Project C: BTree Index

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 sequential, 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. The nodes of a BTree are disk blocks. The leaf nodes hold keys and values, while the interior nodes hold keys and disk block pointers. You can also optionally chain the BTree leaf nodes together into a linked list, making range queries much faster. If you do this, it's known as a "B+Tree".

Your C++ 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, and compare the outputs, pointing out errors in your implementation. We will grade your project based on correctness 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, key sizes, and value sizes.

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 Murphy and do the following:

```
cd ~
tar xvfz ~pdinda/339/HANDOUT/btree/btree_lab.tgz
cd btree_lab
export PATH=$PATH:.
touch .dependencies
make depend clean
```

```
make
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`, and adding other files as you see fit. You'll see "WRITE ME" comments in these files to indicate where you probably want to work. 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.

The code should compile and work on any Unix-like environment, provided you have gcc and Perl. If you want graphical displays of your BTrees, you'll also need to have AT&T's GraphViz package (which is free).   Murphy has all the necessary tools installed. Your code will be tested on Murphy.

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. Your implementation, however, can assume that all keys are unique.

Your BTree implementation (your C++ class) must perform the following operations:
- *Initialize*: create a new BTree structure on the disk --- this is like "format" or "`mkfs`" in a file system.
- *Attach*: open a BTree for use.   In our API, this is combined with initialization (the "create" argument). You can ask for the BTree to be attached with our without initialization.   This is like mounting a file system.
- *Insert (key, value)* – insert the key/value pair
- *Update (key, value)* – change the value associated with an existing key
- *Delete (key)* – delete the key/value pair associated with the given key ⬅ *You do not need to do this for Fall 2014, but you can do it for extra credit*
- *Lookup (key)* - return the value associated with the key
- *SanityCheck()* - do a self-check of the tree looking for problems - this is like "`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 are built on top of these operations. These tools let you manipulate a BTree using your C++ class 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 using your C++ class. The results of each operation are printed. At the very end, the performance statistics for the entire run are printed.

If your *Display()* function also supports output in the dot format, which is described in a comment in the code, you can use `btree_display.pl` to show your tree graphically. The dot format is basically just a description of graph.

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 to other disk blocks.
- Leaf nodes: These store keys and their associated values.

By pointer, what I mean is a disk block number. The disk 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 BTree nodes are the size of disk blocks, we often use the words "node" and "block" interchangeably. On a typical system, the disk blocks are also the same size as the pages the operating system uses to manage memory.

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 need serializers/unserializers that read and write disk blocks into appropriate in-memory structures.[1] The project handout includes BTree C++ data structures with such serialization support. These are in the files `btree_ds.{ch}`. You are welcome to use them, or write your own.

Generally, it a good idea to give your disk a superblock, which is a block, typically stored in block number zero, that describes the BTree. For example, it might contain the size of the keys, the size of the values, a pointer to the root node, a pointer to the free list, etc.

You will also want to have a data structure to keep track of free and allocated blocks on the disk. This data structure will support functions to allow you to allocate and deallocate blocks, similar to C++'s `new` and `delete` (or C's `malloc()`/`free()`), except that all allocations/deallocations are of the same size (one disk block/node).
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 latencies. Allocating a block that is "close" to other blocks that are used with it is very important for performance. If you allocate

---

[1] Make sure that you understand what is meant here because it is very important. You can write your BTree 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. An unserializer is a function that reads a disk block and constructs the relevant C/C++ struct from the content, while a serializer takes the C/C++ struct and writes a disk block with its content.

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. Another alternative is to use a simple bit vector, with a bit for each block on the disk. If the bit is zero, the block is free. A simple free list allocator is included in the code we give you.

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.

The interface includes these C++ classes:

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

DiskSystem: This simulates a single disk disk system. Think of this as a single hard disk. You read and write Blocks from and to a DiskSystem. These read/writes are durable. To see how to use this, look at the various `*disk` tools.

BufferCache: This simulates an in-memory cache of disk blocks, and it is your main interface to the disk. That is, you typically read/write the disk through this interface. While it can be much faster than using the disk directly, read/writes via the buffer cache are also not durable. The cache has the following properties:
- Write back: Writes are cached 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.

In addition, for this project, we are also providing the following partial implementation, again a collection of C++ classes:

BTreeNode: This an implementation of a BTree node (root, interior, and leaf), which is provided in `btree_ds.{ch}`. It is a C++ class that provides serialization to/from the DiskSystem via the BufferCache as Block. That is, a BTreeNode is the same size as a Block. *You are welcome to write your own implementation of a BTree node if you'd like, or you can use this one.*

