Data Structures and Algorithms for Big Databases

Michael A. Bender
Stony Brook & Tokutek

Bradley C. Kuszmaul
MIT & Tokutek
Big data problem

Important and universal problem.

Data ingestion

Queries + answers

Oy vey

42

365

???

Data indexing

Query processor
For on-disk data, one sees funny tradeoffs in the speeds of data ingestion, query speed, and freshness of data.
Funny tradeoff in ingestion, querying, freshness

- “I’m trying to create indexes on a table with 308 million rows. It took ~20 minutes to load the table but 10 days to build indexes on it.”
  - MySQL bug #9544
- “Select queries were slow until I added an index onto the timestamp field... Adding the index really helped our reporting, BUT now the inserts are taking forever.”
  - Comment on mysqlperformanceblog.com
- “They indexed their tables, and indexed them well, And lo, did the queries run quick! But that wasn’t the last of their troubles, to tell– Their insertions, like molasses, ran thick.”
  - Not from Alice in Wonderland by Lewis Carroll
Funny tradeoff in ingestion, querying, freshness

- “I'm trying to create indexes on a table with 308 million rows. It took ~20 minutes to load the table but 10 days to build indexes on it.”
  - MySQL bug #9544
- “Select queries were slow until I added an index onto the timestamp field...Adding the index really helped our reporting, BUT now the inserts are taking forever.”
  - Comment on mysqlperformanceblog.com
Funny tradeoff in ingestion, querying, freshness

- “I'm trying to create indexes on a table with 308 million rows. It took ~20 minutes to load the table but 10 days to build indexes on it.”
  - MySQL bug #9544

- “Select queries were slow until I added an index onto the timestamp field... Adding the index really helped our reporting, BUT now the inserts are taking forever.”
  - Comment on mysqlperformanceblog.com

- “They indexed their tables, and indexed them well, And lo, did the queries run quick! But that wasn’t the last of their troubles, to tell– Their insertions, like treacle, ran thick.”
  - Not from Alice in Wonderland by Lewis Carroll
This tutorial

- Better data structures significantly mitigate the insert/query/freshness tradeoff.
- These structures scale to much larger sizes while efficiently using the memory-hierarchy.
What we mean by Big Data

We don’t define Big Data in terms of TB, PB, EB.

By Big Data, we mean

- The data is too big to fit in main memory.
- We need data structures on the data.
- Words like “index” or “metadata” suggest that there are underlying data structures.
- These data structures are also too big to fit in main memory.
In this tutorial we study the underlying data structures for managing big data.
But enough about databases...

... more about us.
A few years ago we started working together on I/O-efficient and cache-oblivious data structures.

Along the way, we started Tokutek to commercialize our research.
Tokutek sells TokuDB, an ACID compliant, closed-source storage engine for MySQL.
Tokutek sells TokuDB, an ACID compliant, closed-source storage engine for MySQL.
Tokutek sells TokuDB, an ACID compliant, closed-source storage engine for MySQL.

TokuDB also has a Berkeley DB API and can be used independent of MySQL.
Tokutek sells TokuDB, an ACID compliant, closed-source storage engine for MySQL.
Our Mindset

• This tutorial is self contained.
• We want to teach.
• If something we say isn’t clear to you, please ask questions or ask us to clarify/repeat something.
• You should be comfortable using math.
• You should want to listen to data structures for an afternoon.
Topics and Outline for this Tutorial

I/O model and cache-oblivious analysis.

Write-optimized data structures.

How write-optimized data structures can help file systems.

Block-replacement algorithms.

Indexing strategies.

Log-structured merge trees.

Bloom filters.
Data Structures and Algorithms for Big Data

Module 1: I/O Model and Cache-Oblivious Analysis

Michael A. Bender
Stony Brook & Tokutek

Bradley C. Kuszmaul
MIT & Tokutek
If we want to understand the performance of data structures within databases we need algorithmic models for modeling I/Os.

There’s a long history of models for understanding the memory hierarchy. Many are beautiful. Most have not found practical use.

Two approaches are very powerful.

That’s what we’ll present here so we have a foundation for the rest of the tutorial.
How computation works:
- Data is transferred in blocks between RAM and disk.
- The # of block transfers dominates the running time.

Goal: Minimize # of block transfers
- Performance bounds are parameterized by block size $B$, memory size $M$, data size $N$.  

[Aggarwal+Vitter ’88]
Example: Scanning an Array

Question: How many I/Os to scan an array of length $N$?
Question: How many I/Os to scan an array of length $N$?

Answer: $O(N/B)$ I/Os.
Question: How many I/Os to scan an array of length $N$?

Answer: $O(N/B)$ I/Os.
Example: Searching in a B-tree

Question: How many I/Os for a point query or insert into a B-tree with \( N \) elements?

\[ O(\log_B N) \]
Example: Searching in a B-tree

Question: How many I/Os for a point query or insert into a B-tree with $N$ elements?

Answer: $O(\log_B N)$
Example: Searching in an Array

Question: How many I/Os to perform a binary search into an array of size $N$?

$N/B$ blocks
Example: Searching in an Array

Question: How many I/Os to perform a binary search into an array of size $N$?

Answer: $O\left(\log_2 \frac{N}{B}\right) \approx O\left(\log_2 N\right)$
Example: Searching in an Array Versus B-tree

Moral: B-tree searching is a factor of $O(\log_2 B)$ faster than binary searching.

\[ O(\log_2 N) = O\left(\frac{\log_2 N}{\log_2 B}\right) \]
Imagine the following sorting problem:

- 1000 MB data
- 10 MB RAM
- 1 MB Disk Blocks

Here’s a sorting algorithm

- Read in 10MB at a time, sort it, and write it out, producing 100 10MB “runs”.
- Merge 10 10MB runs together to make a 100MB run. Repeat 10x.
- Merge 10 100MB runs together to make a 1000MB run.
I/O-Efficient Sorting in a Picture

1000 MB unsorted data

- sort 10MB runs

10 MB sorted runs:

- merge 10MB runs into 100MB runs

100MB sorted runs

- merge 100MB runs into 1000MB runs

1000 MB sorted data
Why merge in two steps?  We can only hold 10 blocks in main memory.

- 1000 MB data; 10 MB RAM; 1 MB Disk Blocks
Example

- Produce 10MB runs.
- Merge 10 10MB runs for 100MB.
- Merge 10 100MB runs for 1000MB.

becomes in general:

- Produce runs the size of main memory (size=M).
- Construct a merge tree with fanout M/B, with runs at the leaves.
- Repeatedly: pick a node that hasn’t been merged. Merge the M/B children together to produce a bigger run.
Question: How many I/Os to sort \( N \) elements?

- First run takes \( N/B \) I/Os.
- Each level of the merge tree takes \( N/B \) I/Os.
- How deep is the merge tree?

\[
O \left( \frac{N}{B} \log_{M/B} \frac{N}{B} \right)
\]

Cost to scan data \# of scans of data
Question: How many I/Os to sort $N$ elements?

- First run takes $N/B$ I/Os.
- Each level of the merge tree takes $N/B$ I/Os.
- How deep is the merge tree?

\[
O \left( \frac{N}{B} \log_{M/B} \frac{N}{B} \right)
\]

Cost to scan data  # of scans of data

This bound is the best possible.
Merge Sort as Divide-and-Conquer

To sort an array of $N$ objects

- If $N$ fits in main memory, then just sort elements.
- Otherwise,
  - divide the array into $M/B$ pieces;
  - sort each piece (recursively); and
  - merge the $M/B$ pieces.

