How TokuDB Fractal Tree™ Indexes Work

Bradley C. Kuszmaul

Guest Lecture in MIT 6.172 Performance Engineering, 18 November 2010.

My Background

- I'm an MIT alum:

$$\text{MIT Degrees} = 2 \times \text{S.B} + \text{S.M.} + \text{Ph.D.}$$

- I was a principal architect of the Connection Machine CM-5 super-computer at Thinking Machines.

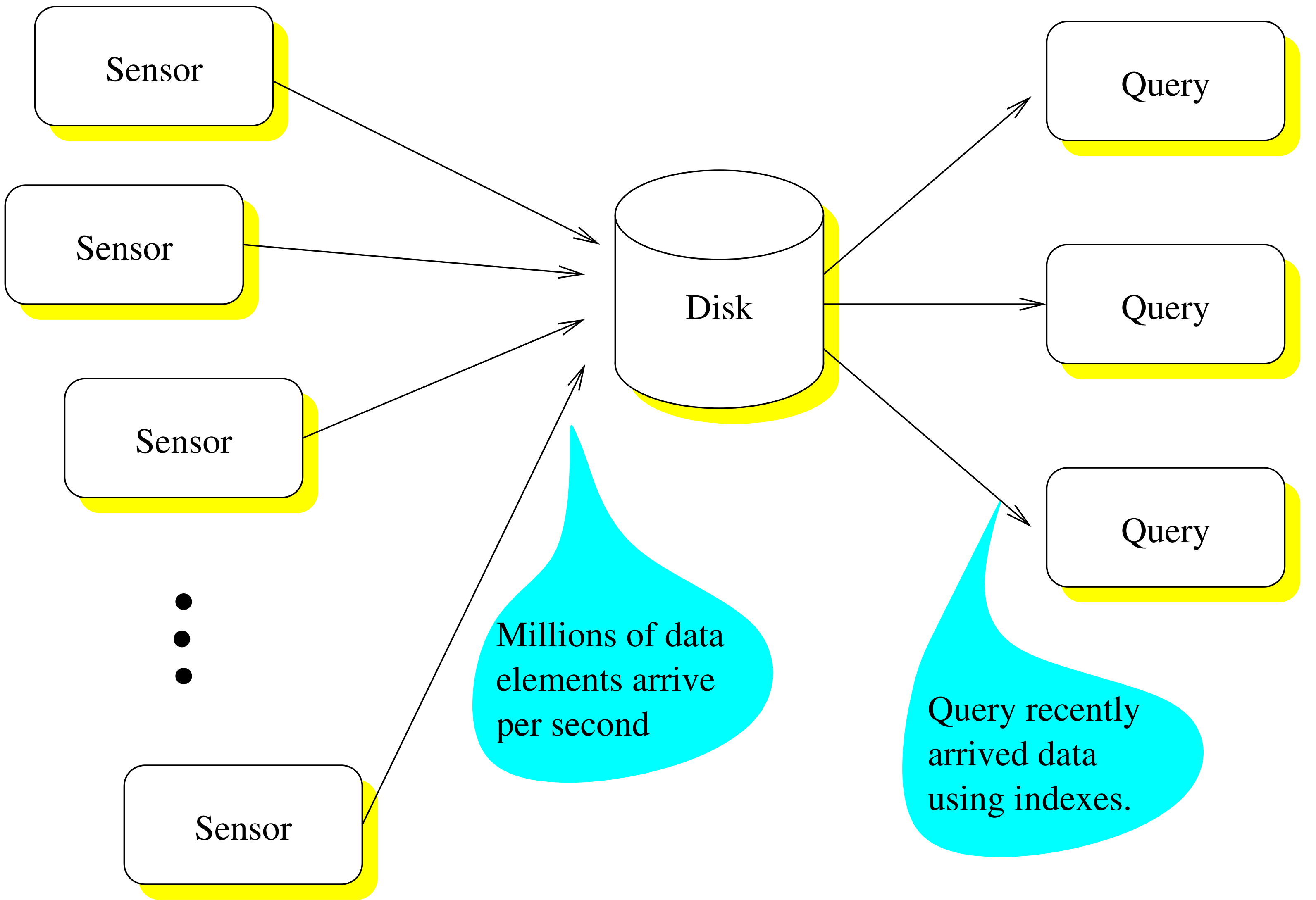
- I was Assistant Professor at Yale.
- I was Akamai working on network mapping and billing.
- I am research faculty in the SuperTech group, working with Charles.

Tokutek

A few years ago I started collaborating with Michael Bender and Martin Farach-Colton on how to store data on disk to achieve high performance.

We started Tokutek to commercialize the research.

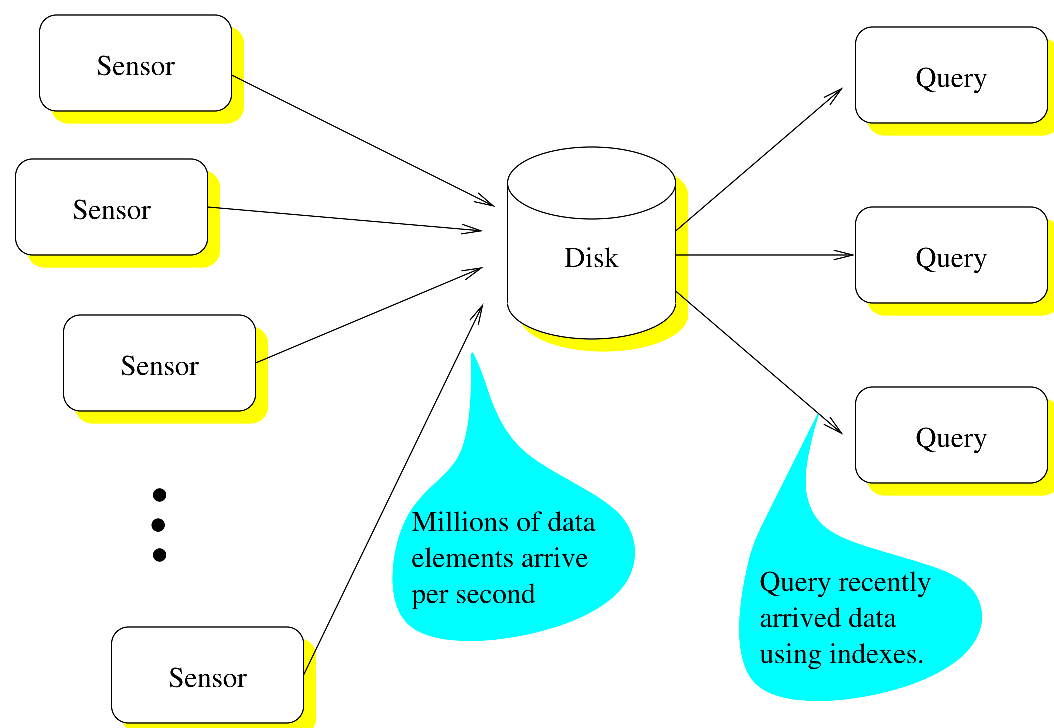
I/O is a Big Bottleneck



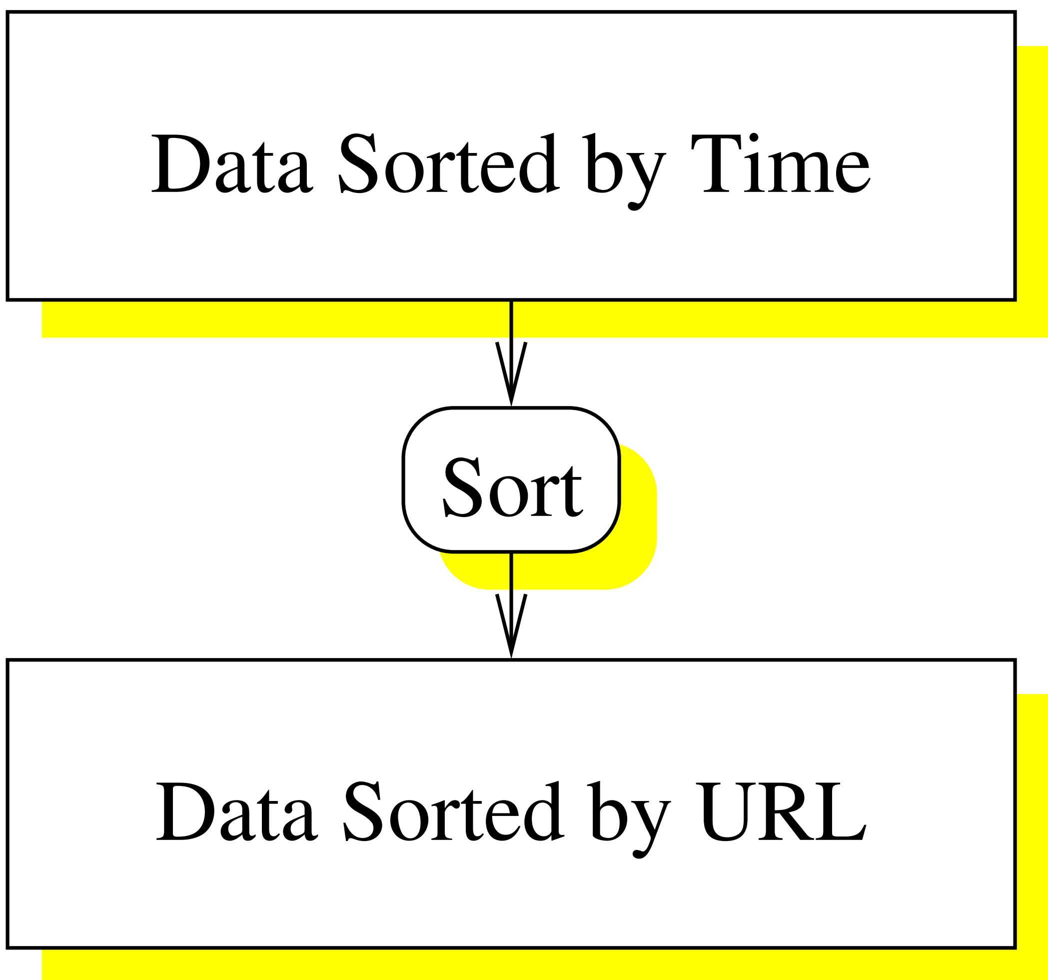
Systems include sensors and storage, and want to perform queries on recent data.

