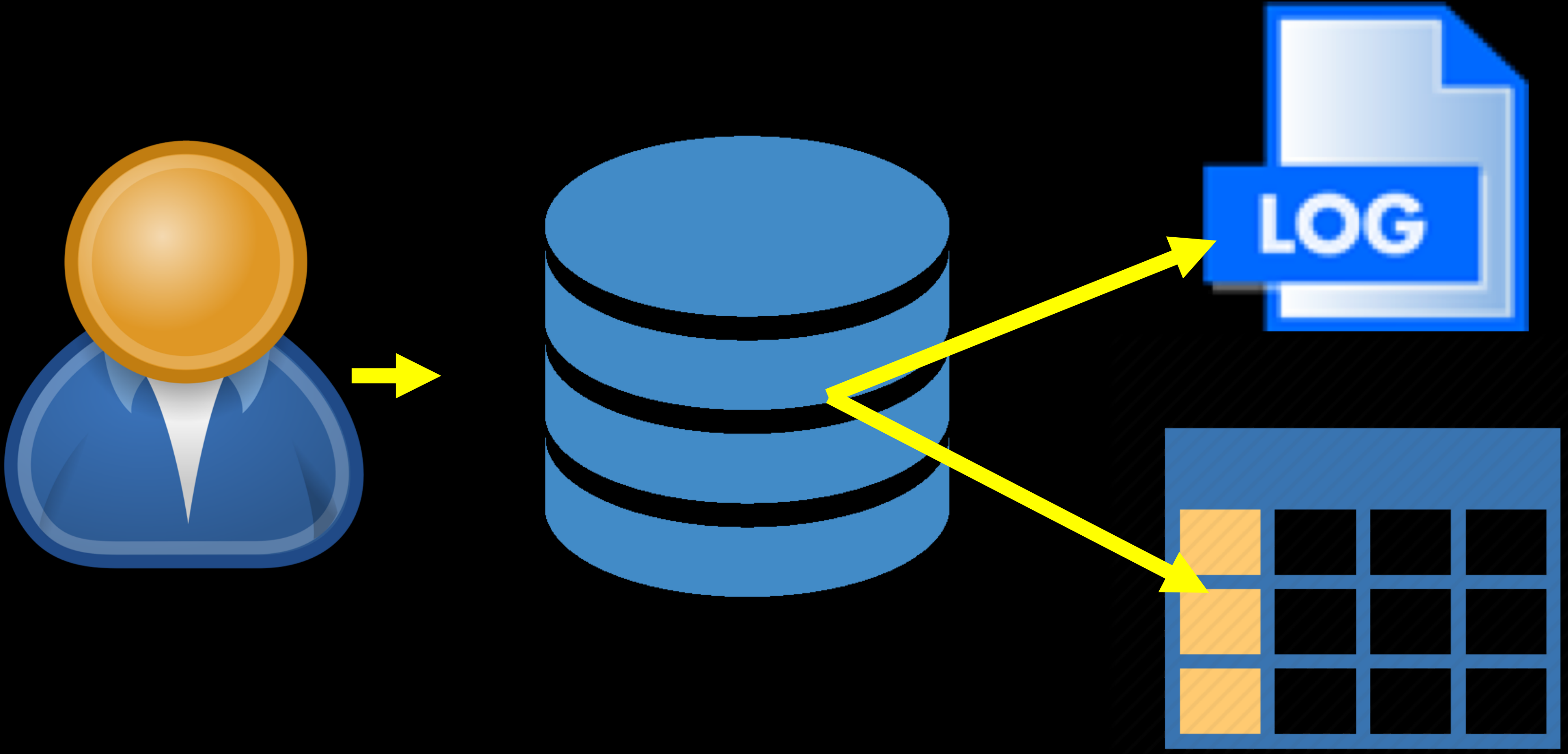
Real-time writes and reasonable reads: The LSM tree in C++

Stathis Karatsiolis & John C. Merfeld

Here’s what we’ll talk about

- Motivating example
- Design goals
- Implementation details (at least, the interesting bits)
- Experimental results

Imagine a DBMS maintaining a transaction log



What constraints apply to our log?

What constraints apply to our log?

- Must ultimately reside on disk

What constraints apply to our log?

- Must ultimately reside on disk (why?)

What constraints apply to our log?

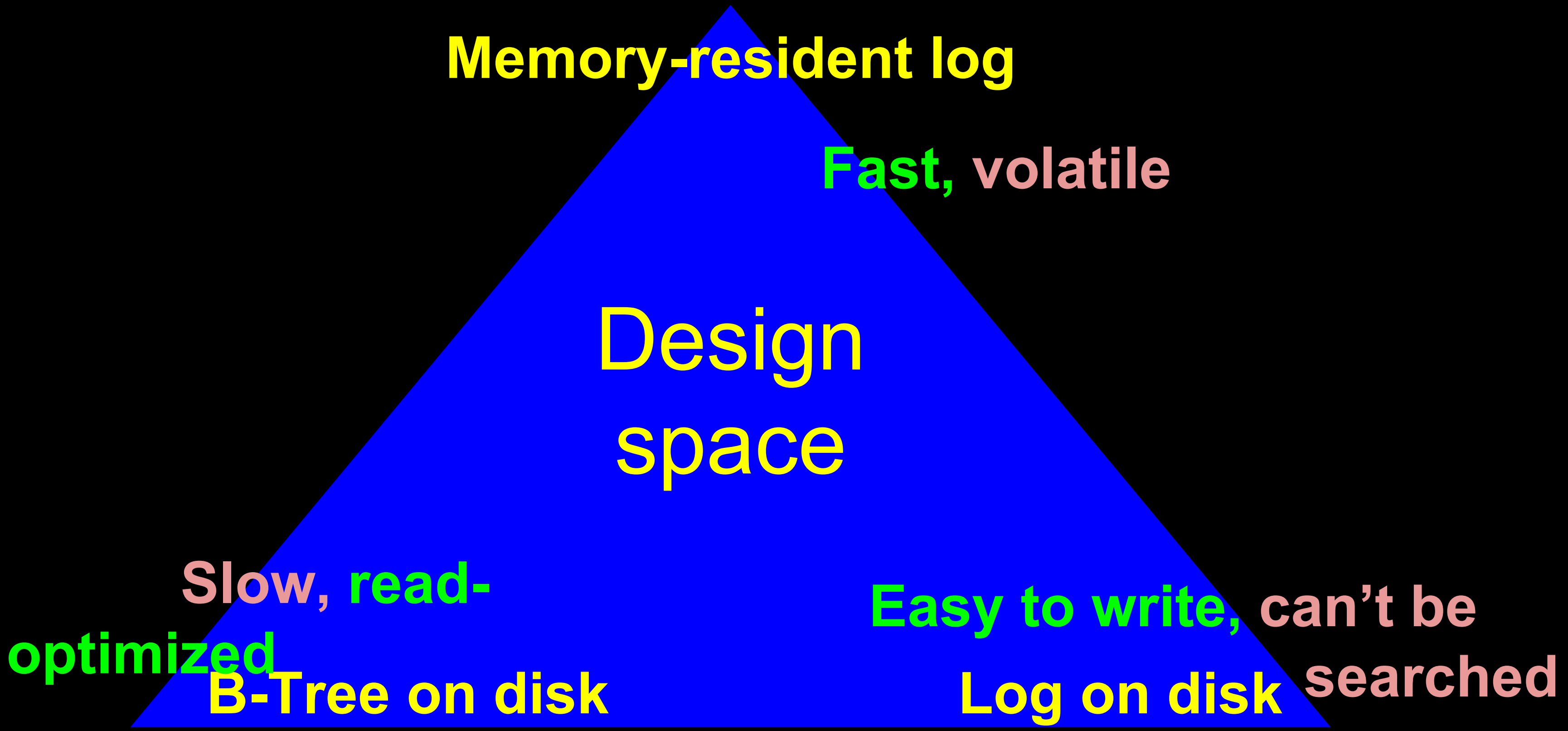
- Must ultimately reside on disk

- Must support efficient lookups

What constraints apply to our log?

- Must ultimately reside on disk
- Must support efficient lookups
- Should not interfere with transaction performance

We can’t perfect all three constraints at once



Memory-resident log

Fast, volatile

Design space

Slow, read-optimized

B-Tree on disk

Easy to write, can’t be searched

Log on disk

We can integrate the benefits of all three designs

- Hold data in memory for as long as possible
- Use some hierarchy and some sorting on disk data
- Keep some lightweight metadata in memory

We can integrate the benefits of all three designs

- Hold data in memory for as long as possible
- Use **some** hierarchy and **some** sorting on disk data
- Keep some lightweight metadata in memory
- "How much is **some**?"

