Project 4: Implementing a Query Engine with an Extension of the Relation Manager

Deadline: Thursday, June 11, 2017, 11:59 pm, on Canvas.
Full Credit: 100 points.

Introduction

In this project, you will first extend the RelationManager (RM) component that you implemented for Project 2 so that the RM layer can orchestrate both the RecordBasedFileManager (RBF) and IndexManager (IX) layers when tuple-level operations happen, and the RM layer will also be managing the catalog information related to indexes at this level. You will need a new catalog tables (in addition to Tables and Columns) that describes Indexes.

After you implement the RM layer extension, you will implement a new QueryEngine (QE) component. The QE component provides classes and methods for answering SQL queries. For simplicity, you only need to implement several basic relational operators. All operators are iterator-based. To give you a jumpstart, we've implemented two wrapper operators on top of the RM layer that provide file and index scanning. See the Appendix section below for more details.

Use the new Project 4 codebase.zip from Resources→Project 4 on Piazza as your starting point. As usual, you may fill it in using your own Project 3 code or our Project 3 solution to do Project 4. See the test cases included in the Project 4 test.sh that is also under Resources→Project 4 on Piazza for examples of how the operators are used.

Part 4.1: RelationManager Extensions

RelationManager

All of the methods that you implemented for Project 2 should now be extended to coordinate modifications of any data file with modifications of all indexes associated with that data file. For example, if you insert a tuple into a table using RelationManager::insertTuple(), the tuple should be inserted into the table (via the RBF layer), and each corresponding entry should be inserted into each index associated with that table (via the IX layer). Also, if you delete a table using RelationManager::deleteTable(), all associated indexes should also be deleted. This applies both to catalog entries that record what's what, and to the file artifacts themselves. The RelationManager class, in addition to enforcing the coordination semantics between data files and the indexes in your existing methods, should also include the following newly-added index-related methods. These methods can all be implemented by simply delegating their work to the underlying IX layer that you built in Project 3. Note that this part of the project is essentially a big wrapper. of the work that you did in Project 3.

```
class RelationManager
{
public:
  ...
  RC createIndex(const string &tableName, const string &attributeName);

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

  // indexScan returns an iterator to allow the caller to go through
qualified entries in index
  RC indexScan(const string &tableName,
                        const string &attributeName,
                        const void *lowKey,
                        const void *highKey,
                        bool lowKeyInclusive,
                        bool highKeyInclusive,
                        RM_IndexScanIterator &rm_IndexScanIterator
      );
  ...
}
```

RC createIndex(const string &tableName, const string &attributeName)

This method creates an index on a given attribute of a given table. It should also enter existence of the index in the Indexes catalog.)

RC destroyIndex(const string &tableName, const string &attributeName)

This method destroys an index on a given attribute of a given table. It should also reflect the non-existence of that index in the Indexes catalog.

RC indexScan(const string &tableName, const string &attributeName, const void *lowKey, const void *highKey, bool lowKeyInclusive, bool highKeyInclusive, RM_IndexScanIterator &rm_IndexScanIterator)

This method should initialize a condition-based scan over the entries in the open index on the given attribute of the given table. If the scan initiation method is successful, a RM_IndexScanIterator object called rm_IndexScanIterator is returned. (Please see the RM_IndexScanIterator class below.)

Once underway, by calling RM_IndexScanIterator::getNextEntry(), the iterator should produce the entries of all records whose indexed attribute key falls into the range specified by the lowKey, highKey, and inclusive flags. If lowKey is NULL, it can be interpreted as -infinity. If highKey is NULL, it can be interpreted as +infinity. The format of the parameter lowKey and highKey is the same as the format of the key in IndexManager::insertEntry().

RM_IndexScanIterator

```
class RM_IndexScanIterator {
 public:
  RM_IndexScanIterator();      // Constructor
  ~RM_IndexScanIterator();     // Destructor

  // "key" follows the same format as in IndexManager::insertEntry()
  RC getNextEntry(RID &rid, void *key);       // Get next matching entry
  RC close();                                 // Terminate index scan
};
```

RC getNextEntry(RID &rid, void *key)

This method should set its output parameters rid and key to be the RID and key, respectively, of the next record in the index scan. This method should return RM_EOF if there are no more index entries satisfying the scan condition. You may assume that RM component clients will not close an index scan while the scan is underway. Hence clients only will close an index scans after that scan returns RM_EOF.

RC close()

This method should terminate the index scan.

