CMPS181_Project2_Description

See CMPS181_Project2 file for general information about Project 2, including information about submission of your Project

Introduction

In this project, you will continue implementing the record-based file manager (RBFM). Once you have finished implementing that, you will build a relation manager (RM) on top of the basic paged file system. The RM manager should meet the following basic requirements.

Basic Requirements (100 points)

Finish RBFM

You need to finish the implementation of the record-based file manager (RBFM) that you started in part 1. Specifically, you should finish implementing the following methods:

RecordBasedFileManager::deleteRecord(), **RecordBasedFileManager::updateRecord()**, **RecordBasedFileManager::readAttribute()**, and **RecordBasedFileManager::scan()**.

Please refer to Project 1 Description for the explanations of the methods, and look at the file **rbfm.h** for the signature of those methods. Once you finish the implementation of those methods and test them well, you will use those methods to implement the relation manager. Please have a look at the file **rm.h** for signatures of the releation manager methods.

Implement the Relation Manager

The RelationManager class is responsible for managing the database tables. It handles the creation and deletion of tables. It also handles the basic operations performed on top of a table (e.g., insert and delete tuples).

Catalog

Create a catalog, a set of tables that holds information about your database. This includes at least the following:

- Table information (e.g., table-name, table-id, etc.).
- For each table, the columns, and for each of these columns: the column name, type, length, and position.
- The name of the record-based file in which the data corresponding to each table is stored.

It is mandatory to store the catalog information by using the RBF layer functions. You should create the catalog's tables and populate them the first time your database is initialized (when the method createCatalog() is called). Once the catalog's tables and columns tables have been created, they should be persisted to disk. Please use the following name and type for the catalog tables and columns. You may add more attributes you want/need to. However, please do not change the given name of these two tables or their attribute names.

- `Tables (table-id:int, table-name:varchar(50), file-name:varchar(50))`
- `Columns(table-id:int, column-name:varchar(50), column-type:int, column-length:int, column-position:int)`

An example of the records that should be in these two tables after creating a table named "Employee" is:

```
Tables
(1, "Tables", "Tables")
(2, "Columns", "Columns")
(3, "Employee", "Employee")

Columns
(1, "table-id", TypeInt, 4 , 1)
(1, "table-name", TypeVarChar, 50, 2)
(1, "file-name", TypeVarChar, 50, 3)
(2, "table-id", TypeInt, 4, 1)
(2, "column-name",  TypeVarChar, 50, 2)
(2, "column-type", TypeInt, 4, 3)
(2, "column-length", TypeInt, 4, 4)
(2, "column-position", TypeInt, 4, 5)
(3, "empname", TypeVarChar, 30, 1)
(3, "age", TypeInt, 4, 2)
(3, "height", TypeReal, 4, 3)
(3, "salary", TypeInt, 4, 4)
```

Note that TypeInt, TypevarChar, and TypeReal are the Enumerated Type values used to for the attribute “column-type”, and they are defined in the rbfm.h file. (PostgreSQL information on SQL Enumerated Types is here.) The only column types that you’ll have to deal with are INT, VARCHAR and REAL.

Also for the file-name attribute, you can use your own naming conventions if you want (e.g,. "Tables.tbl" for Tables table). In the example above, the last row of the Tables table states that the table-id of the "Employee" table is 3, its name is "Employee", and the RBF file associated with this table is "Employee". The last line of the Columns table shows the information about the "salary" column of the "Employee" table. Specifically, the table-id is 3, the column-name is "salary", the type of this column is Int, its length is 4, and the position of this column is 4.

The catalog tables (Tables and Columns) should be created when the method createCatalog() is called. All other subsequent invocations of your database should use the already created catalog tables. The catalog tables should be deleted when the deleteCatalog() is called. Between the createCatalog() and deleteCatalog() methods, the catalog should be persistent on disk. Please note that users should be able to query your catalog's tables just like any other table, but users

should not be allowed to modify its content through the RM API. Modifications of the catalog tables should be only allowed via internal calls (e.g., as a side effect when a user table is created or deleted). To do this, you may need to have a "system" vs. "user" flag in your Tables catalog so you can distinguish "system tables" from "user tables", and make it illegal to do anything but read "system tables" through the RM layer. If you take this design approach, then you'll need to add one more column to the two schemas described above.

