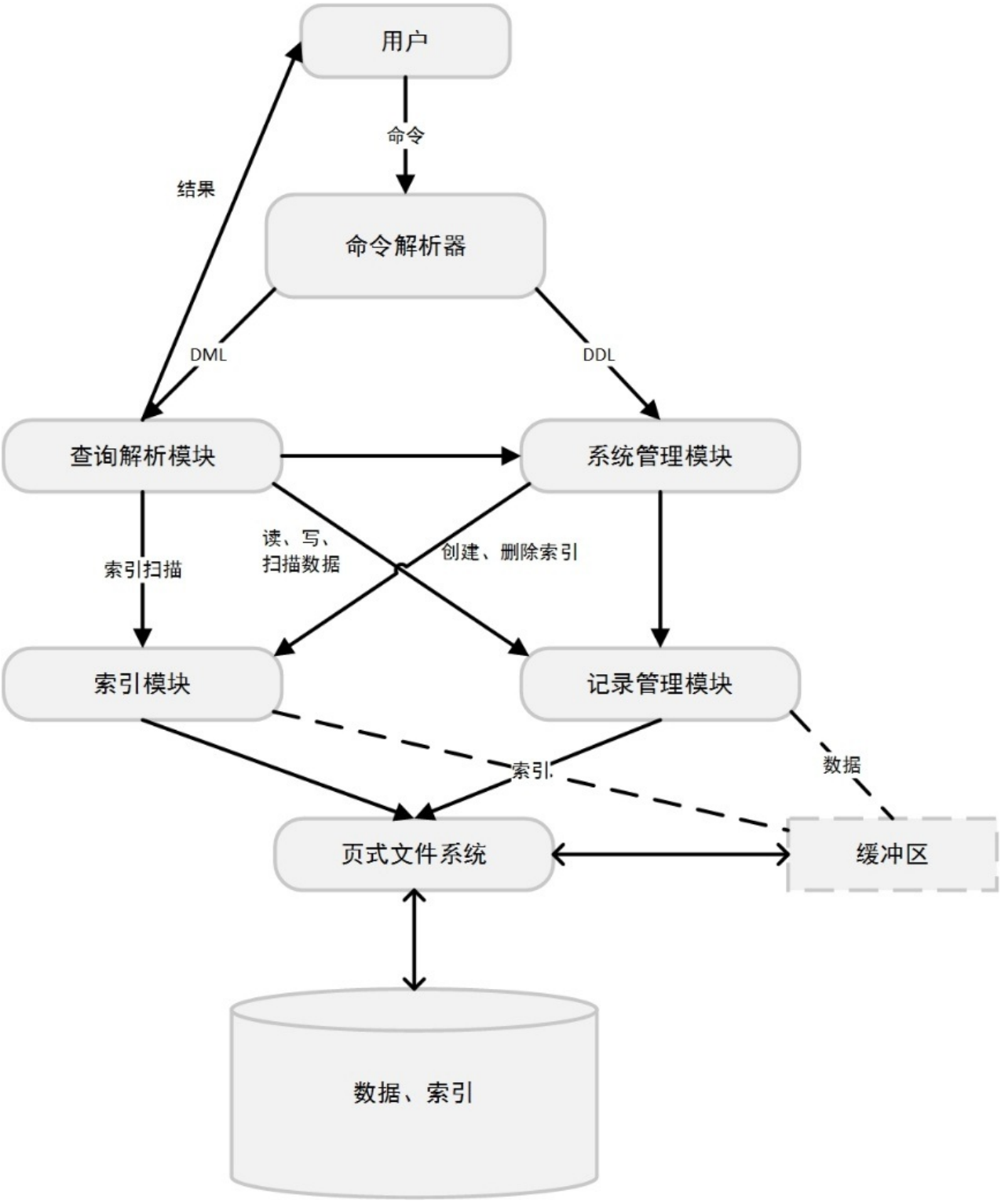
naiveDB总结报告

系统架构设计

我们实现了一个单用户的关系数据库管理系统。本项目分为四个功能模块：

i. 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。该模块依赖于斯坦福大学CS346预先给定的一个页式文件系统，在此基础上扩展而成。
ii. 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。
iii. 系统管理模块：实现基本的数据定义语言（DDL），对数据库和数据表进行管理。
iv. 查询解析模块：实现基本的数据操作语言（DML），对数据库里的数据进行增删改查等基本操作。

