

# Multidimensional & String Indexes for Streaming Data

**Michael A. Bender**  
Stony Brook  
Tokutek, Inc.

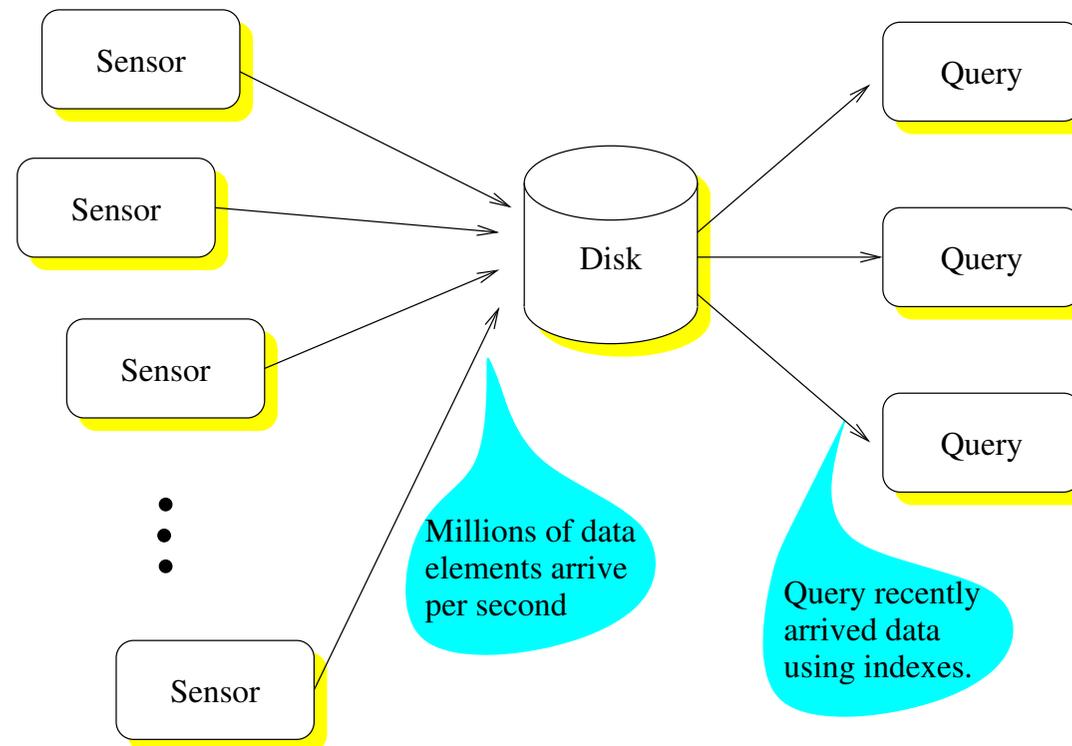
**Martin Farach-Colton**  
Rutgers  
Tokutek, Inc.

**Bradley C. Kuszmaul**  
MIT  
Tokutek, Inc

**Charles E. Leiserson**  
MIT

# Multidimensional and String Indexes for **Streaming Data**

## Millions of data elements arrive per sec from sensors

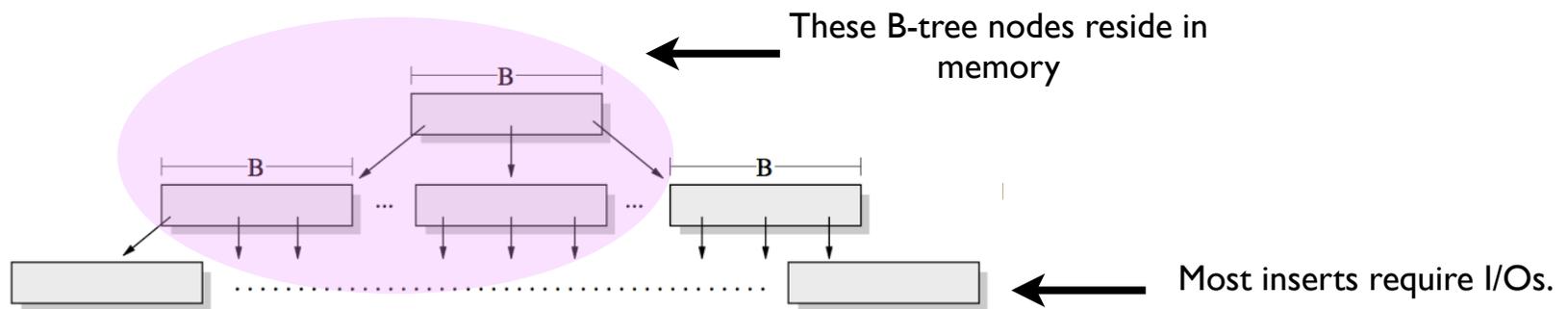


- Sensors may measure mouse clicks on a website, report network attacks, count point-of-sales data, make temperature readings, etc.

# Multidimensional and String Indexes for Streaming Data

**Because we want to query using indexes, we want B-tree-like functionality, but better performance.**

- B-trees support ~100 inserts per sec. per disk in worst case.

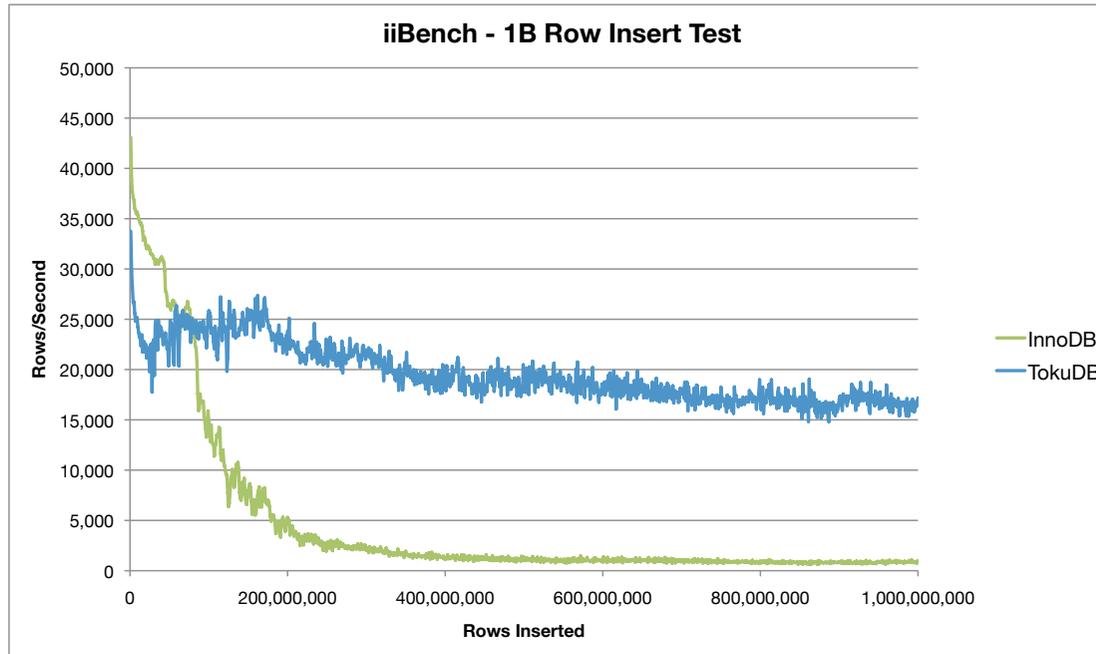


- We would need many tens of thousands of disk to index the data stream using B-trees.

