Log Structured Merge Trees

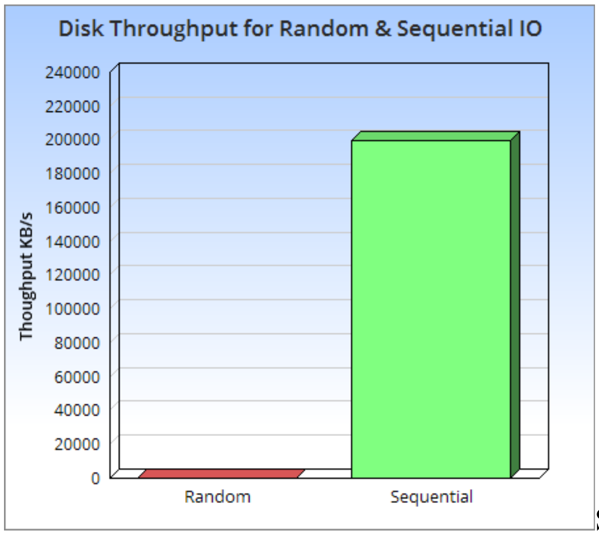
It's nearly a decade since Google released its 'Big Table' paper. One of the many cool aspects of that paper was the file organisation it uses. The approach is more generally known as the Log Structured Merge Tree, after this 1996 paper, although the algorithm described there differs quite significantly from most real-world implementations.

LSM is now used in a number of products as the main file organisation strategy. HBase, Cassandra, LevelDB, SQLite, even MongoDB 3.0 comes with an optional LSM engine, after it's acquisition of Wired Tiger.

What makes LSM trees interesting is their departure from binary tree style file organisations that have dominated the space for decades. LSM seems almost counter intuitive when you first look at it, only making sense when you closely consider how files work in modern, memory heavy systems.

Some Background

In a nutshell LSM trees are designed to provide better write throughput than traditional B+ tree or ISAM approaches. They do this by removing the need to perform dispersed, update-in-place operations.



So why is this a good idea? At its core it's the old problem of disks being slow for random operations, but fast when accessed sequentially. A gulf exists between these two types of access, regardless of whether the disk is magnetic or solid state or even, although to a lesser extent, main memory.

The figures in this ACM report here/here make the point well. They show that, somewhat counter intuitively, sequential disk access is faster than randomly accessing main memory. More relevantly they also show sequential access to disk, be it magnetic or SSD, to be at least three orders of magnitude faster than random IO. This means random operations are to be avoided. Sequential access is well worth designing for.

So with this in mind lets consider a little thought experiment: if we are interested in write throughput, what is the best method to use? A good starting point is to simply append data to a file. This approach, often termed logging, journaling or a heap file, is fully sequential so provides very fast write performance equivalent to theoretical disk speeds (typically 200-300MB/s per drive).

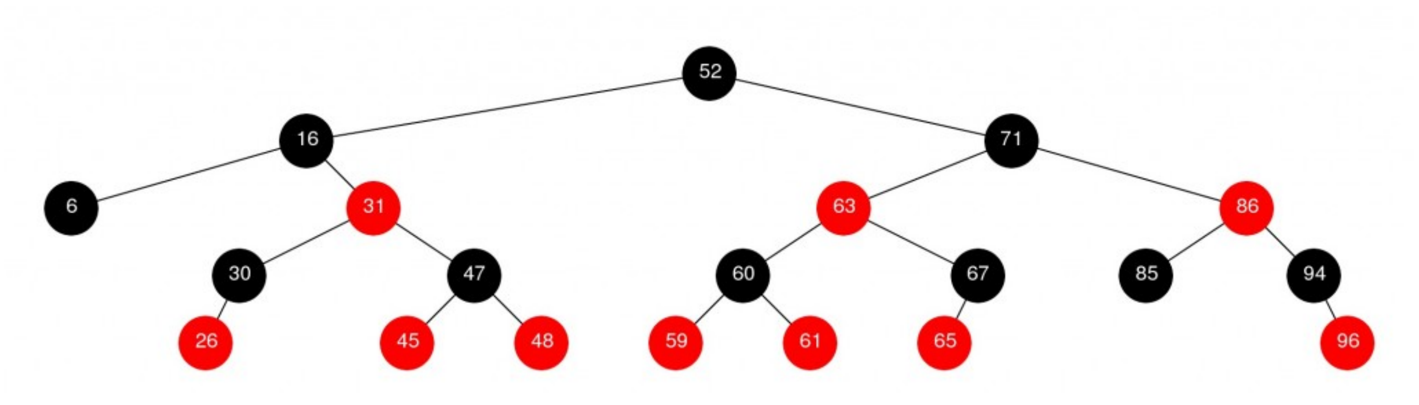
Benefiting from both simplicity and performance log/journal based approaches have rightfully become popular in many big data tools. Yet they have an obvious downside. Reading arbitrary data from a log will be far more time consuming than writing to it, involving a reverse chronological scan, until the required key is found.

