Databases & External Memory
Indexes, Write Optimization, and Crypto-searches

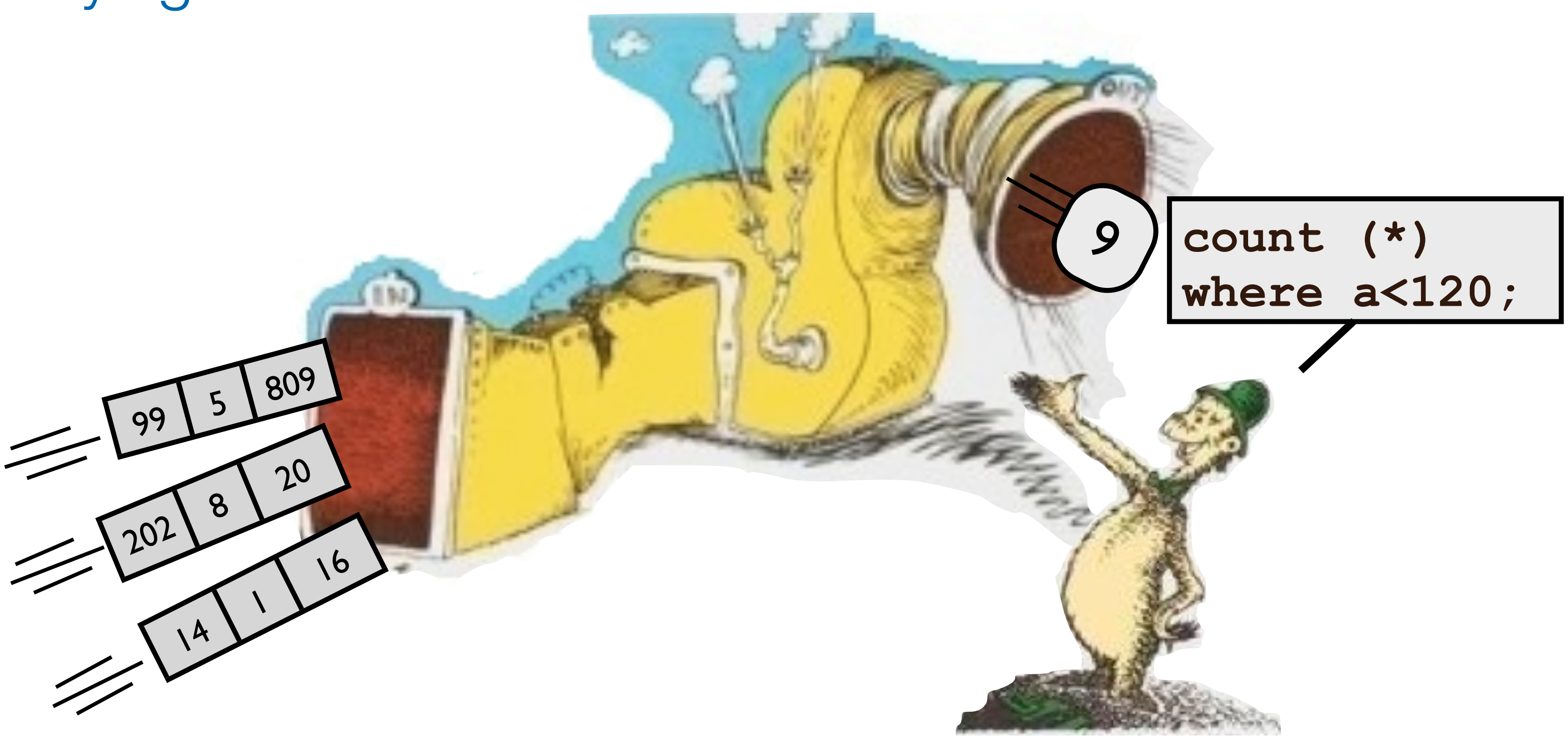
Michael A. Bender
Tokutek & Stony Brook

Martin Farach-Colton
Tokutek & Rutgers

What's a Database?

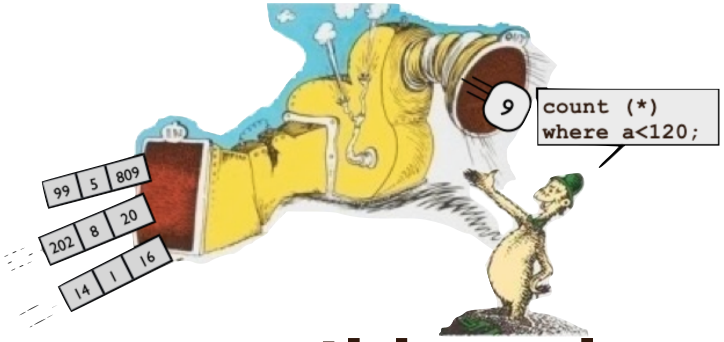
DBs are systems for:

- Storing data
- Querying data.



What's a Database?

DBs are systems for:



- Storing data
- Querying data.

DBs can have SQL interface or something else.

- We'll talk some about so-called NoSQL systems.

Data consists of a set of key,value pairs.

- Each value can consist of a well defined tuple, where each component is called a field (e.g. relational DBs).

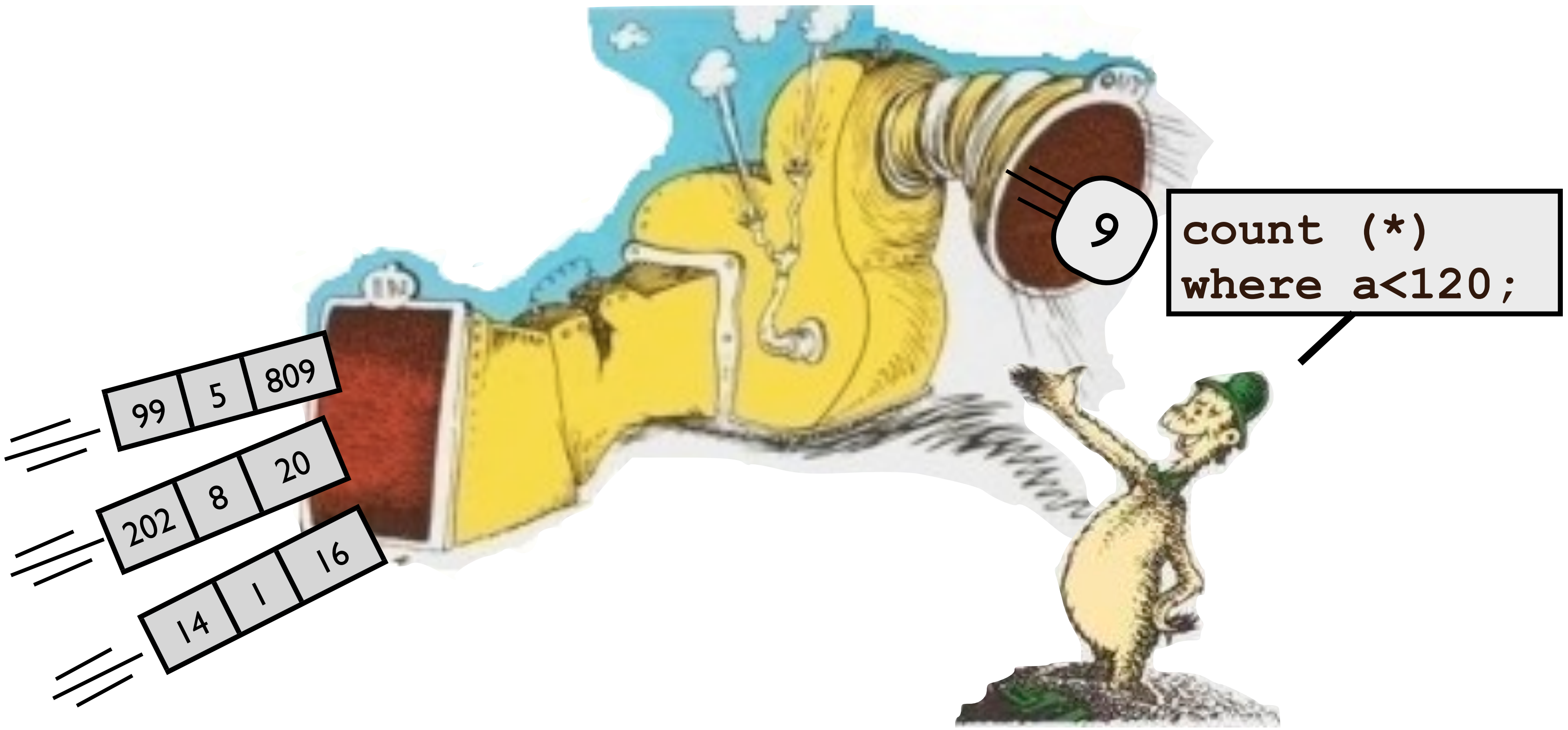
Big Data.

- Big data = bigger than RAM.

What’s this database tutorial?

Traditionally:

- Fast queries ⇒ sophisticated indexes and slow inserts.
- Fast inserts ⇒ slow queries.



We want fast data ingestion + fast queries.

What’s this database tutorial?

