清华大学

数据库系统概论

数据库课程项目报告

作者：

桥本优、钱雨杰、董胤蓬

指导教师：

冯建华、张勇

清华大学

计算机科学与技术系

January 6, 2016

Contents

Chapter 1

项目任务

本项目是清华大学计算机科学与技术系开设的《数据库系统概论》课程项目。

项目的任务是实现一个单用户的关系数据库管理系统。该项目分为四个功能模块：

(1) 记录管理模块：该模块是 DBMS 的文件系统，管理存储数据库记录以及元数据的文件。该模块依赖于我们预先给定的一个页式文件 I/O 系统，在此基础上扩展而成。

(2) 索引模块：为存储在文件中的记录建立 B+ 树索引，加快查找速度。

(3) 系统管理模块：实现基本的数据定义语言（DDL），实现解析器来解析命令行。

(4) 查询解析模块：解析 SQL 语句，能将输入的 SQL 语句解析成关系代数表达式，并生成查询执行计划，访问文件系统执行查询，输出查询结果。

在上述功能的基础上，还可以对该系统进行个性化的功能扩展及性能优化，内容包括但不限于：

(1) 查询优化：基于对查询计划代价的估计，为给定查询选择最有效的查询执行计划。

(2) 支持属性域约束和外键约束。

(3) 创建数据表时支持更多的数据类型。例如 decimal, date 等。

(4) 支持三个或以上表的连接。

(5) 支持更多 SQL 语句。例如聚集查询 AVG,SUM,MIN,MAX,GROUP BY 等。

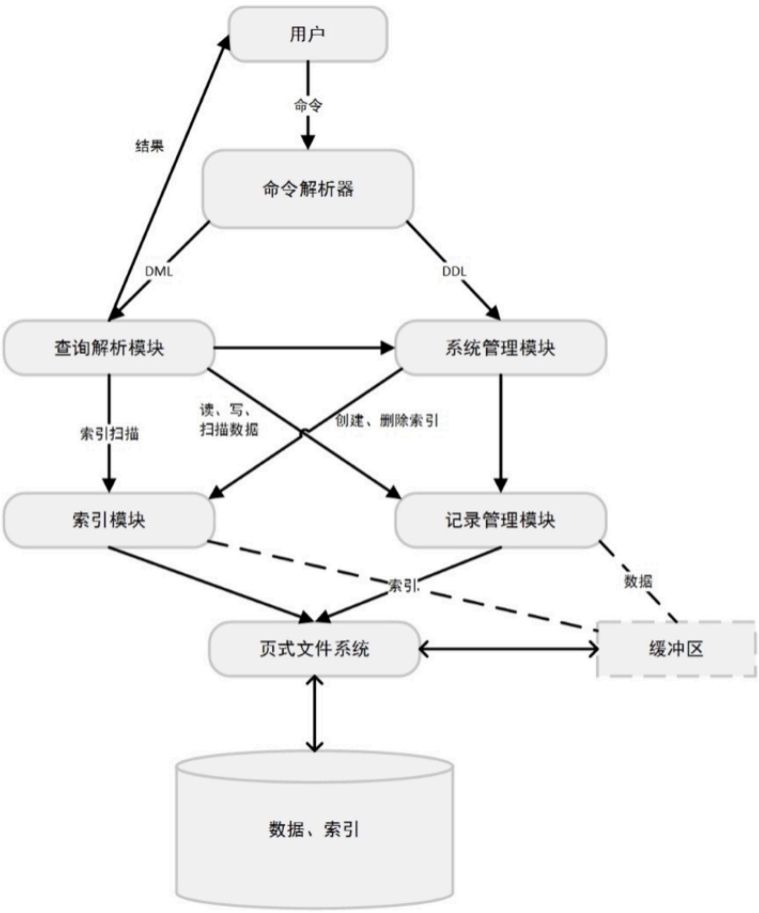
(6) 支持模糊查询。例如 LIKE 关键字以及“%, *, ?”等通配符。

(7) 提供类似 MySQL front 的图形化 UI。

Chapter 2

系统结构设计

数据库的基本架构与实验要求基本相同。我们将主要的模块分为了记录管理模块 (RM_Manager), 系统管理模块 (SM_Manager), 查询解析模块 (QL_Manager), 语句解析模块 (Parse_Manager) 和索引模块 (IX_Manager)。

