A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees

Bradley C. Kuszmaul
Chief Architect
Tokutek
bradley@tokutek.com

April 22, 2014

1 Introduction

I will use this white paper to lead a discussion of how Fractal Trees compare to Log-Structured Merge Trees.

This paper explains the advantages of Fractal-Tree® indexing compared to Log-Structured Merge (LSM) trees. LSM trees were originally described by O'Neil [13], and have been implemented in several systems including [8–10,12,14,17]. Fractal-Tree indexes are based on research on streaming B trees [4], which drew in part on earlier algorithmic work on buffered repository trees [6, 7]. Fractal-Tree indexes appear in Tokutek's database products [16].

I'll compare these data structures both using asymptotic analysis, and for a "typical" database of 10TB of data on a machine with 64GB of RAM, for several measures of quality: write amplification, read amplification, and space amplification. I'll try to express these in terms of the block size B, the fanout k, the memory size M, and the database size N, although it can be difficult to compare different data structures because, for example, they choose different fanouts depending on their characteristics.

2 The Objective

This section describes the quality metrics I'll be using, which include write amplification, read amplification, and space amplification.

2.1 Write Amplification

Write amplification is the amount of data written to storage compared to the amount of data that the application wrote. LSM trees and Fractal-Tree indexes both provide significant advantages over traditional B-trees. For example, if a database row contains 100 bytes, and a B tree such as InnoDB employs 16KiB pages [11], then the B tree may perform 16KiB of I/O to write a single 100-byte row, for a write amplification of 160, compared to a write amplification of 30 to 70 for the other data structures.

This simple measure of write amplification understates the advantage of LSM trees and Fractal Trees, however, since a write amplification of 160 with 16KiB blocks, is an order of magnitude worse than the same write amplification would be with 1MiB blocks (at least for rotating disks). Since the goal of this paper is to compare LSM trees to Fractal Trees, I won't worry too much about the fact this metric makes B trees look better than they are.

Write amplification is a problem on both rotating disks and solid-state disk (SSD), but for different reasons. As I mentioned above, for rotating disks, write amplification does not tell the whole story, because many small writes are slower than one big write of the same total number of bytes. But both LSM trees and Fractal-Tree indexes perform large writes, for which all we care about is bandwidth. Disks often have a limited amount of bandwidth (say, 100MB/s per drive).

For SSD, write amplification is a problem because flash-based SSD's can be written to only a finite number of times. A B tree performing random updates can easily use up a flash. Lower write amplification can help to reduce cost because you can buy cheaper flash (with fewer write cycles), and you may be able to provision a higher fraction of your flash storage to store useful data since the write blocks are larger.

Larger write blocks help SSD for the following reasons. The Flash Translation Layer (FTL) of a flash-based solid-state disk (SSD) can increase the write amplification experienced by the flash memory inside the SSD. For example, for an SSD provisioned at 80% utilization, with an erase-block size of 256KiB and B-tree blocks of 16KiB under a workload with uniform random writes, the FTL induces another factor of 5 of write amplification. In general, if the SSD is provisioned with a fraction r of its storage made available to the user, and blocks are written randomly, and if the written blocks are much smaller than the erase-block size, then the write amplification induced by the FTL is another $1/(1-r)$.

Compression can further reduce write amplification. Reducing the uncompressed write amplification can reduce the CPU load, and hence improve performance, by reducing the amount of data that needs to be compressed. Using lightweight compressors can mitigate this cost, at the cost of reduced compression factors, and hence higher write amplification. So lower uncompressed write amplification can enable the use of a more expensive

compressor that, in turn, further lowers the write amplification seen at the disk.

Depending on the data structure, write amplification can be affected by the application's write pattern. For example, writing random uniformly distributed keys (e.g., as in iiBench [15]) will induce different write amplification than writing skewed keys with a Zipfian distribution [18] (e.g., as in LinkBench [2]).

