数据库设计文档

1.文件结构

src/main/java.cn.edu.thssdb：源代码位置

client包：客户端

server包和service包：服务器

exception包：异常处理

type包：全局类型定义

utils包：全局常量和变量定义

cache包：页式存储管理的缓存系统，以LRU算法作为页面置换算法

index包：B+树和索引

parser包：SQL解析器，包括自带的g4定义，词法分析，句法分析，还有自己实现的语义分析

query包：查找逻辑，查找表和查找结果处理

schema包：元数据管理

src/test/java.cn.edu.thssdb：单元测试代码位置

index包：B+树和索引相关测试

metadata包：元数据管理相关测试

query包：查询相关测试

storage包：存储相关测试

2. 实现功能

基本功能

存储模块

1. 记录的持久化（保存到文件和从文件中恢复记录）
2. 对记录的增删改查
3. 五种数据类型 Int, Long, Float, Double, String
4. 页式存储格式

元数据管理模块

1. 表的创建、删除、修改
2. 数据库的创建、删除、切换
3. 表元数据的持久化，数据库元数据持久化
4. 启动数据库时从文件恢复数据库以及对应的表

查询模块

1. SQL的词法，句法，语义分析
2. Database,Table类中数据库的增加，删除，切换，显示；表的增加，删除，显示；还有数据的增删改查接口
3. 实现了QueryTable，QueryResult类，实现了任意张表的join和distinct关键字
4. 实现了逻辑模块，能够完成（on和where中）多个比较条件的and，or连接，而且实现了sql的null逻辑

事务模块

1. 服务器支持多客户端并发
2. 实现begin transaction和commit。
3. 使用二级锁协议，实现read committed隔离级别。
4. 实现单一事务的WAL机制，可以读写log并恢复数据。
5. 完善数据库存储模块与bug修改。

附加功能

1. 多客户端
2. 页式存储
3. 支持on和where中多个比较条件的and，or连接
4. 支持任意张表的join
5. distinct关键字

3. 功能设计和实现

3.1 存储模块

3.1.1 Table类

因为页式存储较为复杂，涉及到页面的选择、置换等与数据库本身逻辑无关的工作，因此参考了hsqldb的思路，单独实现了管理内存的Cache类，从而将内存的管理从Table类中剥离了；在Table类中，只处理与数据库本身逻辑相关的工作，一切涉及内存的内容通过调用Cache类提供的接口完成。

成员变量

将B+树从Table中移入Cache类中，其他与框架没有变化。

成员函数

- 构造器：参数与框架相同；在构造其中，保存每一列的信息以及主键位置，调用recover方法从文件中恢复持久化的数据。
- 查询记录：`public Row get(Entry entry)` 传入查询的记录的主键作为参数，通过调用Cache类提供的接口进行查询。
- 删除记录：`public void delete(Entry primaryEntry)` 传入待删除的记录的主键作为参数，通过调用Cache类提供的接口进行删除。
- 增加记录：`public void insert(ArrayList<Column> columns, ArrayList<Entry> entries)` 传入待插入的行的列信息和记录信息作为参数，检查每一列和表的schema相同后调用Cache类提供的接口进行增加。
- 修改记录：`public void update(Entry primaryEntry, ArrayList<Column> columns, ArrayList<Entry> entries)` 传入主键、待修改的列及对应的数据作为参数，检查每一个待修改的列位于表的schema中后调用Cache类提供的接口进行修改。
- 恢复数据：`private void recover()` 在默认的文件目录下找到本表格对应的文件并调用Cache类的方法进行恢复。

- 持久化数据：`public void persist()` 直接调用Cache类的方法，将所有页面保存到文件中。

3.1.2 Cache包

Cache包包含Cache类、Page类以及Cache类的内部类EmptyRow类（继承自Row），主要负责内存的管理，包括页面置换等工作。

1) 设计思路

在页式存储的前提下，如何在内存中保存记录，考虑并测试了两种实现方案：

方案一：

将记录(Row)保存在页面(Page类)上，修改框架提供的B+树index的实例化方式，不再从主键Entry映射到Row，而是从Entry映射到保存这条记录的页面编号及页内位置上，每次增删改查操作时，先在B+树种查询位置，再到对应的Page上完成相应操作。

