The TokuFS Streaming File System

John Esmet
Tokutek &
Rutgers

Michael A. Bender
Tokutek &
Stony Brook

Martin Farach-Colton
Tokutek &
Rutgers

Bradley C. Kuszmaul
Tokutek &
MIT

Abstract

The TokuFS file system outperforms write-optimized file systems by an order of magnitude on microdata write workloads, and outperforms read-optimized file systems by an order of magnitude on read workloads. Microdata write workloads include creating and destroying many small files, performing small unaligned writes within large files, and updating metadata. TokuFS is implemented using Fractal Tree indexes, which are primarily used in databases. TokuFS employs block-level compression to reduce its disk usage.

1 Introduction

File system designers often must choose between good read performance and good write performance. For example, most of today's file systems employ some combination of B-trees and log-structured updates to achieve a good tradeoff between reads and writes. TokuFS employs neither B-trees nor log-structured updates, however, and achieves performance that dominates today's file systems by more than an order of magnitude. This paper describes the data structures and algorithms of TokuFS, and presents performance measurements of TokuFS against several traditional file systems.

At one extreme, update-in-place file systems [7, 9, 13] keep data and metadata indexes up-to-date as soon as the data arrives. These file systems optimize for queries by, for example, attempting to keep all the data for a single directory together on disk. Data and metadata can be read quickly, especially for scans of related data that are together on disk, but the file system may require one or more disk seeks per insert, update, or delete.

At the other extreme, logging file systems [4, 12], log file-system updates rapidly to disk. Logging ensures that files can be created and updated rapidly, but queries, such as reads or metadata lookups, may suffer from the lack of an up-to-date index or from poor locality in indexes that are spread through the log.

Large-block reads and writes, which we call ***macrodata*** operations, can easily run at disk bandwidth. For small writes, which we call ***microdata*** operations, in which the bandwidth time to write the data is much smaller than a disk seek, the tradeoff becomes more severe. Examples of microdata operations include creating or destroying ***microfiles*** (small files), performing small writes within large files, and updating metadata (e.g., inode updates).

In this paper, we introduce the TokuFS file system. TokuFS achieves good performance for both reads and writes and for both microdata and macrodata. Compared to ext4, XFS, Btrfs, and ZFS, TokuFS runs at least 18.4 times faster than the nearest competitor (Btrfs) for creating microfiles and 13.5 times faster than the nearest competitor (ext4) for reading data. Although today's file systems make a tradeoff between reads and writes, they are nowhere near the actual optimal tradeoff curve.

TokuFS uses Fractal Tree® indexes, which are sometimes called streaming B-trees [1]. Fractal Tree indexes can improve the performance of databases by allowing systems with few disk drives to maintain many high-entropy indexes without slowing down data ingestion.

The rest of this paper is organized as follows. Section 2 provides an overview for how Fractal Tree indexes operate. Section 3 explains how TokuFS uses Fractal Tree indexes to represent a file system. Section 4 presents a performance study of TokuFS compared to traditional file systems, and Section 5 concludes with a discussion of future work.

2 Fractal Tree indexes

This section presents a high-level description of the Fractal Tree index, a data structure that implements a ***dictionary*** on key-value pairs. Let k be a key, and let v be a value. A dictionary supports the following operations:

Operation	Meaning
$\text{INSERT}(k,v)$	Associate value v with key k.
$v := \text{SEARCH}(k)$	Find the value associated with k.
$\text{DELETE}(k)$	Remove key k and its value.
$k' := \text{SUCC}(k)$	Find the next key after k.
$k' := \text{PRED}(k)$	Find the previous key before k.

These operations form the API for both B-trees and Fractal Tree indexes, and therefore a Fractal Tree index can be thought of as a drop-in replacement for a B-tree.

