

NaiveDB项目文档

作者信息

纳鑫

- 学校：清华大学软件学院
- 学号：2016013276
- Email: naxinlegend@outlook.com

李帅

- 学校：清华大学软件学院
- 学号：2016013270
- Email: lishuai16THU@163.com

王泽宇

- 学校：清华大学软件学院
- 学号：2016013258
- Email: ycdfwzy@outlook.com

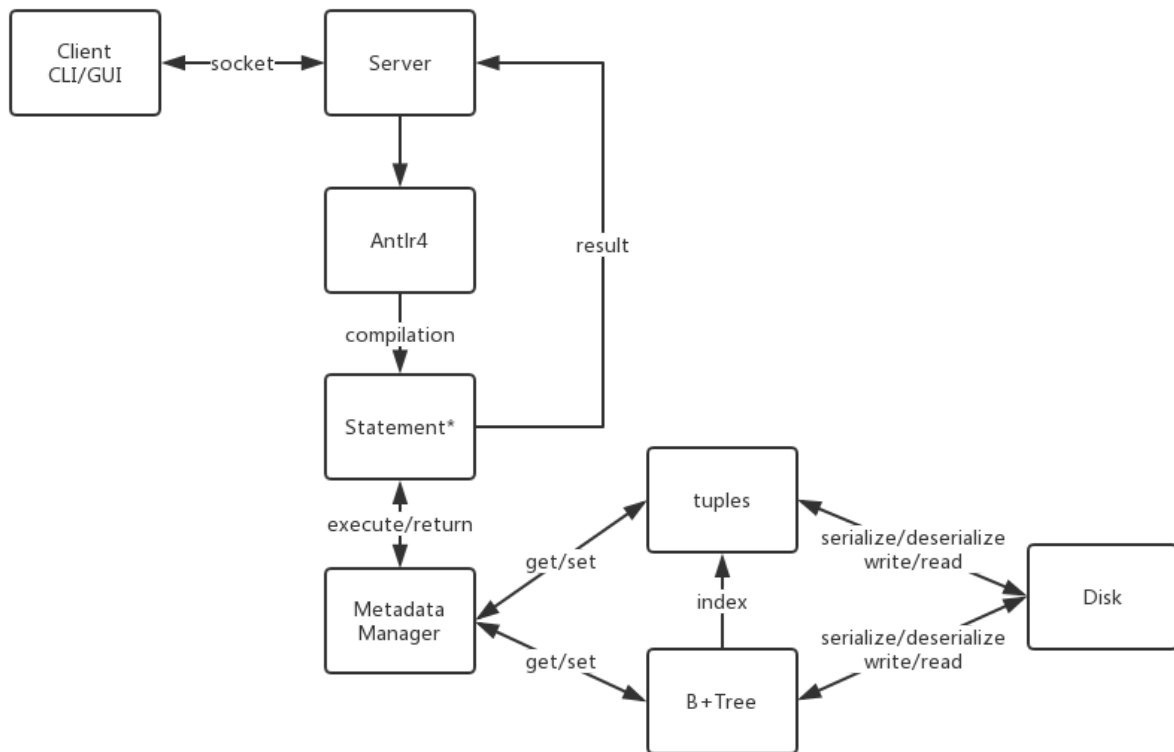
项目简介

NaiveDB是一个简易的数据库，支持五种数据类型：Int, Long, Float, Double, String（定长字符串）；一些简单的数据库操作：建立删除切换Database、创建删除Table、记录的增删改查；客户端（命令行和服务器版）与服务端的连接。

使用介绍

使用介绍请见Github[仓库](#)的[Wiki](#)。

数据库设计



整体流程参见上图

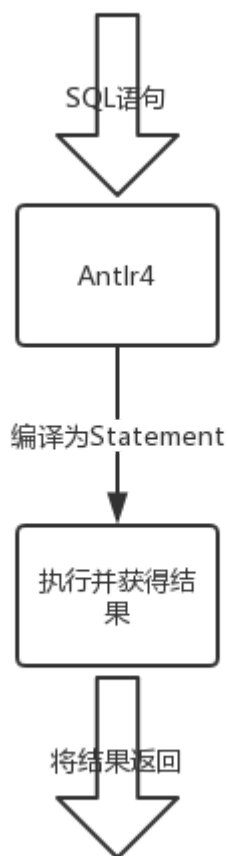
存储模块

见 [存储模块.pdf](#)