The Data Indexing Problem

- Data arrives in one order (say, sorted by the time of the observation).
- Data is queried in another order (say, by URL or location).



Why Not Simply Sort?

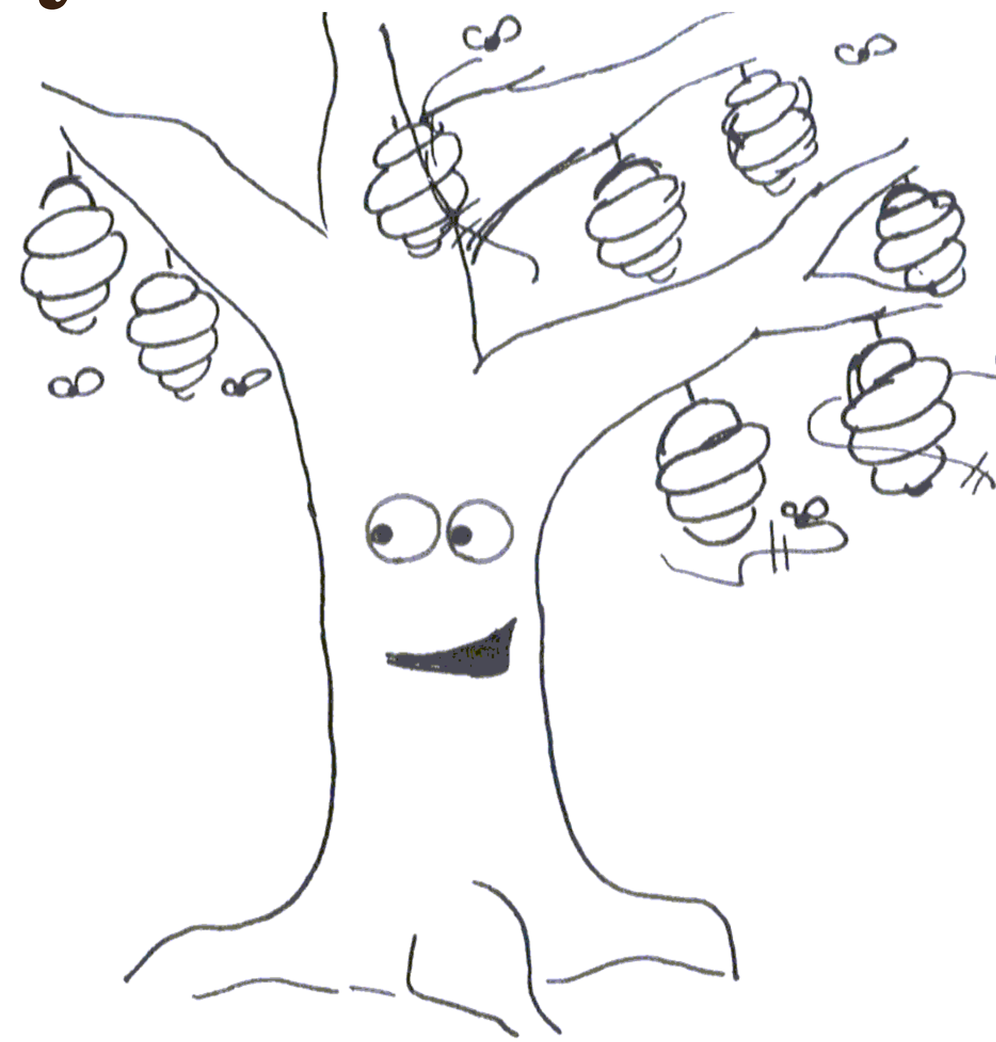


- This is what data warehouses do.
- The problem is that you must wait to sort the data before querying it: typically an overnight delay.

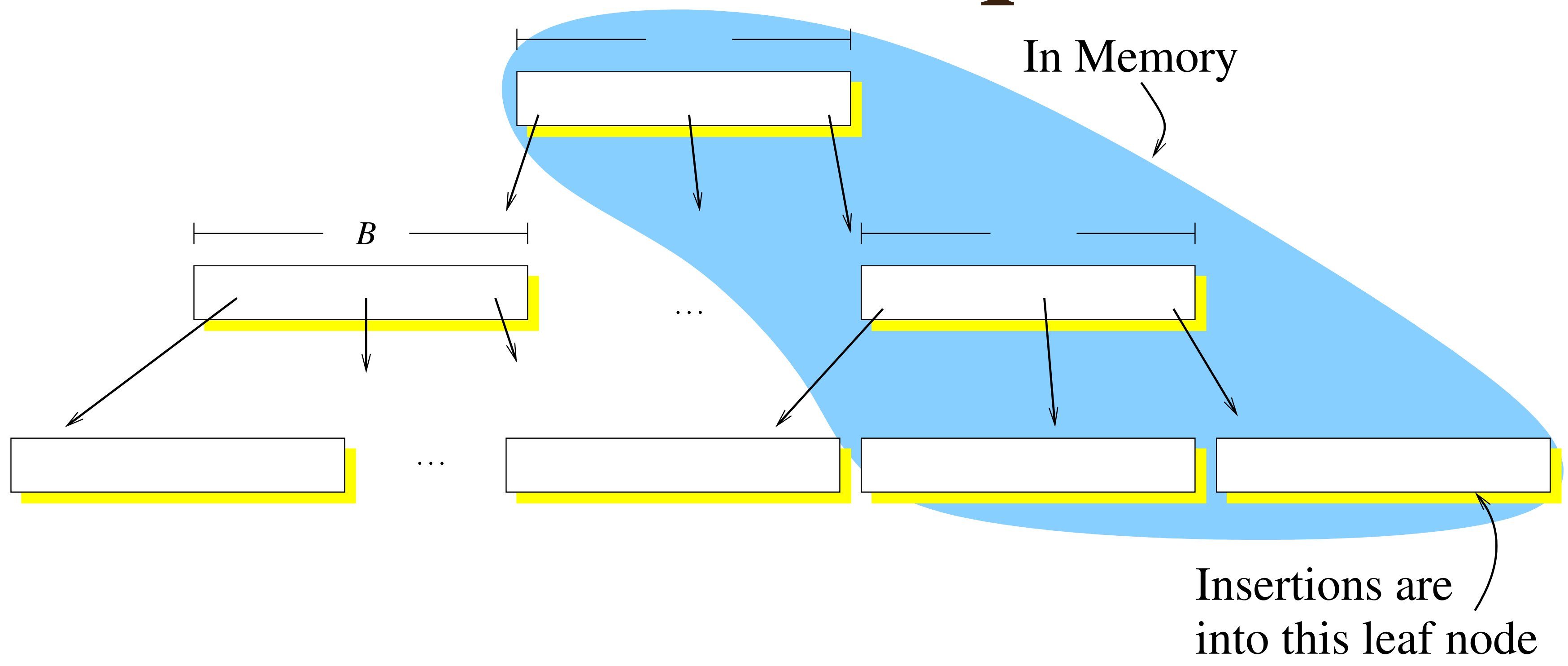
The system must maintain data in (effectively) several sorted orders. This problem is called *maintaining indexes*.

B-Trees are Everywhere

B-Trees show up in database indexes (such as MyISAM and InnoDB), file systems (such as XFS), and many other storage systems.