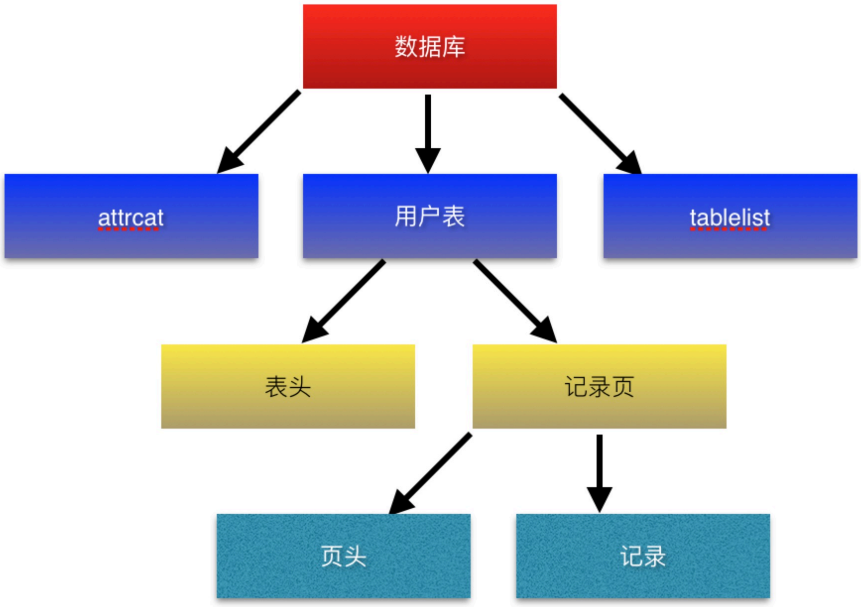


每个数据库在存储空间上对应于一个文件夹，其中包含很多文件，对应于每一个表。每个数据库中有两个专用的表，分别为 attrcat 与 tablelist。attrcat 表用于存

储当前数据库中每个用户表的属性值，而 tabellist 表用于存储当前数据库所有用户表的信息。

数据库中每个表中的记录大小可以不一样，在新建表时指定，但是同一个表中的记录记录是定长的。在每个表中，第一页为存储文件信息的页，后续的各页用于存储记录。为了方便索引，在每页中页头部分记录当前页的使用情况。

下图可以表示出我们设计的数据库的基本结构：



Chapter 3

模块设计

3.1 记录管理模块

记录管理模块主要负责在数据库中创建、打开、删除一个数据表，在表中插入、删除、更新一条记录，以及遍历表找到所有符合条件的记录。

我们在项目中把每张数据表存在独立的文件中。文件均由页式文件系统管理，每个数据表文件的第一页是表文件头页，记录整个数据表的一些信息；之后的页均为数据页，每个数据页的页头记录该页的一些信息。

具体地，表文件头页按以下格式组织：

```
struct FileHead
{
    int recordSize;      //记录长度
    int pageNumber;      //页的个数
    int recordPerPage;   //每页记录个数
    int recordNumber;    //记录的总数
};
```

表文件头页中依次记 4 个整数，分别表示该表中记录长度、页的个数、每页的记录个数和表中记录总数。文件头页其余位置留空，可以用作其他扩展。

数据页每页 8KB，前 96 byte 为页头，记录该页的一些信息，之后依次是固定长度的槽，每个槽可以放一条记录。数据页头按以下格式组织：

```
struct PageHead
{
    int usedSlot;        //已用槽的个数
    char slotMap[84];    //每个槽的状态4
};
```

