

数据库课程设计

实验报告

张轩 5080309672

阎立 5080309073

朱崎峰 5080309079

2011 年 8 月 31 日

目录

一、项目综述.....	2
二、分析 SQL 语句，生成语法树.....	2
1. 概述.....	2
2. 语法分析器实现.....	3
3. 语法树类图.....	3
4. 词法分析中的大小写问题.....	3
三、生成执行计划.....	4
1. 概述.....	4
2. 具体实现.....	4
3. 部分细节说明.....	5
3.1 判断树.....	5
3.2 表达式.....	5
四、执行 SQL 语句.....	6
1. 概述.....	6
2. 具体流程.....	6
2.1 表做积.....	6
2.2 where 筛选.....	6
2.3 order 操作.....	6
2.4 group 操作.....	6
2.5 Having 操作.....	7
2.6 投影.....	7
3. 细节说明.....	7
3.1 where 部分的优化.....	7
3.2 关于 order 和 group 的说明.....	7
五、缓冲及磁盘管理.....	7
1. lowlevel 包.....	7
1.1 lowlevel.struct 包.....	8
1.2 lowlevel.CacheAccessStatus 类.....	9
2. schema 包.....	9
2.1 Schema 类.....	9
3. file.SeqFile 类.....	10
4. db 包.....	10
3.1 DataBase 类.....	10
3.2 HashIndices 类.....	11
3.3 TableHelper 类.....	11
六、用户界面.....	11
七、项目总结.....	12

一、项目综述

本次数据库课程设计实验的目标为设计制作一个小型的数据库管理系统 (DBMS)，实现基本的数据库操作。DBMS 使用 java 语言编写，主要包括编译 SQL 语句，生成查询计划，优化查询计划，执行查询计划，缓冲管理，存储管理，用户界面等部分。

数据库的执行流程如下图所示，本报告将就以下各部分做详细介绍。

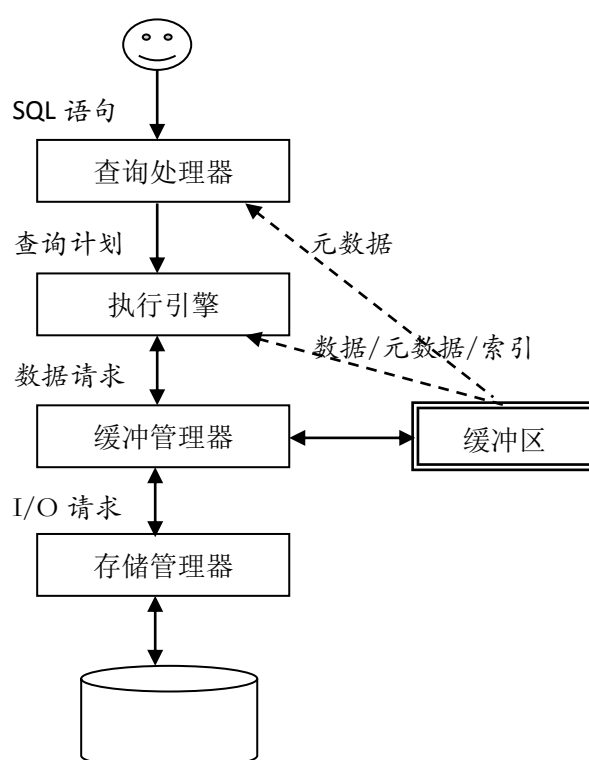


图 1：数据库管理系统组件构成

二、分析 SQL 语句，生成语法树

1. 概述

本阶段以的字符流为输入，识别输入的符号及语法结构，检查语法错误，输出语法树。主要有词法分析和语法分析两个阶段。

词法分析对生成一系列的名字、关键字和标点符号，同时抛弃单词之间的空白符和注释，语法分析对词法分析的输出做进一步检查。这里仿照编译器前端来实现，使用 JLex 和 java_cup 工具，编写 .lex 和 .cup 文件，生成词法分析和语

