EECS 484, Fall 2013

Project 4: Minirel 2K Query Processor

Due: November 25 at 11:55PM

Introduction

In this project, you will implement a query processor and some basic utilities for the Minirel2K system. When your query processor is complete, you will have a simple single-user DBMS that accepts a (small) subset of SQL.

Basics

Once complete, your DBMS will include three executable programs, which you will be able to run from the command line:

1. **dbcreate <dbname>**
   This executable creates the database *dbname*.
2. **minirel <dbname> [SQL-file]**
   This executable allows you to interact with the database named *dbname* by writing (simple) SQL commands. If the optional *SQL-file* is specified, then the program reads SQL from that file.

   While industrial-strength DBMSs support concurrent access to data, for the purposes of this assignment, we will assume that there will be at most one minirel process running at a time, so we do not need to worry about concurrency control.
3. **dbdestroy <dbname>**
   This program deletes the database *dbname.*

Framework Overview

To get your started, we provide a skeleton framework, which consists of a significant quantity of source code, as well as some libraries. Figure 1 gives an overview of the framework. The following is a brief description of each of the main components.

Parser: The **minirel** executable accepts SQL queries and other utility commands. (See the next section for a full description of the supported SQL commands.) We provide a parser, which first parses the input SQL, and then consults the system catalogs to make sure the commands are valid (i.e., the relations and attributes mentioned by the command actually exist in the database.) If the SQL is valid, the parser calls the appropriate query operators and utilities. The parser is implemented for you.

Query Optimizer, Operators, and Utilities: This is where you will do most of your work. If the incoming SQL command is valid, the parser calls the appropriate function to process the command:

- If the command is a utility, the parser calls the appropriate function to process the utility. For this project, you will only be implementing one utility (insert). The function header (Updates::Insert) can be found in the file query.h, and the actual implementation of this function will go in insert.cpp.
- If the command is a query, the parser first determines if the query is a select query (referencing just one table) or a join query. To execute a select query, the parser calls Operators::Select (in select.cpp). To execute a join query, it calls Operators::Join (in join.cpp). The Operator class definition can be found in query.h.

For this assignment, you will implement the insert, select, and join operators.

Storage Manager: Beneath the query processor, there are two main *access methods* for data: indexes and heapfiles. From the query processor, you will need to appropriate public methods. To understand these classes and methods, it should be sufficient to look at the header files (heapfile.h and index.h). The storage manager is implemented for you.

System Catalogs: Recall that the system catalogs are used to store metadata ("data about the data"), including the names of all tables, and the names and types of all attributes. The system catalogs are implemented for you, but you will need to call them from the query processor and insert utility. To understand the classes and methods, it should be sufficient to look at the header file (catalog.h).

