Hashing Assignment
Database Implementation
CS 487/487 - Winter 2018
Due – In Class & D2L – March 15

Introduction

In this assignment you will implement a static hash-based index with index entries containing <key, rid> pairs (called alternative (2) in the slides and text). The learning objective for this assignment is to understand firsthand how hashing and an index are implemented.

Where to Find Code

Copy all the files from the homework page of the class website to your own local directory. The contents are:

- bufmgr.jar, hf.jar These are jar files for the buffer manager and heapfile packages. As in the previous assignment, you can either put these in the search path of your compiler/IDE or use your own source code.
- bufmgr, heap: These directories are empty so that you can insert your own code if you wish to use it instead of the given jar files.
- tests:The tests are in tests.IXTest.
- index: This contains the files referred to below. You are given the files DataEntry.java and SortedPage.java. If you change either of them be sure to indicate so in your README file and include it in your zip/tar file. It also includes the skeletons HashBucketPage.java, HashIndex.java and HashScan.java.
- output.txt: as before. Passing the tests means that the last line says "completed successfully". But if your output is different than this it suggests that I may find something wrong with your code.

Part 1: Search Keys, Data Entries

Because you are modeling an index, you will be working with search keys. Review global/SearchKey.java . Note that a search key consists of a type, a length and a value,and note the getHash( ) method. In an index, using alternative (2), a data entry, according to our text, consists of a search key and an RID. Review index/DataEntry.java .

Part 2: Sorted Pages

In the previous assignment you dealt with DataPages and DirPages, subtypes of HFPages. In this assignment you will deal with pages that hold an index's data entries. In the BTree case these must be sorted, so they are called SortedPages. Otherwise they are similar to the DirPages in the previous assignment. Here they do not inherit from HFPage, they inherit directly from Page. Review index/SortedPage.java .

Part 3: Hash Bucket Pages

The pages of data entries have overflow pages and are called hash buckets. For example, if you are indexing a "customers" table on email addresses, the search keys may average about 25 bytes each. If the RID takes 4 bytes and the slot takes 4 bytes this means only about 30 records can fit on one of our 1K pages. This is not a lot for a hash bucket, especially considering the possibility of duplicates, so we want to be able to chain sorted pages together to form a hash bucket. In order to support large amounts of data with potential duplicates, it becomes necessary for a bucket to span multiple pages, including one primary and possible overflow pages.

A HashBucketPage is thus an extension of a SortedPage that is part of a linked list forming a hash bucket. Since these pages will be used only for a hash index, we will ignore the sorted order of keys. Thus when you insert a record, you can ignore the requirement to insert in sorted order if the page is full and you need to create an overflow page. The skeleton is in index/HashBucketPage.java.

To Implement: index/HashBucketPage.java

Part 4: Hash Index

Hash indexes are particularly useful for equality selections, especially when used in conjunction with certain join algorithms. The basic idea is to partition the data into buckets, which will (hopefully) narrow down the search by a significant factor. Using the HashBucketPage class from part 3, you will implement the static hashing index structure described in the textbook (Section 11.1).

For this assignment, the (static) number of buckets is 128, and the hash function is simply to consider the lower seven bits of the search key. You will find the skeleton in index/HashIndex.java. Be sure to remember the provided getHash() method in SearchKey.java .

To Implement: index/HashIndex.java

Part 5: Hash Scan

Scanning is of course the whole purpose of building an index. A hash scan is simply an iterator that traverses the appropriate bucket to return all entries that match a given search key. The skeleton is in index/HashScan.java. Your implementation of HashScan.java should have at most one page pinned at any given time.

To Implement: index/HashScan.java

Grading & What to Turn In

Project will be graded by demonstration as with the previous assignments. You are also required to turn in your source code. Please post source code on D2L.