Advanced Database Organization - Fall 2023
CS 525 - 04/05
Programming Assignment III: Record Manager
Due: Friday, December 1st 2023 by 11:59pm

---

1. Task

The goal of this assignment is to implement a simple record manager. The record manager handles tables with a fixed schema. Clients can insert records, delete records, update records, and scan through the records in a table. A scan is associated with a search condition and only returns records that match the search condition. Each table should be stored in a separate page file and your record manager should access the pages of the file through the buffer manager implemented in the last assignment.

Hints: This assignment is much more complex than the previous assignments and it is easy to get stuck if you are unclear about how to structure your solution and what data structures to use. Sit down with a piece of paper first and design the data structures and architecture for your implementation.

- `Record Representation`: The data types we consider for this assignment are all fixed length. Thus, for a given schema, the size of a record is fixed too.
- `Page Layout`: You will have to define how to layout records on pages. Also you need to reserve some space on each page for managing the entries on the page. Refresh your memory on the page layouts discussed in class! For example, how would you represent slots for records on pages and manage free space.
- `Table information pages`: You probably will have to reserve one or more pages of a page file to store, e.g., the schema of the table.
- `Record IDs`: The assignment requires you to use record IDs that are a combination of page and slot number.
- `Free Space Management`: Since your record manager has to support deleting records you need to track available free space on pages. An easy solution is to link pages with free space by reserving space for a pointer to the next free space on each page. One of the table information pages can then have a pointer to the first page with free space. One alternative is to use several pages to store a directory recording how much free space you have for each page.

2. tables.h

This header defines basic data structures for schemas, tables, records, record ids (RIDs), and values. Furthermore, this header defines functions for serializing these data structures as strings. The serialization functions are provided (`rm_serializer.c`). There are four datatypes that can be used for records of a table: integer (`DT_INT`), float (`DT_FLOAT`), strings of a fixed length (`DT_STRING`), and boolean (`DT_BOOL`). All records in a table conform to a common schema defined for this table. A record is simply a record id (`rid` consisting of a page number and slot number) and the concatenation of the binary representation of its attributes according to the schema (`data`).

2.1. **Schema.** A schema consists of a number of attributes (`numAttr`). For each attribute we record the name (`attrNames`) and data type (`dataTypes`). For attributes of type `DT_STRING` we record the size of the strings in `typeLength`. Furthermore, a schema can have a key defined. The key is represented as an array of integers that are the positions of the attributes of the key (`keyAttrs`). For example, consider a relation `R(a,b,c)` where a then `keyAttrs` would be `[0]`.

2.2. **Data Types and Binary Representation.** Values of a data type are represented using the `Value` struct. The value struct represents the values of a data type using standard C data types. For example, a string is a `char *` and an integer using a C `int`. Note that values are only used for expressions and for returning data to the client of the record manager. Attribute values in records are stored slightly different if the data type is string. Recall that in C a string is an array of characters ended by a 0 byte. In a record, strings are stored without the additional `0` byte in the end. For example, for strings of length `4` should occupy `4` bytes in the `data` field of the record.

2.3. **Interface.** :

```
#ifndef TABLES_H
#define TABLES_H

#include "dt.h"

// Data Types, Records, and Schemas
typedef enum DataType {
  DT_INT = 0,
  DT_STRING = 1,
  DT_FLOAT = 2,
  DT_BOOL = 3
} DataType;

typedef struct Value {
  DataType dt;
  union v {
    int intV;
    char *stringV;
    float floatV;
    bool boolV;
  } v;
} Value;

typedef struct RID {
  int page;
  int slot;
} RID;

typedef struct Record
{
  RID id;
  char *data;
} Record;

// information of a table schema: its attributes, datatypes,
typedef struct Schema
{
  int numAttr;
  char **attrNames;
  DataType *dataTypes;
  int *typeLength;
  int *keyAttrs;
  int keySize;
} Schema;

// TableData: Management Structure for a Record Manager to handle one relation
typedef struct RM_TableData
{
  char *name;
  Schema *schema;
  void *mgmtData;
} RM_TableData;

#define MAKE_STRING_VALUE(result, value)                                \
  do {                                                                  \
    (result) = (Value *) malloc(sizeof(Value));                         \
    (result)->dt = DT_STRING;                                           \
    (result)->v.stringV = (char *) malloc(strlen(value) + 1);           \
    strcpy((result)->v.stringV, value);                                 \
  } while(0)


#define MAKE_VALUE(result, datatype, value)                             \
  do {                                                                  \
    (result) = (Value *) malloc(sizeof(Value));                         \
    (result)->dt = datatype;                                            \
    switch(datatype)                                                    \
      {                                                                 \
```

```
      case DT_INT:                                                    \
        (result)->v.intV = value;                                     \
        break;                                                        \
      case DT_FLOAT:                                                  \
        (result)->v.floatV = value;                                   \
        break;                                                        \
      case DT_BOOL:                                                   \
        (result)->v.boolV = value;                                    \
        break;                                                        \
      }                                                               \
  } while(0)


// debug and read methods
extern Value *stringToValue (char *value);
extern char *serializeTableInfo(RM_TableData *rel);
extern char *serializeTableContent(RM_TableData *rel);
extern char *serializeSchema(Schema *schema);
extern char *serializeRecord(Record *record, Schema *schema);
extern char *serializeAttr(Record *record, Schema *schema, int attrNum);
extern char *serializeValue(Value *val);

#endif
```