Database tradeoffs:

- There is a query/insertion tradeoff.
  - Algorithmicists mean one thing by this claim.
  - DB users mean something different.
- We’ll look at both.

Overview of Tutorial:

- Indexes: The DB tool for making queries fast
- The difficulty of indexing under heavy data loads
- Theory and Practice of write optimization
- From data structure to database
- Algorithmic challenges from maintaining two indexes

Row, Index, and Table

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

Row

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

Index

- Ordering of rows by key
- Used to make queries fast

Table

- Set of indexes

```
create table foo (a int, b int, c int,
primary key(a));
```

An index can select needed rows

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

```
count (*) where a<120;
```

An index can select needed rows

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

100	5	45
101	92	2

2

```
count (*) where a<120;
```

No good index means slow table scans

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

```
count (*) where b>50 and b<100;
```

No good index means slow table scans

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

3

```
count (*) where b>50 and b<100;
```

You can add an index

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

```
alter table foo add key(b);
```

A selective index speeds up queries

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

```
count (*) where b>50 and b<100;
```

A selective index speeds up queries

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	156
56	256
92	101

56	156
56	256
92	101

3

```
count (*) where b>50 and b<100;
```

Selective indexes can still be slow

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

```
sum(c) where b>50;
```

Selective indexes can still be slow

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

56	156
56	256
92	101
202	198

Selecting on b: fast

```
sum(c) where b>50;
```

Selective indexes can still be slow

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

56	156
56	256
92	101
202	198

Selecting on b: fast

Fetching info for summing c: slow

```
sum(c) where b>50;
```

Selective indexes can still be slow

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

56	156
56	256
92	101
202	198

Selecting on b: fast

156	56	45

Fetching info for summing c: slow

```
sum(c) where b>50;
```

Selective indexes can still be slow

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

56	156
56	256
92	101
202	198

Selecting on b: fast

156	56	45

Fetching info for summing c: slow

```
sum(c) where b>50;
```

Selective indexes can still be slow

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

56	156
56	256
92	101
202	198

Selecting on b: fast

156	56	45
256	56	2

Fetching info for summing c: slow

```
sum(c) where b>50;
```

Selective indexes can still be slow

Poor data locality

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

Selecting on b: fast

56	156
56	256
92	101
202	198

Fetching info for summing c: slow

156	56	45
256	56	2
101	92	2
198	202	56

105

```
sum(c) where b>50;
```

Covering indexes speed up queries

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b,c	a
5,45	100
6,2	165
23,252	206
43,45	412
56,2	256
56,45	156
92,2	101
202,56	198

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

Covering indexes speed up queries

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b,c	a
5,45	100
6,2	165
23,252	206
43,45	412
56,2	256
56,45	156
92,2	101
202,56	198

56,2	256
56,45	156
92,2	101
202,56	198

↓

105

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

DB Performance and Indexes

Read performance depends on having the right indexes for a query workload.

- We've scratched the surface of index selection.
- And there's interesting query optimization going on.

Write performance depends on speed of maintaining those indexes.

Next: how to implement indexes and tables.

An index is a dictionary

Dictionary API: maintain a set S subject to

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

A table is a set of indexes

A table is a set of indexes with operations:

- Add index: `add key(f1,f2,...);`
- Drop index: `drop key(f1,f2,...);`
- 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
- ...

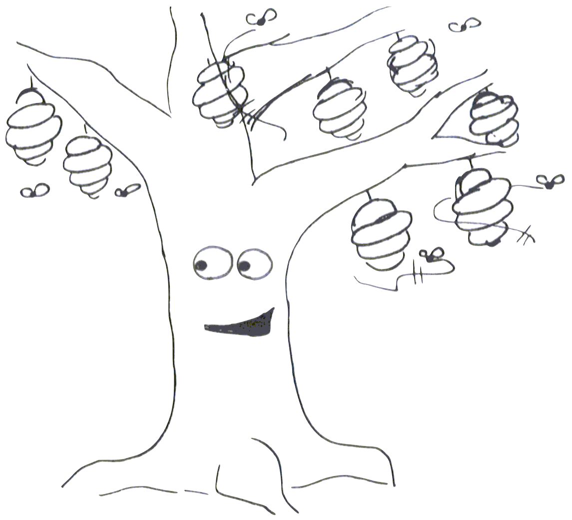
Subject to index correctness constraints.

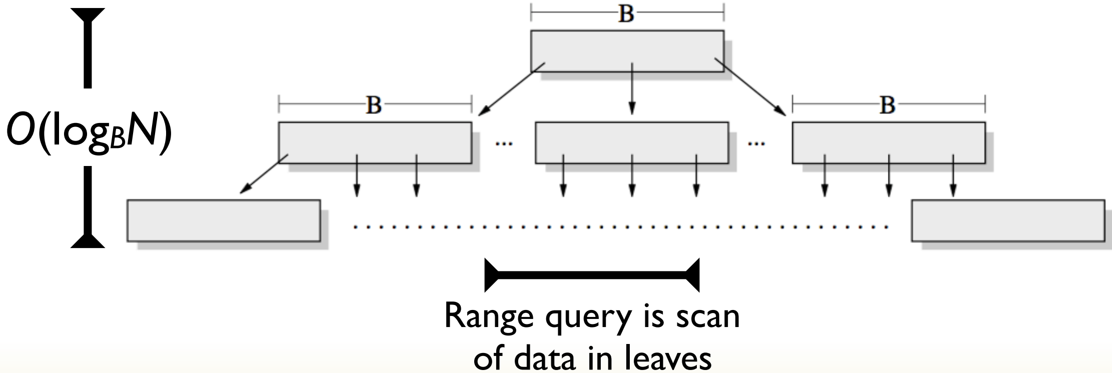
We want table operations to be fast too.

Indexes are typically B-trees

B-tree performance:

- Point queries: $O(\log_B N)$ I/Os.
  - Matches lower bound for DAM model.
- Range queries of K elements: $O(\log_B N + K/B)$ I/Os.
  - We'll talk about B-tree aging later.
- Insertions: $O(\log_B N)$ I/Os.

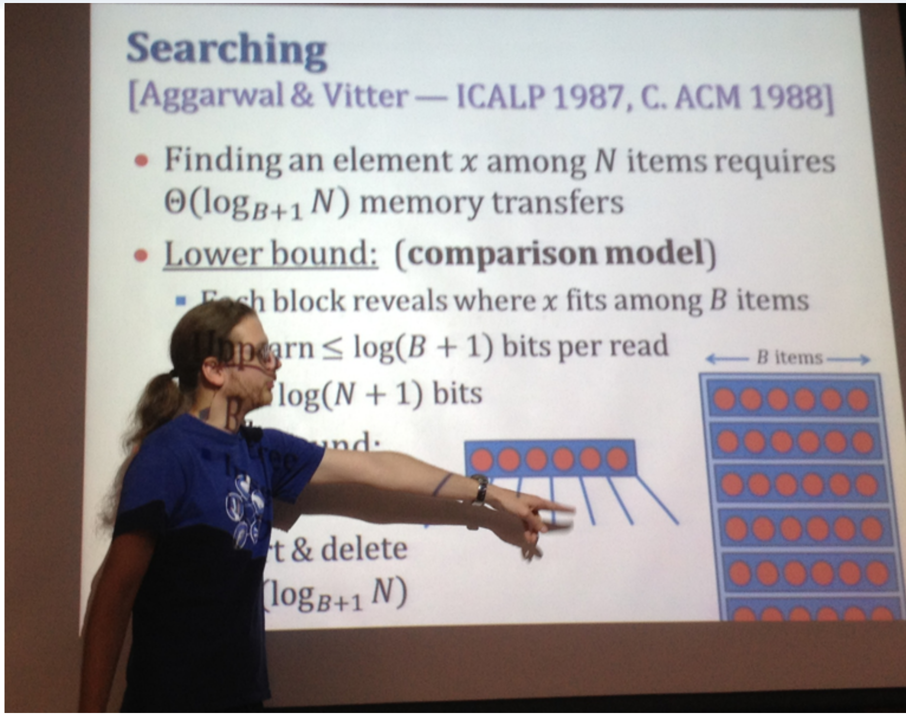


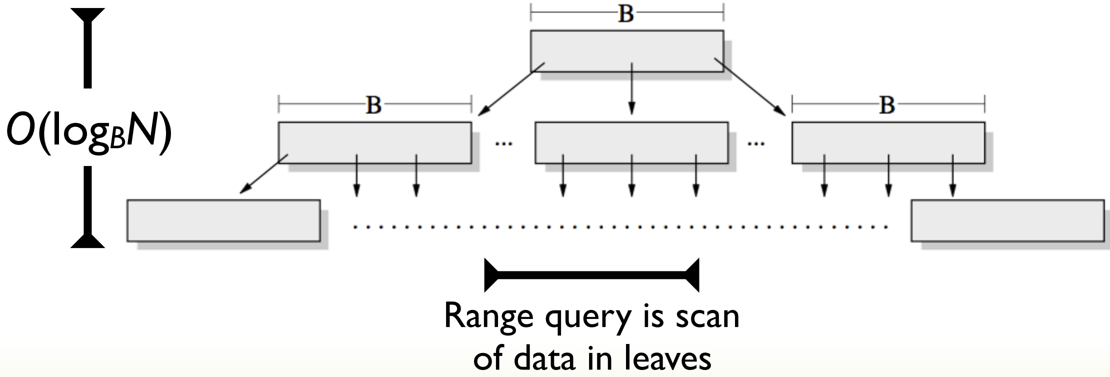


Indexes are typically B-trees

B-tree performance:

- Point queries: $O(\log_B N)$ I/Os.
  - Matches lower bound for DAM model.
- Range queries of K elements: $O(\log_B N + K/B)$ I/Os.
  - We'll talk about B-tree aging later.
- Insertions: $O(\log_B N)$ I/Os.





