# Write-optimization in external memory data structures

Leif Walsh

Two Sigma Investments, LLC.
leif.walsh@gmail.com
@leifwalsh

April 16, 2015

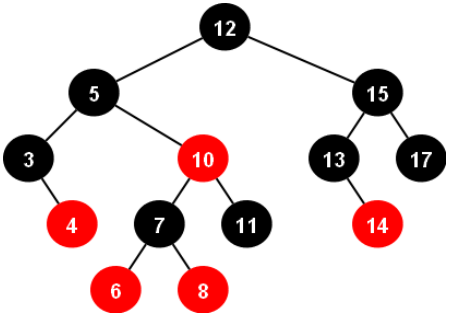
# Background

# Write-optimization in external memory **data structures**

Data structures:

# Write-optimization in external memory **data structures**

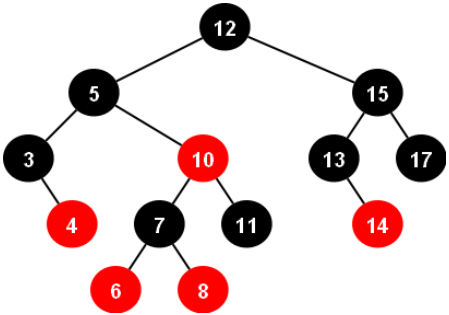
Data structures:



# Write-optimization in external memory **data structures**

Data structures:

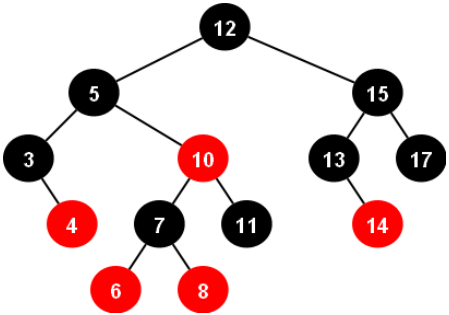
- Provide retrieval of data.
  - *Lookup*(*Key*)
  - *Pred*(*Key*)
  - *Succ*(*Key*)



# Write-optimization in external memory **data structures**

Data structures:

- Provide retrieval of data.
  - *Lookup*(*Key*)
  - *Pred*(*Key*)
  - *Succ*(*Key*)
- *Dynamic* data structures let you change the data.
  - *Insert*(*Key*, *Value*)
  - *Delete*(*Key*)



# Write-optimization in **external memory** data structures

[Aggarwal & Vitter '88]

DAM model

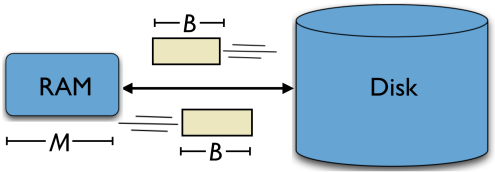
- Problem size *N*.
- Memory size *M*.

# Write-optimization in **external memory** data structures

[Aggarwal & Vitter '88]

DAM model

- Problem size *N*.
- Memory size *M*.
- Transfer data to/from memory in *blocks* of size *B*.



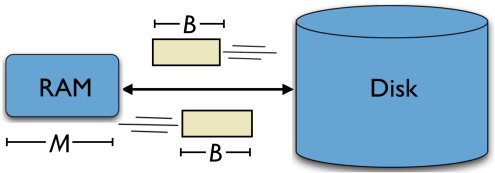
# Write-optimization in **external memory** data structures

[Aggarwal & Vitter '88]

DAM model

- Problem size *N*.
- Memory size *M*.
- Transfer data to/from memory in *blocks* of size *B*.

Efficiency of operations is measured as the *number of block transfers*, a.k.a. IOPS.

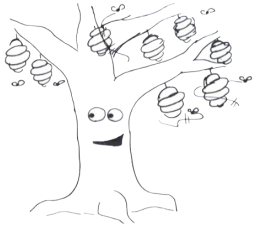


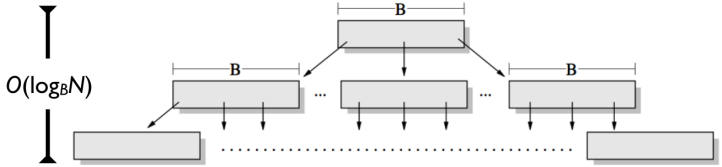
# Write-optimization in **external memory data structures**

A B-tree is an *external memory data structure*:

# Write-optimization in **external memory data structures**

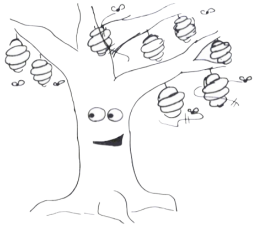
A B-tree is an *external memory data structure*:

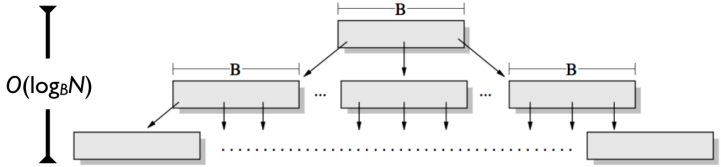




# Write-optimization in **external memory data structures**

A B-tree is an *external memory data structure*:

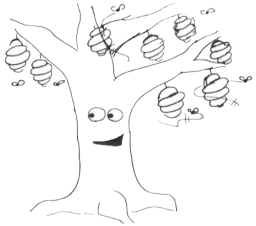


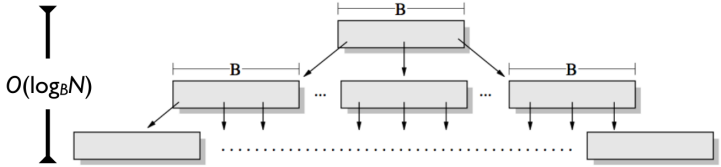



- Balanced search tree.
- Fanout of $B$
  (block size / key size).

