Project : B+ Tree Index

Due on 01/12/2017

INTRODUCTION

As we discussed in class, relations are stored in files. Each file has a particular organization. Each organization lends itself to efficient evaluation of some (not all) of the following operations: scan, equality search, range search, insertion, and deletion. When it is important to access a relation quickly in more than one way, a good solution is to use an index. For this assignment, the index will store data entries in the form **<key, rid> pair**. These data entries in the index will be stored at the leaf level of the index file. The actual records are stored in separate data file. In other words, the index file "points to" the data file where the actual records are stored. Two primary kinds of indexes are hash-based and tree-based, and the most commonly implemented tree-based index is the B+ Tree.

- **`BTreeIndex`**
  Conventionally, an index file name is constructed by concatenating the relational/table name with the offset of the attribute over which the index is built. The general form of the index file name is as follows: `relName.attrOffset`.

B+ TREE INDEX

Your assignment is to implement a B+ Tree index. This B+ Tree will be simplified in a few ways.

- First, you can assume that all records in a file have the same length (so for a given attribute its offset in the record is always the same).
- Second, the B+ Tree only needs to support single-attribute indexing (not composite attribute).
- Third, the indexed attribute may be one of two data types – integer (4 bytes) or string (at least 10 bytes).
- Fourth, in the case of a string, you can assume that all string attributes and search keys in this project are at least 10 characters, and that the first 10 characters of each entry that is inserted into the index is unique. In other words, for strings, you can use the first 10 characters as the key in the B+ Tree.
- Finally, you may assume that we **never** insert two data entries into the index with the same key value. The last part simplifies the B+ Tree implementation.

For this project, the index will be built in memory while database records can be stored either in memory or directly on top of the local file system supported by the OS. If you prefer, you can also store index data records in a file on disk. The data file for

database table that you create is a "raw" file, i.e., it has no page structure on top of it. You will need to maintain (implement) a **slotted page structure** on top of the pages that you will create for generating **RID** (assume that RID is 4 bytes in length – 2 bytes page id and 2 bytes slot number).

Input data set:

1. **The page/block size is assumed to be 512 bytes**
2. **R, Relation-name, key-type, record-length**
   Specify the name of the relation, the data-type (Integer, String) of its index key, and record size (including the key)
3. **I, Relation-name, key-value, "the rest of the record"**
   Could have multiple (key-value, record), separated by ';'
   Example: I, key-value1, record1; key-value2, record2; …
   (Store <key, rest of record> in the corresponding slotted page, which returns page-id and slot-number, ie, RID)
   Note: Size of(the key + rest of the record) will be <= record-length
4. **D, Relation-name, key-value**
   Delete record with key-value from the Relation with Relation-name.

Functions to be tested/checked

1. **(15%) Scan index file**
   Scan Relation-name
   Output (# of leaf pages, # of total index pages)
2. **(15%) Single value index search**
   q Relation-name key-value
   q Relation-name "key-value", if the key is a string type
   Output (Key, size of rest of the record, RID)
3. **(15%) Range query using index**
   q Relation-name key-value1 key-value2
   Output RID list of records with (Key-value1 <= key <= Key-vale2)
4. **(15%) Display data page of a relation/table**
   p relation-name page-id
   Output: the content of the database page
5. **(10%) File, index statistics**

c relation-name
Output: # of index pages, # of slotted data pages on "disk" (or in memory).

6. **(10%) Finally, error handling**
Handle invalid input and display meaningful error message (for example, input starting with something other than <R,I,D, S,q,c,d,p,>, or missing input information)

ADDITIONAL NOTES

In real B+ Tree implementations, when an error occurs, special care is taken to make sure that the index does not end up in an inconsistent state. As you will quickly realize handling errors can be hard in some cases. For example, if you have split the leaf page and are propagating the split upwards, and then encounter a buffer manager error, exiting the method without cleaning up could corrupt the B+ Tree structure.

Sample test data will be provided first, probably next week. Test data for generating final result/report will be provided one week before the due date.

GRADING

The breakup of the grading for this assignment is as follows:

1. **Correctness: 80%**. The correctness part of the grade will be based on the tests that we will provide (**integer** index required; **15% extra credit** for supporting **string**).
2. **Programming Style: 5%**. For your style points, we will check your code for readability (how easy is it to read and understand the code), and for the code organization (do you repeat code over and over again, etc.).
3. **Design report: 15%**. Your design report must describe the following design choices that you make:
    - Any implementation choices that you make. How do you format & manage the slotted page?
    - **Test design:** Describing test cases that test various code paths rigorously. This will not only get you the test points, but will most likely also get you the correctness points..
    - Finally, your design report must also explain how your design/implementation would change if you were to allow duplicate keys in the B+ Tree, i.e., allow multiple data entries with the same key value.