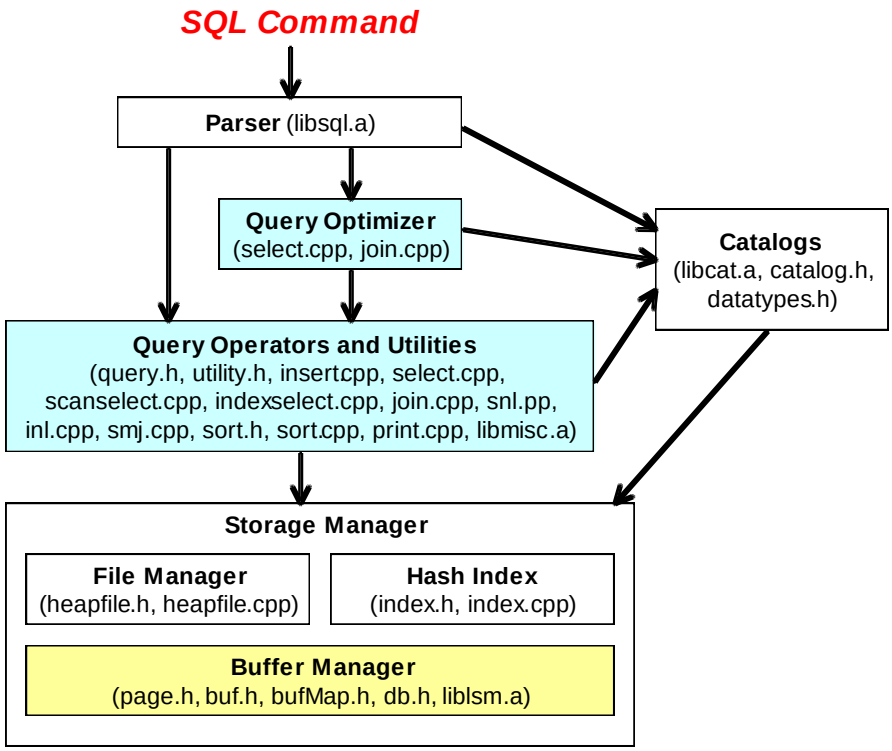

Figure 1: Architecture of Minirel2K

Finally, in Figure 1, notice that in some cases we provide you system components using Linux archive files (with .a extensions) rather than C++ source code (.cpp files). Since we use similar projects in different semester, we do not want to release code that constitutes a solution to another project. However, this should not be a problem for you. Notice that the provided Makefile is already configured to use the archive files. All necessary information about the provided functions (and how to use them) can be found by looking at the appropriate header (.h) files.

Supported SQL Statements and Parser

The SQL parser that we provide understands only a small subset of standard SQL:

1. **CREATE TABLE TBLNAME (ATTRLIST…)**
   Note that the parser does not support any key or table constraints. Supported data types include INTEGER, DOUBLE and CHAR; no attribute values can be null. For example, the following statement creates a table with three attributes:

CREATE TABLE DA (IKEY INTEGER, FILLER CHAR(80), DKEY DOUBLE)

2. **CREATE INDEX RELNAME (ATTRNAME)**
   As you may recall, SQL has no standard command for creating an index. This statement creates a ***hash*** index in Minirel2K. The code for hash index is provided by us (see files index.h and index.cpp).

3. **DROP TABLE RELNAME**
   Deletes the table *RelName*.

4. **DROP INDEX RELNAME(ATTRNAME)**
   Deletes the index on the *AttrName* attribute of the relation *RelName*.

5. **INSERT INTO RELNAME(ATTRNAMELIST) VALUES (VALUELIST)**
   The standard SQL "insert into" command. The only difference is that the AttrNameList is mandatory (not optional as in SQL). For example, the following statement inserts a tuple into the DA table: INSERT INTO DA (IKEY, FILLER, DKEY) VALUES (11, 'SING A SONG'', 111.0);

   When the parser detects an insert command, it calls the "Updates::Insert" method (see file query.h and insert.cpp). Values for doubles in all Minirel2K SQL must be specified in the form "Num.Num". Note that in the insert into statement above the value for *dkey* is specified as 111.0 and not 111.

6. **SELECT PROJLIST FROM RELLIST WHERE PREDICATE**:
   The SQL command for querying. Only a limited class of queries is supported. First, the *RelList* can have at most two relations. Second, the *Predicate* can only be a single predicate (no ANDs or ORs). Third, all attributes must be referenced using the form "RelName.AttrName", not just "AttrName". Fourth, all literal values for doubles must be in the form "Num.Num". Finally, there are no nested select statements.

   Put another way, the following two types of queries are supported:

   - **SELECT PROJLIST FROM RELNAME WHERE RELNAME.ATTR OP LITERAL**
     A simple select query on a single table. When the parser detects a select query, it calls the "Operators::Select" method (see file query.h and select.cpp).

   - **SELECT PROJLIST**
     FROM RELNAME1, RELNAME2
     WHERE RELNAME1.ATTR1 OP RELNAME2.ATTR2
     A simple join query on two tables. When the parser detects this join query, it calls the "Operators::Join" method (see file query.h and join.cpp).

7. **QUIT**:
   The quit command exits the Minirel2K system.

If the parser detects an error in the SQL statement, it prints out an error message and exits the system. Before exiting the system, it cleans up by calling destructors on the buffer manager and catalog objects. However the database may get corrupted if the system crashes at arbitrary points (this may happen when you are testing your code). Since we don't have a recovery manager in Minirel2K, you may have to rebuild the database if it is corrupted. To destroy a database run "dbdestroy" on the database, and then reload the database using your SQL commands.

Catalogs Implementation

Minirel2K has two heapfiles called *relcat* and *attrcat* that are used to store the system catalogs. 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. *relcat* and *attrcat* are instances of the **RelCatalog** and **AttrCatalog** classes respectively, and are derived from **HeapFileScan**. HeapFileScan in turn is derived from the class **HeapFile**.

You will need to use the catalogs to get information about the relations in the queries, the attributes in the relations, type information for the attributes, index information on attributes etc. The interface for the catalog relations is defined in *catalog.h*. There are two main classes: RelCatalog and AttrCatalog.