RelationManager class

The following is more information about the RelationManager class. Your program should create exactly one instance of this class, and all requests for the RM component should be directed to that instance. The public methods of the class declaration are shown first, followed by descriptions of the methods. The last two methods in the class declaration are the constructor and destructor methods and are not explained further. **Note:** your tuple-oriented file system must create a relation manager (RM) that initializes the catalog information you may need to store. It also internally creates a record-based file manager using the implementation from Project 1.

```
class RelationManager
{
public:
  RC createCatalog();

  RC deleteCatalog();

  RC createTable(const string &tableName, const vector<Attribute> &attrs);

  RC deleteTable(const string &tableName);

  RC getAttributes(const string &tableName, vector<Attribute> &attrs);

  RC insertTuple(const string &tableName, const void *data, RID &rid);

  RC deleteTuple(const string &tableName, const RID &rid);

  RC updateTuple(const string &tableName, const void *data, const RID &rid);

  RC readTuple(const string &tableName, const RID &rid, void *data);

  // mainly for debugging
  // Print a tuple that is passed to this utility method.
  RC printTuple(const vector<Attribute> &attrs, const void *data);

  RC readAttribute(const string &tableName, const RID &rid, const string
&attributeName, void *data);

  // scan returns an iterator to allow the caller to go through the results
one by one.
  RC scan(const string &tableName,
      const string &conditionAttribute,
      const CompOp compOp,                  // comparison type such as "<"
and "="
      const void *value,                    // used in the comparison
      const vector<string> &attributeNames, // a list of projected attributes
```

```
      RM_ScanIterator &rm_ScanIterator);

protected:
  RelationManager();
  ~RelationManager();
};
```

RC createCatalog()

This method creates two system catalog tables, Tables and Columns. If they already exist, return an error. The actual files for these two tables should be created, and tuples describing themselves should be inserted into these tables as shown earlier in the catalog section.

RC deleteCatalog()

This method deletes the system catalog tables. The actual files for these two tables should be deleted. It will return an error if the system catalog does not exist.

RC createTable(const string &tableName, const vector<Attribute> &attrs)

This method creates a table called tableName with a vector of attributes (attrs). The actual RBF file for this table should be created. This method should return an error if the table already exists.

RC deleteTable(const string &tableName);

This method deletes a table having the given tableName. The actual RBF file for this table should be deleted. This method should return an error if the table does not exist.

RC getAttributes(const string &tableName, vector<Attribute> &attrs);

This method gets the attributes (attrs) of a table called tableName by looking in the catalog tables. This method should return an error if the table does not exist.

RC insertTuple(const string &tableName, const void *data, RID &rid);

This method inserts a tuple into a table called tableName. This method should return an error if the table does not exist. However, you can assume that the rest of the input is always correct and error-free. That is, you do not need to check if the input tuple has the right number of attributes and/or if the attribute types match. Since there can be NULL values in one or more attributes, the first part in *data contains n bytes to pass the null information about each attributes. For details, see insertRecord() in the Project 1 Description.

RC deleteTuple(const string &tableName, const RID &rid);

This method deletes a tuple with a given rid. This method should return an error if the table does not exist, or if there is no record in the table with the corresponding rid. Also, each time a tuple is deleted, you will need to compact the underlying page. That is, keep the free space together in the middle of the page -- the slot table will be at one end, the record data area will be at the other end, and the free space should be in the middle.

RC updateTuple(const string &tableName, const void *data, const RID &rid);

This method updates a tuple identified by a given rid. This method should return an error if the table does not exist, or if there is no record in the table with the corresponding rid.

Note: if the tuple grows (i.e., the size of the tuple increases) and there is no space in the page to store the tuple (after the update), then the tuple is migrated to a new page with enough free space. Since you will implement an index structure (e.g., B-tree) in Project 3, tuples must be permanently identified by their rids, so when they migrate, you must leave a "forwarding address" behind identifying the new location of the tuple. Also, each time a tuple is updated to become smaller, you need to compact the underlying page. That is, keep the free space in the middle of the page -- the slot table will be at one end, the tuple data area will be at the other end, and the free space should be in the middle. Again, the structure for *data is the same as for insertRecord().

RC readTuple(const string &tableName, const RID &rid, void *data);

This method reads a tuple identified by a given rid. The structure for *data is the same as for insertRecord().This method should return an error if the table does not exist, or if there is no record in the table with the corresponding rid.

RC printTuple(const vector<Attribute> &attrs, const void *data);

This method mainly exists for debugging purposes. This method prints the tuple whose data is passed into this method. The structure for *data is the same as for insertRecord(). For details, refer to printRecord() in the Project 1 Description.