法分析的程序。

2. 语法分析器实现

此语法分析器支持绝大多数标准 SQL 语言,语言标准参照 MySQL 软件实现。语法分析器返回一个 StmList 对象,每一个 SQL 语句都是一个 Stm。查询的 select 语句分为一般的查询 SimpleQuery, 集合操作 SetQuery, 链接操作 JoinQuery 三种。其中 SimpleQuery 的语法结构为:

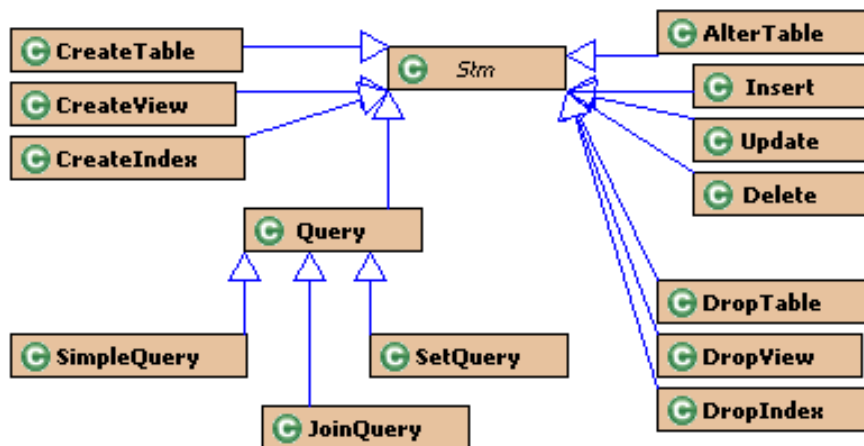
```
SELECT distinct_part field_list FROM tables where_part
        group_part having_part order_part
```

其中 distinct_part 表示是否含有 DISTINCT 标识, where_part 等表示选择语句的条件,当然每一个条件都也可以为空,这里严格按照 SQL 语法标准来实现。以 where_part 为例说明:

Where_part ::= WHERE conditions. Conditions 内可以用 AND, OR, NOT 等来连接 condition。Condition 又分为普通的比较算符, ANY, ALL, IN, Not IN, IS NULL, BETWEEN...AND...等条件。这里不在一一说明。

另外,此语法分析器支持的数据类型包含 DATE, TIME, TIMESTAMP 等。还支持字符串连接算符“||”,“&&”和“||”也可以当作 AND 和 OR 来使用。

3. 语法树类图



4. 词法分析中的大小写问题

由于 SQL 中无视输入的大小写,即 SELECT 写成 select, seLECT 等形式都行,所以对每一种组合进行判断就显得太累赘,这里使用的方法是将所有的词都以 ID 类型读入,然后使用 string.toLowerCase() 方法将 ID 转换成小写之后与"select","create"等字符串进行比较,若相等,则返回此 Token,否则返回 ID:

```

<YYINITIAL> {ID}      {
    String str = yytext().toLowerCase();
    if(str.equals("create"))    return tok(sym.CREATE);
    ...
    return tok(sym.ID, yytext());
}

```

三、生成执行计划

1. 概述

现在已经得到了一棵完整的语法分析树,然后下一步的工作是将这棵翻译成一种中间形式——执行计划,以便后面的执行。

新建这样一棵执行树就我们 DBMS 一些简单的操作而言,看似有点累赘,不过,之所以选择建一棵完整的执行树,而不是选择直接对着语法树直接写执行操作是基于一下一些原因:

1. 格式更为清晰,各层有独立的作用,就类似网络的 7 层协议,执行树就相当于语法层和执行层的接口。
2. 当语法过程或者执行过程修改时,主要只需修改执行树翻译的翻译过程,而不需要进行大面积的修改。
3. 扩展性更强,例如从最基本的查询扩展到支持 **group by**,聚合, **order** 直至最后实现的 **Having** 功能,都只是每层独立修改内容,不需要做全局的变动。
4. 支持优化调整,对于查询的过程,有许多优化算法,如果直接对语法树进行操作的话优化过程难以实现,而对于执行树而言,就相对要容易多了,这有点类似编译中对中间代码进行优化一样。

2. 具体实现

构建的执行计划是一棵树的形式,不过这与构建语法树是有区别的,这棵树构建的主流程是一种自底向上的形式,有点类似一件一件外套往上披。下面做具体说明,对于一个 SQL 的查询,其基本形式是:

```

SELECT fieldList FROM tableList WHERE conditionList GROUP BY groupList
        HAVING having-conditionList ORDER BY orderList

```

这里选择的执行流程为:

```

tableList -> conditionList -> orderList -> groupList -> having-conditionList -> fieldList

```

具体来说就是首先读入表(可以是多个),然后就需要将这些表做笛卡尔积操作,再根据 **condition** 中的内容筛选出一些需要的部分,接着按照 **group**、聚合等算法,在这里做 **order by** 操作可以得到正确的结果而无需做额外的动作(这些细节会在执行查询章节中进行说明)。然后需要 **group**,因为 **Having** 操作内包含聚合操作,需要对当前 **group** 进行,而最后是根据 **fieldList** 做投影工作。

最后得到的执行树的整体结构是:

投影、聚合 $\left(\text{Having 筛选} \left(\text{group 操作} \left(\text{order 操作} \left(\text{where 筛选} (\text{表做积}) \right) \right) \right) \right)$

正如上面说的翻译的过程是由内向外的，过程类似初始一个：

`tree := 表做积`

然后：

`tree := where 筛选(tree)`

以此类推，最后得到一棵完整的执行树。

当然这棵树总体上看是一条线性的操作，而实际有许多分枝，如 `where` 和 `having` 筛选的时候，`condition` 的判断其实是类似深度递归遍历的，而其中的任何一项，譬如 `a`, `b` 是表的两个域，那么类似判断：

`where a + b > b + (a * b) + 3.4`

这里执行判断的树的根部还是比较“茂盛”的。

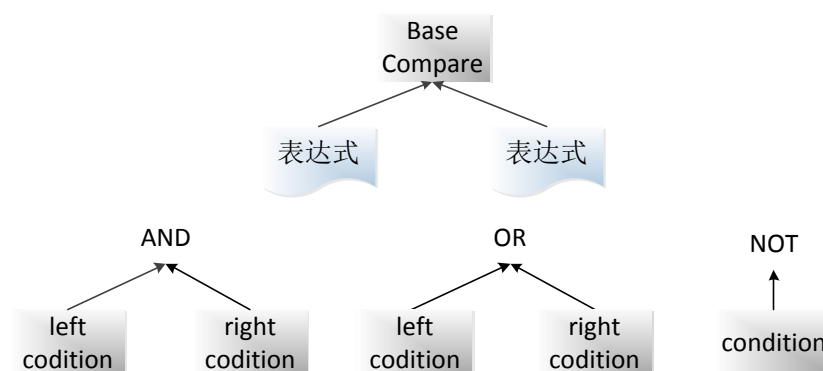
对于 `create` 表之类的语句，语法和执行相对简单，这里不做说明了。

3. 部分细节说明

翻译中最为复杂的部分就是判断树（`condition` 主要在 `where` 和 `having` 中）的建立，然后是一个表达式（任何常数，域，及加减乘除等）的执行树。