# Multidimensional and String **Indexes** for Streaming Data

We studied this problem previously for HECURA.

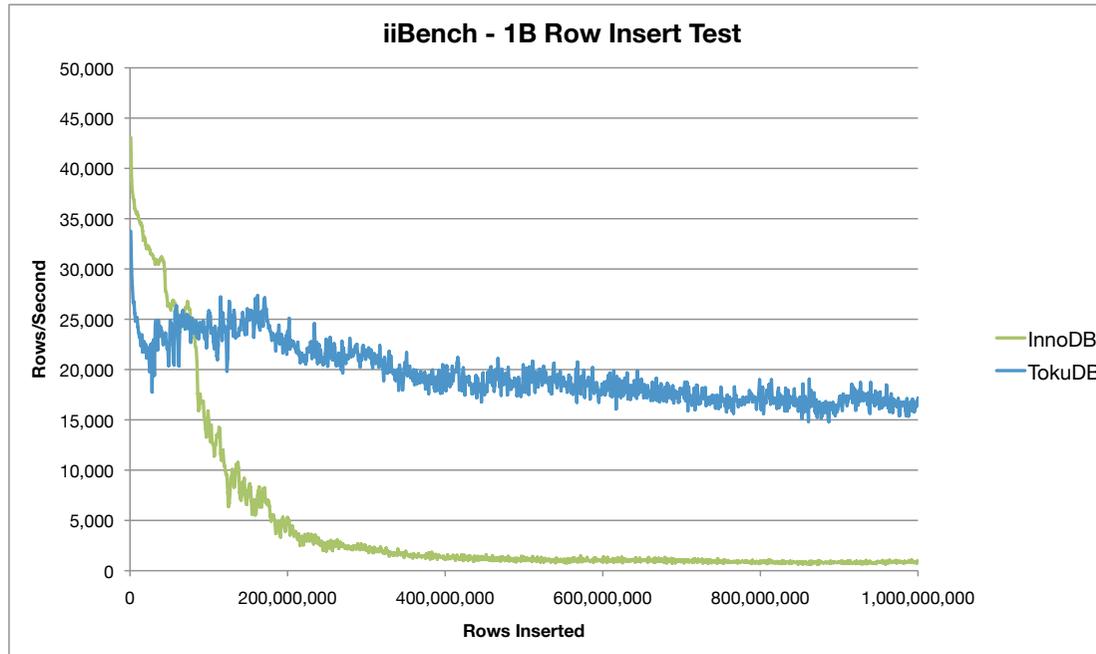
- The **cache-oblivious streaming B-tree** indexes high-entropy (e.g., random) data 10x-100x faster than a B-tree.
- Asymptotically better:  $O((\log_B N)/B)$  I/Os vs.  $O(\log_B N)$  per insert.



# Multidimensional and String **Indexes** for Streaming Data

We studied this problem previously for HECURA.

- The **cache-oblivious streaming B-tree** indexes high-entropy (e.g., random) data 10x-100x faster than a B-tree.
- Asymptotically better:  $O((\log_B N)/B)$  I/Os vs.  $O(\log_B N)$  per insert.

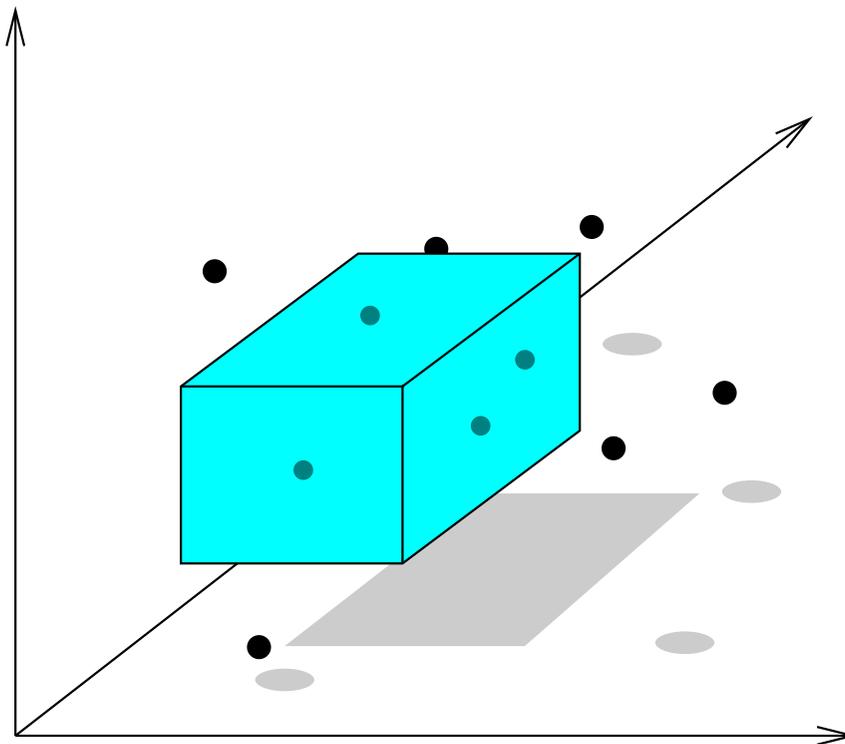


- We transferred the technology to **Tokutek, Inc.**.  
Used in **TokuDB** storage engine for **MySQL**.

# Multidimensional and String Indexes for Streaming Data

## Multidimensional data

- Geometric data, sparse matrices, OLAP cube, MDX, ...



# Multidimensional and **String** Indexes for Streaming Data

## Strings

- File names, URLs, SHA hashes, query strings, webpages, DNA, etc.

## The *string B-tree* indexes strings on external storage

[Ferragina, Grossi 98] [Bender, Farach-Colton, Kuszmaul 06]. [Brodal, Fagerberg 06]

- Impressive queries,  $O(|L| + \log_B N)$  I/Os per search/insert.
- But inserts are slow as B-trees.