# Write-optimization in **external memory data structures**

A B-tree is an *external memory data structure*:



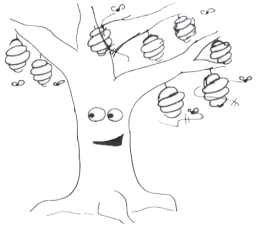


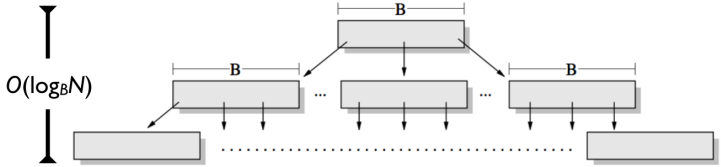

- Balanced search tree.
- Fanout of $B$
  (block size / key size).

- Internal nodes $< M$.
- Leaf nodes $> M$.

# Write-optimization in **external memory data structures**

A B-tree is an *external memory data structure*:



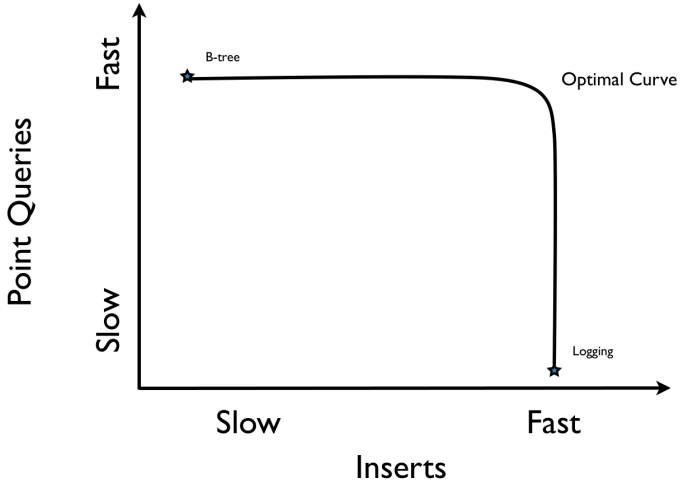



- Balanced search tree.
- Fanout of $B$ (block size / key size).

- Internal nodes $< M$.
- Leaf nodes $> M$.
- Search: $O(\log_B N)$ I/Os
- Insert: $O(\log_B N)$ I/Os

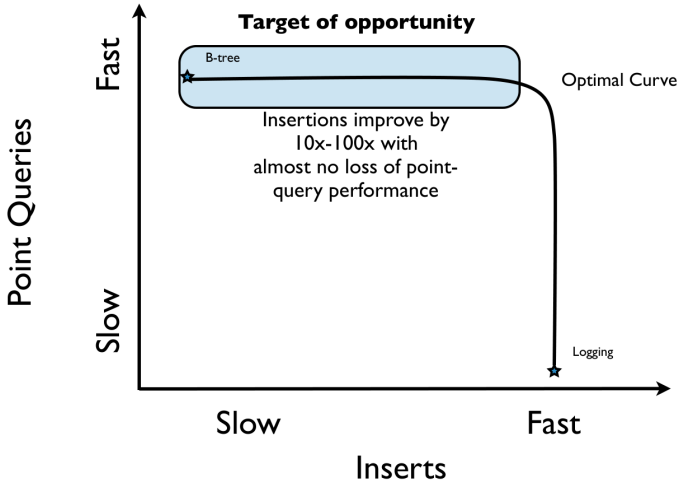
# **Write-optimization** in external memory data structures

[Brodal & Fagerberg '03]



# **Write-optimization** in external memory data structures

[Brodal & Fagerberg '03]



# OLAP

# Write-optimization technique #1: OLAP

OLAP: Online **Analytical** Processing

# Write-optimization technique #1: OLAP

OLAP: Online **Analytical** Processing
Key idea: Analyze data collected in the past.

# Write-optimization technique #1: OLAP

OLAP: Online **Analytical** Processing
Key idea: Analyze data collected in the past.
B-tree inserts are slow, but…logging and sorting are fast.

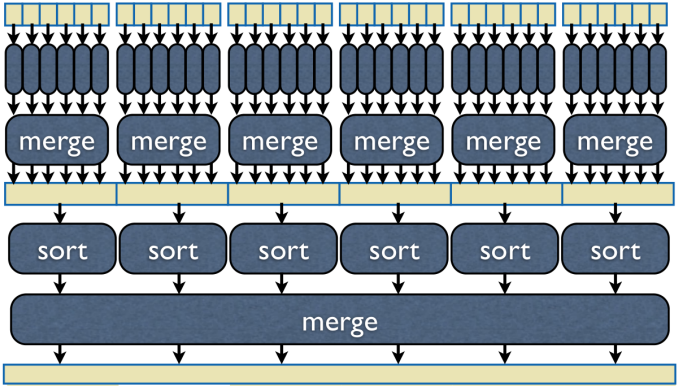
# Write-optimization technique #1: OLAP

OLAP: Online **Analytical** Processing
Key idea: Analyze data collected in the past.
B-tree inserts are slow, but…logging and sorting are fast.

