Assignment 1: In-memory Data Layout

Assigned: January 19th, 2021
Due: February 4th, 2021, 11:59pm Pacific
Starter Code: https://github.com/stanford-futuredata/cs245-as1
Link to class Gradescope: https://www.gradescope.com/courses/218759

Overview

In this assignment, you will implement different in-memory relational stores to measure the effect of data layout and indexing on query performance. You will be working with a minimal query engine written in Java that can load or generate data tables and run four different types of queries on them (implemented as methods on a `Table` class).

We have provided skeletons for `RowTable`, `ColumnTable`, and `IndexedRowTable` in the starter code, which you need to fill in so that they can correctly and efficiently support the four query types we will run (described below). In addition, you will implement a `CustomTable` of your own design, which may adaptively use different data layouts and indices depending on the query to maximize performance. Your `CustomTable` will run in a benchmark competition against the whole class, where the top entries will get bonus points on the assignment.

We will test for the correctness of each of your query methods (`columnSum()`, `predicatedColumnSum()`, etc.) for each of the tables you implement, as well as their execution time: for each table type, your methods need to run at a particular speed (or faster) to receive full points. We will not measure your loading time (the time to run the `load()` methods on your table classes).

We will measure the runtime of each method on each table and workload independently. We measure these runtimes using the Java microbenchmark harness, which runs each query method in a tight loop repeatedly with varying warm-up runs and JVM forks. We then use the best runtime it achieved.

This is a checklist of tasks you'll need to do. We provide more detailed instructions below.

- Finish implementing getter and setter methods in `RowTable`, `ColumnTable`, `IndexedRowTable`.
- Implement our four queries for each of these tables: `columnSum()`, `predicatedColumnSum()`, `predicatedAllColumnsSum()`, and `predicatedUpdate()`.
- Ensure the basic correctness JUnit tests pass.
- Run the Gradescope grader locally to compare runtime performance (note that these timing measurements may differ from the grading machine, but should show relative differences).
- Implement your `CustomTable`, and check its runtime performance locally. You can compare to your already implemented `RowTable`, `ColumnTable`, and `IndexedRowTable` on the provided workload trace (note that the random seed used on Gradescope may differ).
- Submit to Gradescope and check your final results.

Note that this assignment is to be done **individually**. We encourage students to form study groups, but all code must be written independently.

Setup

Software Dependencies

The assignment is written in Java. You will need a Java compiler, Java 11, and Maven version >= 3.3.9. You can download Java 11 here and install Maven using your choice of package manager (brew, apt-get, etc). If you haven't used Java before, you can go through the basics in this Oracle tutorial; or if you are rusty, peruse this simple reference. We highly recommend using an IDE such as IntelliJ community edition to develop in Java, as it is harder to do with just a text editor.

Compilation

The code can be built using your favorite IDE's "Build" functionality or from the command line using,

```
mvn package
```

To get more detailed stack traces, use

```
mvn -DtrimStackTrace=false package
```

JUnit Tests

JUnit tests can be run either using your favorite IDE's "Run all tests" functionality or from the command line using,

```
mvn test
```

To run a specific test (for example, `memstore.table.ColumnSumTest`), run

```
mvn -Dtest=memstore.table.ColumnSumTest test
```

With the starter code, only 4 of 21 tests will pass, including `memstore.grader.GradedTestResultTest`, `memstore.table.RandomizedLoaderTest`, and `memstore.table.CSVLoaderTest`.

Setting Up the Environment on Myth

You're welcome to use your own machine, but if for whatever reason you don't want to, here is how to install all of the dependencies on a Stanford myth machine.

```
bash
mkdir javabin
cd javabin

wget https://download.java.net/java/GA/jdk11/9/GPL/openjdk-11.0.2_linux-x64_bin.tar.gz
tar xvf openjdk-11.0.2_linux-x64_bin.tar.gz
wget http://mirror.reverse.net/pub/apache/maven/maven-3/3.6.3/binaries/apache-maven-3.6.3-bin.tar.gz
tar xvf apache-maven-3.6.3-bin.tar.gz

export JAVA_HOME=$PWD/jdk-11.0.2
export M2_HOME=$PWD/apache-maven-3.6.3
```

```
export M2=$M2_HOME/bin
export PATH="$JAVA_HOME/bin:$M2:$PATH"
```

Then within that bash shell you will be able to run the assignment. The timings should all pass for a correct solution on myth: in fact myth is slightly faster than the gradescope machines.

