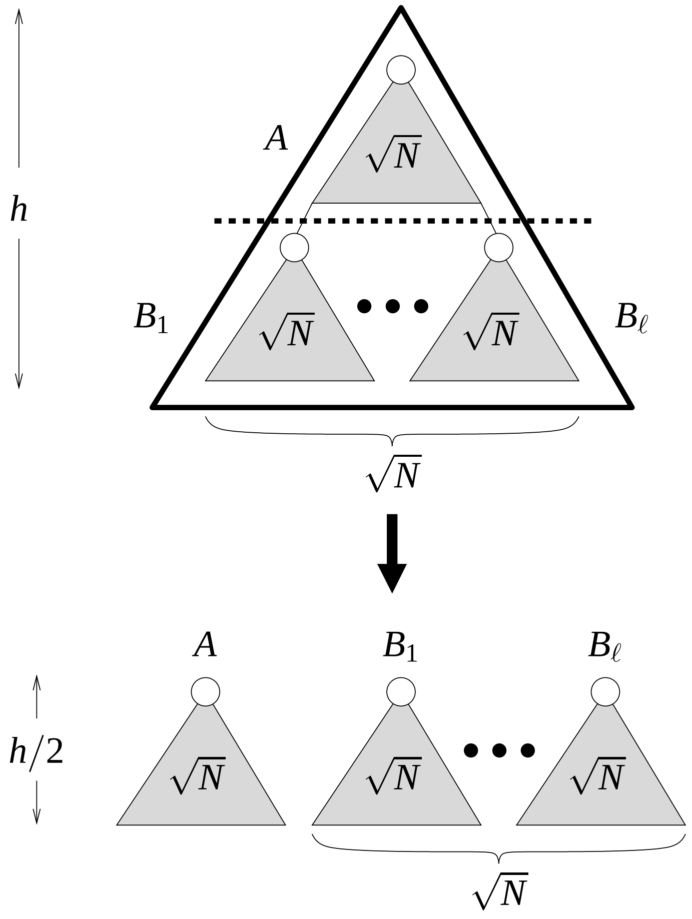
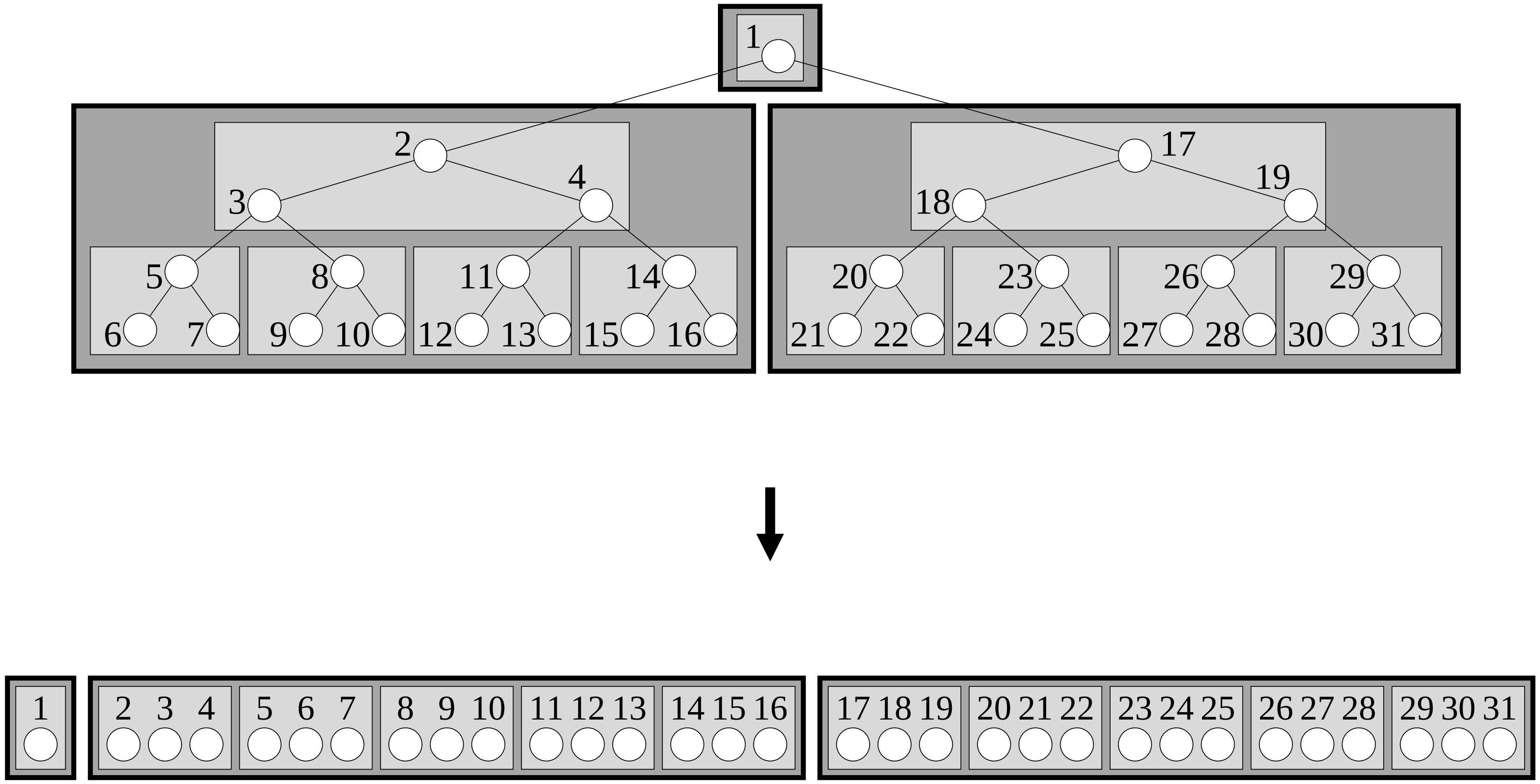
Filesystems for Streaming Databases

