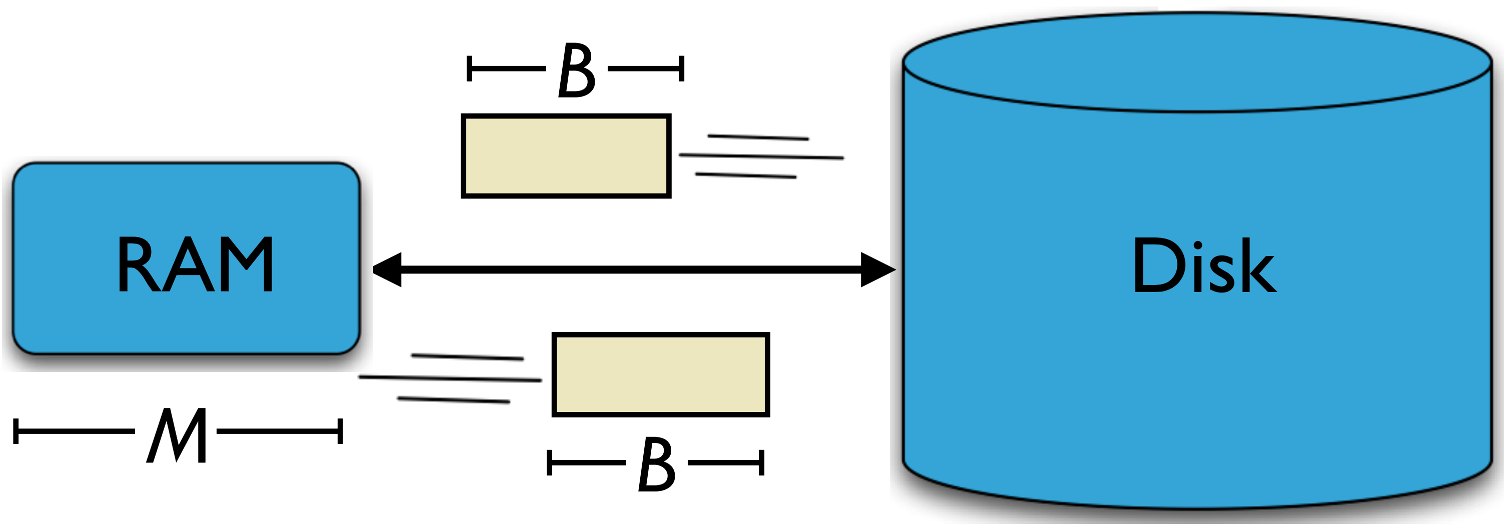
Modeling I/O Using the Disk Access Model

How computation works:

- Data is transferred in blocks between RAM and disk.
- The # 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***.



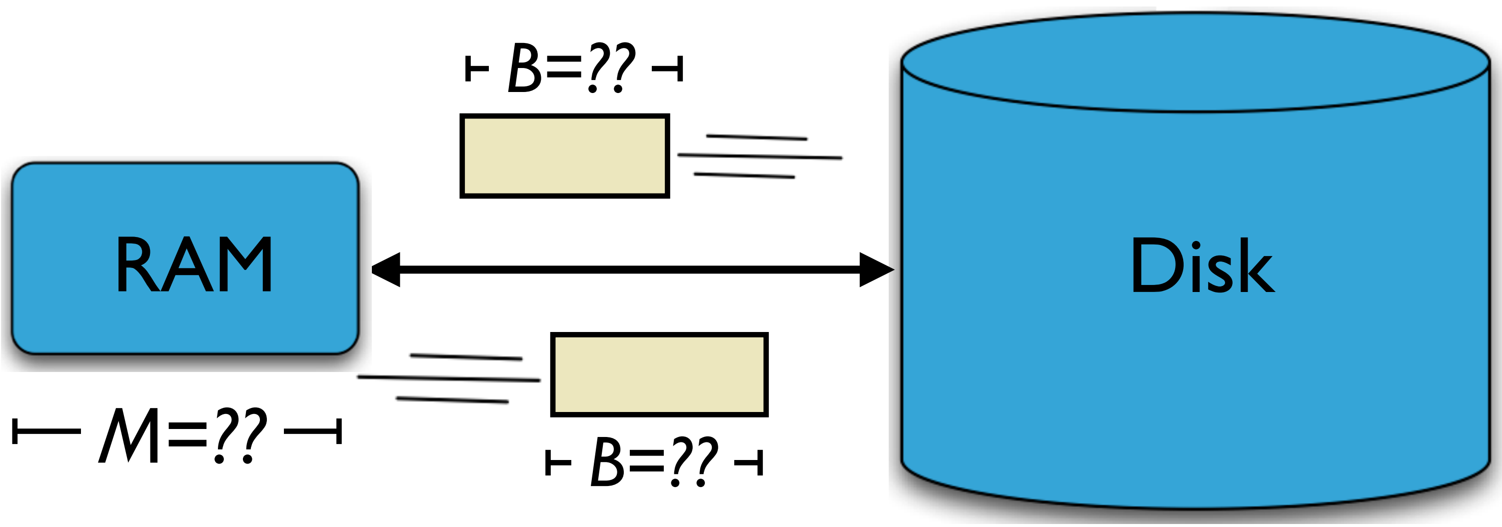
[Aggarwal+Vitter '88]

Cache-Oblivious Analysis

Cache-oblivious analysis:

- Parameters B, M are unknown to the algorithm or coder.
- Performance bounds are parameterized by block size B, memory size M, data size N.

Goal (as before): Minimize # of block transfer



[Frigo, Leiserson, Prokop, Ramachandran ’99]

Write-optimized data structures performance

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 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]