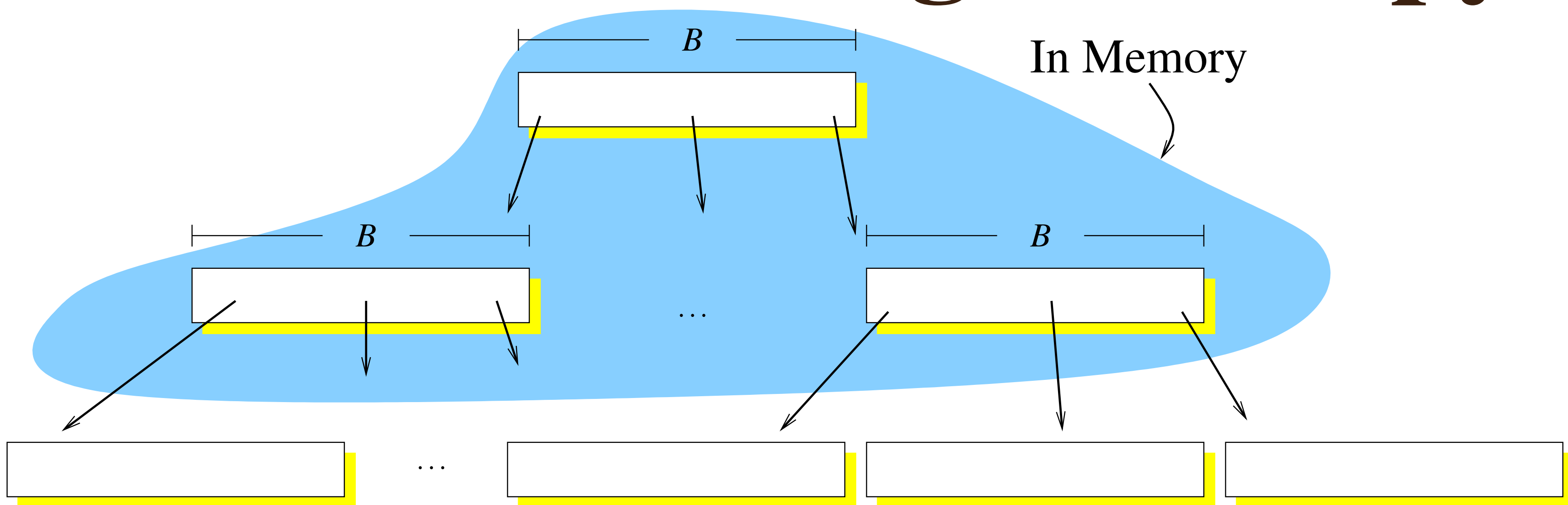


B-Trees are Fast at Sequential Inserts



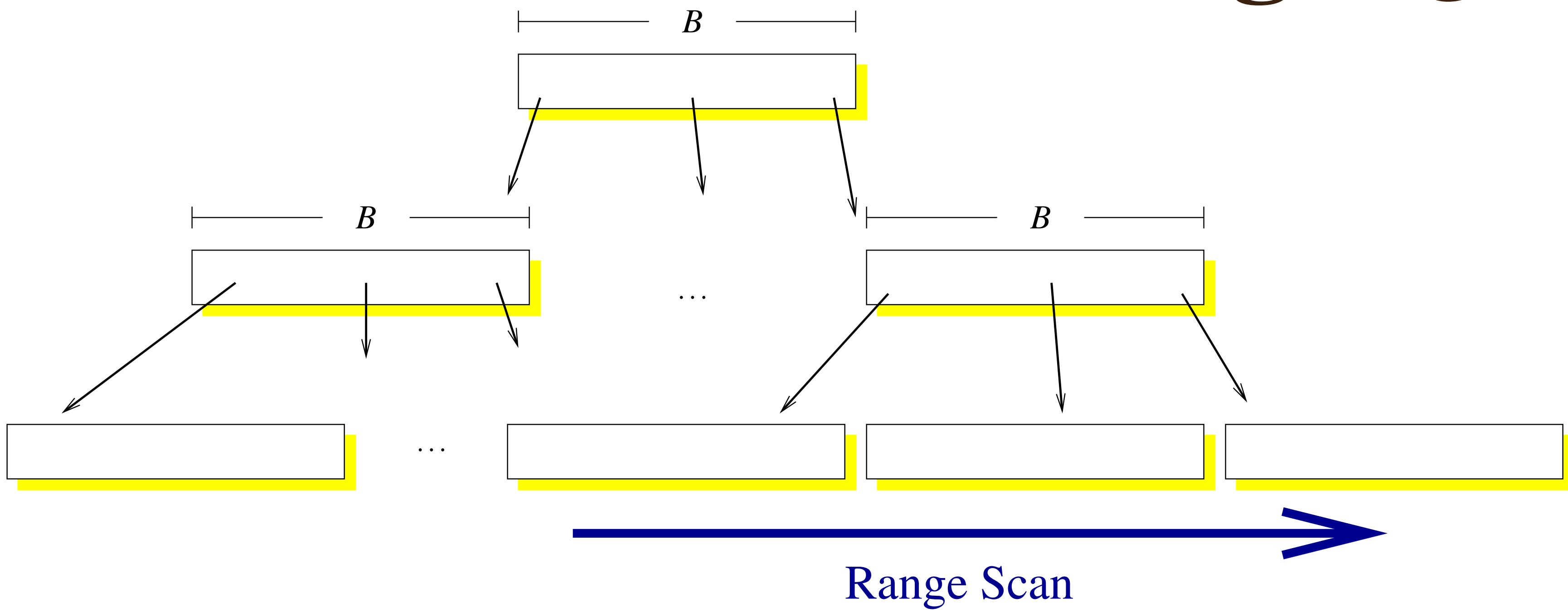
- One disk I/O per leaf (which contains many rows).
- Sequential disk I/O.
- Performance is limited by *disk bandwidth*.

B-Trees are Slow for High-Entropy Inserts



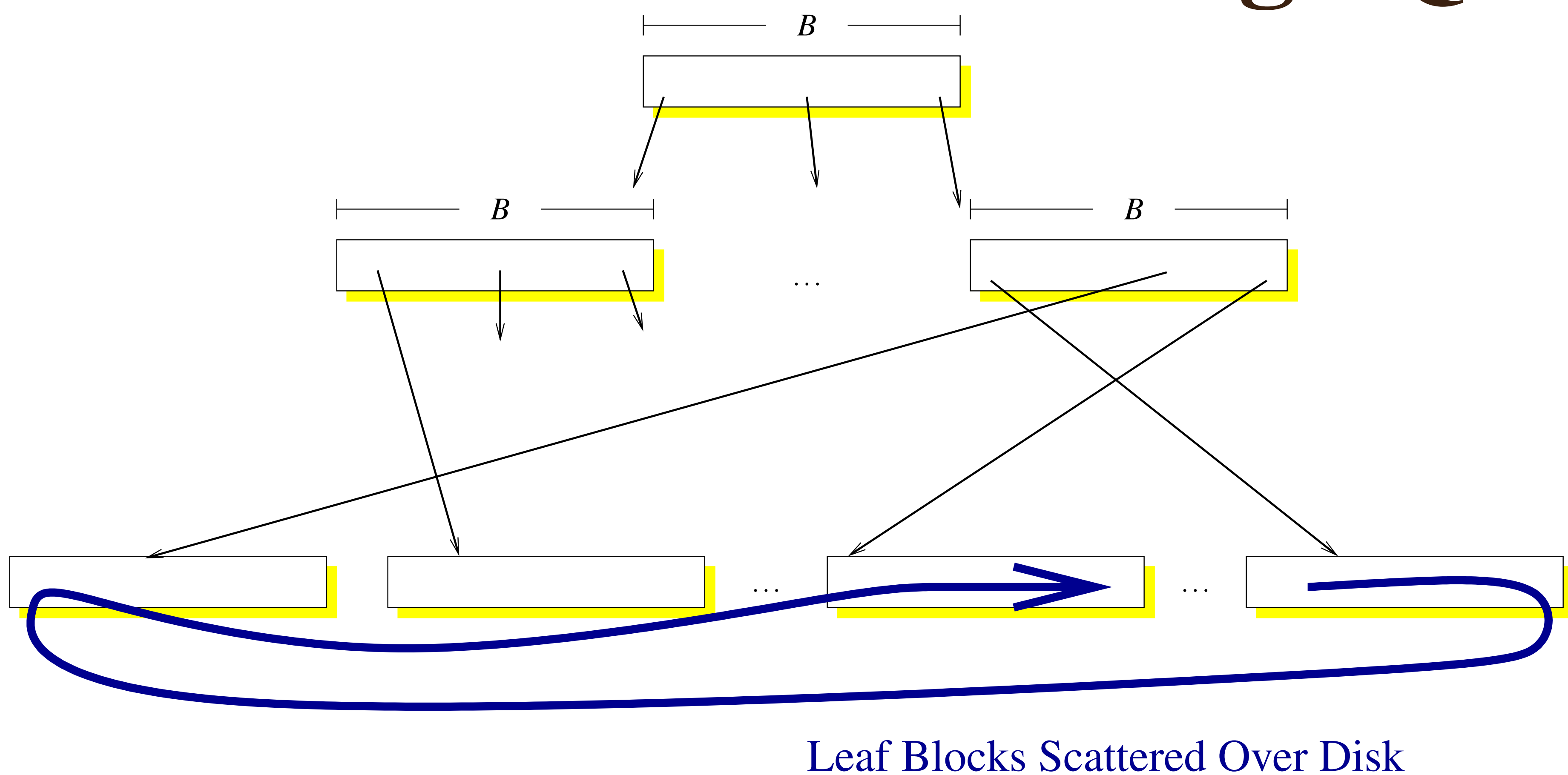
- Most nodes are not in main memory.
- Most insertions require a random disk I/O.
- Performance is limited by *disk head movement*.
- Only 100's of inserts/s/disk ($\leq 0.2\%$ of disk bandwidth).

New B-Trees Run Fast Range Queries



- In newly created B-trees, the leaf nodes are often laid out sequentially on disk.
- Can get near 100% of disk bandwidth.
- About 100MB/s per disk.

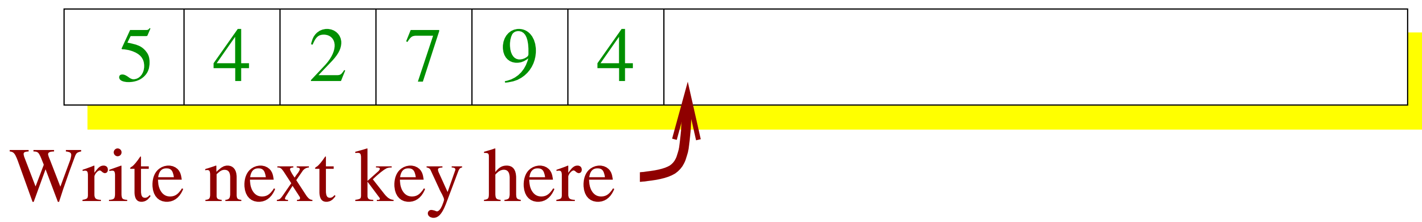
Aged B-Trees Run Slow Range Queries



- In aged trees, the leaf blocks end up scattered over disk.
- For 16KB nodes, as little as 1.6% of disk bandwidth.
- About 16KB/s per disk.

Append-to-file Beats B-Trees at Insertions

Here's a data structure that is very fast for insertions:



Write to the end of a file.