Bradley C. Kuszmaul

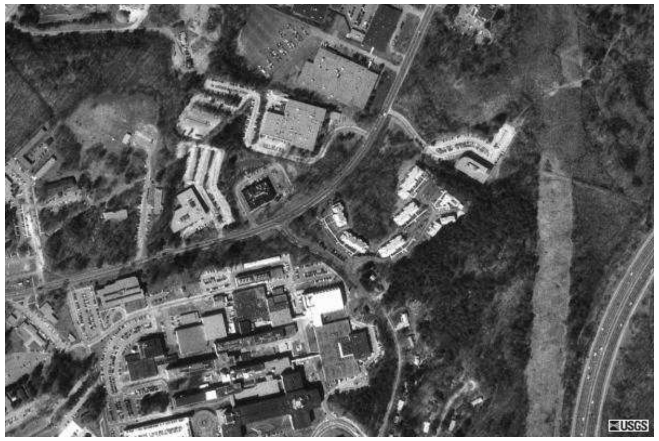
MIT CSAIL





An Important Class of Streaming Applications

- They insert much data, indexed arbitrarily (for example, by geographical coordinate).

“Insert  at key $(71.26585, 42.46053)$.”

- They query the data, asking for all data in a *range* of indices.

“Give me all images within distance 0.1 of $(71.3, 42.0)$.”

- Not all the data is queried.

Performance depends on fast insertions and range queries.

Data Structures for Streaming Applications
or
Streaming B-Trees

Bradley C. Kuszmaul
MIT CSAIL

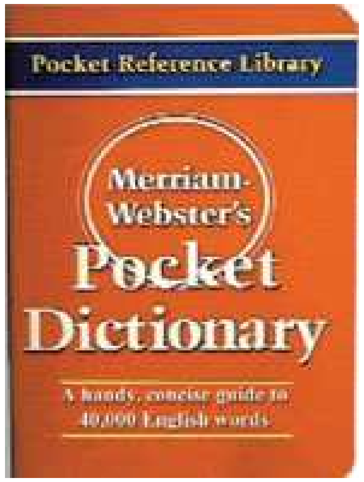
This work represents a collaboration with Michael A. Bender of SUNY Stony Brook and Martin Farach-Colton at Rutgers.

Students: Jeremy Fineman (MIT), Yoni Fogel (Stony Brook), Haodong Hu (Stony Brook), and Jelani Nelson (MIT).

Outline

- Review B-Trees.
- Results.
- The problem with B-Trees.
- How Streaming B-Trees work.
- The future.

Dictionaries Store Key-Value Pairs

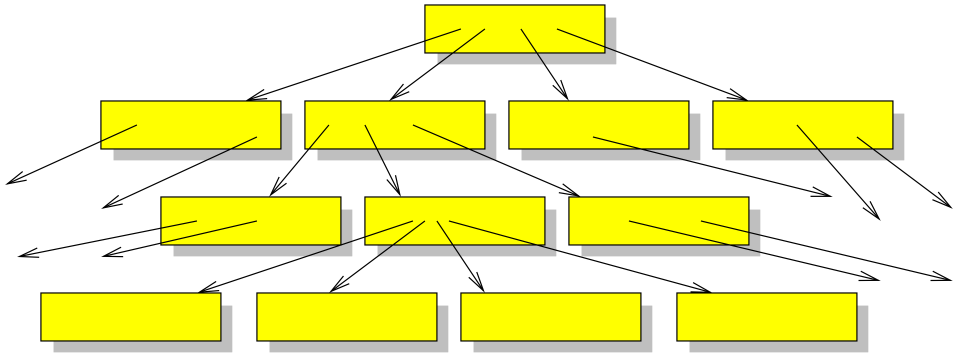


The problem: Store key-value pairs. Operations include

- $\textsc{Insert}(k,v)$: Insert a key-value pair, $k \longrightarrow v$.
- $\textsc{Lookup}(k)$: Find the value associated with a key.
- $\textsc{Next}(k)$: Find the smallest key bigger than k.

A *range query* computes some function on all the key-value pairs whose keys are in a specified range. Range queries can be programmed using $\textsc{Next}$.