This algorithm has the same I/O complexity.
Analysis of divide-and-conquer

Recurrence relation:

\[ T(N) = \frac{M}{B} \cdot T \left( \frac{N}{M/B} \right) + \frac{N}{B} \]

\[ T(N) = \frac{N}{B} \quad \text{when} \quad N < M \]

Solution:

\[ O \left( \frac{N}{B} \log_{M/B} \frac{N}{B} \right) \]

Cost to sort each piece recursively
Cost to merge
Cost to sort something that fits in memory
Cost to scan data
\# of scans of data
The Disk Access Machine (DAM) model

- ignores CPU costs and
- assumes that all block accesses have the same cost.

Is that a good performance model?
The DAM Model is a Simplification

Tracks get prefetched into the disk cache, which holds ~100 tracks.

Fixed-size blocks are fetched.

Disks are organized into tracks of different sizes.
The DAM Model is a Simplification

2kB or 4kB is too small for the model.

• B-tree nodes in Berkeley DB & InnoDB have this size.
• Issue: sequential block accesses run 10x faster than random block accesses, which doesn’t fit the model.

There is no single best block size.

• The best node size for a B-tree depends on the operation (insert/delete/point query).
Cache-oblivious analysis:

- Parameters $B$, $M$ are unknown to the algorithm or coder.
- Performance bounds are parameterized by block size $B$, memory size $M$, data size $N$.

Goal (as before): Minimize # of block transfer

[Frigo, Leiserson, Prokop, Ramachandran '99]
Cache-oblivious algorithms work for all $B$ and $M$...

... and all levels of a multi-level hierarchy.

It’s better to optimize approximately for all $B$, $M$ than to pick the best $B$ and $M$.

[Frigo, Leiserson, Prokop, Ramachandran ’99]
B-trees, k-way Merge Sort Aren’t Cache-Oblivious

Surprisingly, there are cache-oblivious B-trees and cache-oblivious sorting algorithms.

[Frigo, Leiserson, Prokop, Ramachandran ’99] [Bender, Demaine, Farach-Colton ’00] [Bender, Duan, Iacono, Wu ’02] [Brodal, Fagerberg, Jacob ’02] [Brodal, Fagerberg, Vinther ’04]
## Time for 1000 Random Searches

<table>
<thead>
<tr>
<th>B</th>
<th>Small</th>
<th>Big</th>
</tr>
</thead>
<tbody>
<tr>
<td>4K</td>
<td>17.3ms</td>
<td>22.4ms</td>
</tr>
<tr>
<td>16K</td>
<td>13.9ms</td>
<td>22.1ms</td>
</tr>
<tr>
<td>32K</td>
<td>11.9ms</td>
<td>17.4ms</td>
</tr>
<tr>
<td>64K</td>
<td>12.9ms</td>
<td>17.6ms</td>
</tr>
<tr>
<td>128K</td>
<td>13.2ms</td>
<td>16.5ms</td>
</tr>
<tr>
<td>256K</td>
<td>18.5ms</td>
<td>14.4ms</td>
</tr>
<tr>
<td>512K</td>
<td>16.7ms</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Small</th>
<th>Big</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO B-tree</td>
<td>12.3ms</td>
<td>13.8ms</td>
</tr>
</tbody>
</table>

There’s no best block size.

The optimal block size for inserts is very different.

[Source: Bender, Farach-Colton, Kuszmaul ’06]
Algorithmic models of the memory hierarchy explain how DB data structures scale.

- There’s a long history of models of the memory hierarchy. Many are beautiful. Most haven’t seen practical use.

DAM and cache-oblivious analysis are powerful

- Parameterized by block size $B$ and memory size $M$.
- In the CO model, $B$ and $M$ are unknown to the coder.
Data Structures and Algorithms for Big Data
Module 2: Write-Optimized Data Structures

Michael A. Bender  Bradley C. Kuszmaul
Stony Brook & Tokutek  MIT & Tokutek
Big data problem

Important and universal problem.

Hot topic.

query processor

queries + answers

data ingestion

data indexing

oy vey

365

42

???

???
For on-disk data, one sees funny tradeoffs in the speeds of data ingestion, query speed, and freshness of data.

For on-disk data, one sees funny tradeoffs in the speeds of data ingestion, query speed, and freshness of data.
Funny tradeoff in ingestion, querying, freshness

- “I’m trying to create indexes on a table with 308 million rows. It took ~20 minutes to load the table but 10 days to build indexes on it.”
  - MySQL bug #9544

- Typical record of all kinds of metadata is < 150 bytes.
- Different parts of metadata are accessed separately.

- “I’m trying to create indexes on a table with 308 million rows. It took ~20 minutes to load the table but 10 days to build indexes on it.”
  - MySQL bug #9544

- “Select queries were slow until I added an index onto the timestamp field... Adding the index really helped our reporting, BUT now the inserts are taking forever.”
  - Comment on mysqlperformanceblog.com

- “They indexed their tables, and indexed them well, And lo, did the queries run quick! But that wasn’t the last of their troubles, to tell– Their insertions, like molasses, ran thick.”
  - Not from Alice in Wonderland by Lewis Carroll
Funny tradeoff in ingestion, querying, freshness

- “I’m trying to create indexes on a table with 308 million rows. It took ~20 minutes to load the table but 10 days to build indexes on it.”
  - MySQL bug #9544
- “Select queries were slow until I added an index onto the timestamp field... Adding the index really helped our reporting, BUT now the inserts are taking forever.”
  - Comment on mysqlperformanceblog.com
Funny tradeoff in ingestion, querying, freshness

- “I’m trying to create indexes on a table with 308 million rows. It took ~20 minutes to load the table but 10 days to build indexes on it.”
  - MySQL bug #9544
- “Select queries were slow until I added an index onto the timestamp field... Adding the index really helped our reporting, BUT now the inserts are taking forever.”
  - Comment on mysqlperformanceblog.com
- “They indexed their tables, and indexed them well, And lo, did the queries run quick! But that wasn’t the last of their troubles, to tell—Their insertions, like treacle, ran thick.”
  - Not from Alice in Wonderland by Lewis Carroll
This module

- Write-optimized structures significantly mitigate the insert/query/freshness tradeoff.
- One can insert 10x-100x faster than B-trees while achieving similar point query performance.
How computation works:
• Data is transferred in blocks between RAM and disk.
• The number of block transfers dominates the running time.

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

[Aggarwal+Vitter ’88]
B-tree point queries: $O(\log_B N)$ I/Os.
Write-optimized data structures performance

**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:** BigTable, Cassandra, H-Base, LevelDB, TokuDB.

<table>
<thead>
<tr>
<th>Insert/delete</th>
<th>B-tree</th>
<th>Some write-optimized structures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$O(\log_B N) = O\left(\frac{\log N}{\log B}\right)$</td>
<td>$O\left(\frac{\log N}{B}\right)$</td>
</tr>
</tbody>
</table>

- If $B=1024$, then insert speedup is $B/\log B \approx 100$.
- Hardware trends mean bigger $B$, bigger speedup.
- Less than 1 I/O per insert.
**Optimal Search-Insert Tradeoff**  
[Brodal, Fagerberg 03]

<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>
</table>
| **B-tree**  
(\(\varepsilon=1\)) | \( O\left(\log_B N\right) \) | \( O\left(\log_B N\right) \) |
| \(\varepsilon=1/2\) | \( O\left(\frac{\log_B N}{\sqrt{B}}\right) \) | \( O\left(\log_B N\right) \) |
| \(\varepsilon=0\) | \( O\left(\frac{\log N}{B}\right) \) | \( O\left(\log N\right) \) |

10x-100x faster inserts
Illustration of Optimal Tradeoff [Brodal, Fagerberg 03]
Illustration of Optimal Tradeoff [Brodal, Fagerberg 03]

- **Target of opportunity**
  - B-tree
  - Insertions improve by 10x-100x with almost no loss of point-query performance

