Performance of Fractal-Tree Databases

Michael A. Bender

STONY BROOK
STATE UNIVERSITY OF NEW YORK

Tokutek™

The Problem

Problem: maintain a dynamic dictionary on disk.

Motivation: file systems, databases, etc.

The Problem

Problem: maintain a dynamic dictionary on disk.
Motivation: file systems, databases, etc.

State of the art (algorithmic perspective):

- B-tree [Bayer, McCreight 72]
- cache-oblivious B-tree [Bender, Demaine, Farach-Colton 00]
- buffer tree [Arge 95]
- buffered-repository tree [Buchsbaum,Goldwasser,Venkatasubramanian,Westbrook 00]
- B^{ε} tree [Brodal, Fagerberg 03]
- log-structured merge tree [O'Neil, Cheng, Gawlick, O'Neil 96]
- string B-tree [Ferragina, Grossi 99]
- etc, etc!

The Problem

Problem: maintain a dynamic dictionary on disk.
Motivation: file systems, databases, etc.

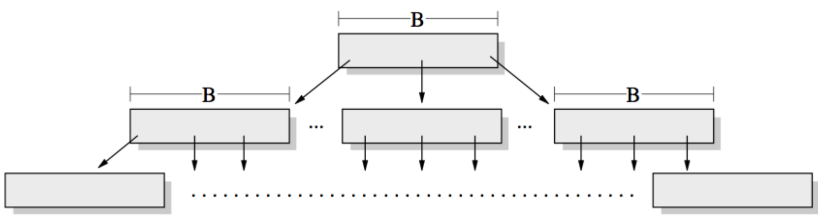
State of the art (algorithmic perspective):

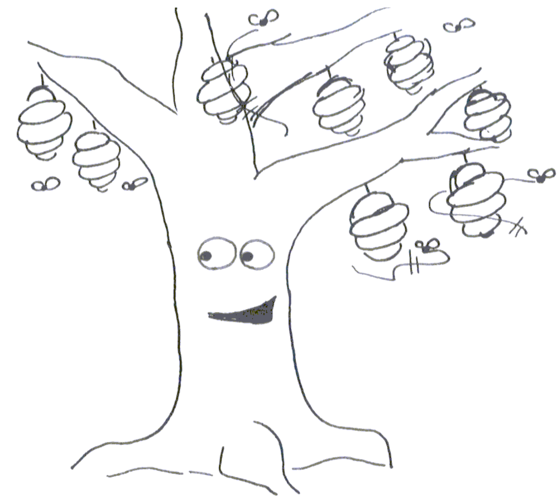
- B-tree [Bayer, McCreight 72]
- cache-oblivious B-tree [Bender, Demaine, Farach-Colton 00]
- buffer tree [Arge 95]
- buffered-repository tree[Buchsbaum,Goldwasser,Venkatasubramanian,Westbrook 00]
- B^ε tree [Brodal, Fagerberg 03]
- log-structured merge tree [O'Neil, Cheng, Gawlick, O'Neil 96]
- string B-tree [Ferragina, Grossi 99]
- etc, etc!

State of the practice:

- B-trees + industrial-strength features/optimizations

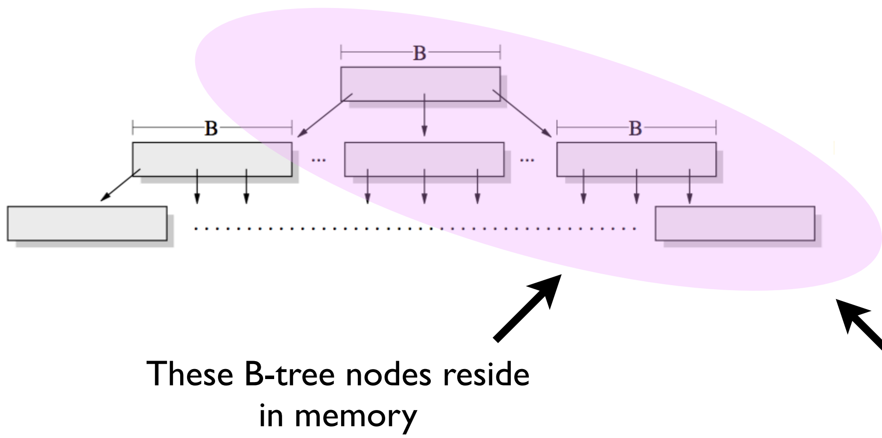
B-trees are Fast at Sequential Inserts

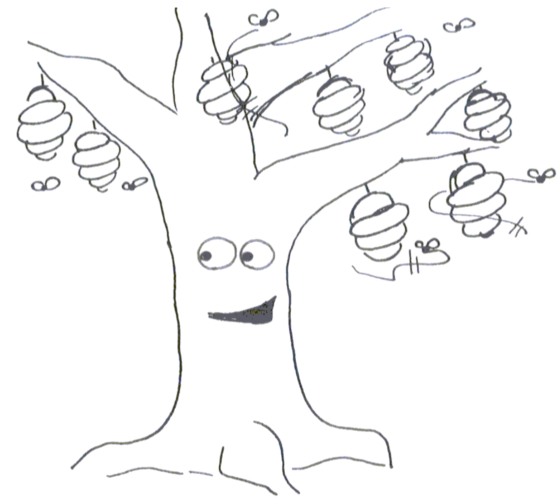




B-trees are Fast at Sequential Inserts

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

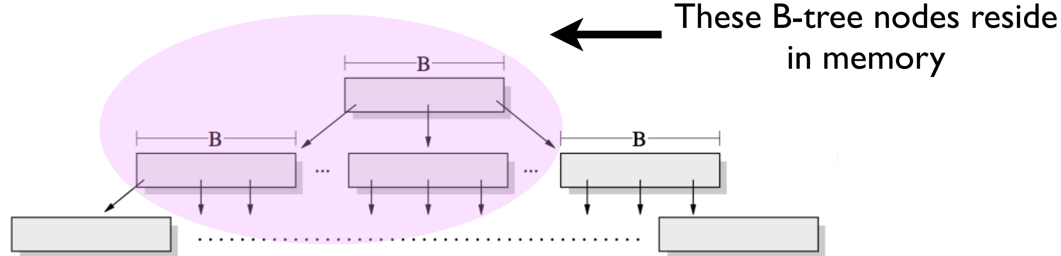




- One disk I/O per leaf (which contains many inserts).
- Sequential disk I/O.
- Performance is disk-bandwidth limited.

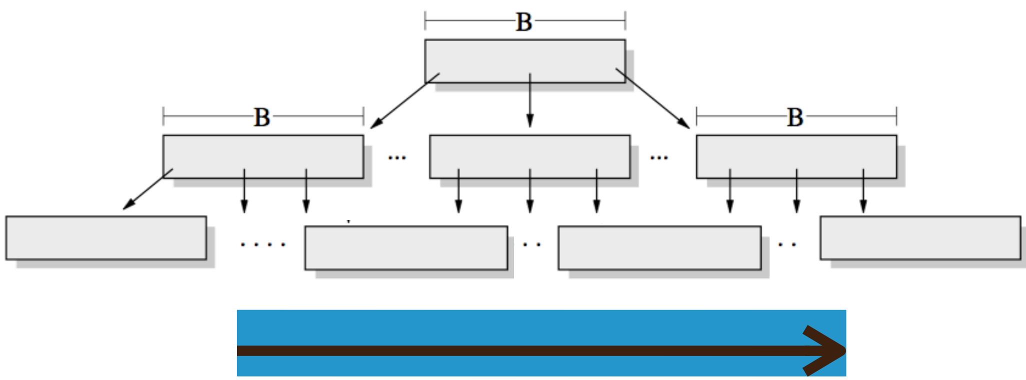
B-Trees Are Slow at Ad Hoc Inserts

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



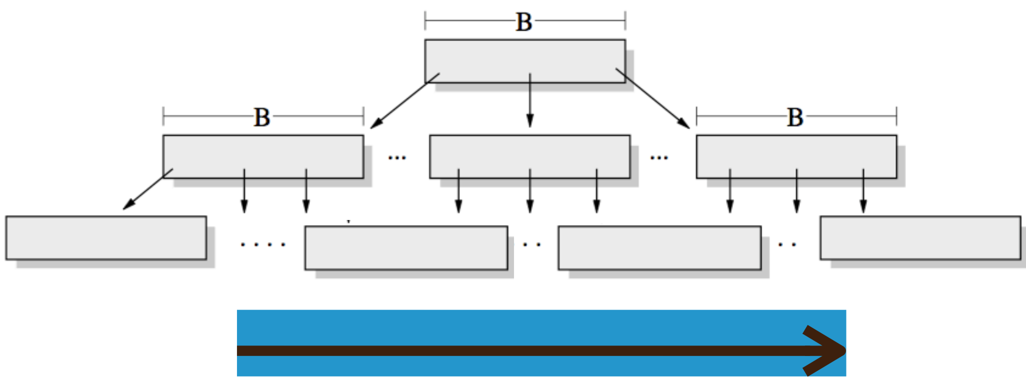
- Most nodes are not in main memory.
- Most insertions require a random disk I/O.
- Performance is disk-seek limited.
- ≤ 100 inserts/sec/disk (≤ 0.05% of disk bandwidth).

B-trees Have a Similar Story for Range Queries



Range queries in newly built B-trees have good locality

B-trees Have a Similar Story for Range Queries



Leaf nodes are scattered across disk in aged B-tree.

Range queries in newly built B-trees have good locality

Range queries in aged B-trees have poor locality

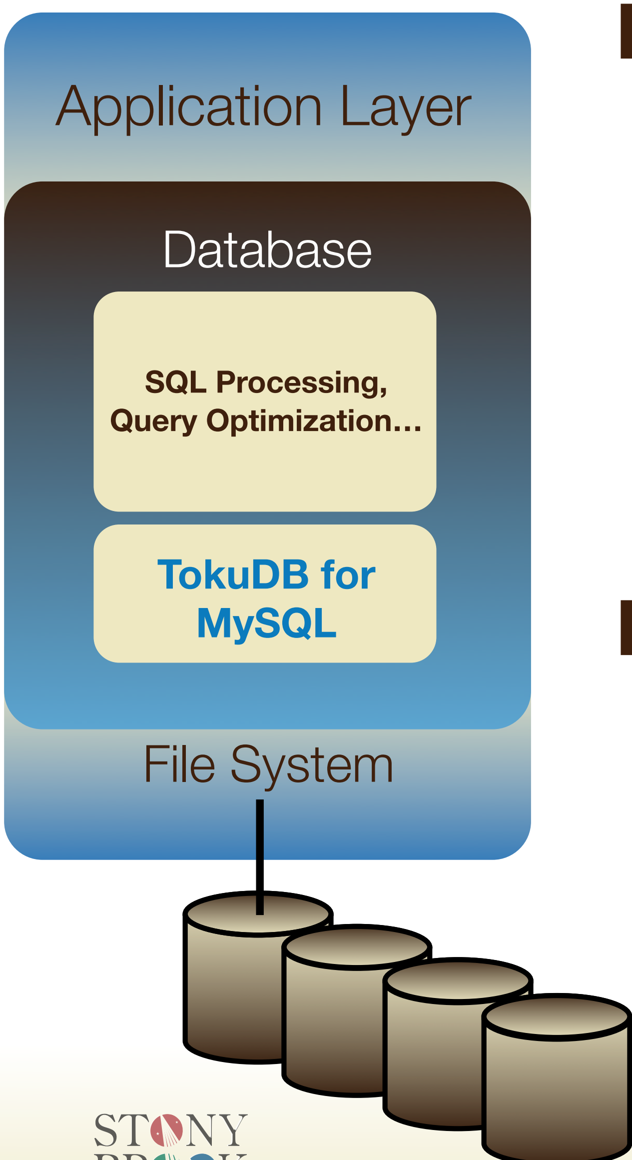
- Leaf blocks are scattered across disk.
- For page-sized nodes, as low as 1% disk bandwidth.