B-Trees are Disk-Efficient Dictionaries

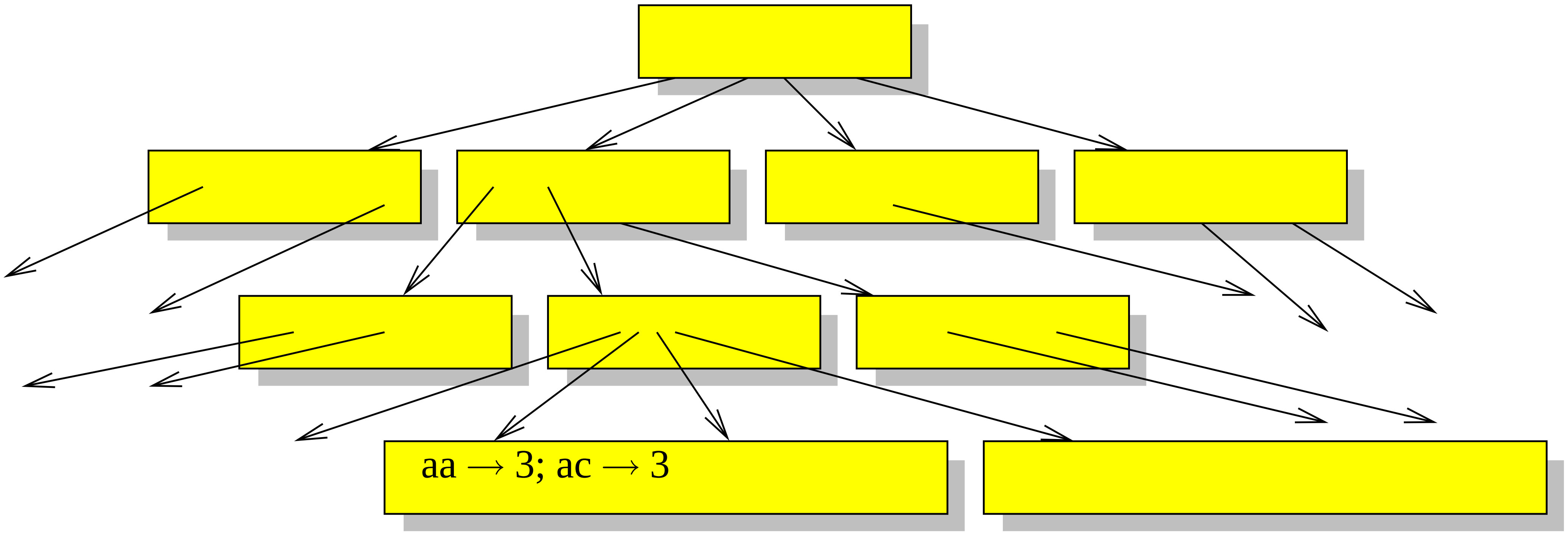


Idea: Whenever you transfer data from disk to memory, try to transfer a whole *block* of useful data.

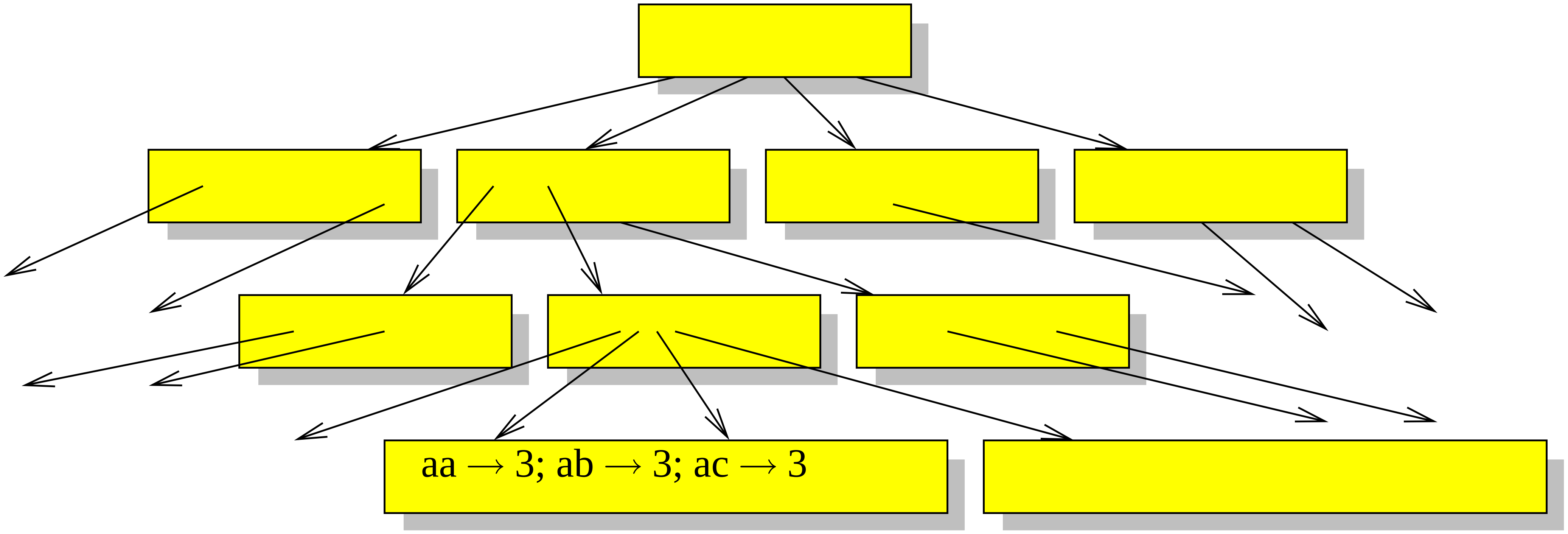
The data structure:

- Data is organized into a tree with blocks of size B.
- For unit-sized data, fanout is $O(B)$.
- The tree depth is $O(\log_B N)$, where N is the number of elements in the dictionary.

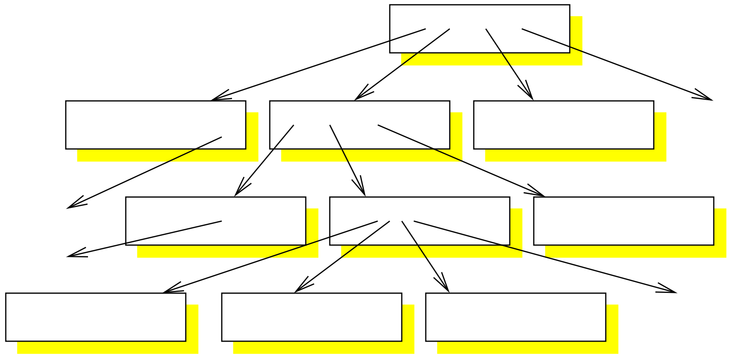
Example INSERT(“ac”, 3) into a B-Tree



becomes



B-Tree Performance



If B is not too big, the time for disk transfers dominate the performance:

- LOOKUP costs $O(\log_B N)$ disk transfers. (Optimal.)
- INSERT costs $O(\log_B N)$ disk transfers. (Not optimal.)
- NEXT costs $O(1/B)$ disk transfers on average, because of caching. (Optimal.)

Outline

- Review B-Trees.
- Results.
- The problem with B-Trees.
- How Streaming B-Trees work.
- The future.

Streaming B-Trees Speed Up INSERT

A B^{ε} tree can, by slowing down LOOKUP by a constant factor ε, speed up insertion by a large factor.

	B-Tree	Streaming B-Tree
LOOKUP	$O(\log_B N)$	$O\left(\frac{\log_B N}{\varepsilon}\right)$
INSERT	$O(\log_B N)$	$O\left(\frac{\log_B N}{\varepsilon B^{\varepsilon}}\right)$
NEXT	$O(1/B)$	$O(1/B)$

For example, if $\varepsilon = 0.5$, and $B = 2^{10}$ data items, then LOOKUP loses a factor of two, and INSERT gains a factor of $\varepsilon B^{\varepsilon} = 256$. Larger blocks gain even more.

