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
What are we going to talk about?

First,

- What we built
- The problem we wanted to address
- What the results were

Then,

- How we did it
- What the system looks like
- What operations in our system look like
- Interesting open problems
TokuFS - the Fractal Tree filesystem

• Wanted to create a filesystem that could handle microdata workloads.

• We built a prototype filesystem using TokuDB.
  ‣ TokuDB is Tokutek’s commercial Fractal Tree implementation.

• Offers orders of magnitude speedup on microdata.
  ‣ Aggregates writes while indexing.
  ‣ So it can be faster than underlying filesystem.
We built TokuFS

TokuFS - the Fractal Tree filesystem

• Wanted to create a filesystem that could handle microdata workloads.

• We built a prototype filesystem using TokuDB.
  ‣ TokuDB is Tokutek’s commercial Fractal Tree implementation.

• Offers orders of magnitude speedup on microdata.
  ‣ Aggregates writes while indexing.
  ‣ So it can be faster than underlying filesystem.

What is microdata?
We built TokuFS

TokuFS - the Fractal Tree filesystem

• Wanted to create a filesystem that could handle microdata workloads.

• We built a prototype filesystem using TokuDB.
  ‣ TokuDB is Tokutek’s commercial Fractal Tree implementation.

• Offers orders of magnitude speedup on microdata.
  ‣ Aggregates writes while indexing.
  ‣ So it can be faster than underlying filesystem.
We built TokuFS

TokuFS - the Fractal Tree filesystem

- Wanted to create a filesystem that could handle microdata workloads.
- We built a prototype filesystem using TokuDB.
  - TokuDB is Tokutek’s commercial Fractal Tree implementation.
- Offers orders of magnitude speedup on microdata.
  - Aggregates writes while indexing.
  - So it can be faster than underlying filesystem.
TokuFS is fully functional

TokuFS is a prototype, but fully functional.

- Implements files, directories, metadata, etc.
- Interfaces with applications via shared library, header.

We wrote a FUSE implementation, too.

- FUSE lets you implement filesystems in userspace.
- But there’s overhead, so performance isn’t optimal.
- The best way to run is through our POSIX-like file API.
Microdata is micro data

Accessing data on disk has two components:

- Seek time -- fixed cost.
- Bandwidth time -- proportional to data size.

Microdata:

- Size of data where bandwidth time < seek time.

Microwrite: Writing microdata

- A random microwrite may pay full seek cost (expensive).
  - Updating in place, like a B-tree or hash structure.
- A random microwrite may share seek cost (cheap).
  - Logging updates to end of file.
The microdata indexing problem

Given:

• A large set of data.
• Stream of microupdates arriving in random key order.

Problem:

• Ingest the stream of updates.
• At any given time, respond to range queries on the data.
The microdata indexing problem

Given:

- A large set of data.
- Stream of microupdates arriving in random key order.

Problem:

- Ingest the stream of updates.
- At any given time, respond to range queries on the data.

Claim: Filesystems face the microdata indexing problem
Filesystem microdata example: `atime`

Whenever a file is touched, its inode is updated with the new access time.

- Updating `atime` is a microwrite.
- `ext4` has `noatime` mount option to avoid the microwrite.
- Also has `relatime` mount option to do less microwrites.

What makes updating `atime` so expensive?

- `ext4`‘s inode table updates in place, so there’s disk I/O.
- Could try and log writes so the write is cheap,
  - But now we need to be able to find an inode in the log.
  - The log exhibits no logical locality, so range queries suffer.
atime can be problematic

atime updates are by far the biggest IO performance deficiency that Linux has today. Getting rid of atime updates would give us more everyday Linux performance than all the pagecache speedups of the past 10 years, combined.

Ingo Molnar, kernel developer
atime can be problematic

atime updates are by far the biggest IO performance deficiency that Linux has today. Getting rid of atime updates would give us more everyday Linux performance than all the pagecache speedups of the past 10 years, combined.

- Many distributions use relatime

Ingo Molnar, kernel developer
**atime** is part of a bigger problem

Maybe **atime** itself isn’t very exciting...

- We can probably live with just **relatime**.

But it exposed a more fundamental problem:

- Updating metadata using an update-in-place data structure incurs too much disk I/O.
- Big scaling problem on disks capable of ~200 seeks/sec.

**Filesystems exhibit the microdata indexing problem, and it can be solved. TokuFS is a prototype that shows this.**
Big speedups on microwrites

We ran microdata-intensive benchmarks

• Compared TokuFS to ext4, XFS, Btrfs, ZFS.
• Stressed metadata and file data.
• Used commodity hardware:
  ▸ 2 core AMD, 4GB RAM
  ▸ Single 7200 RPM disk
  ▸ Simple, cheap setup. No hardware tricks.
• In all tests, orders of magnitude speed up.
Faster on small file creation

Create 2 million 200-byte files in a shallow tree

Randomized small file creation

100000
10000
1000
100
10

files/sec

filesystem

1 thread
4 threads
8 threads

ext4
xfs
btrfs
zfs
tokuFS
Create 2 million 200-byte files in a shallow tree

Randomized small file creation

- 1 thread
- 4 threads
- 8 threads

Log scale

filesystem
Faster on metadata scan

Recursively scan directory tree for metadata

- Use the same 2 million files created before.
- Start on a cold cache to measure disk I/O efficiency
Faster on big directories

Create one million empty files in a directory

- Create files with random names, then read them back.
- Tests how well a single directory scales.
Faster on microwrites in a big file

Randomly write out a file in small, unaligned pieces

![Micropupdate bandwidth graph]

- btrfs
- zfs
- tokufs

write MB/s

 filesystem

MB/sec

2.4
2.0
1.8
1.6
1.2
0.8
0.6
0.4
0.2
0
TokuFS uses Fractal Trees

Key to good performance is good data structures (Fractal Trees)

• As fast as LSM trees on writes.
• As fast as B-trees on reads.

