分布式数据库
实验报告

田洪亮 2009210913
彭　辉 2009210872
张　超 2009210897

2010 元月 17 日

目录

1 需求分析

实现一个满足如下功能的分布式数据库系统。

1.1 系统构建

(一) 构建一个分布式数据库，要求能够创建命名数据库和表，并且支持如下分片方式：
- 水平分片，必选
- 垂直分片，必选
- 混合分片，可选
- 引导分片，可选

(二) 构建数据字典，

(三) 根据数据库定义，分片和分配定义从已有的文件中导入数据到数据库中。

1.2 SQL 语句支持

支持 SQL 语句 *select…from …where…*。where 子句支持 AND 连接的简单谓词条件，但仅限于此，不需要支持其它复杂特性。

1.3 功能需求

图形界面或命令行皆可，只要支持输入如下命令：
- createdb 创建数据库
- dropdb 删除数据库
- createTable 创建一个关系表
- dropTable 从数据库中删除一张表
- select … from … where… 查询语句
- insert 向表中增加一条记录
- delete 从表中删除记录

1.4 系统环境

最终的分布式数据库系统需要能运行在由网络连接起来的三台计算机上。体系结构必须支持 P2P。对于每一个节点而言，本地数据库管理系统可以选用商业数据库或开源数据库。节点之间的通信可以选用 socket，RPC 或者其它，程序设计语言每个项目组可自行选择。

1.5 分布式查询处理

最终的分布式数据系统需要包括查询分解组件，查询本地化组件和查询优化组件，其中查询优化组件需要完成如下功能：

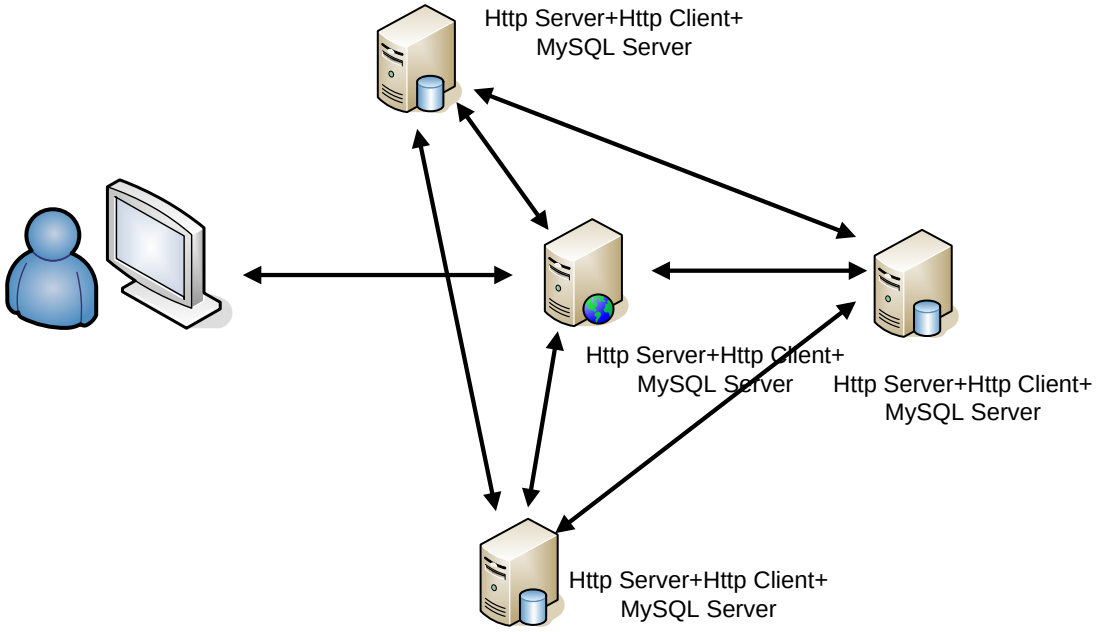
- 优化最初的全局查询树
- 使用分片信息化简查询树
- 网络通信流量优化

2 系统架构

2.1 连接模型

整个分布式数据库管理系统的连接模型如下图所示，客户可以通过网络连接到三个站点中的一个主站点上。主站点分析并处理相应的命令请求，生成查询执行计划，发往各从站点，各从站点执行完自己的查询动作后，再根据设定的评价准则选择最优的组合并返回结果到客户机的方式。

三个场地上共运行四个站点，其中一个场地上会运行两个站点。站点之间的结构是 P2P 的，即无论哪一个站点都可以作为主站点，并且哪一个站点出故障都不会导致系统的整体故障。客户到站点和站点之间的通行采用 HTTP 协议，在服务器端运行 Servlet 容器，项目中采用的是 jetty。站点的本地数据库系统采用 MySQL。

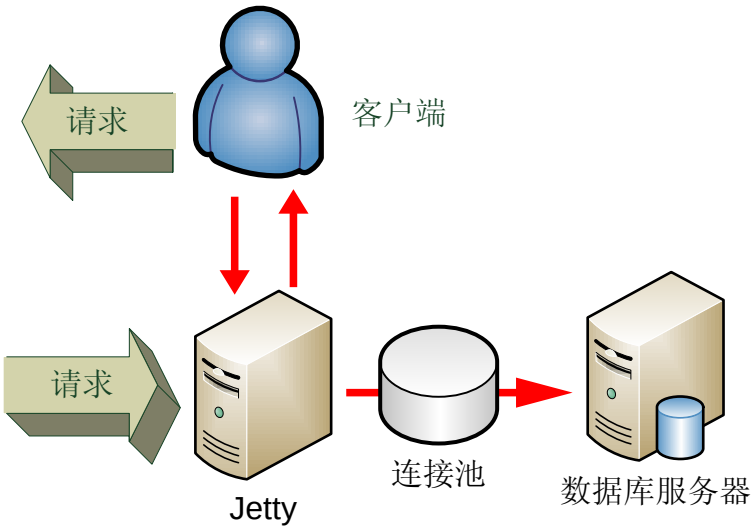


图表 2-1 系统连接模型

2.2 服务器端结构

2.2.1 设计

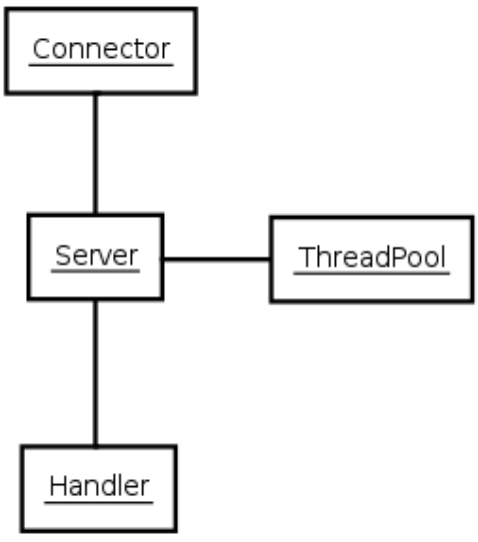
- 节点结构



图表 2-2 节点的结构图

上图是我们实现的分布式数据库系统中，每一个节点的结构图。每一个节点上的前台由一个嵌入式的 http 服务器 jetty 和一个 http 客户端组成，其中 jetty 用于负责对其他服务器的请求进行处理，而客户端用于向其他的服务器发送请求；后台就是一个 MySQL 数据库服务器引擎，当前节点的数据全部保存在这个数据库中。为了提高数据库服务器的性能，在 jetty 和数据库服务器之间加入了一个数据库连接池，所有的数据库访问都通过连接池来完成。

- jetty 结构图



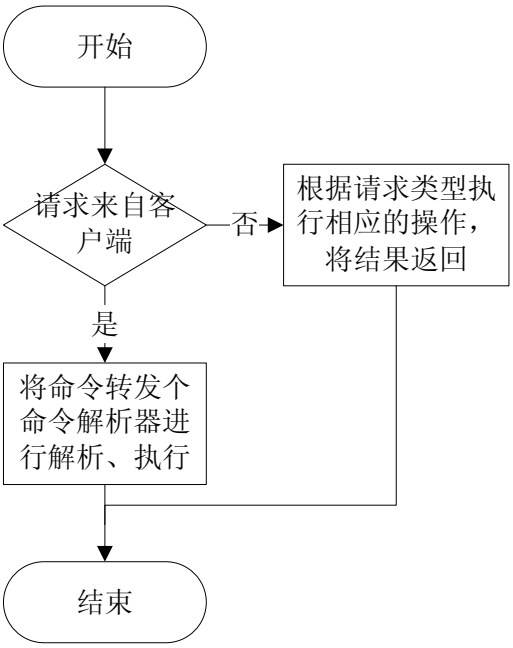
图表 2-3 jetty 结构图

上图是 jetty 的一个简单的结构图。实际上这个服务器主要有三个大模块组成：

（1） 连接器模块：主要负责处理来自 http 客户端的请求；
（2） 事件处理器模块：根据用户的请求进行事务处理并产生 http 响应；
（3） 线程池模块：上面这些实际工作全部由线程池中的线程运行完成的；

