The Algorithmics of Write Optimization

Michael A. Bender
Stony Brook University
Birds-eye view of data storage

incoming data

stored data

queries

answers
Birds-eye view of data storage

Storage systems face a trade-off between the speed of inserts/deletes/updates and the speed of queries.
Parallel computing is about high performance. To get high performance, we need fast access to our data.

The tradeoff hurts.

Storage systems face a trade-off between the speed of inserts/deletes/updates and the speed of queries.
Parallel computing is about high performance. To get high performance, we need fast access to our data.

How should we organize our stored data?

The tradeoff hurts.

Storage systems face a trade-off between the speed of inserts/deletes(updates) and the speed of queries.
Parallel computing is about high performance. To get high performance, we need fast access to our data.

How should we organize our stored data?

This is a data-structural question.

The tradeoff hurts.

Storage systems face a trade-off between the speed of inserts/deletes/updates and the speed of queries.
How should we organize our stored data?
How should we organize our stored data?

Like a librarian?
How should we organize our stored data?

Like a librarian?

Fast to find stuff.
Requires work to maintain.
How should we organize our stored data?

Like a librarian?

Fast to find stuff.
Requires work to maintain.

Like a teenager?

How should we organize our stored data?

Like a librarian?
Fast to find stuff.
Requires work to maintain.

Like a teenager?
Fast to add stuff.
Slow to find stuff.
How should we organize our stored data?

Like a librarian?

Fast to find stuff.
Requires work to maintain.

“Indexing”

Like a teenager?

Fast to add stuff.
Slow to find stuff.

“Logging”
How should we organize our stored data?

indexing
Sort in logical order.

Find a key: fast.
Insert a key: slower.

logging
Sort in arrival order.

Find a key: slow.
Insert a key: fast.
How should we organize our stored data?

**indexing**
Sort in logical order.

- Find a key: fast.
- Insert a key: slower.

| (1,1) | (2,0) | (4,3) | (5,2) | (8,1) |

**logging**
Sort in arrival order.

- Find a key: slow.
- Insert a key: fast.

| (5,2) | (8,1) | (2,0) | (1,1) | (4,3) |
How should we organize our stored data?

**indexing**
Sort in logical order.

Find a key: fast.
Insert a key: slower.

**logging**
Sort in arrival order.

Find a key: slow.
Insert a key: fast.
How should we organize our stored data?

**indexing**
Sort in logical order.

Find a key: fast.
Insert a key: slower.

| (1,1) |
| (2,0) |
| (4,3) |
| (5,2) |
| (8,1) |

**logging**
Sort in arrival order.

Find a key: slow.
Insert a key: fast.

| (5,2) |
| (8,1) |
| (2,0) |
| (1,1) |
| (4,3) |
How should we organize our stored data?

**indexing**

Sort in logical order.

- Find a key: fast.
- Insert a key: slower.

**logging**

Sort in arrival order.

- Find a key: slow.
- Insert a key: fast.

or range of keys
Indexing vs logging spans domains & decades

The tradeoff comes under many different names and guises:

- SQL databases
- NoSQL key-value stores
- File systems
Indexing vs logging spans domains & decades

The tradeoff comes under many different names and guises:

- clustered indexes ➔ unclustered indexes

SQL databases
- InnoDB
- MyISAM

NoSQL key-value stores
- LevelDB
- WiscKey

File systems
Indexing vs logging spans domains & decades

The tradeoff comes under many different names and guises:

- clustered indexes ↔ unclustered indexes
- in-place file systems ↔ log-structured file systems

InnoDB → SQL databases → MyISAM

LevelDB → NoSQL key-value stores → WiscKey

ext4 → file systems → f2fs LFS
Indexing vs logging spans domains & decades

The tradeoff comes under many different names and guises:

- clustered indexes
- unclustered indexes
- in-place file systems
- log-structured file systems
- etc!

file systems

ext4

file systems

file systems

levelDB

NoSQL key-value stores

WiscKey

NoSQL key-value stores

SQL databases

MyISAM

SQL databases

InnoDB

SQL databases

indexing/fast queries

logging/fast data ingestion
Indexing vs logging: universal data structures question

DBs, kv-stores, and file systems are different beasts. But they grapple with the similar data-structures problems.
DBs, kv-stores, and file systems are different beasts. But they grapple with the similar data-structures problems.