Results

Cache-Oblivious Streaming B-tree [Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson 07]

- Replacement for Traditional B-tree
- High entropy inserts/deletes run up to 100x faster
- No aging --> always fast range queries
- Streaming B-tree is cache-oblivious
  - Good data locality without memory-specific parameterization.

Results (cont)



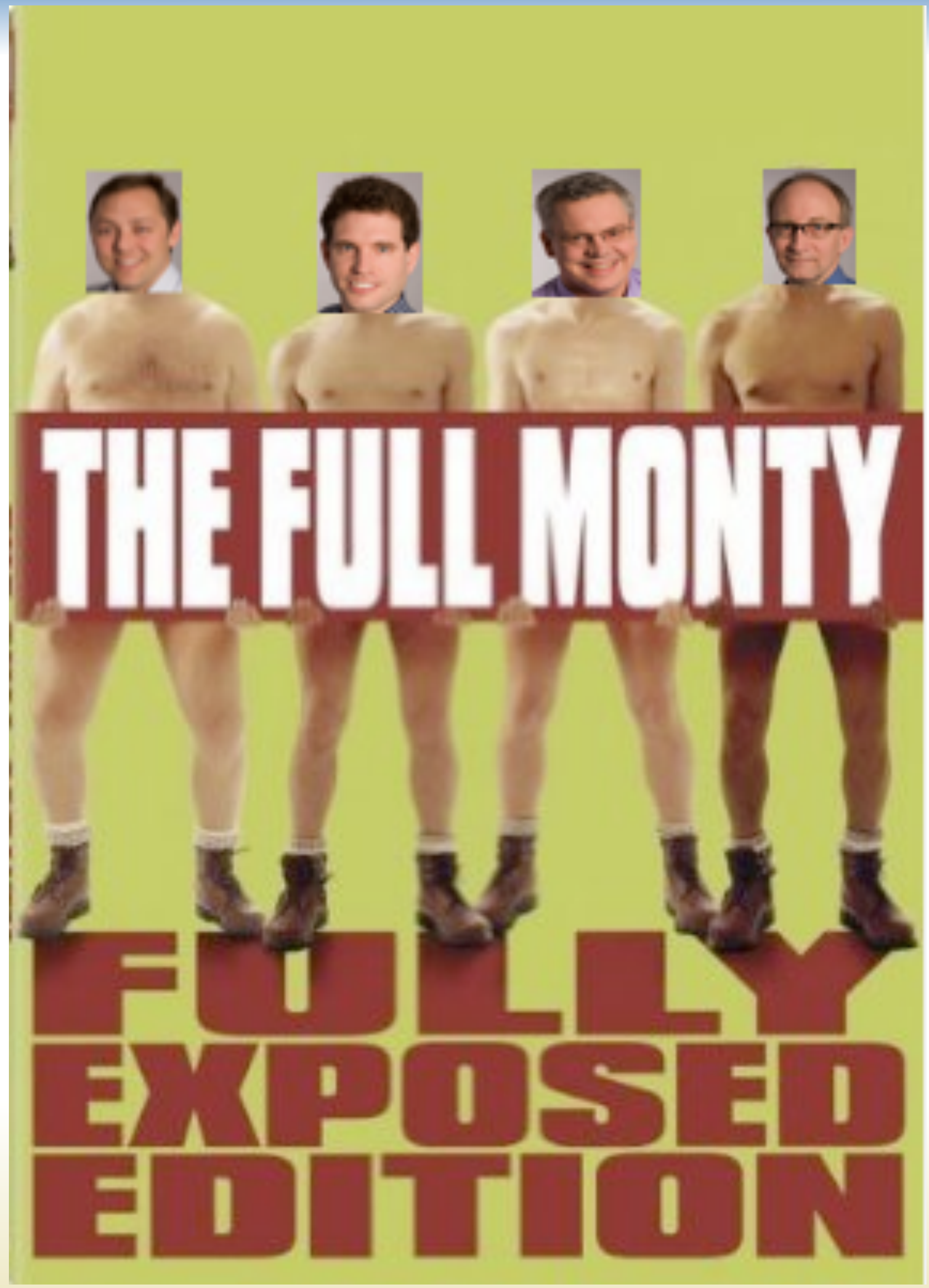
Fractal Tree™ database

- TokuDB is a *storage engine* for MySQL
  - A storage engine is a structure that stores on-disk data.
  - Traditionally a storage engine is a B-tree.
- MySQL is an open-source database
  - Most installations of any database
- Built in context of our startup Tokutek.

Performance

- 10x-100x faster index inserts
- No aging
- Faster queries in important cases

Creative Fundraising for Startup



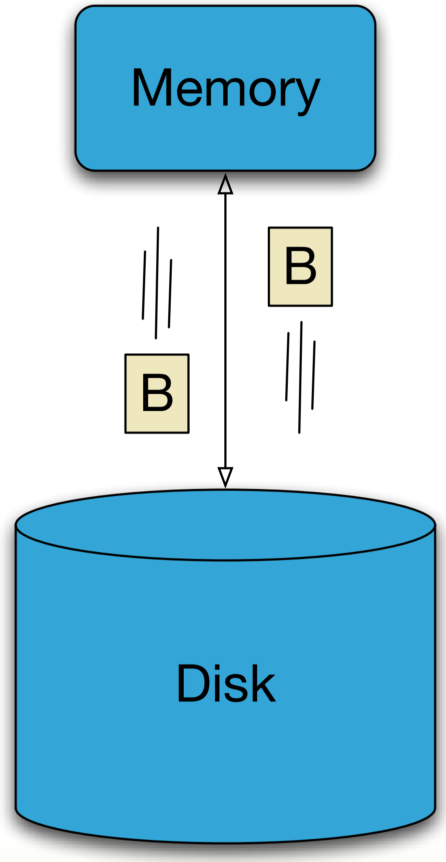
Algorithmic Performance Model

Minimize # of block transfers per operation

Disk-Access Machine (DAM) [Aggrawal, Vitter 88]

- Two-levels of memory.
- Two parameters:

  block-size ***B***, memory-size ***M***.



Algorithmic Performance Model

Minimize # of block transfers per operation

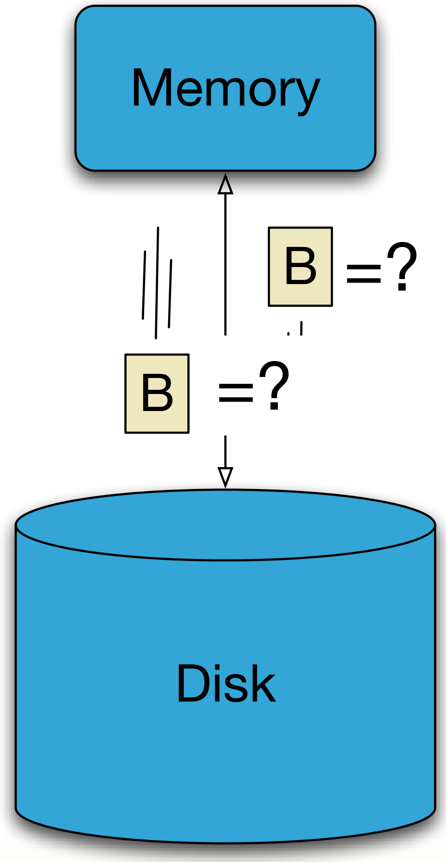
Disk-Access Machine (DAM) [Aggrawal, Vitter 88]

- Two-levels of memory.
- Two parameters:

  block-size ***B***, memory-size ***M***.

Cache-Oblivious Model (CO) [Frigo, Leiserson, Prokop, Ramachandran 99]

- Parameters ***B*** and ***M*** are unknown to the algorithm or coder.
- (Of course, used in proofs.)



Fractal Tree Inserts (and Deletes)

	B-tree	Streaming B-tree
Insert	$O(\log_B N) = O\left(\frac{\log N}{\log B}\right)$	$O\left(\frac{\log N}{B}\right)$

Example: *N*=1 billion, *B*=4096

- 1 billion 128-byte rows (128 gigabytes)
  - $\log_2$ (1 billion) = 30
- Half-megabyte blocks that hold 4096 rows each
  - $\log_2$ (4096) = 12

Fractal Tree Inserts (and Deletes)

	B-tree	Streaming B-tree
Insert	$O(\log_B N) = O\left(\frac{\log N}{\log B}\right)$	$O\left(\frac{\log N}{B}\right)$

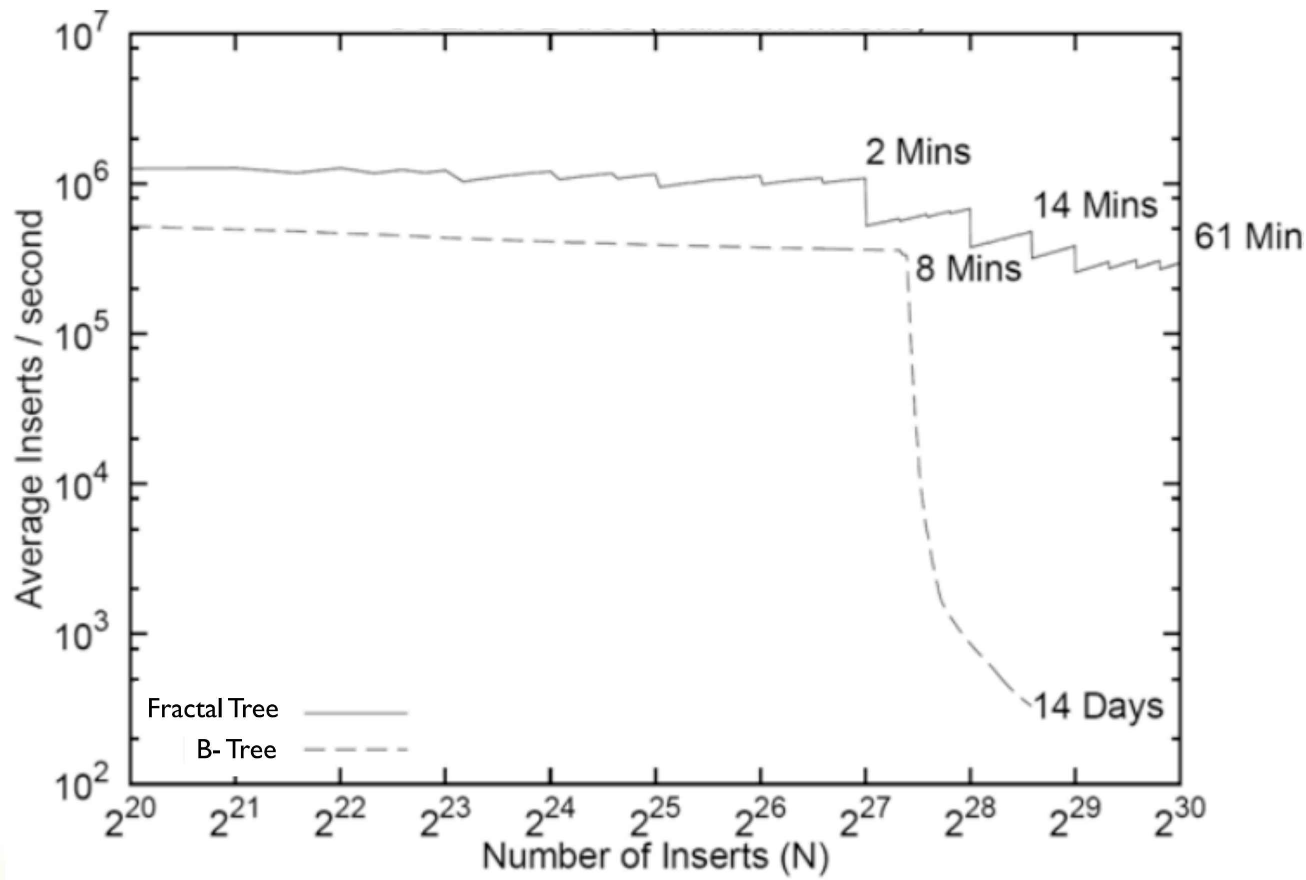
Example: *N*=1 billion, *B*=4096

- 1 billion 128-byte rows (128 gigabytes)
  - $\log_2$ (1 billion) = 30
- Half-megabyte blocks that hold 4096 rows each
  - $\log_2$ (4096) = 12