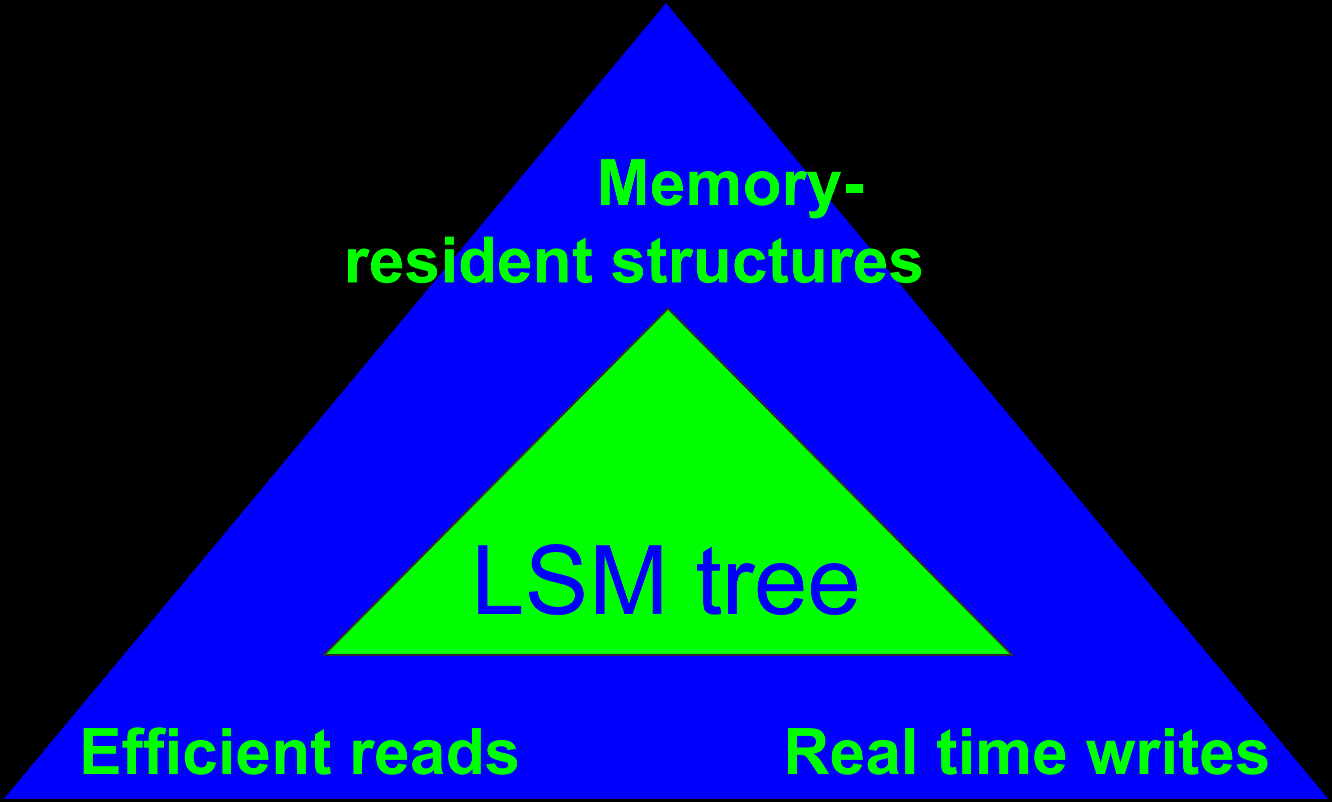
We implemented a log-structured merge tree

- Hold updates in memory

- Merge them to a disk index in batches

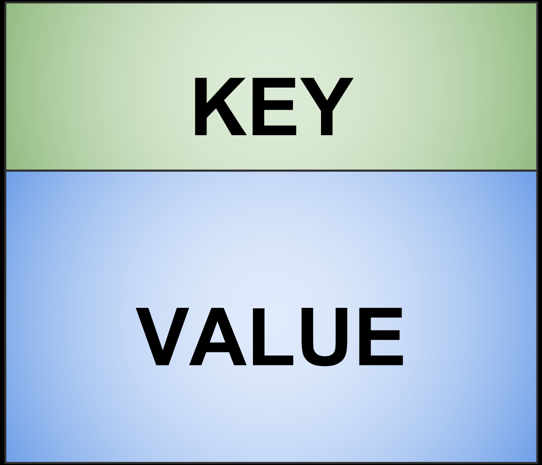
- Retain metadata to assist lookups

The LSM tree fulfills our design goals



OUR IMPLEMENTATION

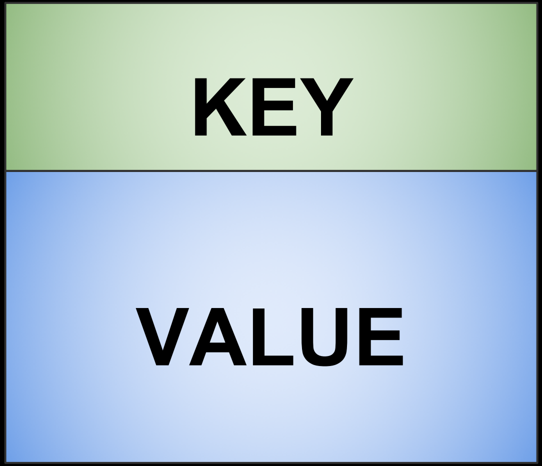
We built a key-value store for integers



KEY

VALUE

We built a key-value store for integers



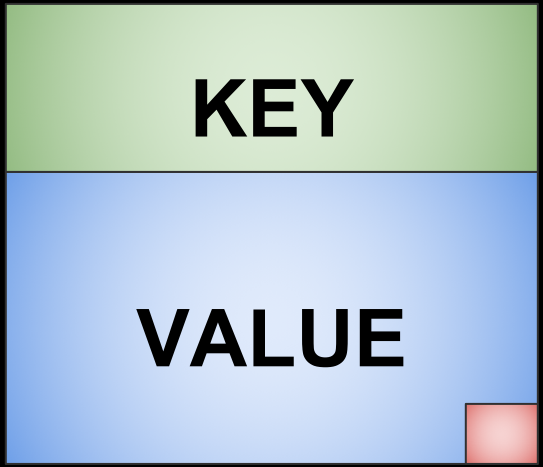
KEY

VALUE

We call this an “Entry”

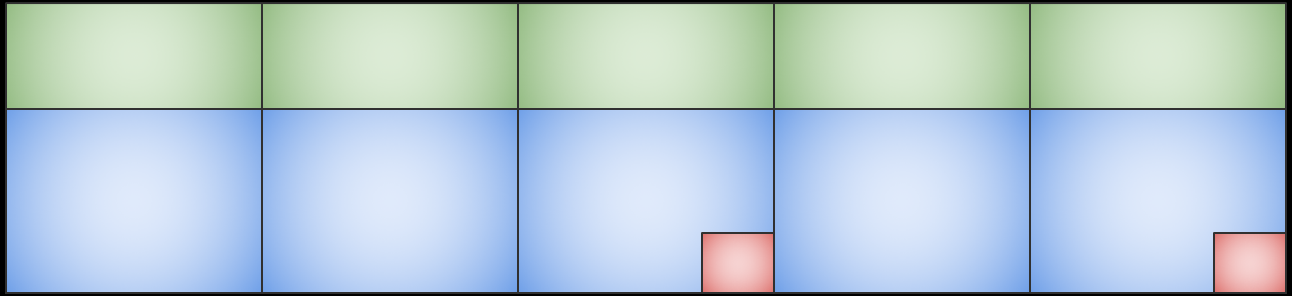
For our purposes, the key and the value were always the same number

We built a key-value store for integers



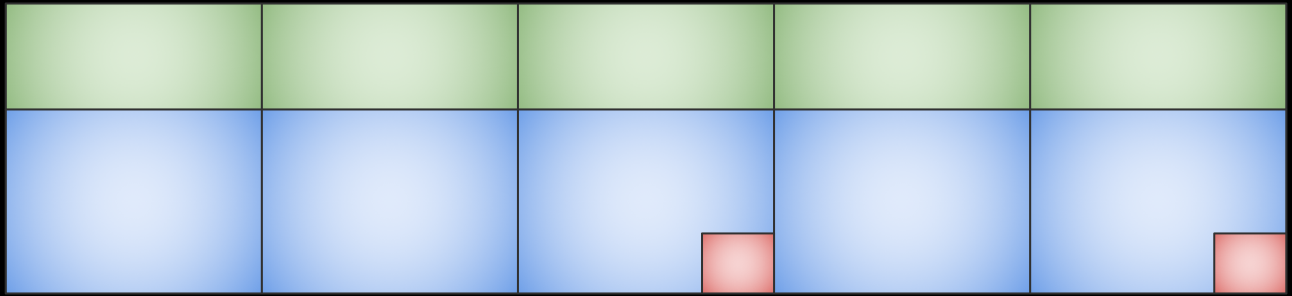
Some entries have a flag indicating they are a delete

In memory, we hold an array of entries



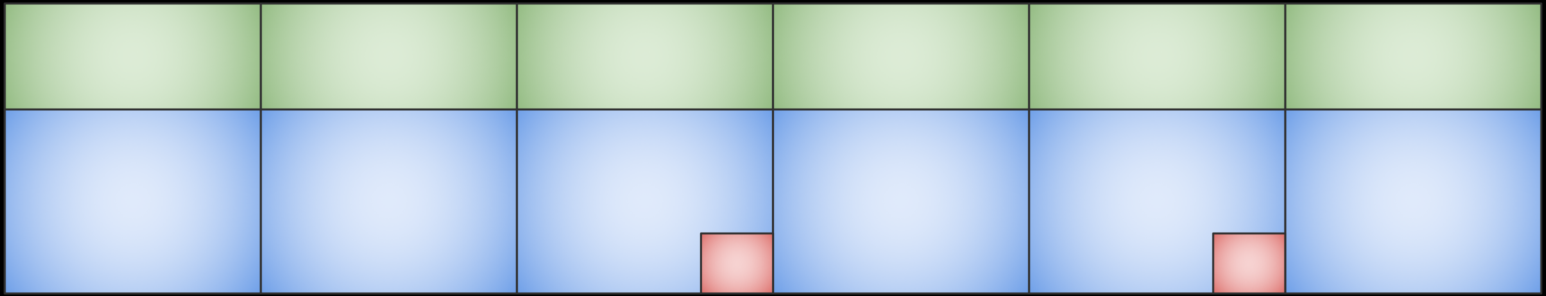
We call this a "Run"

In memory, we hold an array of entries



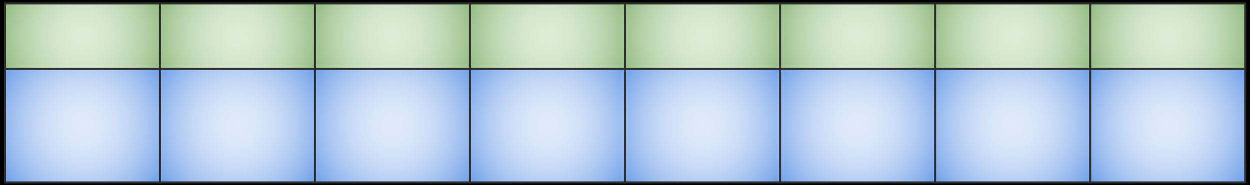
- insert()