This means logs are only really applicable to simple workloads, where data is either accessed in its entirety, as in the write-ahead log of most databases, or by a known offset, as in simple messaging products like Kafka.

So we need more than just a journal to efficiently perform more complex read workloads like key based access or a range search. Broadly speaking there are four approaches that can help us here: binary search, hash, B+ or external.

1. Search Sorted File: save data to a file, sorted by key. If data has defined widths use Binary search. If not use a page index + scan.
2. Hash: split the data into buckets using a hash function, which can later be used to direct reads.
3. B+: use navigable file organisation such as a B+ tree, ISAM etc.
4. External file: leave the data as a log/heap and create a separate hash or tree index into it.

All these approaches improve read performance significantly (n->O(log(n)) in most). Alas these structures add order and that order impedes write performance, so our high speed journal file is lost along the way. You can't have your cake and eat it I guess.



An insight that is worth making is that all four of the above options impose some form of overarching structure on the data.

Data is deliberately and specifically placed around the file system so the index can quickly find it again later. It's this structure that makes navigation quick. Alas the structure must of course be honoured as data is written. This is where we start to degrade write performance by adding in random disk access.

There are a couple of specific issues. Two IOs are needed for each write, one to read the page and one to write it back. This wasn't the case with our log/journal file which could do it in one.

Worse though, we now need to update the structure of the hash or B+ index. This means updating specific parts of the file system. This is known as update-in-place and requires slow, random IO. *This point is important:* in-place approaches like this scatter-gun the file system performing update-in-place*. This is limiting.

