The Right Read Optimization is Actually Write Optimization

Leif Walsh
leif@tokutek.com

Tokutek®

The Right Read Optimization is Write Optimization

Situation: I have some data.

- I want to learn things about the world, so I put it in MySQL, and start querying it.
- To learn more, I go out and get more data.

New Situation: I have a *lot* of data.

- My queries start to **slow down**, and I can't run them all.
  - I also happen to still be collecting data.

Goal: Execute queries in real time against large, growing data sets.

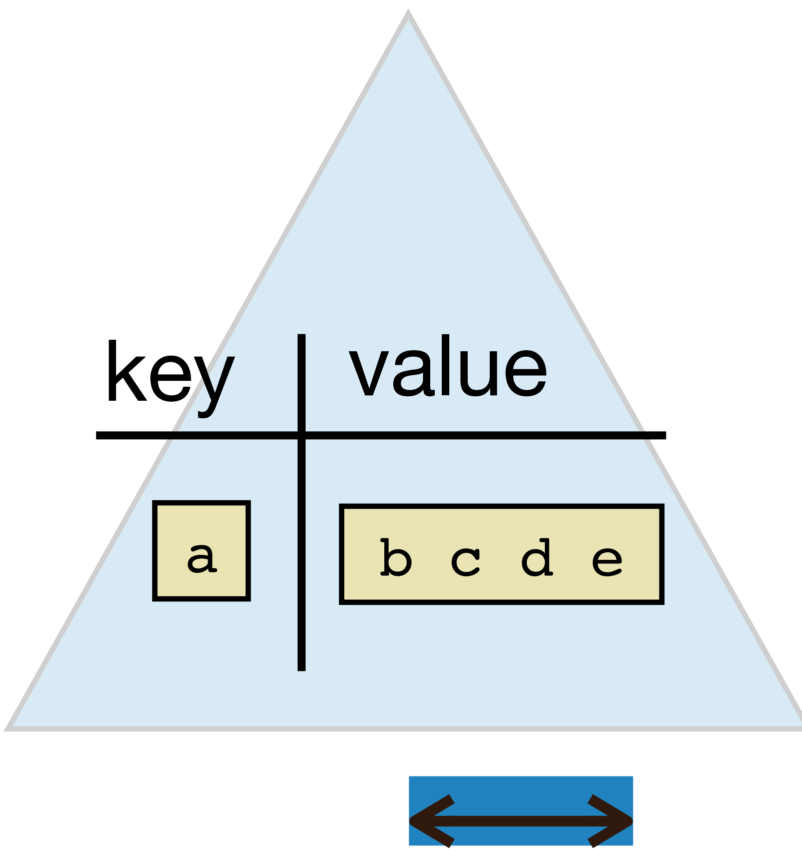
- We need to do some read optimization.

Let's see some ways to optimize reads.