- B-trees require $\frac{\log N}{\log B}$ = 30/12 = 3 disk seeks (modulo caching, insertion pattern)
- Streaming B-trees require $\frac{\log N}{B}$ = 30/4096 = 0.007 disk seeks

Inserts into Prototype Fractal Tree

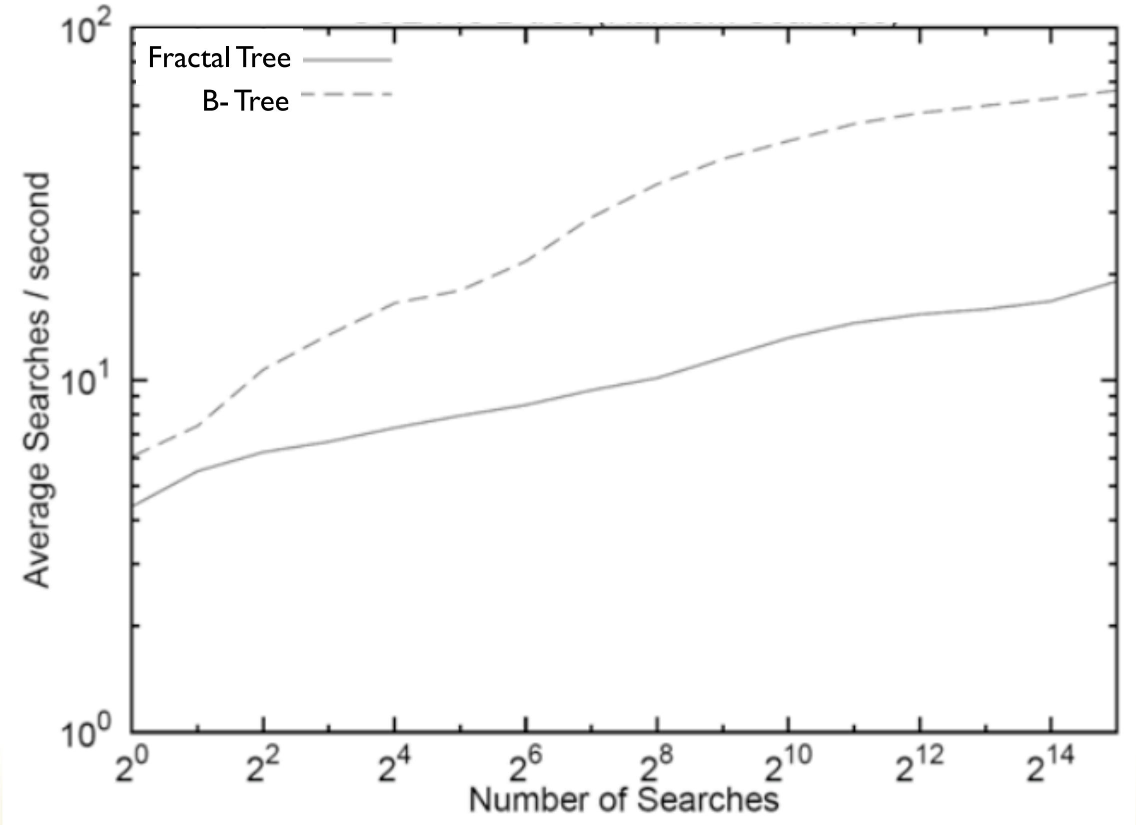
Random Inserts into Fractal Tree ("streaming B-tree") and B-tree (Berkeley DB)



Searches in Prototype Fractal Tree

Point searches ~3.5x slower ($N=2^{30}$)

- Searches/sec improves as more of data structure fits in cache)



Asymmetry Between Inserts and Key Searches

Small specification changes affect complexity

E.g., duplicate keys

- Slow: Return an error when a duplicate key is inserted
  - Hidden search
- Fast: Overwrite duplicates or maintain all versions
  - No hidden search

Asymmetry Between Inserts and Key Searches

Small specification changes affect complexity

E.g., duplicate keys

- Slow: Return an error when a duplicate key is inserted
  - Hidden search
- Fast: Overwrite duplicates or maintain all versions
  - No hidden search

E.g. deletes

- Slow: Return number of elements deleted
  - Hidden search
- Fast: Delete without feedback
  - No hidden search

Asymmetry Between Inserts and Key Searches

Small specification changes affect complexity

E.g., duplicate keys

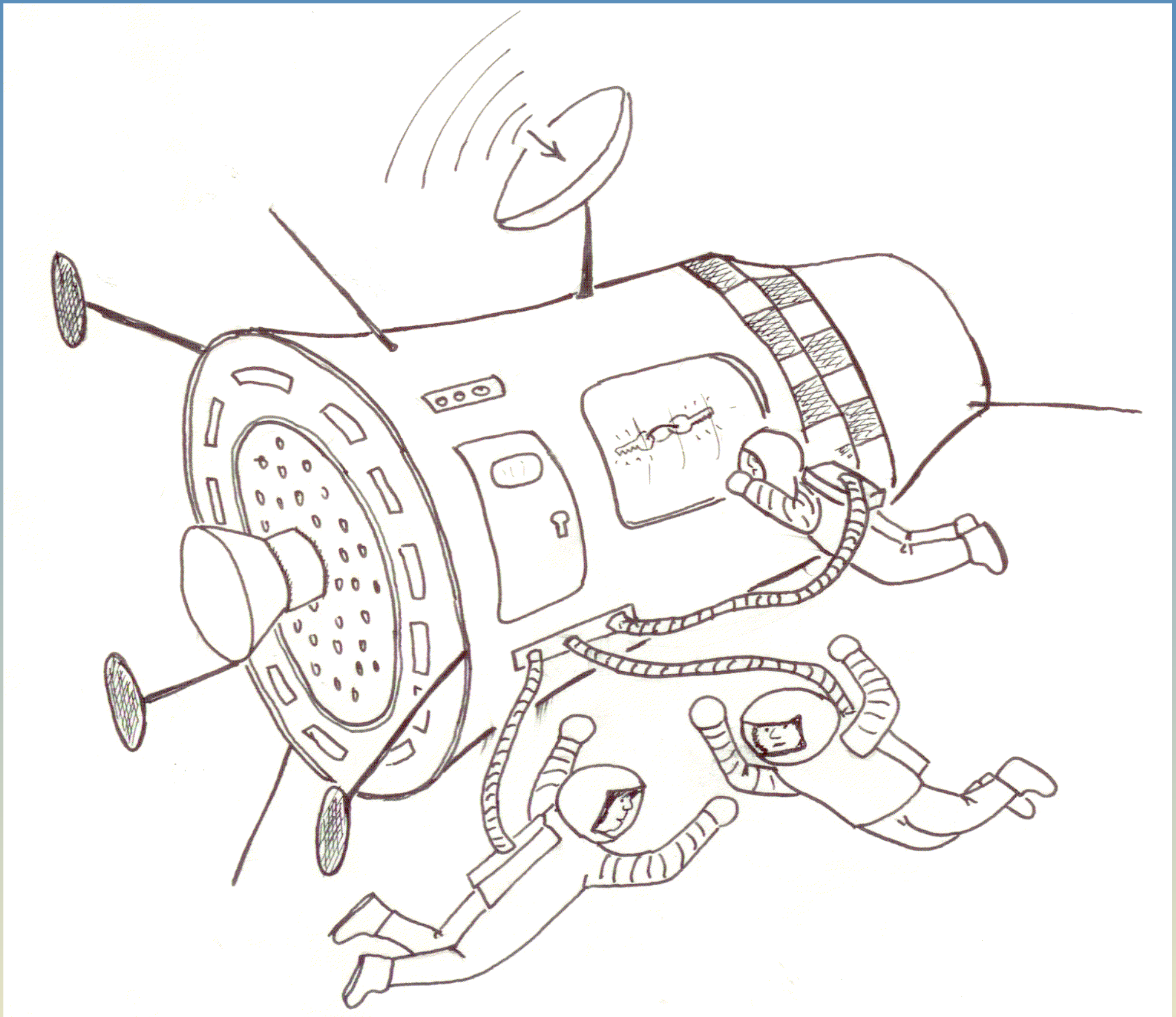
- Slow: Return an error when a duplicate key is inserted
  - Hidden search
- Fast: Overwrite duplicates or maintain all versions
  - No hidden search

E.g. deletes

- Slow: Return number of elements deleted
  - Hidden search
- Fast: Delete without feedback
  - No hidden search

Next slide: extra difficulty of key searches

Extra Difficulty of Key Searches

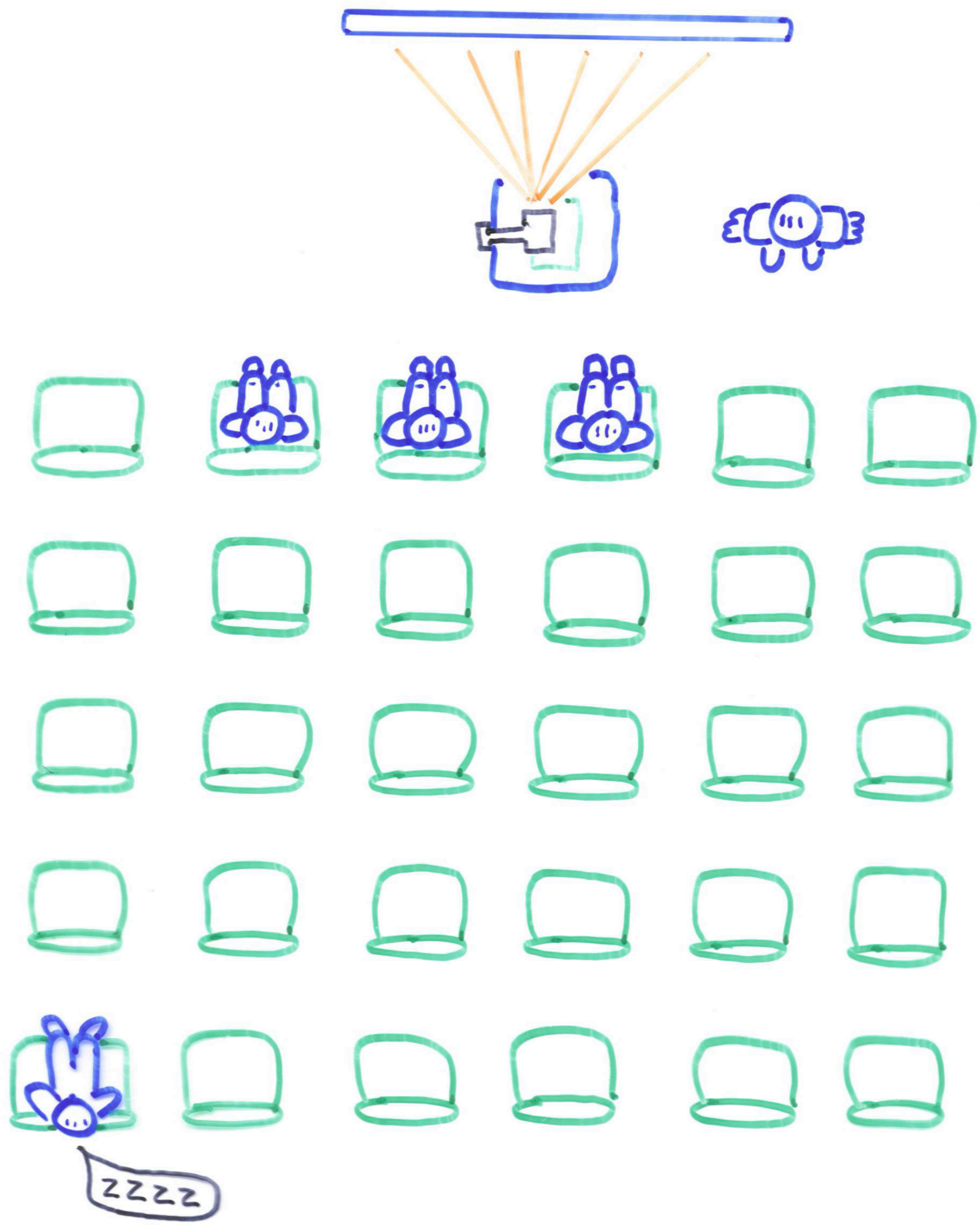


Asymmetry Between Inserts and Key Searches

Inserts/point query asymmetry has impact on

- ***System design.*** How to redesign standard mechanisms (e.g., concurrency-control mechanism).
- ***System use.*** How to take advantage of faster inserts (e.g., to enable faster queries).