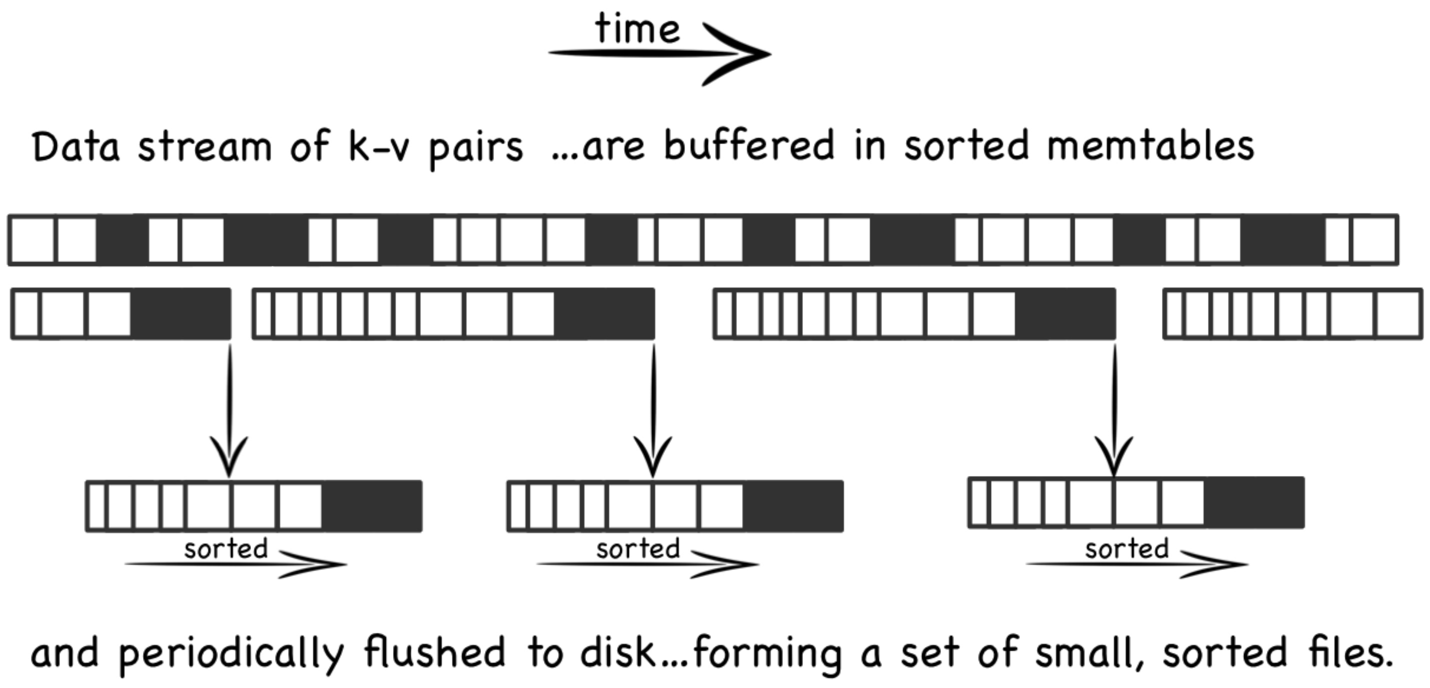
One common solution is to use approach (4) A index into a journal – but keep the index in memory. So, for example, a Hash Table can be used to map keys to the position (offset) of the latest value in a journal file (log). This approach actually works pretty well as it compartmentalises random IO to something relatively small: the key-to-offset mapping, held in memory. Looking up a value is then only a single IO.

On the other hand there are scalability limits, particularly if you have lots of small values. If your values were just say simple numbers then the index would be larger than the data file itself. Despite this the pattern is a sensible compromise which is used in many products from Riak through to Oracle Coherence.

So this brings us on to Log Structured Merge Trees. LSMs take a different approach to the four above. They can be fully disk-centric, requiring little in memory storage for efficiency, but also hang onto much of the write performance we would tie to a simple journal file. The one downside is slightly poorer read performance when compared to say a B+Tree.

In essence they do everything they can to make disk access sequential. No scatter-guns here!

*A number of tree structures exist which do not require update-in-place. Most popular is the append-only Btree, also know as the copy-on-write tree. These work by overwriting the tree structure, sequentially, at the end of the file each time a write occurs. Relevant parts of the old tree structure, including the top level node, are orphaned. Through this method update-in-place is avoided as the tree sequentially redefines itself over time. This method does however come at the cost: rewriting the structure on every write is verbose. It creates a significant amount of write amplification which is a downside unto itself.


time
Data stream of k-v pairs ...are buffered in sorted memtables
sorted
sorted
sorted
and periodically flushed to disk...forming a set of small, sorted files.

The Base LSM Algorithm

Conceptually the base LSM tree is fairly simple. Instead of having one big index structure (which will either scatter-gun the file system or add significant write amplification) batches of writes are saved, sequentially, to a set of smaller index files. So each file contains a batch of changes covering a short period of time. Each file is sorted before it is written so searching it later will be fast. Files are immutable; they are never updated. New updates go into new files. Reads inspect all files. Periodically files are merged together to keep the number of files down.