3. EXPR.H

This header defines data structures and functions to deal with expressions for scans. These functions are implemented in `expr.c`. Expressions can either be constants (stored as a `Value` struct), references to attribute values (represented as the position of an attribute in the schema), and operator invocations. Operators are either comparison operators (equals and smaller) that are defined for all data types and boolean operators `AND`, `OR`, and `NOT`. Operators have one or more expressions as input. The expression framework allows for arbitrary nesting of operators as long as their input types are correct. For example, you cannot use an integer constant as an input to a boolean AND operator. As explained below, one of the parameters of the scan operation of the record manager is an expression representing the scan condition.

3.1. **Interface.** :

```
#ifndef EXPR_H
#define EXPR_H

#include "dberror.h"
#include "tables.h"

// datatype for arguments of expressions used in conditions
typedef enum ExprType {
  EXPR_OP,
  EXPR_CONST,
  EXPR_ATTRREF
} ExprType;

typedef struct Expr {
  ExprType type;
  union expr {
    Value *cons;
    int attrRef;
    struct Operator *op;
  } expr;
} Expr;

// comparison operators
typedef enum OpType {
  OP_BOOL_AND,
  OP_BOOL_OR,
  OP_BOOL_NOT,
  OP_COMP_EQUAL,
  OP_COMP_SMALLER
} OpType;
```

```
typedef struct Operator {
  OpType type;
  Expr **args;
} Operator;

// expression evaluation methods
extern RC valueEquals (Value *left, Value *right, Value *result);
extern RC valueSmaller (Value *left, Value *right, Value *result);
extern RC boolNot (Value *input, Value *result);
extern RC boolAnd (Value *left, Value *right, Value *result);
extern RC boolOr (Value *left, Value *right, Value *result);
extern RC evalExpr (Record *record, Schema *schema, Expr *expr, Value **result);
extern RC freeExpr (Expr *expr);
extern void freeVal(Value *val);

#define CPVAL(_result,_input)                                          \
  do {                                                                 \
    (_result)->dt = _input->dt;                                        \
  switch(_input->dt)                                                   \
    {                                                                  \
    case DT_INT:                                                       \
      (_result)->v.intV = _input->v.intV;                                      \
      break;                                                           \
    case DT_STRING:                                                    \
      (_result)->v.stringV = (char *) malloc(strlen(_input->v.stringV));       \
      strcpy((_result)->v.stringV, _input->v.stringV);                 \
      break;                                                           \
    case DT_FLOAT:                                                     \
      (_result)->v.floatV = _input->v.floatV;                          \
      break;                                                           \
    case DT_BOOL:                                                      \
      (_result)->v.boolV = _input->v.boolV;                            \
      break;                                                           \
    }                                                                  \
} while(0)
#define MAKE_BINOP_EXPR(_result,_left,_right,_optype)                  \
    do {                                                               \
      Operator *_op = (Operator *) malloc(sizeof(Operator));           \
      _result = (Expr *) malloc(sizeof(Expr));                         \
      _result->type = EXPR_OP;                                         \
      _result->expr.op = _op;                                          \
      _op->type = _optype;                                             \
      _op->args = (Expr **) malloc(2 * sizeof(Expr*));                 \
      _op->args[0] = _left;                                            \
      _op->args[1] = _right;                                           \
    } while (0)
#define MAKE_UNOP_EXPR(_result,_input,_optype)                         \
  do {                                                                 \
    Operator *_op = (Operator *) malloc(sizeof(Operator));             \
    _result = (Expr *) malloc(sizeof(Expr));                           \
    _result->type = EXPR_OP;                                           \
    _result->expr.op = _op;                                            \
    _op->type = _optype;                                               \
    _op->args = (Expr **) malloc(sizeof(Expr*));                       \
    _op->args[0] = _input;                                             \
  } while (0)
#define MAKE_ATTRREF(_result,_attr)                                    \
  do {                                                                 \
    _result = (Expr *) malloc(sizeof(Expr));                           \
    _result->type = EXPR_ATTRREF;                                      \
    _result->expr.attrRef = _attr;                                     \
  } while(0)

#define MAKE_CONS(_result,_value)                                      \
  do {                                                                 \
    _result = (Expr *) malloc(sizeof(Expr));                           \
    _result->type = EXPR_CONST;                                        \
    _result->expr.cons = _value;                                       \
  } while(0)
#endif // EXPR
```