- **Fast**
- **Slow**

- **Point Queries**
  - Slow
  - Fast

- **Inserts**
  - Slow
  - Fast

- **Optimal Curve**

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

Illustration of Optimal Tradeoff [Brodal, Fagerberg 03]

- Target of opportunity
  - B-tree
  - Insertions improve by 10x-100x with almost no loss of point-query performance

- Fast
- Slow

- Point Queries
- Inserts

- Slow
- Fast

- Logging
One way to Build Write-Optimized Structures

(Other approaches later)
A simple write-optimized structure

O(log N) queries and O((log N)/B) inserts:
- A balanced binary tree with buffers of size B

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

\(O(\log N)\) queries and \(O((\log N)/B)\) inserts:

- A balanced binary tree with buffers of size \(B\)

Inserts + deletes:

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

O(log $N$) queries and O((log $N$)/$B$) inserts:

- A balanced binary tree with buffers of size $B$

 Inserts + deletes:

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

**O(log N) queries and O((log N)/B) inserts:**
- A balanced binary tree with buffers of size \( B \)

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

$O(\log N)$ queries and $O((\log N)/B)$ inserts:

- A balanced binary tree with buffers of size $B$

Inserts + deletes:

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

O(\log N) queries and O((\log N)/B) inserts:

- A balanced binary tree with buffers of size \( B \)

Inserts + deletes:

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

An insert/delete costs amortized $O((\log N)/B)$ per insert or delete

- A buffer flush costs $O(1)$ & sends $B$ elements down one level
- It costs $O(1/B)$ to send element down one level of the tree.
- There are $O(\log N)$ levels in a tree.
Difficulty of Key Accesses
Difficulty of Key Accesses
Analysis of point queries

To search:

- examine each buffer along a single root-to-leaf path.
- This costs $O(\log N)$. 
Point queries cost $O(\log_{\sqrt{B}} N) = O(\log_B N)$

- This is the tree height.

Inserts cost $O((\log_B N)/\sqrt{B})$

- Each flush cost $O(1)$ I/Os and flushes $\sqrt{B}$ elements.
You can even make these data structures cache-oblivious.

[Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson, SPAA 07]
[Brodal, Demaine, Fineman, Iacono, Langerman, Munro, SODA 10]

This means that the data structure can be made platform independent (no knobs), i.e., works simultaneously for all values of $B$ and $M$. 

Random accesses are expensive.

You can be cache- and I/O-efficient with no knobs or other memory-hierarchy parameterization.
Cache-oblivious write-optimized structures

You can even make these data structures cache-oblivious.

[Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson, SPAA 07]
[Brodal, Demaine, Fineman, Iacono, Langerman, Munro, SODA 10]

This means that the data structure can be made \textit{platform independent (no knobs)}, i.e., works simultaneously for all values of $B$ and $M$. 

Random accesses are expensive.

You can be cache- and I/O-efficient with no knobs or other memory-hierarchy parameterization.
What the world looks like

Insert/point query asymmetry

- Inserts can be fast: >50K high-entropy writes/sec/disk.
- Point queries are necessarily slow: <200 high-entropy reads/sec/disk.

We are used to reads and writes having about the same cost, but writing is easier than reading.
The right read-optimization is write-optimization

The right index makes queries run fast.

- Write-optimized structures maintain indexes efficiently.
The right read-optimization is write-optimization

The right index makes queries run fast.

- Write-optimized structures maintain indexes efficiently.

Fast writing is a currency we use to accelerate queries. Better indexing means faster queries.
The right read-optimization is write-optimization

Add selective indexes.
(We can now afford to maintain them.)
The right read-optimization is write-optimization.

Add selective indexes.
(We can now afford to maintain them.)

Write-optimized structures can significantly mitigate the insert/query/freshness tradeoff.
Implementation Issues
Optimal read-write tradeoff: Easy

Full featured: Hard

- 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
Systems often assume search cost = insert cost

Some inserts/deletes have hidden searches.

Example:

• return error when a duplicate key is inserted.
• return # elements removed on a delete.

These “cryptosearches” throttle insertions down to the performance of B-trees.
Uniqueness checking has a hidden search:

```python
If Search(key) == True
    Return Error;
Else
    Fast_Insert(key, value);
```

In a B-tree uniqueness checking comes for free

- On insert, you fetch a leaf.
- Checking if key exists is no biggie.
Cryptosearches in uniqueness checking

Uniqueness checking has a hidden search:

If Search(key) == True
    Return Error;
Else
    Fast_Insert(key,value);

In a write-optimized structure, that crypto-search can throttle performance

- Insertion messages are injected.
- These eventually get to “bottom” of structure.
- Insertion w/Uniqueness Checking 100x slower.
- Bloom filters, Cascade Filters, etc help.

[Bender, Farach-Colton, Johnson, Kraner, Kuszmaul, Medjedovic, Montes, Shetty, Spillane, Zadok 12]
A simple implementation of pessimistic locking: maintain locks in leaves

- Insert row $t$
- Search for row $u$
- Search for row $v$ and put a cursor
- Increment cursor. Now cursor points to row $w$.

This scheme is inefficient for write-optimized structures because there are cryptosearches on writes.
Performance
iiBench — Over 16x Faster Insertions

iiBench is a popular open-source benchmark developed by Tokutek. It measures how fast a storage engine can insert rows while maintaining secondary indexes. This is often a critical performance measurement since maintaining the right indexes will dramatically improve query performance. The schema consists of short rows that model a retail point-of-sale transaction system. The results below show the insertion of 1 billion rows into a table while maintaining three multicolumn secondary indexes. At the end of the test, TokuDB's insertion rate remained at 17,028 inserts/second whereas InnoDB had dropped to 1,050 inserts/second. That's a difference of over 16x.

Platform: Ubuntu 10.10; 2x Xeon X5460; 16GB RAM; 8x 146GB 10k SAS in RAID10.
Don't Thrash: How to Cache Your Hash in Flash

Platform: Ubuntu 11.04; Intel Corei7/920 @ 3.6Ghz; 12GB RAM; 2x 7.2k SATA.

SysBench — Up to 70% Faster

This is a SysBench comparison of InnoDB 1.1.8 and TokuDB v6.0. Prior to the run we started the database from a cold back-up (the cache is empty at the beginning of the 1 client thread run) and ran for 1 hour at each number of client threads. The following graph shows a significant performance improvement at all levels of concurrency. The values shown are the average transactions per second for the final 15 minutes of the benchmark.
TokuDB on rotating disk beats InnoDB on SSD.
Write-optimization Can Help Schema Changes

**InnoDB**
- Index Creation: 00:31:34
- Column Addition: 17:44:41

**TokuDB**
- Hot Indexing: 00:00:02
- Hot Column Addition: 00:00:03

Platform: CentOS 5.7; 2x Xeon L5520; 72GB RAM; 8x 300GB 10k SAS in RAID10.

Hot Schema – Schema Changes in Seconds, not Hours

TokuDB v5.0 introduced Hot Column Addition (HCAD). You can add or delete columns from an existing table with minimal downtime — just the time for MySQL itself to close and reopen the table. The total downtime is seconds to minutes. We detailed an experiment that showed this in this blog.

TokuDB v5.0 also introduced Hot Indexing. You can add an index to an existing table with minimal downtime. The total downtime is seconds to a few minutes, because when the index is finished being built, MySQL closes and reopens the table. This means that the downtime occurs not when the command is issued, but later on. Still, it is quite minimal, as we showed in this blog.

Platform: CentOS 5.5; 2x Xeon E5310; 4GB RAM; 4x 1TB 7.2k SATA in RAID0.

