

COMP2310/COMP6310

Systems, Networks, & Concurrency

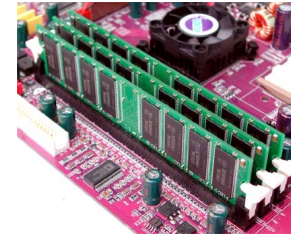
Convener: Shoaib Akram



Australian
National
University

Intention of these slides

- To make your understanding of memory and storage more concrete
 - **Not a lecture on databases or big data analytics**
- Memory
 - Accessed via load/store instructions
 - Fast random-access latency
- Storage
 - Accessed via filesystem calls or mmio (both require kernel support)
 - Fast sequential access and slow random access
 - To some degree the sequential-random gap is true even for solid-state drives that use semiconductor non-volatile memory (e.g., Flash memory) nature of block accesses



Data Intensive Applications

- Many online services today are data-intensive
- CPU power is often not a limiting factor
- Key reason
 - Today, **it is much easier to produce data** than to efficiently store and retrieve it
 - **Storage and retrieval are the new bottlenecks**
- Sources of data
 - Transactions, mobile messaging, social media, web documents, DNA sequencing, weather records, sensors in automobiles and airplanes, etc

Properties of Big Data

- **Velocity**

- Tweets **per second**, Likes **per second**, items added to Amazon buckets **per second**, new jobs appearing on LinkedIn **per second**, CCTV records, Netflix views, IMDB lookups, Whatsapp, new items on shelves at Coles

- **Variety**

- CSV, Email, JSON, JPEG, PDF, strings, MPEG

- **Volume**

- Many sources of easily producing new data leads to high volume
- How much data did you produce today?

What do organizations do with data?

- Service

- Point lookups
- How many items does Amazon have? How many web pages does Google manage?
- How long do you want to wait for a query?

- Insight

- To gain a competitive edge
- To learn patterns and behavior
- To exploit the interaction of online services and human behavior for profit

Discussion: Data vs. Meta-Data

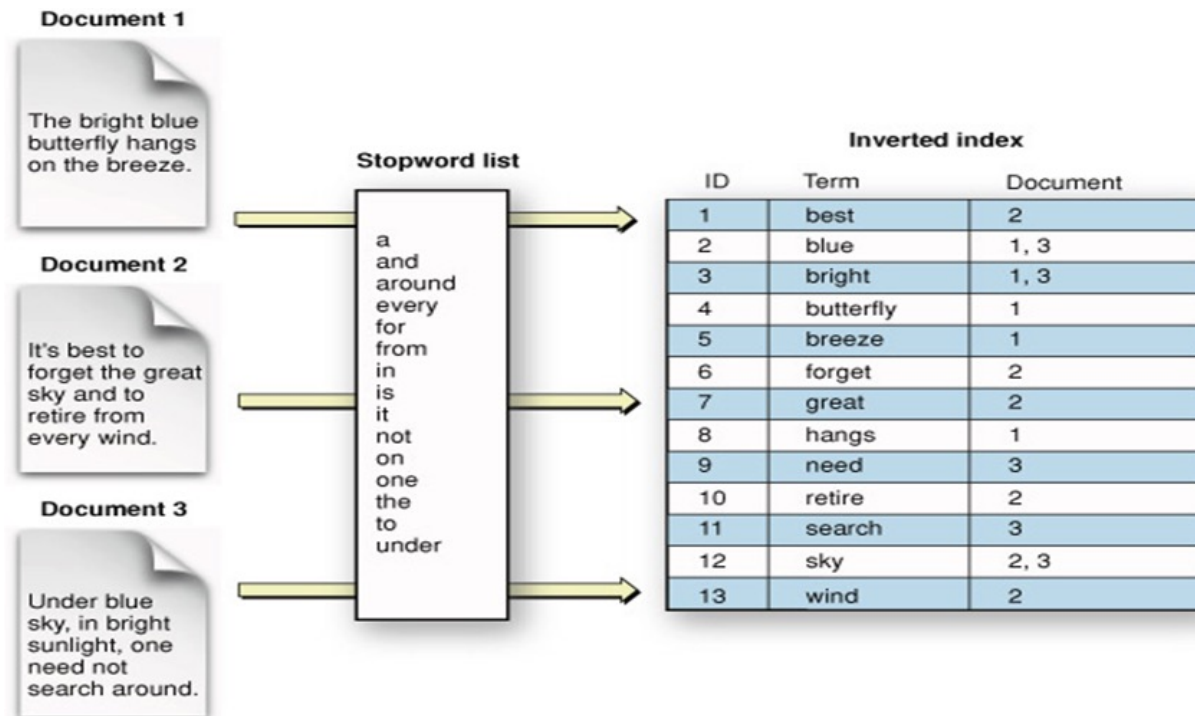
- Having data is not the critical part. Not all data is useful (at any point in time)
- To be able to serve and gain insight in a timely manner is key to survival
- Long-running (slow or tail) queries are often discarded as lost revenue
- Must keep 99.99999% of clients happy for long-term survival
 - Everyone is optimizing for tail latency (past: average latency)
- **Key realization:** Generating insight from data has a deadline
- **Problem:** Need efficient mechanisms to lookup data as fast as possible and to analyze it as efficiently as possible given hardware limitations
 - **Answer: Keep meta-data (typically in memory) to reach data fast**