**Plan:** Log new data unsorted, then build indexes in large batches.

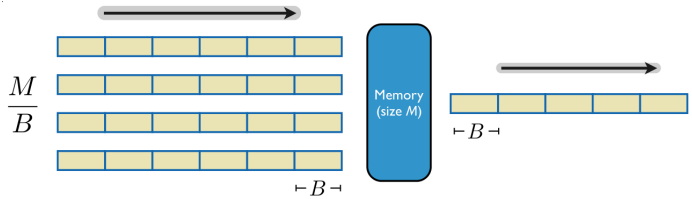
# Write-optimization technique #1: OLAP

Merge sort:



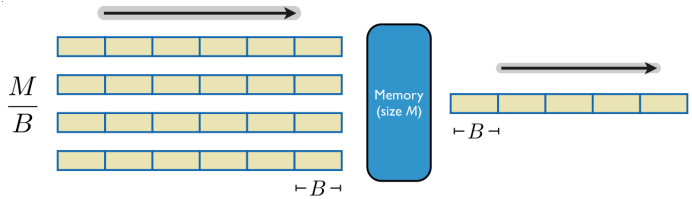
# Write-optimization technique #1: OLAP

Merge sort **in external memory**:



# Write-optimization technique #1: OLAP

Merge sort **in external memory**:

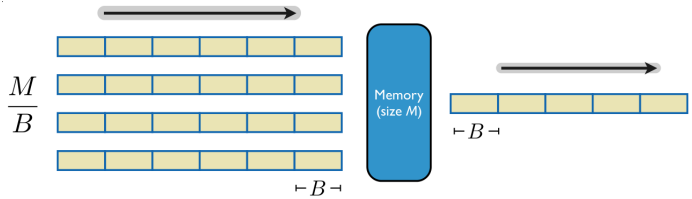



Merge sort cost in DAM model is:

- Cost to scan through all the data once.
- Multiplied by the # of levels in the merge tree.

# Write-optimization technique #1: OLAP

Merge sort **in external memory**:




Merge sort cost in DAM model is:

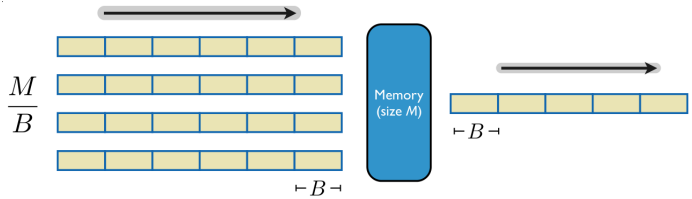
- Cost to scan through all the data once.

  $N/B$

- Multiplied by the # of levels in the merge tree.

# Write-optimization technique #1: OLAP

Merge sort **in external memory**:




Merge sort cost in DAM model is:

- Cost to scan through all the data once.

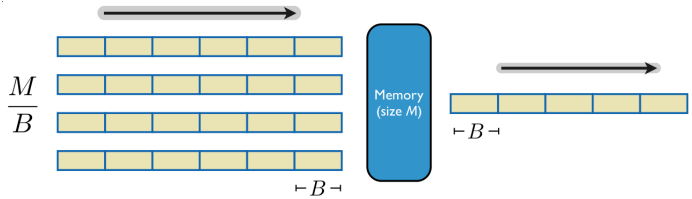
  $N/B$

- Multiplied by the # of levels in the merge tree.

  $\log_{M/B} N/B$

# Write-optimization technique #1: OLAP

Merge sort **in external memory**:




Merge sort cost in DAM model is:

- Cost to scan through all the data once.

  $N/B$

- Multiplied by the # of levels in the merge tree.

  $\log_{M/B} N/B$

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

# Write-optimization technique #1: OLAP

Insert $N$ elements into a B-tree:

$$O\left(N \log_B \frac{N}{M}\right)$$

Merge sort:

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

# Write-optimization technique #1: OLAP

Insert $N$ elements into a B-tree:

$$O\left(N \log_B \frac{N}{M}\right)$$

Merge sort:

$$O\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right) \approx \frac{2N}{B}$$

Typically, $M/B$ is large, so only two passes are needed to sort.

# Write-optimization technique #1: OLAP

Insert $N$ elements into a B-tree:

$$O\left(N \log_B \frac{N}{M}\right) \geq N$$

Merge sort:

$$O\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right) \approx \frac{2N}{B}$$

Typically, $M/B$ is large, so only two passes are needed to sort.
Intuition: Each insert into a B-tree costs ~1 seek, while sorting is close to disk bandwidth.

# Write-optimization technique #1: OLAP

Insert $N$ elements into a B-tree: (assuming 100-1000 byte elements)

$$O\left(N \log_B \frac{N}{M}\right) \geq N \approx 10 - 100\text{kB/s} = 100 \text{ elements/s}$$

Merge sort:

$$O\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right) \approx \frac{2N}{B}$$

Typically, $M/B$ is large, so only two passes are needed to sort.
Intuition: Each insert into a B-tree costs ~1 seek, while sorting is close to disk bandwidth.