Background

The performance of your tables will depend significantly on how their data structures and algorithms use the CPU cache. A CPU cache is a smaller, faster memory used by CPUs to reduce the average cost of accessing data from DRAM by storing copies of data in frequently used memory locations.

When the CPU needs to read or write a location in main memory, it first checks for a corresponding cache entry in any cache lines that might contain that address (a cache line is typically 64 bytes of contiguous memory). If the CPU finds the memory location in the cache, a cache hit occurs, avoiding a slower DRAM access. Otherwise, a cache miss occurs, and the CPU needs to go to DRAM to fulfill the request, and it will put the entire 64-byte cache line it reads from DRAM into the cache.

Caches on modern machines can provide performance benefits when used wisely. Many database queries are bottlenecked on DRAM bandwidth. Therefore, database developers must consider *cache locality* when making design decisions: the fewer accesses the system has to make to main memory, the faster it will operate (since read and write latencies with caches are far lower than with DRAM). Ideally, data accesses that are adjacent in the *program sequence* should also be adjacent in *memory* so that they are loaded into cache together as a single cache line and the program suffers fewer cache misses, consequently leading to better performance.

Row Stores

Row stores lay out data row-wise. Assume that we have a table with N rows and M columns where each field in the table takes B bytes. Then, a row store lays out data in the following format:

```
| row0, col0 | row0, col1 | ... | row0, colM | row1, col0 | ... | row1, colM | ... |
```

In this assignment, you can assume B=4 because we are storing 32-bit integers. Then, one can compute the byte offset of the field in row i and column j (assuming zero-indexing) using the formula,

```
B * ((i * M) + j)
```

Column Stores

Column stores lay out data column-wise so that consecutive values in a column are adjacent in memory. Different columns do not need to be adjacent in memory, but in this assignment we will assume that they are. Similarly, assume a table has N rows and M columns, then our column store lays out data in the following format:

```
| row0, col0 | row1, col0 | ... | rowN, col0 | row0, col1 | ... | rowN, col1 | ... |
```

One can compute the byte offset of the field in row i and column j using a similar formula to the row store.

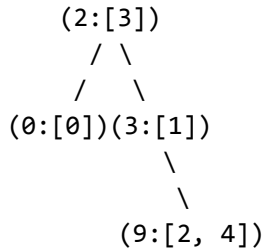
Indexed Stores

We can also use an index on top of a row store to speedup the evaluation of predicates. For example, we can maintain a data structure that stores the row indexes of all rows that contain a certain value in a column. In this assignment we would like indices to accelerate queries with comparison predicates (e.g., `WHERE x > 10`, or `WHERE x < 10`).

You are provided a TreeMap, which is a Red-Black tree (a type of binary search tree) mapping elements in the `columnIndex` column to a list of the IDs of the rows that contain them. For example, consider the following table,

rowId	col0	col1	col2
0	0	1	2
1	3	2	5
2	9	3	4
3	2	4	7
4	9	4	6

Its index on col0 would look like the following,



You can use this index to quickly determine all rows with col0 value less than (or greater than) a certain threshold. For highly selective predicates, indices can be useful since they avoid loading many complete rows; however, they can lead to more random reads and worse performance for less selective predicates.

Performance Tradeoffs

Row, column, and indexed stores are each efficient for different types of queries. For example, in a table with a large number of columns, using a column store for queries that only look at a single column in the table is much more efficient than using a row store. For a query with a highly selective predicate (low number of matching rows) over a table with a large number of rows, using an indexed store is more efficient since it prevents the need to look at all rows. However, there are also queries where indices and column stores impose overheads with few benefits.

Supported Queries

We want all your table implementations to support the following queries,

columnSum()
`SELECT SUM(col0) FROM table;`
Returns the sum of all elements in the first column of the table.

predicatedColumnSum(int threshold1, int threshold2)
`SELECT SUM(col0) FROM table WHERE col1 > threshold1 AND col2 < threshold2;`
Returns the sum of all elements in the first column of the table, subject to the passed-in predicates.

predicatedAllColumnsSum(int threshold)
`SELECT SUM(col0) + ... + SUM(coln) FROM table WHERE col0 > threshold;`
Returns the sum of all elements in the rows which pass the predicate.

predicatedUpdate(int threshold)
`UPDATE table SET col3 = col3 + col2 WHERE col0 < threshold;`
Returns the number of rows updated. If col2 happens to be zero, the row is still considered updated.

