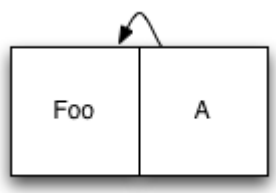
Damn Cool Algorithms: Log structured storage

Posted by Nick Johnson | Filed under tech, damn-cool-algorithms

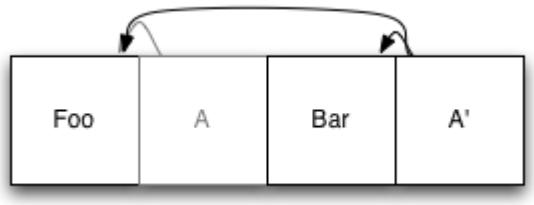
Typically, if you're designing a storage system – such as a filesystem, or a database – one of your major concerns is how to store the data on disk. You have to take care of allocating space for the objects to be stored, as well as storing the indexing data; you have to worry about what happens when you want to extend an existing object (eg, appending to a file), and you have to take care of fragmentation, which happens when old objects are deleted, and new ones take their place. All of this adds up to a lot of complexity, and the solutions are often buggy or inefficient.

Log structured storage is a technique that takes care of all of these issues. It originated as Log Structured File Systems in the 1980s, but more recently it's seeing increasing use as a way to structure storage in database engines. In its original filesystem application, it suffers from some shortcomings that have precluded widespread adoption, but as we'll see, these are less of an issue for database engines, and Log Structured storage brings additional advantages for a database engine over and above easier storage management.

The basic organization of a log structured storage system is, as the name implies, a log – that is, an append-only sequence of data entries. Whenever you have new data to write, instead of finding a location for it on disk, you simply append it to the end of the log. Indexing the data is accomplished by treating the metadata the same way: Metadata updates are also appended to the log. This may seem inefficient, but disk-based index structures such as B-Trees are typically very broad, so the number of index nodes we need to update with each write is generally very small. Let's look at a simple example. We'll start off with a log containing only a single item of data, and an index node that references it:



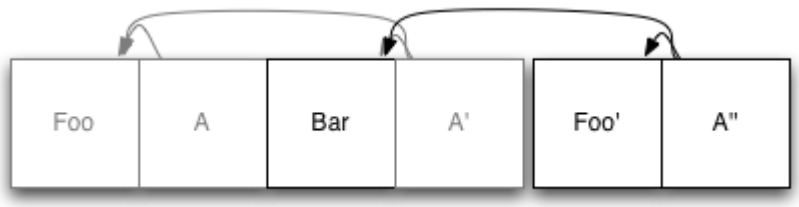
So far so good. Now, suppose we want to add a second element. We append the new element to the end of the log, then we update the index entry, and append the updated version of that to the log, too:



The original index entry (A) is still in the logfile, but it's no longer used: It's been replaced by the new entry, A', which refers to the original, unmodified copy of Foo, as well as the new entry, Bar. When something wants to read our filesystem, it finds the root node of the index, and uses it as it would in any other system using disk-based indexing.

Finding the root of the index warrants a quick aside. The naive approach would simply be to look at the last block in the log, since the last thing we write is always the root of the index. However, this isn't ideal, as it's possible that at the time you try to read the index, another process is halfway through appending to the log. We can avoid this by having a single block – say, at the start of the logfile – that contains a pointer to the current root node. Whenever we update the log, we rewrite this first entry to ensure it points to the new root node. For brevity, we haven't shown this in the diagrams.

Next, let's examine what happens when we update an element. Say we modify Foo:

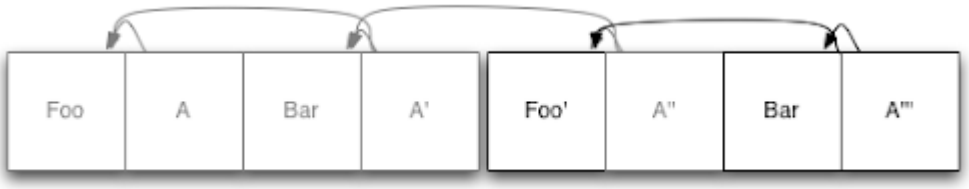


We started by writing an entirely new copy of Foo to the end of the log. Then, we again updated the index nodes (only A' in this example) and wrote them to the end of the log as well. Once again, the old copy of Foo remains in the log; it's just no longer referenced by the updated index.

You've probably realised that this system isn't sustainable indefinitely. At some point, we are going to run out of storage space, with all this old data sitting around taking up space. In a filesystem, this is dealt with by treating the disk as a circular buffer, and overwriting old log data. When this happens, data that is still valid simply gets appended to the log again as if it was freshly written, which frees up the old copy to be overwritten.

