https://hao-ai-lab.github.io/dsc204a-w24/

DSC 204A: Scalable Data Systems
Winter 2024

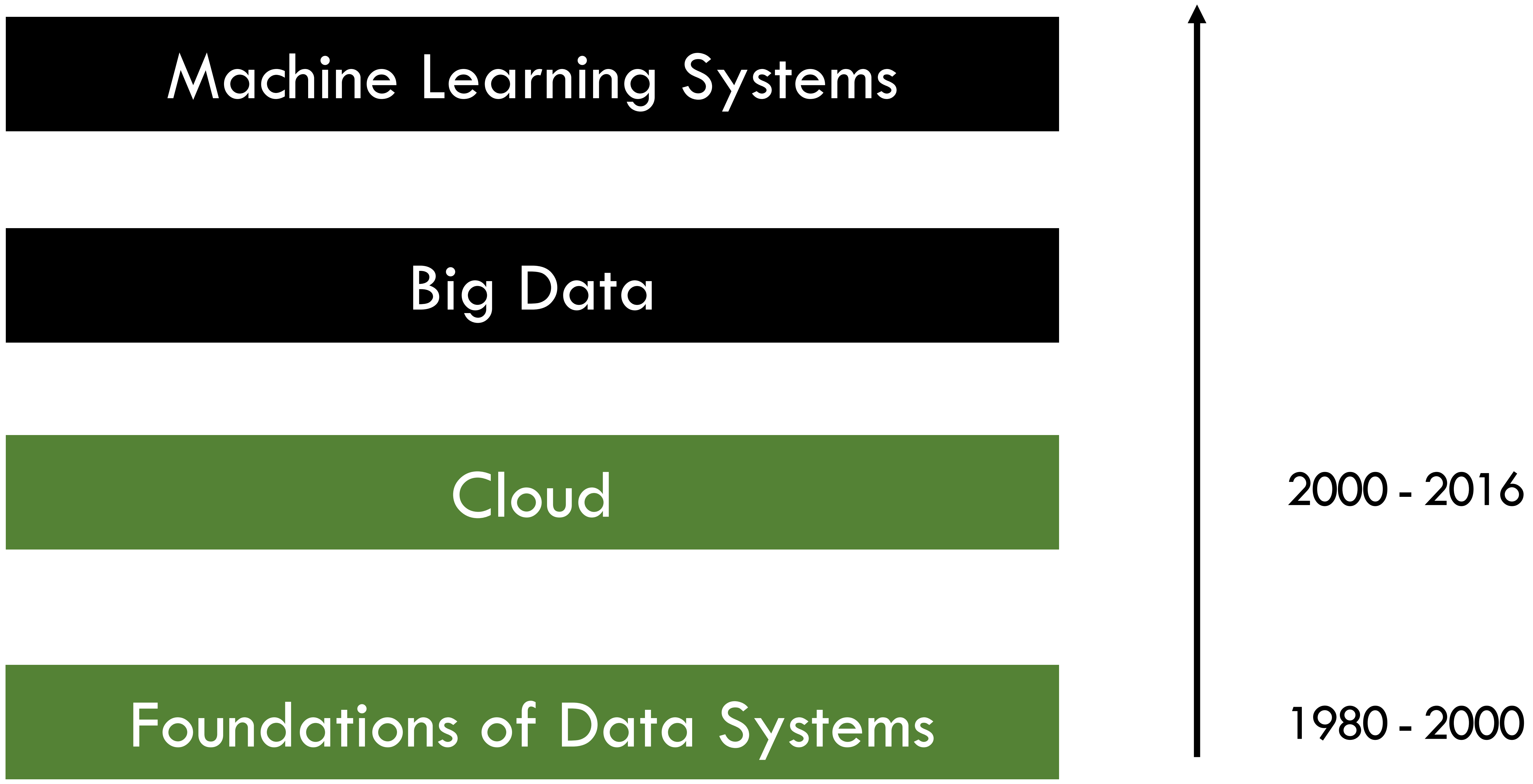
Machine Learning Systems

Big Data

Cloud

Foundations of Data Systems

Where We Are



Machine Learning Systems
Big Data
Cloud
2000 - 2016
Foundations of Data Systems
1980 - 2000

Recap: Collective Pros

- A set of structured / well-defined communication primitives
- Easy to analyze and to understand its performance
- Extremely well-optimized (over the last 40 years)
- Easy to program

Collective Cons

- Lack of Fault Tolerance
  - What if one node (in the ring) is dead?
- Requires **Homogeneity**
  - What if one node computes slower than all other nodes?
  - What if one link has lower bandwidth than the other node?

Real Cluster:

- Need Strong Fault tolerance
- **Heterogeneous** hardware setup

We will come back to this

- **Next week: Parallelism and Big Data processing**
  - We will delve deep to study how we address the drawbacks of Collectives – distributed computing ***with*** fault tolerance

Where we are

Motivations, Economics, Ecosystems, Trends

Cloud

Networking

~~Datacenter networking~~

Collective communication

Storage

(Distributed) File Systems / Database

Cloud storage

Part3: Compute

Distributed Computing

Big data processing

Next: File System, Database, Cloud Storage

- **File system**
- Database
- Column Storage and Data Warehouse

Q: *What is a file?*


MORTGAGE
INVESTMENTS
SAVINGS

Abstractions: File and Directory

- File: A persistent sequence of bytes that stores a logically coherent digital object for an application
  - File Format: An application-specific standard that dictates how to interpret and process a file's bytes
  - 100s of file formats exist (e.g., TXT, DOC, GIF, MPEG); varying data models/types, domain-specific, etc.
  - Metadata: Summary or organizing info. about file content (aka *payload*) stored with file itself; format-dependent
- Directory: A cataloging structure with a list of references to files and/or (recursively) other directories
  - Typically treated as a special kind of file
  - Sub dir., Parent dir., Root dir.

Filesystem

- Filesystem: The part of OS that helps programs create, manage, and delete files on disk (sec. storage)
- Roughly split into *logical level* and *physical level*
  - Logical level exposes file and dir. abstractions and offers System Call APIs for file handling
  - Physical level works with disk firmware and moves bytes to/from disk to DRAM

Filesystem

- Dozens of filesystems exist, e.g., ext2, ext3, NTFS, etc.
  - Differ on how they layer file and dir. abstractions as bytes, what metadata is stored, etc.
  - Differ on how data integrity/reliability is assured, support for editing/resizing, compression/encryption, etc.
  - Some can work with (“mounted” by) multiple OSs

Q: *What is a database? How is it different from just a bunch of files?*

Collection of files?

Virtualization of Files

Binary Representation on Disk storage

- Maintenance
- Performance
- Usability
- Security & privacy
- …

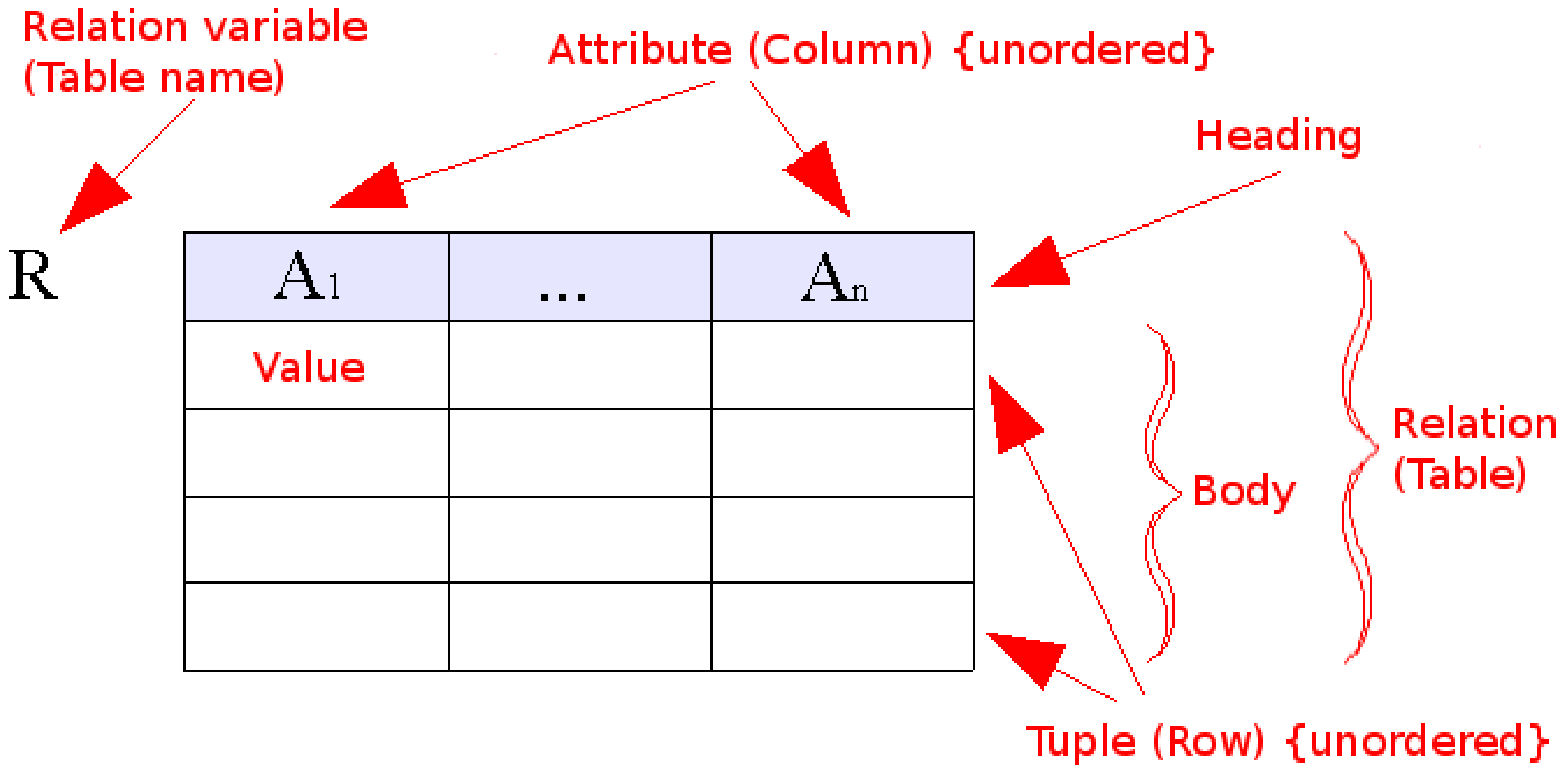
Files Vs Databases: Data Model

- Every database is just an *abstraction* on top of data files!
  - Logical level: Data model for higher-level reasoning
  - Physical level: How bytes are layered on top of files
  - All data systems (RDBMSs, Dask, Spark, TensorFlow, etc.) are application/platform software that use OS System Call API for handling data files

Data as File: Structured

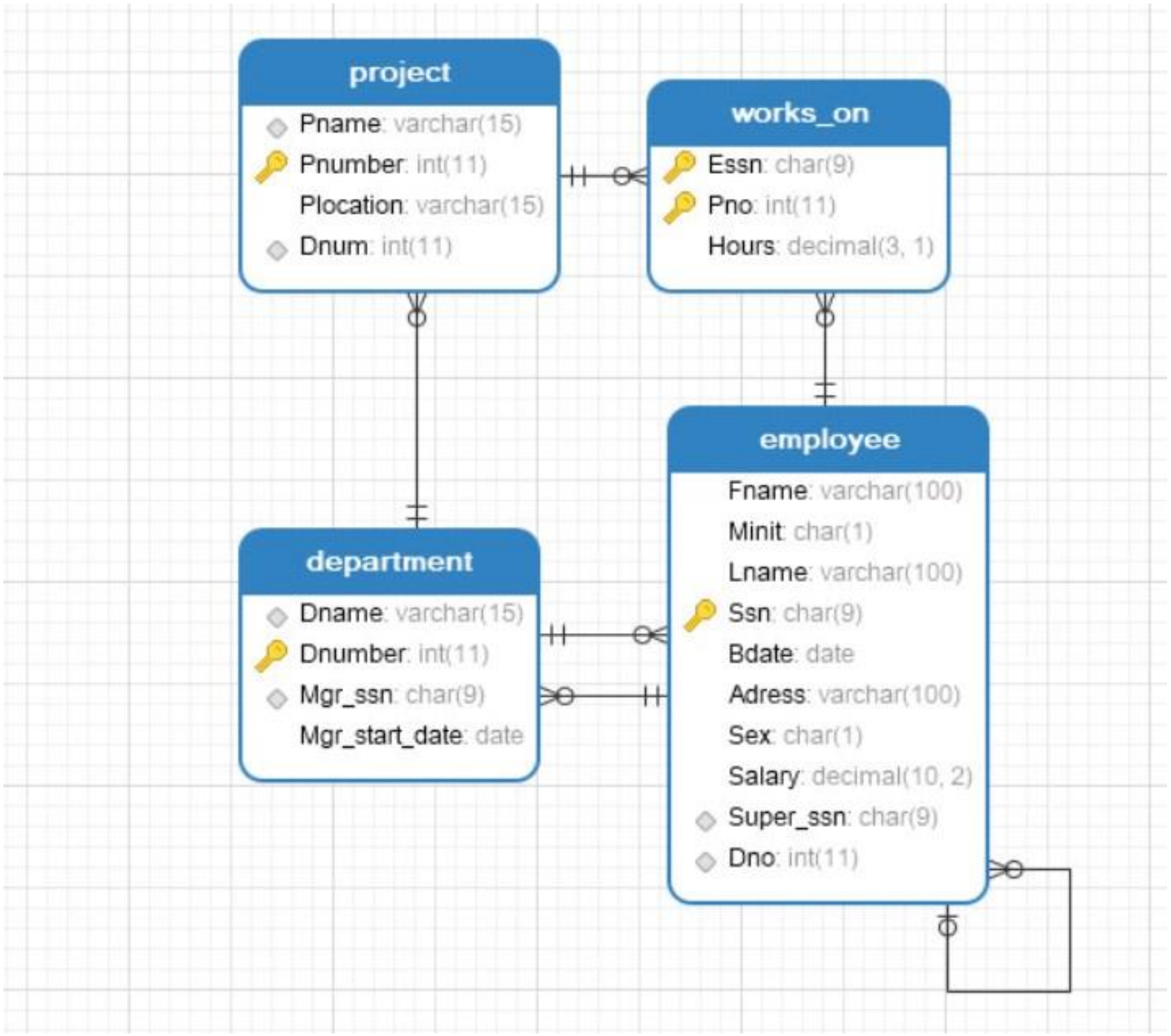
- **Structured Data:** A form of data with regular substructure