Overview of Talk



Overview

External-memory dictionaries

Performance limitations of B-trees

Fractal-Tree data structure (Streaming B-tree)

Search/point-query asymmetry

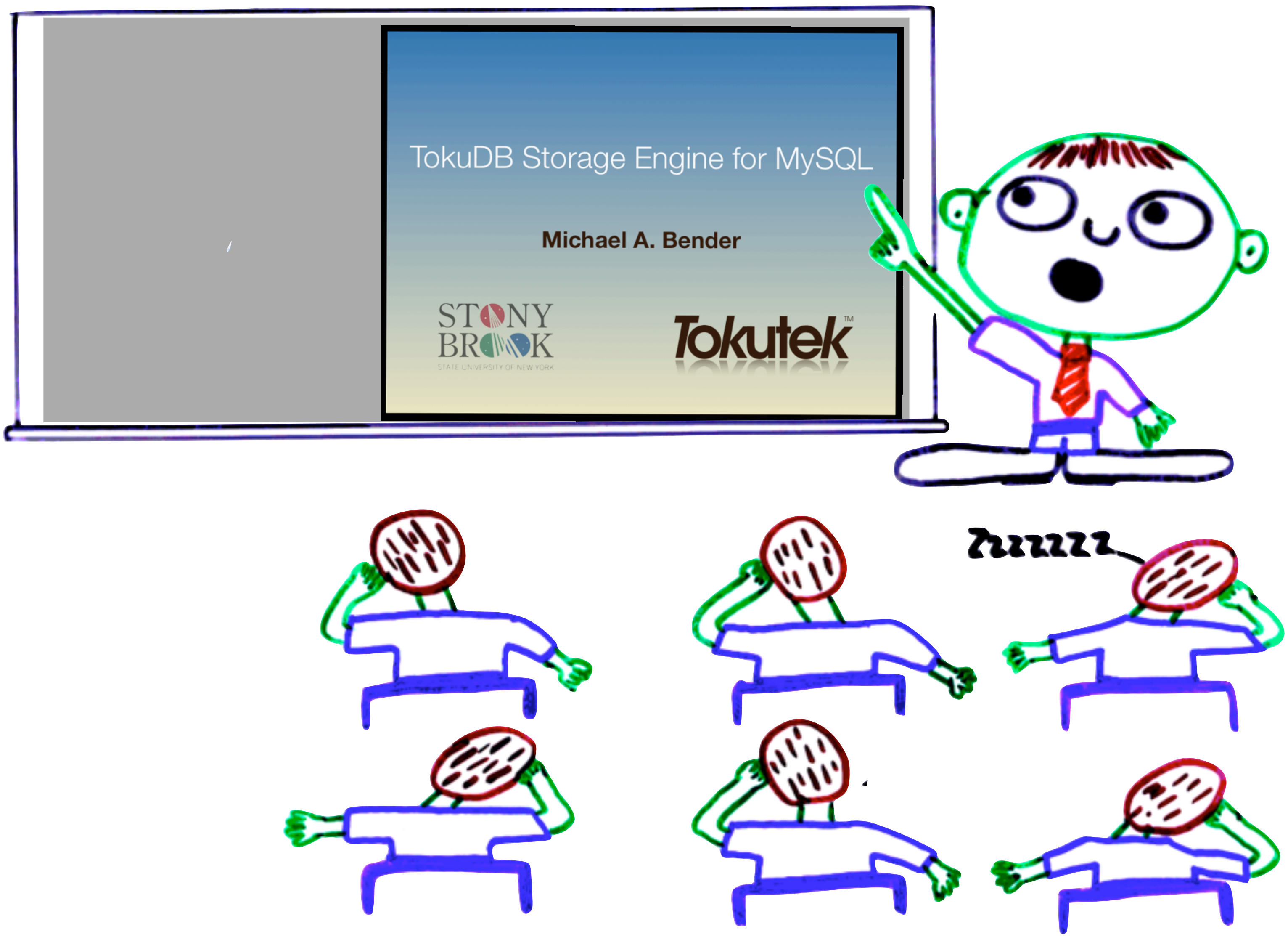
Impact of search/point-query asymmetry on database use

How to build a streaming B-tree

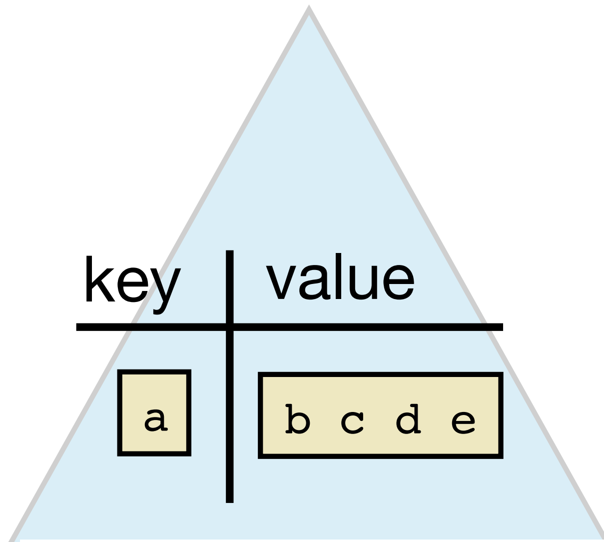
Impact of search/point-query asymmetry on system design

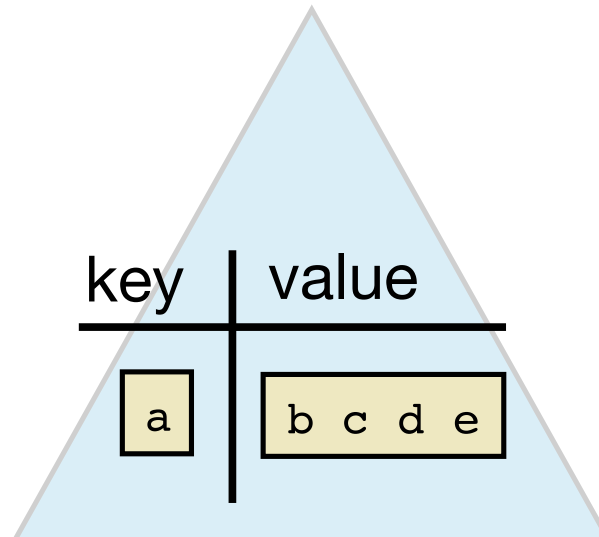
Scaling into the future

Search/point-query asymmetry affecting database use



How B-trees Are Used in Databases



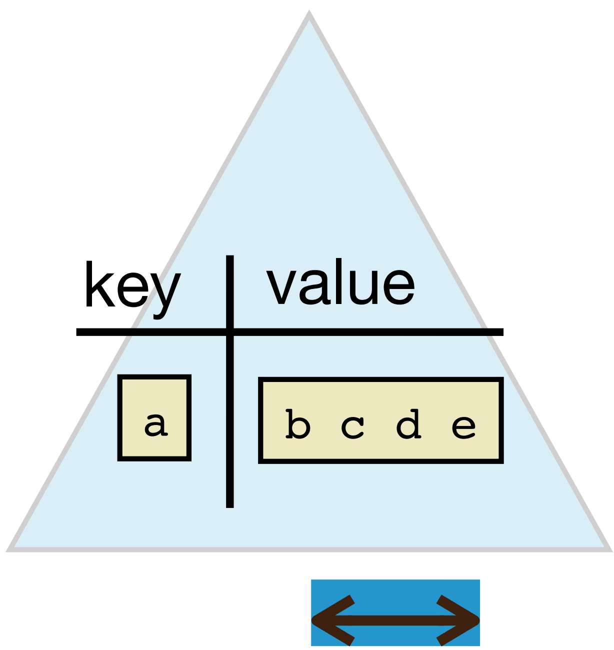


Data maintained in rows and stored in B-trees.

How B-trees Are Used in Databases

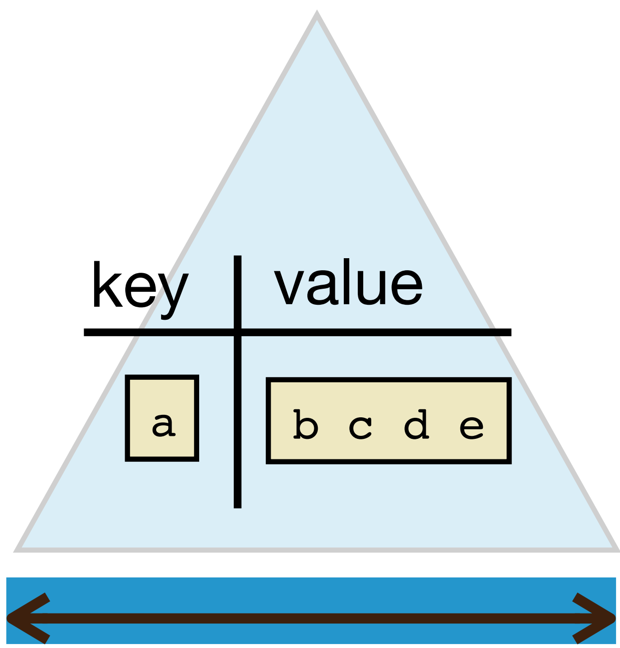
Select via Index

`select d where 270 ≤ a ≤ 538`



Select via Table Scan

`select d where 270 ≤ e ≤ 538`

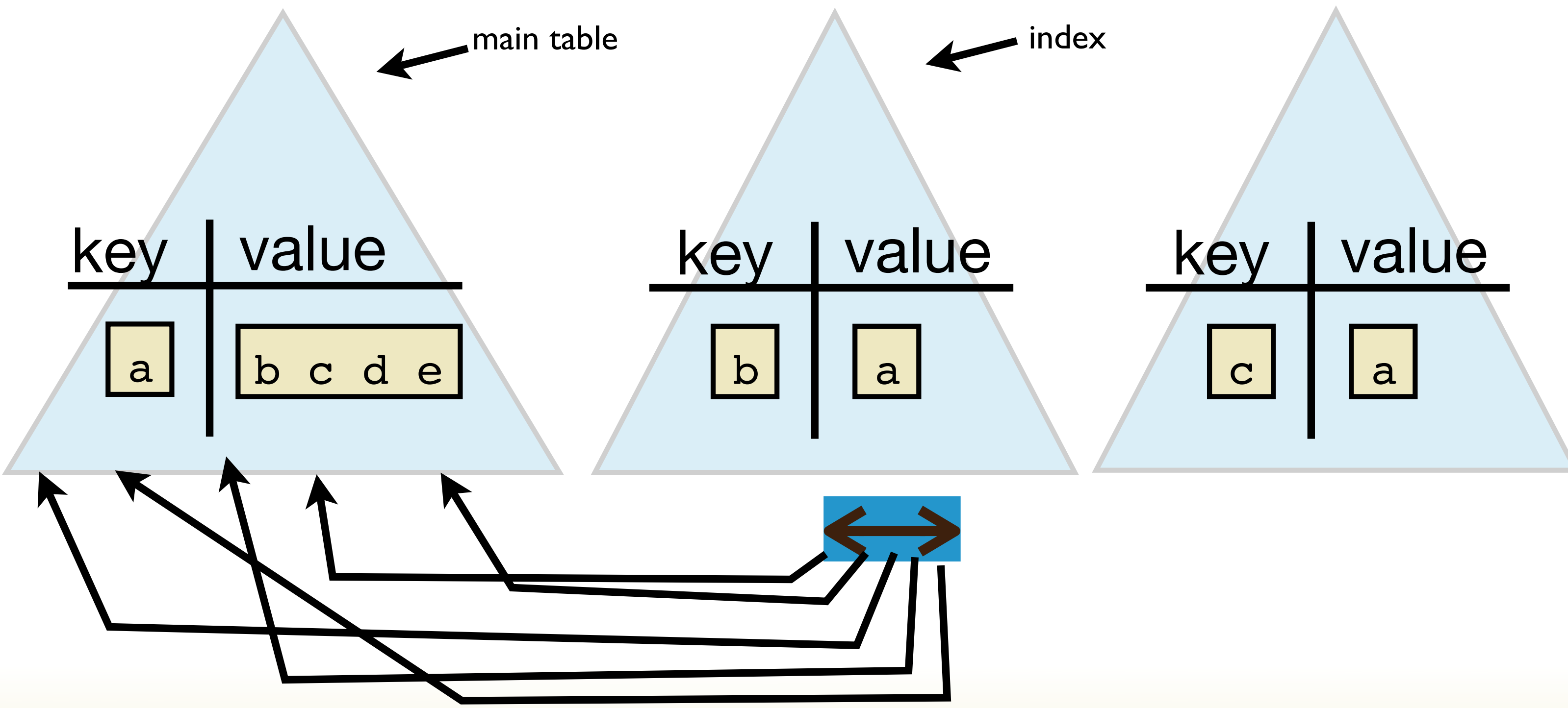


Data maintained in rows and stored in B-trees.

