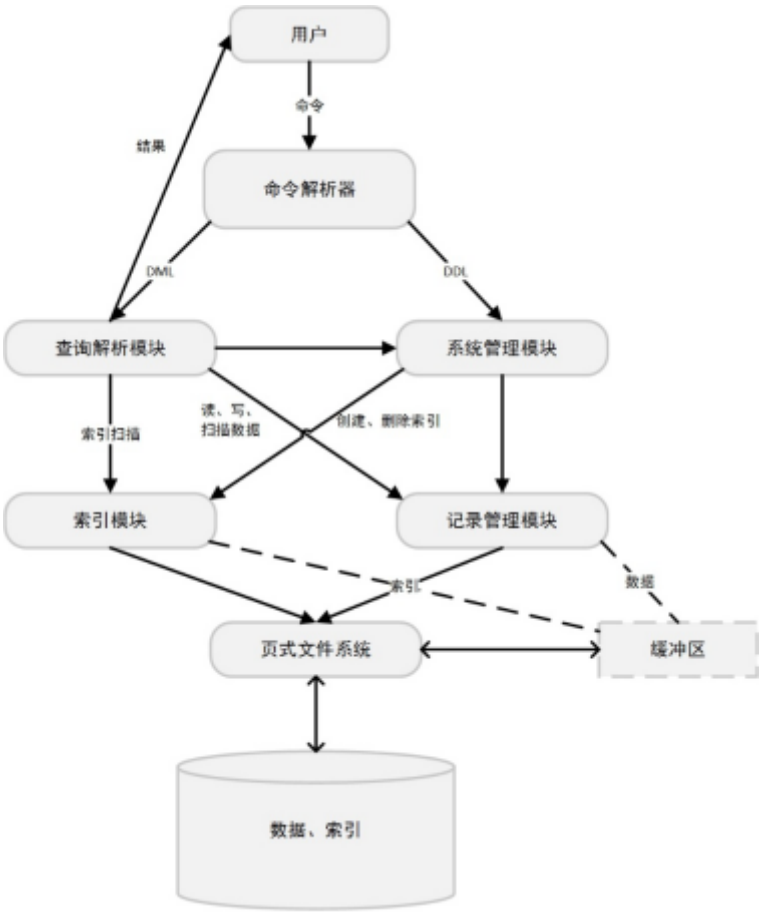
数据库系统概论项目报告

2017011475 潘俊臣

2017011474 郑林楷

1、系统架构设计



总体上数据库的架构与实验指导中的设计思路是类似的。

页式文件系统主要负责直接操作文件，按照页管理存储空间。

记录管理模块完成记录的存储，实现朴素的增删改查。

索引模块实现索引加速，利用 B-Tree 加快搜索。

系统管理模块主要负责对数据库和数据表进行管理。

查询解析模块主要负责将 SQL 语句解析为对索引模块，系统管理模块，系统管理模块的操作。

2、主要模块设计原理

（0）文件系统

文件系统利用给定的页式文件系统，不做过多说明。

（1）记录管理模块

记录管理模块主要负责实现最基本的记录的增删改查，是直接操作文件系统的一个模块。记录管理模块中的操作都是朴素的。

文件构成：

`Table.cpp` `Table.h`

具体功能：

1、表信息的存储

包括表头信息的存储和表内记录的存储。

表头信息以 JSON 格式直接存储对应的数据库配置文件（`database.udb`）中，存储在所属数据库的 table 项内部，多个表按照 List 排列。每一个表的表头信息存储如下：

```
{
    "col_num": 2, // 列的个数
    "col_ty": [   // 列的详细信息
        {
            "char_len": 0,
            "key": 2,
            "name": "id",
            "null": true,
            "ty": 0
        },
        {
            "char_len": 0,
            "key": 2,
            "name": "name",
            "null": true,
            "ty": 0
        }
    ],
    "index": [
        {
            "key": [
                0
            ],
            "key_num": 1,
            "name": "index1",
            "next_del_page": 4294967295,
            "offset": [
                0
            ],
            "page_num": 1,
            "record_size": 12,
            "root_page": 0,
            "ty": [
                0
            ]
        }
    ],
    "index_num": 0,        // 建立在本表的索引个数
    "name": "table1",      // 表的名字
    "p_key_index": -1,     // 表的主键索引号
    "record_num": 5,       // 表内存储的记录个数
    "record_onepg": 1008, // 单个页最多存储的记录个数
    "record_size": 8       // 每个记录的长度（字节）
}
```

记录信息则使用页式管理系统，将记录存储在（`表名.usid`）中。存储中我们为了最大程度利用空间，除了最基本的把数据依次排列以外，还设计了 Exist 位，存储在表的开头，占用 record_per_page 个比特的位置（按照字节向上取整），主要是为了标记对应的 record 是否有效（可能被删除）和标记对饮的 record 是否是空值。