Similar problems $\Rightarrow$ similar solutions
DBs, kv-stores, and file systems are different beasts. But they grapple with the similar data-structures problems. Similar problems \(\Rightarrow\) similar solutions

SQL database
- SQL processing
- query optimization

noSQL database
- key-value operations

file system
- file and directory operations

Indexing vs logging: universal data structures question

Disk/SSD
Some “write-optimized” data structures can mitigate or overcome the indexing-logging trade-off.

At our DB company Tokutek,* we sold open-source write-optimized databases. Since it was sold, we’ve built an open-source file system.

---

**TokuDB**
- SQL database
  - SQL processing
  - query optimization

**TokuMX**
- noSQL database
  - noSQL processing

**BetrFS**
- file and directory operations

---

*acquired by Percona*
Some “write-optimized” data structures can mitigate or overcome the indexing-logging trade-off.

At our DB company Tokutek,* we sold open-source write-optimized databases. Since it was sold, we’ve built an open-source file system.

I’ll talk about my experiences using the same data structure to help all three systems.

*acquired by Percona
The performance landscape is fundamentally changing.

- New data structures
- New hardware
The performance landscape is fundamentally changing.

- New data structures
- New hardware

This has created tons of new research opportunities.

- For algorithmists/theorists
- For systems builders
The performance landscape is fundamentally changing.

- New data structures
- New hardware

This has created tons of new research opportunities.

- For algorithmists/theorists
- For systems builders

There’s still lots to do.
An algorithmic view of the insert-query tradeoff
Performance characteristics of storage
Performance characteristics of storage

Sequential access is fast.
Performance characteristics of storage

Sequential access is fast.
Random access is slower.
How computation works:

- Data is transferred in blocks between RAM and disk.
- The # of block transfers dominates the running time.

Goal: Minimize # of I/Os

- Performance bounds are parameterized by block size $B$, memory size $M$, data size $N$.

Disk-Access Machine (DAM) model [Aggarwal+Vitter '88]
I/Os are slow

RAM: ~60 nanoseconds per access
Disks: ~6 milliseconds per access.

Analogy:

- RAM $\propto$ escape velocity from earth (40,250 kph)
- disk $\propto$ walking speed of the giant tortoise (0.4 kph)
How realistic is the DAM model?
How realistic is the DAM model?

"All models are wrong, but some are useful"

What is reality anyway?

[George Box 1978]
How realistic is the DAM model?

"All models are wrong, but some are useful"

• Unrealistic in various ways.

What is reality anyway?

[George Box 1978]
How realistic is the DAM model?

"All models are wrong, but some are useful”

• Unrealistic in various ways.
• Great for reasoning about I/O and for high-level design.

What is reality anyway?
"All models are wrong, but some are useful"

- Unrealistic in various ways.
- Great for reasoning about I/O and for high-level design.
- You can optimize the model to hone constants.
How computation works:
• Data is transferred in blocks between RAM and disk.
• The # of block transfers dominates the running time.

Goal: Minimize # of I/Os
• Performance bounds are parameterized by block size $B$, memory size $M$, data size $N$. 

Disk-Access Machine (DAM) model [Aggarwal+Vitter ’88]
I/O cost for logging.
What’s the I/O cost for logging?

I/O cost for logging.
I/O cost for logging.

- **query**: scan all blocks $\Rightarrow O(N/B)$
What’s the I/O cost for logging?

I/O cost for logging.

- **query**: scan all blocks \(\Rightarrow O(N/B)\)
- **insert**: append to end of log \(\Rightarrow O(1/B)\)
Q: What’s the I/O cost for indexing?
A: It depends on the indexing data structure.
What’s the I/O cost for indexing?