Appendix – Software Configuration Details
Cumulative Document Insertion Performance (with and without Fractal Tree Indexes)

Documents Inserted

Inserts per Second (cumulative)

MongoDB

MongoDB with FTI

65,000
81,315,000
163,630,000
245,945,000
328,260,000
410,575,000
492,890,000
575,205,000
657,520,000
739,835,000
822,150,000
904,465,000
986,780,000
1069,095,000
1151,410,000
1233,725,000
1316,040,000
1398,355,000
1480,670,000
1562,985,000
1645,300,000
1727,615,000
1809,930,000
1892,245,000
1974,560,000

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

MongoDB with Fractal-Tree Index
Scaling into the Future
Write-optimization going forward


- log high-entropy data sequentially versus index data in B-tree.

<table>
<thead>
<tr>
<th>Year</th>
<th>Size</th>
<th>Bandwidth</th>
<th>Access Time</th>
<th>Time to log data on disk</th>
<th>Time to fill disk using a B-tree (row size 1K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1973</td>
<td>35MB</td>
<td>835KB/s</td>
<td>25ms</td>
<td>39s</td>
<td>975s</td>
</tr>
<tr>
<td>2010</td>
<td>3TB</td>
<td>150MB/s</td>
<td>10ms</td>
<td>5.5h</td>
<td>347d</td>
</tr>
<tr>
<td>2022</td>
<td>220TB</td>
<td>1.05GB/s</td>
<td>10ms</td>
<td>2.4d</td>
<td>70y</td>
</tr>
</tbody>
</table>

Better data structures may be a luxury now, but they will be essential by the decade’s end.
Write-optimization going forward


- log high-entropy data sequentially versus index data in B-tree.

<table>
<thead>
<tr>
<th>Year</th>
<th>Size</th>
<th>Bandwidth</th>
<th>Access Time</th>
<th>Time to log data on disk</th>
<th>Time to fill disk using a B-tree (row size 1K)</th>
<th>Time to fill using Fractal tree* (row size 1K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1973</td>
<td>35MB</td>
<td>835KB/s</td>
<td>25ms</td>
<td>39s</td>
<td>975s</td>
<td></td>
</tr>
<tr>
<td>2010</td>
<td>3TB</td>
<td>150MB/s</td>
<td>10ms</td>
<td>5.5h</td>
<td>347d</td>
<td></td>
</tr>
<tr>
<td>2022</td>
<td>220TB</td>
<td>1.05GB/s</td>
<td>10ms</td>
<td>2.4d</td>
<td>70y</td>
<td></td>
</tr>
</tbody>
</table>

Better data structures may be a luxury now, but they will be essential by the decade’s end.

* Projected times for fully multi-threaded version

- log high-entropy data sequentially versus index data in B-tree.

<table>
<thead>
<tr>
<th>Year</th>
<th>Size</th>
<th>Bandwidth</th>
<th>Access Time</th>
<th>Time to log data on disk</th>
<th>Time to fill disk using a B-tree (row size 1K)</th>
<th>Time to fill using Fractal tree* (row size 1K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1973</td>
<td>35MB</td>
<td>835KB/s</td>
<td>25ms</td>
<td>39s</td>
<td>975s</td>
<td>200s</td>
</tr>
<tr>
<td>2010</td>
<td>3TB</td>
<td>150MB/s</td>
<td>10ms</td>
<td>5.5h</td>
<td>347d</td>
<td>36h</td>
</tr>
<tr>
<td>2022</td>
<td>220TB</td>
<td>1.05GB/s</td>
<td>10ms</td>
<td>2.4d</td>
<td>70y</td>
<td>23.3d</td>
</tr>
</tbody>
</table>

Better data structures may be a luxury now, but they will be essential by the decade’s end.

* Projected times for fully multi-threaded version
Write-optimization can solve many problems.

- There is a provable point-query insert tradeoff. We can insert 10x-100x faster without hurting point queries.
- We can avoid much of the funny tradeoff between data ingestion, freshness, and query speed.
- We can avoid tuning knobs.
Algorithms for Big Data apply to all storage systems, not just databases.

Some big-data users store use a file system.

The problem with Big Data is Microdata...
HEC FSIO Grand Challenges

Store 1 trillion files

Create tens of thousands of files per second

Traverse directory hierarchies fast (`ls -R`)

*B-trees would require at least hundreds of disk drives.*
TokuFS

[Esnet, Bender, Farach-Colton, Kuszmaul HotStorage12]

- A file-system prototype
- >20K file creates/sec
- very fast `ls -R`
- HEC grand challenges on a cheap disk (except 1 trillion files)
**TokuFS**

- A file-system prototype
- >20K file creates/sec
- very fast `ls -R`
- HEC grand challenges on a cheap disk (except 1 trillion files)

**TokuFS offers orders-of-magnitude speedup on microdata workloads.**
- Aggregates microwrites while indexing.
- So it can be faster than the underlying file system.

[Esmer, Bender, Farach-Colton, Kuszmaul HotStorage12]
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, we observed orders of magnitude speed up.
Create 2 million 200-byte files in a shallow tree

Randomized small file creation

- 1 thread
- 4 threads
- 8 threads

(files/sec vs. filesystem)

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

Faster on small file creation
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
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

![Microupdate bandwidth graph]

- btrfs
- zfs
- tokufs

write MB/s

MB/sec

filesystem
TokuFS
Implementation
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 as string**
- All the children of a directory are contiguous in the index
- Reading a directory is simple and fast

**Data block index keys: 【full path, blocknum】**
- So all the blocks for a file are contiguous in the index
- Reading a file is simple and fast
TokuFS compresses indexes

Reduces overhead from full path keys
- Pathnames are highly “prefix redundant”
- They compress very, very well in practice

Reduces overhead from zero-valued padding
- Uninitialized bytes in a block are set to zero
- Good portions of the metadata struct are set to zero

Compression between 7-15x on real data
- For example, a full MySQL source tree
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 user space.
• But there’s overhead, so performance isn’t optimal.
• The best way to run is through our POSIX-like file API.
Microdata is the Problem
Data Structures and Algorithms for Big Data
Module 4: Paging

Michael A. Bender
Stony Brook & Tokutek

Bradley C. Kuszmaul
MIT & Tokutek
This Module

The algorithmics of cache-management.

This will help us understand I/O- and cache-efficient algorithms.
Goal: minimize # block transfers.

- Data is transferred in blocks between RAM and disk.
- Performance bounds are parameterized by $B$, $M$, $N$.

When a block is cached, the access cost is 0. Otherwise it’s 1.
Disk Access Model (DAM Model):
- Performance bounds are parameterized by $B$, $M$, $N$.

**Goal:** Minimize # of block transfers.

**Beautiful restriction:**
- Parameters $B$, $M$ are unknown to the algorithm or coder.

[Frigo, Leiserson, Prokop, Ramachandran ’99]
CO analysis applies to unknown multilevel hierarchies:

- Cache-oblivious algorithms work for all $B$ and $M$...
- ... and all levels of a multi-level hierarchy.

Moral:
- It’s better to optimize approximately for all $B$, $M$ rather than to try to pick the best $B$ and $M$.

[Frigo, Leiserson, Prokop, Ramachandran ’99]
Which blocks are currently cached in RAM?

- The system performs its own caching/paging.
- If we knew $B$ and $M$ we could explicitly manage I/O. (But even then, what should we do?)

![Diagram of RAM and Disk with symbols for $M$ and $B$.]
Which blocks are currently cached in RAM?

- The system performs its own caching/paging.
- If we knew $B$ and $M$ we could explicitly manage I/O. (But even then, what should we do?)

But systems may use different mechanisms, so what can we actually assume?
With cache-oblivious analysis, we can assume a memory system with optimal replacement.