## Two aspects to strings:

- Strings are keys having variable sizes.
- Even when keys have same size, string B-trees deliver performance gains.

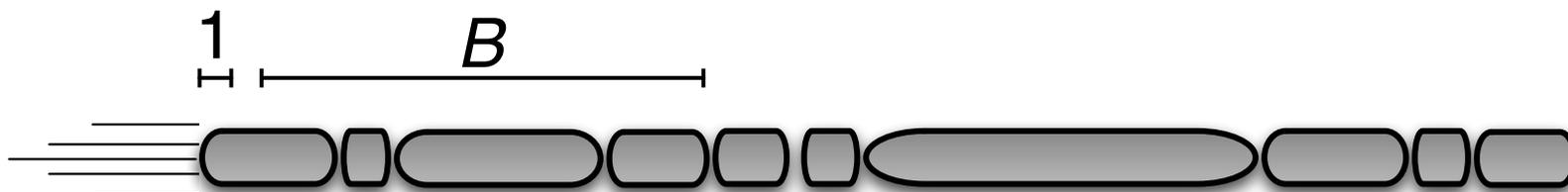
## **Can we build a string streaming B-tree?**

(can we make is cache oblivious, i.e., platform independent?)



## Can we build a string streaming B-tree?

(can we make is cache oblivious, i.e., platform independent?)

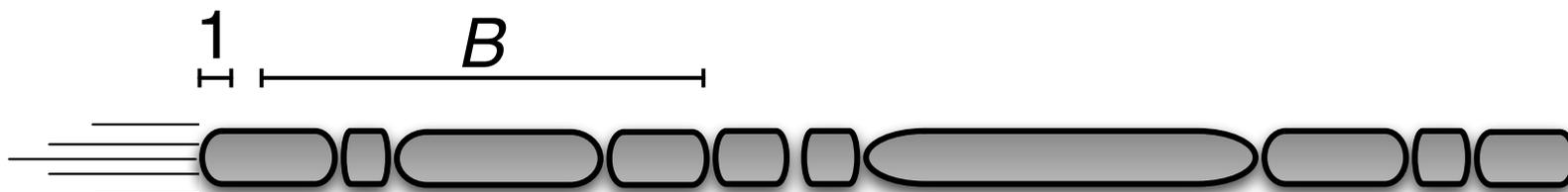


## Interesting case for streaming data:

- strings are smaller than the natural block size  $B$  of the disk but larger than unit size.
- If strings are large, insert cost is dwarfed by cost to scan key

## Can we build a string streaming B-tree?

(can we make is cache oblivious, i.e., platform independent?)



## Interesting case for streaming data:

- strings are smaller than the natural block size  $B$  of the disk but larger than unit size.
- If strings are large, insert cost is dwarfed by cost to scan key.

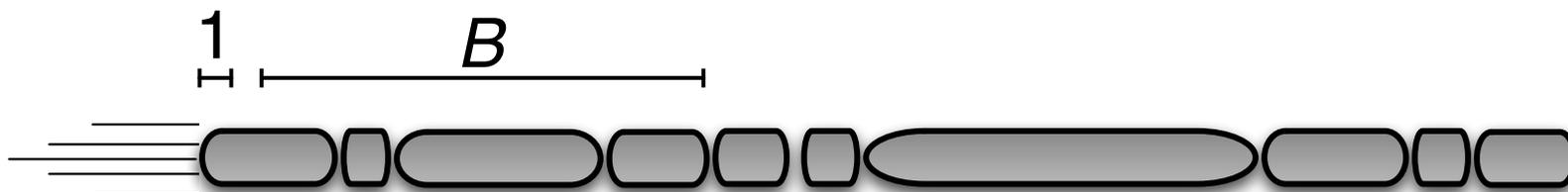
## We have some components:

- cache-oblivious streaming B-tree [Bender, Farach-Colton, Fineman, Fogel, Kuzmaul, Nelson SPAA 07]
- cache-oblivious string B-tree [Bender, Farach-Colton, Kuzmaul PODS 06].
- B-tree with different-size keys [Bender, Kuzmaul PODS 10].

# Multidimensional and **String Indexes for Streaming Data**

## Can we build a string streaming B-tree?

(can we make is cache oblivious, i.e., platform independent?)



## Interesting case for streaming data:

- strings are smaller than the natural block size  $B$  of the disk but larger than unit size.
- If strings are large, insert cost is dwarfed by cost to scan key

We have some components:

- cache-oblivious streaming B-tree
- cache-oblivious string B-tree
- **B-tree with different-size keys**

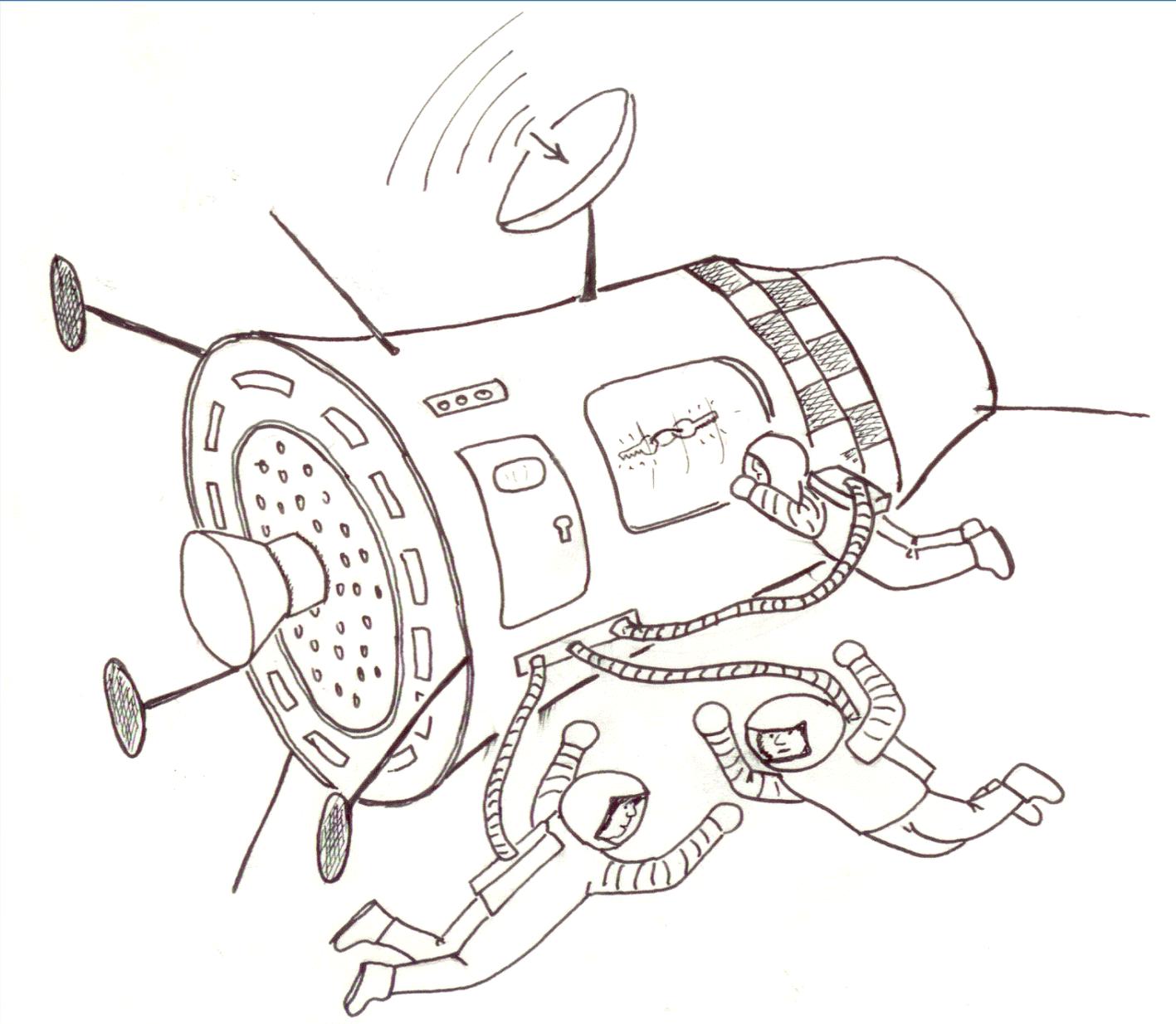
This is what I'm going to talk about.