Meta-Data: Motivation

- Billions of webpages and many terabytes of social media content on the web
- Searching for “land rover”
- If the service infrastructure:
 - stores all data on disk
 - performs a sequential search (e.g., grep)
 -
 -
 - no one will use the service!

Meta-Data: Motivation

- Solution: Index
- Let's look at example index used by search engines



Meta-Data: Motivation

- Another type of index used by key-value databases

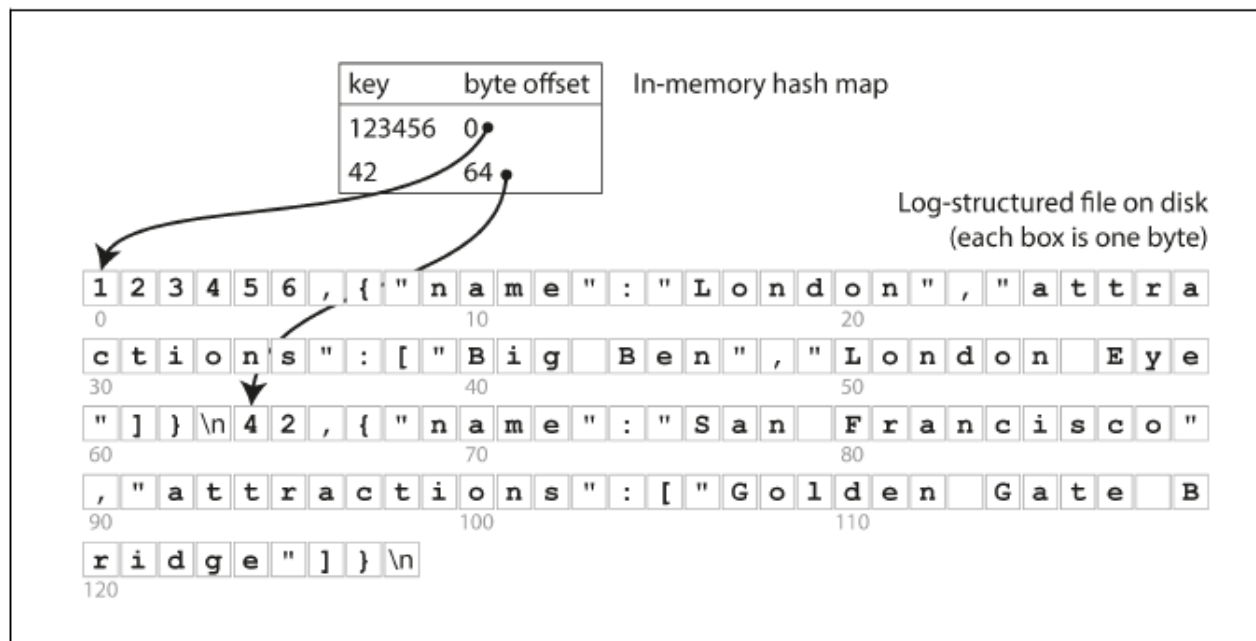


Figure 3-1. Storing a log of key-value pairs in a CSV-like format, indexed with an in-memory hash map.