We have measured $>$ 250-fold speedups.

Outline

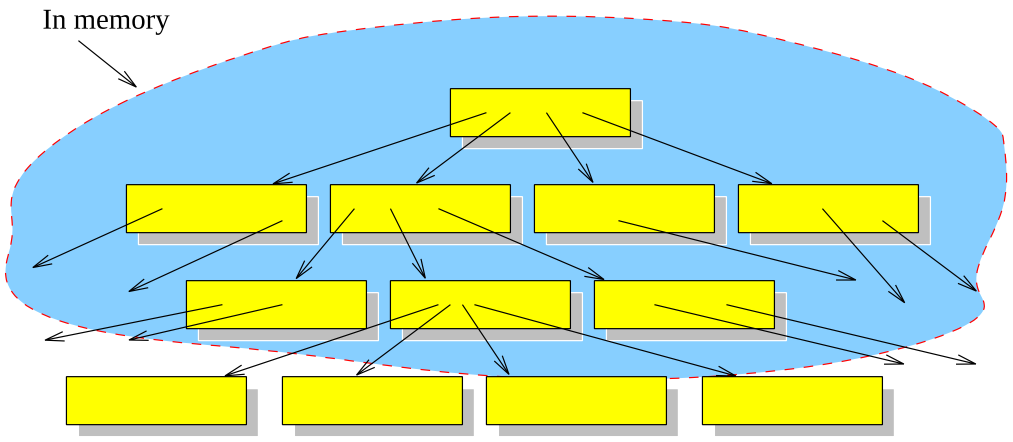
- Review B-Trees.
- Results.
- The problem with B-Trees.
- How Streaming B-Trees work.
- The future.

B-Trees Perform Poorly for Streaming

Consider a random insertion for a database that does not fit in main memory.

A disk I/O is likely to be required to obtain the block in which the pair is to be inserted.

Even if the tree fits in memory, an entire block must eventually be written to disk to insert only a few bytes.



A Back-of-the-Envelope Performance Analysis

B-Tree performance is determined by the disk hardware, and by the workload. Here is an example:

Disk hardware: A high-end 15,000 rpm disk:

- 5ms average seek time.
- 90MB/s bandwidth

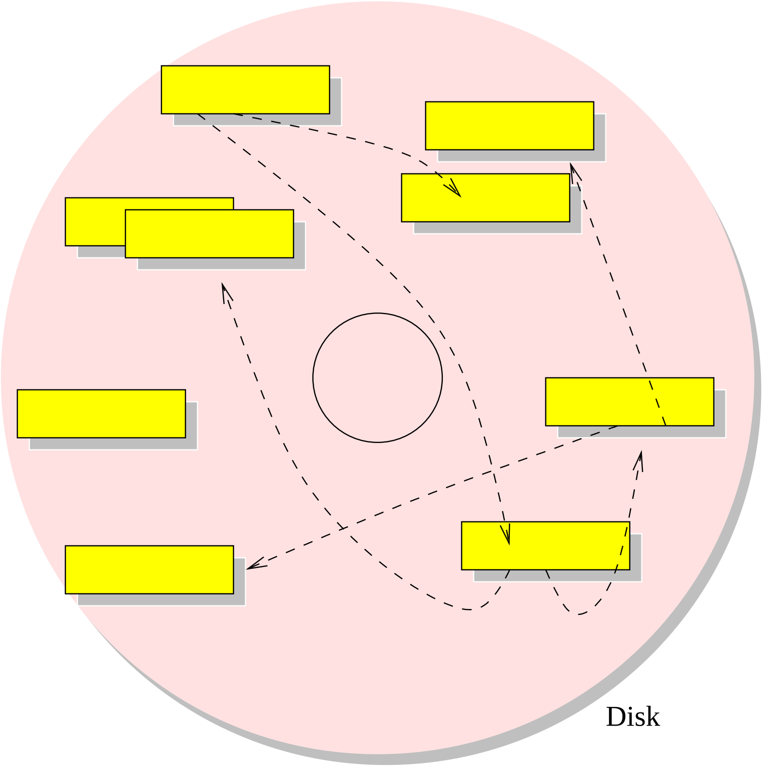
Workload: Random insertion of 16-byte key-value pairs.
Block size is 4KB. Assume one write per INSERT.

Insertion cost:

Seek to the block:	5.00ms
Read the block:	4KB/(90MB/s) = 0.05ms
Total time	5.05ms
Average bandwidth:	$16B/5.05\text{ms} = 3168\text{B/s}$

0.003% of peak disk bandwidth.

