CS564 Fall 2014
Assignment 4 - Front-End and DB Utilities

Due: 11:30PM on Friday, 12th December 2014
Individual Assignment for Epic Section
At most 2 people in a team for On-Campus Section

Introduction

For this part of the project, you will implement the front-end and database utilities of Minirel. You will also have to design and implement the Minirel catalog relations. To help you implement the front-end quickly, we will provide a parser to parse user commands and (a subset of) SQL. The parser works by reading in one line from stdin, parsing the line and making the appropriate calls to the back-end.

Database Utilities

There are three database utilities each of which can be run as a standalone UNIX program

- dbcreate <dbname>
- dbdestroy <dbname>
- minirel <dbname>

We have provided complete implementations of *dbcreate <dbname>* and *dbdestroy <dbname>* and a skeleton of *minirel <dbname>* functions for these three routines in the files minirel.cpp, dbcreate.cpp, and dbdestroy.cpp respectively. Each of these executables can be created by giving appropriate arguments to `make`.

1. dbcreate <dbname>

This creates the database *dbname*. You should examine this code carefully as it will give you some hints about how to implement operations like *create relation*. The program first creates a unix directory to hold the database and then changes the working directory to that directory. Next it creates two relations called RelCatalog and AttrCatalog for the database catalogs. First, of course, it creates these two relations by creating instances of the RelCatalog and AttrCatalog classes. More details about this will be given in the section on Catalogs.

2. dbdestroy <dbname>

This destroys the database *dbname*. This is done by using the following UNIX command:

```
ostrstream sysString;
sysString << "rm -r " << dbname;
system(sysString.str());
```

By doing this, the directory *dbname* and all the files (relations) it contains will be deleted.

3. minirel <dbname>

This is the main routine which is used to access and query the Minirel database *dbname*. When *minirel* is invoked with a dbname, the very first thing it does is to change its working UNIX directory (using `chdir`) to *dbname*. Each Minirel database corresponds to a UNIX directory and each relation within that database corresponds to a file within the directory. The *minirel* utility starts off by creating a buffer manager, opening the relation and attribute catalogs and then calling the function *parse()*.

Important: at this point global variable relCat refers to the relation catalog table and global variable attrCat refers to the attribute catalog table. Keep in mind these two variables, because when you implement various methods of class RelCatalog, you may have to refer to the attribute catalog table, and vice versa, when you implement various methods of class AttrCatalog, you may have to refer to the relation catalog table.

With the call to *parse()*, an infinite loop will start prompting the user for a new command, calling the appropriate routines of the lower layers to execute the command and then prompting the user again. The parser (provided by us) will be responsible for generating the prompt. After a command is parsed, it is transformed by the *interp()* function (in file parser/interp.cpp) into a sequence of operators (e.g. selects, joins, creates, destroys, prints, loads etc.), which are then executed one by one by the database back-end. Executing an operator involves calling some routine which you have or will implement in other parts of the project. For example, when the front-end receives a select-project query, the parser will call a procedure named *select()* that will in turn call the methods of the HeapFileScan class to scan the appropriate heapfile.

SQL DDL commands

Once minirel has been invoked, it can be used to either answer queries (using SQL DML) or to perform various utility operations (via SQL DDL). (Note: DML = Data Manipulation Language, and DDL = Data Definition Language.) Part5 of the project is mainly concerned with implementing the front-end utilities necessary to support a part of SQL DDL. The syntax of these commands will be described using a pseudo context-free grammar. All commands to the Minirel front-end must end with a semi-colon.

Front-End Command Syntax

```
<SQL_COMMAND> ::= <DDL_STATEMENT>;
               | <DML_STATEMENT>;

<DDL_STATEMENT> ::= <CREATE>
                  | <DESTROY>
                  | <LOAD>
                  | <PRINT>
                  | <HELP>
                  | <QUIT>
```

We now describe the semantics of each utility in detail.

```
<CREATE> ::= CREATE TABLE relName (attr1 format1, ..., attrN formatN);
```