2、增删改查

增：利用表头中的 record_num（相当于 RID），计算找到页式文件系统中的对应位置存储即可。

删：直接对某一个 ID 计算标记位的位置，置为 1 即可。

改：利用 ID 直接计算位置，修改对应的值即可。

查：利用 ID 直接计算位置返回结果即可

（2）索引模块

这个部分要负责实现索引的增删和维护，也是直接操作文件系统的一个模块。索引模块使用的是 B-Tree 作为索引，对外提供创建索引，更新索引，利用索引查询等操作。

索引模块主要负责实现索引的增删和维护，也是直接操作文件系统的一个模块。索引模块使用的是 B-Tree 树作为索引，对外提供创建索引，更新索引，利用索引查询等操作。

文件构成：

`BtreeNode.h` `Index.h` `Index.cpp`

具体功能：

1、索引信息的存储

包括索引表头的存储和 B-Tree 索引的存储。

索引表头以 JSON 格式存储在数据库配置文件（`database.udb`）中，具体的格式如下：

```
{
    "key": [ // 索引建立在哪些列上面，记录其 ID
        0
    ],
    "key_num": 1, // 索引列数
    "name": "index1", // 索引名
    "next_del_page": 4294967295, // 下一个未被使用的页的 ID，初始置为 -1 表示 NULL
    "offset": [ // 从文件中取出对应列的数据所需要的字节偏移量
        0
    ],
    "page_num": 1, // 已经使用的页个数
    "record_size": 8, // 储存单个索引值所需要的字节数
    "root_page": 0, // B-Tree 根节点对应的页的 ID
    "ty": [ // 索引对应的列的类型
        0
    ]
}
```

索引的内容，即 B-Tree 的各个节点，实际上是按照页来分配的（`表名_索引名.usid`），也就是说，每一个页就对应了一个 B-Tree 的节点，每一个节点上要记录 fa_index （父节点节点号）， record_cnt （当前节点索引值个数），之后按照「孩子编号 - RID - 索引值 - 孩子编号 - RID - 索引值」的顺序存储 B-Tree 节点的信息。

```
fa_index // 父节点节点号
record_cnt // 当前节点索引值个数
// 假设这个节点有N-1个索引值
child_index_0
record_id_1
record_val_1
child_index_1
.....
child_index_N-1
record_id_N-1
child_index_N
```

2、索引功能的实现

实际上就是 B-Tree 的实现，包括查询，删除，增加，以及内部需要的上溢下溢等等。由于是一个标准的实现就不多做阐述。

（3）系统管理模块

系统管理模块主要负责对数据库和数据表进行管理。支持主外键的创建和删除，列的添加、修改和删除。

文件构成：

数据库和数据表管理：`Database.cpp` `Database.h`

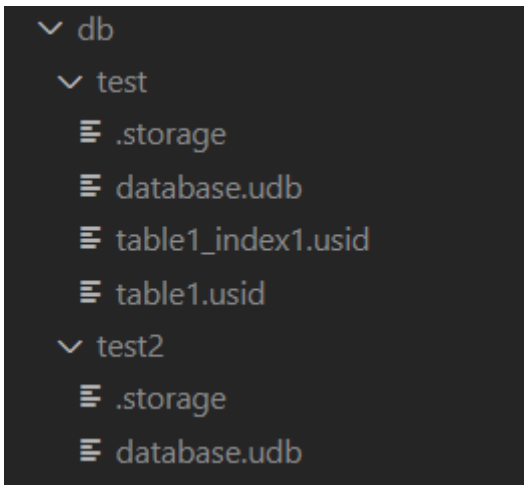
数据表列的增删改：`Table.cpp` `Table.h`

主外键的实现：`Key.h` `Key.cpp`

具体功能：

1、对数据库和数据表的管理

对数据库和数据表的管理逻辑主要实现在 `Database.cpp` 和 `Database.h` 两个文件中。实际实现中，我们将不同的数据库存储在了不同的文件夹下，直接命名为对应的数据名。每个数据库内部有自己的 `database.udb`，存储着数据库的配置信息，以及若干 `TableName.usid` 和 `TableName_IndexName.usid` 对应着记录和索引信息的页式管理文件。一个典型的结构如下：



对于我们的这种实现逻辑，对数据库的管理只需要直接删除或者增加对应的文件夹即可，而对于表的操作，则可以在 `database.udb` 中修改对应的表头（`database.udb` 的结构见上文记录管理模块），再修改相关的文件即可。