这其实是任何一个应用服务器的大致结构图，jetty 相对于其他 http 服务器来说主要不同之处在于 jetty 支持异步请求的处理。所以我们在实现过程中，只需要对 jetty 中的 server 进行配置，用我们自己实现的 Handler 来进行事务处理。

- Handler 的处理流程



图表 2-4 Handler 的处理流程图

上图是我们实现的 Handler 大致流程图，根据请求的 URL 来区分请求时来自客户端还是来自其他服务器，如果是来自客户端的请求，就从请求中将命令提取出来，将命令转发给解析器，对命令进行解析，而后根据命令的语言进行相应的处理；如果请求是直接来之服务器，根据请求的类型，从 http 消息中提取出相应的数 SQL 命令或者执行计划，或者直接将 SQL 命令传给数据库服务器执行；或者对执行计划进行解析，按照执行计划的要求完成相应的任务，最后将结果返回给请求端。

2.2.2 代码实现

服务器端最重要的类是一个单态类 DdbServer，这个类的结构如下图：

图表 2-5 DdbServer 类

其中，main()函数是整个服务器端的入口函数，在这个函数中首先调用 DdbServer 类的 DdbServer.makeConfig（）静态函数读取配置文件到 DdbServerConfig 类型的 config 变量中。会生成一个名叫“server”单态的 DdbServer 实例。

图表 2-6 DdbServerConfig 类

配置文件存放在 conf 目录下的 config.xml 文件中。

本项目中使用的配置文件如下：

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<datapool>
 <server_port>9001</server_port>
```

```
    <local_db_address>localhost</local_db_address>
    <local_db_port>3306</local_db_port>
    <local_db_username>root</local_db_username>
    <local_db_password>666666</local_db_password>
    <local_db_name>ddb_test</local_db_name>
    <local_db_max_connection>50</local_db_max_connection>
    <local_db_timeout>30</local_db_timeout>
    <local_db_max_reconnection>50</local_db_max_reconnection>
  </datapool>
```

完成初始化设置后，就调用 DdbServer.start()函数开始服务器端的运行。所谓运行其实就是指做两件事，一个是监听，另一个是创建连接池。

监听是通过调用 DdbServer. httpServer 成员变量的 start()函数，这个成员变量是 httpServer 类型的，而 DdbHttpServer 类型的 start()函数会启动 jetty Servlet 容器从而完成监听任务。

图表 2-7 DdbHttpServer 类

创建连接池是通过构造 ConnectionPoolManager 类型的 DBUtil.cpm 变量完成的。

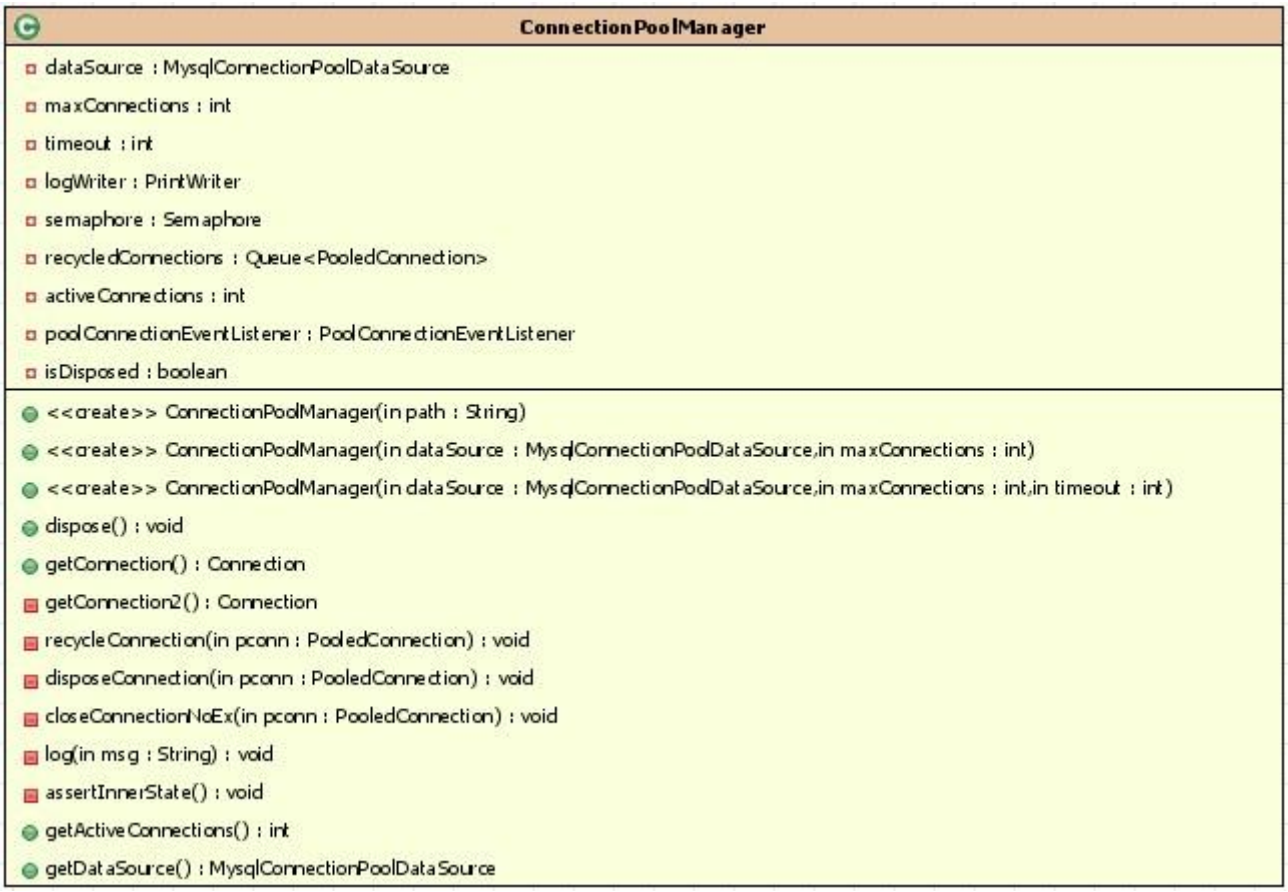


图 3.5 ConnectionPoolManager 类

至此完成了服务器端运行框架的搭建，而具体消息的处理是有 jetty 容器调用 EventsHandler 类的 handle()函数具体处理的。

图表 2-8 EventsHandler 类

3 SQL 解析器

我们使用了 ANTLR 作为构建 SQL 解析器的工具。ANTLR 是一款优秀的 Java 编译器生成工具，它可以以 EBNF 格式接受 LL(k)语法并生成抽象语法树。ANTLR 中将词法规则和语法规则集成在一起放入一个*.g 文件中，在此处为了清晰期间将两者分开描述。

3.1 词法分析

首先在词法规则部分我们定义了如下词法记号：

```
terminator      :   EOF;
STRVAL          :   '\" (~('\\'|'\")|('\\\"))*    '\";
EQ              :   '='   ;
GT              :   '>'   ;
LT              :   '<'   ;
LE              :   '<=';
GE              :   '>=';
OR              :   'or';
AND             :   'and';
TO              :   'to';
CREATE          :   'create';
DELETE          :   'delete';
DOT             :   '.' ;
DROP            :   'drop' ;
INSERT          :   'insert' ;
INTO            :   'into';
FROM            :   'from' ;
SELECT          :   'select' ;
TABLE           :   'table' ;
COMMA           :   ';' ;
INT             :   'int';
STAR            :   '*';
WHERE           :   'where' ;
VARCHAR         :   'varchar' ;
```

