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

Project 4: Implementing a Query Engine with the Extension of the ...
- Introduction
- CodeBase

Part 4.1: RelationManager Extensions
- RelationManager
  - RC createIndex(const std::string &tableName, const std::string ...
  - RC destroyIndex(const std::string &tableName, const std::string ...
  - RC indexScan(const std::string &tableName, const std::string ...
- RM_IndexScanIterator
  - RC getNextEntry(RID &rid, void *key)
  - RC close()

Part 4.2: Query Engine
- Iterator Interface
  - virtual RC getNextTuple(void *data)
  - virtual void getAttributes(vector<Attribute> &attrs)
- Filter Interface
- Project Interface
- Block Nested-Loop Join Interface
- Index Nested-Loop Join Interface
- Grace Hash Join Interface
- Aggregate Interface
- Important Note
- An Example
- Command Line Interface Interpreter
- Appendix
- Testing
- Submission Instructions
- Grading Rubrics
- Q & A

- **Deadline: Friday, March 13th, 2020 at 11:45 pm, on Github.**
- **Full Credit: 100 points**
- **Maximum Extra Credit: 15 points.**
- **As in Projects 2 and 3, you should work with your original team member for this Project.**

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 indices at this level. After the RM layer extension, you will implement a 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 for more details.

CodeBase

As in previous Projects, we have provided you with a framework to implement Project 4. Please download this Project4.patch file, put it in your git repo, and apply it to your codebase with `git am Project4.patch`.

OR, if the patch file does not work, you can follow these steps to fetch the code framework manually.

If you run into conflicts that you do not know how to merge, you can manually copy paste changes in files from the above repo. If needed, please come to office hour for assistance.

Then you can start Project 4:

- Modify the "CODEROOT" variable in makefile.inc to point to the root of your codebase if necessary.
- Implement the extension of Relation Manager (RM) to coordinate data files and the associated indices of the data files.
- Implement Query Engine (QE)
- If you think changes are necessary, please contact us first.

Part 4.1: RelationManager Extensions

RelationManager

All of the methods that you implemented for Project 2 should now be extended to coordinate data files plus any associated indices of the data files. 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 associated index of the table (via the IX layer). Also, if you delete a table using RelationManager::deleteTable(), all associated indices should be deleted, too. 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 indices in your existing methods, will 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. (This part of the project is largely a big wrapper. :-))

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

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

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

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

This method creates an index on a given attribute of a given table. (It should also reflect its existence in the catalogs.)

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

This method destroys an index on a given attribute of a given table. (It should also reflect its non-existence in the catalogs.)

RC indexScan(const std::string &tableName, const std::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 index entries left satisfying the scan condition. You may assume that RM component clients will not close the corresponding open index while a scan is underway.

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(std::vector<Attribute> &attrs) const = 0;
    virtual ~Iterator() = default;
};
```

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 the 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() override = default;

    RC getNextTuple(void *data) override;

    // For attribute in std::vector<Attribute>, name it as rel.attr
    void getAttributes(std::vector<Attribute> &attrs) const override;
};
```

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, we assume this filter only has a single selection condition. The schema of the returned tuples should be the same as the input tuples from the iterator.

Project Interface

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

    RC getNextTuple(void *data) override;

    // For attribute in std::vector<Attribute>, name it as rel.attr
    void getAttributes(std::vector<Attribute> &attrs) const override;
};
```

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 attributes in the vector.

Block Nested-Loop Join Interface

```
class BNLJoin : public Iterator {
    // Block nested-loop join operator
public:
    BNLJoin(Iterator *leftIn,            // Iterator of input R
            TableScan *rightIn,           // TableScan Iterator of input S
            const Condition &condition,   // Join condition
            const unsigned numPages       // # of pages that can be loaded into memory,
            //   i.e., memory block size (decided by the optimizer)
    ) {};

    ~BNLJoin() override = default;

    RC getNextTuple(void *data) override;

    // For attribute in std::vector<Attribute>, name it as rel.attr
    void getAttributes(std::vector<Attribute> &attrs) const override;
};
```

The BNLJoin 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 the TableScan Iterator. We have already implemented the TableScan class for you, which is a wrapper on RM_ScanIterator. 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. Note that **numPages** is the number of outer (left) pages that the algorithm can load into memory at once. In other words, **numPages** is equal to the memory block size (measured in pages) that your algorithm should utilize to make the number of loops through the inner (right) table smaller than a simple tuple-oriented join's would be (by a factor of **numPages**). That is, numPages is the number of buffers that can be used to as a read buffer and hash buckets. However, to make it simple, you can use these buffer pages to read records from R. You can construct an another separate in-memory hash table (e.g., std::map) to keep the records in the numPages buffer. This means std::map will take care of two things: loading tuples and hashing them. **Make sure that the total number of bytes of the loaded tuples in one round does not exceed "numPages * pageSize".** You can also assume that you have one page buffer to read a page from the inner relation **rightIn** and one page output buffer to keep the results.

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() override = default;

    RC getNextTuple(void *data) override;

    // For attribute in std::vector<Attribute>, name it as rel.attr
    void getAttributes(std::vector<Attribute> &attrs) const override;
};
```

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.

