数据库大作业文档

- 组员信息：2018013361余齐齐, 2018013374周彦如, 2018013405彭维方。

实现功能概览

基础功能：

通信模块：

1. 实现thrift/rpc.thrift 中的connect、disconnect和executeStatement 服务。

异常处理模块：

1. SQL解析器在调用语句时透过 try catch 捕获操作时可能出现的错误并返回错误信息给客户端。

存储模块：

1. 利用序列化和反序列化实现记录的持久化。和元数据模块的恢复和持久化有密切关系。
2. 实现对记录的增加、删除、修改、查询。
3. 支持五种数据类型：Int，Long，Float，Double，String。

元数据管理模块：

1. 实现数据库的创建、删除、切换和实现表的创建、删除
2. 实现表和数据库的元数据的持久化。
3. 重启数据库时从持久化的元数据中恢复系统信息。

查询模块：

1. 支持文档中提到的`create`、`drop`、`show`、`insert`、`delete`、`update`、`select`等语句
2. `where`条件支持逻辑运算符（`and`/`or`）
3. 支持多表的`Join`运算

事务模块：

1. 采用普通锁协议(共享锁和排他锁)，实现read committed的隔离级别
2. 实现单一事务的WAL机制，能写log和读log，在重启时能够恢复记录的数据

附加功能：

查询模块：

1. 实现多表JOIN
2. where条件支持逻辑运算符（and/or）
3. 其他标准SQL支持的查询语法如： *，distinct.

事务模块：

1. 实现begin transaction 和 commit;
2. 对单一语句实现 auto begin transaction 和 auto commit

提交作业结构

`Project`根目录下有代码文件夹`ThssDB`和文档文件夹`doc`。其中`ThssDB`文件夹有下面结构：

```
| --ThssDB
    |--src
    |--Data
```

`src`里面放有项目代码，实现的包有：

1. client: 客户端代码
2. exception: 异常处理模块
3. helper: 存储模块辅助类
4. index: 索引模块
5. parser: SQL解析模块
6. query: 查询模块
7. schema: 元数据模块
8. server: 服务器模块
9. service: 服务模块
10. type: 类型管理模块
11. utils: 常量和变量定义

功能实现说明

1. 存储模块

1. 利用序列化和反序列化实现记录的持久化。和元数据模块的恢复和持久化有密切关系。
2. 实现对记录的增加、删除、修改、查询。
3. 支持五种数据类型：Int，Long，Float，Double，String。

存储的操作主要实现在`schema.Table`类里，其成员变量包含：

```
public class Table implements Iterable<Row> {
  ReentrantReadWriteLock lock;
  public String databaseName;
  public String tableName;
  public ArrayList<Column> columns;
  // 列名列表
  private ArrayList<String> columnsName;
  public int schemaLength;
  public BPlusTree<Entry, Row> index;
  private int primaryIndex;
  public ArrayList<Entry> entries;
}
```

