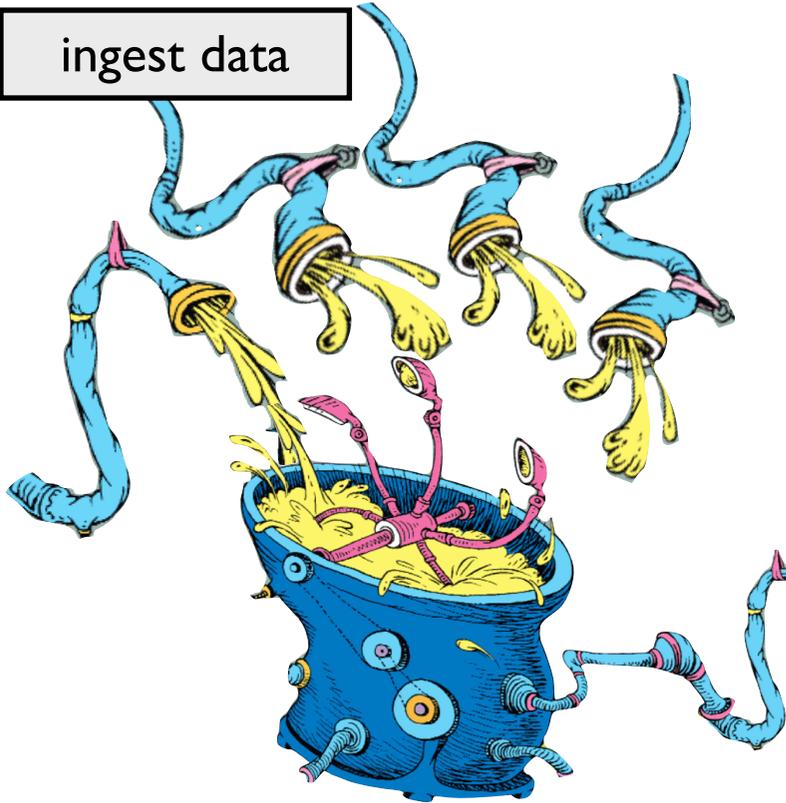


Indexing Big Data

Michael A. Bender

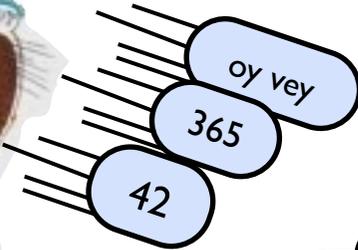
ingest data



organize data on disks

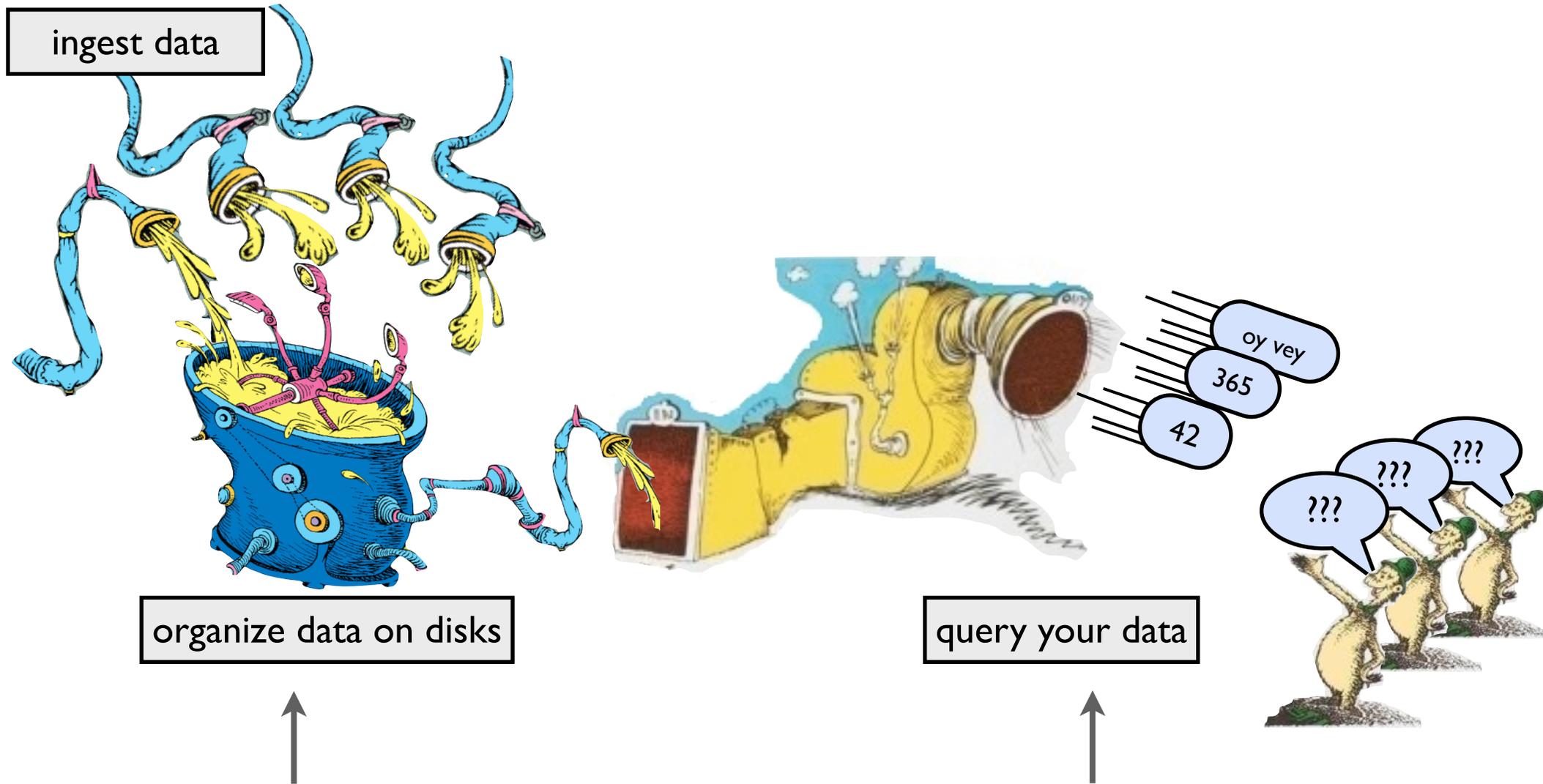


query your data



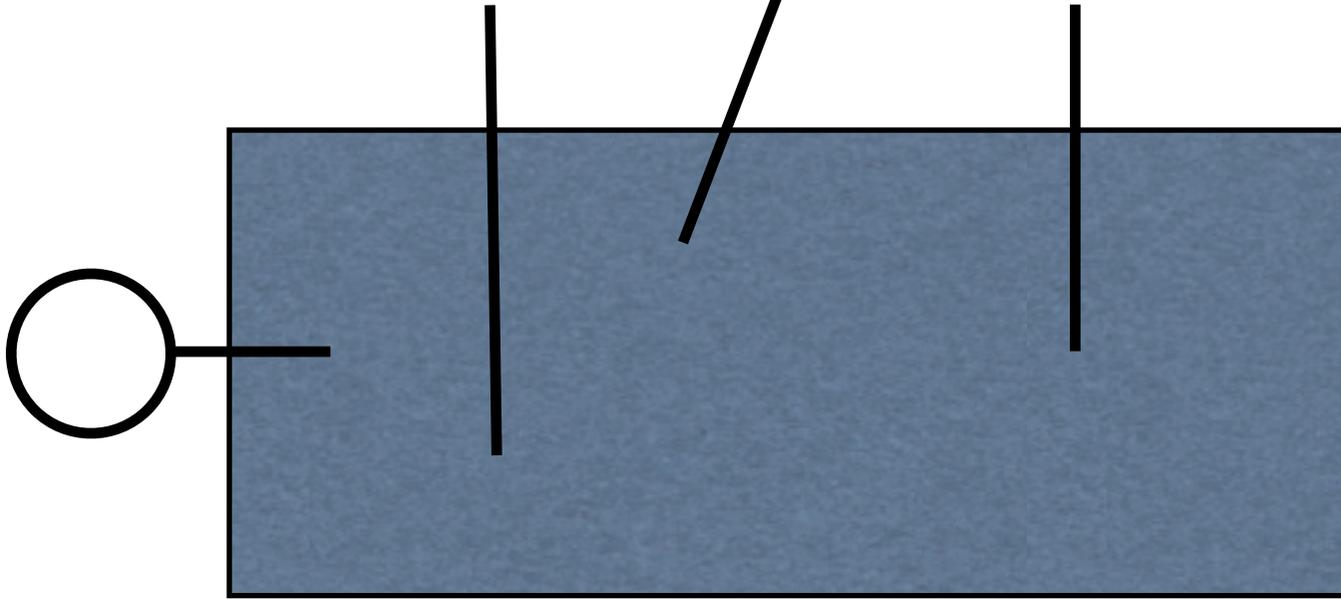
Indexing Big Data

Michael A. Bender



Goal: Index big data so that it can be queried quickly.

Setting of Talk-- “Type II streaming”



Type II streaming: Collect and store streams of data to answer queries.

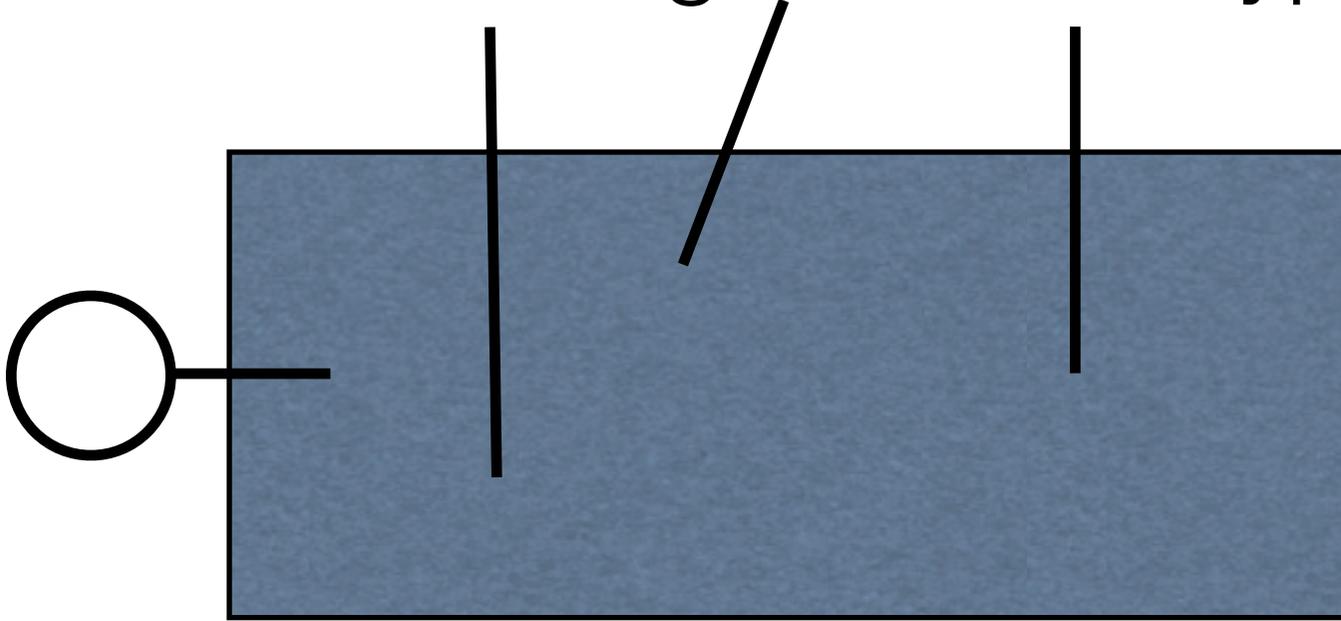
Data is stored on *disks*.

- Dealing with disk latency has similarities to dealing with network latency.

We'll discuss *indexing*.

- We want to store data so that it can be queried efficiently.
- To discuss: relationship with graphs.

Setting of Talk-- “Type II streaming”



Type II streaming: Collect and store streams of data to answer queries.

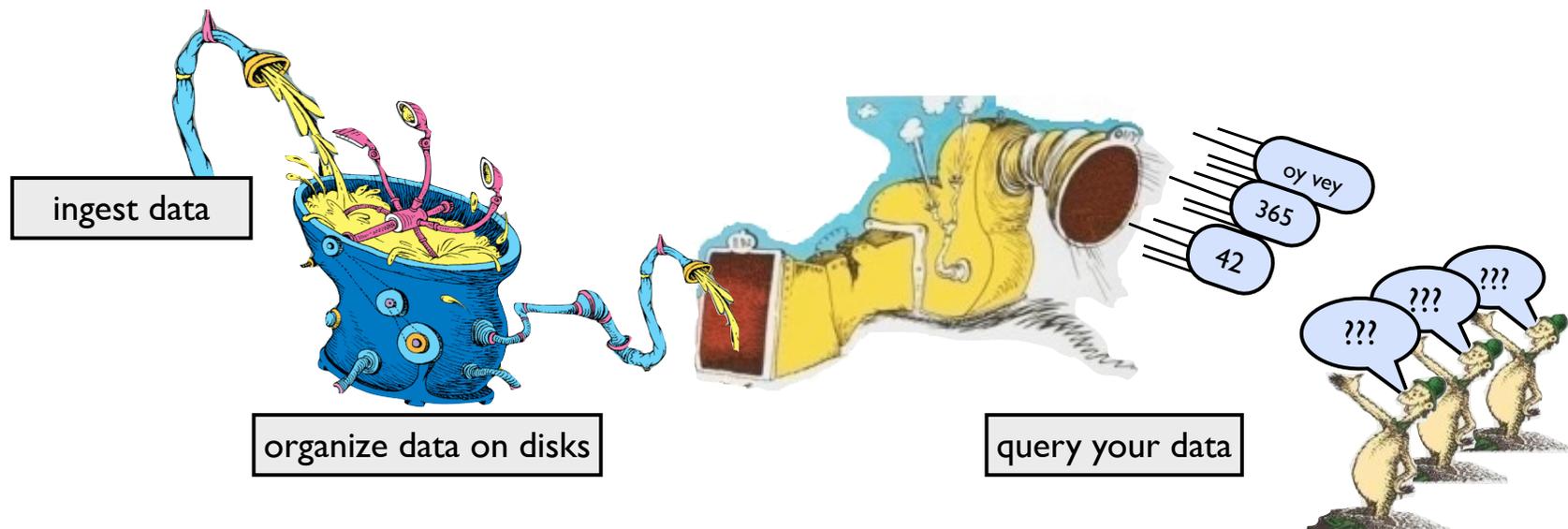
Data is stored on *disks*.