2、主外键功能的实现

主外键功能主要实现在 `Key.cpp` 和 `Key.h` 两个文件中。

每种 Key 又分为 PrimaryKey 和 ForeignKey，PrimaryKey 额外记录与其相关的 ForeignKey 的指针，ForeignKey 额外记录与其相关的 PrimaryKey 的指针。

3、列增删功能的实现

列增删的功能主要实现在 `Table.cpp` 和 `Table.h` 两个文件中。

对于某一个列的修改，体现在表头上其实相对容易，但是处理记录的变化就很困难，因为数据在记录文件中是连续排列的。我们对所有的增删列都采用了 `rebuild` 的方法，直接重新构造对应的存储文件，替换成新的存储格式。对应的可能有现有的索引失效的问题，调用索引模块相关的函数删除这一部分索引即可。

（4）查询解析模块

查询解析模块主要利用 Lex/Yacc 完成了 SQL 语句的解析，将语句转化为对数据库其他模块的操作最终完成数据库的功能。从使用上看，这一部分实际上相当于数据库的入口，所有的使用都通过这一模块解析，并反馈结果或者报错给用户。

文件构成：

Lex/Yacc 解析：`yacc.y` `lex.l` `parser.h`

输出格式的控制：`Print.cpp` `Print.h`

具体功能：

1、语句的解析

语句的解析主要通过 Lex/Yacc 解析来完成，这一部分主要是填写相关的语法规则和对应的处理逻辑，常规的语句解析没有什么特别的实现原理，不做过多说明。

2、嵌套查询的实现

首先在 Yacc 进行嵌套匹配，每个括号内都是一个 Select 语句，对应一个临时数据表，其 ID 从 -1 开始递减。每个 Select 语句会记录其内部的嵌套查询所对应的临时数据表 ID，然后先行完成其查询，再进行自身的查询。支持 Where 子句和 From 子句的嵌套。

3、聚集查询的实现

聚集查询其实处理逻辑不同之处在于查询完后要进行二次处理，将行数变成一行。支持 Min，Max，Average，Sum 和 Count(*) 五种聚集查询函数。

3、主要模块接口说明

从抽象的逻辑上讲，数据库是按照第一部分所说的，四个模块的方式组织的，但是实际在实现中，我们没有完全按照四个部分来组织代码。实际的主框架是由 Database→Table→Index 的结构加上查询解析模块组织起来的，所以以下按照这个实际实现的模块来说明。

（1）Database 模块

Database 模块主要完成的是与数据库，数据表相关的系统管理模块的功能。

```
class Database {
private:
    // 数据库信息的转换函数
    json toJson();
    void fromJson(json j);

    // Select 相关功能
```

```
    bool checkWhere(WhereStmt w);
    void storeData(uint idx);
    void dfsCross(uint idx, uint f_idx);

public:
    // 更新数据库表头
    void update();

    // 构造和析构
    Database(const char* name, bool create);
    ~Database();

    Table* createTable(const char* name, int col_num, Type* col_ty); // 增加数据表
    Table* openTable(const char* name); // 打开数据表
    int deleteTable(const char* name); // 删除数据表
    void showTables(); // SHOW TABLES 语句
    int findTable(std::string s); // 通过名字找数据表

    // 从字符串池中存取 VarChar
    char* getVarchar(uint64_t idx);
    uint64_t storeVarchar(char* str);

    void buildSel(uint idx);
};
```