其中增加的`columnsName`用于在 update 和 insert 时快速定位对应操作列、检查列存在和检查是否指定重复列等。

其成员函数包含：

```
public Row getRow(Entry entry) // 传入欲查询记录主 entry，返回对应的一行记录

// insert 相关
public void insertEntries(ArrayList<Entry> entryList)
// 给定一个按序创建好的 ArrayList<Entry>，实际将数据插入表中，包含插入 index 和插入主键记录中

public void insert(String[] values)
```

```
// 给定一个 String 列表，构建一个 ArrayList<Entry> 之后传给 insert(ArrayList<Entry>
entry_list) 真正进行插入

public void insert(ArrayList<String> columnsName, String[] values)
// 给定一个 Column 列表和 String 列表，依 columns 和 values 的对应位置构建一个
ArrayList<Entry>
//之后传给 insert(ArrayList<Entry> entry_list) 真正进行插入

// delete 相关
public void deleteEntry(Entry entry)
// 提供待删除记录的主 entry，将对应记录自 index 中删除，并在主键记录中删除 entry
public String delete(MultipleCondition conditions)
// 遍历表中的 Row，针对每一行数据进行逻辑判断（是否符合删除的条件）决定是否进行删除行操作

// update 相关
public String update(String columnName, Comparer comparer, MultipleCondition
conditions)
// 将满足条件表达式的行的相应属性更新为相应的值

// 持久化相关
private void serialize(ArrayList<Row> rows,String fileName)
// 将表中数据序列化并将记录写入磁盘文件 DATA/<databaseName>_<tableName>.data 中
public void persist()
// 在接收到用户的 quit 指令后被调用，构建行数据传入 serialize 实现数据持久化

// 恢复相关
private ArrayList<Row> deserialize(File file)
// 读取文件 file 并反序列化重建表中数据
private void recover()
// 在表的构造函数中调用此函数，此函数会进一步调用deserialize回复表中数据

// 事务锁相关
// 返回-1表示加锁失败，返回0表示可以执行但没加锁，返回1表示加锁成功
public int getSLock(Long session)
public int getXLock(Long session)
public void freeSLock(Long session)
public void freeXLock(Long session)

// 展示数据
public String showMeta()
```

在本实现中真正对索引和主键记录的插入和删除的操作函数为 `insertEntries` 和 `deleteEntry`，其中 `insertEntries` 接受已经按列排好的行数据并对 `index` 和 `entries` 分别进行插入。因此外层类调用 `Table` 插入的时候需要判断指定要插入的列是否为空。如果是则按序生成将 values 字符串值利用辅助类 `ValueParser` 生成对应值及 entry 依次利用`ValueParser.checkValid` 检查合法性并插入按列排的行数据，最后调用 `insertEntries` 真正插入表中；如果不是的话则遍历表的列，根据传入的列表和值建立有序 entries。注意对于少于表列的项都默认插入的为 `null` 值，同样需要经过 `ValueParser.checkValid(column, value)` 的合法性检测。
在进行更新操作时需要注意改变的列是否为主键。如果是的话则将根据旧行和新值构建的新行先调用插入函数进行插入，此时会有是否产生重复主键的判断，如果有则抛出异常，没有则继续执行删除操作。如果更新的列不是主键，则直接更新索引中的记录即可。

本次实现中为存储模块设计了一个辅助类 `helper.ValueParser`，其函数包含：

```
public Comparable getValue(Column column, String string){
```

```
        try {
            switch (column.getType()) {
                case INT:
                    return Integer.parseInt(string);
                case LONG:
                    return Long.parseLong(string);
                case FLOAT:
                    return Float.parseFloat(string);
                case DOUBLE:
                    return Double.parseDouble(string);
                case STRING:
                    if(string!=null && string.indexOf('\'') != 0 &&
string.lastIndexOf('\'') != string.length() - 1){
                        throw new ValueTypeMismatchException(column);
                    }
                    return string;
                default:
                    return null;
            }
        } catch (Exception e){
            throw new ValueTypeMismatchException(column);
        }
    }
```

由于 SQL 解析器传入的操作 values 皆为 string 类型，因此需要传入列定义和操作 values 的字符串形式在此函数中进行转换和判断。本次实现利用是否可以`Integer.parseInt(string)`等数值和字符串是否能够转换成功来判断待操作值和列类型定义是否相符合。注意的是因为 String 必定转换成功，但是可能不含字符串识别单引号，因此需要判断头尾是否为单引号来确定传入的是否确实为 String 类型。

```
public void checkValid(Column column, Comparable value){
        // 检查非 null 限制
        if(column.isNotNull() && value == null){
            throw new NullValueException(column.getName());
        }
        // 检查最大长度限制
        if(value != null && column.getType() == ColumnType.STRING ){
            int maxLength = column.getMaxLength();
            int valLength = ((String)value).length();
            if(maxLength > -1 && valLength > maxLength + 2){
                throw new ValueExceedMaxLengthException(column.getName(),
maxLength, valLength);
            }
        }
    }
```