Relation



- Most RDBMSs and Spark serialize a relation **as *binary* file(s)**, often compressed

Relational Database



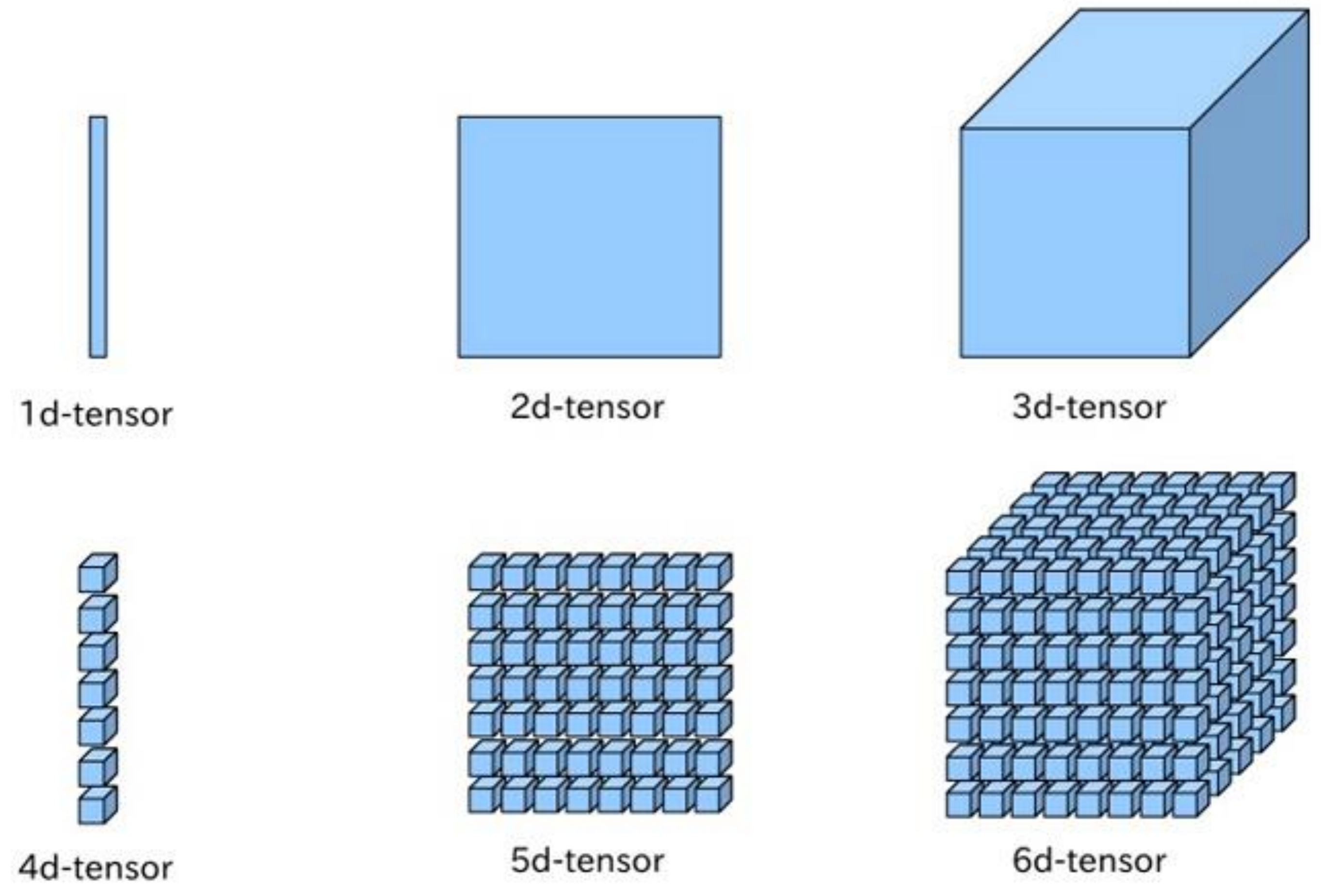
Data as File: Structured

- Structured Data: A form of data with regular substructure

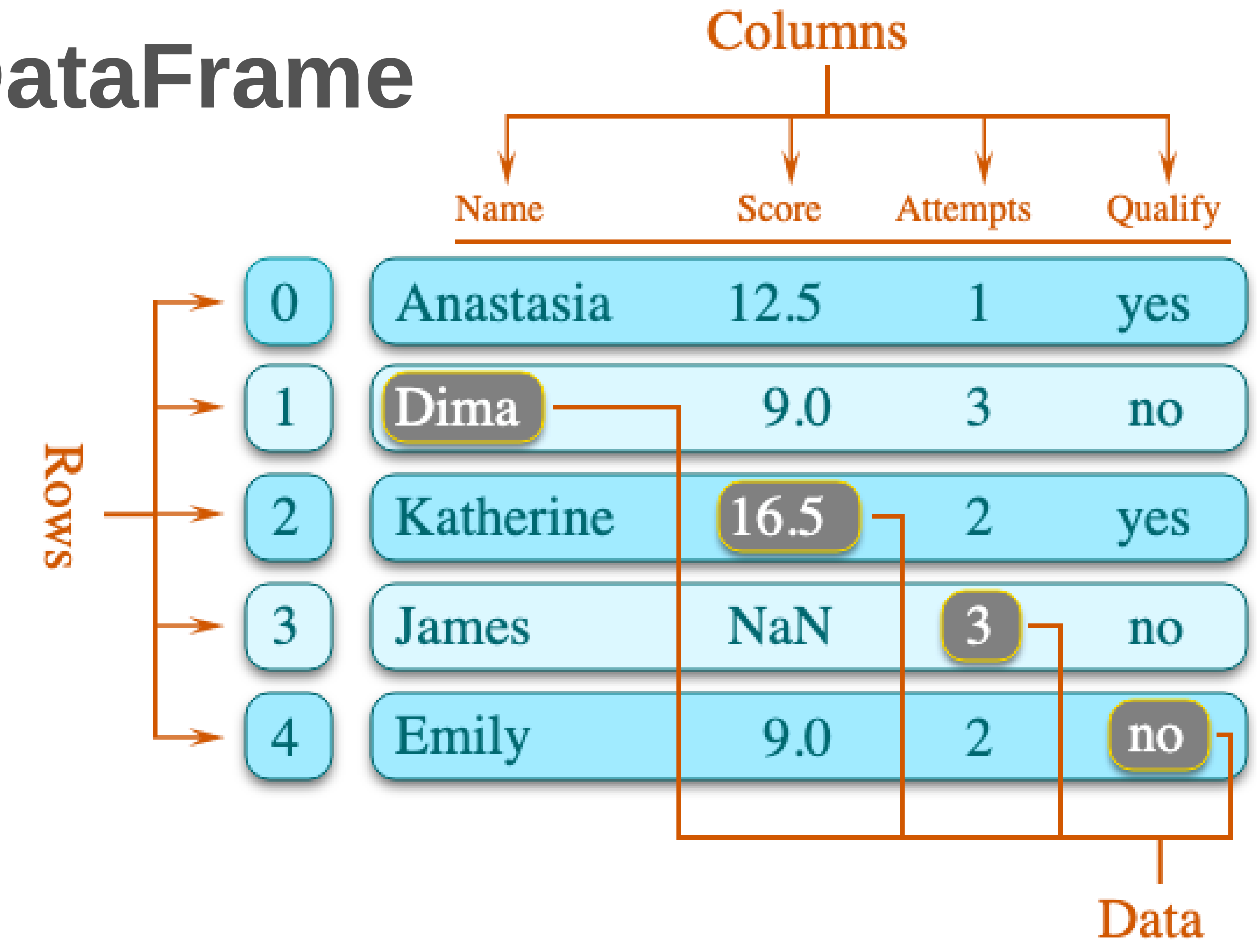
Matrix

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ \vdots \\ m \end{array}\begin{array}{c} \begin{matrix} 1 & 2 & \ldots & n \end{matrix} \\ \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ a_{31} & a_{32} & \ldots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \end{array}$$

Tensor



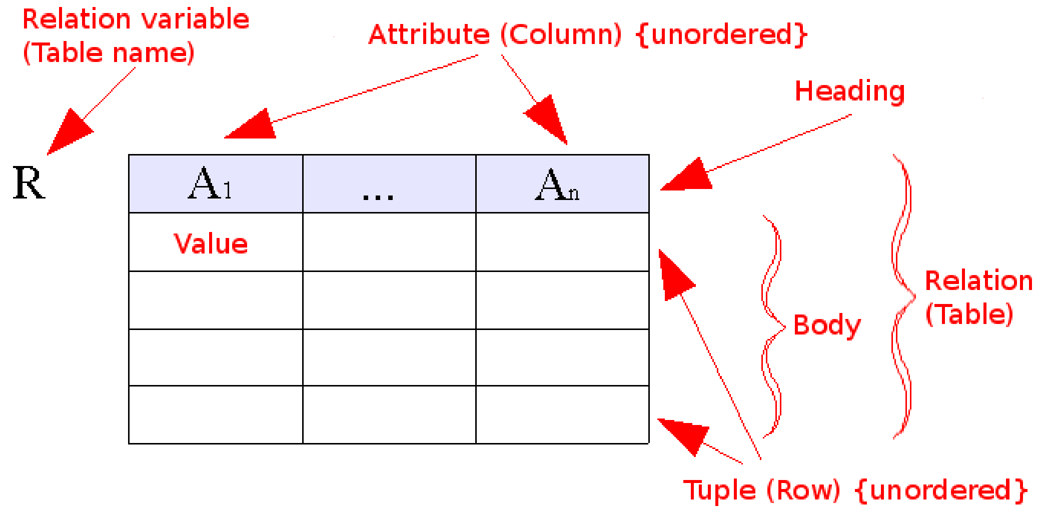
DataFrame

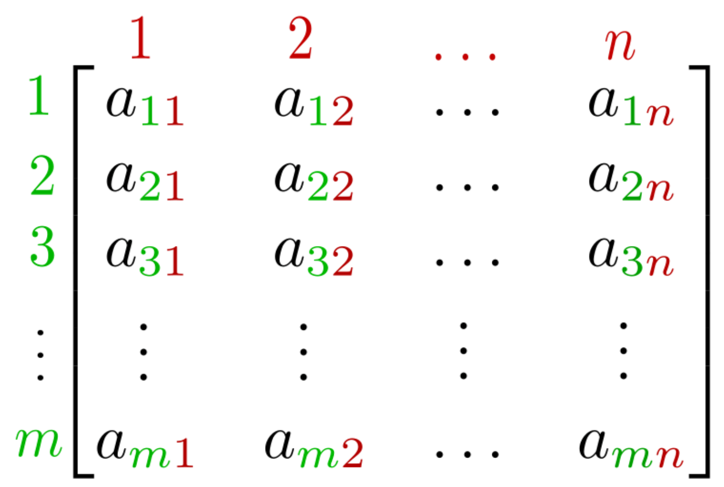


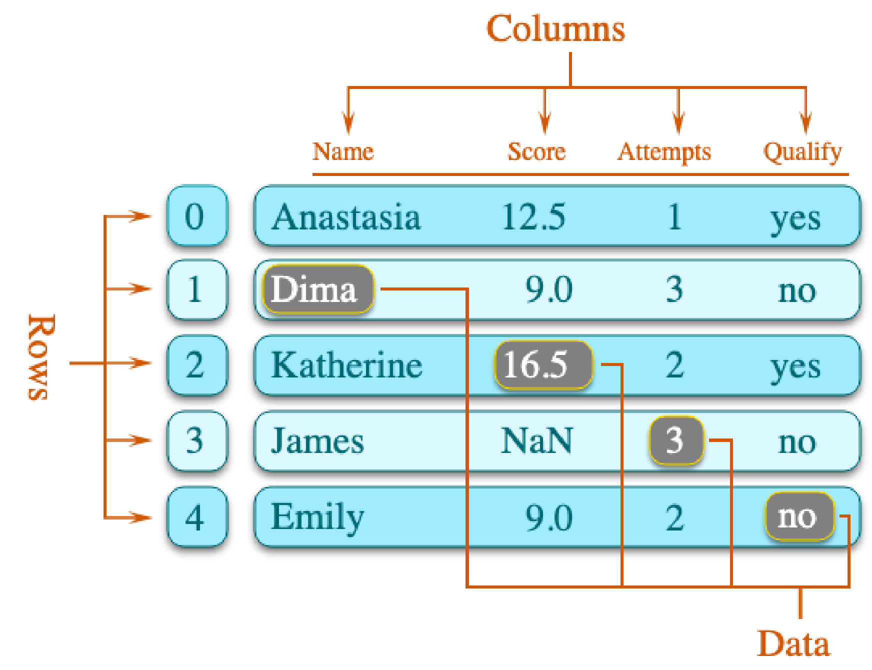
- Typically serialized as restricted ASCII text file (TSV, CSV, etc.)
- Matrix/tensor as binary too
- Can layer on Relations too!