数据页头首先存放一个整数，记录本页已经放了几条记录。之后 84 byte 中每一个 bit 依次对应之后每个槽的状态，0 表示为空，1 表示已经有记录。

本模块的实现主要包括 3 个类，分别是 RM_Manager、RM_FileHandle 以及 RM_FileScan。在实现各个功能时，使用和维护之前所述的文件头、页头的信息。

```
class RM_Manager
{
public:
    RM_Manager      (FileManager *pfm, BufPageManager* bpm);
    int CreateFile  (const char *fileName, int recordSize);
    int DestroyFile (const char *fileName);
    int OpenFile    (const char *fileName, RM_FileHandle &fileHandle);
    int CloseFile   (RM_FileHandle &fileHandle);
};
```

RM_Manager 提供的接口包括：(1)CreateFile：新建一个文件，生成表文件头页；(2)DestroyFile：删除一个文件；(3)OpenFile：打开一个文件,得到一个 RM_FileHandle,可以通过其进行对记录的操作；(4)CloseFile：关闭一个文件。

```
class RM_FileHandle
{
public:
    RM_FileHandle  ();
    ~RM_FileHandle ();
    int GetRec     (const RID &rid, RM_Record &rec) const;
    int InsertRec  (const char *pData, RID &rid);
    int DeleteRec  (const RID &rid);
    int UpdateRec  (const RM_Record &rec);
};
```

RM_FileHandle 提供的接口包括：(1)GetRec：根据 RID 获得记录内容；(2)InsertRec：插入一条记录，返回位置标识 RID；(3)DeleteRec：根据 RID 删除一条记录；(4)UpdateRec：更新一个记录（知晓 RID）。

```
class RM_FileScan {
public:
    RM_FileScan(FileManager *pfm, BufPageManager* bpm);
    ~RM_FileScan();
    int OpenScan   (const RM_FileHandle &fileHandle,
                    AttrType attrType,
                    int attrLength,
                    int attrOffset,
                    CompOp compOp,
                    void *value);
    int GetNextRec (RM_Record &rec);
    int CloseScan() {return 0;}
};
```

RM_FileScan 提供的接口包括：(1)打开一个 Scan，指定数据类型、长度、位置、约束条件；(2)获得一条符合条件的记录；(3)关闭一个 Scan。

3.2 系统管理模块

系统管理模块中需要完成以下几个操作：(1)CREATE Database：新建一个数据库；(2)DROP Database：删除一个数据库；(3)USE Database：使用一个数据库；(4)CREATE TABLE tableName(attrName1 Type1, attrName2 Type2,..., attrNameN TypeN NOT NULL, PRIMARY KEY(attrName1)：新建表，这里实现了 Not NULL 和 Primary key 两个关键字；(5)Show Database：显示这个数据库。

系统管理模块的类的定义如下：

```
class SM_Manager {
  public:
       SM_Manager  (RM_Manager &);                    // 构造函数
       ~SM_Manager ();                                // 析构函数
    int CreateDb    (const char *);
    int OpenDb      (const char *);                   // 打开目录
    int CloseDb     ();                                // 关闭目录
    int DropDb      (const char *);
    int ShowDb      (const char *);
    int ShowDb      ();
    int CreateTable (const char *,                 // 创建表
                     int        ,
                     AttrInfo   *);
    int DropTable   (const char *);                // 删除表
    int ShowTable   (const char *);               // 打印表的信息
    int GetTable    (const char* , int &, DataAttrInfo*&);
    int Exec        (const char *);
 private:
    char DataName[80];
    RM_Manager& rmm;
    RM_FileHandle attrfh;
    RM_FileHandle relfh;
    RM_FileHandle tablefh;
    bool IsOpenDB;
    Log* myLog;
};
```

为了实现这几个操作，所采取的方法是在创建数据库的时候首先需要创建一个相关的文件夹，然后在该文件夹下创建两个文件，一个是属性值列表，一个是关系列表。在删除一个数据库时，需要将该文件夹下所有文件全部删除。而新建一个表的时候需要将该表的属性添加到属性表中和数据库表中。而在属性表中每一项包含如下的信息：

```
// Used by SM_Manager::CreateTable
struct AttrInfo {
    char     attrName[MAXNAME + 1];          // Attribute name
    AttrType attrType;                       // Type of attribute
    int      attrLength;                     // Length of attribute
    bool     notNull;
    bool     primaryKey;
    AttrInfo(){
        memset(attrName, 0, sizeof(attrName));
        attrType = MyINT;
        attrLength = 0;
        notNull = false;
        primaryKey = false;
    }
};
```