The Fractal Tree index is a ***write-optimized*** indexing scheme, in that it can index data orders of magnitude faster than a B-tree. However, unlike many other write-optimized schemes, it can perform queries on indexed data at approximately the same speed as an unfragmented B-tree. And unlike some other schemes, a Fractal Tree index does not require that all the writes occur before all the reads: a read in the middle of many writes is fast and does not slow down the writes.

The B-tree has worst-case insert and search I/O cost of $O(\log_B N)$, though it is common for all internal nodes of a B-tree to be cached in memory, and so most operations require only about one disk I/O. If a query comprises a search or successor query followed by k successor queries, which we refer to as a ***range*** query, the number of disk seeks is $O(\log_B N + k/B)$. In practice, if the keys are inserted in random order, the B-tree becomes fragmented and range queries can be an order of magnitude slower than for keys inserted in sequential order.

An alternative to a B-tree is to append all insertions to the end of a file. Append-to-file optimizes insertions at the expense of queries. Since B inserts can be bundled into one disk write, the cost per operation is $O(1/B)$ I/Os on average. However, performing a search requires reading the entire file, and thus takes $O(N/B)$ I/Os in the worst case.

An LSM tree [11] also misses the optimal read-write tradeoff curve, requiring $O((\log_B^2 N))$ I/Os for queries. (The query time can be mitigated for point queries, but not range queries, by using a Bloom filter [3]; Cassandra [8] uses this approach.)

The Fractal Tree index provides much better write performance than a B-tree and much better query performance than append-to-file or an LSM-tree. Indeed, a Fractal Tree index can be tuned to provide essentially the same query performance as an unfragmented B-tree with orders-of-magnitude improvements in insertion performance. The Fractal Tree index is based on ideas from the buffered repository tree [6] and extended by [1] to provide cache-oblivious results. Here we give a brief sketch of the Fractal Tree index.

Consider a tree with branching factor $b < B$. Associate with each link a buffer of size b/B. When an insert or delete is injected into the system, place an insert/delete command into the appropriate outgoing buffer of the root. When the buffer gets full, flush the buffer and recursively insert the messages in the buffers in the child. As buffers on a root-leaf path fill, an insertion or deletion command makes its way toward its target leaf. During queries, all messages needed to answer a query are in the buffers on the root-leaf search path.

When $b = \sqrt{B}$, the query cost is $O(\log_B N)$, or within a constant of a B-tree, and when caching is taken into account, the query time is comparable. On the other hand, the insertion time is $O((\log_B N)/\sqrt{B})$, which is orders of magnitude faster than a B-tree. This performance meets the optimal read-write tradeoff curve [5].

For TokuFS we use TokuDB [14], Tokutek's implementation of Fractal Tree indexes. TokuDB takes care of many other important considerations such as ACID, MVCC, concurrency, and compression. Fractal Tree indexes do not fragment, no matter the insertion pattern.

3 TokuFS design

TokuFS consists of two Fractal Tree indexes: A metadata index and a data index.

A metadata index is a dictionary that maps pathnames to file metadata:

$$\text{full pathname} \rightarrow \text{size}, \text{owner}, \text{timestamps}, \text{etc}\ldots$$

Files are broken up into data blocks of fixed size. In this paper, we chose the block size to be 512. This choice of block size worked well for microdata and reasonably well for large data. If we wanted to tune for larger files, we would choose a larger value for this parameter.

The blocks can be addressed by path name and block number, according to the data index, defined by

$$\text{pathname}, \text{block number} \rightarrow \text{data}[512].$$

The last block in any file is padded out to the nearest multiple of 512 length. However, the padding does not have a substantial impact on storage space, since Fractal Tree indexes use compression.

Note that path names can be long and repetitive, and thus one might expect that addressing each block by pathname would require a substantial overhead in disk space. However, the sorted path names in this experiment compress by a factor of 20, making the disk-space overhead manageable.