RC readAttribute(const string &tableName, const RID &rid, const string &attributeName, void *data);

This method mainly exists for debugging purposes. This method reads a specific attribute of a tuple identified by a given rid. The structure for *data is the same as for insertRecord(). That is, a null-indicator will be placed in the beginning of *data. However, for this function, since it returns a value for just one attribute, exactly one byte of null-indicators should be returned, not a set of the null-indicators for all of the tuple's attributes. This method should return an error if the table does not exist, or if there is no record in the table with the corresponding rid, or if there is no attribute with the specified attribute name in the table.

RC scan(const string &tableName, const string &conditionAttribute, const CompOp compOp, const void *value, const vector<string> &attributeNames, RM_ScanIterator &rm_ScanIterator);

This method scans the table called tableName. That is, it sequentially reads all of the tuples in the table. This method returns an iterator called rm_ScanIterator to allow the caller to go through the records in the table one by one. A scan has a basic query associated with it, e.g., it consists of a list of attributes to project into the answer, as well as a simple predicate on an attribute and constant (such as "Sal > 40000"). **Note:** the RBFM_ScanIterator should not cache the scan result in memory. In fact, your code should be looking only at one (or a few) page(s) of data at a time when getNextTuple() is called. In this project, let the OS do all the memory management work for you.

As usual, this method should return an error if the table does not exist, or if the attributes named in the predicate or the projection don't exist in the table.

RM_ScanIterator Class

The RM_ScanIterator class is a class that represents an iterator which is used to go through the tuples in the table one by one. The way to use this iterator is as follows:

```
RM_ScanIterator rmsi;

// At this moment, do not execute scan and cache the results in the memory.
Just initialize the scan operator
rc = rm->scan(tableName, conditionAttribute, compOp, value, attributes,
rmsi);

while(rmsi.getNextTuple(rid, returnedData) != RM_EOF){
   // fetch one tuple at at time and process the data;
}
rmsi.close();
```

The public methods of this class are shown next. The first two methods in the class declaration are the constructor and destructor methods and are not explained further.

```
class RM_ScanIterator {
public:
  RM_ScanIterator();
  ~RM_ScanIterator();

  // "data" follows the same format as RelationManager::insertTuple()
  RC getNextTuple(RID &rid, void *data);
  RC close();
};
```

RC getNextTuple(RID &rid, void *data);

This method is used to get the next tuple from the scanned table. It returns RM_EOF when all tuples have been scanned. Note that the structure for *data is the same as for insertRecord(). That is, a null-indicator will be placed in the beginning of *data. However, for this function, since it returns a value for just the attributes that are specified in the scan method, the corresponding byte(s) for those attributes of null-indicators should be returned, not the null-indicators for the entire set of table attributes.

RC close();

This method is used to close the iterator.

Design Assumptions

You can make the following simplifying assumptions when implementing PFM, RBFM, and RM (including ScanIterator):

1. The size of one tuple cannot exceed the size of a page. That is, an empty page can always hold at least one tuple. If an incoming tuple is too big for an empty page, or a tuple grows too much to fit alone on a page, return an error. (However, if a page has two or more tuples, then one of the tuples might grow, and afterwards might not fit on that shared page.)

2. A table maps to a single file, and a single file contains only one table.

Advanced Requirements (10 points)

Advanced features will be treated as extra credit work, worth a maximum of 10 points. Extra credit points will be tracked separately and used when considering effort as a factor when assigning final grades. The following are the advanced features for this part of the project:

```
class RelationManager
{
public:

  ......
  RC addAttribute(const string &tableName, const Attribute &attr);

  RC dropAttribute(const string &tableName, const string &attributeName);
}
```

RC addAttribute(const string &tableName, const Attribute &attr);

This method adds a new attribute (attr) to a table called tableName. **Note:** This operation will update the catalogs but should not involve touching the data itself. That is, when you read a record right after adding an attribute A, the value for the attribute A should be returned as NULL.

RC dropAttribute(const string &tableName, const string &attributeName);

This method drops an attribute called attributeName from a table called tableName. **Note:** This operation will update the catalogs but should not involve touching the data itself. That is, when you read a record right after dropping an attribute A, the value for the attribute A should not be included even though it still actually resides on disk.

HINT: The above two methods will affect how operations access the fields of a record if it was created before such a schema change.

Explanation

The commands listed above are by no means complete, but they do capture the essence of a tuple-oriented file system.

You have a lot of freedom in designing your specific algorithms and building your system. You should spend a significant amount of time in coming up with a design for your system before you start coding.

Grading will be based on the correctness of the implementation unless your code takes an exceedingly long time to execute a test case. Most test cases can be executed within a few seconds.