这里对语法树的要求是判断 `not`, `and`, `or` 的优先级处理完毕，表达式的运算优先级处理完毕，然后对于需要聚合操作的域，特别注明。

3.1 判断树



上面是最基本的，当然还有一些 `any`, `all`, `in`, `not in` 之类的判断，这里不一列举了。

3.2 表达式

这里的表达式虽说比一般的程序语言要简单许多，但是其含义要丰富许多，所以处理起来要更为细心。最基本的是算数表达式，它的左右都是表达式；然后

有常数，域，聚合域。

特别需要注明的是这里把“*”作为一个特殊的域来对待，而对于域，则分为无点域（类似 a），和有点域（类似 T.a），同样对于聚合域也如此。

四、执行 SQL 语句

1. 概述

有了执行树之后，执行就变得相对要简单许多，拿到一棵执行树，判断它是什么操作（投影，选择等），然后递归执行其子树的内容，对子树的结果执行当前操作，然后返回就可以了。

2. 具体流程

2.1 表做积

这里要注意的是，多个表的话，先将前两个做笛卡尔积，然后将结果与第三个做积，以此类推，于是 $a \times b$ 与 $c \times d$ 的表得到的结果是 $(a \times c) \times (b \times d)$ 的一张表。表的体积会比较庞大，这里可以做的一步优化是将 where 部分提前选择，然后才做积。

2.2 where 筛选

where 的处理方式是逐条 Tuple 判断，即遍历表中的每一行，取出该行，判断它是否满足条件，满足的插入新建的表中。

对于 AND 条件，递归判断左边，如果为 false，直接扔掉；同理对于 OR，左边为 true，直接加入。

2.3 order 操作

order 选用的是类似冒泡排序的原理，而且使用标记。譬如降序，就选出未标记中最大的，标记它，把它插入新表。

2.4 group 操作

同样使用标记，执行流程是取出一行，然后在表中找到和它一样的行，全部标记，写入新表，然后重复操作。

2.5 Having 操作

查询中最复杂的部分，对它的处理是把所有的条件判断全部重载一下，加上 group 的起点和终点，以方便做聚合操作。

2.6 投影

投影中的聚合操作同 Having 类似，然后选取需要的列插入新的表就可以了。

3. 细节说明

3.1 where 部分的优化

对于 in, any, all, 我们做了一些优化，将查询的结果临时保存在内存缓存中，不需要每行 Tuple 都执行，例如：

```
select * from Table1 where a in (select b from Table2);
```

当对一条 Tuple 取出的 a 的值判断时，先去缓存看看有没有 select b from Table2 的结果，有点话直接拿出来用，没有的话执行后存入缓存方便日后用。

3.2 关于 order 和 group 的说明

这里要解释一下为什么先做 order 然后是 group。记得我在课上问过李芳老师，group 后去的域没有聚合操作怎么办，李老师当时说应该报错，而我尝试了许多 DBMS 后发现它们就去 group 中的第一行，所以我这里也是这样处理的。于是，进过 order 之后，我选取的 Tuple 进行 group 判断也是按 order 的顺序，也就是说 group 之后，每个 group 的第一个 Tuple 一定是按 order 顺序的，于是即使没有聚合我去第一条也是按 order 排列的。

五、缓冲及磁盘管理

这里就系统关于缓冲和磁盘管理的几个包做分别说明：

1. lowlevel 包

虚拟磁盘(类 Disk)

虚拟文件(类 File)

缓存系统(抽象类 BufferManager 及其子类，类 CacheAccessStatus，类 Page)

杂项(ListHead, 一个用于嵌入在其他结构内用的轻量级双向链表)

1.1 lowlevel.struct 包

Disk 内的数据结构: Bitmaps, DirEntry, INode, SuperBlock

1.1.1 虚拟磁盘

虚拟磁盘的实现根据《操作系统设计与实现 (第三版)》教材 (5.6 MINIX 3 文件系统) 提到的方法进行。用位图进行块管理, 超级块、I 节点等都有不少简化, I 节点大量属性被略去。一些关键点:

1) 块大小为 4K。这 and 现代操作系统是一致的, 也和缓存的页面 (Page) 大小保持一致。

2) 基本接口仿照类 UNIX 操作系统的文件系统调用。根据 Java 面向对象的原则, Disk 类的公有方法不包括读、写、关闭文件等应当属于文件的操作。openFile、createFile、exists 等属于 Disk 类之公有方法, 因为用户调用操作时并没有 File 对象。故公有方法有创建文件、删除文件、打开文件、判断文件是否存在、删除文件、重命名文件; 创建目录、删除目录、罗列目录以及 sync。

3) 数据传输部分使用 java.nio 的 Channel、ByteBuffer 等, 起到直接进行 low level 操作之效。

类 File 实现了 java.nio.ByteChannel 接口, 除了接口指定的,

```
public int read(ByteBuffer dst) throws IOException;
public int write(ByteBuffer src) throws IOException;
public boolean isOpen();
public void close() throws IOException;
```

还有随机访问文件系列的 seek(), position(), size(), truncate() 等。不过距离继承 java.nio.FileChannel 抽象类还有不少距离。

1.1.2 缓存 BufferManager

BufferManager 抽象类及其子类 LRUBufferManager 是 Buffer 系统的主体。采用页缓存法, 每个页面背后都有一个 ByteBuffer 支撑。LRUBufferManager 按线程安全设计, 并可产生两个线程:

syncThread ---- 定时向磁盘写回缓存中的数据

dropThread ---- 定时清理被标记为不再需要的页面

其中 dropThread 在低系统负载情况下不是很有用, LRUBufferManager 本身的换页策略足以应付, 因而没有主动在系统中添加调用通知 BufferManager 有意清理掉某些页面, POSIX 操作系统有 fadvise(posix_fadvise) 系统调用可以有意地通知系统调整缓存使用, 这里保留之。

1.1.3 换页策略

换页策略是维护一个 LRU 链表，这里用到的 ListHead

该类作为一个双向链表节点类，可以利用 Java 的成员类机制做到“嵌入”的效果：

```
class PageNode extends Page {
    public class ListHead extends lowlevel.ListHead {
        public PageNode entry() {
            return PageNode.this;
        }
    }
}
```

于是在遍历链时，走到一个 ListHead，可以用 entry() 方法得到其所在对象的引用。该思路及这种链节点形式来自 Linux Kernel 的 list.h，container_of 宏给人印象深刻。于是页面每次被访问，就放到链表尾部，换页时丢掉链表头上的页面即可。还能利用他做一些遍历。

1.2 lowlevel.CacheAccessStatus 类

lowlevel.CacheAccessStatus 是 Cache 的一个统计信息，可以通过 LRUBufferManager 的 getAccessStatus() 方法获取，一个 int 数组，按照顺序是：访问请求数、缓存未命中数、被标记为可能不再需要的页面又被访问数、换页数。其中第三个，比如已经关掉的文件的页面，会被标记为可能不会再使用。

2. schema 包

表结构包。包括和底层 byte 流打交道需要的函数。

2.1 Schema 类

表结构类，该类是一张表的结构，生成后即为常量不可改变（即不包含任何公有方法修改之）。创建此类的方法是使用 SchemaBuilder 工具类进行。Schema 类实现了 Iterable 接口，遍历访问该结构的各列（各个列的名称，类型）。其中名称使用了 Symbol 类，该类是 String 的一个小封装，内只存 String#intern() 方法的返回值，如此相同内容的 String 不会是不同的对象。类型则是 Type 类，用 enum 标记该 Type 的 category，attribute(primary, not null, default value 等)，对应的 javaType 等。Symbol 和 Type 也都是创建后不可修改的常量。

在以上条件下，Java 运行时编译环境会做出足够的优化，提升代码的性能。

Schema 内的函数比较多，和底层相关的是 getValue(), putValue(), 和 newValues()。

getValue 接受一个低层传回的 byte[] 参数，一个索引或者一个 Symbol，执行

操作从 `byte[]` 中取出索引或 `Symbol` 对应域的值（以对象形式）。
`putValue` 修改 `byte[]` 中的某一个域，返回修改的结果（`byte[]`）
`newValues` 输入各个域的值列表（按顺序），转化成 `byte[]`。