How B-trees Are Used in Databases (Cont.)

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

`select d where 270 ≤ b ≤ 538`

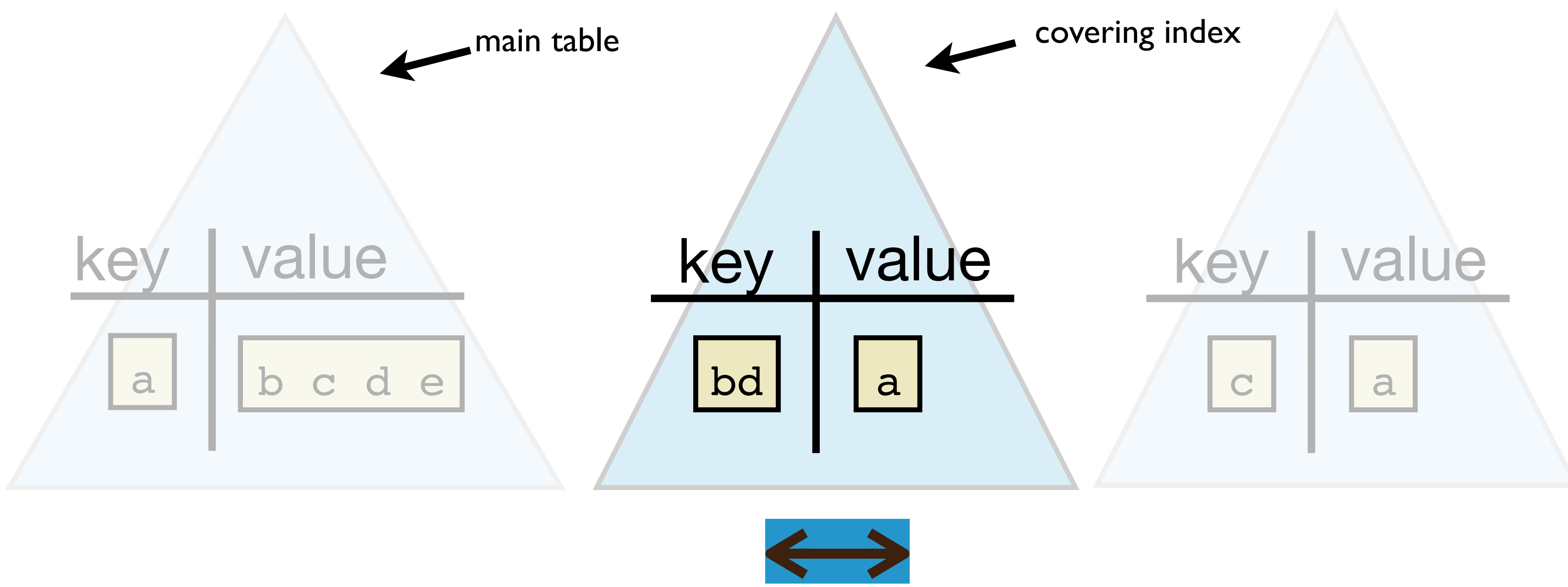


How B-trees Are Used in Databases (Cont.)

Covering index can speed up selects

- Key contains all columns necessary to answer query.

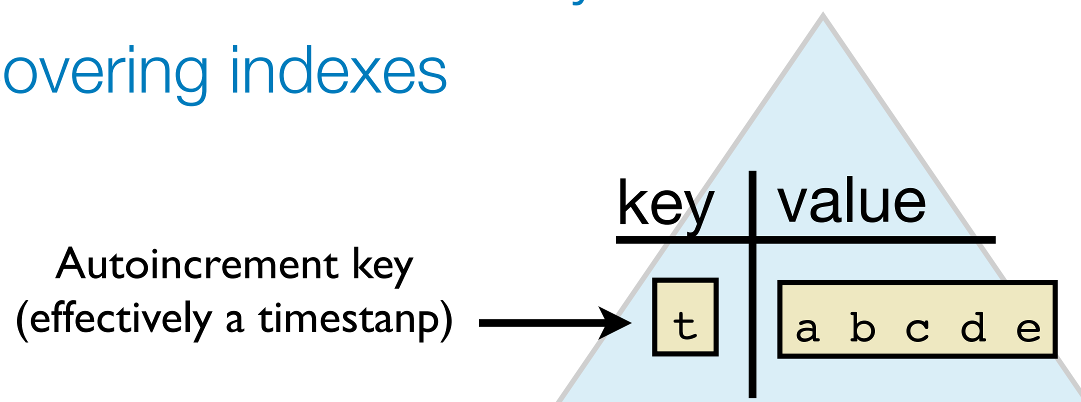
select d where 270 ≤ b ≤ 538



Insertion Pain Can Masquerade as Query Pain

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

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

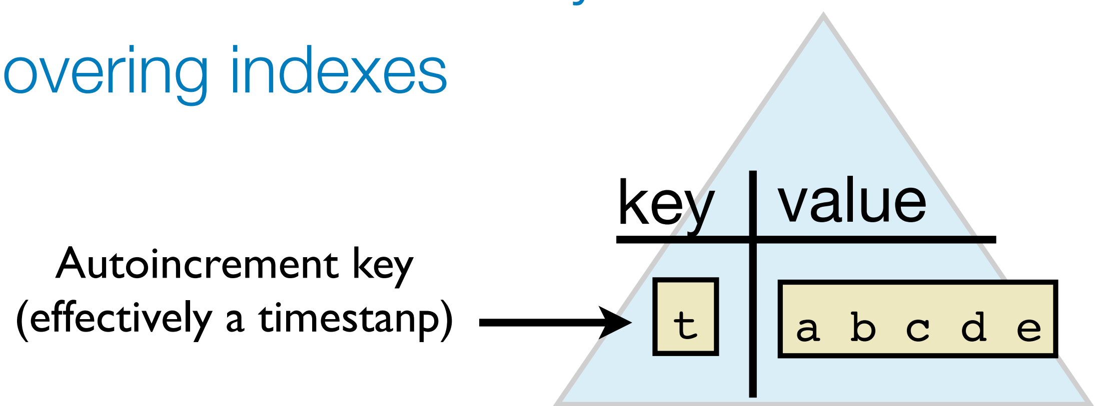


Then insertions are fast but queries are slow.

Insertion Pain Can Masquerade as Query Pain

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

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

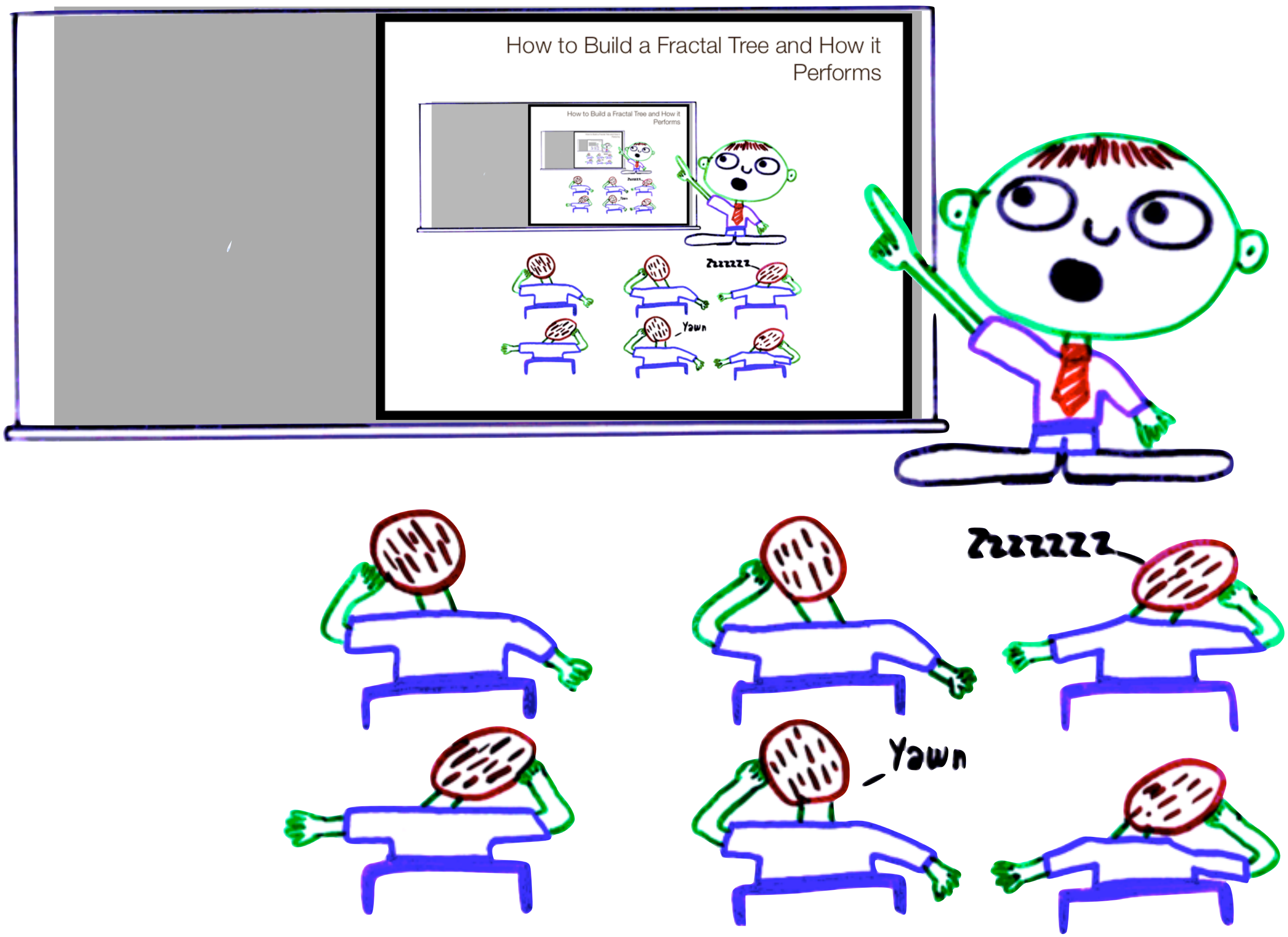


Then insertions are fast but queries are slow.

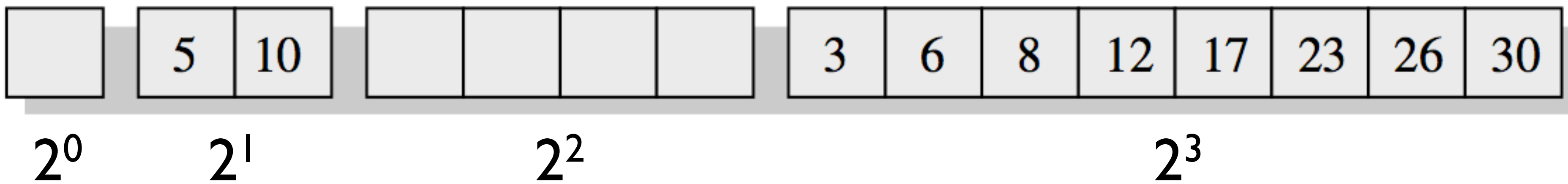
Adding sophisticated indexes helps queries

- B-trees cannot afford to maintain them.
  Fractal Trees can.

