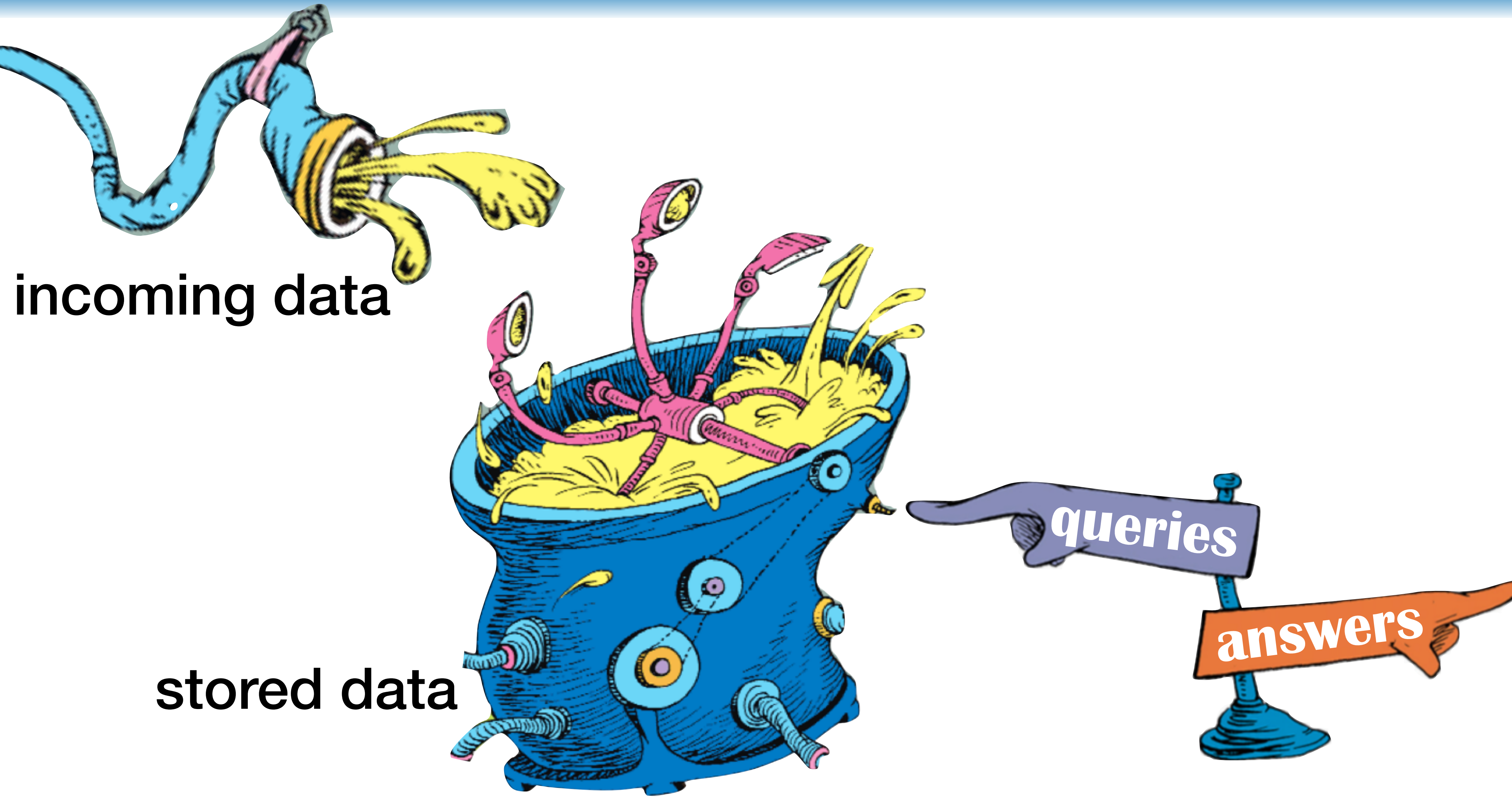
The Algorithmics of Write Optimization

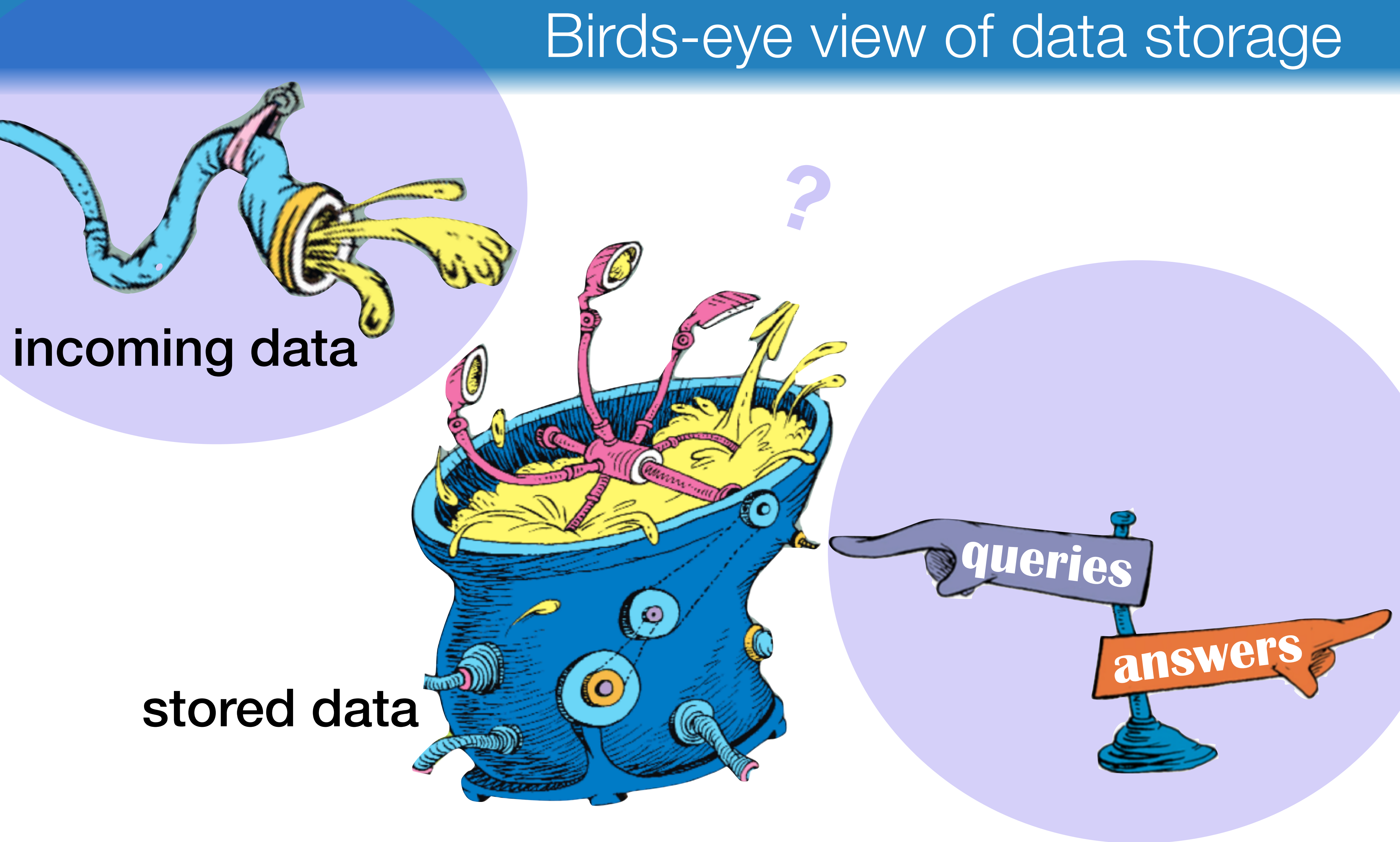
Michael A. Bender

Stony Brook University

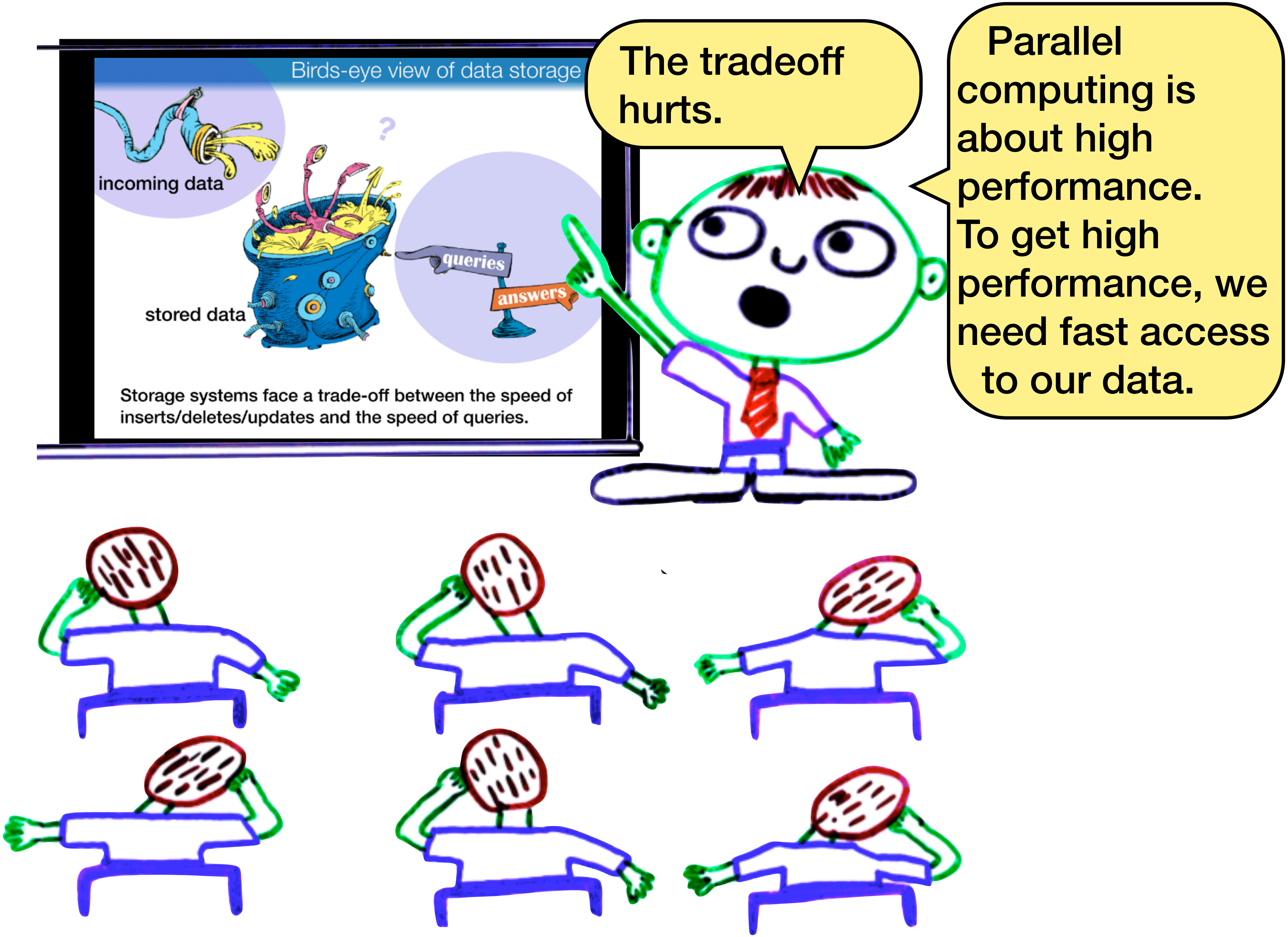
Birds-eye view of data storage

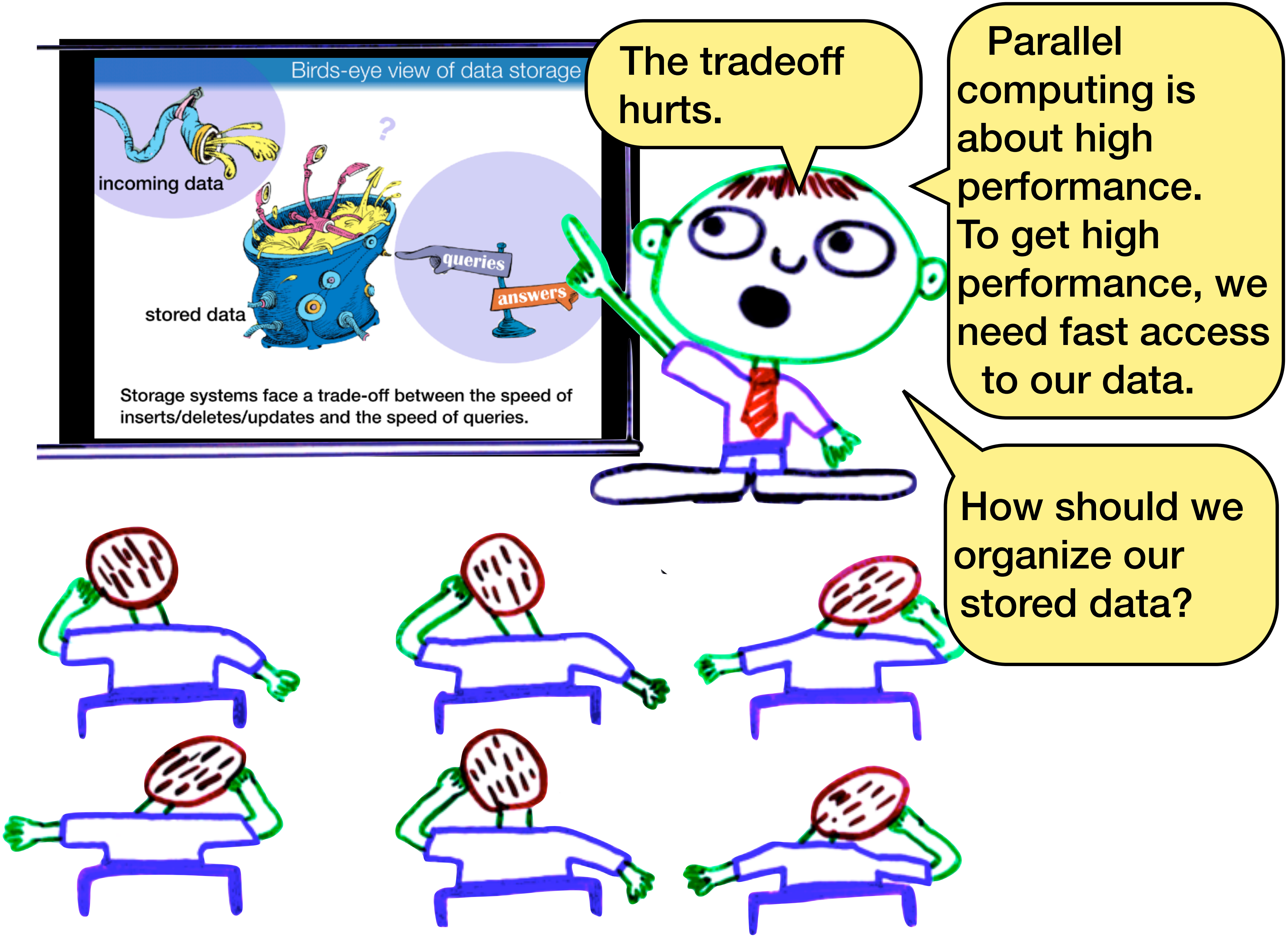


Birds-eye view of data storage

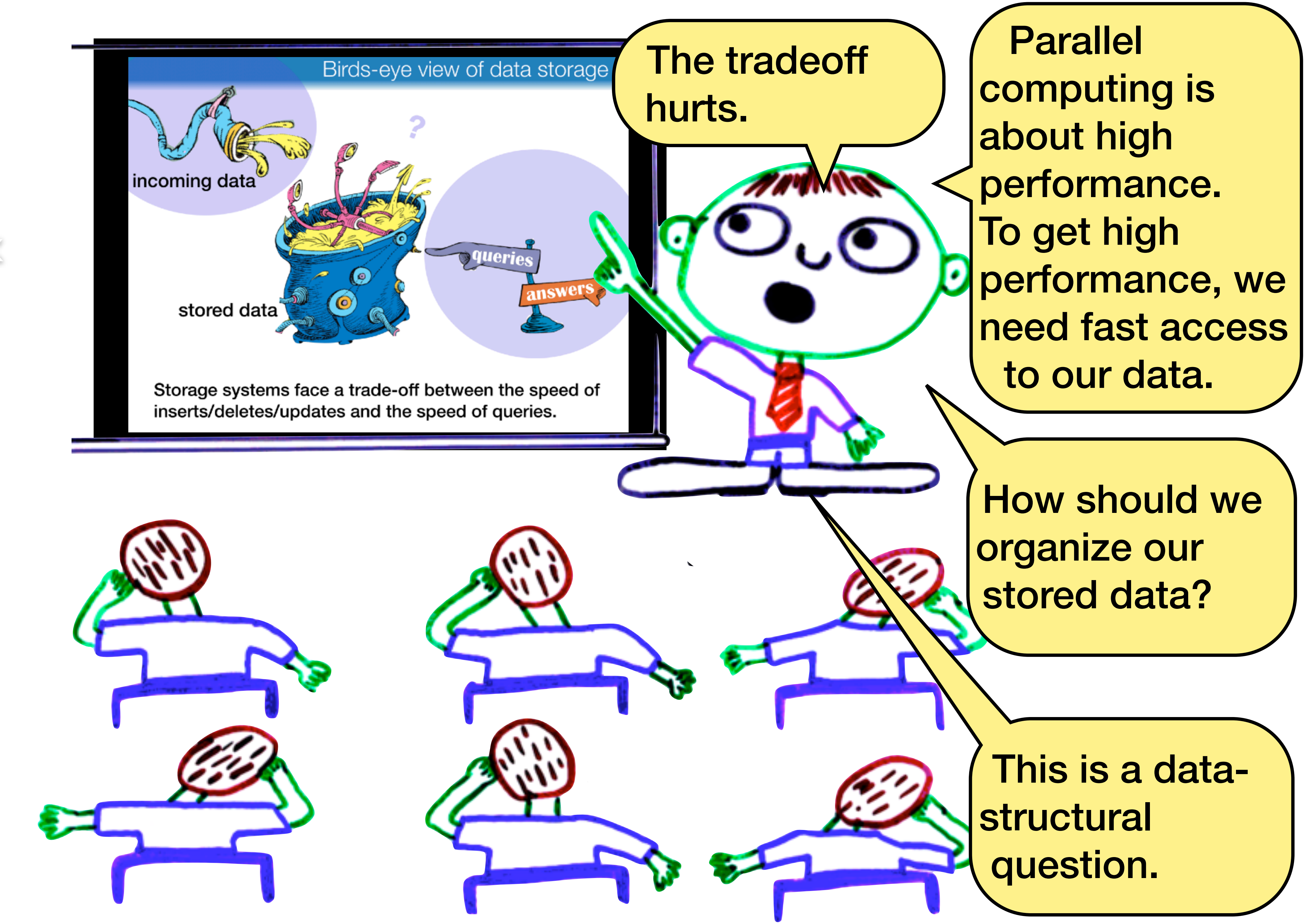


Storage systems face a trade-off between the speed of inserts/deletes/updates and the speed of queries.


Birds-eye view of data storage
incoming data
?
stored data
queries
answers
Storage systems face a trade-off between the speed of inserts/deletes/updates and the speed of queries.
The tradeoff hurts.
Parallel computing is about high performance. To get high performance, we need fast access to our data.



Birds-eye view of data storage
incoming data
stored data
queries
answers
Storage systems face a trade-off between the speed of inserts/deletes/updates and the speed of queries.
The tradeoff hurts.
Parallel computing is about high performance. To get high performance, we need fast access to our data.
How should we organize our stored data?


Birds-eye view of data storage
incoming data
stored data
queries
answers
Storage systems face a trade-off between the speed of inserts/deletes/updates and the speed of queries.
The tradeoff hurts.
Parallel computing is about high performance. To get high performance, we need fast access to our data.
How should we organize our stored data?
This is a data-structural question.

How should we organize our stored data?

How should we organize our stored data?

Like a librarian?

How should we organize our stored data?

Like a librarian?

Fast to find stuff.
Requires work to maintain.

How should we organize our stored data?

Like a librarian?

Like a teenager?

Fast to find stuff.
Requires work to maintain.

How should we organize our stored data?

Like a librarian?

Fast to find stuff.
Requires work to maintain.

Like a teenager?

Fast to add stuff.
Slow to find stuff.

How should we organize our stored data?

Like a librarian?

Fast to find stuff.
Requires work to maintain.

"Indexing"

Like a teenager?

Fast to add stuff.
Slow to find stuff.

"Logging"

How should we organize our stored data?

indexing

Sort in logical order.

(**1**,1)
(**2**,0)
(**4**,3)
(**5**,2)
(**8**,1)

Find a key: fast.
Insert a key: slower.

logging

Sort in arrival order.

(**5**,2)
(**8**,1)
(**2**,0)
(**1**,1)
(**4**,3)

Find a key: slow.
Insert a key: fast.

How should we organize our stored data?

indexing

Sort in logical order.

(**1**,1)
(**2**,0)
(**4**,3)
(**5**,2)
(**8**,1)

Find a key: fast.
Insert a key: slower.

logging

Sort in arrival order.

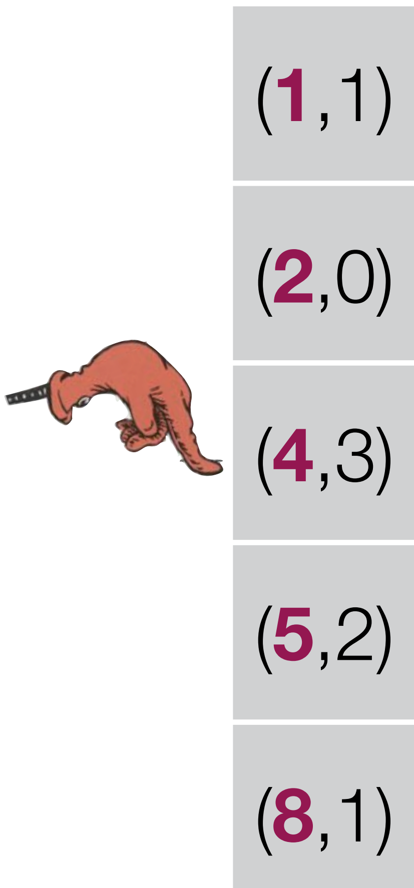
(**5**,2)
(**8**,1)
(**2**,0)
(**1**,1)
(**4**,3)

Find a key: slow.
Insert a key: fast.

How should we organize our stored data?

indexing

Sort in logical order.



Find a key: fast.
Insert a key: slower.

logging

Sort in arrival order.

Find a key: slow.
Insert a key: fast.

How should we organize our stored data?

indexing

Sort in logical order.

(**1**,1)
(**2**,0)
(**4**,3)
(**5**,2)
(**8**,1)

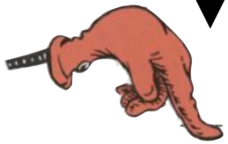
Find a key: fast.

Insert a key: slower.

logging

Sort in arrival order.

(**5**,2)
(**8**,1)
(**2**,0)
(**1**,1)
(**4**,3)



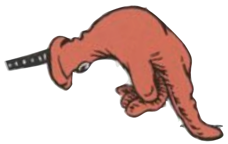
Find a key: slow.

Insert a key: fast.

How should we organize our stored data?

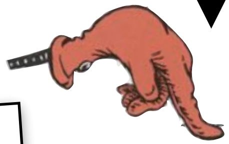
indexing

Sort in logical order.

(1,1)
(2,0)

(4,3)
(5,2)
(8,1)

logging

Sort in arrival order.

(5,2)
(8,1)
(2,0)
(1,1)

(4,3)

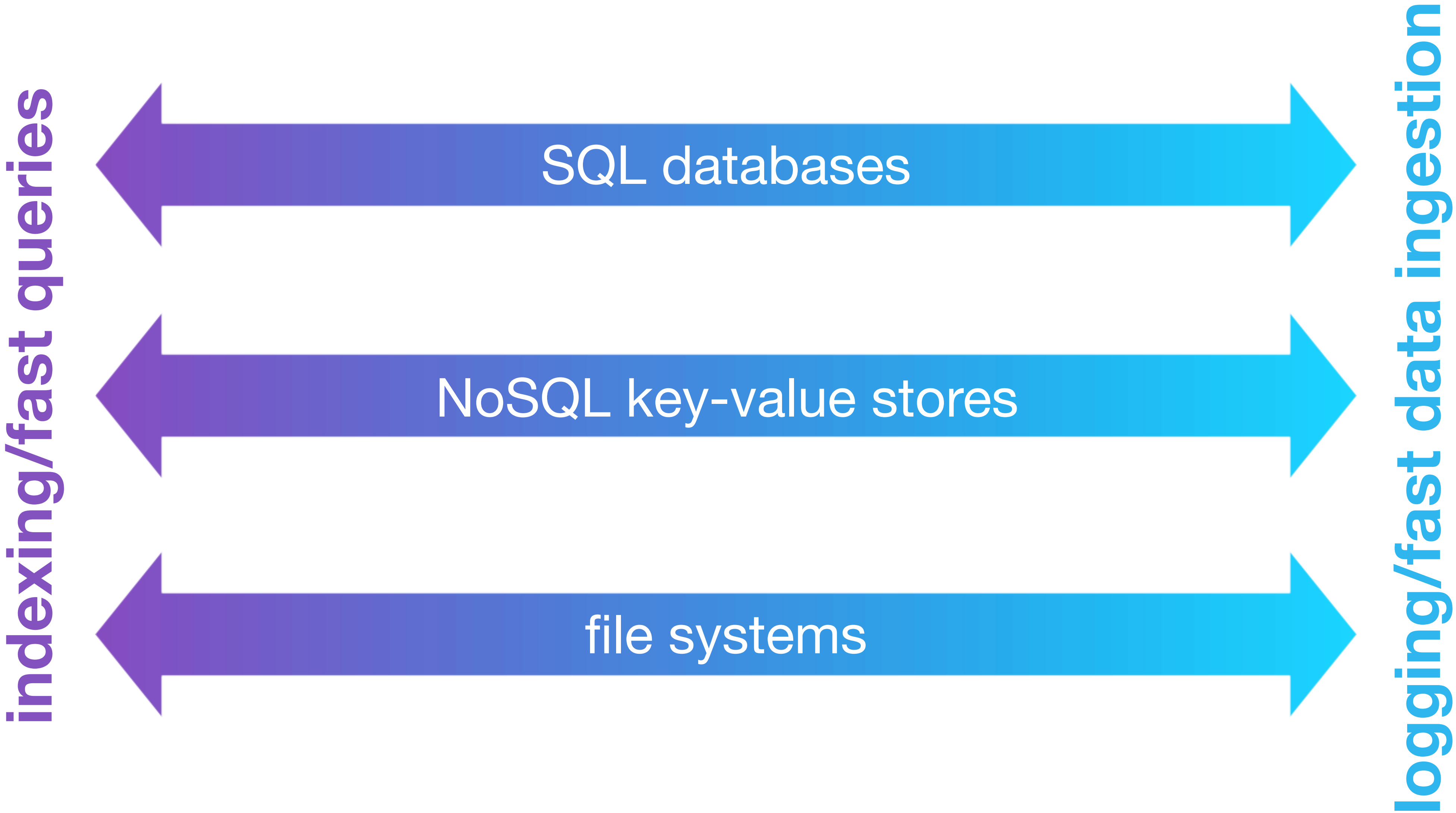
or range of keys

Find a key: fast.
Insert a key: slower.

Find a key: slow.
Insert a key: fast.

Indexing vs logging spans domains & decades

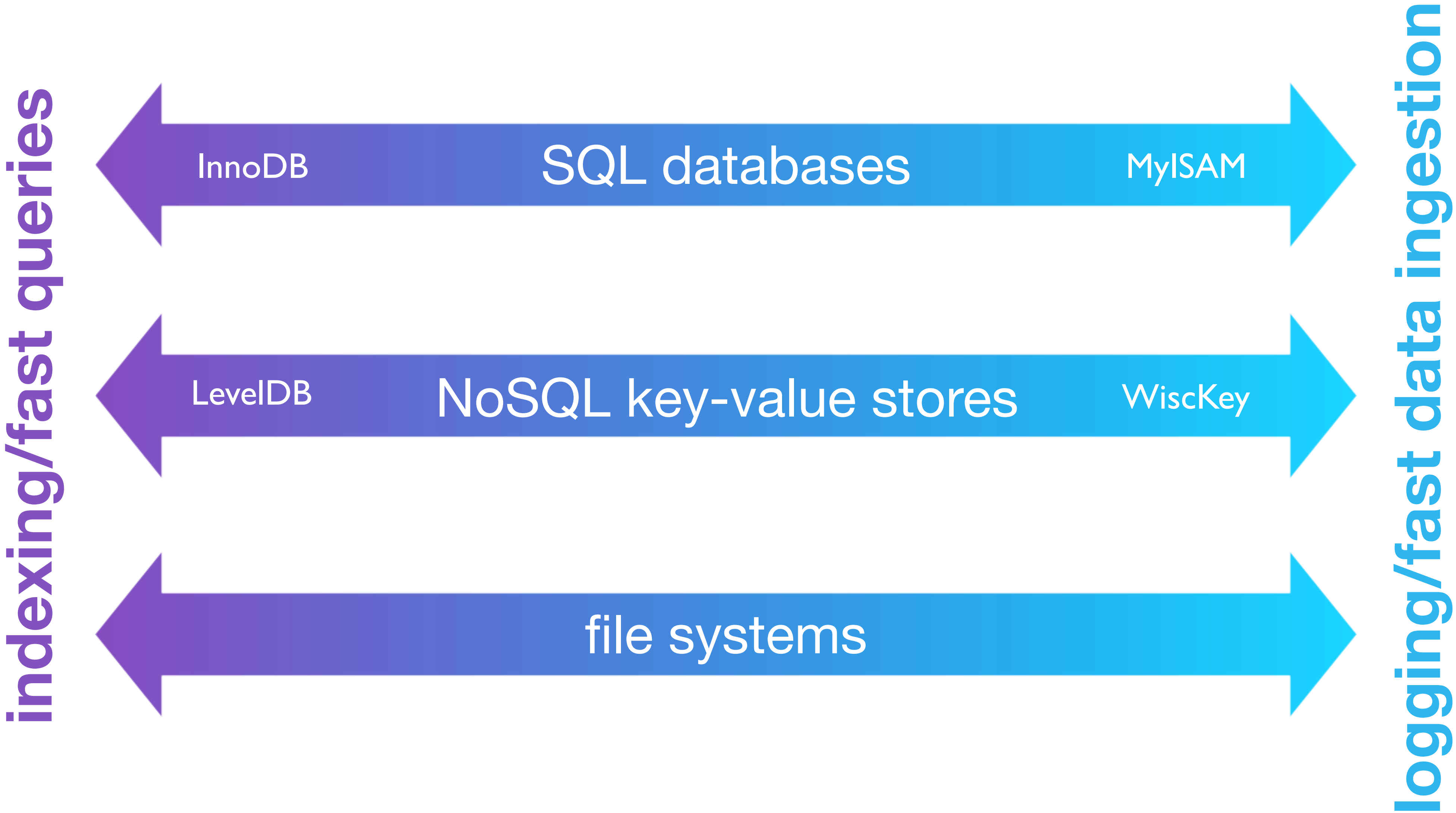
The tradeoff comes under many different names and guises:



Indexing vs logging spans domains & decades

The tradeoff comes under many different names and guises:

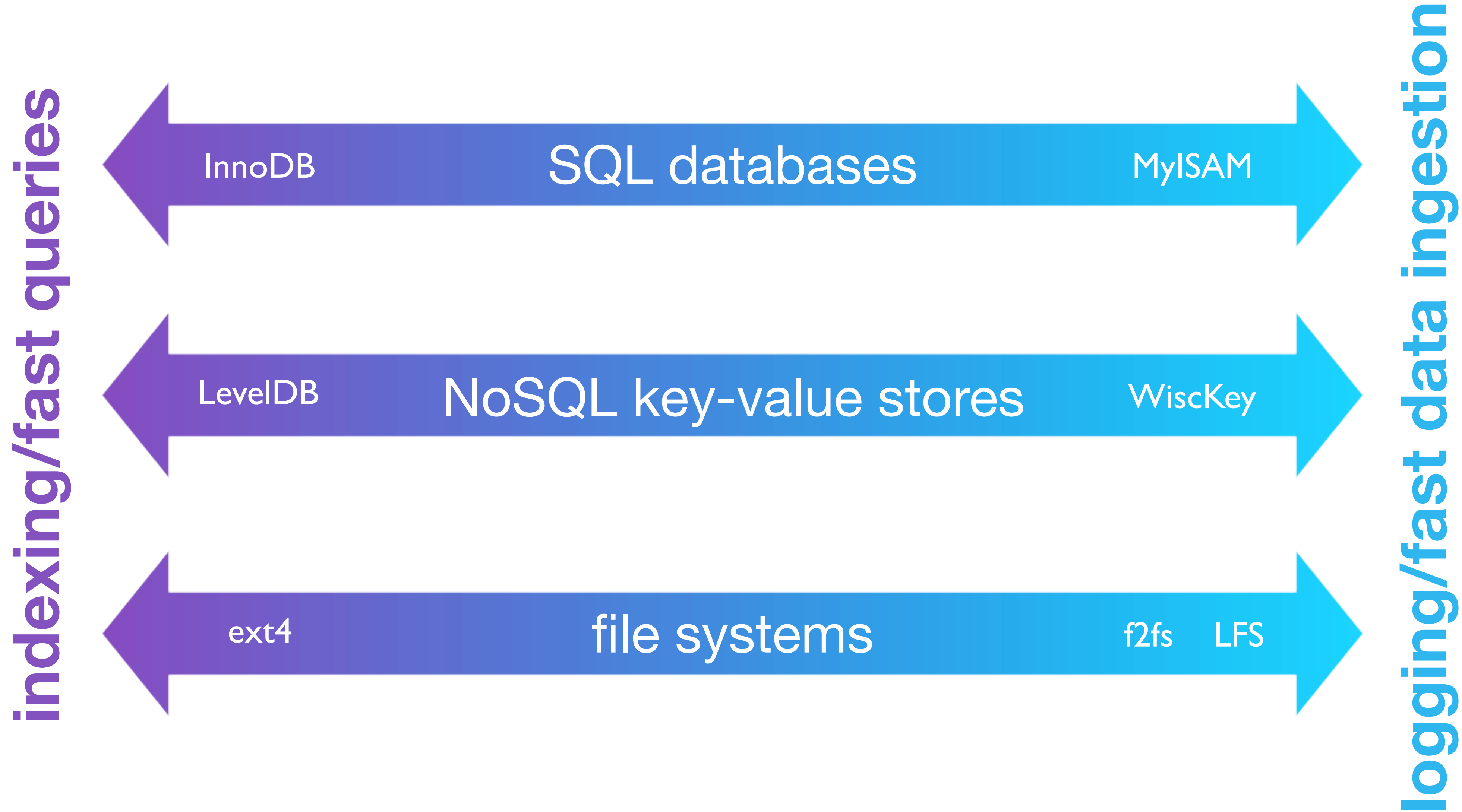
- clustered indexes ⟷ unclustered indexes



Indexing vs logging spans domains & decades

The tradeoff comes under many different names and guises:

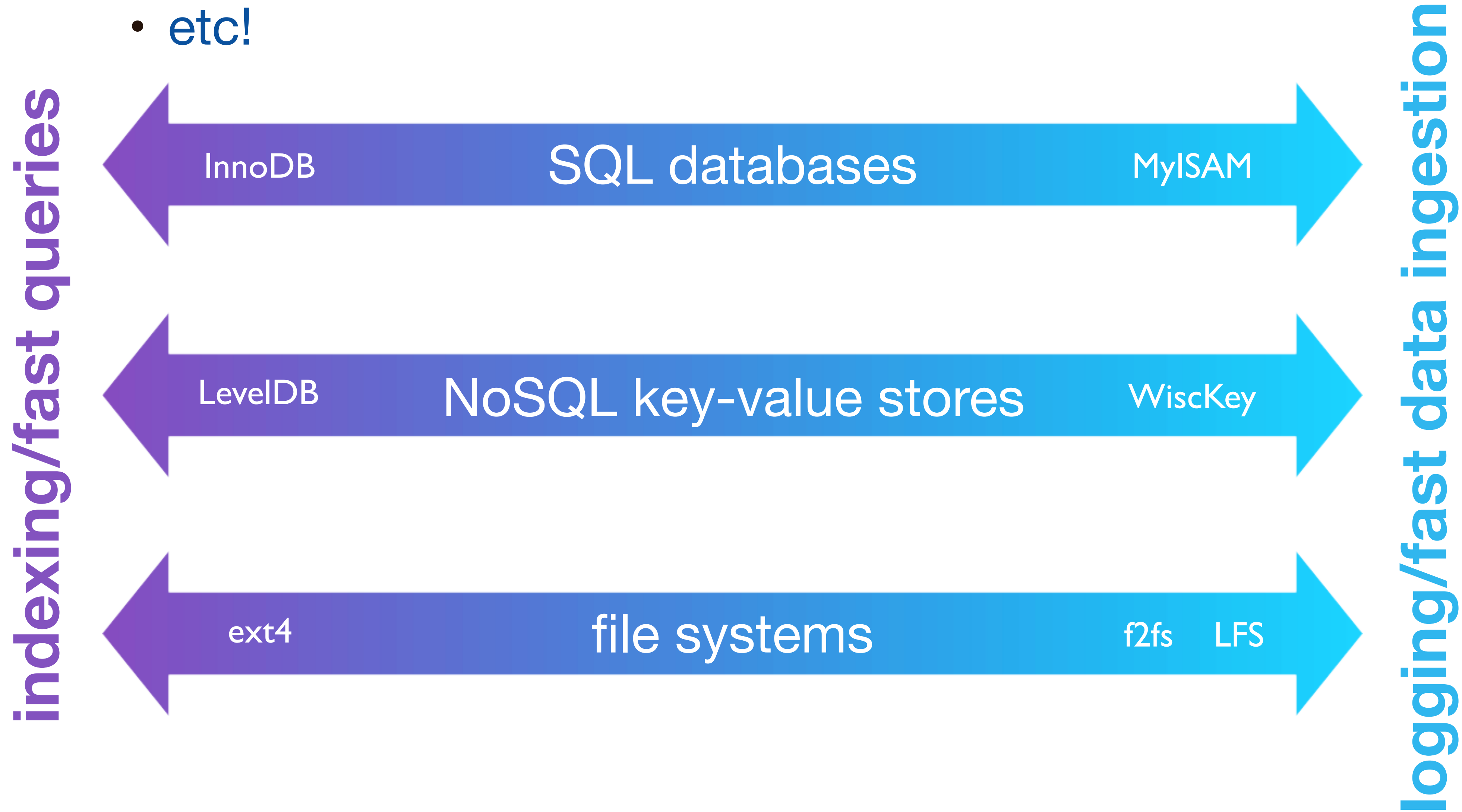
- clustered indexes ⟷ unclustered indexes
- in-place file systems ⟷ log-structured file systems



Indexing vs logging spans domains & decades

The tradeoff comes under many different names and guises:

- clustered indexes ⟷ unclustered indexes
- in-place file systems ⟷ log-structured file systems
- etc!

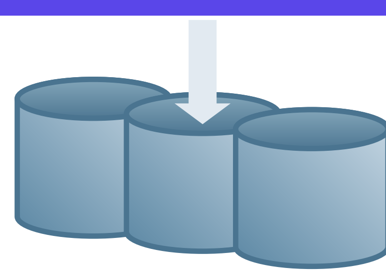


Indexing vs logging: universal data structures question

DBs, kv-stores, and file systems are different beasts.

But they grapple with the similar data-structures problems.

SQL database	noSQL database	file system
• SQL processing • query optimization	• key-value operations	• file and directory operations
persistent data structure	persistent data structure	persistent data structure



Disk/SSD

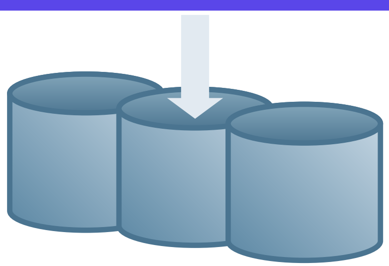
Indexing vs logging: universal data structures question

DBs, kv-stores, and file systems are different beasts.

But they grapple with the similar data-structures problems.

Similar problems ⟹ similar solutions

SQL database	noSQL database	file system
• SQL processing • query optimization	• key-value operations	• file and directory operations
persistent data structure	persistent data structure	persistent data structure



Disk/SSD

Indexing vs logging: universal data structures question

DBs, kv-stores, and file systems are different beasts.

But they grapple with the similar data-structures problems.

Similar problems $\Longrightarrow$ similar solutions

write-optimization

SQL database

- SQL processing
- query optimization

persistent data structure

noSQL database

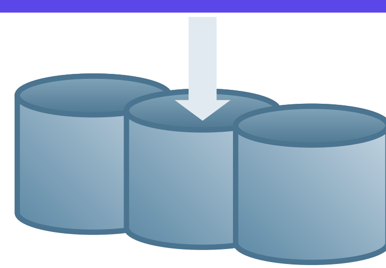
- key-value operations

persistent data structure

file system

- file and directory operations

persistent data structure



Disk/SSD

Indexing vs logging: universal data structures question

Some "write-optimized" data structures can mitigate or overcome the indexing-logging trade-off.

At our DB company Tokutek,* we sold open-source write-optimized databases.

Since it was sold, we've built an open-source file system.

TokuDB
SQL database

- SQL processing
- query optimization

TokuDB core

TokuMX
noSQL database

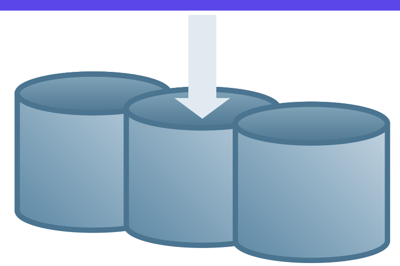
- noSQL processing

TokuDB core

BetrFS
file system

- file and directory operations

TokuDB core



Disk/SSD

*acquired by Percona

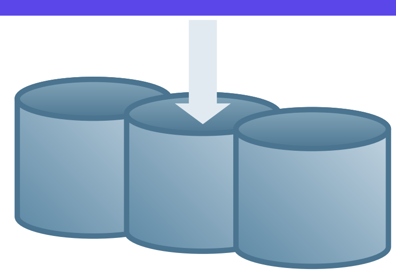
Indexing vs logging: universal data structures question

Some "write-optimized" data structures can mitigate or overcome the indexing-logging trade-off.