The Right **Read Optimization** is Write Optimization

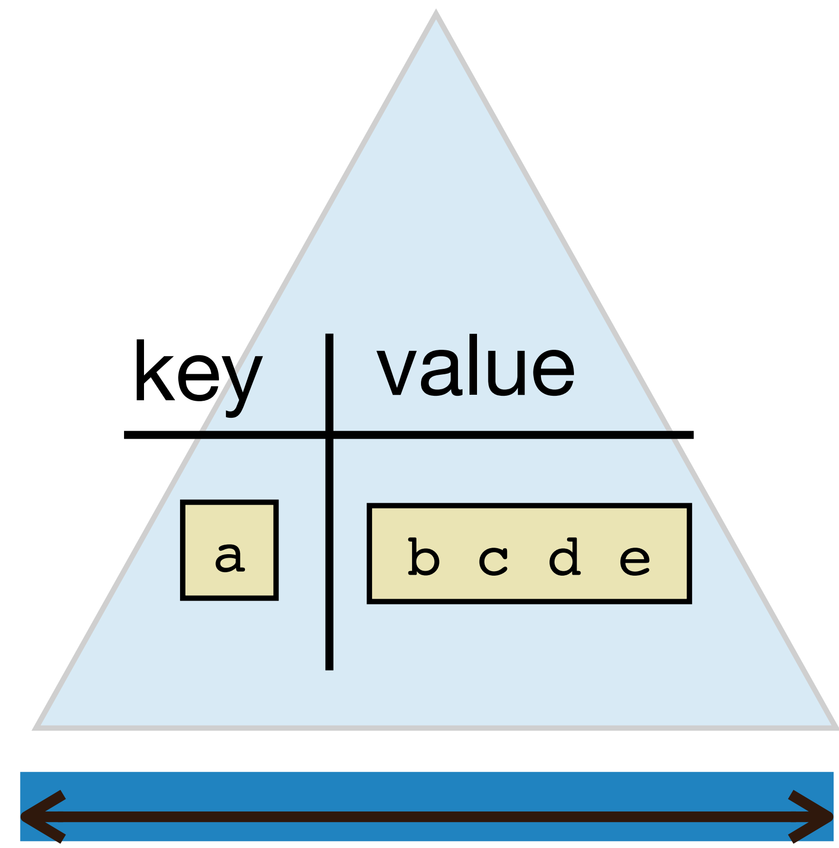
Select via Index

`select d where 270 ≤ a ≤ 538`



Select via Table Scan

`select d where 270 ≤ e ≤ 538`

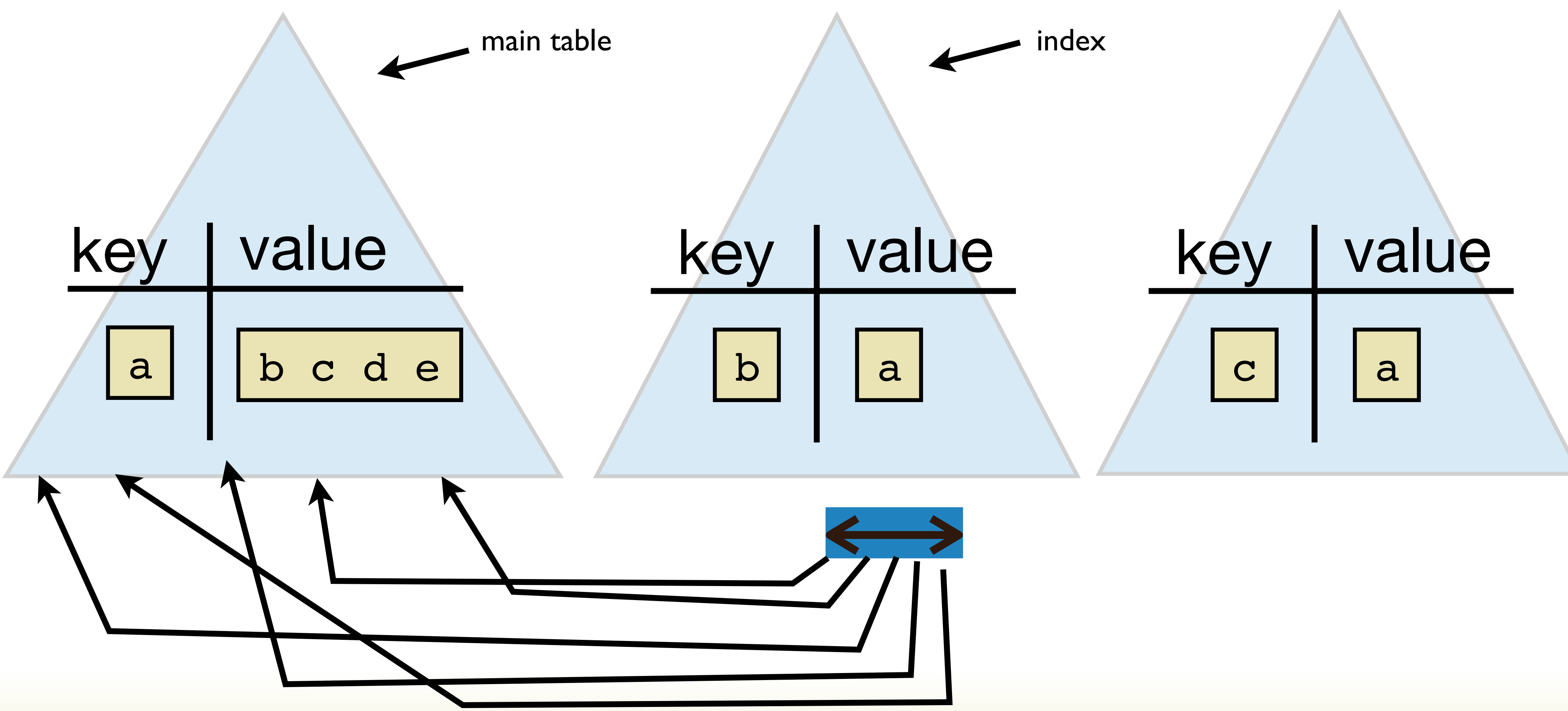


An index with the right key lets you examine less data.

The Right **Read Optimization** is Write Optimization

Selecting via an index can be slow, if it is coupled with point queries.

```
select d where 270 ≤ b ≤ 538
```