In memory, we hold an array of entries

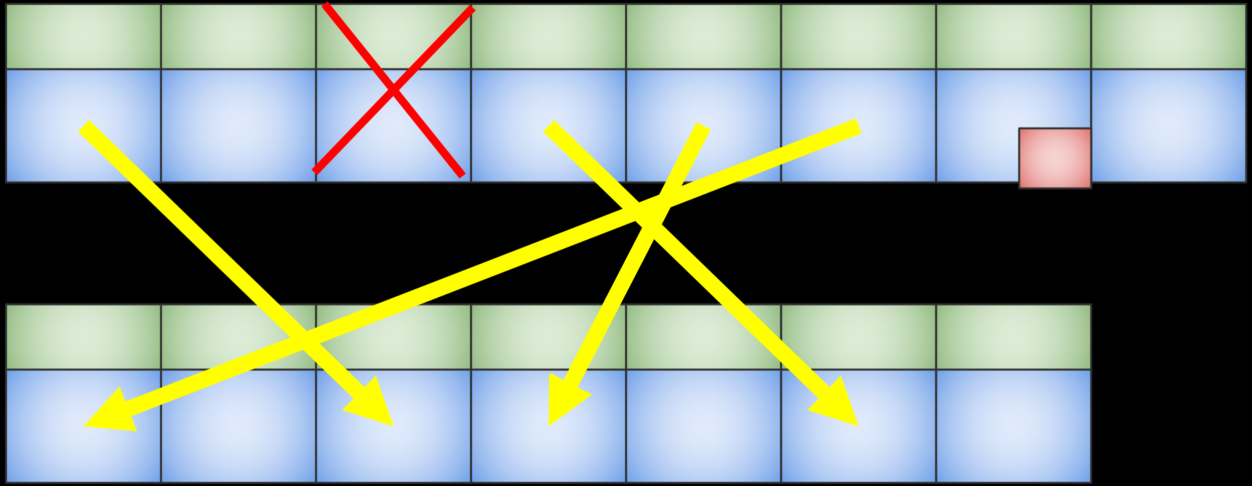


- insert()