```
VALUES          :   'values'    ;
DEFINESITE      :   'definesite';
CREATEDB        :   'createdb';
DROPDB          :   'dropdb';
CREATE_FRAGMENT :   'fragment';
HORIZONTALLY    :   'horizontally';
VERTICALLY      :   'vertically';
ALLOCATE        :   'allocate';
IMPORT          :   'import';
DATABASE        :   'database';

INTEGER: '0' | SIGN? '1'..'9' '0'..'9'*;
NAME: LETTER (LETTER | DIGIT | '_')*;
STRING_LITERAL: "'" NONCONTROL_CHAR* "'";
IP_ADDR: INTEGER'.'INTEGER'.'INTEGER'.'INTEGER;
SIGN: '+' | '-';

fragment NONCONTROL_CHAR: LETTER | DIGIT | SYMBOL | SPACE;
fragment LETTER: LOWER | UPPER;
fragment LOWER: 'a'..'z';
fragment UPPER: 'A'..'Z';
fragment DIGIT: '0'..'9';
fragment SPACE: ' ' | '\t';
fragment SYMBOL: '!' | '#'..'/' | ':'..'@' | '['..'`' | '{'..'~';
WHITESPACE: SPACE+ { $channel = HIDDEN; };
```

3.2 语法分析

在 ANTLR 中，冒号：表示 BNF 中的定义符号“：：=”， “->^”表示改写规则，改写规则用于根据之前的匹配模式构造相应的抽象语法树。

各条命令在语法规则部分定义如下：

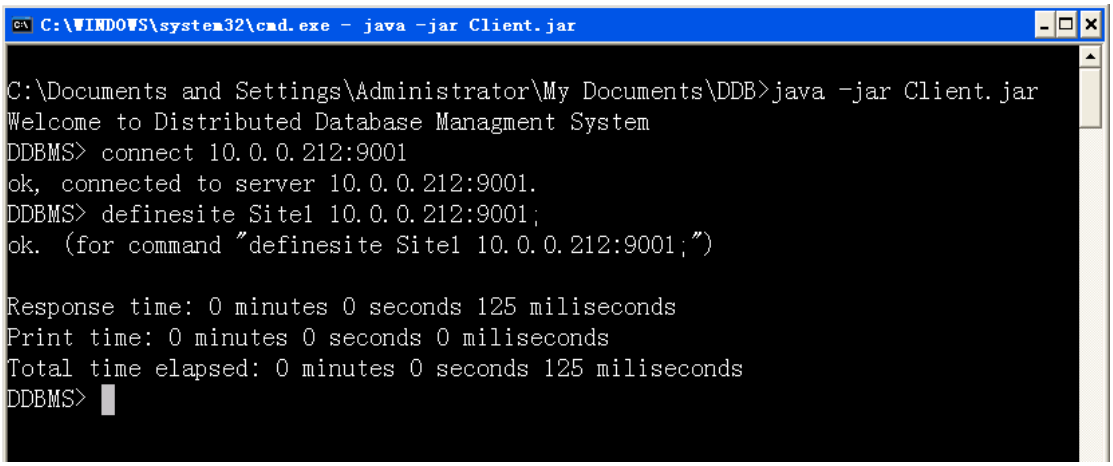
```
statement : (defineSite | createdb | dropdb | createFragment | allocate | importFile | createTable | delete | dropTable | insert | select) COMMA EOF! ;

defineSite : DEFINESITE siteName=NAME ip_addr=IP_ADDR port=INTEGER
 -> ^(DEFINESITE_NODE $siteName $ip_addr $port);

createdb : CREATE DATABASE NAME -> ^(CREATE_DATABASE_NODE NAME);

dropdb : DROP DATABASE NAME -> ^(DROP_DATABASE_NODE NAME);

createFragment : hFragment | vFragment;
```

```
    hFragment : CREATE_FRAGMENT tableName=NAME HORIZONTALLY INTO hFragConditions -> ^(H_FRAGMENT_NODE $tableName hFragConditions);

    hFragConditions : hFragCondition (',' hFragCondition)*
     -> ^(CONDITIONS_NODE hFragCondition ^(hFragCondition)*);

    hFragCondition : term | left=term conj right=term
     -> ^(conj $left $right);

    vFragment
     : CREATE_FRAGMENT tableName=NAME VERTICALLY INTO vFragConditions
     -> ^(V_FRAGMENT_NODE $tableName vFragConditions);

    vFragConditions
     : vFragCondition (',' vFragCondition)*
     -> ^(COLUMNS_SET_NODE vFragCondition ^(vFragCondition)*)  ;

    vFragCondition
     : '(' NAME (',' NAME)* ')'
     -> ^(RESULT_COLUMNS_NODE NAME ^(NAME)*);

    allocate : ALLOCATE tableName=NAME DOT fragNum=INTEGER TO siteName=NAME   -> ^(ALLOCATE_NODE $tableName $fragNum $siteName);

    importFile : IMPORT NAME('.' NAME)? -> ^(IMPORT_NODE NAME ^(NAME)?);

    createTable : CREATE TABLE tableName=NAME '(' columnDef (',' columnDef )* ')' -> ^(CREATE_TABLE_NODE $tableName columnDef (columnDef)*);

    dropTable : DROP TABLE table_name=NAME -> ^(DROP_TABLE_NODE $table_name) ;

    delete : DELETE FROM table_name=NAME -> ^(DELETE_NODE $table_name) | DELETE FROM table_name=NAME WHERE conditions -> ^(DELETE_NODE $table_name conditions);

    insert : INSERT INTO table_name=NAME columns?  VALUES values -> ^(INSERT_NODE $table_name columns? values);

    select :  SELECT resultColumns FROM tableNames -> ^(SELECT_NODE resultColumns tableNames )  |  SELECT resultColumns FROM tableNames WHERE conditions  -> ^(SELECT_NODE resultColumns tableNames conditions);

    columns  :  '(' NAME (',' NAME )* ')'  -> ^( COLUMNS_NODE NAME (NAME)* );

    values  : '(' val (',' val)* ')'  -> ^( VALUES_NODE val (val)* );
```

```
conditions :  expr      -> ^(CONDITIONS_NODE expr) ;

expr :  term |  left=term conj right=expr      -> ^(conj $left $right);

term :  columnOrVal rel columnOrVal -> ^(rel columnOrVal columnOrVal);

rel :  EQ              ->  EQ_NODE
 |  GT                 ->  GT_NODE
 |  LT                 ->  LT_NODE
 |  LE                 ->  LE_NODE
 |  GE                 ->  GE_NODE;

columnOrVal  :  (tablename0=NAME DOT)? col0=NAME  ->  ^(COLUMN_NODE $tablename0? $col0) |  val;

val  :  INTEGER |  STRVAL ;


resultColumns :  STAR |  resultColumn (',' resultColumn)*  -> ^(RESULT_COLUMNS_NODE resultColumn ^(resultColumn)*);

resultColumn : NAME      ->  ^(RESULT_COLUMN_NODE NAME) |  NAME  DOT NAME  ->  ^(RESULT_COLUMN_NODE NAME NAME);