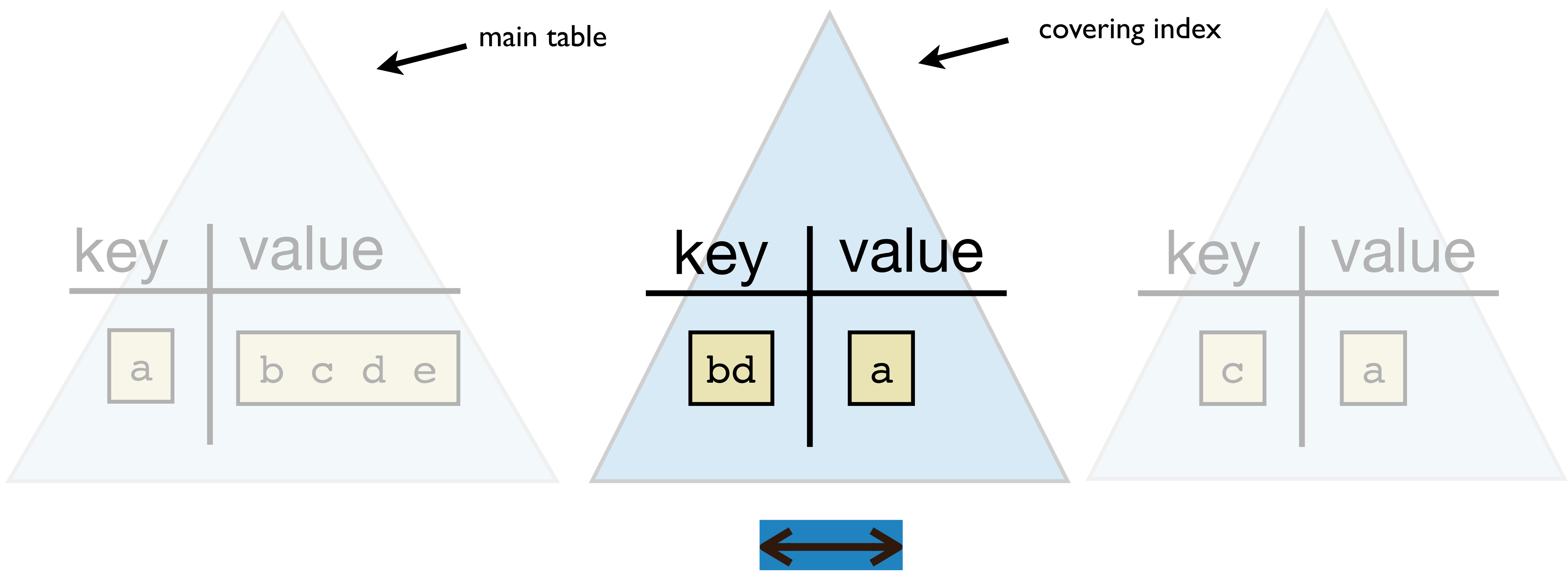


The Right **Read Optimization** is Write Optimization

Covering indexes can speed up queries.

- Key contains all columns necessary to answer query.

```
select d where 270 ≤ b ≤ 538
```



No need to do point queries if you have a covering index.

The Right **Read Optimization** is Write Optimization

Indexes do read optimization.

- Index instead of table scan.
- Covering indexing instead of regular indexing.
- See Zardosht's "Understanding Indexing" talk for more.
  - Avoid post-retrieval sorting in GROUP BY and ORDER BY queries.
  - http://vimeo.com/26454091

Queries run much faster with the proper indexes.

The right read optimization is good indexing!

- But, different queries need different indexes.
- Typically you need lots of indexes for a single table.

Optimizing reads with indexes slows down insertions.

The Right Read Optimization is **Write Optimization**

The case for write optimization is indexed insertion performance.

- “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, they 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 Lewis Carroll

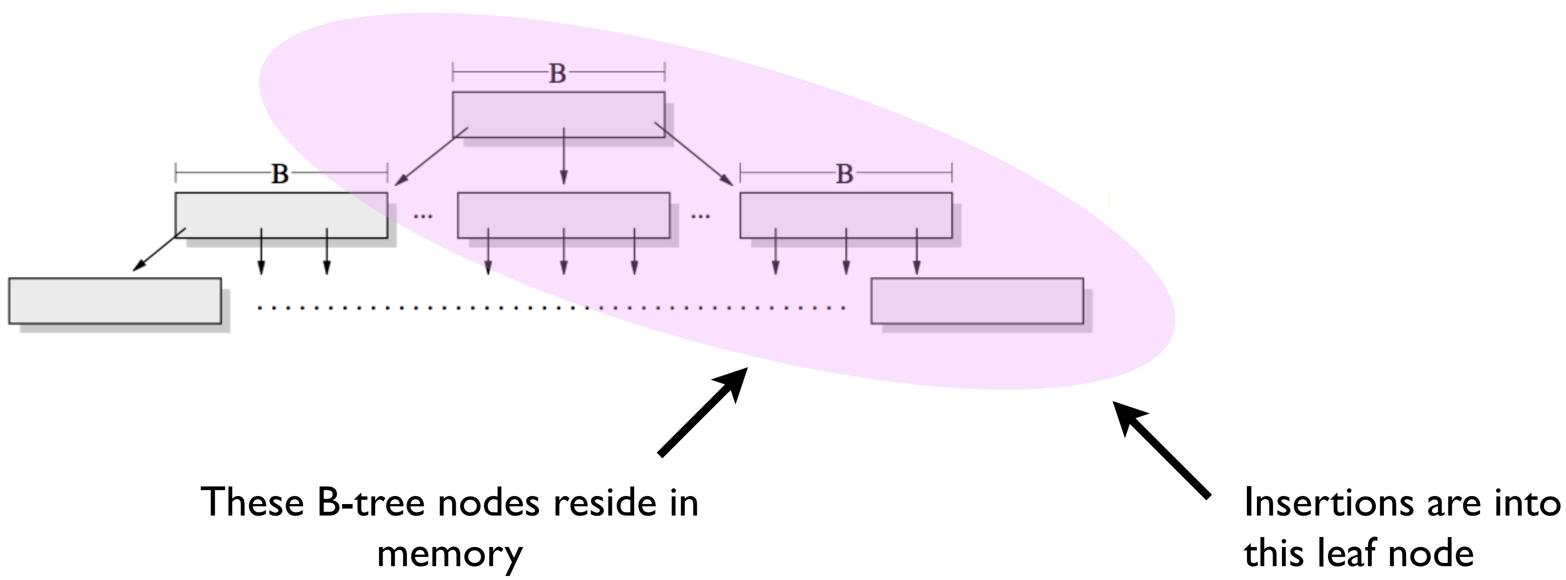
Now, our problem is to optimize writes.