Part 4.2: Query Engine

Iterator Interface

All of the operators that you will implement in this part inherit from the following **Iterator** interface.

```
class Iterator {
    // All the relational operators and access methods are iterators
    // This class is the super class of all the following operator classes
    public:
        virtual RC getNextTuple(void *data) = 0;
        // For each attribute in vector<Attribute>, name it rel.attr
        virtual void getAttributes(vector<Attribute> &attrs) const = 0;
        virtual ~Iterator() {};
};
```

virtual RC getNextTuple(void *data)

This method should set the output parameter **data** of the next record. The format of the **data** parameter, which refers to the next tuple of the operator's output, is the same as that used in previous projects. Also, null-indicators for the given attributes are always placed at the beginning of **data**. That is, the tuple value is a sequence of binary attribute values in which null-indicators are placed first and then each value is represented as follows: (1) For INT and REAL: use 4 bytes; (2) For VARCHAR: use 4 bytes for the length, followed by that number of characters.

virtual void getAttributes(vector<Attribute> &attrs)

This method returns a vector of attributes in the intermediate relation resulted from this iterator. That is, while the previous method returns the tuples from the operator, this method makes the associated schema information for the returned tuple stream available in the query plan. The names of the attributes in vector<Attribute> should be of the form relation.attribute to clearly specify the relation from which each attribute comes.

Filter Interface

```
class Filter : public Iterator {
    // Filter operator
    public:
        Filter(Iterator *input,                          // Iterator of input R
               const Condition &condition                // Selection condition
        );
        ~Filter();

        RC getNextTuple(void *data);
        // For attribute in vector<Attribute>, name it as rel.attr
        void getAttributes(vector<Attribute> &attrs) const;
};
```

Using this iterator, you can do a selection query such as:
"SELECT * FROM EMP WHERE sal > 100000".

This Filter class is initialized by an input iterator and a selection condition. It filters the tuples from the input iterator by applying the filter predicate **condition** on them. For simplicity, you should assume this filter only has a single selection condition. The schema of the returned tuples should be the same as the schema of the tuples returned by the iterator.

Project Interface

```
class Project : public Iterator {
    // Projection operator
    public:
        Project(Iterator *input,                         // Iterator of input R
                const vector<string> &attrNames);
                                  // vector containing attribute names
        ~Project();

        RC getNextTuple(void *data);
        // For attribute in vector<Attribute>, name it as rel.attr
        void getAttributes(vector<Attribute> &attrs) const;
};
```

This Project class takes an iterator and a vector of attribute names as input. It projects out the values of the attributes in the **attrNames**. The schema of the returned tuples should be the attributes in attrNames, in the order of the attributes in the vector.

Index Nested-Loop Join Interface

```
class INLJoin : public Iterator {
    // Index Nested-Loop join operator
    public:
        INLJoin(Iterator *leftIn,                                   // Iterator
of input R
                IndexScan *rightIn,                                 // IndexScan
Iterator of input S
                const Condition &condition                          // Join
condition
        );

        ~INLJoin();

        RC getNextTuple(void *data);
        // For attribute in vector<Attribute>, name it as rel.attr
        void getAttributes(vector<Attribute> &attrs) const;
};
```

The INLJoin iterator takes two iterators as input. The **leftIn** iterator works as the outer relation, and the **rightIn** iterator is the inner relation. The **rightIn** is an object of IndexScan Iterator. Again, we have already implemented the IndexScan class for you, which is a wrapper on RM_IndexScanIterator. The returned schema should be the attributes of tuples from leftIn concatenated with the attributes of tuples from rightIn. You don't need to remove any duplicate attributes.

An Example

Here is an example showing how to assemble the operators to form query plans. Example: "SELECT Employee.name, Employee.age, Employee.DeptID, Department.Name FROM Employee JOIN Department ON Employee.DeptID = Department.ID WHERE Employee.salary > 50000". We are assuming for this example that the optimizer has picked the Index Nested-Loop Join algorithm to execute the join. [For this assignment, Index Nested-Loop Join is the only join method that exists, but in real systems there could be many join methods available.]

```
/****** ****** ****** ****** ******
 *    TABLE SCANS
 ****** ****** ****** ****** ******/

TableScan *emp_ts = new TableScan(rm, "Employee");
TableScan *dept_ts = new TableScan(rm, "Department");

/****** ****** ****** ****** ******
 *    FILTER Employee Table
 ****** ****** ****** ****** ******/

Condition cond_f;
cond_f.lhsAttr = "Employee.Salary";
cond_f.op = GT_OP;
cond_f.bRhsIsAttr = false;
Value value;
value.type = TypeInt;
value.data = malloc(bufsize);
*(int *)value.data = 50000;
cond_f.rhsValue = value;

Filter *filter = new Filter(emp_ts, cond_f);

/****** ****** ****** ****** ******
 *    PROJECT Employee Table
 ****** ****** ****** ****** ******/

vector<string> attrNames;
attrNames.push_back("Employee.name");
attrNames.push_back("Employee.age");
attrNames.push_back("Employee.DeptID");

Project project(filter, attrNames);

/****** ****** ****** ****** ******
 * INDEX NESTED-LOOP JOIN Employee with Dept
 ****** ****** ****** ****** ******/

Condition cond_j;
cond_j.lhsAttr = "Employee.DeptID";
cond_j.op = EQ_OP;
cond_j.bRhsIsAttr = true;
cond_j.rhsAttr = "Department.ID";

INLJoin *inlJoin = new INLJoin(project, dept_ts, cond_j);


void *data = malloc(bufsize);
while(inlJoin.getNextTuple(data) != QE_EOF)
{
  printAttributes(data);
}
```