Grace Hash Join Interface

- **Optional: 10 extra-credit points for *everyone*.**

```
class GHJoin : public Iterator {
    // Grace hash join operator
public:
    GHJoin(Iterator *leftIn,               // Iterator of input R
           Iterator *rightIn,               // Iterator of input S
           const Condition &condition,      // Join condition (CompOp is always EQ)
           const unsigned numPartitions     // # of partitions for each relation (decided by the optimizer)
    ) {};

    ~GHJoin() override = default;

    RC getNextTuple(void *data) override;

    // For attribute in std::vector<Attribute>, name it as rel.attr
    void getAttributes(std::vector<Attribute> &attrs) const override;
};
```

Using this iterator you can do a join query such as "SELECT * FROM EMP, DEPT WHERE EMP.DID = DEPT.DID".

The GHJoin takes two iterators as input. It uses **leftIn** to iterate over the outer relation and the **rightIn** to iterate over the inner relation. Following is a sketch of how to implement this operator:

1- In the partitioning phase, create **numPartitions** partitions for each relation where each partition is an rbfm file. The name of the outer relation partitions must start with the word "left" while the name of the inner relation partitions must start with the word "right". In order to avoid conflicts in the file names (in the case of multiple GHJoins in the query tree) you will have to add a suffix that uniquely identify your partitions. For example, you can have something like left_join1_XX and right_join1_XX for the first join and left_join2_XX and right_join2_XX for the second join. **Note**: It is NOT acceptable to load the entire relation into memory while building the partitions -- you should assume that a query optimizer has chosen the number of partitions based on the amount of memory it has decided to allow this operator to use.

2- In the probing phase, load a partition of either R or S (in fact you might want to load the smaller partition*) into memory, then build an in-memory hash table for such partition. Next, probe the corresponding partition from the other relation for matching tuples.

3- The output will be the join-tuples that must be passed to the next operator. The schema of these join-tuples 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.

* Note that if you load the smaller partition (to compare the sizes of the partitions you can use **fileHandle.getNumberOfPages()**) you may need to rearrange the output attributes if S becomes the left relation. This is NOT a requirement but a closer implementation to what happens in practice :-).

Aggregate Interface

- **Basic aggregation is Mandatory.**
- **Group-based hash aggregation is Optional for everyone. You will get 5 extra-credit points for doing so.**

```
class Aggregate : public Iterator {
    // Aggregation operator
public:
    // Mandatory
    // Basic aggregation
    Aggregate(Iterator *input,                     // Iterator of input R
              const Attribute &aggAttr,            // The attribute over which we are computing an aggregate
              AggregateOp op                          // Aggregate operation
    ) {};

    // Optional for everyone: 5 extra-credit points
    // Group-based hash aggregation
    Aggregate(Iterator *input,                     // Iterator of input R
              const Attribute &aggAttr,            // The attribute over which we are computing an aggregate
              const Attribute &groupAttr,         // The attribute over which we are grouping the tuples
              AggregateOp op                          // Aggregate operation
    ) {};

    ~Aggregate() override = default;
    RC getNextTuple(void *data) override;

    // Please name the output attribute as aggregateOp(aggAttr)
    // E.g. Relation=rel, attribute=attr, aggregateOp=MAX
    // output attrname = "MAX(rel.attr)"
    void getAttributes(std::vector<Attribute> &attrs) const override;
};
```

Basic aggregation: Using the basic aggregation operator, you can execute a query such as: "SELECT MAX(sal) FROM EMP". The basic aggregate method takes an input iterator, an aggregated attribute, and an aggregate function (MIN, MAX, SUM, AVG, COUNT) as the arguments. You can assume we do the aggregation on a numeric attribute (INT or REAL). The returned value is just a single real value (4 bytes), even for the COUNT function. The schema of the (single) returned tuple should be "AggregateOp(relation.attribute)", such as "MAX(emp.sal)". Also, null-indicators always needs to be placed at the beginning of a tuple.