The lexicographic ordering of the keys in the data index guarantees that the contents of a file are logically adjacent. Since Fractal Tree indexes do not fragment, logical adjacency translates into physical adjacency. Thus, a file can be read at near disk bandwidth. Indeed, the lexicographic ordering also places files in the same directory near each other on disk.

In the simple dictionary specification described in Section 2, an index may be changed by inserts and deletes. Consider, however, the case where fewer than 512 bytes need to be changed, or where a write is unaligned with respect to the data index block bounderies. Using the operations specified, would do a $\textsc{Search}(k)$ first, would change the value associated with k to reflect the update, and then a new block would be associated with k via an insertion. Searches are slow, since they require disk seeks. Section 4 describes how to implement *upsert* operations to solve this problem with orders-of-magnitude performance improvements. The alternative would be to index every byte in the file system, which would be slow and have a large on-disk footprint.

We introduce an UPSERT message into the dictionary specification to speed up such cases. A data index UPSERT is specified by $\textsc{Upsert}(k, \text{offset}, v, \text{length})$, where the key k specifies a pathname and block number. If k is not in the dictionary, this operation inserts k with a value of v at position offset of the specified block. Unspecified bytes in the block are set to 0. Otherwise, the value associated with k is changed by replacing the bytes from offset to $\text{offset} + \text{length} - 1$ by the bytes in v. The UPSERT removes the search associated with the naive update method, and provides an order-of-magnitude-or-more boost in performance.

As noted above, the data index maps from path and block number to data block. Although this makes insertions and scans fast, especially on data in a directory tree, it makes the renaming of a directory slow, since the name of a directory is part of the key not only of every data block in every file in the directory, but for every file in the subtree rooted at that directory. Our current implementation does a naive delete from the old location followed by an insert into the new location. An alternative which we did not implement is to move the subtrees around with only $O(\log^2 N)$ work. The pathnames can then be updated with a multicast upsert message (upsert messages are explained below).

The metadata index maps pathname to a so-called ***struct stat*** of its metadata. The struct stat stores all the metadata – permission bits, mode bits, timestamps, link count, etc – that is output by a `stat` command. The stat struct is approximately 150 bytes uncompressed, but it seems to compress well in practice.

The sort order in the metadata index differs from that of the data index. Paths are sorted lexicographically by $(\text{directory depth}, \text{pathname})$. This sort order is useful for reading directories, since all of the children for a particular directory appear sequentially after the parent. With this scheme, the maximum number of files is extremely large and is not fixed at formatting time (unlike, say, ext4, which needs to know how many inodes to create at format time and thus ran out of inodes in the microfile benchmark after a second run because the default was not high enough).

A directory in TokuFS is an entry in the metadata index that maps the directory path to a struct stat with the `O_DIRECTORY` bit set. A directory exists iff there is a corresponding entry in this index. A directory is empty iff the next entry in the meta index does not share the directory path plus a slash as its prefix. This algorithm is easier than tracking whether the directory is empty in the metadata because with that scheme, we would need to update the parent directory every time one of its children was removed.

A directory has no entry in the data index and does not keep a list of its children. Because of the sort order on the metadata index, reading the metadata for the files in a directory consists of a range query, and is thus efficient.

We define a new set of upsert types that plays a critical role in the efficiency of the metadata index. For example, a file created with `O_CREAT` and no `O_EXCL` can be encoded as a message that creates an entry in the metadata if it does not exist, or does nothing if it does. Another example is when a file is written at offset O for N bytes, a message can be injected into the metadata index that updates the modification time for the file and possibly updates the highest offset of the file to be $O + N$ (i.e., its size). Or when a file is read, we can insert a message into the meta index to update the access time efficiently. Some file systems have mount options to avoid doing this altogether because updating the read time has a measurable performance hit in other implementations. These upsert messages share in common that they avoid a search into the metadata index and encode enough information to update the struct stat once the upsert message makes it to the leaf.