- Dealing with disk latency has similarities to dealing with network latency.

We'll discuss *indexing*.

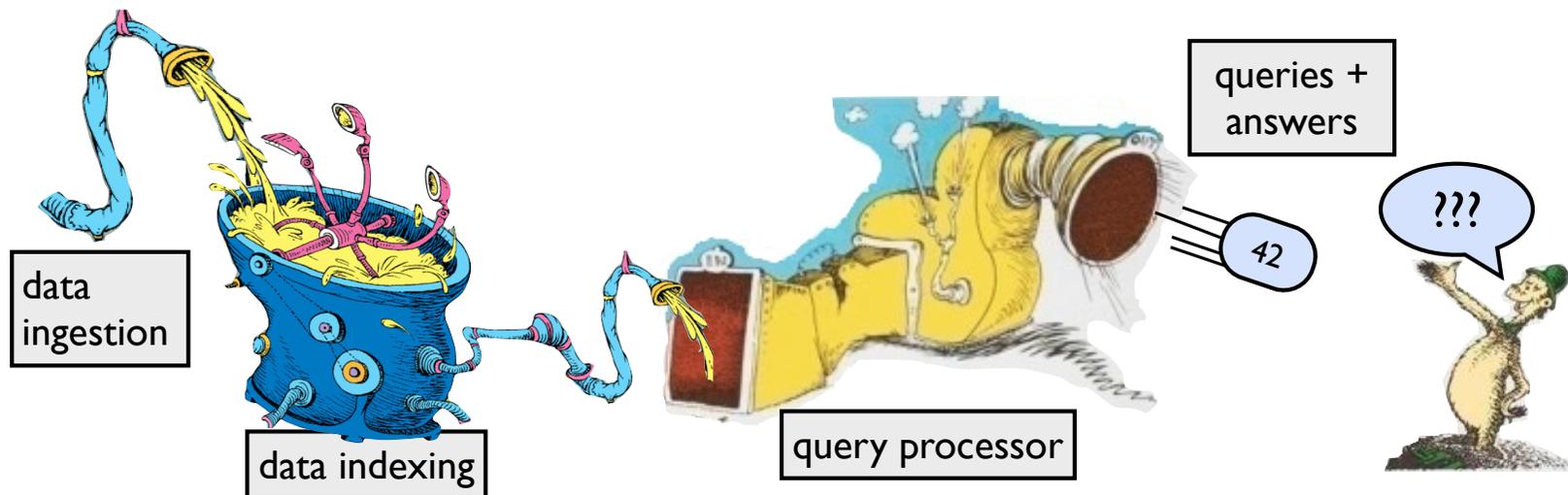
- We want to store data so that it can be queried efficiently.
- To discuss: relationship with graphs.

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



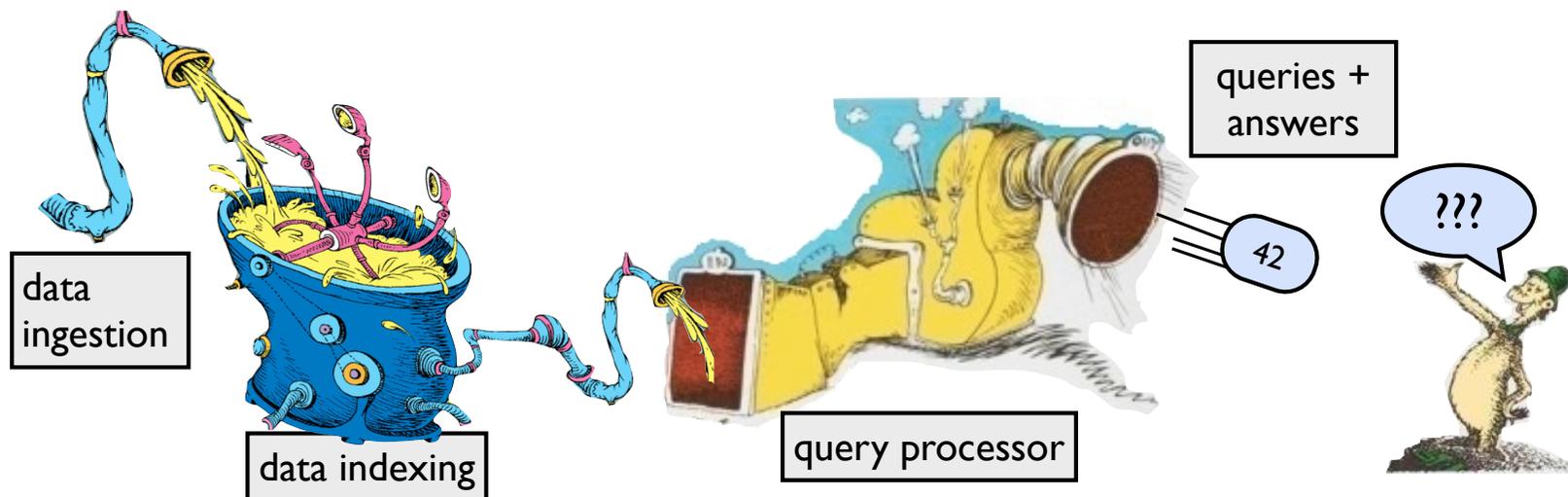
Funny tradeoff in ingestion, querying, freshness

- “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](https://www.mysqlperformanceblog.com)



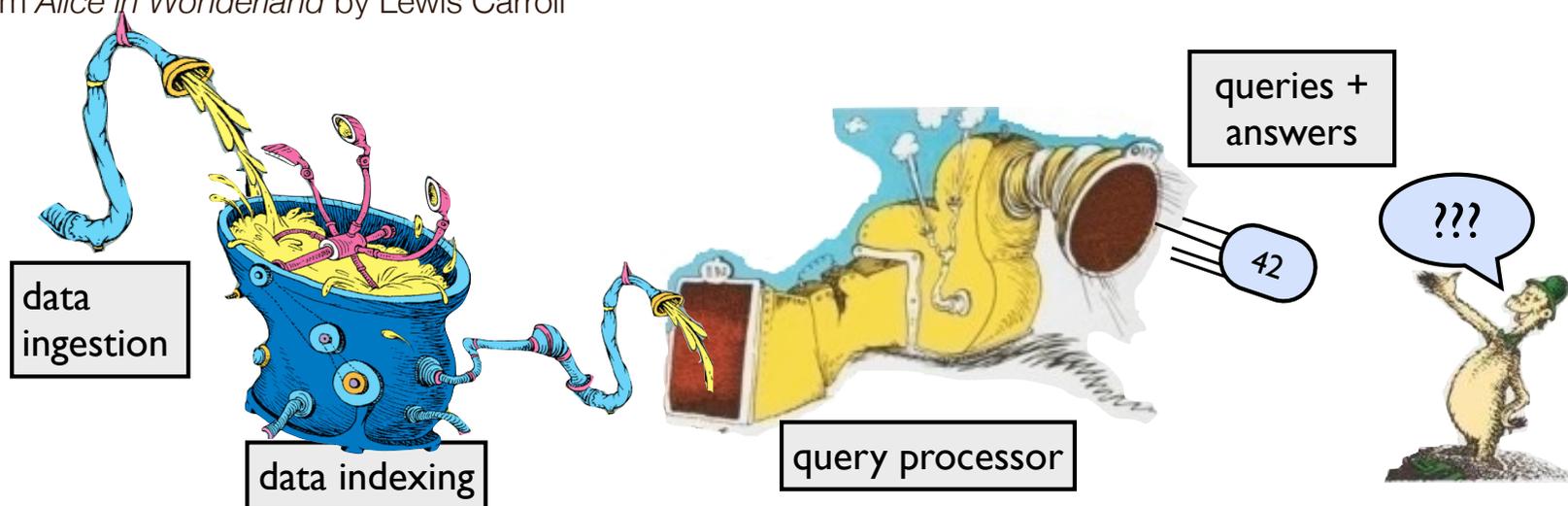
Funny tradeoff in ingestion, querying, freshness

- “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
- “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



Funny tradeoff in ingestion, querying, freshness

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



Tradeoffs come from different ways to organize data on disk

Like a
librarian?



Tradeoffs come from different ways to organize data on disk

Like a
librarian?



Fast to find stuff.
Slow to add stuff.

“Indexing”

Tradeoffs come from different ways to organize data on disk

Like a
librarian?



Fast to find stuff.
Slow to add stuff.

“Indexing”

Like a
teenager?



Tradeoffs come from different ways to organize data on disk

Like a
librarian?



Fast to find stuff.
Slow to add stuff.

“Indexing”

Like a
teenager?



Fast to add stuff.
Slow to find stuff.

“Logging”



This talk: we don't need tradeoffs

Write-optimized data structures:

- Faster indexing (10x-100x)
- Faster queries
- Fresh data

These structures efficiently scale to very big data sizes.

Our algorithmic work appears in two commercial products

Tokutek's high-performance MySQL and MongoDB.

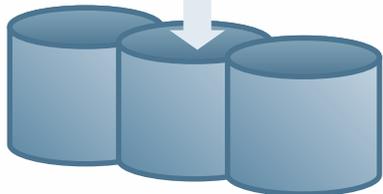
TokuDB

Application

MySQL Database
-- SQL processing,
-- query optimization

libFT

File System



Disk/SSD

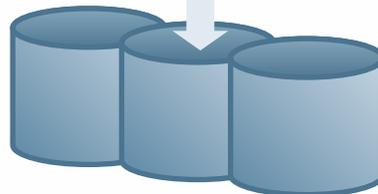
TokuMX

Application

Standard MongoDB
-- drivers,
-- query language, and
-- data model

libFT

File System



Our algorithmic work appears in two commercial products

Tokutek's high-performance MySQL and MongoDB.

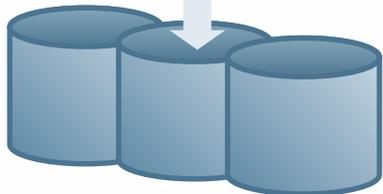
TokuDB

Application

MySQL Database
-- SQL processing,
-- query optimization

libFT

File System



Disk/SSD

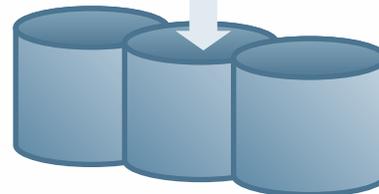
TokuMX

Application

Standard MongoDB
-- drivers,
-- query language, and
-- data model

libFT

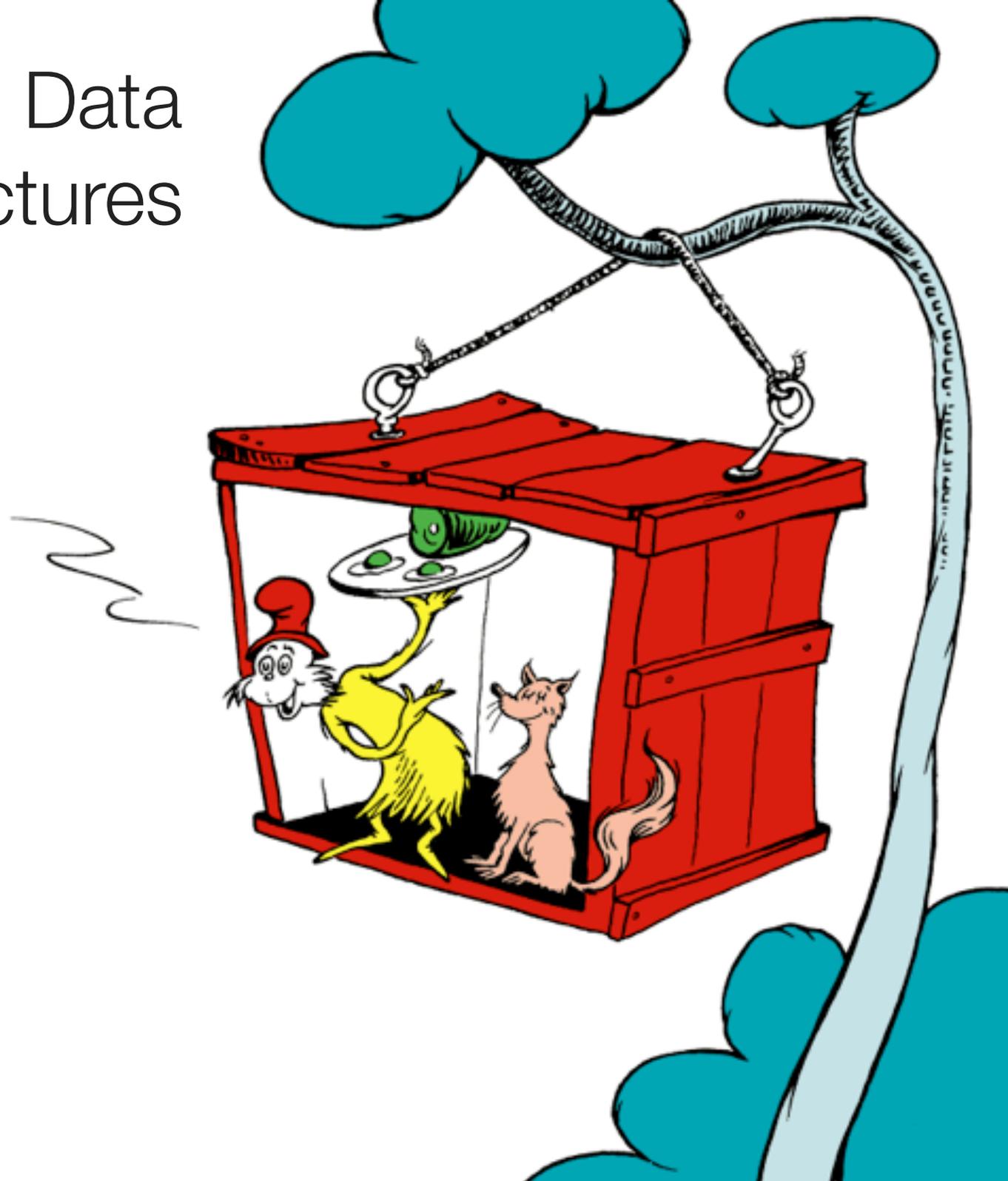
File System



The *Fractal Tree engine* implements the persistent structures for storing data on disk.