Group-based hash aggregation: Using the group-based hash aggregation operator, you can execute a query such as: "SELECT city, MAX(sal) FROM EMP GROUP BY city".
To implement the *group-by* feature, you need to implement the group-based hash aggregation where we add one more argument to the argument list: **groupAttr**, which is the group-by attribute. Unlike Grace Hash Join, you are not required to implement hash-partitioned aggregation using partitions on disk. You can assume that all of the groups' aggregation values will fit in a hash table in memory while the operation is executing. (E.g., think group by age or group by state -- where the number of groups is reasonable.) Each returned tuple should include the group-by attribute value followed by the aggregation value. The group-by attribute can be INT, REAL, or VARCHAR. The aggregated attribute can be INT or REAL. The schema of the returned tuples should be the group-by attribute and the aggregation attribute, such as "emp.city MAX(emp.sal)". Null-indicators always needs to be placed at the beginning of each tuple. **5 extra-credit points**

Important Note

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

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 Grace Hash Join algorithm to execute the join.

```
/****** ****** ****** ****** ******
 *    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);

/****** ****** ****** ****** ******
 *   GRACE HASH 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";

GHJoin *ghJoin = new GHJoin(project, dept_ts, cond_j, 100);

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

Command Line Interface Interpreter

- **Optional: There is NO implementation needed for CLI part**

Instead of having to manually assemble the operators to form query plans, as shown in the above example, we are also providing you with a Command Line Interface (CLI) that takes a SQL-like command and executes that command. This will hopefully provide a better, more flexible test environment than the manual approach presented above (e.g., assembling query plans manually, running them, and even debugging them). The CLI runs in interactive mode so that you can type commands and see their results interactively. To get more information about the CLI, please visit this page. Note that the CLI is provided for your convenience. We will not be using the CLI to test your code.

Important Notes:

1. In order to run the CLI on Ubuntu you might need to install this library **libreadline-dev**. You can run the following command to install it:

   ```
   sudo apt-get install libreadline-dev
   ```

2. CLI uses C++11 features. You might need to use g++-4.8 to compile CLI. Please refer to this page to install GCC 4.8 on Ubuntu. You can ignore this step if you already have a GCC with version >= 4.8

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 {
    std::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
    std::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 std::string &tableName, const char *alias = NULL);       // constructor

    void setIterator();                                                                    // Start a new iterator

    RC getNextTuple(void *data);                                                           // Return the next tuple

    void getAttributes(vector<Attribute> &attrs) const;                                    // Return the attributes

    ~TableScan();                                                                          // destructor
};
```

```
class IndexScan : public Iterator
{
    IndexScan(RelationManager &rm, const std::string &tableName, const std::string &attrName, const char *alias = NULI
```

```
    void setIterator(void* lowKey, void* highKey, bool lowKeyInclusive, bool highKeyInclusive);             // Star

    RC getNextTuple(void *data);                                                                            // Retu

    void getAttributes(vector<Attribute> &attrs) const;                                                     // Retu

    ~IndexScan();                                                                                           // dest
};
```

Testing

Please use the provided test files included in the codebase 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. The test code includes provisional points for each test case. The points are subject to change if needed.

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: master / {rbf, rm, ix, qe, cli, data, makefile.inc, CMakelists.txt, README.md, project1_report.txt, project2_report.txt, project3_report.txt, project4_report.txt} where rbf, rm, ix, and qe folders include your source code and the makefile.

Grading Rubrics

Full Credit: 100 points

1. Project Report & Implementation details (20 points)
   - Clearly document the design of your project by using the provided template.
   - Implement all the required functionalities
   - Implementation should follow the basic requirements of each function.

2. Pass the provided test suite (50 points)
   - Each of which is graded as pass/fail. Some of them need the eyeball check.

3. Pass the private test suite (30 points)
   - There are a number of private tests, most of which are graded as pass/fail. Some of them need the eyeball check

4. Extra credits (15 points)

  - Grace Hash Join. Optional for everyone. (10)
  - Group-based hash aggregation. Optional for everyone (5)

Q & A

- **Q1**: For the grace hash-join and block nested loop join, can I use a std::map() as an in-memory table?
  A1: Yes. You can.

  Q2: For the block-nested loop join, we are supposed to use numPages buffer to read tuples from the leftIn relation. How to use the memory based on this parameter?
  A2: Suppose we do a BNLP for two inputs R and S, where R is the left child. To simplify the implementation, it is acceptable to read "numPages" pages from R, then build an in-memory hash table (with additional memory) for these records. In addition, it is also acceptable to use the getNext() API of R to read enough records to fill in *one* page, then immediately add them to the in-memory hash table. Then we continue the process until we have read enough records for numPages pages (one page at a time). Since the records from S are pipelined, you can join a record from S.getNext() immediately using the hash table of R. You are required to have an output buffer. You pause the join process whenever the output buffer is full.

- **Q3**: What types of joins do we need to support?
  A3: Based on the scope of this project, we only need you to implement equi-join on single join attribute. However, you do need to support the case that multiple records share the same join key value.

- **Q4**: Do I need to read a page at a time from the right table of BNL join, since TableScan only supports a tuple-based API?
  A4: For simplicity, you can read a tuple from the right table at a time and generate join tuple directly.

- **Q5**: How to handle NULL values in Join,Aggregation, and GroupBy?
  A5: Based on the scope of this project, you do not need to support NULL values in these operators. However, you should make sure you index does not contain NULL keys (they should be filtered out).

Last modified on 2020-2-27 下午03:22:34

附件 (1)