系统设计如下：



模块说明

RM

RM提供用于管理文件中无序记录的类和方法，组件中所有类名和方法名都以RM作为前缀。RM组件存储在由PF组件提供的页式文件系统中，每个文件的第一页存储文件的可用空间信息等元数据，之后的每一页开头会记录一个bitmap，表示这一页中所有的slot，哪些已经存储了记录，哪些还未存储记录。记录为定长设计，这样我们能够通

过RID确定一个记录。
由于需要实现NULL，所以对每一条记录，我们还会在每条记录的最后存储num bits来表示该记录对应的每个属性是否为null，其中num为一条记录的属性个数。
RM_Manager类负责文件的创建/删除，打开/关闭。
RM_FileHandle类用于操作打开的RM文件中的记录。
RM_FileScan类负责对文件中的记录进行扫描，可以基于指定的条件。

IX

IX提供管理无序记录的索引的类和方法，组件中所有类名和方法名都以IX作为前缀。索引最终用于加速基于条件的选择，更新和删除操作。和记录本身一样，索引也存在由PF组件提供的页式文件系统中。我们使用B+树实现索引模块，以加快查找速度。
IX_Manager类处理索引的创建/插入/删除，打开/关闭。
IX_IndexHandle类用于插入、删除和查找索引记录。此类使用B+树实现。
IX_IndexScan类负责对索引中的条目进行扫描，可以基于指定的条件。

SM

SM模块实现了基本的数据定义语言（DDL），对数据库和数据表进行管理。
SM_Manager类实现了数据库的添加、删除和选择，对某数据库的表进行添加、删除和查询，以及建立和删除索引操作。

QL

QL作为查询语言组件，提供了增删查改的类和方法，组件中所有类名和方法名都以QL作为前缀。
QL_Manager类实现了增删查改。所有操作在所选属性已建立索引的情况下将优先使用索引查找记录，否则从无序记录中遍历寻找满足条件的记录。其中Select支持不超过两个表的连接，Insert/Delete/Update的同时也将更新索引。

接口说明

RM

```
class RM_Manager {
  public:
       RM_Manager  (PF_Manager &pfm);            // Constructor
       ~RM_Manager ();                           // Destructor
    RC CreateFile  (const char *fileName, int recordSize);
                                                 // Create a new file
    RC DestroyFile (const char *fileName);       // Destroy a file
    RC OpenFile    (const char *fileName, RM_FileHandle &fileHandle);
                                                 // Open a file
    RC CloseFile   (RM_FileHandle &fileHandle);  // Close a file
};
```

`RC CreateFile (const char *fileName, int recordSize)`
创建名为fileName的文件,其中文件中记录的长度为recordSize

`RC DestroyFile (const char *fileName)`
删除名为fileName的文件

`RC OpenFile (const char *fileName, RM_FileHandle &fileHandle)`
打开名为fileName的文件,调用成功后fileHandle作为操作文件内容的句柄

`RC CloseFile (RM_FileHandle &fileHandle)`

关闭fileHandle这个句柄

```
class RM_FileHandle {
  public:
       RM_FileHandle  ();                                    // Constructor
       ~RM_FileHandle ();                                    // Destructor
    RC GetRec         (const RID &rid, RM_Record &rec) const;
                                                             // Get a record
    RC InsertRec      (const char *pData, RID &rid);        // Insert a new record,
                                                             //   return record id
    RC DeleteRec      (const RID &rid);                     // Delete a record
    RC UpdateRec      (const RM_Record &rec);               // Update a record
    RC ForcePages     (PageNum pageNum = ALL_PAGES) const; // Write dirty page(s)
                                                             //   to disk
};
```