conj :  AND ->  AND_NODE  | OR ->  OR_NODE;
tableNames  :  NAME (',' NAME)* -> ^(TABLE_NAMES_NODE NAME (NAME)*) ;
columnDef  : NAME type -> ^(COLUMN_DEF_NODE NAME type) ;
type  : integer | string;
string  :  VARCHAR '(' INTEGER ')' -> ^(TYPE_STR_NODE INTEGER);
integer :  INT -> ^(TYPE_INT_NODE) ;
```

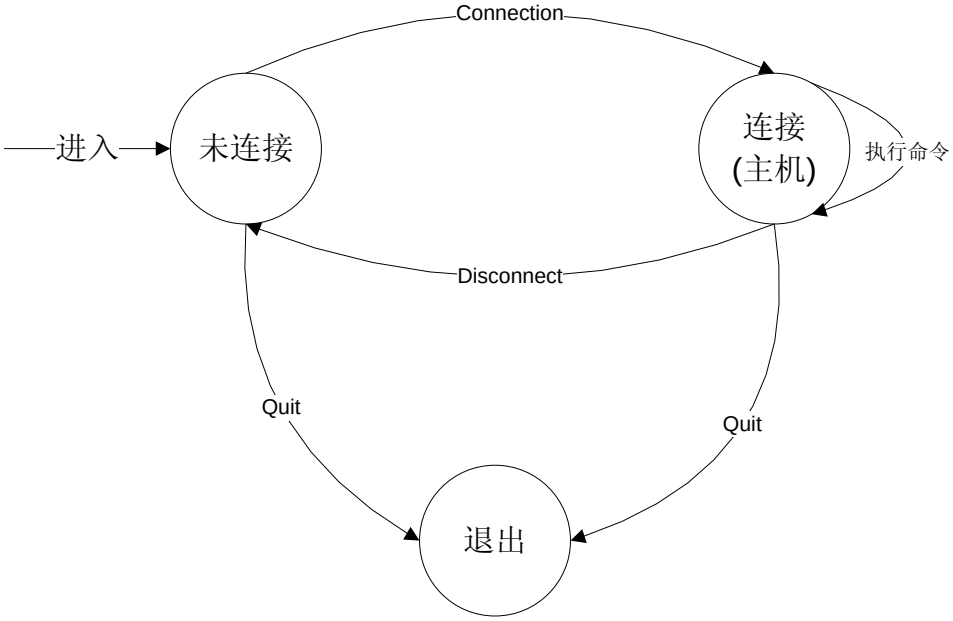
4 客户端

这个分布式数据库系统的客户端做的相对简单，所有操作都是在命令行中完成，数据库系统返回的结果也是全部打印在命令行窗口中。如下图 1 所示。



图表 4-1 命令行界面

在使用过程中，用户首先需要连接到一个到主机上，才能进行其他操作。当用户连接上某个主机之后，客户端就以这个主机作为 Master 节点，下图 2 是这个客户端的状态转移图。



图表 4-2 客户端程序的状态图

在实现上，客户端就是遵循图 2 中的状态转移图，采用自动机的模型。启动客户端的时候，向用户打印欢迎信息，而后就是根据用户输入的命令进行相应的操作。具体的程序流程如下图 3 所示。

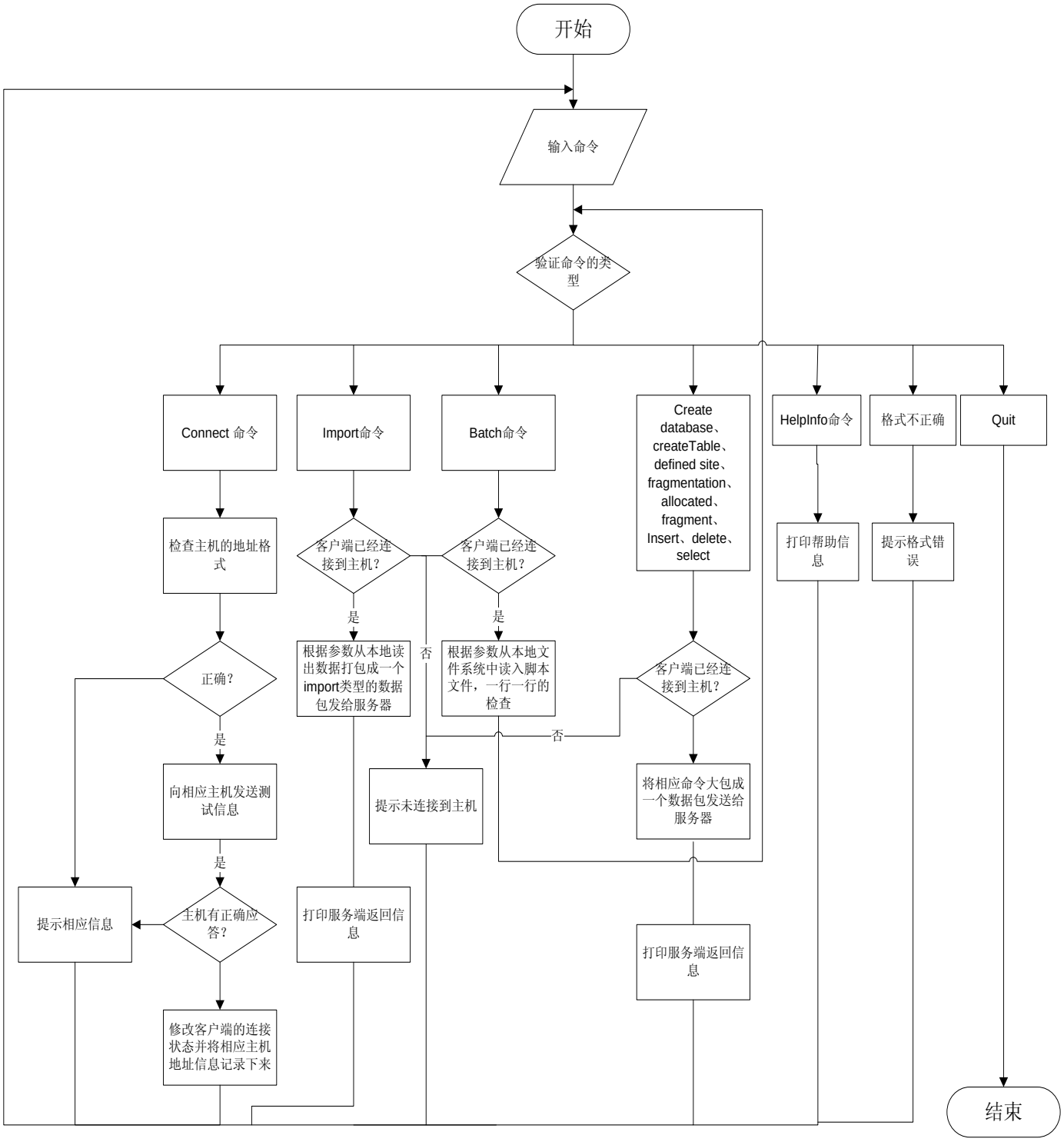
对于用户输入的命令有如下三类：

（1） 仅仅在客户端执行的命令；像断开连接（disconnect），打印帮助（HelpInfo），退出；

（2） 仅仅在服务端执行的命令；像定义站点（definesite），建立数据库，建表，分片插入删除，查询等，这类命令客户端仅仅将这些命令发送给服务器，服务器执行完成之后将结果返回给客户端；

（3） 既需要客户端执行，也需要服务器端执行的命令；像 batch，import，前者需要从用户提供的命令脚本文件中将命令一条一条读出来，而后将这些命令发送给服务器，后者需要从用户提供的数据文件中将数据读取出来，打成数据包发送

给服务器端，服务器将数据导入到相应的数据库中。

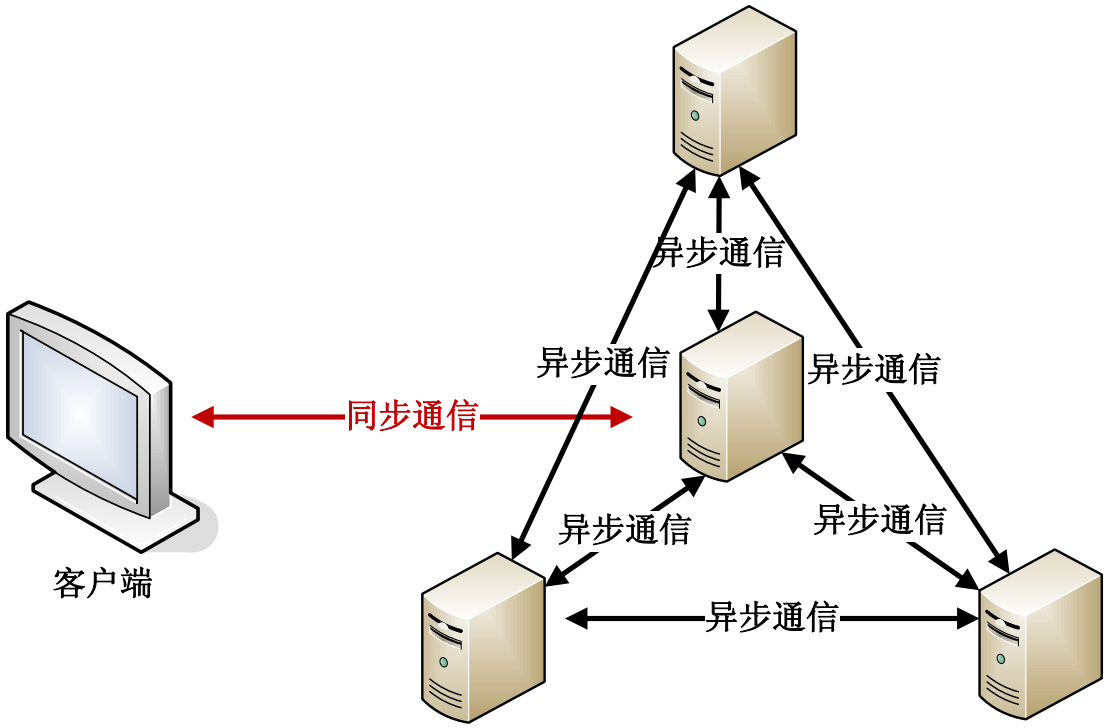


图表 4-3 客户端流程图

5 通信

在这次实验中为了实现的方便，我们直接使用 http 协议，因为目前已经有很多开源的 http 协议客户端和服务器，我们可以直接拿来使用，我们只要在 http 协议的基础上简单定义几种消息格式就可以做为我们的通信协议。

在这次实验中，我们利用了开源的 HTTP 客户端工具包有 apache-http-client 客户端，jetty-http-client 客户端以及 jetty http server，apache-http-client 特点是文档比较丰富，基本上没有什么使用问题，但是利用这个客户端工具包开发异步通信相刚对比较困难。Jetty-http-client 刚好相反，基本上没有什么可用的文档，但是它非常方便实现和服务器的异步通信。而 jetty 服务器是一个轻量级的嵌入式 http 服务器，额外开销很小。所以在这次实验中，在客户端和 master 节点之间的通信就利用 apache-http-client 进行同步通信，系统中各个服务器之间的通信就利用 jetty- http-client 进行异步通信，如下图 4 所示。



图表 5-1 通信模式

所以我们的通信协议分为两部分，客户端-master 之间通信协议和服务器-服务器之间通信协议。因为利用 http 协议进行通信时使用的地址是 URL，所以我们是利用不同的 URL 来区分客户端-master 之间的通信还是服务器-服务器之间通信的。

表格 5-1 客户端-Master 之间通信协议

命令类型	HEADER	BODY	URL
除 import 外的命令	Command：命令	空	http://server:port/client

Import 命令	Command：命令	要导入的数据	http://server:port/client

上表格 1 是我们定义的客户端-Master 之间的通信协议。客户端当用户需要导入数据的时候，将数据放在 http 消息的 BODY 部分，将命令以一个 http HEADER 中；对于其他命令，就直接将消息以一个 HEADER 的形式发送到 Master 节点上。

对于 Master 发送给客户端的返回消息，就直接将所有的消息放在 BODY 之中，客户端收到回复的时候，直接将 http 消息中的 BODY 部分打印给用户。

服务器-服务器之间的通信协议相对要复杂的多。对于这一部分的协议又可以分成两块：

表格 5-2 服务器-服务器之间通信请求协议

请求类型	HEADER	BODY	URL
插入	REQ_TASKTYPE：insert Command：命令	空	http://server:port/server
删除	REQ_TASKTYPE：delete Command：命令	空	http://server:port/server
查询	REQ_TASKTYPE：select Command：命令	执行计划	http://server:port/server