When memory fills, we create some metadata

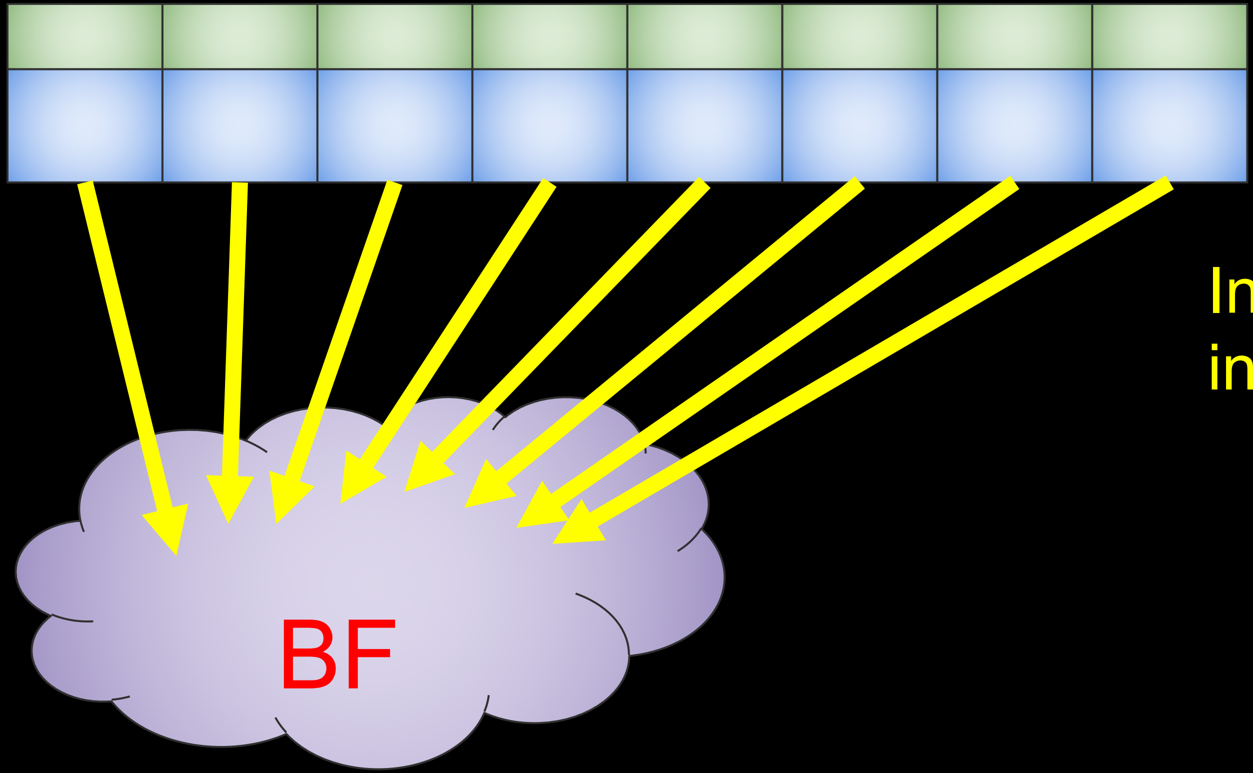


When memory fills, we create some metadata



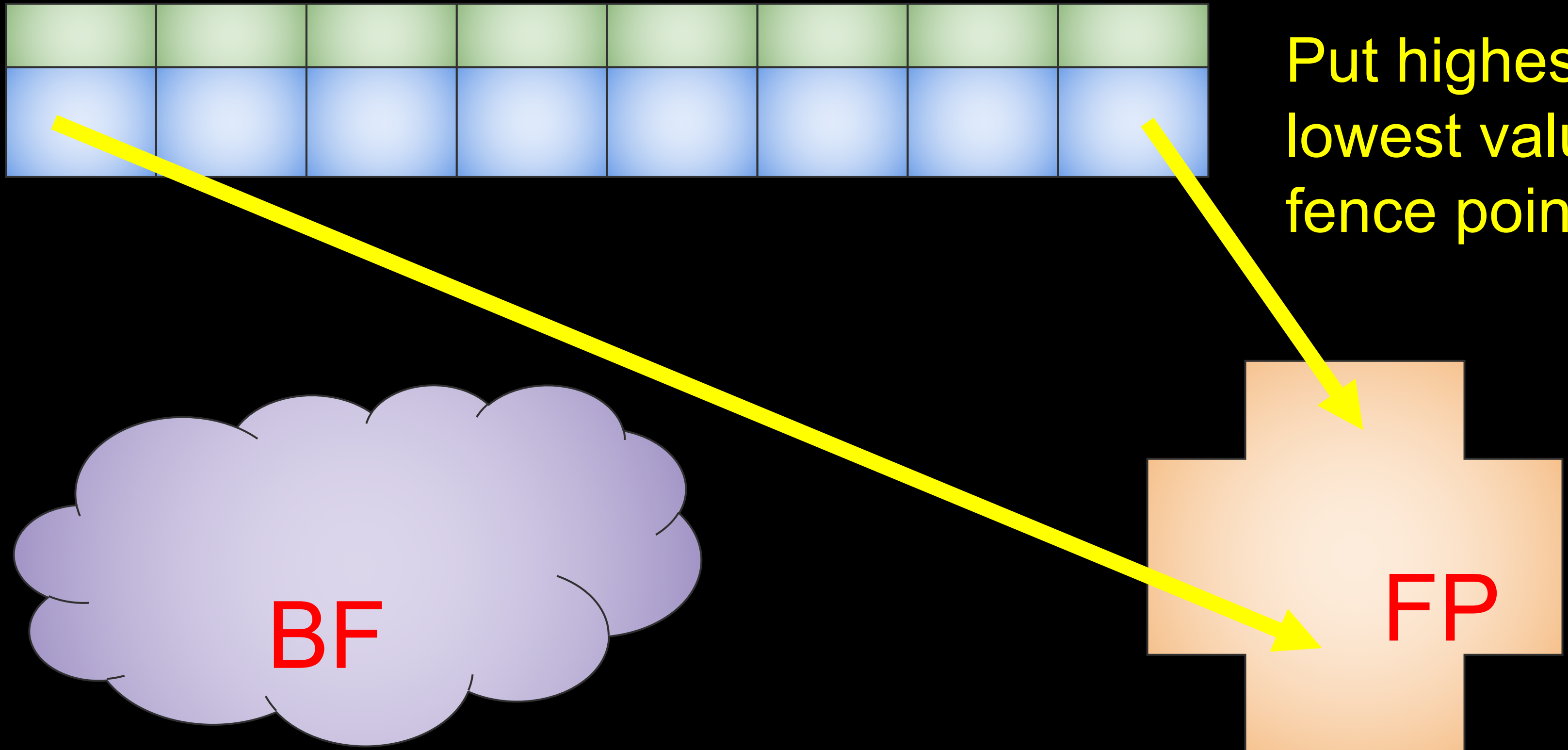
Sort, remove duplicates

When memory fills, we create some metadata



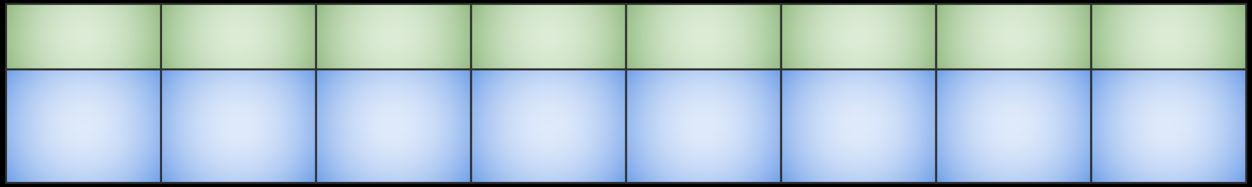
Insert everything
into a bloom filter

When memory fills, we create some metadata

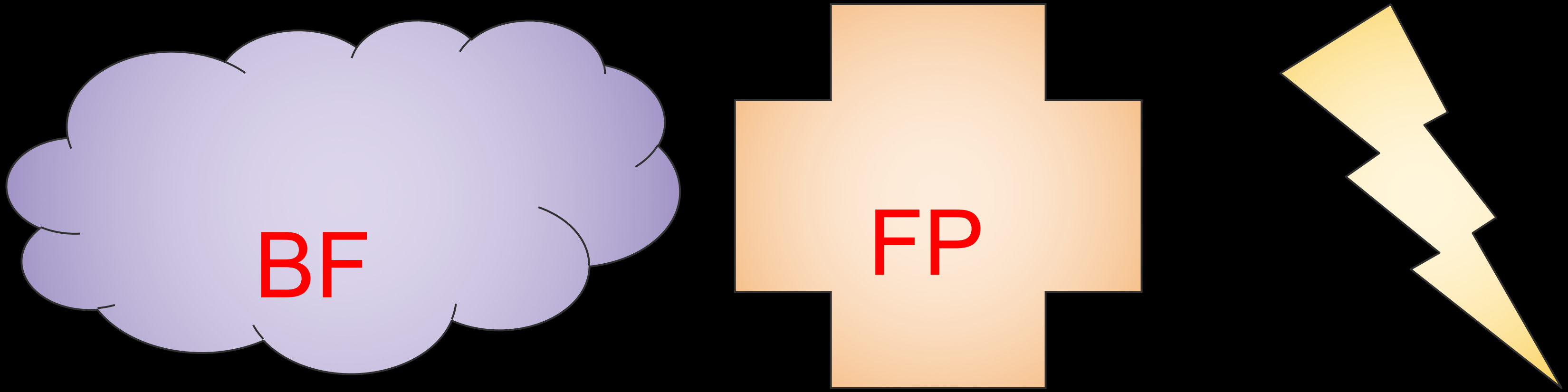


Put highest and lowest values in a fence pointer

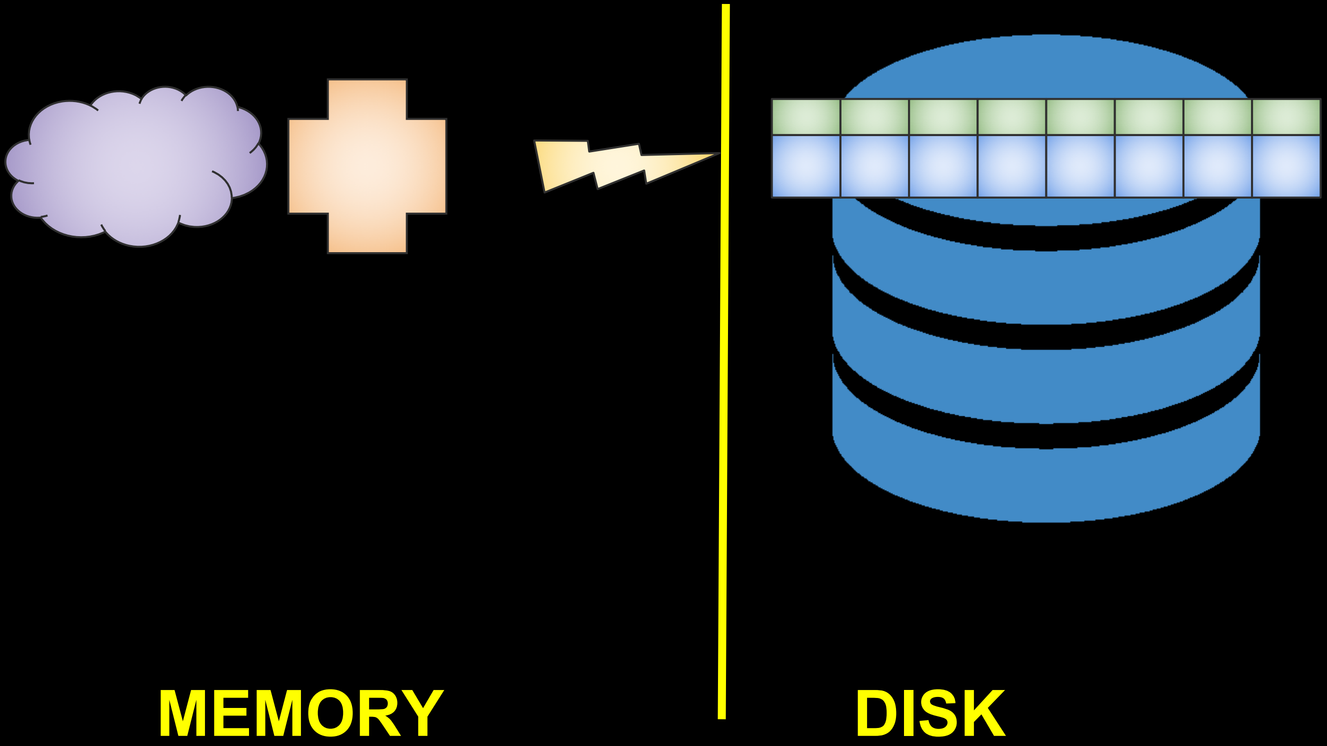
When memory fills, we create some metadata



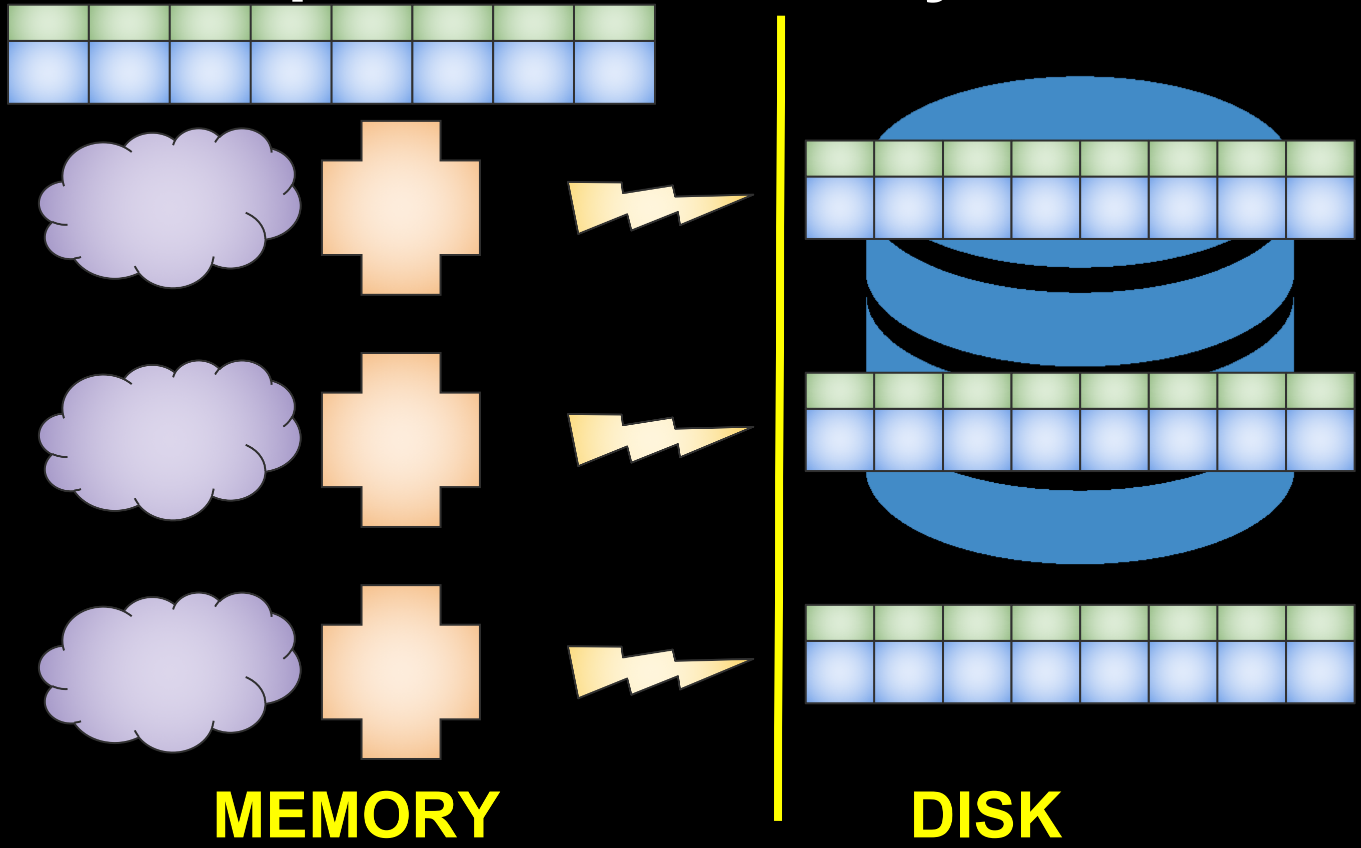
Generate a filename and get a pointer to a disk file



We write our run to a file and keep the metadata



At its simplest, this is our system!



How does this fulfill our design goals?

- Inserts, updates, deletes just append to a memory array (Real time writes!)

How does this fulfill our design goals?

- Inserts, updates, deletes just append to a memory array
  (Real time writes!)

- Sorted runs on disk prevent full scans
  (Reasonable reads!)

How does this fulfill our design goals?

- Inserts, updates, deletes just append to a memory array
  (Real time writes!)

- Sorted runs on disk prevent full scans
  (Reasonable reads!)

- Metadata allow for data-skipping during queries
  (Memory-resident structures!)