Worst-case analysis is a fail

Some indexes are easy to maintain, others not.

- The database is chugging along nicely.
- You add an index.
- Performance tanks --- inserts can run 100x slower.

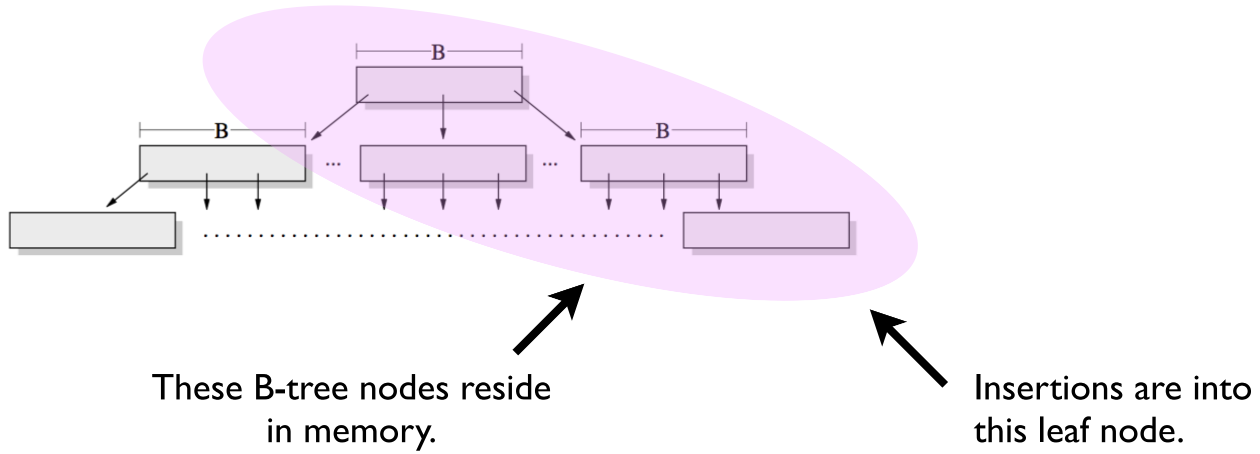
Indexes and Performance Anomalies

Databases can exhibit performance anomalies when indexes are modified.

- “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

B-trees and Caching: Sequential Inserts

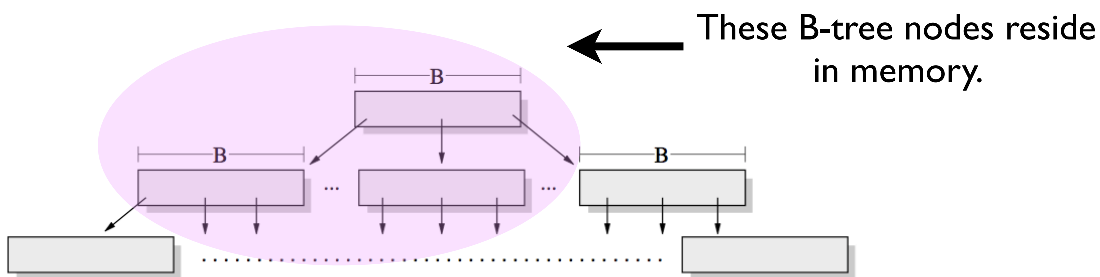
Sequential B-tree inserts run fast because of near-optimal data locality.



- One disk I/O per leaf (though many elements are inserted).
- O(1/B) I/Os per row inserted.
- Performance is limited by disk-bandwidth.

B-trees and Caching: Random Inserts

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



- Most leaves are not in main memory.
- This achieves worst case performance: O($\log_B N$).
- ≤ 100’s inserts/sec/disk (≤ 0.2% of disk bandwidth).
  - Two orders of magnitude slower than sequential insertions.

B-tree insertion: DB Practice

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

- Sequential inserts via an autoincrement key.
  - Makes insertions fast but queries slow.
- Few indexes, few covering indexes.

t	b	c
1	5	45
2	92	2
3	56	45
4	6	2
5	202	56
6	23	252
7	56	2
8	43	45

B-tree insertion: DB Practice

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

t	b	c
1	5	45
2	92	2
3	56	45
4	6	2
5	202	56
6	23	252
7	56	2
8	43	45

- Sequential inserts via an autoincrement key.
  - Makes insertions fast but queries slow.
- Few indexes, few covering indexes.

Adding sophisticated indexes helps queries.

- B-trees cannot afford to maintain them.

B-tree insertion: DB Practice

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

- Sequential inserts via an autoincrement key.
  - Makes insertions fast but queries slow.
- Few indexes, few covering indexes.

t	b	c
1	5	45
2	92	2
3	56	45
4	6	2
5	202	56
6	23	252
7	56	2
8	43	45

Adding sophisticated indexes helps queries.

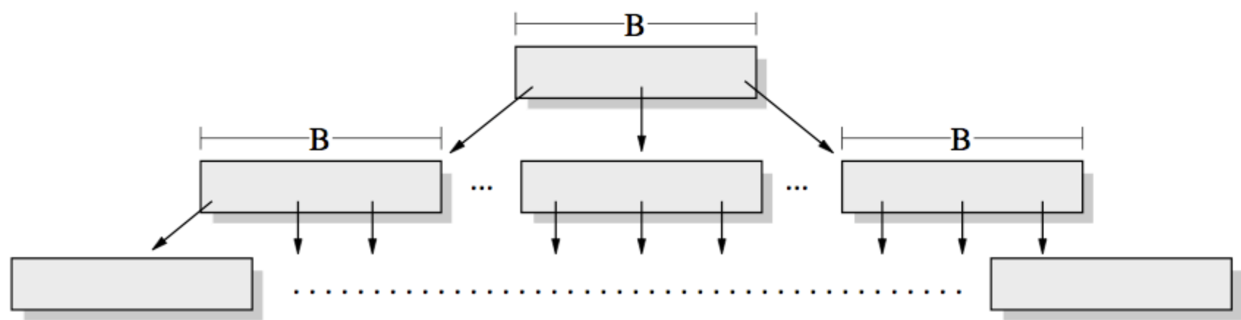
- B-trees cannot afford to maintain them.

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

Write-Optimized External Dictionaries

What we want:

- B-tree API. Better insert/delete performance.

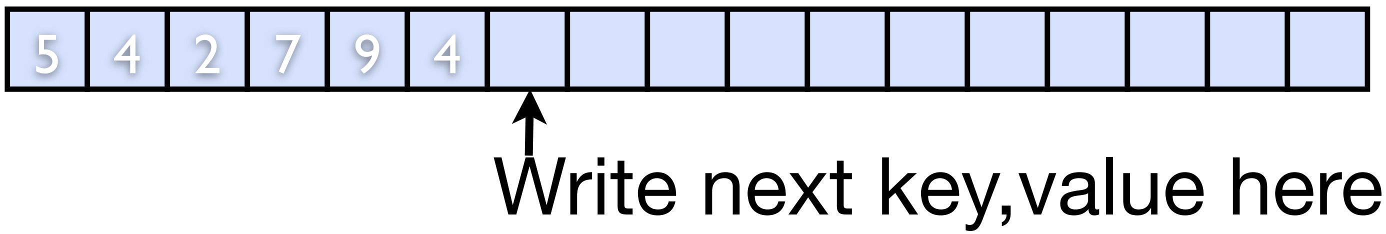


There's a $\Omega(\log_B N)$ lower bound for searching...

... *but not for inserting.*

Write-Optimized External Dictionaries

Append-to-file beats B-trees at insertions.



Pros:

- Achieve disk bandwidth even for random keys.
- Ie, inserts cost amortized O(1/*B*).

Cons:

- Looking up anything requires a table scan.
- Searches cost O(*N*/*B*).