[Bender, Kuszmaul PODS 10].

# Difficulty of Key Search (with Different-Size Keys)

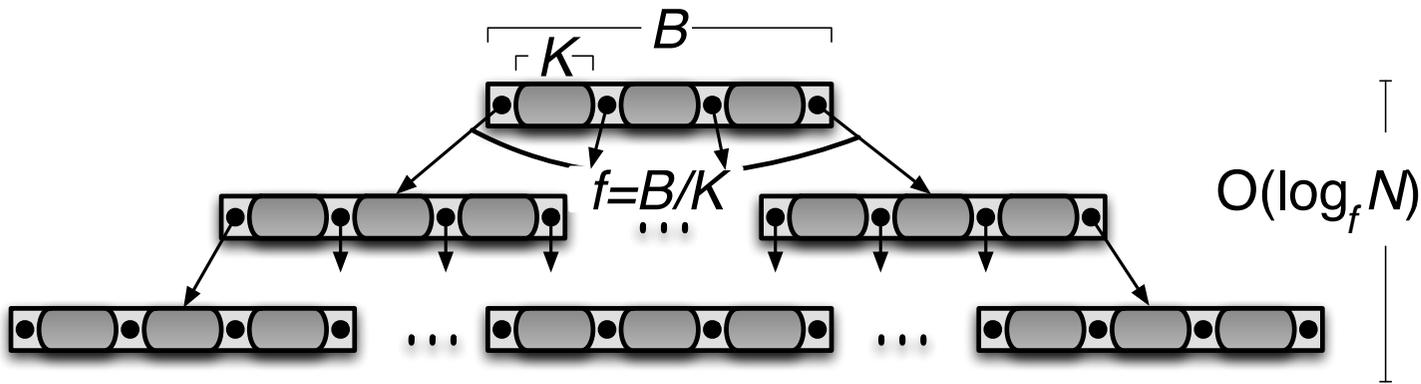


# Difficulty of Key Search (with Different-Size Keys)



# B-trees with Different-Sized Keys

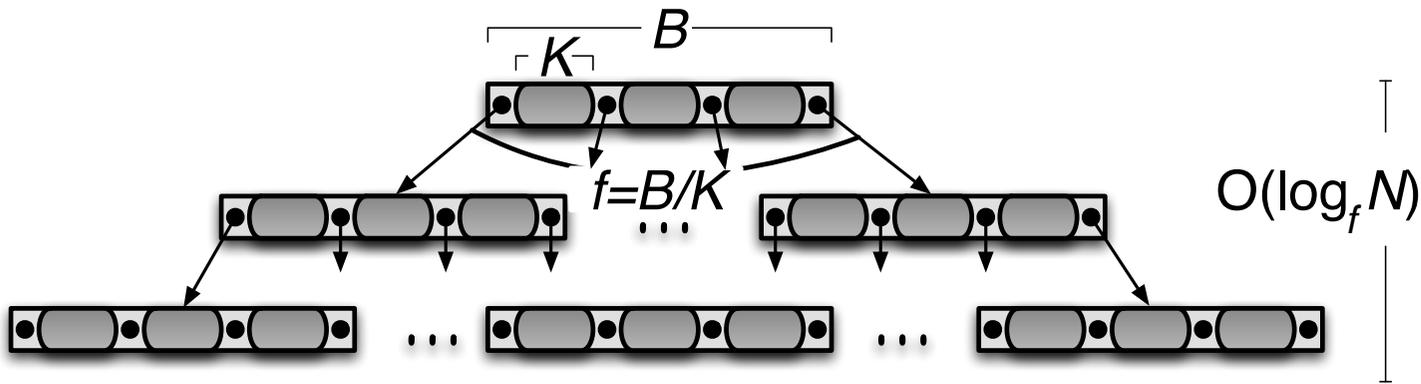
In B-trees in textbooks, all keys have the same size.



Production B-trees support different-size keys...

# B-trees with Different-Sized Keys

**In B-trees in textbooks, all keys have the same size.**



**Production B-trees support different-size keys...**

*But with no nontrivial performance guarantees.*

*Rest of talk: Can we give provably good guarantees in an only slightly modified B-tree?*