Even though the system manages memory, we can assume all the advantages of explicit memory management.
An LRU-based system with memory $M$ performs cache-management $< 2x$ worse than the optimal, prescient policy with memory $M/2$.

Achieving optimal cache-management is hard because predicting the future is hard.

But LRU with $(1+\varepsilon)M$ memory is almost as good (or better), than the optimal strategy with $M$ memory.

[Sleator, Tarjan 85]
The paging/caching problem

A program is just sequence of block requests:

\[ r_1, r_2, r_3, \ldots \]

Cost of request \( r_j \)

\[
\text{cost}(r_j) = \begin{cases} 
0 & \text{block } r_j \text{ is already cached,} \\
1 & \text{block } r_j \text{ is brought into cache.}
\end{cases}
\]
RAM holds only $k = \frac{M}{B}$ blocks.

*Which block should be ejected when block $r_j$ is brought into cache?*
Paging Algorithms

**LRU (least recently used)**
- Discard block whose most recent access is earliest.

**FIFO (first in, first out)**
- Discard the block brought in longest ago.

**LFU (least frequently used)**
- Discard the least popular block.

**Random**
- Discard a random block.

**LFD (longest forward distance) = OPT**
- Discard block whose next access is farthest in the future.

[Belady 69]
LFD (Longest Forward Distance) [Belady ’69]:

- Discard the block requested farthest in the future.
LFD (Longest Forward Distance) [Belady '69]:

- Discard the block requested farthest in the future.

**Cons:** Who knows the Future?!

Page 5348 shall be requested tomorrow at 2:00 pm
LFD (Longest Forward Distance) [Belady ’69]:

- Discard the block requested farthest in the future.

Cons: Who knows the Future?!

Pros: LFD can be viewed as a point of comparison with online strategies.
An online algorithm $A$ is $k$-competitive, if for every request sequence $R$:

$$\text{cost}_A(R) \leq k \text{ cost}_{\text{opt}}(R)$$

Idea of competitive analysis:

- The optimal (prescient) algorithm is a yardstick we use to compare online algorithms.
LRU is no better than k-competitive

Memory holds 3 blocks

\[ M/B = k = 3 \]

The program accesses 4 different blocks

\[ r_j \in \{1, 2, 3, 4\} \]

The request stream is

1, 2, 3, 4, 1, 2, 3, 4, \ldots
LRU is no better than k-competitive

There’s a block transfer at every step because LRU ejects the block that’s requested in the next step.
LRU is no better than k-competitive

LFD (longest forward distance) has a block transfer every k=3 steps.
In fact, LRU is $k=M/B$-competitive.

- I.e., LRU has $k=M/B$ times more transfers than OPT.
- A depressing result because $k$ is huge so $k \cdot \text{OPT}$ is nothing to write home about.

LFU and FIFO are also $k$-competitive.

- This is a depressing result because FIFO is empirically worse than LRU, and this isn’t captured in the math.
On the other hand, the LRU bad example is fragile

If $k = M/B = 4$, not 3, then both LRU and OPT do well. If $k = M/B = 2$, not 3, then neither LRU nor OPT does well.
LRU is 2-competitive with more memory [Sleator, Tarjan 85]

**LRU** is at most twice as bad as **OPT**, when **LRU** has twice the memory.

\[
\text{LRU}_{|\text{cache}| = k(R)} \leq 2 \text{OPT}_{|\text{cache}| = k/2(R)}
\]

In general, **LRU** is nearly as good as **OPT** when **LRU** has a little more memory than **OPT**.
LRU is 2-competitive with more memory \[ \text{[Sleator, Tarjan 85]} \]

LRU is at most twice as bad as OPT, when LRU has twice the memory.

\[ \text{LRU}_{\text{cache}} = k(R) \leq 2 \text{OPT}_{\text{cache}} = k/2(R) \]

LRU has more memory, but OPT=LFD can see the future.

In general, LRU is nearly as good as OPT when LRU has a little more memory than OPT.
LRU is 2-competitive with more memory \cite{SleatorTarjan85}

LRU is at most twice as bad as OPT, when LRU has twice the memory.

\[
\text{LRU}_{\text{cache}} = k(R) \leq 2 \text{OPT}_{\text{cache}} = k/2(R)
\]

LRU has more memory, but OPT=LFD can see the future.

In general, LRU is nearly as good as OPT when LRU has a little more memory than OPT.
LRU is 2-competitive with more memory \[\text{[Sleator, Tarjan 85]}\]

LRU is at most twice as bad as OPT, when LRU has twice the memory.

\[
\text{LRU}_{\text{cache}} = k(R) \leq 2 \text{OPT}_{\text{cache}} = k/2(R)
\]

LRU has more memory, but OPT=LFD can see the future.

In general, LRU is nearly as good as OPT when LRU has a little more memory than OPT.

(These bounds don’t apply to FIFO, distinguishing LRU from FIFO).
Divide LRU into phases, each with k faults.

\[ r_1, r_2, \ldots, r_i, r_i+1, \ldots, r_j, r_j+1, \ldots, r_\ell, r_\ell+1, \ldots \]
Divide LRU into phases, each with k faults.

\[ r_1, r_2, \ldots, r_i, r_{i+1}, \ldots, r_j, r_{j+1}, \ldots, r_\ell, r_{\ell+1}, \ldots \]

**OPT[k]** must have \( \geq 1 \) fault in each phase.

- Case analysis proof.
- LRU is k-competitive.
Divide LRU into phases, each with $k$ faults.

$\{r_1, r_2, \ldots, r_i, r_{i+1}, \ldots, r_j, r_{j+1}, \ldots, r_\ell, r_{\ell+1}, \ldots\}$

**OPT[$k$] must have $\geq 1$ fault in each phase.**
- Case analysis proof.
- LRU is $k$-competitive.

**OPT[$k/2$] must have $\geq k/2$ faults in each phase.**
- Main idea: each phase must touch $k$ different pages.
- LRU is $2$-competitive.
Moral: with cache-oblivious analysis, we can analyze based on a memory system with optimal, omniscient replacement.

- Technically, an optimal cache-oblivious algorithm is asymptotically optimal versus any algorithm on a memory system that is slightly smaller.
- Empirically, this is just a technicality.

This is almost as good or better... ...than this.
Moral: There’s not much performance on the table for new cache-replacement policies.

- Bad instances for LRU versus LFD are fragile and very sensitive to $k=M/B$.

There are still research questions:

- What if blocks have different sizes [Irani 02][Young 02]?
- There’s a write-back cost? (Complexity unknown.)
- LRU may be too costly to implement (clock algorithm).
Data Structures and Algorithms for Big Data
Module 5: What to Index

Michael A. Bender
Stony Brook & Tokutek

Bradley C. Kuszmaul
MIT & Tokutek
This module explores indexing.

Traditionally, (with B-trees), indexing speeds queries, but cripples insert.

But now we know that maintaining indexes is cheap. So what should you index?
Add selective indexes.

This is a graph from a real user, who added some indexes, and reduced the I/O load on their server. (They couldn’t maintain the indexes with B-trees.)
What is an Index?