The classic indexing structure is the B-tree.

$O(\log_B N)$
What’s the I/O cost for indexing?

The classic indexing structure is the B-tree.

Queries: $O(\log_B N)$
Inserts: $O(\log_B N)$
The classic indexing structure is the B-tree.

Queries: $O(\log_B N)$
Inserts: $O(\log_B N)$
What’s the I/O cost for indexing?

The classic indexing structure is the B-tree.

Queries: $O(\log_B N)$

Inserts: $O(\log_B N)$

not optimal!!
Beating B-tree Bounds

Queries: $O(\log_B N)$
Inserts: $O(\log_B N)$

optimal

not optimal!!
Beating B-tree Bounds

Queries: $O(\log_B N)$
Inserts: $O(\log_B N)$

Goal:
Inserts that run faster than a B-tree.
Queries that don’t run slower.

optimal

not optimal!!
Start with a regular B-tree
Reduce the fanout.

- Now the nodes are mostly empty
Put $B^{1/2}$-sized buffers in each internal node.
Inserts + deletes:

- Send insert/delete messages down from the root and store them in buffers.
- When a buffer fills up, flush.
Inserts and deletes in a $B^\varepsilon$ tree

**Inserts + deletes:**

- Send insert/delete messages down from the root and store them in buffers.
- When a buffer fills up, flush.
Inserts + deletes:

- Send insert/delete messages down from the root and store them in buffers.
- When a buffer fills up, flush.
Inserts + deletes:

- Send insert/delete messages down from the root and store them in buffers.
- When a buffer fills up, flush.
Inserts + deletes:

- Send insert/delete messages down from the root and store them in buffers.
- When a buffer fills up, flush.
Inserts + deletes:

- Send insert/delete messages down from the root and store them in buffers.
- When a buffer fills up, flush.
Inserts and deletes in a $B^\varepsilon$ tree

**Inserts + deletes:**

- Send insert/delete messages down from the root and store them in buffers.
- When a buffer fills up, flush.
Inserts + deletes:

- Send insert/delete messages down from the root and store them in buffers.
- When a buffer fills up, flush.
Inserts + deletes:

- Send insert/delete messages down from the root and store them in buffers.
- When a buffer fills up, flush.
Inserts + deletes:

- Send insert/delete messages down from the root and store them in buffers.
- When a buffer fills up, flush.

Deletes are tombstone messages.
Difficulty of key searches
Difficulty of key searches

This is a keynote talk... so note the keys.
Searches cost $O(\log_{B} N)$

- Look in all buffers on root-to-leaf path.
Searches cost $O(\log_B N)$

- Look in all buffers on root-to-leaf path.
Insertions analysis in $B^\epsilon$ tree

**Inserts cost $O((\log_B N)/\sqrt{B})$ per insert/delete.**

- Each flush cost 1 I/O and flushes $\sqrt{B}$ elements.
- Flush cost per element is $1/\sqrt{B}$.
- There are $O(\log_B N)$ levels in a tree.
<table>
<thead>
<tr>
<th>Fanout</th>
<th>Insert</th>
<th>Point Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>$O(\log_B N)$</td>
<td>$O(\log_B N)$</td>
</tr>
<tr>
<td>$B^{1/2}$</td>
<td>$O\left(\frac{\log_B N}{\sqrt{B}}\right)$</td>
<td>$O(\log_B N)$</td>
</tr>
</tbody>
</table>
Write-optimization

[Brodal, Fagerberg 03]

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>point query</th>
</tr>
</thead>
<tbody>
<tr>
<td>fanout $B$</td>
<td>$O(\log_B N)$</td>
<td>$O(\log_B N)$</td>
</tr>
<tr>
<td>fanout $B^{1/2}$</td>
<td>$O\left(\frac{\log_B N}{\sqrt{B}}\right)$</td>
<td>$O(\log_B N)$</td>
</tr>
</tbody>
</table>