B-Tree Hacks Do Not Help Much



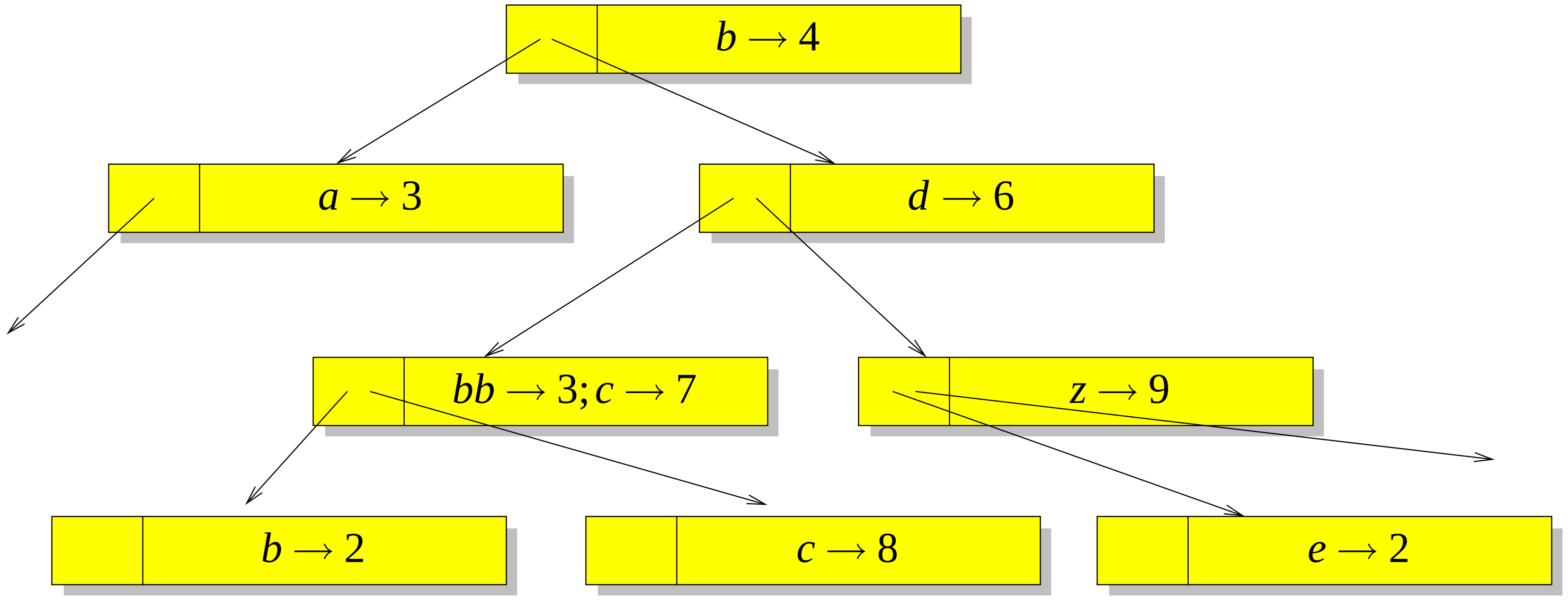
- **Big blocks**: Queries run faster, but big blocks do not help insertions. They also introduce other performance issues.
- **Clustering**: (E.g., cylinder groups, allocation groups, etc.) Range queries run faster, but clustering does not help insertions. Moreover, as the system ages, clusters become fragmented, reducing performance.

Outline

- Review B-Trees.
- Results.
- The problem with B-Trees.
- How Streaming B-Trees work.
- The future.

How the B^{ε} Streaming B-Tree Works

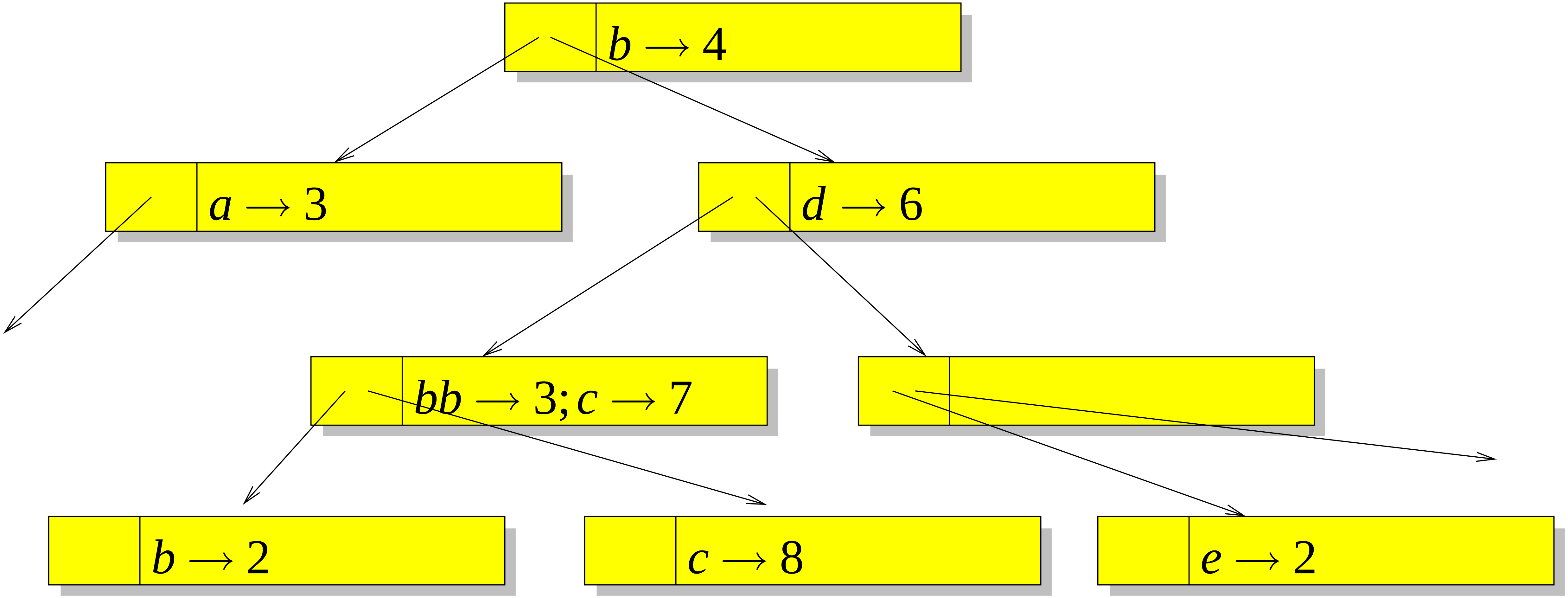
Idea: Put buffers at internal nodes.