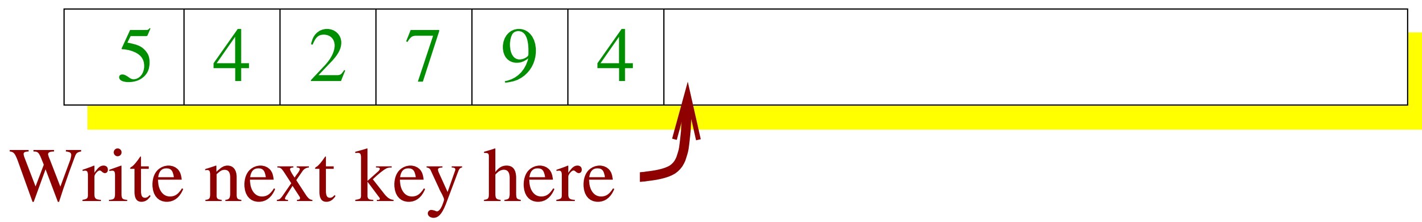
Pros:

- Achieve disk bandwidth even for random keys.

Cons:

Append-to-file Beats B-Trees at Insertions

Here's a data structure that is very fast for insertions:



Write to the end of a file.

Pros:

- Achieve disk bandwidth even for random keys.

Cons:

- Looking up anything requires a table scan.

A Performance Tradeoff?

Structure	Inserts	Point Queries	Range Queries
B-Tree	Horrible	Good	Good (young)
Append	Wonderful	Horrible	Horrible

- B-trees are good at lookup, but bad at insert.
- Append-to-file is good at insert, but bad at lookup.
- Is there a data structure that is about as good as a B-tree for lookup, but has insertion performance closer to append?

A Performance Tradeoff?

Structure	Inserts	Point Queries	Range Queries
B-Tree	Horrible	Good	Good (young)
Append	Wonderful	Horrible	Horrible
Fractal Tree	Good	Good	Good

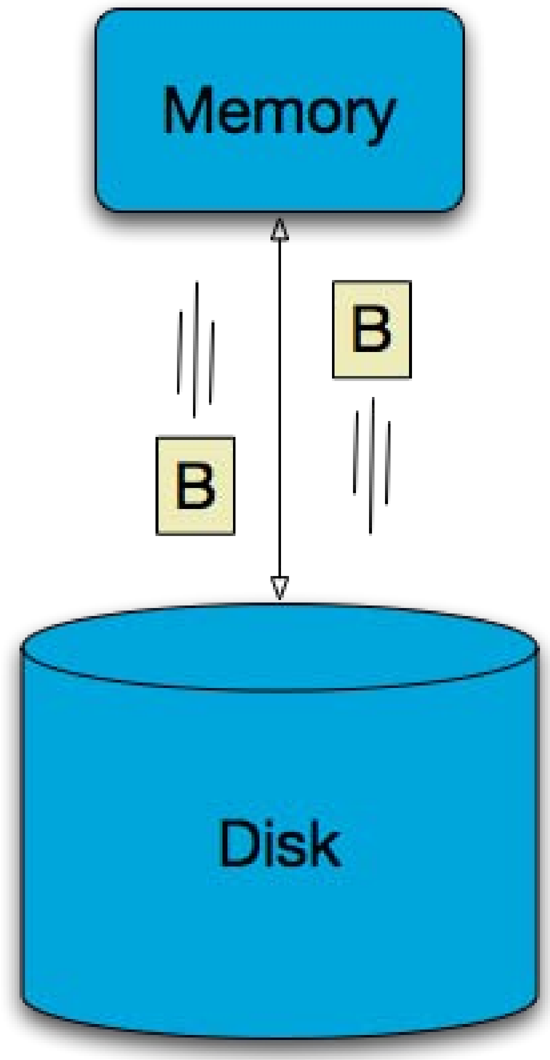
- B-trees are good at lookup, but bad at insert.
- Append-to-file is good at insert, but bad at lookup.
- Is there a data structure that is about as good as a B-tree for lookup, but has insertion performance closer to append?

Yes, Fractal Trees!

An Algorithmic Performance Model

To analyze performance we use the Disk-Access Machine (DAM) model. [Aggrawal, Vitter 88]

- Two levels of memory.
- Two parameters: block size B, and memory size M.
- The game: Minimize the number of block transfers. Don't worry about CPU cycles.



Theoretical Results

Structure	Insert	Point Query
B-Tree	$O\left(\frac{\log N}{\log B}\right)$	$O\left(\frac{\log N}{\log B}\right)$
Append	$O\left(\frac{1}{B}\right)$	$O\left(\frac{N}{B}\right)$
Fractal Tree	$O\left(\frac{\log N}{B^{1-\varepsilon}}\right)$	$O\left(\frac{\log N}{\varepsilon \log B^{1-\varepsilon}}\right)$

Example of Insertion Cost

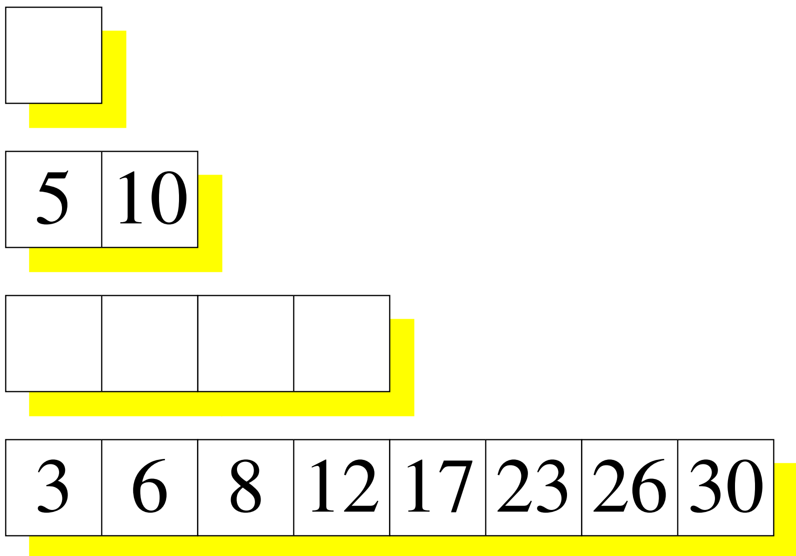
- 1 billion 128-byte rows. $N = 2^{30}$; $\log(N) = 30$.
- 1MB block holds 8192 rows. $B = 8192$; $\log B = 13$.

B-Tree: $$O\left(\frac{\log N}{\log B}\right) = O\left(\frac{30}{13}\right) \approx 3$$

Fractal Tree: $$O\left(\frac{\log N}{B}\right) = O\left(\frac{30}{8192}\right) \approx 0.003.$$

Fractal Trees use << 1 disk I/O per insertion.

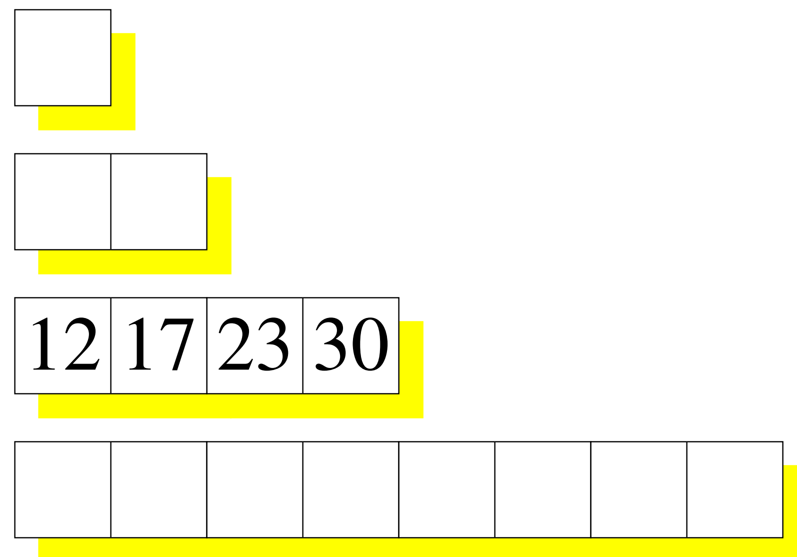
A Simplified Fractal Tree



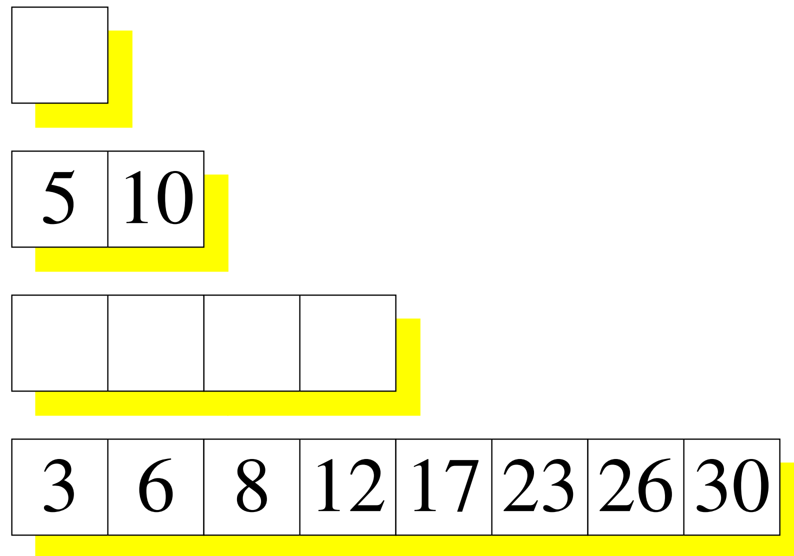
- $\log N$ arrays, one array for each power of two.
- Each array is completely full or empty.
- Each array is sorted.