(2) Table 模块

Table 模块主要完成的是记录管理部分的功能和与表结构相关的系统管理模块的功能。

```
class Table {
private:
    // 一些内部辅助函数
    void insert(Any& val, enumType ty, uint size, BufType& buf);
    void fetch(BufType& buf, enumType ty, uint size, Any& val);
    void fetchWithOffset(BufType& buf, enumType ty, uint size, Any& val, uint
offset);
    uint genOffset(uint index);
    char* getStr(BufType buf, uint size);

public:
    // 构造析构，存储相关
    json toJson();
    Table(Database* db, json j);
    Table() {}
    Table(uint _sel) : sel(_sel) {}
    ~Table();

    // 计算储存每条记录所需字节数
    uint calDataSize();

    // 建表
    int createTable(uint table_id,Database* db, const char* name, uint col_num,
Type* col_ty, bool create = false);

    // 增删改查（记录管理模块）
    int insertRecord(Any* data);
    int removeRecord(const int record_id);
```

```
    Anys queryRecord(const int record_id);
    int queryRecord(const int record_id, Any* &data);
    int updateRecord(const int record_id, const int len, Any* data);

    // 检查 Where 子句相关功能
    bool checkWhere(Anys data, WhereStmt &w);
    bool checkWheres(Anys data, std::vector<WhereStmt> &where);
    int removeRecords(std::vector<WhereStmt> &where);
    int updateRecords(std::vector<Pia> &set, std::vector<WhereStmt> &where);

    // 比较两个 Anys
    bool cmpRecord(Anys a, Anys b, enumOp op);
    bool cmpRecords(Anys data, enumOp op, bool any, bool all);

    // 对数据表的索引的增删和找
    int findIndex(std::string s);
    uint createIndex(std::vector<uint> key_index,std::string name);
    void removeIndex(uint index_id);

    // 对数据表列的增删改（系统管理模块）
    int createColumn(Type ty);
    int removeColumn(uint key_index);
    int modifyColumn(uint key_index, Type ty);
    void updateColumns();

    // 重构数据表（在对列增删改时使用）
    void rebuild(int ty, uint key_index);

    // 主键和外键的删除和创建
    int createForeignKey(ForeignKey* fk, PrimaryKey* pk);
    int removeForeignKey(ForeignKey* fk);
    int createPrimaryKey(PrimaryKey* pk);
    int removePrimaryKey();

    // 输出
    void printCol();
    void print();

    // 通过名字找到特定的列标号
    int findCol(std::string a);

    // 数据完整性检查
    int constraintCol(uint col_id);
    int constraintKey(Key* key);
    int constraintRow(Any* data, uint record_id, bool ck_unique);
    int constraintRowKey(Any* data, Key* key);

    // 自定义指针
    int pointer;
    void setPointer(int pointer);
    bool movePointer();
    Any getPointerColData(uint idx);
    Anys getPointerData();
};
```

（3）Index 模块

Index 模块主要完成的是索引部分的功能。

```
class Index {
private:
    void overflow_upstream(BtreeNode* now);                      // B-Tree 节点
上溢
    void overflow_downstream(BtreeNode* now);                    // B-Tree 节点
下溢
    std::vector<int> queryRecord(Anys* info, BtreeNode* now);    // B-Tree 内部
搜索
    void insertRecord(Anys* info, int record_id, BtreeNode* now); // B-Tree 内部
插入
    void removeRecord(Anys* info, int record_id, BtreeNode* now); // B-Tree 内部
删除

public:
    // 与构造和存储相关的接口
    Index() {}
    Index(Table* table, const char* name, std::vector<uint> key,int
btree_max_per_node);
    Index(Table* table, json j);
    json toJson();
    void open();
    void close();
    void remove();

    // 节点的存储和读取
    BtreeNode* convert_buf_to_BtreeNode(int index);
    void convert_BtreeNode_to_buf(BtreeNode* node);

    // 增删查，公共接口
    std::vector<int> queryRecord(Anys* info);
    void insertRecord(Anys* info, int record_id);
    void removeRecord(Anys* info, int record_id);

    // 调试接口
    void Debug();
    void debug(BtreeNode* node);
};
```

4、实验结果

（1）实际实现的功能

1、数据库

（1）数据库的创建和删除

2、数据表

（1）数据表的重命名

（2）数据表的创建和删除

（3）数据表的列重命名，列增加，列删除

3、主外键

（1）主键的增加和删除

（2）外键的增加和删除

（3）主外键相关的数据完整性维护

4、索引

（1）显式创建和删除索引

（2）主外键的默认索引（方便维护完整性）

（3）联合索引

5、增删改查

（1）parser中提到的基础增删改查

（2）嵌套查询（where嵌套和from嵌套）

（3）聚集查询（Min，Max，Average，Sum 和 Count(*)）。

6、其他部分

（1）相对详细的报错信息（例如主键重复，数据不匹配等）

（2）多表连接

5、小组分工

潘俊臣：文档报告编写，索引模块编写，查询解析模块部分代码编写

郑林楷：记录管理模块，系统管理模块，查询解析模块部分代码编写

6、参考文献及使用的开源代码

（1）Lex/Yacc

查询解析模块的实现中我们利用了现成工具 Lex/Yacc 实现了 SQL 文法的解析。

（2）JSON for Modern C++

开源库地址： https://github.com/nlohmann/json

在实现一些配置信息的存储时（比如表的列信息，索引的列信息，数据库的信息等等），我们使用了一个 JSON 的开源库。将这些只需要单次读取的配置信息的存储简化，不使用提供的文件管理系统。

7、其他需要说明的问题

（1）空值 NULL 的处理

我们在实现中将所有 bits 全为 1 的值统一处理为空值，也就是说对于 INT 类数据，其最大值为 $2^{32}-2$ 。

（2）VarChar 的处理

VarChar 我们统一存储在了每个数据库对应的一个 `.storage` 文件内，通过偏移量来计算每一个 VarChar 所在的储存位置来存取 VarChar。

（3）项目地址：https://github.com/Konano/UselessDatabase