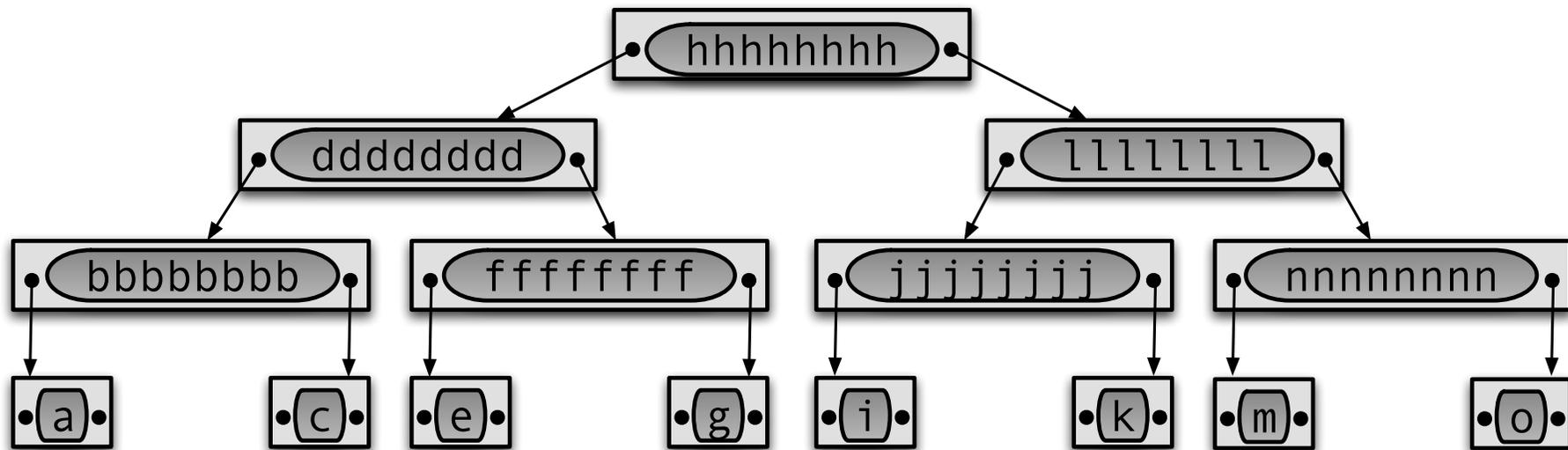
# Example Showing Problem

(a) bbbbbbbb (c) dddddddd (e) ffffffff (g) hhhhhhhh  
(i) jjjjjjjj (k) llllllll (m) nnnnnnnn (o)

# Example Showing Problem

(a) bbbbbbbb (c) dddddddd (e) ffffffff (g) hhhhhhhh  
(i) jjjjjjjj (k) llllllll (m) nnnnnnnn (o)

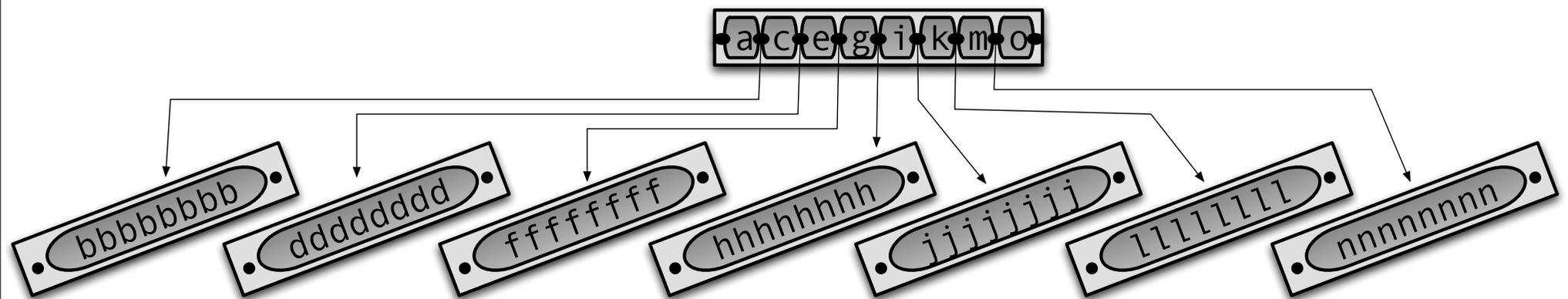
**When the length-8 keys are pivots (and the block size is 8), the tree height is 4:**



# Example Showing Problem

(a) bbbbbbbb (c) dddddddd (e) ffffffff (g) hhhhhhhh  
(i) jjjjjjjj (k) llllllll (m) nnnnnnnn (o)

**When the length-1 keys are pivots (and the block size is 8), the tree height is 2:**



***Choice of pivot affects the B-tree performance.***

# Keys are Atomic

Cannot compare first byte of `bbbbbbbb` with `c`.

# Keys are Atomic

Cannot compare first byte of `bbbbbbbb` with `c`.

Only the comparison function understands the keys.

Keys are opaque. Need to send entire key to comparison function and store entire key in node.



# Keys are Atomic

Cannot compare first byte of `bbbbbbbb` with `c`.

Only the comparison function understands the keys.

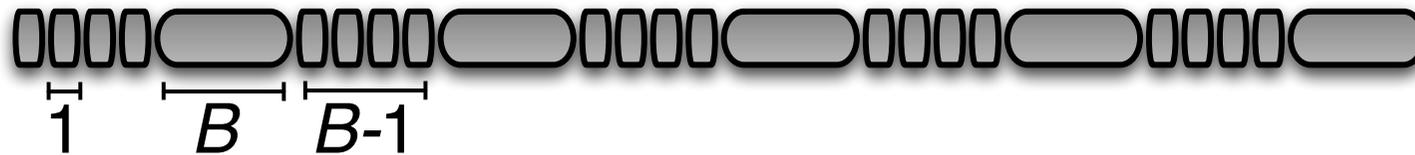
Keys are opaque. Need to send entire key to comparison function and store entire key in node.



# Choice of Pivot Matters For Variable-Size Keys

**Example:  $N$  keys with average size  $< 2$ .**

- $N/B$  keys with size  $B$  and  $N-N/B$  keys with size 1.



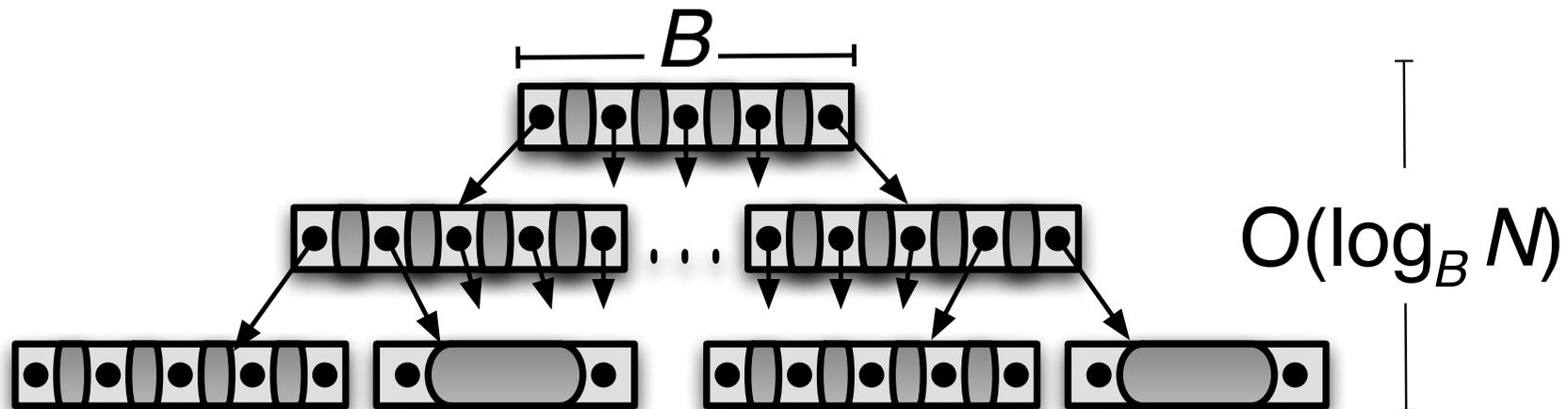
# Choice of Pivot Matters For Variable-Size Keys

**Example:  $N$  keys with average size  $< 2$ .**

- $N/B$  keys with size  $B$  and  $N-N/B$  keys with size 1.