Example (4 elements)

If there are 4 elements in our fractal tree, the structure looks like this:



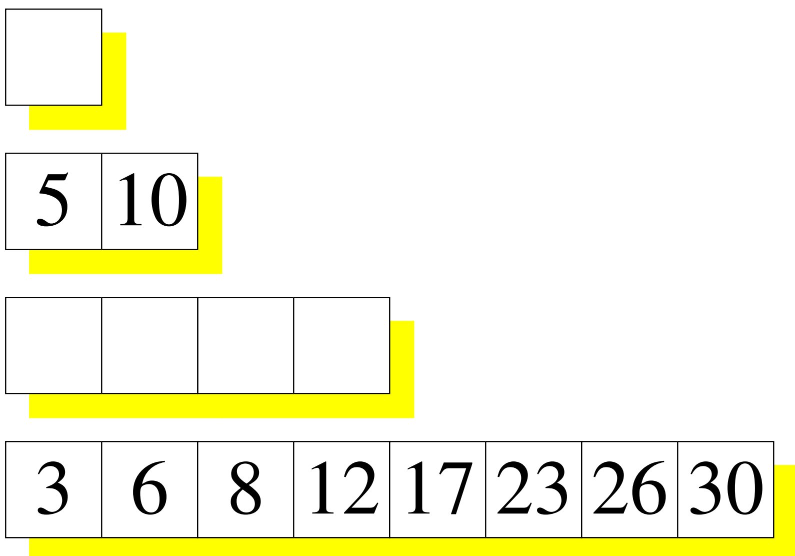
If there are 10 elements in our fractal tree, the structure might look like this:



But there is some freedom.

- Each array is full or empty, so the 2-array and the 8-array must be full.
- However, which elements go where isn't completely specified.

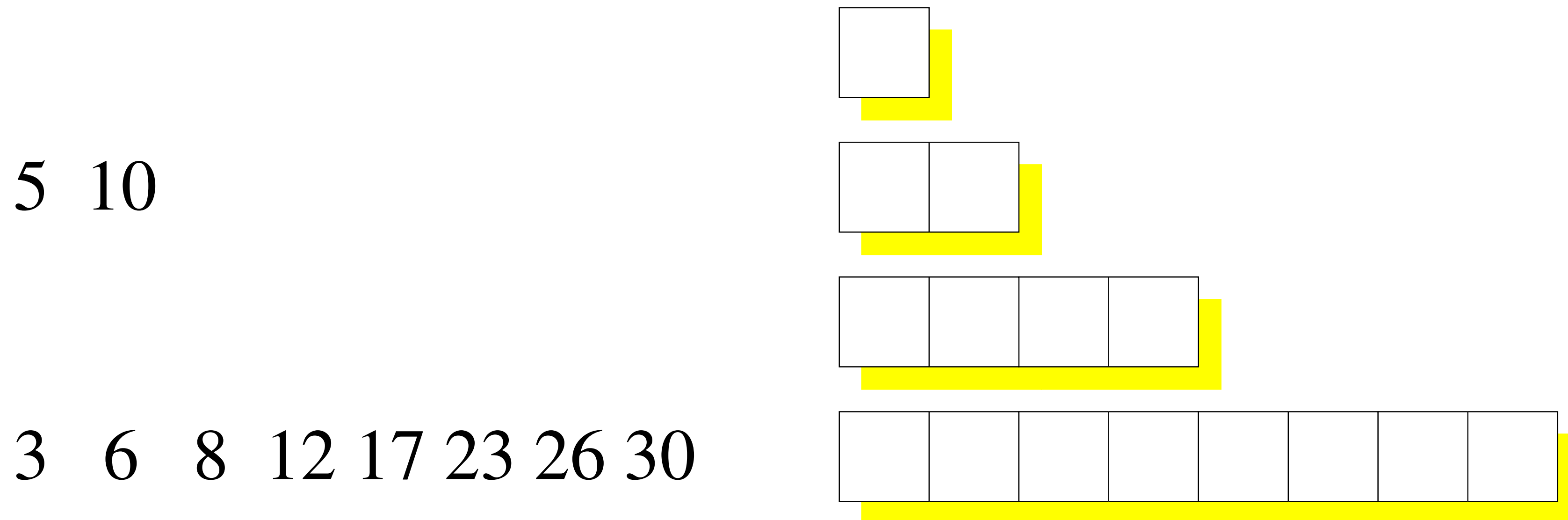
Searching in a Simplified Fractal Tree



- Idea: Perform a binary search in each array.
- Pros: It works. It's faster than a table scan.
- Cons: It's slower than a B-tree at $O(\log^2 N)$ block transfers.

Let's put search aside, and consider insert.

Inserting in a Simplified Fractal Tree

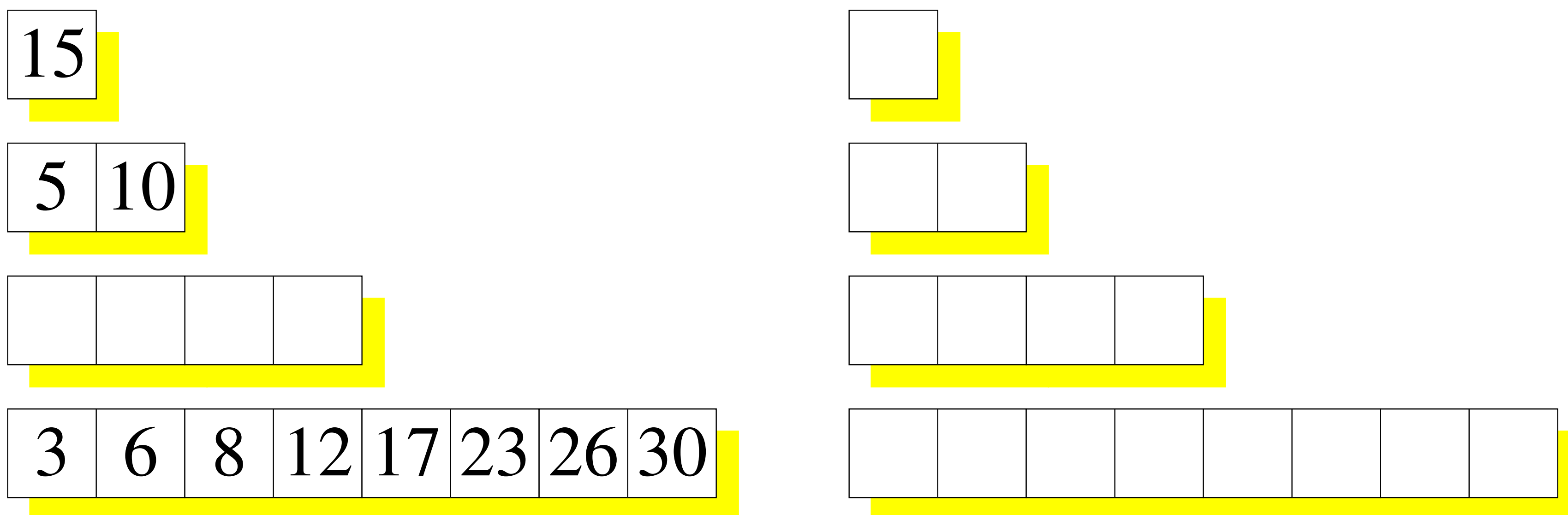


Add another array of each size for temporary storage.

At the beginning of each step, the temporary arrays are empty.

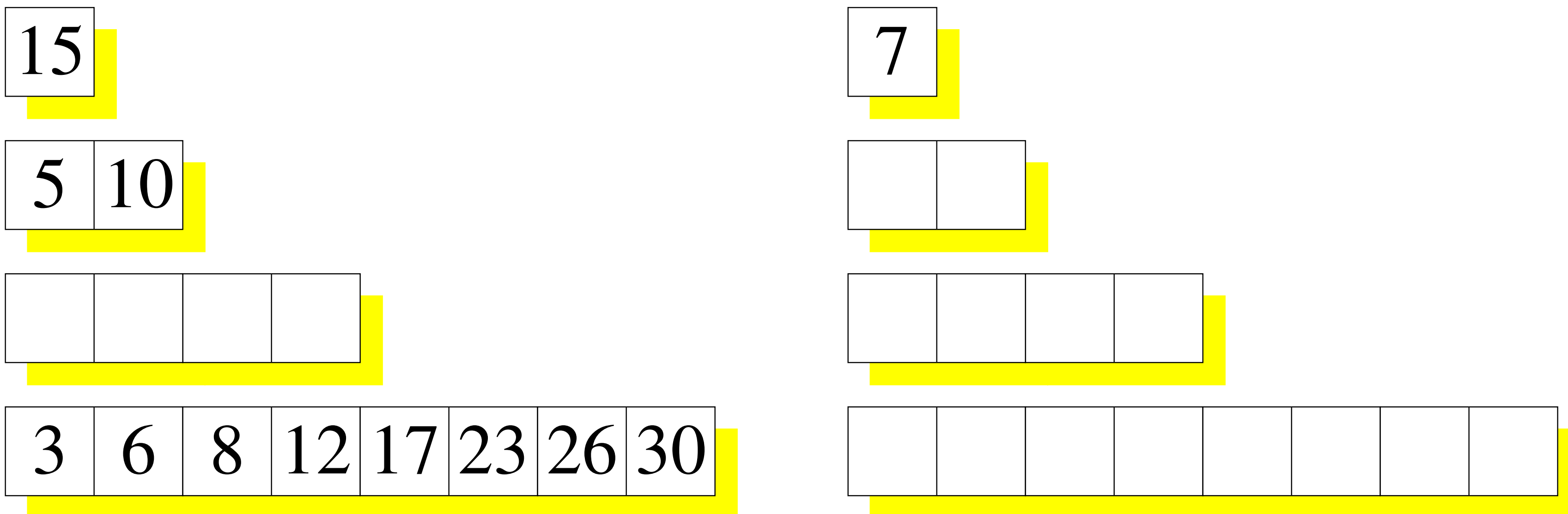
Insert 15

To insert 15, there is only one place to put it: In the 1-array.