在进行完操作值转换后，还需要根据列定义检查值是否符合非 null 限制和对于 String 类型的最大长度限制进行合法性判断。

```
public Comparable compararToComparable(Column column, Comparer comparer) throws
ValueTypeMismatchException{
        // null
        if(comparer == null || comparer.getValue() == null || comparer.getType()
== ComparerType.NULL){
            return null;
        } else if (comparer.getType() == ComparerType.NUMERIC){
            // num
```

```
            String valueString = comparer.getValue().toString();
            try{
                switch (column.getType()){
                    case INT:
                        return Integer.parseInt(valueString.substring(0,
valueString.indexOf('.')));
                    case LONG:
                        return Long.parseLong(valueString.substring(0,
valueString.indexOf('.')));
                    case FLOAT:
                        return Float.parseFloat(valueString);
                    case DOUBLE:
                        return Double.parseDouble(valueString);
                    default:
                }
            }catch (Exception e){
                throw new ValueTypeMismatchException(column);
            }

        }else if (comparer.getType() == ComparerType.STRING && column.getType()
== ColumnType.STRING){
            // string
            String string = comparer.getValue().toString();
            if(string.indexOf('\'') == 0 && string.lastIndexOf('\'') ==
string.length() - 1){
                return string;
            }
        }

        // 如果不属于上面任何一种情况，表示类型有错误
        throw new ValueTypeMismatchException(column);
    }
```

由于条件判断的参数类型为 `Comparer`，而列定义为 `Comparable`，因此需要根据两者的类型进行转换判断。转换成功之后同样需要调用 `checkValid(column, value)` 进行值合法性判断。

2. 元数据管理模块

实现功能

1. 在 Manager 中实现数据库 Database 的创建、删除、切换，在 Database 中实现表 Table 的创建、删除。
2. 实现表和数据库的元数据的持久化。
3. 重启数据库时从持久化的元数据中恢复系统信息。

关系分析

- `Manager` 管理 `Database`，`Database` 管理 `Table`

数据库的创建、删除与切换

`Manager` 类的重要成员变量：

```
private HashMap<String, Database> databases;
private Database currentDatabase;
```

- 创建
  `public void createDatabaseIfNotExists(String databaseName)`
  - 参数说明：`databaseName`为要创建的数据库的名称
  - 功能实现：先判断`databases`中是否已含有该数据库，若没有才调用`Database`的构造函数创建新对象并加入`databases`，同时判断`currentDatabase`是否为空，为空则指向新创建的数据库
- 删除
  `public void dropDatabase(String databaseName)`
  - 参数说明：`databaseName`为要删除的数据库的名称
  - 功能实现：同理，先判断数据库是否存在，若不存在则抛异常，否则根据名称获取`database`对象并调用其`dropSelf`函数，然后将其置为`null`，并从`databases`中移除

  `Database`中：`public void dropSelf()`
  - 功能实现：删除`database`对应的文件，并遍历`tables`中所有的`table`，逐个调用`dropTable`函数，最后将`tables`置为`null`
- 切换
  `public void switchDatabase(String databaseName)`
  - 参数说明：`databaseName`为要切换到的数据库的名称
  - 功能实现：同理，先判断数据库是否存在，若不存在则抛异常，否则将`currentDatabase`指向根据名称获取的`database`对象

表的创建与删除

`Database`类的重要成员变量：

```
private String name;
private HashMap<String, Table> tables;
```

- 创建
  `public void createTableIfNotExists(String tableName, Column[] tableColumns)`
  - 参数说明：`tableName`和`tableColumns`分别为要创建的表的名称和属性数组
  - 功能实现：先判断`tables`中是否已含有该名称的表，若没有才调用`Table`的构造函数，并将新创建的表加入`tables`
- 删除
  `public void dropTable(String tableName)`
  - 参数说明：`tableName`为要删除的表的名称
  - 功能实现：同理，先判断表是否存在，若不存在则抛异常，否则根据名称获取`table`对象并调用其`drop`函数，然后将其置为`null`，并从`tables`中移除