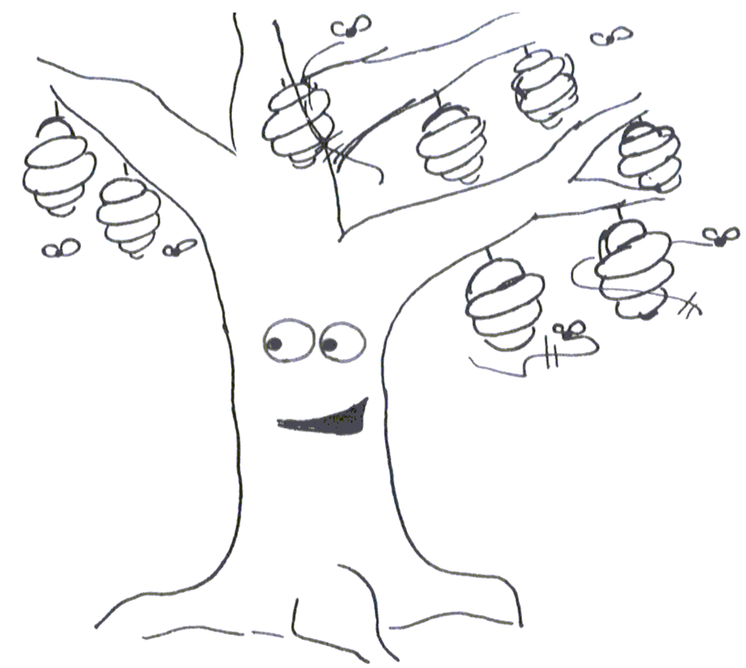
- We need to understand how writes work in indexes.

B-tree Basics

B-trees are Fast at Sequential Inserts

Sequential inserts in B-trees have near-optimal data locality.

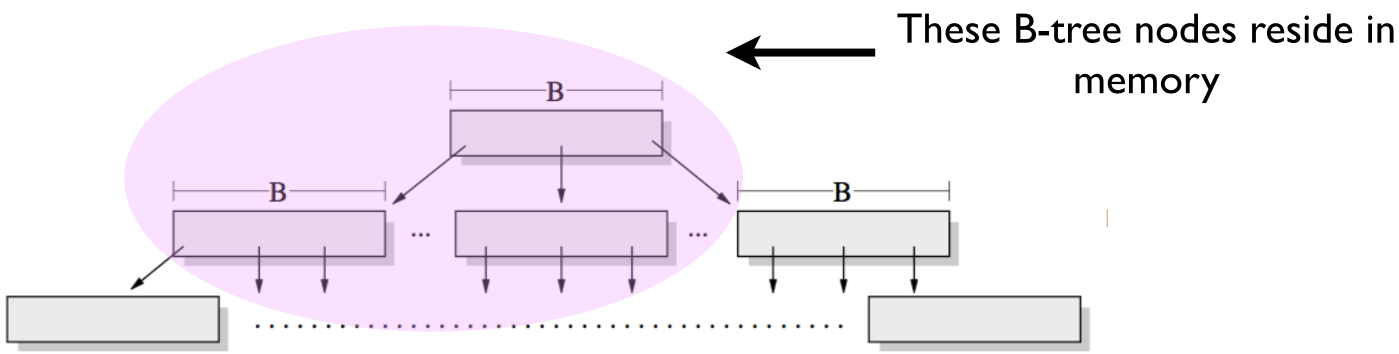




- One disk I/O per leaf (which contains many inserts).
- Sequential disk I/O.
- Performance is disk-bandwidth limited.

B-Trees Are Slow at Ad Hoc Inserts

High entropy inserts (e.g., random) in B-trees have poor data locality.



- Most nodes are not in main memory.
- Most insertions require a random disk I/O.
- Performance is disk-seek limited.
- ≤ 100 inserts/sec/disk (≤ 0.05% of disk bandwidth).

Good Indexing is Hard With B-trees

With multiple indexes, B-tree indexes are slow.