# Write-optimization technique #1: OLAP

Insert $N$ elements into a B-tree: (assuming 100-1000 byte elements)

$$O\left(N \log_B \frac{N}{M}\right) \geq N \approx 10 - 100\text{kB/s} = 100 \text{ elements/s}$$

Merge sort:

$$O\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right) \approx \frac{2N}{B} \approx 50\text{MB/s} = 50\text{k} - 500\text{k elements/s}$$

Typically, $M/B$ is large, so only two passes are needed to sort.
Intuition: Each insert into a B-tree costs ~1 seek, while sorting is close to disk bandwidth.

# Write-optimization technique #1: OLAP

So, how does OLAP work?

# Write-optimization technique #1: OLAP

So, how does OLAP work?

- Log new data unindexed until you accumulate a lot of it (∼10% of the data set).

# Write-optimization technique #1: OLAP

So, how does OLAP work?

- Log new data unindexed until you accumulate a lot of it (∼10% of the data set).
- Sort the new data.

# Write-optimization technique #1: OLAP

So, how does OLAP work?

- Log new data unindexed until you accumulate a lot of it (∼10% of the data set).
- Sort the new data.
- Use a merge pass through existing indexes to incorporate new data.

# Write-optimization technique #1: OLAP

So, how does OLAP work?

- Log new data unindexed until you accumulate a lot of it (∼10% of the data set).
- Sort the new data.
- Use a merge pass through existing indexes to incorporate new data.
- Use indexes to do analytics.

# Write-optimization technique #1: OLAP

So, how does OLAP work?

- Log new data unindexed until you accumulate a lot of it (∼10% of the data set).
- Sort the new data.
- Use a merge pass through existing indexes to incorporate new data.
- Use indexes to do analytics.

Moral: OLAP techniques can handle high insertion volume, but query results are *delayed*.

# Write-optimization technique #1: OLAP

So, how does OLAP work?

- Log new data unindexed until you accumulate a lot of it (~10% of the data set).
- Sort the new data.
- Use a merge pass through existing indexes to incorporate new data.
- Use indexes to do analytics.

Moral: OLAP techniques can handle high insertion volume, but query results are *delayed*.

# LSM-trees

# Write-optimization technique #2: LSM-trees

The insight for LSM-trees starts by asking: how can we reduce the queryability delay in OLAP?

# Write-optimization technique #2: LSM-trees

The insight for LSM-trees starts by asking: how can we reduce the queryability delay in OLAP? The buffer is small, let's index it!

# Write-optimization technique #2: LSM-trees

The insight for LSM-trees starts by asking: how can we reduce the queryability delay in OLAP? The buffer is small, let's index it!

- Inserts go into the "buffer B-tree".
- When the buffer gets full, we merge it with the "main B-tree".

Queries have to touch both trees and merge results, but results are available immediately.

# Write-optimization technique #2: LSM-trees

The insight for LSM-trees starts by asking: how can we reduce the queryability delay in OLAP? The buffer is small, let's index it!

- Inserts go into the "buffer B-tree".
- When the buffer gets full, we merge it with the "main B-tree".

Queries have to touch both trees and merge results, but results are available immediately.

(This specific technique (which is not yet an LSM-tree) is used in InnoDB and is called the "change buffer".)

# Write-optimization technique #2: LSM-trees

Why is this fast?

# Write-optimization technique #2: LSM-trees

Why is this fast?

- The buffer is in-memory, so inserts are fast.
- When we merge, we put many new elements in each leaf in the main B-tree (this *amortizes* the I/O cost to read the leaf).

# Write-optimization technique #2: LSM-trees

Why is this fast?

- The buffer is in-memory, so inserts are fast.
- When we merge, we put many new elements in each leaf in the main B-tree (this *amortizes* the I/O cost to read the leaf).

Eventually, we reach a problem:

# Write-optimization technique #2: LSM-trees

Why is this fast?

- The buffer is in-memory, so inserts are fast.
- When we merge, we put many new elements in each leaf in the main B-tree (this *amortizes* the I/O cost to read the leaf).

Eventually, we reach a problem:

- If the buffer gets too big, inserts get slow.
- If the buffer stays too small, the merge gets inefficient (back to $O(N \log_B N)$).

# Write-optimization technique #2: LSM-trees

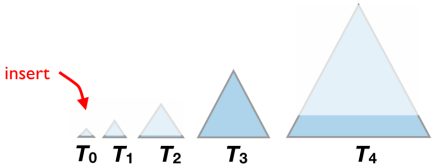
How can we fix this?

# Write-optimization technique #2: LSM-trees

How can we fix this? More buffering!

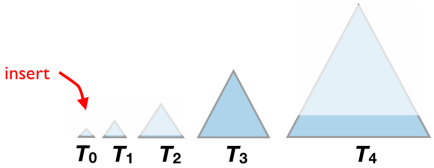
# Write-optimization technique #2: LSM-trees

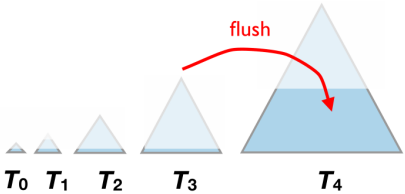
How can we fix this? More buffering!



# Write-optimization technique #2: LSM-trees

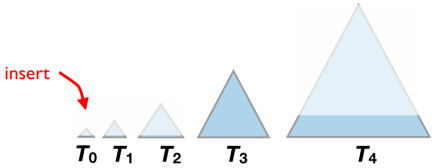
How can we fix this? More buffering!