To understand what to index, we need to get on the same page for what an index is.
Row, Index, and Table

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>101</td>
<td>92</td>
<td>2</td>
</tr>
<tr>
<td>156</td>
<td>56</td>
<td>45</td>
</tr>
<tr>
<td>165</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>198</td>
<td>202</td>
<td>56</td>
</tr>
<tr>
<td>206</td>
<td>23</td>
<td>252</td>
</tr>
<tr>
<td>256</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>412</td>
<td>43</td>
<td>45</td>
</tr>
</tbody>
</table>

**Row**
- Key, value pair
- key = a, value = b, c

**Index**
- Ordering of rows by key (dictionary)
- Used to make queries fast

**Table**
- Set of indexes

```
cREATE TABLE foo (a INT, b INT, c INT, primary key(a));
```
An index is a dictionary

Dictionary API: maintain a set $S$ subject to

- $\text{insert}(x)$: $S \leftarrow S \cup \{x\}$
- $\text{delete}(x)$: $S \leftarrow S - \{x\}$
- $\text{search}(x)$: is $x \in S$?
- $\text{successor}(x)$: return $\min y > x$ s.t. $y \in S$
- $\text{predecessor}(y)$: return $\max y < x$ s.t. $y \in S$

We assume that these operations perform as well as a B-tree. For example, the successor operation usually doesn’t require an I/O.
A table is a set of indexes with operations:

- **Add index**: `add key(f_1, f_2, ...);`
- **Drop index**: `drop key(f_1, f_2, ...);`
- **Add column**: adds a field to primary key value.
- **Remove column**: removes a field and drops all indexes where field is part of key.
- **Change field type**
- ...

Subject to index correctness constraints.

*We want table operations to be fast too.*
Next: how to use indexes to improve queries.
Indexes provide query performance

1. Indexes can reduce the amount of retrieved data.
   • Less bandwidth, less processing, ...

2. Indexes can improve locality.
   • Not all data access cost is the same
   • Sequential access is MUCH faster than random access

3. Indexes can presort data.
   • GROUP BY and ORDER BY queries do post-retrieval work
   • Indexing can help get rid of this work
Indexes provide query performance

1. **Indexes can reduce the amount of retrieved data.**
   - Less bandwidth, less processing, ...

2. **Indexes can improve locality.**
   - Not all data access cost is the same
   - Sequential access is MUCH faster than random access

3. **Indexes can presort data.**
   - GROUP BY and ORDER BY queries do post-retrieval work
   - Indexing can help get rid of this work
An index can select needed rows

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>101</td>
<td>92</td>
<td>2</td>
</tr>
<tr>
<td>156</td>
<td>56</td>
<td>45</td>
</tr>
<tr>
<td>165</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>198</td>
<td>202</td>
<td>56</td>
</tr>
<tr>
<td>206</td>
<td>23</td>
<td>252</td>
</tr>
<tr>
<td>256</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>412</td>
<td>43</td>
<td>45</td>
</tr>
</tbody>
</table>

count (*) where a<120;
An index can select needed rows

count (*) where a<120;
No good index means slow table scans

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>101</td>
<td>92</td>
<td>2</td>
</tr>
<tr>
<td>156</td>
<td>56</td>
<td>45</td>
</tr>
<tr>
<td>165</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>198</td>
<td>202</td>
<td>56</td>
</tr>
<tr>
<td>206</td>
<td>23</td>
<td>252</td>
</tr>
<tr>
<td>256</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>412</td>
<td>43</td>
<td>45</td>
</tr>
</tbody>
</table>

`count (*) where b>50 and b<100;`
No good index means slow table scans

count (*) where b>50 and b<100;
You can add an index

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>100</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>101</td>
<td>92</td>
<td>2</td>
</tr>
<tr>
<td>156</td>
<td>56</td>
<td>45</td>
</tr>
<tr>
<td>165</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>198</td>
<td>202</td>
<td>56</td>
</tr>
<tr>
<td>206</td>
<td>23</td>
<td>252</td>
</tr>
<tr>
<td>256</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>412</td>
<td>43</td>
<td>45</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>165</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>206</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>412</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>101</td>
<td></td>
</tr>
</tbody>
</table>

```
alter table foo add key (b);
```
A selective index speeds up queries

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>100</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>101</td>
<td>92</td>
<td>2</td>
</tr>
<tr>
<td>156</td>
<td>56</td>
<td>45</td>
</tr>
<tr>
<td>165</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>198</td>
<td>202</td>
<td>56</td>
</tr>
<tr>
<td>206</td>
<td>23</td>
<td>252</td>
</tr>
<tr>
<td>256</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>412</td>
<td>43</td>
<td>45</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>165</td>
</tr>
<tr>
<td>23</td>
<td>206</td>
</tr>
<tr>
<td>43</td>
<td>412</td>
</tr>
<tr>
<td>56</td>
<td>156</td>
</tr>
<tr>
<td>56</td>
<td>256</td>
</tr>
<tr>
<td>92</td>
<td>101</td>
</tr>
<tr>
<td>202</td>
<td>198</td>
</tr>
</tbody>
</table>

count (*) where b>50 and b<100;
A selective index speeds up queries

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>101</td>
<td>92</td>
<td>2</td>
</tr>
<tr>
<td>156</td>
<td>56</td>
<td>45</td>
</tr>
<tr>
<td>165</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>198</td>
<td>202</td>
<td>56</td>
</tr>
<tr>
<td>206</td>
<td>23</td>
<td>252</td>
</tr>
<tr>
<td>256</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>412</td>
<td>43</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>165</td>
</tr>
<tr>
<td>23</td>
<td>206</td>
</tr>
<tr>
<td>43</td>
<td>412</td>
</tr>
<tr>
<td>56</td>
<td>156</td>
</tr>
<tr>
<td>56</td>
<td>156</td>
</tr>
<tr>
<td>56</td>
<td>256</td>
</tr>
<tr>
<td>92</td>
<td>101</td>
</tr>
</tbody>
</table>

count (*) where b>50 and b<100;
Selective indexes can still be slow

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>101</td>
<td>92</td>
<td>2</td>
</tr>
<tr>
<td>156</td>
<td>56</td>
<td>45</td>
</tr>
<tr>
<td>165</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>198</td>
<td>202</td>
<td>56</td>
</tr>
<tr>
<td>206</td>
<td>23</td>
<td>252</td>
</tr>
<tr>
<td>256</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>412</td>
<td>43</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>165</td>
</tr>
<tr>
<td>23</td>
<td>206</td>
</tr>
<tr>
<td>43</td>
<td>412</td>
</tr>
<tr>
<td>56</td>
<td>156</td>
</tr>
<tr>
<td>56</td>
<td>256</td>
</tr>
<tr>
<td>92</td>
<td>101</td>
</tr>
<tr>
<td>202</td>
<td>198</td>
</tr>
</tbody>
</table>

\[ \text{sum}(c) \text{ where } b>50; \]
Selective indexes can still be slow

$$\text{sum}(c) \text{ where } b>50;$$
### Selective indexes can still be slow

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>101</td>
<td>92</td>
<td>2</td>
</tr>
<tr>
<td>156</td>
<td>56</td>
<td>45</td>
</tr>
<tr>
<td>165</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>198</td>
<td>202</td>
<td>56</td>
</tr>
<tr>
<td>206</td>
<td>23</td>
<td>252</td>
</tr>
<tr>
<td>256</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>412</td>
<td>43</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>165</td>
</tr>
<tr>
<td>23</td>
<td>206</td>
</tr>
<tr>
<td>43</td>
<td>412</td>
</tr>
<tr>
<td>56</td>
<td>156</td>
</tr>
<tr>
<td>56</td>
<td>256</td>
</tr>
<tr>
<td>92</td>
<td>101</td>
</tr>
<tr>
<td>202</td>
<td>198</td>
</tr>
</tbody>
</table>

**sum(c) where b>50;**
Selective indexes can still be slow

```sql
sum(c) where b>50;
```
Selective indexes can still be slow