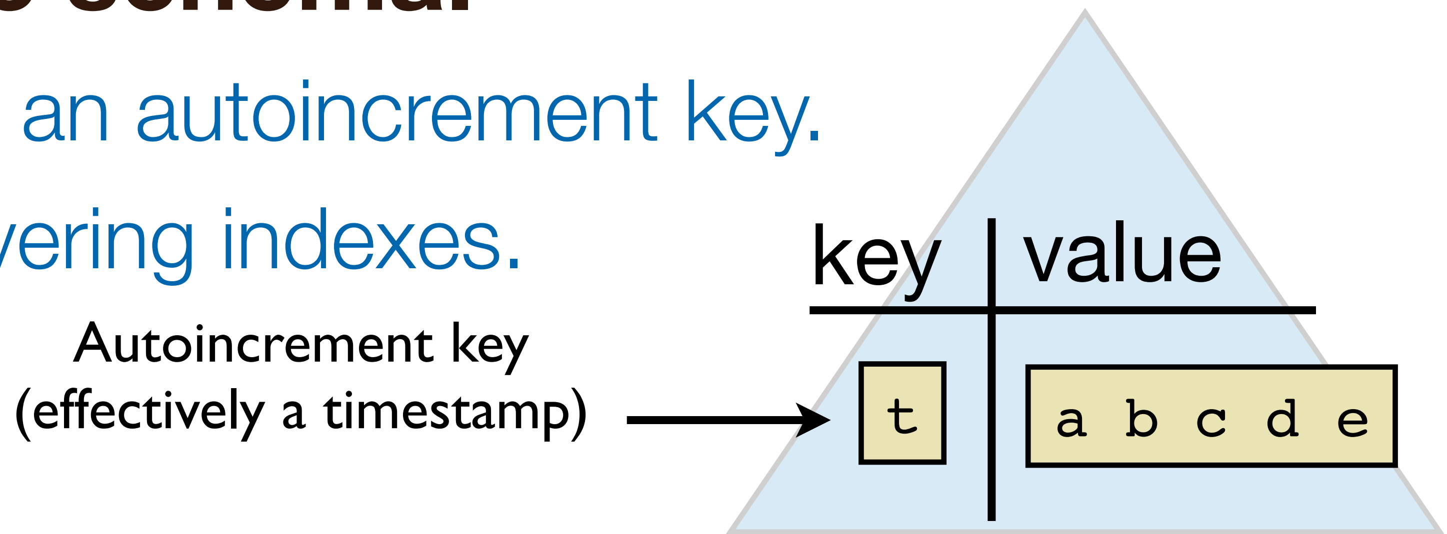
- Secondary indexes are not built sequentially.
  - If they have the same sort order as the primary key, why bother storing them?
- For read optimization, we would like multiple secondary indexes per table.
- So inserts become multiple random B-tree insertions.
- That’s slow, so we can’t keep up with incoming data.

We can’t run queries well without good indexes, but we can’t keep good indexes in B-trees.

The Right Read Optimization is **Write Optimization**

People often don't use enough indexes.
They use simplistic schema.

- Sequential inserts via an autoincrement key.
- Few indexes, few covering indexes.



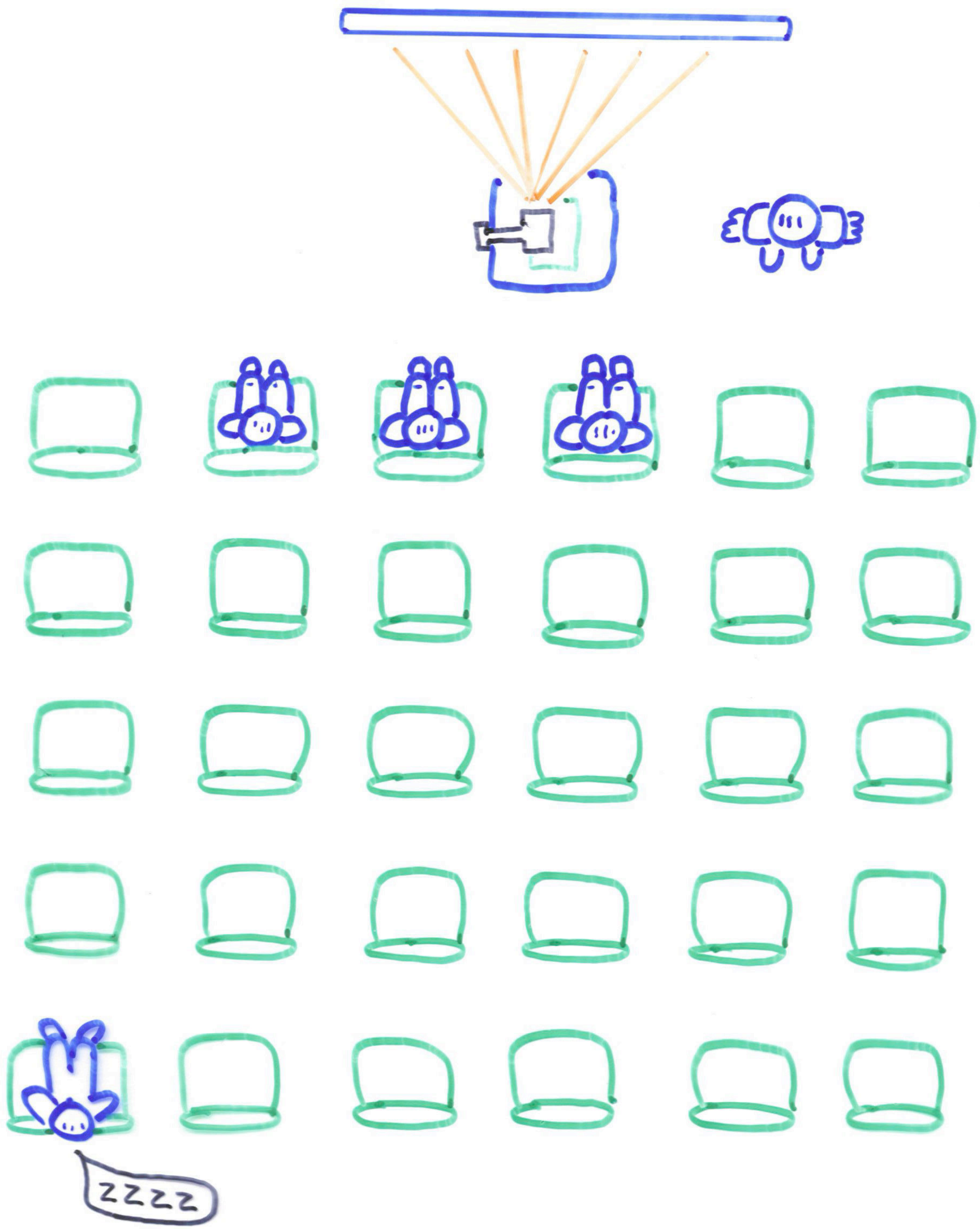
Then insertions are fast but queries are slow.