We now discuss the interface of the record manager as defined in `record_mgr.h`. There are five types of functions in the record manager:

- functions for table and record manager management,
- functions for handling the records in a table,
- functions related to scans,
- functions for dealing with schemas, and
- function for dealing with attribute values and creating records.

We now discuss each of these function types

4.1. **Table and Record Manager Functions.** Similar to previous assignments, there are functions to initialize and shutdown a record manager. Furthermore, there are functions to create, open, and close a table. Creating a table should create the underlying page file and store information about the schema, free-space, ... and so on in the Table Information pages. All operations on a table such as scanning or inserting records require the table to be opened first. Afterwards, clients can use the `RM_TableData` struct to interact with the table. Closing a table should cause all outstanding changes to the table to be written to the page file. The `getNumTuples` function returns the number of tuples in the table.

4.2. **Record Functions.** These functions are used to retrieve a record with a certain `RID`, to delete a record with a certain `RID`, to insert a new record, and to update an existing record with new values. When a new record is inserted the record manager should assign an `RID` to this record and update the record parameter passed to `insertRecord`.

4.3. **Scan Functions.** A client can initiate a scan to retrieve all tuples from a table that fulfill a certain condition (represented as an `Expr`). Starting a scan initializes the `RM_ScanHandle` data structure passed as an argument to `startScan`. Afterwards, calls to the `next` method should return the next tuple that fulfills the scan condition. If `NULL` is passed as a scan condition, then all tuples of the table should be returned. `next` should return `RC_RM_NO_MORE_TUPLES` once the scan is completed and `RC_OK` otherwise (unless an error occurs of course). Below is an example of how a client can use a scan.

4.4. **Interface.** :

```
RM_TableData *rel = (RM_TableData *) malloc(sizeof(RM_TableData));
RM_ScanHandle *sc = (RM_ScanHandle *) malloc(sizeof(RM_ScanHandle));
Schema *schema;
Record *r = (Record *) malloc(sizeof(Record));
int rc;

// initialize Schema schema (not shown here)
// create record to hold results
createRecord(&r, schema);
// open table R for scanning
openTable(rel, "R");

// initiate the scan passing the scan handle sc
startScan(rel, sc, NULL);

// call next on the RM_ScanHandle sc to fetch next record into r
while((rc = next(sc, r)) == RC_OK)
  {
  // do something with r
  }

// check whether we stopped because of an error or because the scan was finished
if (rc != RC_RM_NO_MORE_TUPLES)
  // handle the error

// close scanhandle & table
closeScan(sc);
closeTable(rel);
```

Closing a scan indicates to the record manager that all associated resources can be cleaned up.

4.5. **Schema Functions.** These helper functions are used to return the size in bytes of records for a given schema and create a new schema.

4.6. **Attribute Functions.** These functions are used to get or set the attribute values of a record and create a new record for a given schema. Creating a new record should allocate enough memory to the data field to hold the binary representations for all attributes of this record as determined by the schema.

