CS4432 D Term 2014

Project 2

Develop and Evaluate Indexes, Join Processing Methods and Query Optimizer Strategies for SimpleDB Database System

Assigned: April 17th, 2014
Due: May 1st, 2014 (11:59PM)

This is a relatively large project. We encourage you to budget your time wisely to assure successful completion of this project. Practice your managerial skills, and make sure to put together a task schedule and to regularly verify that your team is indeed on track. In this project, you will mainly touch the following packages in the SimpleDB software system:

simpledb.index
simpledb.index.btree
simpledb.index.hash
simpledb.index.planner
simpledb.index.query
simpledb.materialize
simpledb.opt
simpledb.parse
simpledb.planner
simpledb.query
simpledb.record

You will learn about disk layout, indexing, query processing, and query optimization.

The project is large yet modular. That is, several of the core project tasks can be done in parallel. Furthermore, several of the tasks do not depend on each other, and partial credit can be obtained for some of them accordingly. As example of a possible task distribution, one team member could work on the SQL parser, while the second team member may be working on the index implementation. Whoever finishes first can start exploring the later tasks of the sort-and-merge join strategy. Similarly, later on in Stage 2 of the project, the evaluation and performance testing can again be tackled with a divide and conquer strategy.

Task 1: Index Type MetaData Preparation

The SimpleDB system by default always uses a static hash index whenever you specify an index in SQL using the **"create index on relation (attribute)"** command. In this project task, you will now enable the support for different types of indices.

In particular, you will first modify the simpledb.parse.Parser.java file to support the specification of different type of indices. In particular, you will override the createIndex() method, so it can parse the statement *"create [IndexType] index [IndexName] on [TableName] ([FieldName])"*. The type of the index will be denoted as { **sh, bt, eh** } which stands for static hash, B-Tree and extensible hash index, respectively.

Second, you will also need to change the **BasicUpdatePlanner** to the **IndexUpdatePlanner**. While the **BasicUpdatePlanner** is responsible for performing the updates on data records, the **IndexUpdatePlanner** takes care of also updating the index files of all indices associated with the updated relation. This way you assure that your new index will be maintained up-to-date by the SimpleDB system.

Third, you will need to modify the **CreateIndexData** class to hold the information about the index type that has been constructed over a given relation and attribute pair. This **CreateIndexData** object will be passed the **IndexMgr** as metadata using the executeCreateIndex() method in the **IndexUpdatePlanner**:

```
public int executeCreateIndex(CreateIndexData data, Transaction tx) {
      SimpleDB.mdMgr().createIndex(data.indexName(), data.tableName(),
data.fieldName(), tx);
      return 0; }
```

The executeCreateIndex() method then calls createIndex**()** in the **MetaDataMgr** class.

```
public void createIndex(String idxname, String tblname, String fldname,
Transaction tx) {
      idxmgr.createIndex(idxname, tblname, fldname, tx);    }
```

It then calls createIndex()method in **IndexMgr** class.

```
public void createIndex(String idxname, String tblname, String fldname,
Transaction tx) {
      RecordFile rf = new RecordFile(ti, tx);
      rf.insert();
      rf.setString("indexname", idxname);
      rf.setString("tablename", tblname);
      rf.setString("fieldname", fldname);
      rf.close();

   }
```

Fourth, you need to override or overload methods in **IndexUpdatePlanner**, **MetaDataMgr** and **IndexMgr** so that the type information about the new index is passed along correctly.

Currently, only three properties, namely, the index name, table name and filed name, are stored by the **IndexMgr** object. But now you will need to make sure that the **IndexMgr** also stores the index type name.

Up to this point, we have prepared the relevant information about the metadata concerning the index type and its association with which table and which attributes. However, so far the actual index has not yet been created. The actual index structure needs to be created when for the first time a tuple is actually inserted into the relation! Thereafter, after each update of the underlying relation, the index then needs to be maintained.

Remember from above, that SimpleDB system uses the **IndexUpdatePlanner** to handle both the updates to the table and updates to the index. You need to make sure the index type is passed to **IndexInfo** as well.