Adding sophisticated indexes helps queries.

- B-trees cannot afford to maintain them.

If we speed up inserts, we can maintain the right indexes, and speed up queries.

Overview of Talk



The Right Read Optimization is Write Optimization

Read Optimization Techniques

Write Optimization is Necessary for Read Optimization

Write Optimization Techniques

- Insert Batching
  - OLAP
- Bureaucratic Insert Batching
  - LSM Trees
- How the Post Office Does Write Optimization
  - Fractal Trees

Reformulating The Problem

Random insertions into a B-tree are slow because:

- Disk seeks are very slow.
- B-trees incur a disk seek for every insert.

Here is another way to think about it:

- B-trees only accomplish one insert per disk seek.

A simpler problem:

- Can we get B-trees to do more useful work per disk seek?

Insert Batching

Insert Batching

Recall that sequential insertions are faster than random insertions.

- The argument before holds for empty trees.
- But even for existing trees, you can bunch of a set of insertions (say, a day's worth) and:
  - Sort them
  - Insert them in sorted order
- Inserting batches in sorted order is faster when you end up with multiple insertions in the same leaf.
- This happens a lot in practice, so batch-sort-and-insert is standard practice.

Insert Batching Example

Here's a typical B-tree scenario:

- 1 billion 160-byte rows = 160GB
- 16KB page size
- 16GB main memory available

That means:

- Each leaf contains 100 rows.
- There are 10 million leaves.
- At most (16GB / 160GB) = 10% of the leaves fit in RAM.
  - So most leaf accesses require a disk seek.

Insert Batching Example

Back of the envelope analysis:

- Let's batch 16GB of data (100 million rows).
  - Then sort them and insert them into the B-tree.
- That's 10% of our total data size, and each leaf has 100 rows, so each leaf has about 10 row modifications headed for it.
- Each disk seek accomplishes 10 inserts (instead of just one).
- So we get about 10x throughput.

But we had to batch a lot of rows to get there.

- Since these are stored unindexed on disk, we can't query them.
- If we had 10 billion rows (1.6TB), we would have had to save 1 billion inserts just to get 10x insertion speed.

Insert Batching Results

OLAP is insert batching.

- The key is to batch a constant fraction of your DB size.
  - Otherwise, the math doesn't work out right.

Advantages

- Get plenty of throughput from a very simple idea.
  - 10x in our example, more if you have bigger leaves.

Disadvantages

- Data latency: data arrives for insertion, but isn't available to queries until the batch is inserted.
  - The bigger the DB, the bigger the batches need to be, and the more latency you experience.

Learning From OLAP's Disadvantages

We got latency because:

- Our data didn't get indexed right away, it just sat on disk.
- Without an index, we can't query that data.

We could index the buffer.

- But we need to make sure we don't lose the speed boost.

Learning From OLAP’s Disadvantages

Let’s try it:

- One main B-tree on disk.
- Another smaller B-tree, as the buffer.
  - Maximum size is a constant fraction of the main B-tree’s size.
- Inserts go first to the small B-tree.
- When the small B-tree is big enough, merge it with the larger B-tree.
- Queries need to be done on both trees, but at least all the data can be queried immediately.