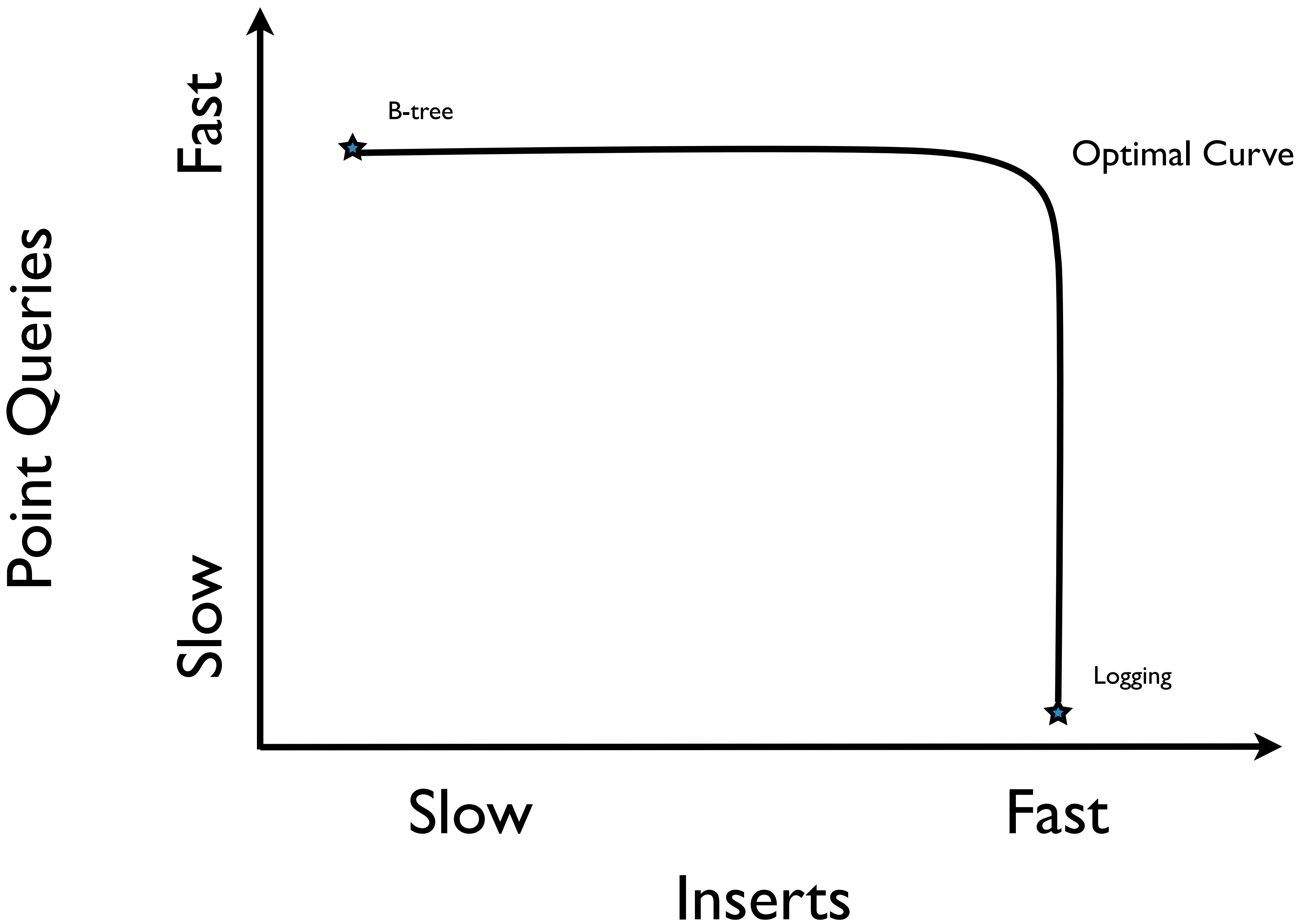
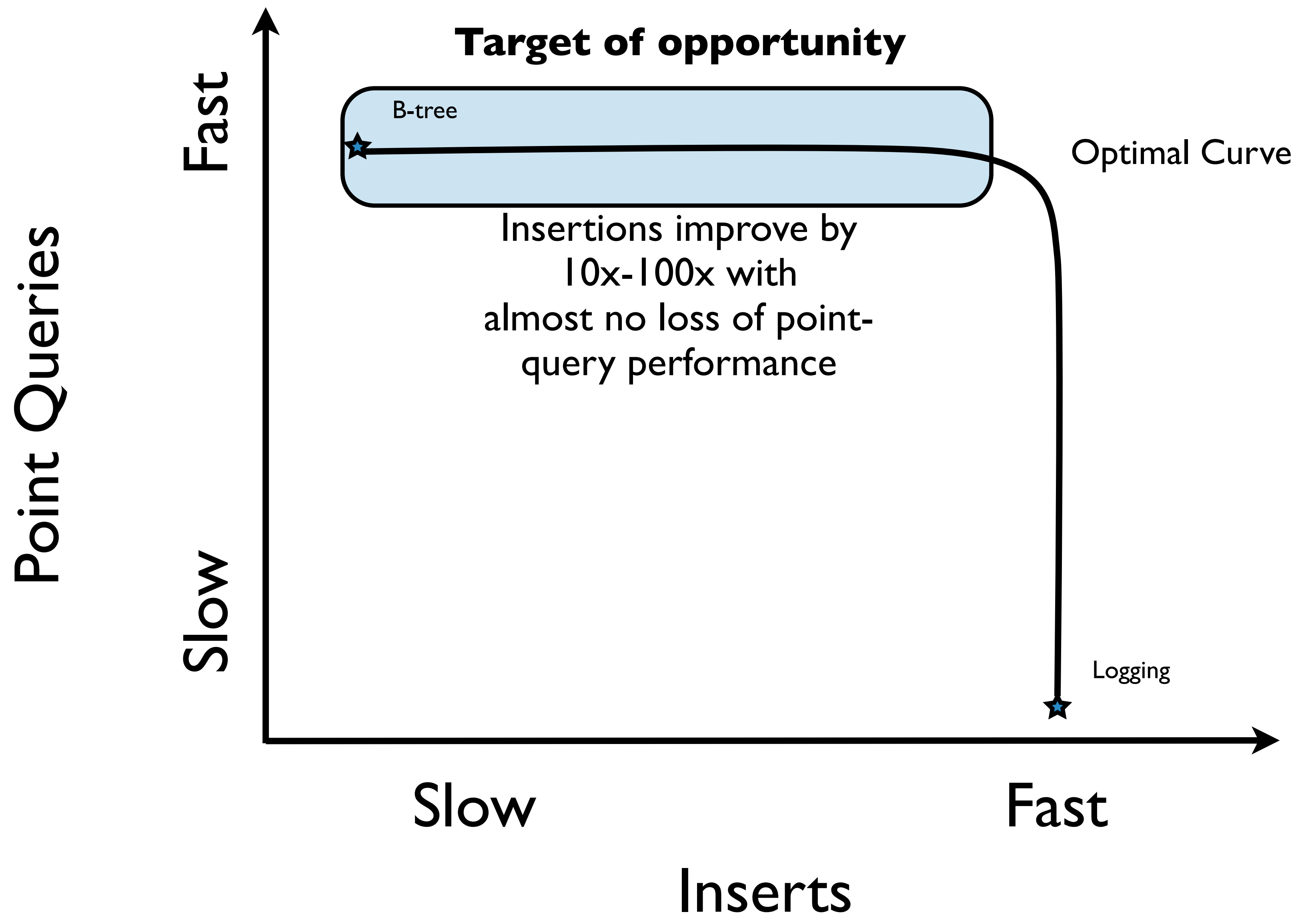


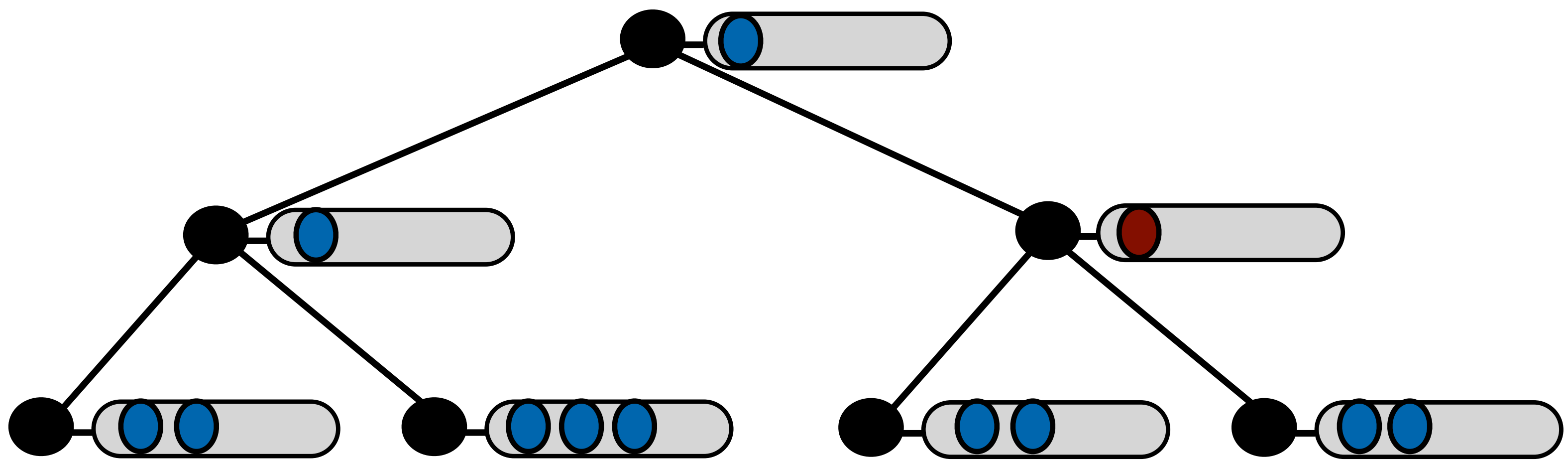
Illustration of Optimal Tradeoff [Brodal, Fagerberg 03]



A simple write-optimized structure

O(log *N*) queries and O((log *N*)/*B*) inserts:

- A balanced binary tree with buffers of size *B*



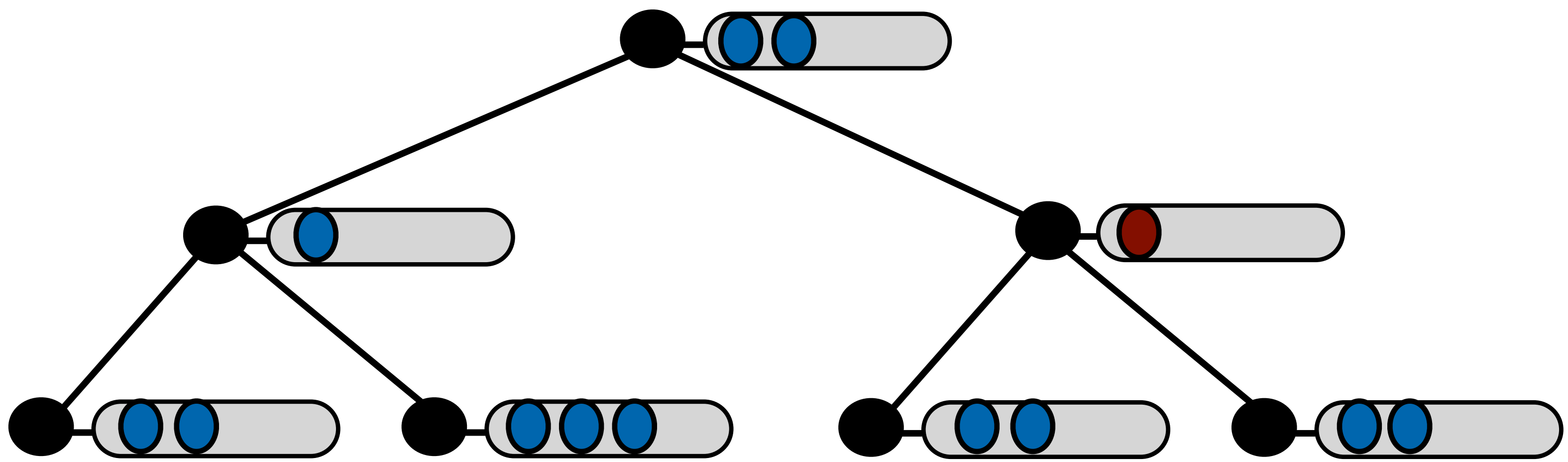
Inserts + deletes:

- 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*



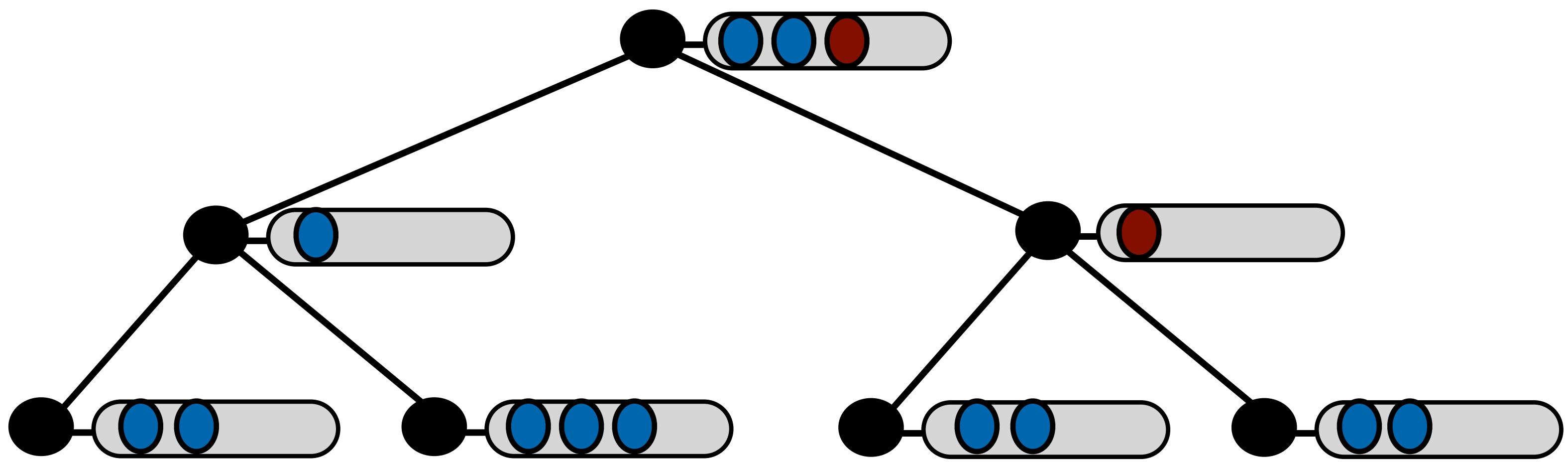
Inserts + deletes:

- 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*



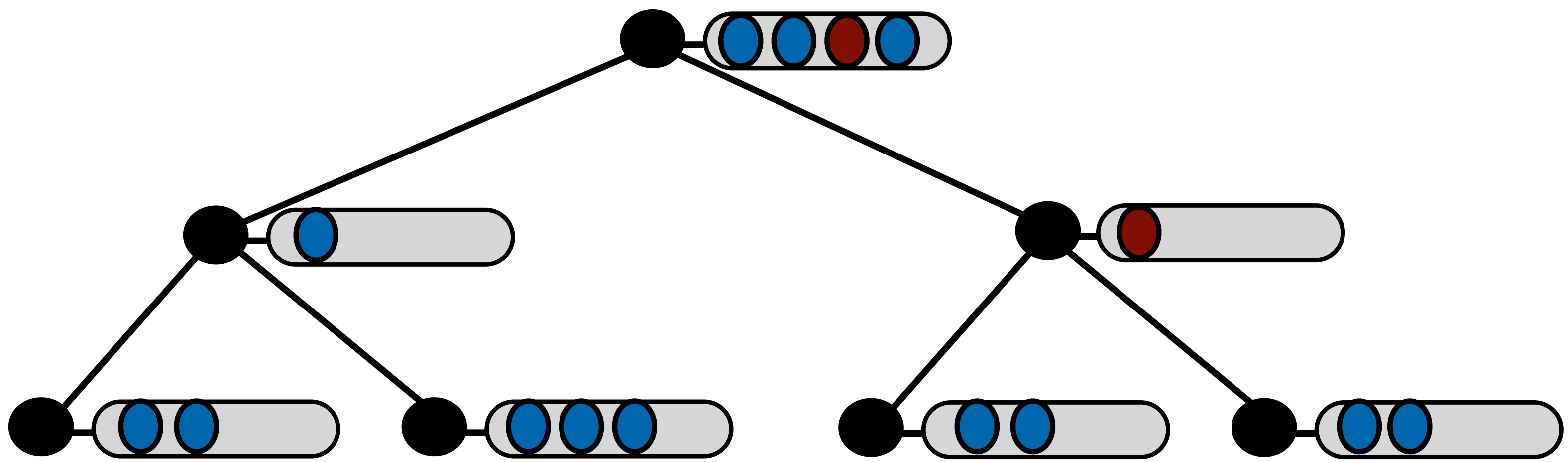
Inserts + deletes:

- 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*



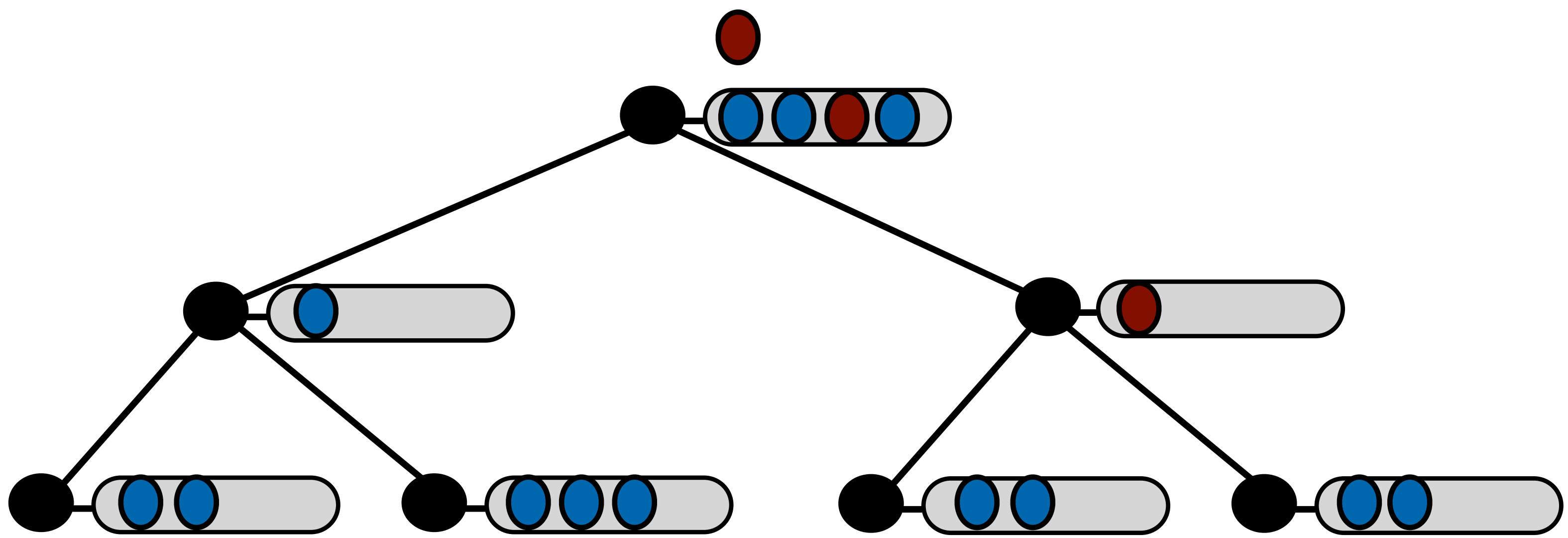
Inserts + deletes:

- 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*



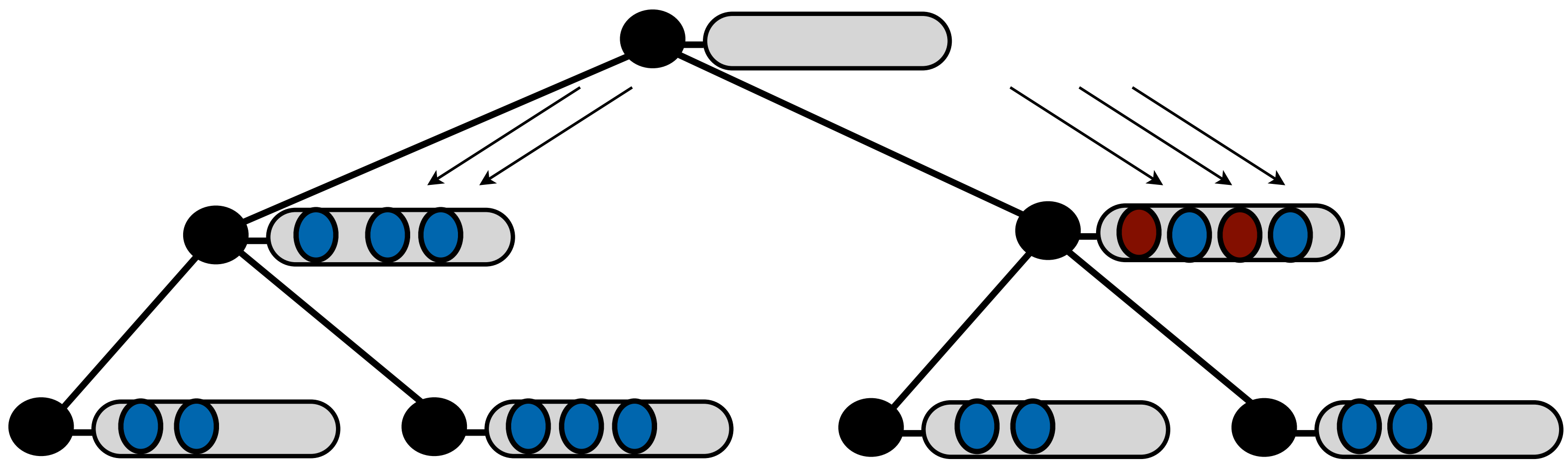
Inserts + deletes:

- 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*



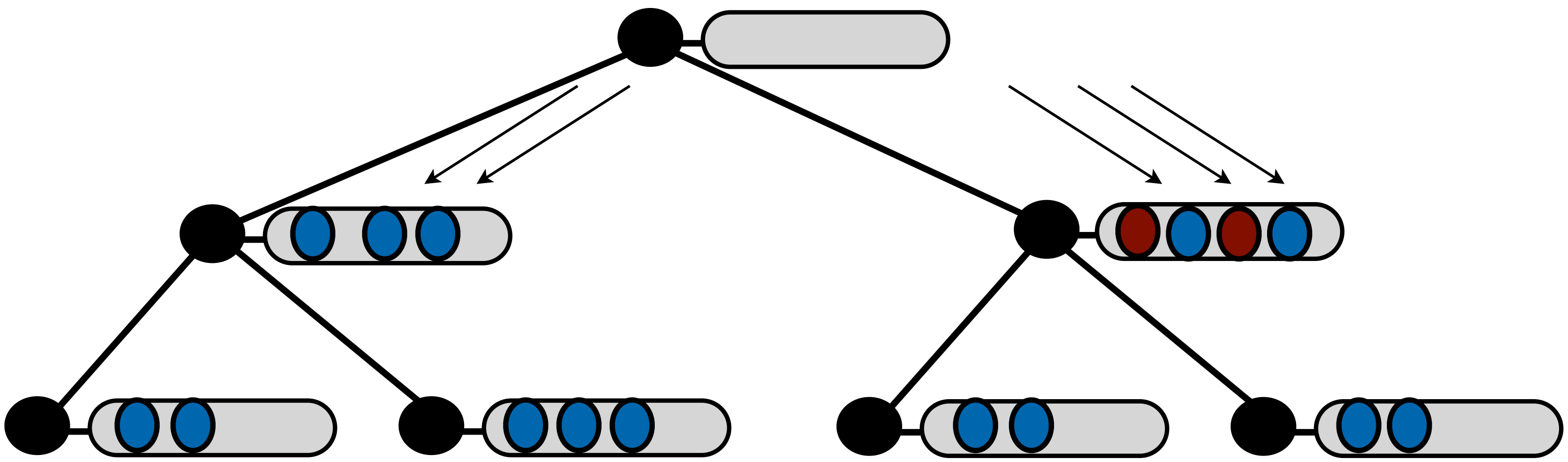
Inserts + deletes:

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

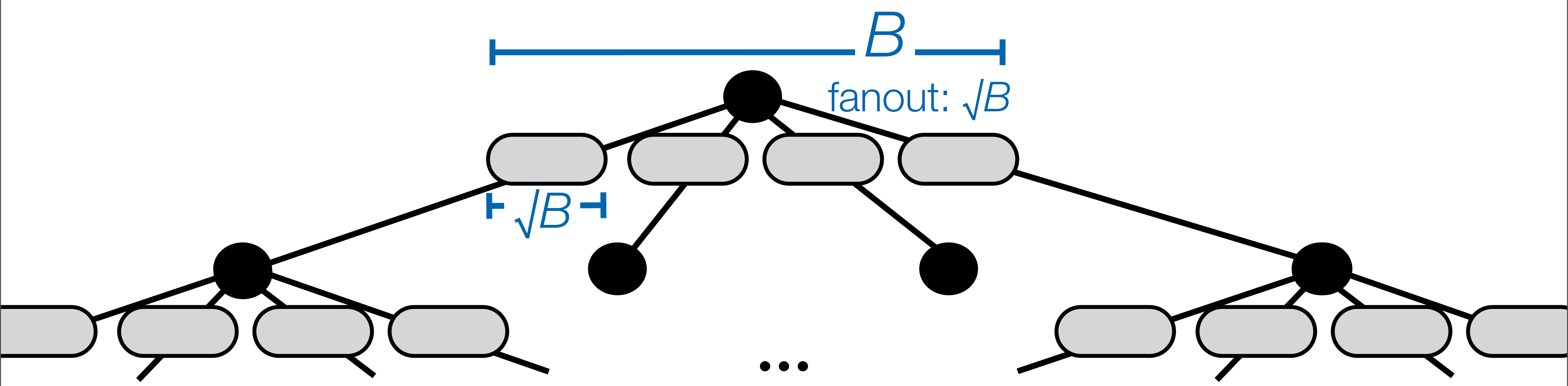
Analysis of writes

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.



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.

Write optimization. ✔ What's missing?

Optimal read-write tradeoff: Easy

Full featured: Hard

- 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

Log Structured Merge Trees

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

Log structured merge trees are write-optimized data structures developed in the 90s.

Over the past 5 years, LSM trees have become popular (for good reason).

Accumulo, Bigtable, bLSM, Cassandra, HBase, Hypertable, LevelDB are LSM trees (or borrow ideas).

http://nosql-database.org lists 122 NoSQL databases. Many of them are LSM trees.

Log Structured Merge Tree

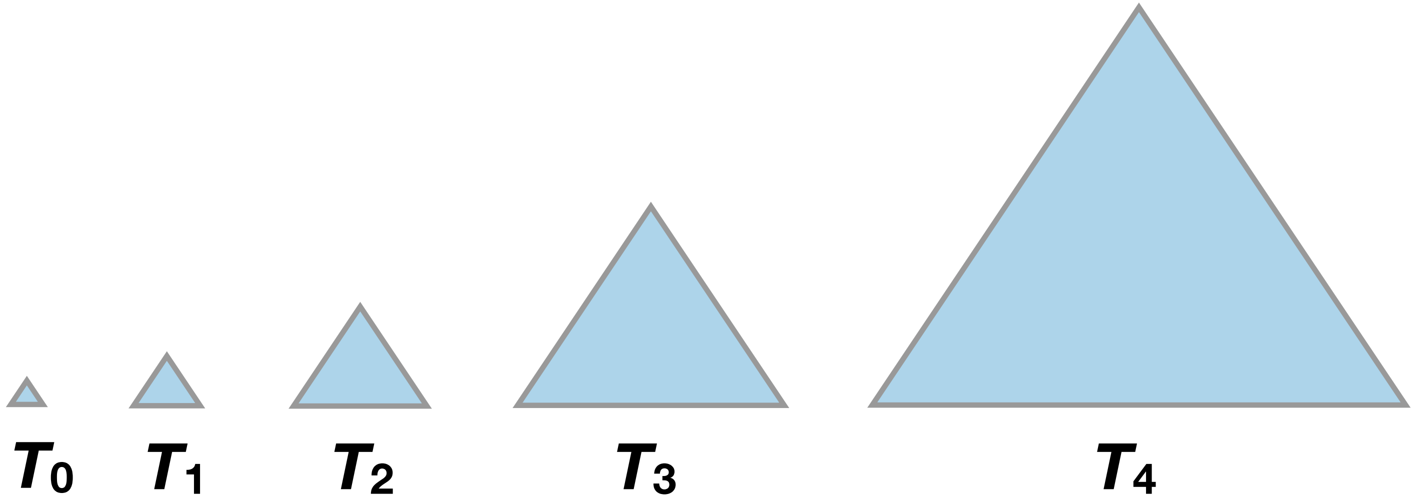
[O'Neil, Cheng, Gawlick, O'Neil 96]

An LSM tree is a cascade of B-trees.