At our DB company Tokutek,* we sold open-source write-optimized databases.

Since it was sold, we've built an open-source file system.

TokuDB SQL database	TokuMX noSQL database	BetrFS file system
• SQL processing • query optimization	• noSQL processing	• file and directory operations
TokuDB core	TokuDB core	TokuDB core



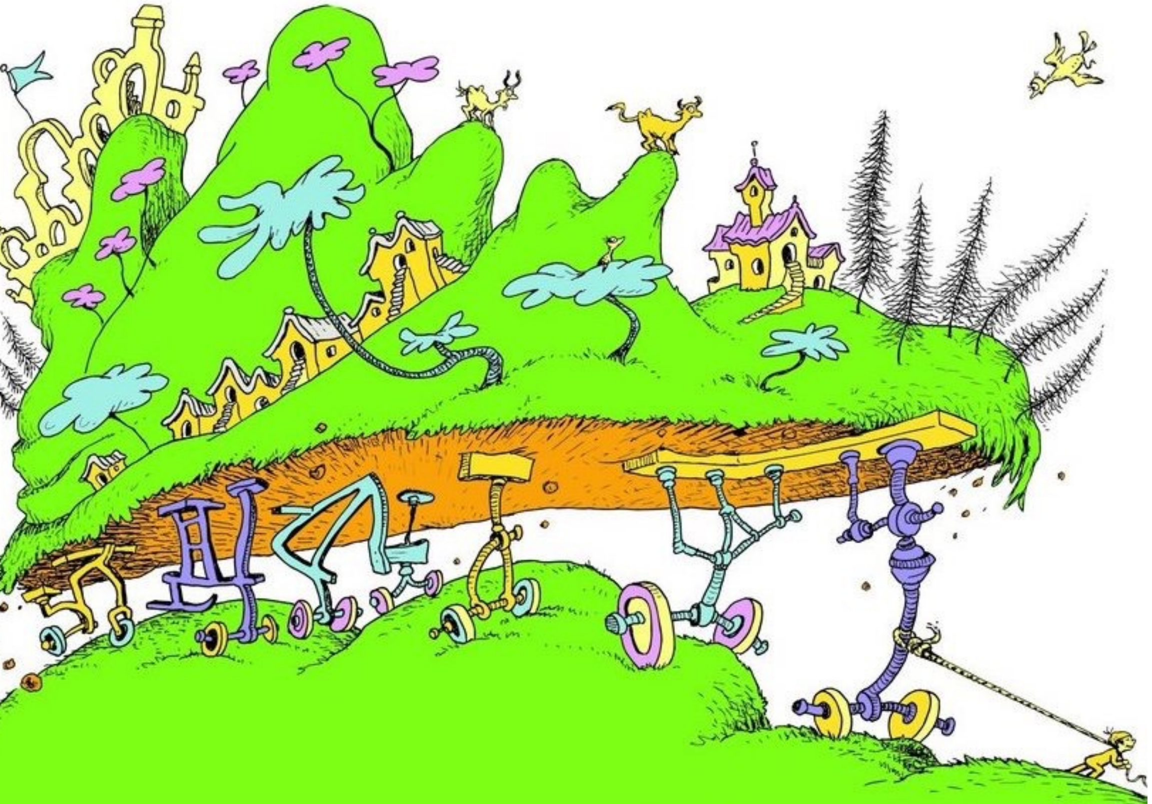
Disk/SSD

I'll talk about my experiences using the same data structure to help all three systems.

*acquired by Percona

The performance landscape is fundamentally changing.

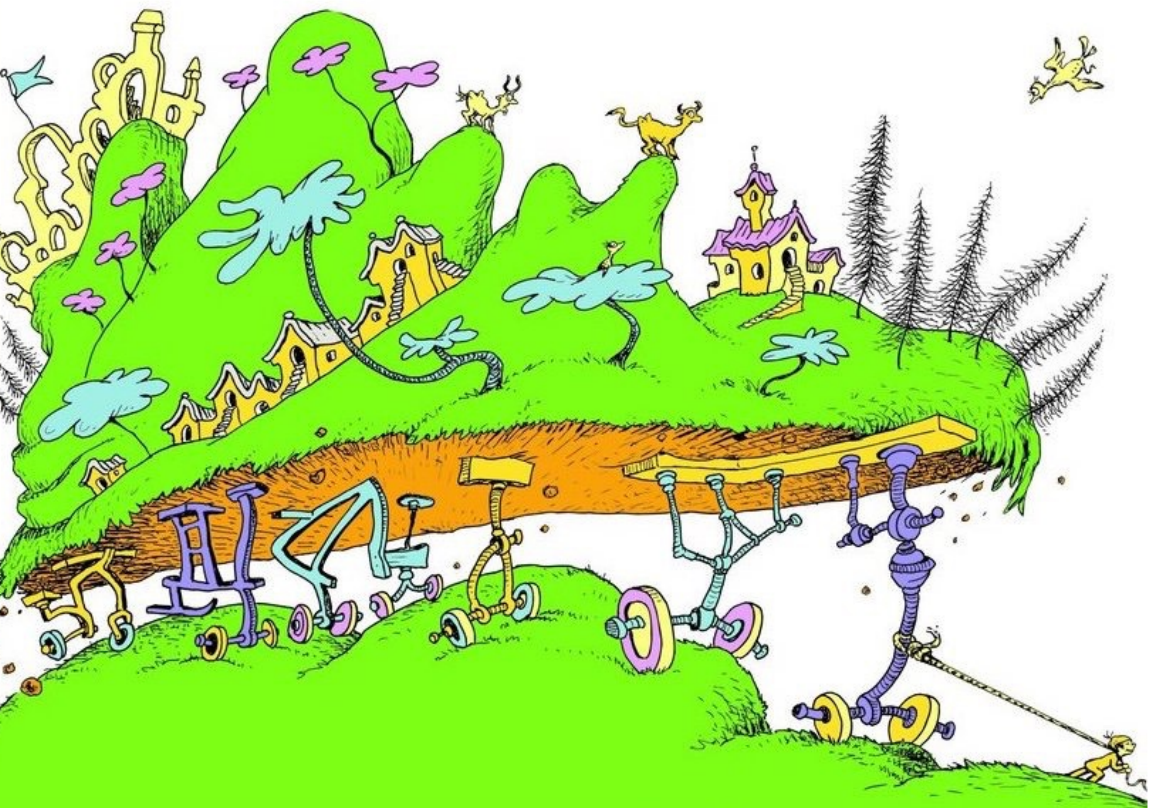
- New data structures
- New hardware



Point of this talk

The performance landscape is fundamentally changing.

- New data structures
- New hardware



This has created tons of new research opportunities.

- For algorithmists/theorists
- For systems builders

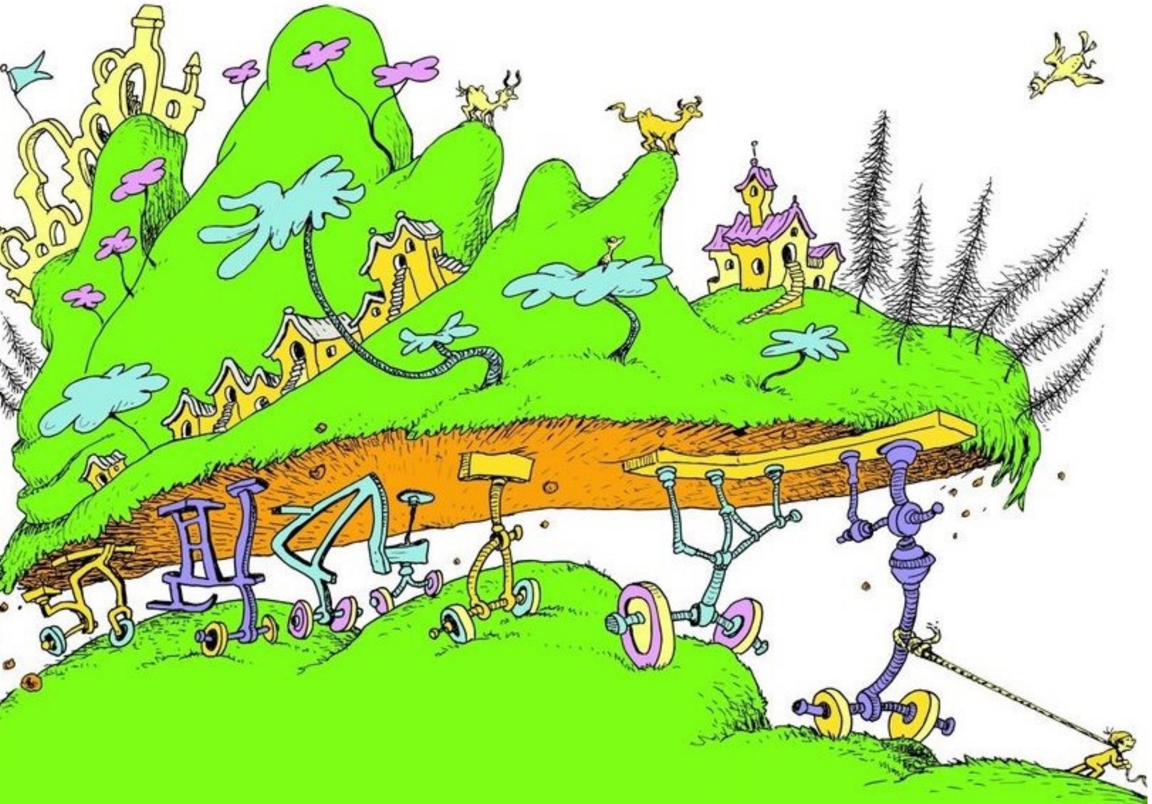
Point of this talk

The performance landscape is fundamentally changing.

- New data structures
- New hardware

This has created tons of new research opportunities.

- For algorithmists/theorists
- For systems builders



There's still lots to do.

An algorithmic view of the insert-query tradeoff

Performance characteristics of storage

Sequential access is fast.

Performance characteristics of storage

Sequential access is fast.

Random access is slower.

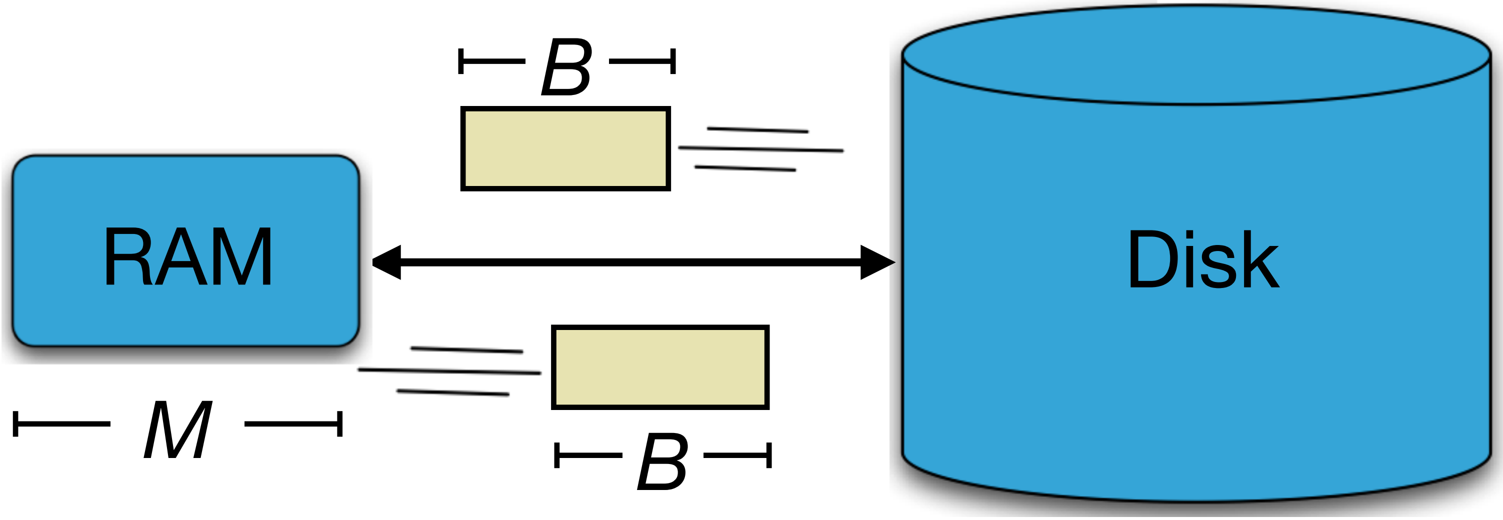
A model for I/O performance

How computation works:

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

Goal: Minimize # of I/Os

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



Disk-Access Machine (DAM) model **[Aggarwal+Vitter '88]**

I/Os are slow

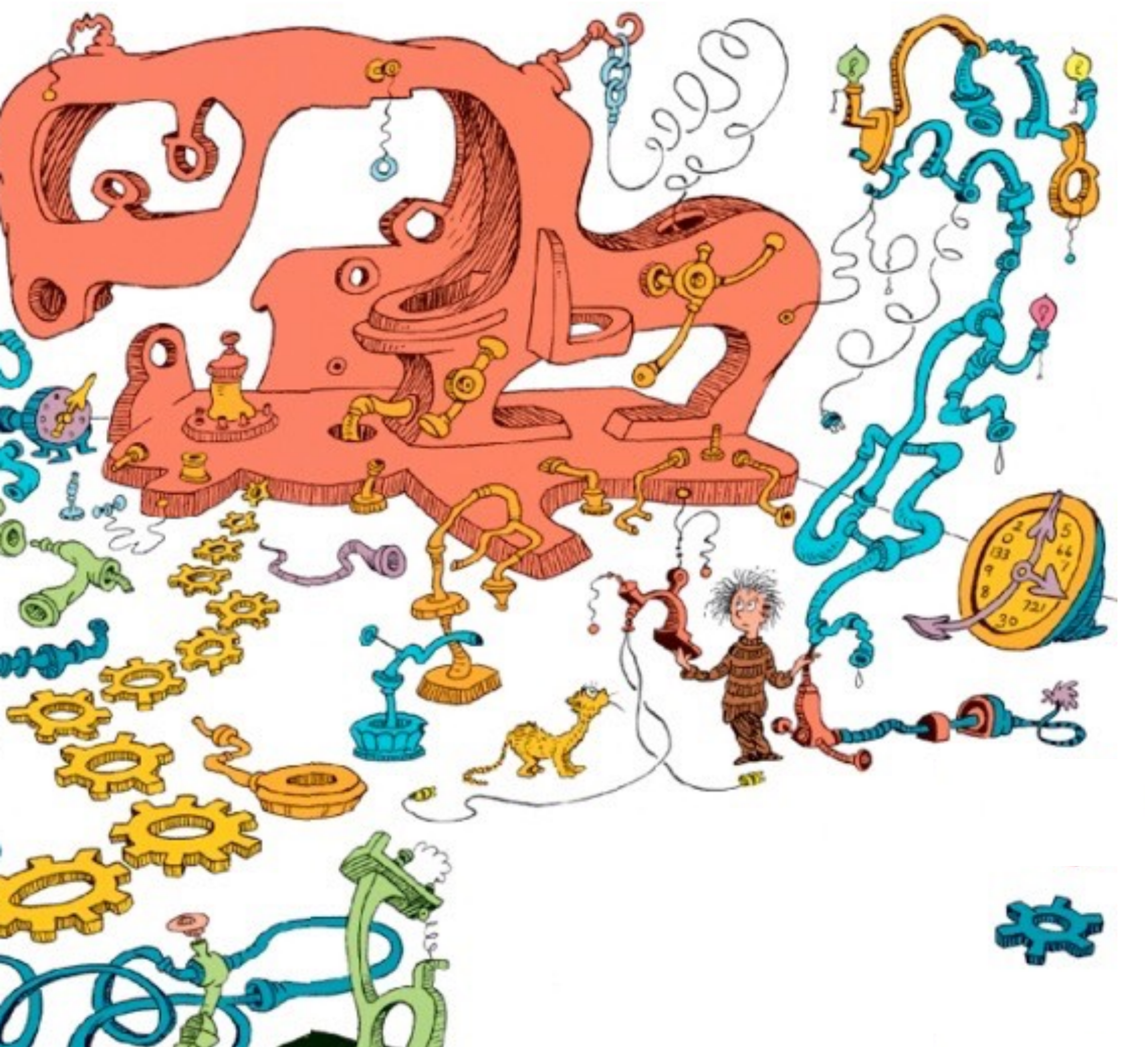
RAM: ~60 nanoseconds per access

Disks: ~6 milliseconds per access.

Analogy:

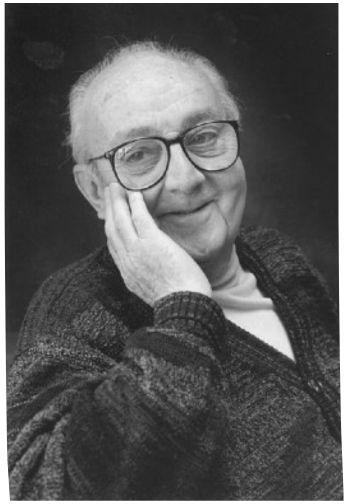
- RAM $\propto$ escape velocity from earth (40,250 kph)
- disk $\propto$ walking speed of the giant tortoise (0.4 kph)

How realistic is the DAM model?

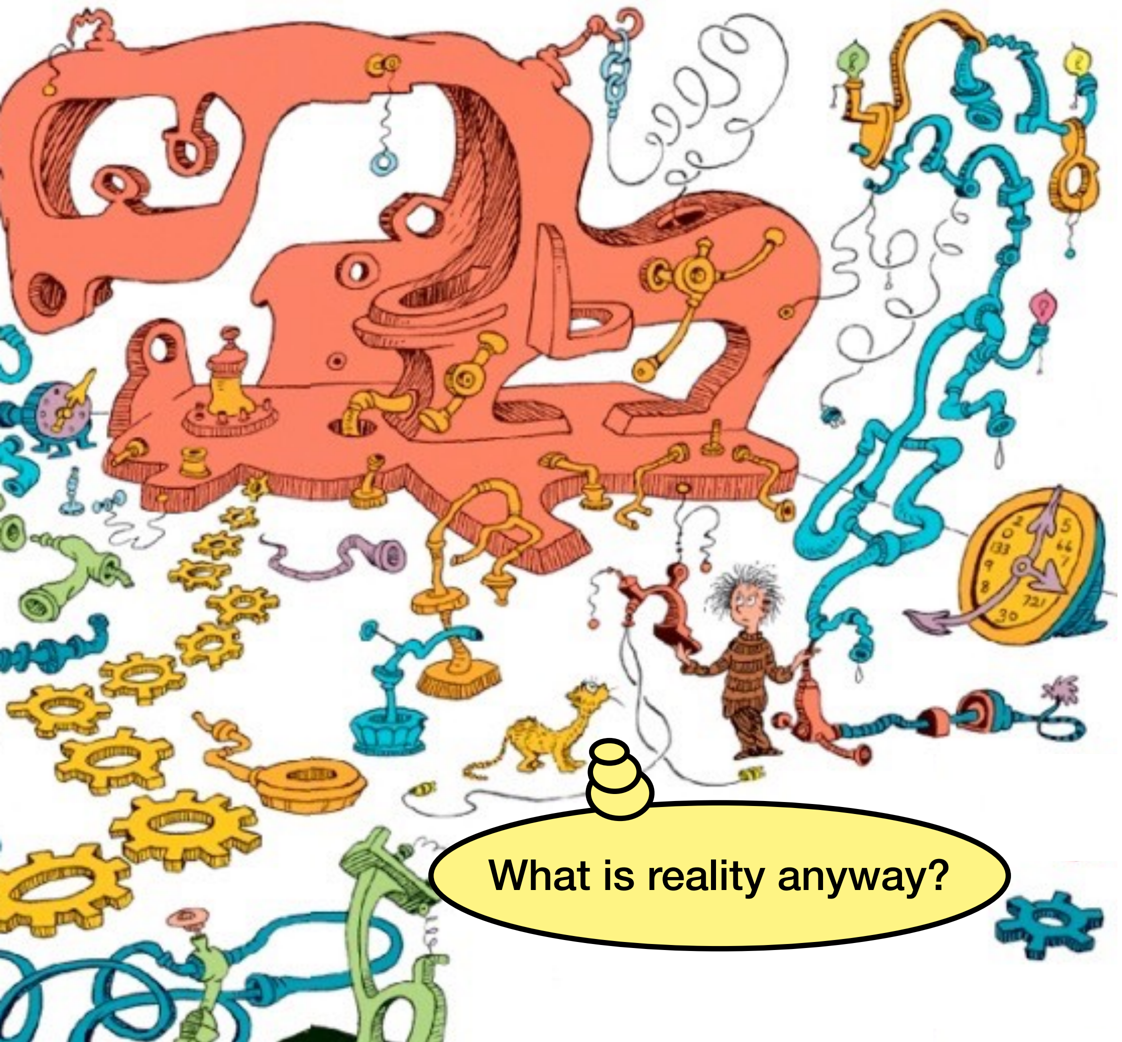


How realistic is the DAM model?

"All models are wrong, but some are useful"

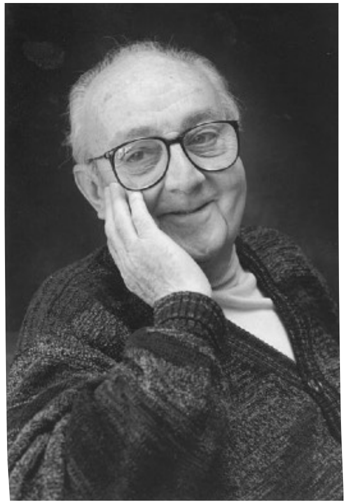


[George Box 1978]

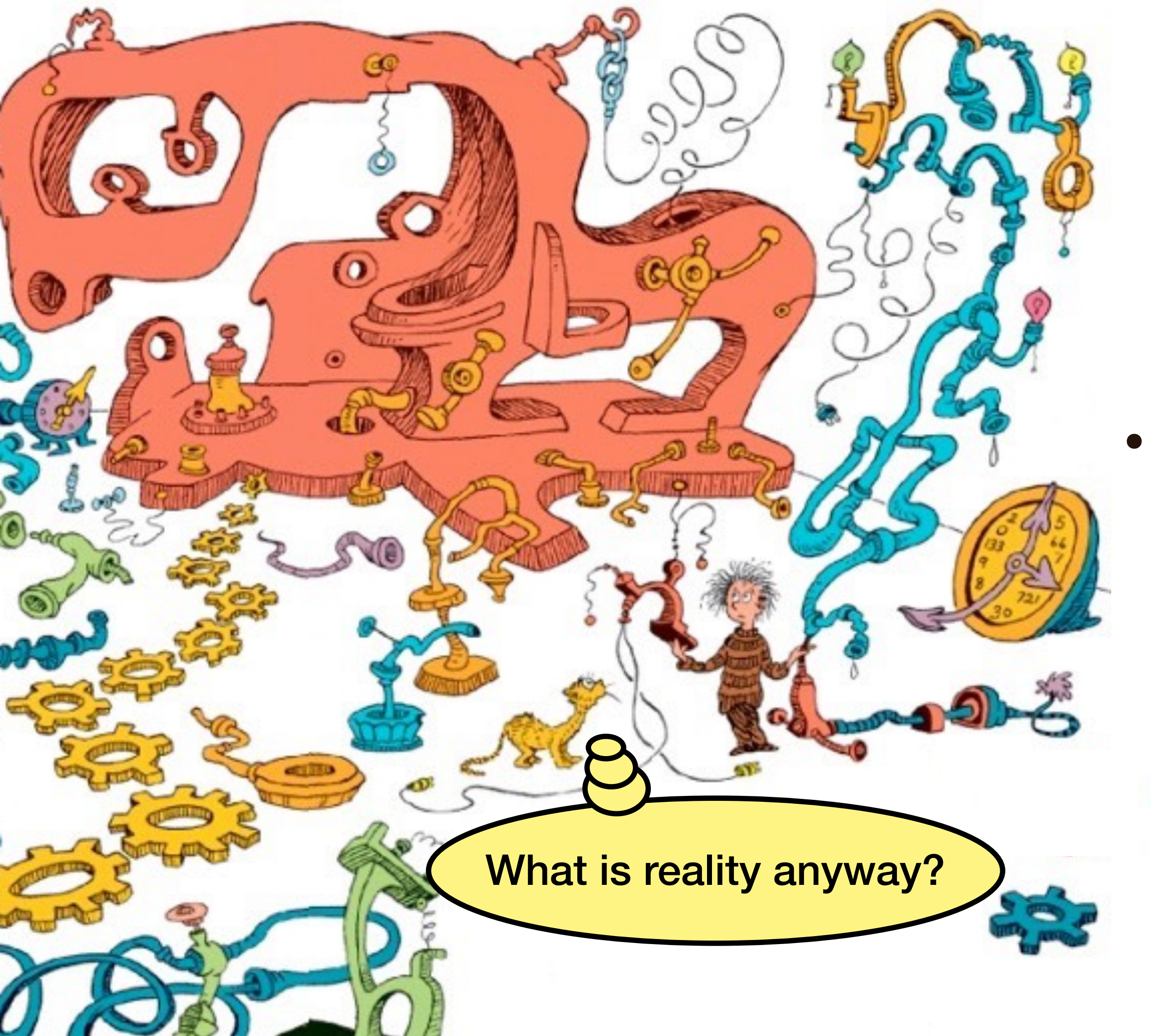


How realistic is the DAM model?

"All models are wrong, but some are useful"



[George Box 1978]

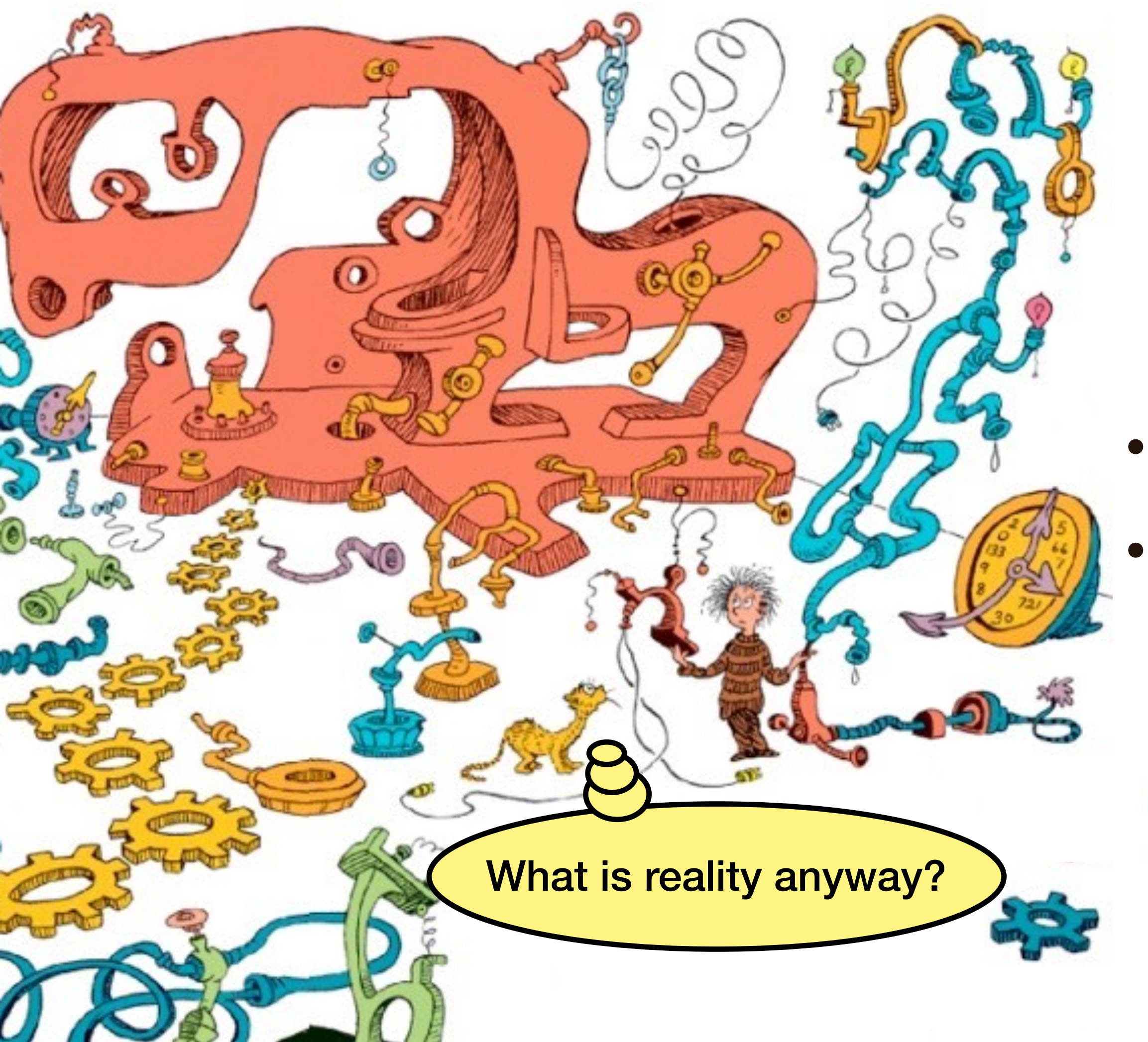


- Unrealistic in various ways.

How realistic is the DAM model?

"All models are wrong, but some are useful"

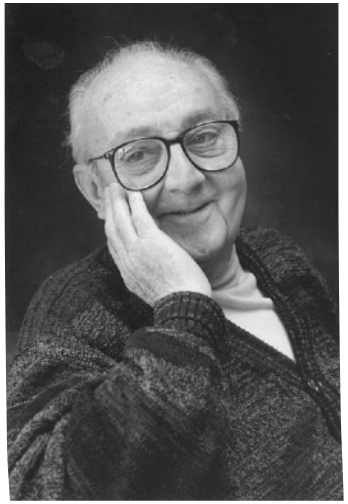
[George Box 1978]



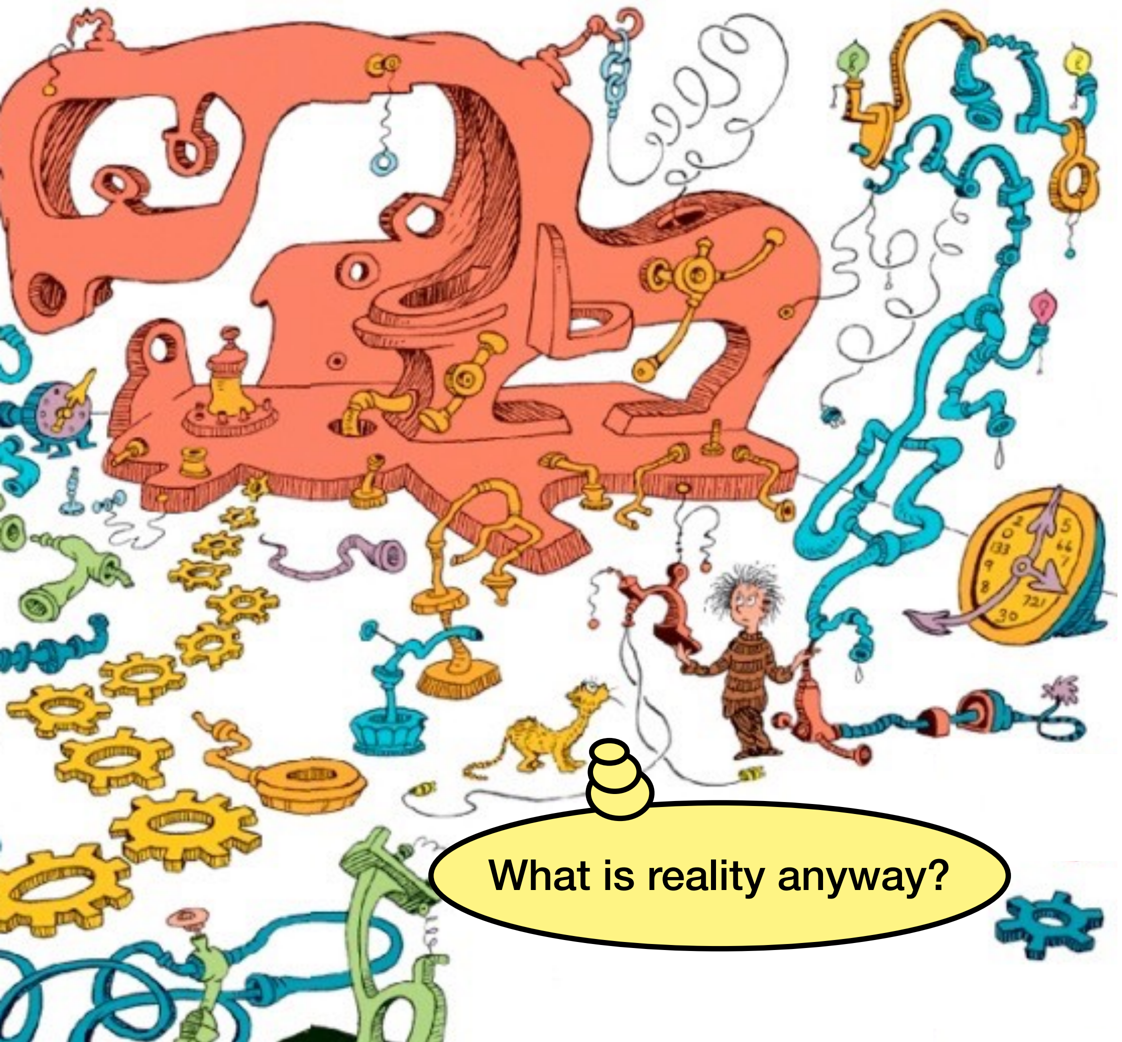
- Unrealistic in various ways.
- Great for reasoning about I/O and for high-level design.

How realistic is the DAM model?

"All models are wrong, but some are useful”



[George Box 1978]



- Unrealistic in various ways.
- Great for reasoning about I/O and for high-level design.
- You can optimize the model to hone constants.

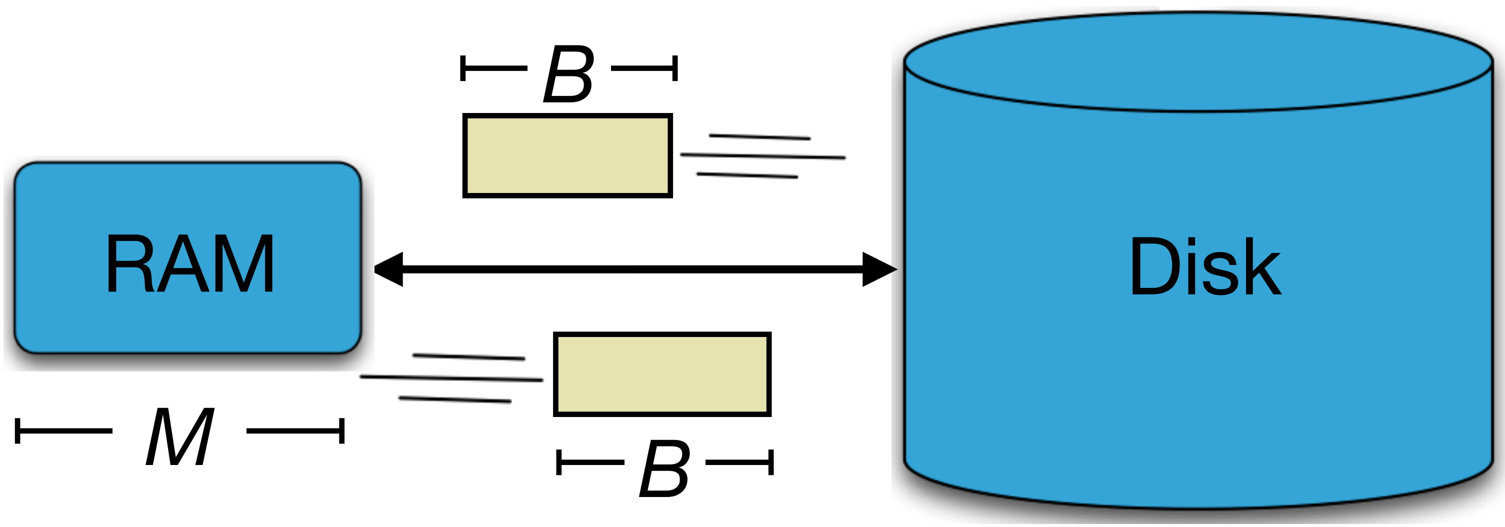
A model for I/O performance

How computation works:

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

Goal: Minimize # of I/Os

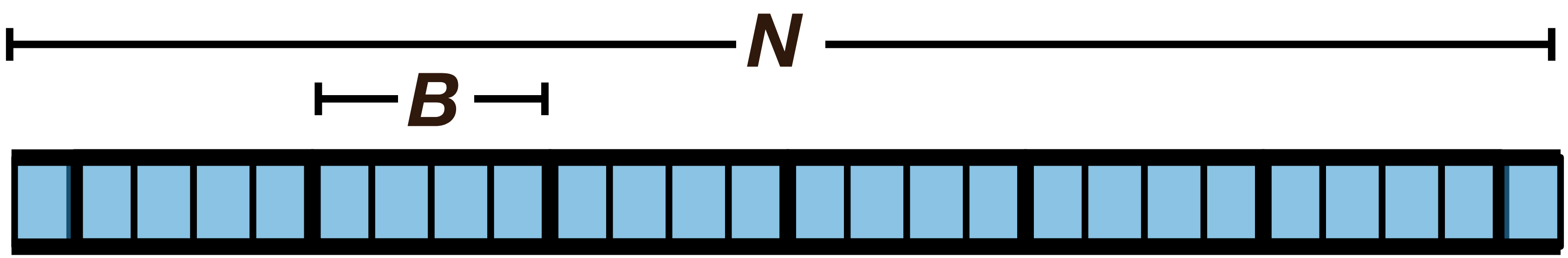
- Performance bounds are parameterized by block size ***B***, memory size ***M***, data size ***N***.



Disk-Access Machine (DAM) model **[Aggarwal+Vitter '88]**

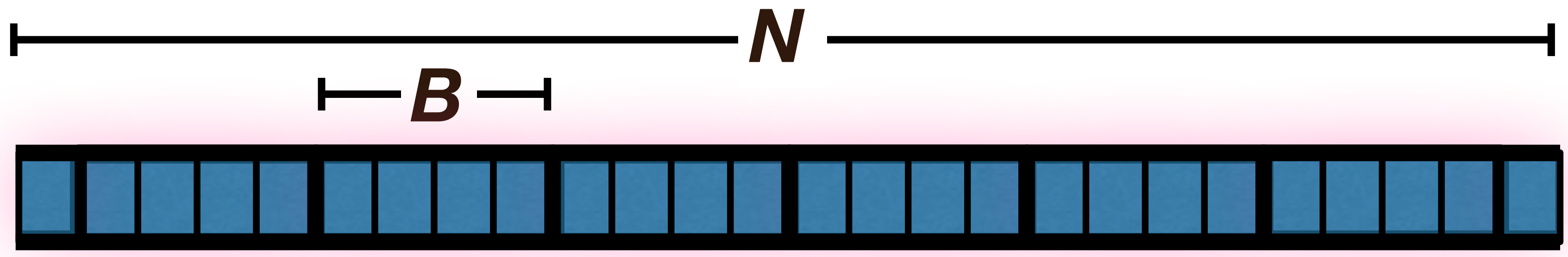
What's the I/O cost for logging?

I/O cost for logging.



What's the I/O cost for logging?

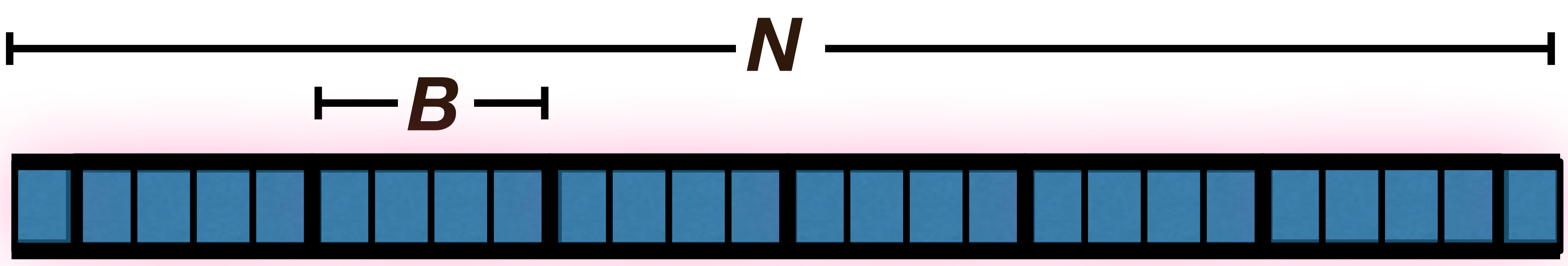
I/O cost for logging.



What's the I/O cost for logging?

I/O cost for logging.

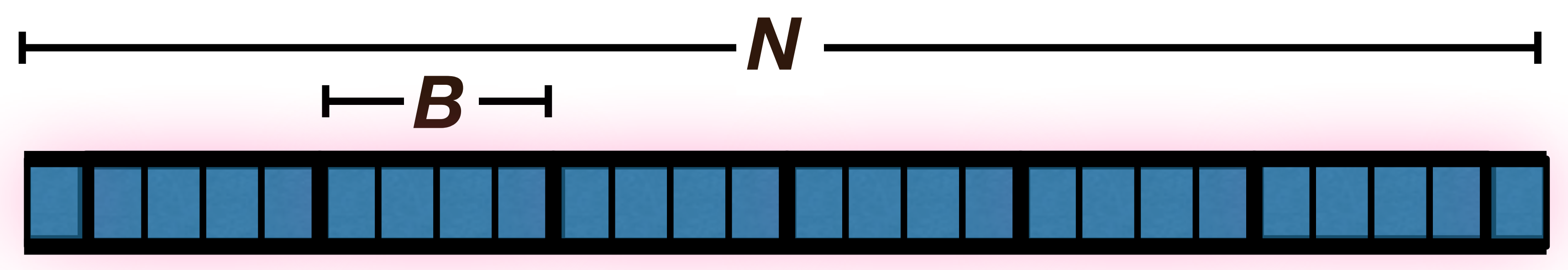
- **query:** scan all blocks $\Longrightarrow$ $O(N/B)$



What's the I/O cost for logging?

