构设计

我们使用的是 Etcd 存储表的元信息，Etcd 是一个分布式的键值对数据库，所有写入的元数据实际上就是 key-value 对：

Key	Value
/gdd_table	表名,如 student,course
/gdd_table/student	属性列表，如 id,name,email
/gdd_table/student/id	属性类型，如 int
/gdd_key/student	主键，如 id
/part_schema/student	分片类型，H 表示水平，V 表示垂直
/part_schema/student/H	水平分片所依赖的属性,如 id(垂直不写)
/part_info/student	分片数量
/part_info/student/student.1	分片的条件，如 id>20 或者 id,name
/part_site/student.1	分片所在站点，如 st0

表 3-2-1　元数据结构

（2）元数据的存取

元数据是在每条 SQL 的执行时写入的，对于一条 SQL 语句我们解析出一个包含了语句所有信息的结构体,然后将对应的元数据通过 storage 的 putValue 函数写入 Etcd:

```
// just write gdd
func (d DDL) ExecStmt() error {
    if strings.ToLower(d.Action) == "create" {
        err := writeGddTable(gddGet.Client, &d)
        if err != nil {
            fmt.Println( a: "Write etcd failed!")
            return err
        }
        fmt.Println( a: "Create语句执行成功.")
        return nil
    }
    return nil
}
```

图 3-2-1　create 语句执行时写入元数据

元数据的获取函数是在 gddGet.go 中，主要是通过 storage 包的 GetValue 获

取 Etcd 中的值:

```
// Given table name return the all fragment such as Student.1,Student.2
func GetFragment(tableName string) ([]string, error) {
    numBytes, err := storage.GetValue(Client, "/part_info/"+tableName)
    if err != nil {
        return nil, err
    }
    var s []string
    numStr := byteToString(numBytes)
    num, _ := strconv.Atoi(numStr)
    for i := 1; i <= num; i++ {
        s = append(s, tableName+"."+strconv.Itoa(i))
    }
    return s, nil
}
```

图 3-2-2　分片信息的元数据获取函数

其中因为在 Etcd 中读取出来的字节流，所以我们最后将其转换为字符串返回。

3.2.3 SQL 语句的执行

在 statement 文件中，我们定义了一个 Statement 接口:

```
type Statement interface {
    PrintStmt()
    ExecStmt() error
}
```

图 3-2-3　statement 接口

Statement 接口也是 Parse 函数解析 SQL 语句的结果，对于 Create, fragment, allocate, insert, delete , load 语句有分别对应的结构体实现了 Statement 的接口。PrintStmt 函数主要是前期用于解析结果查看，ExecStmt 是语句的执行函数。

3.2.3.1 Create 语句的执行

执行 Create 语句时，如图 3-2-1 所示我们只是将全局表的信息写入到 Etcd 中，不做其他操作。

3.2.3.2 Fragment 语句的执行

Fragment 语句执行也只是将元数据写入到 Etcd 而不做其他操作。

3.2.3.3 Allocate 语句的执行

Allocate 语句中我们指定哪个分片分配给哪个站点，此时不仅要在 etcd 中写入元信息，同时需要给相应的站点创建一个代表分片的表，通过 allocate 结构体与 Etcd 中的元信息，封装相应的 SQL 语句的 String，调用 grpc 发送给目标站点执行，目标站点的执行函数在 server.protoImp 中。

3.2.3.4 Insert 语句的执行

对于 Insert 语句我们分为如下两种执行策略：

（1） 目标表是水平划分

此时我们将 insert 的 value 通过 exprParser 代入到分片的条件中，然后将语句发送到对于的站点执行。其中 expreParser 也是一个 lex&yacc 解析器，如果