方案二：

保留框架提供的B+树index的实例化方式，在内存中仍然把Row作为B+树的Value保存；在页面上只存储记录的主键和记录的长度信息，每次查询直接在index中完成，需要页面置换、保存、恢复时再通过Page中记录的主键到index中获取对应的Row并进行保存/恢复。

比较和选择

两种方案各有优缺点。对于方案一，优点是除了删除操作，不需要对B+树进行大的更新，同时页面置换也与B+树无关，因此在测试中对于遍历数据库2000条记录的操作，这种方案平均用时比方案二少200ms；缺点在于一旦涉及到删除，需要对同属一个页面的其他记录也更新位置信息，一旦实际使用的页面较大（测试时为了测试页面置换的正确性和效率采用了256Byte的很小的页面大小），同时更新的记录会很多，而且对于其他操作，获取记录都需要两部，即先获取位置再访问页面，对单条记录的访问效率较低。对于方案二，优点是对于较大的页面比较友好，查询、修改、增加记录都没有额外操作；但是对于页面置换来说，需要修改整个页面对应的所有记录。

综上，考虑到实际实现中页面大小远大于测试使用的大小，同时从访问记录的用时上考虑，最终选择了方案二，即保留框架中index的实例化方式，在B+树中存储记录的方案。

2) 实现方式

Cache类

```
public class Cache {
    private static final int maxPageNum = 100;
    private HashMap<Integer, Page> pages;
    private int pageNum;
    private BPlusTree<Entry, Row> index;
    private String cacheName;

    // 公有接口
    public Cache(String databaseName, String tableName);
    public Iterator<Pair<Entry, Row>> getIndexIter();
    public boolean insertPage(ArrayList<Row> rows, int primaryKey);
    public void insertRow(ArrayList<Entry> entries, int primaryKey);
    public void deleteRow(Entry entry, int primaryKey);
    public void updateRow(Entry primaryEntry, int primaryKey,
                          int[] targetKeys, ArrayList<Entry> targetEntries);
    public Row getRow(Entry entry, int primaryKey);
    public void persist();

    // 页面选择、置换相关私有接口
```

```
    private void exchangePage(int pageId, int primaryKey);
    private boolean addPage();
    private void expelPage();
    private void serialize(ArrayList<Row> rows, String filename);
    private ArrayList<Row> deserialize(File file);

    // 内部类 EmptyRow
    public class EmptyRow extends Row {
        public EmptyRow(int position)
        {
            super();
            this.position = position;
        }
    }
}
```

Page类

```
public class Page {
    public static final int maxSize = 256;
    private int id;                // unique id
    private int size;              // size of the page
    private ArrayList<Entry> entries;  // primary keys
    private String pageFileName;  // filename on the disk
    private long timeStamp;        // timestamp of last visit
    private Boolean edited;        // this page has been edited, need to be write
back to file
}
```

页式存储管理方式

在Cache类中保存每个页面编号对应的Page，并记录当前页面数量，当页面数量超过上限时，就开始执行置换：

- expelPage

  遍历当前在内存中的所有页面，根据LRU算法选择时间戳最早的一个进行驱逐。将该页面所对应的记录在index中置为EmptyRow，同时将原来的Row存入文件中。

- readPage

  从文件中读入一个页面对应的行，用主键在index中更新记录，并在Cache类中恢复该页的状态。

- exchangePage

  遇到访问的记录在index中是EmptyRow的情况时，先选择一个页面驱逐，再读入对应的页面对该记录（及所有该页面上的记录）进行恢复。

3.2 元数据管理模块

3.2.1 Database类

主要成员变量

- `tables` 用哈希表以表的名字为键存储数据库对应的所有表
- `name` 数据库本身的名字

主要成员函数

公有方法

- `public void create(String name, Column[] columns)` 传入表的名字和表头信息，创建一张新表；如果名字已存在，会抛出`DuplicateTableException`异常。
- `public Table get(String name)` 根据表的名字获取一张表；如果名字不存在，会抛出`TableNotExistException`异常。
- `public void drop(String name)` 根据表的名字删除一张表，包括在数据库中的记录、内存中的信息以及外存中的文件；如果名字不存在，会抛出`TableNotExistException`异常。
- `public void dropSelf()` 删除自身这个数据库，包括其中的每一张表和数据库的元数据文件。
- `public void quit()` 退出系统时调用，对数据库的元数据以及每一张表进行持久化。