I/O cost for logging.

- **query:** scan all blocks $\Longrightarrow$ $O(N/B)$
- **insert:** append to end of log $\Longrightarrow$ $O(1/B)$



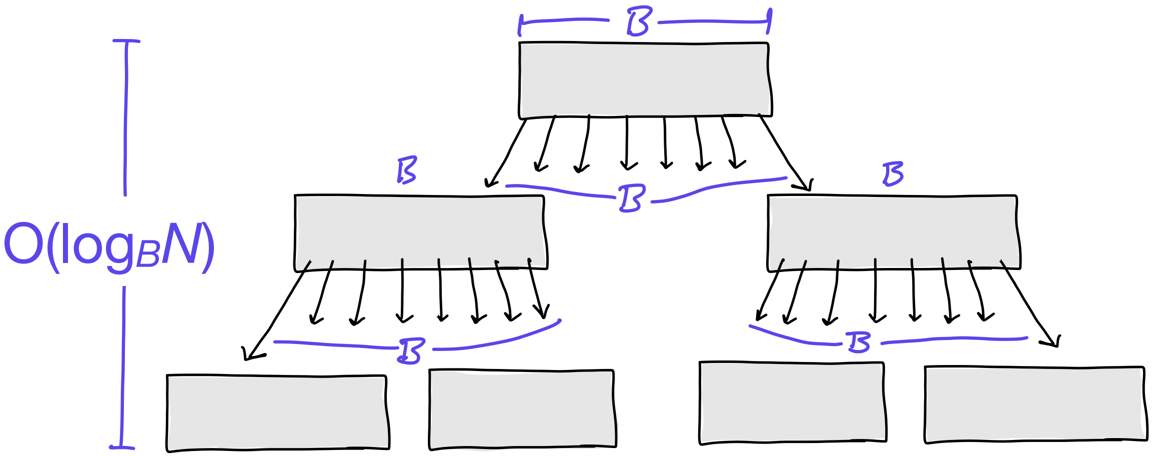
What's the I/O cost for indexing?

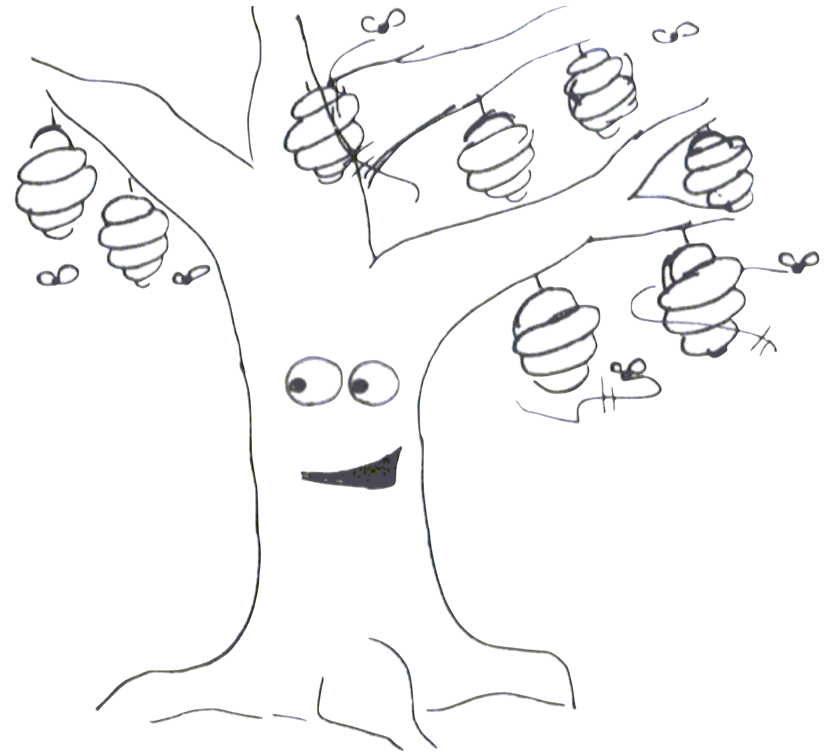
Q: What's the I/O cost for indexing?

A: It depends on the indexing data structure.

What's the I/O cost for indexing?

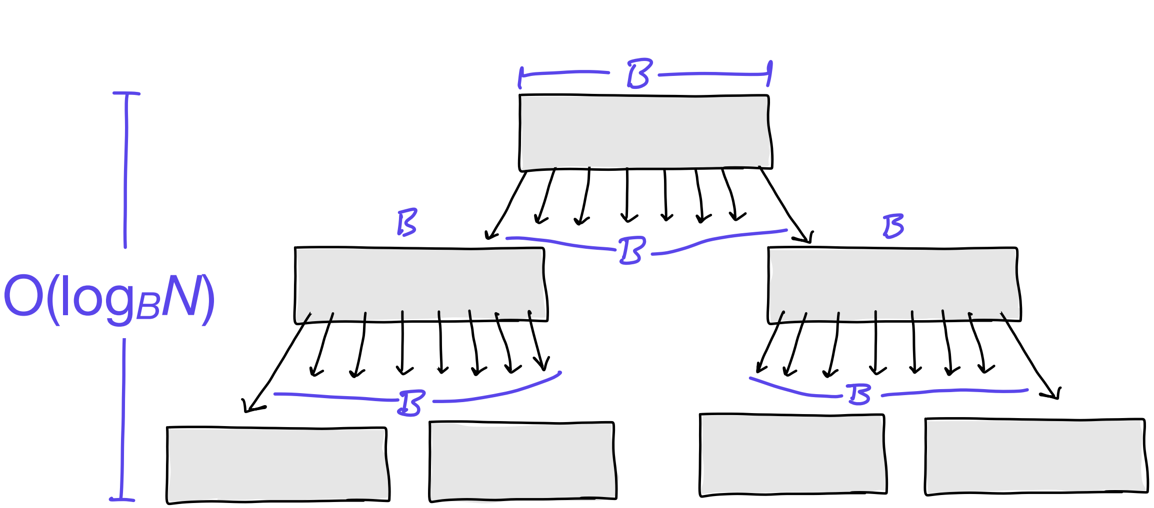
The classic indexing structure is the B-tree.

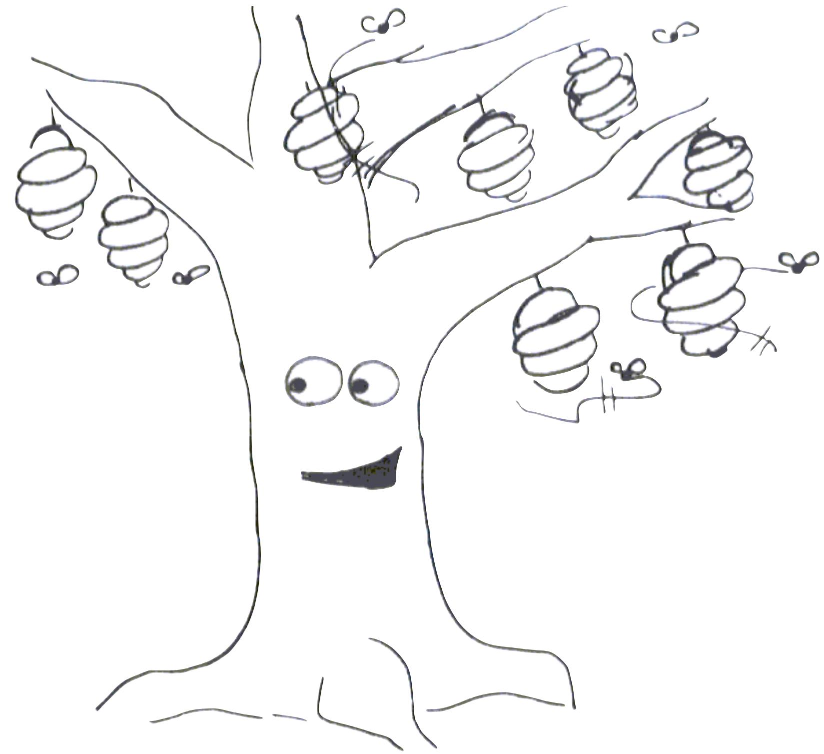




What's the I/O cost for indexing?

The classic indexing structure is the B-tree.

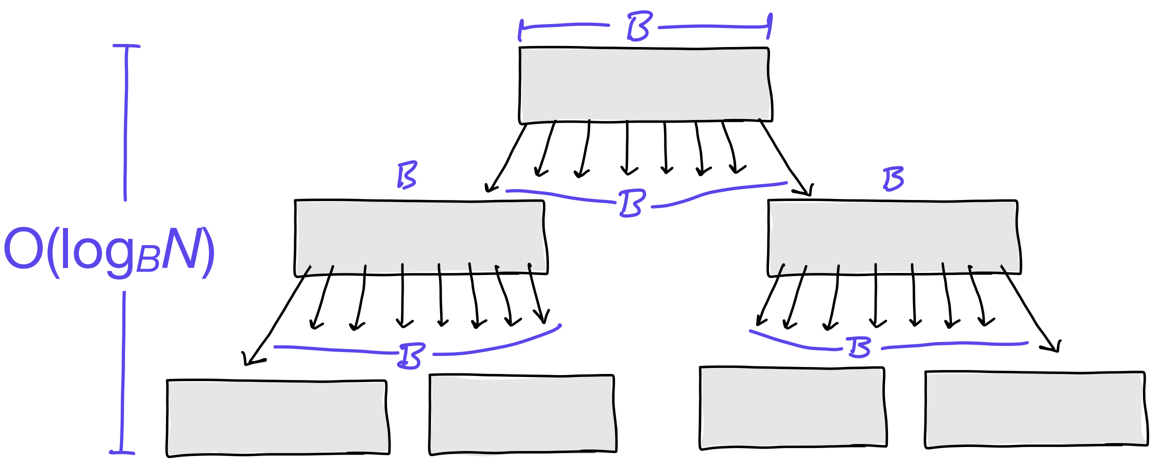


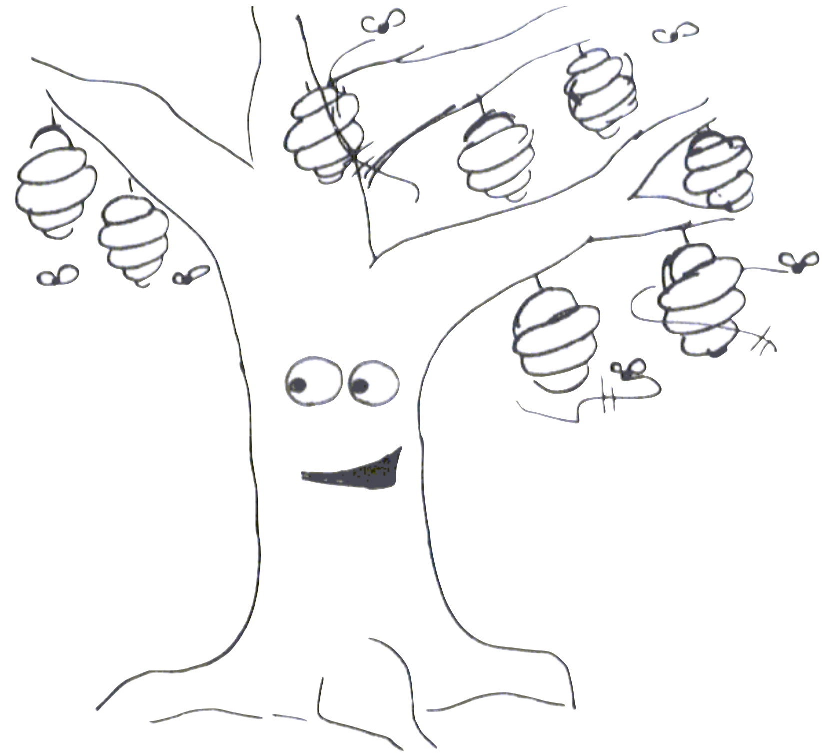


Queries: O(log$_B$*N*)
Inserts: O(log$_B$*N*)

What's the I/O cost for indexing?

The classic indexing structure is the B-tree.



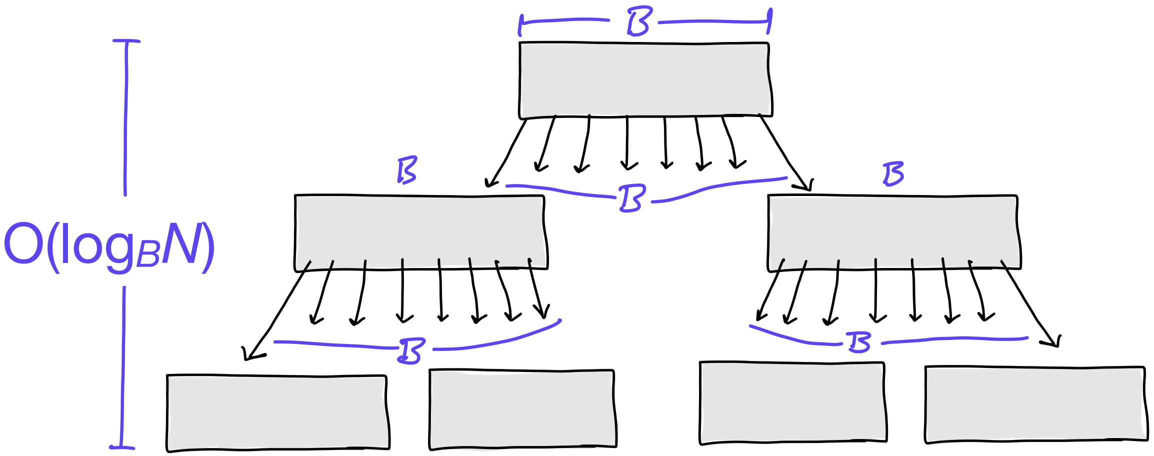


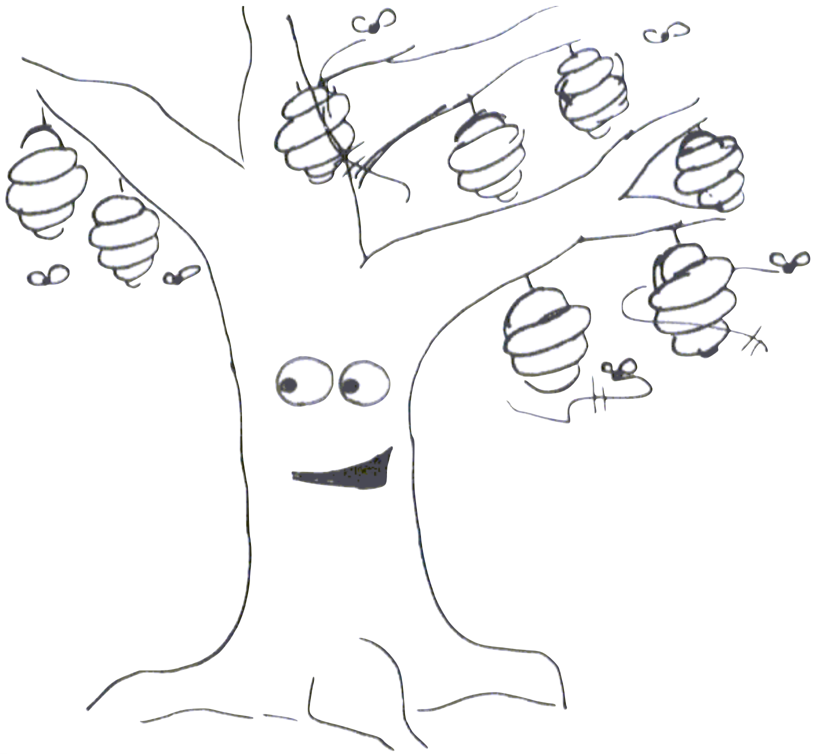
Queries: $O(\log_B N)$

Inserts: $O(\log_B N)$

What's the I/O cost for indexing?

The classic indexing structure is the B-tree.

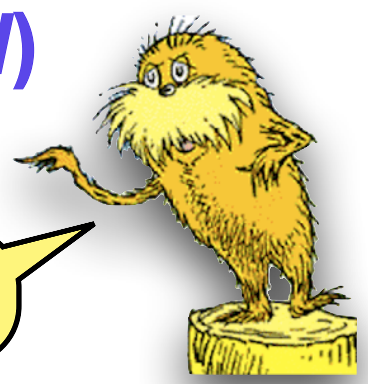




optimal

Queries: $O(\log_B N)$

Inserts: $O(\log_B N)$



not optimal!!

Beating B-tree Bounds

Queries: $O(\log_B N)$

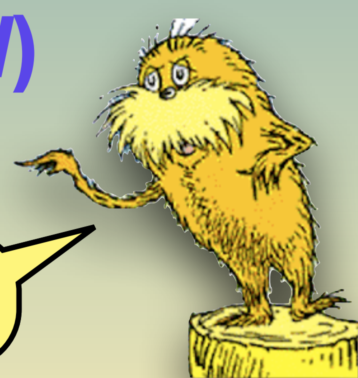
Inserts: $O(\log_B N)$

Goal:
Inserts that run faster than a B-tree.
Queries that don't run slower.

Beating B-tree Bounds

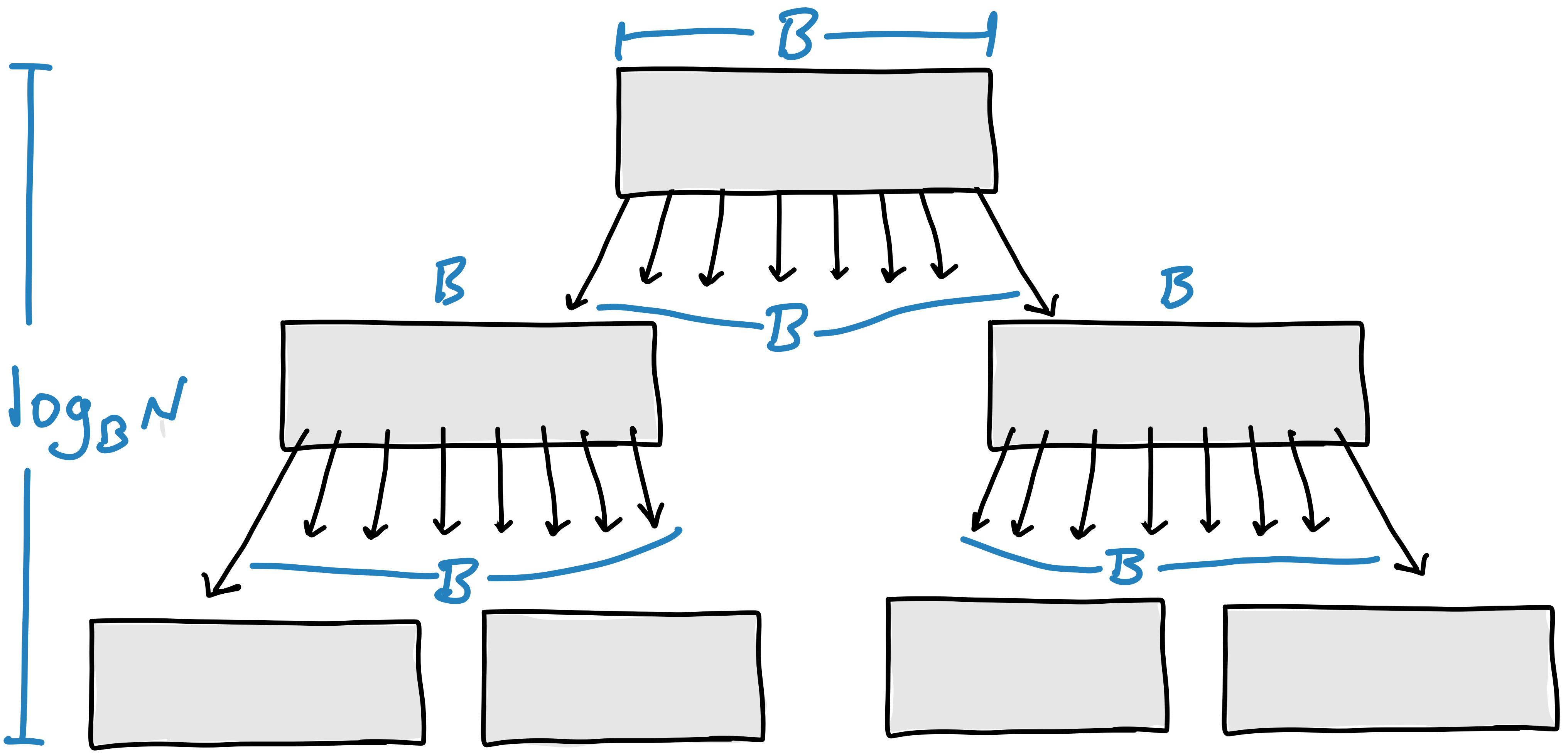
optimal

Queries: $O(\log_B N)$
Inserts: $O(\log_B N)$



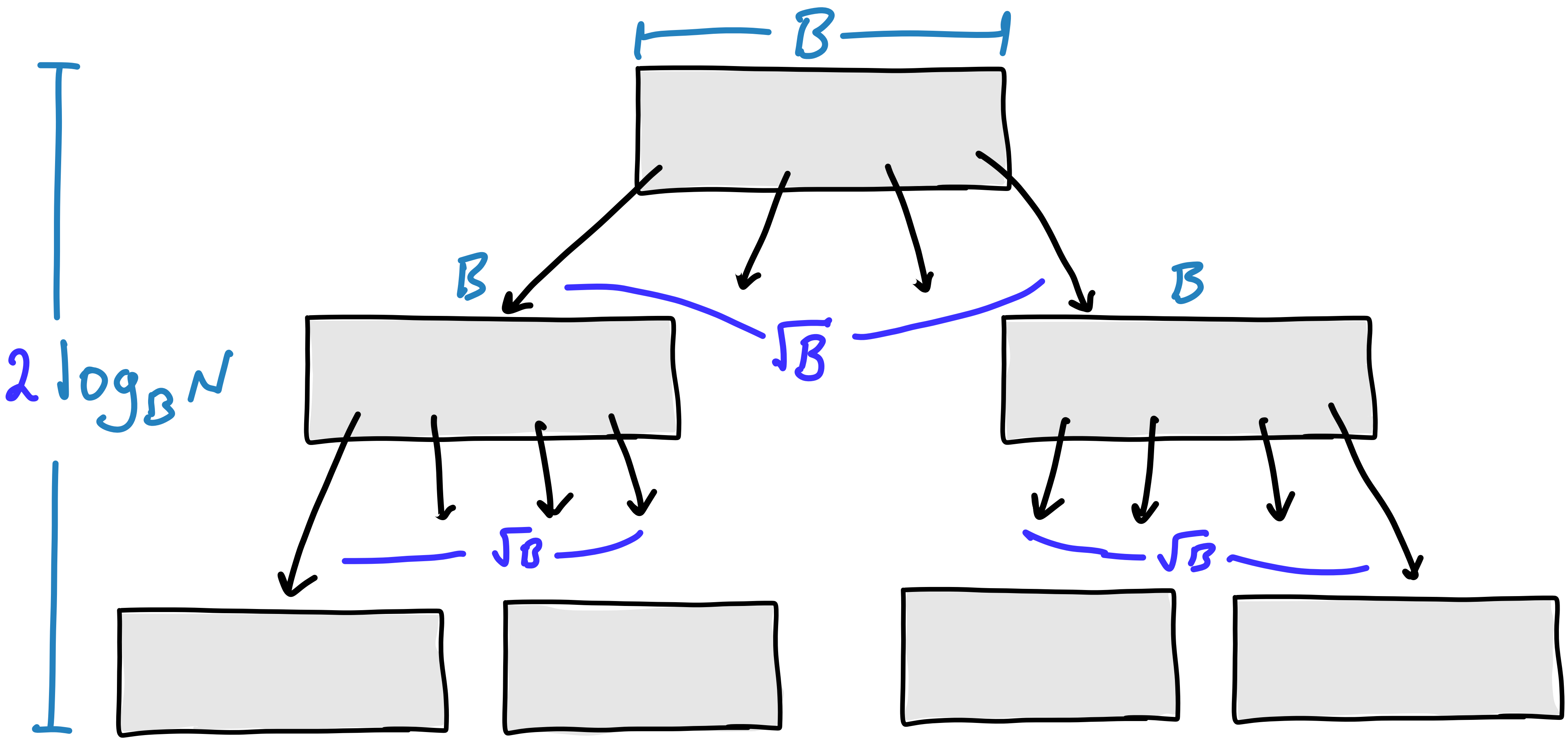
not optimal!!

Start with a regular B-tree

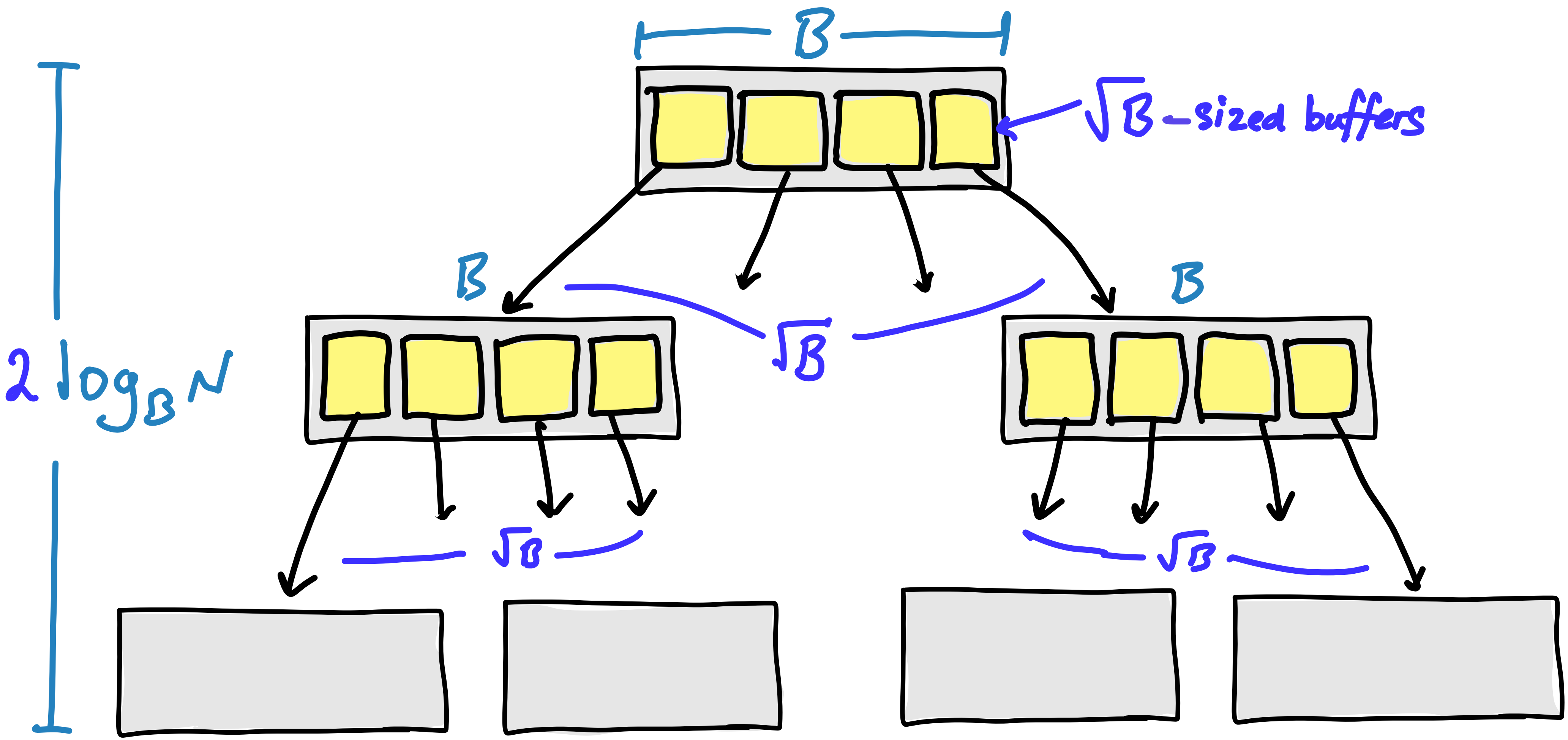


Reduce the fanout.

- Now the nodes are mostly empty

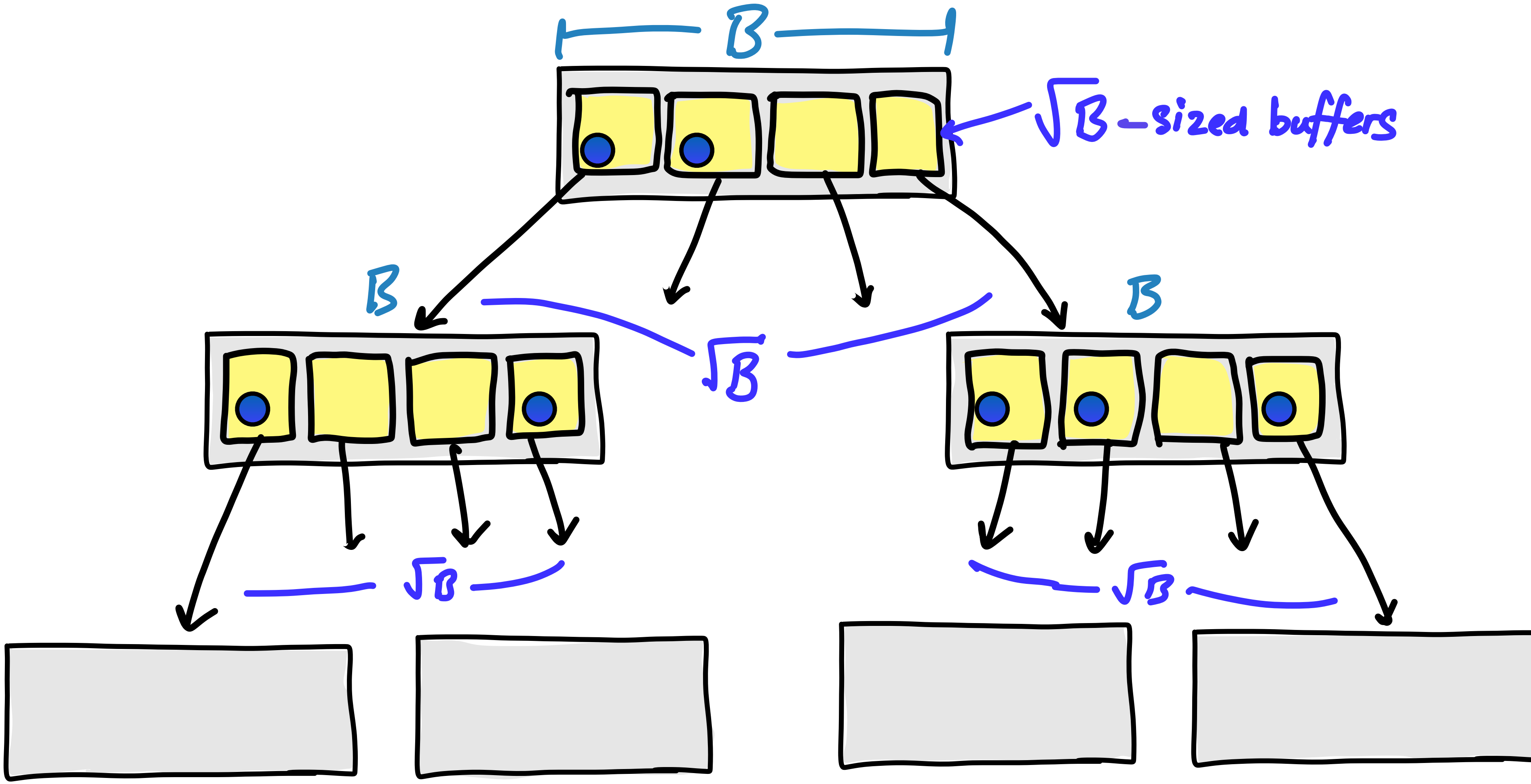


Put $B^{1/2}$-sized buffers in each internal node.



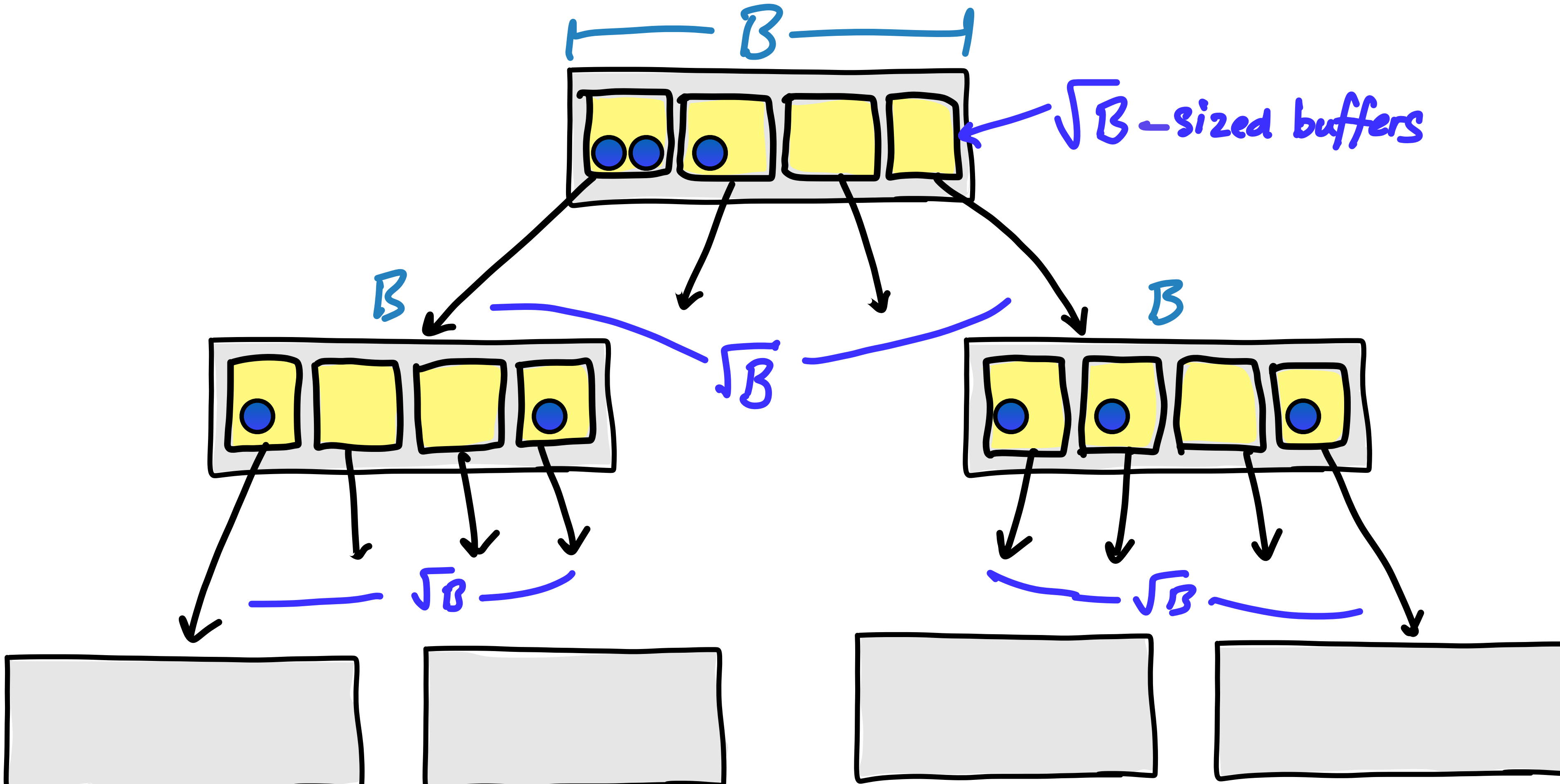
Inserts + deletes:

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



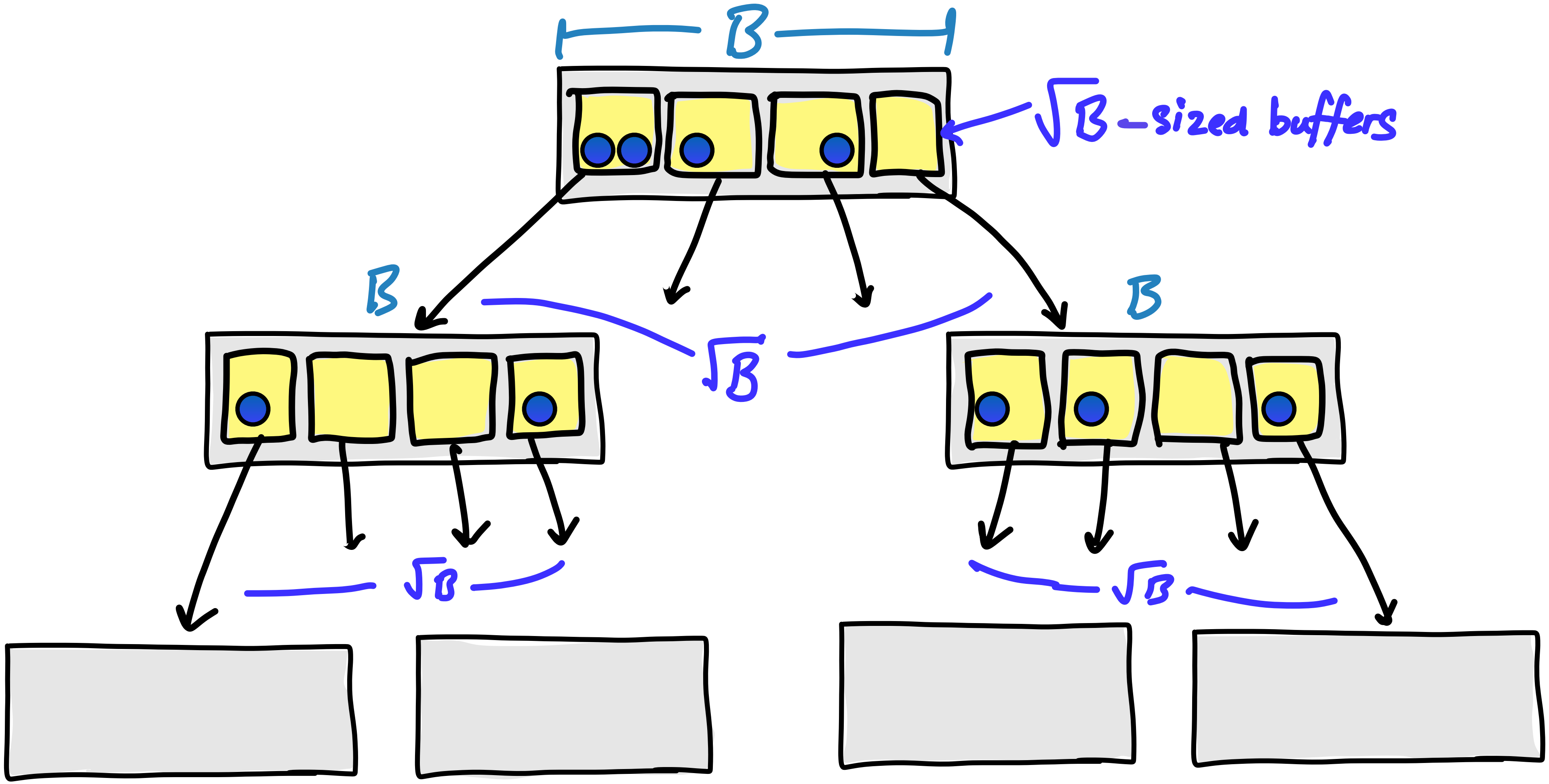
Inserts + deletes:

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



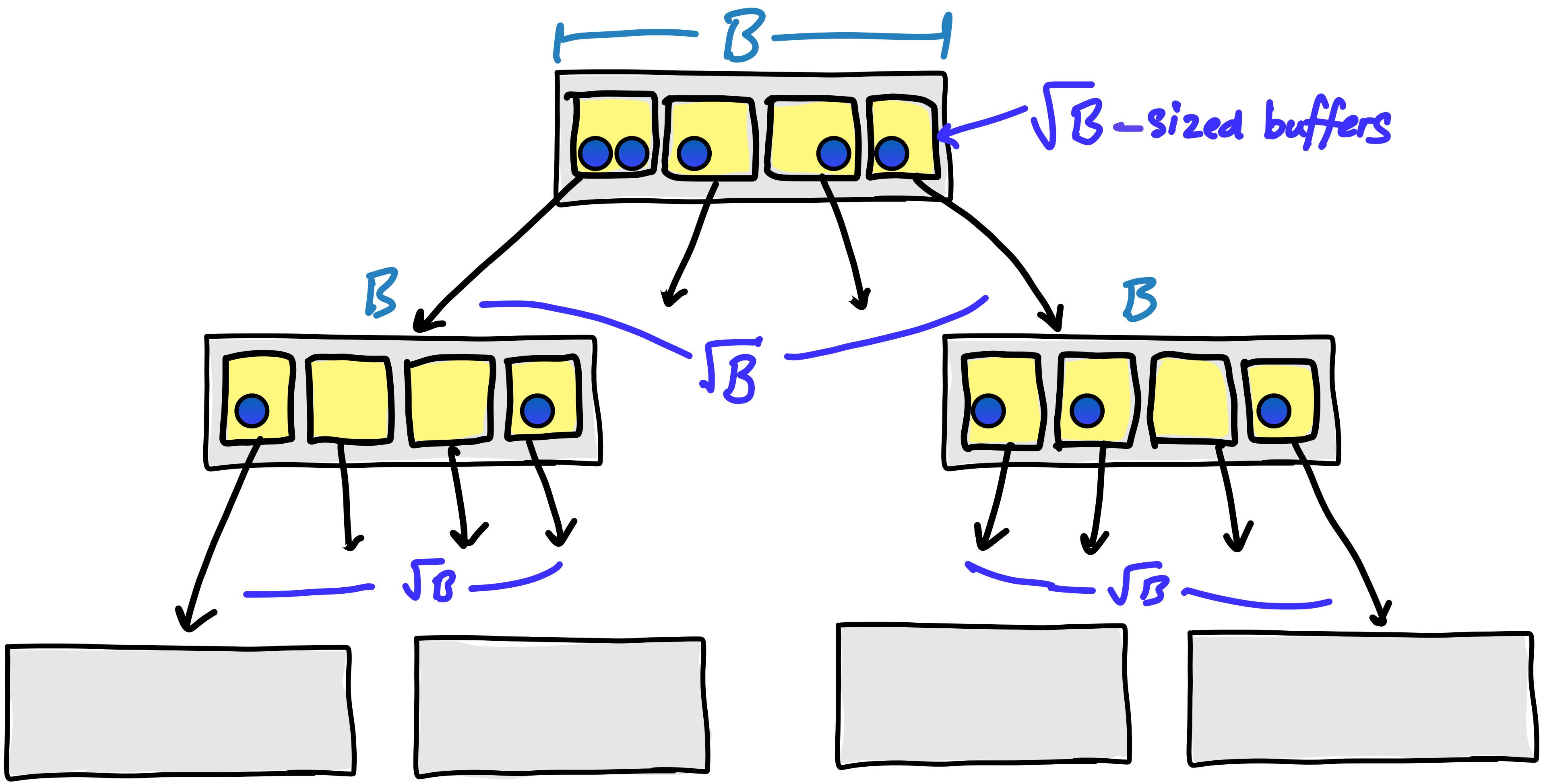
Inserts + deletes:

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



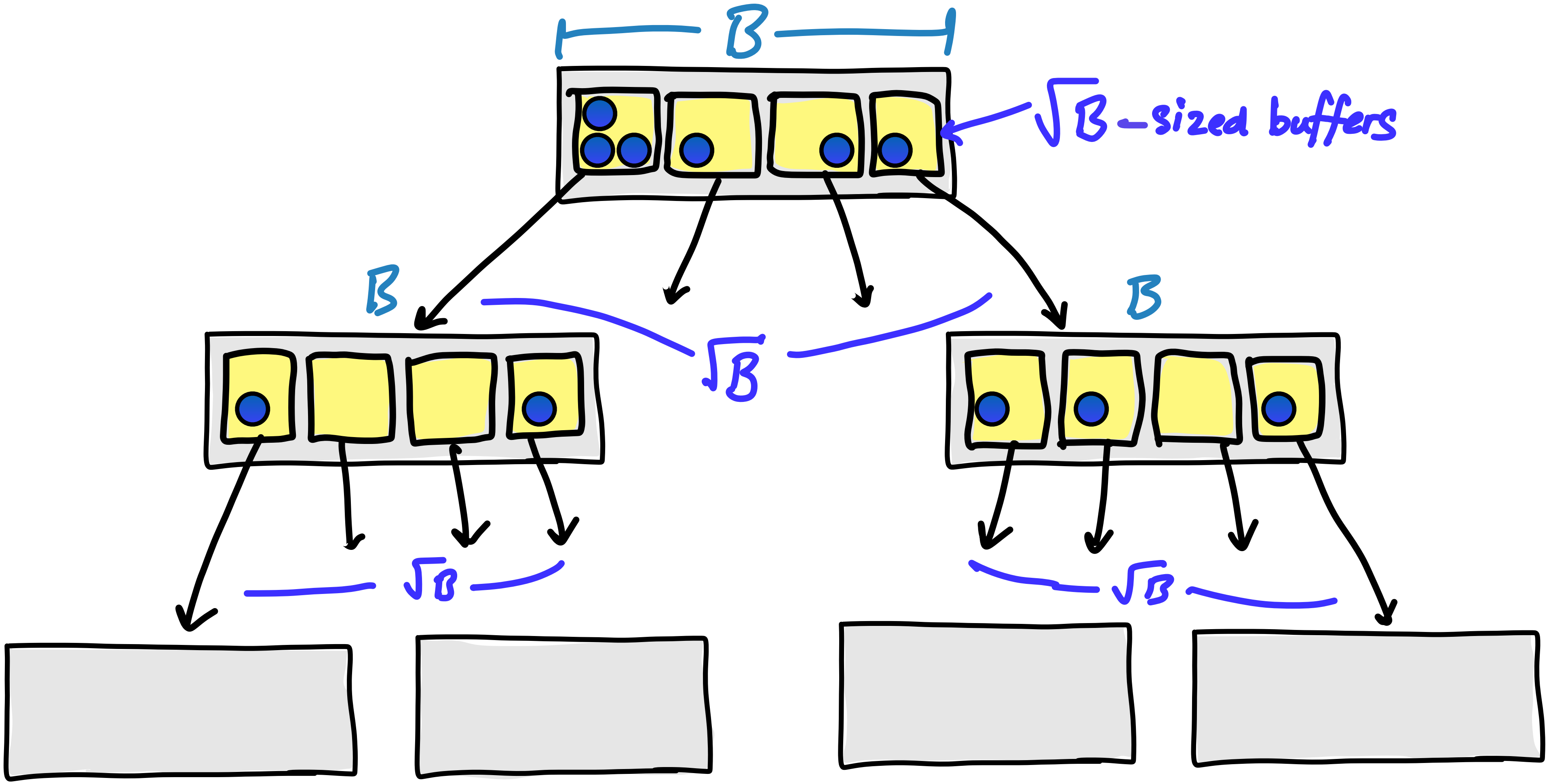
Inserts + deletes:

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



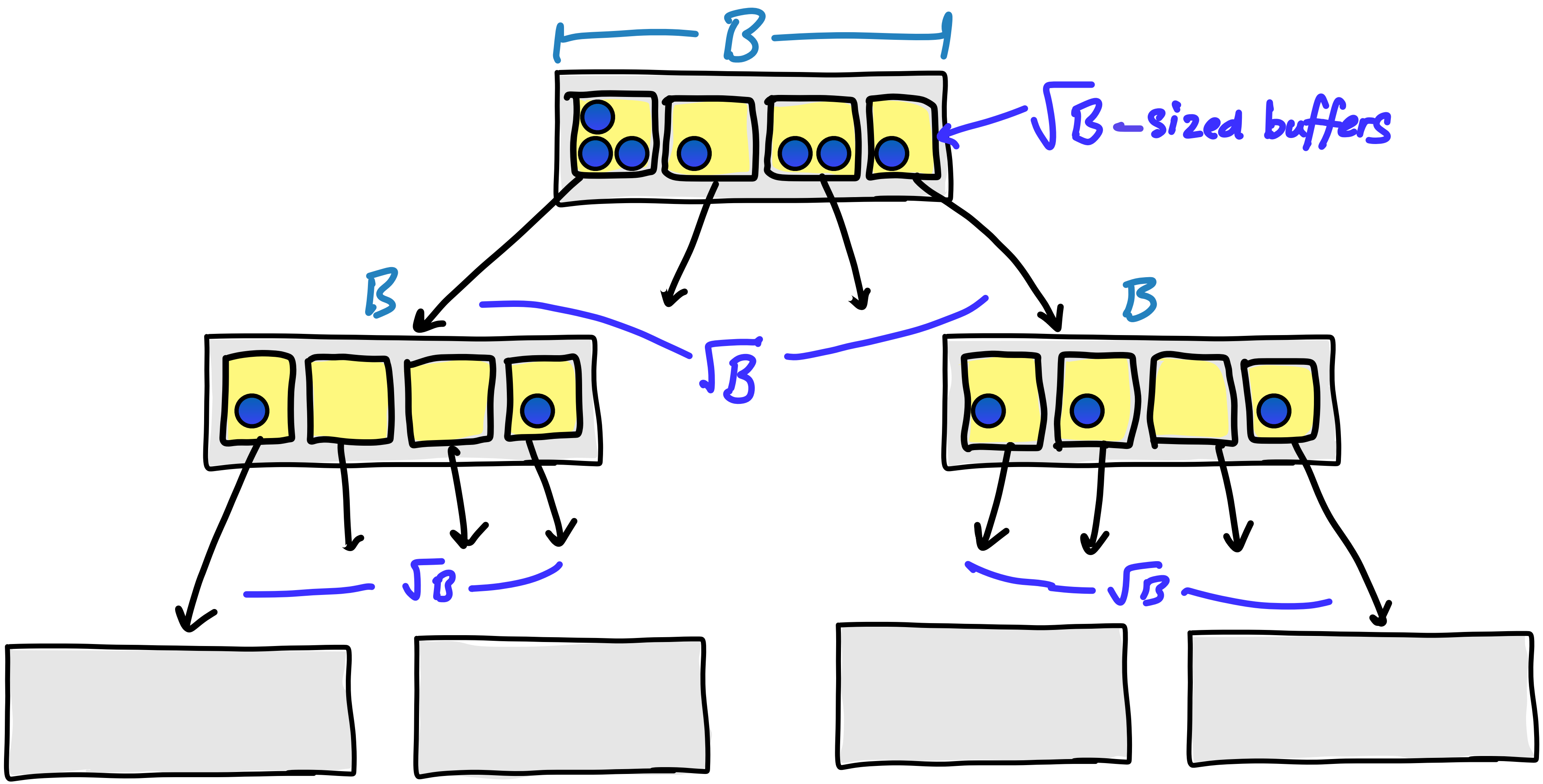
Inserts + deletes:

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



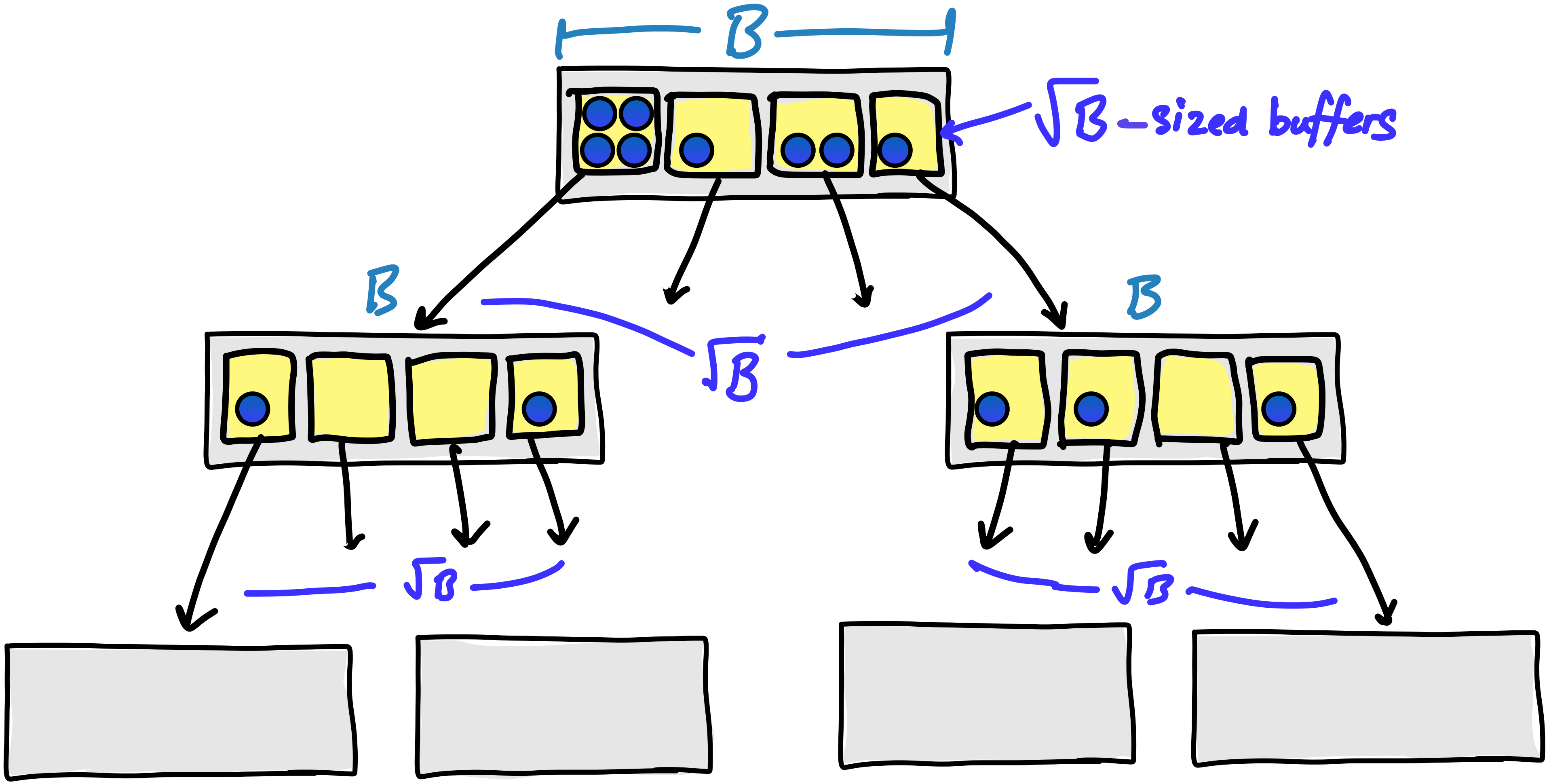
Inserts + deletes:

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



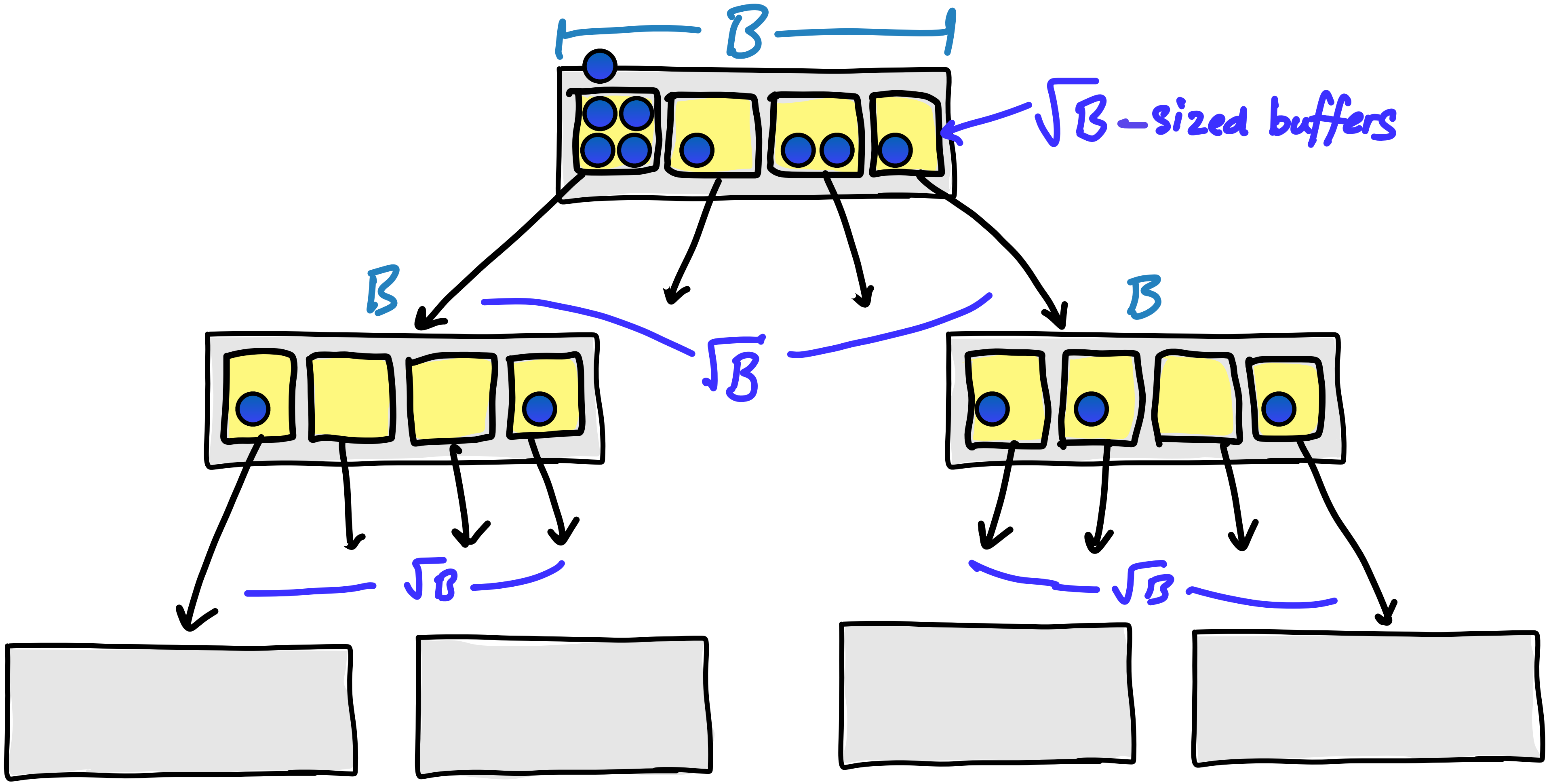
Inserts + deletes:

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



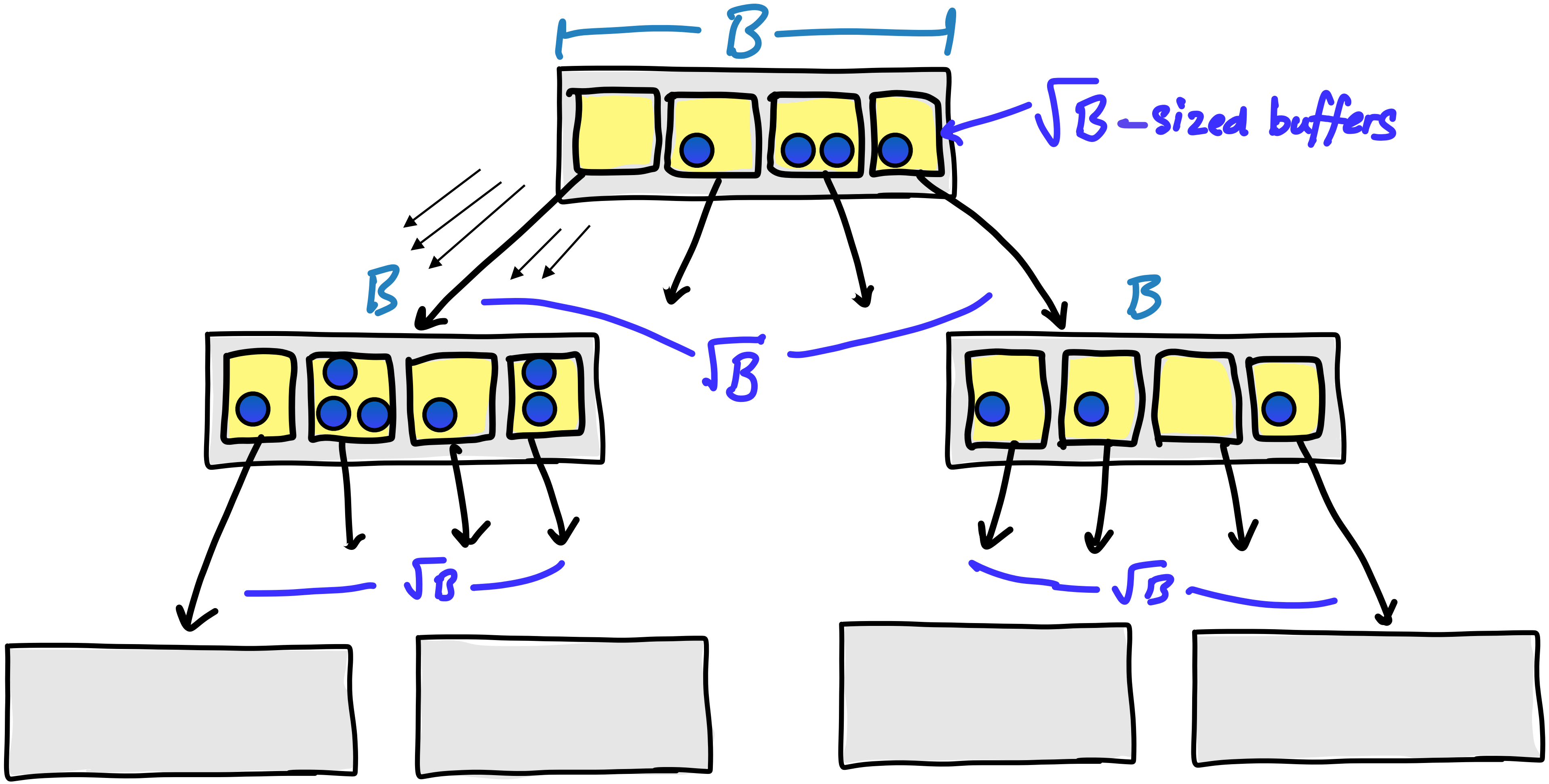
Inserts + deletes:

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



Inserts + deletes:

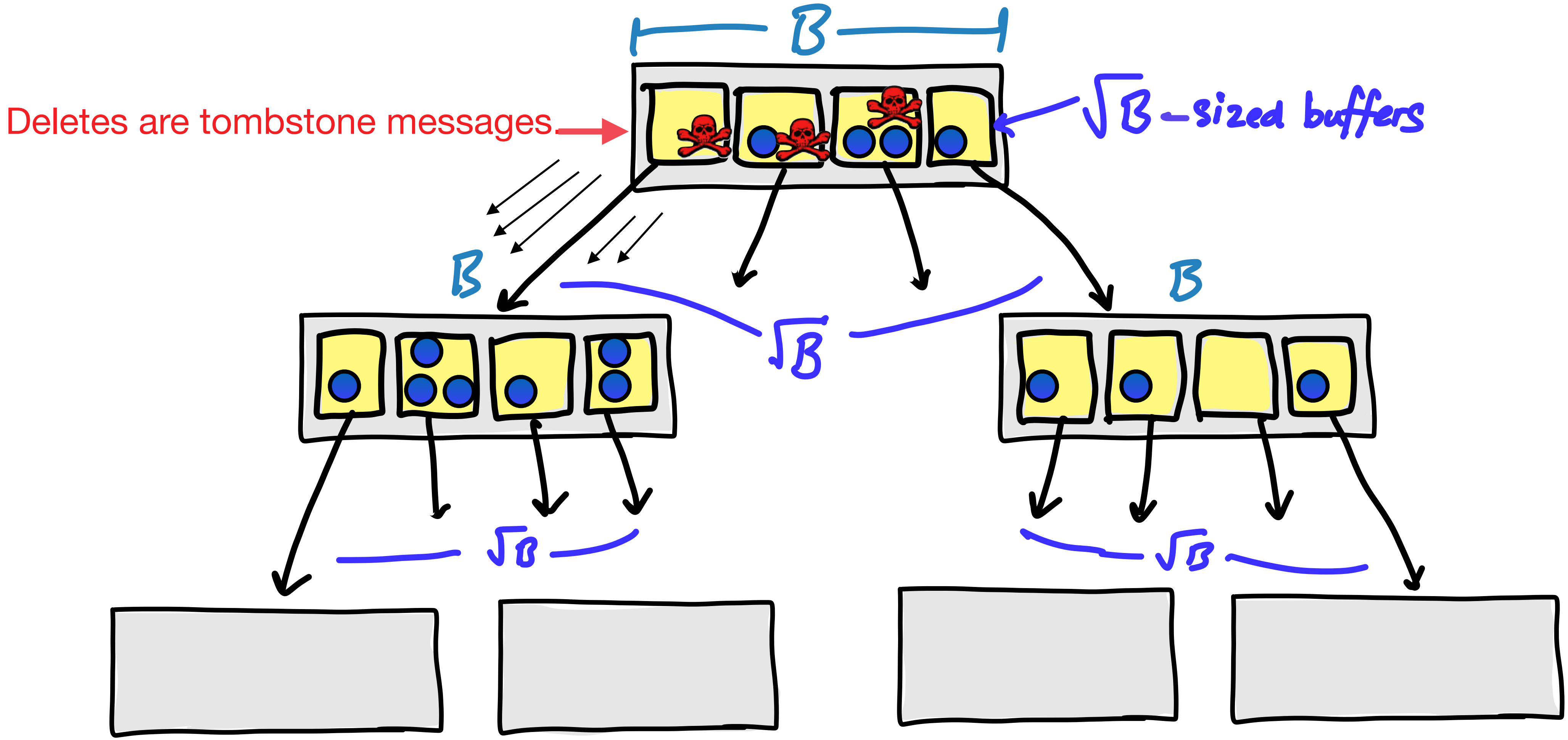
- Send insert/delete messages down from the root and store them in buffers.
- When a buffer fills up, flush.



Inserts and deletes in a B$^{\varepsilon}$ tree

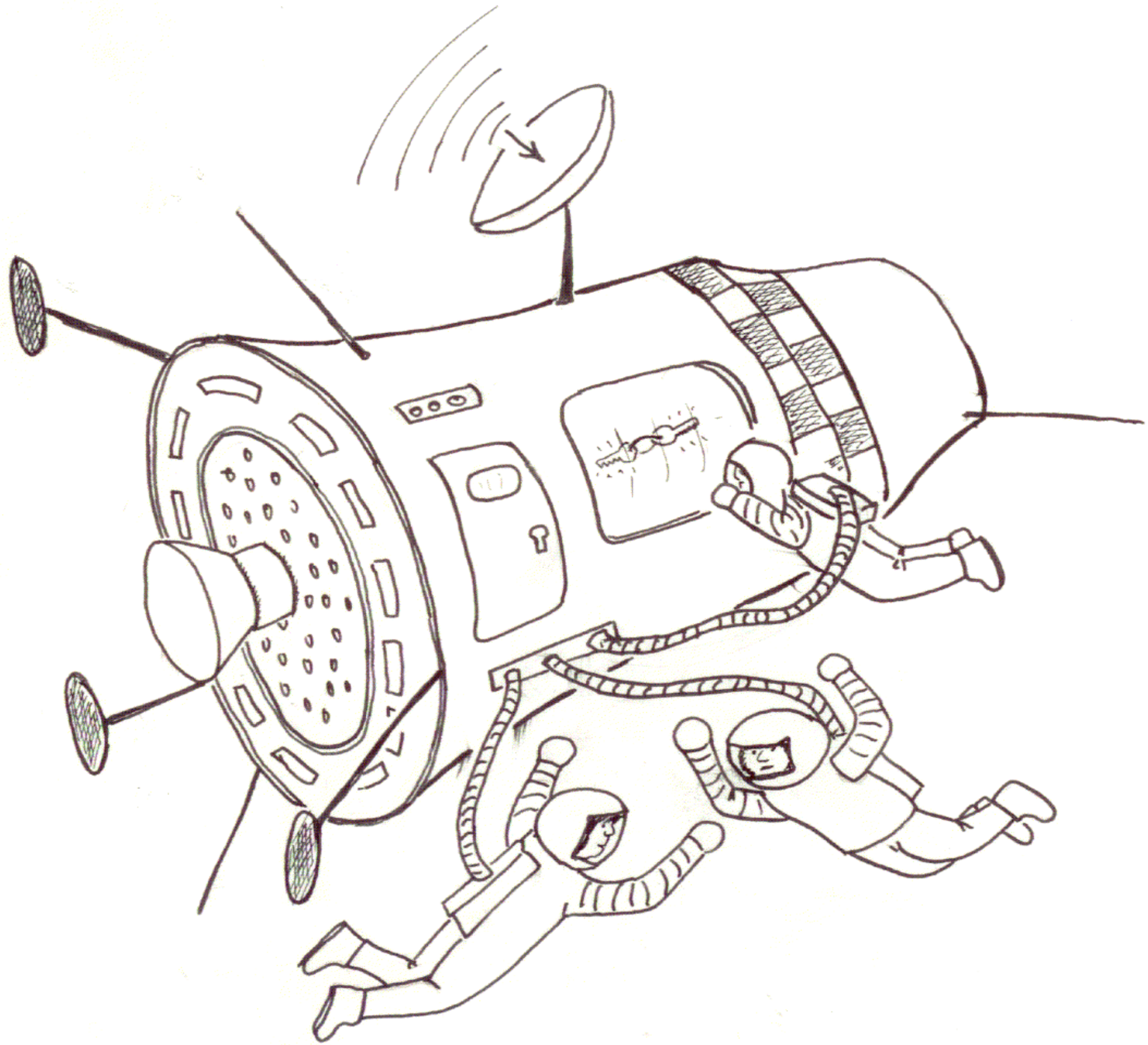
Inserts + deletes:

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

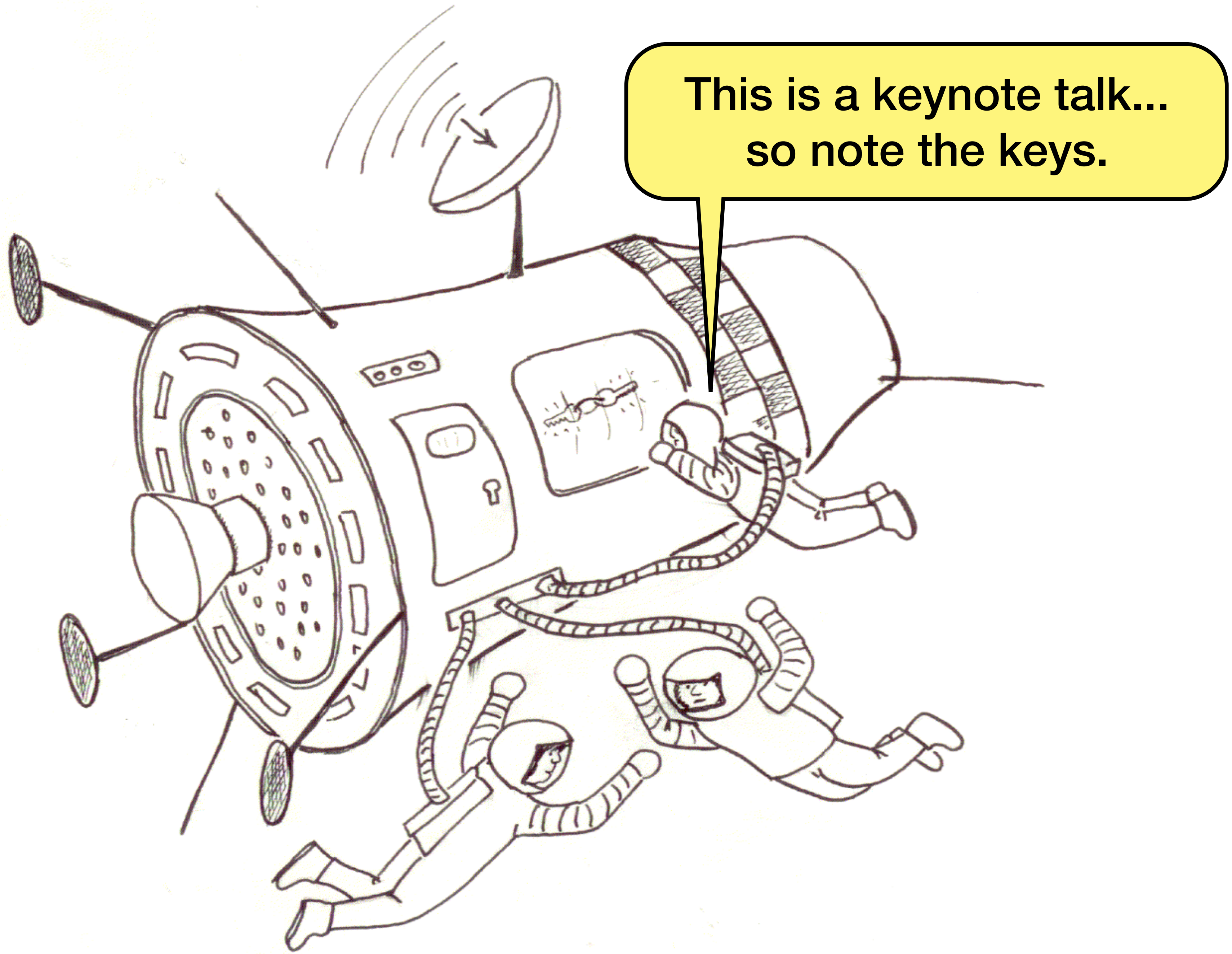


Difficulty of key searches

Difficulty of key searches

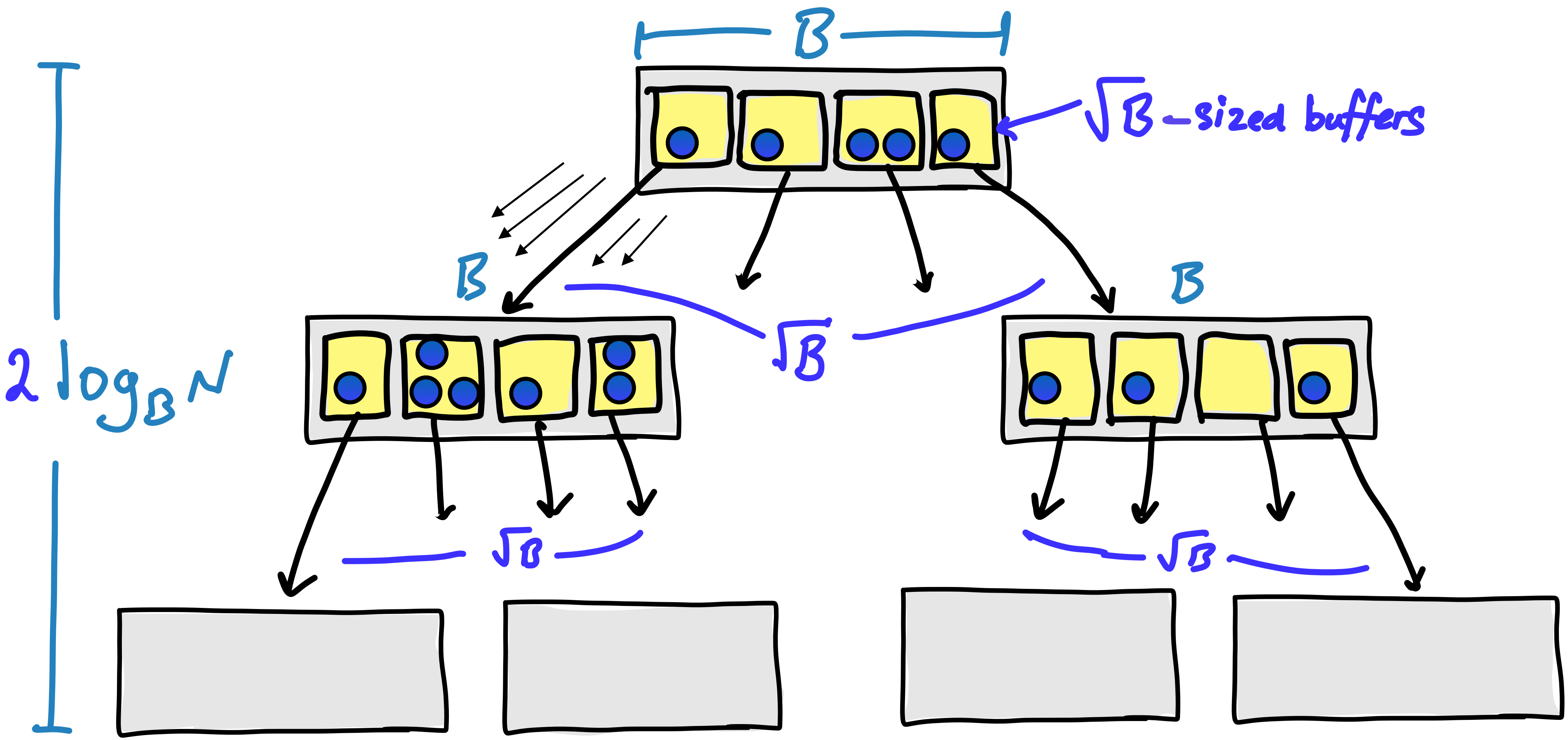


Difficulty of key searches



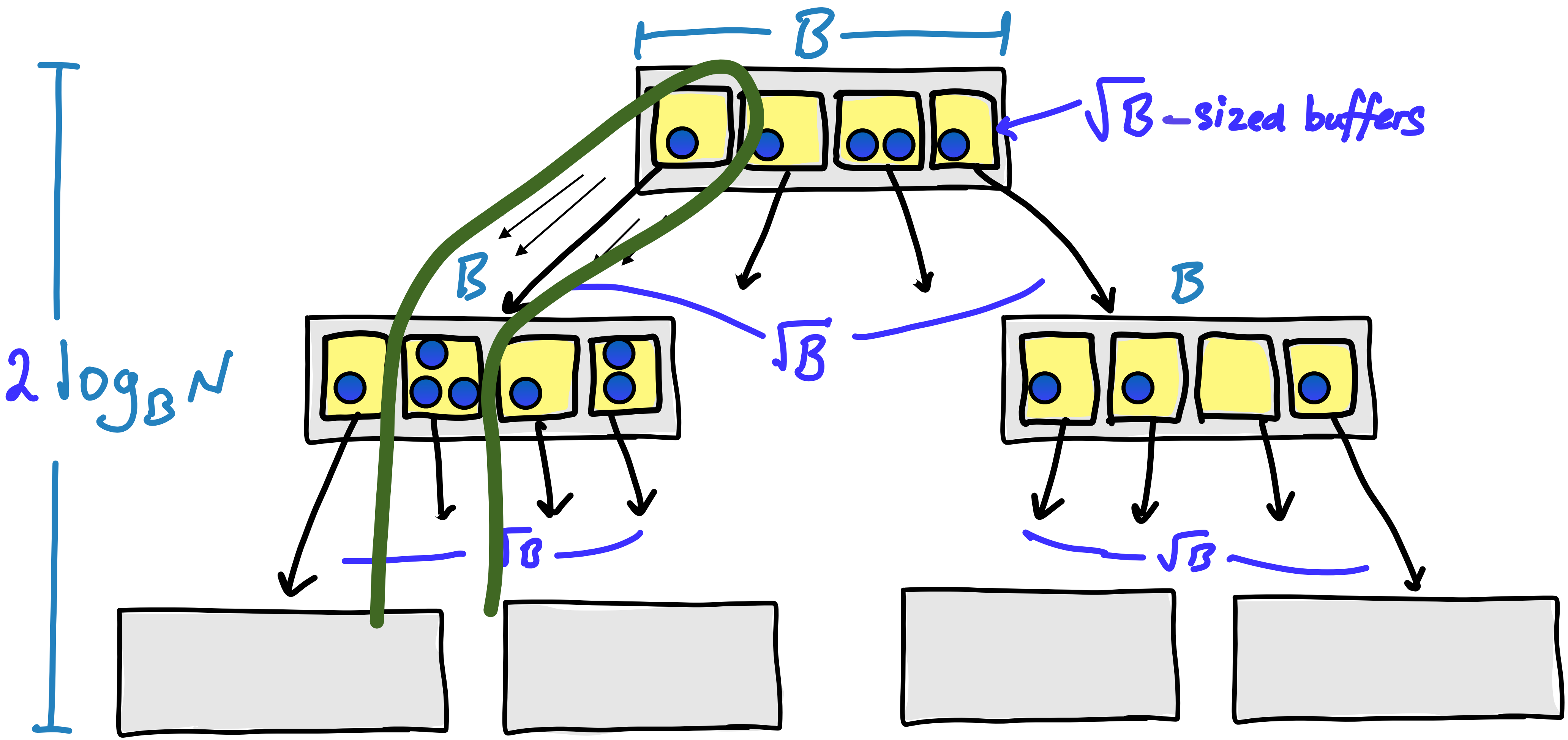
Searches cost $O(\log_B N)$

- Look in all buffers on root-to-leaf path.



Searches cost $O(\log_B N)$

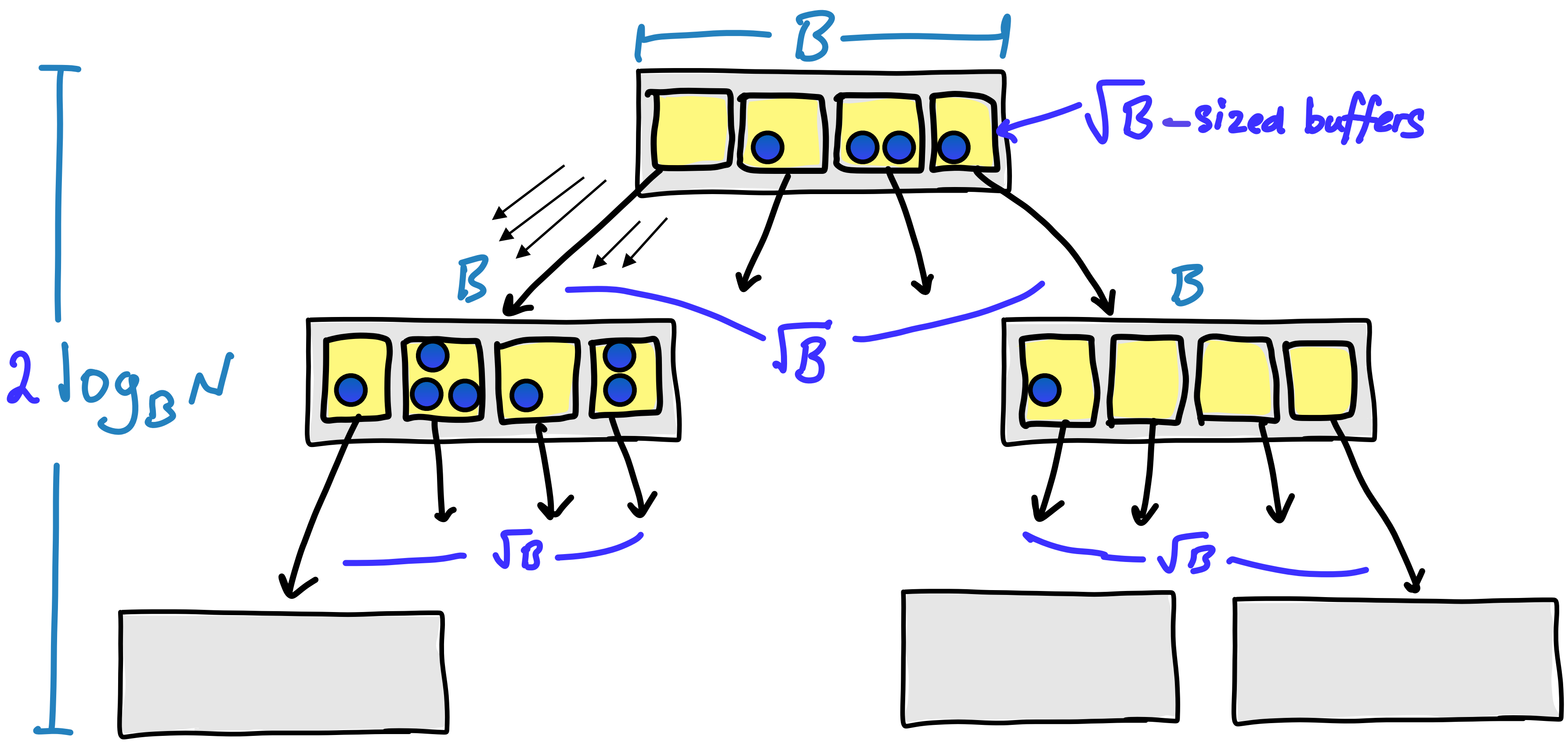
- Look in all buffers on root-to-leaf path.