Lets look at this in a little more detail. When updates arrive they are added to an in-memory buffer, which is usually held as a tree (Red-Black etc) to preserve key-ordering. This ‘memtable’ is replicated on disk as a write-ahead-log in most implementations, simply for recovery purposes. When the memtable fills the sorted data is flushed to a new file on disk. This process repeats as more and more writes come in. Importantly the system is only doing sequential IO as files are not edited. New entries or edits simply create successive files (see fig above).

So as more data comes into the system, more and more of these immutable, ordered files are created. Each one representing a small, chronological subset of changes, held sorted.

As old files are not updated duplicate entries are created to supersede previous records (or removal markers). This creates some redundancy initially.

Periodically the system performs a *compaction.* Compaction selects multiple files and merges them together, removing any duplicated updates or deletions (more on how this works later). This is important both to remove the aforementioned redundancy but, more importantly, to keep a handle on the read performance which degrades as the number of files increases. Thankfully, because the files are sorted, the process of merging the files is quite efficient.

When a read operation is requested the system first checks the in memory buffer (memtable). If the key is not found the various files will be inspected one by one, in reverse chronological order, until the key is found. Each file is held sorted so it is navigable. However reads will become slower and slower as the number of files increases, as each one needs to be inspected. This is a problem.

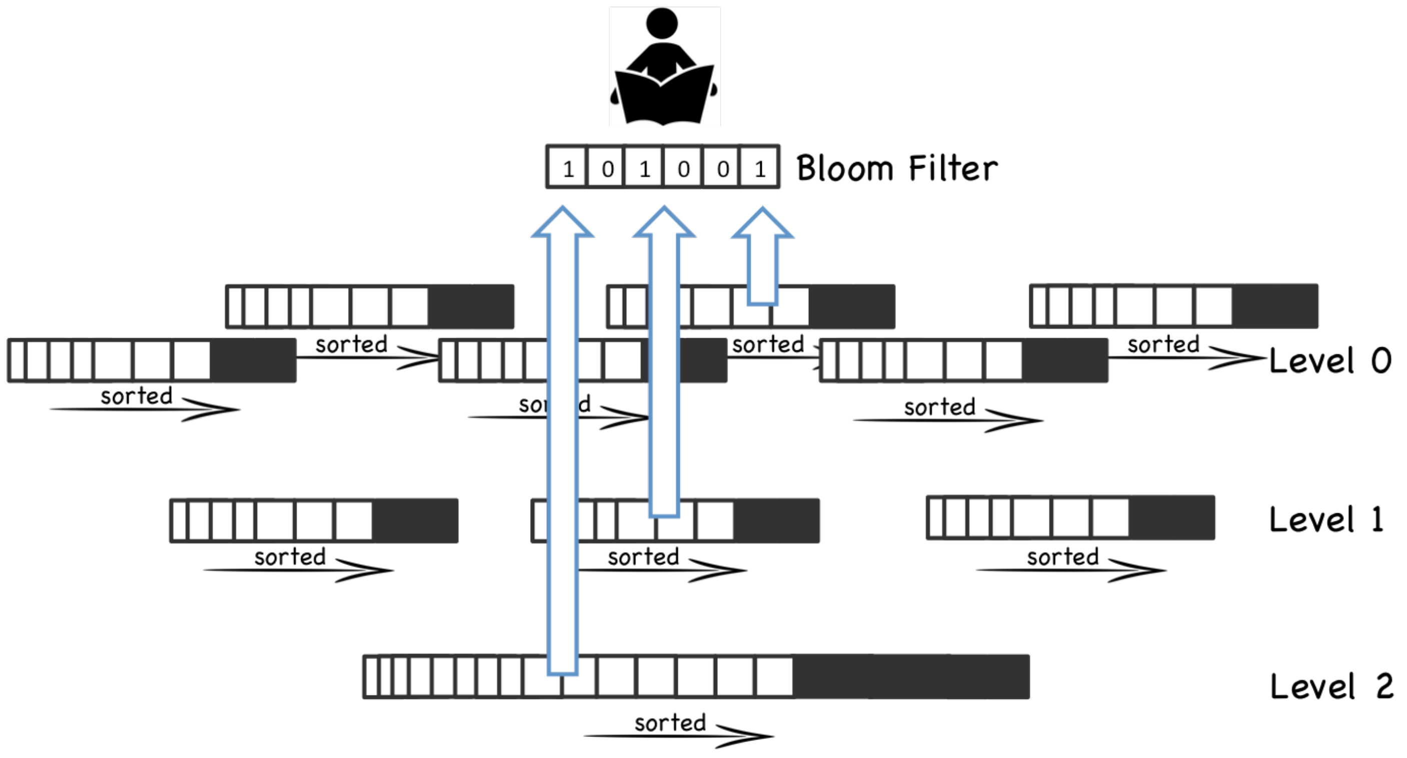
So reads in LSM trees are slower than their in-place brethren. Fortunately there are a couple of tricks which can make the pattern performant. The most common approach is to hold a page-index in memory. This provides a lookup which gets you ‘close’ to your target key. You scan from there as the data is sorted. LevelDB, RocksDB and BigTable do this with a block-index held at the end of each file. This often works better than straight binary search as it allows the use of variable length fields and is better suited to compressed data.