表和数据库的元数据的持久化和恢复

- 关于manager/database/table中recover和persist的关系
  - manager的数据存储在`manager.data`，用于存储manager中所有database的database_name;
  - database的数据存储在多个`meta_databaseName_tableName.data`文件中，用于存储database中table的metaInfo。
  - table的数据存储在`databaseName_tableName.data`文件中，用于存储table中的rowInfo。

- 关于recover时要做的判断
  manager recover时要创建table,database recover时要创建table,table创建时要对文件中的row和schema比较，`assert row.entries.size == columns.size`。
- 关于收到用户quit指令时的操作
  在quit函数中会调用`persist`函数来持久化元数据，在构造函数中调用`recover`函数来重建对象结构，恢复元数据。

3. 查询模块

实现功能

1. 支持文档中提到的`create`、`drop`、`show`、`insert`、`delete`、`update`、`select`等语句
2. `where`条件支持逻辑运算符（`and`/`or`）
3. 支持多表的`Join`运算
4. 其他标准SQL支持的查询语法： *，distinct.

SQL语句解析和执行部分

主体思路

- `Client`：客户端读取输入的内容（`String`形式），以此构造`ExecuteStatementReq`类，发送给服务器并将收到的`ExecuteStatementResp`进行解析，如果为查询语句则将结果中的`columnList`和`RowList`进行展示。
- `IserviceHandler`：服务器处理时先检查客户端的`session`是否处于连接状态，然后用分号将输入内容分隔，对每条指令做处理后利用`SQLLexer`进行词法分析，`SQLParser`进行句法分析，并在`SQLBaseVisitor`类的基础上重写了`Visitor`类以进行语义分析，将`visitParse`返回的值作为`QueryResult`数组，最后加入到响应并返回给客户端。

关键类的设计

- `StatementVisitor`类：继承`SQLBaseVisitor`类，重写了其中的各类visit statement函数，在`visitParse`函数中调用`visitSql_stmt_list`函数，其中遍历`ctx.sql_stmt()`中的每个`subCtx`，并对其调用`visitSql_stmt`函数，来对语句类型分类并调用相应的visit函数；在各类具体的visit函数中利用当前的`manager`调用`Database`/`Table`中的相应接口。
- 增加 `QueryResult` 参数为 `String` 的构造函数，使得`QueryResult`除了可以返回查询信息之外也能用来方便地返回执行结果信息。

各类语句的执行逻辑

- 各种 create, drop 和 show：
  根据需要创建的对象（数据库和表）调用`Manager`类中的相应接口，同时由于不区分大小写，在遍历语法树时会将数据库名、表名和列名都转换成小写（于`StatementVisitor`的成员函数`visitDatabase_name`, `visitTable_name`, `visitColumn_name`中进行转换）。在创建表时，需要解析`ctx`中的列定义（包含列名、数据类型和最大长度限制、主键和非空限制等）去生成对应的列`columns`并和`tableName`一起作为参数传进`Database.createTableIfNotExists(tableName, columns)`来新建一张表。
- insert:
  从`manager`中获取当前`database`，从`ctx`中获取要更新的表名`tableName`并生成插入值的字符串列表`values`，根据是否有限定列（即`ctx.column_name()`是否为空）判断是否需要解析出插入列名`columnsName`并和`values`一起传入通过`tableName`查找到的相应`table`的`insert`函数。
- delete:
  从`manager`中获取当前`database`，从`ctx`中获取要更新的表名`tableName`，并且根据是否有

`where`语句以及`where`语句后所接的条件表达式`conditions`调用通过`tableName`查找到的相应`table`的`delete`函数。