Symbolic links are supported by storing the target pathname as the file data for the source pathname. For simplicity, our implementation does not support hard links, though it could in the future. Hard links could be emulated using the same algorithm we use for symbolic links. We would also kept track of the link count for every file, so when a target pathname reaches a link count of zero, the file can finally be removed.

4 Performance

This section compares the performance of TokuFS to several traditional file systems. One big advantage of TokuFS is that it can handle microwrites, so we measured two kinds of microwrite benchmarks: writing many small blocks spread throughout a large file, and writing many small files in a directory hierarchy. We also measured the performance of large writes, which is where traditional file systems do well, and TokuFS is relatively slower. In the future, we hope to include optimizations for large file creation.

All of these experiments were performed on a Dual-Core AMD Opteron Processor 1222 running Ubuntu 10.04, with a 1TB Hitachi 7200rpm SATA disk drive. We chose this machine to demonstrate that the microdata problem can be addressed with cheap hardware.

Although our TokuFS FUSE module works correctly, it does not show TokuFS in its best light. We found that before and after every file creation, FUSE performs a `stat` system call, which transforms our write-only workload into a read-intensive workload. For any file system, reads require disk seeks, since only a small fraction of the files can be cached, and thus this workload is not useful in differentiating file systems. Instead, we ran our TokuFS benchmarks using a user-space library. In the future, we hope to adapt FUSE so that it does not perform so many read operations on a write-only workload. (When we run the same benchmarks using Berkeley DB [10], the performance is also bad because B-trees perform insertions slowly compared to Fractal Tree indexes. In that case, both the file system layer and the underlying data structures cause microdata performance problems.)

Figure 1 shows the time to create and scan 5 million 200-byte files in a balanced directory hierarchy in which each directory contains at most 128 entries. TokuFS is faster than the other file systems by one to two orders of magnitude for both reads and writes. Btrfs does well on writes, compared to the other traditional file systems. Btrfs gets the advantage of a log-structured file system for creating files, but suffers on reads since the resulting directory structure has lost spacial locality; surprisingly ZFS performs poorly, although it does better on a higher thread write workload. Perhaps ZFS requires a disk sync to create a file. XFS performs poorly on file creation, but relatively well on scans. The ext4 file system performs better than the other traditional file systems on the scan, probably because its hashed directory scheme preserves locality on scans.

	creation			scan
	1 thread	4 threads	8 threads	1 thread
ext4	217	365	458	10,629
XFS	196	154	143	731
Btrfs	928	671	560	204
ZFS	44	194	219	303
TokuFS	17,088	16,960	16,092	143,006

Figure 1: Microfile creation and scan performance in a directory hierarchy. The first column names the file system, the next three columns show write performance in files per second when there are varying number of threads, and the last column shows the scan rate, that is how many files per second can be traversed in a recursive directory walk.

	create files/s	scan files/s
ext4	10,426	8,995
TokuFS	89,540	231,332

Figure 2: File creation and scan rates for one million empty files in the same directory. The performance is measured in files per second.

File systems such as ext2 perform badly if you create a single directory with many files in it. Figure 2 shows that ext4 does reasonably well in this situation, in fact it does better than for the directory hierarchy. TokuFS is slightly faster in one directory than in a hierarchy, and is at least an order of magnitude faster than ext4.

Figure 3 shows the performance when performing 575-byte nonoverlapping random writes into a 10 GB file. We chose 575-bytes because it is slightly larger than one 512-byte sector and is unaligned. (For similar rea-

	write MB/s
Btrfs	0.049
ZFS	0.032
TokuFS	2.01

Figure 3: Microupdate performance. This table shows the rate (in MB/s) at which each file system can write 575-byte nonoverlapping blocks and random offsets.

sons Bent et al. [2] employed a 47,001-byte block size in a similar benchmark for parallel file systems, stating that this size was "particularly problematic.") The traditional file systems achieve only tens of kilobytes per second for this workload, whereas TokuFS achieves 2 MB/s. Although TokuFS performance in absolute terms seems small (utilizing only 2% of the bandwidth of the underlying disk drive), it is still two orders of magnitude better than the alternatives.