How to Build a Fractal Tree and How it Performs



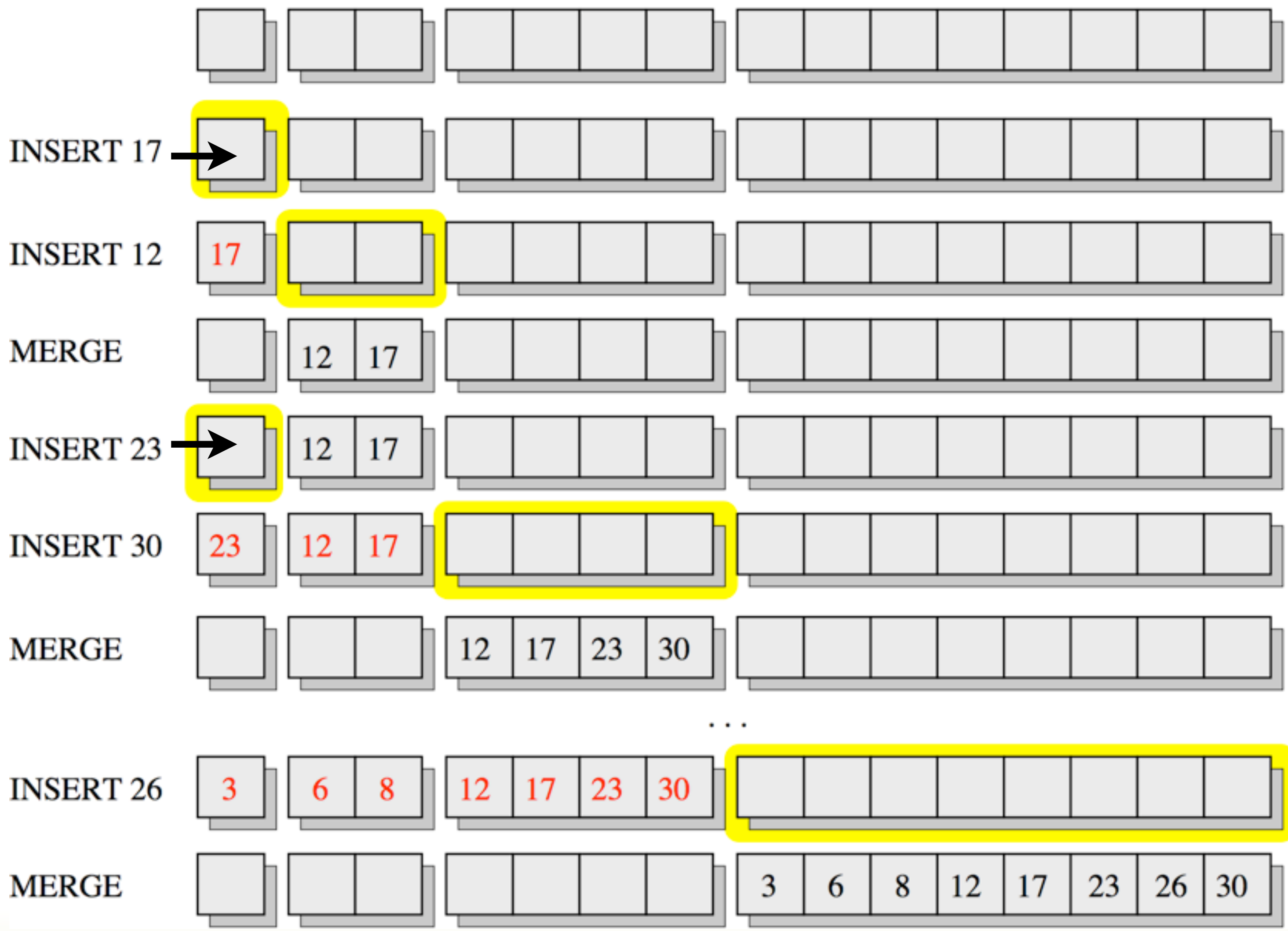
Simplified (Cache-Oblivious) Fractal Tree



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

- Sorted arrays of exponentially increasing size.
- Arrays are completely full or completely empty (depends on the bit representation of # of elmts).
- Insert into the smallest array.
  Merge arrays to make room.

Simplified (Cache-Oblivious) Fractal Tree (Cont.)



Analysis of Simplified Fractal Tree

17	5	10	13	41	57	90	3	6	8	12	17	23	26	30

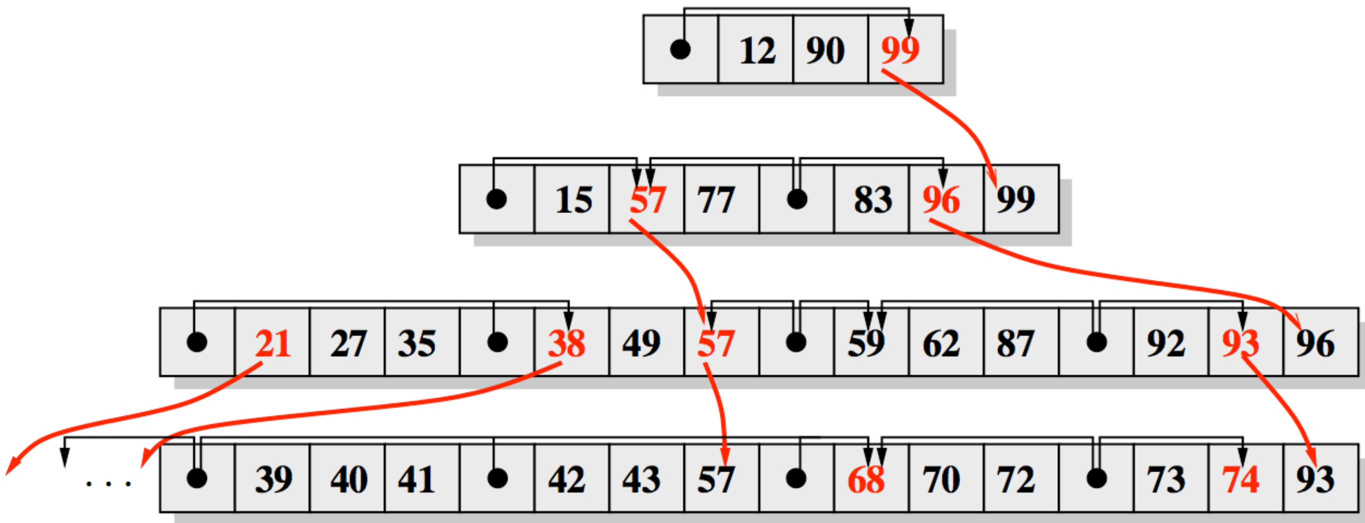
Insert Cost:

- 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 Fractal Tree



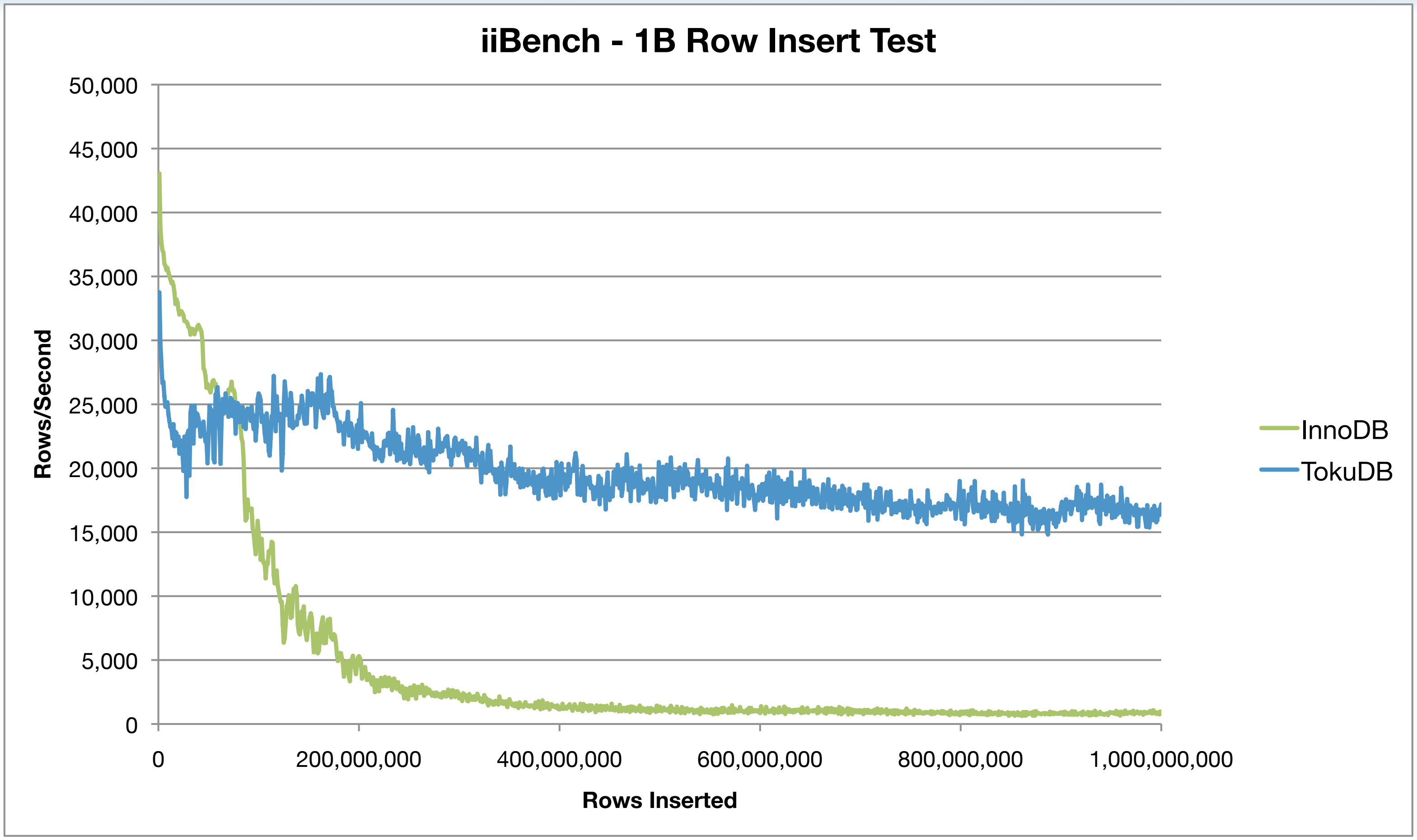
O(log (*N*/*B*)) search cost

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

Why The Previous Data Structure is a Simplification

- Need concurrency-control mechanisms
- Need crash safety
- Need transactions, logging+recovery
- Need better search cost
- Need to store variable-size elements
- Need better amortization
- Need to be good for random and sequential inserts
- Need to support multithreading.
- Need compression

