CSE 562: Project 2 (4 pages)

Due Date: 11:59 AM on Sunday, March 25
Groups: You may work in groups of 2-3 people.
Submission Instructions: Compress (as tar.gz or zip) a single directory containing:

- All of the your Java Source files for the project, including those provided as part of the project itself (they may be organized in subdirectories).
- A file named TEAM containing each team member's UBIT, one per line.

Have one group member upload your file to `timberlake.cse.buffalo.edu`, then log in and run the command:

```
/util/bin/submit_cse562 [filename]
```

Introduction

Firstly, due to popular distaste for Makefiles, I've switched to ant. Ant works more smoothly with eclipse, and java in general. The structure of the source tree should also be a bit more friendly to both systems.

3 packages have been added/modified in Project 2. The buffer package is based around a new component: the BufferManager. The BufferManager is a simplified version of a buffer manager that might be found in a traditional DBMS; It keeps a sequence of frames, and allocates them on an as-needed basis to files.

Your primary interface to the BufferManager will be the FileManager and ManagedFile classes. Each ManagedFile corresponds to a single file on disk, and gives you per-page access to the contents of that file. ManagedFile interacts with the BufferManager to limit memory usage to an amount specified when the BufferManager is first created. FileManager provides a means of creating new ManagedFile instances.

Second, two new classes have been added to the data package: DatumBuffer and DatumSerialization. A DatumBuffer instance wraps around a ByteBuffer, and provides functionality for organizing (variably-sized) Datum[] rows in a ByteBuffer so that they can be read back out.

Finally, note the index package, which includes several interfaces defining how to interact with an index. This includes several simple interfaces for the indexes themselves, as well as a class called IndexKeySpec, which is an abstract specification of how index key is related to the corresponding data rows. The primary operation is createKey() which extracts the key columns for a given row. For example, if the key were the first column of the relation, IndexKeySpec.createKey() would return a new Datum[] containing a single element: the contents of the first column. IndexKeySpec also contains several methods for obtaining schema information and comparing keys to keys and hashing keys.

A utility class called GenericIndexKeySpec should provide you with *most* of the functionality that you need for this purpose.

For the purposes of this project you MAY use native java memory for internal state of your database, and a **constant** or bounded amount of temporary state. All indexing structures and relation data must be stored through the BufferManager/ManagedFile API.

In short, if I ask you tobuild an index over 1 million tuples with a buffer manager set to 1024 tuples, java should not be using substantially more memory than if I ask you to build an index over 1000 tuples with the buffer manager set similarly.

The index script

The utility script `index` has been provided in the sql directory. This invokes the main method of edu.buffalo.cse.sql.Index, which is a testing rig for indexes. Index utilizes a data stream generator, which provides a repeatable stream of relational rows, built to order.

The command line options of the index script are as follows:

- `-keys numKeys` Set the number of key columns to `numKeys`. The total number of columns in the generated relation is equal to the number of key columns plus the number of value columns. (default: 1)
- `-values numValues` Set the number of key columns to `numValues`. The total number of columns in the generated relation is equal to the number of key columns plus the number of value columns. (default: 4)
- `-rows numRows` Set the number of rows generated to `numRows`. (default: 100)
- `-frames numFrames` Set the number of frames in the bufferPool to `numFrames`. (default: 1024)
- `-keychaos chaos` The higher the `chaos` parameter is set to, the more distinct the keys that are generated will be.
- `-hash` Generate/validate a `hash` index.
- `-isam` Generate/validate a `ISAM` index.
- `-indexSize size` Set the (default) index size to `size`. For the hash index, this will be the number of hash buckets to create. For the ISAM Index, this value is ignored
- `-get key` Perform a lookup on the indicated comma-separated-integer key, returning exactly one (arbitrary) value matching the key.
- `-validate` Validate an already generated index. Scan on ranges defined by `-from` or `-to`, or over the entire index otherwise.

- `-from` Set the lower bound for the index validation pass.
- `-to` Set the upper bound for the index validation pass.

Parameters not matching the above are assumed to be filenames, and the last filename will be used to store the index.

Part 1: Static Hash Index (40 points)

In this part, we will construct a *dense* static hash index, as represented by the placeholder class `edu.buffalo.cse.sql.index.HashIndex`. You should create a create() function to generate the index, and define the class to subsequently access the index. Note that as we are creating a *dense* index, data pages should store entire rows, and not just pointers/record IDs.

The design of this class is mostly up to you. I will leave you with several observations:

- Recall the static hash index design from lecture. This design involves a fixed number of (preallocated) sequential pages to store the hash table buckets, and a set of additional overflow pages. Each data page consists of a sequence of (unordered) data tuples, and a pointer to the first overflow page (i.e., forming a linked list).
- The DatumBuffer class provides functionality for encoding data tuples into ByteBuffers. Normally, the DatumBuffer class consumes the entire ByteBuffer. However, note that when initializing a DatumBuffer, you can request that it reserve some space in the page (e.g., for a linked list).
- The IndexKeySpec class provides functionality for hashing both keys and rows. The hash value typically falls in the range 0 to Int Max. However, remember how we the modulus operator to cut a big hash range down to a smaller range.
- The directory size is variable. You'll need to find someplace in the file to store this information permanently (as well as any other information that needs to be persisted).

Grading: Your hash index will be tested by repeatedly (1) generating a hash index, with an arbitrarily chosen set of values for the integer parameters to the testing script, and (2) doing one or more Gets on the index. Values returned should correspond to known keys.

The following parameters, and/or combinations thereof will be used to test your code.

- Number of Frames: 1024 (default),100, 10
- Number of Rows: 100, 200, 500, 10000, 1 Million
- Index Size: 10, 100, 1000, 10000
- Key Chaos: 2, 3, 4, 5, 10, 20

- Number of Keys: 1 to 10
- Number of Values: 1 to 10

Note: There are 36,000 different possible combinations of the above parameters. Testing them all may be difficult. Which combinations are the most meaningful. Which identify special cases in your code? For example, what happens to the Hash index when there are a very low number of distinct keys? (e.g., number of rows 100, key chaos 2, number of keys 2)

Part 2: ISAM Tree Index (60 points)

As in Part 1, you're going to construct a *dense* static ISAM tree. A placeholder for this index class is provided in `edu.buffalo.cse.sql.index.ISAMIndex`. Some more observations:

- Recall the structure of the ISAM index. Data pages are stored sequentially on disk, and tree index pages store alternating keys/pointers.
- Because record sizes are variable, it may not be possible to predict the number of pages used until they've actually been written out. This means you might need to perform 2 passes over the file. First, to write the data pages out (on sequential pages), and then afterwards to index the records.
- The DatumBuffer class may be insufficient for the needs of an index tree page (due to the use of pointers). You may want to add a wrapper similar to/on top of DatumBuffer.

Grading: Your ISAM index will be tested (1) in the same way as the Hash index, by running a sequence of Get operations, and then (2) by validating one or more range scans on a selected range/selected set of ranges. Test the ISAM index yourself by similarly testing on lookups for a range of keys.

In both cases, succeeding on all comthe same

Extra Credit: Dynamic Indexes (up to 20 bonus points)

Extra credit will be given for teams who implement a Dynamic Index (A B+ Tree, An Extendible Hash, A Linear Hash). If you choose to do the extra credit portion of this project, your team will be expected to demonstrate your work to the instructor in person.