Example:

- Record size: 128 bytes
- Node size: 128 KB
- $B$: 1024 records
- Speedup: $\approx \frac{\sqrt{1024}}{2} = 16$
Write-optimization

[Brodal, Fagerberg 03]

<table>
<thead>
<tr>
<th>fanout $B$</th>
<th>fanout $B^{1/2}$</th>
<th>insert</th>
<th>point query</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$O \left( \log_{B} N \right)$</td>
<td>$O \left( \log_{B} N \right)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O \left( \frac{\log_{B} N}{\sqrt{B}} \right)$</td>
<td>$O \left( \log_{B} N \right)$</td>
</tr>
</tbody>
</table>

Example:

Record size: 128 bytes
Node size: 128 KB
B: 1024 records
Speedup: $\approx \frac{\sqrt{1024}}{2} = 16$

Inserts run 1-2 orders of magnitude faster than in a B-tree.
Optimal insertion-search tradeoff curve

[Brodal, Fagerberg 03]
Change the fanout from $B^{1/2}$ to $B^\varepsilon$. 

[Diagram showing the change in fanout from $B^{1/2}$ to $B^\varepsilon$.]
<table>
<thead>
<tr>
<th>Optimal tradeoff (function of $\varepsilon=0...1$)</th>
<th>(O\left(\frac{\log_{1+B^{\varepsilon}} N}{B^{1-\varepsilon}}\right))</th>
<th>(O\left(\log_{1+B^{\varepsilon}} N\right))</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>B-tree</strong> ((\varepsilon=1))</td>
<td>(O\left(\log_B N\right))</td>
<td>(O\left(\log_B N\right))</td>
</tr>
<tr>
<td>(\varepsilon=1/2)</td>
<td>(O\left(\frac{\log B N}{\sqrt{B}}\right))</td>
<td>(O\left(\log_B N\right))</td>
</tr>
<tr>
<td>(\varepsilon=0)</td>
<td>(O\left(\frac{\log N}{B}\right))</td>
<td>(O\left(\log N\right))</td>
</tr>
</tbody>
</table>

10x-100x faster inserts

Change the fanout from \(B^{1/2}\) to \(B^\varepsilon\).
Illustration of Optimal Tradeoff [Brodal, Fagerberg 03]

Optimal tradeoff
(function of $\varepsilon=0\ldots1$)

$$O \left( \frac{\log_{1+B^\varepsilon} N}{B^{1-\varepsilon}} \right) O \left( \log_{1+B^\varepsilon} N \right)$$

I/O per Point Query

Fast

Slow

B=10^5, N=10^{12}
Illustration of Optimal Tradeoff [Brodal, Fagerberg 03]

**Optimal tradeoff**

\[ O \left( \frac{\log_{1+B^\varepsilon} N}{B^{1-\varepsilon}} \right) \]

(\text{function of } \varepsilon = 0 \ldots 1)

Insertions improve by large factors with almost no loss of point-query performance

**Target of opportunity**

- Fast
- I/O per Point Query
- Slow

- Insert
- Fast
- I/O per Insert
- Slow

\[ B=10^5, \ N=10^{12} \]
Illustration of Optimal Tradeoff [Brodal, Fagerberg 03]

Don't Thrash: How to Cache Your Hash in Flash

logging

Optimal tradeoff (function of $\varepsilon=0\ldots1$)  

$O\left( \log_{1+B^\varepsilon} N \right)$  

$O\left( \log_{1+B^\varepsilon} N \right)$  

Insertions improve by large factors with almost no loss of point-query performance

Target of opportunity

B-tree

I/O per Point Query

Fast

Slow

Fast  I/O per Insert  Slow

B=10^5, N=10^{12}
Write performance on large data

iiBench Benchmark (throughput)
TokuMX vs. MongoDB
(higher is better)

Toot your own horn
Other WODS
The most famous write-optimized data structure is the log structured merge tree  

[O’Neil, Cheng, Gawlick, O’Neil 96]

Data structures: [O’Neil, Cheng, Gawlick, O’Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach, Fineman, Fogel, Kuszmaul, Nelson’07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10], [Spillane, Shetty, Zadok, Archak, Dixit 11]. 