导入数据	REQ_TASKTYPE：import Command：命令	要导入的数据	http://server:port/server

对于请求这一部分，在实验中，我们只定义了这些请求方法，http 消息中加入了一个消息类型 HEADER，并将这个消息的要执行的 SQL 命令放到另一个 HEADER 中。对于查询命令，由于我们的查询执行是以递归方式执行的，所以，要将一个查询计划发送到其他服务器上。关于这一点，在后面的查询执行小节有详细的介绍。

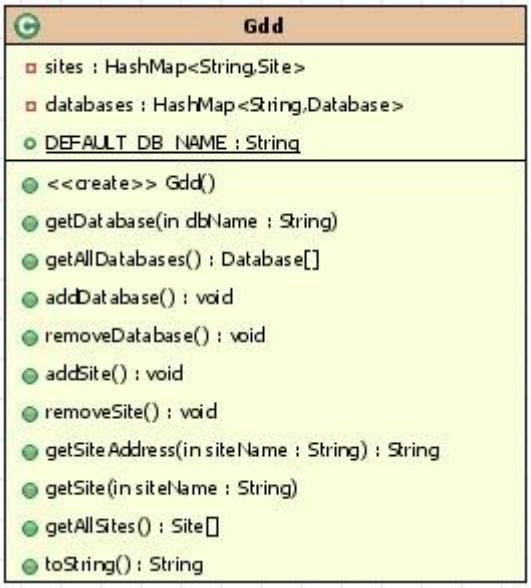
表格 5-3 服务器-服务器之间通信回复协议

回复类型	HEADER	BODY	URL
查询结果	RES_TYPE：RES_TYPE_RS RES_BYTES_TRANS：数据传输量	查询结果集	
执行状态	RES_TYPE：RES_STATUS	正常情况下数据库服务器返回的执行状态结果	
出错	RES_TYPE：RES_TYPE_ERR	数出现异常的情况下服务器返回的执行状态结果	

对于服务器返回给其他服务器的请求，我们定义了如上表中的三类回复，如果服务器在执行过程中出现了异常，则返回一个出错回复，并将错误的异常的具体信息放在 http 消息的 Body 部分；如果请求是一个更新操作，则返回一个执行状态的回复，将服务器执行的状态结果放在 http 消息的 BODY 部分中返回给其他服务器；如果请求是一个查询操作，则返回一个查询结果类型的回复，同时将数据传输量以一个 HEADER 放在 http 消息中，最后将查询结果放在 http 消息的 BODY 中。

6 全局数据字典

全局数据字典是通过单态类 Gdd 实现的，Gdd 类的结构图如下：

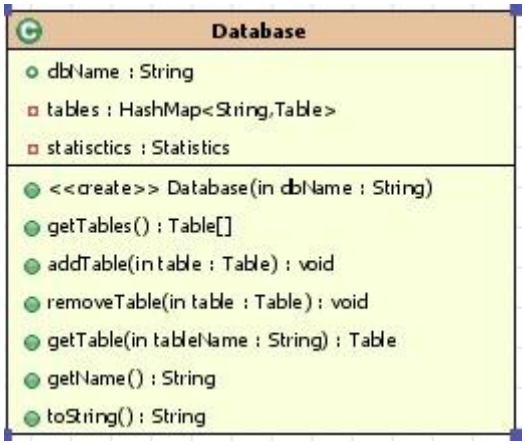


图表 6-1 Gdd 类

通过全局数据字典的接口函数可以使用 Gdd 的下列功能：

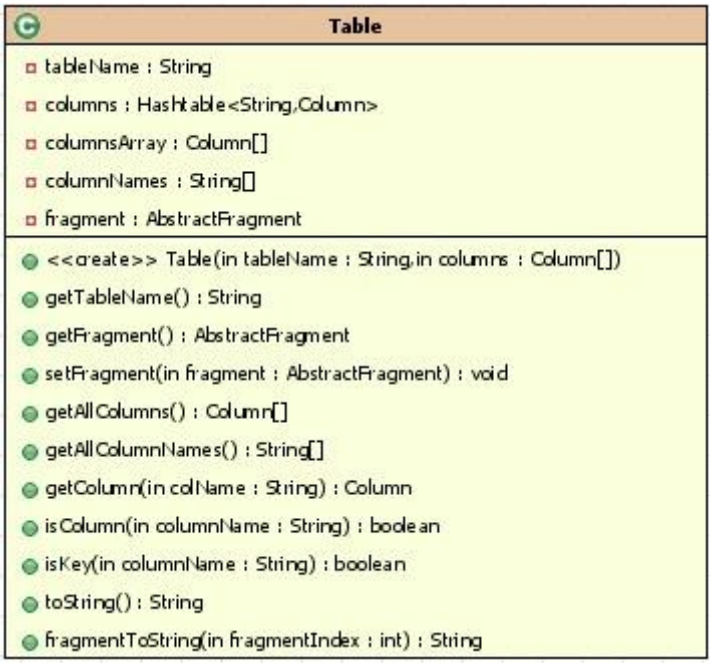
- *添加数据库*
- *获取数据库*
- *删除数据库*
- *添加站点*
- *获取站点*
- *删除站点*

另外数据库信息是存储在 Database 类中的。



图表 6-2 Database 类

表信息存储在 Table 类中。

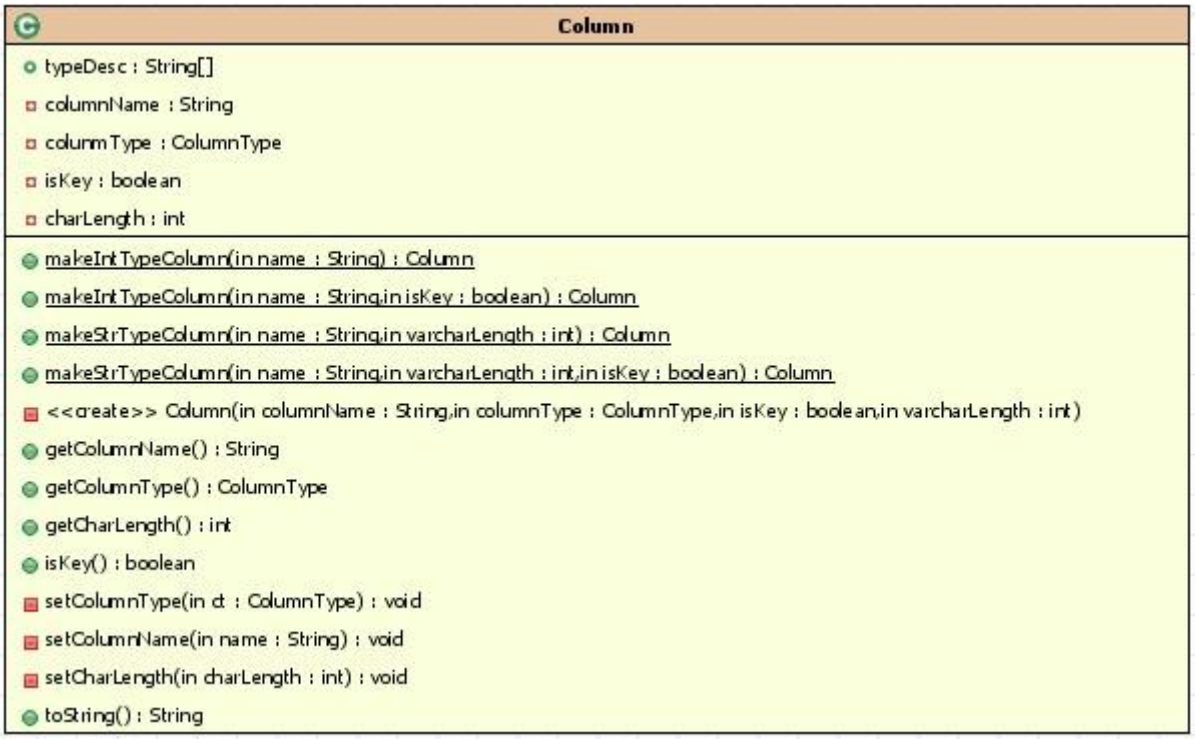


图表 6-3 Table 类

站点信息存储在 Site 类中。

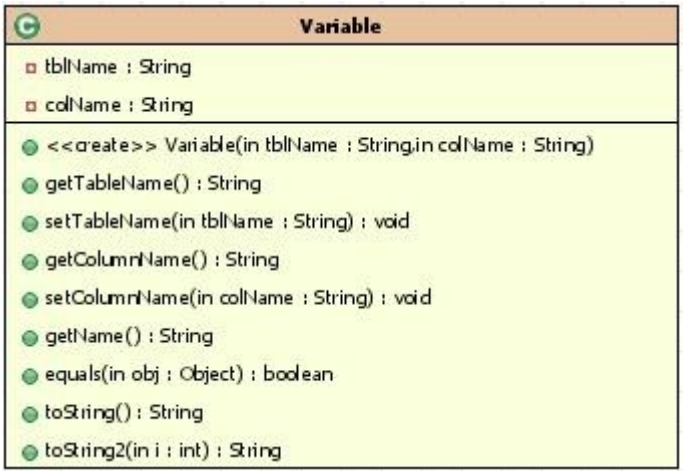
图表 6-4 Site 类

列信息存储在 Column 类中。



图表 6-5 Column 类

变量信息存储在 Variable 类中。



图表 6-6 Variable 类

7 全局查询处理

全局查询处理主要由四步组成：查询分解（Decomposition），数据本地化（Data Localization），全局优化（Global Optimization）以及查询计划（Plan Generation）生成。

在我们的程序中，查询语句用一个叫做 SelectClientCommand 的类表示。而上面这四步是由三个分别实现了 Decompositioner，Localizer 和 Optimizer 三个接口（Interface）的类完成的。

- 查询分解

根据教材上的介绍，查询分解主要分为四步：正规化（Normalization），分析（Analysis），

除冗余（Elimination of Redundancy），改写（Rewriting）。

正规化这一步的目标是把查询中可能任意复杂的 where 部分的谓词化成合取范式（CNF ，Conjunction Normal Form）。由于测试的查询语句的 where 部分都比较简单和规整，要么是单独的谓词，要么是全部由 AND 连接的谓词，所以我们的实现中就省略的这一步，假定输入的查询语句本来就符合合取范式。

分析主要是为了排除类型或者语义上有不正确的语句。我们通过读取 GDD 检查查询中涉及到的数据库表和表中的列是否存在，以及谓词的类型是否符合数据库表的列的类型定义。

我们的实现中除冗余这一步被省略了。

改写是查询分解中最重要的组成部分。查询分解的输出是一棵关系代数（Relational Operator）的树，同时这也是数据本地化的输入。查询分解主要由两步组成，（1）构造关系代数的树；（2）重构这棵树以优化性能。

我们的关系代数树中有下面 7 种类型的节点，而在改写这一步只使用到了 Join、Select、Project 和 AccessPath 这四个：

- ➔ Union：联合，用于把很横向分割的表组合到一起
- ➔ Join：连接，用语把纵向分割的表组合到一起，或者把多个不同的表组合在一起
- ➔ Select：选择表中的满足一些谓词的条目
- ➔ Project：投影
- ➔ AccessPath：访问本地数据库表
- ➔ Receive：用于通信，表示需要接受从远程服务器上返回的数据

  Send：用于通信，表示需要把产生的结果发送给远程服务器

改写中的第二步，即重构这棵树以改进性能，我们把它推迟到了优化那一步再做。

■ 数据本地化