4.7. **Interface.** :

```
#ifndef RECORD_MGR_H
#define RECORD_MGR_H

#include "dberror.h"
#include "expr.h"
#include "tables.h"

// Bookkeeping for scans
typedef struct RM_ScanHandle
{
  RM_TableData *rel;
  void *mgmtData;
} RM_ScanHandle;

// table and manager
extern RC initRecordManager (void *mgmtData);
extern RC shutdownRecordManager ();
extern RC createTable (char *name, Schema *schema);
extern RC openTable (RM_TableData *rel, char *name);
extern RC closeTable (RM_TableData *rel);
extern RC deleteTable (char *name);
extern int getNumTuples (RM_TableData *rel);

// handling records in a table
extern RC insertRecord (RM_TableData *rel, Record *record);
extern RC deleteRecord (RM_TableData *rel, RID id);
extern RC updateRecord (RM_TableData *rel, Record *record);
extern RC getRecord (RM_TableData *rel, RID id, Record *record);

// scans
extern RC startScan (RM_TableData *rel, RM_ScanHandle *scan, Expr *cond);
extern RC next (RM_ScanHandle *scan, Record *record);
extern RC closeScan (RM_ScanHandle *scan);

// dealing with schemas
extern int getRecordSize (Schema *schema);
extern Schema *createSchema (int numAttr, char **attrNames, DataType *dataTypes, int *typeLength, int keySize, int
    *keys);
extern RC freeSchema (Schema *schema);

// dealing with records and attribute values
extern RC createRecord (Record **record, Schema *schema);
extern RC freeRecord (Record *record);
extern RC getAttr (Record *record, Schema *schema, int attrNum, Value **value);
extern RC setAttr (Record *record, Schema *schema, int attrNum, Value *value);

#endif // RECORD_MGR_H
```

5. Optional Extensions

You can earn up to `20%` bonus points for implementing optional extensions. A good implementation of one or two extensions will give you the maximum of `20%` points. So rather than implementing `5` incomplete extensions, I suggest you to focus on one extension first and if there is enough time, then add additional ones.

- `TIDs and tombstones`: Implement the TID and Tombstone concepts introduced in class. Even though your implementation does not need to move around records, because they are fixed size, TIDs and Tombstones are important for real systems.
- `Null values`: Add support for SQL style `NULL` values to the data types and expressions. This requires changes to the expression code, values, and binary record representation (e.g., you can use the NULL bitmaps introduced in class).
- `Check primary key constraints`: On inserting and updating tuples, check that the primary key constraint for the table holds. That is you need to check that no record with the same key attribute values as the new record already exists in the table. Ordered scans: Add an parameter to the scan that determines a sort order of results, i.e., you should pass a list of attributes to sort on. For dare-devils: Implement this using external sorting, so you can sort arbitrarily large data.
- `Interactive interface`: Implement a simple user interface. You should be able to define new tables, insert, update, and delete tuples, and execute scans. This can either be a shell or menu-based interface.
- `Conditional updates using scans`: Extend the scan code to support updates. Add a new method `updateScan` that takes a condition (expression) which is used to determine which tuples to update and a pointer to a function which takes a record as input and returns the updated version of the record. That is the user of the `updateScan` method should implement a method that updates the record values and then pass this function to `updateScan`. Alternatively, extend the expression model with new expression types (e.g., adding two integers) and let `updateScan` take a list of expressions as a parameter. In this case the new values of an updated tuple are produced by applying the expressions to the old values of the tuple. This would closer to real SQL updates.

6. Source Code Structure

You source code directories should be structured as follows. You should reuse your existing storage manager and buffer manager implementations. So before you start to develop, please copy your storage manager and buffer manager implementations.

- Put all source files in a folder `assign3` in your git repository
- This folder should contain at least
  - the provided header and `C` files
  - a make file for building your code `Makefile`.
  - a bunch of `*.c` and `*.h` files implementing the record manager
  - `README.txt/README.md`: A markdown or text file with a brief description of your solution

  Example, the structure may look like that:

```
git
   assign3
       Makefile
       buffer_mgr.h
       buffer_mgr_stat.c
       buffer_mgr_stat.h
       dberror.c
       dberror.h
       expr.c
       expr.h
       record_mgr.h
       rm_serializer.c
       storage_mgr.h
       tables.h
       test_assign3_1.c
       test_expr.c
       test_helper.h
```

- `test_helper.h`
  - Defines several helper methods for implementing test cases such as `ASSERT_TRUE`.
- `test_expr.c`
  - This file implements several test cases using the `expr.h` interface. Please let your make file generate a `test_expr` binary for this code. You are encouraged to extend it with new test cases or use it as a template to develop your own test files.
- `test_assign3_1.c`
  - This file implements several test cases using the `record_mgr.h` interface. Please let your make file generate a `test_assign3` binary for this code. You are encouraged to extend it with new test cases or use it as a template to develop your own test files.