Appendix

Below we list the APIs for the three classes used in the operators. For more detailed implementation information, please refer to the **qe.h** header file in the code base. Note that in the TableScan and IndexScan classes, the argument **alias** is used to rename the input relation. In the case of self-joins, at least one of the uses of the relations must be renamed to differentiate the two from each other in terms of attribute naming.

```
struct Condition {
    string lhsAttr;         // left-hand side attribute
    CompOp  op;             // comparison operator
    bool    bRhsIsAttr;
                            // TRUE if right-hand side is an attribute
                            //   and not a value; FALSE, otherwise
    string  rhsAttr;        // right-hand side attribute if bRhsIsAttr = TRUE
    Value   rhsValue;       // right-hand side value if bRhsIsAttr = FALSE
};

class TableScan : public Iterator
{
TableScan(RelationManager &rm, const string &tableName, const
                        char *alias = NULL);  // constructor
void setIterator();
// Start a new iterator
RC getNextTuple(void *data);
// Return the next tuple from the iterator
void getAttributes(vector<Attribute> &attrs) const;
// Return the attributes from this iterator
~TableScan();
// destructor
};

class IndexScan : public Iterator
{
IndexScan(RelationManager &rm, const string &tableName,
      const string &attrName, const char *alias = NULL); // constructor
void setIterator(void* lowKey, void* highKey, bool lowKeyInclusive,
           bool highKeyInclusive);
                    // Start a new iterator given the new compOp and value
RC getNextTuple(void *data);
// Return the next tuple from the iterator
void getAttributes(vector<Attribute> &attrs) const;
// Return the attributes from this iterator
~IndexScan();
// destructor
};
```

Important Note

You must make sure that all operators which create temporary rbfm files clean up after themselves. That is, such files must be deleted when the operator is closed.

Testing

- Please use the test.sh file that appears under Resources→Lab4 on Piazza to test your code. Note that this file will be used to grade your project partially since we also have our own private test cases. This is by no means an exhaustive test suite. Please feel free to add more cases to this, and test your code thoroughly. As with previous projects, we will test your code using a private test cases suite.

Submission Instructions

The following are requirements on your submission. Points may be deducted if they are not followed.

- Write a report to briefly describe the design and implementation of your query engine module.

- You need to submit the source code under the "rbf" ,"rm" ,"ix", "qe", and data folder. Make sure you do a "make clean" first, and do NOT include any useless files (such as binary files and data files). Your makefile should make sure the files *qetest_XX.cc* ***compile and run properly. We will use our own*** **qetest_XX.cc** files to test your module.

- Please organize your project in the following directory hierarchy: project4-*teamID* / codebase / {rbf, rm, ix, qe, data, makefile.inc, readme.txt, project4-report} where rbf, rm, ix, and qe folders include your source code and the makefile.

- Compress project4-team*ID* into a SINGLE zip file. Each team only submits one file, with the name "project4-*teamID*.zip". (e.g., project4-01.zip)

- Put the test.sh script and the zip file under the same directory. Run test.sh to check whether your project can be properly unzipped and tested (use your own makefile.inc when you are testing the script). If the script doesn't work correctly, it's most likely that your folder organization doesn't meet the requirement. Our grading will be automatically done by running script. The usage of the script is:

  ```
  ./test.sh ''project4-teamID''
  ```

- Upload the zip file "project4-*teamID*.zip" to Canvas.

Grading Rubrics

Full Credit: 100 points

You must follow the usual submission requirements in order to receive credit for this assignment, but there are no separate points given for this.

- Code is structured as required
- Makefile works
- Code compiles and links correctly

Please make sure that your submission follows these requirements and can be unzipped and built by running the script we provided. Failure to follow the instructions could make your project unable to build on our machines, which may result in loss of points.

1. Project Report (10%)
    - Include team numbers, submitter info, and names of other people on your team.
    - Clearly document the design of your project using the template provided.
2. Implementation details (15%)
    - Implement all the required functionalities as specified in the descriptions
3. Pass the provided test suite. (50%)
    - Each of which is graded as pass/fail.
4. Pass the private test suite. (20%)
    - There are a number of private tests, each of which is graded as pass/fail.
5. Valgrind (5%)
    - You should not have any memory leaks. A valgrind check should not fail.