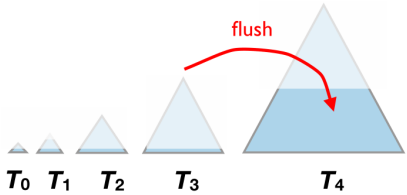


# Write-optimization technique #2: LSM-trees

How can we fix this? More buffering!

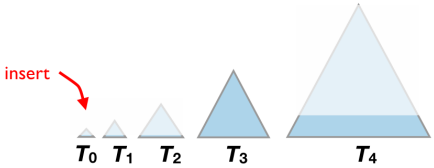


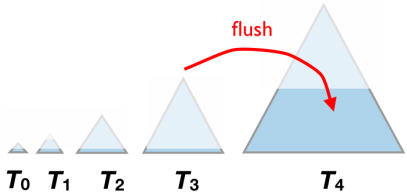




Each level is twice as large as the previous level, for some value of 2.

# Write-optimization technique #2: LSM-trees

How can we fix this? More buffering!



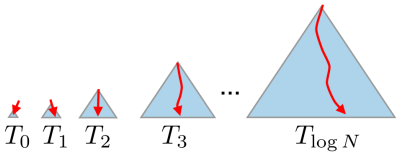

Each level is twice as large as the previous level, for some value of 2. We'll use 2.

# Write-optimization technique #2: LSM-trees

How do queries work?

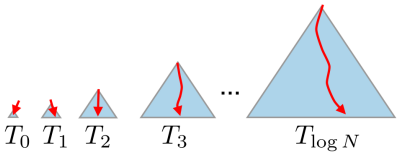
# Write-optimization technique #2: LSM-trees

How do queries work?



# Write-optimization technique #2: LSM-trees

How do queries work?

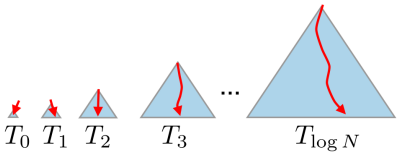



Search cost is:

$$\log_B B + \ldots + \log_B \frac{N}{8} + \log_B \frac{N}{4} + \log_B \frac{N}{2} + \log_B N$$

# Write-optimization technique #2: LSM-trees

How do queries work?



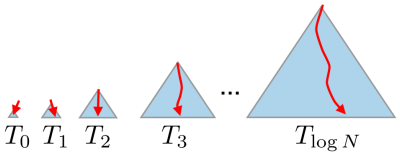

Search cost is:

$$\log_B B + \ldots + \log_B \frac{N}{8} + \log_B \frac{N}{4} + \log_B \frac{N}{2} + \log_B N$$

$$= \frac{1}{\log B}(1 + \ldots + \lg(N) - 3 + \lg(N) - 2 + \lg(N) - 1 + \lg(N))$$

# Write-optimization technique #2: LSM-trees

How do queries work?



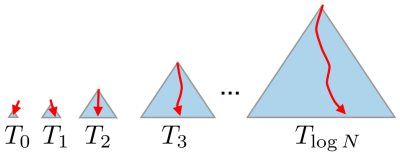

Search cost is:

$$\log_B B + \ldots + \log_B \frac{N}{8} + \log_B \frac{N}{4} + \log_B \frac{N}{2} + \log_B N$$

$$= \frac{1}{\log B}\left(1 + \ldots + \lg(N) - 3 + \lg(N) - 2 + \lg(N) - 1 + \lg(N)\right)$$

# Write-optimization technique #2: LSM-trees

How do queries work?




Search cost is:

$$\log_B B + \ldots + \log_B \frac{N}{8} + \log_B \frac{N}{4} + \log_B \frac{N}{2} + \log_B N$$

$$= \frac{1}{\log B}\left(1 + \ldots + \lg(N) - 3 + \lg(N) - 2 + \lg(N) - 1 + \lg(N)\right) = O(\log N \cdot \log_B N)$$

# Write-optimization technique #2: LSM-trees

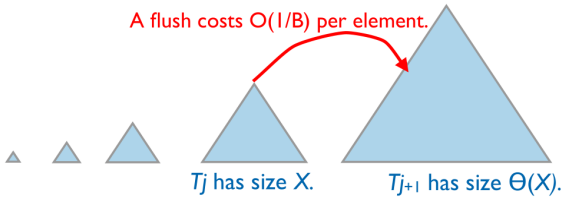
How much do inserts cost?

# Write-optimization technique #2: LSM-trees

How much do inserts cost?
Cost to flush a tree $T_j$ of size $X$ is $O(X/B)$.

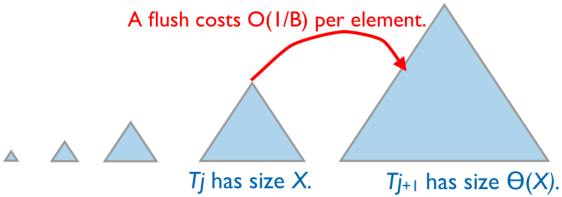
# Write-optimization technique #2: LSM-trees

How much do inserts cost?
Cost to flush a tree $T_j$ of size $X$ is $O(X/B)$.
Cost per element to flush $T_j$ is $O(1/B)$.