数据本地化的输入是上边得到的关系代数树。在分解中，访问某一个数据库表的操作是用 AccessPath 来表示的。但是我们知道，在分布式数据库中一张数据库表实际上被横向（Horizontally）或者纵向（Vertically）地分割成了几张划分表（Fragment），而这些划分表可能分布在不同的节点上。数据本地化就是要指名这些划分表，以及在哪个站点上存放的。对于横向划分的数据库表，我们使用 Union 操作把划分表组合到一起；对于纵向划分的，我们使用 Join 操作把他们组合起来。

■ 全局优化

优化是整个分布式数据库的最重要的部分，同时也是本项目难点。为了能取得比较理想的优化结果，我们查阅了大量的分布式数据库查询优化的文献，包括经典的 System R*和 Distributed Ingris 的论文。最早介绍这两个分布式数据库鼻祖的文献可以追述到 30 年前的。看了这些几十年前的论文，不得不感慨一下计算机排版和印刷的技术的进步。在 Distributed Ingris 的论文中，由于当时的网络条件很差(1980 年左右)，UC Berkley 的研究人员用的本地网络媒介竟然都不是以太网！总之，最早的有关 System R*和 Distributed Ingris 的论文由于历史太久远参考价值有限。我们查到的对我们项目最用的是两篇文章是《The State-of-the-Art in Distributed Query Processing》和《Iterative Dynamic Programming：A New Class of Query Optimization Algorithms》。

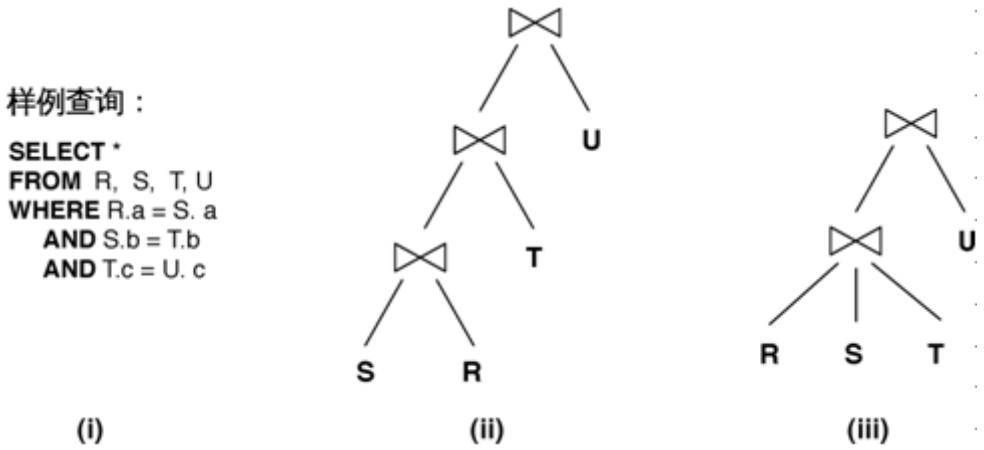
查询优化算法主要有三类穷举搜索（Exhaustive search），启发式算法（Heuristics），随机算法（Randomized Algorithms）。下面简要介绍了三种算法，并总结了它们的优劣势。

- 穷举搜索，即在解空间内穷举所有可能的解并从中找到最优的。这种方法的好处是，

如果代价函数（Cost Function）足够精确，那么这种方法可以保证找到最好的解。但是这种算法的时间和空间开销是以指数增长的。代表例子有：动态规划（Dynamic Programming），比如 R*系统就采用的是这种方法；A*搜索；基于变换的技术，比如在 EXODUS、Volcano 和其他商用系统。

- 启发式算法。这种算法的优点是它的时空复杂度是多项式的，但是这种方法产生的执行计划常常比穷举搜索的差一个甚至几个数量级。代表例子有：贪心算法（Greedy Algorithm），比如 Distributed Ingris 就是用的是一种贪心的算法决定 Join 的顺序，而且只考虑了左深度计划(Left-Deep Plan)而完全不管更一般的 Bushy Plan；Minimum Selectivity 也是一种贪心算法。
- 随机算法。优点是常数的空间开销，对复杂查询要比穷举搜索或者启发式算法快，而且可以在很多情况下产生好的查询计划。但是随机算法处理简单的查询就要比前两钟方法慢，而且有时候产生的查询计划的开销会比最优的计划高一个甚至几个数量级。

综合以上考虑，作者觉得由于测试的查询语句最多涉及到 4 个数据表，这样用动态规划穷举的开销就不会很大，而且动态规划在理论上能给出最优的解（如果开销（Cost）函数足够精确），所以作者决定选择基于动态规划的穷举搜索作为查询优化的算法。穷举搜索的一个直接的好处是不仅仅 Lef t Deep Tree（下图中间），而且 Bushy Tree（下图右边）也会被考虑。



图表 7-1 Left deep tree vs. bushy tree

另一个问题是是否使用 Semi-join 的技术。文献[1]指出“Experimental work indicates that semijoin programs are typically not very attractive for join processing in standard (relational) distributed database systems because the additional computational overhead is usually higher than the savings in communication costs [Lu and Carey 1985; Mackert and Lohman 1986]”。所以作者对采用 Semi-join 能达到的效果持怀疑态度，因此我们没有采用 Semi-join 作为一项优化技术。

虽然在设计之初我们考虑很多查询优化的方法，但是到项目的最后阶段我们发现已经没有时间做优化。程序中最终只实现了 Select 和 Project 下移这种简单的优化。所以很自然我们在助教评测的时候测得的数据传输比较大，但是由于执行模块写得非常高效，所以响应时间都非常短（最短的小于 1s，最长的不超过 10s）。这充分说明了分布式数据库性能的好坏，一方面受查询计划优劣的影响，另一方面执行效率的高低也是很重要的影响因素。

- 全局优化
- 查询处理

8 本地查询执行

8.1 查询计划的递归处理算法

查询处理模块的输入是查询语句，而输出是查询计划。这个查询计划会被发到各个站点执行。在内存中，查询计划是以增加了 Send 和 Receive 原语的关系代数树的形式表示的，而网络传输时，查询计划是以 XML 形式表达。比如查询语句

select * from Customer

的查询计划以关系代数树的形式可以表示成：