Systems: BetrFS, BigTable, Cassandra, H-Base, LevelDB, PebblesDB, RocksDB, TokuDB, TableFS, TokuMX.
The most famous write-optimized data structure is the log structured merge tree \cite{oneil1996}

There are many others (B$^\epsilon$-tree, buffered repository tree, COLA, x-dict, write-optimized skip list).

**Data structures:** \cite{oneil1996,buchsbaum2000,argel2003,graeffe2003,bender2007,brodal2010,spillane2011}  

**Systems:** BetrFS, BigTable, Cassandra, H-Base, LevelDB, PebblesDB, RocksDB, TokuDB, TableFS, TokuMX.
Other write-optimized data structures

The most famous write-optimized data structure is the log structured merge tree [O'Neil, Cheng, Gawlick, O'Neil 96]

There are many others (B$^\varepsilon$-tree, buffered repository tree, COLA, x-dict, write-optimized skip list).

Write optimization is having a large impact on systems.

Data structures: [O'Neil, Cheng, Gawlick, O'Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach, Fineman, Fogel, Kuszmaul, Nelson'07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10], [Spillane, Shetty, Zadok, Archak, Dixit 11].

Systems: BetrFS, BigTable, Cassandra, H-Base, LevelDB, PebblesDB, RocksDB, TokuDB, TableFS, TokuMX.
Overview of Talk
Overview of Talk
Overview of Talk

A write-optimized dictionary (WOD) data structure....
A write-optimized dictionary (WOD) data structure...

... searches like a B-tree...
A write-optimized dictionary (WOD) data structure...

... searches like a B-tree...

... but inserts asymptotically faster.
A write-optimized dictionary (WOD) data structure....

... searches like a B-tree...

... but inserts asymptotically faster.

WODs beat the tradeoff from the beginning of the talk.
Write-optimization in databases
ACID-compliant database

- Application
  - SQL processing
  - Query optimization

- Database index
  (traditionally a B-tree)

- File system

- Disk/SSD
ACID-compliant database built on a B$^\varepsilon$-tree

- SQL processing
- Query optimization

SQL database

Database index
(traditionally a B-tree)

File system

Disk/SSD
ACID-compliant database built on a $B^\varepsilon$-tree

Replace the B-tree with a $B^\varepsilon$-tree.

(The $B^\varepsilon$-tree must be full-featured and persistent to power a database.)
ACID-compliant database built on a $\mathbb{B}^\varepsilon$-tree

We built a write-optimized SQL databases at our DB company Tokutek.

SQL database

- application
  - SQL processing
  - query optimization

- database index
  (traditionally a B-tree)

- file system

Replacing the B-tree with a $\mathbb{B}^\varepsilon$-tree.

(The $\mathbb{B}^\varepsilon$-tree must be full-featured and persistent to power a databases.)
ACID-compliant database built on a B^{ε}-tree

We built a write-optimized SQL databases at our DB company Tokutek.

Replace the B-tree with a B^{ε}-tree.

(The B^{ε}-tree must be full-featured and persistent to power a databases.)

Result: indexing runs 10x-100x faster than traditional structures.
Everything else

- Variable-sized rows
- Concurrency-control mechanisms
- Multithreading
- Transactions, logging, ACID-compliant crash recovery
- Optimizations for the special cases of sequential inserts and bulk loads
- Compression
- Backup
Everything else

- Variable-sized rows
- Concurrency-control mechanisms
- Multithreading
- Transactions, logging, ACID-compliant crash recovery
- Optimizations for the special cases of sequential inserts and bulk loads
- Compression
- Backup

Yeah, but what about transactions?
Transactions, logging, ACID-compliant crash recovery

