Andrew Nolan and Jackson Powell

4/25/19

CS4432

Database Systems 2 Project 2:

Report

**Installation:**

The modified version of SimpleDB is installed and functions in the same way as the original SimpleDB program. But with our optimized indexing feature as an option.

Installing:

The code for the simpledb is contained in the zip file, simply import the project into eclipse as an 'already existing project'

In order to run SimpleDB:

1. Run Startup.java with a parameter describing the name of

the working directory for the SimpleDB server. The Startup.java

file can be found under the simpledb.server package in the src folder.

2. The database may then be interacted with through the studentClient/simpledb folder by either:

a. Running set commands in the java files found in the studentClient/simpledb folder that are labeled as testing files

b. Running SQLInterpreter.java may be run from this location to provide a prompt, where a subset of SQL may be directly entered to interact with database.

Simpledb will run as it normally does in order to use our additional features one must create an index on a table to do this, create a table and then run the SQL:

“create [IndexType] index [IndexName] on [TableName] ([FieldName])”

The type of the index must be {sh, bt, or eh}

**Testing:**

Our testing files included 3 distinct SQL tests to ensure that the extensible hash not only maintains the ‘stellar integrity’ of simpledb but also improves performance compared to non-indexed tables. The results from these tests can be seen in our test results report.

The first test is a simple select statement with a where clause on the indexed column:

select a1, a2 from [table] where a1 =432

This test is performed on the static hash indexed table, btree indexed table, nonindexed table, and extensible hash indexed table.

The second test is a join between the indexed table and a non-indexed table. The test was performed with all 4 tables again, joining them with the same table:

select a1, a2, a1, a2 from test5, [table] where a1 = [table].a1

The final test is two parts. First a delete is performed on the table, next the same select from test 1 is performed to ensure the integrity of the system.

For our tests we set the maxsize to be 5,000 records to avoid hanging when inserting larger numbers of records. It also was needed to avoid buffer aborts during testing joins.

To run our tests simply startup SimpleDB and then in the studentClient/simpledb folder run the file CreateTestTables.java. This file will generate 4 tables each with 5000 records. One with no index, one with a static hashing index, one with our extensible hash index, and one with a BTree index. It will also generate a non-indexed 2500 record table to test joins with the indexed tables. There are four more test files: ExtensibleHashTest.java, NoIndexTest.java, BTreeIndexTest.java, and StaticHashTest.java. Each of these files will perform the same tests on the different tables. You can run them to compare the results. The times are printed out by these test files. The I/O’s are printed out in startup.java’s console.

**Design Description:**

In order to develop a clean extension of the SimpleDB index and planning structure, we designed an extensible hash index system.

First, we modified the parser and activated the use of the heuristic planner to allow for the user to decide which type of index they would like to use. If nothing is specified SimpleDB defaults to its original state using static hashing for indexing. Btree and extensible hash are both enabled as well now. To make this work an additional piece of metadata was added to the indexinfo and indexmgr, they now store the type of index to be used.

To implement the extensible hashing index we created a new package and three classes: the index itself, a class to represent the buckets of the hash, and a class to represent the data nodes. To implement the extensible hash we used a hashmap that maps the hashvalues of the datavals to buckets using binary modulus. We begin with 2 buckets that can each hold 64 records; we chose 64 records so that bucket splitting did not happen too frequently with large amounts of inserts. When a bucket will become overfull we empty the bucket and split it into two buckets and then recursively add the nodes from the old bucket back into their new proper buckets. An example of splitting can be seen below. Local depth and global depth are also maintained. If the a bucket splits and has the same depth as the global depth, the global depth is increased and more pointers are added to the hashmap. If the local depth is less than the global depth the bucket is split and the pointers in the hasmap are not increased but redirected.

Example output for inserting into extensible hash index:

Inserting 452 with hash: 0

Old global table

Hash Local Depth Number of records Global Depth = 1

0 1 65

1 1 60

Increasing global depth from 1 to 2

New table:

Hash Local Depth Number of records Global Depth = 2

0 2 33

1 1 60

2 2 32

3 1 60

Inserting 676 with hash: 0

Inserting 536 with hash: 0

Inserting 358 with hash: 2

Inserting 504 with hash: 0

Inserting 600 with hash: 0

Inserting 486 with hash: 2

Inserting 598 with hash: 2

Inserting 908 with hash: 0

Inserting 865 with hash: 1

Inserting 859 with hash: 3

Inserting 615 with hash: 3

Inserting 522 with hash: 2

Inserting 192 with hash: 0

Inserting 1 with hash: 1

Inserting 688 with hash: 0

Inserting 241 with hash: 1

Inserting 485 with hash: 1

Old global table

Hash Local Depth Number of records Global Depth = 2

0 2 40

1 1 65

2 2 36

3 1 65

New table:

Hash Local Depth Number of records Global Depth = 2

0 2 40

1 2 37

2 2 36

3 2 28

Looking at this output you can see the two different kinds of splits. First when one of the buckets overfilled and it’s local depth was the same as the global depth. The number of buckets doubled and that bucket split into two buckets. The other bucket remained the same and just had two pointers that both pointed to it.

In the second split a bucket that had a local depth smaller than the global depth overflowed. Consequently it split without doubling the number of pointers. Now all of the pointers point to unique buckets.

**Files Added/Modified:**

--All changes can be viewed in the Diff file--

simpledb.buffer.BasicBufferMgr.java

simpledb.buffer.BufferMgr.java

simpledb.buffer.LRUBufferMgr.java

simpledb.index.extensible.ExtensibleHash.java

simpledb.index.extensible.HashBlock.java

simpledb.index.extensible.Node.java

simpledb.index.planner.IndexUpdatePlanner.java

simpledb.metadata.IndexInfo.java

simpledb.metadata.IndexMgr.java

simpledb.metadata.MetadataMgr.java

simpledb.opt.HeuristicQueryPlanner.java

simpledb.parse.CreateIndexData.java

simpledb.parse.Parser.java

simpledb.planner.BasicUpdatePlanner.java

simpledb.record.TableInfo.java

simpledb.server.SimpleDB.java

simpledb.tx.Transaction.java

simpledb.BTreeIndexTest.java

simpledb.CreateTestTables.java

simpledb.CreateTestTables2.java

simpledb.ExtensibleHashTest.java

simpledb.NoIndexTest.java

simpledb.Project2Testing.java

simpledb.StaticHashTest.java