2.2 Read Amplification

Read amplification is the number of I/O's required to satisfy a particular query. There are two important cases for read amplification: cold cache, and warm cache. For example, for a B tree in the cold-cache case, a point query requires $O(\log_B N)$ I/O's, whereas for most reasonable configurations of RAM and disk, in the warm-cache case the internal nodes of the B tree are cached, and so a B tree requires at most one I/O per query. The choice of uniform random vs. skewed can also affect read amplification

Some reads are performed for *point queries*, where we are simply looking up a particular key. Some reads are performed for range queries, where we want all key-value pairs for which the key lies in a particular range. These two problems are different because, for example, Bloom filters [5] can help with point queries, but do not seem to help with range queries.

Note that the units are different for write amplification and read amplification. Write amplification measures how much more data is written than the application thought it was writing, whereas read amplification counts the number of I/O's to perform a read.

2.3 Space Amplification

The space required by a data structure can be inflated by fragmentation or requirements for temporary copies of the data. For example, B trees typically achieve only 75% space utilization due to fragmentation inside the B-tree blocks. Thus B trees suffer a space amplification of $4/3$.

3 B Trees

Before diving into LSM trees and fractal trees, I'll first quickly review B trees. A B-tree index is a search tree in which every node of the tree can have many children.[1] A B tree

[1]There are many flavors of B trees, including, for example, plain B trees, B^+ trees, and B^* trees. It turns out that to a first approximation, the differences between these flavors is unimportant. The trees that I'll describe are B^+ trees, but I'll just call them "B trees". Unfortunately the literature on B trees employs

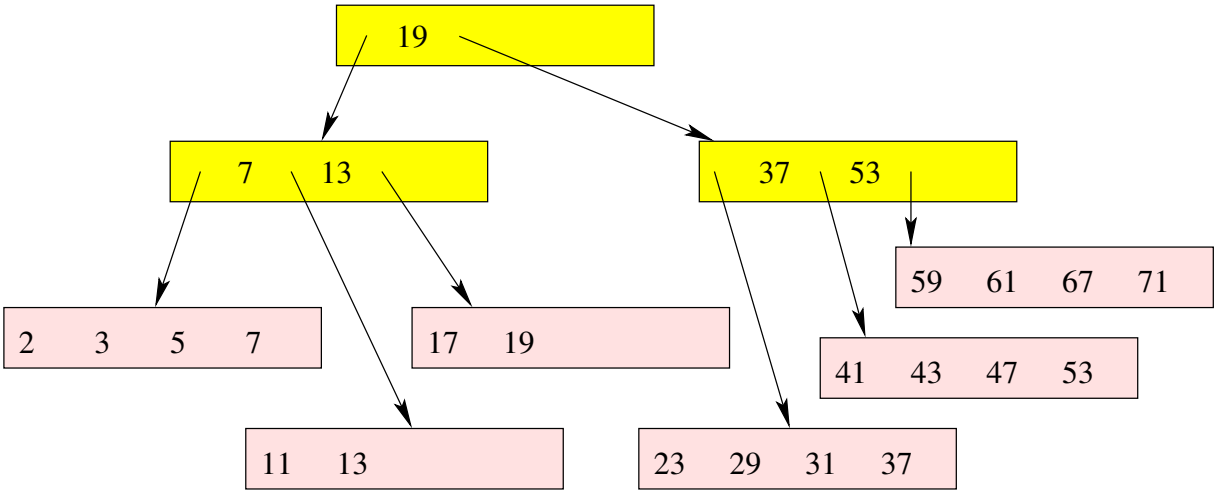

Figure 1: A B tree (adapted from [3]) containing the first twenty prime numbers. The internal nodes of the tree are shaded yellow, and contain pivot keys, whereas the leaf nodes are shaded pink and contain only data records. The root node is shown at the top of the tree, and in this case happens to contain a single pivot (19), indicating that records with key k where $k \leq 19$ are stored in the first child, and records with key k where $k > 19$ are stored in the second child. The first child contains two pivot keys (7 and 13), indicating that records with key k where $k \leq 7$ is stored in the first child, those with $7 < k \leq 13$ are stored in the second child, and those with $k > 13$ are stored in the third child. The leftmost leaf node contains four values (2, 3, 5, and 7).

contains two kinds of nodes, leaf nodes, and internal nodes. A leaf node contains data records and has no children, whereas an internal node has children and *pivot keys*, but no data records. There is a single root node, which has no parent. Every other node has a parent. An example B tree appears in Figure 1.