The *create* command creates the relation *relName* with the given schema. Relation and attribute names can be at most MAXNAME = 32 characters long. Each format argument has a type specification ('int', 'real' , 'char' or 'char' followed by an integer inside a pair of parenthesis specifying the length of the string. e.g. 'char(20)'). For the 'char' format valid lengths are between 1 to 255 (both inclusive). This command results in an invocation of the RelCatalog::createRel() method that updates the catalogs appropriately and then creates a HeapFile to store the tuples of the relation. *Thus, once you have implemented the method RelCatalog::createRel(), parse() will call that method to execute this command.*

```
<DESTROY> ::= DESTROY TABLE relName;
```

The *destroy* command should destroy the HeapFile for relation relName. Finally it should remove all relevant catalog entries for that relation. All this is done by invoking the RelCatalog::destroyRel method. *Thus, once you have implemented the method RelCatalog::destroyRel, parse() will call that method to execute this command.*

```
<PRINT> ::= PRINT TABLE relName;
```

The *print* routine prints the contents of the specified relation in a reasonably tidy format. This is done by the UT_Print function. *Thus, once you have implemented the function UT_Print, parse() will call that function to execute this command.*

```
<LOAD> ::= LOAD TABLE relname FROM (filename);
```

The load utility bulk loads tuples into a relation (which must have already been created) using data in a binary UNIX file named filename. The load command begins by examining the relation catalog to determine length of the tuple length. Then it opens the UNIX file filename containing the input data. Each input tuple is in binary format (i.e. there is no space wasted for delimiters). The next step is to create an instance of the InsertFileScan class, passing the relname as parameter which cause the appropriate HeapFile to be opened. Finally, one tuple at a time should be read from the UNIX file and inserted into the HeapFile using the InsertFileScan::insertRecord() method. After loading all the tuples, the UNIX file should be closed and the InsertFileScan object should be deleted. This will result in the underlying DB files being closed. All this happens in the UT_Load function (described below). *Thus, once you have implemented the function UT_Load, parse() will call that function to execute this command.*

```
<HELP> ::= HELP [TABLE relName];
```

If a relname is not specified, help lists the names of all the relations in the database. Otherwise, it lists the name, type, length and offset of each attribute together with any other information you feel is useful. This is done by the RelCatalog::help function. *Thus, once you have implemented the function RelCatalog::help, parse() will call that function to execute this command.*

```
<QUIT> ::= QUIT;
```

The quit command exits minirel, causing all buffers and relations to be flushed to disk. Prior to terminating, the system should close down any open files (including catalogs). This is achieved by the UT_Quit function. *Thus, once you have implemented the function UT_Quit, parse() will call that function to execute this command.*

Summary: To implement the above commands, you will have to implement RelCatalog::createRel(), RelCatalog::destroyRel(), RelCatalog::help(), UT_Print(), UT_Load(), and UT_Quit(), among others. These methods and functions are described below.

Implementing Catalogs

One of the neat things about a relational database is that in addition to the data, even the metadata that describe what relations exist in the database, the types of their attributes and so on are also stored in relations, called catalogs. Minirel has two catalog relations: *relcat* and *attrcat*. The relcat relation contains one tuple for every relation in the database (including itself). The attrcat relation contains one tuple for every attribute of every relation (including the catalog relations) and this tuple contains information about the attribute. Both attrcat and relcat are created by the dbcreate utility and together they contain the schema of the database. Since each database has its own schema, relcat and attrcat must be placed in the UNIX directory corresponding to the database. relcat and attrcat are instances of the RelCatalog and AttrCatalog classes respectively.

The RelCatalog Class

```
#define RELCATNAME "relcat"     // name of relation catalog
#define ATTRCATNAME "attrcat"   // name of attribute catalog
#define MAXNAME 32              // length of relName, attrName
#define MAXSTRINGLEN 255        // max length of string attribute

// schema for tuples in the relation relcat:
// relation name : char(32) <-- lookup key
// attribute count : integer(4)
typedef struct {
    char relName[MAXNAME];      // relation name
    int attrCnt;                // number of attributes
} RelDesc;

class RelCatalog : public HeapFile {
        public:
                // open relation catalog
                RelCatalog(Status& status);

                // get relation descriptor for a relation
                const Status getInfo(const string& relName, RelDesc& record);

                // add information to catalog
                const Status addInfo(RelDesc& record);

                // remove tuple from catalog
                const Status removeInfo(const string& relName);

                // create a new relation
                const Status createRel(const string& relName,
                                       const int attrCnt,           // number of elements in attrList[]
                                       const attrInfo attrList[]);  // see defn of attrInfo below

                // define attrInfo
                typedef struct {
                        char relName[MAXNAME];     // relation name
                        char attrName[MAXNAME];    // attribute name
                        int attrType;              // INTEGER, FLOAT, or STRING
                        int attrLen;               // length of attribute in bytes
                        void *attrValue;           // ptr to binary value
                } attrInfo;

                // destroy a relation
                const Status destroyRel(const string& relName);

                // print catalog information (relation may be an empty string)
                const Status help(const string& relName);

                // get rid of catalog
                ~RelCatalog();
};
```