iiBench Insertion Benchmark



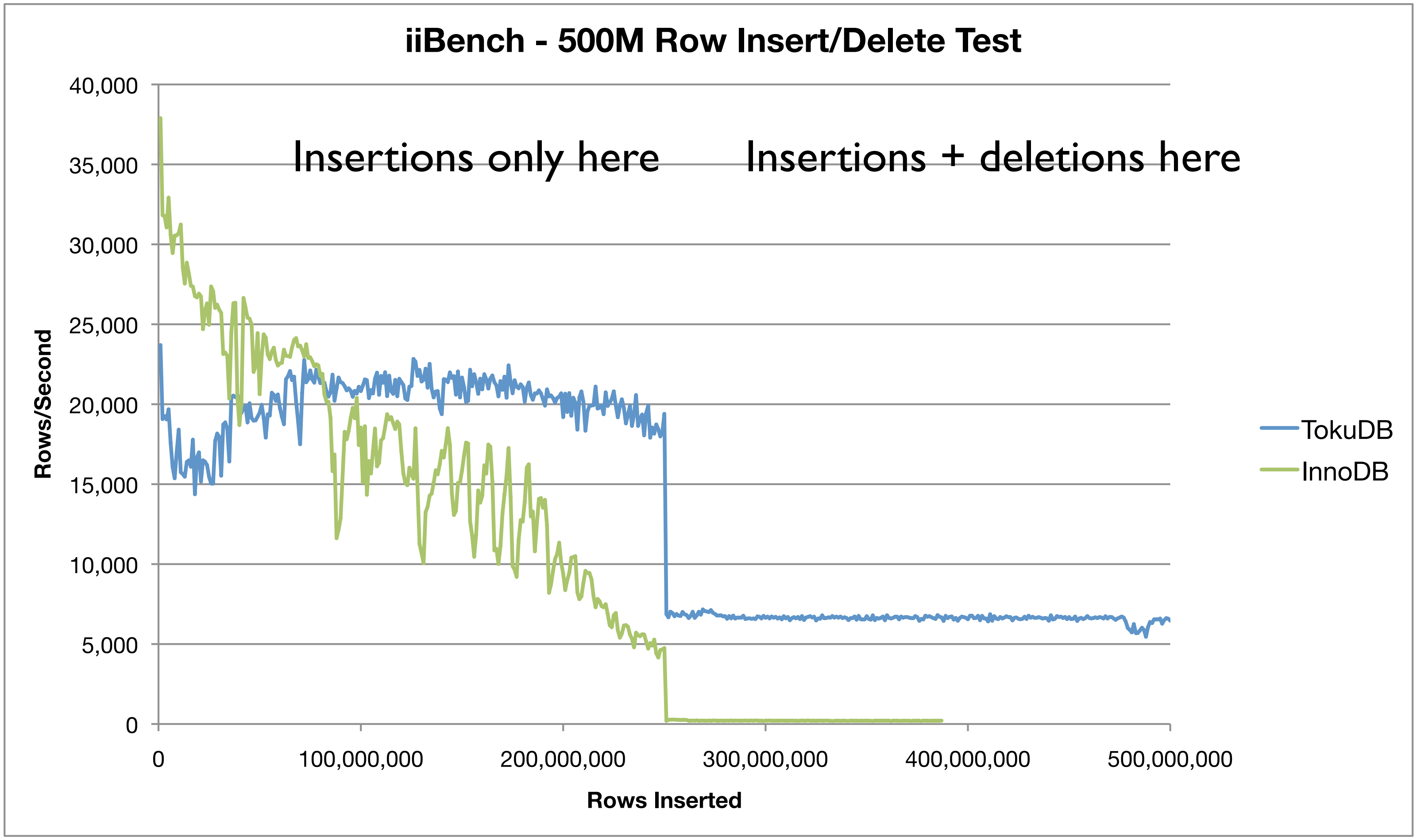
Fractal Trees scale with disk bandwidth not seek time.

- In fact, now we are compute bound, so cannot yet take full advantage of more cores or disks. (This will change.)

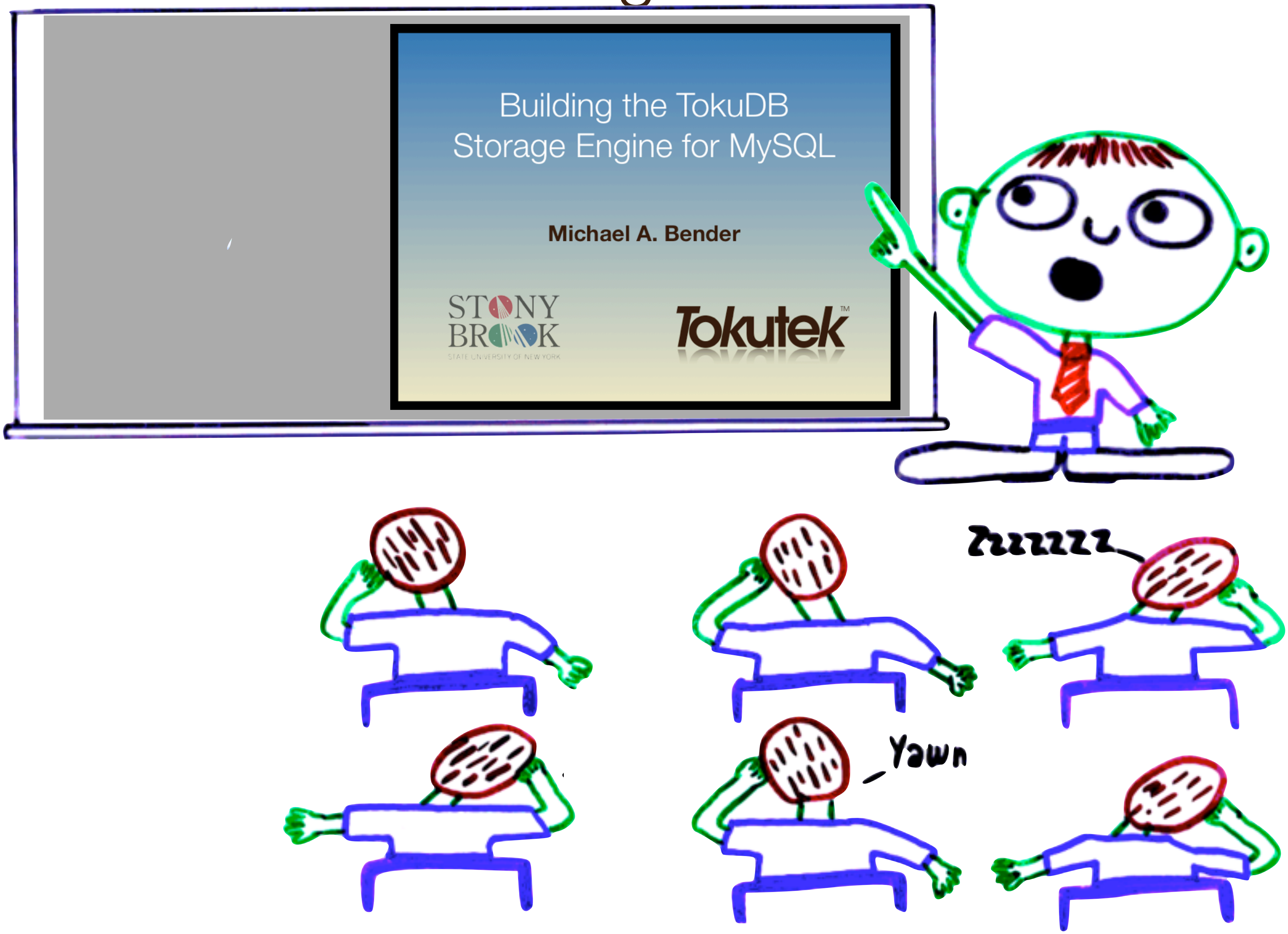
STONY BROOK

Tokutek™

iiBench Deletions



Search/point query asymmetry when building Fractal-Tree Database



Building TokuDB Storage Engine for MySQL

Engineering to do list

- Need concurrency-control mechanisms
- Need crash safety
- Need transactions, logging+recovery
- Need better search cost
- Need to store variable-size elements
- Need better amortization
- Need to be good for random and sequential inserts
- Need to support multithreading.
- Need compression

Building TokuDB Storage Engine for MySQL

Engineering to do list

- **Need concurrency-control mechanisms**
- Need crash safety
- Need transactions, logging+recovery
- Need better search cost
- Need to store variable-size elements
- Need better amortization
- Need to be good for random and sequential inserts
- Need to support multithreading.
- Need compression

Concurrency Control for Transactions

Transactions

- Sequence of durable operations.
- Happen atomically.

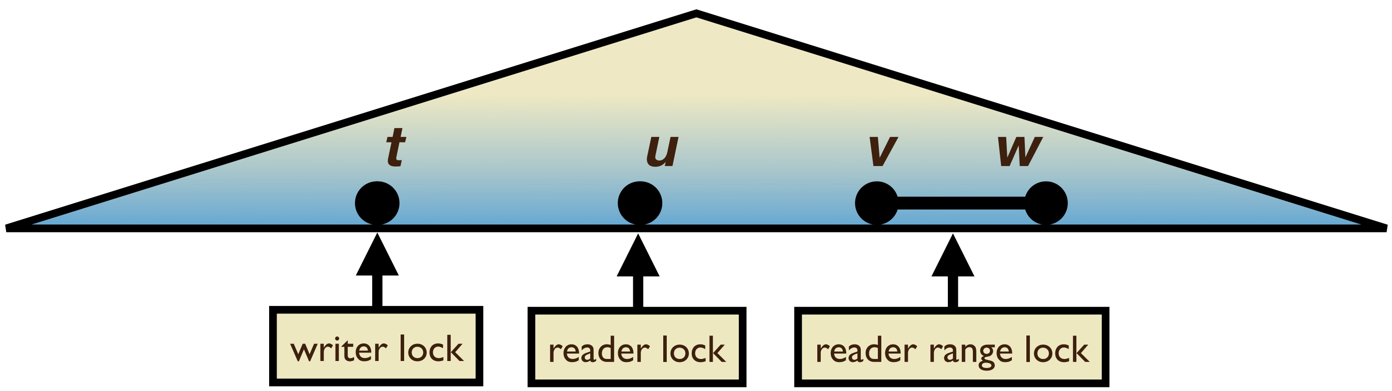
Atomicity in TokuDB via pessimistic locking

- **readers lock:** ***A*** and ***B*** can both read row ***x*** of database.
- **writers lock:** if ***A*** writes to row ***x***, ***B*** cannot read ***x*** until ***A*** completes.

Concurrency Control for Transactions (cont)

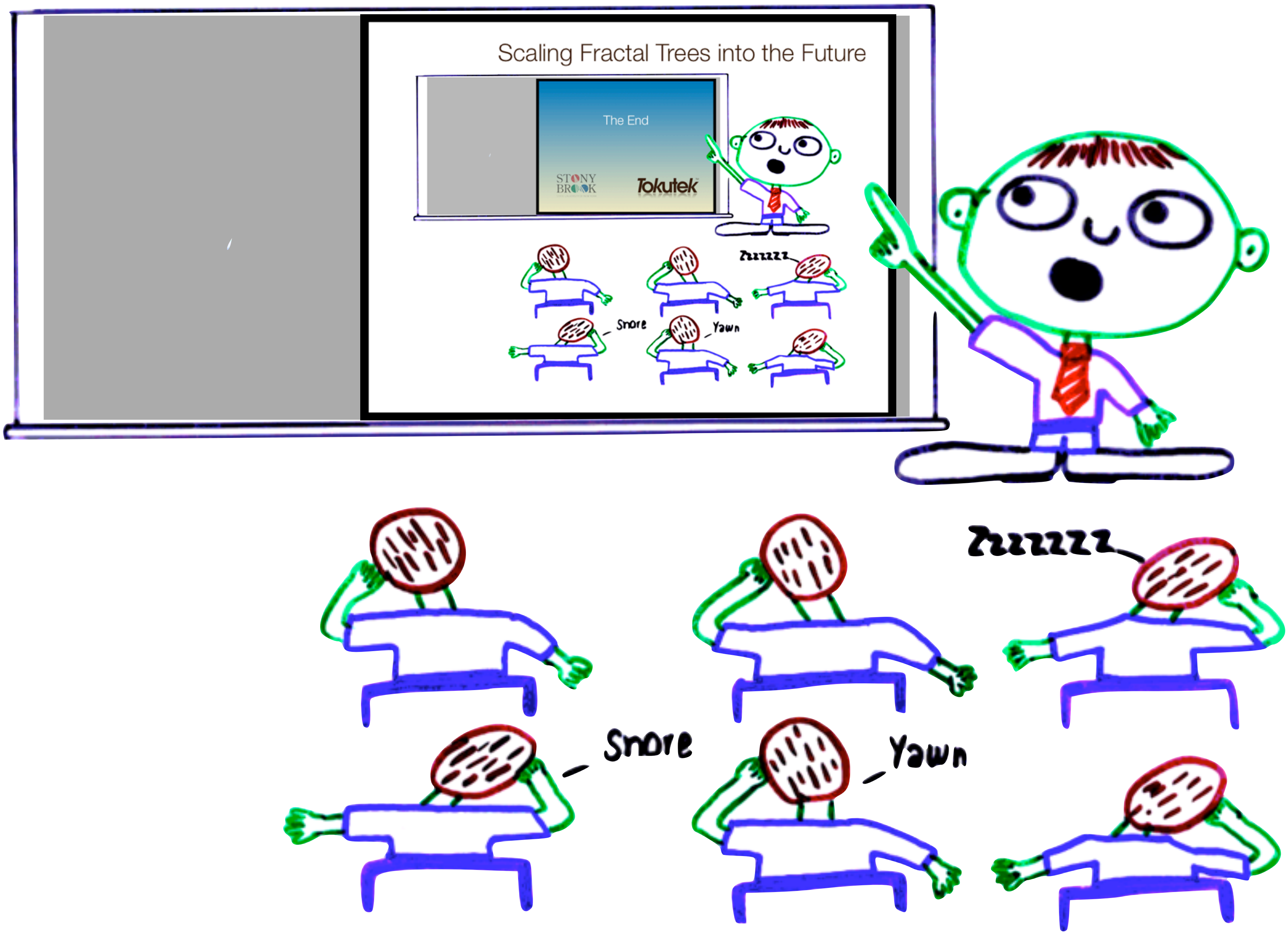
B-tree implementation: maintain locks in leaves