在关系列表中需要记录每个用户表的名字。故类的定义如下：

```
struct TableInfo {
    char    tableName[MAXNAME + 1];
    TableInfo() {
        memset(tableName, 0, sizeof(tableName));
    }
};
```

3.3 查询解析模块

查询模块需要根据指令解析模块 Parse_Manager 得到的结果来查询出正确的结果。解析模块传入以下三种参数：

Value 类

```
struct Value {
    AttrType type;     // type of value
    void     *data;    // value
    friend ostream &operator<<(ostream &s, const Value &v){ ...
    int getlength() const { ...
    }
};
```

type 为数据类型；Data 将数据转化为 void* 的指针。

condition 类

```
struct Condition {
    RelAttr lhsAttr;      // left-hand side attribute
    CompOp  op;           // comparison operator
    int     bRhsIsAttr;   // TRUE if right-hand side is an attribute
                          //   and not a value
    RelAttr rhsAttr;      // right-hand side attribute if bRhsIsAttr = TRUE
    Value   rhsValue;     // right-hand side value if bRhsIsAttr = FALSE
    //friend ostream &operator<<(ostream &s, const Condition &c);
};
```

lhsAttr 为左侧的属性值；op 判断符号；bRhsIsAttr 为右侧是否为属性值。如果是属性值则 rhsAttr 为右侧属性值，如果不是则 rhsValue 为右侧的值

relation 类

```
struct RelAttr {
    AggeType type;
    char *relName;      // relation name (may be NULL)
    char *attrName;     // attribute name
    char *outkeyrel;
    char *outkeyattr;
    //friend ostream &operator<<(ostream &s, const RelAttr &ra);
};
```

type 为数据类型；relName 为所属关系名称；attrName 为属性名称；outkeyrel 为关联关系名称；outkeyattr 为关联关系属性

Ql_manager 负责整合前几个模块，其类结构如图所示：

```
class QL_Manager {
    public:
                                            // Constructor
        QL_Manager (SM_Manager *_smm, RM_Manager *_rmm, IX_Manager *_ixm);
        ~QL_Manager ();
        void Print(int nSelAttrs, const RelAttr selAttrs[],
                const char* relName, int nrid, RID* rid);
        void Search(const char* relName, const Condition* cond,
                int& nrid, RID*& rid);
        void Delete (const char *relName, int nrid, const RID rid[]);
        void Select (int nSelAttrs, RelAttr selAttrs[], int nRelations,
                char *relations[], int isGroup, int nConditions,
                Condition conditions[], const RelAttr& groupAttr);
         void Insert (const char *relName,int nValues, const Value values[],
                int nattr, const AttrInfo attr[]);
         void Update(const char *relName, const RelAttr &updAttr,
                const int bIsValue, const RelAttr& rhsRelAttr,
                const Value &rhsValue, int nrid, const RID rid[]);
```

其中五个主要部分说明如下：

(1)void Insert(const char *relName, int nValues, const Value values[],int nattr, const AttrInfo attr[]) 根据输入的关系名字和 value 域个数以及各个 value 的值插入到表中，如果插入时提供了有属性列表，那么需要将其他位置的值置为 null，需要注意的是如果该属性为 not null，提示本条插入信息无效。

(2)void Print(int nSelAttrs, const RelAttr selAttrs[], const char* relName, int nrid, RID* rid) 根据之前查询所得到的 RID 列表，某一个 Relation 中对应的属性依次输出。先判断 select 域是否为 *，如果是，则需要查询 attr 列表，并且将所有的属性值返回，如果不是，查询 attr 文件，根据属性名字和关系名字查询出所有需要输出的属性并且依次存放。