Don’t worry, it’s still a tree

- Hold the metadata in a 2D array

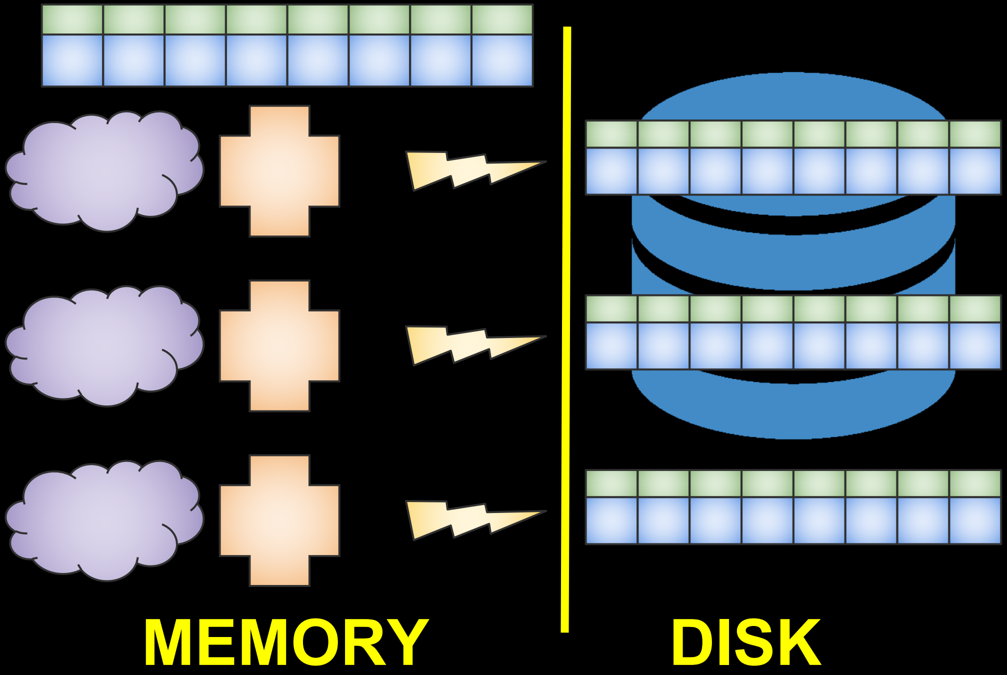
- When a row of the array fills:
    - Load its runs into memory and sort-merge them

    - Consolidate the metadata and write to new file

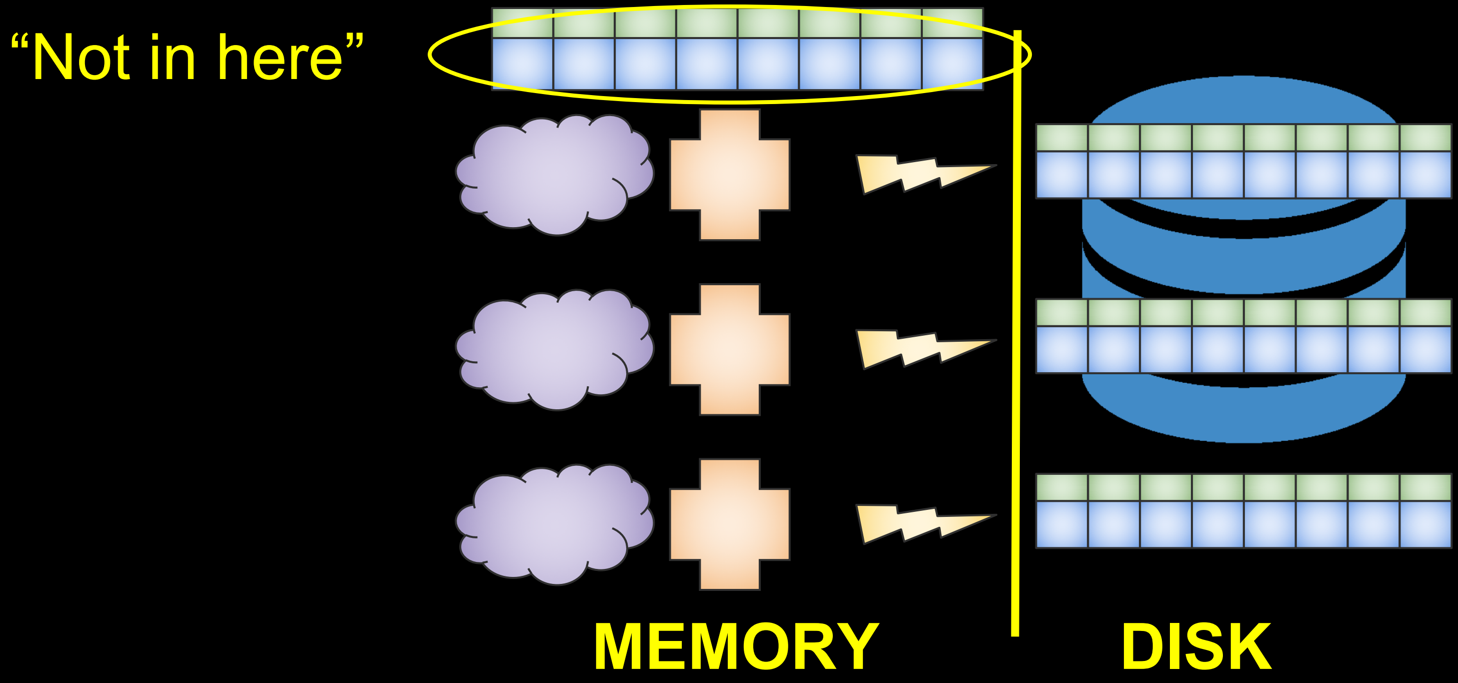
    - Push the metadata down a level in the array

Queries operate about how you'd expect

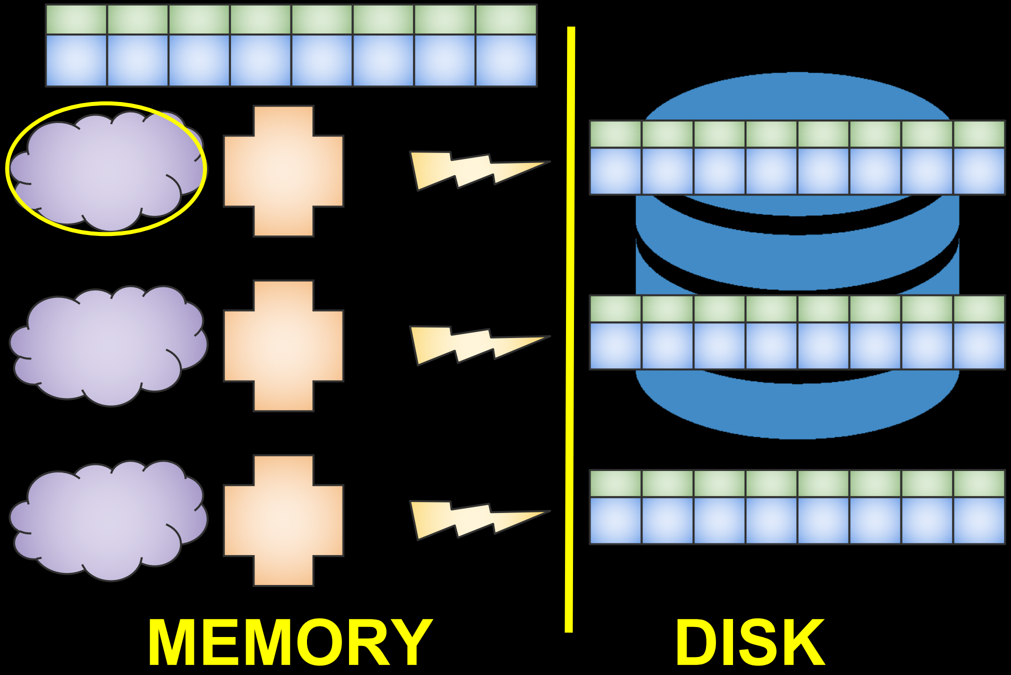
First ask memory,
then examine disk
runs as needed



Queries operate about how you'd expect



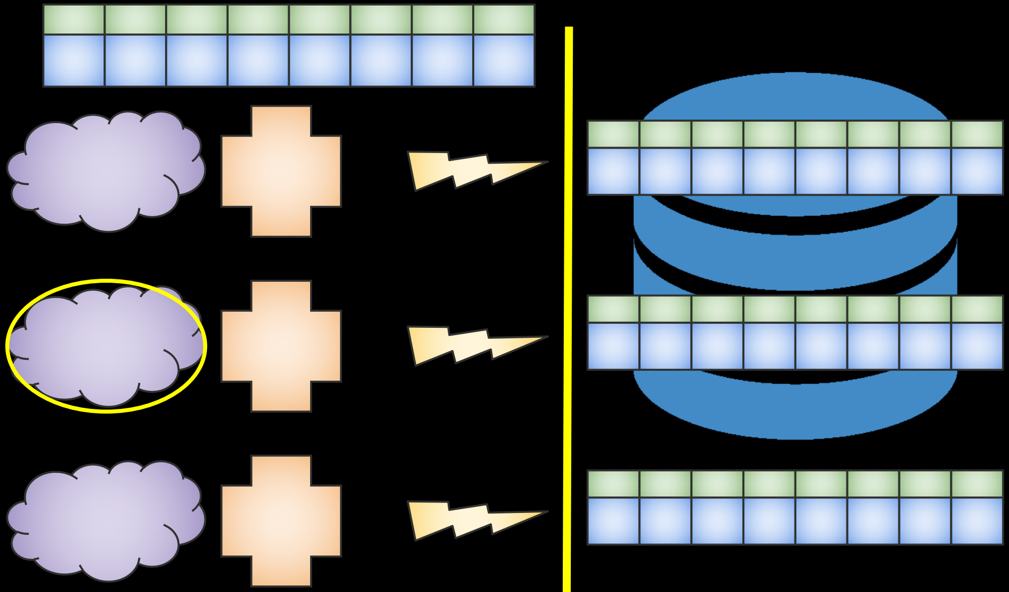
Queries operate about how you'd expect



"Not in here"