To LOOKUP, look in root fufer. If not there, recursively look in the proper subtree. Examples:

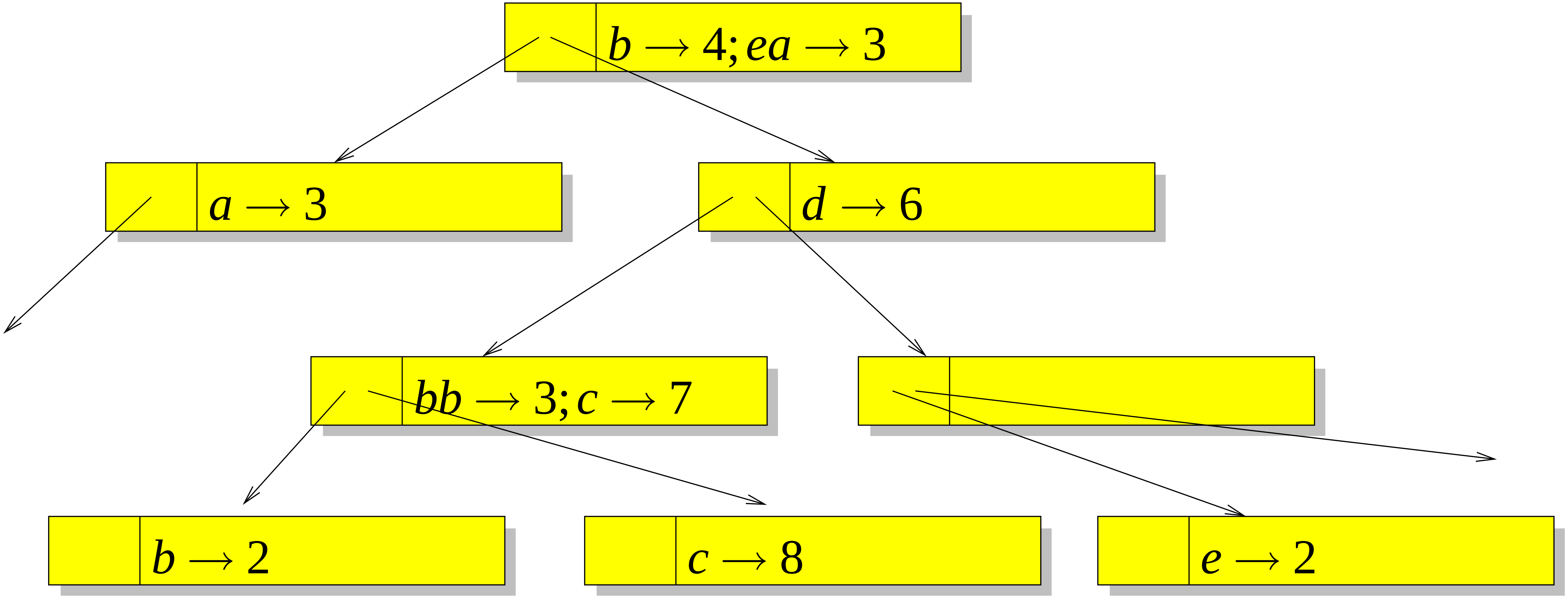
- LOOKUP(“a”): not found in root, found in left child.
- LOOKUP(“e”): must traverse all the way to a leaf.
- LOOKUP(“b”): finds the value at the root, ignoring another value for “b” at a leaf.

INSERT in a B^{ε} Streaming B-Tree



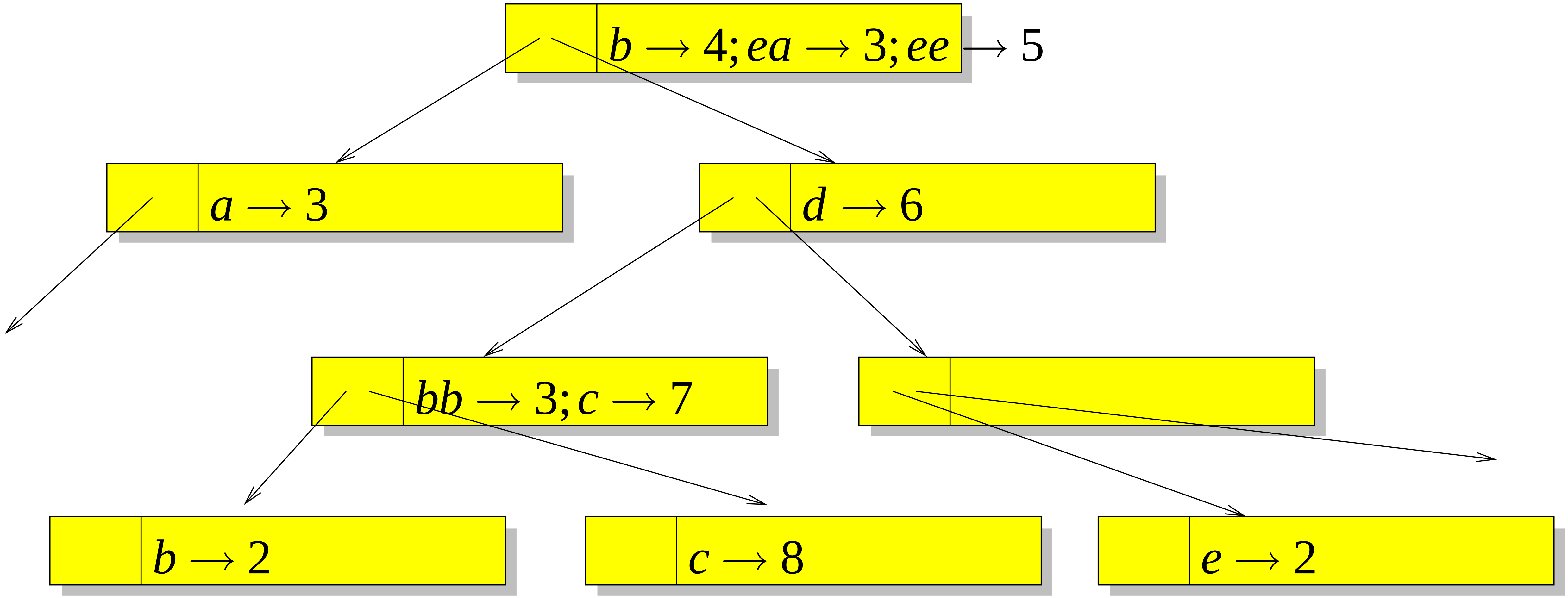
- Store items in the buffer at the root.
- If the root fills up, send many items together to one of the children. Buffer there if possible. Recurse.
- Disk writes always move a big fraction of a block.