In a regular filesystem, this is where one of the shortcomings I mentioned earlier rears its ugly head. As the disk gets fuller, the filesystem needs to spend more and more of its time doing garbage collection, and writing data back to the head of the log. By the time you reach 80% full, your filesystem practically grinds to a halt.

If you're using log structured storage for a database engine, however, this isn't a problem! We're implementing this on top of a regular filesystem, so we can make use of it to make our life easier. If we split the database into multiple fixed-length chunks, then when we need to reclaim some space, we can pick a chunk, rewrite any still active data, and delete the chunk. The first segment in our example above is beginning to look a bit sparse, so let's do that:



All we did here was to take the existing copy of 'Bar' and write it to the end of the log, followed by the updated index node(s), as described above. Now that we've done that, the first log segment is entirely empty, and can be deleted.

This approach has several advantages over the filesystem's approach. For a start, we're not restricted to deleting the oldest segment first: If an intermediate segment is nearly empty, we can choose to garbage collect that, instead. This is particularly useful for databases that have some data that stays around for an extended time, and some data that gets overwritten repeatedly: We don't want to waste too much time rewriting the same unmodified data. We also have some more flexibility about when to garbage collect: we can usually wait until a segment is mostly obsolete before garbage collecting it, further minimising the amount of extra work we have to do.

The advantages of this approach for a database don't end there, though. In order to maintain transactional consistency, databases typically use a "Write Ahead Log", or WAL. When a database wants to persist a transaction to disk, it first writes all the changes to the WAL, flushes those to disk, then updates the actual database files. This allows it to recover from a crash by 'replaying the changes recorded in the WAL. If we use log structured storage, however, the Write Ahead Log *is* the database file, so we only need to write data once. In a recovery situation, we simply open the database, start at the last recorded index header, and search forward linearly, reconstructing any missing index updates from the data as we go.

Taking advantage of our recovery scheme from above, we can further optimise our writes, too. Instead of writing the updated index nodes with every write, we can cache them in memory, and only write them out to disk periodically. Our recovery mechanism will take care of reconstructing things in a crash, as long as we provide it some way to distinguish completed transactions from incomplete ones.

Backups are also easier with this approach: We can continuously, incrementally back up our database by copying each new log segment to backup media as it is completed. To restore, we just run the recovery process again.

One final major advantage to this system relates to concurrency and transactional semantics in databases. In order to provide transactional consistency, most databases use complex systems of locks to control which processes can update data at what times. Depending on the level of consistency required, this can involve readers taking out locks to make sure data is not modified while they are reading it, as well as writers locking data for write, and can cause significant performance degradation even with relatively low write rates, if enough concurrent reads are occurring.

We can beat this with Multiversion Concurrency Control, or MVCC. Whenever a node wants to read from the database, it looks up the current root index node, and uses that node for the remainder of its transaction. Because existing data is never modified in a log-based storage

system, the process now has a snapshot of the database at the time it grabbed the handle: Nothing a concurrent transaction can do will affect its view of the database. Just like that, we have lock-free reads!

When it comes to writing data back, we can make use of Optimistic concurrency. In a typical read-modify-write cycle, we first perform our read operations, as described above. Then, to write our changes, we take the write lock for the database, and verify that none of the data we read in the first phase has been modified. We can do this quickly, by looking at the index, and checking if the address for the data we care about is the same as when we last looked. If it's the same, no writes have occurred, and we can proceed with modifying it ourselves. If it's different, a conflicting transaction occurred, and we simply roll back and start again with the read phase.

With me singing its praises so loudly, you may be wondering what systems already use this algorithm. There are surprisingly few that I'm aware of, but here's a few notable ones:

- Although the original Berkeley DB uses a fairly standard architecture, the Java port, BDB-JE uses all of the components we just described.
- CouchDB uses the system just described, except that instead of dividing logs into segments and garbage collecting them, it rewrites its entire database when enough stale data accumulates.
- PostgreSQL uses MVCC, and its writeahead logs are structured so as to permit the incremental backup approach we described.
- The App Engine datastore is based on Bigtable, which takes a different approach to on-disk storage, but the transactional layer uses optimistic concurrency.

If you know of other database systems that use the ideas detailed in this post, let us know in the comments!

14 December, 2009

Previous Post Next Post

Comments

Blogroll

- Nick Johnsonz
- Bill Katz
- Coding Horror
- Craphound
- Neopythonic
- Schneier on Security

Home | Atom | CSS | XHTML