Write-Optimized External Dictionaries

structure	insert	point query
B-tree	$O\left(\frac{\log N}{\log B}\right)$	$O\left(\frac{\log N}{\log B}\right)$
append-to-file	$O\left(\frac{1}{B}\right)$	$O\left(\frac{N}{B}\right)$
write-optimized	$O\left(\frac{\log N}{\varepsilon B^{1-\varepsilon} \log B}\right)$	$O\left(\frac{\log N}{\varepsilon \log B}\right)$
write-optimized (ε=1/2)	$O\left(\frac{\log N}{\sqrt{B}}\right)$	$O\left(\frac{\log N}{\log B}\right)$

Some optimal write optimized structures:

- Buffered repository tree [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00]
- B^{ε}-tree [Brodal, Fagerberg 03]
- Streaming B-tree [Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson 07]
- Fractal Tree Index [Tokutek]
- xDict [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10]

Write optimization techniques in production

Online insert buffer

- InnoDB, Vertica

Offline insert buffers

- OLAP, OLAP, OLAP

Heuristic techniques for getting more productive work done when we touch a B-tree leaf

Cascading

- LSM trees in Bigtable, Cassandra, H-Base

Asymptotically optimal data structures

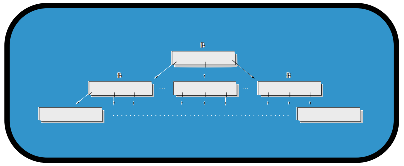
- Buffered Repository Trees (BRT), B^{ε} -tree: Tokutek
- Cache-oblivious Lookahead Arrays (COLA): Tokutek, Acunu

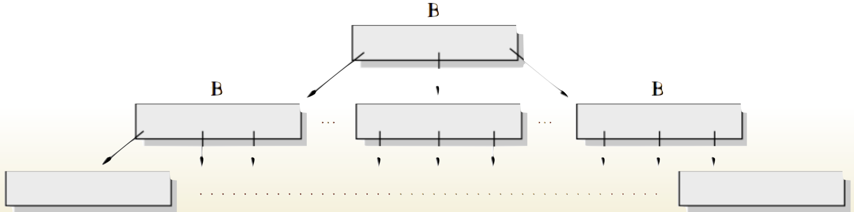
Write optimization techniques in production

B-trees with an online in-RAM buffer

- Flush multiple operations to same leaf
- To query: search in buffer and in B-tree.

in-RAM buffer





Write optimization techniques in production

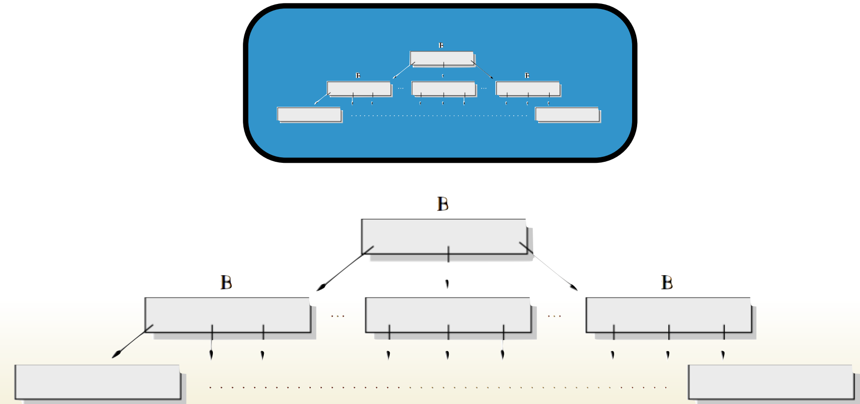
B-trees with an online in-RAM buffer

- Flush multiple operations to same leaf
- To query: search in buffer and in B-tree.

~~Analysis~~ Experience

- Smooths out the "dropping out of memory" cliff.
- Improves inserts by a small constant (say, 1x-4x).

in-RAM buffer



Write optimization techniques in production

B-trees with an online in-RAM buffer

- Flush multiple operations to same leaf
- To query: search in buffer and in B-tree.

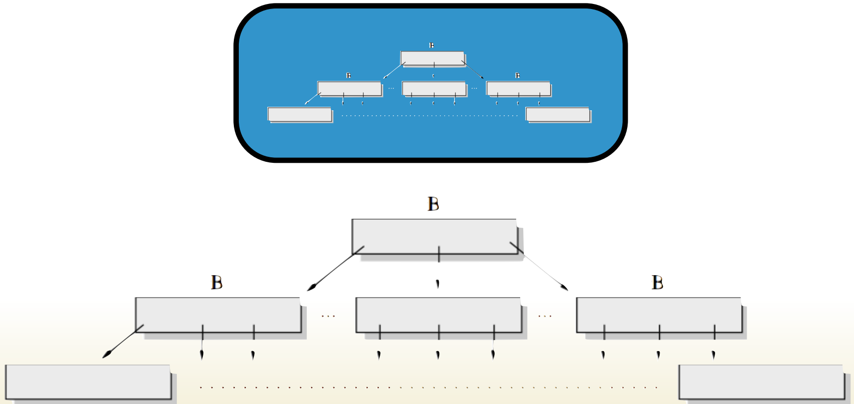
~~Analysis~~ Experience

- Smooths out the “dropping out of memory” cliff.
- Improves inserts by a small constant (say, 1x-4x).

Used in

- InnoDB, Vertica, ...

in-RAM buffer



Write optimization techniques in production

OLAP: B-tree with an offline log of inserts

- When log gets big enough (say cN), sort and insert.
  - Or do this operation during scheduled down time.
- To queries: search in B-tree.
  - There's a time lag before data gets into queryable B-tree, so queries are on stale data.
  - This is called data latency.

Write optimization techniques in production

OLAP: B-tree with an offline log of inserts

- When log gets big enough (say cN), sort and insert.
  - Or do this operation during scheduled down time.
- To queries: search in B-tree.
  - There's a time lag before data gets into queryable B-tree, so queries are on stale data.
  - This is called data latency.

Analysis

- cB insertions per leaf.
- Increasing c increases throughput but also latency.

Write optimization techniques in production

OLAP: B-tree with an offline log of inserts

- When log gets big enough (say cN), sort and insert.
  - Or do this operation during scheduled down time.
- To queries: search in B-tree.
  - There's a time lag before data gets into queryable B-tree, so queries are on stale data.
  - This is called data latency.

Analysis

- cB insertions per leaf.
- Increasing c increases throughput but also latency.

Marketing is king

- Not clear why this is OnLine Analytical Processing.

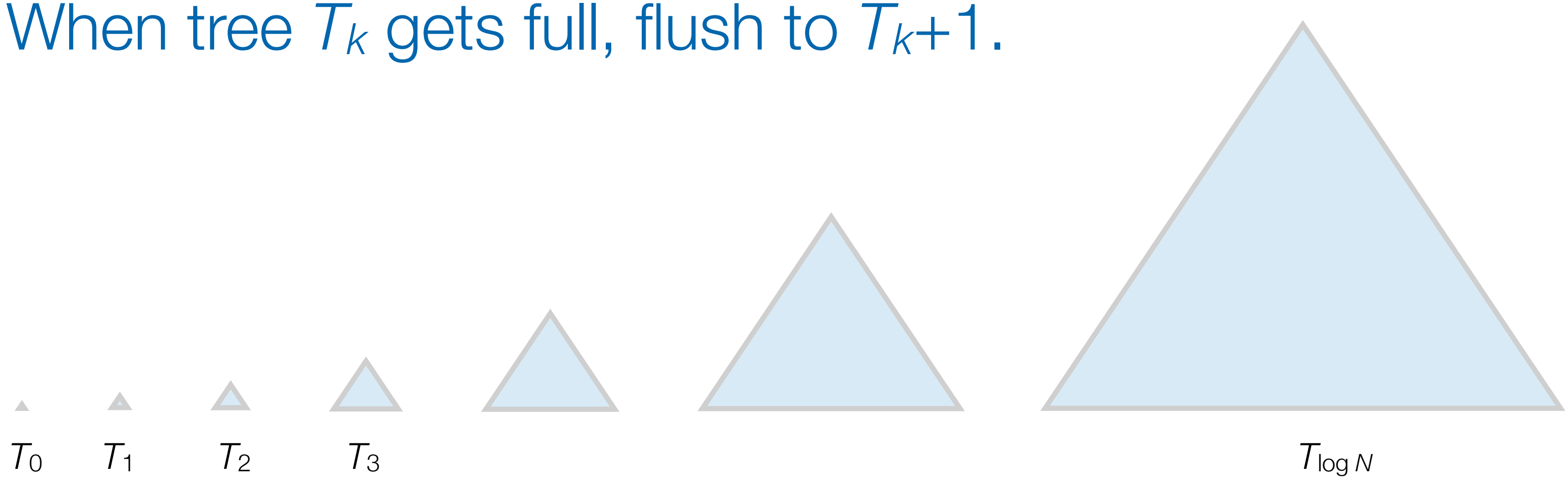
Write optimization techniques in production

Cascading:

[O'Neil,Cheng, Gawlick, O'Neil 96]

Log structured merge (LSM) trees

- Maintain cascading B-tree T_1, ..., $T_{\log N}$, $|T_i| < c^i$
- When tree T_k gets full, flush to T_k+1.