No performance cliff when scaling out of RAM

• Performance decreases smoothly as data grows.
• Index never fragments.

Efficient on mixed workloads

• Reads just as fast even if mixed with writes.
• Writes just as fast even if mixed with reads.
TokuFS employs two indexes

**Metadata index:**
- The metadata index maps pathname to file metadata.
  - `/home/esmet` ➞ mode, file size, access times, ...
  - `/home/esmet/tokufs.c` ➞ mode, file size, access times, ...

**Data index:**
- The data index maps pathname, blocknum to bytes.
  - `/home/esmet/tokufs.c, 0` ➞ [ block of bytes ]
  - `/home/esmet/tokufs.c, 1` ➞ [ block of bytes ]
- Block size is a compile-time constant: 512.
  - good performance on small files, moderate on large files
Common queries exhibit locality

**Metadata index keys: Full path**

- All the children of a directory are contiguous in the index.
- Ordered by number of slashes first, then lexicographically.
- Reading a directory is simple and fast.

```
/home/esmet
/home/michael
/home/esmet/file.c
/home/esmet/talk
/home/michael/notes.txt
/home/esmet/talk/talk.pdf
```
Common queries exhibit locality

Metadata index keys: Full path

- All the children of a directory are contiguous in the index.
- Ordered by number of slashes first, then lexicographically.
- Reading a directory is simple and fast.

Example:
Reading /home/esmet
Common queries exhibit locality

Metadata index keys: Full path

- All the children of a directory are contiguous in the index.
- Ordered by number of slashes first, then lexicographically.
- Reading a directory is simple and fast.

Example:

Reading /home/esmet

```
/home/esmet
/home/michael
/home/esmet/file.c
/home/esmet/talk
/home/michael/notes.txt
/home/esmet/talk/talk.pdf
```

(child)

(child)
Common queries exhibit locality

**Metadata index keys: Full path**

- All the children of a directory are contiguous in the index.
- Ordered by number of slashes first, then lexicographically.
- Reading a directory is simple and fast.

```
/home/esmet/file.c
/home/esmet/talk
/home/michael/notes.txt
/home/esmet/talk/talk.pdf
```

Example:

Reading `/home/esmet`

- child
- child
- not a child
Common queries exhibit locality

Data block index keys: <full path, blocknum>

- Keys ordered such that all data blocks are contiguous.
  - So reading a file is simple and fast.
- Keys are also ordered such that all the files for a particular subtree are contiguous.
  - Helps implement directory rename. More on this later.
TokuFS compresses indexes

Reduces overhead from full path keys

- Pathnames are highly *prefix redundant*.
- They compress well in practice.

Reduces overhead from zero-valued padding

- Uninitialized bytes in a block are set to zero.
Upsert speeds up read-modify-write

Read-Modify-Write

- Update by reading old value, writing new value.
- Slow if implemented with a search + insert.

Upserts (update + insert) make read-modify-write fast

- Update with a message, targeted at a key.
- Message starts at root, but affects queries immediately.
- When root has too many messages, flush to children.
- When message reaches the value, it is applied.
Use upsert for metadata

To create a file:

• Message says: create an entry if one doesn’t exists else do nothing

To update atime:

• Message says: replace old atime with new atime

To update file size:

• Message says: set file size to max(old_offset, new)
To update $N$ bytes of a block at offset $K$:

- **Message says:**
  - Replace $N$ bytes at offset $K$.
  - If block didn’t exist, create one with zeros, then update.

To update a file:

- Send a message to each block to be modified.
- Unaligned writes aren’t too much slower.
  - If a write spans a block boundary, we can send a message to each block.
  - This is nice compared to two I/Os on an update-in-place data structures.
Upsert can be used when an operation:
  • Will have all the information it needs when applied.
  • Doesn’t need the result to return.

But some operations have hidden searches.
  • Creating a file in exclusive mode.
  • Here, the create may need to return error if file exists.
  • To know if the file exists, we need to search.

Avoid hidden searches
  • When performance is a concern, architect a system to not require a search from your write operations.
  • Or use Bloom filters.
Large file I/O has room for improvement

- Experiments show a factor of 3 to be gained compared to XFS in writing out a 400MB file sequentially.

Ideas:

- Try different compression methods
  - Trade space for speed.
  - The data is highly redundant, should work well even with a faster/lighter compressor

- Use a dynamically growing block sizes for files
  - Block sizes could grow exponentially as file size grows, to a limit, maybe 128Kb.
  - Eliminate the need to tune block size for the expected
Directory rename is slow

- Since we use full pathnames in keys, renaming a directory requires modifying the key of every child.
- For feature completeness, we implement a brute force algorithm that renames each child by deleting the old new and inserting the new.
- This scales poorly as the directory subtree grows.

Idea:

- Implement a *lifted* fractal tree, nodes have key prefix. *Clip-and-move* subtrees along a root to leaf path for the renamed directory, moving each to the new location.
Thank you!

To learn more about Fractal Tree indexing:

• “Why the best read optimization is a write optimization”
  ‣ Leif Walsh, Tokutek, talk given at Percona Live Expo 2012

• “How TokuDB Fractal Tree indexes work”
  ‣ Bradley C. Kuszmaul, Tokutek and MIT, from MySQL UC 2010

To discuss ways to leverage them for high-performance filesystems:

• John Esmet

• esmet@tokutek.com