私有方法

- `private void persist()` 持久化方法，将数据库的元数据保存在以meta_开头的外存文件中。
- `private void recover()` 恢复方法，从外存中读取对应该数据库的元数据文件，并根据内容创建相应的表。

3.2.2 Manager类

主要成员变量

- `databases` 使用哈希表以数据库的名字为键存储所有的数据库
- `currentDB` 当前数据库

主要成员函数

公有方法

- `public static Manager getInstance()` 静态方法获得全局唯一的Manager实例对象。
- `public void createDatabaseIfNotExists(String databaseName)` 根据数据库名创建数据库，供外部调用以及自身的`recover`方法调用。
- `public Database get(String databaseName)` 根据数据库名获取数据库对象；如果名字不存在，会抛出`DatabaseNotExistException`异常。
- `public void deleteDatabase(String databaseName)` 根据数据库名删除数据库对象；如果名字不存在，会抛出`DatabaseNotExistException`异常。
- `public void switchDatabase(String databaseName)` 根据数据库名切换当前数据库；如果名字不存在，会抛出`DatabaseNotExistException`异常。
- `public void quit()` 退出系统，会自动调用`persist()`保存manager的元数据到外存文件，并调用所有database的`quit()`方法进行退出和保存。

私有方法

- `private void persist()` 持久化，将manager的元数据，包括数据库名等数据保存到外存文件。
- `private void recover()` 从文件中恢复manager的元数据，并调用`createDatabaseIfNotExists`创建所有的表。

3.3 查询模块

3.3.1 逻辑模块

逻辑模块在实现删除，更新，查询的where语句，以及多表连接的on语句中都很重要。

一个逻辑表达式（multiple_condition），实际上是0个到多个条件表达式（condition）连接成的，SQL的语法树是递归定义这种连接的

```
multiple_condition :
    condition
    | multiple_condition AND multiple_condition
    | multiple_condition OR multiple_condition ;
```

而一个condition对象，则是由两个运算表达式（expression）连接成的，SQL语法树实现如下：

```
condition :
    expression comparator expression;
```

但是我并没有实现运算表达式，运算表达式我直接使用一个元素（comparer）来替代了。

```
expression :
    comparer
```

因此，我定义了三个对应的类：

Logic类：一个完整或者部分完整的逻辑，它可能是只有一个condition的，也可能由左右两个逻辑连接而成。

```
    public Logic(Logic left, Logic right, LogicType type) {
        this.mTerminal = false;
        this.mLeft = left;
        this.mRight = right;
        this.mType = type;
    }

    public Logic(Condition condition) {
        this.mTerminal = true;
        this.mCondition = condition;
    }
```

Condition类：一个比较表达式，由左右两个comparer连接而成。

```
    public Condition(Comparer left, Comparer right, ConditionType type) {
        this.mLeft = left;
        this.mRight = right;
        this.mType = type;
    }
```

Comparer类：一个值包装器，只有值和类型。

```
    public Comparer(ComparerType type, String value) {
        this.mType = type;
        switch (type) {
            case NUMBER:
                this.mValue = Double.parseDouble(value);
                break;
            case STRING:
            case COLUMN:
```

```
                this.mValue = value;
                break;
            default:
                this.mValue = null;
        }
    }
```

通过Logic类的GetResult方法，我实现了一个逻辑表达式对一个广义数据行（JointRow）（一个行或者多个行连接而成的行）的逻辑判定。在Logic类里，我进行递归判定，用左右子逻辑/自己唯一的条件表达式的值求得当前逻辑的值；在Condition类里，我先比较左右两个比较器的类型，如果有COLUMN类型，就去数据行里读取对应的值，并且更新类型。之后按照比较器类型进行比较和异常处理。值得一提的是，我在这里实现了sql的null逻辑：null和谁比较都是unknown，unknown的逻辑运算我也实现了。

3.3.2 查找表和结果获取

我实现了QueryTable类和其两个子类：SingleTable和JointTable。在框架的基础上，我们实现了hasNext()函数，next()函数和PrepareNext()函数，用于找到下一个符合条件的元素。其中，PrepareNext()函数是核心。