Ingredients

- a regular write-optimized structure
- a log
- periodic checkpoints of the WOD

Yeah, but what about transactions?
Transactions, logging, ACID-compliant crash recovery

Ingredients

• a regular write-optimized structure

• a log: sequential access

• periodic checkpoints of the WOD

Yeah, but what about transactions?
Transactions, logging, ACID-compliant crash recovery

Ingredients

• a regular write-optimized structure

• a log

• periodic checkpoints of the WOD

Yeah, but what about transactions?
Transactions, logging, ACID-compliant crash recovery

Components

• a regular write-optimized structure
• a log
• periodic checkpoints of the WOD

Result: even with crash recovery and transactional semantics, indexing is 10x-100x faster than traditional structures.

Yeah, but what about transactions?
WODs change the performance landscape.

WODs help in ways that, at first glance, have little to do with fast insertions.
Remember the logging/indexing dilemma?

TokuDB

a company
Remember the logging/indexing dilemma?

With TokuDB you can index data 10x-100x faster.
Remember the logging/indexing dilemma?

With TokuDB you can index data 10x-100x faster.

We don’t have an insertion bottleneck. We have a query bottleneck.
Remember the logging/indexing dilemma?

With TokuDB you can index data 10x-100x faster.

In this case, put a rich set of database indexes in your DB schema.

We don’t have an insertion bottleneck. We have a query bottleneck.
Remember the logging/indexing dilemma?

With TokuDB you can index data 10x-100x faster.

We don’t have an insertion bottleneck. We have a query bottleneck.

In this case, put a rich set of database indexes in your DB schema.

We can’t.
Remember the logging/indexing dilemma?

With TokuDB you can index data 10x-100x faster.

In this case, put a rich set of database indexes in your DB schema.

Why not?

We don’t have an insertion bottleneck. We have a query bottleneck.

We can’t.
Remember the logging/indexing dilemma?

With TokuDB you can index data 10x-100x faster.

In this case, put a rich set of database indexes in your DB schema.

Why not?

We don’t have an insertion bottleneck. We have a query bottleneck.

We can’t.

We cannot get our insertion rate to keep up.
Remember the logging/indexing dilemma?

With TokuDB you can index data 10x-100x faster.

In this case, put a rich set of database indexes in your DB schema.

Why not?

We don’t have an insertion bottleneck. We have a query bottleneck.

We can’t.

We cannot get our insertion rate to keep up.

Moral: insertion problems often masquerade as query problems.
Remember the logging/indexing dilemma?

With TokuDB you can index data 10x-100x faster.

In this case, put a rich set of database indexes in your DB schema.

Why not?

We can insert data 10x-100x faster.

We don’t have an insertion bottleneck. We have a query bottleneck.

We can’t.

We cannot get our insertion rate to keep up.

Moral: insertion problems often masquerade as query problems.
Remember the logging/indexing dilemma?

With TokuDB you can index data 10x-100x faster.

In this case, put a rich set of database indexes in your DB schema.

Why not?

We can insert data 10x-100x faster.

We don’t have an insertion bottleneck. We have a query bottleneck.

We can’t.

We cannot get our insertion rate to keep up.

Our inserts are fine. Our queries run too slowly.

Moral: insertion problems often masquerade as query problems.
Remember the logging/indexing dilemma?

With TokuDB you can index data 10x-100x faster.

In this case, put a rich set of database indexes in your DB schema.

Why not?

We can insert data 10x-100x faster.

Just use a richer set of indexes.

We don’t have an insertion bottleneck. We have a query bottleneck.

We can’t.

We cannot get our insertion rate to keep up.

Our inserts are fine. Our queries run too slowly.

Moral: insertion problems often masquerade as query problems.
Remember the logging/indexing dilemma?

"With TokuDB you can index data 10x-100x faster."

"In this case, put a rich set of database indexes in your DB schema."

"Why not?"

"We can insert data 10x-100x faster."

"Just use a richer set of indexes."

"We don’t have an insertion bottleneck. We have a query bottleneck."