`RC GetRec (const RID &rid, RM_Record &rec) const;`
根据指定的rid（包括页号和slot编号），返回对应位置存储的数据RM_Record。

`RC InsertRec (const char *pData, RID &rid);`
将pData指向的数据作为新记录插入文件中。如果成功，返回参数&rid应该指向新插入记录的RID（包括页号和slot编号）。

`RC DeleteRec (const RID &rid);`
文件中删除标识符为rid的记录。

`RC UpdateRec (const RM_Record &rec);`
用rec的当前内容替换文件中记录的现有内容。RM_Record中包含rid，即可以通过rec得知需要替换的记录位置。

`RC ForcePages (PageNum pageNum = ALL_PAGES) const;`
调用相应的方法PF_FileHandle::ForcePages，以便将文件的一个或所有脏页的内容从缓冲池复制到磁盘。

```
class RM_FileScan {
  public:
       RM_FileScan  ();                                  // Constructor
       ~RM_FileScan ();                                  // Destructor
    RC OpenScan     (const RM_FileHandle &fileHandle,  // Initialize file scan
                     AttrType      attrType,
                     int           attrLength,
                     int           attrOffset,
                     CompOp        compOp,
                     void          *value,
                     ClientHint    pinHint = NO_HINT);
    RC GetNextRec   (RM_Record &rec);                  // Get next matching record
    RC CloseScan    ();                                // Terminate file scan
};
```

```
RC OpenScan (const RM_FileHandle &fileHandle, AttrType attrType, int attrLength, int attrOffset, Co
mpOp compOp, void *value, ClientHint pinHint = NO_HINT)
```

初始化由fileHandle管理的打开文件的扫描,在扫描期间,只会获取某个属性满足条件的技术.attrOffset指定了属性在每条记录中的位置,attrType和attrLength指定了属性的类型和长度.compOp指定了比较的方式,value指定比较的值,如果value是空指针,则没有条件.pinHint用于指定扫描时的页面固定策略

`RC GetNextRec (RM_Record &rec)`

获取满足扫描条件的下一条记录的副本,如果成功返回,rec中将包含记录的副本及其记录标识符,没有满足条件的记录则返回RM_EOF

`RC CloseScan ()`
关闭当前扫描

IX

```
class IX_Manager {
  public:
       IX_Manager   (PF_Manager &pfm);              // Constructor
       ~IX_Manager  ();                             // Destructor
    RC CreateIndex  (const char *fileName,          // Create new index
                     int        indexNo,
                     AttrType   attrType,
                     int        attrLength);
    RC DestroyIndex (const char *fileName,          // Destroy index
                     int        indexNo);
    RC OpenIndex    (const char *fileName,          // Open index
                     int        indexNo,
                     IX_IndexHandle &indexHandle);
    RC CloseIndex   (IX_IndexHandle &indexHandle);  // Close index
};
```

`RC CreateIndex (const char *fileName, int indexNo, AttrType attrType, int attrLength)`
为名为fileName的数据建立编号为indexNo的索引,使用时需要确保indexNo唯一且是非负的,被索引属性的类型和长度由attrType和attrLength指定

`RC DestroyIndex (const char *fileName, int indexNo)`
删除为名为fileName的数据建立的编号为indexNo的索引

`RC OpenIndex (const char *fileName, int indexNo, IX_IndexHandle &indexHandle)`
打开为名为fileName的数据建立的编号为indexNo的索引,调用成功后indexHandle作为操作索引内容的句柄

`RC CloseIndex (IX_IndexHandle &indexHandle)`
关闭indexHandle这个索引句柄

```
class IX_IndexHandle {
  public:
       IX_IndexHandle  ();                              // Constructor
       ~IX_IndexHandle ();                              // Destructor
    RC InsertEntry     (void *pData, const RID &rid);  // Insert new index entry
    RC DeleteEntry     (void *pData, const RID &rid);  // Delete index entry
    RC ForcePages      ();                             // Copy index to disk
};
```