Insertions analysis in B$^{\varepsilon}$ tree

Inserts cost $O((\log_B N)/\sqrt{B})$ per insert/delete.

- Each flush cost 1 I/O and flushes $\sqrt{B}$ elements.
- Flush cost per element is $1/\sqrt{B}$.
- There are $O(\log_B N)$ levels in a tree.



Write-optimization [Brodal, Fagerberg 03]

	insert	point query
fanout B	$O\left(\log_B N\right)$	$O\left(\log_B N\right)$
fanout $B^{1/2}$	$O\left(\frac{\log_B N}{\sqrt{B}}\right)$	$O\left(\log_B N\right)$

Write-optimization [Brodal, Fagerberg 03]

	insert	**point query**
fanout B	$O\left(\log_B N\right)$	$O\left(\log_B N\right)$
fanout $B^{1/2}$	$O\left(\frac{\log_B N}{\sqrt{B}}\right)$	$O\left(\log_B N\right)$

Example:

Record size: 128 bytes
Node size: 128 KB
B: 1024 records
Speedup: $\approx \frac{\sqrt{1024}}{2} = 16$

Write-optimization [Brodal, Fagerberg 03]

	insert	point query
fanout B	$O(\log_B N)$	$O(\log_B N)$
fanout $B^{1/2}$	$O\left(\frac{\log_B N}{\sqrt{B}}\right)$	$O(\log_B N)$

Example:

Record size: 128 bytes
Node size: 128 KB
B: 1024 records
Speedup: $\approx \frac{\sqrt{1024}}{2} = 16$

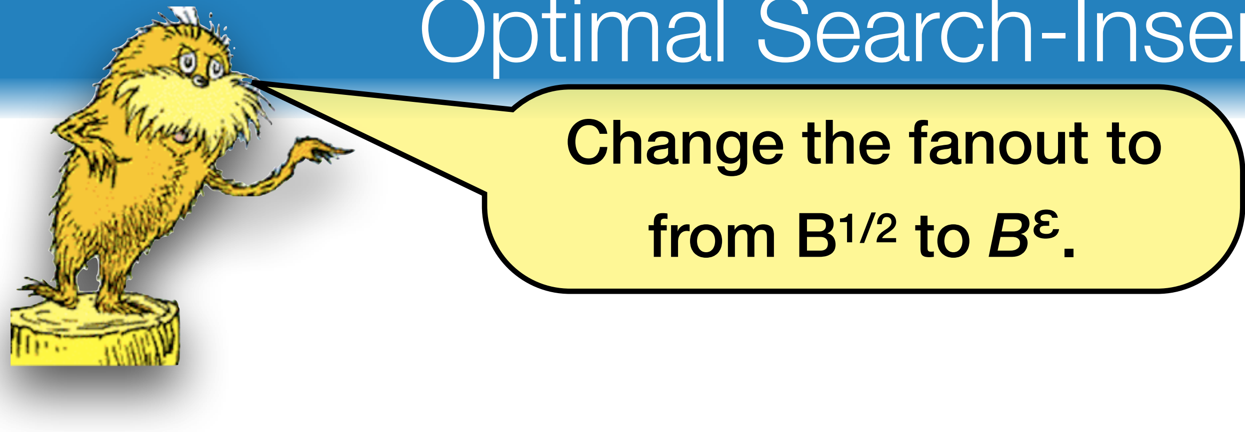
Inserts run 1-2 orders of magnitude faster than in a B-tree.

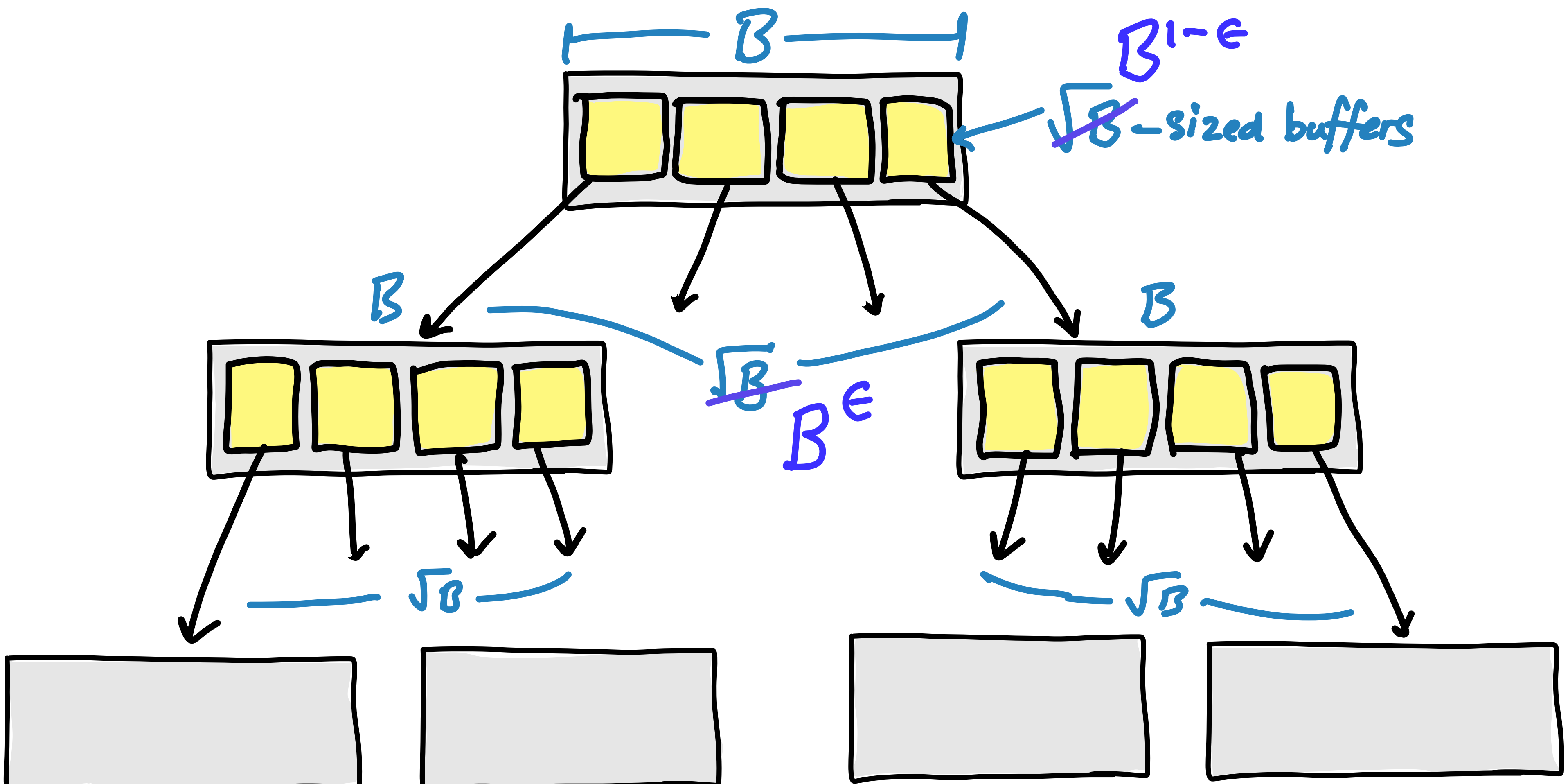
Optimal insertion-search tradeoff curve

[Brodal, Fagerberg 03]

Optimal Search-Insert Tradeoff

[Brodal, Fagerberg 03]





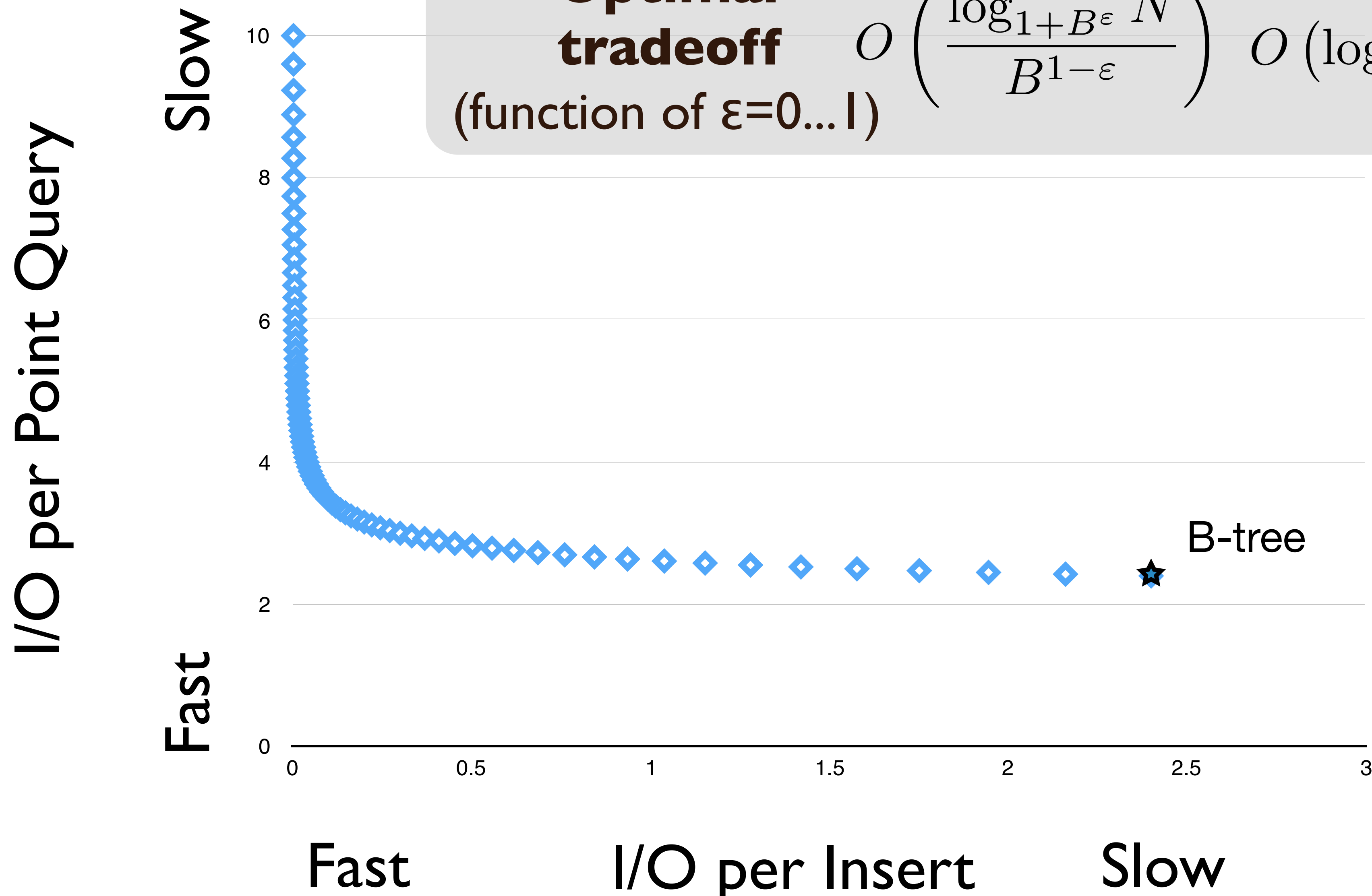
Optimal Search-Insert Tradeoff [Brodal, Fagerberg 03]

	insert	point query
Optimal tradeoff (function of ε=0...1)	$O\left(\frac{\log_{1+B^{\varepsilon}} N}{B^{1-\varepsilon}}\right)$	$O\left(\log_{1+B^{\varepsilon}} N\right)$
B-tree (ε=1)	$O\left(\log_B N\right)$	$O\left(\log_B N\right)$
ε=1/2	$O\left(\frac{\log_B N}{\sqrt{B}}\right)$	$O\left(\log_B N\right)$
ε=0	$O\left(\frac{\log N}{B}\right)$	$O\left(\log N\right)$

10x-100x faster inserts

Illustration of Optimal Tradeoff [Brodal, Fagerberg 03]

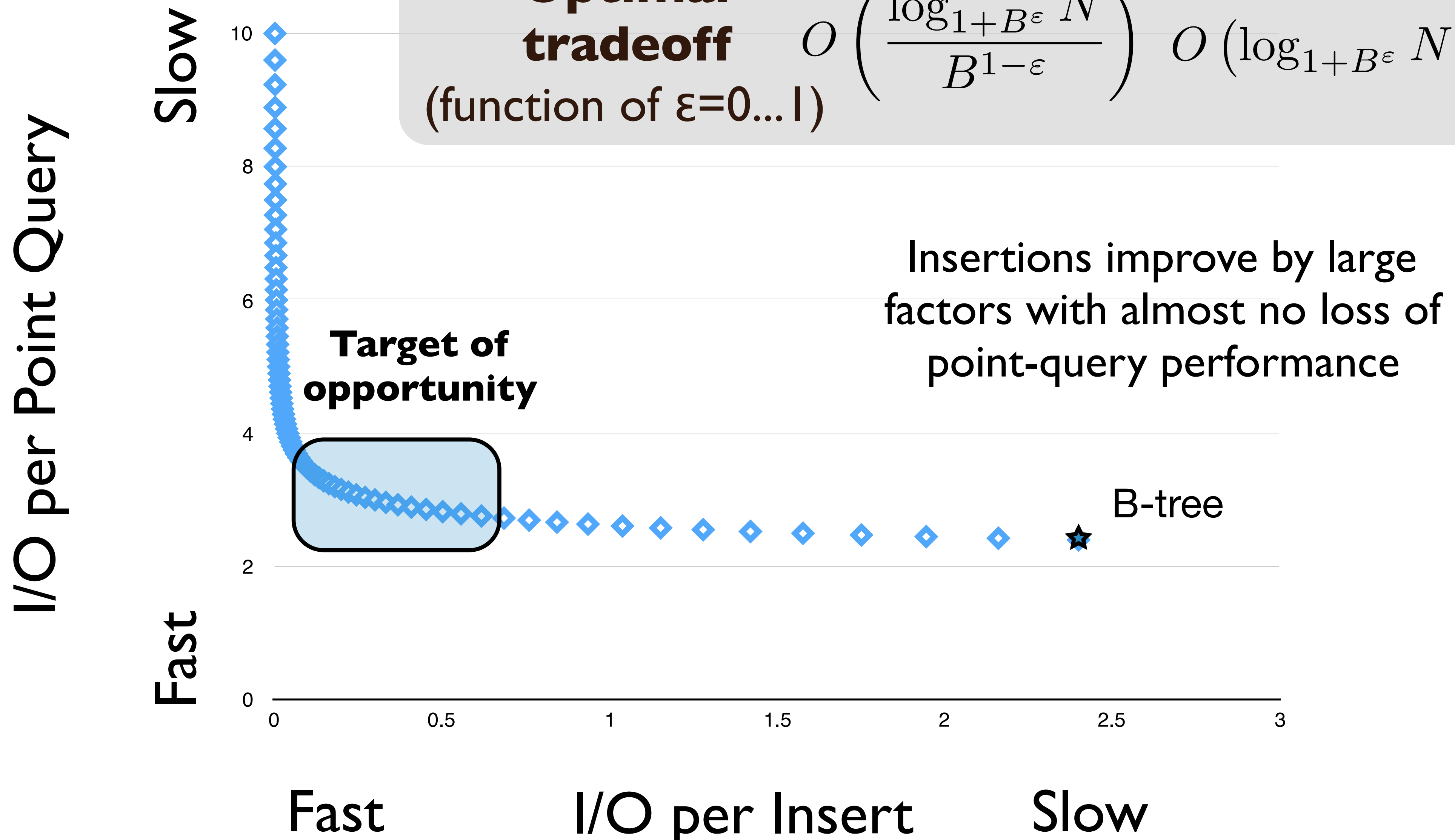
	insert	point query
Optimal tradeoff (function of ε=0...1)	$O\left(\frac{\log_{1+B^\varepsilon} N}{B^{1-\varepsilon}}\right)$	$O\left(\log_{1+B^\varepsilon} N\right)$



$B=10^5$, $N=10^{12}$

Illustration of Optimal Tradeoff [Brodal, Fagerberg 03]

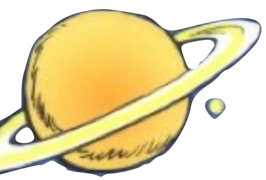
	insert	point query
Optimal tradeoff (function of ε=0...1)	$O\left(\frac{\log_{1+B^\varepsilon} N}{B^{1-\varepsilon}}\right)$	$O\left(\log_{1+B^\varepsilon} N\right)$

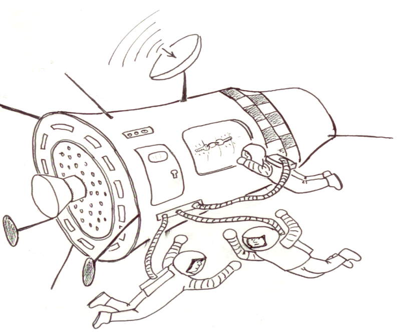


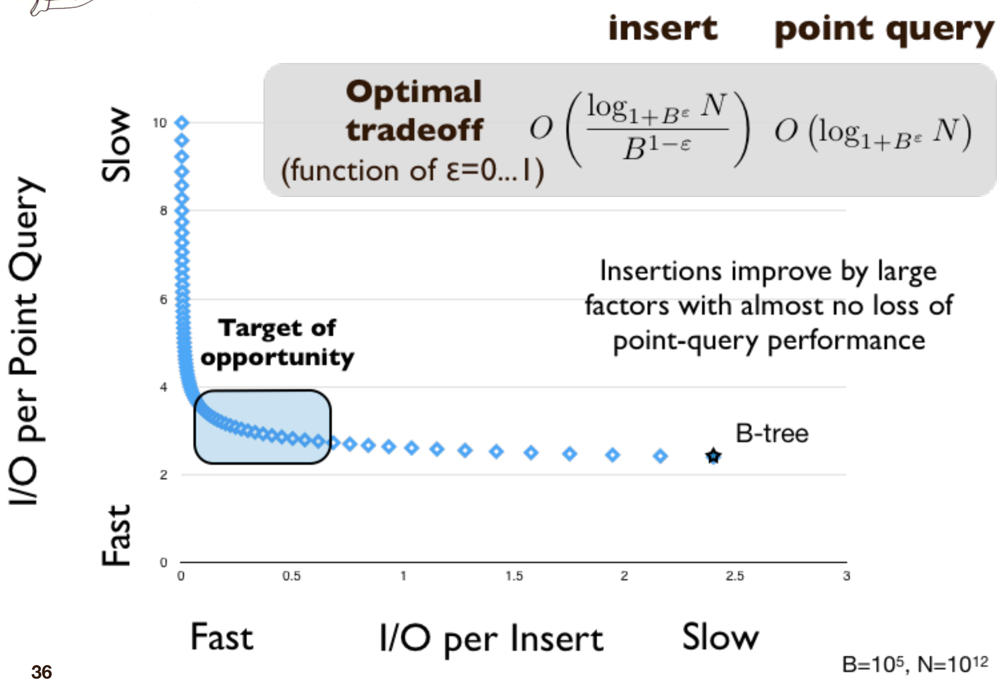
$B=10^5$, $N=10^{12}$

Illustration of Optimal Tradeoff [Brodal, Fagerberg 03]

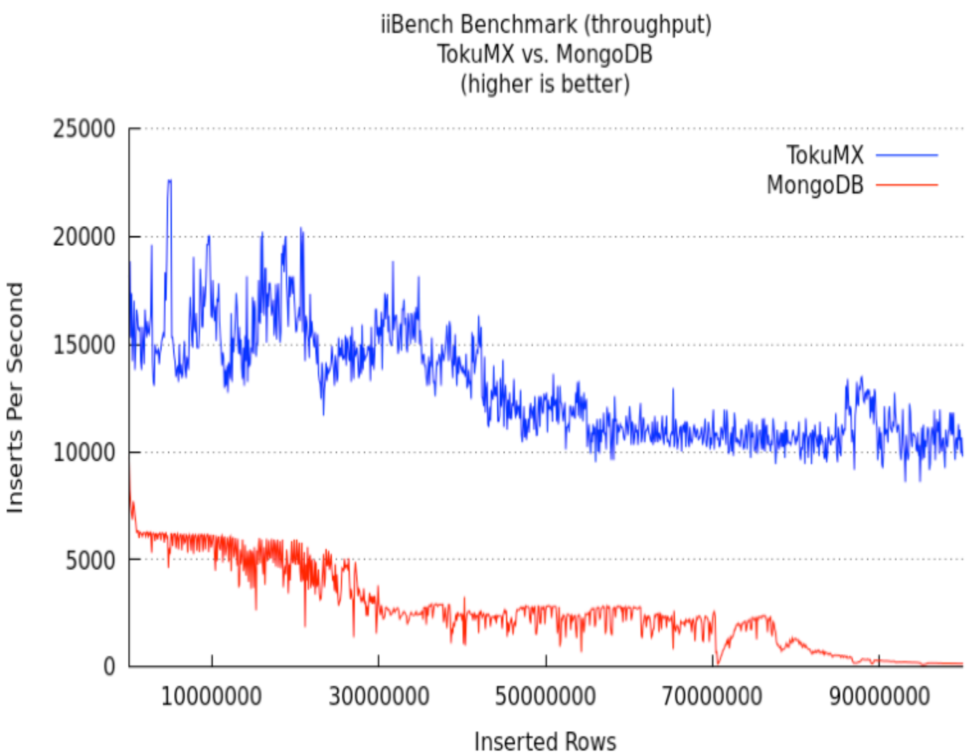
logging

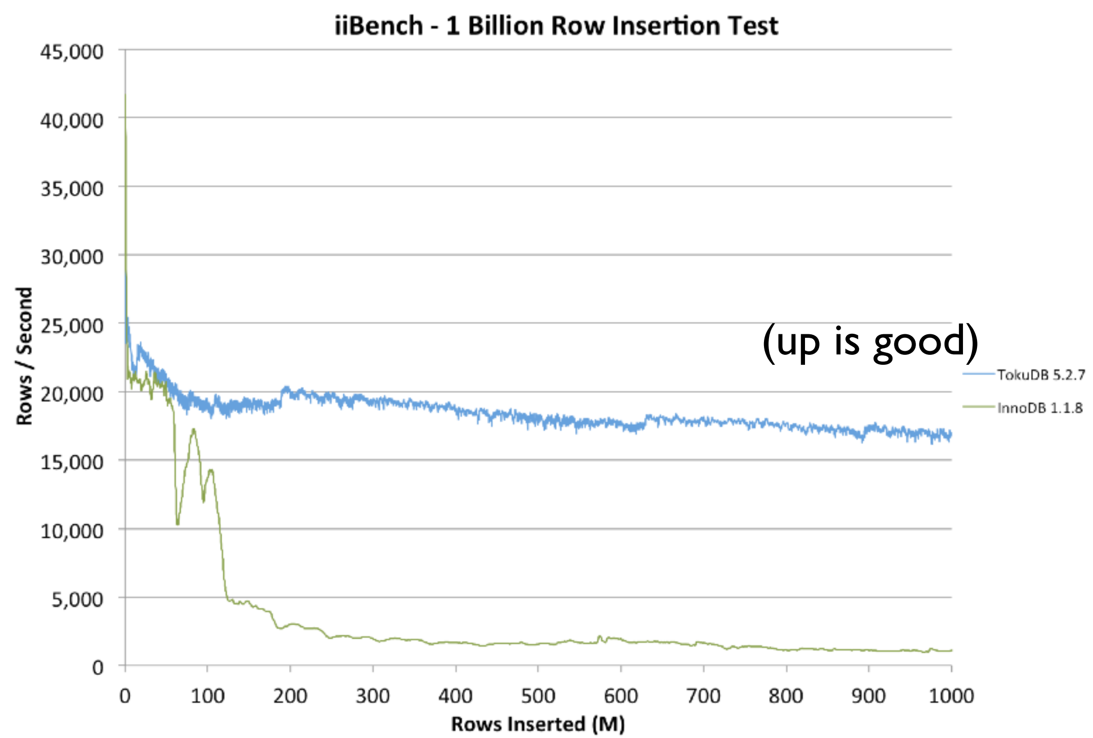






Write performance on large data



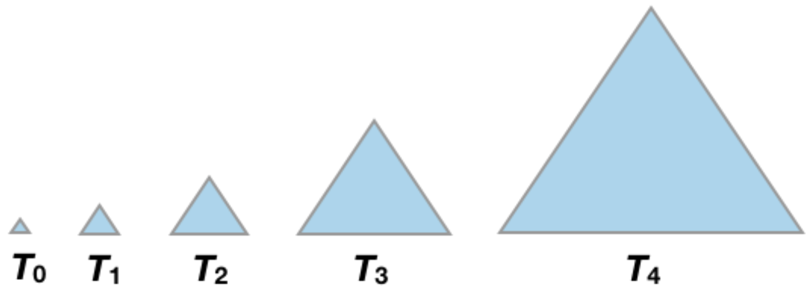


MongoDB **MySQL**

Other WODS

Other write-optimized data structures

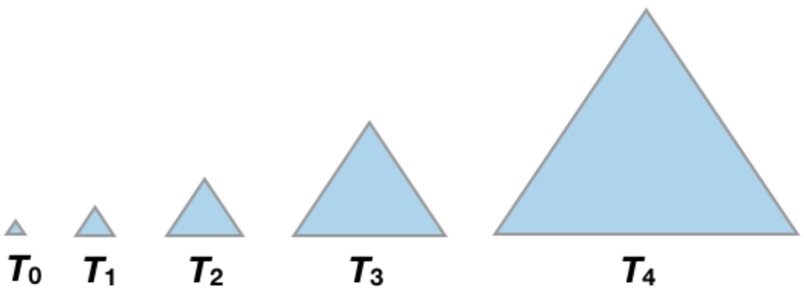
The most famous write-optimized data structure is the log structured merge tree [O'Neil,Cheng, Gawlick, O'Neil 96]



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: BetrFS, BigTable, Cassandra, H-Base, LevelDB, PebblesDB, RocksDB, TokuDB, TableFS, TokuMX.

Other write-optimized data structures

The most famous write-optimized data structure is the log structured merge tree [O'Neil,Cheng, Gawlick, O'Neil 96]

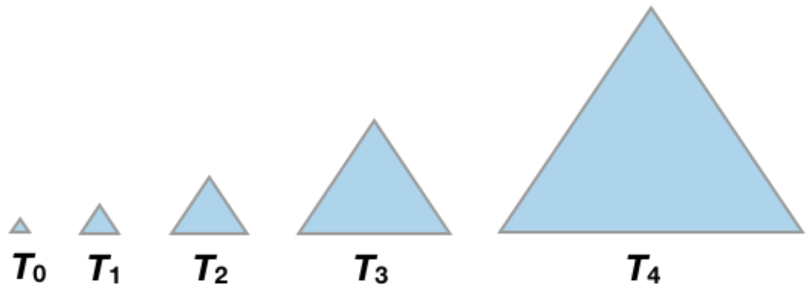


There are many others (B$^{\varepsilon}$-tree, buffered repository tree, COLA, x-dict, write-optimized skip list).

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: BetrFS, BigTable, Cassandra, H-Base, LevelDB, PebblesDB, RocksDB, TokuDB, TableFS, TokuMX.

Other write-optimized data structures

The most famous write-optimized data structure is the log structured merge tree [O'Neil,Cheng, Gawlick, O'Neil 96]



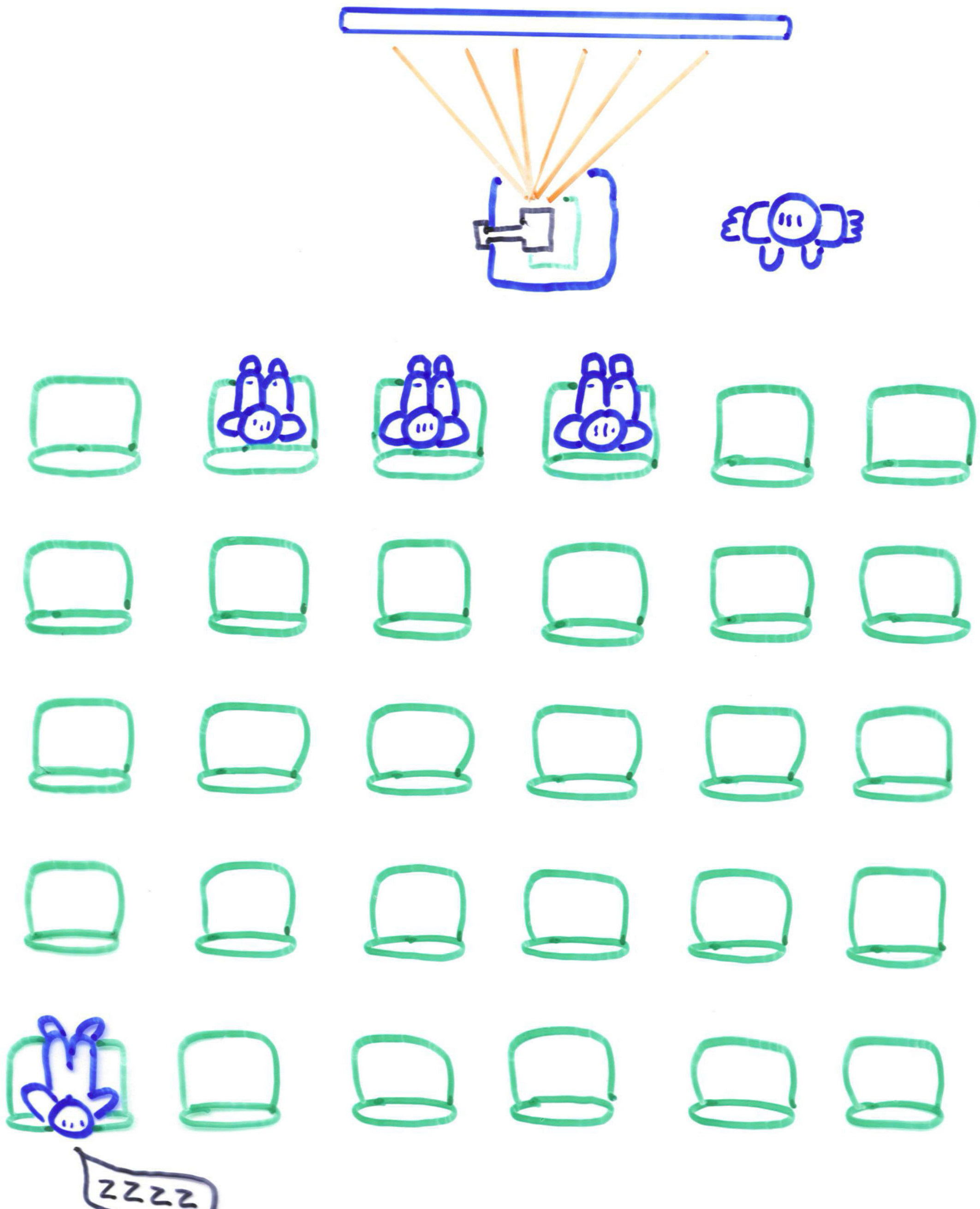
There are many others (B^{ε}-tree, buffered repository tree, COLA, x-dict, write-optimized skip list).

Write optimization is having a large impact on systems.

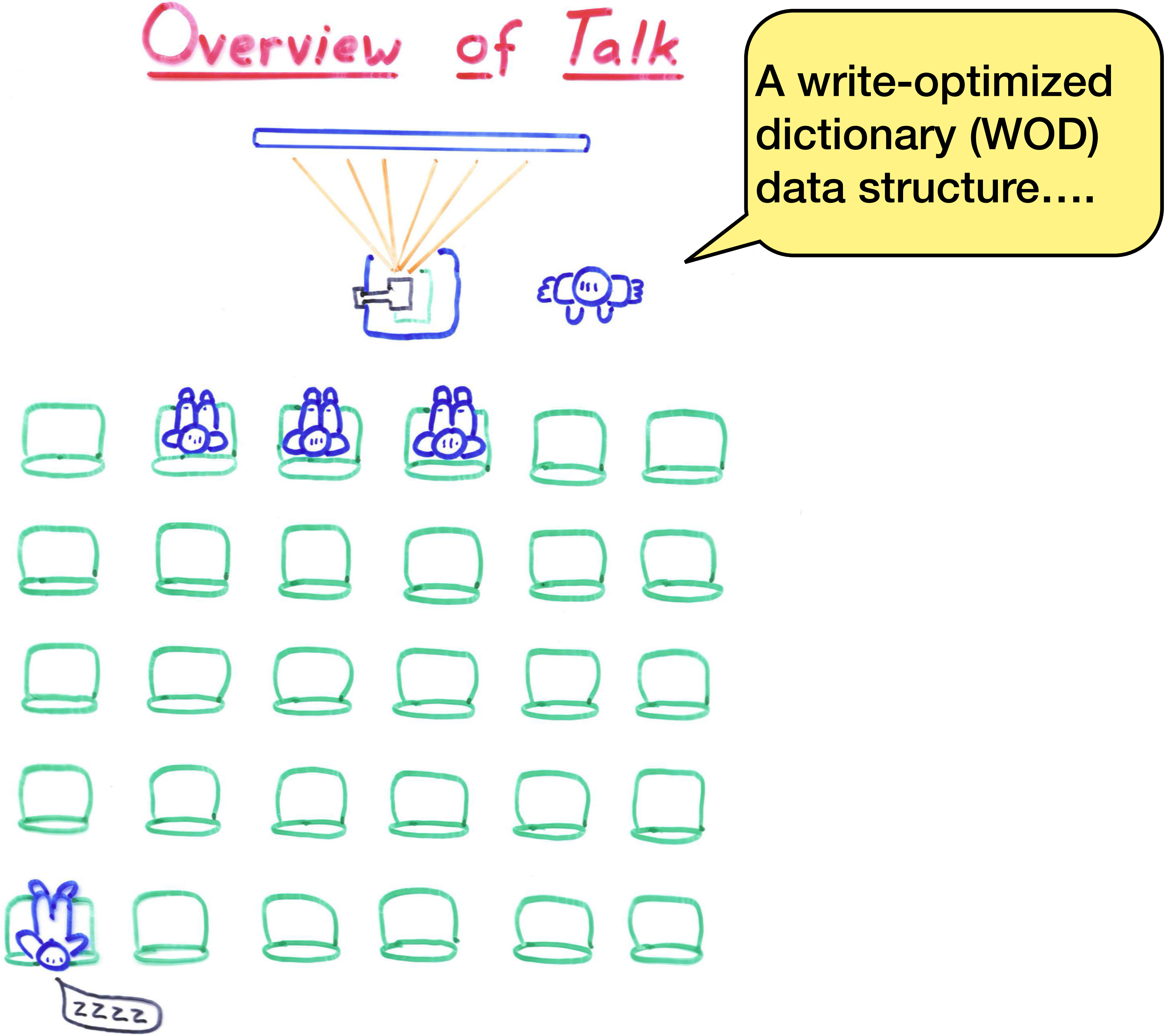
Data structures: [O'Neil,Cheng, Gawlick, O'Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach,Fineman,Fogel, Kuszmaul, Nelson'07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10], [Spillane, Shetty, Zadok, Archak, Dixit 11]. [
Systems: BetrFS, BigTable, Cassandra, H-Base, LevelDB, PebblesDB, RocksDB, TokuDB, TableFS, TokuMX.

Overview of Talk

Overview of Talk



Overview of Talk



Overview of Talk

A write-optimized dictionary (WOD) data structure….

… searches like a B-tree…

Overview of Talk

A write-optimized dictionary (WOD) data structure….

… searches like a B-tree…

… but inserts asymptotically faster.

Overview of Talk

A write-optimized dictionary (WOD) data structure….

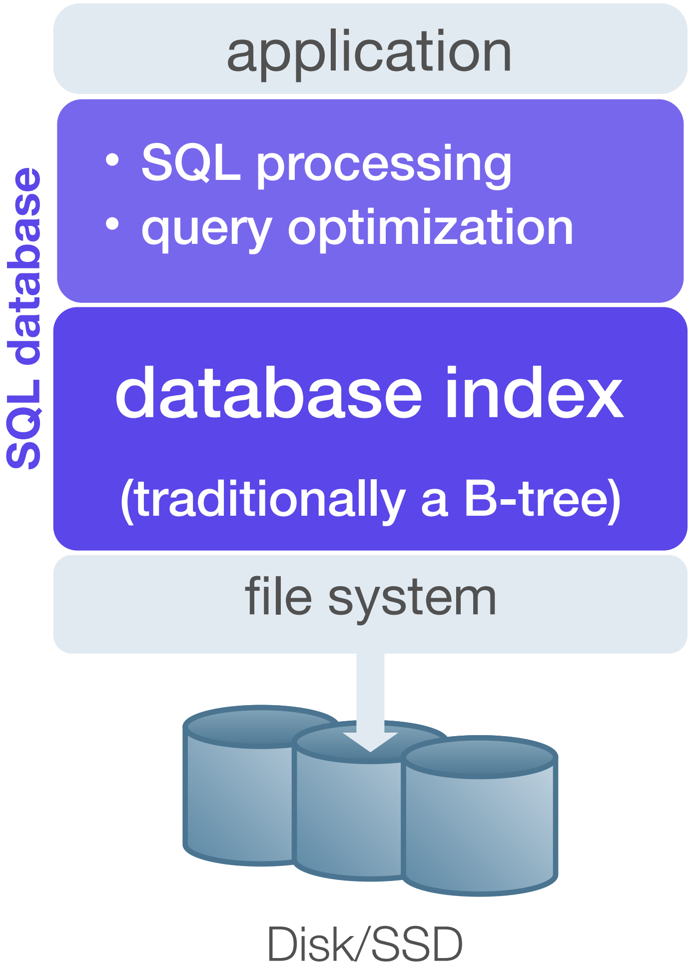
… searches like a B-tree…

… but inserts asymptotically faster.

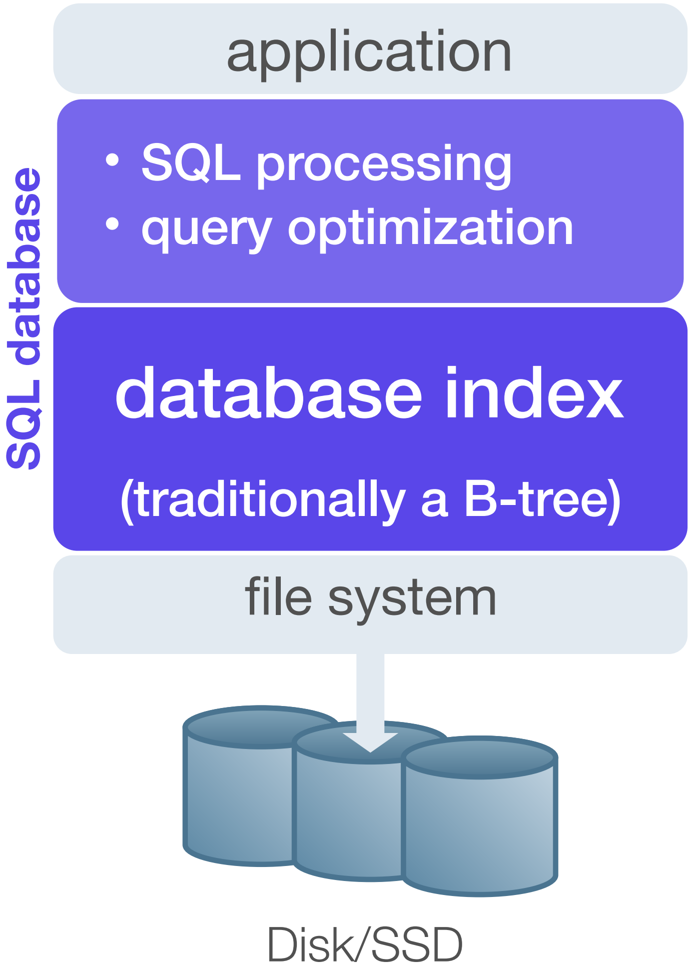
WODs beat the tradeoff from the beginning of the talk.