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>101</td>
<td>92</td>
<td>2</td>
</tr>
<tr>
<td>156</td>
<td>56</td>
<td>45</td>
</tr>
<tr>
<td>165</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>198</td>
<td>202</td>
<td>56</td>
</tr>
<tr>
<td>206</td>
<td>23</td>
<td>252</td>
</tr>
<tr>
<td>256</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>412</td>
<td>43</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>165</td>
</tr>
<tr>
<td>23</td>
<td>206</td>
</tr>
<tr>
<td>43</td>
<td>412</td>
</tr>
<tr>
<td>56</td>
<td>156</td>
</tr>
<tr>
<td>56</td>
<td>256</td>
</tr>
<tr>
<td>92</td>
<td>101</td>
</tr>
<tr>
<td>92</td>
<td>101</td>
</tr>
<tr>
<td>202</td>
<td>198</td>
</tr>
</tbody>
</table>

sum(c) where b>50;
Selective indexes can still be slow

**Table:**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>101</td>
<td>92</td>
<td>2</td>
</tr>
<tr>
<td>156</td>
<td>56</td>
<td>45</td>
</tr>
<tr>
<td>165</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>198</td>
<td>202</td>
<td>56</td>
</tr>
<tr>
<td>206</td>
<td>23</td>
<td>252</td>
</tr>
<tr>
<td>256</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>412</td>
<td>43</td>
<td>45</td>
</tr>
</tbody>
</table>

**Another Table:**

<table>
<thead>
<tr>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>165</td>
</tr>
<tr>
<td>23</td>
<td>206</td>
</tr>
<tr>
<td>43</td>
<td>412</td>
</tr>
<tr>
<td>56</td>
<td>156</td>
</tr>
<tr>
<td>56</td>
<td>256</td>
</tr>
<tr>
<td>92</td>
<td>101</td>
</tr>
<tr>
<td>202</td>
<td>198</td>
</tr>
</tbody>
</table>

**Explanation:**

- **Selecting on b:** Fast
- **Fetching info for summing c:** Slow

**SQL Query:**

```sql
sum(c) where b > 50;
```
Selective indexes can still be slow

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>101</td>
<td>92</td>
<td>2</td>
</tr>
<tr>
<td>156</td>
<td>56</td>
<td>45</td>
</tr>
<tr>
<td>165</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>198</td>
<td>202</td>
<td>56</td>
</tr>
<tr>
<td>206</td>
<td>23</td>
<td>252</td>
</tr>
<tr>
<td>256</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>412</td>
<td>43</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>165</td>
</tr>
<tr>
<td>23</td>
<td>206</td>
</tr>
<tr>
<td>43</td>
<td>412</td>
</tr>
<tr>
<td>56</td>
<td>156</td>
</tr>
<tr>
<td>56</td>
<td>256</td>
</tr>
<tr>
<td>92</td>
<td>101</td>
</tr>
<tr>
<td>202</td>
<td>198</td>
</tr>
</tbody>
</table>

Poor data locality

```
sum(c) where b>50;
```
Selective indexes can still be slow

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>101</td>
<td>92</td>
<td>2</td>
</tr>
<tr>
<td>156</td>
<td>56</td>
<td>45</td>
</tr>
<tr>
<td>165</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>198</td>
<td>202</td>
<td>56</td>
</tr>
<tr>
<td>206</td>
<td>23</td>
<td>252</td>
</tr>
<tr>
<td>256</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>412</td>
<td>43</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>165</td>
</tr>
<tr>
<td>23</td>
<td>206</td>
</tr>
<tr>
<td>43</td>
<td>412</td>
</tr>
<tr>
<td>56</td>
<td>156</td>
</tr>
<tr>
<td>56</td>
<td>101</td>
</tr>
<tr>
<td>202</td>
<td>198</td>
</tr>
</tbody>
</table>

`sum(c) where b>50;`
Indexes provide query performance

1. Indexes can reduce the amount of retrieved data.
   • Less bandwidth, less processing, ...

2. Indexes can improve locality.
   • Not all data access cost is the same
   • Sequential access is MUCH faster than random access

3. Indexes can presort data.
   • GROUP BY and ORDER BY queries do post-retrieval work
   • Indexing can help get rid of this work
Covering indexes speed up queries

```
alter table foo add key(b,c);
sum(c) where b>50;
```
Covering indexes speed up queries

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>101</td>
<td>92</td>
<td>2</td>
</tr>
<tr>
<td>156</td>
<td>56</td>
<td>45</td>
</tr>
<tr>
<td>165</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>198</td>
<td>202</td>
<td>56</td>
</tr>
<tr>
<td>206</td>
<td>23</td>
<td>252</td>
</tr>
<tr>
<td>256</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>412</td>
<td>43</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b,c</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,45</td>
<td>100</td>
</tr>
<tr>
<td>6,2</td>
<td>165</td>
</tr>
<tr>
<td>23,252</td>
<td>206</td>
</tr>
<tr>
<td>43,45</td>
<td>412</td>
</tr>
<tr>
<td>56,2</td>
<td>256</td>
</tr>
<tr>
<td>56,45</td>
<td>156</td>
</tr>
<tr>
<td>92,2</td>
<td>101</td>
</tr>
<tr>
<td>202,56</td>
<td>198</td>
</tr>
</tbody>
</table>

```
alter table foo add key(b,c);
sum(c) where b>50;
```
Indexes provide query performance

1. Indexes can reduce the amount of retrieved data.
   - Less bandwidth, less processing, ...

2. Indexes can improve locality.
   - Not all data access cost is the same
   - Sequential access is MUCH faster than random access

3. Indexes can presort data.
   - GROUP BY and ORDER BY queries do post-retrieval work
   - Indexing can help get rid of this work
Indexes can avoid post-selection sorts

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>101</td>
<td>92</td>
<td>2</td>
</tr>
<tr>
<td>156</td>
<td>56</td>
<td>45</td>
</tr>
<tr>
<td>165</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>198</td>
<td>202</td>
<td>56</td>
</tr>
<tr>
<td>206</td>
<td>23</td>
<td>252</td>
</tr>
<tr>
<td>256</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>412</td>
<td>43</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b,c</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,45</td>
<td>100</td>
</tr>
<tr>
<td>6,2</td>
<td>165</td>
</tr>
<tr>
<td>23,252</td>
<td>206</td>
</tr>
<tr>
<td>43,45</td>
<td>412</td>
</tr>
<tr>
<td>56,2</td>
<td>256</td>
</tr>
<tr>
<td>56,45</td>
<td>156</td>
</tr>
<tr>
<td>92,2</td>
<td>101</td>
</tr>
<tr>
<td>202,56</td>
<td>198</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>sum(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>23</td>
<td>252</td>
</tr>
<tr>
<td>43</td>
<td>45</td>
</tr>
<tr>
<td>56</td>
<td>47</td>
</tr>
<tr>
<td>92</td>
<td>2</td>
</tr>
<tr>
<td>202</td>
<td>56</td>
</tr>
</tbody>
</table>

```sql
select b, sum(c) group by b;
sum(c) where b>50;
```
Log Structured Merge Trees

Log structured merge trees are write-optimized data structures developed in the 90s.

Over the past 5 years, LSM trees have become popular (for good reason).

Accumulo, Bigtable, bLSM, Cassandra, HBase, Hypertable, LevelDB are LSM trees (or borrow ideas).

http://nosql-database.org lists 122 NoSQL databases. Many of them are LSM trees.
Recall Optimal Search-Insert Tradeoff

**Optimal tradeoff**
(function of $\varepsilon=0...1$)

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

- Point query:
  
  $$O \left( \log_{1+B\varepsilon} N \right)$$

**LSM trees don’t lie on the optimal search-insert tradeoff curve.**

**But they’re not far off.**

**We’ll show how to move them back onto the optimal curve.**
An LSM tree is a cascade of B-trees. Each tree $T_j$ has a target size $|T_j|$. The target sizes are exponentially increasing. Typically, target size $|T_{j+1}| = 10 |T_j|$.

[O'Neil, Cheng, Gawlick, O'Neil 96]
LSM Tree Operations

Point queries:

\[ T_0 \quad T_1 \quad T_2 \quad T_3 \quad T_4 \]
Point queries:

Range queries:
Insertions:

- Always insert element into the smallest B-tree $T_0$.

