Project 3
B+Tree

Out: October 20, 2014
***Due:** November 10, 2014, 11:59 P.M.*

1 Introduction

We have seen how the B+ Tree works in class, and how it is one of the most powerful and most widely used indices for databases now in use by the industry.

Have you ever dreamt about B+ Trees? Get ready for it, because in this assignment you will write your own B+ Tree implementation. You will also spend some time on testing the tree and analyzing its performance. No matter how efficient your design will be, its performance will depend on the parameters (i.e. the node size.) You will hopefully have a solid understanding of B+ Trees after you complete these tasks:

- Implement the missing parts of the code
- Write scripts to test the correctness of the B+Tree
- Test the performance of your B+ Tree
- Write a report documenting the test results

2 Loading and Running the Project

The stencil code for this project is available for you in `/course/cs127/pub/btree/stencil.tgz`. Once you have successfully copied the code to your directory and unzipped it, open it as a new Eclipse project. In Eclipse, go to `New->Java Project`, uncheck *Use default location* and direct to the project directory.

To run the project, `cd` into your btree folder and type `ant run`

If everything goes right [1], the program will display a separate window for each relation in the database. Since the program is configured to load the sample dataset from `"test/files/OneEmptyTable.xml,"` you will see one window, `TestFoods`, which visualizes the B+ Tree index for that table. You can use the buttons on the bottom on these windows to insert and delete values from the tree. We hope that this will help you to debug your project!

[1] Things won't go right before you implement the B+ Tree. Initially, the program will throw NotYetImplmentedException marking places in the code you still need to implement.

You can test your program on a different dataset by specifying the `file` property in `build.xml`. Alternatively, you can pass the name of the test file as a parameter to ant:

```
ant run -Dfile="test/files/OneEmptyTable.xml"
```

3 Implementing the Tree

The pseudocode for the major methods of the tree has been outlined in our textbook, but the project we have written for you is slightly different. Therefore, we will give you the modified pseudocode we expect you to implement. Please refer to **Appendix 2** for details.

3.1 The stencil code

The `BDIndex` interface must be implemented by all database indexes. `BDBPlusTreeIndex`, which we provide, implements this interface. However, there are five incomplete methods in its inner class `BDBPlusTree` which you must fill in. This class contains the general methods for insertion, deletion, and search of the tree (incidentally, the three methods you must fill in). It owns the tree nodes and manages splits and merges.
The following methods in `src/bdsim/server/system/index/BDBPlusTreeIndex.java` are the ones you shall care about:

- `delete(BDTuple t)`
- `delete_entry(BDBPlusTreeNode N, Comparable K, BDTuple P)`
- `delete_entry(BDBPlusTreeNode N, Comparable K, BDBPlusTreeNode P)`
- `insert(BDTuple t)`
- `insert_in_parent(BDBPlusTreeNode N, Comparable Kprime, BDBPlusTreeNode Nprime)`
- `find(Comparable value)`

You may consider adding an insert method in BDBPlusTreeNode to handle inserting into the leaf node.

MODIFICATION BEYOND THESE METHODS IS NOT RECOMMENDED.

Before you start coding, take a few minutes to look at `BDBPlusTreeNode.` Since the entire B+ Tree is made of these nodes, you should understand very well how they work inside. The class is well documented, and we added some notes to help you understand it. Hopefully you won't need to modify this class, but feel free to do so (and let us know if you find any bugs there.)

4 Testing scripts

You can run test scripts by clicking "Load Test Script" in the visualizer. The results will pop up a few moments later (the script runs synchronously, so the UI will hang while it's running). Scripts look like this:

```
insert 5
delete 5
checkNotExists 5
insert 10
checkExists 10
checkNotExists 5
insert 11
insert 12
insert 13
delete 10
checkNotExists 10
```

More formally,

- `insert <value>` Create a row whose primary key is `value` and whose other fields are null.
- `delete <value>` Delete rows with primary key `value`.
- `checkExists <value>` Ensures that exactly one row has primary key `value`.
- `checkNotExists <value>` Ensures that exactly zero rows have primary key `value`.

The TAs have given you a general stress test that you can run from the command line. `cd` into your project directory and run `ant student-test`. This only works if you are on the department machine.

We expect you to provide us with a thorough testing package for your B+ tree using this scripting language. Try to account for all edge cases you can think of, and document this in your README file.

5 Analysis and Tuning

Take your B+ tree and track the number of times that your tree will write to and read from the disk [2].We suggest that you use the fields `m_readNumDiskOps` and `m_writeNumDiskOps`. You should already have some stress tests that perform thousands of operations, so run them again and record the number of IOs. Now try changing the d-value (defined in

[2]We are looking for the number of times the code would read or write data to a disk block if each node were kept in its own disk block

`server.conf`) to at least four other values (if you're feeling unimaginitive, you can use 3, 6, 12, and 24) and report how the number of IOs changes with d. Provide graphs and a writeup explaining any observed performance trends.

6 Demo

The demo of the project is available in `/course/cs127/pub/btree/demo/`. It can be run by typing the following command in the demo directory:

```
java -jar demo.jar
```

The demo program is configured to pre-load the sample dataset from `test/files/Bank.xml`. And this command below will allow you to specify the dataset that you want to pre-load:

```
java -jar demo.jar DATASET-XML-FILE
```

(i.e. `java -jar demo.jar test/files/OneEmpTable.xml`)

Please keep in mind that only one copy of the code can be running on a computer at a time: when you are done looking at the demo, please be sure to close it (use Ctrl-C to end the task) so that you don't impede the work of other students. Conversely, if you get a networking error when starting the application, that means someone else is running their application on your computer: you should find another computer to work on.