Write-optimization in databases

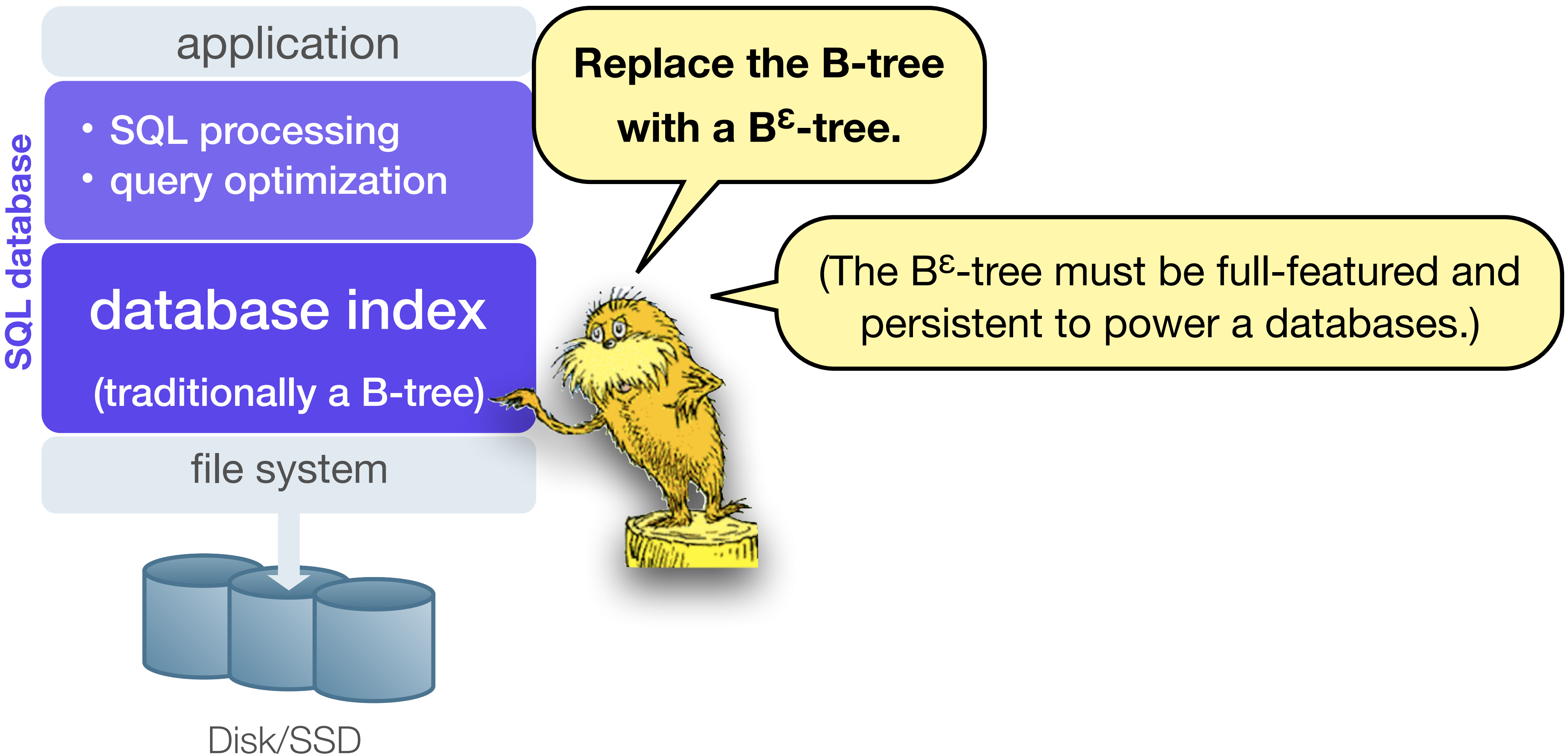
ACID-compliant database



ACID-compliant database built on a B$^{\varepsilon}$-tree

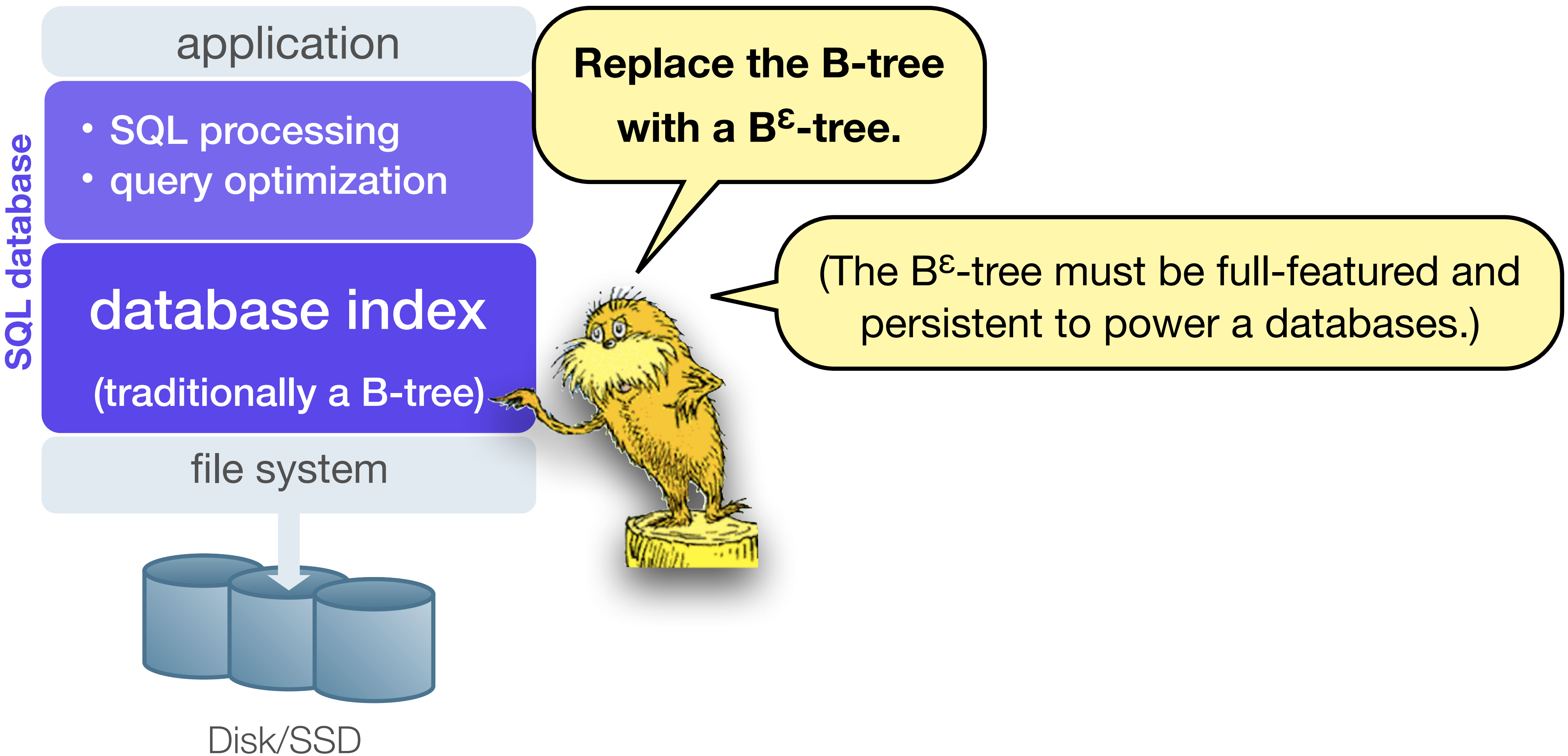


ACID-compliant database built on a B^{ε}-tree



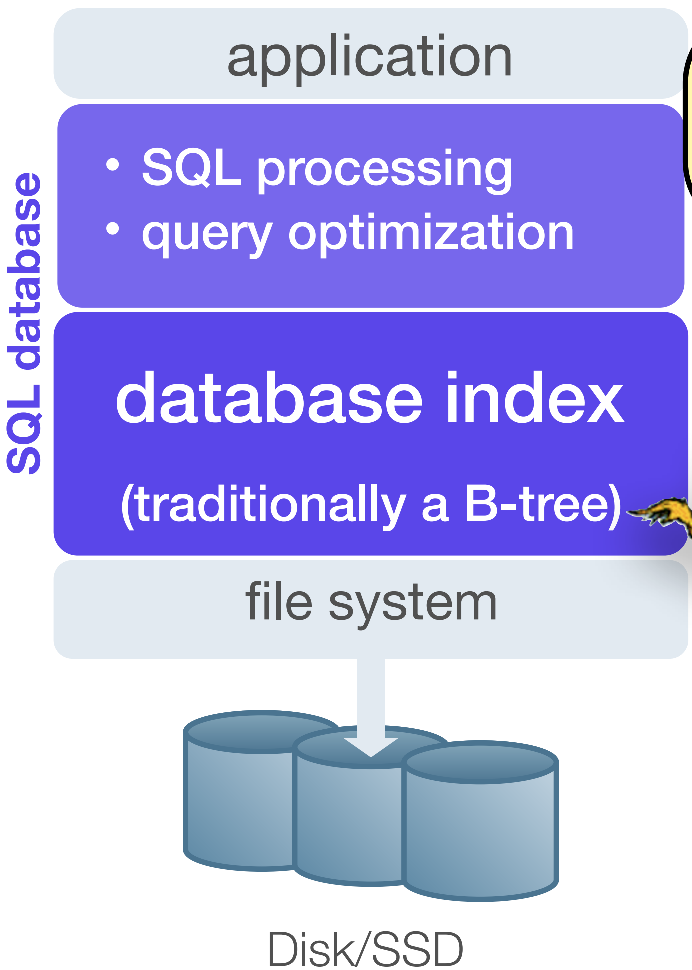
ACID-compliant database built on a B$^{\varepsilon}$-tree

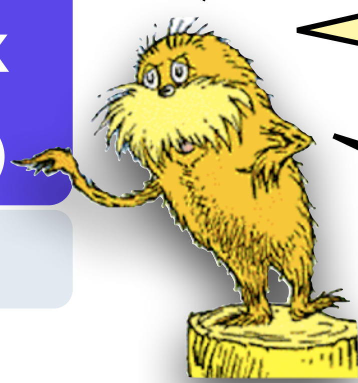
We built a write-optimized SQL databases at our DB company Tokutek.



ACID-compliant database built on a B^{ε}-tree

We built a write-optimized SQL databases at our DB company Tokutek.





Replace the B-tree with a B^{ε}-tree.

(The B^{ε}-tree must be full-featured and persistent to power a databases.)

Result: indexing runs 10x-100x faster than traditional structures.

Write optimization. ✔ What's missing?

Everything else

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

Write optimization. ✔ What's missing?

Everything else

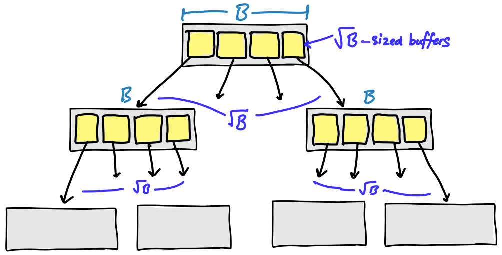
- Variable-sized rows
- Concurrency-control mechanisms
- Multithreading

- Transactions, logging, ACID-compliant crash recovery
- Optimizations for the special cases of sequential inserts and bulk loads
- Compression
- Backup

Transactions, logging, ACID-compliant crash recovery

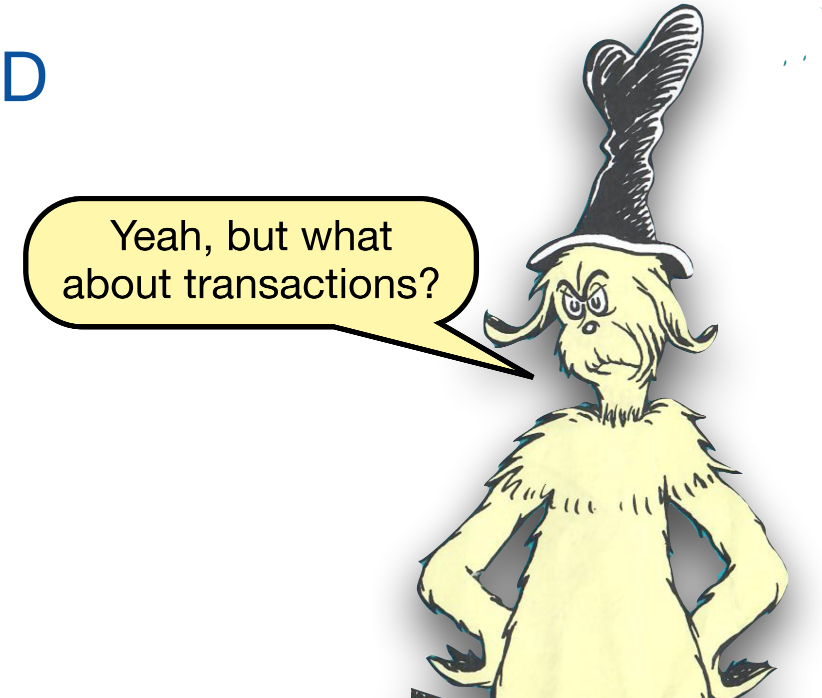
Ingredients

- a regular write-optimized structure



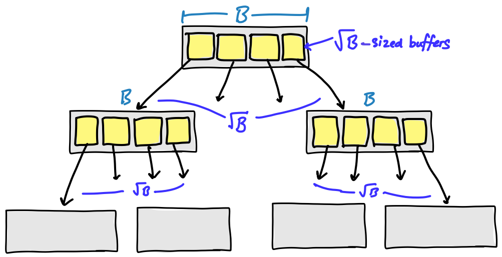
- a log

- periodic checkpoints of the WOD



Ingredients

- a regular write-optimized structure



- a log

sequential access

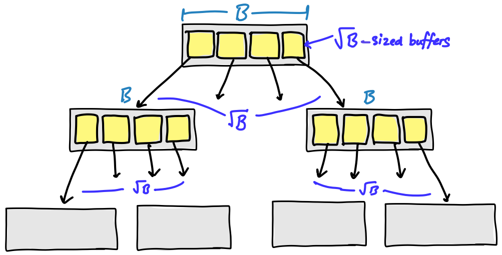
- periodic checkpoints of the WOD

Yeah, but what about transactions?

Transactions, logging, ACID-compliant crash recovery

Ingredients

- a regular write-optimized structure



- a log

sequential access

sequential access…
… if checkpoints are done right

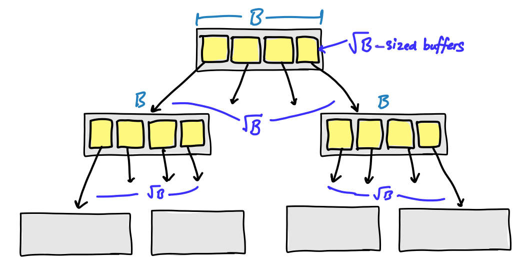
- periodic checkpoints of the WOD

Yeah, but what
about transactions?

Transactions, logging, ACID-compliant crash recovery

Ingredients

- a regular write-optimized structure



- a log 

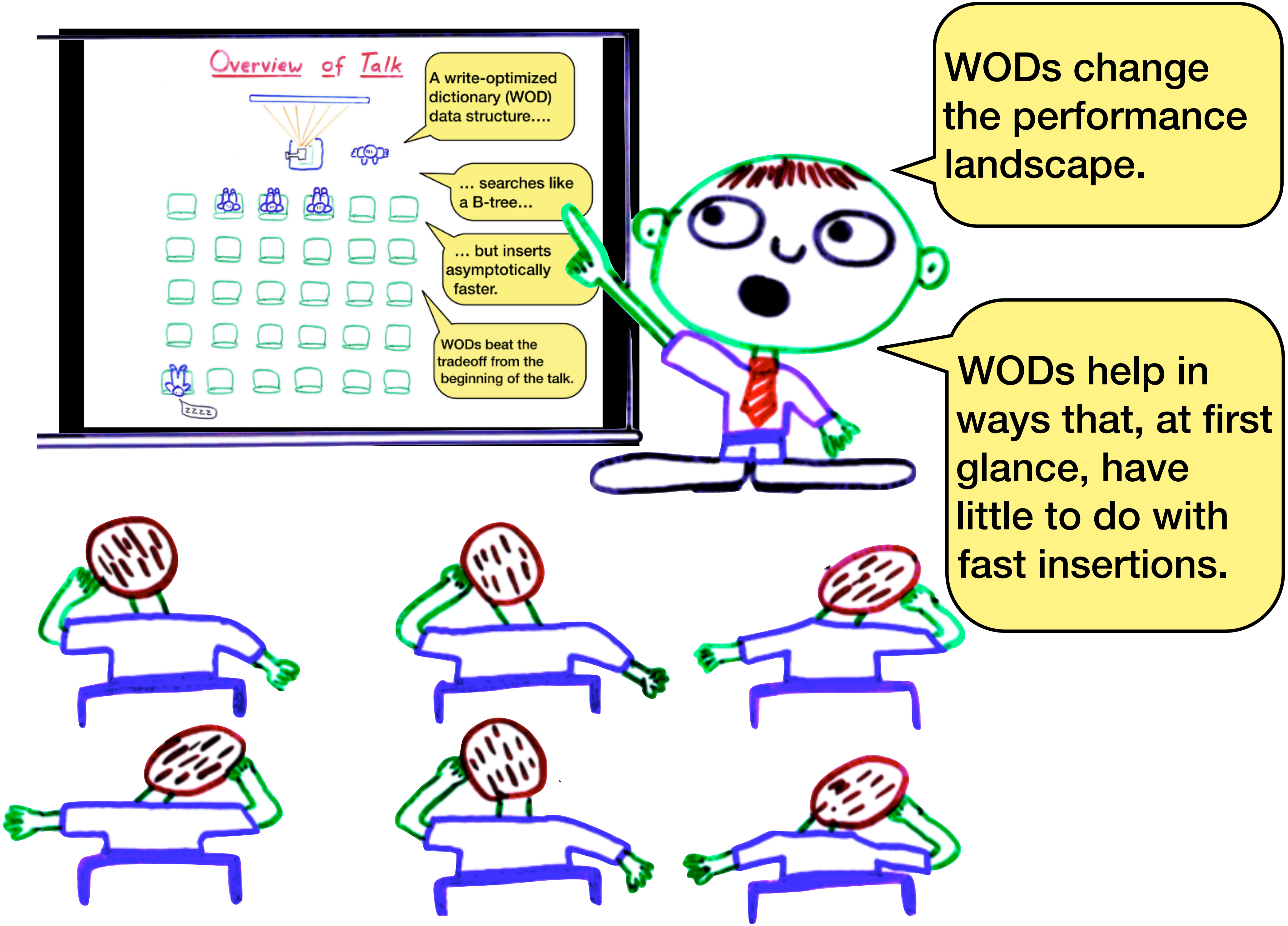
sequential access

sequential access…
… if checkpoints are done right

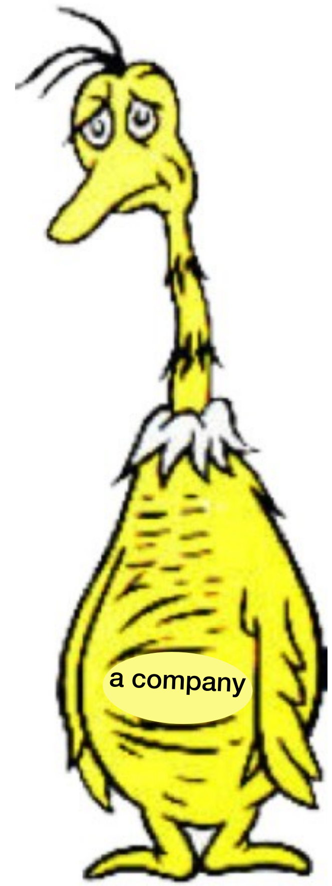
- periodic checkpoints of the WOD

Yeah, but what
about transactions?

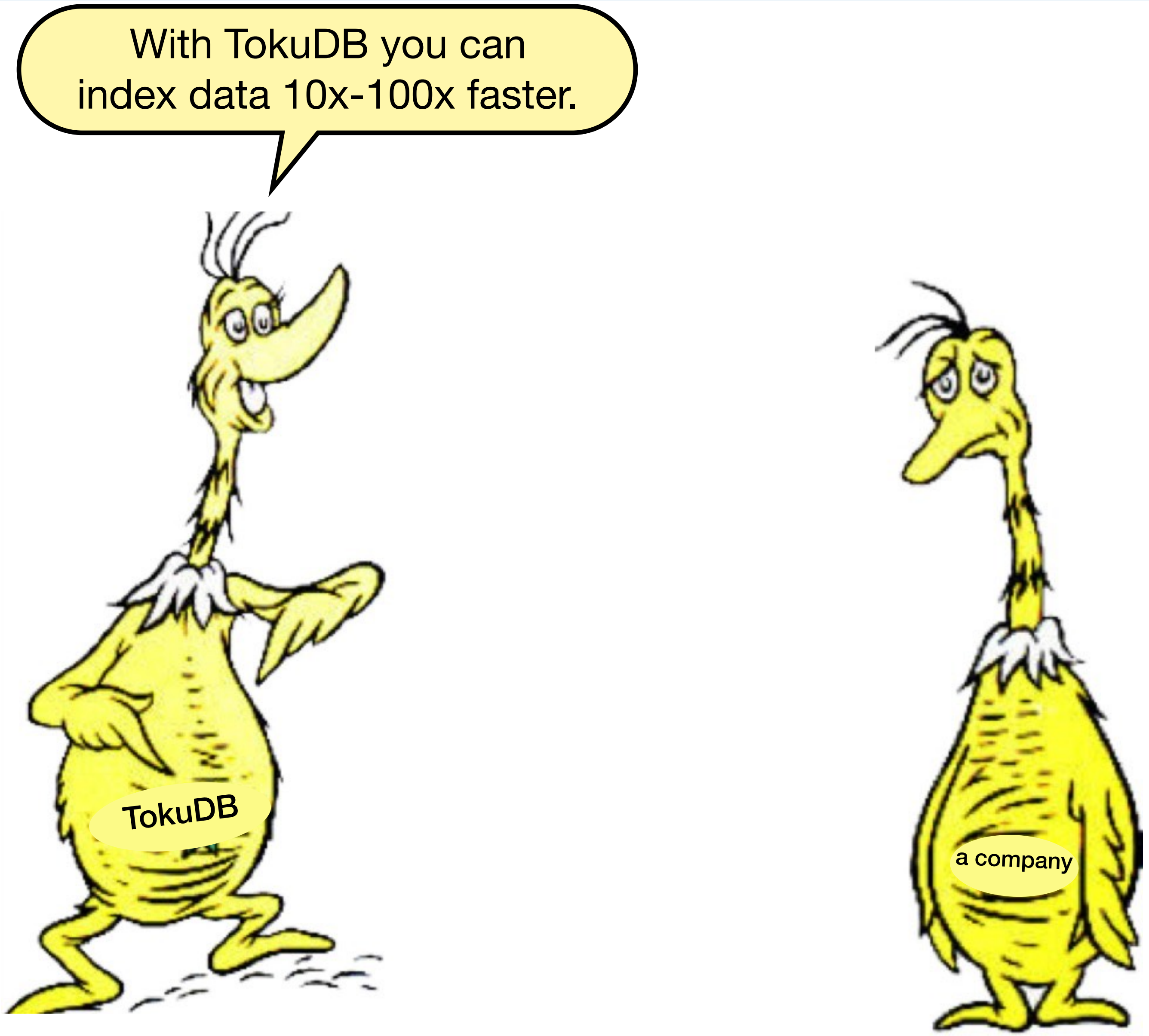
Result: even with crash recovery and transactional semantics, indexing is 10x-100x faster than traditional structures.


Overview of Talk
A write-optimized dictionary (WOD) data structure....
... searches like a B-tree...
... but inserts asymptotically faster.
WODs beat the tradeoff from the beginning of the talk.
zzzz
WODs change the performance landscape.
WODs help in ways that, at first glance, have little to do with fast insertions.

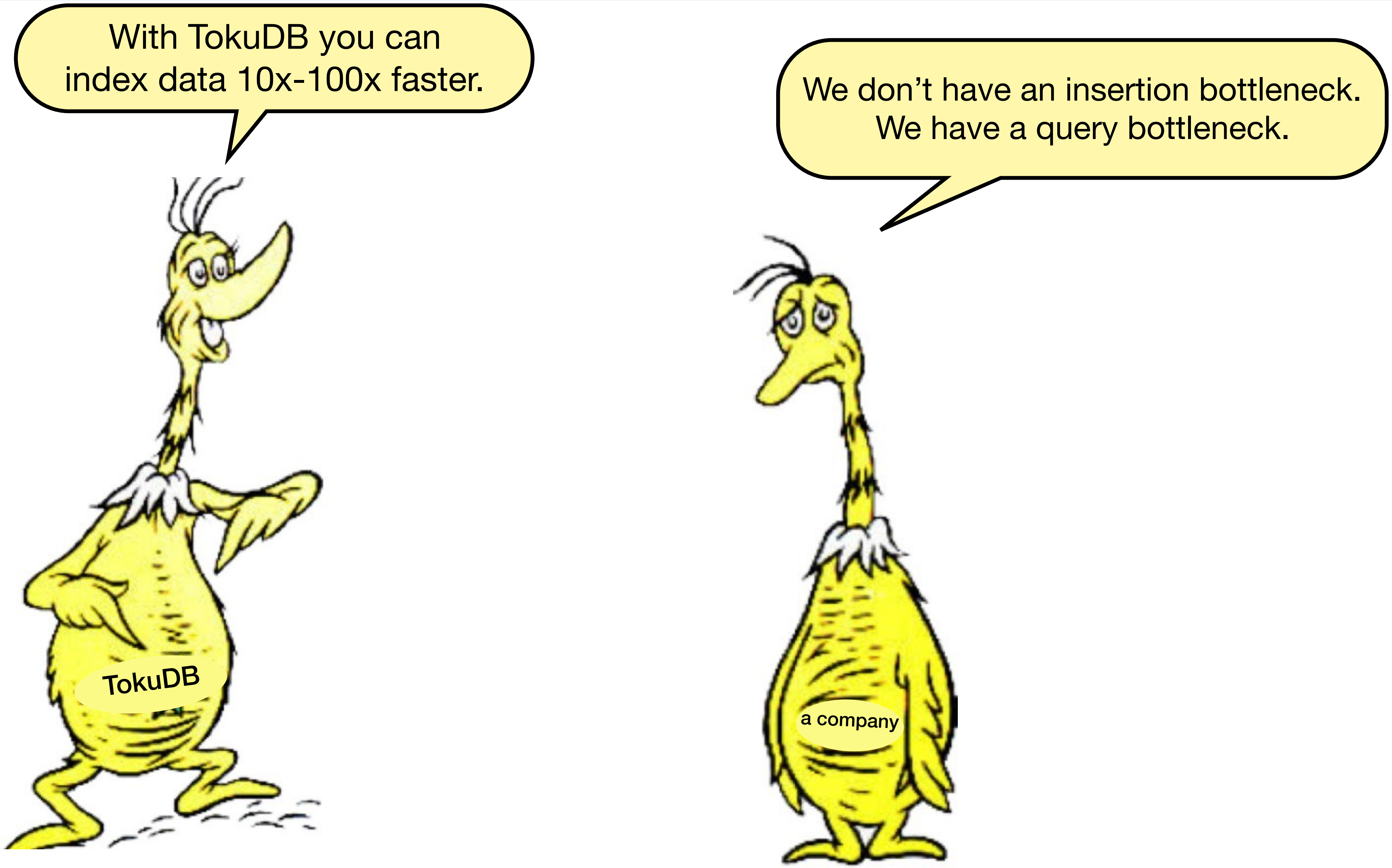
Remember the logging/indexing dilemma?



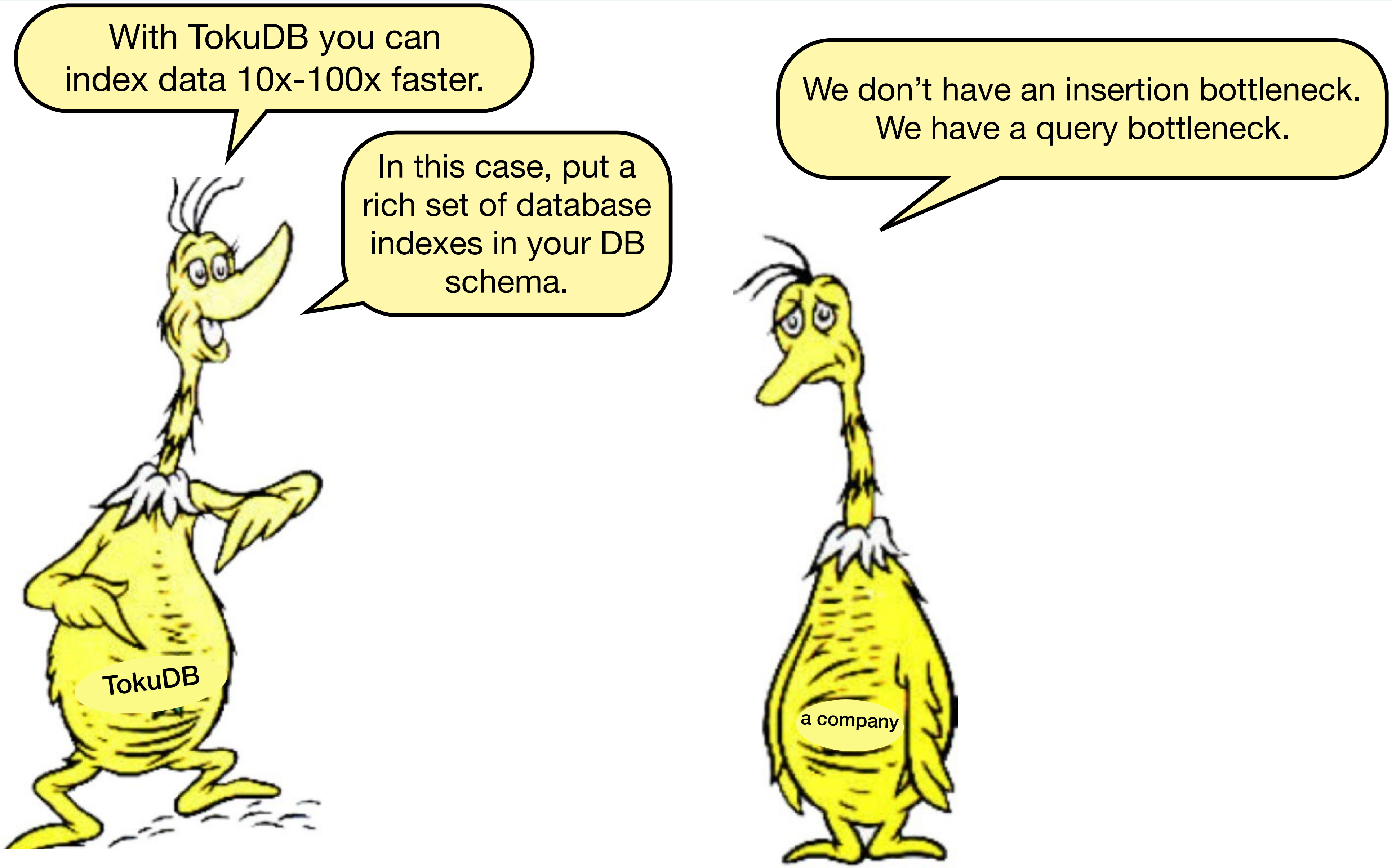
Remember the logging/indexing dilemma?



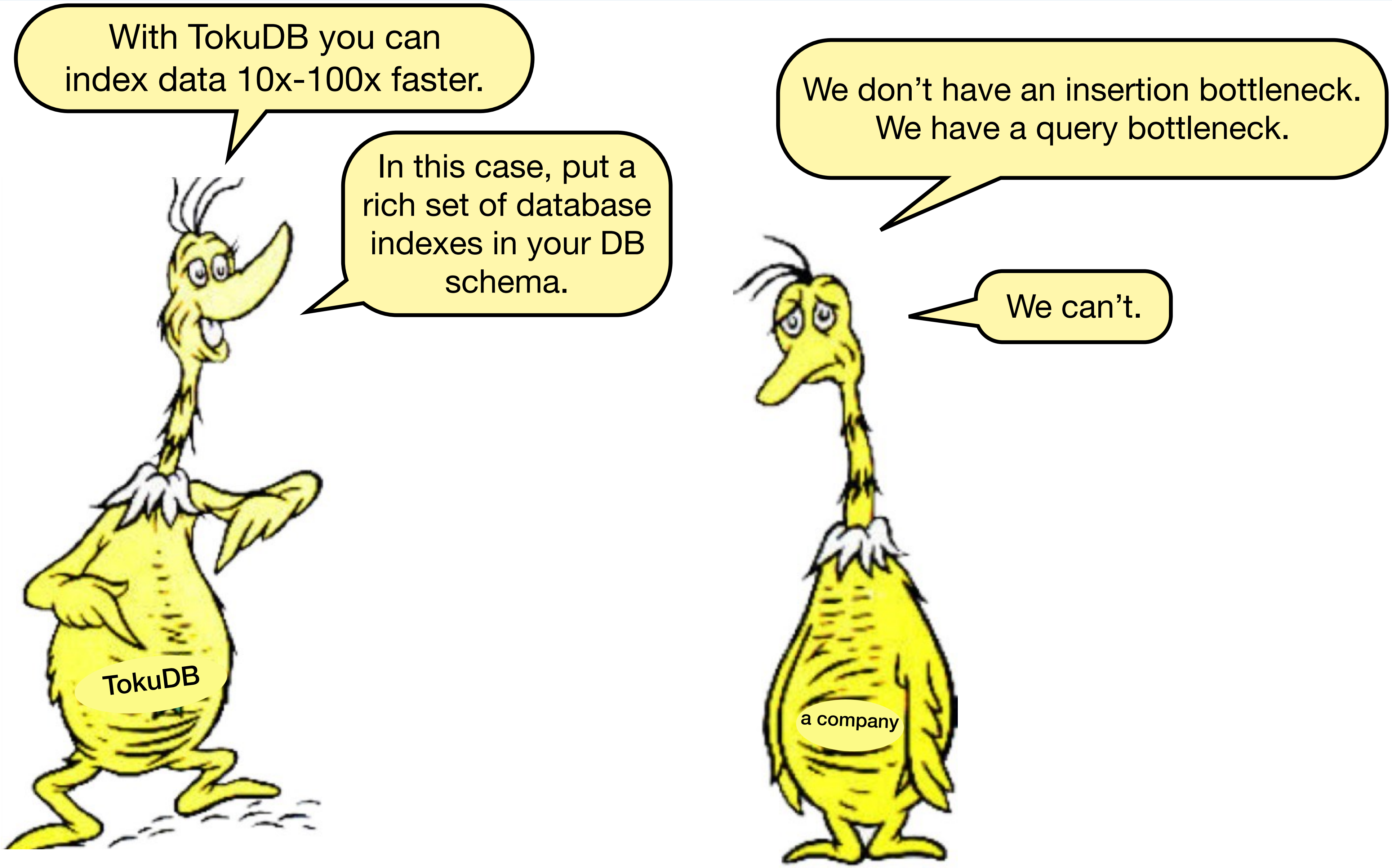
Remember the logging/indexing dilemma?



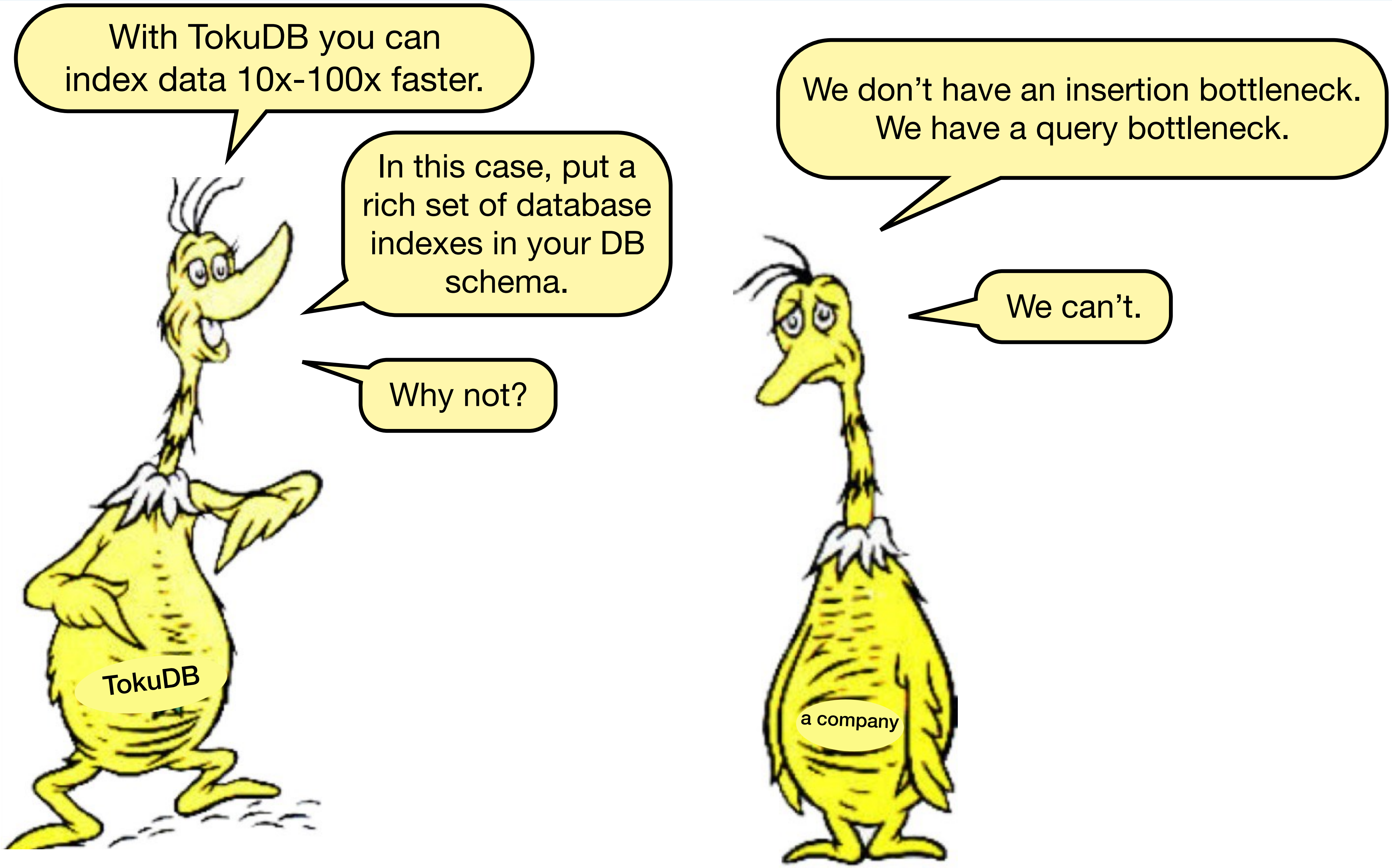
Remember the logging/indexing dilemma?



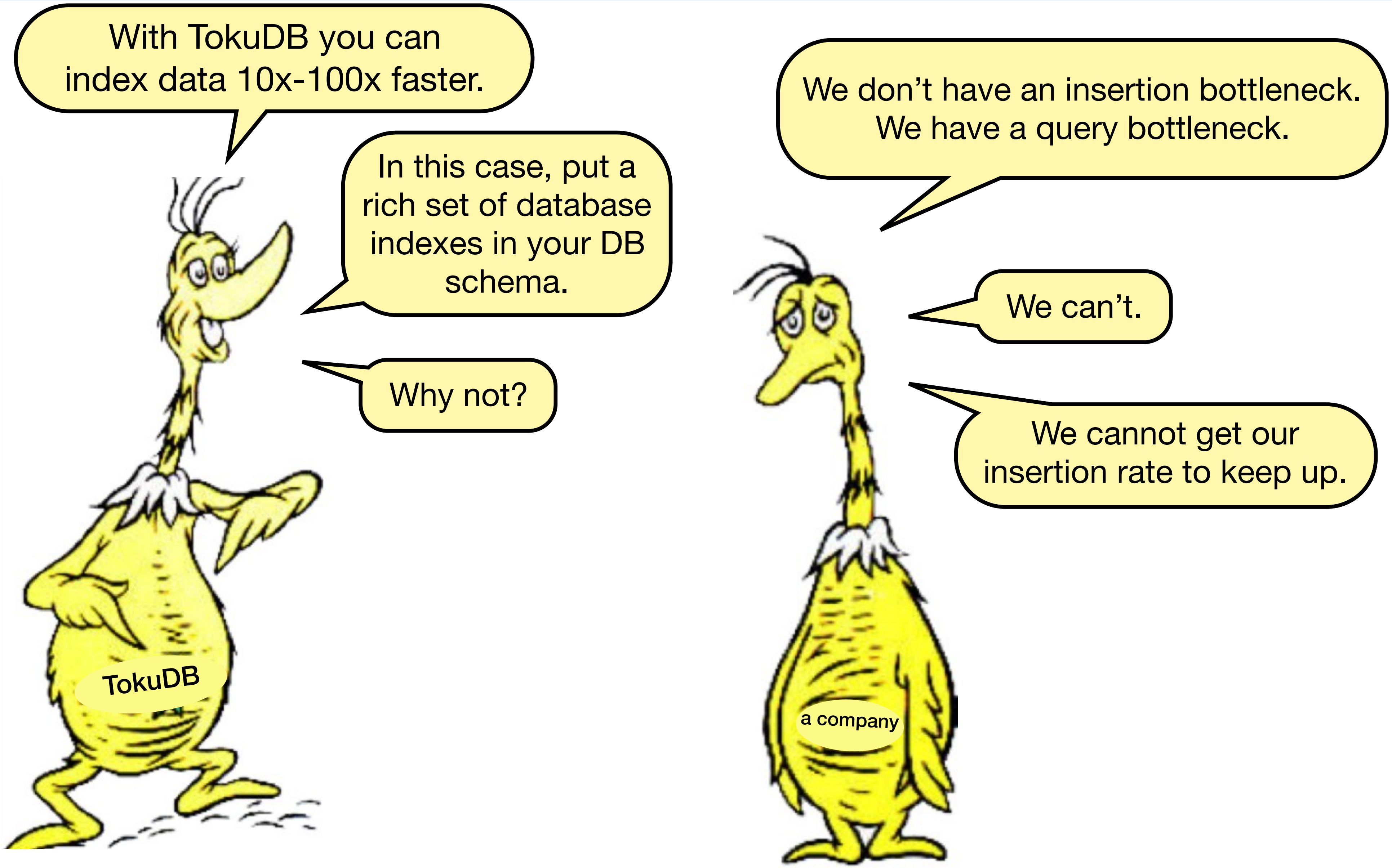
Remember the logging/indexing dilemma?