`RelCatalog::RelCatalog()`

The constructor just invokes the HeapFile constructor with the constant RELCATNAME (declared above).

`RelCatalog::~RelCatalog()`

You need to do nothing here as the underlying HeapFile will be close automatically.

`const Status RelCatalog::createRel(const string& relName, const int attrCnt, const attrInfo attrList[])`

First make sure that a relation with the same name doesn't already exist (by using the getInfo() function described below). Next add a tuple to the relcat relation. Do this by filling in an instance of the RelDesc structure above and then invoking the RelCatalog::addInfo() method. Third, for each of the attrCnt attributes, invoke the AttrCatalog::addInfo() method of the attribute catalog table (**remember that this table is referred to by the global variable attrCat**), passing the appropriate attribute information from the attrList[] array as an instance of the AttrDesc structure (see below). Finally, create a HeapFile instance to hold tuples of the relation (hint: there is a procedure to do this which we have seen in the last project stage; you need to give it a string that is the relation name). *Implement this function in create.cpp*

`const Status RelCatalog::destroyRel(const string& relName)`

First remove all relevant schema information from both the relcat and attrcat relations. Then destroy the heapfile corresponding to the relation (hint: again, there is a procedure to destroy heap file that we have seen in the last project stage; you need to give it a string that is the relation name). *Implement this function in destroy.cpp*

`const Status RelCatalog::addInfo(RelDesc& record)`

Adds the relation descriptor contained in *record* to the relcat relation. RelDesc represents both the in-memory format and on-disk format of a tuple in relcat. First, create an InsertFileScan object on the relation catalog table. Next, create a record and then insert it into the relation catalog table using the method insertRecord of InsertFileScan. *Implement this function in catalog.cpp*.

`const Status RelCatalog::getInfo(const string& relName, RelDesc& record)`

Open a scan on the relcat relation by invoking the startScan() method on itself. You want to look for the tuple whose first attribute matches the string relName. Then call scanNext() and getRecord() to get the desired tuple. Finally, you need to memcpy() the tuple out of the buffer pool into the return parameter *record*. *Implement this function in catalog.cpp*.

`const Status RelCatalog::removeInfo(const string& relName)`

Remove the tuple corresponding to relName from relcat. Once again, you have to start a filter scan on relcat to locate the rid of the desired tuple. Then you can call deleteRecord() to remove it. *Implement this function in catalog.cpp*

`const Status RelCatalog::help(const string& relName)`

If relation.empty() is true (empty() is a method on the string class), print (to standard output) a list of all the relations in relcat (including how many attributes it has). Otherwise, print all the tuples in attrcat that are relevant to relName. *Implement this function in help.cpp*

The AttrCatalog Class

```
// schema of tuples in the attribute catalog:
typedef struct {
        char relName[MAXNAME];  // relation name
        char attrName[MAXNAME]; // attribute name
        int  attrOffset;        // attribute offset
        int  attrType;          // attribute type
        int  attrLen;           // attribute length
} AttrDesc;

class AttrCatalog : public HeapFile {
        friend RelCatalog;

        public:
                // open attribute catalog
                AttrCatalog(Status& status);

                // get attribute catalog tuple
                const Status getInfo(const string& relName,
                                     const string& attrName,
                                     AttrDesc& record);

                // add information to catalog
                const Status addInfo(AttrDesc& record);

                // remove tuple from catalog
                const Status removeInfo(const string& relName,
                                        const string& attrName);

                // get all attributes of a relation
                const Status getRelInfo(const string& relName,
                                        int& attrCnt,
                                        AttrDesc*& attrs);

                // delete all information about a relation
                const Status dropRelation(const string & relName);

                // close attribute catalog
                ~AttrCatalog();
};
```

`AttrCatalog::AttrCatalog(Status& status)`

Just call the new HeapFile constructor (see below), passing the constant ATTRCATNAME (defined above).