# Write-optimization technique #2: LSM-trees

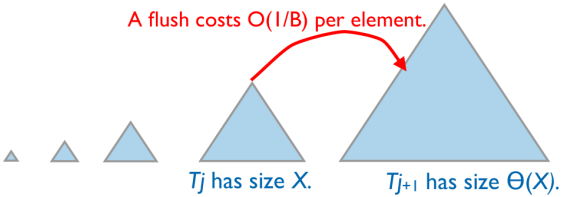
How much do inserts cost?
Cost to flush a tree $T_j$ of size $X$ is $O(X/B)$.
Cost per element to flush $T_j$ is $O(1/B)$.




Each element moves $\leq \log N$ times.

# Write-optimization technique #2: LSM-trees

How much do inserts cost?
Cost to flush a tree $T_j$ of size $X$ is $O(X/B)$.
Cost per element to flush $T_j$ is $O(1/B)$.




Each element moves $\leq \log N$ times.
Total amortized insert cost per element is $O\left(\frac{\log N}{B}\right)$.

# Fractal Trees

# Write-optimization technique #3: Fractal Trees

The pain in LSM-trees is doing a full $O(\log_B N)$ search in each level.

# Write-optimization technique #3: Fractal Trees

The pain in LSM-trees is doing a full $O(\log_B N)$ search in each level.
We use *fractional cascading* to reduce the search per level to $O(1)$.

# Write-optimization technique #3: Fractal Trees

The pain in LSM-trees is doing a full $O(\log_B N)$ search in each level.
We use *fractional cascading* to reduce the search per level to $O(1)$.

The idea is that once we've searched $T_i$, we know where the key would be in $T_i$, and we can use that information to guide our search of $T_{i+1}$.

# Write-optimization technique #3: Fractal Trees

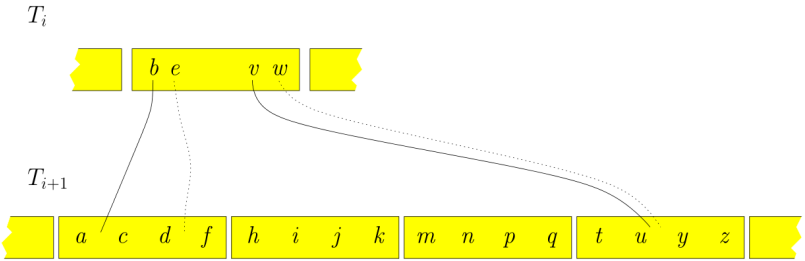
The pain in LSM-trees is doing a full $O(\log_B N)$ search in each level.
We use *fractional cascading* to reduce the search per level to $O(1)$.

The idea is that once we've searched $T_i$, we know where the key would be in $T_i$, and we can use that information to guide our search of $T_{i+1}$.

Let's examine the leaves of two consecutive levels of the LSM-tree…

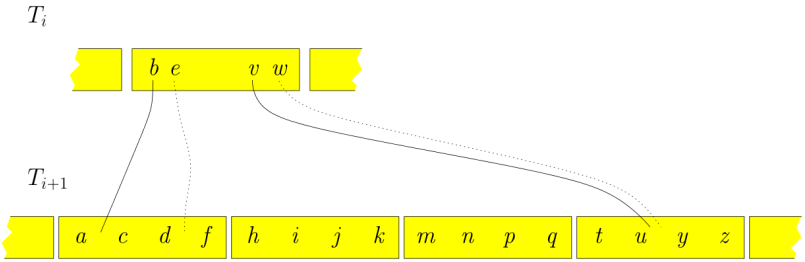
# Write-optimization technique #3: Fractal Trees

Add *forwarding pointers* from leaves in $T_i$ to leaves in $T_{i+1}$ (but remove the redundant ones that point to the same leaf):



# Write-optimization technique #3: Fractal Trees

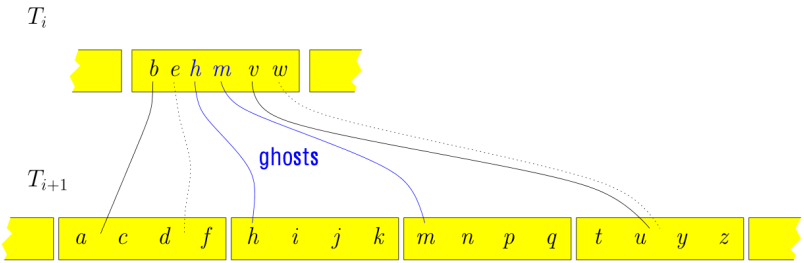
Add *forwarding pointers* from leaves in $T_i$ to leaves in $T_{i+1}$ (but remove the redundant ones that point to the same leaf):




Now, from a leaf node in $T_i$, we can jump forward to some of the leaves in $T_{i+1}$ without searching the whole tree at $T_{i+1}$.