- When a B-tree $T_j$ fills up, flush into $T_{j+1}$.
Deletes are like inserts:

• Instead of deleting an element directly, insert tombstones.
• A tombstone knocks out a “real” element when it lands in the same tree.
Static-to-Dynamic Transformation

An LSM Tree is an example of a “static-to-dynamic” transformation [Bentley, Saxe '80].

- An LSM tree can be built out of static B-trees.
- When $T_3$ flushes into $T_4$, $T_4$ is rebuilt from scratch.
This Module

Let’s analyze LSM trees.
Recall the cost of searching in an array versus a B-tree.

\[ O(\log_B N) = O\left(\frac{\log_2 N}{\log_2 B}\right) \]
Recall: Searching in an Array Versus B-tree

Recall the cost of searching in an array versus a B-tree.

\[
O\left(\log_2 \frac{N}{B}\right) \approx O\left(\log_2 N\right)
\]

\[
O\left(\log_B N\right) = O\left(\frac{\log_2 N}{\log_2 B}\right)
\]
Analysis of point queries

Search cost:

\[ \log_B N + \log_B N/2 + \log_B N/4 + \cdots + \log_B B \]

\[ = \frac{1}{\log B} (\log N + \log N - 1 + \log N - 2 + \log N - 3 + \cdots + 1) \]

\[ = O(\log N \log_B N) \]
The cost to flush a tree $T_j$ of size $X$ is $O(X/B)$.

- Flushing and rebuilding a tree is just a linear scan.

The cost per element to flush $T_j$ is $O(1/B)$.

The # times each element is moved is $\leq \log N$.

The insert cost is $O((\log N)/B)$ amortized memory transfers.
## Samples from LSM Tradeoff Curve

### Tradeoff

<table>
<thead>
<tr>
<th>Insert</th>
<th>Point Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O \left( \frac{\log_{1+B^\varepsilon} N}{B^{1-\varepsilon}} \right) )</td>
<td>( O \left( (\log_B N)(\log_{1+B^\varepsilon} N) \right) )</td>
</tr>
</tbody>
</table>

### Sizes grow by \( B \)

- \( \varepsilon = 1 \)
  - \( O(\log_B N) \)
  - \( O((\log_B N)(\log_B N)) \)

### Sizes grow by \( B^{1/2} \)

- \( \varepsilon = 1/2 \)
  - \( O\left( \frac{\log_B N}{\sqrt{B}} \right) \)
  - \( O((\log_B N)(\log_B N)) \)

### Sizes double

- \( \varepsilon = 0 \)
  - \( O\left( \frac{\log N}{B} \right) \)
  - \( O((\log_B N)(\log N)) \)
How to improve LSM-tree point queries?

Looking in all those trees is expensive, but can be improved by

- caching,
- Bloom filters, and
- fractional cascading.

\[ T_0 \quad T_1 \quad T_2 \quad T_3 \quad T_4 \]
Caching in LSM trees

When the cache is warm, small trees are cached.

When the cache is warm, these trees are cached.
Bloom filters in LSM trees

Bloom filters can avoid point queries for elements that are not in a particular B-tree.

We’ll see how Bloom filters work later.
Fractional cascading reduces the cost in each tree

Instead of avoiding searches in trees, we can use a technique called *fractional cascading* to reduce the cost of searching each B-tree to $O(1)$.

Idea: We’re looking for a key, and we already know where it should have been in $T_3$, try to use that information to search $T_4$. 
Searching one tree helps in the next

Looking up c, in $T_i$ we know it’s between b, and e.

$T_i$

```plaintext
b e v w
```

$T_{i+1}$

```plaintext
a c d f h i j k m n p q t u y z
```
If we add *forwarding pointers* to the first tree, we can jump straight to the node in the second tree, to find c.
Remove redundant forwarding pointers

We need only one forwarding pointer for each block in the next tree. Remove the redundant ones.
We need a forwarding pointer for every block in the next tree, even if there are no corresponding pointers in this tree. Add **ghosts**.

\[ T_i \]

\[ T_{i+1} \]
LSM tree + forward + ghost = fast queries

With forward pointers and ghosts, LSM trees require only one I/O per tree, and point queries cost only $O(\log_R N)$.

[Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson 07]
This data structure no longer uses the internal nodes of the B-trees, and each of the trees can be implemented by an array.

[Reference: Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson 07]
Data Structures and Algorithms for Big Data
Module 7: Bloom Filters

Michael A. Bender
Stony Brook & Tokutek

Bradley C. Kuszmaul
MIT & Tokutek
Approximate Set Membership Problem

We need a space-efficient in-memory data structure to represent a set $S$ to which we can add elements. We want to answer *membership queries* approximately:

- If $x$ is in $S$ then we want $\text{query}(x, S)$ to return true.
- Otherwise we want $\text{query}(x, S)$ to usually return false.

Bloom filters are a simple data structure to solve this problem.
How do approximate queries help?

Recall for LSM trees (without fractional cascading), we wanted to avoid looking in a tree if we knew a key wasn’t there.

Bloom filters allow us to *usually* avoid the lookup.

Bloom filters don’t seem to help with range queries, however.
Simplified Bloom Filter

Using hashing, but instead of storing elements we simply use one bit to keep track of whether an element is in the set.

- Array $A[m]$ bits.
- Uniform hash function $h: S \rightarrow [0,m)$.
- To insert $s$: Set $A[h(s)] = 1$;
- To check $s$: Check if $A[h(s)] = 1$. 
Example using Simplified Bloom Filter

Use an array of length 6. Insert

- insert \(a\), where \(h(a)=3\);
- \(b\), where \(h(b)=5\).

Look up

- \(a\): \(h(a)=3\)  Answer is yes. Maybe \(a\) is there. (And it is).
- \(b\): \(h(b)=5\)  Answer is yes. Maybe \(b\) is there. (And it is).
- \(c\): \(h(c)=2\)  Answer is no. Definitely \(c\) is not there.
- \(d\): \(h(d)=3\)  Answer is yes. Maybe \(d\) is there. (Nope.)
If \( n \) items are in an array of size \( m \), then the chances of getting a YES answer on an element that is not there is \( \approx 1 - e^{-n/m} \).

If you fill the array about 30% full, you get about a 50% odds of a false positive. Each object requires about 3 bits.

How do you get the odds to be 1% false positive?
One way would be to fill the array only 1% full. Not space efficient.

Another way would be to use 7 arrays, with 7 hash functions. False positive rate becomes 1/128.

Space is 21 bits per object.
Idea: Don’t use 7 separate arrays, use one array that’s 7 times bigger, and store the 7 hashed bits.

For a 1% false positive rate, it takes about 10 bits per object.
Other Bloom Filters

Counting bloom filters [Fan, Cao, Almeida, Broder 2000] allow deletions by maintaining a 4-bit counter instead of a single bit per object.

Buffered Bloom Filters [Canin, Mihaila, Bhattacharhee, and Ross, 2010] employ hash localization to direct all the hashes of a single insertion to the same block.

Cascade Filters [Bender, Farach-Colton, Johnson, Kraner, Kuszmaul, Medjedovic, Montes, Shetty, Spillane, Zadok 2011] support deletions, exhibit locality for queries, insert quickly, and are cache-oblivious.
Closing Words
We want to feel your pain.

We are interested in hearing about other scaling problems.

Come to talk to us.

bender@cs.stonybrook.edu
bradley@mit.edu
The problem with big data is microdata.

Sometimes the right read optimization is a write-optimization.

As data becomes bigger, the asymptotics become more important.

Life is too short for half-dry white-board markers and bad sushi.