`AttrCatalog::~AttrCatalog()`

Do nothing. File gets closed automatically by HeapFile destructor.

`const Status AttrCatalog::getInfo(const string& relName, const string& attrName, AttrDesc &record)`

Returns the attribute descriptor record for attribute attrName in relation relName. Uses a scan over the underlying heapfile to get all tuples for relation and check each tuple to find whether it corresponds to attrName. (Or maybe do it the other way around !) This has to be done this way because a predicated HeapFileScan does not allow conjuncted predicates. Note that the tuples in attrcat are of type AttrDesc (structure given above). *Implement this function in catalog.cpp*

`const Status AttrCatalog::getRelInfo(const string& relName, int& attrCnt, AttrDesc*& attrs)`

While getInfo() above returns the description of a single attribute, this method returns (by reference) descriptors for all attributes of the relation via *attr*, an array of AttrDesc structures, and the count of the number of attributes in *attrCnt*. The attrs array is allocated by this function, but it should be deallocated by the caller. *Implement this function in catalog.cpp*.

`const Status AttrCatalog::addInfo(AttrDesc & record)`

Adds a tuple (corresponding to an attribute of a relation) to the attrcat relation. *Implement this function in catalog.cpp*.

`const Status AttrCatalog::removeInfo(const string& relName, const string& attrName)`

Removes the tuple from attrcat that corresponds to attribute attrName of relation. *Implement this function in catalog.cpp*.

`const Status AttrCatalog::dropRelation(const string & relName)`

Deletes all tuples in attrcat for the relation relName. Again use a scan to find all tuples whose relName attribute equals relation and then call deleteRecord() on each one. *Implement this function in destroy.cpp*

Implementing Utility Routines

When the Minirel parser reads in a command, it will parse the command and call the necessary function to execute it. Some of the front-end DDL commands such as create and destroy are implemented by invoking the appropriate methods on the catalog classes. The implementation of the other utility functions load, print and quit are described below.

`void UT_quit(void)`

This function should simply delete the RelCatalog and AttrCatalog objects (causing the underlying files to be closed), delete the BufMgr object (causing any remaining dirty pages to be flushed to disk) and then terminate the minirel program by calling exit(). *Implement this function in quit.cpp*

`const Status UT_load(const string& relName, const string& fileName)`

Loads the tuples from the UNIX file fileName into the specified relation. The relation must already exist before UT_load is invoked. *Implement this function in load.cpp*

`const Status UT_print(const string & relName)`

Prints the values of all the attributes of all tuples of the specified relation. We will give you this code.

EXTRA CREDIT

Minirel Query and Update Operators

For extra credit, you will implement facilities for querying or updating Minirel databases. Specifically, you will implement one of selection, or update. You will use the same parser as the one used for utilities. For extra credit, you should implement either the Query DML statement (modify QU_Select) **OR** the Update DML statement (modify QU_Insert AND QU_delete) but not both. The meaning of the Query and Update DML statements are explained below.

SQL DML Commands

The syntax of this language is described below using pseudo context-free grammar that is a continuation of the grammar described above. Recall that optional parts of statements are enclosed in square brackets.

A DML statement can either be a query or an update.

```
<DML_STATEMENT> ::= <QUERY>
                  | <UPDATE>
```

We now describe the format of a query.

```
<QUERY> ::= SELECT <TARGET_LIST> [into relname] FROM <TABLE_LIST> [WHERE <QUAL>]
```

The result of a query is either printed on the screen or stored in a relation named *relname*. To simplify the first case (i.e. print to the screen), the Minirel query interpreter creates a temporary relation which it then prints and destroys. Thus, in both cases, the name of a result relation is provided and will be passed to function SQ_Select (see below). If no qualification is specified, then the query simply prints the columns requested in the target list.

```
<TARGET_LIST> ::= (relname1.attr1, relname2.attr2, ..., relnameN.attrN)
```