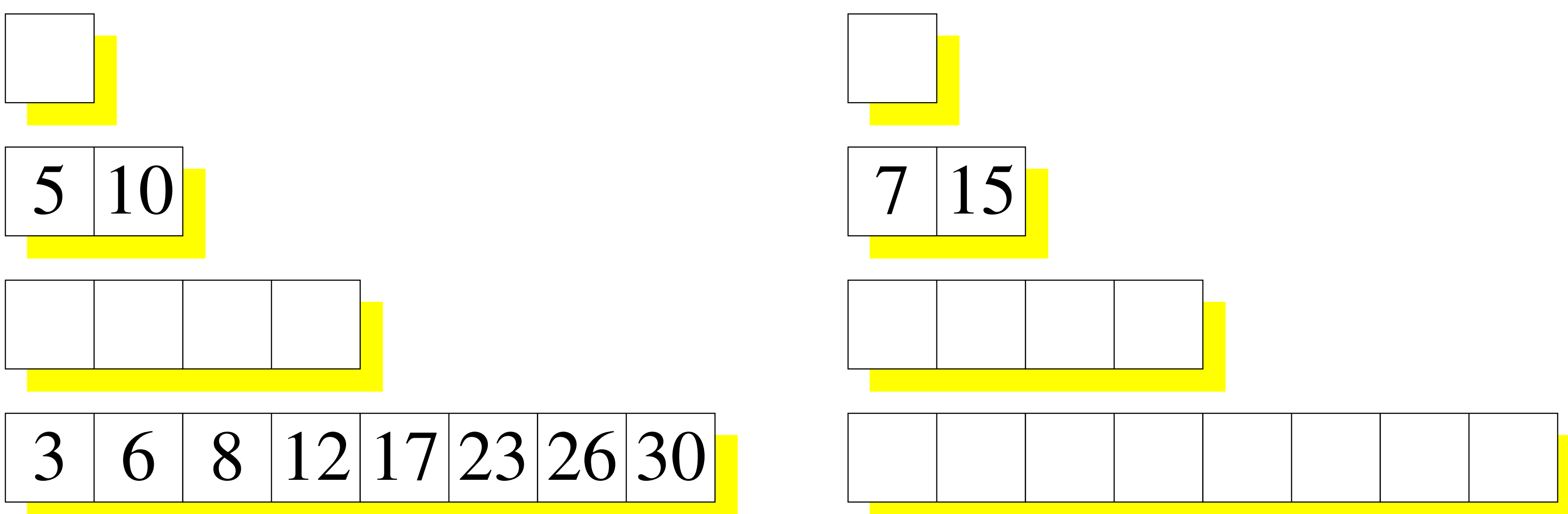


Insert 7

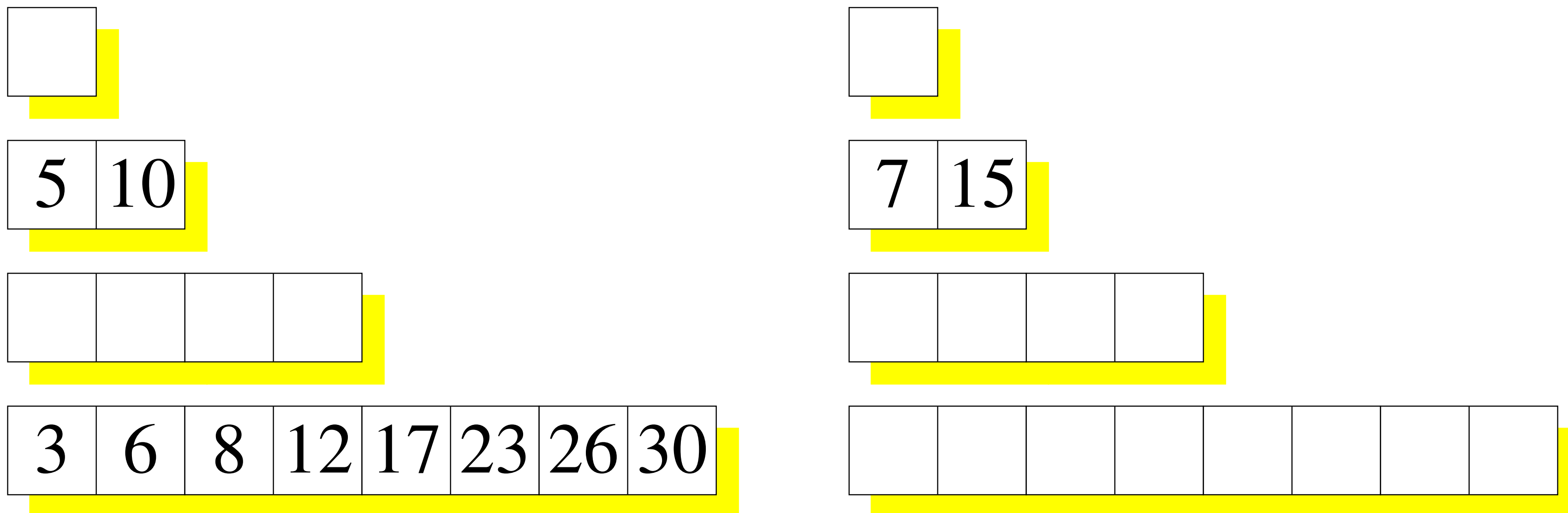
To insert 7, no space in the 1-array. Put it in the temp 1-array.



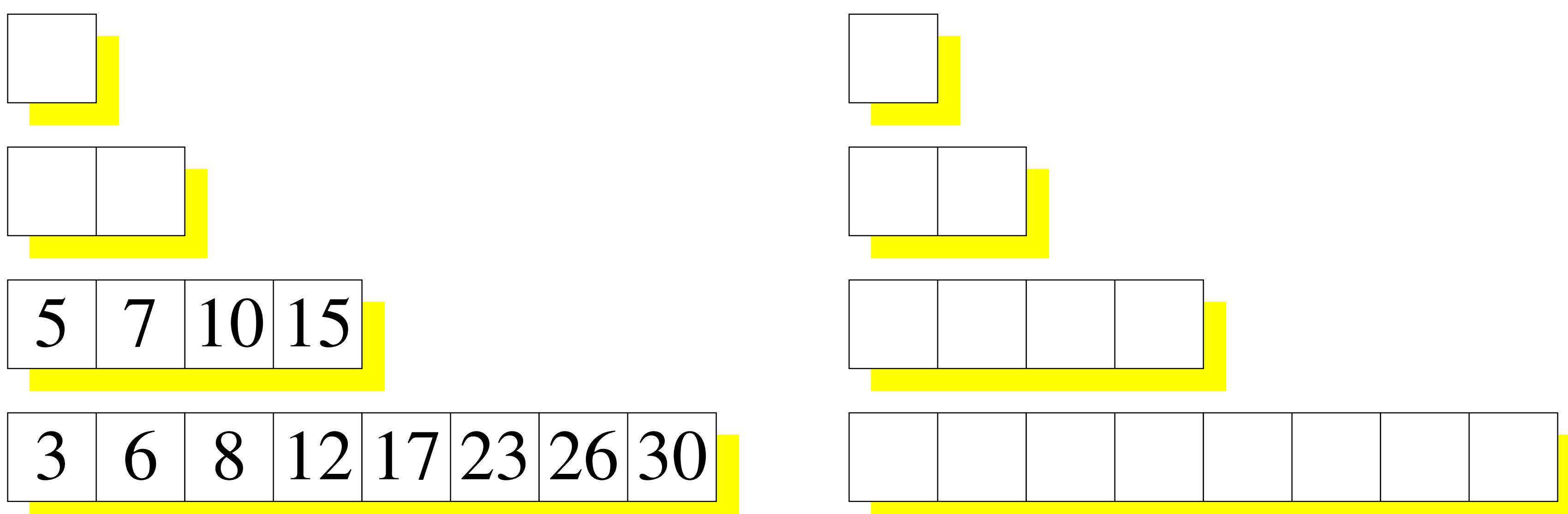
Then merge the two 1-arrays to make a new 2-array.