It looks like we solved the latency problem.

If At First You Don't Succeed, Recurse

We didn't maintain our speed boost.

- At first, the smaller B-tree fits in memory, so inserts are fast.
- When your DB grows, the smaller tree must grow too.
  - Otherwise, you lose the benefit of batching – remember, you need a constant fraction like 10%.
- Eventually, even the small B-tree is too big for memory.
- Now we can't insert into the small B-tree fast enough.

Try the same trick again:

- Stick an insert buffer in front of the small B-tree.
- But now you get latency, so index the new buffer.
- ...

This brings us to our next write optimization.

LSM Trees

LSM Trees

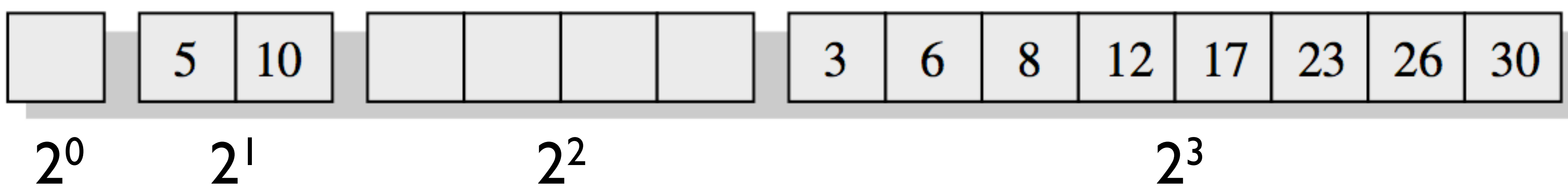
Generalizing the OLAP technique:

- Maintain a hierarchy of B-trees: B_0, B_1, B_2, ...
  - B_k is the insert buffer for B_{k+1}.
- The maximum size of B_{k+1} is twice that of B_k.
  - “Twice” is a simple choice but it’s not fixed.
- When B_k gets full, merge it down to B_{k+1}, and empty B_k.
- These merges can cascade down multiple levels.

This is called a Log-Structured Merge Tree.

Visualizing the LSM Tree

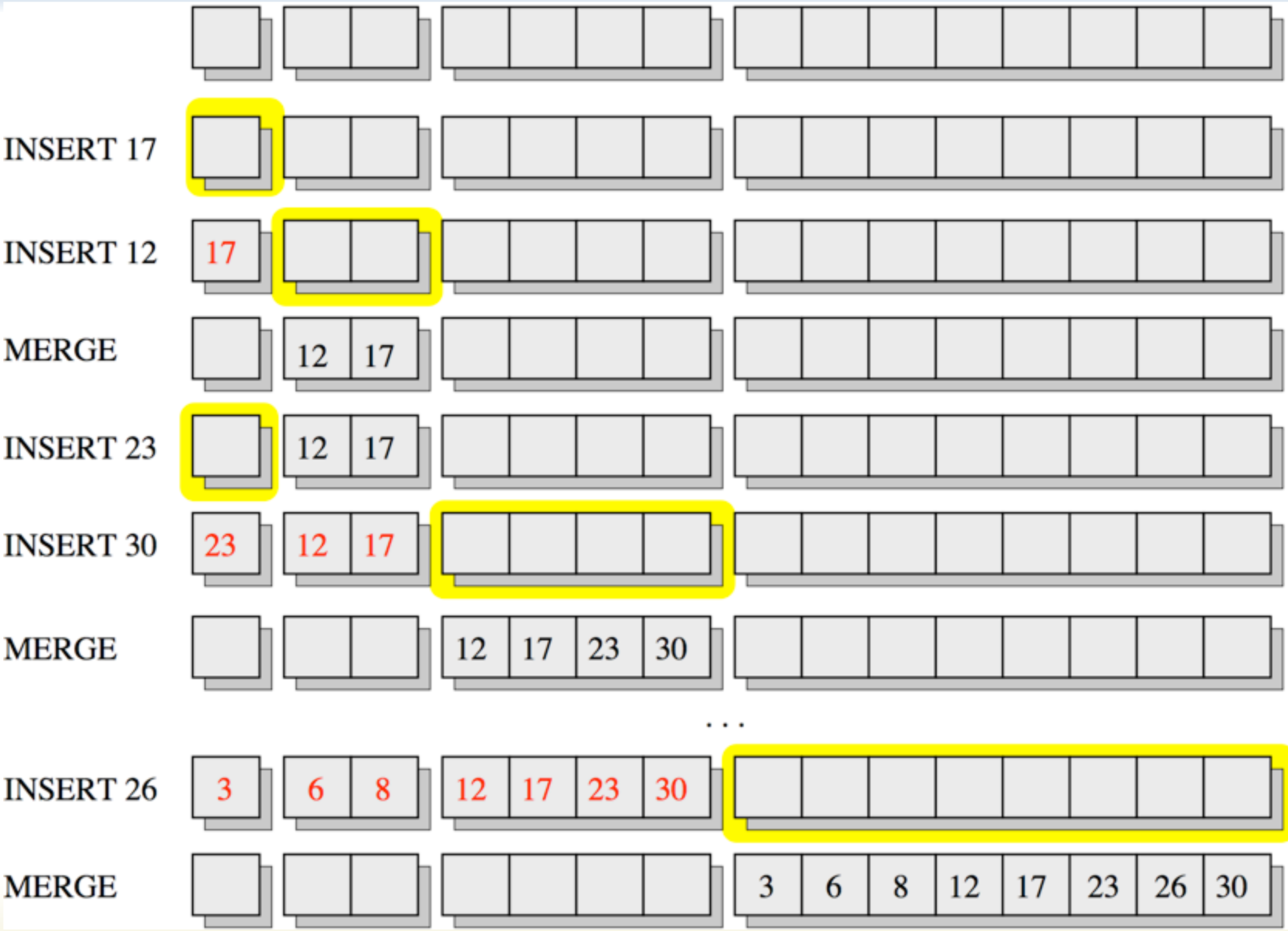
- B-trees are a bit like arrays, the way we use them here.
  - If we simplify things a tiny bit, all we do is merge B-trees, which is fast.
  - Merging sorted arrays is fast too (mergesort uses this).