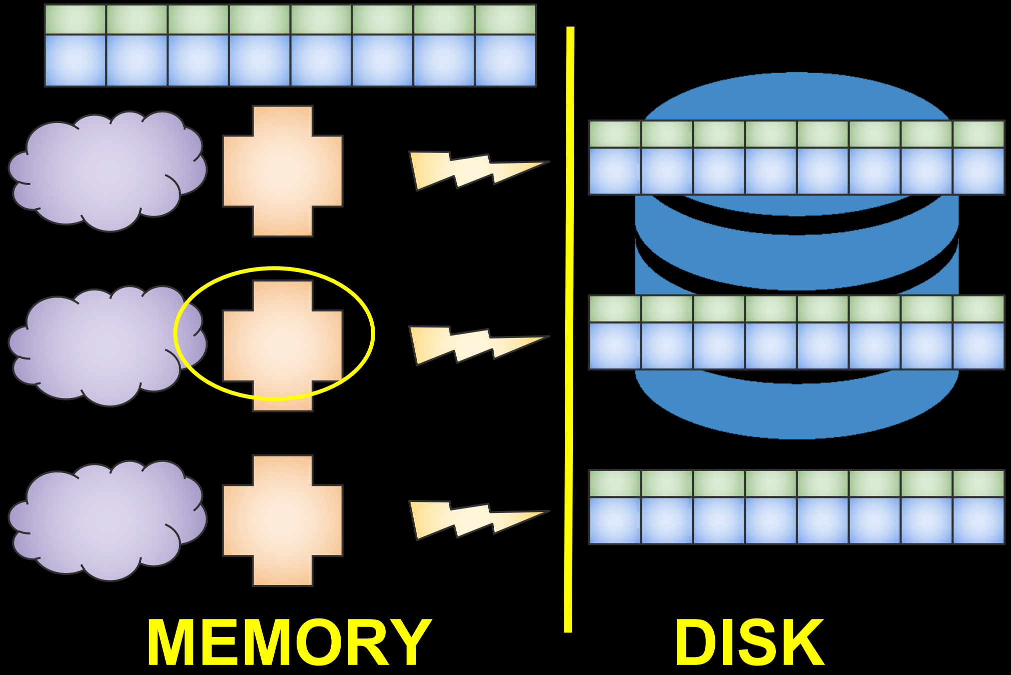
Queries operate about how you’d expect

“Might be in here”



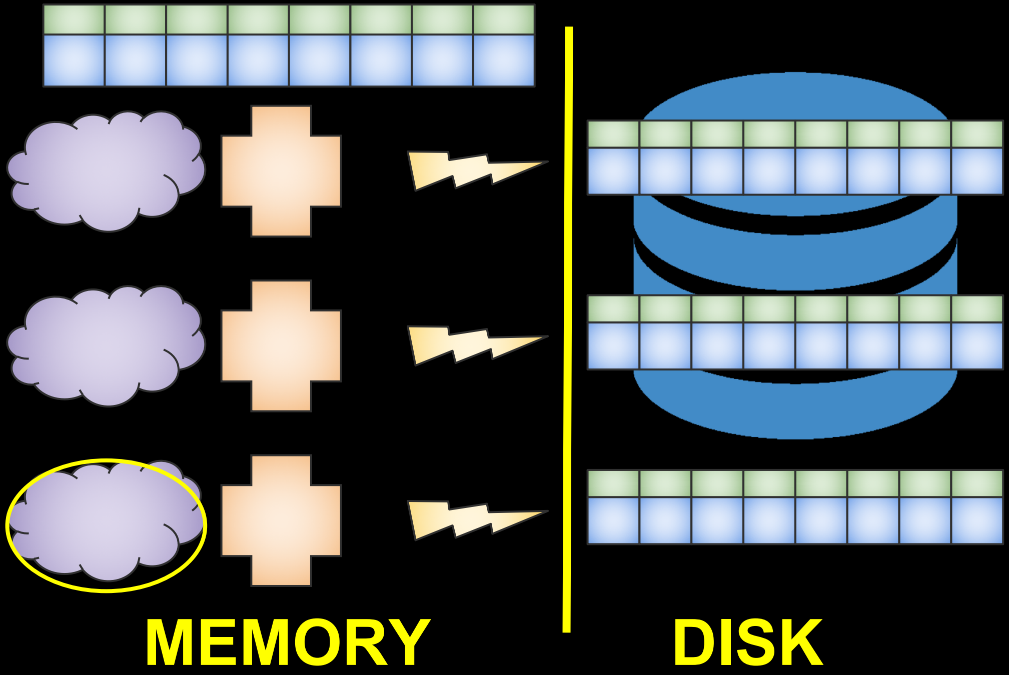
Queries operate about how you'd expect

“Nope, not in range”



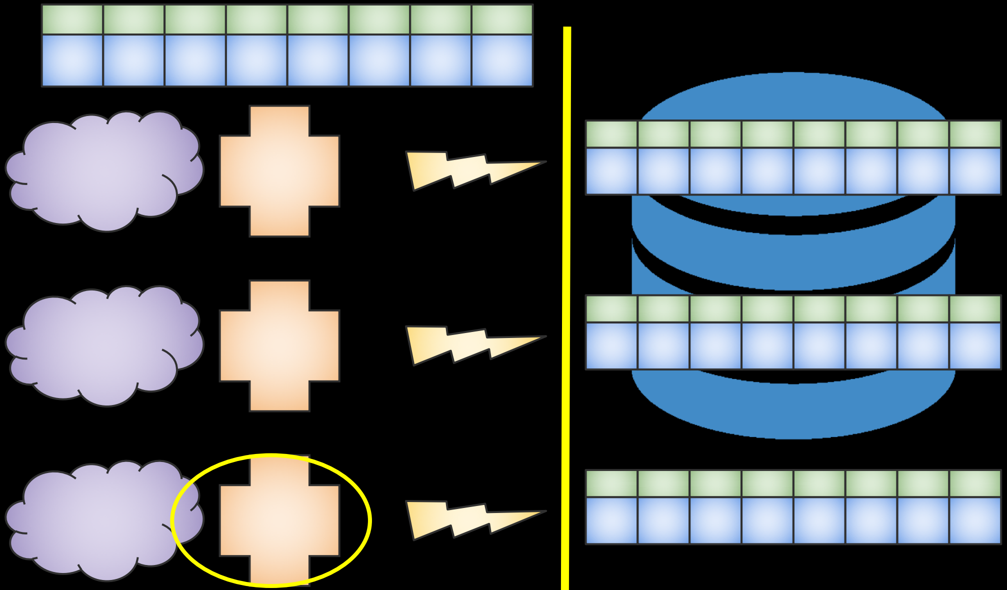
Queries operate about how you’d expect

“Might be in here”

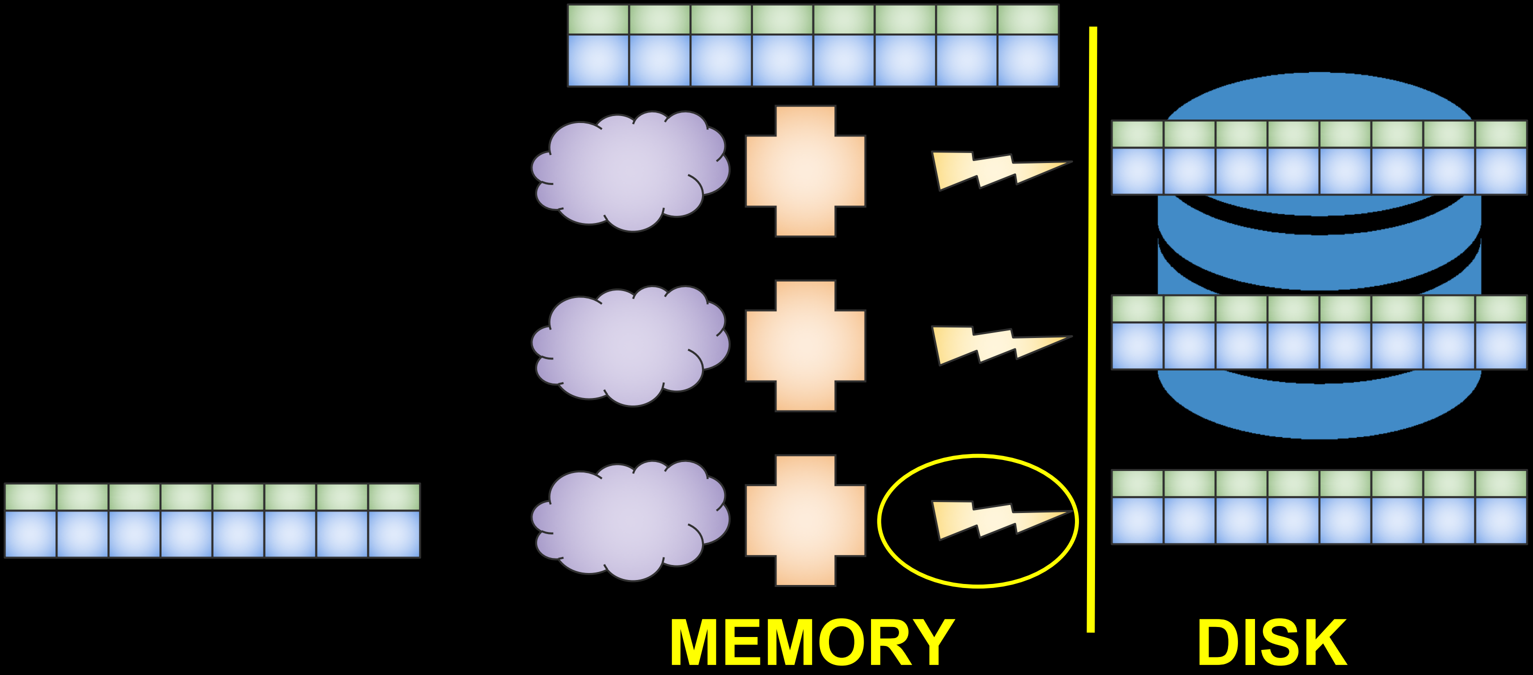


Queries operate about how you'd expect

“In my range”