Each tree T_j has a *target size* $|T_j|$.

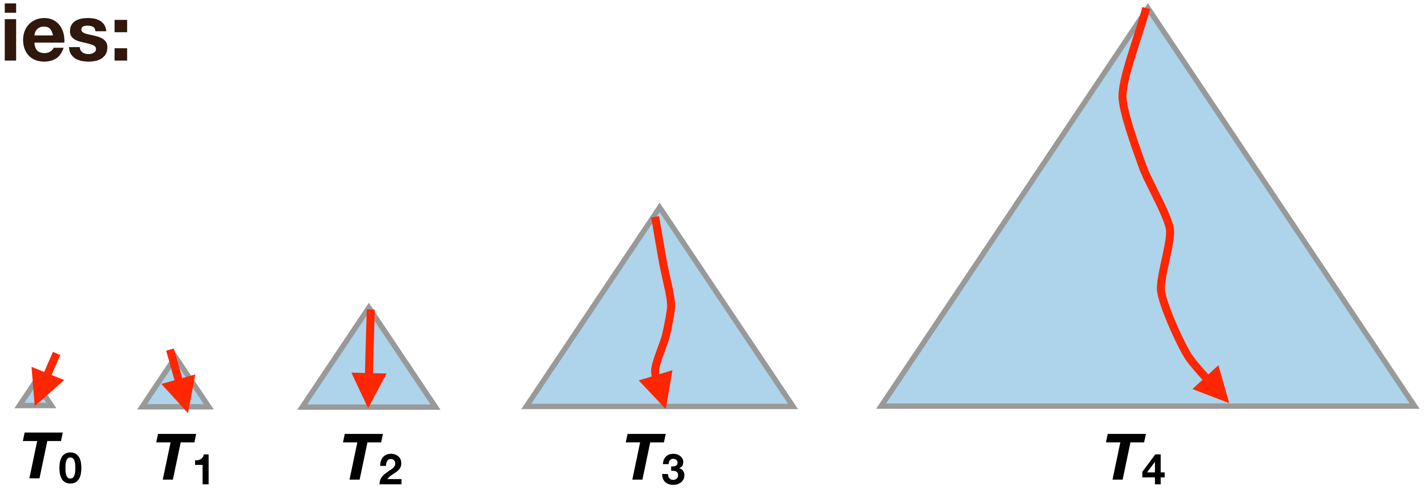
The target sizes are exponentially increasing.

Typically, target size $|T_{j+1}| = 10\,|T_j|$.



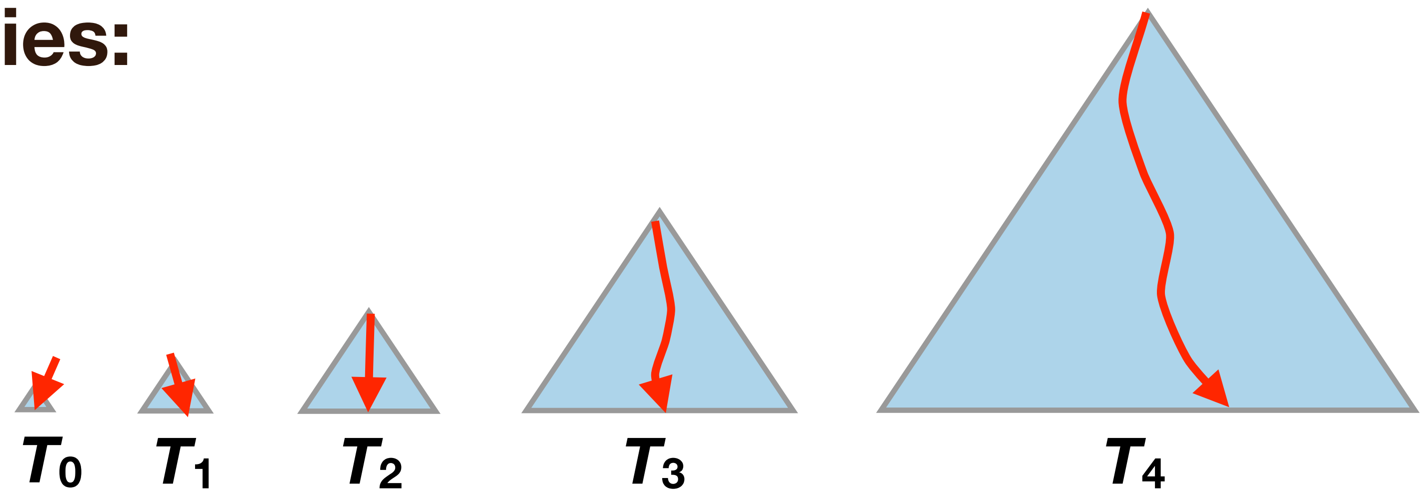
LSM Tree Operations

Point queries:

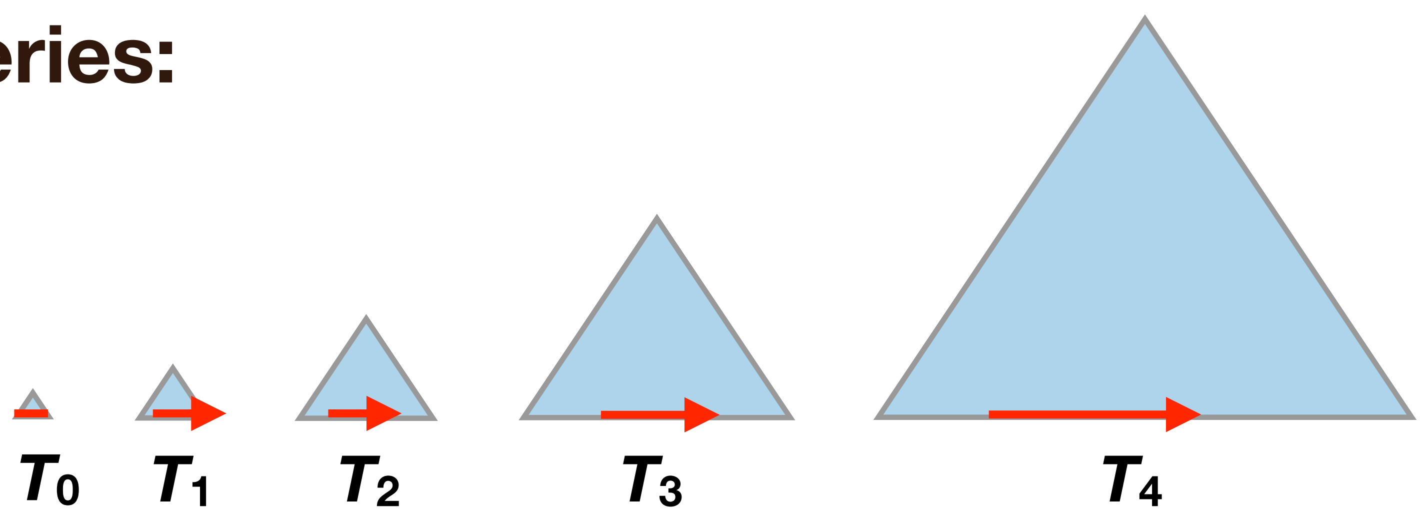


LSM Tree Operations

Point queries:



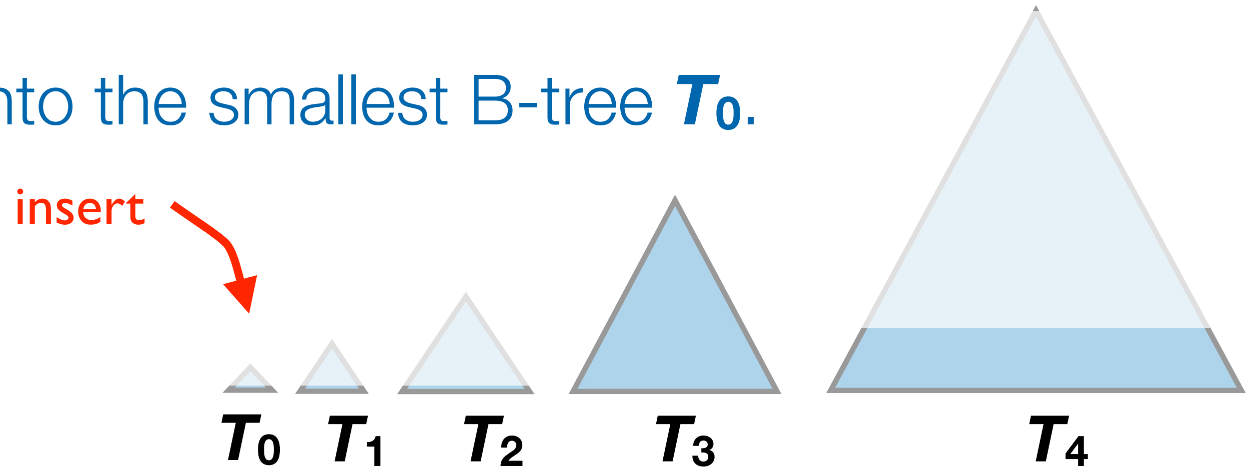
Range queries:



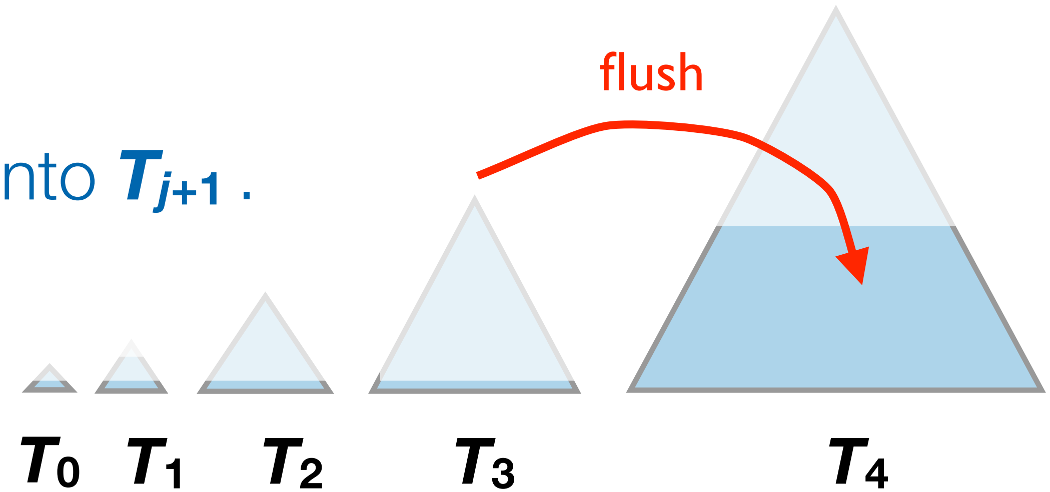
LSM Tree Operations

Insertions:

- Always insert element into the smallest B-tree $\boldsymbol{T_0}$.



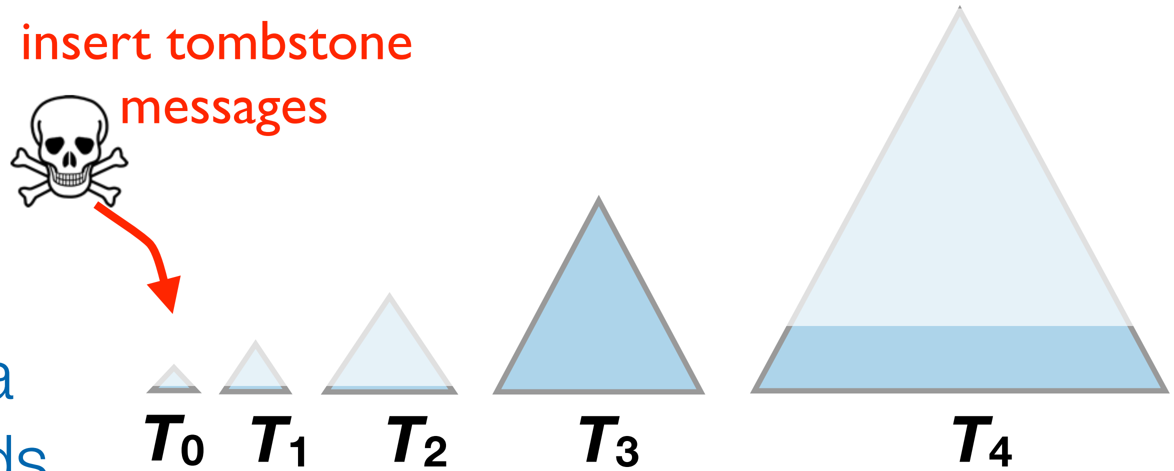
- When a B-tree $\boldsymbol{T_j}$ fills up, flush into $\boldsymbol{T_{j+1}}$.

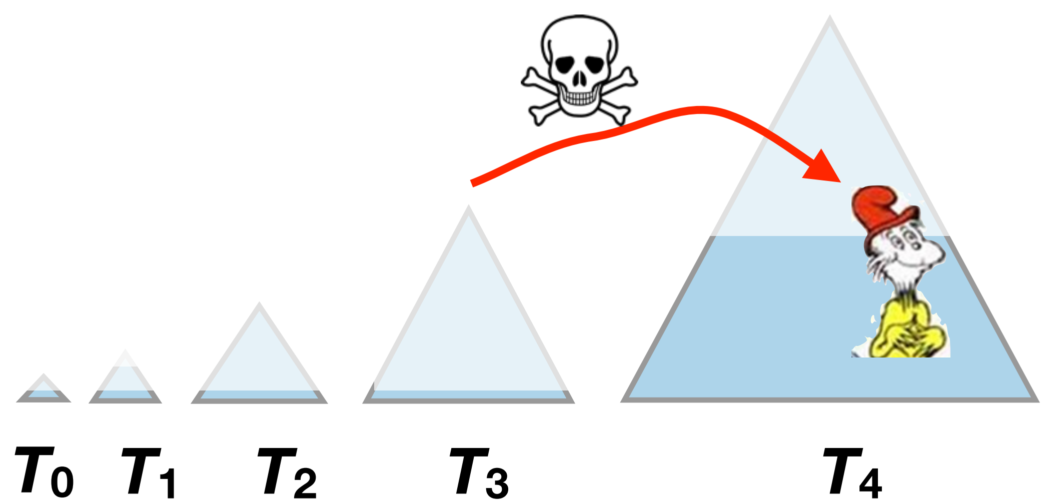


LSM Tree Operations

Deletes are like inserts:

- Instead of deleting an element directly, insert tombstones.
- A tombstone knocks out a "real" element when it lands in the same tree.

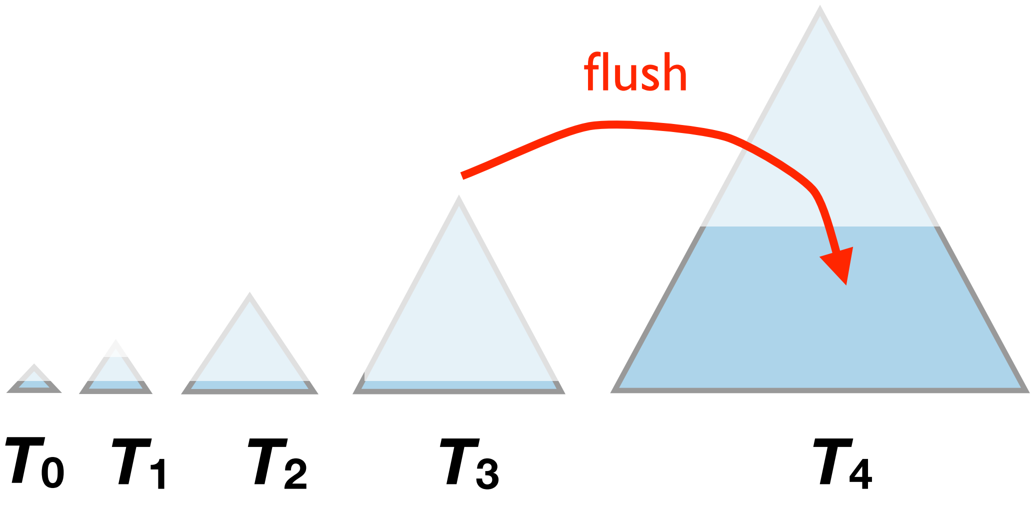




Static-to-Dynamic Transformation

An LSM Tree is an example of a "static-to-dynamic" transformation **[Bentley, Saxe '80]** **.**

- An LSM tree can be built out of ***static B-trees.***
- When T_3 flushes into T_4, T_4 is rebuilt from scratch.