元数据处理管理模块

见 [元数据处理管理模块.pdf](#)

查询模块



首先将输入的SQL语句编译为Statement类。这里编译采用了Antlr4的Visitor模

式。Statement类拥有exec()的执行函数，可以将执行编译好的SQL语句，并将返回结果，遇到执行错误会抛出异常。对于where子句和on子句后面的条件判断，我们支持了and/or以及复杂的逻辑嵌套。我们通过实现Conditions类来实现这个功能：Conditions类本质上是一个二叉树节点类，每个节点或者有两个Conditions子节点，代表 cond1 and cond2 或者 cond1 or cond2，或者有两个Expression类的成员变量，代表 expr1 op expr2，op 是二元比较副，包括等于、不等于、小于、小于等于、大于、大于等于。Expression类的实现方式类似于Conditions类，也是一个二叉树节点类，不过它标识了一个表达式的求值结构，具体实现就不再赘述。

Join的实现

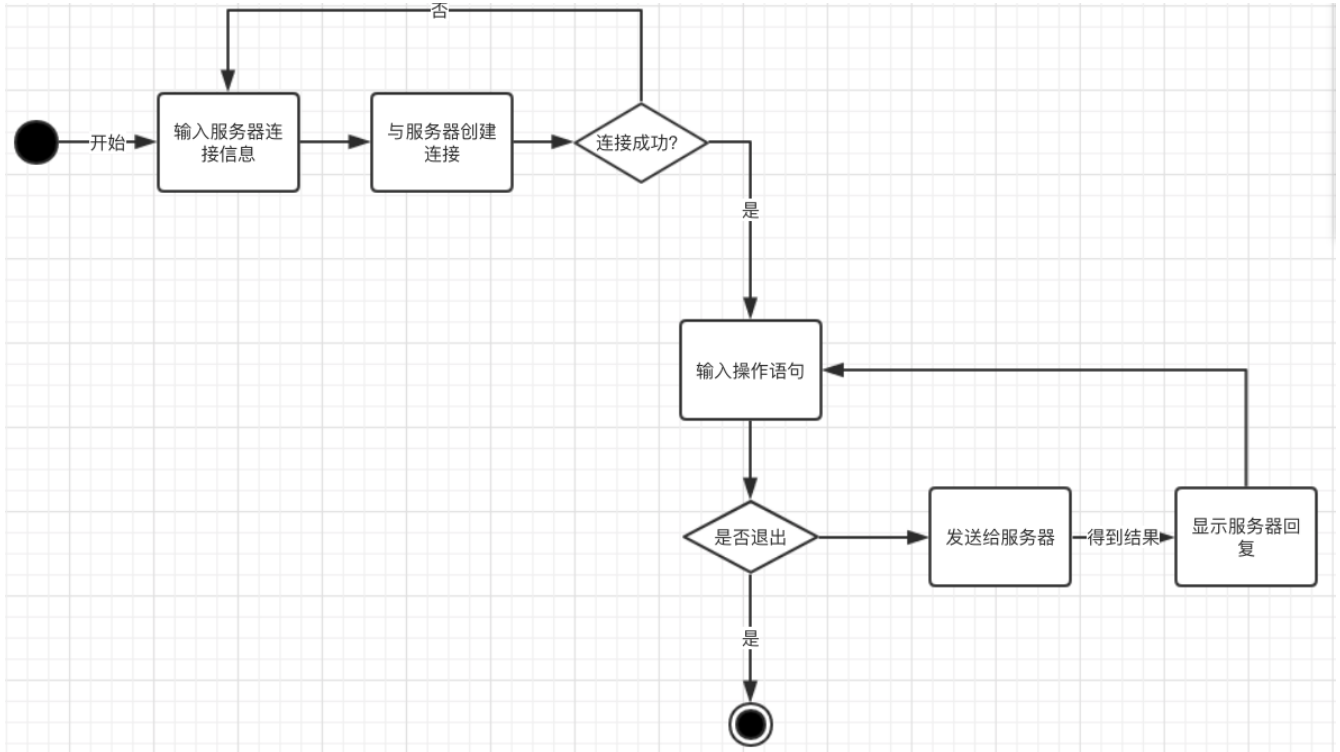
我们实现了对完整的Join操作的支持，包括

- natural join
- join...on...
- (left/right/full) outer join
- (inner) join
- 笛卡尔积
- 以上多种Join操作的复合（多表join）