(3)void Update(const char *relName, const RelAttr &updAttr, const int bIsValue, const RelAttr& rhsRelAttr, const Value &rhsValue, int nrid, const RID rid[]) 更新数据库，将 rid 列表中所有项依次取出，根据 updata 的 set 域中的每一个条件，

更新每一个记录。其操作类似于 print 操作，首先需要查询属性列表，获得相应的每一个属性信息，包括 offset 等，然后在记录中找到相关属性的位置，然后判断是否为 null，然后更新记录，使用 filehandle 根据 Rid 来 update。

(4)void Delete(const char *relName,int nrid, const RID rid[]) 删除 rid 中的每一项，使用 filehandle 根据 rid 来删除记录。

(5)void Search(const char* relName, const Condition* cond, int& nrid, RID*& rid) 这一部分是 select，print，delete 之前的必要操作，该部分的作用是根据条件域得到一个 rid 列表，这里是一个关系，一个条件下的所有 rid 列表，rid 列表得到后根据交并操作得到最终的 rid 列表传给以上的三个函数。

(6)void Select (int nSelAttrs, SelAttr selAttrs[], int nRelations, char relations[], int isGroup, int nConditions, Condition conditions[], const RelAttr& groupAttr) 这一部分用于处理多表查询以及其他的一些特殊模块。另外还处理一些特殊情况，包括 select*，没有 from 等。这一部分将 search 和 print 功能整合起来。对于多表查询部分在 3.6.4 部分有详细说明。如果在 select 的时候没有 from 域，那么求出该表中的所有 rid。

最后，在每一个部分中都需要进行相关的判错处理，增加系统的鲁棒性。在任何操作时都要判断 relation 名字是否在当前数据库中存在，并且需要判断属性值是否也在该数据库中存在，如果存在缺省的关系名称，需要在数据库中搜索该属性值，如果存在两个或者以上关系表具有该属性名称，则需要报错处理。在插入时，当插入数据类型与数据库中该属性不匹配时报错处理，并且根据该属性值是否为 Not null 判断插入是否合法。相对应的，在更新数据库的时候也需要做类似的判断。

3.4 语句解析模块

语句解析模块负责将 SQL 语句进行解析，生成相应的关系代数表达式，并调用查询解析模块的函数进行插入、删除、查找等操作。我们采用正则表达式匹配的方法进行语句的解析工作，并对一些语法错误进行了检查。

由于 Python 语言对正则表达式的处理很好，所以在语句解析模块调用 Python 来进行正则表达式的匹配，并按照规定的格式将匹配的结果输出到临时文件。在 Parse_Manager 中读取临时文件的中间表达形式，并根据语句调用正确的函数。

```
class Parse_Manager {
    string filename;
    FILE *fp;
    QL_Manager* ql_manager;

public:
    Parse_Manager(string filename_) : filename(filename_) {}
    void MainLoop(SM_Manager *smm, RM_Manager *rmm, IX_Manager *ixm);
    char* readName();
    void readAttr(int& nattr, AttrInfo*& attrs);
    void readInsertData(char* relName, int nattr, AttrInfo* attrs);

    void readInt(void*& data);
    void readString(void*& data);
    void readFloat(void*& data);

    void readWhere(char* relName, int&nrid, RID*& rrid);
    void readWhere(int& ncond, Condition*& cond);

    void caland(RID*& rid1, RID* rid2, int& size1, int size2);
    void calor(RID*& rid1, RID* rid2, int& size1, int size2);

    void readRelAttr(RelAttr& relattr);
};
```

其中 fp 为 Python 解析原始 SQL 语句后产生的临时文件的句柄。其包含的函数功能为：(1)readAttr：读入属性信息，共 nattr 个；(2)readInsertData：读入插入数据；(3)readInt：读入一个整数；(4)readFloat：读入一个浮点数；(5)readString：读入字符串；(6)readWhere：读入 Where 子句；(7)readRelAttr：读入关系表信息。