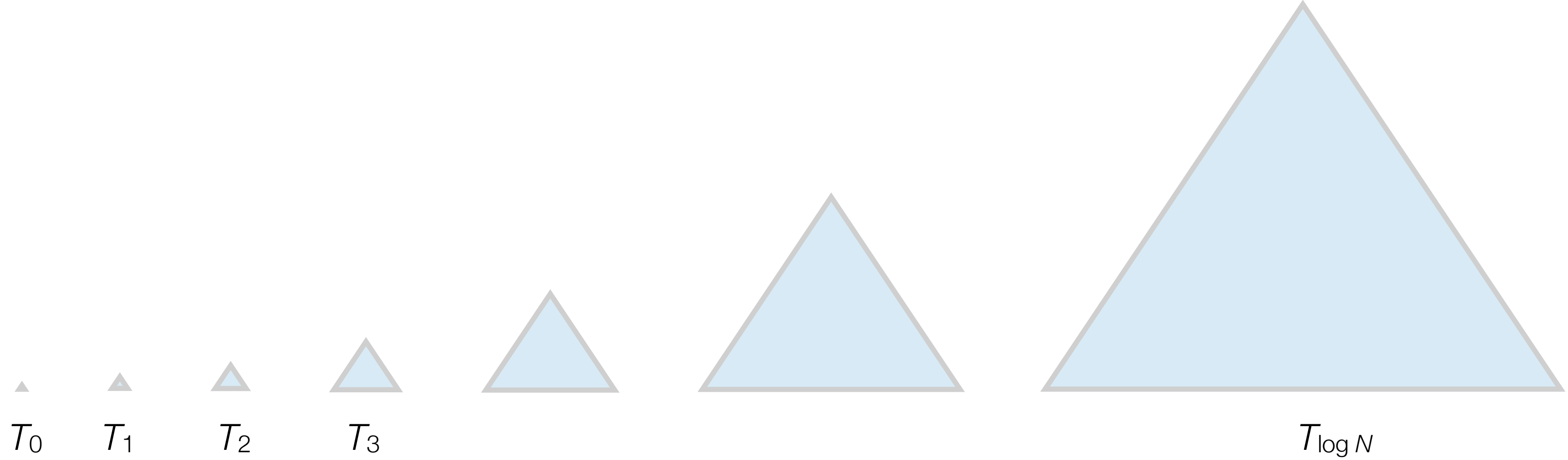
LSM Analysis

- Inserts: $O((\log N)/B)$. ✔
- Queries: $O(\log N \log_B N)$. ✘

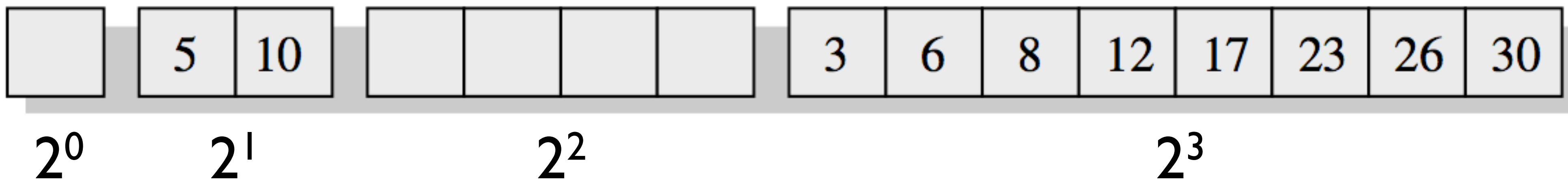
Write optimization techniques in production

LSMs in production:

- Big-Table, Cassandra, H-base
- Bloom filters to improve performance.
- Some LSM implementations (e.g. Cassandra, H-base) don't even have a successor operation, because it runs too slowly.



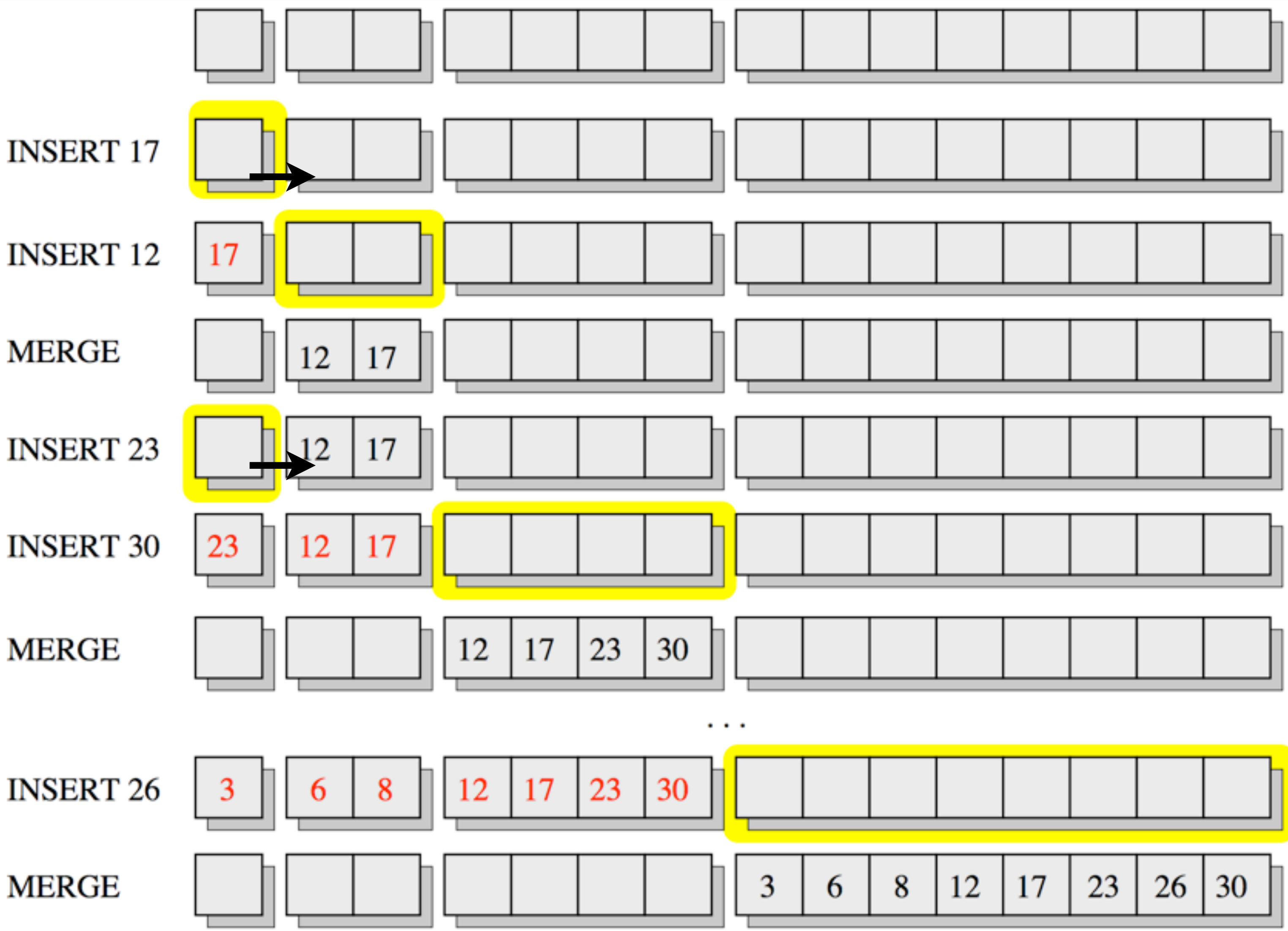
Simplified CO Lookahead Tree (COLA)



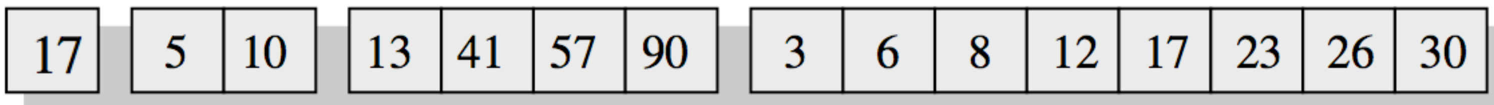
O((log*N*)/*B*) insert cost & O(log²*N*) search cost

- It's an LSM, except we keep arrays instead of B-trees.
- A factor of O($\log_B N$) slower on searches than an LSM.
- We'll use "fractional cascading" to search in O(log *N*).

COLA Insertions (Similar to LSMs)



Analysis of COLA



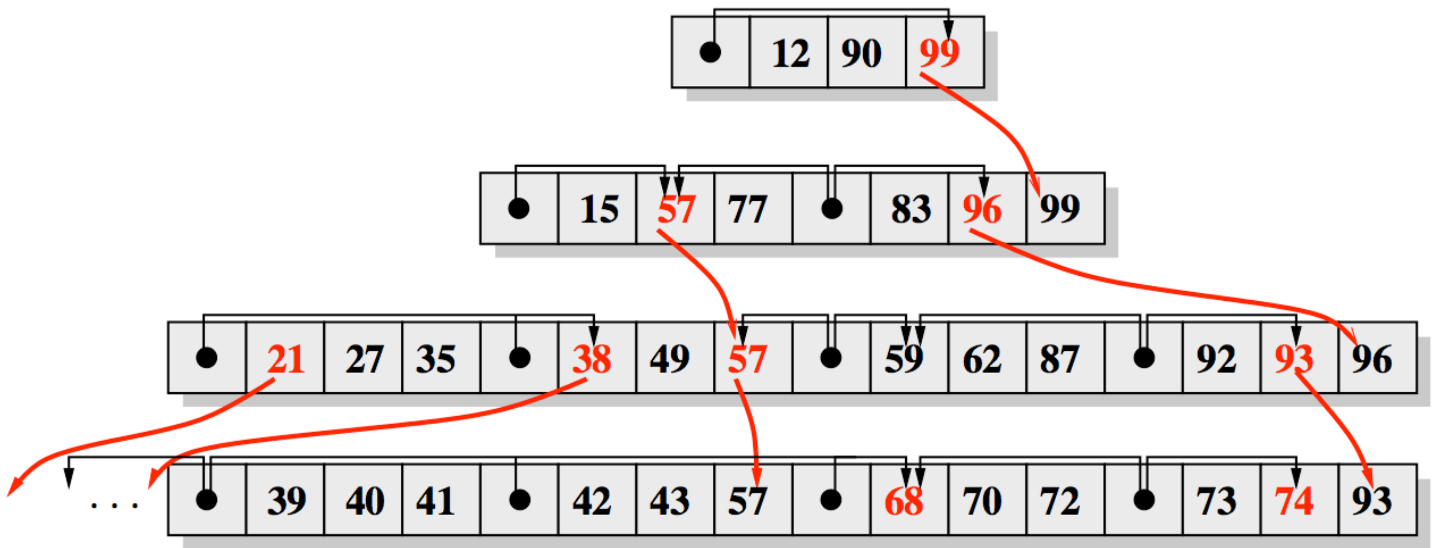
Insert Cost:

- cost to flush buffer of size X = $O(X/B)$
- cost per element to flush buffer = $O(1/B)$
- max # of times each element is flushed = $\log N$
- insert cost = $O((\log N))/B)$ amortized memory transfers

Search Cost

- Binary search at each level
- $\log(N/B) + \log(N/B) - 1 + \log(N/B) - 2 + \ldots + 2 + 1$
  $= O(\log^2(N/B))$

Idea of Faster Key Searches in COLA



O(log (*N*/*B*)) search cost

- Some redundancy of elements between levels
- Arrays can be partially full
- Horizontal and vertical pointers to redundant elements
- (Fractional Cascading)

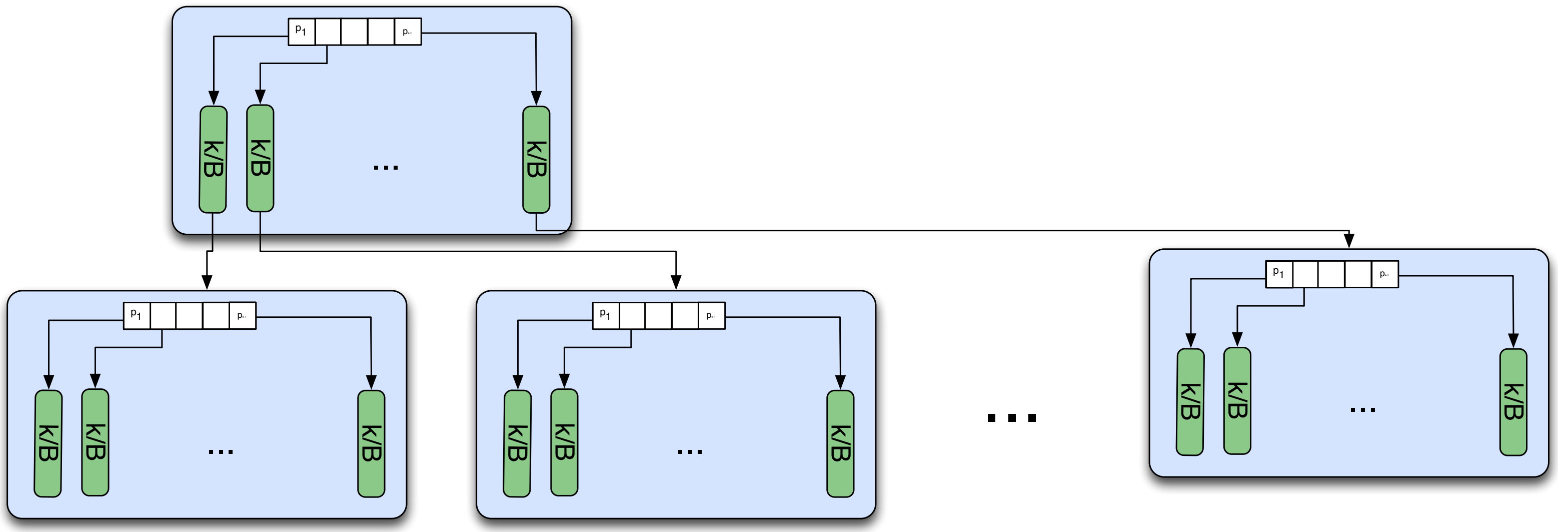
COLA Tradeoff

Arrays can grow by bigger factors

- Larger growth makes insertions slower but queries faster.
- It's the same tradeoff curve as a B^{ε}-tree. (See below.)
- In order to get the full tradeoff curve, you need a growth factor that depends on *B*.
  - You loose cache-obliviousness.
- The xDict achieves $O(\log_B N)$ searches while staying cache-oblivious.
  - We don't know of an implementation, but would love to hear about it.

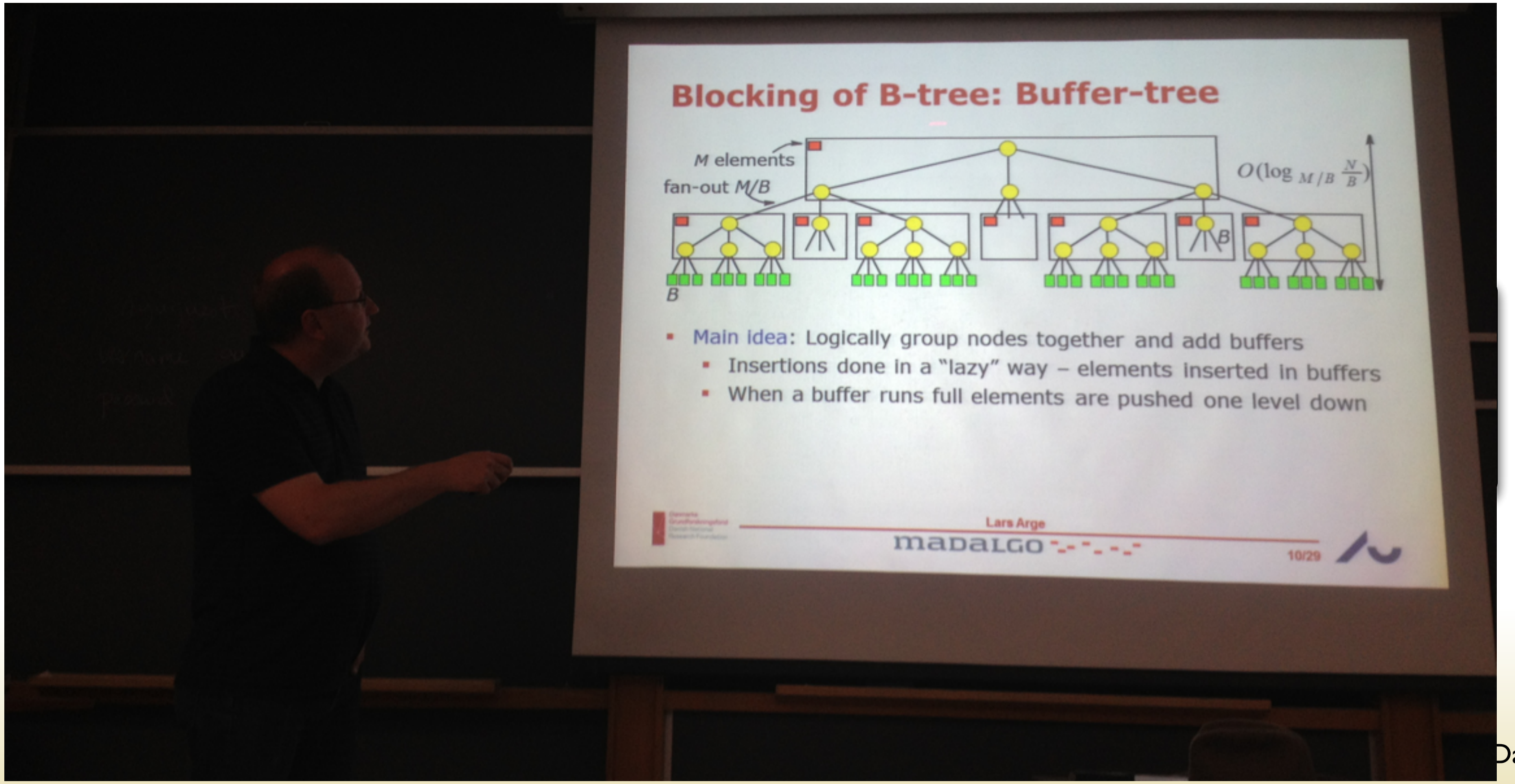
k-tree with k/B edge buffers

- Branching factor of k.
- Each branch gets a buffer of size k/B.
  - All buffers in a node total size B.
- When a buffer fills, flush to child.



k-tree with k/B edge buffers

- Branching factor of k.
- Each branch gets a buffer of size k/B.
  - All buffers in a node total size B.
- When a buffer fills, flush to child.