我们的基本实现采用了Nested Loop Join。对于等值连接的操作我们采用了Hash Join进行优化，并且对于连接条件涉及到单表的操作我们也进行了一些优化，达到了较好的效果。具体实现请看下方亮点部分。

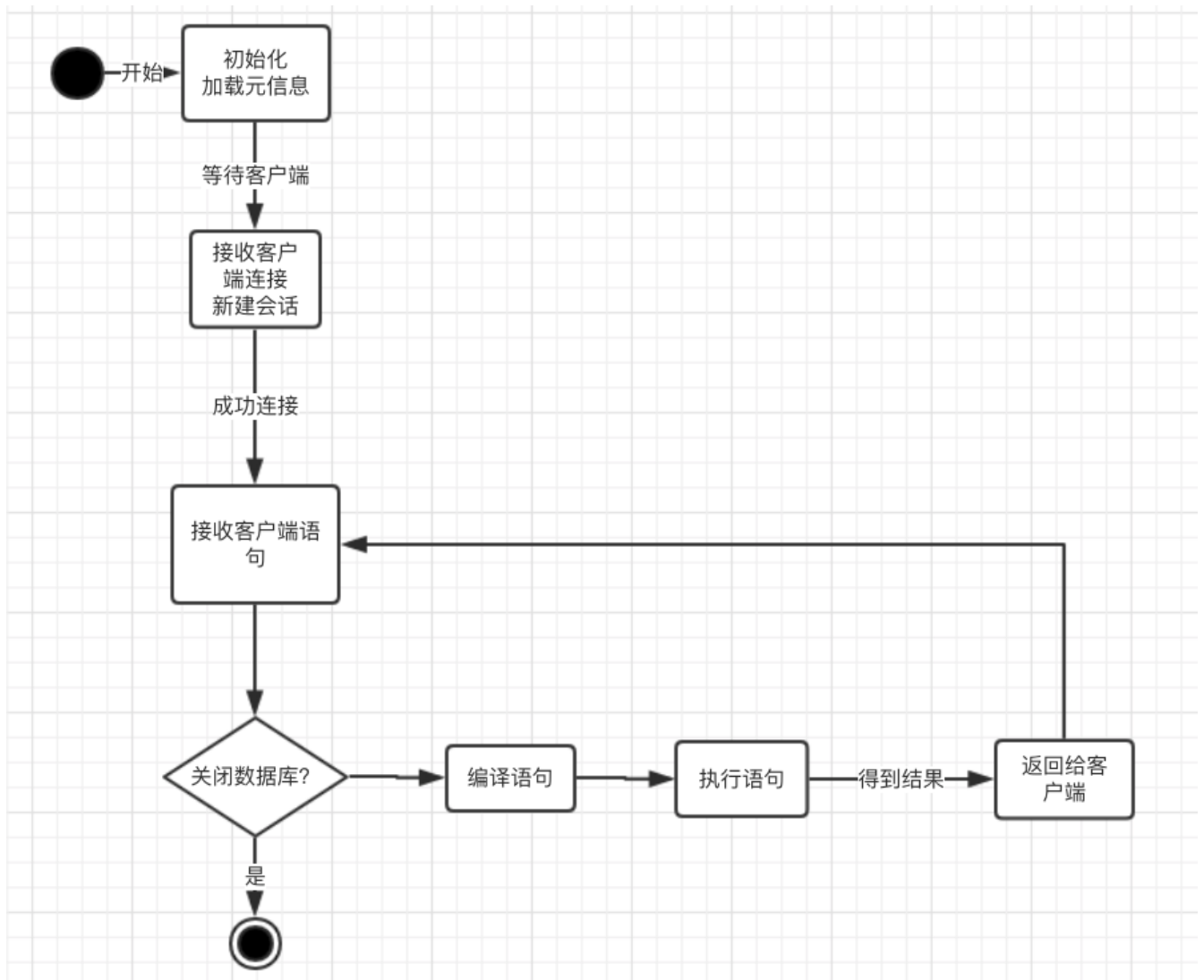
客户端通信模块

- 客户端总体流程



客户端启动时建立和数据库的连接，之后进入REPL循环，接收一条输入语句，将其传递给服务端，得到结果并展示，若为退出则传递给服务端后退出即可。

- 服务端总体流程



服务端等待与客户端建立的连接，建立成功后接收给出的语句，首先进行编译，之后执行，在编译-执行过程中出现错误都记录到结果中，最终一起返回给客户端，接收到关闭数据库的消息则关闭数据库。

- 通信方式

通信方面直接使用Socket通信，服务端和客户端通过套接字传递信息。在客户端发送给服务端的信息中，只包含了要传输的SQL语句（先传入一个整数表示SQL语句长度）。客户端在接收了SQL语句后直接执行，将其结果保存在一个 `ServerResult` 的对象的实例中，然后使用Java io的ObjectStream即对象流传输，其可以在套接字上直接传输一个对象，这样就省去了发送网络请求等麻烦。客户端接收到此对象的实例，然后根据其中的内容进行显示即可。

- 客户端实现

我们实现了命令行的客户端和GUI的客户端，具体使用方法请见使用介绍。命令行客户端方面，使用了jline包包装REPL，支持命令的编辑，查看历史命令以及命令的自动补。GUI方面使用JavaSwing编写，具体请见下方亮点部分。

亮点

B+树与查询优化

存储模块实现了B+树索引，并且利用B+树对下列查询做了优化（假设主键为 `pk`）：

- 查询条件为 `pk=x`，那么可以利用B+树做精确查询。

- 查询条件为 `pk<x`，`pk<=x`，`pk>x`，`pk>=x`，可以利用B+树做区间查询
- 查询条件为 `x<pk<y`，`x<=pk<y`，`x<pk<=y`，`x<=pk<=y`，同样可以利用B+树做区间查询。

如下图，当使用主键id为查询关键字时，从十万条数据中筛选仅需要9ms，而不使用主键，则需要高达1074ms。