Write-Optimized Data Structures



Write-Optimized Data Structures

Would you like them with an algorithmic performance model?



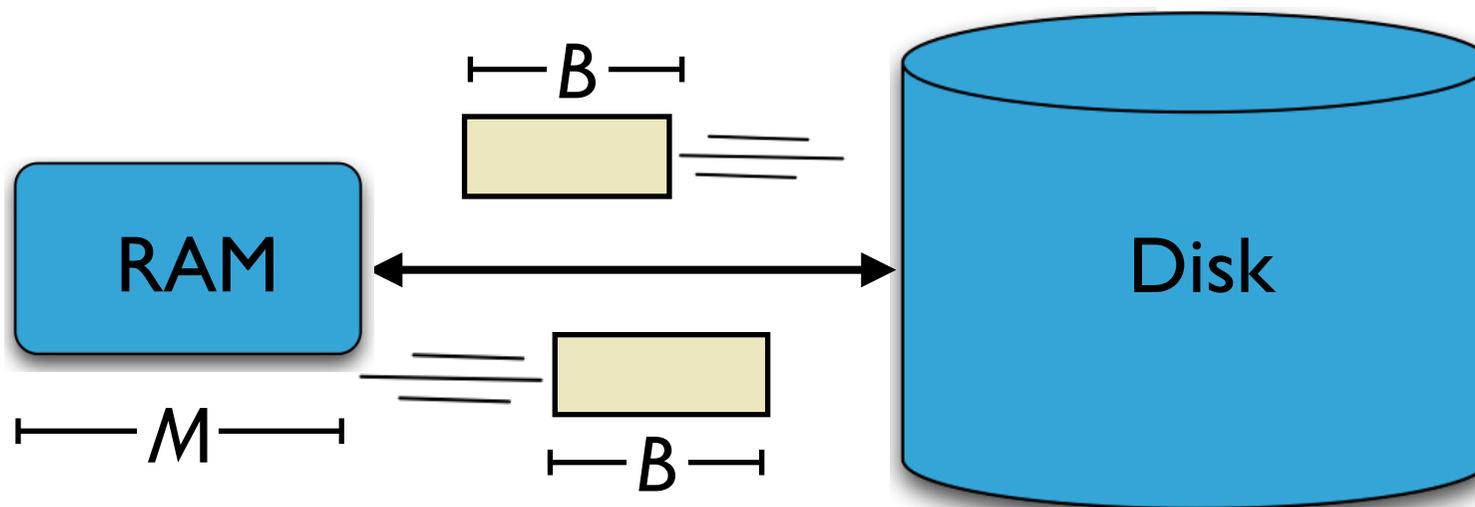
An algorithmic performance model

How computation works:

- Data is transferred in blocks between RAM and disk.
- The number of block transfers dominates the running time.

Goal: Minimize # of block transfers

- Performance bounds are parameterized by block size B , memory size M , data size N .



[Aggarwal+Vitter '88]

Memory and disk access times

Disks: ~6 milliseconds per access.

RAM: ~60 nanoseconds per access



Memory and disk access times

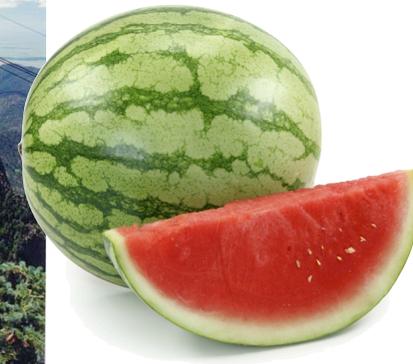
Disks: ~6 milliseconds per access.

RAM: ~60 nanoseconds per access



Analogy:

- disk = elevation of Sandia peak
- RAM = height of a watermelon



Memory and disk access times

Disks: ~6 milliseconds per access.

RAM: ~60 nanoseconds per access



Analogy:

- disk = elevation of Sandia peak
- RAM = height of a watermelon

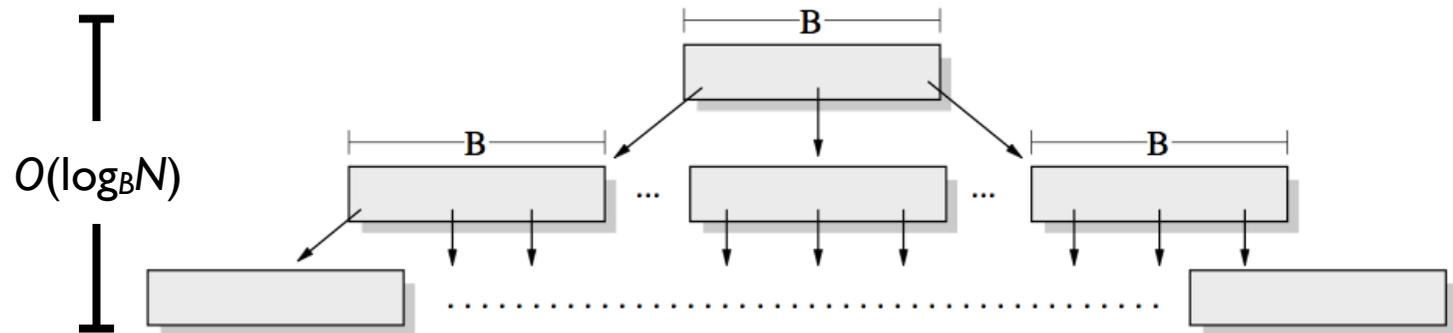
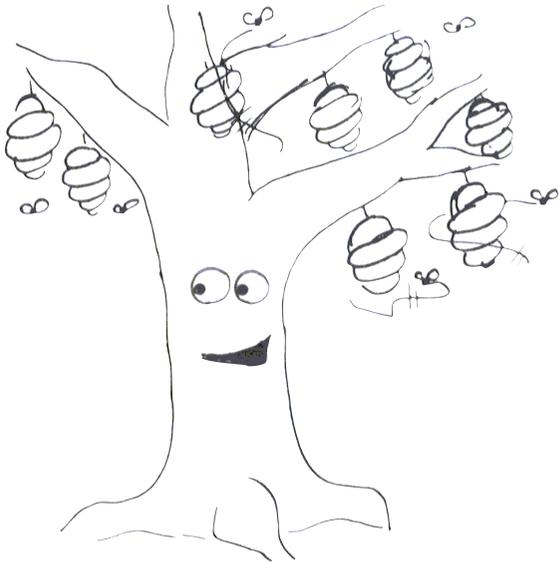


Analogy:

- disk = walking speed of the giant tortoise (0.3mph)
- RAM = escape velocity from earth (25,000 mph)



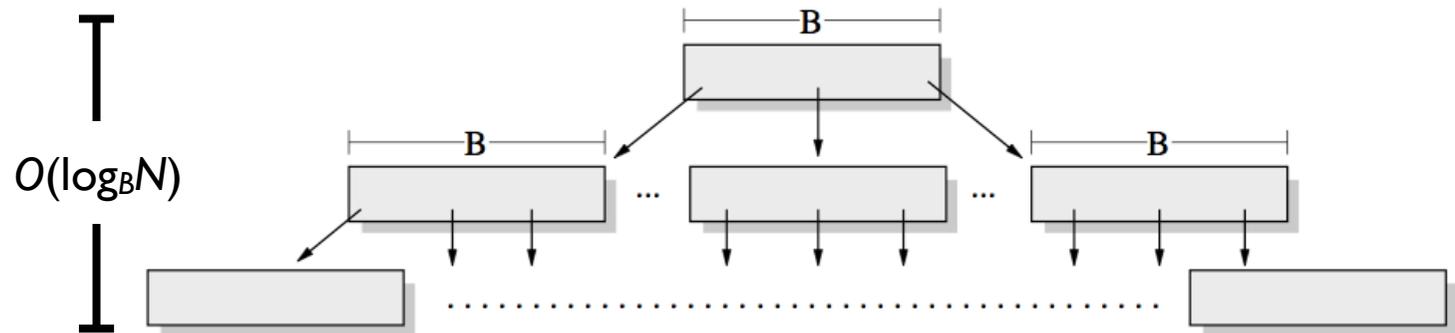
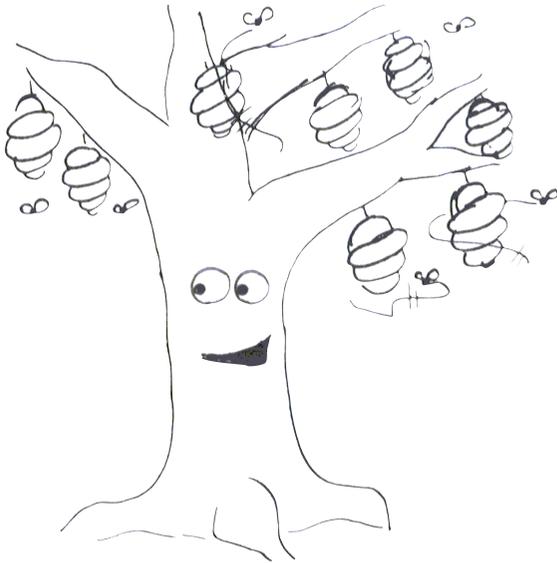
The traditional data structure for disks is the B-tree



The traditional data structure for disks is the B-tree

Adding a new datum to an N -element B-tree uses $O(\log_B N)$ block transfers in the worst case.

(Even paying one block transfer is too expensive.)



Write-optimized data structures perform better

Data structures: [O'Neil, Cheng, Gawlick, O'Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach, Fineman, Fogel, Kuszmaul, Nelson'07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10], [Spillane, Shetty, Zadok, Archak, Dixit 11].

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

	B-tree	Some write-optimized structures
Insert/delete	$O(\log_B N) = O\left(\frac{\log N}{\log B}\right)$	$O\left(\frac{\log N}{B}\right)$

- If $B=1024$, then insert speedup is $B/\log B \approx 100$.
- Hardware trends mean bigger B , bigger speedup.
- Less than 1 I/O per insert.

Optimal Search-Insert Tradeoff

[Brodal, Fagerberg 03]

insert

point query

**Optimal
tradeoff**

(function of $\varepsilon=0\dots 1$)

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

$$O(\log_{1+B^\varepsilon} N)$$

B-tree
($\varepsilon=1$)

$$O(\log_B N)$$

$$O(\log_B N)$$

$\varepsilon=1/2$

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

$$O(\log_B N)$$

$\varepsilon=0$

$$O\left(\frac{\log N}{B}\right)$$

$$O(\log N)$$

10x-100x faster inserts

Illustration of Optimal Tradeoff [Brodal, Fagerberg 03]