So whenever a table is changed (insert and delete) by a transaction, **IndexMgr** would provide an **IndexInfo** object which contains the index type information, if any, for each fieldname in that table. The **IndexUpdatePlanner** will call the corresponding methods as needed. One example may be :

```
public int executeDelete(DeleteData data, Transaction tx)
{
/*
Codes for delete the data and modify the index,
If no index, just delete the data
*/}
```

Important Note: In this Project, the "Update" statement will not be tested. Therefore, you can focus only on making the "Insert" and "Delete" operations working.

Task 2: Index Design and Development

Currently**, IndexInfo.open()** always creates a static hash index. Your task is to change that, so that **IndexInfo.open()** would instead create the correct index type based on the type information that had been indicated in the earlier create index statement. In this task, you would first implement your own extensible hash index. Your extensible hash index should implement the Index interface which the B-tree and the static hash index in the current SimpleDB system are both implementing. All other details about the index design and development are under your jurisdiction.

You must provide a description of your design challenges and final design decisions. Also, lastly you need to provide us your detailed testing plans that you utilized to convince yourself (and also us) that your extensible hash index code implements the index solution. This clearly must be done before doing actual performance. For this testing, we suggest that again as in Project1, you will override the toString() method of your extensible hash index class, so you and we can print out the current index information.

To fully integrate your new index solution into the SimpleDB engine so that SimpleDB query optimizer recognizes and makes use of indices in the query plan, you will need to change the **BasicQueryPlanner** in SimpleDB to the **HeuristicQueryPlanner.**

Task 3: Evaluation Index-based Query Performance – Selection and Join Queries

In this task you will test the effectiveness of your newly designed index against other query processing alternatives.

For this, you need to follow the testing setup as explained below, and in addition you can optionally perform additional testing scenarios of your own choice. First, please create four different tables containing 100,000 tuples each. Each table should contain roughly the same (randomly generated) data. For this, we will provide you with a small java jdbc program via MyWPI to create these tables.

DataGenerator Tool: This tool is given to you as .txt file, you should make a .java file, and make sure it compiles and connects to the DB. Variable "maxSize" controls how many tuples to generate in the tables, so adjust this variable as requested in the project description.

**If for any reason the tool did not compile, just take its logic and write your similar code (it should be straightforward)

- The first table Table1 would have no index;
- The second table Table 2 would have a static hash index
- The third table Table 3 would have an extensible hash index,
- The last table Table4 would have a B-tree index.

To test the relative effectiveness of different indexing solutions versus no index, you should query all 4 tables with a selection query of the form

"Select [ALL Attributes] from [YourTABLE] Where [YourAttribute] = [SomeConstant]"

Again, you need to keep track of performance statistics you observe about these runs. That is, you should keep track of the relative runtime of each query, and in addition you should also report the number of IOs.

Furthermore, you also should design a testing scenario to better understand how indices are being used for join query processing and their effectiveness in speeding up join query processing. For this testing scenario, you would now also create a fifth table (Table 5) using the same random number generated above, which only holds 50,000 records. This Table 5 will *NOT* have any indices defined on it. Now you can test the join performance by querying your tables with four queries similar to the testing scenario above always involving join with Table5 as indicated below:

"Select [ALL ATTRIBUTEs] from Table5, TableX Where [Table5.Attribute] = [TableX.Attribute]".

Where “TableX” will be replaced by Table1, Table2, Table3, or Table4 in the different queries. Again, as above, you need to keep track of the running time and I/O counts for each query.

Task 4: Develop SmartMergeJoin in SimpleDB

The SimpleDB MergeJoin will always re-sort all records of the two participating tables regardless of the records are already sorted or not. This is not very efficient. In this task, you need to improve that. As first step, you need to keep track of whether a table has been sorted or not (you may do so in TableInfo.) At the beginning, none of the tables are sorted. The current **SortScan** of the **sortmergejoin** will sort the table and write the records back out to a temporary table for later use (materialization). However, the original table is never sorted.