```
naivedb> select * from stu where id='50000';
+-----+-----+
| ID    | NAME    |
+-----+-----+
| 50000 | name50000 |
+-----+-----+
1 rows in set (9 mill sec)

naivedb> select * from stu where name='name50000';
+-----+-----+
| ID    | NAME    |
+-----+-----+
| 50000 | name50000 |
+-----+-----+
1 rows in set (1074 mill sec)
```

缓存机制

我们在三个部分使用了缓存机制：持久化记录、B+树、表格Table。在持久化记录中，我们允许在内存中常驻一定大小的记录（默认限制时1M，可以在Consts类中修改），缓存中的记录超过这一大小时，随机将一些记录写回磁盘。在B+树中，一个节点存在一个page中，同样我们允许内存中常驻一定大小的节点（默认限制时1M，可以在Consts类中修改），缓存中的节点大小超过这一大小时，随机将一些节点写回磁盘。对于表格Table，将其缓存到Database里，每个Database可以缓存一定数量的Table（默认缓存上限是10个，可以在Consts类中修改），缓存中的Table数量超过这一大小时，随机将一些Table写回磁盘。在没有实现缓存机制时，我们测试执行1000条插入语句（每次必须将Table写回），共执行了9120ms（除去antlr的编译时间），加入了缓存及之后，仅执行了31ms（除去antlr的编译时间），效果拔群。

多数据库实例

我们支持多个数据库，在数据库首次初始化时，会创建 `data` 文件夹，在此文件夹中，每个文件夹是一个独立的数据库，这样数据库之间不会互相影响。我们使用一个DatabaseManager类管理数据库，由于数据库管理者应该使用单体模式，故此类中的变量与方法都是静态的，调用时直接使用类名调用。在初始时会创建一个默认数据库“PUBLIC”，且此数据库不能被删除。与数据库相关的SQL语句的使用方法请见使用介绍。