满足分片条件，返回 ture，反之返回 flase。

（2）目标表是垂直划分

基于 gdd 的元信息，我们将一条 SQL 语句转换成对于不同站点的不同 SQL 语句执行

3.2.3.5 delete 语句的执行

对于 delete 语句我们分为如下两种执行策略：

（1） where 条件为空

此时我们只需根据分片信息将对应的 delete 语句发送到对应的站点

（2） where 条件不为空且表是水平划分

此时我们将 delete 语句发送到所有的站点执行，因为水平划分时一条记录只可能存在一个站点中。

（3） where 条件不为空且表是垂直划分

如果 where 条件是包含主键的条件，我们可以将 delete 语句发送到所有站点执行，如果不包含主键，那么我们从 where 条件中的属性所在站点 select 到主键，然后将 where 条件用主键的信息替代发送到每个站点执行。

3.2.3.6 load 语句的执行

我们现在根据元信息在本地建一个临时表，将 tsv 文件导入，之后根据元信息，select 出某分片的数据，发送给对应的站点，这里我们使用的是 insert 一条一条插入数据，插入数据总共花 3 分钟左右，其余的时间主要消耗在字符串的封装和发送上。

3.3 其它模块

其它模块在设计上其实没太多好说的，RPC 主要是使用了 gRPC 的 proto 文件定义了通讯接口，其后通过工具直接生成.go 文件；etcd 搭建 embed 的过程稍有难度，但也没有太多设计上的内容好说；和数据库的连接使用的是 go 自带的 sql 包的接口。

主要还是反思几个设计上的问题。

一个是 RPC 接口在设计初期就只考虑了 string 的传输方式。结果大量的数据需要先进行封装，传输后还需要进行解析。封装和解析花费的时间随着数据规模的扩大指数型增长，导致了虽然通讯很快但是传输前后耗时很久的问题。实际这类通讯传输应该依靠 RPC 进行流式传输，每一条都及时传输处理，对性能的提升应该会非常明显。

另一个是 etcd 的封装。前期设计的时候是设想把 key 的生成写在 storage 包中，由封装 etcd 的接口负责硬编码和管理。但是没有坚持，最后实际实现是在 Data Manager 中另行生成，导致代码量不必要的增长和维护成本的增大。

4 个人总结

4.1 柴茗珂

本学期的分布式数据库让我收获很多。首先范老师用简单易懂的语言介绍了分布式数据库的基本原理还有事务冲突的处理等。有时讲的知识和张孝老师的数据库知识相互呼应就会觉得非常有意思。最重要的是，通过大作业的开发，对分布式数据库架构有了更加清楚的认识，还新学了 Go 语言和 pg 的使用，以及 RPC 的调用。和队友的合作也体会到了团队合作的乐趣，也非常感谢队友们的鼓励和支持。但是，在编码的过程中，随着代码量的增大，考虑因素的增多，会发现自己设计的策略的不足，也会根据程序的问题产生很多新想法新策略，正所谓实践出真知。对于本系统，非常遗憾的是表的数据封装为 String 再传输消耗时间过多，流处理应该会更好。还有由于并行执行会有很多冲突，应该改变策略增加 channel 等。

4.2 沈运龙

记得当初上课的时候范老师劝我们选课需谨慎，没想到当初头铁还是选了。因为本科期间学的不是计算机基本上也没写过太多的代码，从一开始焦头烂额不知所措，到最后终于憋出了 2000 多行的代码，虽然说代码的结构确实有点混乱，质量也不是特别高，Load 也超级慢，但是好歹也实现了要求的所有功能😊，完成了大作业的任务，自己的代码功力也有了很大提升。最后感谢一直以来队友的支持和鼓励，感谢范老师为我们带来的精彩的分布式数据库课程以及这个具有挑战性的大作业。

4.3 洪殷昊

这门课程让我收获颇丰，特别是有些内容和我在做的项目能相互对照时，就觉得非常有意义。我的工作主要是搭建程序框架和运用一些库的 API 建起自己的存储和通讯的接口。因为工作量没有前面两位同学大，就另外负责了一些提供思路和沟通之类的工作。对我来说也是提升很多，涨了一些管理项目的经验。但是这个过程中工作做的还是不够好，有一些一开始设计时产生的缺陷，带来了后续的性能问题；有些地方因为沟通不够顺畅，走了弯路或者没有能够实现我们所想到的最好的办法。通过这次大作业，以后也能尽可能避免少踩这些坑。