Typically, B trees are organized to have uniform depth (that is, the distance from the root a leaf node is same for every leaf node).

How big are the nodes of a B tree? The theory literature (see, for example, the Disk-Access Model (DAM) of [1]) usually assumes that the underlying disk drives employ a fixed-sized block, and sets the block size of the B tree to match that device block size. To simplify the analysis, the theory assumes that all blocks are equally expensive to access (real disks exhibit locality between bocks, and furthermore the blocks near the outer edge of an disk are faster than the ones near the inner edge). The theory further assumes that

inconsistent terminology, so I'll have to explain what I mean by terms such as "leaf".

there is no such thing as a small block or a large block, and that all we need to do is count the number of blocks fetched from and stored to disk. The theory denotes the block size of the tree as B, and assumes that keys, pointers, and records are constant sized, so that each internal node contains $O(B)$ children and each leaf node contains $O(B)$ data records. (The root node is a special case, and can be nearly empty in some situations.) In spite of all these unrealistic assumptions, the DAM theory works very well. In the DAM model, the depth of a B tree is

$$O(\log_B N/B) = O\left(\frac{\log N/B}{\log B}\right),$$

where N is the size of the database.

How expensive is it to search or insert data records? The number of disk I/O's required to search for an arbitrary key is at most $O(\log_B N/B)$, since that's the depth of the tree. An insertion also requires $O(\log_B N/B)$ disk I/O's. In a system with cache, once the cache is warmed up all the internal nodes of the tree are typically cached, and so a search or insertion may cost only one I/O.

One common misconception that I have heard is that insertions are expensive because maintaining the tree's balance is difficult. (To maintain tree's "balance" includes maintaining the invariant that the tree has uniform depth and every node has $O(B)$ children.) Although maintaining B-tree balance is tricky (the code to do so is often quite complex), almost no additional I/O's are incurred to keep the tree balanced, however.

In practice, most engineered B trees choose a block size B measured in bytes. The keys and data records are of various sizes, so real B trees typically simply fill up their nodes as much as possible. That means that the number of children at each node (and hence the depth of the tree) depends on the sizes of the pivot keys.

What is the write amplification for a B tree? For the worst-case insertion workloads, every insertion requires writing the leaf block containing the record, so the write amplification is B. You might get lucky and write two or more records into the same block before evicting the block from the cache, but if your database is substantially larger than your cache, and you perform random insertions, most insertions incur a full block write. To understand how bad this write amplification is, we need to examine how block sizes are chosen for B-trees.

The block size for B trees varies quite a bit depending on the implementation: Many databases employ 4KiB blocks or 16KiB blocks, but some use larger blocks. Databases that perform many writes seem to favor small block sizes, whereas databases that perform mostly reads seem to favor large block sizes, sometimes going up to a mgabyte or more. This is because large blocks worsen the write amplification, but improve search performance by making the tree shallower.

It turns out that 4KiB or 16KiB is far too small to get good performance on rotating