There are two global variables defined in the Minirel2K programs (minirel.cpp, dbcreate.cpp) which provide handles to the systems relation and attribute catalogs. Use these handles to invoke methods on the catalog classes. The declaration of these variables is:

```
RelCatalog *relCat;
AttrCatalog *attrCat;
```

The RelCatalog Class

This class manages the catalog about the relations. It stores the relation catalogs in a heapfile called "relcat" (defined by the variable RELCATNAME in the file catalog.h). There are two main functions that are defined on this class:

1. **const Status getInfo(const string & rName, RelDesc& record)**
   This method is used to retrieve the catalog information for the relation *rName.* This function returns via the second function argument, *record*, a RelDesc struct for the relation. For each relation in the system, the relation catalog stores in the underlying heapfile a record that has the structure defined by RelDesc. The RelDesc structure is defined as follows:

```
typedef struct {
  char relName[MAXNAME]; // relation name
  int attrCnt;           // number of attributes in the relation
  int indexCnt;          // number of indexed attrs
} RelDesc;
```

2. **const Status addInfo(RelDesc& record)**
   This function adds the relation described by *record*, to the relation catalog. Each member of the RelDesc struct becomes an attribute in a tuple, which is written to the heapfile named "relcat". RelDesc represents the in memory format of a tuple in "relcat" heapfile. The tuple format on disk is exactly the same as the in-memory format, since we have designed the RelDesc structure to have the same byte alignment in both the memory and the disk formats.

Both these functions return OK if there are no errors in executing the function, else they return some error code that can be printed using the error class defined in the files error.h and error.cpp.

In addition to the methods described above, there are a number of other methods defined on the RelCatalog class. You don't need to use any of the other methods for this assignment, however if you are curious about these other methods take a look at the definition of **RelCatalog** in the file *catalog.h*

The AttrCatalog Class

This class manages the catalog information about the attributes in the relations. For each attribute in each table in the database, there is a tuple stored in the heapfile called "attrcat" (defined by the variable ATTRCATNAME in catalog.h). The information stored in these tuples has the following C++ format:

```
typedef struct {
  char relName[MAXNAME];   // relation name
  char attrName[MAXNAME];  // attribute name
  int attrOffset;          // attribute offset
  int attrType;            // attribute type
```

```
  int attrLen;              // attribute length
  int indexed;              // TRUE (==1) if indexed
} AttrDesc;
```

Here *relName* is the name of relation to which the attribute belongs. *attrName* is the name of the attribute, *attrOffset* is the offset of the attribute value from the start of the record data as stored on disk. In Minirel2K's disk representation attributes are places next to each other without any byte-padding for aligning the attribute values at some fixed boundaries (such a 4 bytes or 8 bytes). This is typical in database implementations as it is important to have a compact representation of records on disk. *attrType* is the type of the attribute. The legal data types are defined in the file datatypes.h. *attrLen* is the length of the attribute and *indexed* indicates if an index has been created on the attribute.

There are three important functions that are defined on the AttrCat class:

1. **const Status getInfo(const string & rName, const string & attrName, AttrDesc &record)**
   Returns via the last argument, the attribute descriptor *record* for attribute *attrName* in relation *rName*.

2. **const Status addInfo(AttrDesc & record)**
   Adds a record describing an attribute to the attribute catalog. The information described in *record* is converted to a tuple and stored in the heapfile "attrcat".

3. **const Status getRelInfo(const string & rName, int &attrCnt, AttrDesc *&attrs)**
   This function is used to get information on all the attributes of a *relation.* Returns by reference an array of AttrDesc structures via the function argument *attrs,* and a count of the number of attributes via *attrCnt*. The *attrs* array is allocated by this function, and should be deallocated by the caller.

All these functions return OK if there are no errors, else they return an appropriate error code.

Your Assignment

For this project, your assignment is to implement important relational operators and utilities, namely Insert, Select, and Join. You can find the class definitions and function signatures in query.h. You will fill in the implementations of these operators (in files insert.cpp, select.cpp, join.cpp, indexselect.cpp, scanselect.cpp, snl.cpp, inl.cpp, and smj.cpp). The remainder of this section describes each of the functions in greater detail.