Comparing Struct. Data Models

***Q:** What is the difference between Relation, Matrix, and DataFrame?*





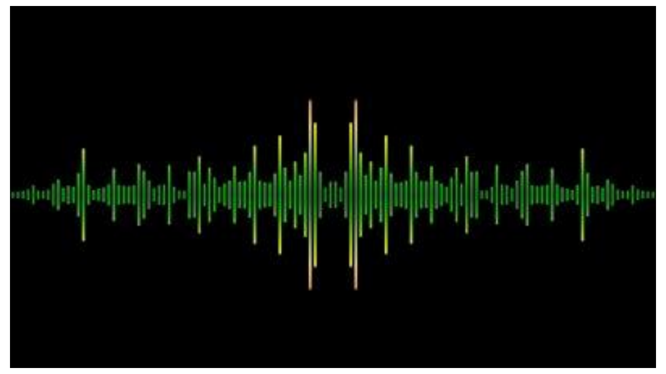
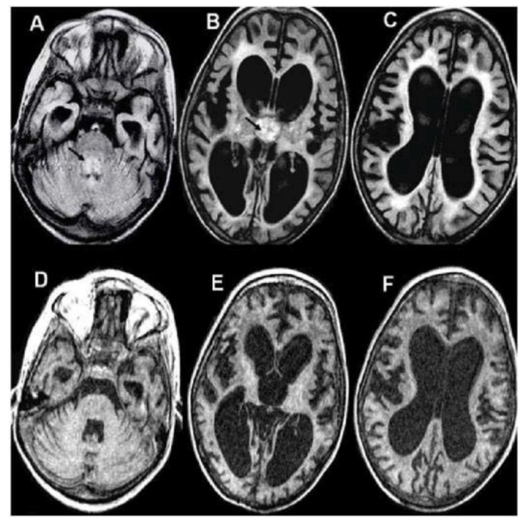


- Ordering: Matrix and DataFrame have row/col numbers; Relation is orderless on both axes!
- Schema Flexibility: Matrix cells are numbers. Relation tuples conform to pre-defined schema. DataFrame has no pre-defined schema but all rows/cols can have names; col cells can be mixed types!
- Transpose: Supported by Matrix & DataFrame, not Relation

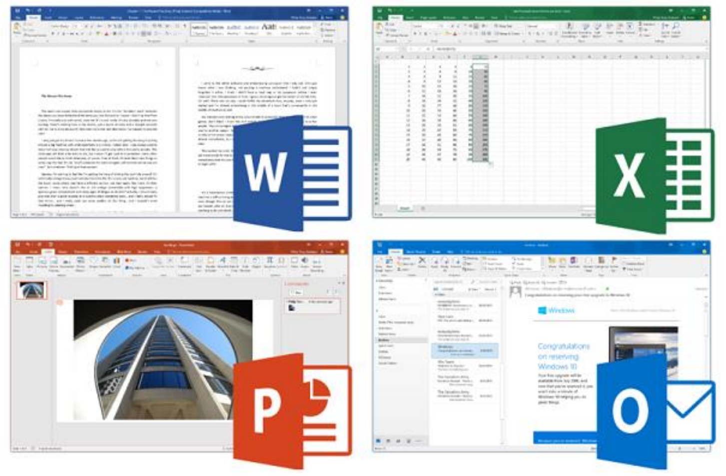
If interested in reading more: https://towardsdatascience.com/preventing-the-death-of-the-dataframe-8bca1c

Data as File: Other Common Formats

- Machine Perception data layer on tensors and/or time-series
- Myriad binary formats, typically with (lossy) compression, e.g., WAV for audio, MP4 for video, etc.


- Text File (aka plaintext): Human-readable ASCII characters
- Docs/Multimodal File: Myriad app-specific rich binary formats



PDF Adobe

jupyter

ChatGPT

Q1: In What format are GPT-3 weights stored?

Q2: In what format are GPT-3 training data stores? Structured or Unstructured?

Rule of Thumb: unstructured data are way more difficult to manage and deal with than structured data.

Next: File System, Database, Cloud Storage

- File system
- **Database**
  - **Strawman**
  - HashTable
  - SSTable and LSM-Trees
  - B-Tree (optional)
- Column Storage and Data Warehouse

The simplest database (demo)

```
#!/bin/bash

db_set () {
    echo "$1,$2" >> database
}

db_get () {
    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

1. Search the lines that start with a parameter.

2. Only output the value part.

3. Only output the last line.

The simplest database (write)

```
#!/bin/bash

db_set () {
    echo "$1,$2" >> database
}

db_get () {
    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

- **Append only.**
  - Writing is efficient.
  - Application:
    - Database Log
- **How to address the following challenges?**
  - Concurrency
  - Disk space
  - Handling errors
  - …