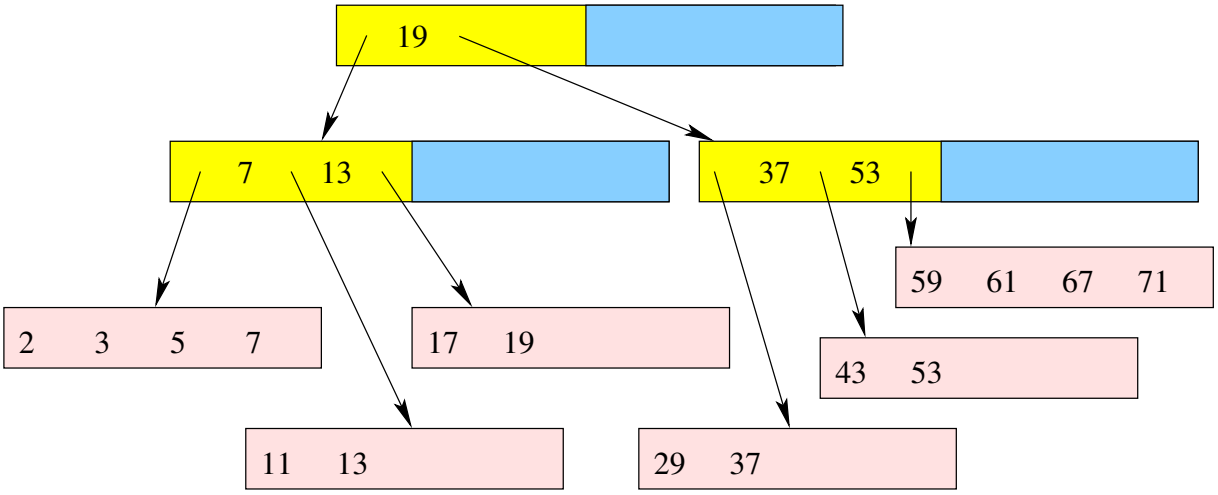

Figure 2: An FT index containing the first twenty numbers (except that 23, 31, 41, and 43 are missing). This data structure contains buffers in the internal nodes (shown in blue). In the FT index shown here, the buffers are empty.

disk, since for a typical disk, the seek time might be 5ms and the transfer rate might be 100MB/s. If you write 4KiB blocks, you will spend 5ms seeking, and 0.04ms transferring data, thus achieving only 819KB/s transfer rates on a disk that should have been able to do 100MB/s. To make good use of the disk, we really need the block size to be at the large end of the range (close to a megabyte). B trees thus force database engineers into a unpleasant corner.

4 Fractal Tree Indexes

Having discussed B trees, I'll now discuss a look at Fractal Tree indexes, which I'll abbrevate as FT indexes. The idea behind FT indexes is to maintain a B tree in which each internal node of the tree contains a buffer.

An example FT index is shown in Figure 2. This tree is very similar to the B tree of Figure 1, except it has extra buffers, which happen to be empty.

Figure 3 shows how an FT index uses the buffers. When a data record is inserted into the tree, instead of traversing the entire tree the way a B tree would, we simply insert the data record into the buffer at the root of the tree.

Eventually the root buffer will fill up with new data records. At that point the FT index copies the inserted records down a level of the tree. Eventually the newly inserted records

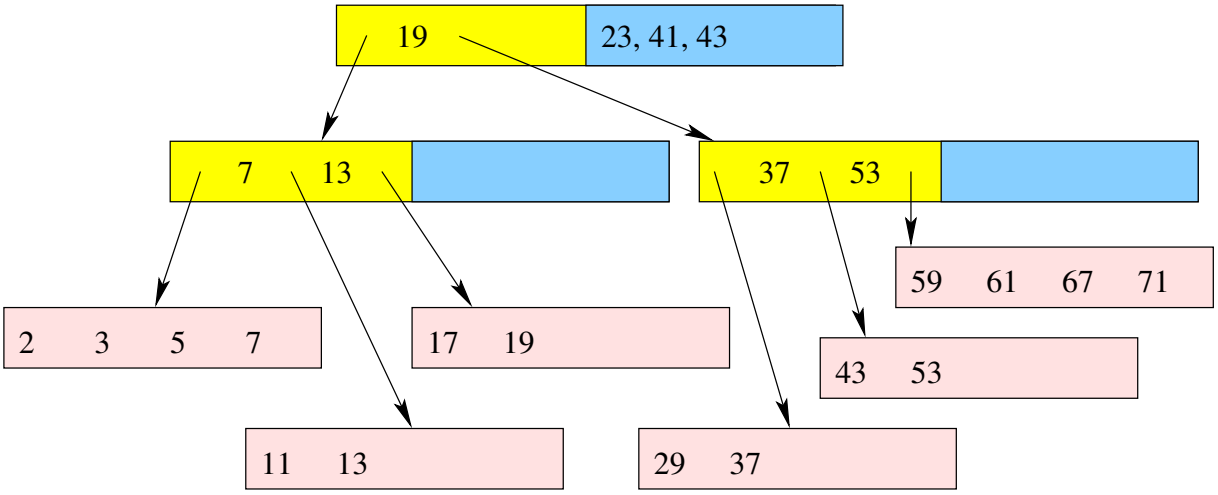

Figure 3: The FT index of Figure 2 after we have inserted 23, 41, and 43. Those newly inserted records now appear in the buffer at the root of the tree.

will reach the leaves, at which point they are simply stored in a leaf node as a B tree would store them.