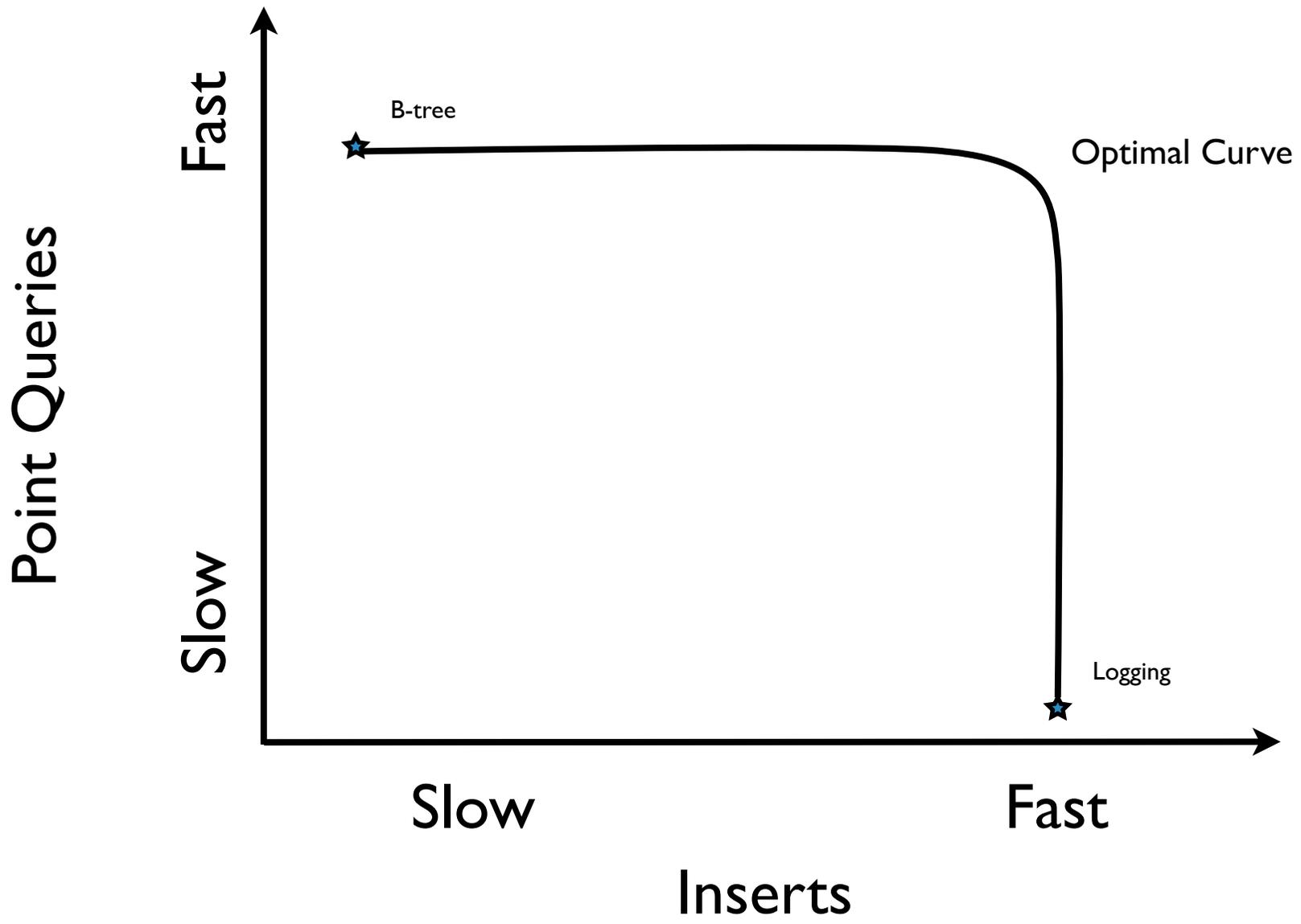
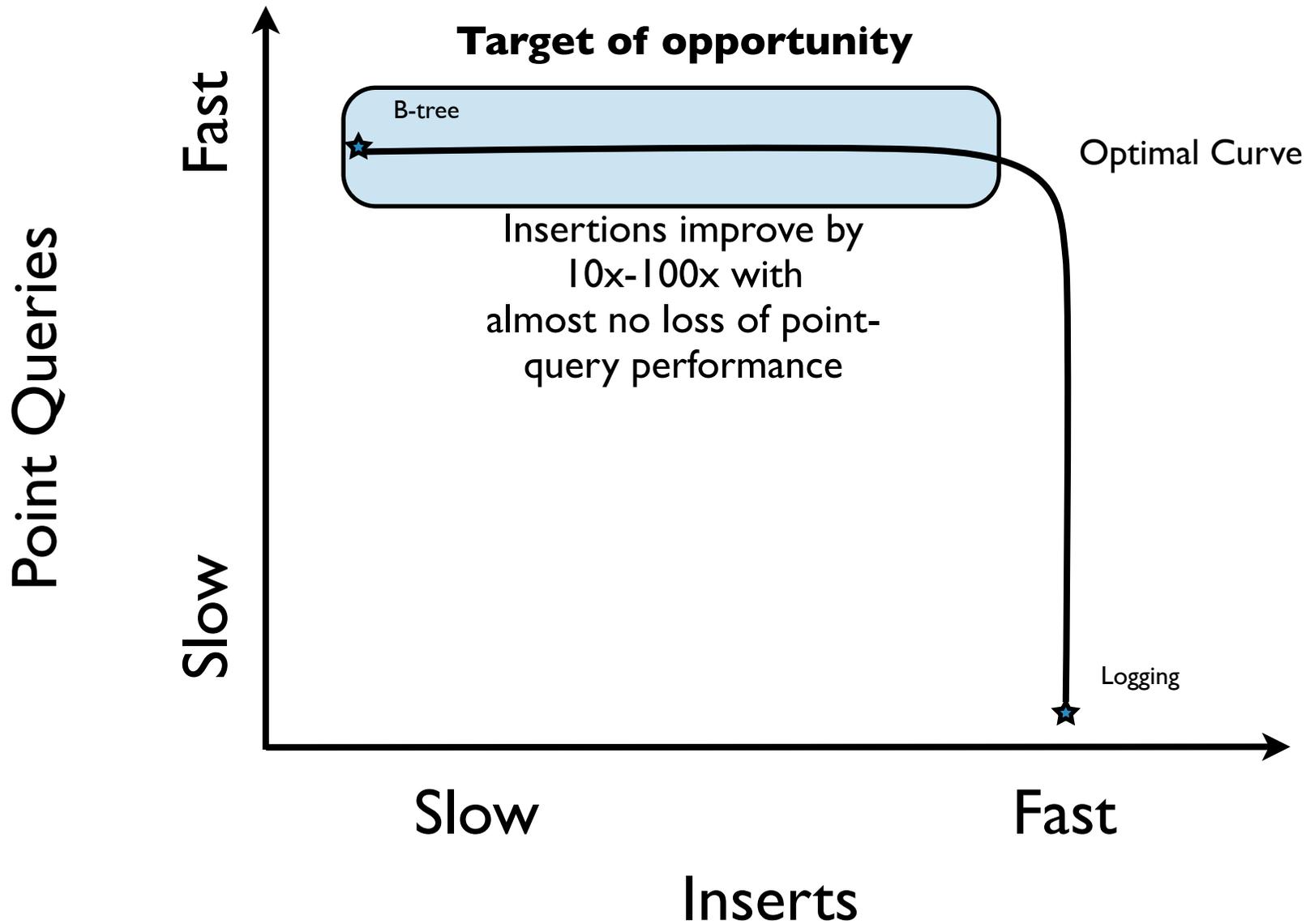
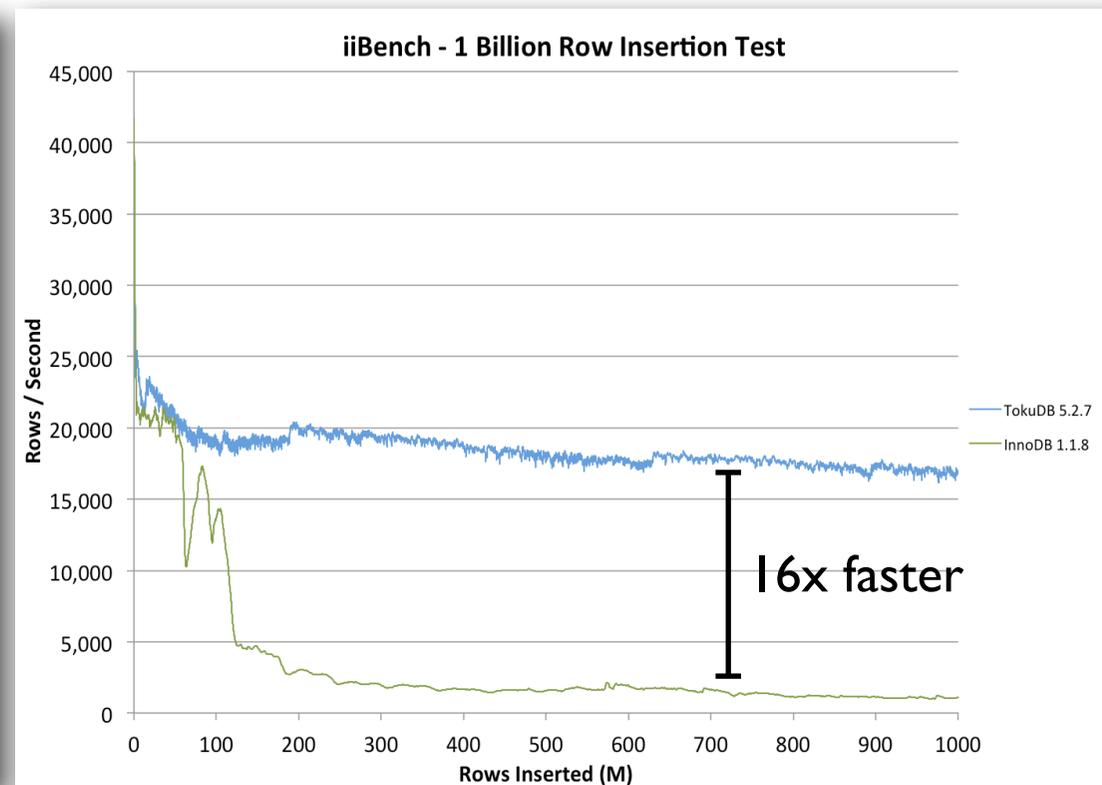
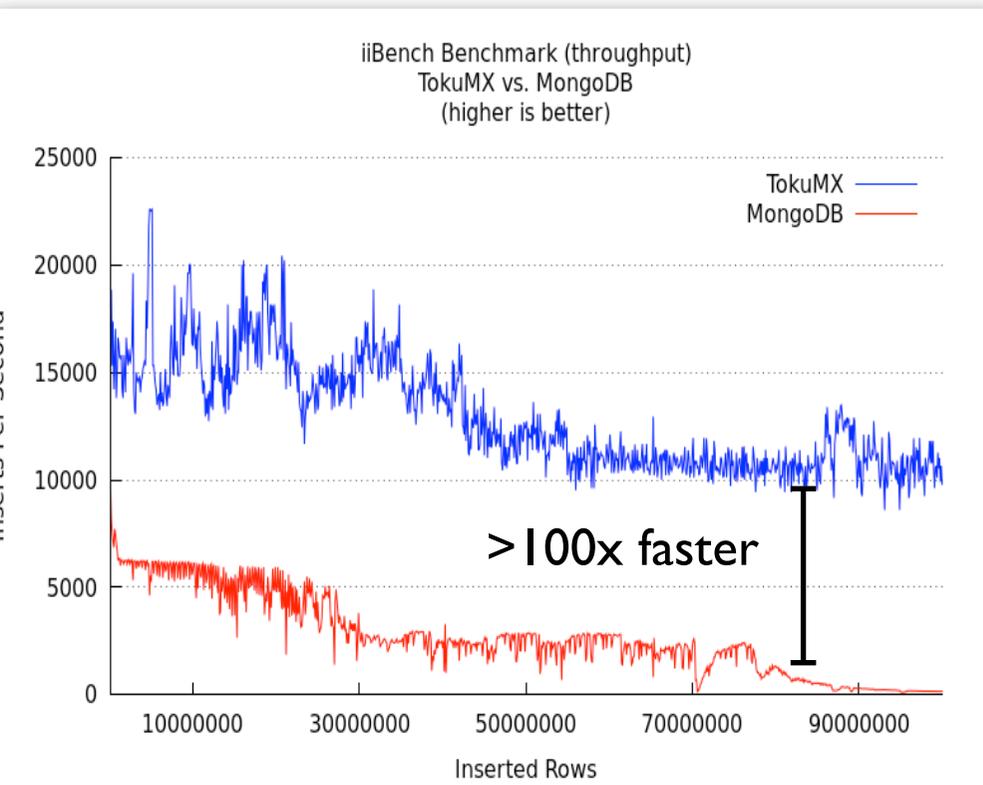


Illustration of Optimal Tradeoff [Brodal, Fagerberg 03]



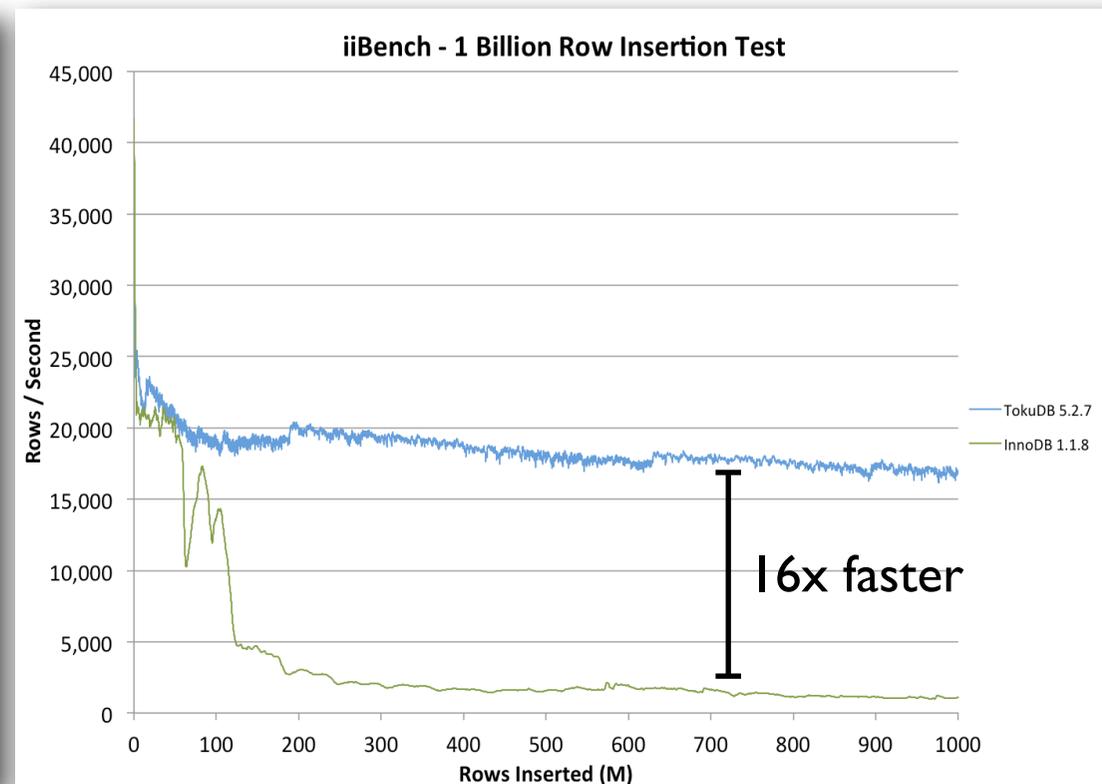
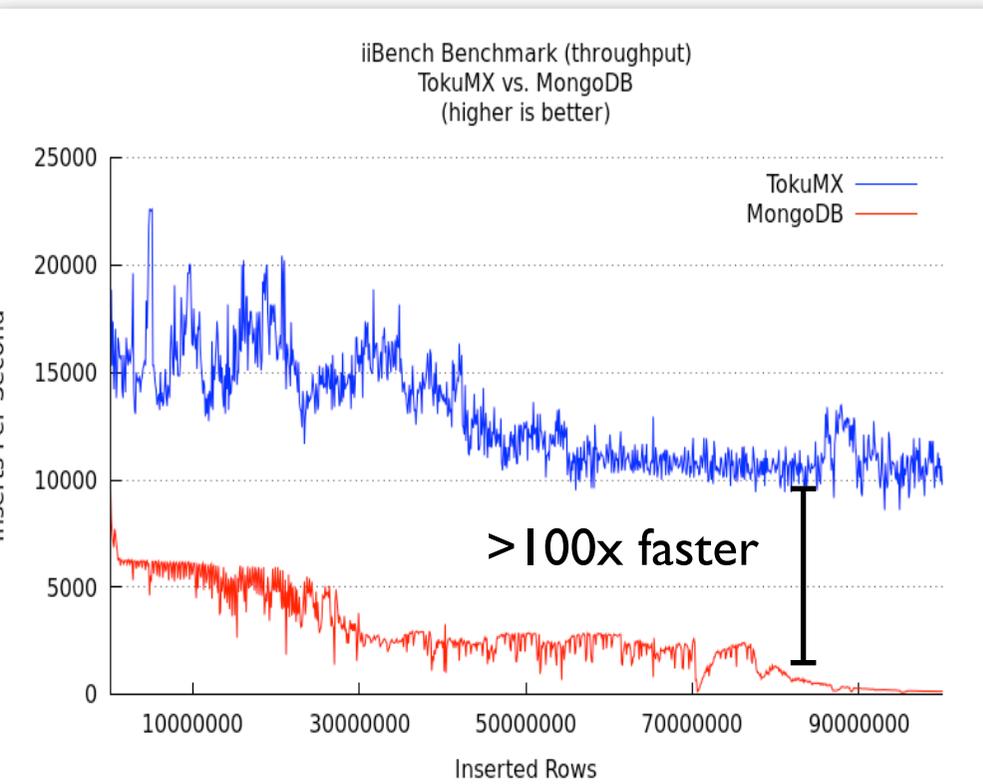
Performance of write-optimized data structures

Write performance on large data



Performance of write-optimized data structures

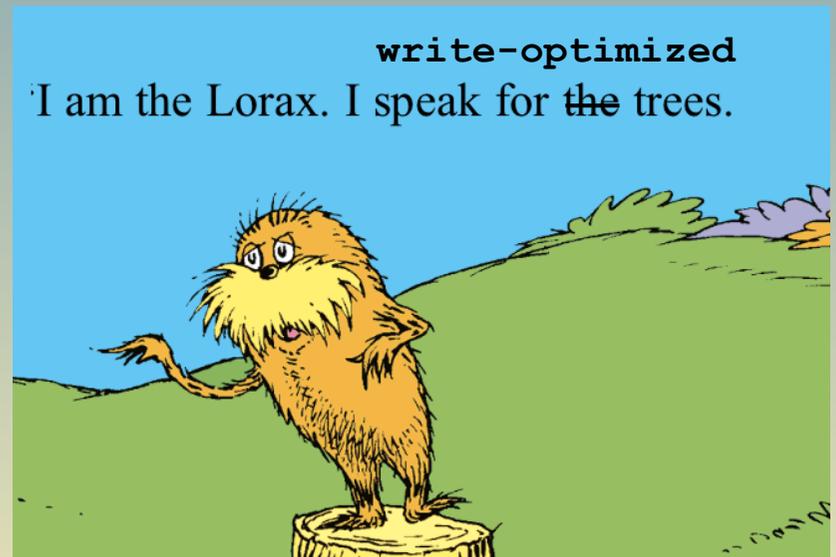
Write performance on large data



Later: why fast indexing leads to faster queries.