This is a list of attributes attr1, attr2, â€¦, attrN from relations relname1, relname2,... relnameN respectively that constitutes the target list of a query. All attributes in the target list should come from the same relation, except when the qualification is a join, in which case the attributes can come from the two relations begin joined . The "relnameN" qualifier can only be omitted if the query is on a single relation. It can be an alias of a relation if the alias is defined in TABLE_LIST. Please refer to <TABLE_LIST> (see below) for more of relation alias. The projection should be performed *on the fly* while the qualification is being evaluated, i.e. while the selection or join is being processed. You **should not** create a temporary relation with the result of the selection or join and then project on the desired columns. You do not have to worry about eliminating duplicates.

```
<TABLE_LIST> ::= (relname1 [alias1], relname2 [alias2],... relnameN [aliasN])
```

This list defines the target table list of a query. Aliases for each relation can be specified optionally. Once an alias is specified, it can be used as relation qualifier in TARGET_LIST and QUAL clause.

```
<QUAL> ::= <SELECTION>
         | <JOIN>
```

Qualifications are very simple, either a selection or a join clause.

```
<SELECTION> ::= relname.attr <OP> value
```

A selection condition compares an attribute with a constant. The "relname" qualifier can be omitted if the query is on a single relation, or can be an alias if the alias is defined in TABLE_LIST clause.

```
<JOIN> ::= relname1.attr1 <OP> relname2.attr2
```

A join condition compares the join attribute value of one relation with another. The "relname" qualifier can be an alias if the alias is defined in TABLE_LIST clause. Non-equi joins can only be performed using the nested loops join method.

```
<UPDATE> ::= <DELETE>
           | <INSERT>
```

An update is either an insert or a delete operation i.e. you cannot use Minirel to modify the value of an existing tuple in a relation.

```
<DELETE> ::= DELETE FROM relname [where <SELECTION>]
```

This operations specifies the deletion of tuples that satisfy the specified selection condition.

```
<INSERT> ::= INSERT INTO relname (attr1, attr2,... attrN) VALUES (val1, val2,... valN)
```

This will insert the given values as a tuple into the relation relname. Note that the values of the attributes may need to be reordered before the insertion is performed in order to conform to the offsets specified for each attribute in the AttrCat table. You can do this by using memcpy to move each attribute in turn to its proper offset in a temporary array before calling insertRecord.

Implementing the Relational Operators

For this part of the project you will need to implement the following routines. They will be called by the parser with the appropriate parameter values in response to various SQL statements submitted by the user. Thus you should not add or remove parameters unless you are prepared to modify the parser.

```
const Status QU_Select(const string& result, const int projCnt, const attrInfo projNames[], const attrInfo* attr, const Operator op, const char *a
```

A selection is implemented using a filtered HeapFileScan. The result of the selection is stored in the result relation called *result (a* heapfile with this name will be created by the parser before QU_Select() is called). The project list is defined by the parameters *projCnt* and *projNames*. Projection should be done on the fly as each result tuple is being appended to the result table. A final note: the search value is always supplied as the character string *attrValue*. You should convert it to the proper type based on the type of attr. You can use the atoi() function to convert a char* to an integer and atof() to convert it to a float. If attr is NULL, an unconditional scan of the input table should be performed.

```
const Status QU_Join(const string& result, const int projCnt, const attrInfo projNames[], const attrInfo *attr1, const Operator op, const attrInfo
```

The two attributes being compared are *attr1* and *attr2*. As with the select operator, the result table will be created by the parser before QU_Join is called. The names of the two relations being joined can be found in the relName component of the two attrInfo[] parameters. If the two relations being joined have attributes with the same name, one of them will disambiguated using the suffix "_1" in the result schema. You should perform projection on the fly using information given in projCnt and projNames.