对于单一查找表，我分成两种情况讨论：首先，如果查询的是主键，而且只查询主键，我就使用index索引直接获取所需元素。否则，我就继续遍历table，找到下一个满足逻辑表达式的元素。

对于复合查找表，我使用类似进位的方法来获取这些表笛卡尔积的下一个元素，代码如下：

```
private JointRow JoinRows() {
        if (mRowsToBeJoined.isEmpty()) {
            for (Iterator<Row> iter : mIterators) {
                if (!iter.hasNext()) {
                    return null;
                }
                mRowsToBeJoined.push(iter.next());
            }
            return new JointRow(mRowsToBeJoined, mTables);
        } else {
            int index;
            for (index = mIterators.size() - 1; index >= 0; index--) {
                //类似加法进位一样的机制：一直重新设置iterator，类似进位，直到有一个
iterator有能进位的为止
                mRowsToBeJoined.pop();
                if (!mIterators.get(index).hasNext()) {
                    mIterators.set(index, mTables.get(index).iterator());
                }
                else {
                    break;
                }
            }
            if (index < 0) {
                return null;
            }
            //再把进位的元素补回去--重设iterator，类似补0
            for (int i = index; i < mIterators.size(); i++) {
                if (!mIterators.get(i).hasNext())
                    return null;
                mRowsToBeJoined.push(mIterators.get(i).next());
            }
            return new JointRow(mRowsToBeJoined, mTables);
```

```
        }
    }
```

通过以上的方法，我能够实现对几个表笛卡尔积的遍历。这样我只需要遍历这些表的笛卡尔积，然后判断是否符合on和where条件即可。

基于框架，我还实现了QueryResult类。这个类的功能是待查的column对查询结果做映射。其中，我利用哈希表实现了distinct关键字。

3.3.3 Parser和语义分析

词法，句法分析部分，我使用框架提供的SQL.g4,SQLLexer,SQLParser实现。语义分析部分，我继承了框架提供的SQLVisitor实现，实现于MyVisitor，通过递归解析语法树，调用Table，Database等类的代码实现。我实现了SQLHandler来调用词法，句法，语义分析，编译器异常处理部分，代码如下：

```
    public String evaluate(String statement) {
        //词法分析
        SQLLexer lexer = new SQLLexer(CharStreams.fromString(statement));
        lexer.removeErrorListeners();
        lexer.addErrorListener(MyErrorListener.instance);
        CommonTokenStream tokens = new CommonTokenStream(lexer);

        //句法分析
        SQLParser parser = new SQLParser(tokens);
        parser.removeErrorListeners();
        parser.addErrorListener(MyErrorListener.instance);

        //语义分析
        try {
            MyVisitor visitor = new MyVisitor(manager);
            return String.valueOf(visitor.visitParse(parser.parse()));
        } catch (Exception e) {
            return "Exception: illegal SQL statement! Error message: " +
e.getMessage();
        }
    }
```

语义分析遇到的主要困难是insert，delete，update，select的类型问题了。在insert中，所有属性值被解析为string类型，我直接使用五大类型对应parser来将其转换成想要的。在update，delete，select中，所有属性值都被解析为comparer类型，因此我只进行了string，null，column的判断，对于数字则是直接强制类型转换成double类型，再强制类型转换成所需的。因此，因为语法树的规定，这些地方我没有做仔细的数字类型判断。

3.4 事务模块

3.4.1 多客户端支持

只需修改一行代码即可，thrift官方实现了支持多客户端的服务器。

```
server = new TThreadPoolServer(new
TThreadPoolServer.Args(transport).processor(processor));
```

3.4.2 事务功能实现

由于给定代码的parser不支持事务的相关语句，因此需要重构cfg文件并使用antlr重新生成相关文件。

在重构parser后，实现begin transaction和commit两个命令主要就是在MyVisitor.java中完成`visitBegin_transaction_stmt`和`visitCommit_stmt`两个方法，同时对普通语句作为一个单元素的事务看待，使用autocommit模式，在提交前后分别插入`autobegin transaction`和`autocommit`，这两个方法也需要实现（和显示方法基本一致）。