How to Build Write-Optimized Structures

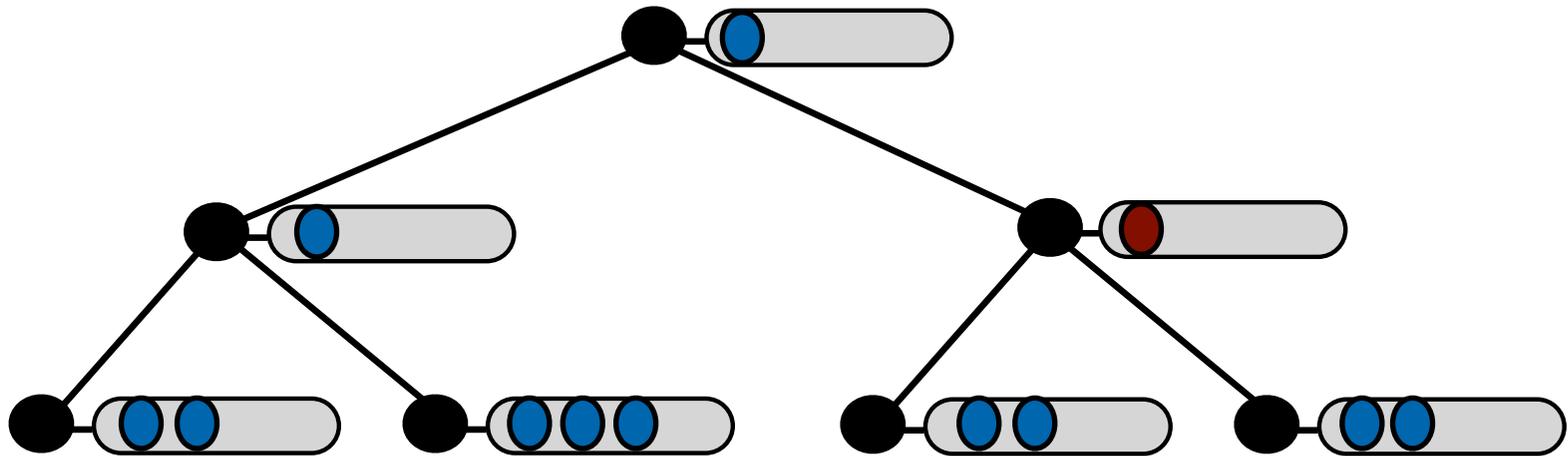
How to Build Write-Optimized Structures



A simple write-optimized structure

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

- A balanced binary tree with buffers of size B



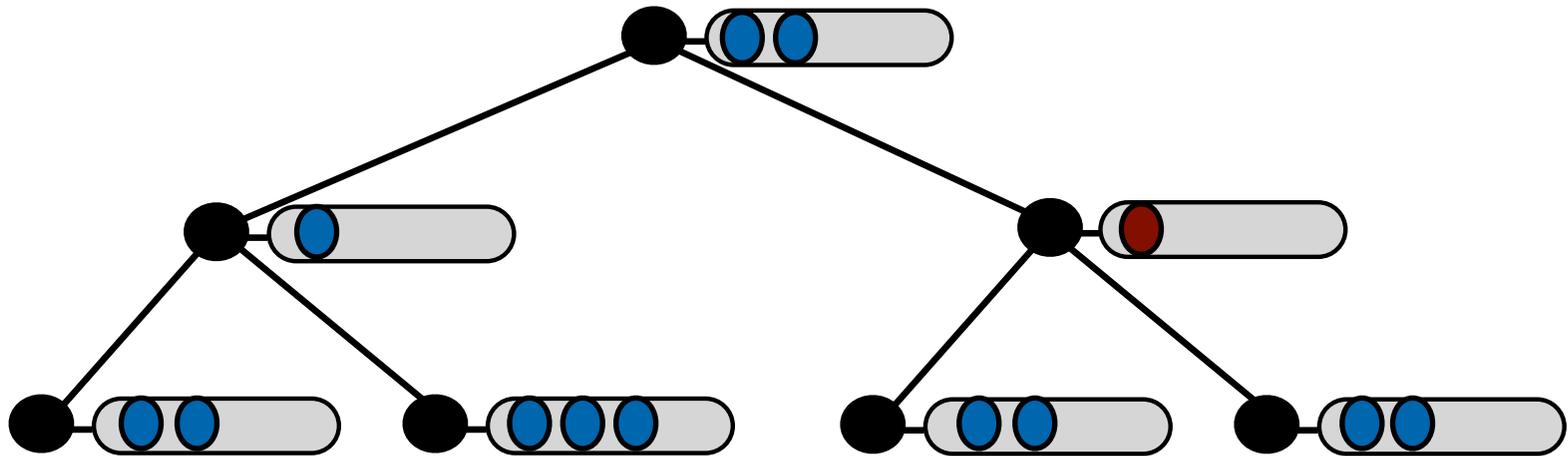
Inserts + deletes:

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

A simple write-optimized structure

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

- A balanced binary tree with buffers of size B



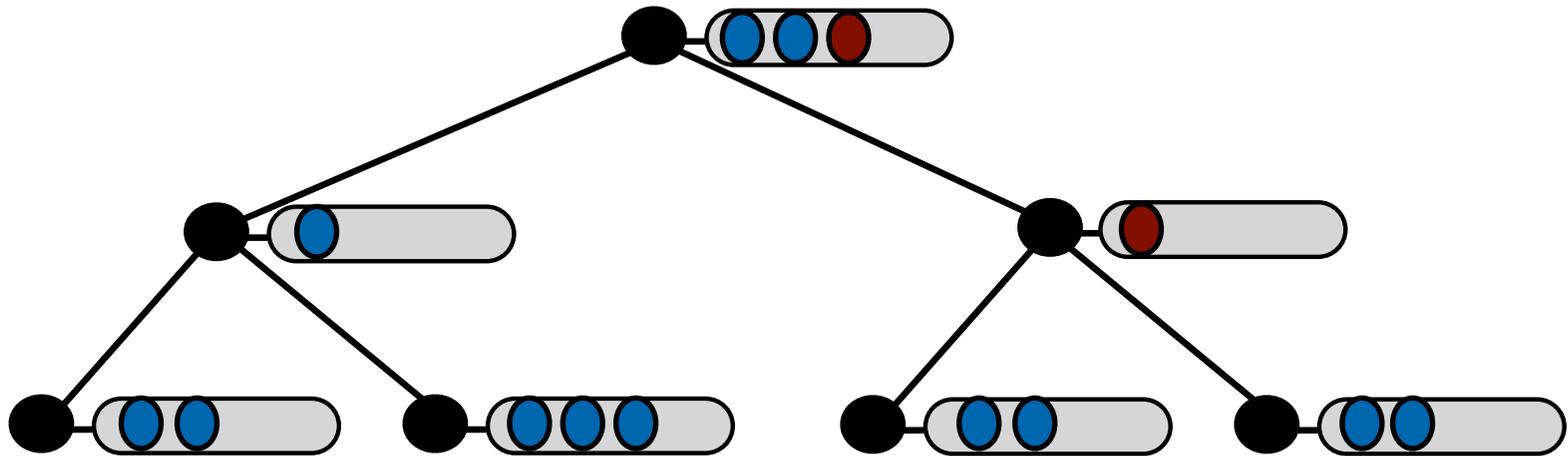
Inserts + deletes:

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

A simple write-optimized structure

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

- A balanced binary tree with buffers of size B



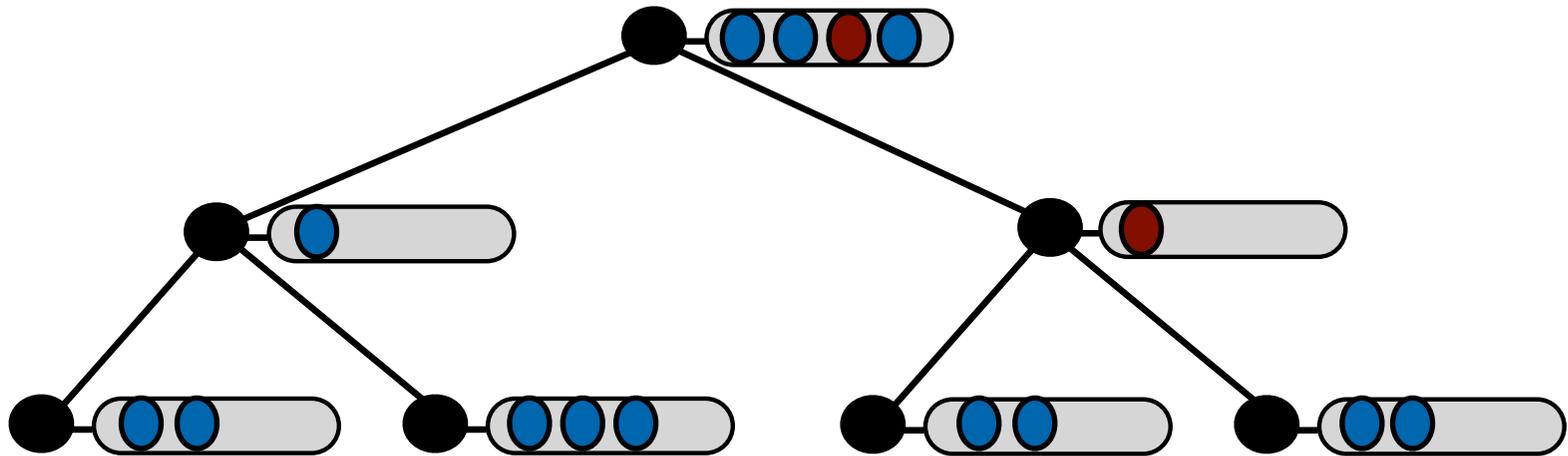
Inserts + deletes:

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

A simple write-optimized structure

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

- A balanced binary tree with buffers of size B



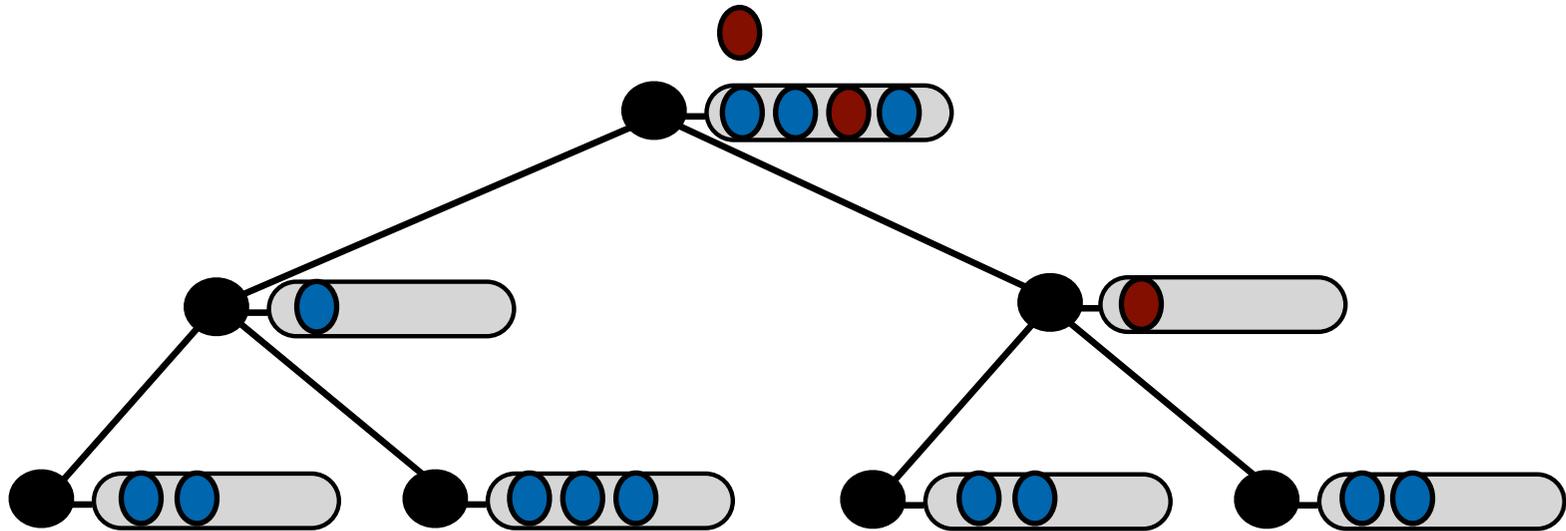
Inserts + deletes:

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

A simple write-optimized structure

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

- A balanced binary tree with buffers of size B



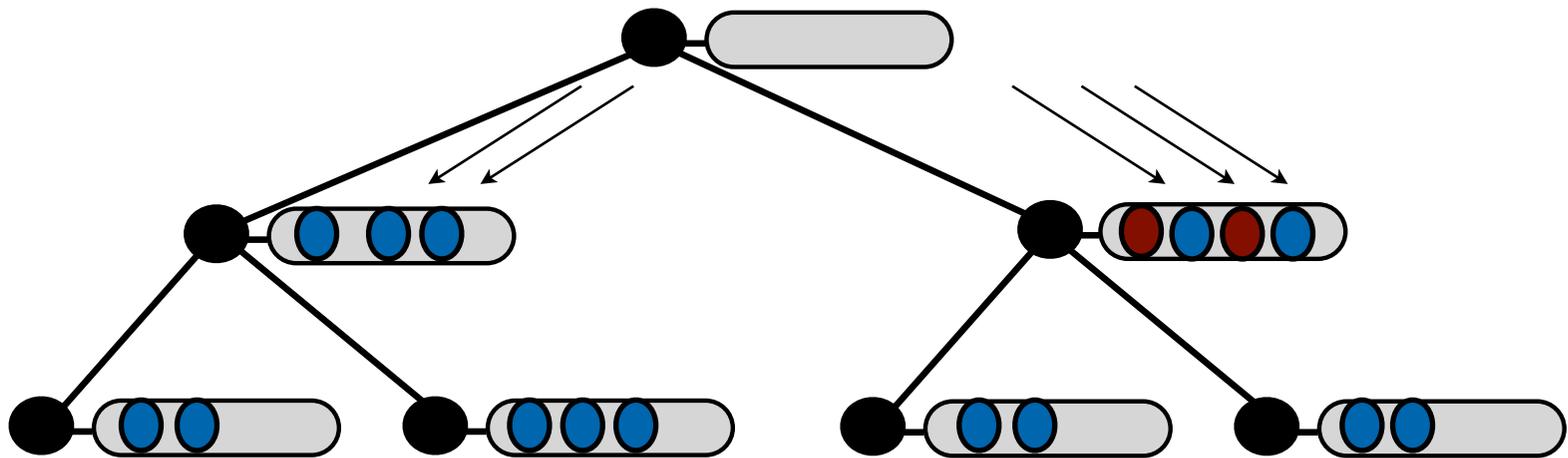
Inserts + deletes:

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

A simple write-optimized structure

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

- A balanced binary tree with buffers of size B



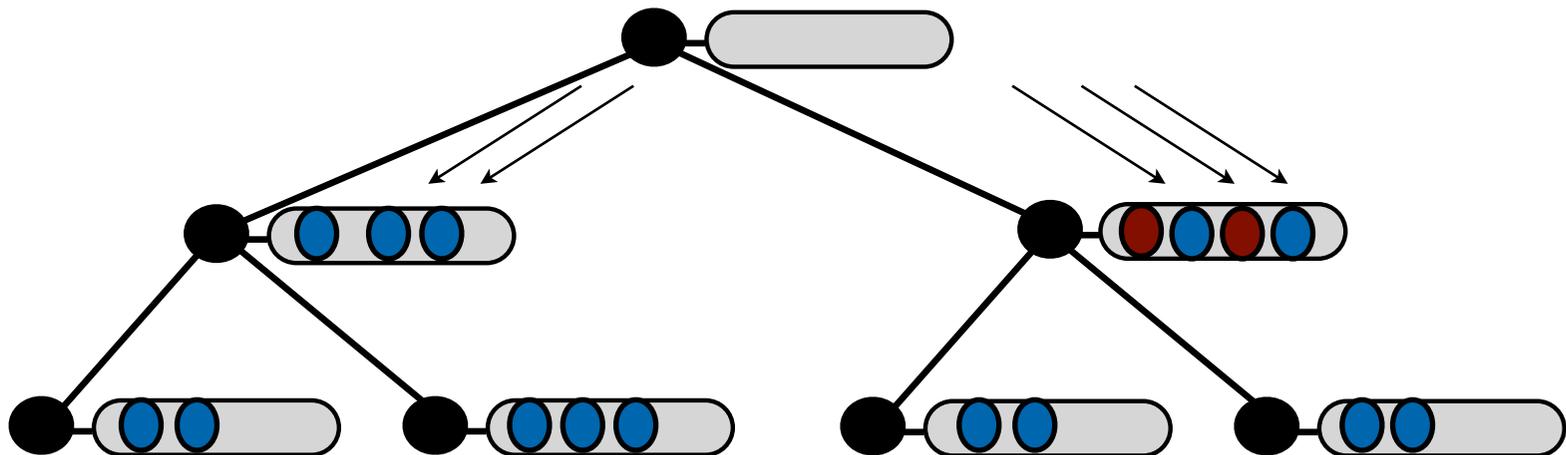
Inserts + deletes:

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

Analysis of writes

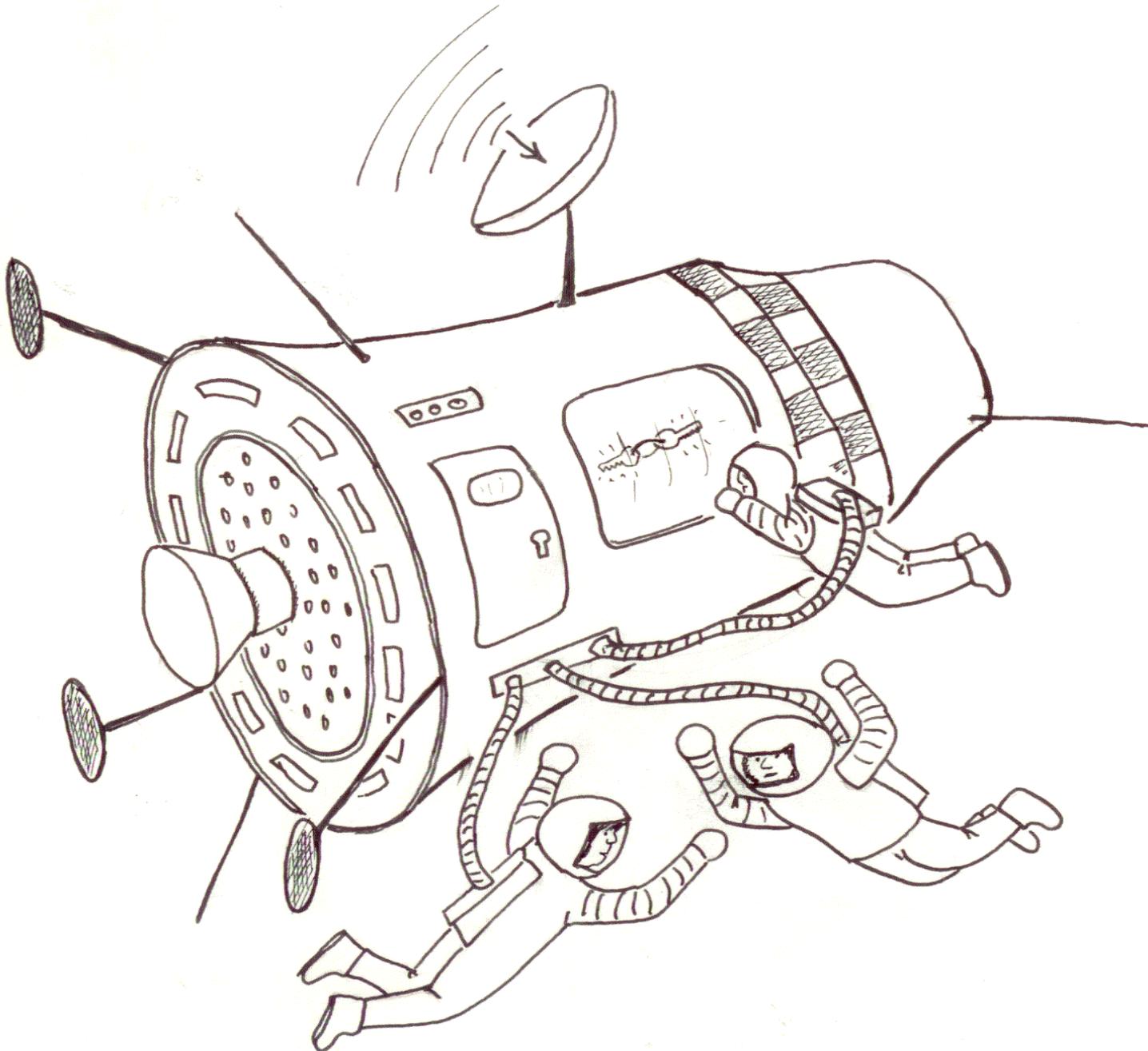
An insert/delete costs amortized $O((\log N)/B)$ per insert or delete

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



Difficulty of Key Accesses

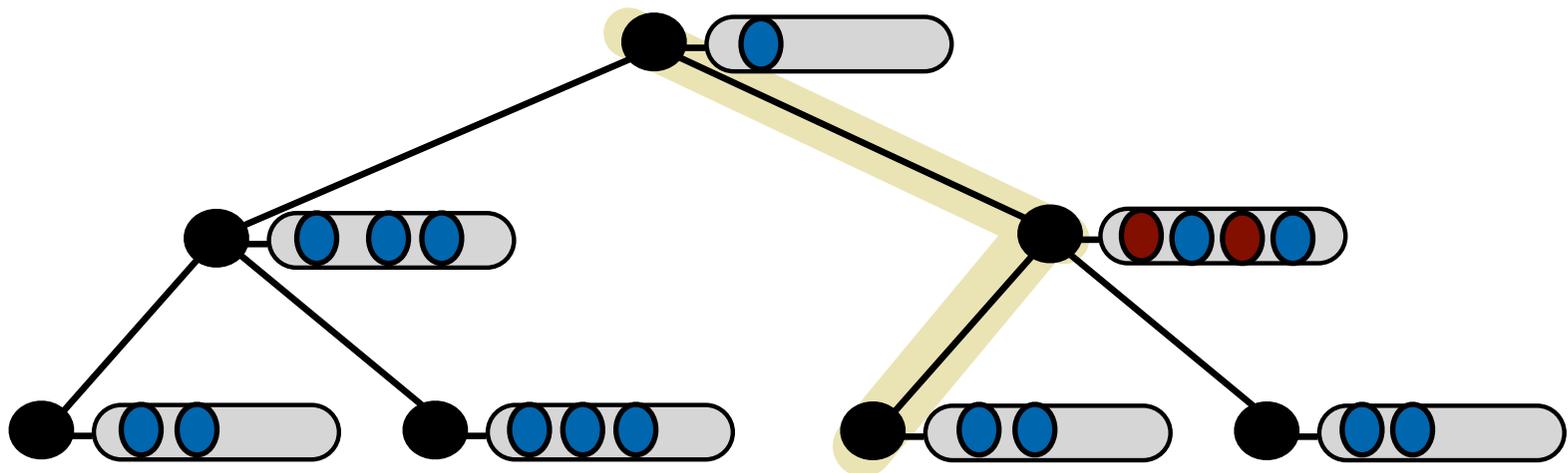
Difficulty of Key Accesses



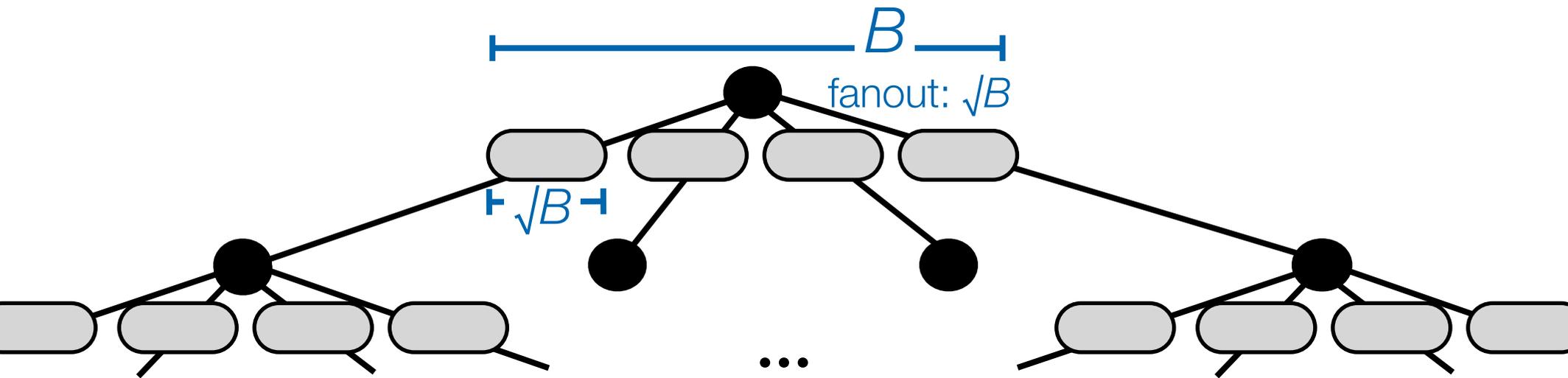
Analysis of point queries

To search:

- examine each buffer along a single root-to-leaf path.
- This costs $O(\log N)$.



Obtaining optimal point queries + very fast inserts



Point queries cost $O(\log_{\sqrt{B}} N) = O(\log_B N)$

- This is the tree height.

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

- Each flush cost $O(1)$ I/Os and flushes \sqrt{B} elements.

Powerful and Explainable

Write-optimized data structures are very powerful.

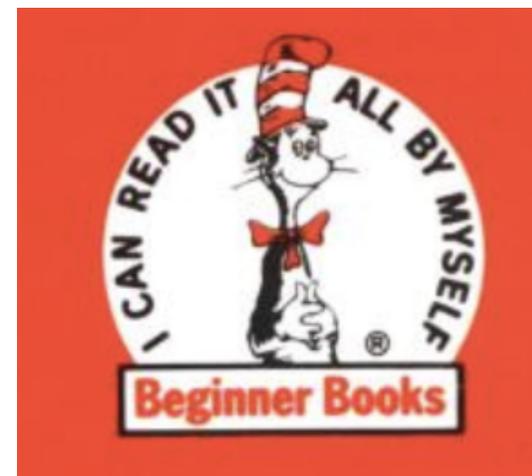
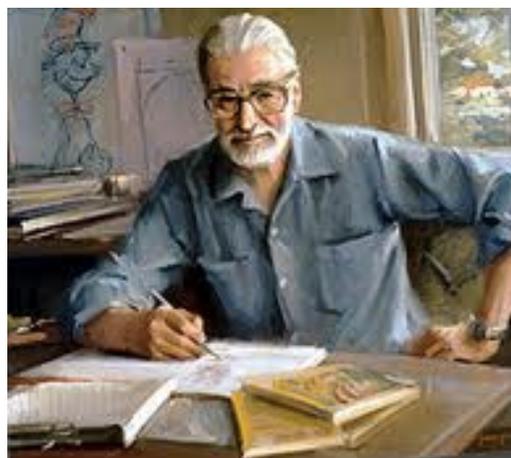
They are also not hard to teach in a standard algorithms course.