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

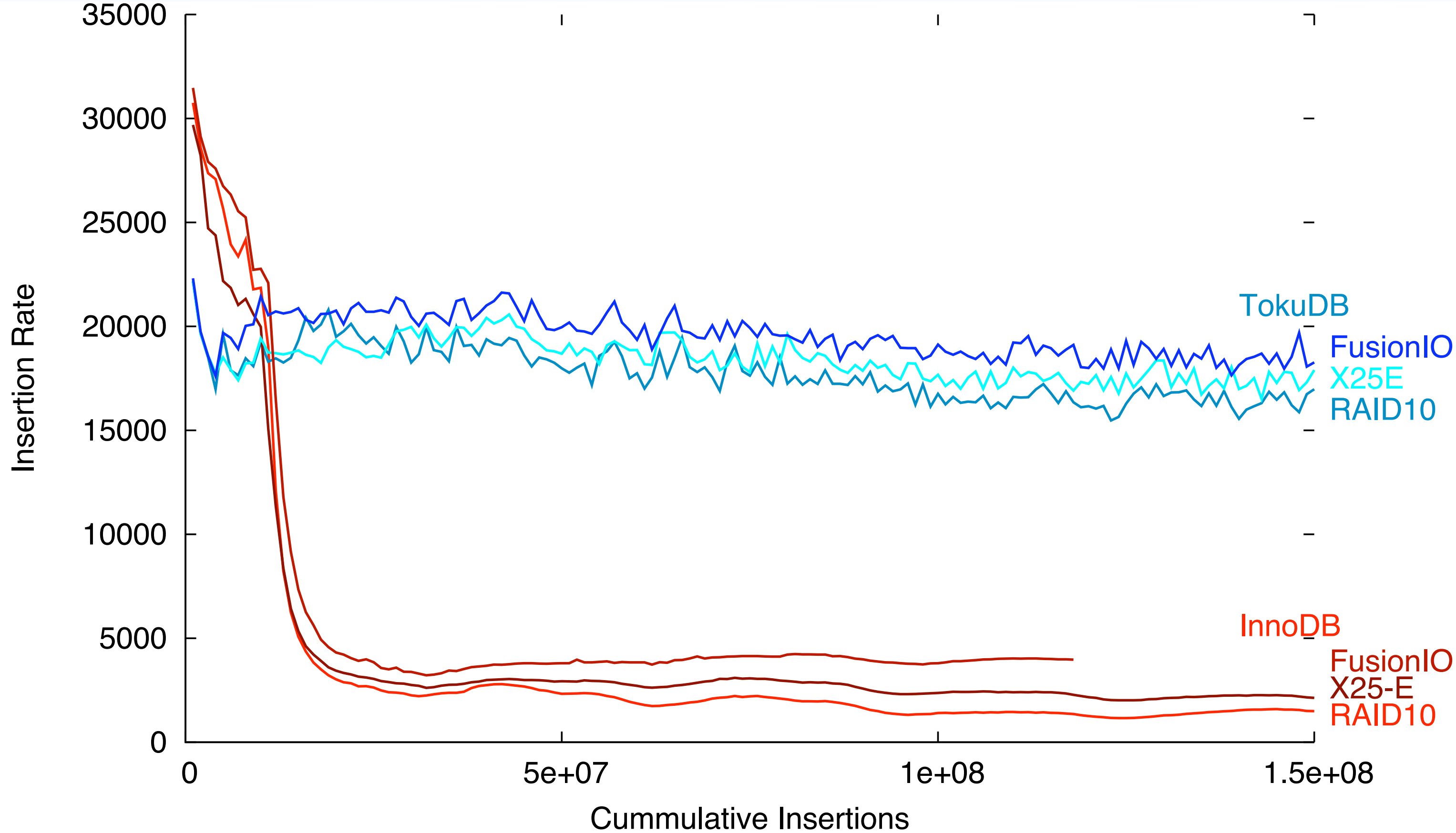


Doesn't work for Fractal Trees: maintaining locks involves implicit searches on writes.

Scaling Fractal Trees into the Future



iiBench on SSD

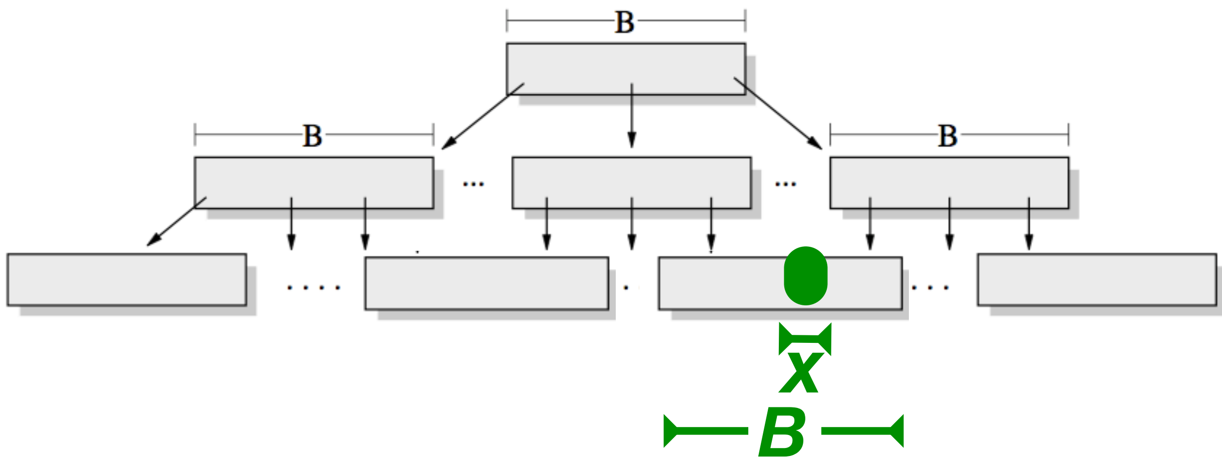


B-trees are slow on SSDs, probably b/c they waste bandwidth.

- When inserting one row, a whole block (much larger) is written.

B-tree Inserts Are Slow on SSDs

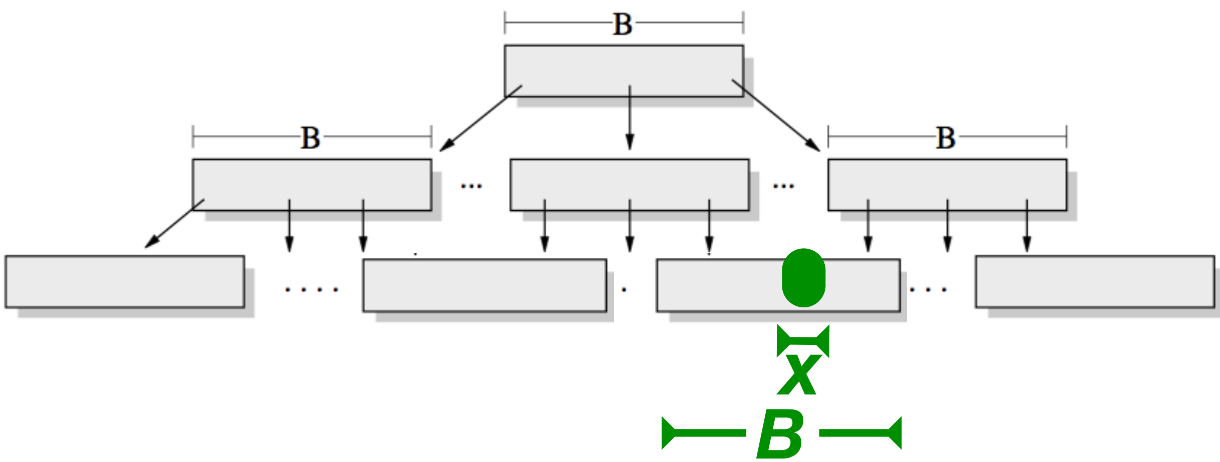
Inserting an element of size *x* into a B-tree dirties a leaf block of size *B*.



We can write keys of size *x* into a B-tree using at most a O(*x*/*B*) fraction of disk bandwidth.

B-tree Inserts Are Slow on SSDs

Inserting an element of size *x* into a B-tree dirties a leaf block of size *B*.



We can write keys of size *x* into a B-tree using at most a O(*x*/*B*) fraction of disk bandwidth.

Fractal trees do efficient inserts on SSDs because they transform random I/O into sequential I/O.

Disk Hardware Trends

Disk capacity will continue to grow quickly

Year	Capacity	Bandwidth
2008	2 TB	100MB/s
2012	4.5 TB	150MB/s
2017	67 TB	500MB/s

but seek times will change slowly.

- Bandwidth scales as square root of capacity.

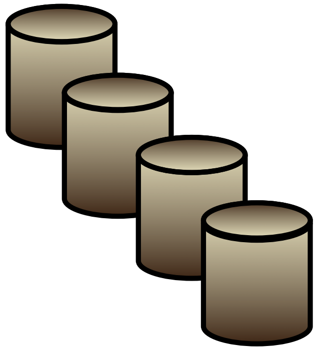
Source: http://blocksandfiles.com/article/4501

Tokutek™

Fractal Trees Enable Compact Systems

B-trees require capacity, bandwidth, and random I/O

- B-tree based systems achieve large random I/O rates by using more spindles and lower capacity disks.



Fractal Trees require only capacity & bandwidth

- Fractal Trees enable the use of high-capacity disks.

Fractal Trees Enable Big Disks

B-trees require capacity, bandwidth, and seeks.

Fractal trees require only capacity and bandwidth.

Today, for a 50TB database,

- Fractal tree with 25 2TB disks gives 500K ins/s.
- B-tree with 25 2TB disks gives 2.5K ins/s.
- B-tree with 500 100GB disks gives 50K ins/s but costs $, racks, and power.

In 2017, for a 1500TB database:

- Fractal tree with 25 67TB disks gives 2500K ins/s.
- B-tree with 25 67TB disks gives 2.5K ins/s.

B-trees need spindles, and spindle density increases slowly.

Tokutek™

Using Big Disks Also Saves Energy

Power consumption of disks

- Enterprise 80 to 160 GB disk runs at 4W (idle power).
- Enterprise 1-2 TB disk runs at 8W (idle power).

Data centers/server farms use 80-160 GB disks

- Use many small-capacity disks, not large ones.

Using large disks may save factor >10 in Storage Costs

- Other considerations modify this factor
  - e.g., CPUs necessary to drive disks, scale-out infrastructure, cooling, etc.
  - Metric: e.g., Watts/MB versus Inserts/Joule