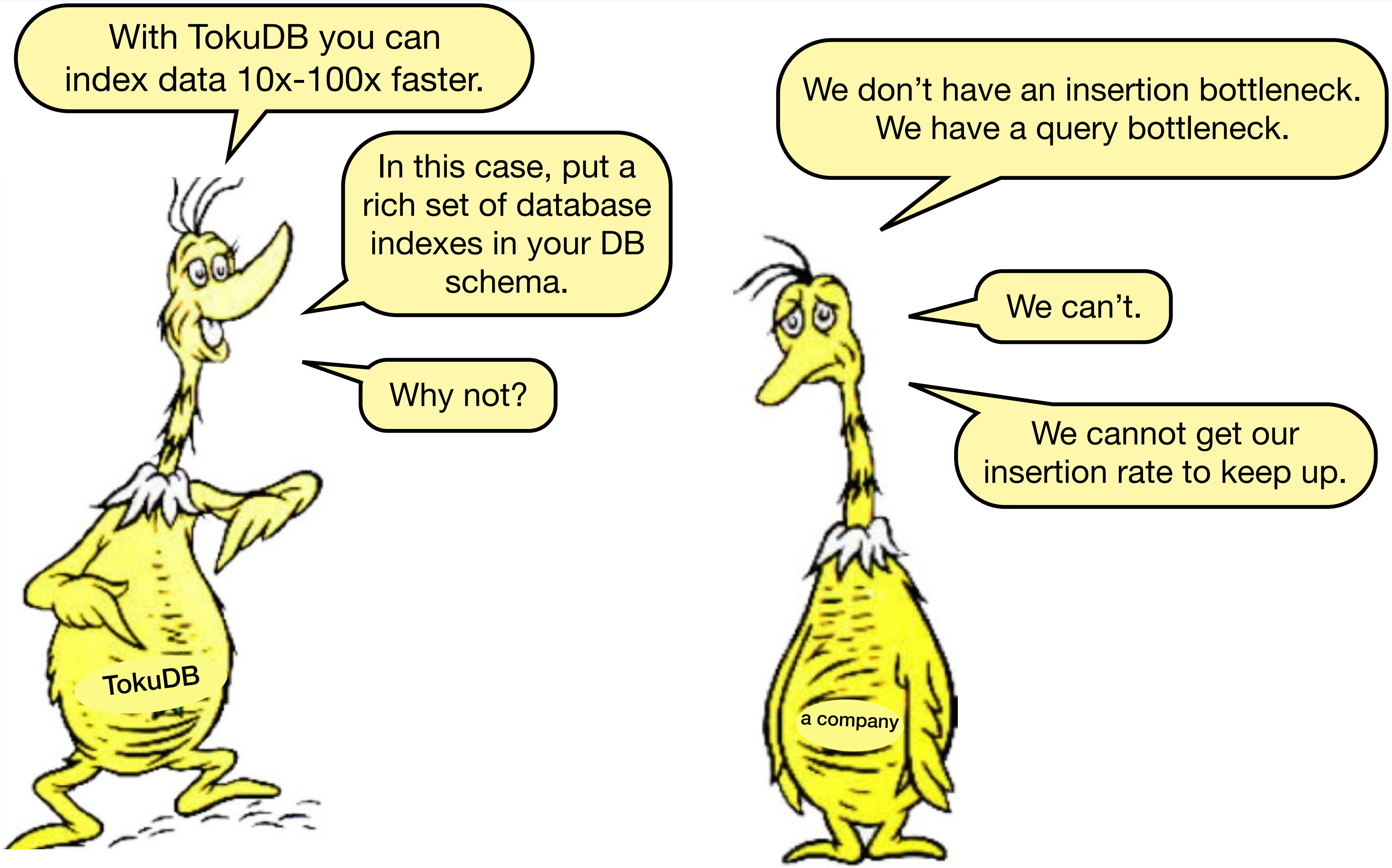
Remember the logging/indexing dilemma?



Remember the logging/indexing dilemma?

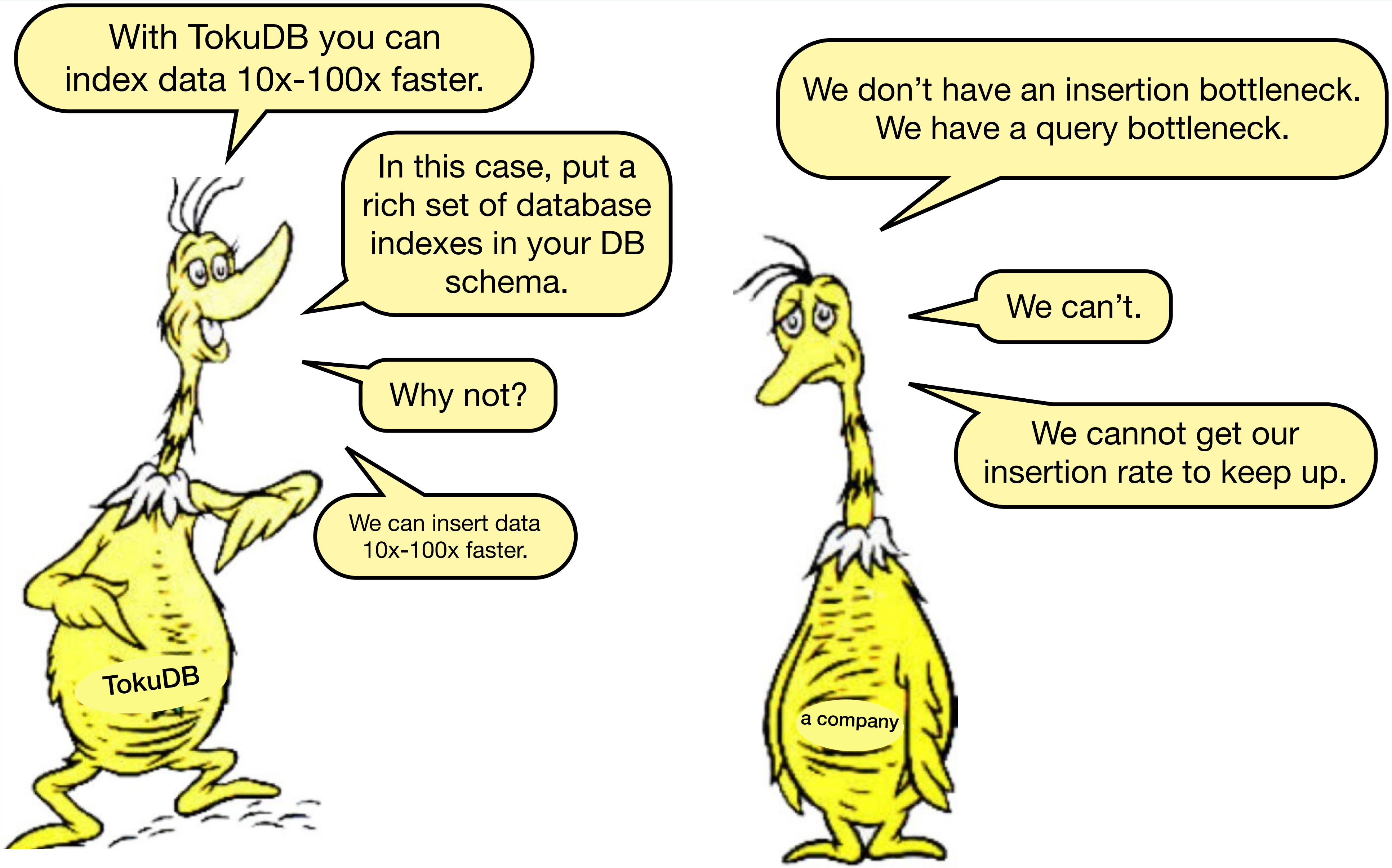


Remember the logging/indexing dilemma?



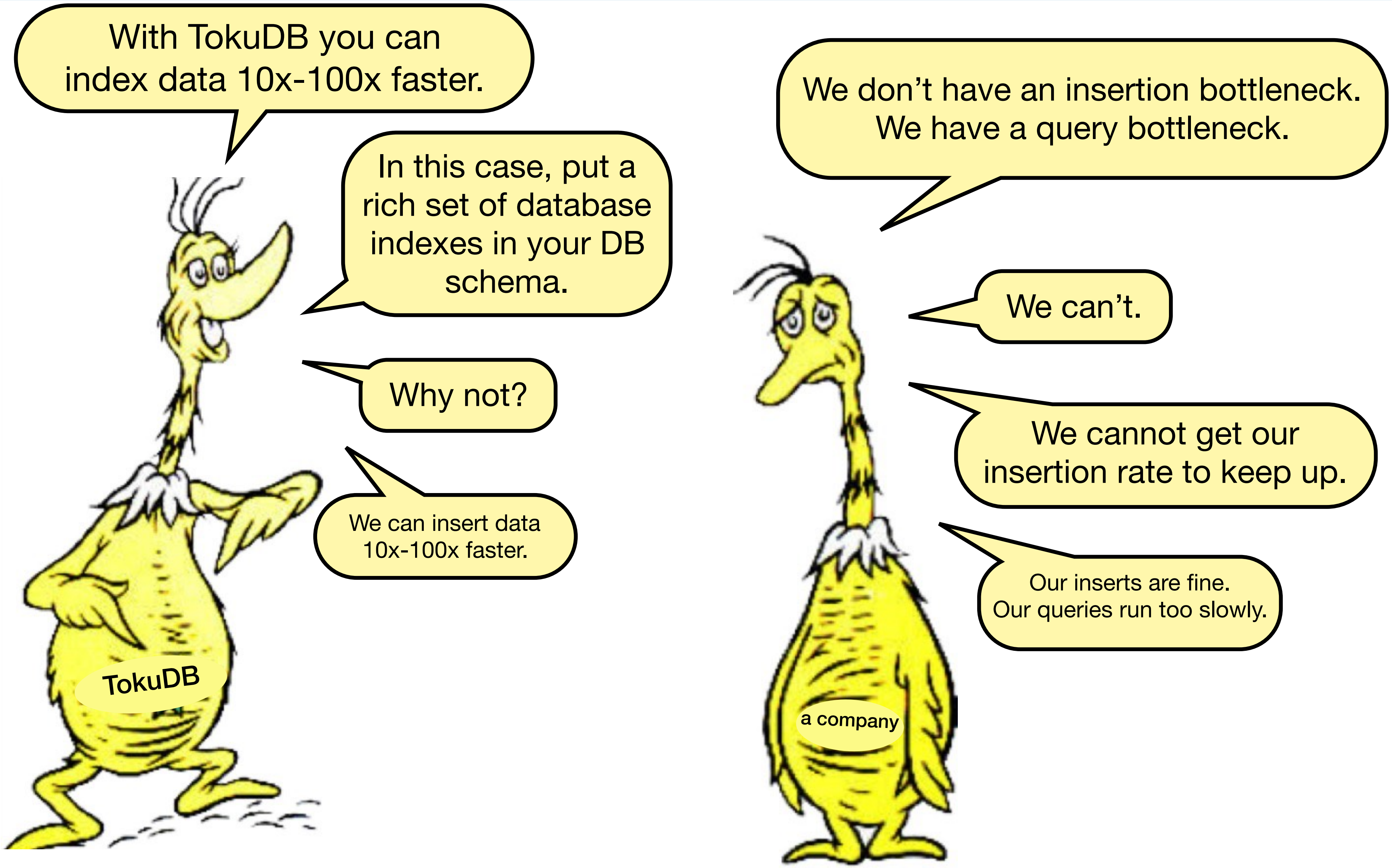
Moral: insertion problems often masquerade as query problems.

Remember the logging/indexing dilemma?



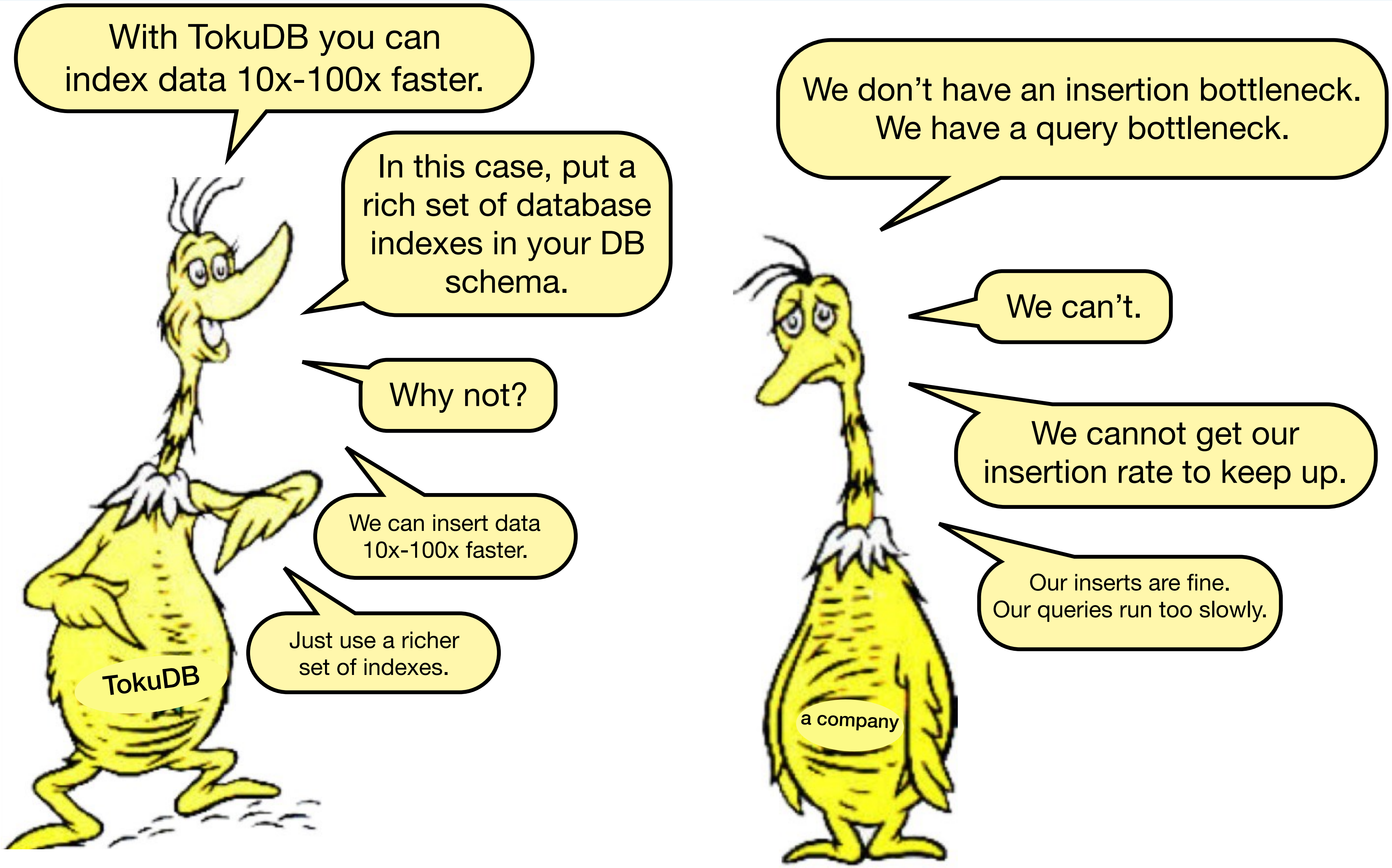
Moral: insertion problems often masquerade as query problems.

Remember the logging/indexing dilemma?



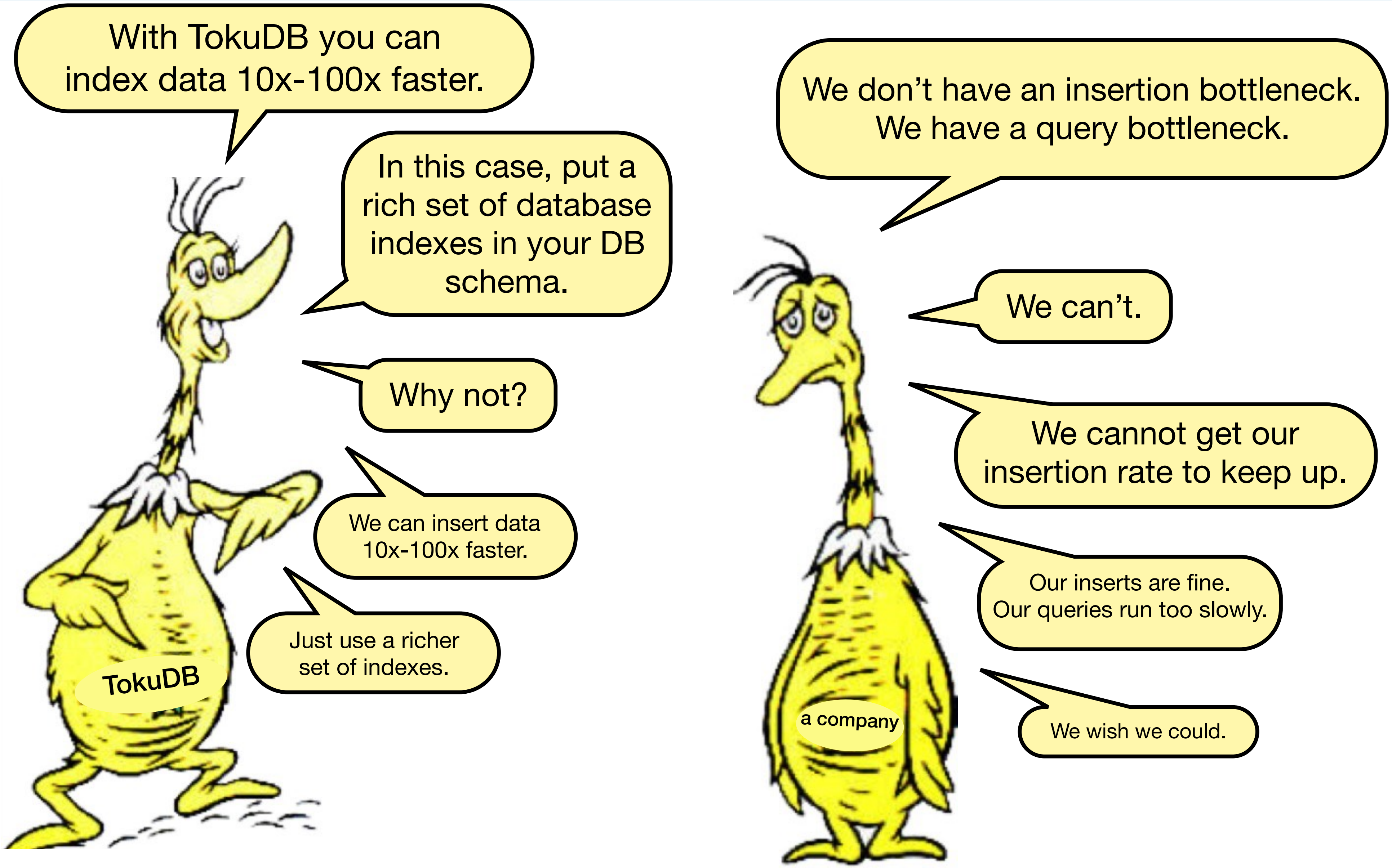
Moral: insertion problems often masquerade as query problems.

Remember the logging/indexing dilemma?



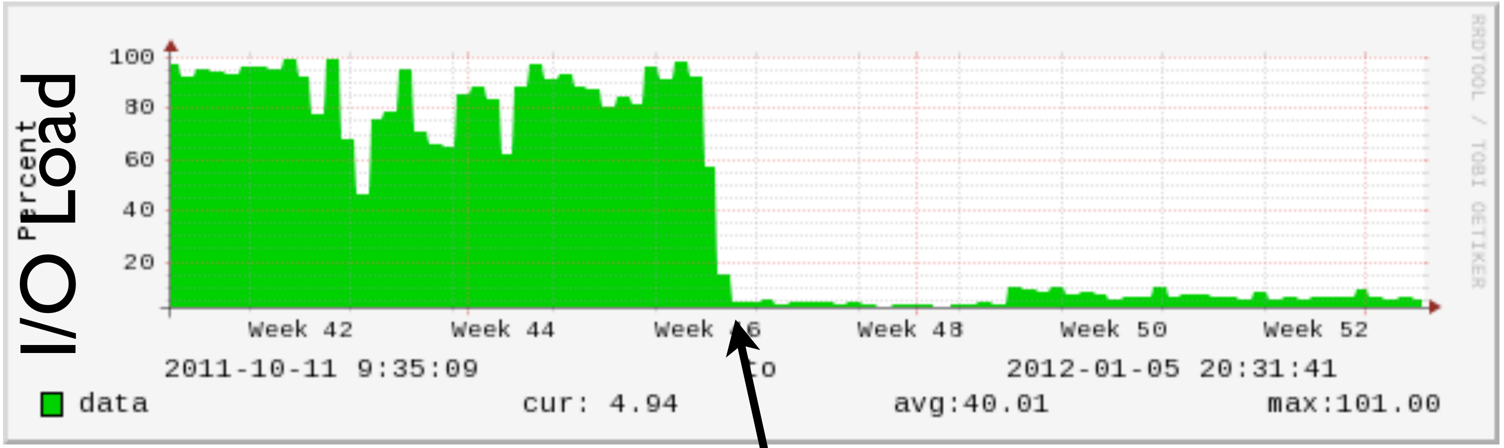
Moral: insertion problems often masquerade as query problems.

Remember the logging/indexing dilemma?



Moral: insertion problems often masquerade as query problems.

The right read optimization is write optimization



The right index makes queries run fast.
WODS can maintain them.

Fast writing is a currency we use to make queries faster.

WODs force you to reexamine your system design…

Insert/point query asymmetry

- Inserts can be fast:
  50-100K random writes/sec on a disk.
- Point queries are provably slow:
  <200 random reads/sec on a disk.

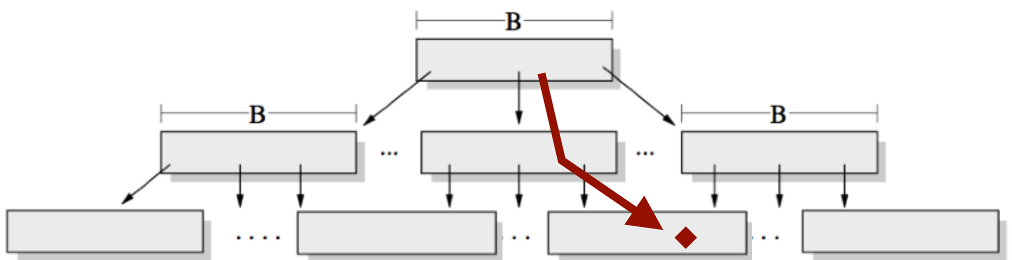
Systems are often designed assuming reads and writes have about the same cost.

In fact, writing is easier than reading.

Systems often assume search cost = insert cost

Ancillary search—a search with each insert.

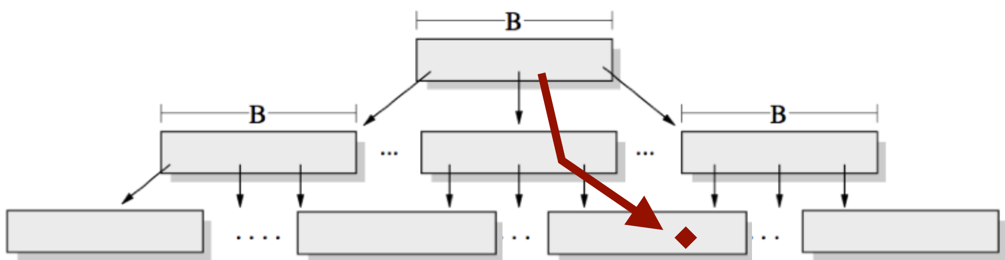
- Insert with uniqueness check—
  is the key is already present?
- Delete with acknowledgement—
  was a key actually deleted?



Systems often assume search cost = insert cost

Ancillary search—a search with each insert.

- Insert with uniqueness check—
  is the key is already present?
- Delete with acknowledgement—
  was a key actually deleted?

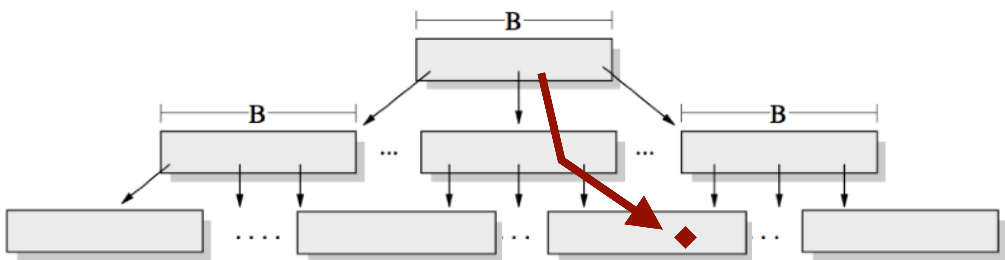


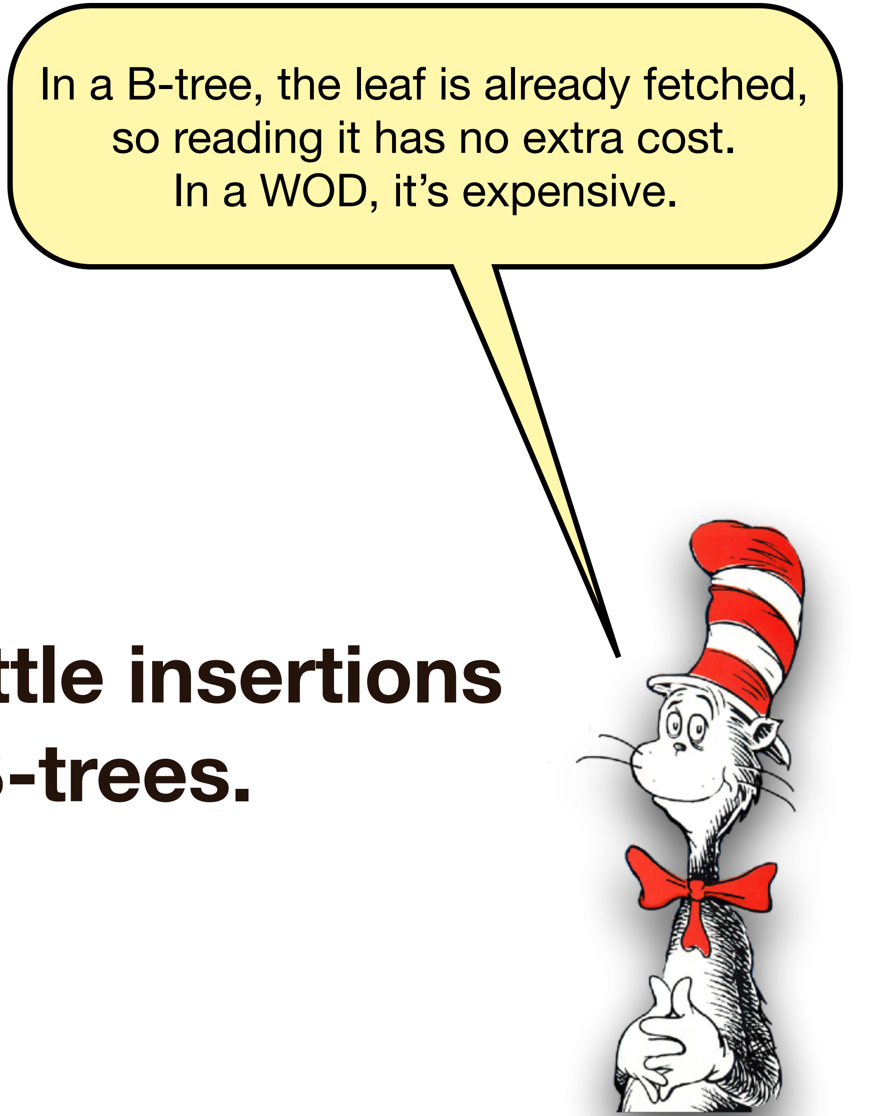
These ancillary searches throttle insertions down to the performance of B-trees.

Systems often assume search cost = insert cost

Ancillary search—a search with each insert.

- Insert with uniqueness check—is the key is already present?
- Delete with acknowledgement—was a key actually deleted?

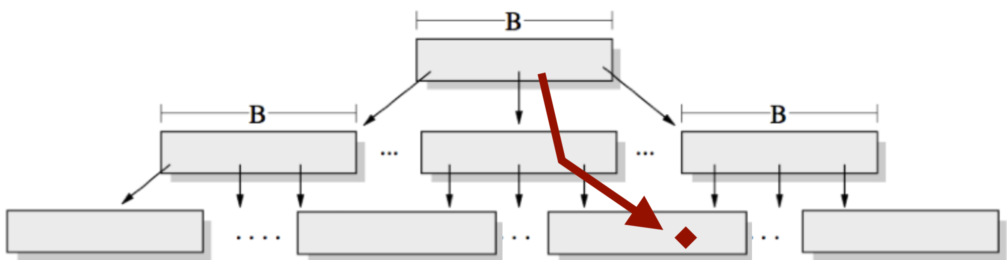


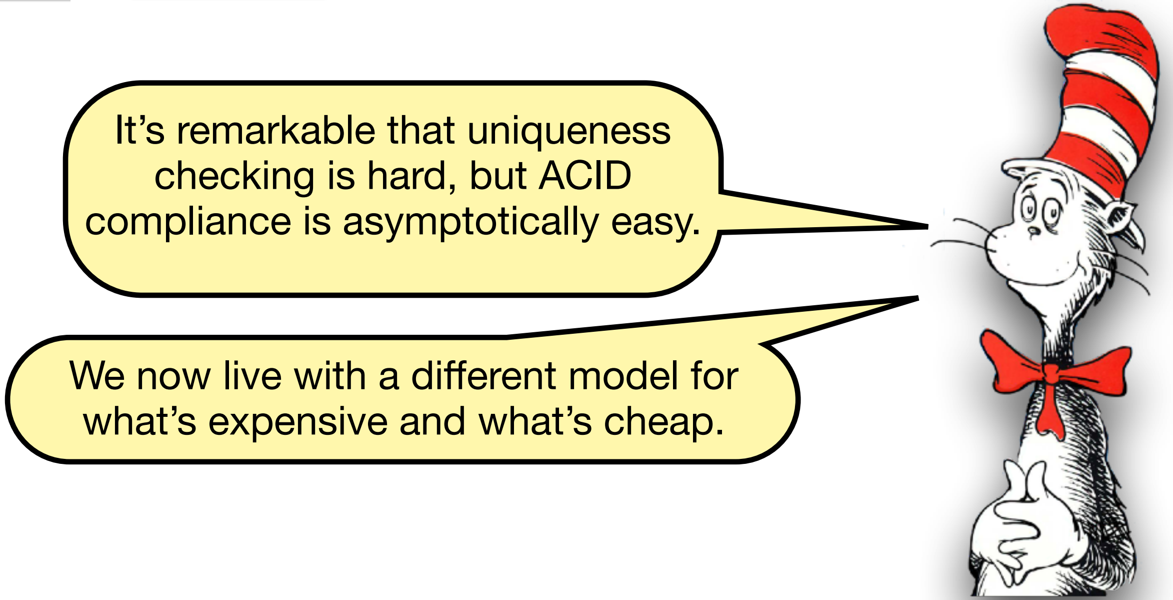


These ancillary searches throttle insertions down to the performance of B-trees.

How can we get rid of ancillary searches?

Write-optimized systems must get rid of or mitigate ancillary searches whenever possible.





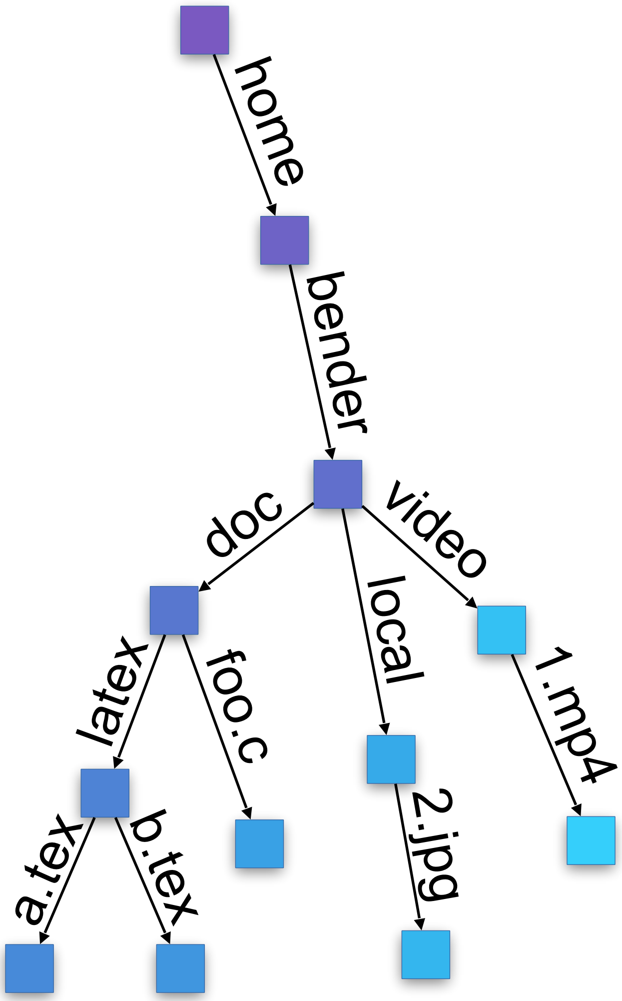
Using WODs in File Systems

(BetrFS, TokuFS, TableFS are examples of write-optimized file systems. I'll talk about BetrFS)

Using WODs in File Systems

(BetrFS, TokuFS, TableFS are examples of write-optimized file systems. I'll talk about BetrFS)

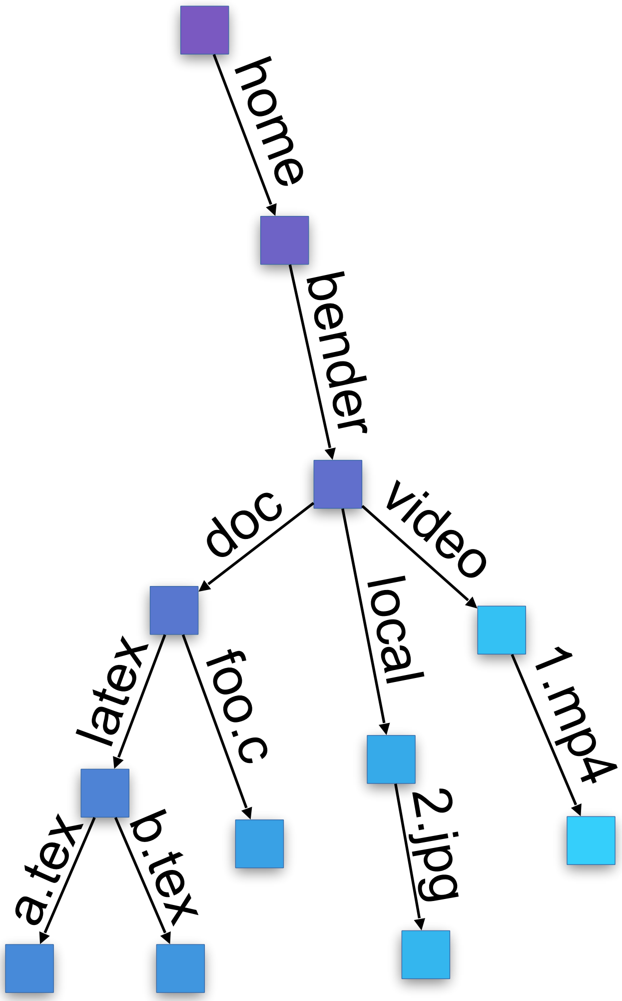
Logging versus indexing tradeoff in file systems



directory tree

Logging versus indexing tradeoff in file systems

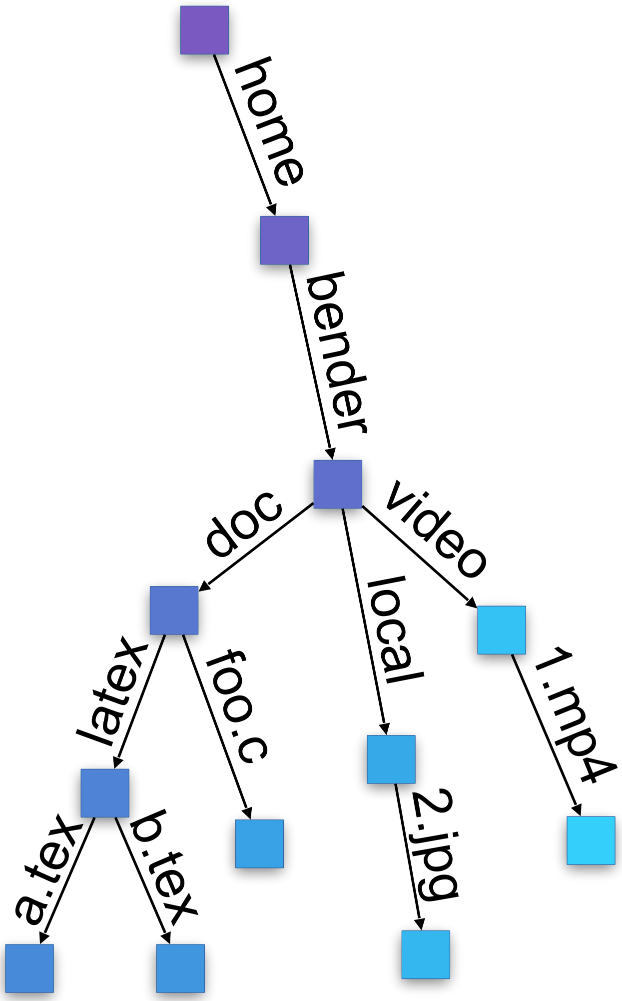
How should we organize the files on disk?



directory tree

Logging versus indexing tradeoff in file systems

How should we organize the files on disk?



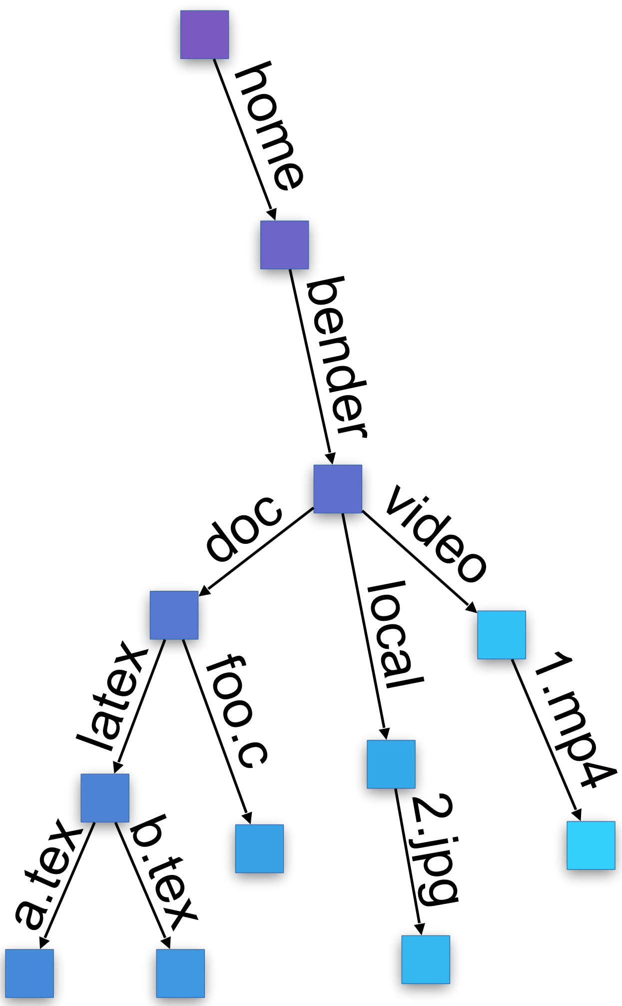
directory tree

logical order ⟹ sequential scans are fast

- grep -r “bar” .
- ls -R .