What the world looks like

Insert/point query asymmetry

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

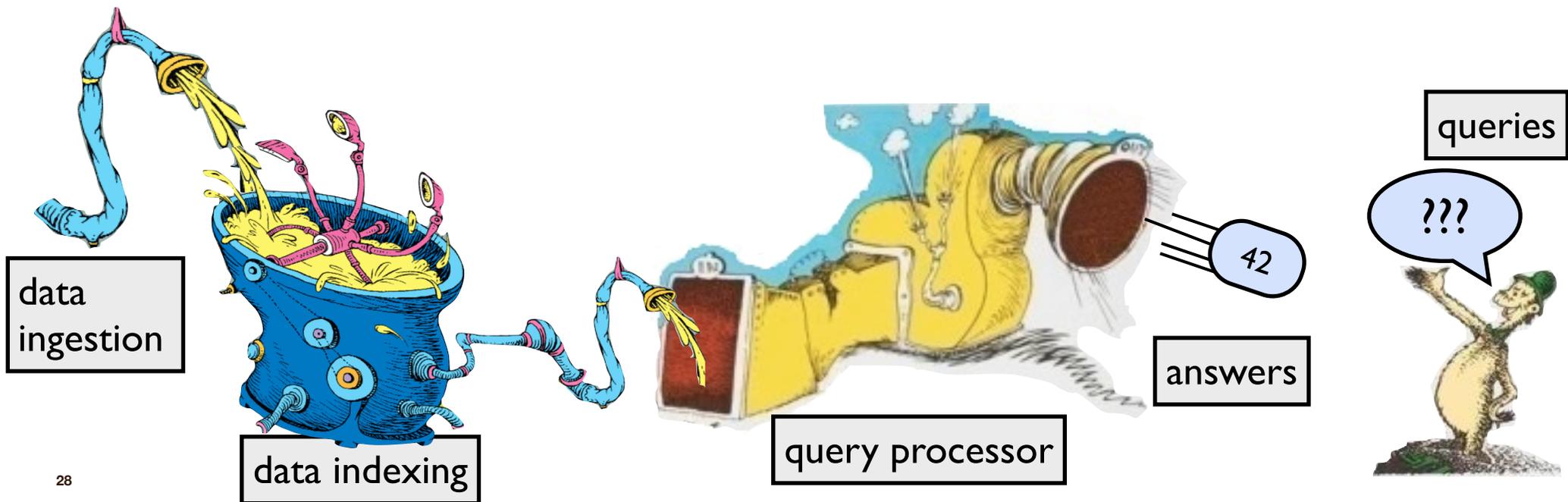
We are used to reads and writes having about the same cost, but writing is easier than reading.



The right read optimization is write optimization

The right index makes queries run fast.

- Write-optimized structures maintain indexes efficiently.

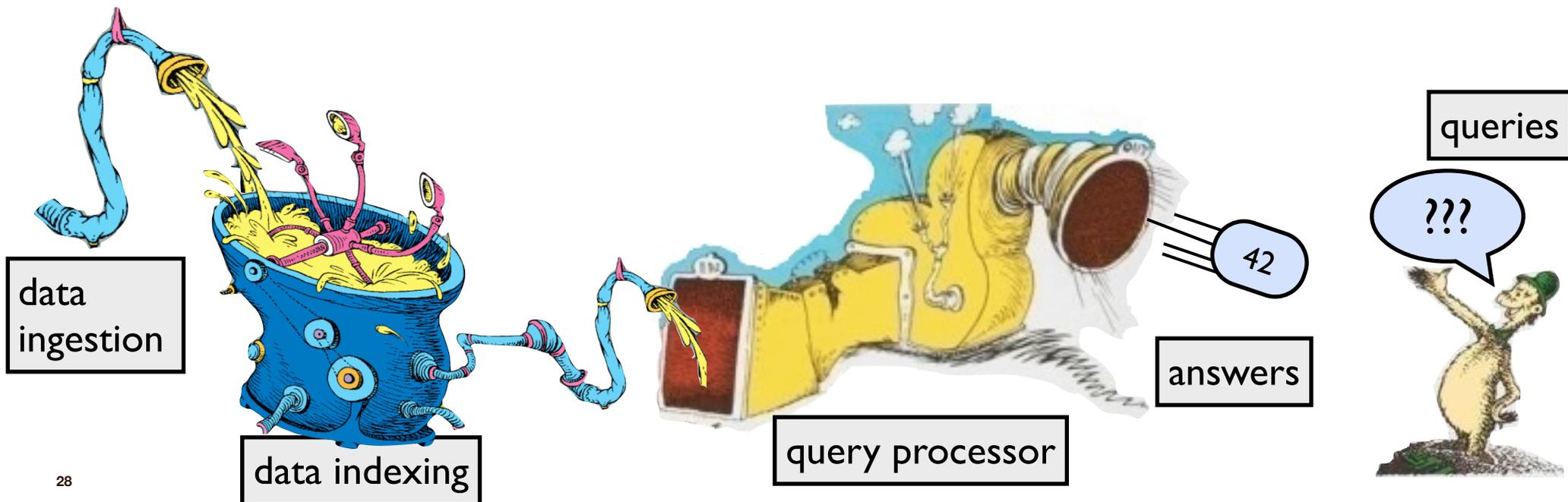


The right read optimization is write optimization

The right index makes queries run fast.

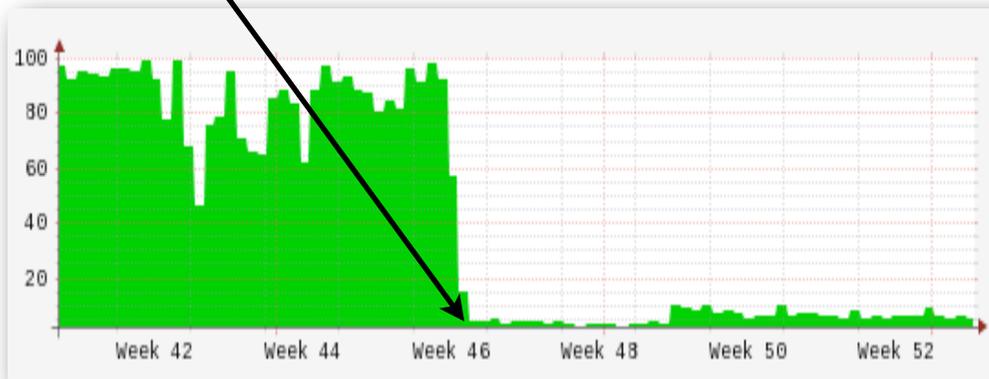
- Write-optimized structures maintain indexes efficiently.

Fast writing is a currency we use to accelerate queries. Better indexing means faster queries.



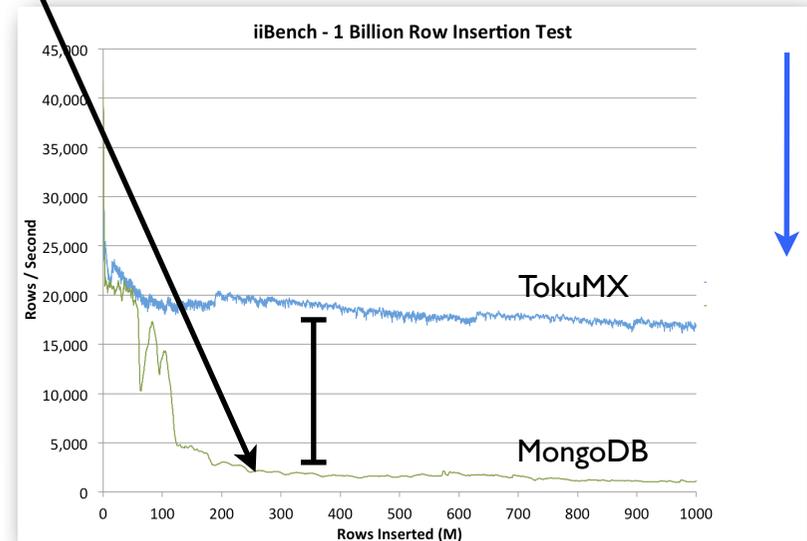
The right read optimization is write optimization

Adding more indexes leads to faster queries.



query I/O load on a production server

Index maintenance has been notoriously slow.

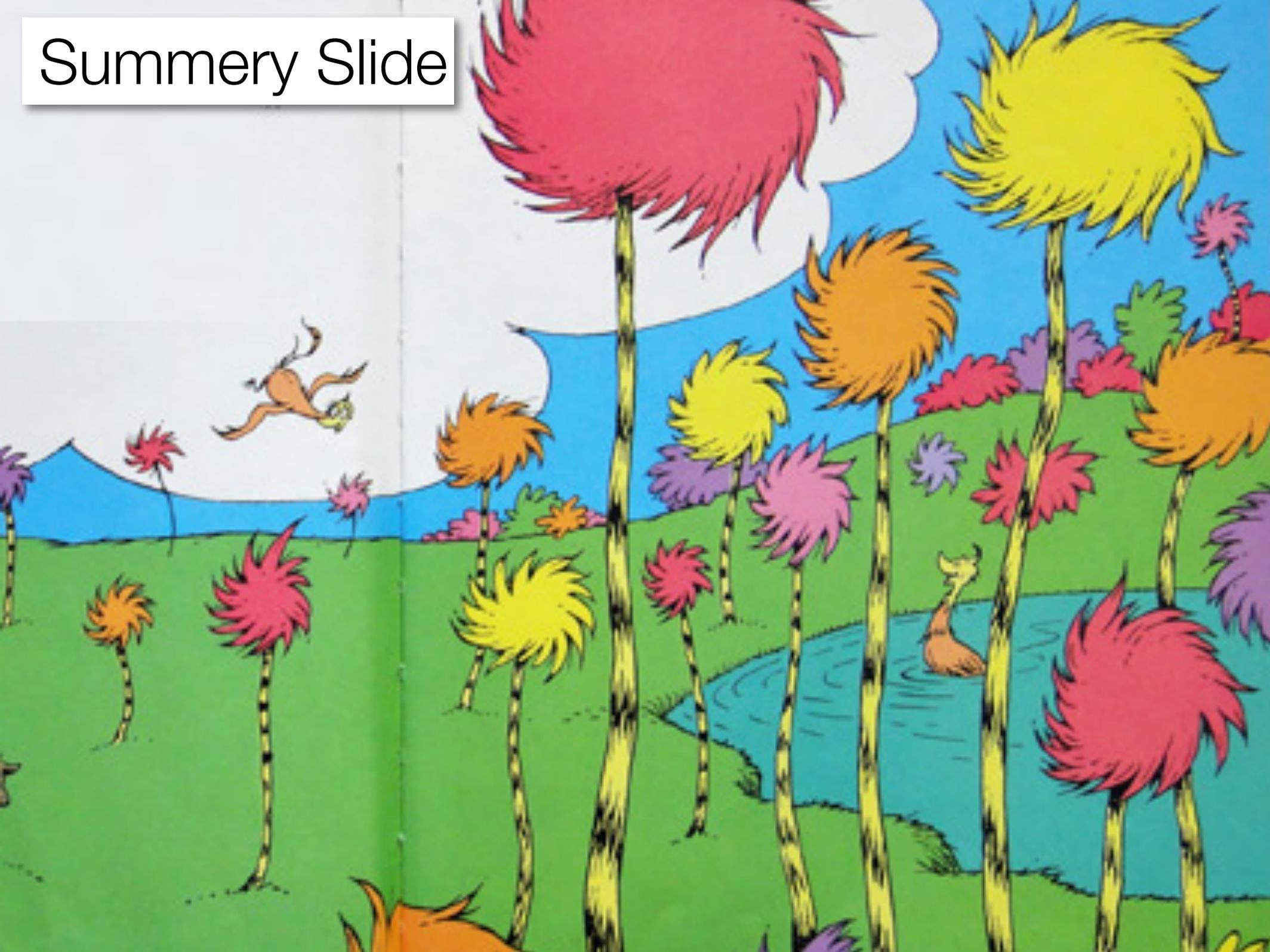


indexing rate

If we can insert faster, we can afford to maintain more indexes (i.e., organize the data in more ways.)