`RC InsertEntry (void *pData, const RID &rid);`
在与IX_IndexHandle相关联的索引中插入一个新条目。参数pData指向要插入到索引中的属性值，参数rid表示pData相对应的记录位置（包含页号和slot编号）。

`RC DeleteEntry (void *pData, const RID &rid);`
从与IX_IndexHandle相关联的索引中删除(*pData，rid)对的条目。

`RC ForcePages ();`
将所有与IX_IndexHandle相关联的页面复制到磁盘上。通过调用索引文件的PF _ FileHandle : : ForcePages，索引

页面内容被强制放到磁盘上。

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

```
RC OpenScan (const IX_IndexHandle &indexHandle, CompOp compOp, void *value, ClientHint pinHint = NO
_HINT)
```

对indexHandle引用的索引中的条目进行基于条件的扫描,索引属性值以指定的方式与指定的值进行比较.compOp指定了比较的方式,value指定比较的值,如果value是空指针,则没有条件

`RC GetNextEntry (RID &rid)`
将rid设置为索引扫描中下一条记录的RID,没有满足条件的记录则返回IX_EOF

`RC CloseScan ()`
关闭当前扫描

SM

```
class SM_Manager {
  public:
       SM_Manager  (IX_Manager &ixm, RM_Manager &rmm);  // Constructor
       ~SM_Manager ();                                  // Destructor
    RC CreateDb   (const char* dbName);                 // Create database
    RC DestroyDb(const char* dbName);                   // Destroy database
    RC OpenDb      (const char *dbName);                // Open database
    RC CloseDb     ();                                  // Close database
    RC CreateTable (const char *relName,                // Create relation
                    int        attrCount,
                    AttrInfo   *attributes);
    RC DropTable   (const char *relName);               // Destroy relation
    RC CreateIndex (const char *relName,                // Create index
                    const char *attrName);
    RC DropIndex   (const char *relName,                // Destroy index
                    const char *attrName);
    RC PrintDBs();                                      // Print information for all databases
    RC PrintTables();                                   // Print all tables in one database
    RC PrintTable(const char* relName);                 // Print information for one table
};
```

`RC CreateDb (const char* dbName);`
创建名为dbName的数据库，即创建名为dbName的文件夹，并为其创建catalog。

`RC DestroyDb(const char* dbName);`
删除名为dbName的数据库，即删除名为dbName的文件夹和其中所有内容。

`RC OpenDb (const char *dbName);`

打开名为dbName的数据库，转到dbName对应的文件夹（用系统调用chdir），并打开这个数据库的catalog文件。

`RC CloseDb ();`
关闭与数据库相关的所有文件。文件关闭会自动地把相关buffer给flush到磁盘。

`RC CreateTable (const char *relName, int attrCount, AttrInfo *attributes);`
应该首先修改catalog。一条关于新的表的记录应该被加入catalog relcat中。关于每个属性的记录应该被加入catalog attrcat中。为了加入catalog中，需要计算记录长度及offset。
之后调用RM_Manager的CreateFile即可。

`RC DropTable (const char *relName);`
删除表以及表对应的所有索引。是通过attrcat来查找所有索引的，并通过调用IX_Manager的DestroyIndex来删除索引。表文件是通过RM_Manager的DestroyFile删除的。同时，要删除relcat和attrcat中的相关信息。

`RC CreateIndex (const char *relName, const char *attrName);`
要对已有的内容建立索引。应该首先通过attrcat判断是否已经对这个属性建立了索引，如果已有了，应该返回一个非零信息。之后更新attrcat，插入新增的index。之后建立索引、打开索引、通过RM来scan已有的记录并插入索引、关闭索引。