Logging versus indexing tradeoff in file systems

How should we organize the files on disk?



directory tree

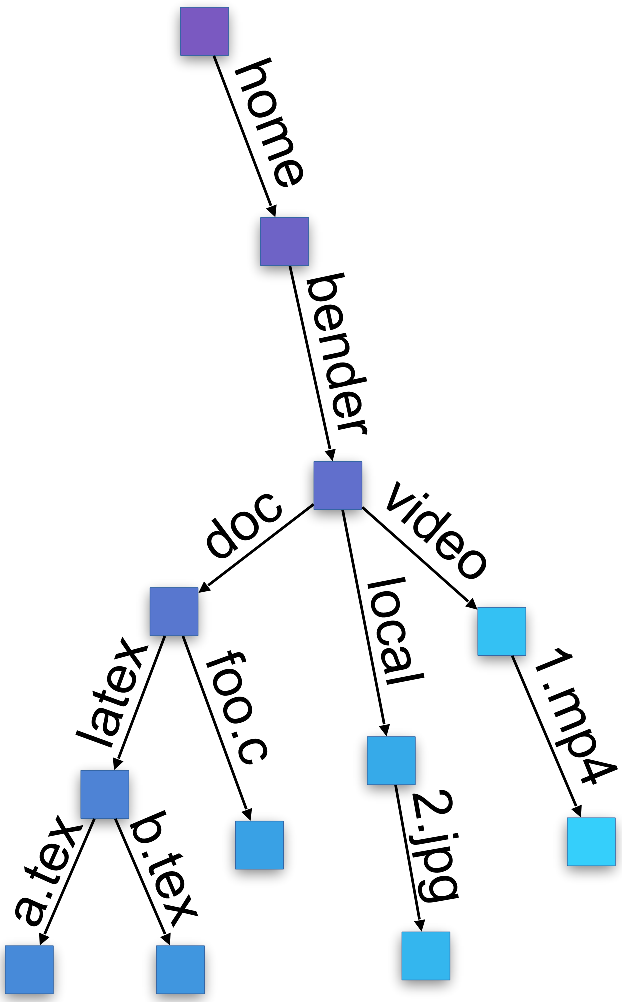
logical order ⟹ sequential scans are fast

- grep -r “bar” .
- ls -R .

update order ⟹ small writes are fast

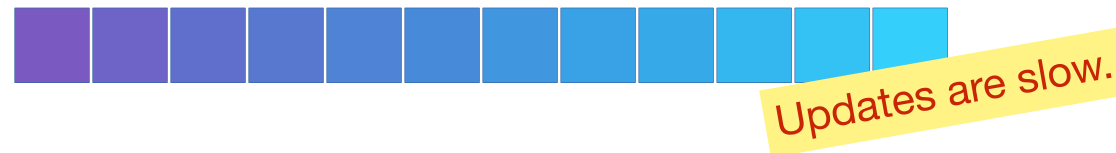
Logging versus indexing tradeoff in file systems

How should we organize the files on disk?



directory tree

logical order ⟹ sequential scans are fast

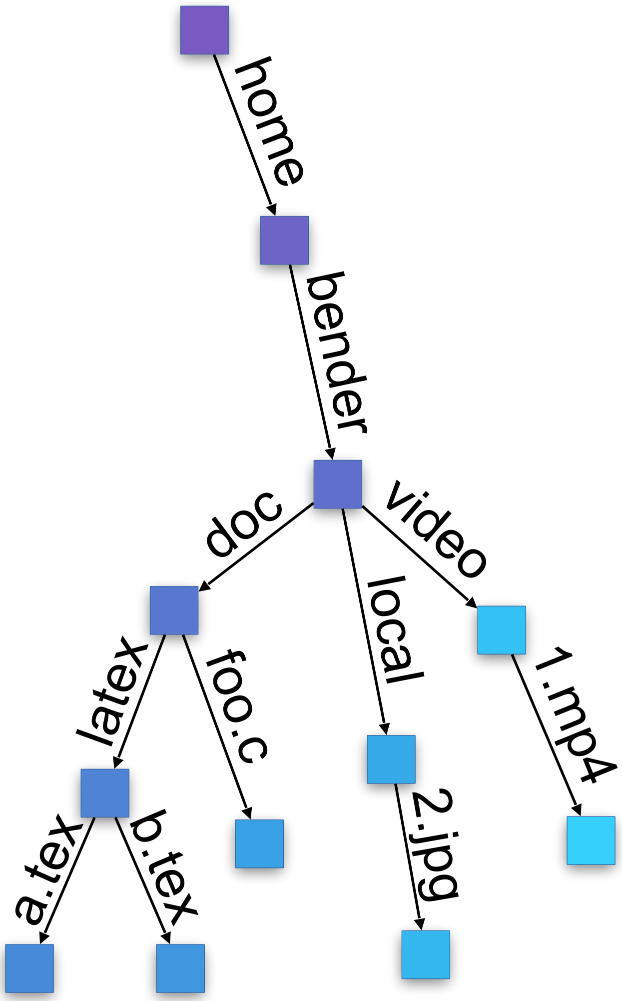


- grep -r “bar” .
- ls -R .

update order ⟹ small writes are fast

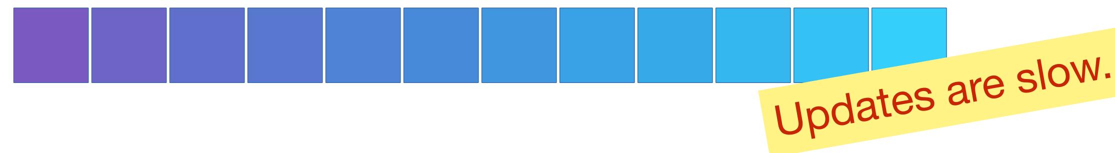
Logging versus indexing tradeoff in file systems

How should we organize the files on disk?



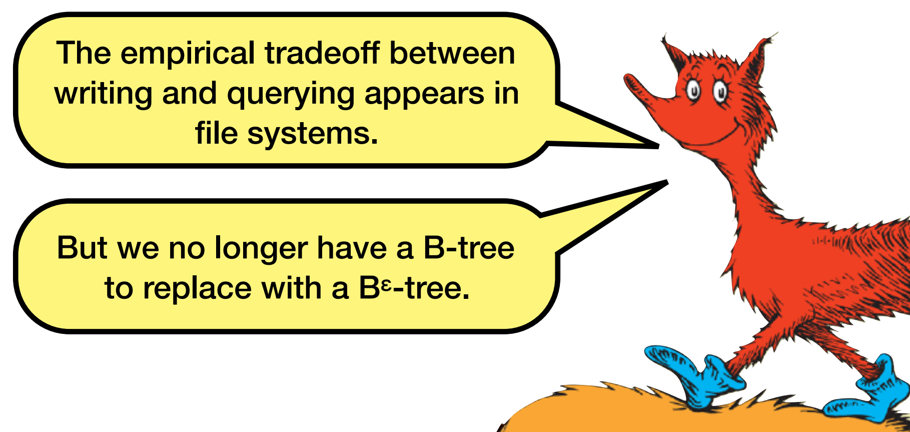
directory tree

logical order ⟹ sequential scans are fast



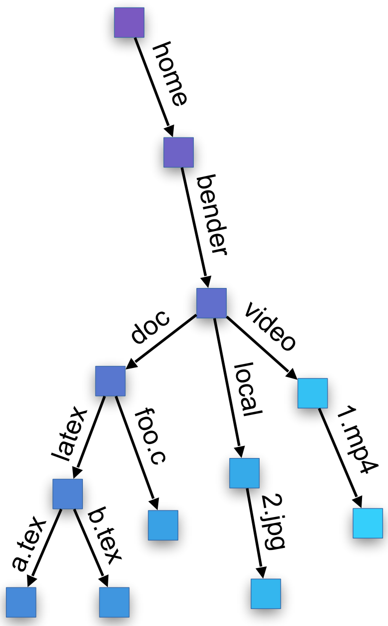
- grep -r “bar” .
- ls -R .

update order ⟹ small writes are fast



Full-path indexing

Maintain two WODs, each indexed on the path names.



directory tree

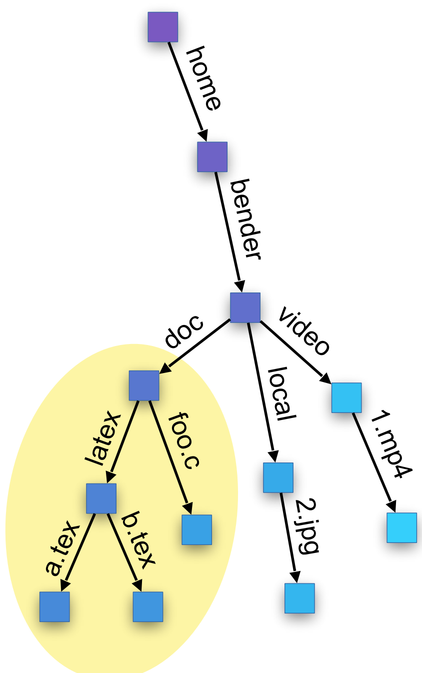
<path, file metadata>
...
/home/bender/doc
/home/bender/doc/latex/
/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex
/home/bender/doc/foo.c
/home/bender/local
...
...
...

<path, file data>
...
/home/bender/doc
/home/bender/doc/latex/
/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex
/home/bender/doc/foo.c
/home/bender/local
...
...
...

File-system operations → inserts and range queries.

Full-path indexing

Maintain two WODs, each indexed on the path names.



directory tree

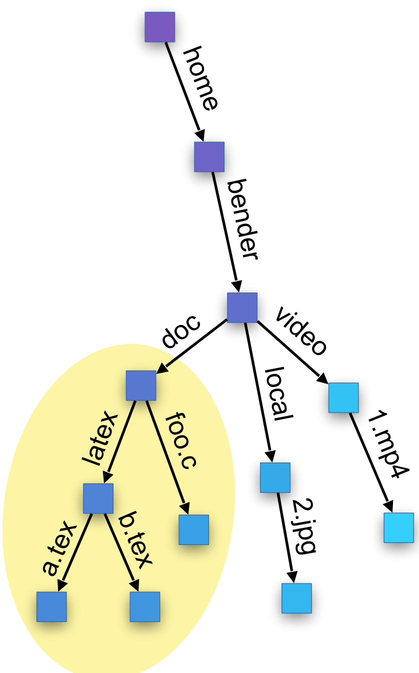
<path, file metadata>
...
/home/bender/doc
/home/bender/doc/latex/
/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex
/home/bender/doc/foo.c
/home/bender/local
...
...
...

<path, file data>
...
/home/bender/doc
/home/bender/doc/latex/
/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex
/home/bender/doc/foo.c
/home/bender/local
...
...
...

File-system operations → inserts and range queries.

Full-path indexing

Maintain two WODs, each indexed on the path names.



directory tree

ls -R

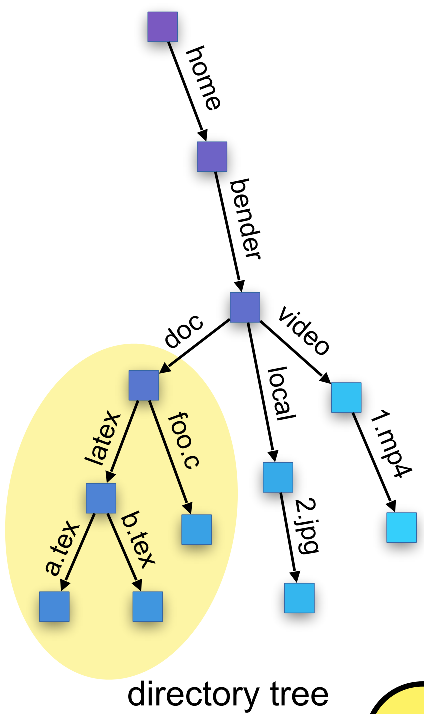
<path, file metadata>
...
/home/bender/doc
/home/bender/doc/latex/
/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex
/home/bender/doc/foo.c
/home/bender/local
...
...
...

<path, file data>
...
/home/bender/doc
/home/bender/doc/latex/
/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex
/home/bender/doc/foo.c
/home/bender/local
...
...
...

File-system operations → inserts and range queries.

Full-path indexing

Maintain two WODs, each indexed on the path names.



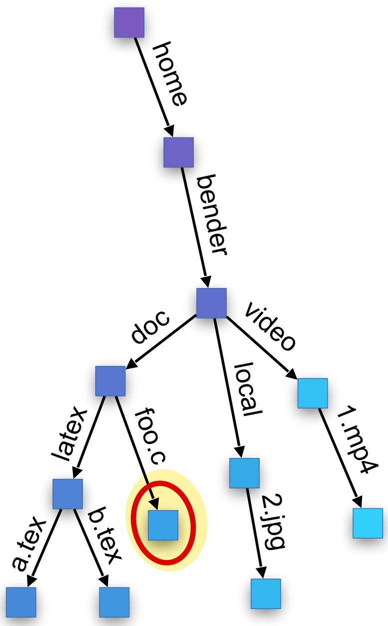
<path, file metadata>	<path, file data>
...	...
/home/bender/doc	/home/bender/doc
/home/bender/doc/latex/	/home/bender/doc/latex/
/home/bender/doc/latex/a.tex	/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex	/home/bender/doc/latex/b.tex
/home/bender/doc/foo.c	/home/bender/doc/foo.c
/home/bender/local	/home/bender/local
...	...
...	...
...	...

ls -R

grep -r

File-system operations → inserts and range queries.

Maintain two WODs, each indexed on the path names.



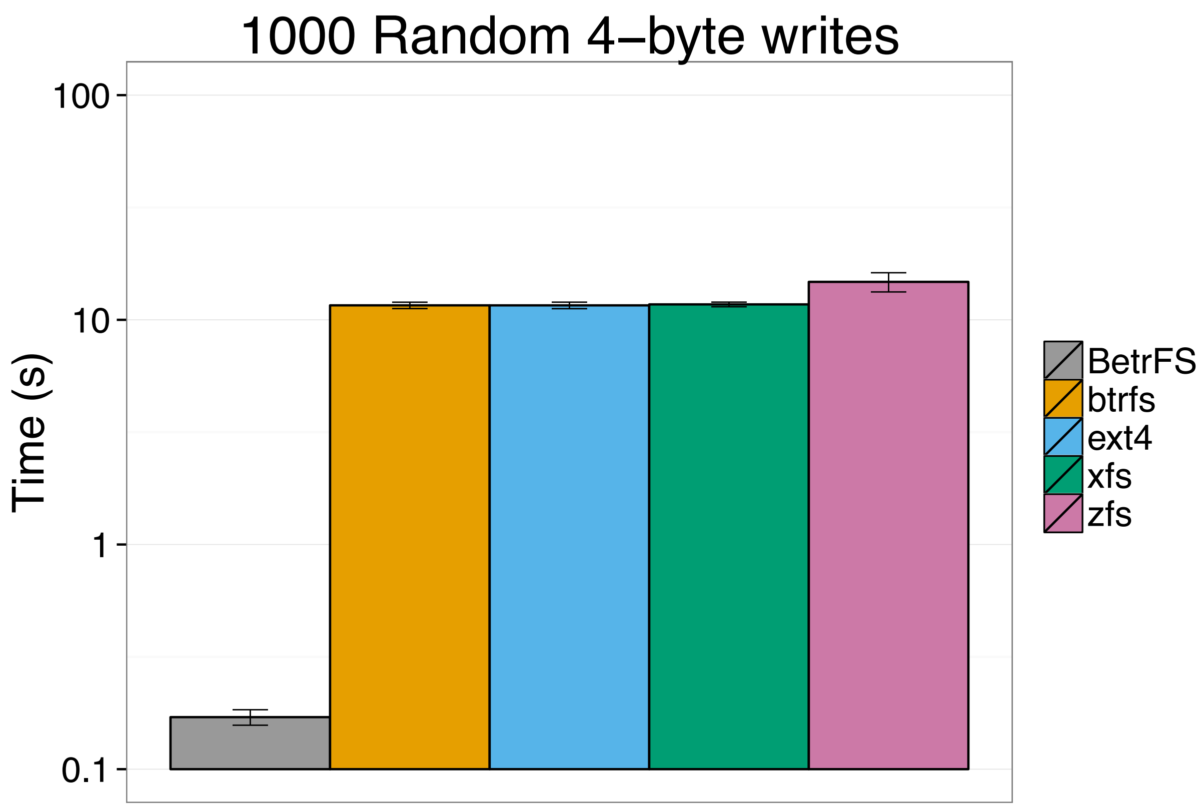
directory tree

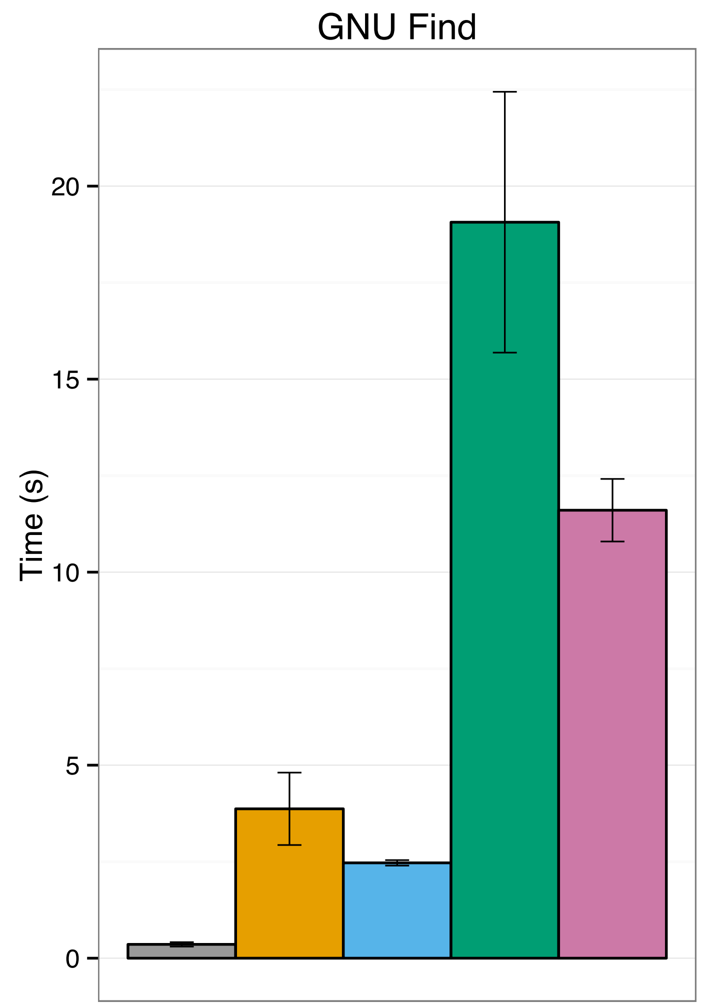
<path, file metadata>
...
/home/bender/doc
/home/bender/doc/latex/
/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex
~~/home/bender/doc/foo~~ rm
/home/bender/local
...
...
...

<path, file data>
...
/home/bender/doc
/home/bender/doc/latex/
/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex
~~/home/bender/doc/foo~~.c
/home/bender/local
...
...
...

File-system operations → inserts and range queries.

Microwrite and Scan Performance on BetrFs





1 GiB file, random data

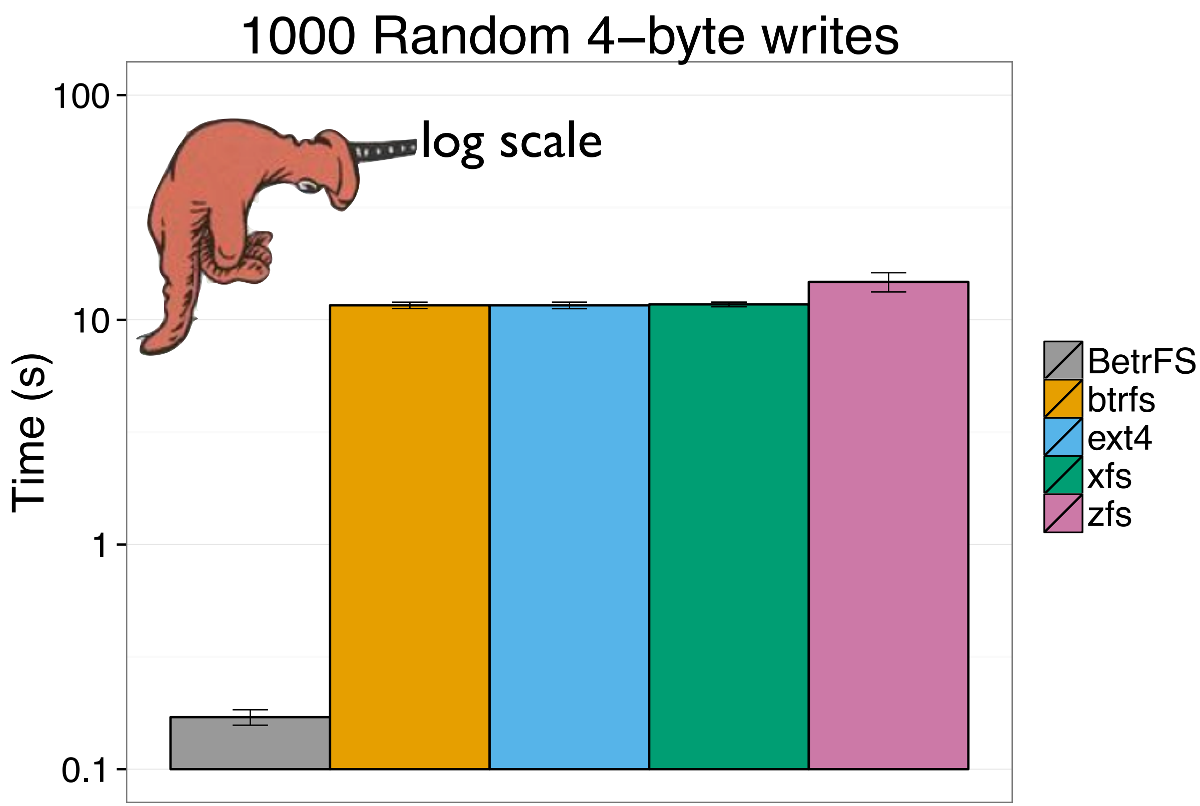
1,000 random 4-byte writes

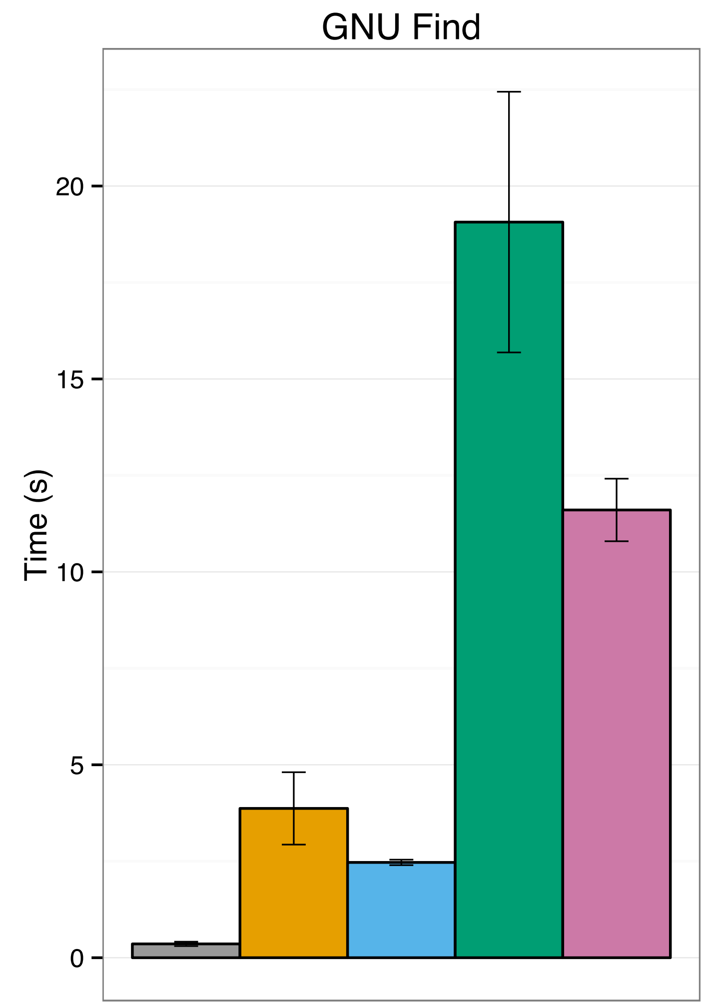
`fsync()` at end

*lower is better

[BetrFS: Jannen, Yuan, Zhan, Akshintala, Esmet, Jiao, Mittal, Pandey, Reddy, Walsh, Bender, Farach-Colton, Johnson, Kuszmaul, Porter, FAST 15]

Microwrite and Scan Performance on BetrFs





1 GiB file, random data

1,000 random 4-byte writes

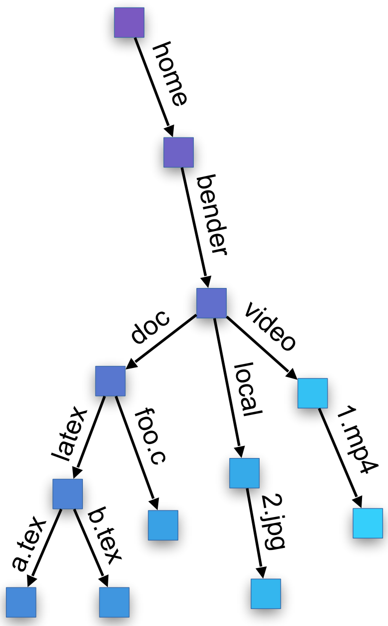
`fsync()` at end

*lower is better

[BetrFS: Jannen, Yuan, Zhan, Akshintala, Esmet, Jiao, Mittal, Pandey, Reddy, Walsh, Bender, Farach-Colton, Johnson, Kuszmaul, Porter, FAST 15]

Full-path indexing

Maintain two WODs, each indexed on the path names.



directory tree

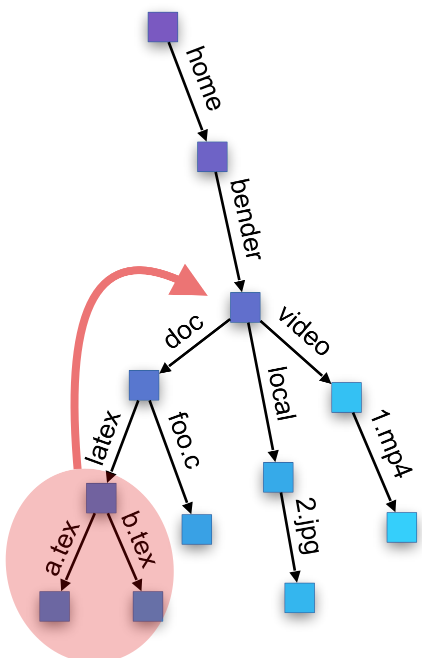
<path, file metadata>
...
/home/bender/doc
/home/bender/doc/latex/
/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex
/home/bender/doc/foo.c
/home/bender/local
...
...
...

<path, file data>
...
/home/bender/doc
/home/bender/doc/latex/
/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex
/home/bender/doc/foo.c
/home/bender/local
...
...
...

Some file-system operations don’t seem to map cheaply.

Full-path indexing

Maintain two WODs, each indexed on the path names.

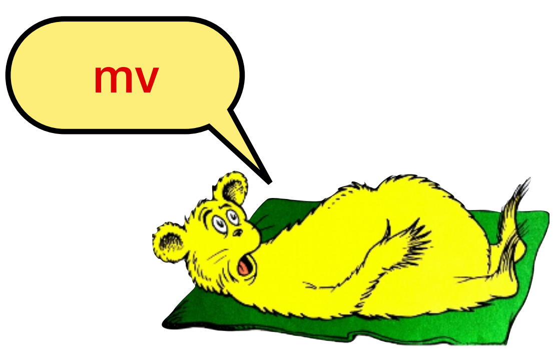


directory tree

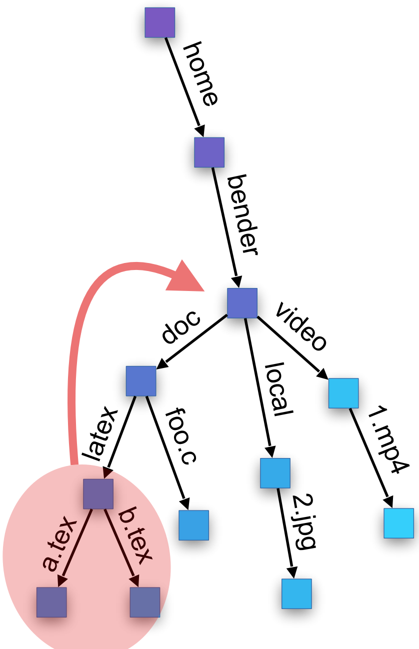
<path, file metadata>
...
/home/bender/doc
/home/bender/doc/latex/
/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex
/home/bender/doc/foo.c
/home/bender/local
...
...
...

<path, file data>
...
/home/bender/doc
/home/bender/doc/latex/
/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex
/home/bender/doc/foo.c
/home/bender/local
...
...
...

Some file-system operations don’t seem to map cheaply.



Maintain two WODs, each indexed on the path names.



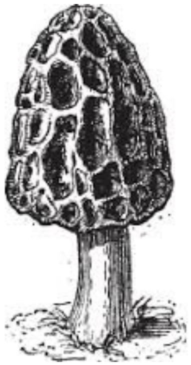
directory tree

<path, file metadata>
...
/home/bender/doc
/home/bender/doc/latex/
/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex
/home/bender/doc/foo.c
/home/bender/local
...
...
...

<path, file data>
...
/home/bender/doc
/home/bender/doc/latex/
/home/bender/doc/latex/a.tex
/home/bender/doc/latex/b.tex
/home/bender/doc/foo.c
/home/bender/local
...
...
...

Some file-system operations don't seem to map cheaply.

These keys change their names.
They move to a different place in the order.



Moral: how can we make write-optimized data structures that support the richer set of operations needed by the applications?

We need more than just insert and delete.

Other WODs advantages

Other WODs advantages

B$^{\varepsilon}$-trees can use bigger nodes than B-trees

- Better compression
- Less fragmentation.

B$^{\varepsilon}$-trees file systems do not age the way B-tree based file systems do.

[Conway, Bakshi, Jiao, Zhan, Bender, Jannen, Johnson, Kuszmaul, Porter, Yuan, Farach-Colton 17]

Other WODs advantages

B^{ε}-trees can use bigger nodes than B-trees

- Better compression
- Less fragmentation.

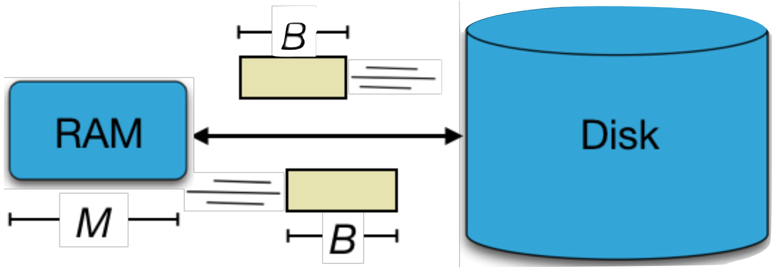
B^{ε}-trees file systems do not age the way B-tree based file systems do.

[Conway, Bakshi, Jiao, Zhan, Bender, Jannen, Johnson, Kuszmaul, Porter, Yuan, Farach-Colton 17]

We cannot see this in the DAM model.
We need a more refined model.

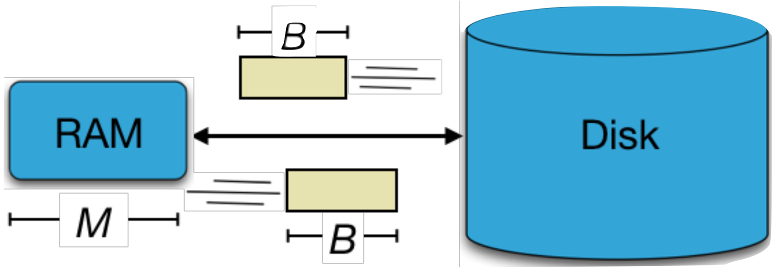
Revisit I/O model to harness full power of write-optimization

DAM is realistic enough to make powerful predictions.



Some things it doesn't predict, such as aging.

Revisit I/O model to harness full power of write-optimization



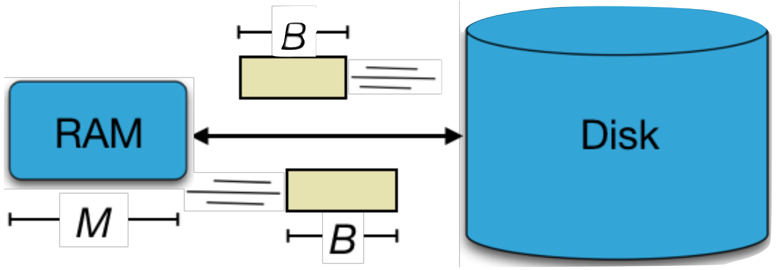
DAM is realistic enough to make powerful predictions.

Some things it doesn't predict, such as aging.

Technology is changing.

- I/O speeds are accelerating faster than CPU.
- Storage technology supports lots of I/Os in parallel.

Revisit I/O model to harness full power of write-optimization



DAM is realistic enough to make powerful predictions.

Some things it doesn't predict, such as aging.

Technology is changing.

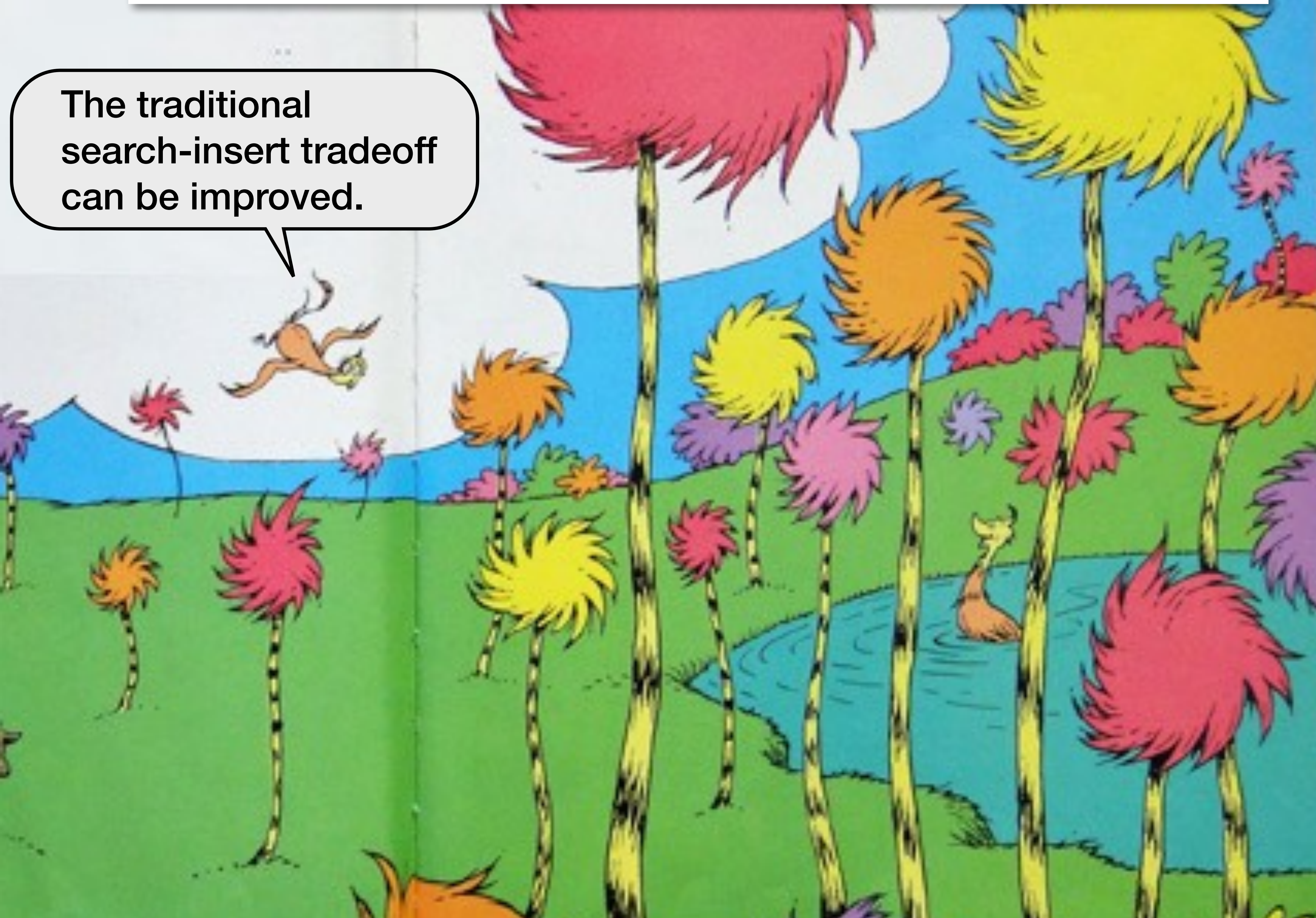
- I/O speeds are accelerating faster than CPU.
- Storage technology supports lots of I/Os in parallel.

Need multithreading and lots of parallel I/Os to drive the device to its capacity.

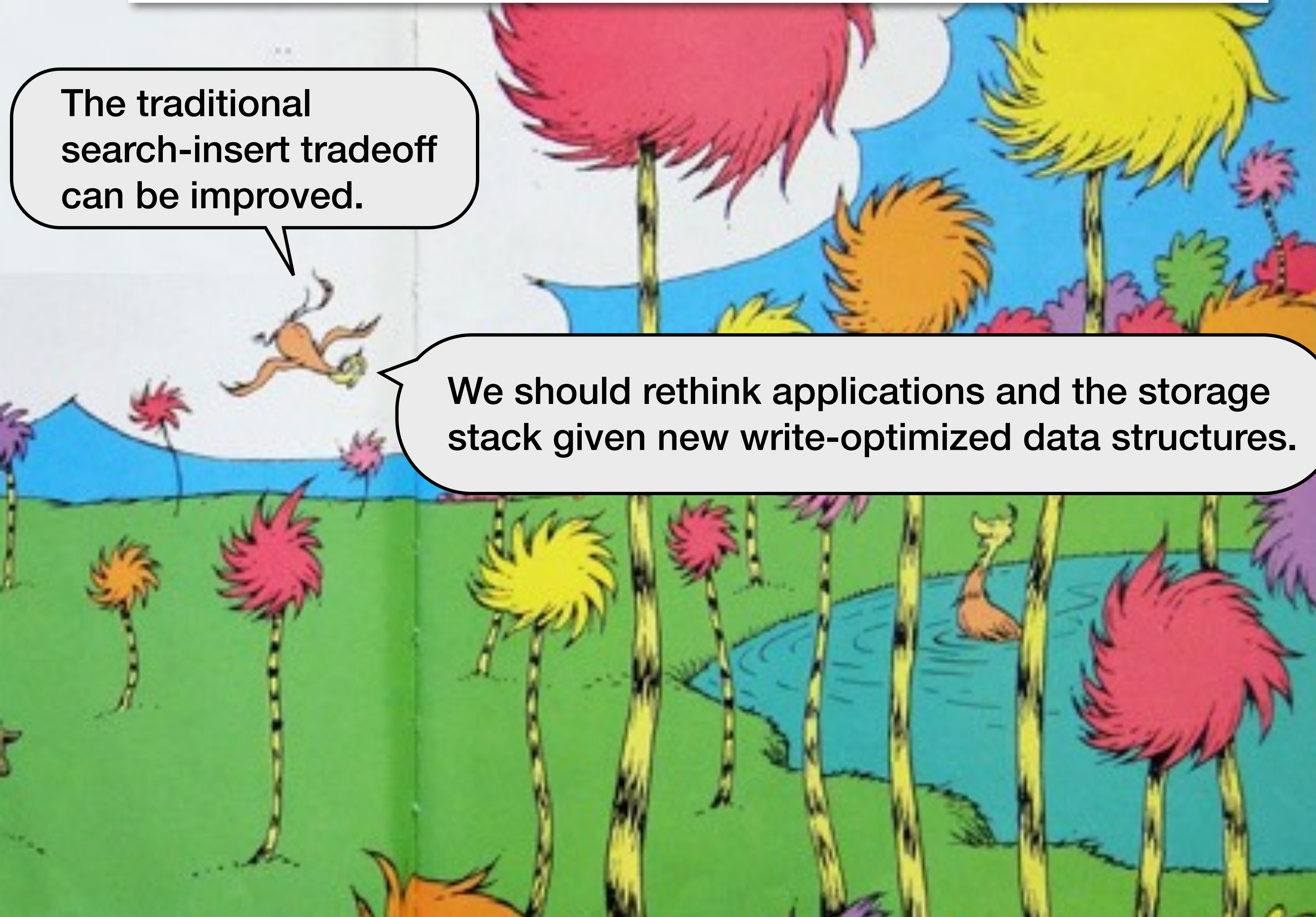
- Data structures for older storage don't work so well now.

Summery Slide

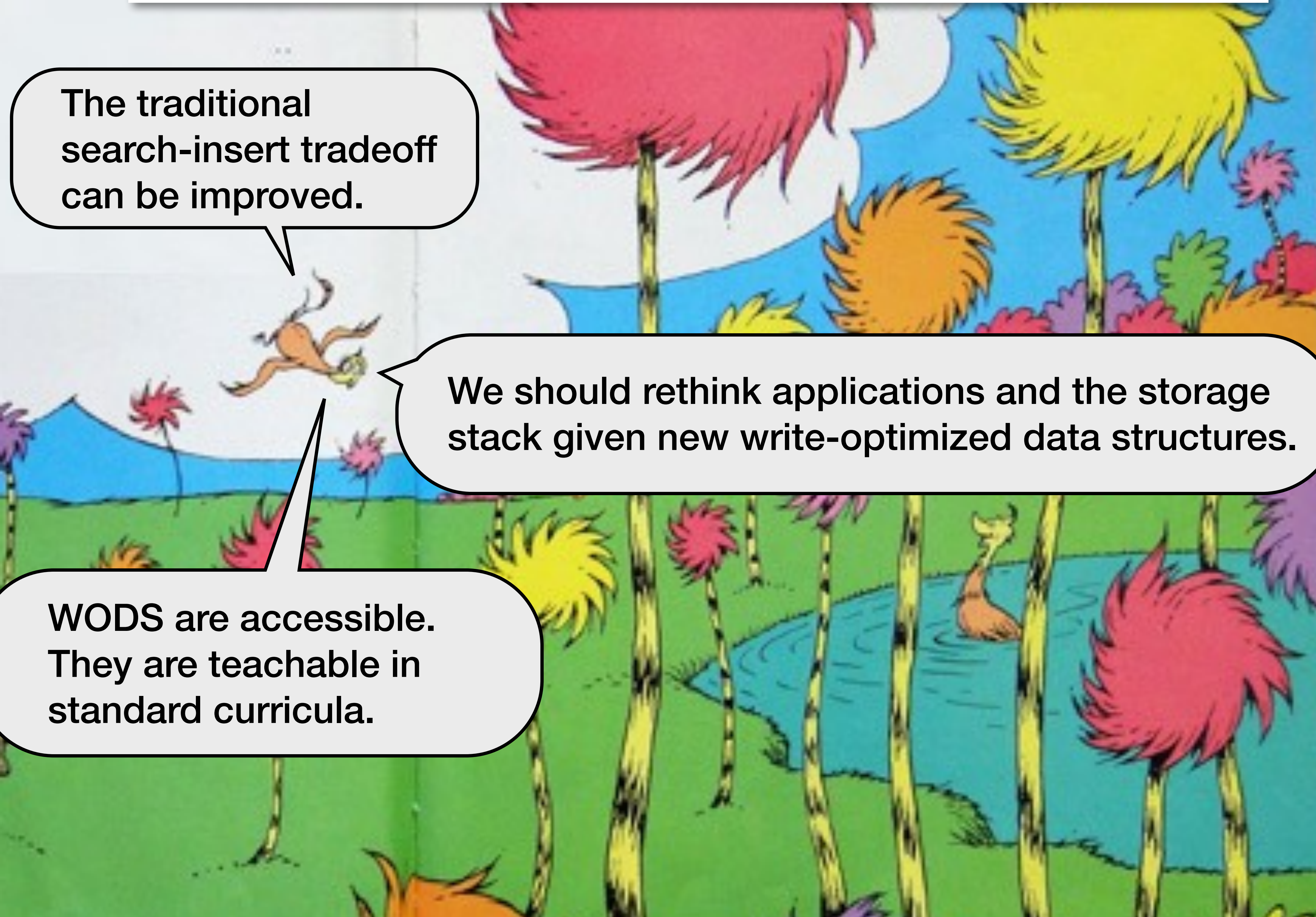
Summery Slide



Summery Slide



Summery Slide



Summery Slide