You need not do anything in this file for extra credit as we have already implemented simple nested loops join for you. However, the implementation for Sort-Merge join and Hash join are left empty. You do not need to implement these for extra credit. **Adding anything in these two methods will not count towards extra credit. However, if you do choose to implement them out of interest, you can specify the join type in the invocation of the minirel utility on the command line by providing an extra third parameter of 'SM' for sort merge or 'HJ' for hash join.** We have set it up now so that all joins are simple nested loops.

```
const Status QU_Delete(const string& relation, const string& attrName, const Operator op, const Datatype type, const char *attrValue)
```

This function will delete all tuples in relation satisfying the predicate specified by *attrName*, *op,* and the constant *attrValue*. *type* denotes the type of the attribute. You can locate all the qualifying tuples using a filtered HeapFileScan.

```
const Status QU_Insert(const string& relation, const in attrCnt, const attrInfo attrList[])
```

Insert a tuple with the given attribute values (in *attrList*) in relation. The value of the attribute is supplied in the *attrValue* member of the attrInfo structure. Since the order of the attributes in attrList[] may not be the same as in the relation, you might have to rearrange them before insertion. If no value is specified for an attribute, you should reject the insertion as Minirel does not implement NULLs.

END OF EXTRA CREDIT

Error Handling

You should continue to use the same error handling mechanism that you used for the previous parts of the project. Feel free to augment this with new error codes and messages as needed.

Reminder

Remember to clean up after yourself in your code: if you allocate an object, a scan, or a file handle, then you are responsible to free it or close it. Many (painful) bugs experienced in the minirel projects are due to a failure to follow this rule. By the way, this does **not** mean that that every object will be created and free'd in the same block of code. For example, it is possible that a block of code will allocate an object, and the caller of the block of code will have the responsibility to free it.

Getting Started

Start by making a copy of front_end_utilities.tar.gz from the Assignment 4 directory in Learn@UW. In it you will find the following files along with files used in the lower layers:

- `minirel.cpp` - Main program for Minirel. Can be used without modification.
- `dbcreate.cpp, dbdestroy.cpp` - Main programs for these utilities. We have provided complete implementations of these functions.
- `create.cpp, destroy.cpp` - Stubs of catalog functions for creating and dropping relations. **You have to fill these out.**
- `catalog.cpp` - Stubs for other catalog functions. **You have to implement these.**
- `load.cpp` - Utility for loading a relation from a UNIX file. **You have to implement this.**
- `print.cpp` - Utility for printing the data in a relation.
- `help.cpp` - Utility for printing out schema. **You have to implement this.**
- `quit.cpp` - Utility for cleaning up and exiting Minirel.
- `join.cpp` - Contains implementation of Simple nested loops join.
- `select.cpp, insert.cpp, delete.cpp` - Stubs for functions to be used in extra credit portion. Change this file iff you are doing extra credit work.
- Other `.h` files - These contain the relevant class definitions and function prototypes. You need not modify these.
- `Makefile` - To compile this part of the project.
- Files in the parser subdirectory - This contains the SQL parser and interpreter. You should not modify these.

Testing

In ths directory you will find a shell program called **uttest** along with two directories uttestqueries and uttestdata. If you look at the files in the directory uttestqueries you will notice that they contain a progressively more difficult series of tests. The directory uttestdata contains some small tables that will get loaded by the test queries.
To run all the tests, simply invoke **uttest** while in the part 5 directory. If you want to run one test at a time simply type **uttest i** where where i is a number between 1 and 10. For example, **uttest 3** will run the test in the file ut.3 in the uttestqueries directory.

Testing for Extra Credit Work

The test files are named qu.1, qu.2, etc. in directory "testqueries". Feel free to augment these files and use the augmented versions to further evaluate your code.
Certain files in "testqueries" test joins, supposedly with indexing, except that we have turned off the indexing step for now. You can use them to further test your code.
You can run the above files by manually feeding the commands in the files to your code, or you can modify the script **qutest**, and then use it to run the above files.

Handing In

To handin the stage of your project:

- Follow the same steps as the previous assignment and upload you tar file to the *Assignment 5 Submission* dropbox folder on Learn@UW.
- Do not forget to include a README file with your submission with your teammates name in it (If you worked in a team).
- Please remember it is due at 11:30PM on Friday, 12th December 2014. The late policy will be as mentioned here. If you do not have the assignment complete by that time, please turn in whatever you have in hopes of getting partial credit.
- If you attempt the extra credit section, please mention which of the two DML statements you have implemented in you README file (QUERY/UPDATE).