The simplest database (read)

```
#!/bin/bash

db_set () {
    echo "$1,$2" >> database
}

db_get () {
    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

- Always output the latest matched line.
  - Read is super slow.
    - Scan entire database.
    - The cost of lookup O(n).
      - Double lines => Double time

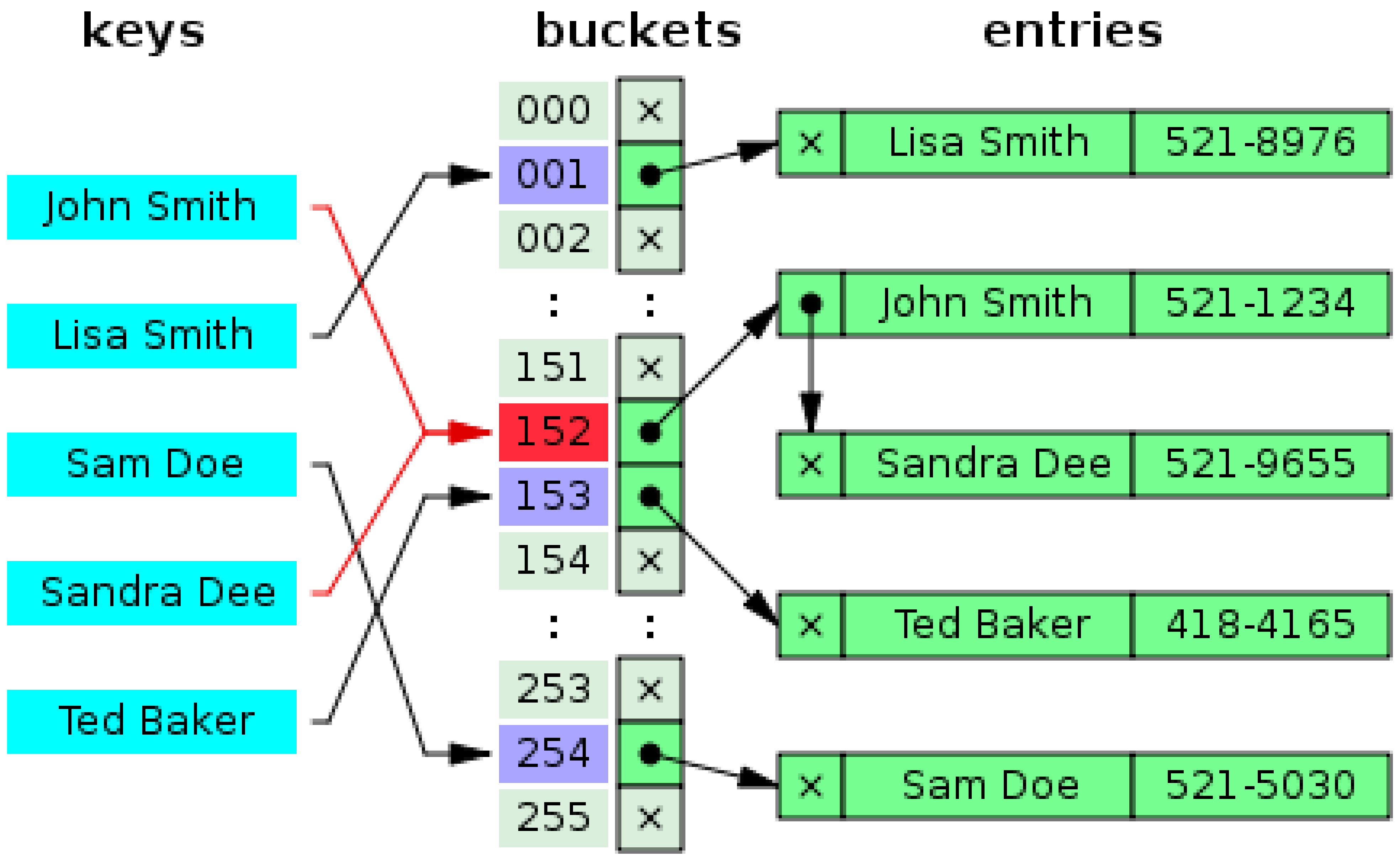
Improvement: Index

- Keep some additional metadata on the side, which acts as a signpost and helps you to locate the data you want.
- Faster to find the data.
- Update/remove/add the index is cheap.
- No free lunch!
  - Slows down the write.
  - Often needs to update the index.
- Choose your index wisely!
  - Index speeds up read, but slow down writes
  - Based on domain knowledge.
  - Balance the tradeoffs.

Hash map/table

A hash table is a very fast approach to dictionary storage

- hash functions
- Search, insert, delete: ~ O(1).



Time Complexity

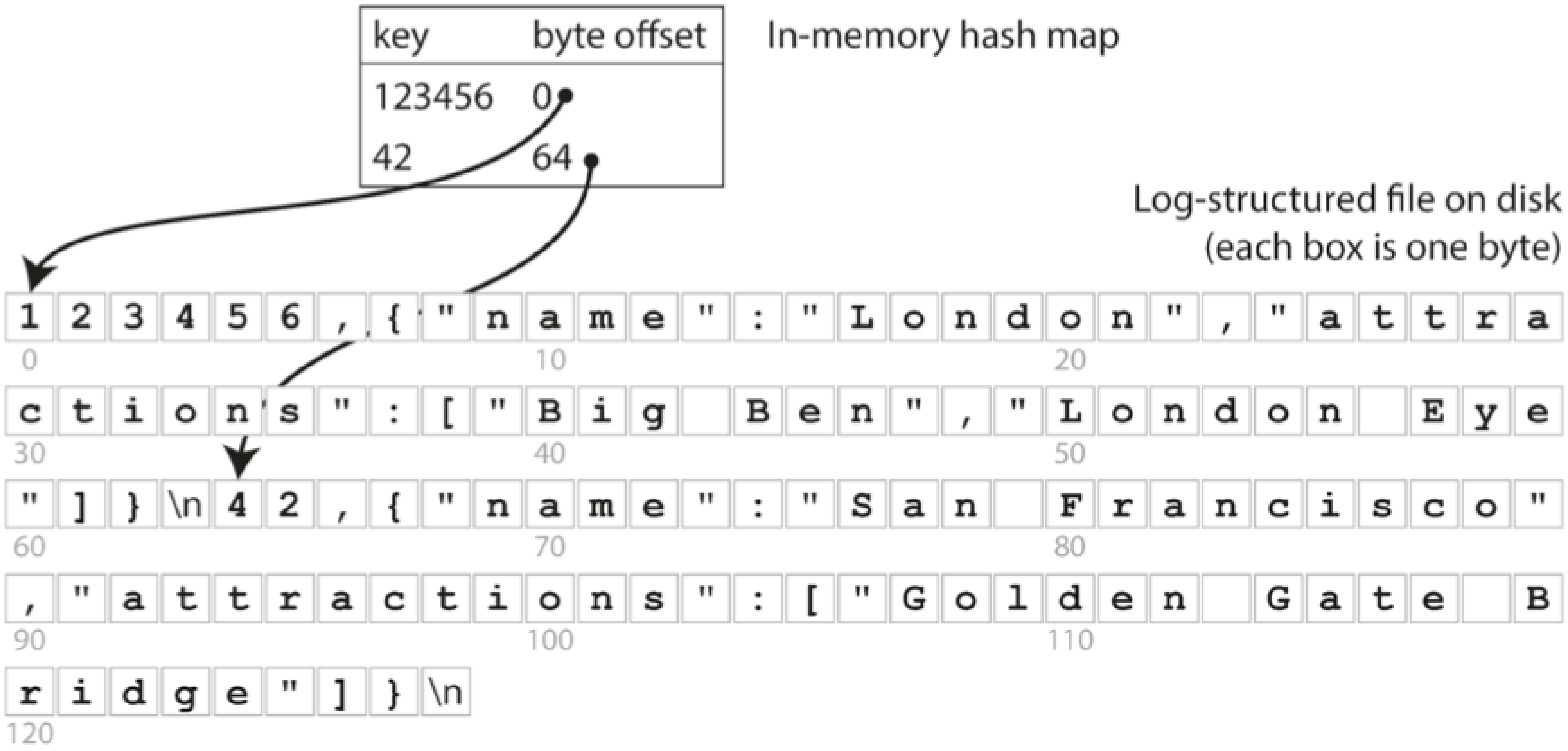
Average Case	Add	Remove	Search
Array	O(1)	O(n)	O(n)
Sorted Array	O(n)	O(lg n)	O(lg n)
Linked List	O(1)	O(n)	O(n)
BST	O(lg n)	O(lg n)	O(lg n)
Hash Table	~O(1)	~O(1)	~O(1)

Note: For sorted array and BST, keys have to be ordered.

8

See details in https://algs4.cs.princeton.edu/lectures/keynote/34HashTables.pdf

Hash map in Memory Hierarchy



- Keys: small and in memory
- Values: Large and in disk
- High performance reads and writes.
- Capacity:
  - All keys need to fit in the available RAM.
  - Values can be load from a disk. Much larger!!!

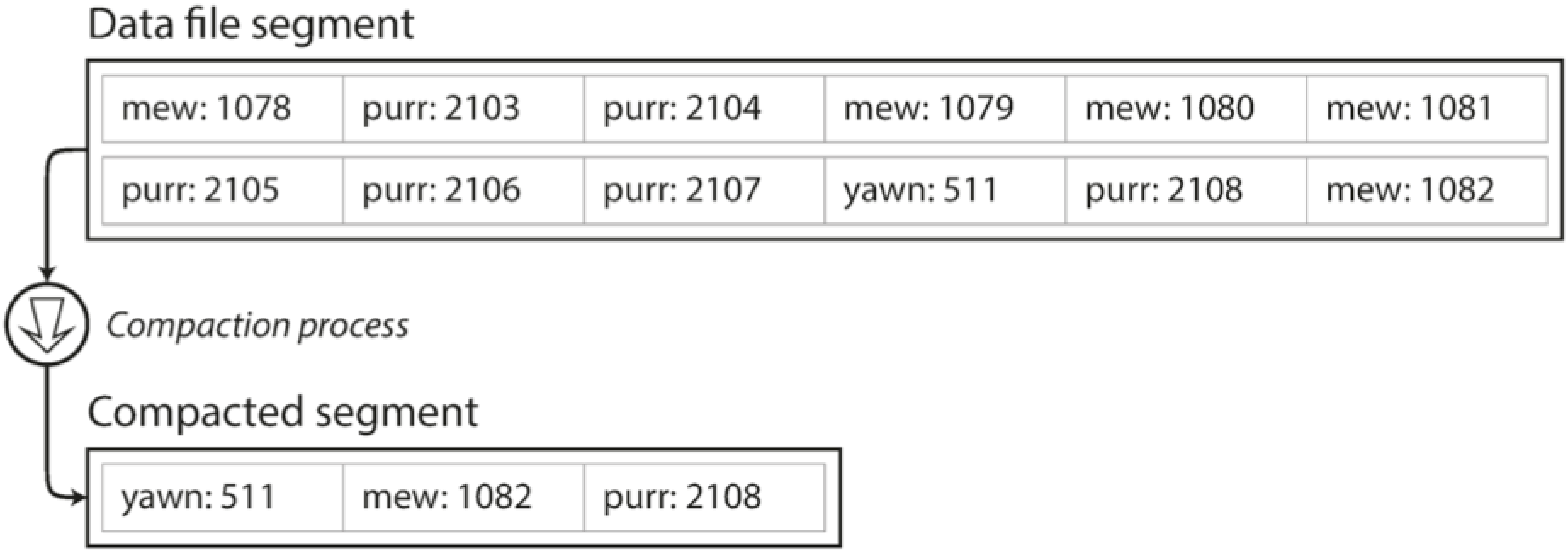
An example application:

- Track the number of times a video has been played.
  - Increment every time someone hits the play button.
- Memory capacity
  - 64 GB
  - URL: 2048 char = 2048 byte = 2KB
  - 64 GB/2KB = 32 million.

- **Problem: YouTube has over 800 million videos. Need to keep all the keys in Memory?**
  - We'll improve this later using **SSTable**

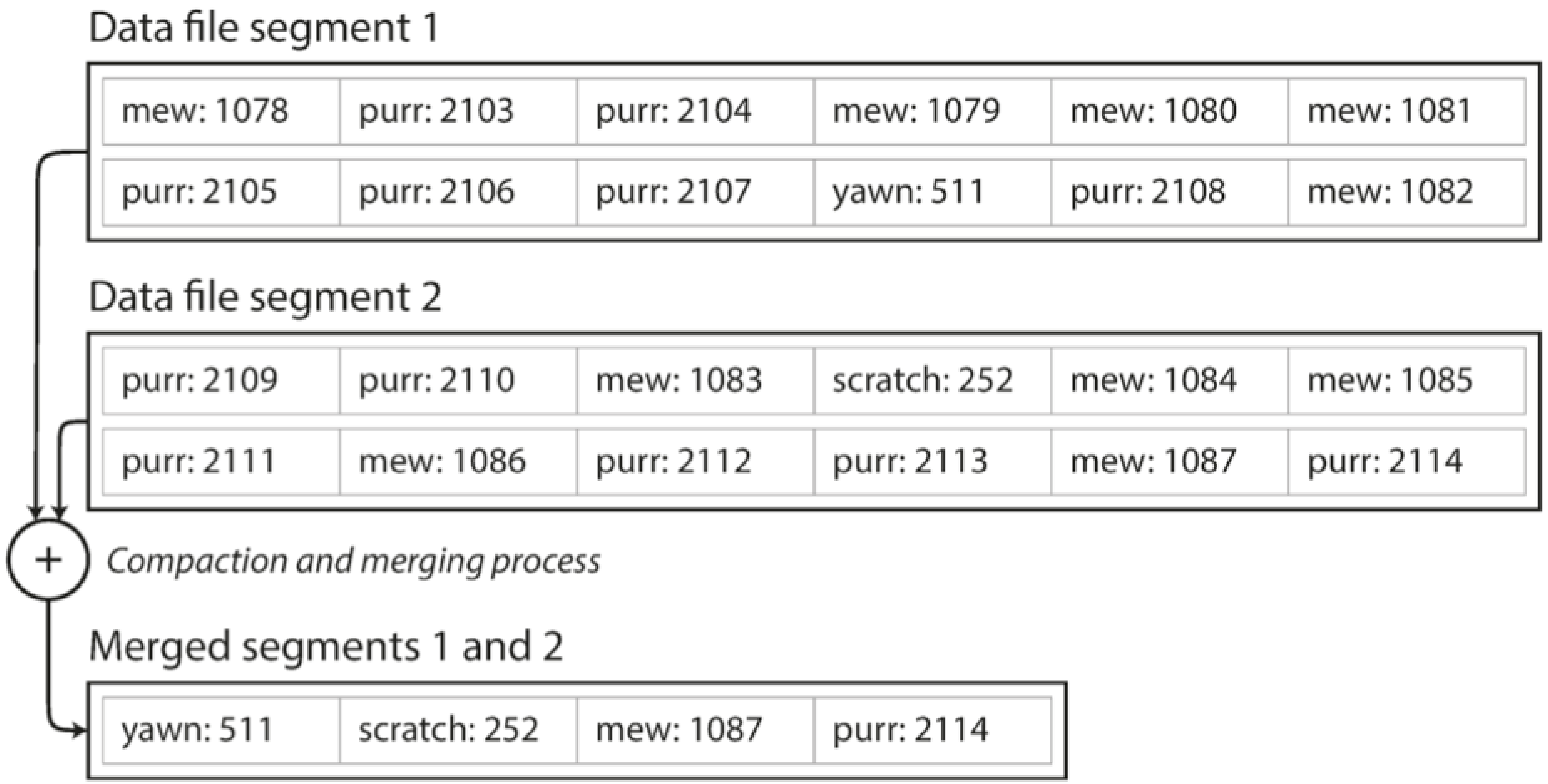
Run out of disk space? Segment compaction

- Segments of a certain size.