The data records descending through the buffers of the tree can be thought of as messages that say "insert this record". FT indexes can use other kinds of of messages, such as messages that delete a record, or messages that update a record.

Since we are using part of the B-tree internal node for a buffer, that means there is less space for pivot keys. How much space should we allocate for the buffer, and how much for the children.

To make the analysis work out in theory, we can set the number of children $k = \sqrt{B}$. (Recall, for a B-tree, $k = B$).

In practice we might set $k = 10$ and the block size B to 64KB. Internal nodes can be bigger if the buffers are full. Several leaf nodes are grouped together into a single 4MB block for writing, so that for writes the block size is 4MB (before compression) and reads are 64KB (before compression).

FT write amplification can be analyzed as follows. In the worst case, each object moves down the tree, once per level. Moving B/k objects down the tree incurs the write cost of B, for write amplification per level of k, and a total write amplification of $O(k \log_k N/B)$. Skewed writes can achieve substantially better write amplification, sometimes ending up with performance as good as $O(\log_k N/B)$.

Read cost is $O(\log_k N/B)$ for the cold-cache case, and typically one I/O for the warm

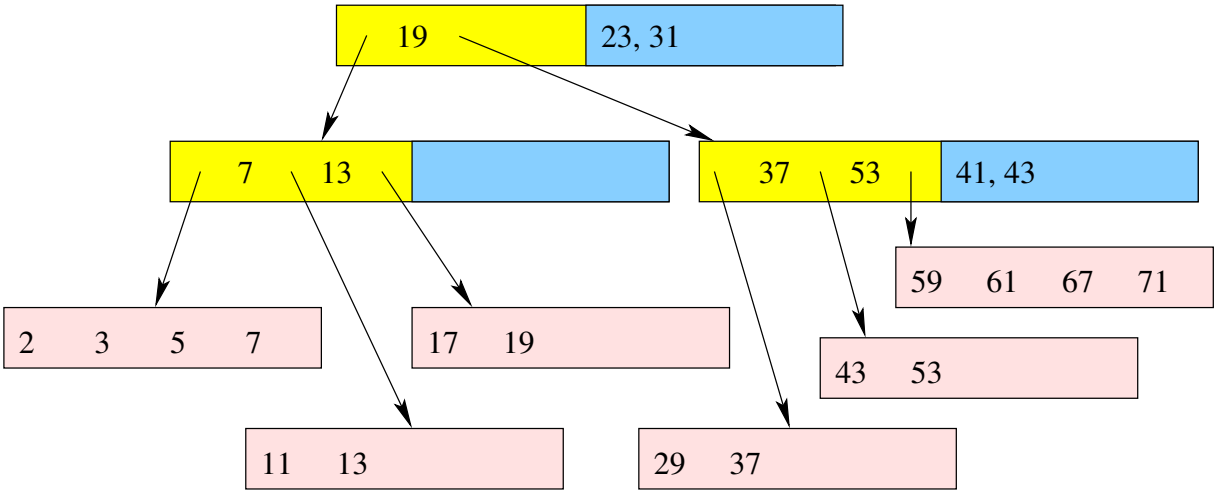

Figure 4: The FT index of Figure 3 after we have inserted 31, and the tree has moved 41 and 43 into a child of the root.

cache case, both for point queries and range queries.

5 LSM Trees

The idea behind LSM trees is to maintain a collection of sorted *runs* of data, each run containing key-value pairs with the keys sorted in ascending order. In some systems a run is managed as a single file, and in some systems a run is managed as a collection of smaller files. Some systems do both: For example, Cassandra's leveled compaction strategy organizes runs into files that are at most 5MB by default, whereas their size-tiered compaction strategy uses a single file per run [9].