7 Handin

Please don't forget to handin all parts of the project. To make it clear, make sure you are handing in the following:

- The code.
- Your testing package
- README file. Describe your testing package. Very briefly describe your B+tree implementation, and mention all bugs you are aware of.
- Your test report with graphs and performance discussion. Please use some universally readable format, preferably pdf.

You can handin your project by running the following command from your btree directory:

```
/course/cs127/bin/cs127_handin btree
```

Good luck!
-your TA's

8 Appendix 1: Some Friendly Pointers(Pun Definitely Intended)

8.1 d-value

The d-value is used to define the size of each node. More formally, every node will have size 2*d. Playing with this value serves as a good testing mechanism for your structure. The variable can be found in `conf/server.conf`.

8.2 Getting a BDTuple's Key Attribute

To save you the agony of having to learn the entire system's inner workings, it might be useful to know how to obtain a BDTuple's key. Here is one way to do it:

```
Comparable K = (Comparable)tuple.getObject(m_keyName);
```

8.3 Pointers at the Leaf Nodes

Unlike in standard B+ Trees, your leaf nodes do not have to implement pointers to the next node.

8.4 The `insertChild` Method

Notice that the `insertChild` method in `BDBPlusTreeNode` takes in one key and one child node. You will recall from class that every node has n keys and n+1 pointers. The `insertChild` stencil code will take care of this for you to properly insert the last child.

8.5 Duplicates

Your code should prevent insertion of duplicate keys. This will make it easier to reason about deletion.

9 Appendix 2: B+ Tree Project Pseudocode

Adapted from *Silberschatz, Korth, Sudarshan: Database System Concepts.*

9.1 find

```
find(value V) {
 clear search path
 C = root node and add to path
 while (C is not a leaf node) {
  K_i = smallest search-key value, if any, greater than V
  if (no such value) {
   m = number of pointers in the node
   C = node pointed to by P_m
  } else {
   C = node pointed to by P_i
  }
  save node in search path
 }
 if (key value K_i  in C s.t. K_i = V) {
  pointer P_i  directs us to desired record or bucket
 } else {
  no record w/ key value k exists
 }
}
```

9.2 insert

```
insert(tuple T) {
 K = the key from T
 find(the leaf node L that should contain key value K)
 if (L has < n key values) {
  L.insertTuple(K,T)
 } else {
  create node L'
  copy LP_1...LK_n to block B that can hold n+1 pairs
  B.insertTuple(K,T)
  erase LP_1...LK_n  from L
  copy BP_1...BK_{n/2} from B into L
  copy BP_{n/2 + 1}...BK_n from B into L'
  K' = smallest key-value in L'
  insert_in_parent(L,K',L')
 }
}
```

9.3 insert_in_parent

```
insert_in_parent(node N, value K', node N') {
 if (N = root) {
  create new node R containing N, K', N'
  make R root of tree
  return
 }
 P = parent(N)
 if (P has < n + 1 pointers) {
  insert (K', N') in P just after N
 } else {
  ***SPLIT P***
  copy P to block T that can hold P and (K',N')
  insert (K', N') into T just after N
  erase all entries from P
  copy T.P_1...T.P_{n/2} into P
  create node  P'
  K'' = T.K_{n/2}
  copy T.P_{n/2 + 1}...T.P_{n+1} into P'
  insert_in_parent(P,K'',P')
 }
}
```

9.4 delete

```
delete(tuple T) {
     find(leaf node L that contains key K from T)
     delete_entry(L,K,T)
}
```

9.5 delete_entry (leaf)

```
delete_entry(leaf node N, K, tuple P) {
 delete(K, P) from N
 if (N has too few values/pointers) {
  N' = previous or next child of parent(N)
  K' = value between pointers N and N' in parent(N)
  if (entries in N and N' can fit in a single node) {
   ***COALESCE NODES***
   if(N is a predecessor to N') {
    swap variables (N, N')
   }
   append all (K_i,P_i) pairs in N to N'
   delete_entry(parent(N), K', N)
  } else {
   ***REDISTRIBUTION: BORROW AN ENTRY FROM N' ***
   if(N' predecessor to N) {
    m = s.t. (N'.P_m, N'.K_m) is last tuple in N'
    replace K' in parent(N) with N'.K_m
    remove (N'.P_m, N'.K_m) from N'
    insert (N'.P_m, N'.K_m) as first pointer/value in N
   } else {
     insert (N'.P_1, K_1) into N
     remove (N'.P_1, K_1) from N'
     replace K' with new N'.K_1 in parent(N)
   }
  }
 }
}
```

9.6 delete_entry (inner node)

```
delete_entry(inner node N, value K, node P) {
 delete(K, P) from N
 if(N = root && N has 1 remaining child) {
  make the child of N the new root of the tree and delete N
 } else if (N has too few values/pointers) {
  N' = previous or next child of parent(N)
  K' = value between pointers N and N' in parent(N)
  if (entries in N and N' can fit in a single node) {
   ***COALESCE NODES***
   if(N is a predecessor to N') {
    swap variables (N, N')
   }
   append K' and all pointers and values in N to N'
   delete_entry(parent(N), K', N)
   delete node N
  } else {
   ***REDISTRIBUTION: BORROW AN ENTRY FROM N'***
   if(N' predecessor to N (ie. if N is NOT a leaf node)) {
    m = s.t. (N'.P_m last pointer in N'
    insert (N'.P_m, K') as first pointer/value in N
    replace K'  in parent(N) by N'.K_{m-1}
    remove (N'.K_{m-1}, N'.P_m) from N'
   } else {
    symmetric to the THEN case...
   }
  }
 }
}
```