Project C: BTree

In this last project, you will implement a BTree 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 BTree 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 BTree 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 Btree are of fixed size and given when the Btree 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" BTree. Many databases implement a B+Tree, which is a relatively straightforward extension and makes doing range queries much faster. You are welcome to do a B+Tree for extra credit.

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 and performance 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. At the end of the class, we will make available a web page showing the performance/correctness of each implementation in an anonymized form. We may also provide opportunities for competition earlier during the project for those who are interested.

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
```

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. Note test.pl – it is the test harness mentioned above. ref_impl.pl is the reference implementation. Your implementation will be executed via sim.cc

Btree operations and the command-line

At a high-level of abstraction, a Btree is a mapping from keys to values. Btrees can require that all keys be unique, but it's not necessary – there is a distinction between a key in a Btree 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. If the index is implemented as a Btree, then it must be possible for the "key" (the values that the set of attributes takes on) not to be unique.

A Btree implementation must perform the following operations:

- Initialize: create a new Btree 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
- Range Lookup (key1,key2): returns a list of (key,value) pairs, ranging from those associated with key1 to those associated with key2.

In addition, your Btree 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 BTree, 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 btree 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 Btree look like on the disk?

A BTree on the disk looks a lot like a file system on a disk. The blocks of the disk are used to store BTree nodes. BTree 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. Do not worry about pointer swizziling.

The size of a block is determined when the disk is created. The size of a key and the size of a value are determined when the BTree 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.

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

You will also want to have a data structure to keep track of free and allocated blocks on the disk. Notice that you can always discover all the allocated blocks by doing a traversal of the tree. However, this is quite inefficient. Some approaches you could track free blocks:

- Use a free space bitmap: you can create an in-memory bitmap of the allocation status of the blocks by doing a traversal at startup time. This doesn't really scale (eventually the bitmap is big enough that you have to put *it* on the disk, in which case you have the same problem you started with). However, it's OK for the project.
- Tie together the free blocks in their own data structure. For example, since the block is free, you might as well store a pointer in it. Using one pointer per block, you can tie all the free blocks into a linked list.

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.

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.

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.

The interface also provides prefetching, meaning that you can request blocks at any time and come back for them later, thus letting you overlap I/O with other work, and also to generate parallelism for the cache and to give it a chance to plan a seek strategy. NOTE: This is currently UNIMPLEMENTED and is available for extra credit. To see how to use the buffer cache, look at the various *buffer tools.

BTreeIndex: The interface you will provide is that of a BTree index for fixed length binary keys. Notice that a range scan is possible. 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 BTree information in the book. You do not want to start this project without understanding what a BTree node should contain, and how BTrees are kept balanced.
2. Design your on-disk data structures: superblock, interior node, and leaf node.
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”.
5. Write and test your code for initialization. (Write the superblock to format, read the superblock to initialize)
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 BTree 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.
8. Implement balancing. We suggest that you start with the balance steps needed for insertion and then do the ones needed for 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.
- ⇨ Newsgroups and instant messaging, 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 (30% Maximum)

Here are extra credit directions, ordered from easy to hard (in our estimation):

- B+Tree: Add B+Tree support and modify the range query to use it. Essentially, you will now need to store two pointers in each leaf node.
- Prefetch: Add prefetch support to the BufferCache.
- Roll back: Make it possible to take your BTree "go back in time" using an undo log. For extra points, put the undo log on the virtual disk.
- Atomicity and durability: Tag transaction begins and ends in your log and use them to provide these transactional semantics.
- Concurrency: Allow multiple threads to manipulate your BTree simultaneously. Note: outside of the obvious, very slow "big giant lock on the whole enchilada" approach, this is subtle and interesting. We can point you to a survey paper on BTree locking if you're interested. We strongly suggest you try the other extra credit first
- Independence. If you have logs and you have concurrency, you can implement independence. Again, check with us before you decide to do this.