INSERT in a B^ε Streaming B-Tree



- Store items in the buffer at the root.
- If the root fills up, send many items together to one of the children. Buffer there if possible. Recurse.
- Disk writes always move a big fraction of a block.

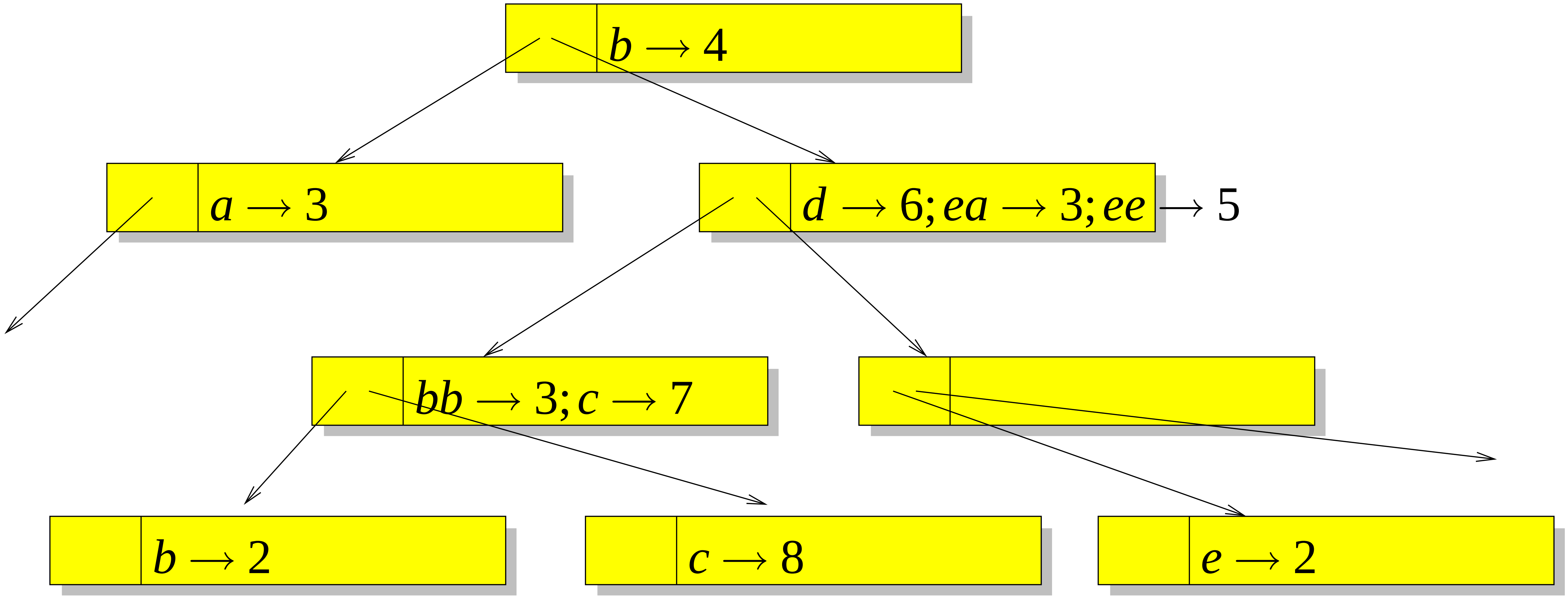
INSERT **in a B^{ε} Streaming B-Tree**



$b \longrightarrow 4; ea \longrightarrow 3; ee \longrightarrow 5$
$a \longrightarrow 3$
$d \longrightarrow 6$
$bb \longrightarrow 3; c \longrightarrow 7$
$b \longrightarrow 2$
$c \longrightarrow 8$
$e \longrightarrow 2$

- Store items in the buffer at the root.
- If the root fills up, send many items together to one of the children. Buffer there if possible. Recurse.
- Disk writes always move a big fraction of a block.