Insert

The signature for the insert function is in query.h, and you must fill in the code for this function in insert.cpp.

const Status Updates::Insert(const string & relation, const int attrCnt, const attrInfo attrList[])

This function inserts a tuple with the given attribute values (in *attrList*) into the specified relation. The type *attrInfo* is defined in the file catalog.h. The value of the attribute is in *attrList[I].attrValue*, and the name of the attribute is in *attrList[I].attrName*. For the SQL INTEGER (DOUBLE) data type, *attrValue* is a pointer to an integer (double). Similarly for an attribute of type STRING (SQL CHAR type), *attrValue* points to a character string.

The attrList array may not list the attributes in the same order as they appear in the relation, so you may have to rearrange the attribute values before inserting the tuple into the relation. **If no value is specified for an attribute in *attrList*, you should reject the insertion.** (In a real database system, missing attribute values are implemented using NULLs.)

In addition to inserting the tuple, this operator must also update all the hash indices on the relation. Use the Index::insertEntry to insert an entry into the index. Look up the system catalogs to find out information about the relation and attributes. (See Section "Getting Started" for more details.)

Select

The signature for the select function is in query.h, and you will fill in code to implement this function in select.cpp, indexselect.cpp, and scanselect.cpp.

const Status Operators:: Select(const string & result, const int projCnt, const attrInfo projNames[], const attrInfo* attr, const Operator op, const void *attrValue)

This function implements the select operator, selecting all tuples that match the input predicate. In your case, the predicate is specified via three variables: *attr, op,* and *attrValue*. (If *attr* is null, this means that the selection is unconditional.) You must implement two access methods for the select operator: one using a HeapFileScan and the other using an IndexScan. These access methods will be implemented as separate functions called **Operators::ScanSelect** (in the file scanselect.cpp) and **Operators::IndexSelect** (in the file indexselect.cpp).

The **Operators::Select** function must implement a very simple rule for "optimizing" select queries. It must check if an index exists on the attribute in the predicate, and if the predicate is an equality predicate. If both these conditions are met, it calls the **IndexSelect** function; otherwise it calls **ScanSelect**. This criteria works since hash indices are generally very efficient for evaluating equality predicates.

The results of the selection are inserted into a result file (a HeapFile) called *result*, which is created by the parser before calling Select. The names of the attributes of this relation are derived from the corresponding attributes in projNames, and have as suffix the attribute number.

Projection, defined by *projCnt* and *projNames*, should be done on the fly when each result tuple is being written out. Don't worry about eliminating duplicates during the projection.

Finally, the search value, *attrValue*, is a pointer to a value that has the same type as *attr*. For SQL data types INTEGER, DOUBLE, CHAR, the *attrValue* points to the C++ types int&, double&, and char* respectively.

Join

The signature for the join function is in query.h, and you will fill in code to implement this function in join.cpp, snl.cpp, inl.cpp, and smj.cpp.

const Status Operators:: Join(const string & result, const int projCnt, const attrInfo projNames[], const attrInfo *attr1, const Operator op, const attrInfo *attr2)

This function joins two relations based on the predicate specified using the variables "*attr1 op attr2"*. You must implement three join algorithms: simple nested-loops (not the page oriented or block nested-loops), indexed-nested loops and sort-merge join. These three algorithms must be implemented as separate function called **SNL**, **INL**, and **SMJ** respectively, in the files snl.cpp, inl.cpp and smj.cpp.

Operators::Join chooses amongst these alternatives based on the join predicate and the index availability on the join attributes. The order of preference for the algorithms is first **INL,** then **SMJ** and finally **SNL**. Collectively these algorithms allow evaluating both equi-joins and non-equi-joins. Non-equi-join must be processed using **SNL**. If it is an equi-join and an index exists on either *attr1* or *attr2*, you should use the **INL** join algorithm. If indices exist on both you can arbitrarily choose which index to use. Finally, if it is an equi-join and no indices exist on either of the join attributes, you should use the **SMJ** algorithm.

For implementing sort-merge join, you can use the sorting code provided in files sort.h and sort.cpp, but you will need to implement the merge phase. When performing a sort, you will use 80% of the unpinned pages for the sort. Start by calling the **BufMgr::numUnpinnedPages** to determine the number of pages that are unpinned in the buffer pool. Set the number of pages that you will use for the sort at 80% of this number (as with most resources performance starts going down because of thrashing if the resource is overcommitted). Now you will have a number, *k*, that tells you how many pages in the buffer pool you can use for the sort. Consult the system catalogs and determine the size of the tuple (in number of bytes). Since Minirel pages are 1KB (defined by PAGESIZE), you can calculate how many tuples, *n*, would be contained in *k* pages.