Figure 5 shows a simple LSM tree with three runs in which two runs (one of length 4 and one of length 6) are merged into one run of length 10, leaving the third run unchanged.

Given a sorted run, one way to perform a query is to perform binary search on the data. Such a query requires $O(\log N/B)$ I/O's in a cold cache, where N is the database size and B is the read block size. That is a lot of I/Os: For a 10TB database with a 1MB read-block size, the cold-cache read cost is about 26 reads. Changing the read-block size might help, but smaller reads would require more I/Os and larger reads are more expensive, and my calculations suggest that a 1MB read-block size is close to optimal for rotating disks. The warm-cache case can be much better: If the system can keep 100 million keys in main memory, then it can decide which block to read and read the right block with a single I/O.

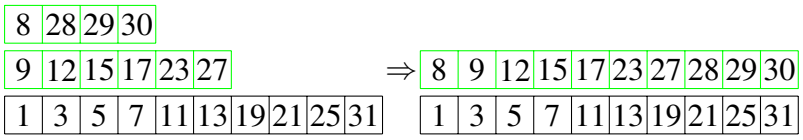

Figure 5: An LSM with three runs (left) in which two runs are merged (right). The runs being merged are rendered in green.

If there are multiple runs, then a query may require a single I/O for each run. In practice, the smaller runs are typically cached. To get a better handle on the tradeoffs between reads and writes, we need to look closer at how LSM trees handle compaction. LSM trees never overwrite old data runs. Instead, they write all updates into new runs. Over time, many versions of a row may exist in different runs, and finding a row could involve looking in all those runs. To maintain good read performance *compaction* merges two or more runs together into a single run. This merging can be done at disk bandwidth, since the various runs being merged can be read sequentially, and the new run can be written sequentially.

LSM trees come in two flavors: *Leveled*, and *Size-tiered*. I'll analyze each flavor separately in the rest of this section.

5.1 Leveled LSM Trees

In *leveled* LSM trees, data is organized into levels. Each level contains one run. Data starts in level 0, then gets merged into the level 1 run. Eventually the level 1 run is merged into the level 2 run, and so forth. Each level is constrained in its sizes. For example, in some systems, Level i contains between 10^{i-1} and 10^i megabytes. So Level 1 would be between 1MB and 10MB, Level 2 would be between 10MB and 100MB, Level 3 would be up to 1GB, and so forth. So a 10TB database would contains 7 levels. Some systems use different growth factors. I'll analyze for the general case and for the particular case in which the growth factor equals 10.

Figure 6 shows a leveled LSM with runs that grow by a factor of 5. A run of length 5 is shown being merged into a run of length 15.

In the single-file-per-run case, merging a run from one level into the next requires reading both runs, and writing a new run. This merging can require temporary storage which is twice the size of the database.

In the multiple-files-per-run case, merging can be performed by reading one file at a time from the first level, and merging it into the relevant files in the second run. In the

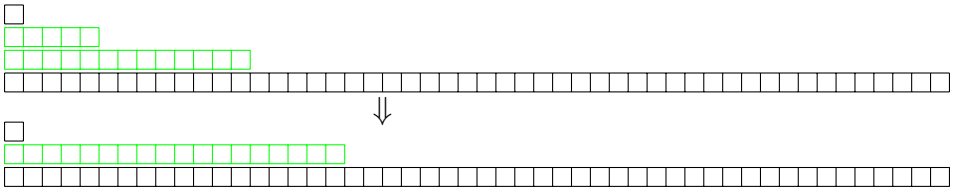

Figure 6: A leveled LSM with runs that grow by a factor of 10. Before merging two rows (above) and after (below).

worst case, this approach can also require twice the space, since one could have a single file that contains a row of data destined to be merged with each file from the next level.

Leveled compaction is implemented by Cassandra [9], LevelDB [12], and RocksDB [14], and was also the approach in the original LSM paper [13].