**Size 1 keys as pivots: optimal.**



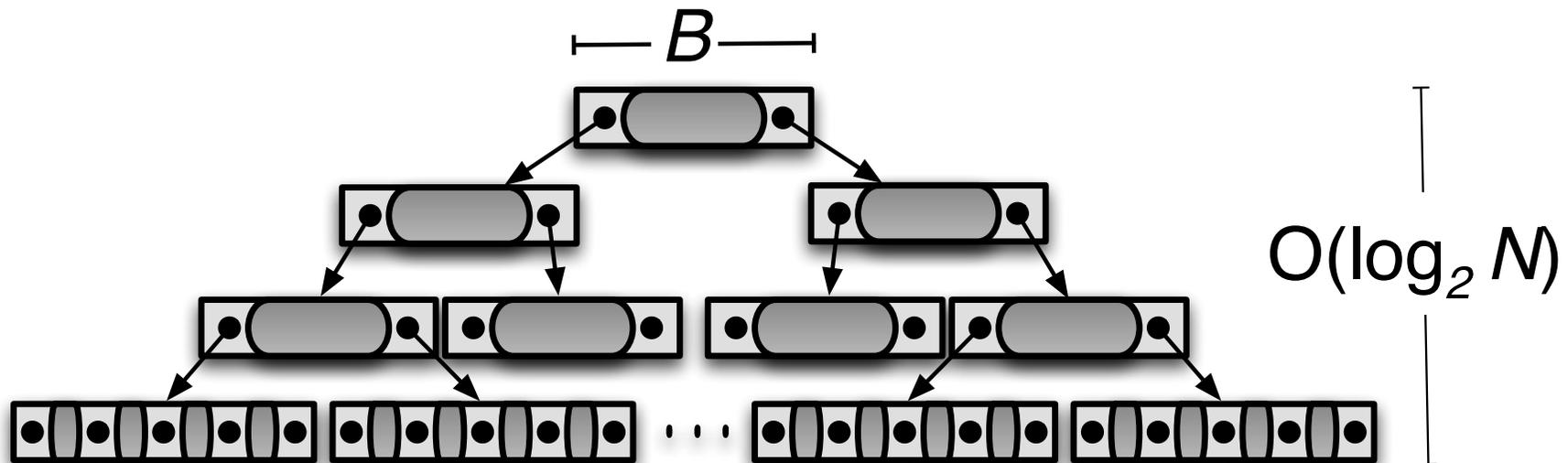
# Choice of Pivot Matters For Variable-Size Keys

**Example:  $N$  keys with average size  $< 2$ .**

- $N/B$  keys with size  $B$  and  $N-N/B$  keys with size 1.



**Size  $B$  keys as pivots:  $O(\log B)$  factor worse.**



# Desired Guarantee

Let  $K$  be the *average* key size.

**Goal:  $O(\log_{B/K} N)$  memory transfers per operation.**

- Generalizes what happens if keys all have the same size  $K$ .

# Desired Guarantee

Let  $K$  be the *average* key size.

**Goal:  $O(\log_{B/K} N)$  memory transfers per operation.**

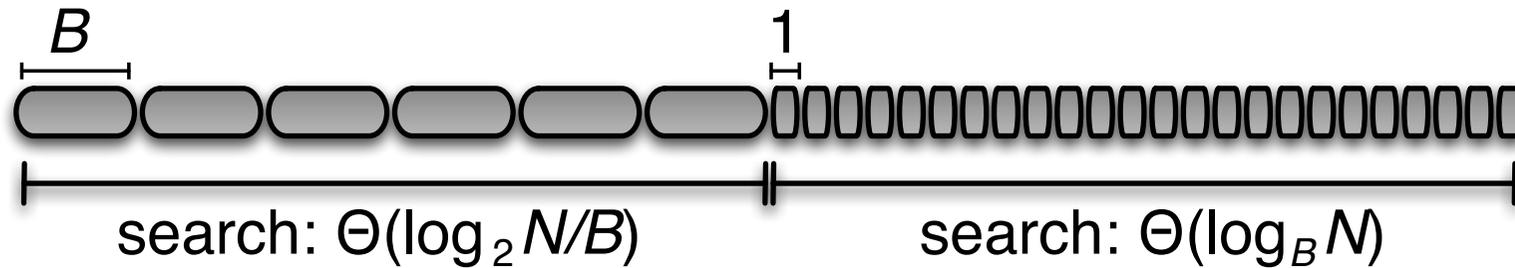
- Generalizes what happens if keys all have the same size  $K$ .