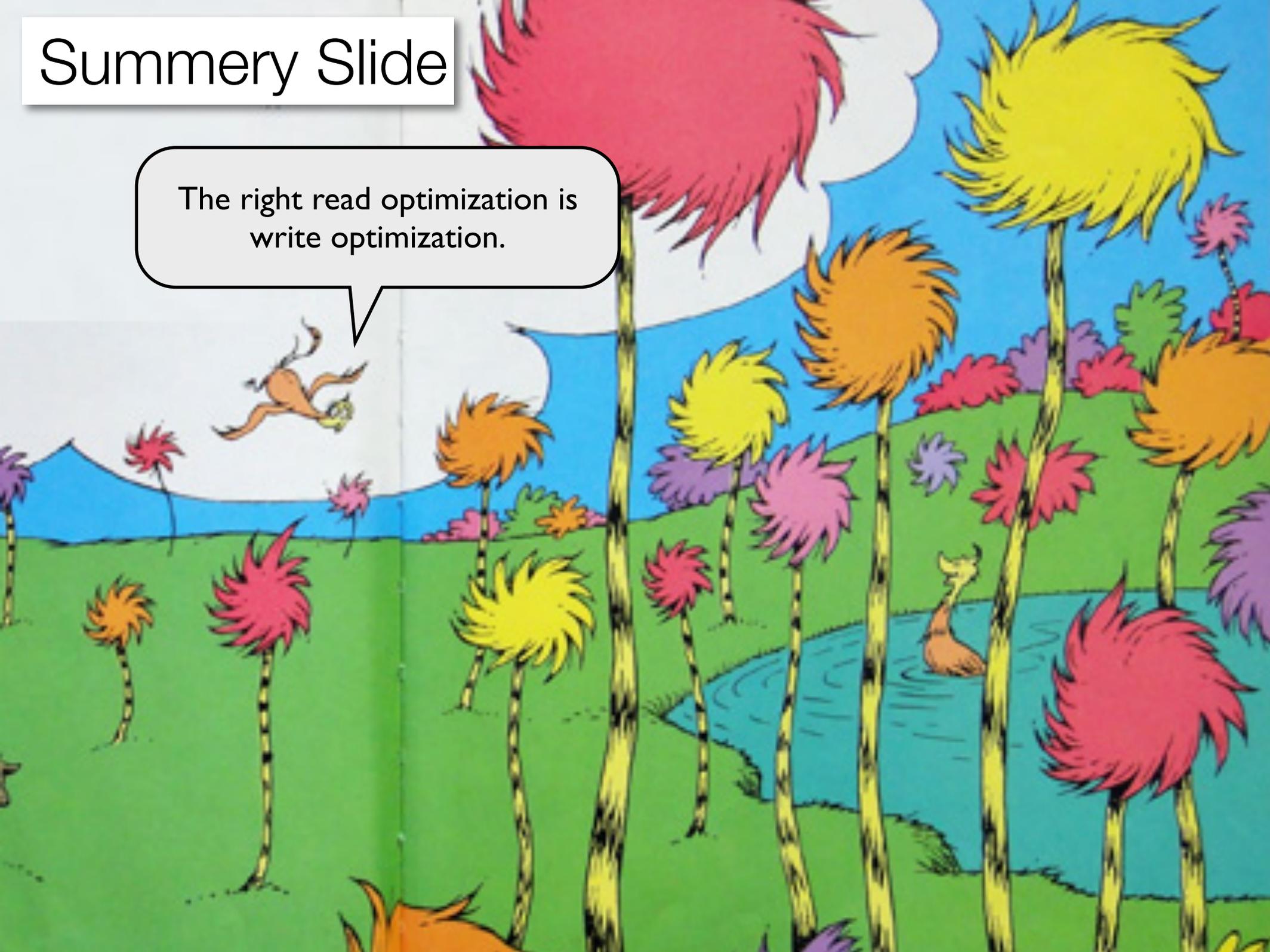


Summery Slide



Summery Slide

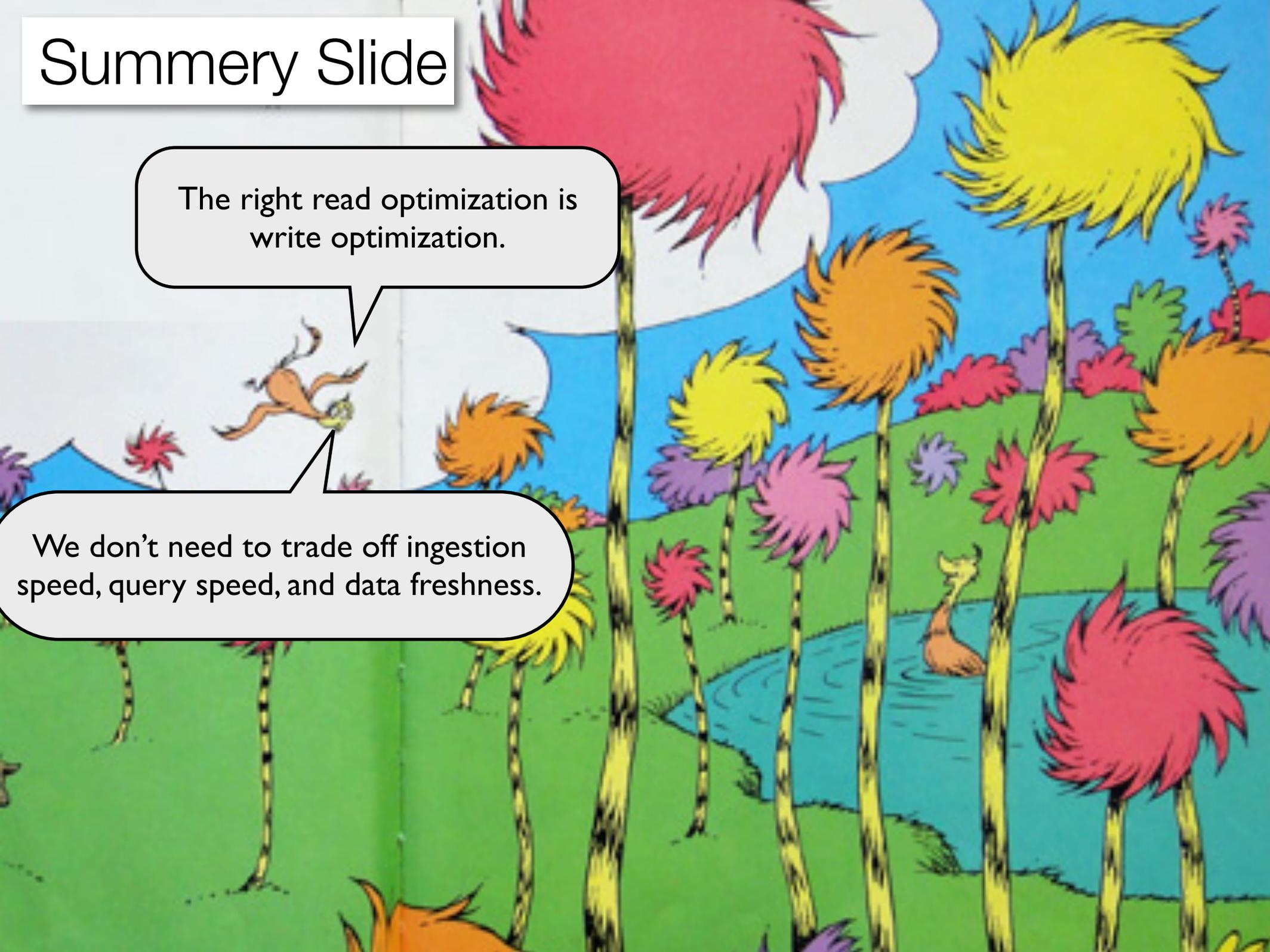
The right read optimization is
write optimization.



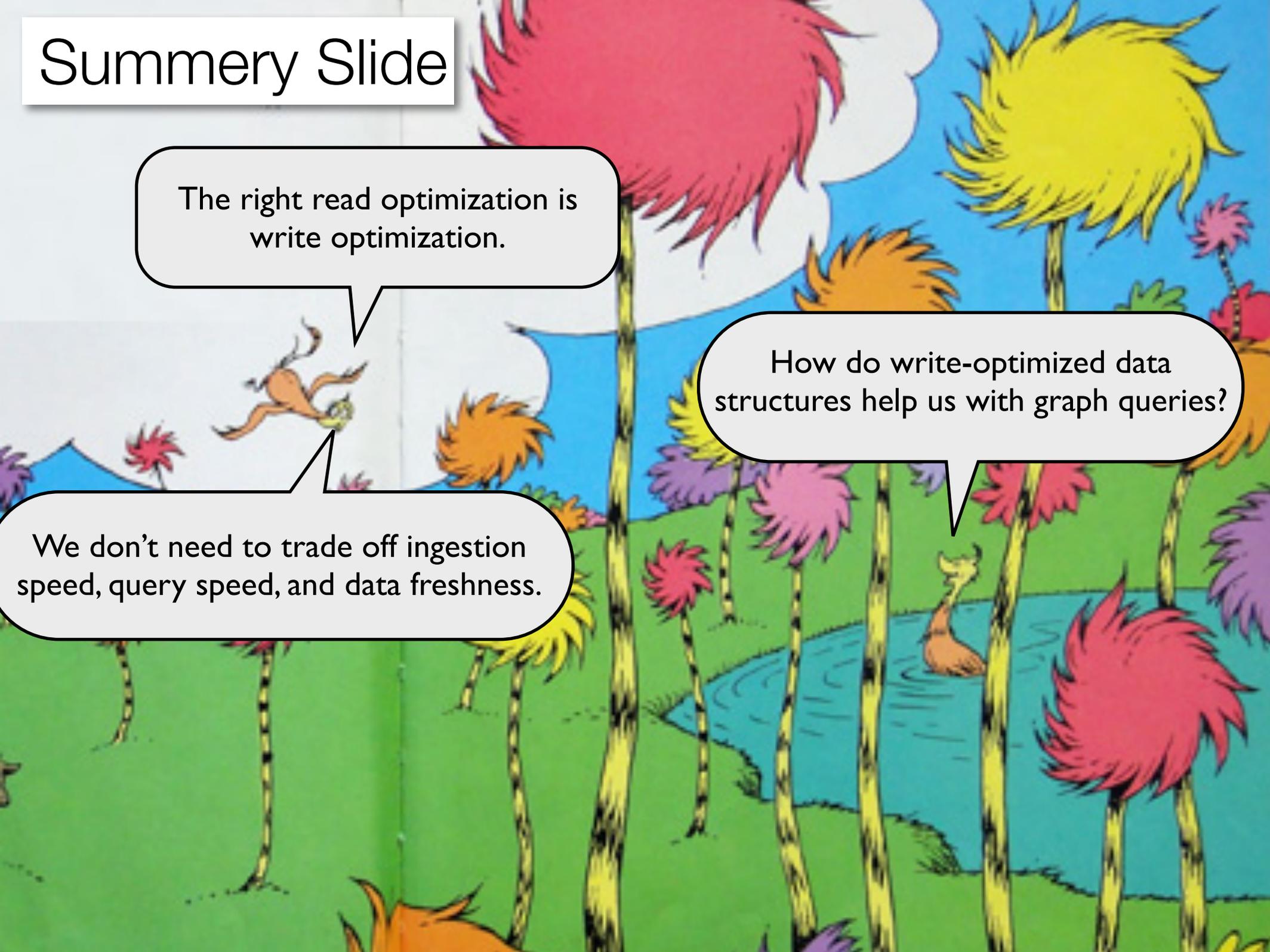
Summery Slide

The right read optimization is write optimization.

We don't need to trade off ingestion speed, query speed, and data freshness.



Summery Slide

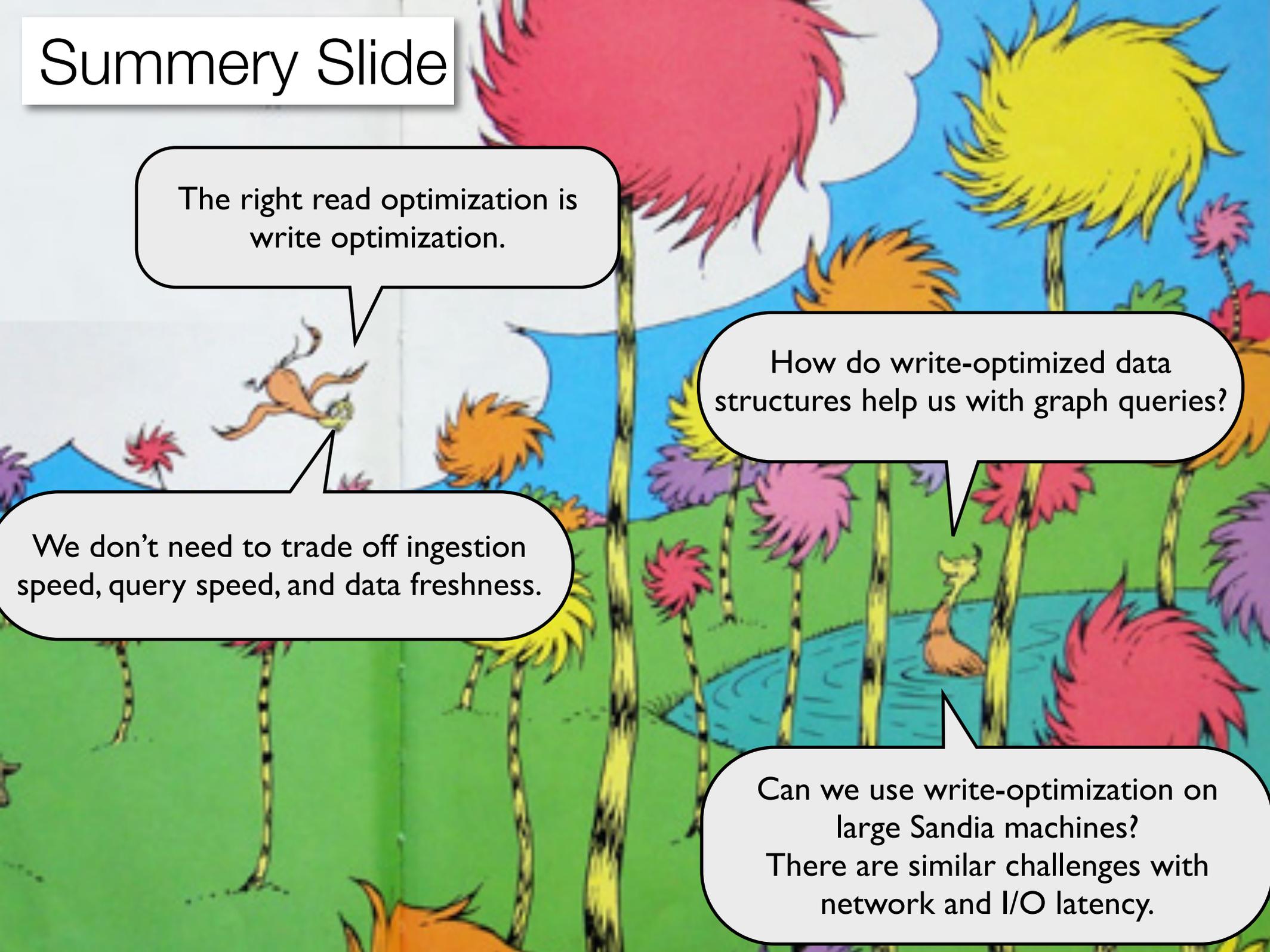


The right read optimization is write optimization.

How do write-optimized data structures help us with graph queries?

We don't need to trade off ingestion speed, query speed, and data freshness.

Summery Slide



The right read optimization is write optimization.

How do write-optimized data structures help us with graph queries?

We don't need to trade off ingestion speed, query speed, and data freshness.

Can we use write-optimization on large Sandia machines?
There are similar challenges with network and I/O latency.