We can analyze leveled LSM trees as follows. If the growth factor is k and the smallest level is a single file of size B, then the number of levels is $\Theta(\log_k N/B)$. Data must be moved out of each level once, but data from a given level is merged repeatedly with data from the previous level. On average, after being first written into a level, each data item is remerged back into the same level about $k/2$ times. So the total write amplification is $\Theta(k \log_k N/B)$.

To perform a short range query in the cold cache case, we must perform a binary search on each of the runs. The biggest run is $O(N)$, thus requires $O(\log N/B$ I/O's. The next run is size $O(N/k)$, and so requires $O(\log(N/(kB))) = O((\log N/B) - \log k)$ I/O's. The next run requires $O(\log(N/(k^2B))) = O((\log N/B) - 2\log k)$ I/O's. The total number of I/O's is thus, asymptotically,

$$R = (\log N/B) + (\log N/B - \log k) + (\log N/B - 2\log k) + \cdots + 1$$

which has solution $R = O((\log N/B)(\log_k N/B)) = O((\log^2 N/B)/\log k)$. For our hypothetical 10TB database, that comes out to about 93 reads for a cold cache.

For a warm cache we need one read per level that doesn't fit in main memory. For a 64GB machine and a 10TB database, all but the last three levels fit in main memory, so the system requires 3 reads. That factor of 3 compares unfavorably with a B tree's single read for a short range query.

Point queries can run more efficiently by adding a Bloom filter to each run, avoiding the need to do a disk I/O for levels that do not actually contain the row of interest. In most cases, this means that a point query requires one read.

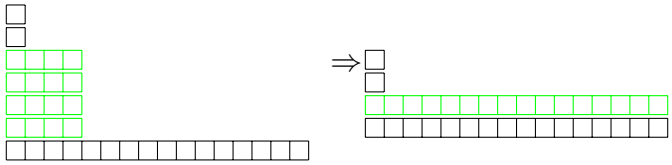

Figure 7: A size-tiered LSM with runs that grow by a factor of 4. There can be up to 4 runs of each size. Before merging four rows (left) and after (right).

5.2 Size-tiered LSM Trees

A second approach to LSM maintenance is called size-tiered compaction. For size-tiered systems, the runs are implemented each with a single file, and the runs are not a fixed size either for analysis nor in practice. The size-tiered files tend to get larger as they get older.

Figure 7 shows a size-tiered LSM being compacted. Four runs of length 4 are combined to make a single run of length 16.

A typical approach is used in Cassandra [9], where they find four runs that are about the same size, and merge them together to make a larger run. For a 10TB database, one could have a large run that's near 10TB, four runs which are between 500GB and 2.5TB, four runs that are between 100GB and 500GB, and so forth.

The worst-case write amplification for this system can be analyzed as follows. If the fanout is k ($k = 4$ for Cassandra, for example) there are $O(\log_k N/B)$ levels. Data is written to each level once. So writes incur $O(\log_k N/B)$ write amplification. This is substantially less than the $O(k \log_k N/B)$ write amplification of leveled LSM trees even though k is typically smaller for size-tiered: making the factor of $\log_k N/B$ larger (by making k smaller) is not nearly as important as multiplying the number of writes by k.

Reads can require checking up to $O(k \log_k N/B)$ files. For the cold-cache case, reading the files requires $O(k(\log^2 N/B)/\log k)$ which is k times more than the leveled approach since one must read k files at each level. This works out to 17 I/Os for our hypothetical 100TB database with 64GB RAM.

For the warm cache case, for our hypothetical 10TB database on a 64GB machine, there can be 13 or more files that are bigger than main memory. For example, one might find a situation in which there are

- 1 run of size 4TB,
- 4 runs of size 1TB,
- 4 runs of size 250GB, and

- 4 runs of size 65GB,

for a total of 13 files. Asymptotically that's $O(k \log_k N/M)$ files, where M is the memory size. A bloom filter can avoid most of those reads in the case of a point query, but not for range queries.