**Unfortunately, we cannot get this for worst-case searches, but we'll get it in expectation.**

# Why We Cannot Attain Good Worst-Case Bounds

**Example:  $N$  keys with average size  $K < 2$ .**

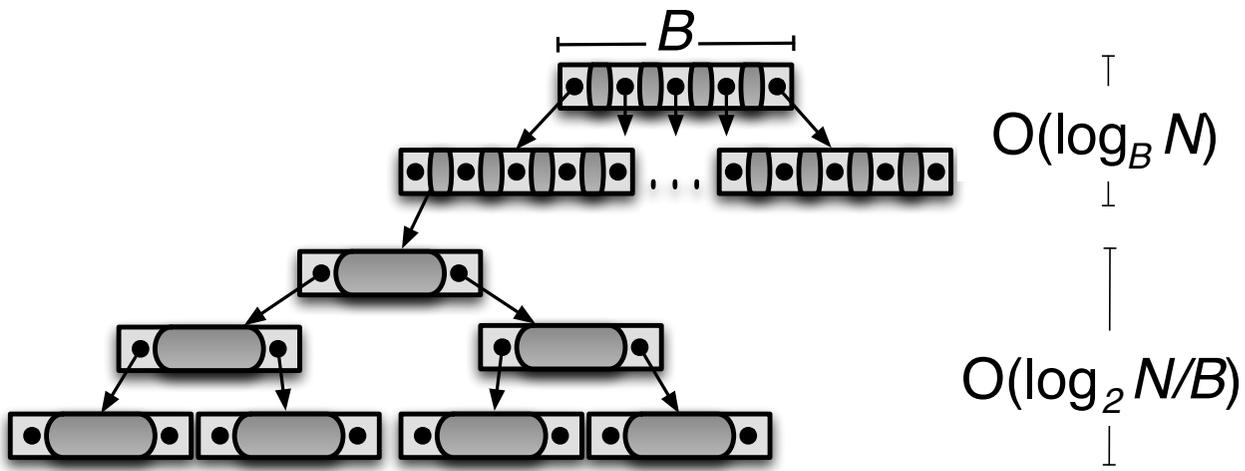
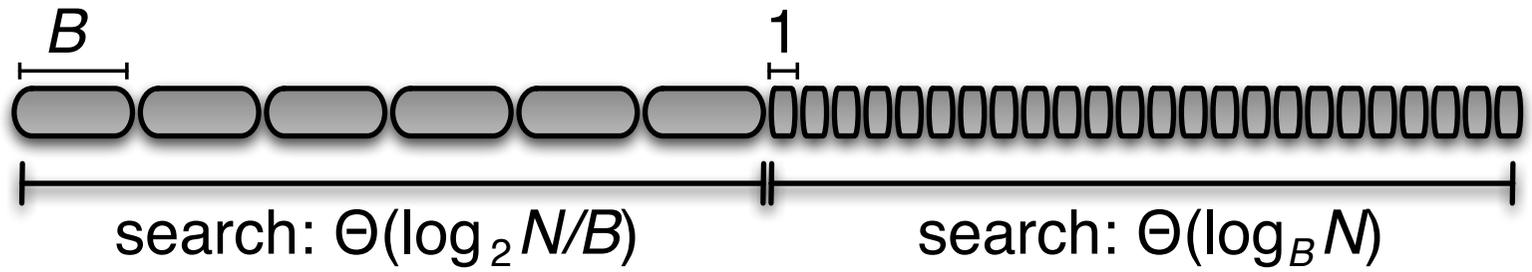
- $N/B$  keys with size  $B$  and  $N - N/B$  keys with size 1.



# Why We Cannot Attain Good Worst-Case Bounds

**Example:  $N$  keys with average size  $K < 2$ .**

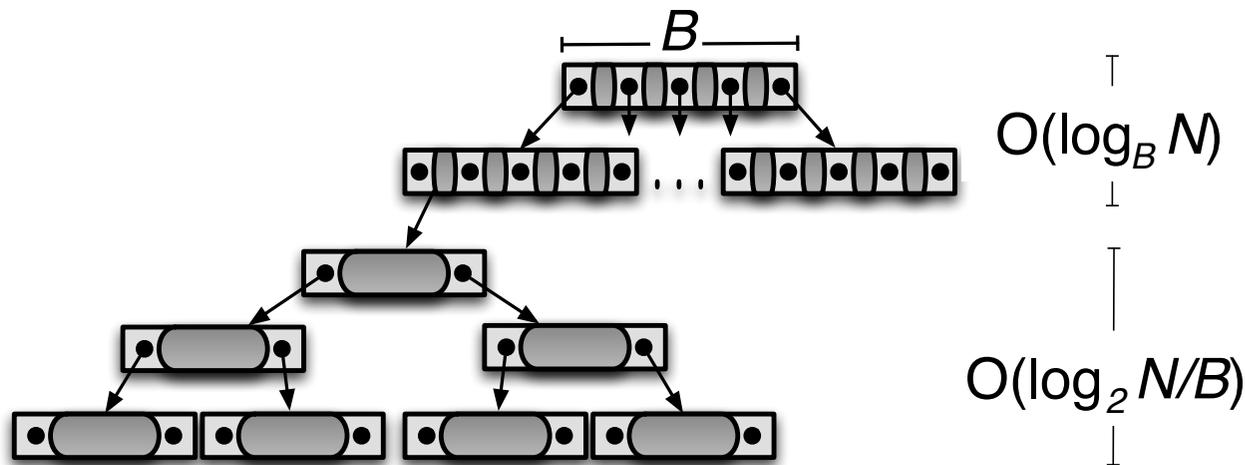
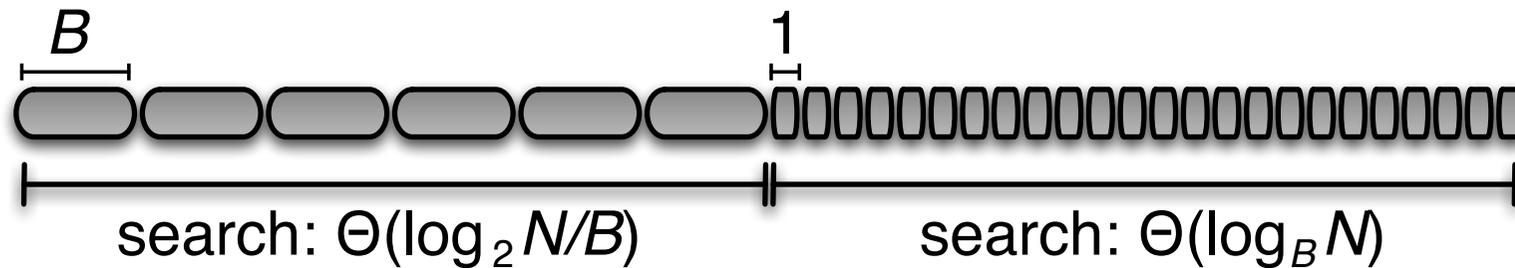
- $N/B$  keys with size  $B$  and  $N - N/B$  keys with size 1.



# Why We Cannot Attain Good Worst-Case Bounds

**Example:  $N$  keys with average size  $K < 2$ .**

- $N/B$  keys with size  $B$  and  $N - N/B$  keys with size 1.

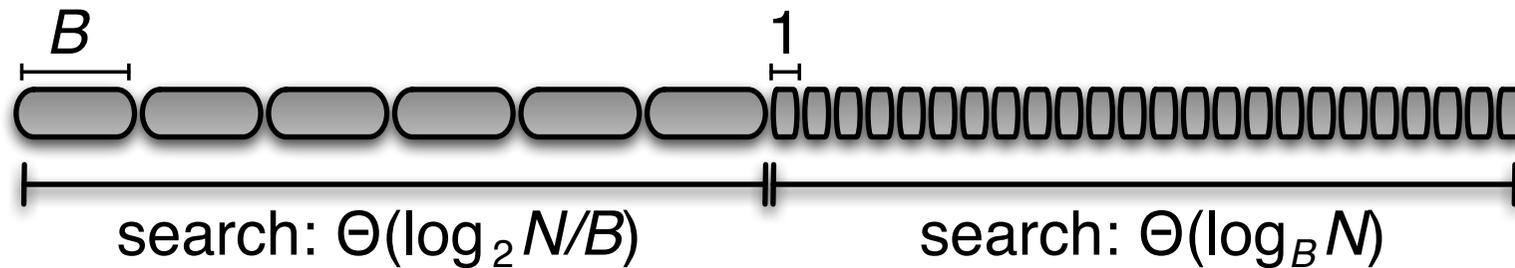


But we are ok on average:  $(1 - 1/B) \log_B N + (1/B) \log_2 N = O(\log_B N)$ .