```
Join (Customer.id)
|---ScanAccessPath (Customer.1, Site1)
|---Receive (from site Site2@127.0.0.1:9002)
    |---Send (to site Site1@127.0.0.1:9001)
        |---ScanAccessPath (Customer.2, Site2)
```

而以 XML 形式表示就是

```
<?xml version="1.0" encoding="UTF-8"?>
<Join key_col="Customer.id">
    <ScanAccessPath site="Site1@127.0.0.1:9001" table="Customer" frag_index="0">
    </ScanAccessPath>
    <Receive send_to="Site2@127.0.0.1:9002">
        <Send receive_from="Site1@127.0.0.1:9001">
            <ScanAccessPath site="Site2@127.0.0.1:9002" table="Customer"
frag_index="1">
            </ScanAccessPath>
        </Send>
    </Receive>
</Join>
```

而收到这种XML形式的查询计划的节点会把它再转换成内存中的关系代数树，然后递归执行。

查询计划的递归执行算法的伪代码如下：

```
If 是叶子节点
//叶子节点必是 AccessPath，即执行访问数据库表
返回数据库划分表的数据；
ELSE IF 本节点的类型是 RECEIVE
把本节点的子树（以 SEND 作为根节点）发送给指定的站点执行，作为结果返回；
ELSE
遍历所有叶子节点，递归调用本算法，把叶子节点的结果保留下来；
SWITCH(本节点的类型)
    CASE JOIN：
        把叶子节点的结果 JOIN 起来，作为结果返回；
    CASE UNION：
        把叶子节点的结果 UNION 起来，作为结果返回；
```