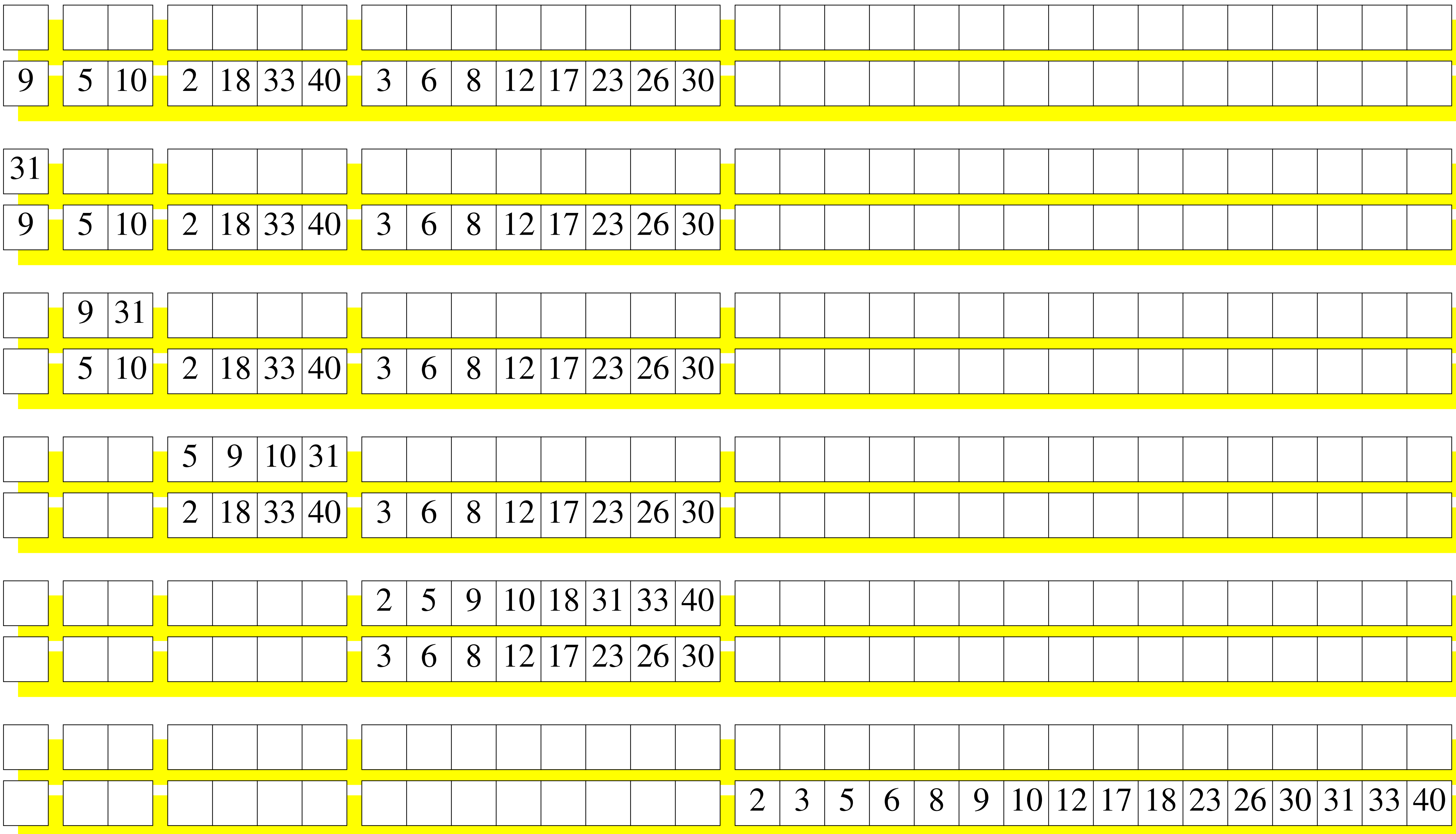
Not done inserting 7



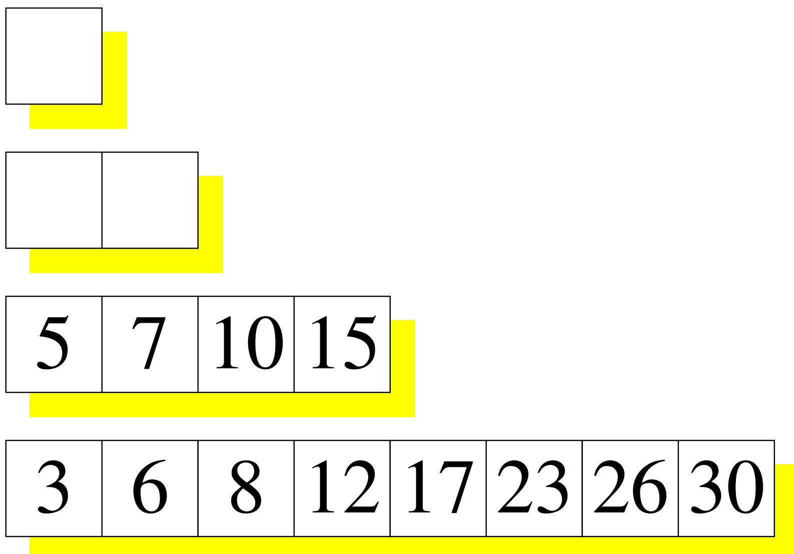
Must merge the 2-arrays to make a 4-array.



An Insert Can Cause Many Merges



Analysis of Insertion into Simplified Fractal Tree



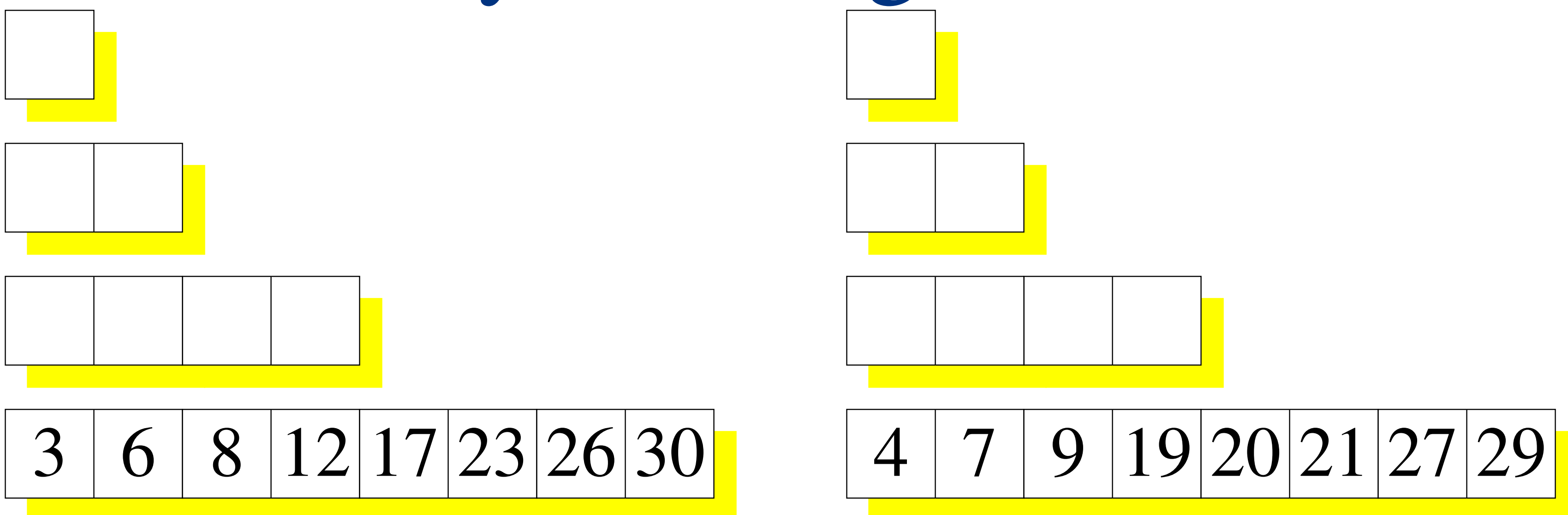
- Cost to merge 2 arrays of size X is $O(X/B)$ block I/Os.

  Merge is very I/O efficient.

- Cost per element to merge is $O(1/B)$ since $O(X)$ elements were merged.
- Max # of times each element is merged is $O(\log N)$.
- Average insert cost is $O\left(\frac{\log N}{B}\right)$.

Improving Worst-Case Insertion

Although the *average* cost of a merge is low, occasionally we merge a *lot* of stuff.

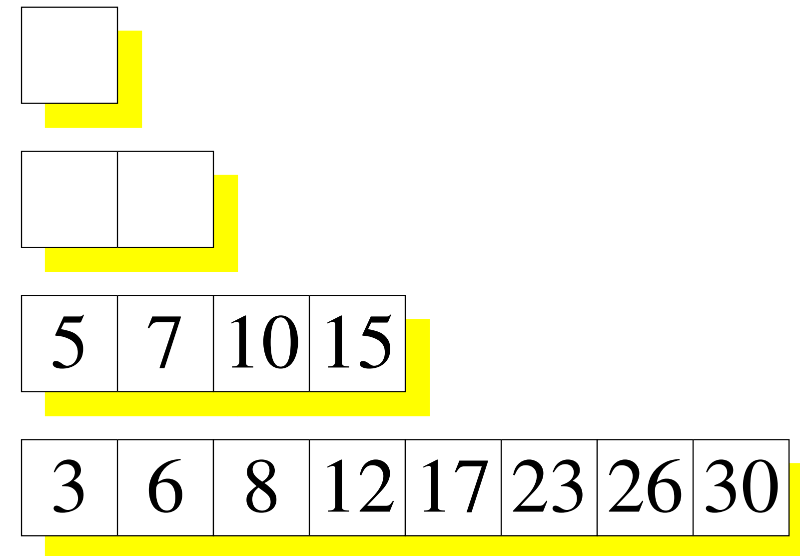


Idea: A separate thread merges arrays. An insert returns quickly.

Lemma: As long as we merge $\Omega(\log N)$ elements for every insertion, the merge thread won't fall behind.

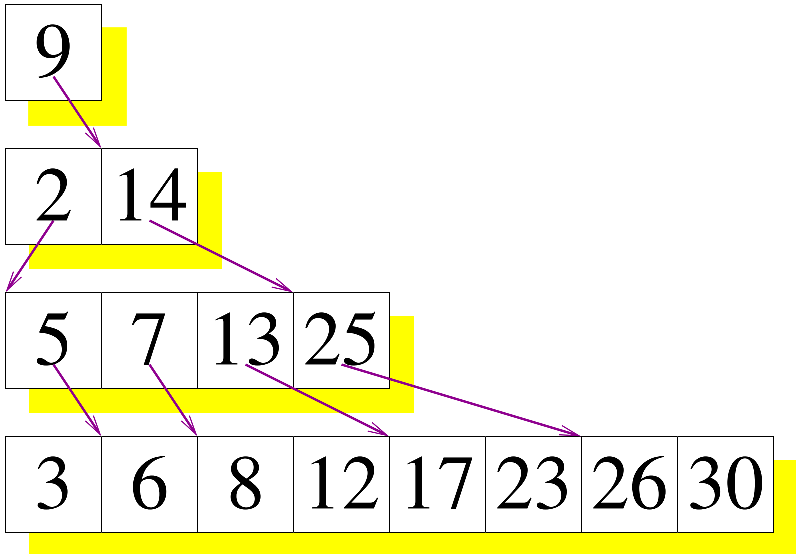
Speeding up Search

At $\log^2 N$, search is too expensive.
Now let's shave a factor of $\log N$.