# Why We Cannot Attain Good Worst-Case Bounds

**Example:  $N$  keys with average size  $K < 2$ .**

- $N/B$  keys with size  $B$  and  $N - N/B$  keys with size 1.



## Related work: how to optimize B-tree height

[Vaishnavi, Kriegel, Wood 80] [Gotleib 81] [Huang, Vishwanathan 90] [Becker 94]

- static (no inserts/deletes)
- DP-based
- (far from our target guarantee)

## Static atomic-key B-tree (only searches)

- Expected leaf search cost:  $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Linear construction cost for sorted data:  $O(NK/B)$

## Static atomic-key B-tree (only searches)

- Expected leaf search cost:  $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Linear construction cost for sorted data:  $O(NK/B)$

Captures  $K=O(B)$   
and  $K \leq \Omega(B)$



## Static atomic-key B-tree (only searches)

- Expected leaf search cost:  $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$  ← Captures  $K=O(B)$  and  $K \leq \Omega(B)$
- Linear construction cost for sorted data:  $O(NK/B)$  ← Scan bound since total length =  $NK$

## Static atomic-key B-tree (only searches)

- Expected leaf search cost:  $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$  ← Captures  $K=O(B)$  and  $K \leq \Omega(B)$
- Linear construction cost for sorted data:  $O(NK/B)$  ← Scan bound since total length =  $NK$

## Dynamic atomic-key B-tree

- Expected leaf search cost :  $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Cost to insert/delete/search for key  $L$  of *random* rank (amort):  
 $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N + |L|/B)$
- Cost to insert/delete/search for key of *arbitrary* rank:  
***modification cost is dominated by search cost.***

## Static atomic-key B-tree (only searches)

- Expected leaf search cost:  $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$  ← Captures  $K=O(B)$  and  $K \leq \Omega(B)$
- Linear construction cost for sorted data:  $O(NK/B)$  ← Scan bound since total length =  $NK$

## Dynamic atomic-key B-tree

- Expected leaf search cost :  $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Cost to insert/delete/search for key  $L$  of *random* rank (amort):  
 $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N + |L|/B)$  ←  $O(|L|/B)$  is cost to read  $L$  into memory
- Cost to insert/delete/search for key of *arbitrary* rank:  
***modification cost is dominated by search cost.***

## Static atomic-key B-tree (only searches)

- Expected leaf search cost:  $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$  ← Captures  $K=O(B)$  and  $K \leq \Omega(B)$
- Linear construction cost for sorted data:  $O(NK/B)$  ← Scan bound since total length =  $NK$

## Dynamic atomic-key B-tree

- Expected leaf search cost :  $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Cost to insert/delete/search for key  $L$  of *random* rank (amort):  
 $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N + |L|/B)$  ←  $O(|L|/B)$  is cost to read  $L$  into memory
- Cost to insert/delete/search for key of *arbitrary* rank:  
***modification cost is dominated by search cost.*** ← important  
For streaming B-tree, should be **much** less than search cost.

# Atomic-Key B-tree Intuition

## Greedy construction algorithm

- Greedily select pivot elements for the root node
- Proceed recursively on all subtrees of the root.

## Intuition

- Pick small keys in root to maximize fanout.
- Pick evenly distributed keys to reduce the search space.

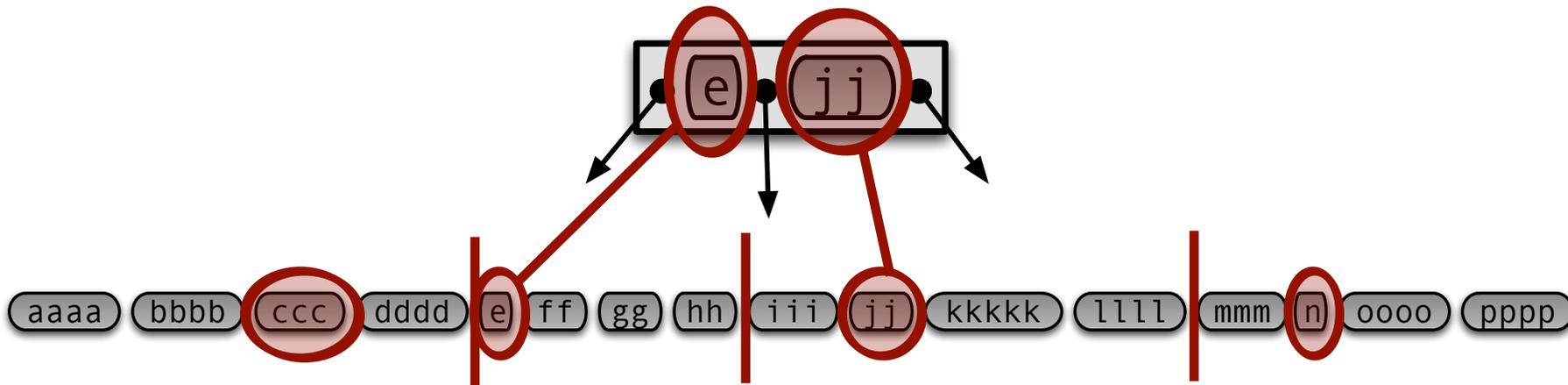
## To prove

- Root has a good structure.
- Recursive substructures achieve good performance, even though subtrees may have different average key sizes.

## How fast can we insert?

# Root Construction

1. Divide keys into equal-size groups.
2. Pick (one of the) short keys in each group.
3. Store these keys in root



**Satisfies constraints that keys are small and evenly distributed.**

**Enables inserts/deletes.**

# What can we tell you about what your hardware can do?

You **cannot** have 1 I/O/sec/GB (cheaply).

You **can** insert data in any order at near disk bandwidth.

You **can** query data fast if the data is organized in the right order (e.g., has locality for that query).

You **must** organize your system so queries have locality.

- E.g., *find* on a file system can be fast if the data is indexed correctly.

You **must** use indexing to organize data for queries.

***Indexing is the currency that we can use to make queries go faster.***