- update：
  从`manager`中获取当前`database`，从`ctx`中获取要更新的表名`tableName`，要更新的属性名`columnName`，更新后的相应类型及值`comparer`，以及`where`语句后所接的条件表达式`conditions`，并将这些参数传入`Database`的`update`函数，进而通过`tableName`查找到相应`table`，调用`Table`中的`update`函数接口
- select：
  从`manager`中获取当前`database`，从`ctx`中获取是否有`distinct`关键词，所有要选择的属性名`columnNames`，通过`join`连接的表`queryTable`，以及`where`语句后所接的条件表达式`conditions`，并将这些参数传入`Database`的`select`函数，
- 异常处理
  由于在`StatementVisitor`中，对于任何操作我们基本上都是使用 try - catch 结构，同时我们重载了`QueryResult`对的构造函数，让`QueryResult`可以打印捕获异常的字符串信息返回给客户端。

逻辑条件部分

`MultipleCondition`类为对`SQL.g4`中的`Where/on`语法中的`multipleCondition`的实现，负责底层的逻辑验证。
`QueryTable`类利用`MultipleCondition`中提供的`JudgeMultipleCondition`接口，来选择符合条件的行。
`QueryResult`从`QueryTable`中拿到符合条件的行，从中解析出选择的列。

- `MultipleCondition`相关类：
  `SQL.g4`中选择逻辑部分的语法如下：

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

condition :
    expression comparator expression;

expression :
    comparer
    | expression ( MUL | DIV ) expression
    | expression ( ADD | SUB ) expression
    | '(' expression ')';

comparer :
    column_full_name
    | literal_value ;

literal_value :
    NUMERIC_LITERAL
    | STRING_LITERAL
    | K_NULL ;

column_full_name:
    ( table_name '.' )? column_name ;

comparator :
    EQ | NE | LE | GE | LT | GT ;
```

据此设计`MultipleCondition,Condition,Comparer`类（认为expression等价为comparer）。`MultipleCondition`和`Condition`分别对外提供`JudgeMultipleCondition`、`JudgeCondition`接口来验证条件。在`MultipleCondition`中实现了AND/OR两种连接符。

- `QueryTable`相关类：
  将`QueryTable`设为抽象类，拥有`next,hasNext`等接口来查询符合条件的行。
  `QueryTable`由`SingleTable`和`JointTable`两个类拓展。分别实现单个表的行的选择和多个表的选择。这两个类都需要补全父类的`addNext`方法，来获得下一个可选的行。
  - 对于`SingleTable`，实现相对较为简单，只需要针对单个Table来选择符合条件的行。同时对无条件/条件为常量比较的情况都做了优化，其余的都利用了`MultipleCondition`相关类。
  - 对于`JointTable`,作为多个表的补全，为实现多个表的笛卡尔积，采用类似进位的方法保证通过迭代器能够实现每种可能的笛卡尔积都被遍历到。

  同时针对`QueryTable`相关类，还设计了`QueryRow`类——作为Row类的补充，增加了`MetaInfoList`来实现`Join`后的row中多个`Table`与`Columns`的对应。
- `QueryResult`类：
  `QueryResult`类利用`QueryTable`类筛选出的行，再筛选一遍用户选择的列，生成查询结果。
  `QuertResult`有两种构造函数，一种传递正常的搜索结果，一种传递错误消息，实现代码的一致性。

4. 事务模块

实现功能

1. 实现begin transaction 和 commit;
2. 对单一语句实现 auto begin transaction 和 auto commit
3. 采用普通锁协议(共享锁和排他锁)，实现read committed的隔离级别
4. 实现单一事务的WAL机制，能写log和读log，在重启时能够恢复记录的数据

主要设计

- 增加g4文法支持：
  添加相应指令及关键字
- 通过thrift代码提供对多客户端的支持：
  查询资料发现，需要在ThssDB.java的setUp函数中修改server的创建代码，将server由`TSimpleServer`类型修改为`TThreadPoolServer`类
- Manager类中新增成员变量：
  - 两个`list`：`transactionSessions`记录处于事务状态的所有session，`blockedSessions`记录处于等待获得锁的状态的所有session
  - 两个`hashmap`：`sLockDict`记录所有session和它加上s锁的所有table，`xLockDict`记录所有session和它加上x锁的所有table
  - 两个函数：`writeLog`和`readLog`函数
    - writeLog: 获取当前的数据库，根据相应命名获取.log记录文件并将指令追加写入
    - readLog: 获取.log记录文件, 扫描一遍命令，重新执行除在transaction中但未被commit之外的所有指令，同时相应修改.log文件
- `Table`类中新增成员变量和函数：
  - 变量lockLevel: 表示table目前的加锁级别，0表示未加任何锁，1表示有加共享锁，2表示有加排他锁

- 两个`list`：`sLockSessions`记录当前对该表加s锁的所有session，`xLockSessions`记录当前对该表加x锁的所有session
- 四个函数：加s锁、加x锁、释放s锁、释放x锁 (下图代码以s锁为例)