BTreeIndex: The interface you will provide is that of a BTree index for fixed length binary keys.   You can optionally add range queries based on leaf-node linking (see extra credit). A detailed, commented version of the interface is available in `btree.h`. *BTreeIndex is partially implemented*. **Init, Attach, Lookup, and Display are already implemented for you. We also provide a simple free list allocator**. *You are welcome to write your own implementation of BTreeIndex or extend this one.*

Advice

It's very easy to become overwhelmed with the code in this project.   There are almost 3400 lines of C++! When you're asked to work in the context of an existing codebase, which is almost always the case, you have to adopt strategies that do not involve trying to read and understand all of the code. Some strategies that are useful include:

- Walk through several specific execution paths instead of the whole codebase. In this project, you could focus on understanding *Init/Attach* and *Lookup* to start with.   If you don't know how to use a symbolic debugger (i.e., gdb) to do this, now is a great time to learn.
- Follow the abstractions only as far as you need to.   In this project, there is a lot of code that implements the buffer cache and the disk system simulation.   In order to create a correct implementation of BTree, you don't need to know any of it.   You just need to know how to make a virtual disk, how to read and write the buffer cache, and how to assure that the buffer cache is flushed when your tree is detached. Later, you may want to dig deeper if you decide to make your implementation faster.
- Play with it. Code is intended to be run, not stared at. This codebase includes a number of tools for exploration, including graphical display (`btree_display.pl`).

For a sense of scope of the project, my implementation is about 650 lines of C++ code in addition to those I've given you, and that includes the Delete functionality, which you do not need to do.

Suggested 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.   You may also find Comer's survey article on BTrees helpful.
2. Understand the on-disk data structures: superblock, interior node, and leaf node. **We provide implementations of these in `btree_ds.{h,cc,README}`. You are welcome to use them or write your own. We strongly suggest you look at our on-disk data structures implementation first.**
3. Make sure you understand serialization/unserialization of your superblock, interior node, leaf node, and any free space management data structures. We provide an implementation for the nodes in `btree_ds.{h,cc,README}` which you're welcome to use, or you can roll your own.

4. Make sure you understand the code for initialization and attaching the BTree, or write your own.
5. Understand the implementation of *Lookup*, or write your own.
6. Extend the code to implement *Update*.   This should be very easy to do if you understand *Lookup*.   Note that you won't be able to test your code for this until you have at least *Insert* also working.
7. Determine whether you will use the free space allocator we provide or if you will write your own.   The code we give you sets up a simple explicit free list. You're welcome to design your own free space management approach to replace this one.
8. Implement *Insert*.   *Insert* is not really something that you can implement incrementally. You need to design your approach and implement it en masse. Fortunately, the descriptions given in the book and in Comer are very detailed. Overall, you will probably find that this is simplest to do using recursion, walking down the tree to figure out where to insert, and then walking up the tree as you split nodes.   Then there is a special case if you have to split the root.
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 BTrees 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://pdinda.org/codes.html for more. Please note that attempting to copy+paste from this code will be nearly impossible.
⇨ Our course discussion group. Don't forget to help others with problems that you've already overcome.
⇨ **OFFICE HOURS AND RECITATION SECTION.** Make sure to use the office hours made available by the instructor and the TAs.
⇨ We may provide additional code snippets/etc over time. If we do, we will announce them and make them available in ~pdinda/339/HANDOUT/btree.

Hand-in

We will send email about this.

Extra Credit (30% Maximum)

As with previous projects, if you are interested in doing extra credit, talk to us first so that we can determine the maximum number of points possible.

Implement Delete.   The first step here is to implement *Delete* without key recovery from internal nodes. *Delete* is more difficult than *Insert* because the key to be deleted can exist both in the leaves and in interior nodes.   Begin with an implementation that leaves deleted keys in internal nodes and simply removes them from leaves. Notice that this still requires that you do merging of nodes, so you'll search down to a leaf, delete, and then merge back up to the root. Again, this will be easiest to do with recursion. Once you've completed this step, you'll have a perfectly correct BTree implementation.

The only problem is that it will slowly accumulate garbage (deleted keys in interior nodes).   The next step will be to do key recovery from internal nodes.   Be sure you read the book and Comer carefully before starting this part. Note that to test your *Delete* functionality, you will need to modify gen_test_sequence.pl and uncomment the DELETE and DELETE_EXISTS operations.

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. A range query for keys between min and max then boils down to a *Lookup* for the min key, and then a forward scan through the linked list until the max key is found.   This will turn your BTree into a B+Tree, allowing you to do range queries in $O(\log(n)+m)$ time, where n is the number of keys in the tree, and m is the number of keys in the range.

Add support for variable length keys and values. This is self-explanatory.

Implement a locking protocol (something better than just one big lock) for the tree so that operations do not have to be serialized.

Implement a logging approach (redo or undo logging) so that you can rollback changes to the tree in case of a failed transaction.