- Perform compaction.
  - Throw away duplicate logs and keep only the most recent update.

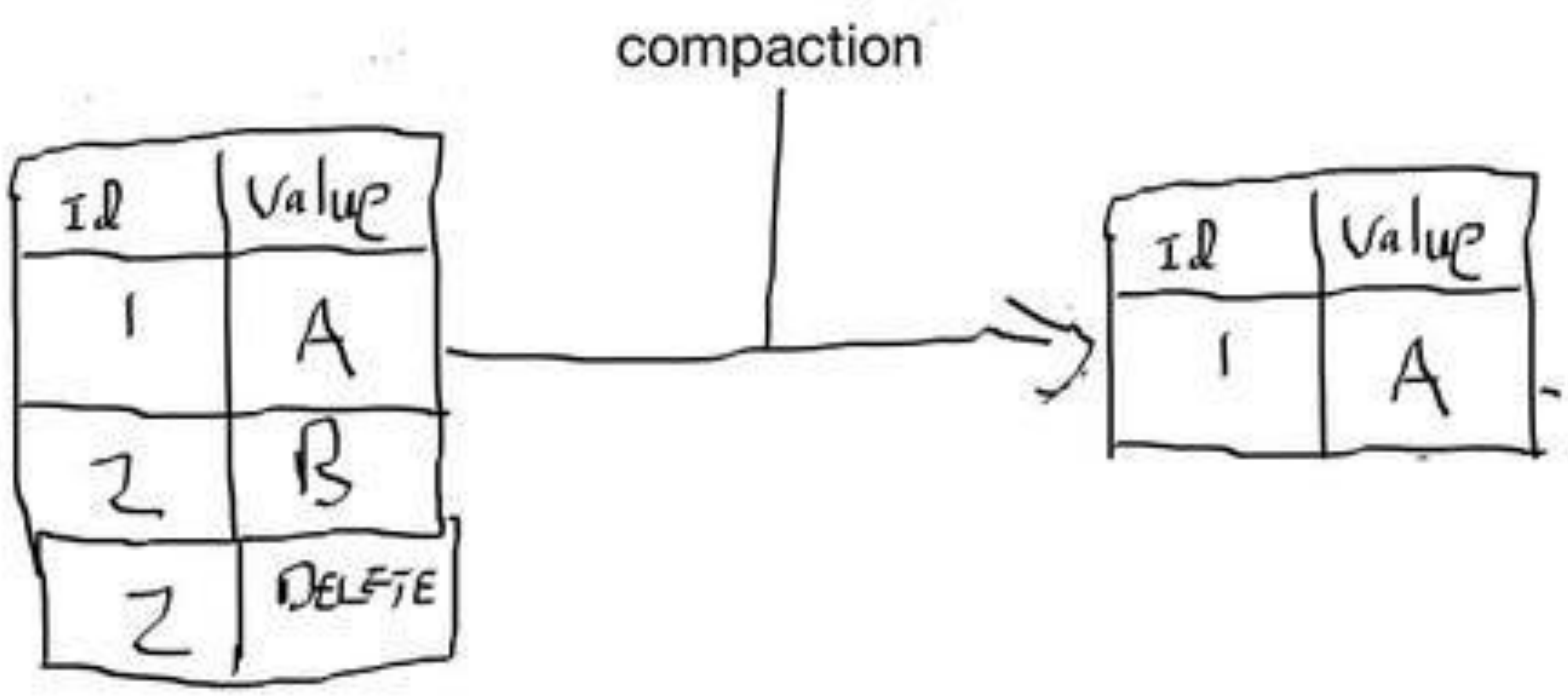


Concurrent R/W and Compaction?

- Frozen segments. Never modified.
  - Only merge frozen segments and write the output to a new file.
- The read and write can work as normal using the old segment files.
- After the merging,
  - Read requests from the merged file.
  - Delete old segment files



How to delete a record?



Crash recovery

- Restart a database.
  - Segments are often large.
    - Loading is slow.
    - Store the segments' hash maps on disk.
- Partially written records. e.g., lose power?
  - Checksums for each record.
  - Detect and ignore corrupted parts.

Hash Table Index

- Advantages (Append-only & imputable):
  - Very fast write.
    - Recall how hard drive works.
  - Simple concurrency and crash recovery.
    - No need to worry about partially written records.
  - Avoid the problems of fragmented data files.
- Disadvantage
  - The hash table index must fit in memory.
    - Can we put hash table index on disk?

Data indexes

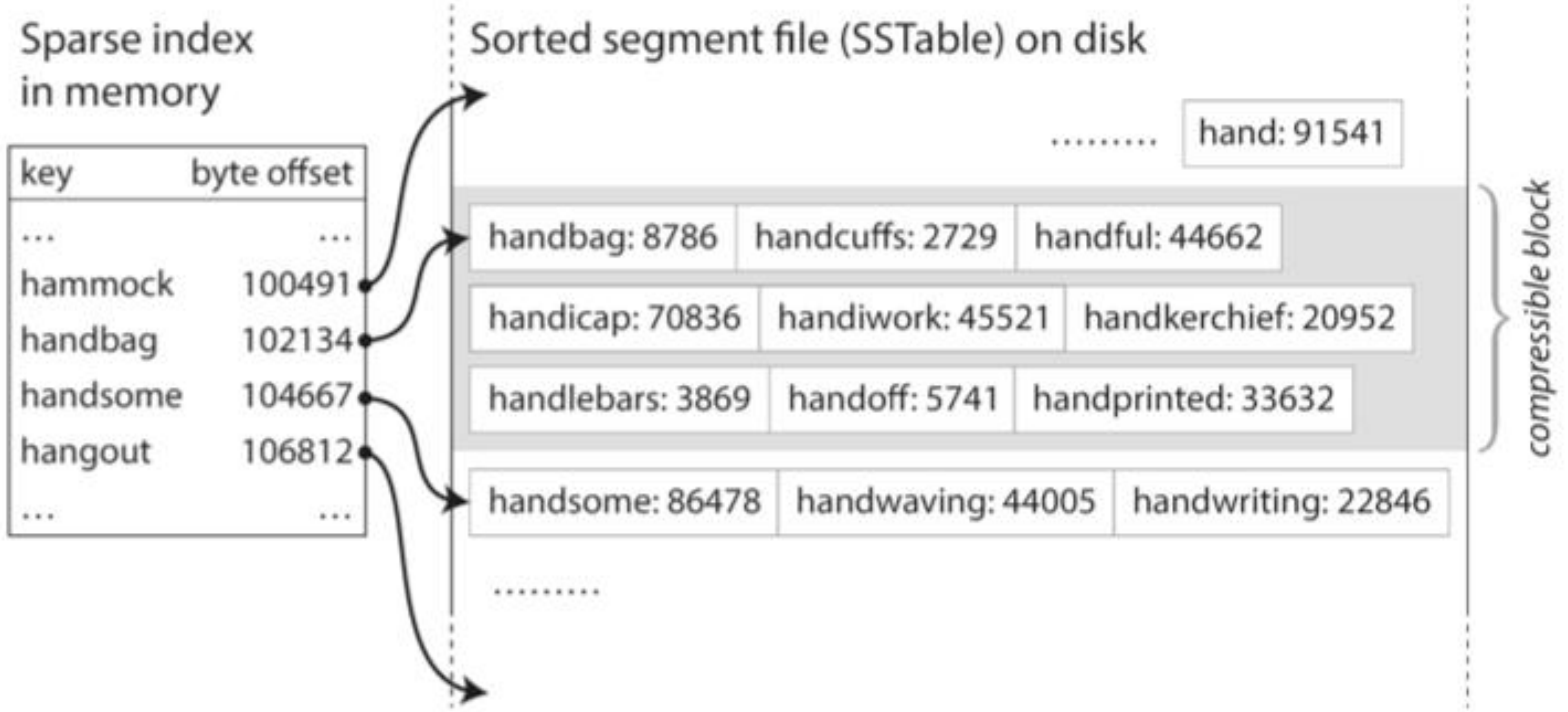
- Straw-man design (bash script, get, set, append-only)
  - Fast write
  - Slow read
  - Large storage space.
- Hashtable (all keys in the memory, all values on the disk, background compaction)
  - Fast write & read
  - Less storage space
  - All keys need to fit in memory.
- ????

SSTable (sorted string table)

- Change the format of the segment files
  - Sorted by keys

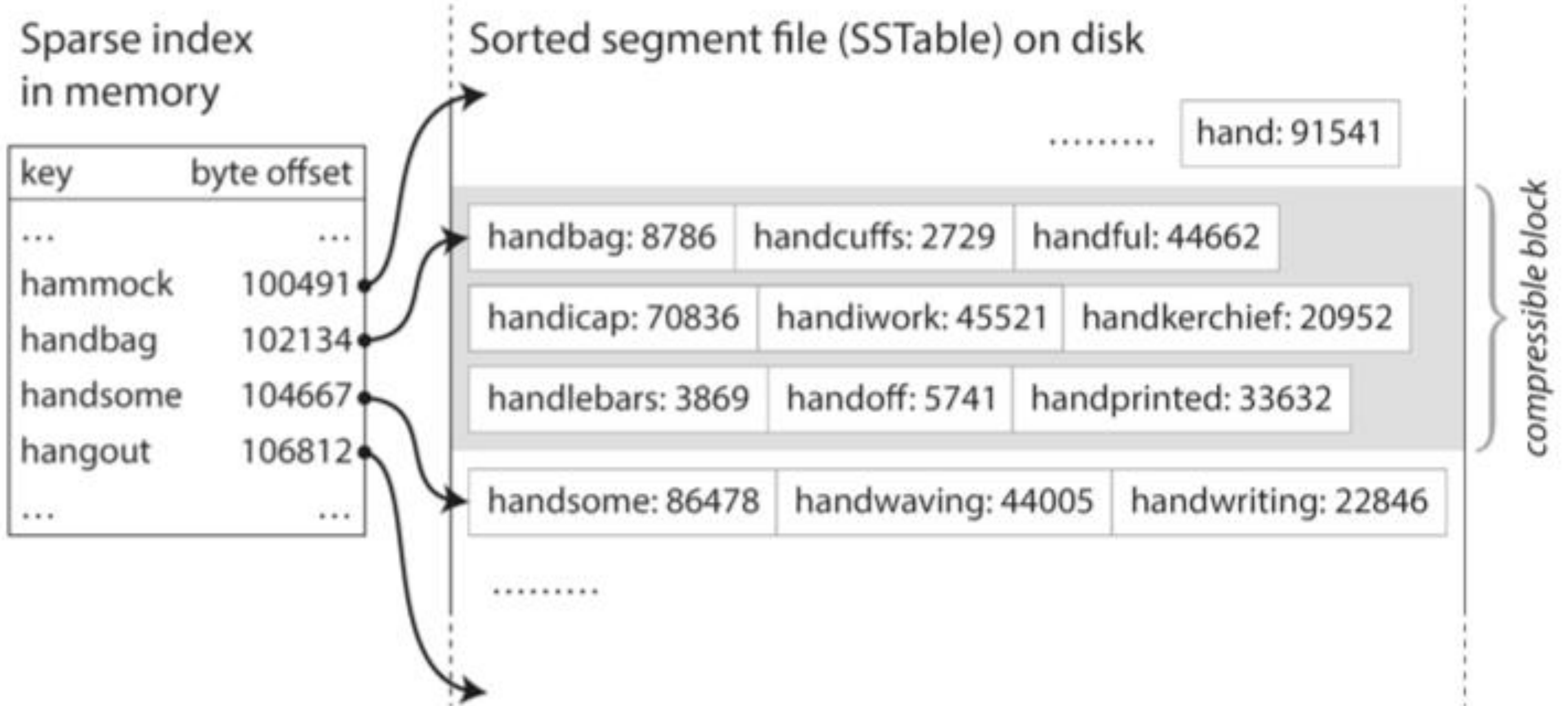
SSTable

- Merging segments is simple and efficient
  - Merge sort: in your PA1
- No longer need to keep an index of all the keys in memory.
  - Jump to the range.
  - Similar idea as Hash table.



SSTable implementation

- Sparse in-memory index
- Each segment file for a few KB-MB.
- “Better idea”:
  - Assume that the keys and values had a fixed size, use binary search on a segment file and avoid the in-memory index.
    - Only useful in special applications.
- Compressible blocks.