"We can’t."

"We cannot get our insertion rate to keep up."

"Our inserts are fine. Our queries run too slowly."

"We wish we could."

Moral: insertion problems often masquerade as query problems.
The right read optimization is write optimization.

The right index makes queries run fast. WODS can maintain them.

Fast writing is a currency we use to make queries faster.
WODs force you to reexamine your system design...
What the world looks like

Insert/point query asymmetry

- Inserts can be fast: 50-100K random writes/sec on a disk.
- Point queries are provably slow: <200 random reads/sec on a disk.

Systems are often designed assuming reads and writes have about the same cost.

In fact, writing is easier than reading.
Systems often assume search cost = insert cost

Ancillary search—a search with each insert.

- Insert with uniqueness check—is the key is already already present?
- Delete with acknowledgement—was a key actually actually deleted?
Ancillary search—a search with each insert.

- Insert with uniqueness check—is the key is already present?
- Delete with acknowledgement—was a key actually deleted?

These ancillary searches throttle insertions down to the performance of B-trees.
Ancillary search—a search with each insert.

- Insert with uniqueness check—is the key is already present?
- Delete with acknowledgement—was a key actually deleted?

These ancillary searches throttle insertions down to the performance of B-trees.

In a B-tree, the leaf is already fetched, so reading it has no extra cost. In a WOD, it’s expensive.
How can we get rid of ancillary searches?

Write-optimized systems must get rid of or mitigate ancillary searches whenever possible.

It’s remarkable that uniqueness checking is hard, but ACID compliance is asymptotically easy.

We now live with a different model for what’s expensive and what’s cheap.
Using WODs in File Systems

(BetrFS, TokuFS, TableFS are examples of write-optimized file systems. I’ll talk about BetrFS)
Using WODs in File Systems

(BetrFS, TokuFS, TableFS are examples of write-optimized file systems. I’ll talk about BetrFS)

The empirical tradeoff between writing and querying appears in file systems.
Logging versus indexing tradeoff in file systems
Logging versus indexing tradeoff in file systems

How should we organize the files on disk?

directory tree
How should we organize the files on disk?

- logical order $\implies$ sequential scans are fast
  - grep -r "bar" .
  - ls -R .
How should we organize the files on disk?

- **logical order** $\Rightarrow$ sequential scans are fast
  
  - grep -r "bar" .
  - ls -R .

- **update order** $\Rightarrow$ small writes are fast
How should we organize the files on disk?

**logical order** ⇒ sequential scans are fast

- grep -r "bar" .
- ls -R .

**update order** ⇒ small writes are fast

- Scans are slow.
- Updates are slow.
How should we organize the files on disk?

**logical order ⇒ sequential scans are fast**

- `grep -r "bar" .`
- `ls -R .`

**update order ⇒ small writes are fast**

**The empirical tradeoff between writing and querying appears in file systems.**

But we no longer have a B-tree to replace with a $B^\varepsilon$-tree.

Updates are slow.

Scans are slow.
Maintain two WODs, each indexed on the path names.

File-system operations → inserts and range queries.
Maintain two WODs, each indexed on the path names.

File-system operations → inserts and range queries.
Maintain two WODs, each indexed on the path names.

File-system operations $\rightarrow$ inserts and range queries.
Maintain two WODs, each indexed on the path names.

<table>
<thead>
<tr>
<th>&lt;path, file metadata&gt;</th>
<th>&lt;path, file data&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>/home/bender/doc</td>
<td>/home/bender/doc</td>
</tr>
<tr>
<td>/home/bender/doc/latex/</td>
<td>/home/bender/doc/latex/</td>
</tr>
<tr>
<td>/home/bender/doc/latex/a.tex</td>
<td>/home/bender/doc/latex/a.tex</td>
</tr>
<tr>
<td>/home/bender/doc/latex/b.tex</td>
<td>/home/bender/doc/latex/b.tex</td>
</tr>
<tr>
<td>/home/bender/doc/foo.c</td>
<td>/home/bender/doc/foo.c</td>
</tr>
<tr>
<td>/home/bender/local</td>
<td>/home/bender/local</td>
</tr>
</tbody>
</table>

File-system operations → inserts and range queries.