完整的Join语句支持

- 基本实现

我们通过实现RangeVariable类来实现Join功能。RangeVariable类是对查询范围的一个抽象，其本质是一个多叉树节点类。该节点分为内部节点和叶子节点，内部节点中包含一个RangeVariable的数组，其中保存了子节点的引用，表示一个 `t1 join t2 join t3.....` 类型的子句；叶子节点中包含一个Table类型的成员变量，保存了一个Table实例的引用，代表一张单独的表。我们join的基础实现方式是Nested-Loop Join，其伪代码如下：

```
for row1 in t1
    for row2 in t2
        if (row1 combine row2) satisfied condition then add (row1 combine row2) to
temptable
```

为了防止内存的溢出，我们在执行该算法的中间过程并不保存实际的数据，而是保存每条数据在各表中的行号，只有在最内部判断该数据是否满足join condition时才会根据其行号和表实例的数组来获取完整的数据。因此每个RangeVariable中保存了一个名为`ArrayList<ArrayList<Long>> rowNumLists`的成员变量，表示每个子节点执行join操作所生成的中间结果。RangeVariable中还保存了一个`ArrayList<Table> tableList`的成员变量，表示每个行号所属的表。我们在RangeVariable类中通过combineRows()函数来实现该过程。该函数会递归地调用子节点中的combineRows()函数，从而递归地进行join操作并生成相关的结果。在执行select语句时当执行select操作时，Antlr4会将输入的SQL语句中的from子句解析成一棵RangeVariable树，然后调用根节点的exec()函数，该函数会调用根节点的combineRows()函数，该函数递归地实现了上述的Nested-Loop Join算法，其结果是上述的行号的列表，最后在exec()函数中根据tableList和rowNumLists获取实际数据，插入到一个TempTable实例中，最后将该TempTable返回。

- natural join实现

在解析from子句时，若一个join是natural join，在生成RangeVariable时会将其中的isNatural属性设为true。在执行combine()函数时，若当前RangeVariable的isNatural属性为true，则会调用processNatural()函数将当前的表的列名与前一个表的列名进行比较，当找到同名的列时就生成一个condition并加入当前的join condition中，然后将相同两列中的其中一列加入到ignoreColumns数组中，之后就按照普通的join操作进行处理。

连接优化

- Hash Join优化

在每次执行join操作时，我们先判断join condition是否是等值连接，若是则采用一个类似Hash Join的算法来执行join操作。其伪代码如下：

```
for row1_value, row1_num in t1
    add (row1_value, row1_num) to HMap
for row2_value, row2_num in t2
    if HMap.containsKey(row2_value) then add (HMap.getByKey(row2_value) combine row2_num) to
temptable
```

即首先遍历第一张表，使用一个HashMap来保存第一个表中的值到行号的映射，然后遍历第二张表，若在HashMap存在以第二张表中的某个值为键的映射，则将该键对应的行号取出，然后与第二张表中的对应的行号进行连接并加入到中间结果中。由于HMap.containsKey()和HMap.getByKey都采用Hash来进行查询，因此都是 $O(1)$ 的时间复杂度，故该算法的复杂度是 $O(m+n)$ ，其中 m 和 n 是两张表的大小。对比Nested Loop Join的 $O(mn)$ 的复杂度具有极大的提升。

友好的图形界面客户端

我们的GUI客户端如下：

Naive DB Manager Swing

FileCommandRecentHelp

clearexecute

```
select * from department natural join classroom where department.budget > 30000 and (classroom.capacity < 60 or classroom.capacity > 100);
```

DEPARTMENT.DEPT_NAME	DEPARTMENT.BUILDING	DEPARTMENT.BUDGET	CLASSROOM.ROOM_NUM...	CLASSROOM.CAPACITY
Music	Packard	80000	101	500
Finance	Painter	120000	514	10
History	Painter	50000	514	10
Biology	Watson	90000	100	30
Physics	Watson	70000	100	30
Biology	Watson	90000	120	50
Physics	Watson	70000	120	50

* Ready, 7 rows in set, finished in 17 mill sec

上方菜单栏中，File菜单下可以打开、保存、执行一个脚本文件；Command菜单下可以快捷输入指令，执行指令等；Recent下保存了最近执行的20条指令。下方分别是命令输入框以及结果显示框。最下方显示了上次执行的结果，行数以及执行时间。