B^ε -tree Performance

Searches: $O(\log_k N)$

Insertions:

- Cost to flush a buffer: $O(1)$.
- Cost to flush a buffer, per element: $O(k/B)$.
- # of flushes per element = height of tree: $O(\log_k N)$.
- Total amortized cost to flush to a leaf: $O(k \log_k N/B)$.
- Pick $k = B^\varepsilon$
  - Searches: $O((1/\varepsilon)\log_B N)$, as good as B-tree for constant ε.
  - Insertions: $O(\log_B N/\varepsilon B^{1-\varepsilon})$, as good as LSMs.

Write optimization. ✔ What's missing?

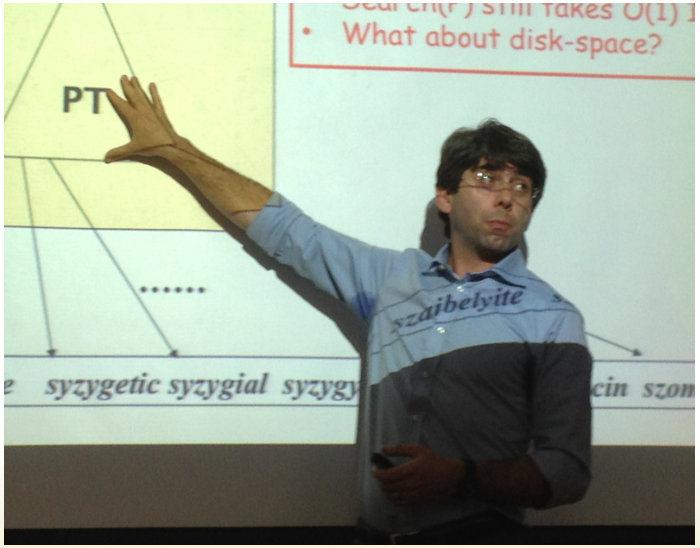
A real implementation must deal with

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

Write optimization. ✔ What's missing?

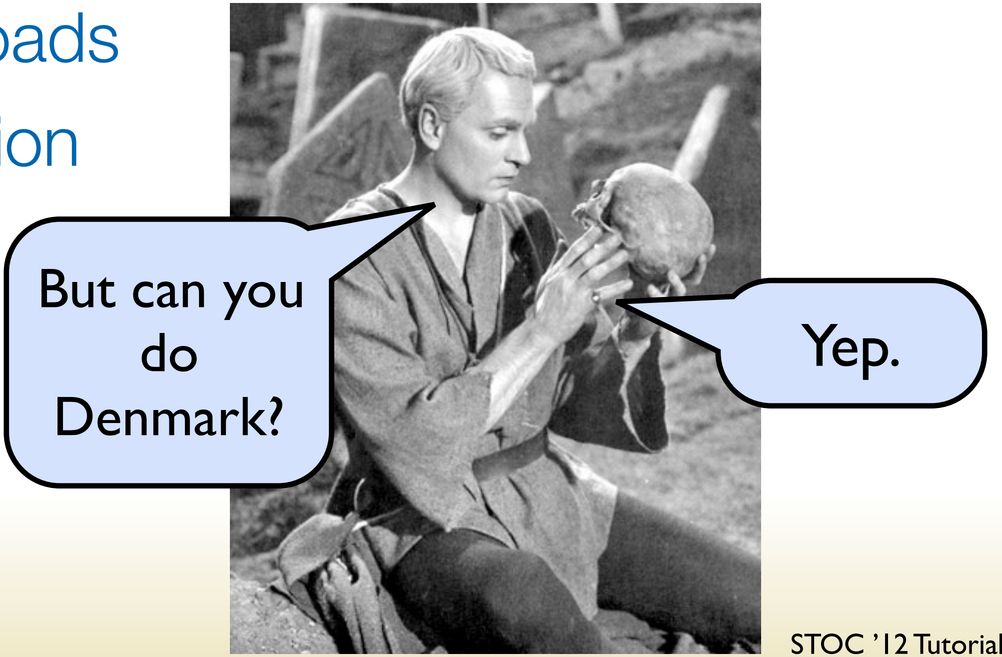
A real implementation must deal with

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

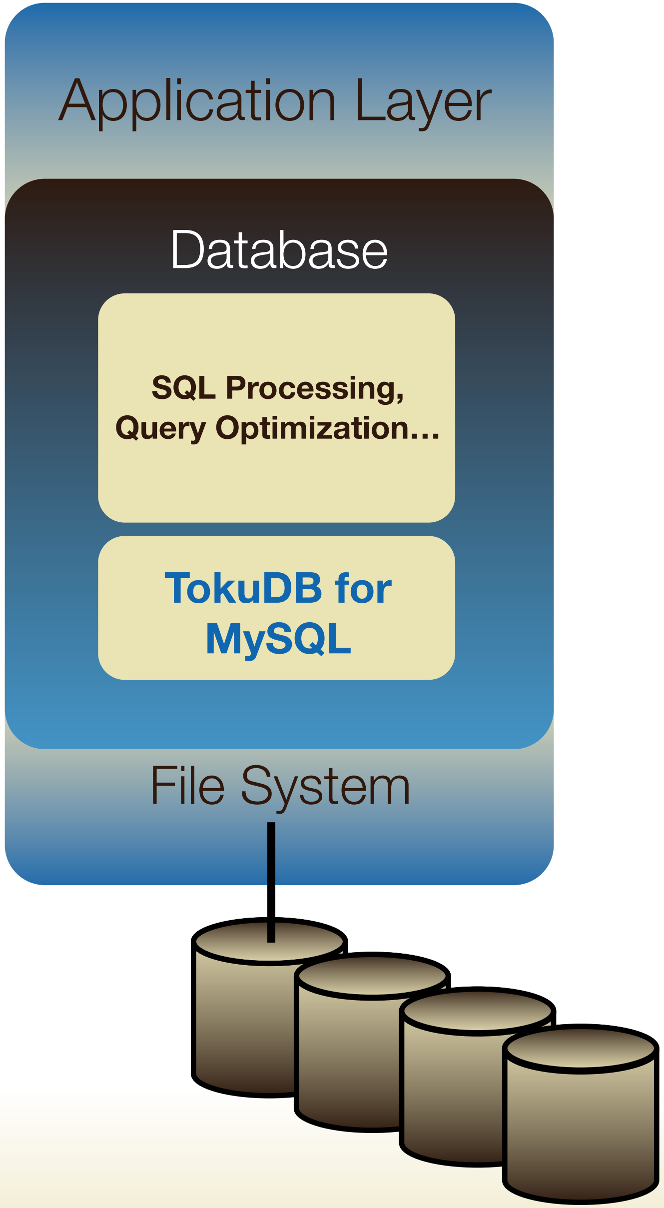


A real implementation must deal with

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



Fractal Trees are B^{ε}-tree+COLA+Stuff



TokuDB®, an industrial-strength Fractal Tree

- Berkeley DB API (a B-tree API)
- Full featured (ACID, compression, etc).

TokuDB is a storage engine for MySQL

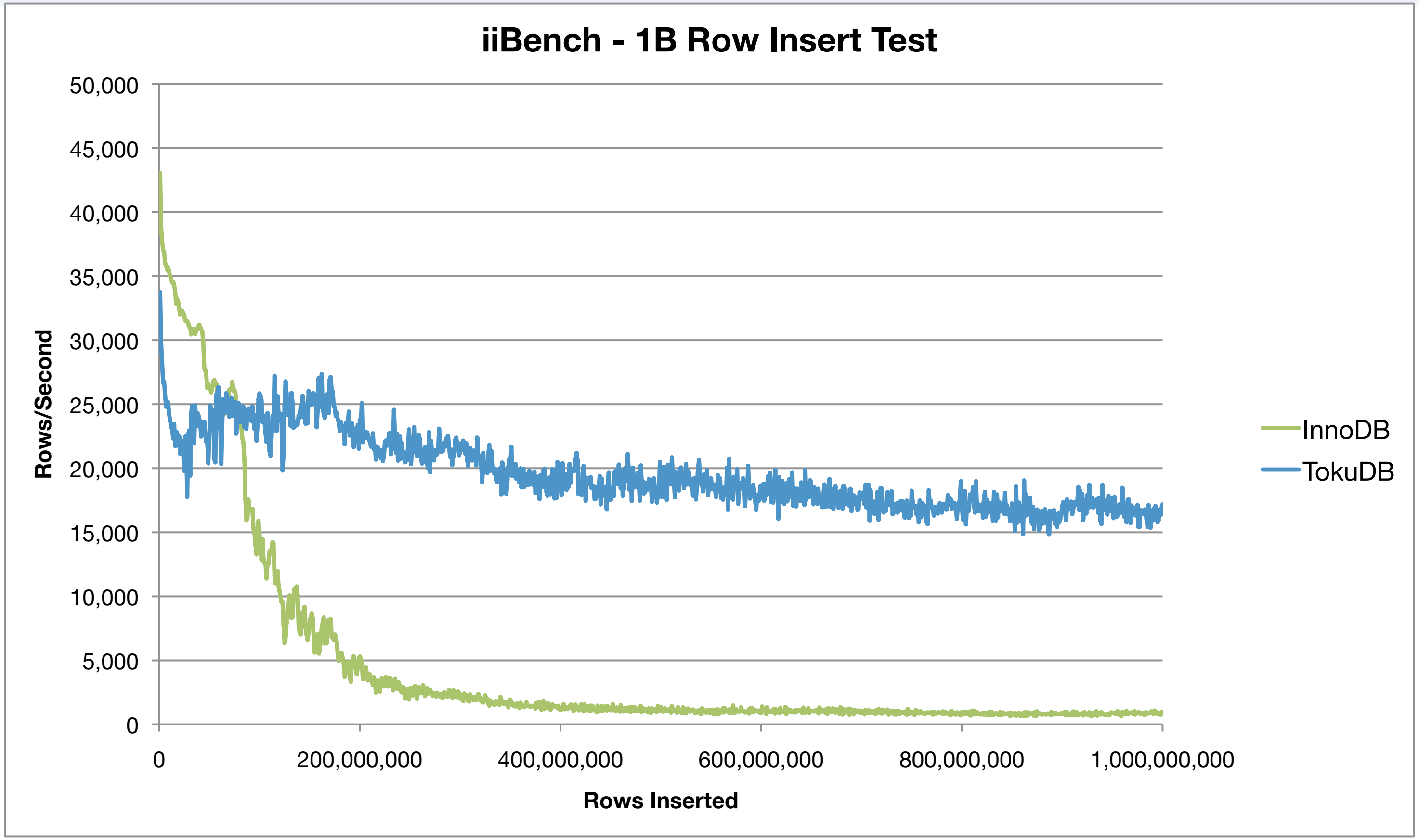
- Role of storage engine: maintains on-disk data