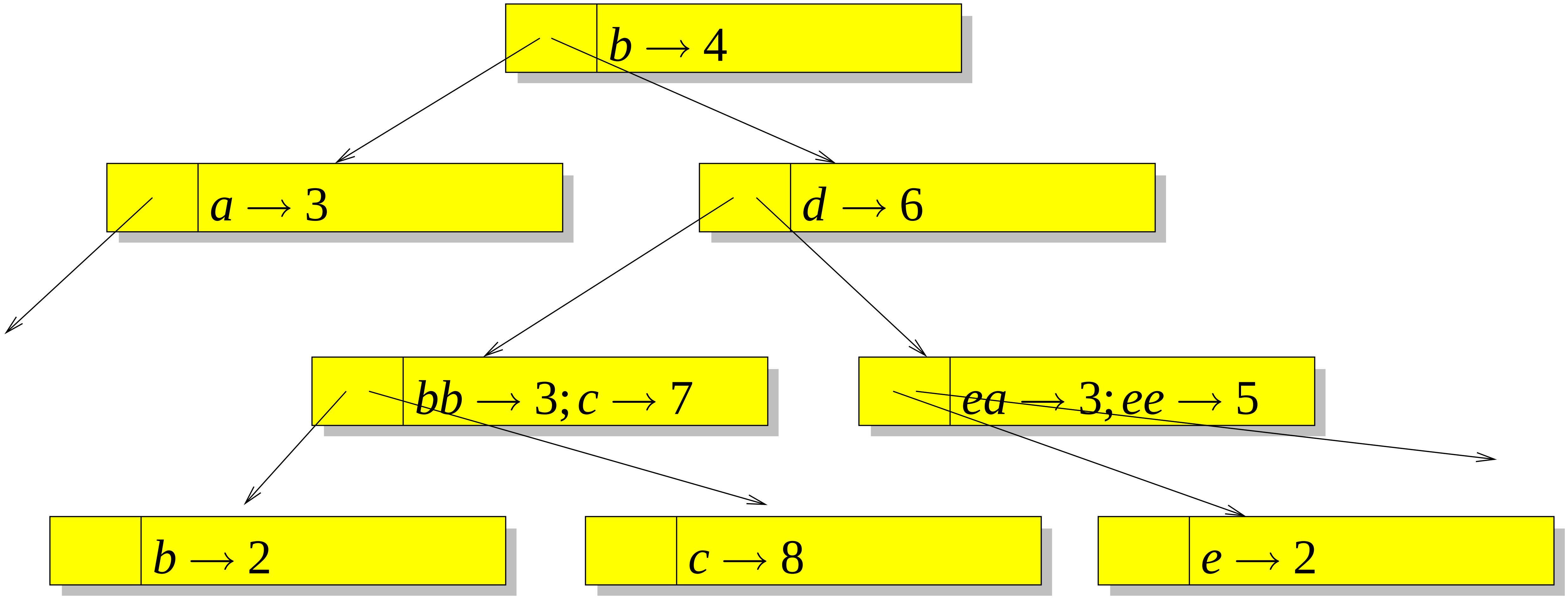
INSERT in a B^{ε} Streaming B-Tree



$b \longrightarrow 4$
$a \longrightarrow 3$
$d \longrightarrow 6; ea \longrightarrow 3; ee \longrightarrow 5$
$bb \longrightarrow 3; c \longrightarrow 7$
$b \longrightarrow 2$
$c \longrightarrow 8$
$e \longrightarrow 2$

- Store items in the buffer at the root.
- If the root fills up, send many items together to one of the children. Buffer there if possible. Recurse.
- Disk writes always move a big fraction of a block.

INSERT in a B^{ε} Streaming B-Tree



- Store items in the buffer at the root.
- If the root fills up, send many items together to one of the children. Buffer there if possible. Recurse.
- Disk writes always move a big fraction of a block.

Streaming B-Tree Performance Numbers

In Spring 2006, Jelani Nelson implemented a streaming B-tree with $B = 4096$ and fixed-size key-value pairs.

Achieved 17-fold speedup on INSERT, with a slowdown of about 3-fold for LOOKUP.

We figured that larger blocks would help...

Streaming B-Tree Performance Numbers (large *B*)

Sequential INSERTs, followed by random INSERTs.

	B-Tree		B^{ε} Tree		Speedup
$2 \cdot 10^7$ serial	418s	47802/s	647s	30911/s	0.6
$50,000$ random	375s	134/s	1.4s	34658/s	258
CPU Load		11%		63%	

- B^{ε} streaming B-Tree has block size$=$ 1MB, degree $= 16$.
- For the B-Tree, used Berkeley DB (which employs special-case code for serial INSERT), block size = 4KB.
- Key-value pairs of varying size, around 16-bytes.
- Haven't yet measured query cost.

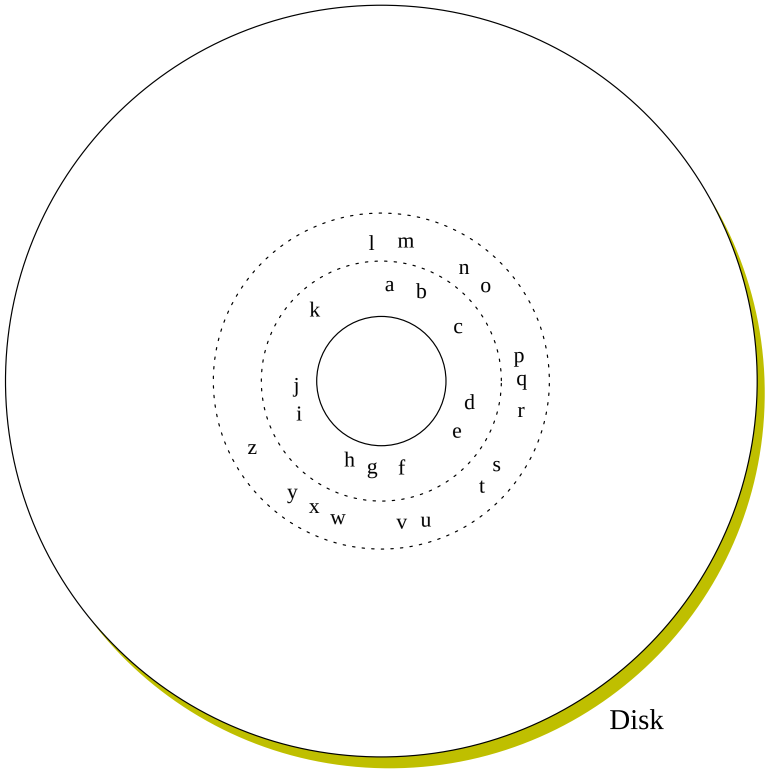
Outline

- Review B-Trees.
- Results.
- The problem with B-Trees.
- How Streaming B-Trees work.
- The future.

Cache-Oblivious Data Structures

We are also investigating the technology of ***cache-oblivious*** data structures for streaming applications, which provide several advantages:

- **Passive self-tuning:** no ***"voodoo"*** parameters. ("How many cylinders belong in an allocation group?" "What is the right block size?")
- **Provable guarantees** on locality (e.g., most data is stored in sorted order on disk, thereby guaranteeing good range query performance).
- No file-system **aging** problems.



Preliminary: The cache-oblivious streaming B-tree implemented by Yoni Fogol may be another 3x faster.

Applications of Streaming B-Trees

- File systems: In the file-system community, one of the "grand challenges" is to achieve $30,000$ file creations per second, and to handle a trillion files.

  Streaming B-trees on a single PC with one disk come near that performance.

  We developed an I/O-only version of the HPCS SSCA#3 benchmark (available from `highproductivity.org`).

- Databases: Many database applications have trouble loading data fast enough. Achieving 800 INSERT/s in MySQL is considered a big deal.

We are working to create both a file system and a database based on streaming B-trees. We're looking for "customers".