3. file.SeqFile 类

`file.SeqFile` 该类半底层，是一个 `record based file` 的实现。

作为记录遍历的基础，简要说明如下：

首先是若干 `bytes` 域作为文件的 `header`（这里选取 512bytes，方便扩展），最重要的信息有两个，第一条记录的位置和第一个空洞前一条记录的位置。这里有点特殊，记录也可以是个空洞（空洞不会是个记录）。

整个数据部分形成两个链，一个链是所有的记录，每条记录开头的 4 字节存放：

下一条“真”记录的位置

或

下一个空洞的位置的负值。

通过正负判断下一个是空洞还是真的记录。记录的长度就是读出来这个数的绝对值和当前位置之差。

以上也是空洞链会记录空洞前一条记录的位置的原因，因为在那个位置存有一个负值，表示下面是个空洞，如果该空洞被抹去，前条记录的那个负值就会变正。所有空洞组成的链，在所有空洞记录的第 4~8 字节存放：下一个空洞前面那条真实记录的位置。（过短的空洞会被算在前面那条真实记录的长度里面，`Schema` 类的 `getValue` 等保证多余的数据不会被读出来）空洞链以 0 作为结束标记。记录链则是以超出文件边界（文件长度）作为结束标记。设计时尽力不产生连续的空洞（空洞合并）。

扩展 `java.util.Iterator` 接口，支持 `replace` 和 `replaceForce`（这两个方法都要在调用 `next` 后调用，和 `Iterator` 本身的 `remove` 保持一致，区别是前者仅做原地替换尝试）插入时，遍历空洞链，找到一个合适的空洞，写入数据，可能还有够大空洞，或者没有空洞，维护空洞链，同时也要修正空洞前记录的链节点值。该操作不可能产生比原来更大的空洞。如果找不到合适的空洞，定位到文件末尾写入。

遍历中可以执行 `remove`，删掉刚才得到的那条记录，删除时关注空洞的相对位置，合理的调整前后链指针的值，`replace` 类似进行。

4. db 包

3.1 DataBase 类

该类是一些关于创建、删除表和 `VIEW` 的接口，关于表的存放也在此说明：

一张表有三个虚拟文件，一是 `metadata`，也就是 `Schema` 类，以序列化存储。

二是 `data`，用 `SeqFile` 作为低层，里面所有的 `primitive type` 都是原样存放，如 `INTEGER` 四个字节，`byte[20]` 就是 20bytes。不定长字符串根据 `DataOutput`

接口的 writeUTF 方法存储。

三是索引文件，将 java 的哈希表序列化存储于此。

底层 Disk 分为/sys 和/usr 两个目录，分别存储系统表和用户表。每个目录各分 tbl 和 view 两个文件夹。整个做法很低级。也亏低层的 Disk 实现的 OK。