- B_k's maximum size is 2^k.
- The first few levels* are just in memory.

* If memory size is M, that's $\log_2(M)$ levels

LSM Tree Demonstration



LSM Tree Insertion Performance

LSM Trees use I/O efficiently.

- Each merge is 50% of the receiving tree's size.
- So each disk seek done during a merge accomplishes half as many inserts as fit in a page (that's a lot).
  - In our earlier example, that's 50 inserts per disk seek.
- But there are $\log_2(n) - \log_2(M)$ levels on disk, so each insert needs to get written that many times.
  - That would be ~3 times.
- Overall, we win because the boost we get from batching our inserts well overwhelms the pain of writing data multiple times.
  - Our database would get about a 16x throughput boost.

LSM Trees have very good insertion performance.

LSM Tree Query Performance

LSM Trees do a full B-tree search once per level.

- B-tree searches are pretty fast, but they do incur at least one disk seek.
- LSM trees do lots of searches, and each one costs at least one disk seek.

Queries in LSM trees are much slower than in B-trees.

- Asymptotically, they're a factor of log(n) slower.

LSM Tree Results

Advantages

- Data is available for query immediately.
- Insertions are very fast.

Disadvantages

- Queries take a nasty hit.

LSM trees are almost what we need.

- They can keep up with large data sets with multiple secondary indexes and high insertion rates.
- But the indexes you keep aren't as effective for queries. We lost some of our read optimization.

Fractal Tree® Indexes

Getting the Best of Both Worlds

LSM Trees have one big structure per level.

- But that means you have to do a global search in each level.

B-trees have many smaller structures in each level.

- So on each level, you only do a small amount of work.

A Fractal Tree® Index is the best of both worlds.

- Topologically, it looks like a B-tree, so searches are fast.
- But it also buffers like an LSM Tree, so inserts are fast.

Building a Fractal Tree® Index

Start with a B-tree.

Put an unindexed buffer (of size B) at each node.

- These buffers are small, so they don't introduce data latency.

Insertions go to the root node's buffer.

When a buffer gets full, flush it down the tree.

- Move its elements to the buffers on the child nodes.
- This may cause some child buffers to flush.

Searches look at each buffer going to a leaf.

- But they can ignore all the rest of the data at that depth in the tree.

Fractal Tree® Index Insertion Performance

Cost to flush a buffer: O(1).

Cost to flush a buffer, *per element*: O(1/B).

- We move B elements when we flush a buffer.

# of flushes per element: O(log(N)).

- That's just the height of the tree – when the element gets to a leaf node, it's done moving.

Cost to flush an element all the way down: O(log(N)) * O(1/B) = O(log(N) / B).

- (Full cost to insert an element)
- By comparison, B-tree insertions are $O(\log_B(N)) = O(\log(N) / \log(B))$.

Fractal Tree Indexes have very good insertion performance.

- As good as LSM Trees.

Fractal Tree® Index Query Performance

Fractal Tree searches are the same as B-tree searches.

- Takes a little more CPU to look at the buffers, but the same # of disk seeks.
  - There are some choices to make here, about caching and expected workloads, but they don't affect the asymptotic performance.

So Fractal Trees have great query performance.

Fractal Tree® Index Results

Advantages

- Insertion performance is great.
  - We can keep all the indexes we need.
- Query performance is great.
  - Our indexes are as effective as they would be with B-trees.

Disadvantages

- Introduces more dependence between tree nodes.
  - Concurrency is harder.
- Insert/search imbalance: inserts are a lot cheaper than searches, only as long as inserts don't require a search first.
  - Watch out for uniqueness checks.

Other benefits

- Can afford to increase the block size.
  - Better compression, no fragmentation.
- Can play tricks with "messages" that update multiple rows.
  - HCAD, HI, HOT (online DDL).

Thanks!

Come see our booth and our lightning talk

leif@tokutek.com

Tokutek®