Project C: B+Tree

In this last project, you will implement a B+Tree index in C++. At the end of the project, you will have a C++ class that conforms to a specific interface. Your class is then used by various command-line tools. The tools will let you create and manipulate persistent B+Tree indexes stored in virtual disks and accessed through a buffer cache that manipulates disk blocks or pages. The tools will also tell you what the performance is, in terms of how long individual operations take and how many disk reads and writes you do. The I/O model of computation is used – we only count disk time.

You can assume that requests to the B+Tree are serialized, meaning that you can finish a request before starting the next one. In a real database system, however, locking and logging are used to allow multiple requests to simultaneously execute on the tree. If you really feel ambitious, you can add support for this for extra credit.

You can assume that keys and values in the B+Tree are of fixed size and given when the B+Tree is initialized. In a real database system, however, keys and values can be of variable size. You are welcome to add support for variable length keys and values for extra credit.

You will be implementing a "pure" B+Tree. In a B+Tree, leaf nodes hold keys and values, while interior nodes hold keys and block pointers. You can also optionally chain the B+Tree leaf nodes together into a linked list, making range queries much faster.

Your class will be evaluated using a test harness that will evaluate its correctness and performance. The test harness will generate a random, but repeatable stream of requests, run them through your implementation and a reference implementation, compare the outputs, pointing out errors in your implementation, and presenting a performance number. We will grade your project based on correctness[1] using a random request stream generated from a particular seed. We won't tell you which seed, but you can test your program using lots of different seeds.

This project may be done in groups of up to three people.

Getting and installing the framework

To install the framework, log into your account on tlab-login and do the following:

```
cd ~
tar xvfz ~pdinda/HANDOUT/btree_lab.tgz
cd btree_lab
more README
```

[1] For this year, performance will **not** be part of the grading criteria.

The README file will give you detailed instructions on how to configure the framework and verify that it is working. You will be writing btree.cc and btree.h, and adding other files as you see fit. Note test.pl – it is the test harness mentioned above. ref_impl.pl is the reference implementation. You can use test_me.pl to run your implementation against the reference implementation. Your implementation will be executed via sim.cc

B+Tree operations and the command-line

At a high-level of abstraction, a B+Tree is a mapping from keys to values. B+Trees can require that all keys be unique, but it's not necessary – there is a distinction between a key in a B+Tree and a key in relational database terminology. This is also necessary for SQL. In SQL, it is perfectly OK to create an index on some attribute or set of attributes that form neither a key or superkey. Your implementation can assume that all keys are unique.

A B+Tree implementation must perform the following operations:

- Initialize: create a new B+Tree structure on the disk – "format" or "mkfs" in a file system
- Insert (key, value)
- Update (key, value)
- Delete (key)
- Lookup (key) : returns value associated with the key

In addition, your B+Tree implementation will also support the following operations:

- Sanity Check: do a self-check of the tree looking for problems – "chkdsk" or "fsck" in a file system.
- Display: do a traversal of the B+Tree, printing out the sorted (key,value) pairs in ascending order of the keys.

The btree_* tools that are built implement these operations. This lets you manipulate a B+Tree from the command line. At the end of each execution, the performance statistics are printed.

The sim tool reads a sequence of these operations, starting with an initialization, from standard input and applies them. The results of each operation are printed. At the very end, the performance statistics for the entire run are printed.

What does the B+Tree look like on the disk?

A B+Tree on the disk looks a lot like a file system on a disk. The blocks of the disk are used to store B+Tree nodes. B+Tree nodes come in two forms:

- Internal nodes: These store keys and pointers.
- Leaf nodes: These store keys and their associated values.

By pointer, what I mean is a disk block number (the blocks are numbered from 0 to the total number of blocks minus one.

The size of a block is determined when the disk is created. Since the B+Tree nodes are the size of disk blocks, we often use the words "node" and "block" interchangeably

The size of a key and the size of a value are determined when the B+Tree is initialized and need not be the same (and generally are not). Because of this variation, you will probably have to write serializers/unserializers that read and write disk blocks into appropriate in-memory structures.[2]

Generally, it a good idea to give your disk a superblock, a block, typically stored in block number zero, that describes the B+Tree (size of key, size of value, pointer to root node, pointer to free list).

You will also want to have a data structure to keep track of free and allocated blocks on the disk.[3] Notice that you can always discover all the allocated blocks by doing a traversal of the tree. However, this is quite inefficient. There are many approaches you could use to keep track of free blocks. Allocation of free space is very important in disk systems because they have non-uniform access. Allocating a block that is "close" to other blocks that are used with it is very important for performance. If you allocate blocks all in random locations, you'll have lousy performance because you'll be doing big seeks as you walk the tree.