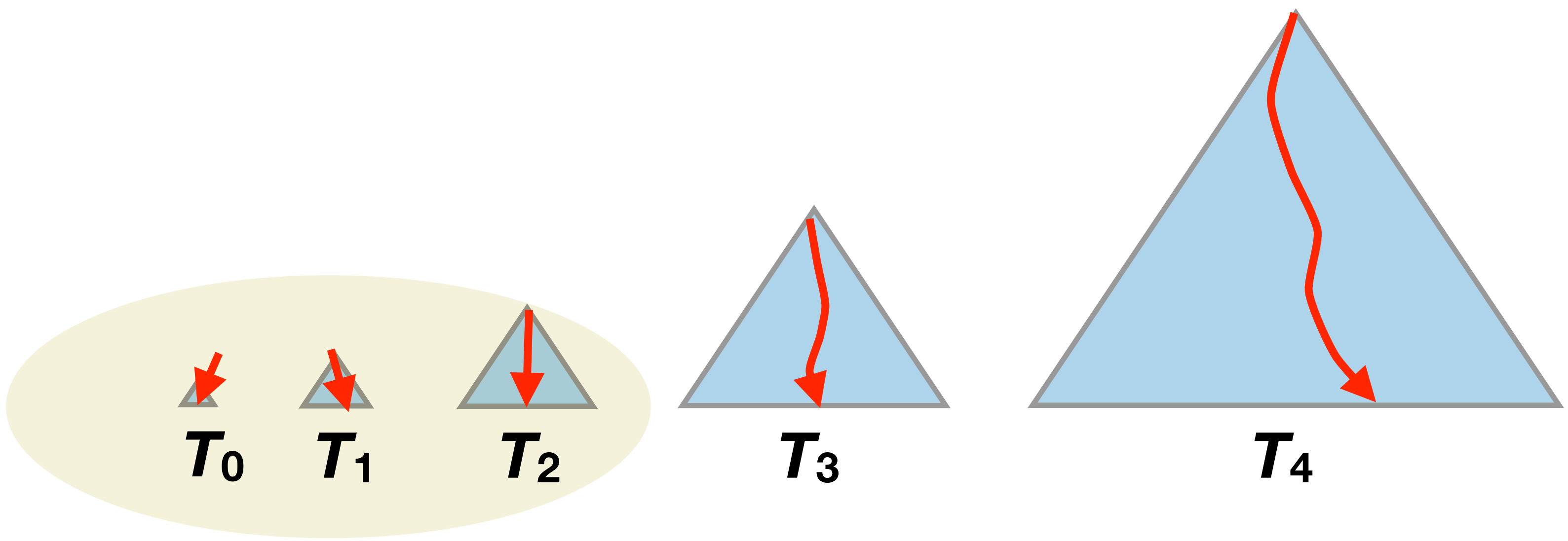
Samples from LSM Tradeoff Curve

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

How to improve LSM-tree point queries?

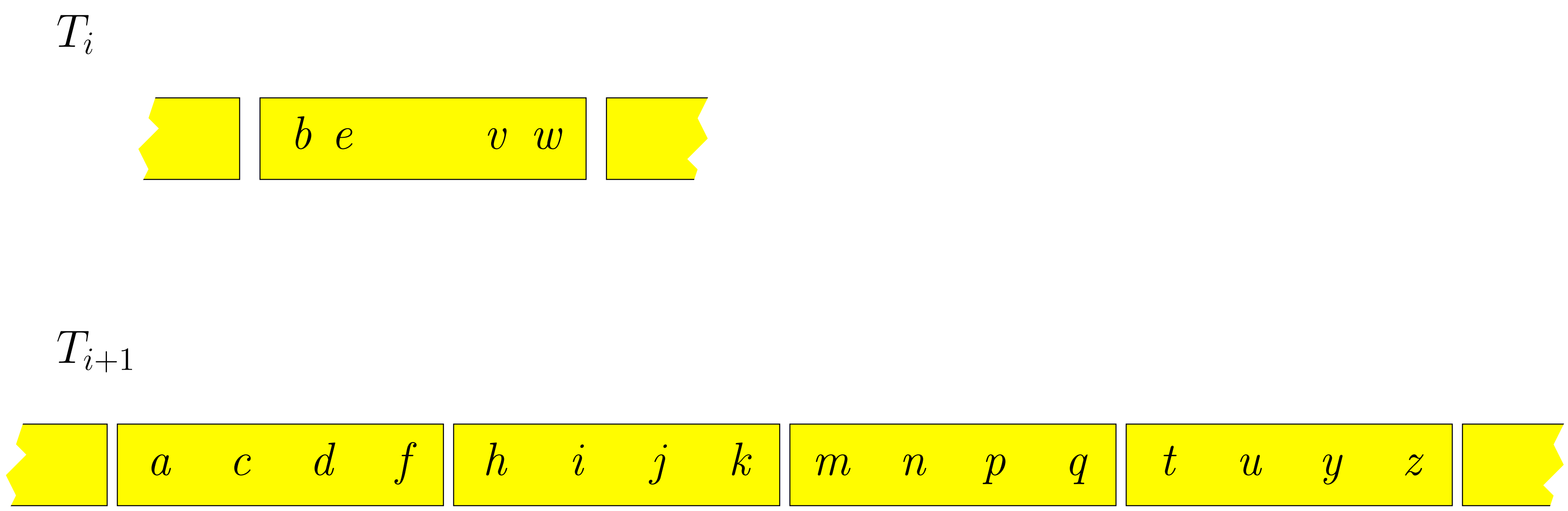
Looking in all those trees is expensive, but can be improved by

- caching,
- Bloom filters, and
- fractional cascading.



Searching one tree helps in the next

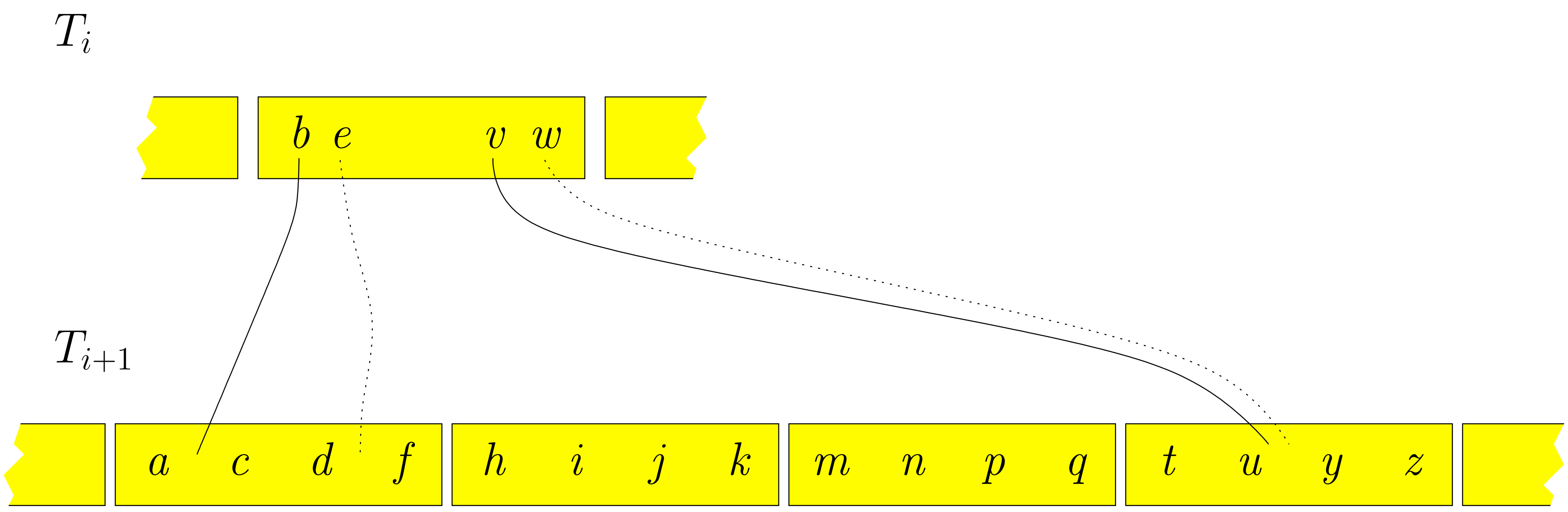
Looking up *c*, in *Ti* we know it's between *b*, and *e*.