	time	size	file bandwidth	disk bandwidth
TokuFS	15.74s	72MB	27MB/s	1.7MB/s
XFS	5.53s	426MB	77MB/s	77MB/s
gzip -5	9.23s	52MB	46MB/s	5.6MB/s

Figure 4: Time to write a 426MB uncompressed tar file (MySQL source). Since TokuFS compresses, we measured the size on disk, the file bandwidth (the original size divided by time), and the disk bandwidth (the size on disk divided by time).

Figure 4 shows the performance when writing a single large file. We used an uncompressed MySQL source tarball. If we assume that XFS is achieving 100% of the write bandwidth at 77MB/s, then TokuFS achieves only about 35% of the underlying disk bandwidth. Part of the reason for this performance hit is that TokuFS compresses files using zlib, the same compressor used in gzip. To try to understand how much of the TokuFS performance hit is from compression, we timed gzip to compress the same file, and found that the compression time is about the same as the difference in time between TokuFS and XFS. For this workload, TokuFS runs faster on a higher core-count server. We believe that there is the potential for further optimizations in the Fractal Tree index implementation for large, incompressible files.

The biggest open problem for TokuFS is to map it into the kernel file system in such a way as to avoid performing extra read operations when creating files.

5 Conclusion

In the future, we intend to investigate the use of Fractal Tree file systems in the context of supercomputing. For small file systems, the difference between Fractal Trees and B-trees may be inconsequential, but as a file system gets larger, poor microupdate scaling becomes serious. Bent et al. [2] showed that an append-to-end file system can solve the insertion problem at the expense of read performance. Fractal Trees, which can support both efficient writes and reads, may offer a more general solution to the problem of maintaining a supercomputing file system.

Acknowledgments

Richard Prohaska provided invaluable help and advice for getting this project started. This work was supported by DOE Grant DE-FG02-08ER25853 and by NSF grants 1058565, 0937860, and 0937829.

References

[1] Bender, M. A., Farach-Colton, M., Fineman, J. T., Fogel, Y. R., Kuszmaul, B. C., and Nelson, J. Cache-oblivious streaming B-trees. In *SPAA* (2007), pp. 81–92.

[2] Bent, J., Gibson, G. A., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., and Wingate, M. PLFS: A checkpoint filesystem for parallel applications. In *SC* (2009).

[3] Bloom, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (1970), 422–426.

[4] Bonwick, J. ZFS: the last word in file systems. `https://blogs.oracle.com/video/entry/zfs_the_last_word_in`, Sept. 14 2004.

[5] Brodal, G. S., and Fagerberg, R. Lower bounds for external memory dictionaries. In *SODA* (2003), pp. 546–554.

[6] Buchsbaum, A. L., Goldwasser, M., Venkatasubramanian, S., and Westbrook, J. R. On external memory graph traversal. In *SODA* (2000), pp. 859–860.

[7] Card, R., Ts'o, T., and Tweedie, S. Design and implementation of the Second Extended Filesystem. In *Proc. of the First Dutch International Symposium on Linux* (1994), pp. 1–6.

[8] Cassandra wiki. `http://wiki.apache.org/cassandra/`, 2008.

[9] Mason, C. Btrfs design. `https://btrfs.wiki.kernel.org/articles/b/t/r/Btrfs_design.html`, Dec. 2010.

[10] Olson, M. A., Bostic, K., and Seltzer, M. Berkeley DB. In *USENIX FREENIX Track* (June 6–11 1999).

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

[12] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. In *SOSP* (Oct. 1991), pp. 1–15.

[13] Sweeny, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability in the XFS file system. In *USENIX* (San Diego, CA, Jan. 1996), pp. 1–14.

[14] Tokutek Inc. TokuDB. `http://www.tokutek.com/`, 2011.