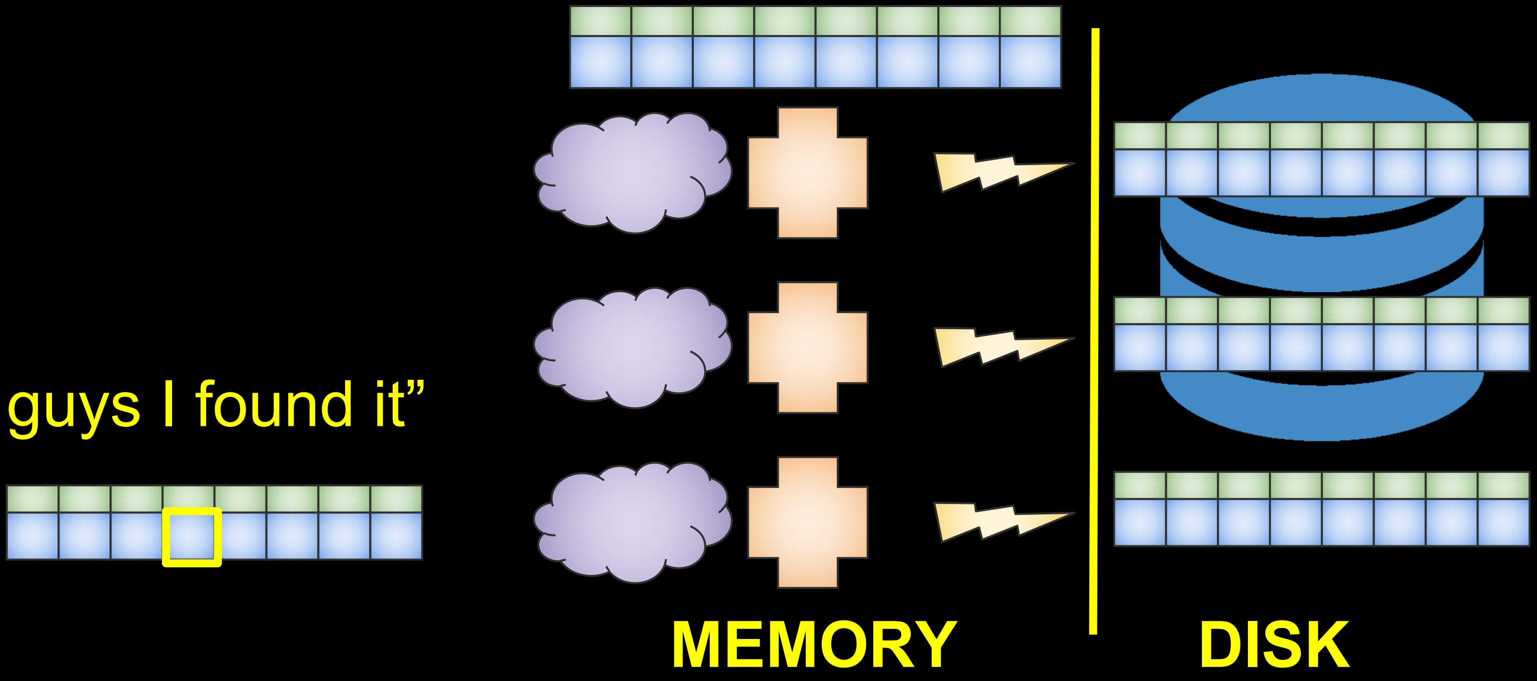


Queries operate about how you'd expect

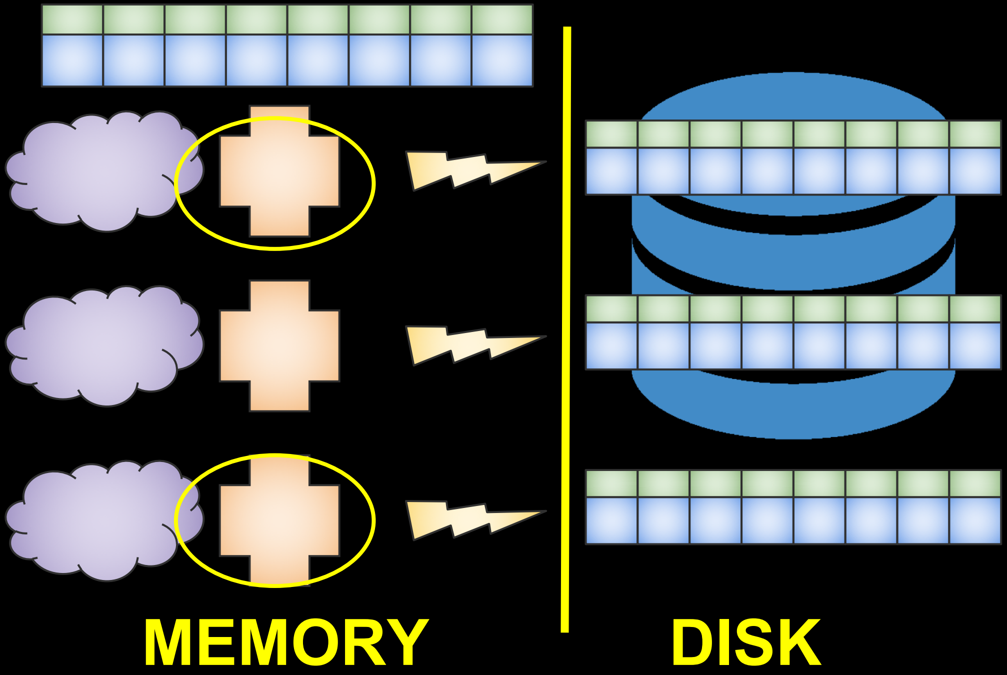


Queries operate about how you'd expect

“Hey guys I found it”