When merging two sorted files, to handle duplicates, you will sometimes need to move the scan **backwards**! The SortedFile class has two new functions ***setMark()*** which sets a marker at the current tuple being scanned. Call the ***setMark()*** only after you have called ***next()*** to get a record. A marker will be set on the record that was fetched by the last call to ***next()***. To go back to scanning from the mark point, use the method ***gotoMark()***, which moves the scan back to the last mark point, and retrieves the record at the marker. After calling ***gotoMark()***, you can call ***next()*** to keep scanning forward from the marker point.

The *HeapFileScan* class also has a new ***scanNext()*** method that fetches both the *rid* and the *record* of the next record. Use this ***scanNext()*** method if don't want to call the old ***scanNext()*** followed by ***getRecord()***.

Just as is the case of selection, the results of the join are inserted into a result file (a HeapFile) called *result*, which is created by the parser before calling Select.

Additional notes

- All results are printed using the code in the file *print.cpp.*
- All the methods that you add must return OK if there are no errors.
- The general rule for propagating errors when calling other Minirel2K methods is the same as before: If the method (caller) calls another Minirel2K method (callee), and if the callee returns an error code, then the caller should return back that same error code.
- In Minirel2K's disk representation of records, attributes are placed next to each other without any byte-padding for aligning the attribute values at some fixed boundaries. You must follow this packing when constructing result tuples that are written to the result heapfile and to the catalog heapfiles during the bootstrapping process. See the code in print.cpp to see how the attributes in the tuples are interpreted by the system.
- The methods in the buffer manger "getBufStats" and "clearBufStats" can be used to check the number of I/Os and buffer accesses that are incurred by your implementation.
- You do not need to match the statistics exactly. It is sufficient to be within approximately 50% of these statistics as it tells us you are doing things correctly.

Submitting Your Assignment

For this project, you should only need to modify the following files:

- insert.cpp: The code for inserting tuples goes here.
- query.h: You may add additional private methods here if you like to help implement the various operators.
- select.cpp, scanselect.cpp, indexselect.cpp: The code for selection goes here.
- join.cpp, snl.cpp, inl.cpp, smj.cpp: The code for join goes here.

It is very important that you do not add any additional files and that you do not modify the public APIs.

To submit your assignment, please follow these instructions:

1. Log into a CAEN Linux machine.
2. Create a directory called P4. Copy into this directory all of the files listed above, as well as a file called "report.pdf" which describes your testing methodology.
3. For testcases, make sure to include a sql/ folder that contains all the testcases that you have written.
4. Make sure your directory does not contain any other files (object files, executables, etc.)
5. From the directory immediately above P4, type the following command:

   > tar czvf P4.tgz P4/

   You will see a message indicating that a tar file is being created.
6. Use the CTools account of any single group member to submit your work. (Please only submit one version, and list the names of both partners in the comment box.) Upload the file P4.tgz into CTools using the "Upload local file" option. (Do not use the URL option!)

When grading, we will compile your code on a CAEN Linux machine using the latest version of the g++ compiler. Thus, before submitting, you should verify that your code compiles on a CAEN machine using our provided Makefile (even if you are using an IDE or another machine). We will also link your files with our (hidden) test driver, so it is very important to be faithful to the public API we provide.

Grading

The breakdown of the grading for this assignment is as follows:

1. **Correctness - 80%:** The correctness part of the grade will be based on the tests that we have provided, and additional (more rigorous) tests that we will run on your submitted projects.
2. **Programming Style and code clarity - 10%**: For your style points, we will check your code for readability (how easy is it to read and understand the code), and for the code organization (reasonable use of functions, etc.).
3. **Testing and Test report – 10%**: You should think carefully about testing so that various paths in your code are tested. Put any additional test cases in the sql directory and use the report to describe what your additional tests tested.

Getting Started

You should start by downloading the skeleton files from CTools. Before modifying any of the files, you should be able to compile using the provided Makefile on a CAEN Linux machine:

```
> make clean
> make
```

After doing this, you should be able to see three executables: *dbcreate*, *dbdestroy*, and *minirel*.

Let's get started by creating a test database called "testdb"

```
> dbcreate testdb
```

You can delete the database as follows

```
> dbdestroy testdb
```

Let's create it again and try using it

```
> dbcreate testdb
```

Since our database is stored as files in the Linux file system, you can check the contents of the directory "testdb." This is where the files storing data for testdb will reside. You will notice that there are already two files in this directory: "attrcat" and "relcat". These files store the system catalogs for the (initially empty) testdb database.