```
        CASE SELECT：
            选择叶子节点结果中满足谓词的数据，作为结果返回；
        CASE PROJECT：
            选择叶子节点结果中的某些列的数据，作为结果返回；
        CASE SEND：
            把本节点的唯一子节点的结果发送给等待这个结果的远程站点；
    ENDSWTICH
ENDIF
```

从上面这个非常简洁优雅的查询计划执行的递归算法中可以看到，由于把Send和Receive这两个通信原语引入到关系代数树中，我们不需要为每个操作指定其执行的站点，默认情况都是在收到这个查询计划的本地站点执行的。只有Send和Receive时才需要考虑站点和通信问题。

Receive和Send总是同时搭配使用的，正如我们举的select * from Customer的例子。Receive只有一个唯一的子节点，并且一定是Send；任意Send的父节点一定是Receive。Receive节点的功能很简单，就是把以Send为根节点的子树发送给Receive中指定发送站点。而远程节点在接到了Send为根节点的子树之后，会执行Send以下的子树，然后把得到结果发送回在Send中指定的接受站点。

8.2 数据操作的高效执行

本系统的一个最大优势就在于高效的数据操作实现。我们听说大部分小组的 union、join、select、project 等操作都是采用了在数据库中存零时表，依靠数据来执行这些操作的。但是稍加思考就可以发现，理论上来说，直接在内存中对中间结果进行 union、join、select、project 等一系列操作显然要比先把数据导到数据库中（很可能有 IO 开销）再让数据库操作这些数据，然后再读回内存要快。自己写这些数据操作一来很费时间，二来容易出错。但是，我们就这么做了！而结果是，我们的执行效率非常之高。10 个查询测试包括 2 个隐藏查询都正确，并且响应时间最快的不到 1s，最慢的超不过 10s。

执行效率之所以很高，一方面得益于所有数据操作都在内存中执行，另一方面在于我们在实现的时候力图做到“零”数据拷贝。比如 Project 操作，我们所做的，只是把需要的列标记出来，而没有做任何数据拷贝。Select 操作也是类似，根据 Select 操作的谓词，符合的数据条目被标记出来，不符合的不标记，没有任何数据被删去，或者被拷贝。Union 就更不需要拷贝了。Join 也没有拷贝任何数据， 而是使用了一个 hash 表作为索引，使得原本 O(n^2) 时间复杂度的操作降低为 O(n)。

综上，我们数据操作的实现具有非常高的效率。

9 代码组织

源代码的分为 src 和 test 两个目录。src 下是主要的源代码，test 目录下是单元测试。下表列出了 src 目录下的所有 17 个 Java 包（共包含 131 个类）并给出了各个包的简要描述。

- org.ddb.client：客户端；
- org.ddb.commands：用户能够执行的各种命令，比如 create table, import, select, print gdd 等等；

- org.ddb.commands.select：查询处理相关，包括查询分解，数据本地化和优化；
- org.ddb.commands.select.ddbresultset：查询处理涉及到的各种查询结果，包括 Join、Union、Select、Project 和 AccessPath 等操作的查询结果的表示；
- org.ddb.commands.select.relationaloperator：关系代数树（同时也是执行计划）的数据结构；
- org.ddb.commands.select.ralationaloperator.visitor：遍历关系代数树的访问者，比如执行查询计划的访问者，输出成 XML 的访问者；
- org.ddb.communication：通信，包括 HTTP Server，服务器-客户端通信，服务器-服务器通信；
- org.ddb.gdd：GDD 及其相关的类，包括 GDD 本身，以及 GDD 序列化和反序列化的类；
- org.ddb.parser：Antlr 根据语法文件生成的词法器和语法器；
- org.ddb.query：Insert，Delete 和 Select 等命令的服务器-服务器通信的表示；
- org.ddb.server：分布式数据库服务器的主类，以及读取配置文件；
- org.ddb.sqlentities：GDD 中存放的各种信息的数据表示，比如数据库表，站点，列，统计信息等；
- org.ddb.sqlentities.fragment：划分表的数据表示，包括横向和纵向划分；
- org.ddb.sqlentities.predicate：Insert、Delete 和 Select 等命令中的 where 从句中的谓词；
- org.ddb.sqlentities.predicate.visitor：谓词的访问者；
  org.ddb.util：工具类；

10 分工和工作量

姓名	任务
田洪亮	整体框架 GDD 查询处理 查询优化 查询执行
彭辉	客户端 通信 本地数据库 搭建测试环境和 svn 服务器
张超	Parser 查询命令生成
田洪亮、彭辉、张超	系统整合 集成测试 文档

11 个人总结

11.1 田洪亮的总结

在本学期的所有课中，分布式数据库带给我的收获最大，而分布式数据库大作业则是这门课的精华。

首先，这个大作业极大地锻炼了我的编程能力。实际上，这是我写的第一个总代码行数超过 10，000 行的软件。我们的程序总计有 1.2 万行代码，其中 4000 余行是 Antlr 根据语法文件自动生成的，而在剩下的 8000 多行手写的代码中有 6000 行左右是我写的。以前从来没有在这么短的时间内写如此之多的代码。第 15、16 周，甚至连元旦假期，我都全身心的投入到分布式数据库大作业的编程中。最终，我们的程序能够高效地跑对全部 10 个测试查询语句，包括 2 个隐藏测试，我感到非常欣慰和自豪。

其次，这个大作业给我一个难得的设计复杂系统的机会。以前写的程序或者软件，功能比较单一，结构也相对简单。但我们实现的分布式数据库总计包含 26 个 Java 包，156 个 Java 类。为了写出功能完善，结构清晰，组织合理且易于维护的程序，我花了一周时间查阅分布式数据系统以及查询优化的文献，设计整体框架、流程以及 GDD 等关键数据结构，画出了超出好几个屏幕大的 UML（统一建模语言）设计图。虽然有队友提出质疑，认为设计过于“复杂”，在某些设计问题上（比如执行计划的表示）有分歧，但是最终的实现基本还是按照我原先的设计，而且证明了这套设计是可行的。看到自己的想法付诸实施并得到验证是件令人很有满足感的事情。

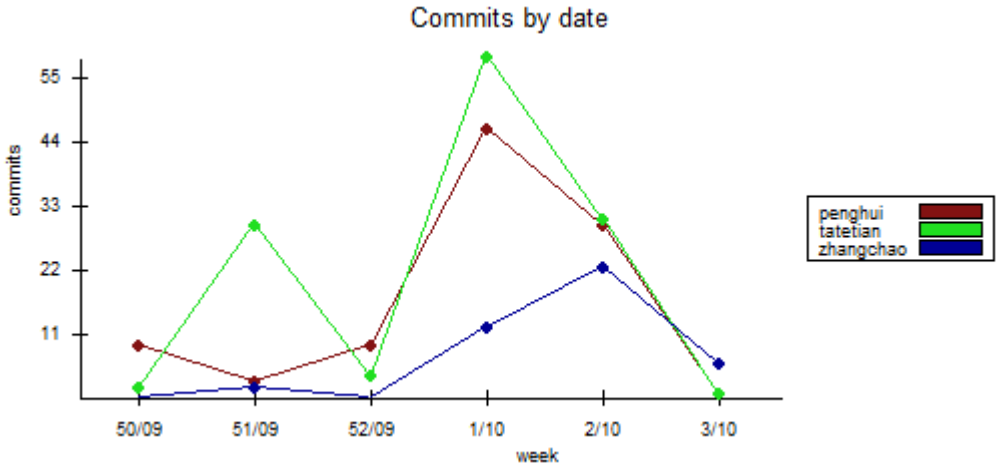
还有，在这次大作业中，我深深体会到了团队合作的重要性和艰巨性。 作为我们组的组长，我自然希望负起责任、带领整个团队走向成功。但是每个人的想法、个性和背景都不同，统一思想、团结协作并不是说起来那么简单的。如何能够有效沟通？如何能够激励他人？如何能够说服他人接受自己的观点？更重要的是，如何倾听别人的观点并且自己主动做出妥协？在经历过这次大作业的磨练以后，我想我在人与人的相处、交往和合作方面变得更成熟了。

最后，我想感谢两位队友的鼎力相助，说想说：“缺了我们三个任何一个，我们都不可能成功！”

11.2 彭辉的总结

总的来说，这个课程设计对于我来讲是一个比较大的挑战。我主要想谈一谈我的感受和一些经验总结。

第一， 一定要规划好时间，准确把握好项目进行过程中的每一个时间点。我们组这次就没有规划好。下面这个图是我们实验过程中代码提交的情况，实际上，前期我们基本上没有编写多少代码，大部分代码都是要检查前两周写的，所以导致我们很多优化结果都没有来得及做。



第二， 设计上越简单越好。其实我们都知道，越简单的东西，越可靠，实现起来也越容易，但是简单的东西不一定能满足我们现实的要求；所以这是一对矛盾。但是，我个人觉得，我们在设计的时候，应该从一个最简单的模型开始进行设计和实现，在实现了基本功能的基础上，再考虑其他的高级功能或者效果。

第三， 分工上很有讲究；我个人觉得，在一个小组中，大家一起做事情，如果分工很好，能够做到各部分的耦合度达到最小，相互之间的工作进度影响能够做到最小的话，这是最理想的状态，但是实际上往往也很难做到；我们这次做的也不是很理想，出现了相互卡壳的现象。

第四， 利用开源工具固然有好处，可以有效减少自己的工作量，但是很多开源软件或者工具包都有一个致命的弱点就是开发文档不是很规范，出现问题很难找出来，在这次试验过程中，jetty-http-client 就有一个比较严重的问题，但是 jetty 的文档里面连 JAVA DOC 都写得特别简单，更不用说他的其他文档了。另外，我们在这次使用的 HTTP 协议来传输数据，实际上是从某种程度提高了我们的数据传输量的，因为 http 协议是一个文本传输协议，只支持文本传输，所以，所有的数据都是以文本的形式进行传输的，这一点从某种角度上增大了我们的数据传输量。所以，我个人觉得，像我们这种非常简单的协议，一种更可取的方法是直接自己开发协议。

11.3 张超的总结

在选这门课之前就得知，这门分布式数据库系统课以“难”和“好”著称，到现在完全上完这门课后对此更有体会。“难”对每一个选了这门课的人都是显而易见的，倒不是理论的艰深和两次考试内容的难度大，而是对这门课的重头戏分布式数据库项目而言的。无论从最开始的分析，设计，中间的编码，到最后准备检查时的彻夜奋战，个中心酸每个参与的人都明白。不能像有些科目一样，一学期不用上课，最后突击也能过，分布式数据库不行，没有详细的规划和认真的工作就不可能按时拿的出满足要求的代码。“好”就是“难”的必然结果了，只有通过这些扎实的训练，才能真正的理解所学的概念和实现过程。

就项目本身而言，团队合作绝对是成功的关键。每组三人，大家的背景不同，擅长不同，能力也有大有小。只有充分的合作才能保证最终的完工。合作的过程中感受就是讨论问题时大家就要畅所欲言，枚举出各种可能情况和选择的利弊，而一旦确定方案后，就应全部遵守。

最后要衷心感谢同组的两位队友，是他们出色的设计和实现能力完成了项目的绝大部分工作，并使我们的系统最终顺利通过评测。