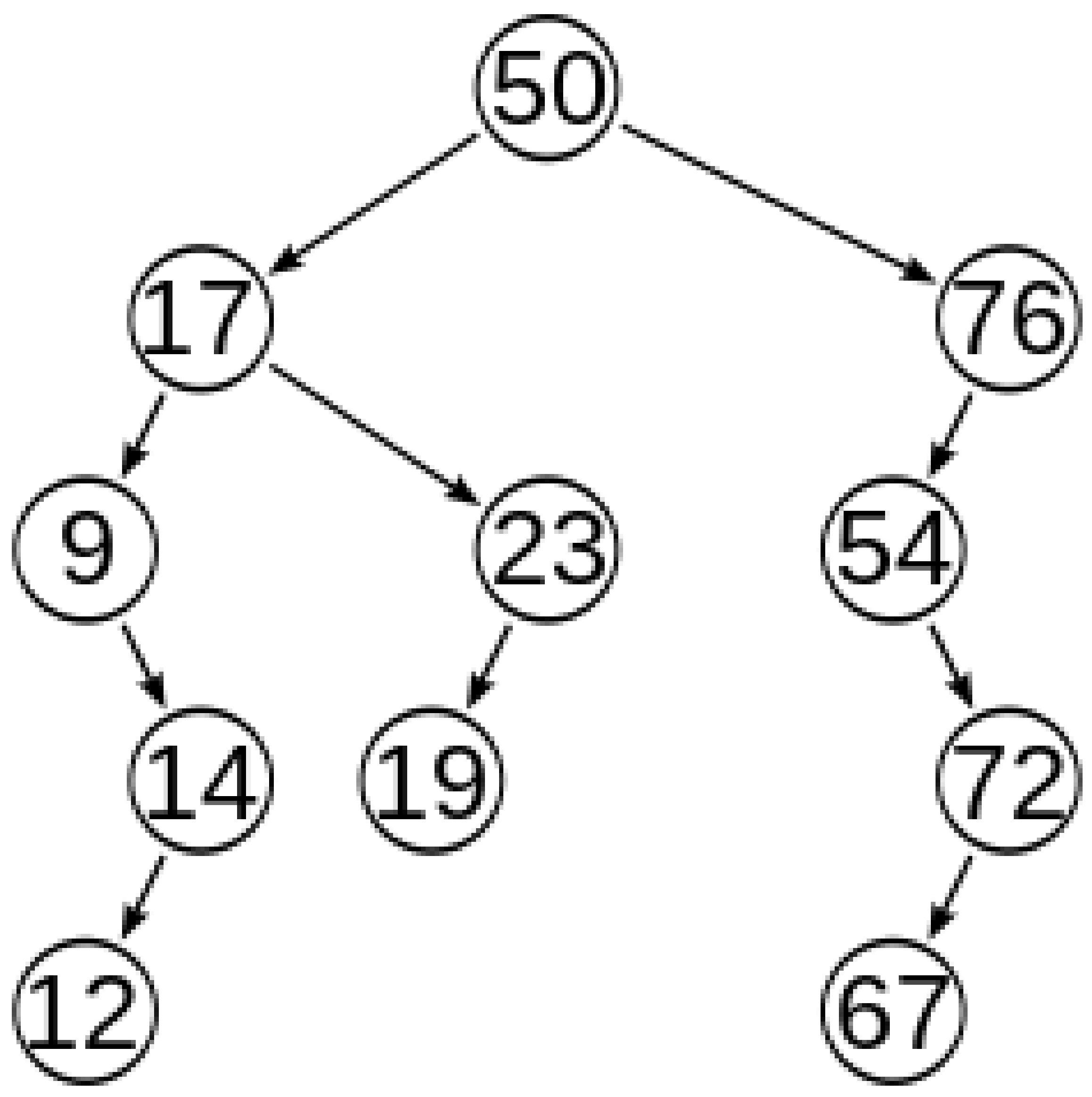
How do you get your data to be sorted by key in the first place?

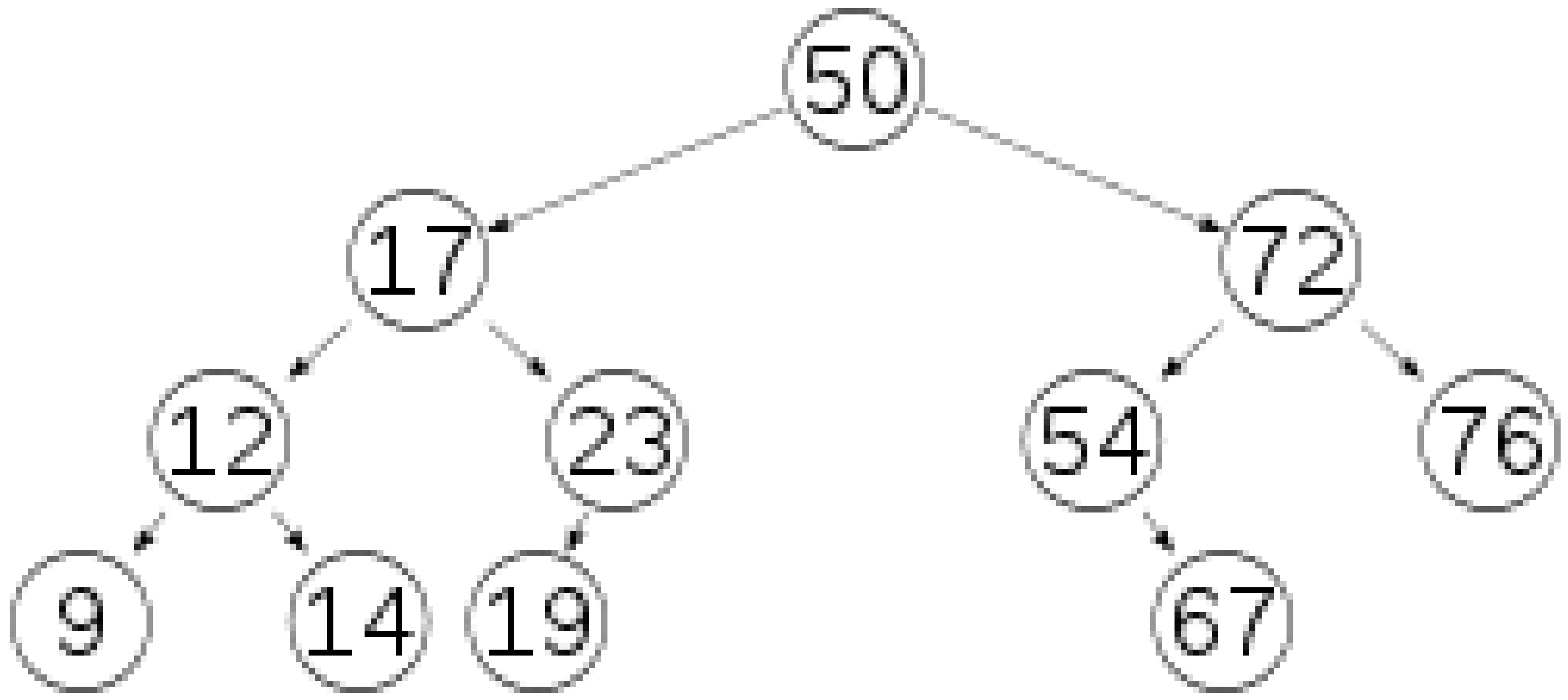
Memtable: Sorted structure in memory

- Easier to manipulate data in memory than disk.
  - Why?
- Maintain a sorted data structure in memory.

Self-balanced trees

- Any node-based binary search tree that automatically keeps its height (maximal number of levels below the root) small in the face of arbitrary item insertions and deletions.
  - E.g., Red-black trees or AVL trees
  - Height O(log n)





Complexity Comparison of Various Structures

Operation	Sequential List (Sorted Array)	Linked List	AVL Tree
Search for x	$O(\log n)$	$O(n)$	$O(\log n)$
Search for kth item	$O(1)$	$O(k)$	$O(\log n)$
Delete x	$O(n)$	$O(1)$[1]	$O(\log n)$
Delete kth item	$O(n - k)$	$O(k)$	$O(\log n)$
Insert x	$O(n)$	$O(1)$[2]	$O(\log n)$
Output in order	$O(n)$	$O(n)$	$O(n)$

[1]Doubly linked list and position of x known.

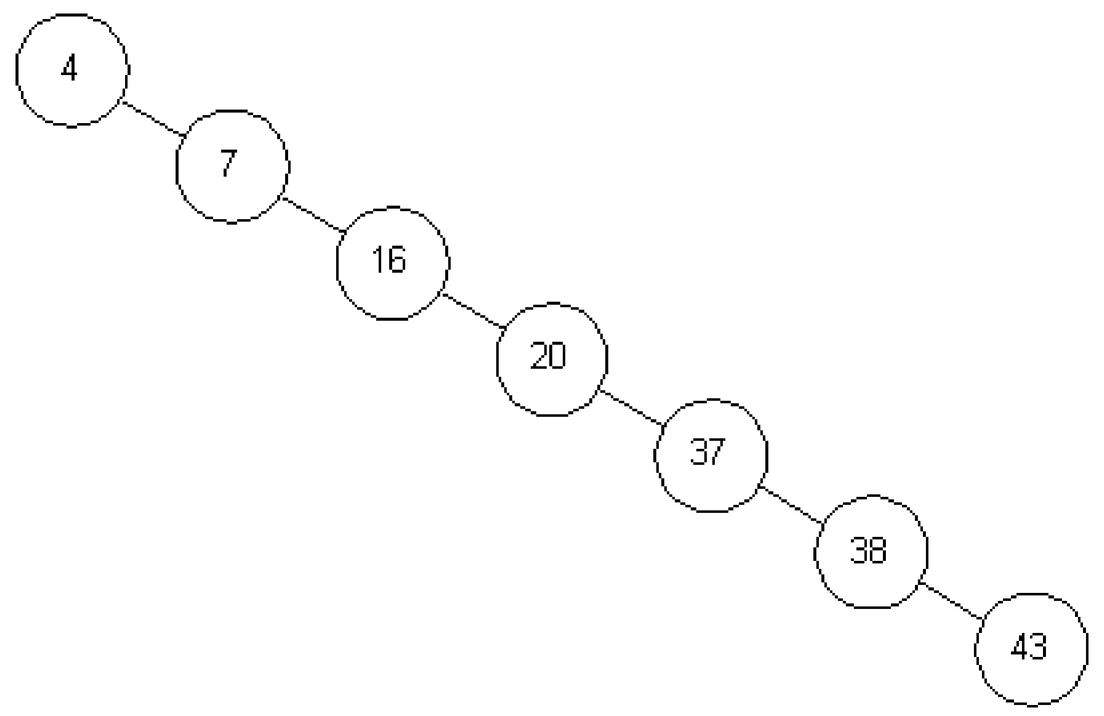
[2]Position for insertion known

10-16

AVL v.s Binary Search Tree

AVL tree		
Type	Tree	
Invented	1962	
Invented by	G.M. Adelson-Velskii and E.M. Landis	
Time complexity in big O notation		
	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Binary search tree		
Type	tree	
Invented	1960	
Invented by	P.F. Windley, A.D. Booth, A.J.T. Colin, and T.N. Hibbard	
Time complexity in big O notation		
Algorithm	**Average**	**Worst case**
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$



How a LSM (Log-structured merged-tree) storage engine works

- Write:
  - When a write comes in, add it to the memtable.
  - If the memtable > a threshold, save the memtable as the most recent segment.
- Read:
  - Check if the key in the memtable.
  - Then go through the segments.
- Background:
  - Merge and compact.

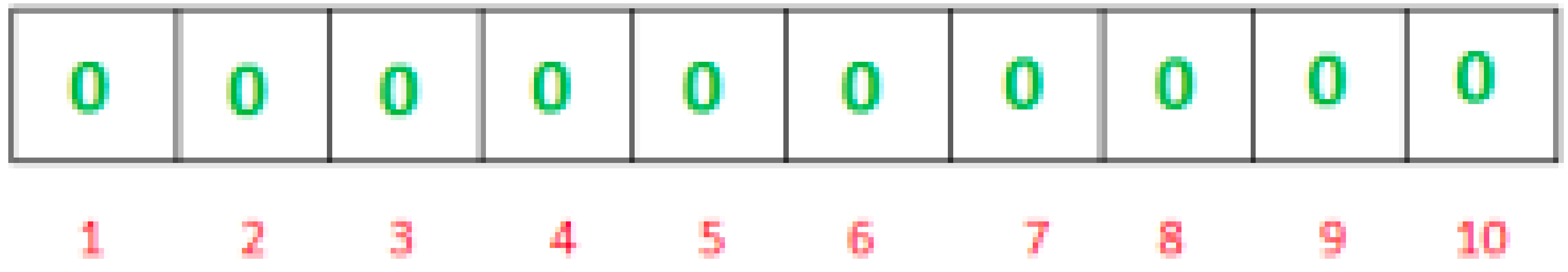
One issue of LSM

- What will happen if we want to look up keys that do not exist in the database?
  - Check the memtable
  - Check the segments all the way back to the oldest
- Optimization:
  - Use a bloom filter to test whether a key exist.

Bloom filters

- A space efficient probabilistic data structure
  - It can test whether an element is a member of a set.
  - Computation: O(k) and Space: O(m).
- Cost: probabilistic?
  - False positive:
    - It might tell that an element is a member of a set while it is not.

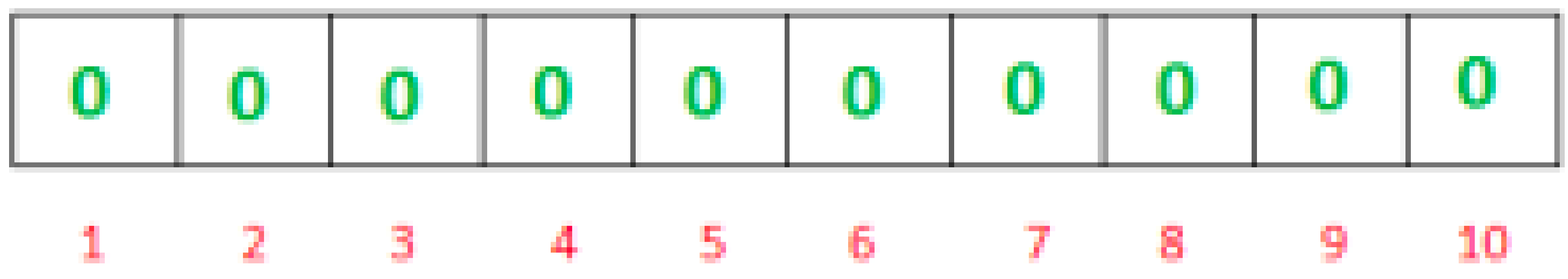
Initialization (m):



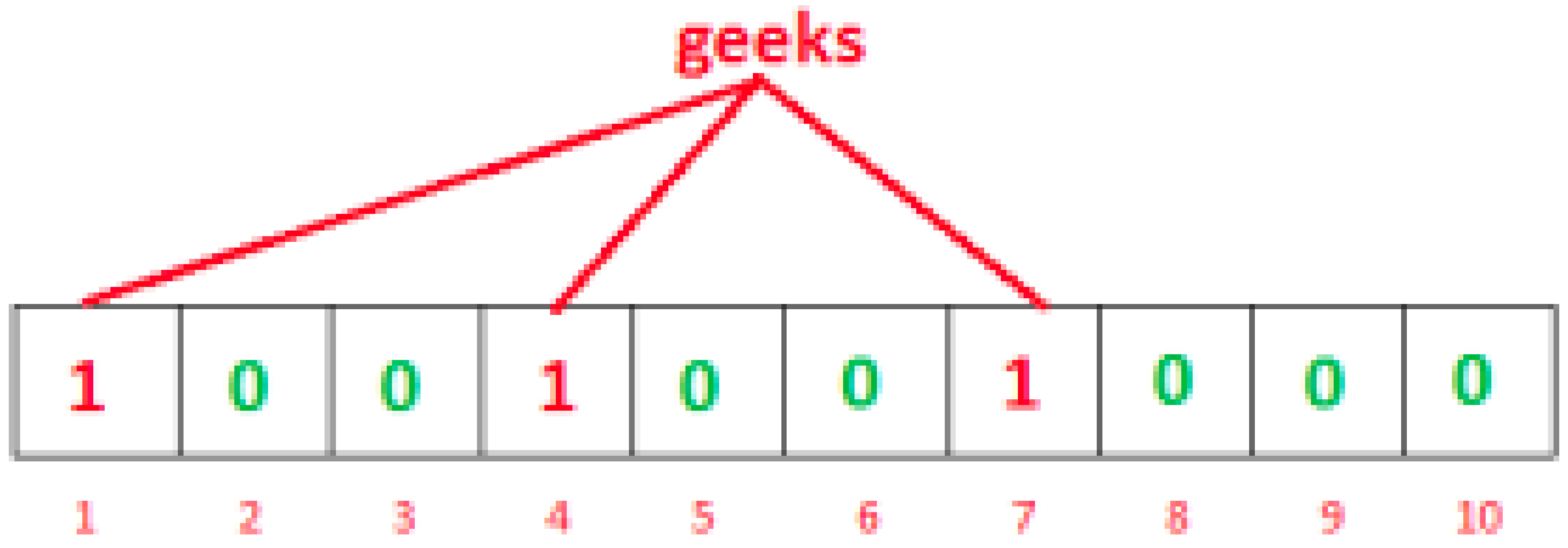
Three hashing functions (k): h1, h2, h3

Bloom filters (read and write)

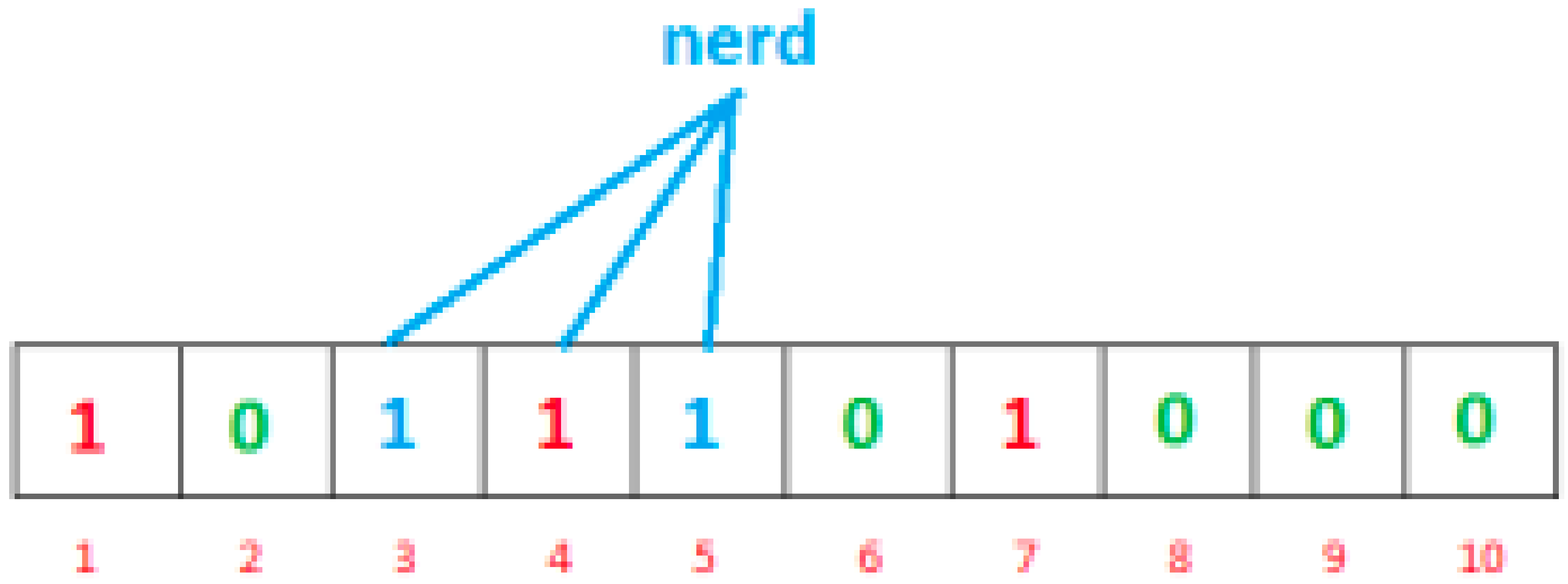
A set of words: {“geeks”, “nerd”}



h1(“geeks”) % 10 = 1
h2(“geeks”) % 10 = 4
h3(“geeks”) % 10 = 7



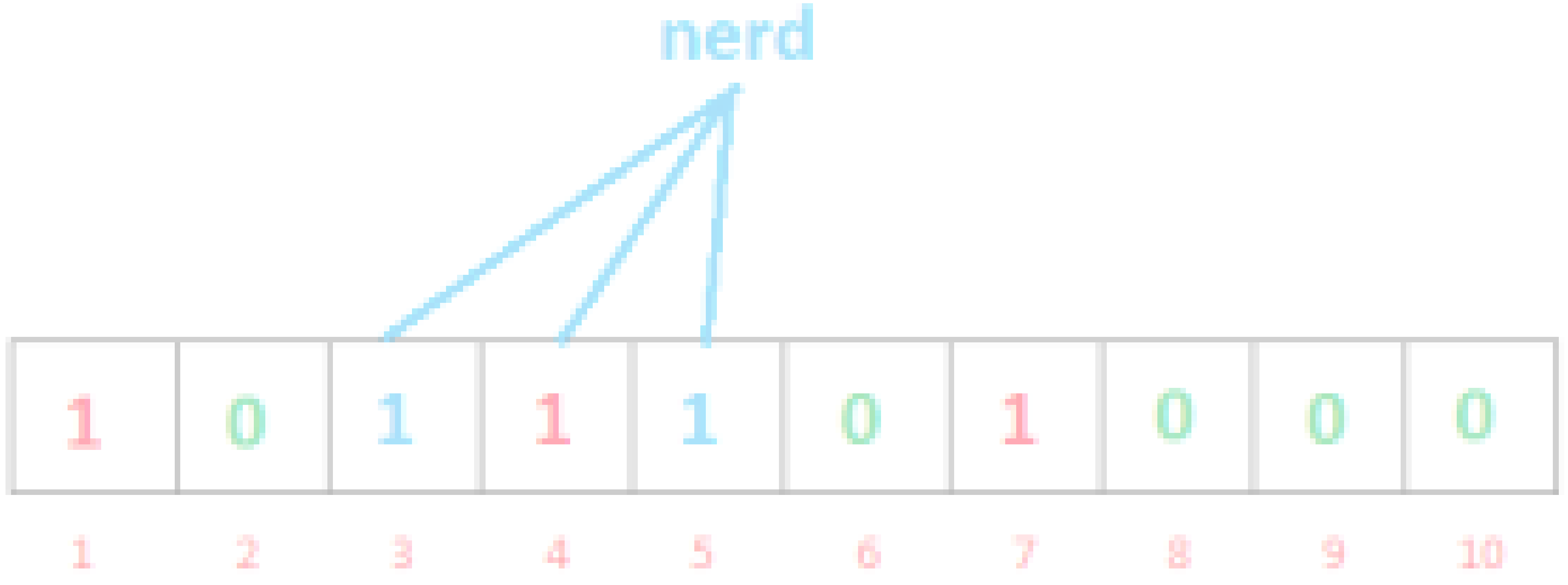
h1(“nerd”) % 10 = 3
h2(“nerd”) % 10 = 5
h3(“nerd”) % 10 = 4



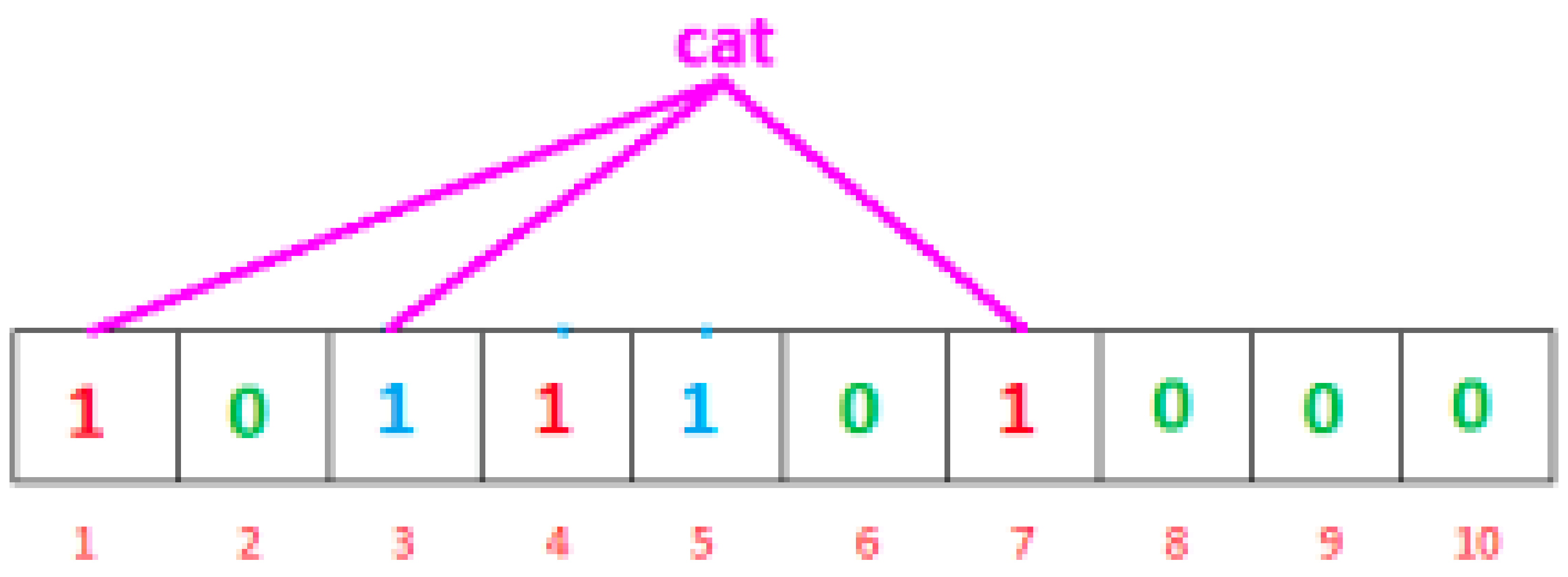
Bloom filters - False positive

h1("geeks") % 10 = 1
h2("geeks") % 10 = 4
h3("geeks") % 10 = 7

h1("nerd") % 10 = 3
h2("nerd") % 10 = 5
h3("nerd") % 10 = 4



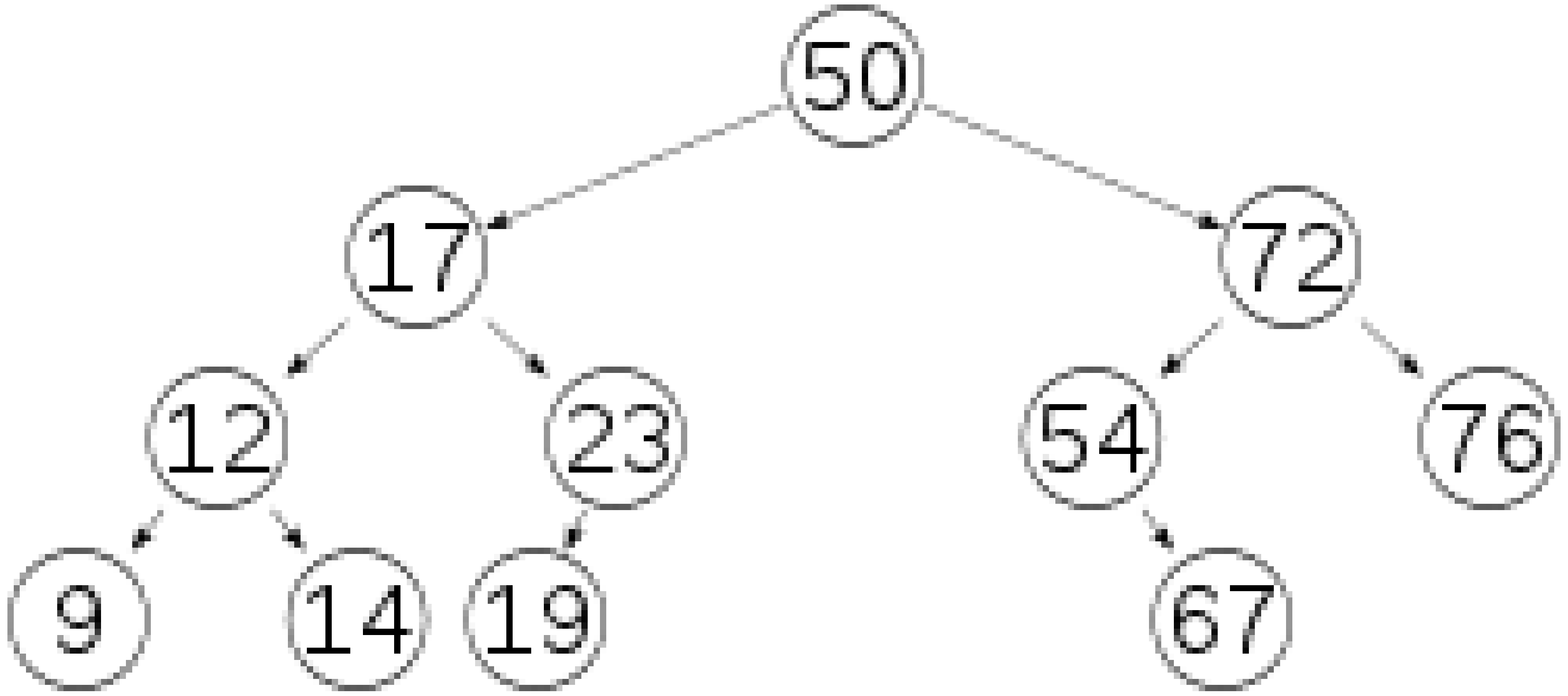
h1("cat") % 10 = 1
h2("cat") % 10 = 3
h3("cat") % 10 = 7



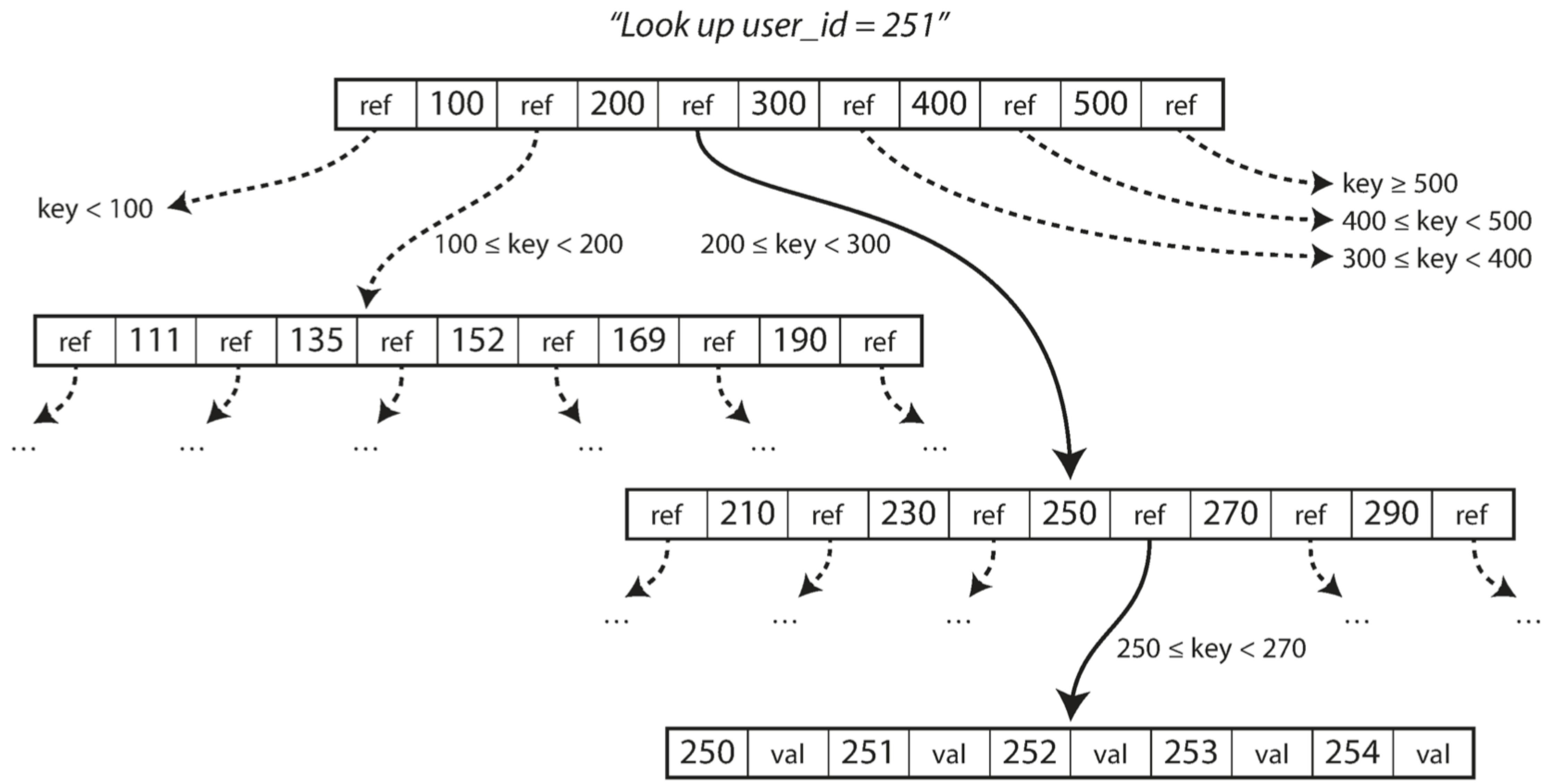
Data indexes

- Straw-man design (bash script, get, set, append-only)
  - Fast write
  - Slow read
  - Large storage space.
- Hashtable (all keys in the memory, all values on the disk, background compaction)
  - Fast write & read
  - Less storage space
  - All keys need to fit in memory.
- SSTable (HashTable + Sorted Segment + Sparse keys in the memory)
  - Works even if the size of keys in dataset is bigger than the memory.
  - Good performance for ranging queries as well.
  - Further compression

B-tree



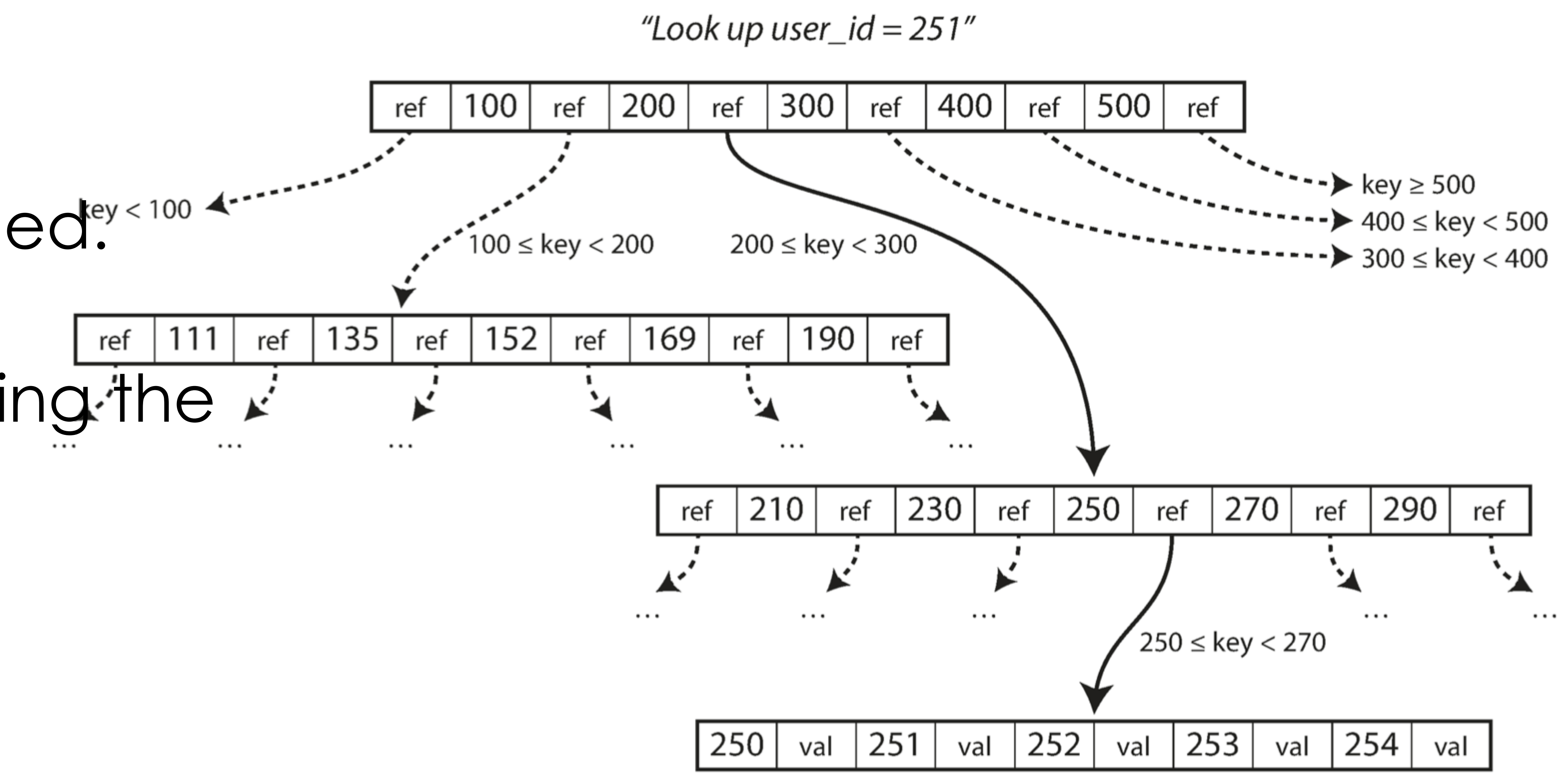
Self-balanced BST



B-Tree

B-tree

- Corresponds more closely to the underlying hardware, as disks are also arranged in fixed-size blocks.
- Root = kept in main memory.
  - Loaded into memory when needed.
- Not append only.
  - Search for the leaf page containing the target key
  - Change the value in that page
  - Write the page back to disk.
  - Do not change the references.



B-Tree