为了实现read committed级别的事务隔离，我采用了二级锁协议：

> 1.一级封锁协议
> 一级封锁协议是：事务T在修改数据R之前必须先对其加X锁，直到事束才释放。事务结束包括正常结束（COMMIT）和非正常结束（ROLLBACK）。
> 一级封锁协议可以防止丢失修改，并保证事务T是可恢复的。使用一级封锁协议可以解决丢失修改问题。
> 在一级封锁协议中，如果仅仅是读数据不对其进行修改，是不需要加锁的，它不能保证可重复读和不读“脏”数据。
> 2.二级封锁协议
> 二级封锁协议是：一级封锁协议加上事务T在读取数据R之前必须先对其加S锁，读完后方可释放S锁。
> 二级封锁协议除防止了丢失修改，还可以进一步防止读“脏”数据。但在二级封锁协议中，由于读完数据后即可释放S锁，所以它不能保证可重复读。

这种协议是一种表级别锁协议，在具体实现上，我在table.java中定义了以下数据：

```
int tplock = 0;
public ArrayList<Long> s_lock_list;
public ArrayList<Long> x_lock_list;
```

其中tplock代表当前表的锁级别，可取值0，1，2分别代表无锁，s锁和x锁，`s_lock_list`和`x_lock_list`两个list储存了拥有当前表的s锁和x锁的session列表，其中由于x锁的特性，`x_lock_list`的最大长度为1。

定义了不同锁的获取、释放方法：

```
public int get_s_lock(long session){
  int value = 0;                        //返回-1代表加锁失败  返回0代表成功但未加锁  返回1代表成功加锁
  if(tplock==2){
    if(x_lock_list.contains(session)){   //自身已经有更高级的锁了 用x锁去读，未加锁
      value = 0;
    }else{
      value = -1;                       //别的session占用x锁，未加锁
    }
  }else if(tplock==1){
    if(s_lock_list.contains(session)){    //自身已经有s锁了 用s锁去读，未加锁
      value = 0;
    }else{
      s_lock_list.add(session);          //其他session加了s锁 把自己加上
      tplock = 1;
      value = 1;
    }
  }else if(tplock==0){
    s_lock_list.add(session);              //未加锁 把自己加上
    tplock = 1;
    value = 1;
  }
  return value;
}
```

```
public int get_x_lock(long session){
  int value = 0;                    //返回-1代表加锁失败  返回0代表成功但未加锁  返回1代表成功加锁
  if(tplock==2){
    if(x_lock_list.contains(session)){     //自身已经取得x锁
      value = 0;
    }else{
      value = -1;                       //获取x锁失败
    }
  }else if(tplock==1){
    value = -1;                           //正在被其他s锁占用
  }else if(tplock==0){
    x_lock_list.add(session);
    tplock = 2;
    value = 1;
  }
  return value;
}

public void free_s_lock(long session){
  if(s_lock_list.contains(session))
  {
    s_lock_list.remove(session);
    if(s_lock_list.size()==0){        //之前就没有其他s锁
      tplock = 0;
    }else{
      tplock = 1;                 //还有其他session有s锁
    }
  }
}

public void free_x_lock(long session){
  if(x_lock_list.contains(session))
  {
    tplock = 0;                   //释放x锁必然回归无锁状态
    x_lock_list.remove(session);
  }
}
```

在manager.py中，定义了一下数据结构用于锁管理：

```
public ArrayList<Long> transaction_sessions;    //处于transaction状态的session列表
public ArrayList<Long> session_queue;           //由于锁阻塞的session队列
public HashMap<Long, ArrayList<String>> s_lock_dict;       //记录每个session取得了哪些表的s锁
public HashMap<Long, ArrayList<String>> x_lock_dict;       //记录每个session取得了哪些表的x锁
```