The idea: Having searched an array for a row, we know where that row would belong in the array. We can gain information about where the row belongs in the next array

Forward Pointers



Each element gets a forward pointer to where that element goes in the next array using *Fractional Cascading*. [Chazelle, Guibas 1986]

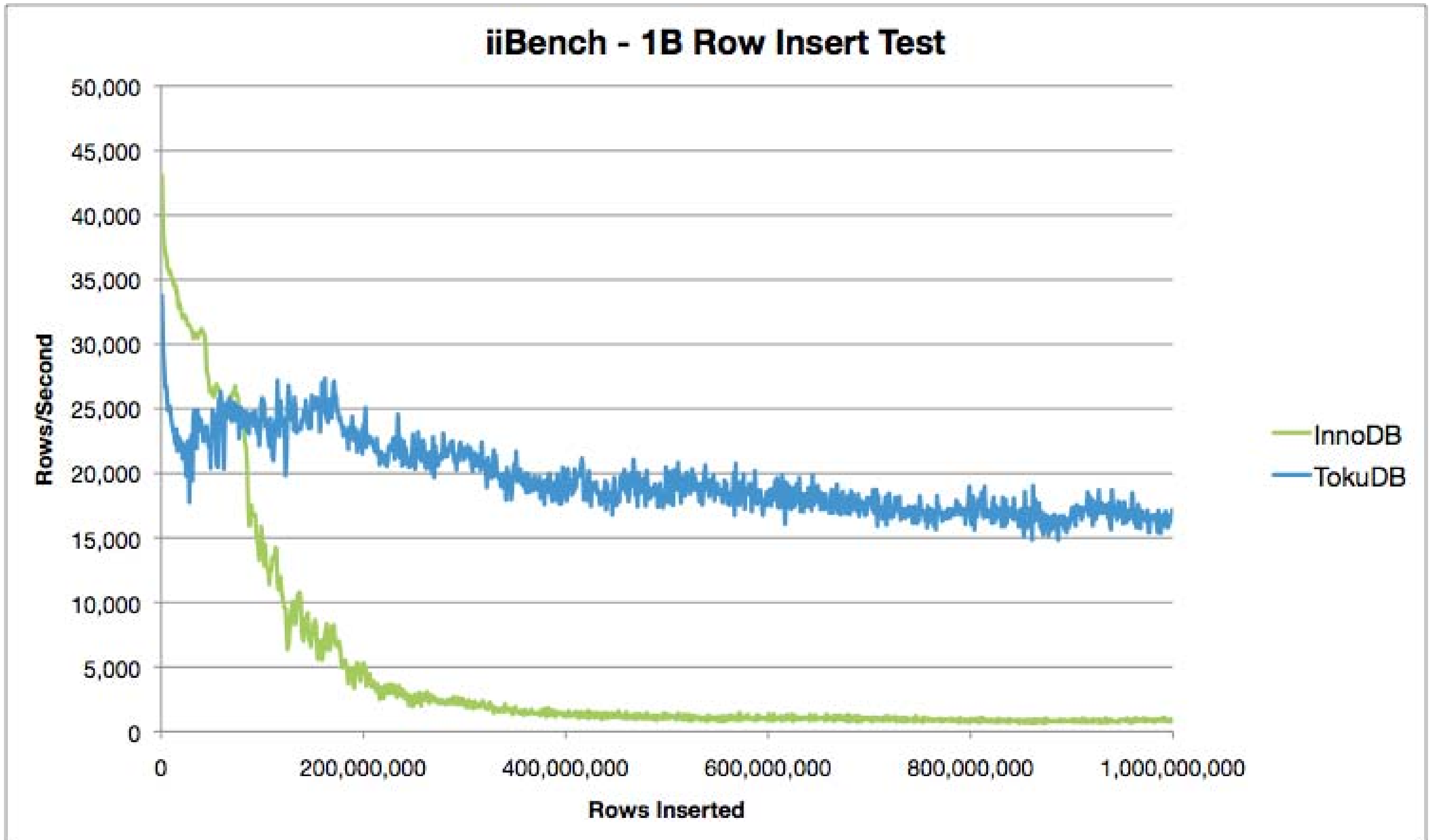
If you are careful, you can arrange for forward pointers to land frequently (separated by at most a constant). Search becomes $O(\log N)$ levels, each looking at a constant number of elements, for $O(\log N)$ I/Os.

Industrial-Grade Fractal Trees

A real implementation, like TokuDB, must deal with

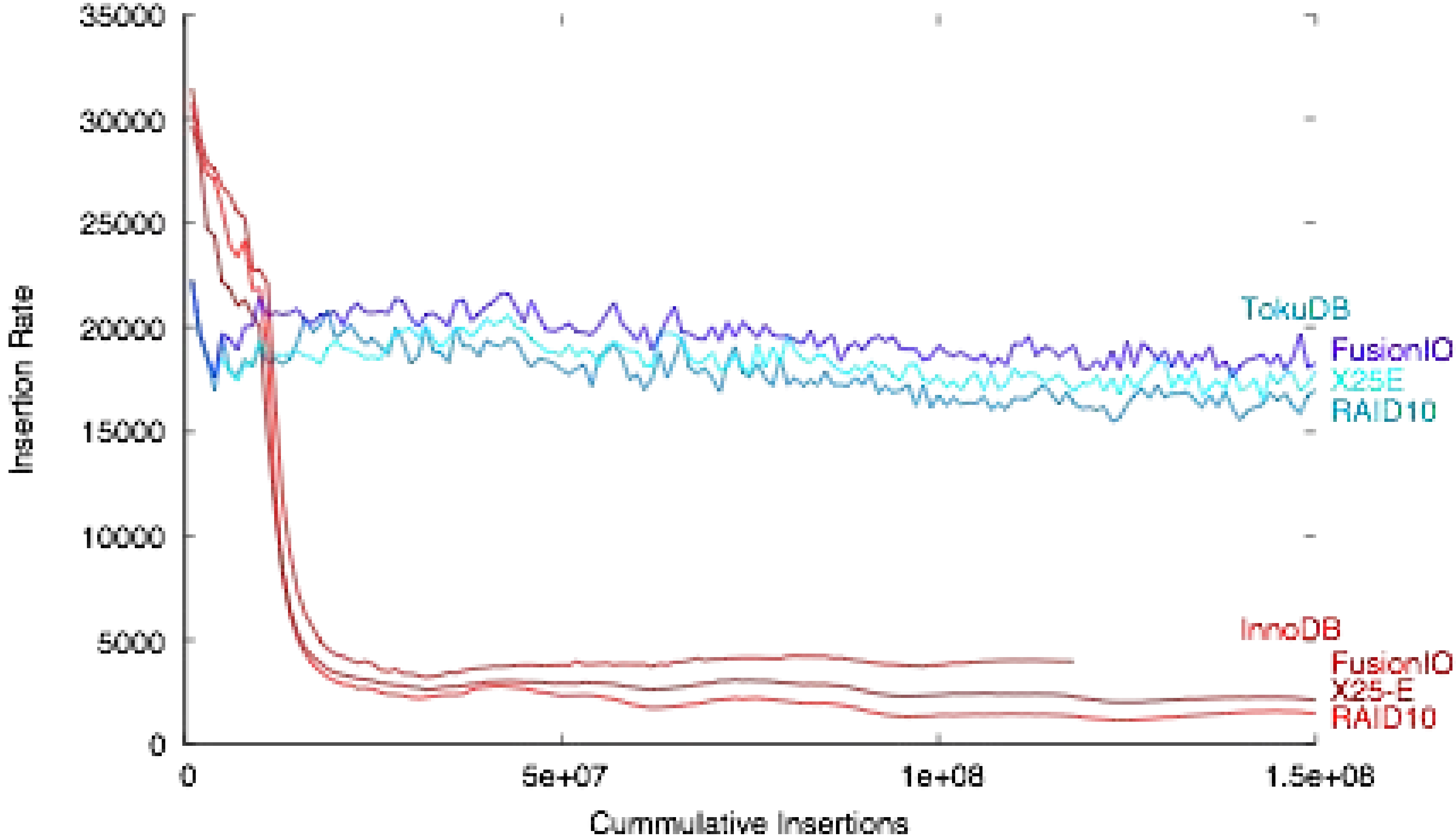
- Variable-sized rows;
- Deletions as well as insertions;
- Transactions, logging, and ACID-compliant crash recovery;
- Must optimize sequential inserts more;
- Better search cost: $O(\log_B N)$, not $O(\log_2 N)$;
- Compression; and
- Multithreading.

iiBench Insert Benchmark



iiBench was developed by us and Mark Callaghan to measure insert performance.
Percona took these measurements about a year ago.

iiBench on SSD



TokuDB on rotating disk beats InnoDB on SSD.

Disk Size and Price Technology Trends

- SSD is getting cheaper.
- Rotating disk is getting cheaper faster. Seagate indicates that 67TB drives will be here in 2017.
- Moore's law for silicon lithography is slower over the next decade than Moore's law for rotating disks.

Conclusion: big data stored on disk isn't going away any time soon.

Fractal Tree indexes are good on disk.

One cannot simply indexes in main memory. One must use disk efficiently.

Speed Trends

- Bandwidth off a rotating disk will hit about 500MB/s at 67TB.
- Seek time will not change much.

Conclusion: Scaling with bandwidth is good. Scaling with seek time is bad.

Fractal Tree indexes scale with bandwidth.

Unlike B-trees, Fractal Tree indexes can consume many CPU cycles.

Power Trends

- Big disks are much more power efficient per byte stored than little disks.
- Making good use of disk bandwidth offers further power savings.

Fractal Tree indexes can use 1/100th the power of B-trees.

CPU Trends

- CPU power will grow dramatically inside servers over the next few years. 100-core machines are around the corner. 1000-core machines are on the horizon.
- Memory bandwidth will also increase.
- I/O bus bandwidth will also grow.

Conclusion: Scale-up machines will be impressive.

Fractal Tree indexes will make good use of cores.

The Future

- Fractal Tree indexes dominate B-trees theoretically.
- Fractal Tree indexes ride the right technology trends.
- In the future, all storage systems will use Fractal Tree indexes.