其中 Where 子句的读入被重载了两次，第一个为利用栈进行表达式计算，第二个为多表连接等操作时不考虑 Where 子句含有表达式情况。具体 Where 表达式计算见附加功能部分。

3.5 索引模块

索引模块的功能是为表中的某一属性建立索引，提高查询速度。

我们项目中索引使用 B+ 树的数据结构实现，每个索引存放在一个独立的文件中。索引文件的命名方式是 RelName.AttrName.index。索引文件由页式文件系统管理，有三种页，分别是索引文件头页、中间节点页、叶子节点页。

具体地，索引文件头页按以下方式组织：

```
struct IX_FileHead
{
    AttrType attrType;
    int      attrLength;
    int      maxN;
    int      pageNum;
    int      firstEmptyPage;
    int      root;
};
```

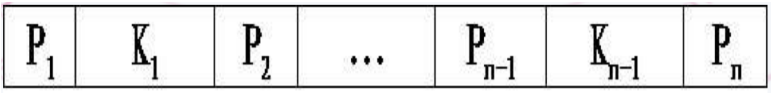
索引文件头页中记录索引数据的类型、长度、每页最多放多少索引项、已使用页的数量、根节点页等。

之后的每个页页头也需要记录一些内容，包括

```
struct IX_PageHead
{
    IX_PageType type;  // 0 for node, 1 for leaf
    int n;
};
```

type 表示该页是中间节点还是叶子节点，n 表示该页当前有的索引项个数。

中间节点页与叶子节点页的数据结构是相同的，如下图所示。



其中 P 为指针，K 为索引码值。区别在于，如果是中间节点，指针指向子节点（索引文件内的一个页），即 P 存储的是本文件的一个 RID；如果是叶子节点，每个索引码值对应一条数据记录，P 存储的是该记录在数据表页中的 RID；叶子节点最后一个索引码值后面的指针 P 指向后面一个叶子节点，这样一个索引文件中存储的所有索引项按顺序可以形成一个列表。

索引的插入、删除、查询等功能按照 B+ 树的要求实现。

索引模块的实现仿照记录管理模块,主要包括 3 个类,即 IX_Manager、IX_IndexHandle 以及 IX_IndexScan。

```
class IX_Manager
{
  public:
    IX_Manager   (FileManager *pfm_, BufPageManager* bpm_)
        : pfm(pfm_), bpm(bpm_) {}
    ~IX_Manager  ();
    int CreateIndex  (const char *fileName,          // Create new index
                      const char *indexName,
                      AttrType   attrType,
                      int        attrLength);
    int DestroyIndex (const char *fileName,          // Destroy index
                      const char *indexName);
    int OpenIndex    (const char *fileName,          // Open index
                      const char *indexName,
                      IX_IndexHandle &indexHandle);
    int CloseIndex   (IX_IndexHandle &indexHandle);  // Close index
  private:
    FileManager *pfm;
    BufPageManager *bpm;
};
```

IX_Manager 提供的接口包括新建索引、打开索引、关闭索引和删除索引。

```
class IX_IndexHandle
{
  public:
    IX_IndexHandle  ();
    ~IX_IndexHandle ();
    int InsertEntry     (void *pData, const RID &rid);  // Insert new index entry
    int DeleteEntry     (void *pData, const RID &rid);  // Delete index entry
    int ForcePages      ();                             // Copy index to disk
    void PrintEntries   ();
    int GetLowerBound   (void *pData, RID &indexid);
    int GetUpperBound   (void *pData, RID &indexid);
    int GetFirst        (RID &indexid);
};
```

IX_IndexHandle 提供的接口包括插入索引项、删除索引项，以及为了支持查询提供首项、LowerBound 和 UpperBound 的查询。B+ 树的主要功能实现在这个类中完成。

```
class IX_IndexScan
{
  public:
    IX_IndexScan  () : data(NULL) {}                 // Constructor
    ~IX_IndexScan () {}                              // Destructor
    int OpenScan      (IX_IndexHandle &indexHandle,     // Initialize index scan
                       CompOp      compOp,
                       void        *value);
    int GetNextEntry  (RID &rid);                       // Get next matching entry
    int CloseScan     ();                               // Terminate index scan
};
```