在MyVisitor.py中实现并修改相关接口：

```
//开始transaction
//初始化相关状态和数据结构
```

```
public String
visitBegin_transaction_stmt(SQLParser.Begin_transaction_stmtContext ctx) {
    try{
        if (!manager.transaction_sessions.contains(session)){
            manager.transaction_sessions.add(session);
            ArrayList<String> s_lock_tables = new ArrayList<>();
            ArrayList<String> x_lock_tables = new ArrayList<>();
            manager.s_lock_dict.put(session,s_lock_tables);
            manager.x_lock_dict.put(session,x_lock_tables);
        }else{
            System.out.println("session already in a transaction.");
        }

    }catch (Exception e){
        return e.getMessage();
    }
    return "start transaction";
}
```

```
//commit transaction
//根据记录释放相关表的x锁，清空相关记录并查看log文件大小，超过一定大小后持久化数据并清空log
public String visitCommit_stmt(SQLParser.Commit_stmtContext ctx) {
    try{
        if (manager.transaction_sessions.contains(session)){
            Database the_database = GetCurrentDB();
            String db_name = the_database.get_name();
            manager.transaction_sessions.remove(session);
            ArrayList<String> table_list = manager.x_lock_dict.get(session);
            for (String table_name : table_list) {
                Table the_table = the_database.get(table_name);
                the_table.free_x_lock(session);
                the_table.unpin();
            }
            table_list.clear();
            manager.x_lock_dict.put(session,table_list);

            //查看log文件大小，超过一定大小后持久化数据并清空log
            String log_name = DATA_DIRECTORY + db_name + ".log";
            File file = new File(log_name);
            if(file.exists() && file.isFile() && file.length()>50000)
            {
                System.out.println("Clear database log");
                try
                {
                    FileWriter writer=new FileWriter(log_name);
                    writer.write( "");
                    writer.close();
                } catch (IOException e)
                {
                    e.printStackTrace();
                }
                manager.persistdb(db_name);
            }
        }else{
            System.out.println("session not in a transaction.");
        }
        //System.out.println("sessions: "+manager.transaction_sessions);
```

```
    }catch (Exception e){
        return e.getMessage();
    }
    return "commit transaction";
}
```

`autobegin transaction`和`autocommit`与以上实现类似。

同时修改insert，delete，update，select方法的接口，具体代码不在此处展示，主要思路是对于一个新的transaction，尝试获取锁，如果获取锁失败则加入等待队列，在等待队列开头的事务会定期尝试获取锁，其余事务处于阻塞休眠状态。如果获取相应的锁成功则出队执行操作，如果是select操作，则执行完操作后立刻释放s锁，其余操作等待commit后再释放x锁。

3.4.3 WAL机制实现

wal机制的主要思路：将对表的修改语句记录写入log文件，这样数据就不需要立即持续化，而是定期由某种机制持久化，同时清空log文件；在重新启动数据库时，数据库会根据log的记录进行数据库的恢复，当log中存在某个事务未完成（begin transaction语句和commit语句数目不等）时，执行到开始事务之前的一句，并把log中未完成事务的记录删除，这样就保证了事务的原子性。