Even with per-file indexes read operations will still slow as the number of files increases. This is kept in check by periodically merging files together. Such compactions keep the number of files, and hence read performance, within acceptable bounds.

Even with compaction reads will still need to visit many files. Most implementations void this through the use of a Bloom filter. Bloom filters are a memory efficient way of working out whether a file contains a key.

So from a ‘write’ perspective; all writes are batched up and written ***only*** in sequential chunks. There is an additional, periodic IO penalty from compaction rounds. Reads however have the potential to touch a large number of files when looking up a single row (i.e. scatter-gun on read). This is simply the way the algorithm works. We’re trading random IO on write for random IO on read. This trade off is sensible if we can use software tricks like bloom filters or hardware tricks like large file caches to optimise read performance.

As elements of a record could be in any level all levels must be consulted. Thus bloom Filters are used to avoid files unnecessary reads.



Basic Compaction

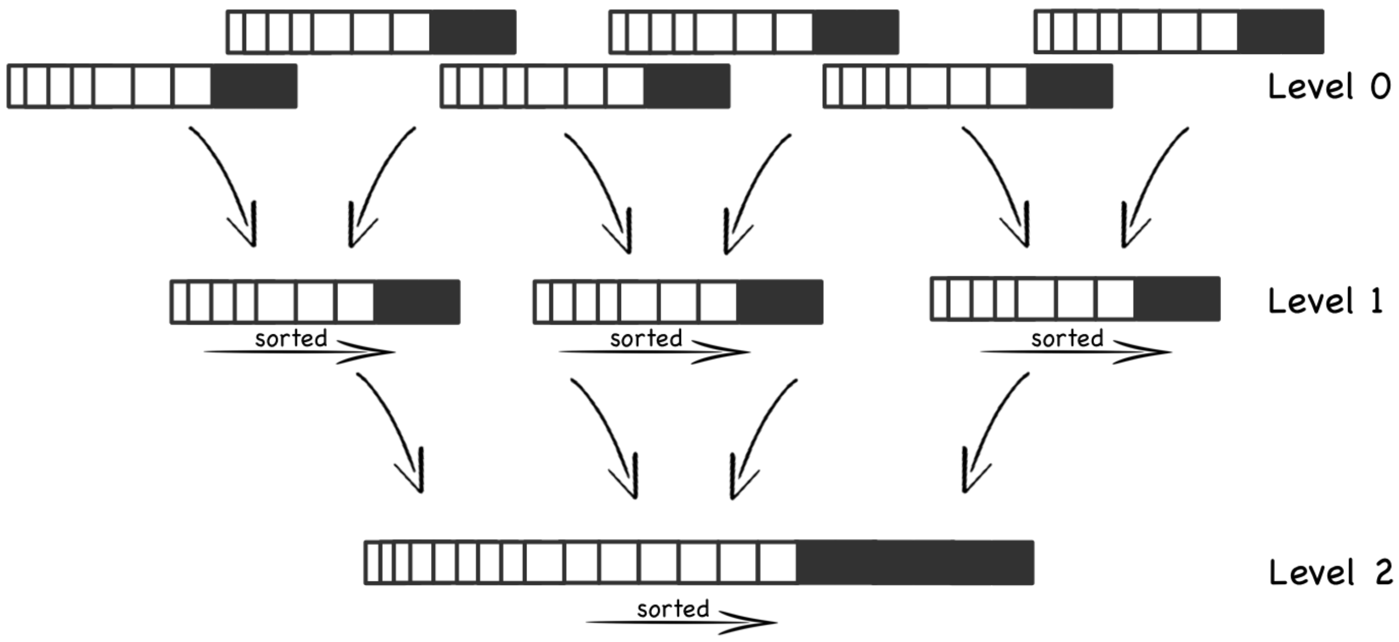
To keep LSM reads relatively fast it’s important to manage-down the number of files, so lets look more deeply at compaction. The process is a bit like generational garbage collection:

When a certain number of files have been created, say five files, each with 10 rows, they are merged into a single file, with 50 rows (or maybe slightly less) .

This process continues with more 10 row files being created. These are merged into 50 row files every time the fifth file fills up.

Eventually there are five 50 row files. At this point the five 50 row files are merged into one 250 row file. The process continues creating larger and larger files. See fig.

The aforementioned issue with this general approach is the large number of files that are created: all must be searched, individually, to read a result (at least in the worst case).



Compaction continues creating fewer, larger and larger files

Levelled Compaction

Newer implementations, such as those in LevelDB, RocksDB and Cassandra, address this problem by implementing a level-based, rather than size-based, approach to compaction. This reduces the number of files that must be consulted for the worst case read, as well as reducing the relative impact of a single compaction.

This level-based approach has two key differences compared to the base approach above:

1. Each level can contain a number of files and is guaranteed, as a whole, to not have overlapping keys within it. That is to say the keys are partitioned across the available files. Thus to find a key in a certain level only one file needs to be consulted.

The first level is a special case where the above property does not hold. Keys can span multiple files.

2. Files are merged into upper levels one file at a time. As a level fills, a single file is plucked from it and merged into the level above creating space for more data to be added. This is slightly different to the base-approach where several similarly sized files are merged into a single, larger one.

These changes mean the level-based approach spreads the impact of compaction over time as well as requiring less total space. It also has better read performance. However the total IO is higher for most workloads meaning some of the simpler write-oriented workloads will not see benefit.

Summarising

So LSM trees sit in the middle-ground between a journal/log file and a traditional single-fixed-index such as a B+ tree or Hash index. They provide a mechanism for managing a set of smaller, individual index files.

By managing a group of indexes, rather than a single one, the LSM method trades the expensive random IO associated with update-in-place in B+ or Hash indexes for fast, sequential IO.

The price being paid is that reads have to address a large number of index files rather than just the one. Also there is additional IO cost for compaction.

If that's still a little murky there are some other good descriptions here and here.

Thoughts on the LSM approach

So are LSM approaches really better than traditional single-tree based ones?

We've seen that LSM's have better write performance albeit a cost. LSM has some other benefits though. The SSTables (the sorted files) a LSM tree creates are immutable. This makes the locking semantics over them much simpler. Generally the only resource that is contended is the memtable. This is in contrast to singular trees which require elaborate locking mechanisms to manage changes at different levels.

So ultimately the question is likely to be about how write-oriented expected workloads are. If you care about write performance the savings LSM gives are likely to be a big deal. The big internet companies seem pretty settled on this subject. Yahoo, for example, reports a steady progression from read-heavy to read-write workloads, driven largely by the increased ingestion of event logs and mobile data. Many traditional database products still seem to favour more read-optimised file structures though.