The multi-file leveled approach used by LevelDB and RocksDB has a space amplification of about 1.1 since about 90% of the data is the highest level. Size-tiered compaction does not seem able to get space amplification that low. RocksDB is configured by default to have space amplification of 3, but it may be possible to reduce that at the cost of one hurting read amplification or write amplification.

Size-tiered compaction is implemented by Cassandra [9], WiredTiger [17], HBase [10], and RocksDB [14]. The Bigtable paper [8] did not describe their LSM carefully enough to know for sure: it looks like a size-tiered compaction scheme, which is confusing since LevelDB, which is often described as an open-source version of the same data structure, is leveled.

Like for file-per-run leveled LSM's, one potential problem is that the system may require temporary storage equal to the size of the database to perform compaction.

6 Summary

Figure 8 shows a summary of the various kinds of amplification.

- FT indexes provide excellent write amplification, read amplification, and space amplification, both asymptotically and in practice.

- Leveled LSM's can basically match the write amplification of FT indexes. For many-files-per-run Leveled LSM's do pretty well for space amplification, but not so well for file-per-run. Leveled LSM's incur significantly more read amplification than FT indexes both asymptotically and in practice. Very few applications should use leveled LSM's.

- Size-tiered LSM's can provide much lower write amplification than Leveled LSM's or FT indexes at a horrible cost for range query and pretty bad space amplification. A workload that performs few reads and is not sensitive to space amplification, but is very sensitive to write costs might benefit from size-tiered LSM's. In my experience, it is an unusual application that is so short on write bandwidth that it can gladly give up a factor of 13 on read amplification.

Data Structure	Write Amp (worst case)	Read Amp (range) (cold cache)	Read Amp (range) (warm)	Space Amp
B Tree	$O(B)$	$O(\log_B N/B)$	1	1.33
FT index	$O(k \log_k N/B)$	$O(\log_k N/B)$	1	negligible
LSM leveled	$O(k \log_k N/B)$	$O((\log^2 N/B)/\log k)$	3	2 (file-per-run) 1.1 (many files)
LSM size-tiered	$O(\log_k N/B)$	$O(k(\log^2 N/B)/\log k)$	13	3

Figure 8: A summary of the write amplification, read amplification for range queries, and space amplification of the various schemes.

- B trees have much higher write amplification than the other alternatives, and are good at read amplification, and pretty good at space amplification. Very few applications should use B trees.

The data structure used in a database is only part of the decision. Features such as MVCC, transactions with ACID recovery and two-phase distributed commit, backup, and compression often dominate the engineering challenges of these systems and often determine whether the database actually meets the needs of an application. For example, compression works very well for FT indexes, and can dramatically reduce write amplification. Tokutek's implementation of fractal tree indexes provides MVCC, transactions, distributed transactions, backup, and compression.

Acknowledgments

Mark Callaghan helped me understand how to explain LSM trees.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. 31(9):1116–1127, September 1988.

[2] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: A database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1185–1196, New York, NY, 2013.

[3] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, Feb. 1972.

[4] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 81–92, San Diego, CA, June 9–11 2007.

[5] B. H. Bloom. Spacetime trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 39–48, 2002.

[7] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00)*, pages 859–860, 2000.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2), June 2008. Article No. 4.

[9] J. Ellis. Leveled compaction in Apache Cassandra. `http://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra`, Oct. 2011. Viewed April 22, 2014.

[10] Apache HBase. `https://hbase.apache.org/`. Viewed April 14, 2014.

[11] InnoBase. InnoDB pluggable storage engine for MySQL.

[12] LevelDB. `https://code.google.com/p/leveldb/`. Viewed April 14, 2014.

[13] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[14] RocksDB. `rocksdb.org`, 2014. Viewed April 19, 2014.

[15] Tokutek. iiBench Contest. `http://blogs.tokutek.com/tokuview/iibench`, Nov. 2009.

[16] Tokutek. MongoDB, MySQL, and MariaDB performance using Tokutek. `tokutek.com`, 2014.

[17] WiredTiger. wiredtiger.com. Viewed April 14, 2014.

[18] G. K. Zipf. *The Psychobiology of Language*. Houghton-Mifflin, 1935.