- `ls -R`
- `grep -r`
Maintain two WODs, each indexed on the path names.

File-system operations \(\rightarrow\) inserts and range queries.
Microwrite and Scan Performance on BetrFs

1000 Random 4-byte writes

*lower is better

GNU Find

[BetrFS: Jannen, Yuan, Zhan, Akshintala, Esmet, Jiao, Mittal, Pandey, Reddy, Walsh, Bender, Farach-Colton, Johnson, Kuszmaul, Porter, FAST 15]
Microwrite and Scan Performance on BetrFs

1000 Random 4-byte writes

*lower is better

[BetrFS: Jannen, Yuan, Zhan, Akshintala, Esmet, Jiao, Mittal, Pandey, Reddy, Walsh, Bender, Farach-Colton, Johnson, Kuszmaul, Porter, FAST 15]
Maintain two WODs, each indexed on the path names.

```
<path, file metadata>

/home/bender/doc
/home/bender/doc/latex/
/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex
/home/bender/doc/foo.c
/home/bender/local

<path, file data>

/home/bender/doc
/home/bender/doc/latex/
/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex
/home/bender/doc/foo.c
/home/bender/local
```

Some file-system operations don’t seem to map cheaply.
Maintain two WODs, each indexed on the path names.

Some file-system operations don’t seem to map cheaply.
Maintain two WODs, each indexed on the path names.

Some file-system operations don’t seem to map cheaply.

These keys change their names. They move to a different place in the order.
Moral: how can we make write-optimized data structures that support the richer set of operations needed by the applications?

We need more than just insert and delete.
Other WODs advantages
Bε-trees can use bigger nodes than B-trees

- Better compression
- Less fragmentation.

Bε-trees file systems do not age the way B-tree based file systems do.

[Conway, Bakshi, Jiao, Zhan, Bender, Jannen, Johnson, Kuszmaul, Porter, Yuan, Farach-Colton 17]
Other WODs advantages

\textbf{B}^\varepsilon\textbf{-trees} can use bigger nodes than \textbf{B}-trees

\begin{itemize}
  \item Better compression
  \item Less fragmentation.
\end{itemize}

\textbf{B}^\varepsilon\textbf{-trees} file systems do not age the way \textbf{B}-tree based file systems do.

[Conway, Bakshi, Jiao, Zhan, Bender, Jannen, Johnson, Kuszmaul, Porter, Yuan, Farach-Colton 17]

We cannot see this in the DAM model. We need a more refined model.
DAM is realistic enough to make powerful predictions.

Some things it doesn’t predict, such as aging.
DAM is realistic enough to make powerful predictions.

Some things it doesn’t predict, such as aging.

Technology is changing.

- I/O speeds are accelerating faster than CPU.
- Storage technology supports lots of I/Os in parallel.
DAM is realistic enough to make powerful predictions.

Some things it doesn’t predict, such as aging.

Technology is changing.

- I/O speeds are accelerating faster than CPU.
- Storage technology supports lots of I/Os in parallel.

Need multithreading and lots of parallel I/Os to drive the device to its capacity.

- Data structures for older storage don’t work so well now.
The traditional search-insert tradeoff can be improved.
The traditional search-insert tradeoff can be improved.

We should rethink applications and the storage stack given new write-optimized data structures.
WODS are accessible. They are teachable in standard curricula.

The traditional search-insert tradeoff can be improved.

We should rethink applications and the storage stack given new write-optimized data structures.
We should revisit the performance model.

To get performance now we need parallelism everywhere.

WODS are accessible. They are teachable in standard curricula.

The traditional search-insert tradeoff can be improved.

We should rethink applications and the storage stack given new write-optimized data structures.

We should revisit the performance model. To get performance now we need parallelism everywhere.