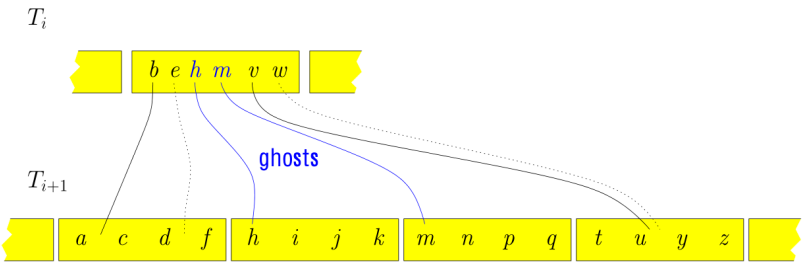
# Write-optimization technique #3: Fractal Trees

Add *ghost pointers* to leaves not pointed to in $T_{i+1}$ in leaves in $T_i$:



# Write-optimization technique #3: Fractal Trees

Add *ghost pointers* to leaves not pointed to in $T_{i+1}$ in leaves in $T_i$:

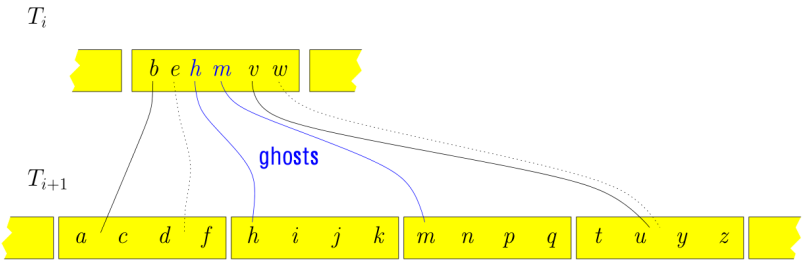



Now *every* leaf in $T_{i+1}$ can be reached by a pointer in a leaf node in $T_i$.

# Write-optimization technique #3: Fractal Trees

[Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, & Nelson '07]

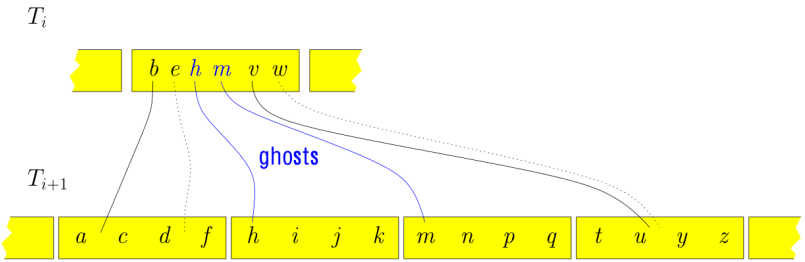
After searching $T_i$ for a missing element $c$, we look left and right for forwarding or ghost pointers, and follow them down to look at $O(1)$ leaves in $T_{i+1}$.



# Write-optimization technique #3: Fractal Trees

[Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, & Nelson '07]

After searching $T_i$ for a missing element $c$, we look left and right for forwarding or ghost pointers, and follow them down to look at $O(1)$ leaves in $T_{i+1}$.




This way, search is only $O(\log_R N)$ (in our example, $R = 2$).

# Write-optimization technique #3: Fractal Trees

[Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, & Nelson '07]

The internal node structure in each level is now redundant, so we can represent each level as an array. We can forget about the B-tree structure above the leaves in each level!
This is called a *Cache-Oblivious Lookahead Array*.

# Write-optimization technique #3: Fractal Trees

The amortized analysis says our inserts are fast, but we flush a very large level to the next one, we might see a big stall. Concurrent merge algorithms exist, but we can do better.

# Write-optimization technique #3: Fractal Trees

The amortized analysis says our inserts are fast, but we flush a very large level to the next one, we might see a big stall. Concurrent merge algorithms exist, but we can do better.

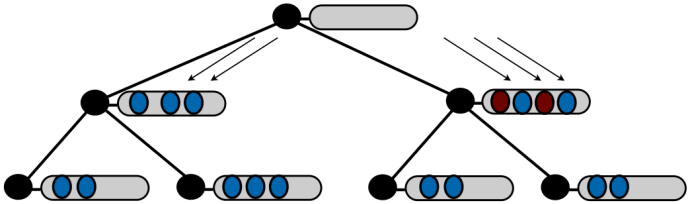
We break each level's array into chunks that can be flushed independently. Each chunk flushes to a small region of a few chunks in the next level down, found using its forwarding pointers.

# Write-optimization technique #3: Fractal Trees

The amortized analysis says our inserts are fast, but we flush a very large level to the next one, we might see a big stall. Concurrent merge algorithms exist, but we can do better.

We break each level's array into chunks that can be flushed independently. Each chunk flushes to a small region of a few chunks in the next level down, found using its forwarding pointers.

Now we have a tree again!



# Write-optimization technique #3: Fractal Trees

Fractal Tree Advantages over COLA:

# Write-optimization technique #3: Fractal Trees

Fractal Tree Advantages over COLA:

1. Easier to manage an LRU cache of blocks.

# Write-optimization technique #3: Fractal Trees

Fractal Tree Advantages over COLA:

1. Easier to manage an LRU cache of blocks.
2. More flexible with “hotspots”, or non-uniform workload distributions.

# Write-optimization technique #3: Fractal Trees

Fractal Tree Advantages over COLA:

1. Easier to manage an LRU cache of blocks.
2. More flexible with “hotspots”, or non-uniform workload distributions.
3. Flushes are $O(1)$, so easier to reduce latency and increase concurrency with client work.

# Write-optimization technique #3: Fractal Trees

Fractal Tree Advantages over COLA:

1. Easier to manage an LRU cache of blocks.
2. More flexible with “hotspots”, or non-uniform workload distributions.
3. Flushes are $O(1)$, so easier to reduce latency and increase concurrency with client work.
4. Easier to implement a concurrent checkpoint algorithm with small flushes.