Queries are called by calling the respective method.

Suggested Implementation Order

We suggest implementing methods for *all tables* in the following order,

- `getIntField()` / `putIntField()`: After implementing these methods, `TableLoadTest` should now pass.
- `load()`: You will need to implement the `load` function for `IndexedRowTable`.
- `columnSum()`: After implementing these methods, tests in `ColumnSumTest` should now pass.
- `predicatedColumnSum()`: After implementing these methods, tests in `predicatedColumnSumTest` should now pass.
- `predicatedAllColumnSum()`: After implementing these methods, tests in `predicatedAllColumnSumTest` should now pass.
- `predicatedUpdate()`: After implementing these methods, tests in `predicatedUpdateTest` should now pass.

Feel free to add more unit tests to test your implementation. For example, you might want to write a unit test to test your index when implementing the `IndexedRowTable`.
The main `GraderRunner` class is the same grader that will be run on Gradescope, but we reserve the right to use different seeds. Unless otherwise announced due to bugs or important patches, results visible on gradescope after your submission completes reflect the grade we will actually assign your submission, though we will tabulate bonus points for the leaderboard at the end.

Custom Table

Real world workloads involve many different kinds of queries on the same table, and database system designers are forced to optimize data layout and indexes over the data in order to obtain good average case performance on this query mix -- it is too expensive to try to move data into a new layout every time a new query comes in. In this assignment, you will write a `CustomTable` implementation that executes a provided query mix as fast as possible.

You can make the following assumptions about the workload we will test you on,

- Total size of all fields in the table is ~100-150MB.
- Number of columns pulled from a uniform distribution [4, 1004).
- Number of rows is computed as the greatest integer function of (tableSize / numCols).
- Each integer in the table is in the range [0, 1024).
- Thresholds for `predicatedColumnSum()`, `predicatedAllColumnsSum()`, `predicatedUpdate()` drawn independently from a uniform distribution [0, 1024).
- Queries: 100 queries each of `columnSum()`, `predicatedColumnSum()`, `predicatedAllColumnsSum()`, `predicatedUpdate()`, and `putIntField()`.

The code that creates this query mix is in `src/main/java/memstore/workloadbench/CustomTableBenchAbstract.java`.

You may reuse your existing `RowTable`, `ColumnTable`, and `IndexedRowTable` implementations if you like. One possible approach is to determine the threshold ranges where these different table implementations work well, and then call the corresponding table's implementation based on the query. However, there are probably more clever (and more efficient) approaches! The fastest CustomTable submissions in the class will get up to 6 bonus points on this assignment.

Grading

To run the autograder tests locally, you can run

```
mvn clean
mvn package
java -Xmx1328m -Xms500m -jar target/benchmarks.jar -m all -i 60 -ci 10
```

The last line of the output shows how many points were earned on each test.

Note that this runs the non-`CustomTable` tests 60 times each, and the `CustomTable` tests 10 times each -- you can reduce these numbers down to 1 or 2 when testing locally.

More detailed instructions for compiling and running tests are in `README.md`, including instructions on how to replicate the scripts we will use for final grading on your local machine.

Important Note on PerformanceTests

For the 3 basic tables you implement, we will only use a single workload to performance test each query. You are free to tailor each individual method in each table for the workload. Each workload is summarized below, but you can also view the benchmarks in `src/main/java/memstore/benchmarks/`.

PredicatedAllColumnSumBench (Calls `predicatedAllColumnsSum`)
* Queries that perform operations on many columns for a large fraction of the rows will benefit from row-based layouts due to memory locality.

PredicatedColumnSumBench (Calls `predicatedColumnSum`)
* Highly selective predicates should allow indexes to perform very well.

PredicatedUpdateBench (Calls `predicatedUpdate`)
* Highly selective predicates should allow indexes to perform very well. Note that the updates in this query do not affect the columns being filtered or added.

Submission Instructions

When done, run the `create_submission.sh` script to produce a `student_submission.zip` file -- upload this file on Gradescope. Note that this script should only contain five files (`Table.java`, `RowTable.java`, `ColumnTable.java`, `IndexedRowTable.java`, `CustomTable.java`), so please put whatever code you implemented in these five files.