Recall Lecture 4 (Memory hierachy):

What's Inside A Disk Drive?

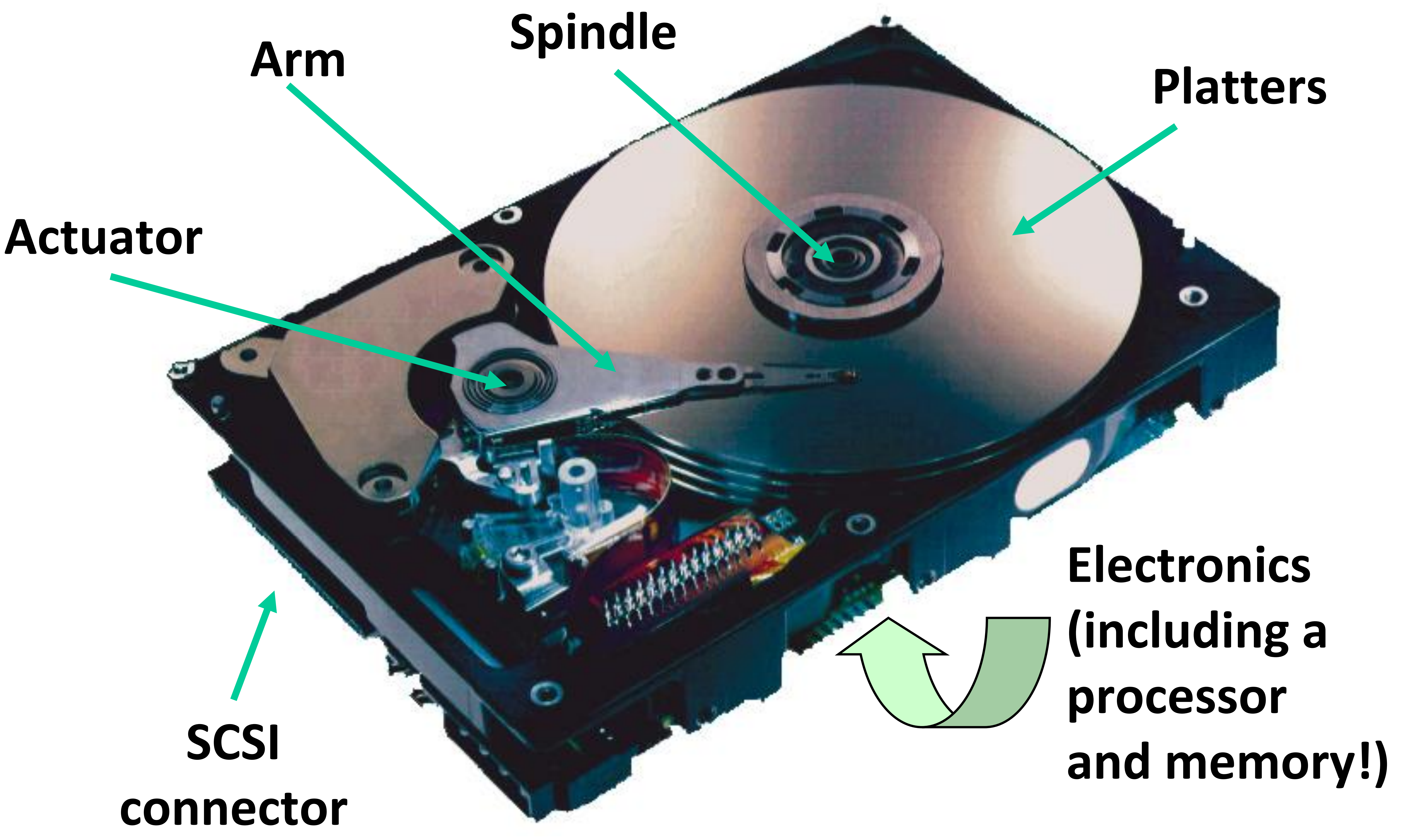
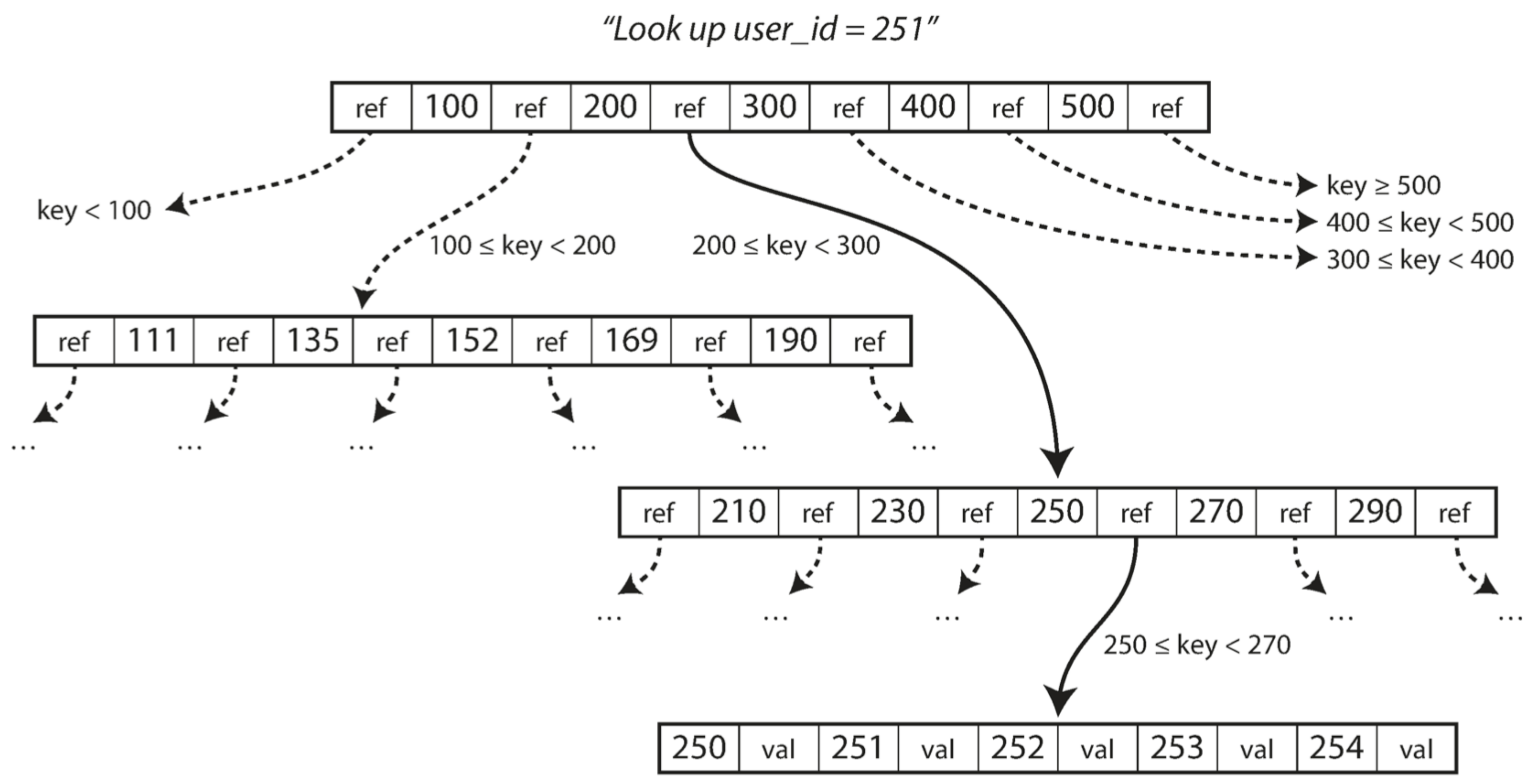


Image courtesy of Seagate Technology

B-tree

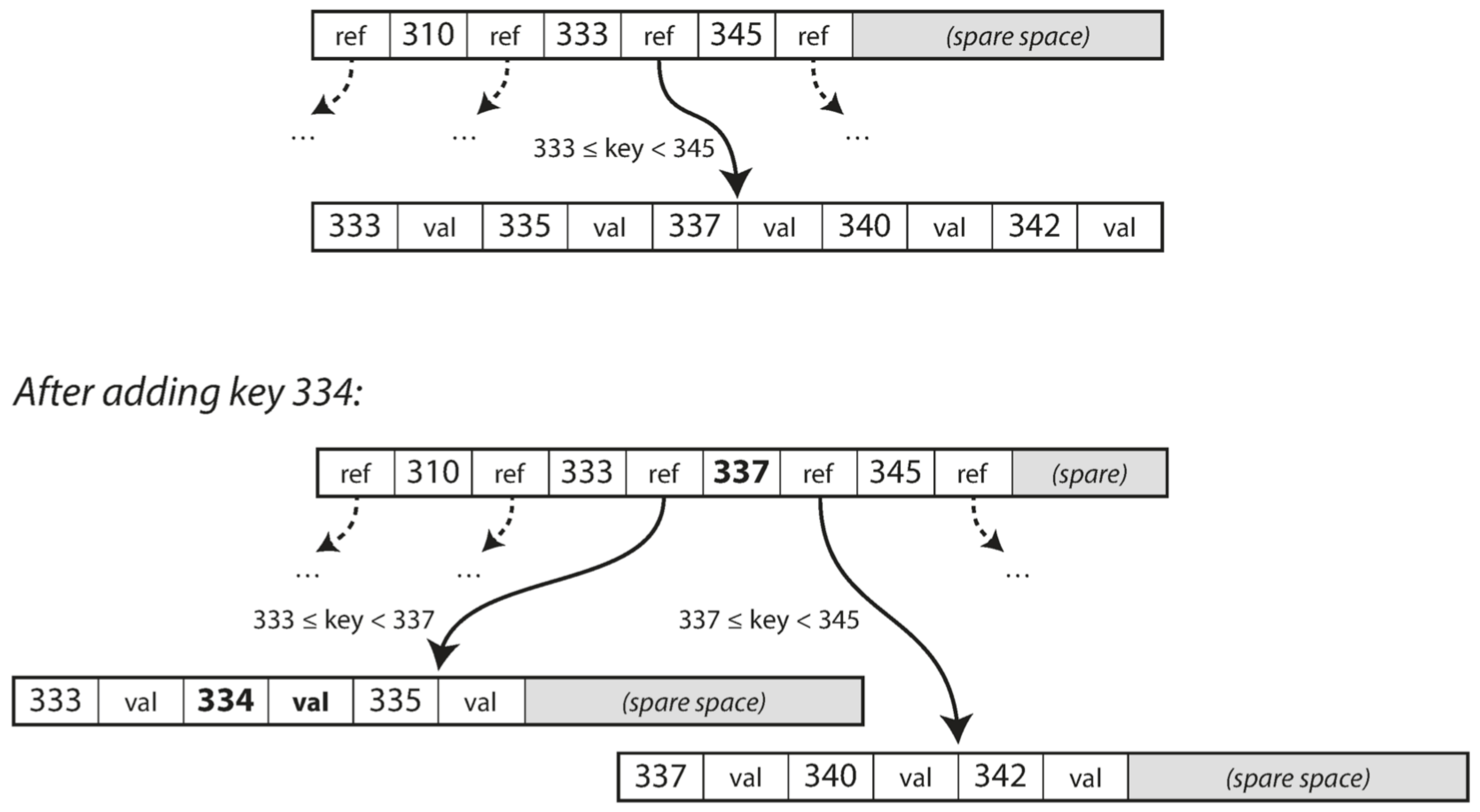
- Branching factors:
  - The number of references to child pages.
    - Typically several hundred.
- I/O is proportional to tree height.
  - Height can be less than BST.
- Fit more volume of data into the memory.
  - Most DBs are 3 or 4 levels deep.
  - A four-level tree of 4KB pages with a branching factor of 512 can store up to 256 TB.
    - (512^4) x 4kb = 256 TB (Disk)
    - Memory?
- B-tree was invented in 1970s.



B-Tree

Page splitting in B-tree

- What if we want to add a key and there is not enough space?
  - Split a page in a B-tree.
- B-tree is also a self-balance tree.



LSM-trees v.s. B-trees

- LSM-Trees
  - Faster for writes
    - Append-only
  - Slower for reads
    - Need to check multiple data structures
      - At different stages of compactions
  - Better compression
  - Higher CPU usages
  - What if write too fast? => Compaction configuration.
- B-trees
  - Faster for reads
    - Consistent data structure.
  - Slower for writes
    - Need to write to a log to address the implications of append-only.
  - Storage Fragmentation

In-memory database

- Why so much complexity?
  - Magnetic Disks and SSDs are awkward to deal with.
  - Slow, Donot support random address access.
  - But they are durable/persistent and cheap.
- New trends
  - RAM becomes cheaper and larger.
  - Battery powered RAM.
- In-memory database
  - Memcached, Memsql, Oracle TimesTen, Redis

In-memory database

- Multiple implementations.
  - Use in-memory database for caches only
  - Use disks as an append-only log only.
- Advantages
  - Counter intuitive!
    - Not because disk is slower.
      - Modern OSs do caching well.
    - Because of the data serialization.
      - Data representations in the memory and the disk
    - Simpler implementations.
    - Cost: Disk < Memory < Developers