# Write-optimization technique #3: Fractal Trees

Fractal Tree Advantages over COLA:

1. Easier to manage an LRU cache of blocks.
2. More flexible with “hotspots”, or non-uniform workload distributions.
3. Flushes are $O(1)$, so easier to reduce latency and increase concurrency with client work.
4. Easier to implement a concurrent checkpoint algorithm with small flushes.
5. Enables good tradeoffs for queries, and allows that computation to be cached without inducing I/O (this is enough complexity for a whole other talk).

# Results

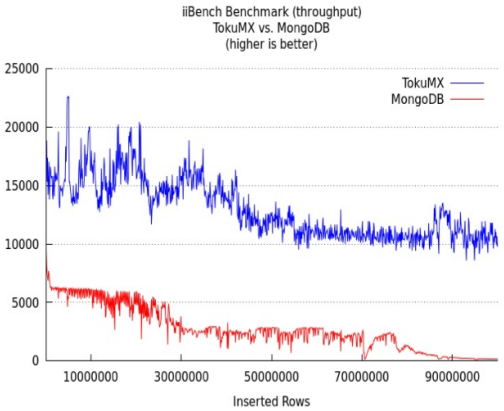
- Modified B-tree-like dynamic (inserts, updates, deletes) data structure that supports point and range queries.
- Inserts are a factor $B/\log B$ (typically 10-100x in practice) faster than a B-tree: $O\left(\frac{\log N}{B}\right) < O\left(\frac{\log N}{\log B}\right)$.
- Searches are a factor $\log B/\log R$ slower than a B-tree: $O\left(\frac{\log N}{\log R}\right) > O\left(\frac{\log N}{\log B}\right)$.
- To amortize flush costs over many elements, we want each block we write to be large (∼4MB), much larger than typical B-tree blocks (∼16KB). These compress well.

# Applications

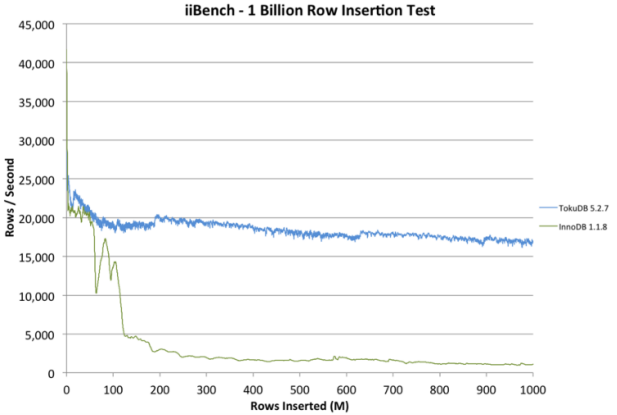
TokuDB for MySQL, TokuMX for MongoDB:

- Faster indexed insertions.
- Hot schema changes.
- Compression.
- Read-free replication on secondaries.
- Fast (no read before write) updates with messages in buffers.
- ACID transactions.
- Mixed workload concurrency.
- Faster sharding migrations (TokuMX).

# Benchmarks

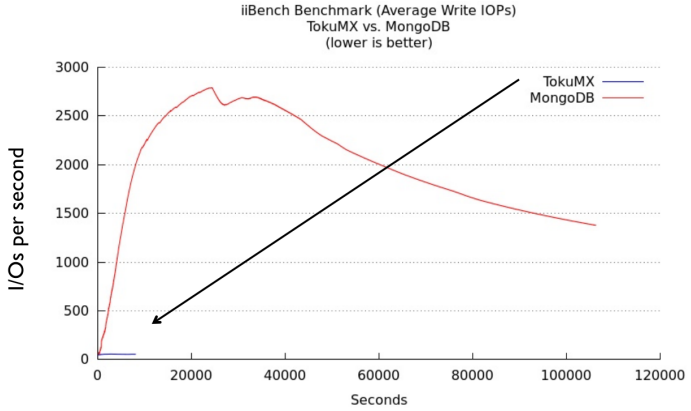



MongoDB



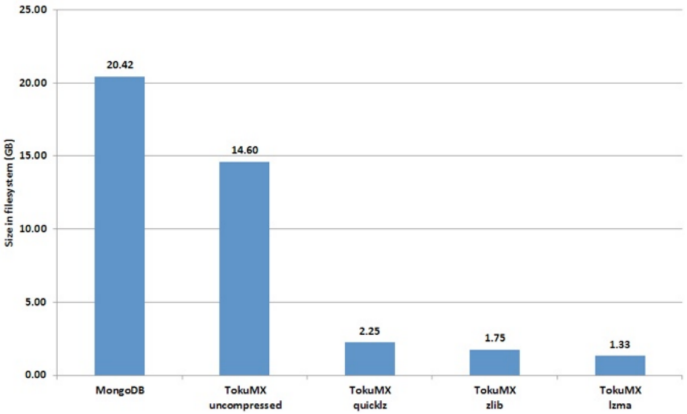

MySQL

# Benchmarks




100M inserts into a collection with 3 secondary indexes

# Benchmarks


25.00
20.00
15.00
10.00
5.00
0.00
Size in filesystem (GB)
20.42
14.60
2.25
1.75
1.33
MongoDB
TokuMX uncompressed
TokuMX quicklz
TokuMX zlib
TokuMX lzma

# Questions?

Leif Walsh
@leifwalsh
Downloads: www.tokutek.com/downloads
Docs: docs.tokutek.com
Slides: bit.ly/1au1uvr