`RC DropIndex (const char *relName, const char *attrName);`
首先检查attrcat是否存在索引，如果不存在就返回非零信息。然后更新attrcat，并删除索引。

`RC PrintDBs();`
输出所有database，即输出目录下的所有文件夹的名字。

`RC PrintDBs();`
输出对应database中所有表及表中的属性，需要遍历relcat和attrcat。

`RC PrintTable(const char* relName);`
输出一个表的所有属性。需要遍历attrcat，并将其中所属表名为relName的所有属性输出。

QL

```
class QL_Manager {
public:
    QL_Manager (SM_Manager &smm, IX_Manager &ixm, RM_Manager &rmm);
    ~QL_Manager();                       // Destructor

    RC Select  (int nSelAttrs,           // # attrs in select clause
        const RelAttr selAttrs[],        // attrs in select clause
        int   nRelations,                // # relations in from clause
        const char * const relations[],  // relations in from clause
        int   nConditions,               // # conditions in where clause
        Condition conditions[]);   // conditions in where clause

    RC Insert  (const char *relName,     // relation to insert into
        int   nValues,                   // # values
        Value values[]);           // values to insert

    RC Delete  (const char *relName,     // relation to delete from
        int   nConditions,               // # conditions in where clause
        Condition conditions[]);   // conditions in where clause

    RC Update (const char *relName,      // relation to update
        int nColumns,                    // # columns and values in set clause
        const char* const columnNames[],       // columnNames in set clause
```

```
        const Value values[],            // values in set clause
        int   nConditions,               // # conditions in where clause
        Condition conditions[]);   // conditions in where clause

    RC SetPrintPara(bool p) {printPara = p; }
};
```

```
RC Select (int nSelAttrs, const RelAttr selAttrs[], int nRelations, const char * const relations[],
 int nConditions, Condition conditions[] )
```

该方法的六个参数逻辑上包括三对,每对的第一个参数是一个整数n,表示该对的第二个参数中的项目数.第二个参数是包含实际项的长度为n的数组.
参数nSelAttrs包含所选属性的数量，而参数selAttrs是包含实际属性的长度为nSelAttrs的数组.
参数nRelations包含所选表名的数量，而参数relations是包含实际表名的长度为nRelations的数组.
参数nConditions条件的数量，而参数conditions是包含实际条件长度为nConditions的数组.

`RC Insert (const char *relName, int nValues, const Value values[])`
在关系relName中使用指定的属性值创建新元组,应确认参数values中的值的数量和类型与关系的模式匹配

`RC Delete (const char *relName, int nConditions, Condition conditions[])`
删除满足指定条件的关系relName中的所有元组,如果没有条件,则此方法删除relName中的所有元组

```
RC Update (const char *relName, const RelAttr &updAttr, const int bIsValue, const RelAttr &rhsRelAt
tr, const Value &rhsValue, int nConditions, Condition conditions[]
)
```

更新满足指定条件的关系relName中的所有元组,并将每个更新的元组中的updAttr的值设置为新值,如果isNULL或isNotNULL不为0,则不使用比较.否则如果bIsValue=1,那么新的更新值应该是rhsValue中的常量，否则新的更新值应该从rhsRelAttr指定的属性中获取.如果没有条件,则此方法应更新relName中的所有元组.

实验结果

可以完成所有基本要求，包括：

- 创建和删除数据库
- 创建和删除表
- 增删查改记录
- Select时两个表联合
- 支持INT、FLOAT和STRING基本类型
- 支持NULL与PRIMARY KEY

同时，我们也完成了额外要求：

- 创建、删除与使用索引

小组分工

师天麾负责：

- RM_FileHandle & RM_FileHeader & RM_PageHeader & RM_BitMap & RM_Record
- IX_IndexHandle & IX_FileHeader & BTreeNode
- SM_Manager

罗华一负责：

- RM_Manager & RM_FileScan
- IX_Manager & IX_IndexScan
- QL_Manager

参考文献

http://web.stanford.edu/class/cs346/