3.2 HashIndices 类

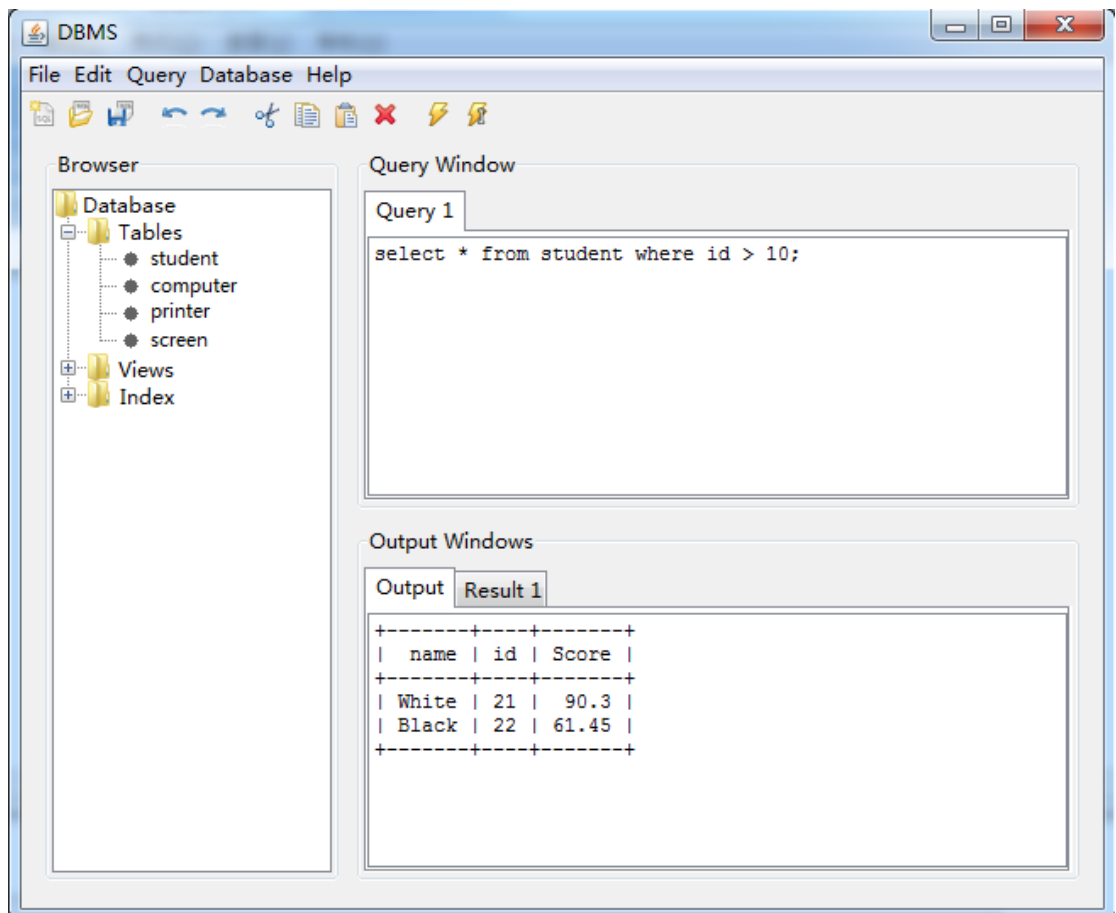
HashIndices 是个简单的索引，DbTable 是基于 SeqFile 的一个对表封装，封装了 SeqFile 和索引修改的功能。



3.3 TableHelper 类

TableHelper 类主体是一堆 sample/功能测试代码，里面的 package private 方法是实际的用于 Table 的帮助函数。

六、用户界面

用户启动程序后，首先需要验证用户密码，然后进入系统主界面，如下图所示：



界面左边的树(JTree)显示这个 Database 中的内容,包括表,视图和索引。在右上方的文本框中输入 SQL 命令,然后点击执行按钮执行文本框中的所有语句,或者按钮执行光标所在行的 SQL 命令,结果在下方的 Output 文本框中显示。

此用户界面支持用户新建,打开,保存 SQL 文件,支持剪切,复制,粘贴,删除文本的编辑命令,支持撤销(UNDO, REDO)等指令。用户界面人性化,使用方便,功能强大。

欢迎您使用!

七、项目总结

经过几个星期三个人的忙碌,我们的 DBMS 数据库管理系统已经基本完成。此次课程设计使我们加深对《数据库原理》课程中学到的基本概念、基本原理和基本技术的理解,使我们能够清晰地明白 SQL 命令的执行流程,数据库的组成模式等,已经完成此次课程设计的目标。

感谢李芳老师的辛勤授课,感谢助教提供的帮助。

小组成员:

张轩 5080309672

阎立 5080309073

朱崎峰 5080309079