相关接口主要在manager.java中实现：

```
//写数据，只在更改数据库时记录，读取数据库时不记录
public void writelog(String statement)
{
    Database current_base = getCurrent();
    String database_name = current_base.get_name();
    String filename = DATA_DIRECTORY + database_name + ".log";
    try
    {
        FileWriter writer=new FileWriter(filename,true);
        System.out.println(statement);
        writer.write(statement + "\n");
        writer.close();
    } catch (IOException e)
    {
            e.printStackTrace();
    }
```

```
//读取log并恢复数据库
//如果遇到执行不完整的transaction，则执行到事务开启之前的语句，不完整的事务语句清除。
public void readlog(String database_name)
{
    String log_name = DATA_DIRECTORY + database_name + ".log";
    File file = new File(log_name);
    if(file.exists() && file.isFile()) {
        System.out.println("log file size: " + file.length() + " Byte");
        System.out.println("Read WAL log to recover database.");
        evaluate("use " + database_name);

        try {
            InputStreamReader reader = new InputStreamReader(new
FileInputStream(file));
            BufferedReader bufferedReader = new BufferedReader(reader);
```

```
            String line;
            ArrayList<String> lines = new ArrayList<>();
            ArrayList<Integer> transcation_list = new ArrayList<>();
            ArrayList<Integer> commit_list = new ArrayList<>();
            int index = 0;
            while ((line = bufferedReader.readLine()) != null) {
                if (line.equals("begin transaction")) {
                    transcation_list.add(index);
                } else if (line.equals("commit")) {
                    commit_list.add(index);
                }
                lines.add(line);
                index++;
            }
            int last_cmd = 0;
            if (transcation_list.size() == commit_list.size()) {
                last_cmd = lines.size() - 1;
            } else {
                last_cmd = transcation_list.get(transcation_list.size() - 1) -
1;
            }
            for (int i = 0; i <= last_cmd; i++) {
                evaluate(lines.get(i));
            }
            System.out.println("read " + (last_cmd + 1) + " lines");
            reader.close();
            bufferedReader.close();

            //清空log并重写实际执行部分
            if (transcation_list.size() != commit_list.size()) {
                FileWriter writer1 = new FileWriter(log_name);
                writer1.write("");
                writer1.close();
                FileWriter writer2 = new FileWriter(log_name, true);
                for (int i = 0; i <= last_cmd; i++) {
                    writer2.write(lines.get(i) + "\n");
                }
                writer2.close();
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

通过这样的机制，即可实现数据库的恢复与事务的原子性保证。

4.测试情况

在test/实现了单元测试，可通过 `mvn test` 执行。

4.1 存储模块

增加（访问）测试

插入了2000条记录，随机抽取若干记录访问并测试了记录内容的正确性。对不存在的主键进行了访问的测试，会抛出KeyNotExisted异常并提示不存在的主键名。

遍历测试

遍历所有2000条记录，测试了遍历的顺序是有序的，同时记录的内容是正确的。

更新测试

随机抽取了若干记录进行更新，测试了包括更新主键、不更新主键的情况；以及更新后的主键已经存在等情况，对于后者，会抛出DuplicatedKey异常并提示重复的主键名。

删除测试

随机抽取了几个主键删除，测试包括了待删除的记录的主键存在和不存在的情况；前者能够正确删除，后者会抛出KeyNotExisted异常并提示不存在的主键名。

持久化/恢复测试

经过上述增删改查后，对表进行持久化，然后从文件中恢复。测试访问了经过上述测试后某些被修改的记录、被删除的记录、以及额外增加的记录，内容均正确。

4.2 元数据管理模块

初始化

创建了一个管理者Manager类的实例对象，在manager中创建了三个数据库University, HighSchool, MiddleSchool，每个数据库中创建了Student和Teacher两张表。

添加记录

向每一张表中添加2000条记录，并随机访问若干条记录检查正确性。

删除

- 删除数据库

  删除MiddleSchool数据库，检查是否还能访问已删除内容

- 删除表

  删除University数据库中的Teacher表，检查是否还能访问已删除内容

- 删除记录

  删除University数据库中的Student表的1000条记录，检查是否还能访问已删除内容

持久化和恢复

先调用manager的 `quit()` 方法退出系统，将所有信息保存在外存，再进行恢复。检查在删除测试中删除的内容是否能访问，以及未删除的内容是否正确。

4.3 查询模块

逻辑测试

位置在test中的query/QueryTest类/Logictest函数

我手工构造了13条逻辑，包括空，单一逻辑，复合逻辑等多种情况，用他们判断一些列是否满足这些逻辑。

SQL测试

位置在client中的ClientTest类,直接运行即可。

我直接构造了若干条sql语句，对应作业需求的数据库，表，数据的各种操作，用于检测这些操作是否正确。

4.4 事务模块

由于事务模块主要需要人为观察数据库运行结果，并没有编写自动化单元测试代码，下面是我手动进行的一些操作：

单事务测试

测试了begin transaction和commit命令，在其中执行多个语句，发现结果运行正常，处于事务状态时，客户端会显示处于(T模式)。

多事务并发测试

开启多个客户端事务，具有以下特点：

1. 当一个事务只执行读取操作时，并不影响其余事务对table的读取。
2. 当一个事务执行了对table的修改但未commit，此时其余事务的读写操作都无法获取锁，被阻塞等待。当commit相关操作后，会按照阻塞事务的操作加入时间选择下一个执行的事务。
3. 结果表示，相关操作解决了脏读问题，但不能保证两次读取结果一致，实现了read committed级别的隔离。

单事务WAL恢复

关闭数据库，发现此时储存的二级制文件并没有改变，重启数据库，数据库会根据log文件执行相关操作恢复数据库，查看相关信息发现数据库恢复无误。

当完成一个transaction后且log文件大小大于5MB时，会自动清空log文件且对数据进行持久化。

事务原子性

在事务执行一半未commit时，关闭数据库程序以模仿掉电故障，重启数据库查看数据，可发现数据库已回退到未执行事务之前的状态。