IX_IndexScan 提供的接口和 RM_FileScan 完全一致，区别在于 IX_IndexScan 的查询操作是通过索引在 B+ 树上完成的，效率更高。

其他的模块在执行新建索引以及插入、删除、更新记录的同时，要更新相关的索引。

3.6 扩展功能

3.6.1 扩展数据类型

我们除了整数 (int) 和字符串 (string) 外，还扩展了浮点数 (float) 类型，支持建表时指定 float 的宽度，即精度。同时在插入数据时可以有 float 的表达式计算，在查找语句 select 时可以对 float 类型查找。由于 float 基本原理与 int 一致，故不再赘述。

3.6.2 域完整性约束

域完整性约束，即建表时有 CHECK 关键字，约束某一个属性可能的取值。

为实现这一功能，我们对每个有域完整性约束的属性，新建一个和索引文件一样的约束文件，命名方式为 RelName.AttrName.check.index，也用 IX_Manager 管理。在这个文件中，将该属性可能的取值作为索引项依次插入。

插入记录时，检查所有有域完整性约束的属性值是否在其约束文件中存在。如果存在，允许插入，否则拒绝并报错。

3.6.3 外键约束

外键约束，用于与另一张表的关联，以保持数据的一致性。外键一定是另一张表的主键，因此一定是建好索引的。

所以，在更新一个有外键约束的记录时，打开关联表的主键索引，检查更新后的值是否存在。如果存在，允许更新，否则拒绝并报错。

3.6.4 模糊匹配

模糊匹配支持 LIKE 关键字以及“%, *, ?" 通配符。其中% 匹配零个或多个字符，* 匹配一个或多个字符，？匹配零个或一个字符。模糊匹配的实现在 RM_FileScan 中实现，增加操作符 LIKE_OP 表示模糊匹配查询。

在模糊匹配时，考虑到匹配的鲁棒性以及正确性，我们采用了 Python 正则表达式的方法，这样可以减少错误发生的几率。这时我们仅需要将%, *, ? 换成 Python 中对应的正则表达式匹配的语法即可。

```
s2 = s2.replace('?', '(.?)')
s2 = s2.replace('*', '(.+)')
s2 = s2.replace('%', '(.*)')
```

3.6.5 Where 子句表达式

对于 Where 子句的处理部分，支持 Where 条件的逻辑表达式形式，即通过逻辑词 and，or 和括号连接的逻辑表达式。为了实现此功能，我们在 Parse_Manager 中得 readWhere 函数中建立两个栈用于存储操作数 (RID 的列表) 和操作符 (包括'and', 'or', '(', ')')，并根据优先级顺序进行计算。每次遇到一个条件，首先通过 search 函数查找符合此条件的 RID 的集合，然后通过 and，or 运算求这些 RID 集合的交或并。等到所有的条件输入完毕，将求出的 RID 的集合对应的记录值输出。

3.6.6 三个表以上的连接

三个表以上的查询实现在 select 函数中，所采取的方式是笛卡尔积的方式，首先读出每一个表中所有的 RID，存放在各自的列表中，然后生成一系列的笛卡尔积，然后类似查询解析中 select 的实现，根据 value 域对结果进行筛选。

3.6.7 聚集查询

聚集查询根据要求支持 4 个操作，MAX，MIN，AVG 和 SUM，分别为最大，最小，平均和总和，这一部分的实现在 select 函数中实现，根据传入参数中是否具有聚集属性来判断是否需要返回聚集查询结果。该部分实现比较简单，只需要在输出的时候根据聚集属性将结果取最大，最小，平均和求和即可，其中求平均时需要统计一个总数。

3.6.8 图形用户界面

图形用户界面使用 Qt 实现，在文本框中输入 SQL 语句，点击后通过命令行执行，然后将输出的结果以表格形式显示。

Chapter 4

实验成果

Chapter 5

小组分工

Chapter 6

实验总结