However, since you are not going to be graded on performance, I suggest you use a simple free list. Have the superblock point to the first free node and have every free node point to the next free node. Then simply insert and remove free nodes from the front of the list.

What are the interfaces?

The framework provides the following interface to you. Notice that the interface is of a buffer cache, an intermediary between the data storage and indexing systems and the raw disk system. It keeps track of reads and writes to the actual underlying disk system. To see how to use the interface, take a look at the btree_*, and *buffer tools.

We use the I/O model of computation here and assume that your performance is dominated by these read and writes. The framework also keeps track of virtual time – the time in milliseconds that has passed since you started using the buffer cache. Virtual time passes according to I/Os.

Block: This abstracts a linear array of bytes and provides memory management.

[2] Make sure that you understand what is meant here because it is very important. You can write your B+Tree data structures as C/C++ structs, but then you also need a way to write those structs to disk blocks and read them back again. The functions that do this reading and writing are typically called serializers and unserializers.

[3] This part should sound to you very much like the malloc lab from CS 213, except here all requests are for the same size.

DiskSystem: This simulates a single disk disk system. Think of this as an IDE disk. You read and write Blocks from and to a DiskSystem. To see how to use this, look at the various *disk tools.

BufferCache: This is your main interface to the disk. It has the following properties:

- Write back: Writes are caches as well as reads.
- Write allocate: A write to a block that is not in the cache puts it into the cache.
- LRU: When a block needs to be evicted, the one that was used the longest time ago is chosen.

BTreeIndex: The interface you will provide is that of a B+Tree index for fixed length binary keys. If you implement a B+Tree (see extra credit), range scans will be much faster. A detailed, commented version of the interface is available in btree.h

Project Steps

We suggest that you take the following steps:

1. Carefully read and understand the B+Tree information in the book. You do not want to start this project without understanding what a B+Tree node should contain, and how B+Trees are kept balanced.
2. Design your on-disk data structures: superblock, interior node, and leaf node. A good way to do this is to create C/C++ structs for each one and then figure out how the struct would be represented on the disk.
3. Decide on how to do free space management and design your data structures for that.
4. Write serializers/unserializers for your superblock, interior node, leaf node, and any free space management data structures. A serializer takes an in-memory representation (an interior node, for example), and writes it to a disk block in an appropriate way. An unserializer does the reverse. If you are particularly clever, you may be able to make these very "thin".[4]
5. Write and test your code for initialization. For example, your implementation might write a superblock, a root node, and set up a free list by writing all of the free blocks..
6. Write and test your code for free space management (allocate and deallocate blocks). If you tell BufferCache when you allocate or deallocate a block, it will also keep track of things in its own internal representation. The advantage of this is that it may simplify debugging because it will warn you when you try to allocate a block that's already allocated or deallocate a block that was never allocated.)
7. Implement the B+Tree without balancing. Notice that if you set the key size large enough, you can effectively force this to be a binary tree, which is helpful for getting started. You should be able to get insert, update, delete, and lookup working and test correctness at this stage.

[4] Do not get hung up on serializers and unserializers. Ask for help and don't be afraid to do the simplest thing. For example, if your C structs contain no memory pointers and are padded to be as long as a block, you could just simply copy them in and out of memory. We'll be happy to provide help on the newsgroup.

8. Implement balancing. We suggest that you start with the balance steps needed for insertion and then do the ones needed for deletion. Insertion is much easier than deletion.
9. Do extra credit if you have time!

Where to go for help

- ⇨ Take a look at Comer's Ubiquitous B-Tree article (linked from the course web page)
- ⇨ You might find the B+-tree code in the MacFS filesystem to be interesting. The Macintosh's HFS and HFS+ filesystems use B+Trees to store directories and logical to physical block mappings. However, note that it is rather Mac-specific, and it implements variable-length keys. See http://www.cs.northwestern.edu/~pdinda/codes.html for more. Please note that attempting to copy+paste from this code will be nearly impossible.
- ⇨ Newsgroup as described on the course web page. Don't forget to help others with problems that you've already overcome.
- ⇨ Office hours. Make sure to use the office hours made available by the instructor and the TAs.

Hand-in

We will send email about this.

Extra Credit (10% Maximum)

Add fast range queries. To do this, you'll need to stitch your leaf nodes into a linked list and extend the interface of BTreeIndex.