Next, let's create some user tables. The following executes the SQL commands in the file sql/insert.sql:

```
> minirel testdb sql/insert.sql
```

The first two commands in sql/insert.sql are

```
CREATE TABLE soaps(soapid integer, name char(32),
        network char(4), rating double);
CREATE TABLE stars(starid integer, real_name char(20),
        plays char(12), soapid integer)
```

After executing the command, you should see two additional files in the testdb directory.

```
> ls testdb
attrcat  relcat  soaps  stars
```

We just created two relations: soaps and stars. The CREATE TABLE command is already implemented for you. The catalog data for the two tables is also added to attrcat and relcat files.

The sql/insert.sql file also contains INSERT commands. Those commands are read by the parser (already implemented). The parser invokes the insert method in insert.cpp, but this method doesn't do anything yet. A good starting point is to implement the insert method first. That will orient you to using most of the other classes effectively, such as Catalog, Heapfile, and Index.

Let's look at what should happen for the first INSERT command in sql/insert.sql:

```
INSERT INTO stars(starid, real_name, plays, soapid)
    VALUES (100, 'Posey, Parker', 'Tess', 6);
```

To insert a record into "stars", the insert method needs to:

1. Insert the record into the Heapfile corresponding to "stars".
2. Insert the record ID into each index that exists on 'stars".

To insert the record, you first need to map the data from the arguments to insert into a Record object. The Record object is defined in page.h as:

```
struct Record
{
  void* data;
  int length;
};
```

The data that is passed in is in *attrList*, which is an array of *attrInfo* objects. attrInfo is defined as follows in catalog.h:

```
typedef struct {
```

```
  char relName[MAXNAME];              // relation name
  char attrName[MAXNAME];             // attribute name
  int  attrType;                      // INTEGER, DOUBLE, or STRING
  int  attrLen;                       // length of attribute in bytes
  void *attrValue;                    // ptr to binary value (used
                                      //    by the parser for insert into
                                      //    statements)
} attrInfo;
```

For the above INSERT command, the 4 elements of the array will be something like the following:

[<"stars", "starid", INTEGER, -1, pointer to 4 bytes containing 100>,
<"stars", "real_name" STRING, -1, pointer to "Posey, Parker">
<"stars", "plays", STRING, -1, pointer to "Tess">
<stars", "soapid", INTEGER, -1, pointer to 4 bytes containing 6>]

The attrCnt argument to insert.cpp will be 4. Note that strings are null-terminated C-strings.

We have to pack the above attribute values into a record for the relation. The CREATE command for stars specified the order of fields, types, etc. So, we have to look up the System Catalog to determine the attributes of "stars", their offset in the record, and their data type. Note that it is not necessary that the attrList array contains the attributes in the correct order.

attrCat is a global variable that points to the attribute table for the Catalog.

You can use **attrCat->getRelInfo(const string &rName, int &attrCnt, RelDesc& relattrs)** to retrieve information about the attributes of the "stars" relation. This will tell you the offset, data type, and the size of each attribute for packing into the record. In this case, we should expect to find that "starid" has offset 0 and length 4. "real_name" has offset 4 and length 20, "plays" has offset 24 and length 12, and "soapid" has offset 36 and length "4". The total length of the record becomes 40.

At this point, you have all the information to allocate the memory for a record, fill it with data, and then insert it in the heapfile. To allocate the memory for the record, you need to know the record size. Let's say the size of the record is determined to be 40. We need to create a Record object and then allocate 40 bytes for the data field of the record.

Then, the data field needs to be filled in with the data from the attrList. You will need to use the C++ built-in function *memcpy* to do that. You will need to rely on the data from the RelDesc object returned by the getInfo call to determine the offsets for each attribute and the # of bytes to copy.

Once the record is created, you can call insertRecord on the heapfile for the relation. First the constructor for the "stars" heapfile (which creates a heapfile if it does not exist, else opens it) needs to be called. Then, the record you created needs to be inserted. That will return a Record ID.

This record ID needs to be inserted in each index for the relation. To determine if there is one or more index for the relation, you need to look at the catalogs. Then, use the interfaces defined in index.h.

Make sure you free up memory when you are done with various objects. Either keep the objects, such as the index object, or free them up when you are done with them.

Try getting the insert() method working. Once you are able to insert records in your tables, you are on your way to going to the next step of implementing the SELECT operators. Implementing the JOIN will take substantial time. Make sure to start early, and set intermediate goals to avoid falling behind.