Showing only the bottom level of each B-tree.

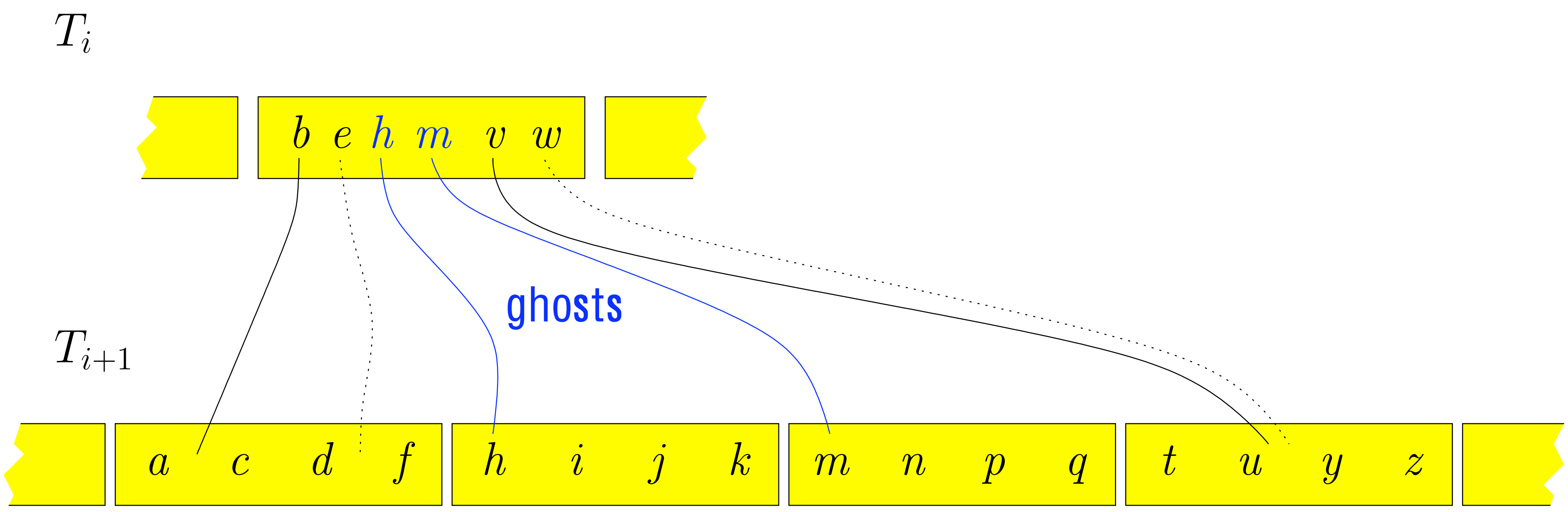
Remove redundant forwarding pointers

We need only one forwarding pointer for each block in the next tree. Remove the redundant ones.



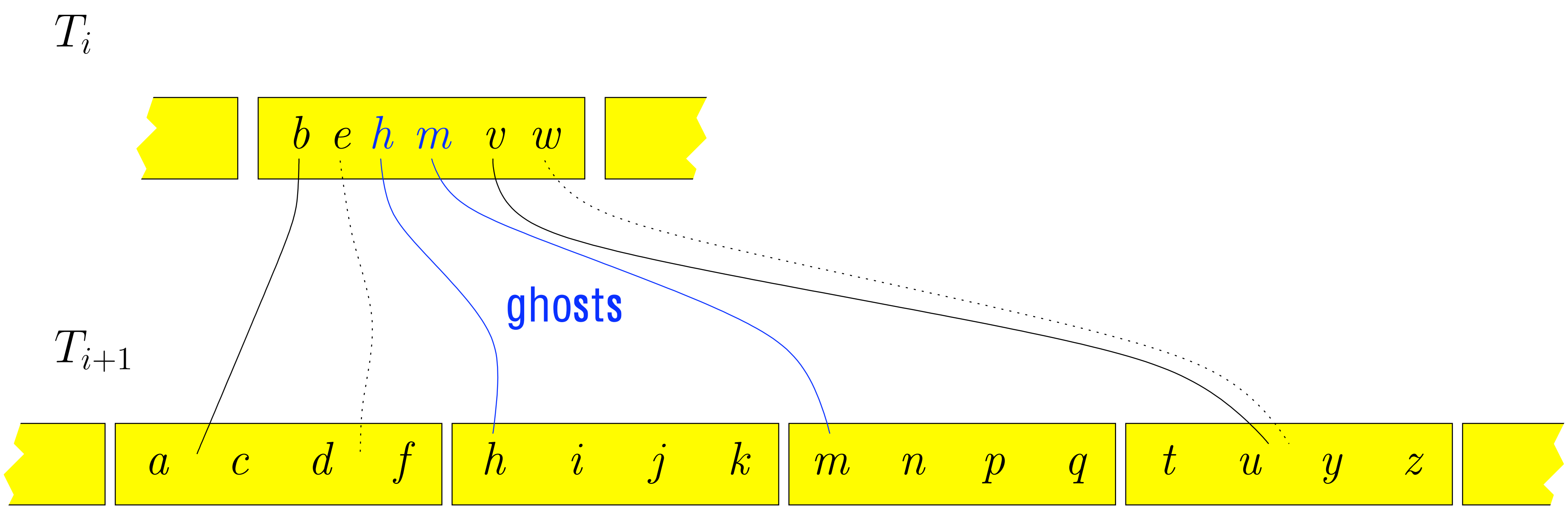
Ghost pointers

We need a forwarding pointer for every block in the next tree, even if there are no corresponding pointers in this tree. Add ghosts.



LSM tree + forward + ghost = fast queries

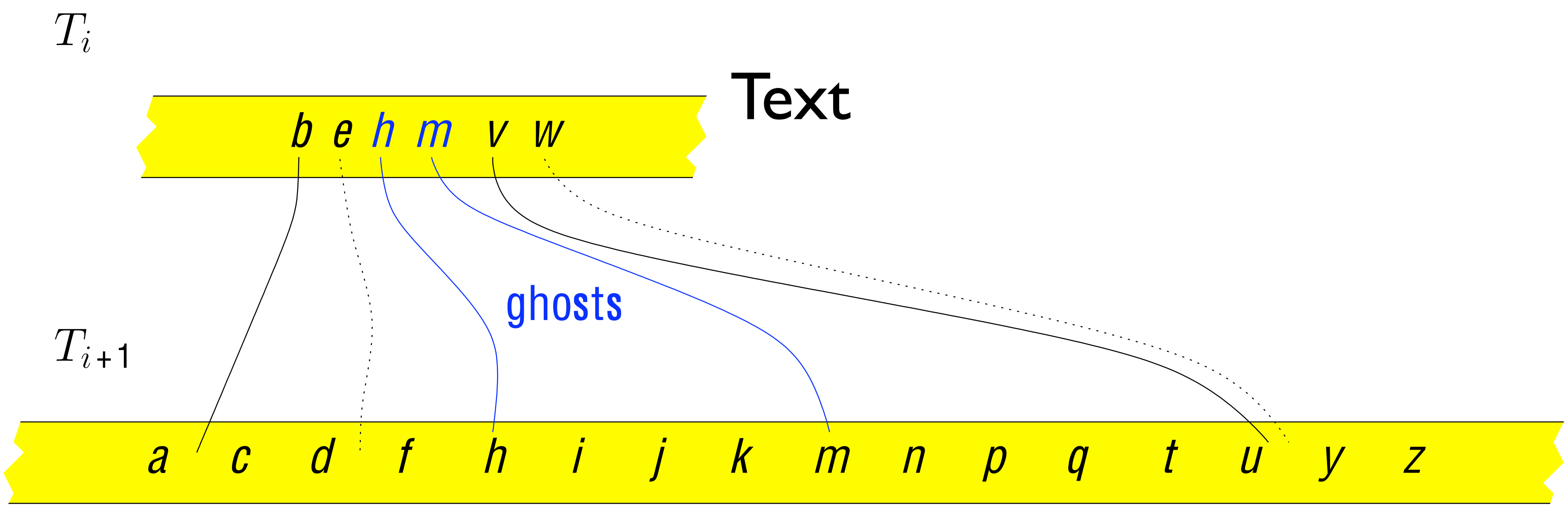
With forward pointers and ghosts, LSM trees require only one I/O per tree, and point queries cost only $O(\log_R N)$**.**



[Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson 07]

LSM tree + forward + ghost = COLA

This data structure no longer uses the internal nodes of the B-trees, and each of the trees can be implemented by an array.



[Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson 07]