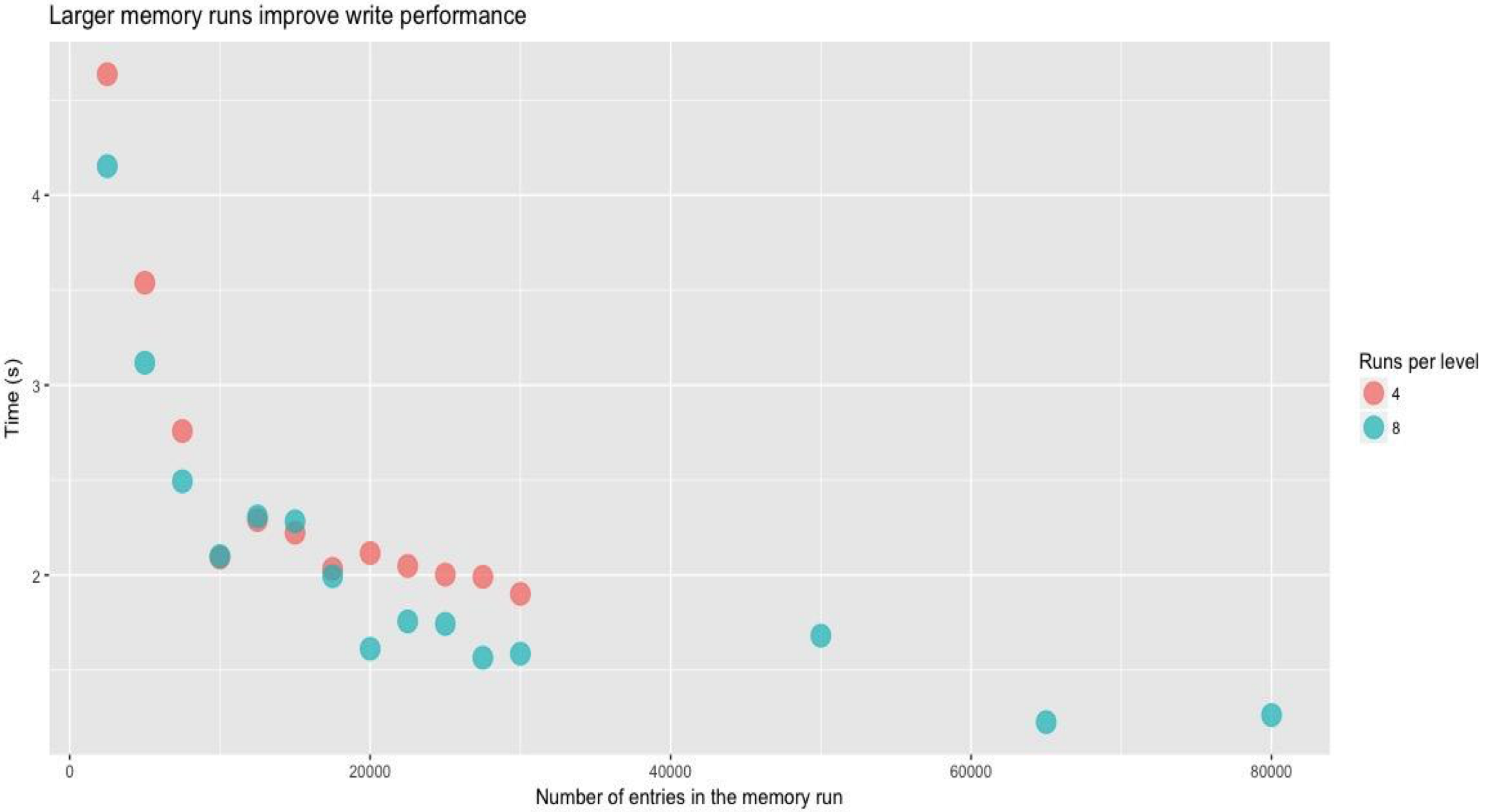


Range queries check every run whose fence pointer overlaps with the query range

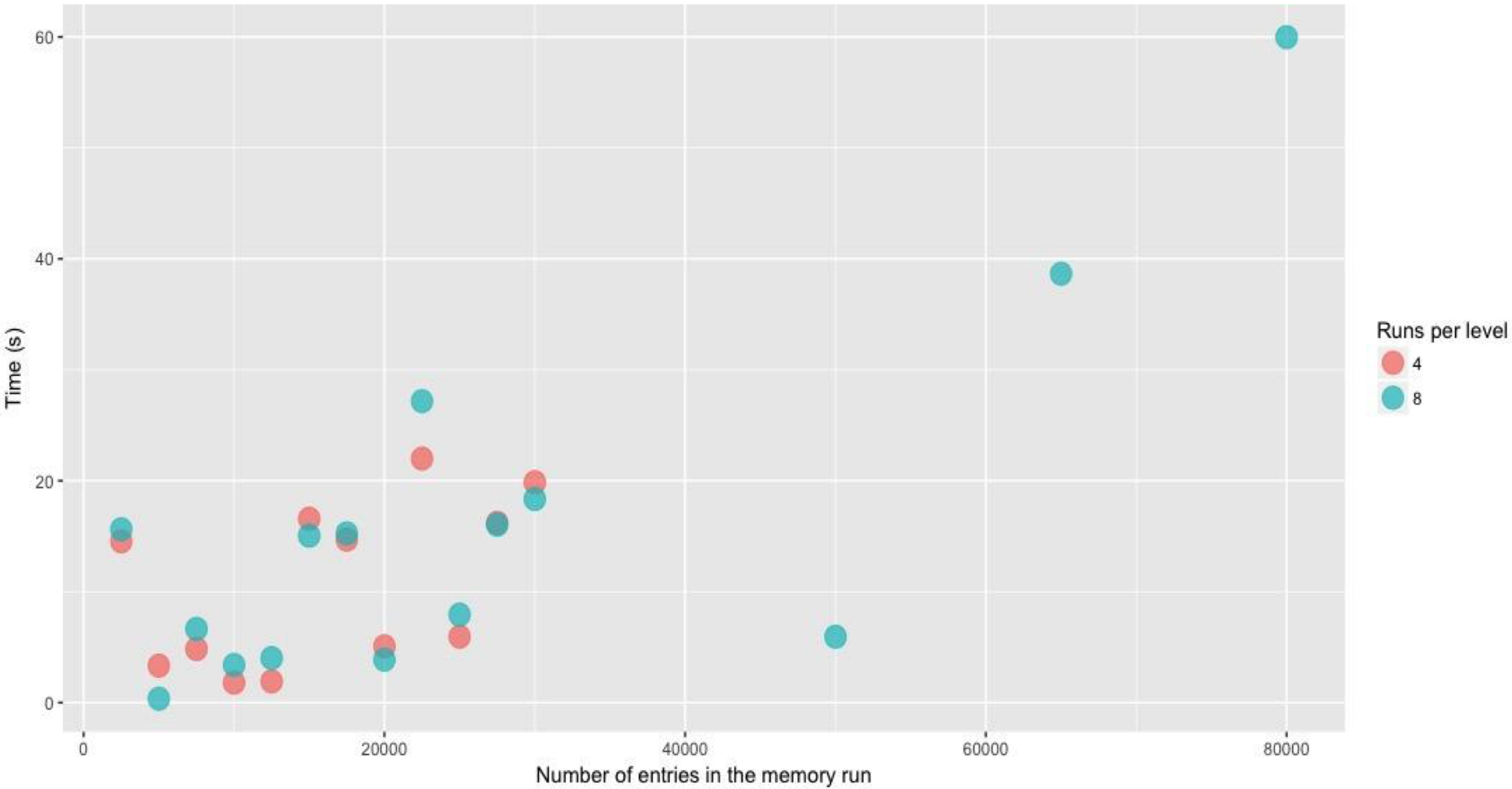


EXPERIMENTAL EVALUATION

(Or, what happened once we got it to compile)


Larger memory runs improve write performance
Time (s)
4
3
2
0
20000
40000
60000
80000
Number of entries in the memory run
Runs per level
4
8

The relationship with read performance is less clear



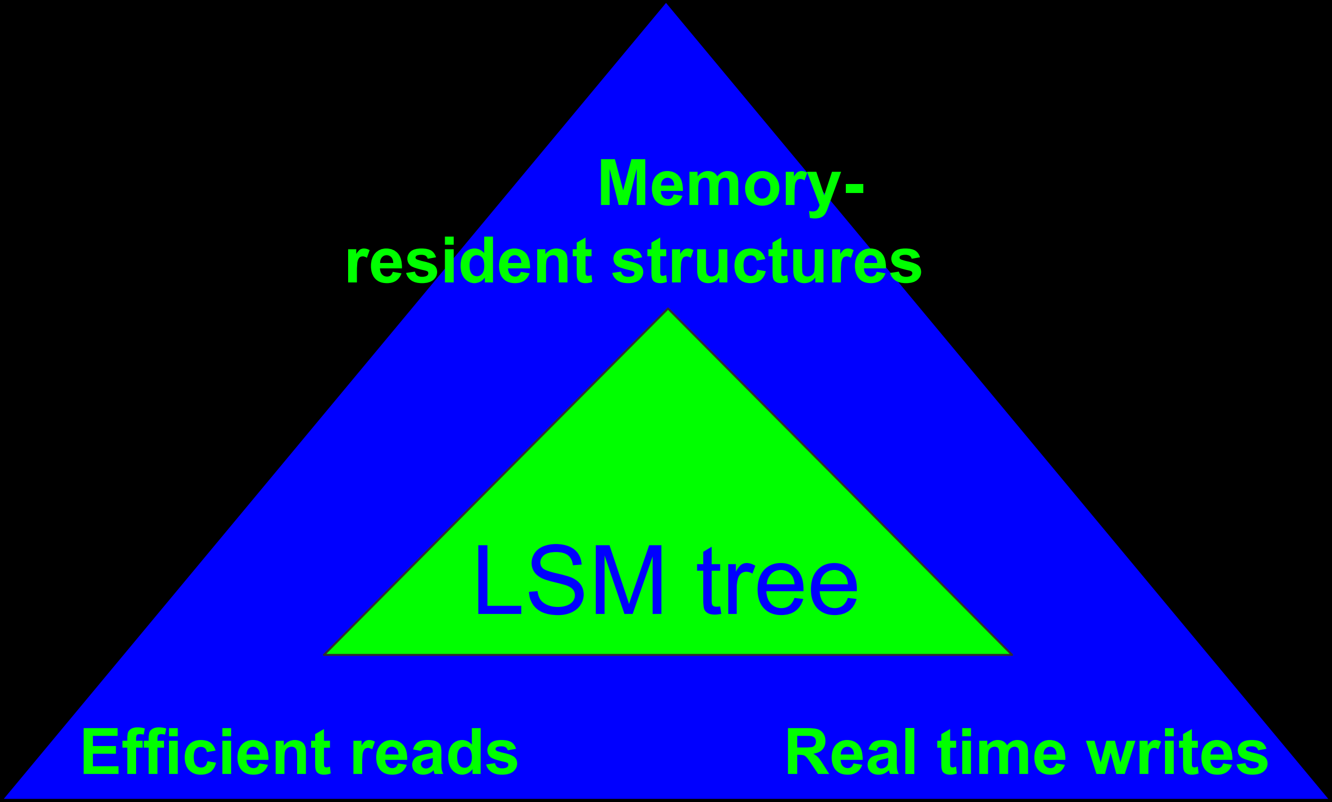
We have theories about the poor read performance

- Pages sizes might not perfectly align with the sizes of our Memory Runs

- Set of Fence Pointers per run vs Set of Fence Pointers per Page in a Run

- Sequential scan of Disk Run vs Binary Search

In conclusion, recall our design goals



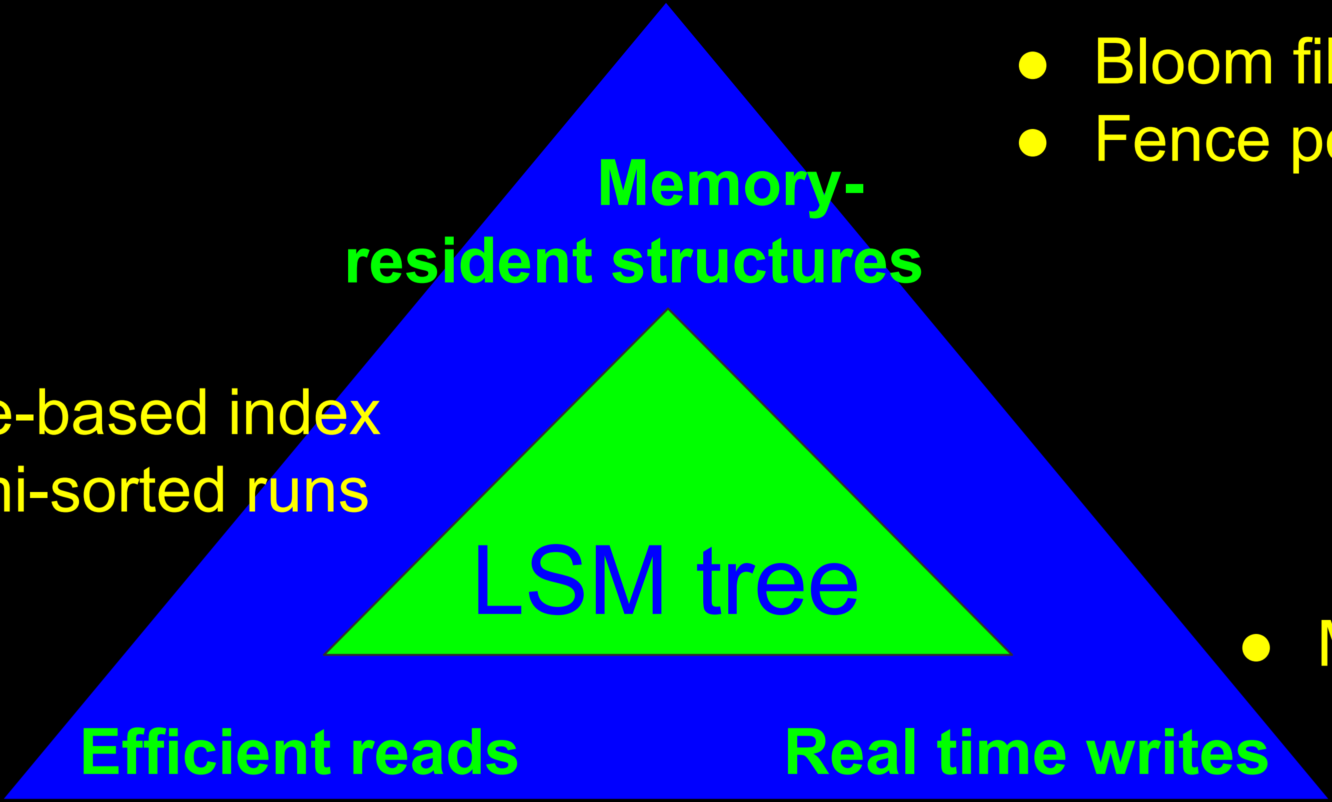
Memory-resident structures

LSM tree

Efficient reads

Real time writes

In conclusion, recall our design goals



Memory-resident structures

LSM tree

Efficient reads

Real time writes

- Bloom filters
- Fence pointers

- Tree-based index
- Semi-sorted runs

- Memory run

There are some obvious next steps for us

- Implement leveled tree
- Fix read performance issues
- Refine experiments to identify bottlenecks

Here’s who did what, in very broad terms

STATHIS:

- Reading and writing to files, backends for metadata and tree restructuring

- Experimental setup and execution

JOHN C:

- Tree API, navigating the tree during queries,, and operations on runs

- Code for benchmarking and visualization