TokuDB inherits Fractal Tree speed

- 10x-100x faster index inserts

Tokutek is marketing this technology.

iiBench Insert Benchmark is CPU bound



Fractal Trees scale with disk bandwidth not seek time.

- But in practice, there's another bottleneck -- we are CPU bound.
  We cannot (yet) take full advantage of more cores or disks. This must change.
  (not what I expected from theoretical mode).

Fun Thing about Write Optimization

Time to fill a disk in 1973, 2010, and 2022.

- log data sequentially, index data in B-tree, index in Fractal Trees.

Year	Size	Bandwidth	Access Time	Time to log data on disk	Time to fill disk using a B-tree (row size 1K)	Time to fill using Fractal tree* (row size 1K)
1973	35MB	835KB/s	25ms	39s		
2010	3TB	150MB/s	10ms	5.5h		
2022	220TB	1.05GB/s	10ms	2.4d		

Fancy indexing structures may be a luxury now, but they will be essential by the decade's end.

* Projected times for fully multi-threaded version

Fun Thing about Write Optimization

Time to fill a disk in 1973, 2010, and 2022.

- log data sequentially, index data in B-tree, index in Fractal Trees.

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

Fancy indexing structures may be a luxury now, but they will be essential by the decade's end.

* Projected times for fully multi-threaded version

Fun Thing about Write Optimization

Time to fill a disk in 1973, 2010, and 2022.

- log data sequentially, index data in B-tree, index in Fractal Trees.

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

Fancy indexing structures may be a luxury now, but they will be essential by the decade's end.

* Projected times for fully multi-threaded version

Remember Tables?

A table is a set of indexes where:

- One index is distinguished as *primary.*
  - The key of the primary index is *unique*.
- Every other index is called *secondary.*
  - There's a bijection 'twixt the rows of a secondary and primary indexes.
  - The value of a secondary index is the key of the primary index for the corresponding row.

Constraints have performance impact.

- Why?

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

Remember Tables?

A table is a set of indexes where:

- One index is distinguished as *primary.*
  - The key of the primary index is *unique*.
- Every other index is called *secondary.*
  - There's a bijection 'twixt the rows of a secondary and primary indexes.
  - The value of a secondary index is the key of the primary index for the corresponding row.

Constraints have performance impact.

- Why?

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

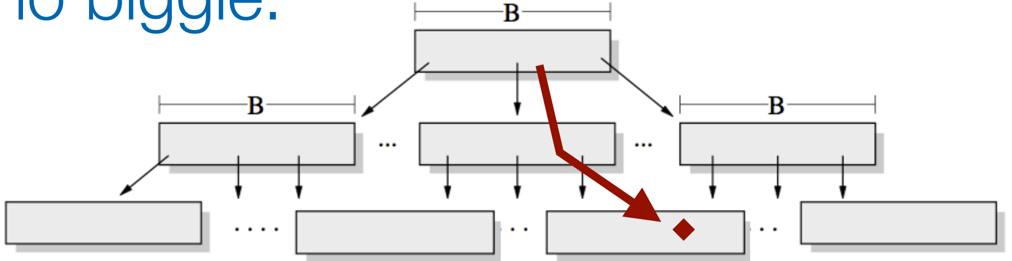
Uniqueness Checking...

Uniqueness checking has a hidden search:

```
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.



Uniqueness Checking...

Uniqueness checking has a “crypto-search”:

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

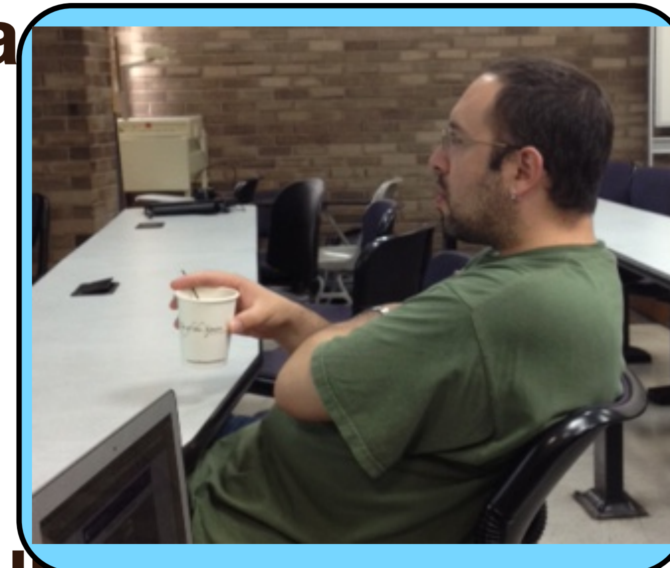
In a write-optimized structure, that pesky search can throttle performance

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

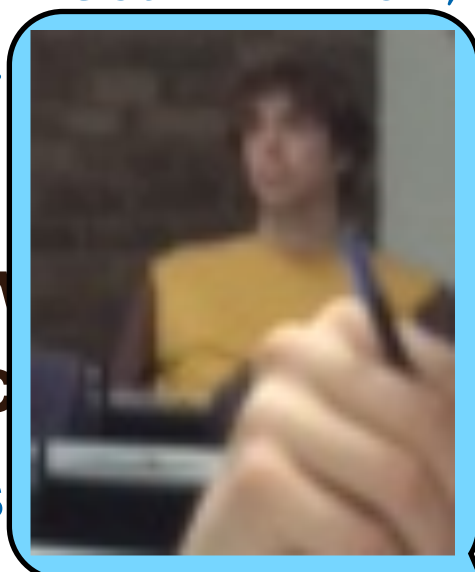
[Bender, Farach-Colton, Johnson, Kraner, Kuszmaul, Medjedovic, Montes, Shetty, Spillane, Zadok 12]

Uniqueness Checking...

Uniqueness checking has a



```
If Search(key) == True
    Return Error;
El
           ey,value);
```



In a w ed structure, that pesky
searc e performance

- Ins s 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]

Are there other crypto searches?

A table is a set of indexes where:

- One index is distinguished as *primary.*
  - The key of the primary index is *unique*.
- Every other index is called *secondary.*
  - There's a bijection 'twixt the rows of a secondary and primary indexes.
  - The value of a secondary index is the key of the primary index for the corresponding row.

Constraints have performance impact.

- Why?

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

Are there other crypto searches?

A table is a set of indexes where:

- One index is distinguished as *primary.*
  - The key of the primary index is *unique*.
- Every other index is called *secondary.*
  - There's a bijection 'twixt the rows of a secondary and primary indexes.
  - The value of a secondary index is the key of the primary index for the corresponding row.

Constraints have performance impact.

- Why?

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

How do deletes work?

Tombstone message?

- A tombstone message is a message that kills a row once it meets up with it.
- But in order to insert a tombstone message into secondary indexes, we need to know the value by which they have been indexed.
- This requires a search in the primary index.

How do we solve this?

- In this case, by considering the use cases for deletions in DBs.
- We can make range deletions fast, for example.

Are there other crypto searches?

Oh yes!

- Traditional row locking for transactions.
  - Solve by having a new data structure for locking, rather than locking at the leaf.
- Deletions with primary-only index.
  - Even without secondary indexes, the semantics of deletion usually require a return message to say that the deleted key existed, and an error if you try to delete something that didn't exist to begin with.
  - Solve by convincing customers to accept the faster semantics.
- …

Conclusion

DB indexing is a rich field.

Data structures make a big difference here.

There are loads of open problems.

- But they are usually only interesting if you take the time to really learn the use case.

What does Tokutek mean?

What does Tokutek mean?

Toku → 得 → To Profit

↓

匿

↓

Cacher → Cache → Cache-Oblivious Analysis

↓

To Hide

What does Tokutek mean?

Toku → 得 → To Profit

↓

匿

↓

Cacher → Cache → Cache-Oblivious Analysis

↓

To Hide

1. Cache-Oblivious Analysis
2. ???
3. Profit