As with Log Structured file systems [see footnote] the key argument stems from the increasing availability of memory. With more memory available reads are naturally optimised through large file caches provided by the operating system. Write performance (which memory doesn't improve with more) thus becomes the dominant concern. So put another way, hardware advances are doing more for read performance than they are for writes. Thus it makes sense to select a write-optimised file structure.

Certainly LSM implementations such as LevelDB and Cassandra regularly provide better write performance than single-tree based approaches (here and here respectively).

Beyond Levelled LSM

There has been a fair bit of further work building on the LSM approach. Yahoo developed a system called Pnuts which combines LSM with B trees and demonstrates better performance. I haven't seen openly available implementations of this algorithm though. IBM and Google have done more recent work in a similar vein, albeit via a different path. There are also related approaches which have similar properties but retain an overarching structure. These include Fractal Trees and Stratified Trees.

This is of course just one alternative. Databases utilise a huge range of subtly different options. An increasing number of databases offer pluggable engines for different workloads. Parquet is a popular alternative for HDFS and pushes in pretty much the opposite direction (aggregation performance via a columnar format). MySQL has a storage abstraction which is pluggable with a number of different engines such as Toku's fractal tree based index. This is also available for MongoDB. Mongo 3.0 includes the Wired Tiger engine which provides both B+ & LSM approaches along with the legacy engine. Many relational databases have configurable index structures that utilise different file organisations.

It's also worth considering the hardware being used. Expensive solid state disks, like FusionIO, have better random write performance. This suits update-in-place approaches. Cheaper SSDs and mechanical drives are better suited to LSM. LSM's avoid the small random access patters that thrash SSDs into oblivion**.

LSM is not without it critics though. It's biggest problem, like GC, is the collection phases and the effect they have on precious IO. There is an interesting discussion of some of these on this hacker news thread.

So if you're looking at data products, be it BDB vs. LevelDb, Cassandra vs. MongoDb you may tie some proportion of their relative performance back to the file structures they use. Measurements appear to back this philosophy. Certainly it's worth being aware of the performance tradeoffs being selected by the systems you use.

**In SSDs each write incurs a clear-rewrite cycle for a whole 512K block. Thus small writes can induce a disproportionate amount of churn on the drive. With fixed limits on block rewrites this can significantly affect their life.

Further Reading

- There is a nice introductory post here.
- The LSM description in this paper is great and it also discusses interesting extensions.
- These three posts provide a holistic coverage of the algorithm: here, here and here.
- The original Log Structured Merge Tree paper here. It is a little hard to follow in my opinion.
- The Big Table paper here is excellent.
- LSM vs Fractal Trees on High Scalability.
- Recent work on Diff-Index which builds on the LSM concept.
- Jay on SSDs and the benefits of LSM
- Interesting discussion on hackernews regarding index structures.

Footnote on log structured file systems

Other than the name, and a focus on write throughput, there isn't that much relation between LSM and log structured file systems as far as I can see.

Regular filesystems used today tend to be 'Journaling', for example ext3, ext4, HFS etc are tree-based approaches. A fixed height tree of inodes represent the directory structure and a journal is used to protect against failure conditions. In these implementations the journal is logical, meaning it only internal metadata will be journaled. This is for performance reasons.

Log structured file systems are widely used on flash media as they have less write amplification. They are getting more press too as file caching starts to dominate read workloads in more general situations and write performance is becoming more critical.

In log structured file systems data is written only once, directly to a journal which is represented as a chronologically advancing buffer. The buffer is garbage collected periodically to remove redundant writes. Like LSM's the log structured file system will write faster, but read slower than its dual-writing, tree based counterpart. Again this is acceptable where there is lots of RAM available to feed the file cache or the media doesn't deal well with update in place, as is the case with flash.

Posted on February 14th, 2015 in Analysis and Opinion, Big Data, Blog, Top4

---

12 Comments

Jump to comment form | comments rss