```
public int getSLock(Long session) {
// 返回-1表示加锁失败 返回0表示可以执行但没加锁 返回1表示加锁成功
    if (lockLevel == 0) {
      // 直接加
      sLockSessions.add(session);
      lockLevel = 1;
      return 1;
    }
    else if (lockLevel == 1) {
      if (sLockSessions.contains(session)) {
        // 已有s锁
        return 0;
      }
      else {
        sLockSessions.add(session);
        return 1;
      }
    }
    else {
      // 此时lockLevel == 2
      if (xLockSessions.contains(session)) {
        // 自身已有x锁，不必加
        return 0;
      } else {
        // 被别的锁占用，不能加
        return -1;
      }
    }
}

public void freeSLock(Long session) {
      if (sLockSessions.contains(session)) {
        sLockSessions.remove(session);
        if (sLockSessions.size() == 0) {
          lockLevel = 0;
        }
      }
}
```

- `StatementVisitor`中，在`select/insert/update/delete`及`transaction`相关语句的visit函数做了事务处理：
  - begin transaction中对Manager中的相关变量进行初始化，commit中进行相应清除同时判断.log文件大小，超过一定限制则清空并写入.data文件
  - 以insert为例，首先通过`manager.transactionSessions.contains(session)`判断session是否处于事务状态，如果不在则正常执行，如果在则对该session引入一个while循环，等能跳出循环后再正常执行：

```
while (true) {
    if (!manager.blockedSessions.contains(session)) {
        // 尚未被阻塞，看看能不能加x锁
```

```
            // 能加锁/成功，跳出循环正常执行
            int result = table.getXLock(session);
            if (result != -1) {
                if (result == 1) {
                    // 能加锁
                    ArrayList<String> tmp = manager.xLockDict.get(session);
                    tmp.add(tableName);
                    manager.xLockDict.put(session,tmp);
                }
                // 移出等待队列
                manager.blockedSessions.remove(session);
                break;
            }
            // 不能则加入等待队列
            manager.blockedSessions.add(session);
        }
        else if (manager.blockedSessions.get(0).equals(session)) {
            // 若该session排在阻塞队列的第一位，试图加锁
            int result = table.getXLock(session);
            if (result != -1)
            {
                if (result == 1)
                {
                    // 成功加锁则更新manager中的列表
                    ArrayList<String> tmp = manager.xLockDict.get(session);
                    tmp.add(tableName);
                    manager.xLockDict.put(session,tmp);
                }
                // 能成功执行则从阻塞队列移除
                manager.blockedSessions.remove(0);
                break;
            }
        }
        // 因为阻塞，所以休眠一会   -- 繁忙等待
        try {
            Thread.sleep(300);
        }
        catch(Exception e) {
            System.out.println(e.getMessage());
        }
    }
```

- `IserviceHandler` 类中的 `handleCommand` 函数在执行 `select/insert/update/delete` 四种单一语句前后分别会执行 `auto begin transaction` 和 `commit`，特定session在执行前调用 `writeLog` 函数。