You need to extent the **SortScan** class, so that your modified SortScan sorts the base table as well and then sets the Sorted flag to true, thus indicating the table has been sorted. You could do this either by copying the sorted file back into the existing file blocks, or alternatively you could record the appropriate information in the system metadata. In particular, you may replace the **RecordFile** of the table you are sorting to the **RecordFile** of the temp table, or copy every record in the **RecordFile** of temp table to the **RecordFile** of your base table.

For simplicity, whenever a table is scanned by **UpdateScan** (meaning it may possible have been updated and thus may be out of order), the sorted flag should be re-set to false. **UpdateScan** here refers to the possibly update of data in the underling relation.

You also need to extent the **MergeJoin** class, so it uses your improved **SortScan**. Thus your improved **SmartMergeJoin** Operator would sort the tables first only if the tables are not sorted already. So that if it gets lucky, it can completely skip the first phase of the **SortJoin** method, the sorting phase.

Task 5: Testing SmartSortMergeJoin in SimpleDB

In order to test your SortMergeJoin operator, you would need to implement your own **Query Planner**, which always uses SortMergeJoin. for query processing, instead of using the given **HeuristicQueryPlanner.** Name your newly designed QueryPlanner the **ExploitSortQueryPlanner.** For this, your **ExploitSortQueryPlanner** would need to implement the **QueryPlanner** Interface, which is also implemented by both **BasicQueryPlanner** and **HeuristicQueryPlanner.**

You would create two tables each with 100,000 records and no index. Write a query to sort merge join these two tables. Keep track of the runtime of this query execution process. Then execute the identical query a second time. This time your second run should be much faster than the first run, because it should skip the first sorting process.

Note on SimpleDB: *SimpleDB requires the index to be created at the start while the relational table is still empty. Only data records that are inserted into the indexed table, after the index has been set up will be retrievable via the index.*

Deliverables (Packaged in one single zip file)

1. A project2 report file (doc or pdf) that contains at least the following information:
   a. Precise **installation** instructions for the CS4432 staff to be able to run your modified SimpleDB. We will blindly follow your step-by-step instructions. In some circumstances, we may also set up a one-to-one meeting with your team in which you will discuss your findings related to project1. This also includes an overview of what your document contains.
   b. Queries used in your **testing** should be described. *In addition, we will also provide you with one small JDBC file that you should use for doing one performance testing run of both the index-based SQL selection queries and the join-based SQL queries. Also, do submit your results for this test.*
   c. **Design description** in English of your effort in designing and developing a clean extension of the SimpleDB index structure, query planner, and other classes you have touched.
2. Your **testing scenarios** for verifying that your extensible index solution works correctly, and is in fact being exploited by the query optimizer for query optimization. This should describe all testing scenarios that you have designed, a rationale for each test case, what it tests and how, and how you determined that your code indeed is correct. This should include the **output of your key testing runs** that you have conducted above.
3. Results of your **experimental study** showing the performance of your system, which displays the *number of I/Os* and *the time* it took to complete different Selection queries and Join queries. Provide a clear analysis of your results. Did they match your expectation in terms of performance results? Discuss the respective performance differences of your methods, or lack thereof.
4. **Bug report.** Make sure you report all aspects of your system that are not properly working. If your code is detected to not be working, or to have serious omissions and you have not documented this shortcoming, you will be graded more harshly.
5. Fully working and **self-contained** code, supported **with inline comments** and indications of what lines of code were changed and why. You will be graded not only on the correctness of your code, but also its clarity and the quality of your software documentation. You should also include in your project submission a **DIFF file** (using Vim Diff or another other tool) showing the differences between your modified and the prior version of the simpleDB code base.

Note on Grading

Project 2 will have higher weight than Project 1. That is, Project 1 will be 8% of the total grade while Project 2 will be 12%.

What to Submit

A single file .zip from each team containing the files mentioned above.
Make sure in the report file the team member names are written.

Where to Submit

Only through the Blackboard System.