Source: *Designing Data-Intensive Applications* by Martin Kleppmann

Discussion: Hardware Limitations

- Hash tables and indices typically reside in **main memory**
- The data they index reside on **storage**
- Main memory is **capacity limited**. What can be done about it?
 - move meta-data to storage
 - what is the **problem with storing hash maps on disk?**
 - find indexing structures that reduce main memory requirements of meta-data
 - find DRAM alternatives (phase-change memory, nanotubes, Spin-Torque Transfer memory → difficult uptake)

Typical Data Intensive System

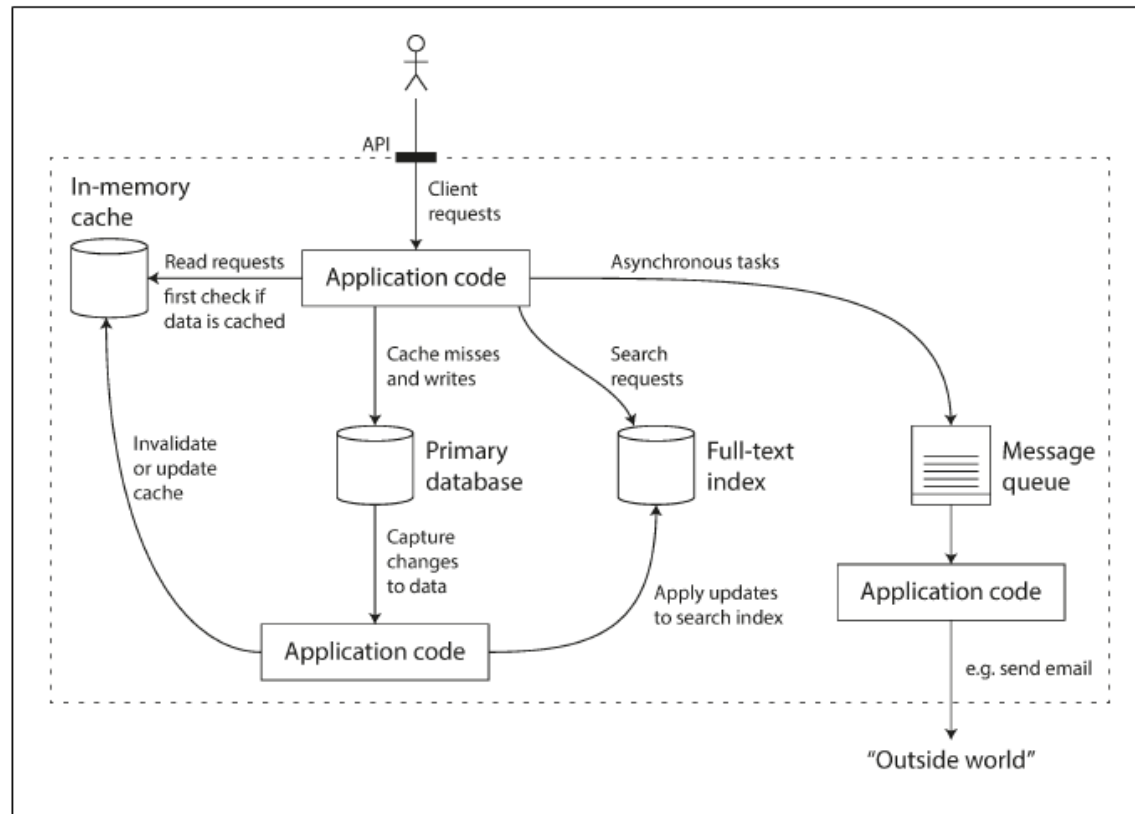
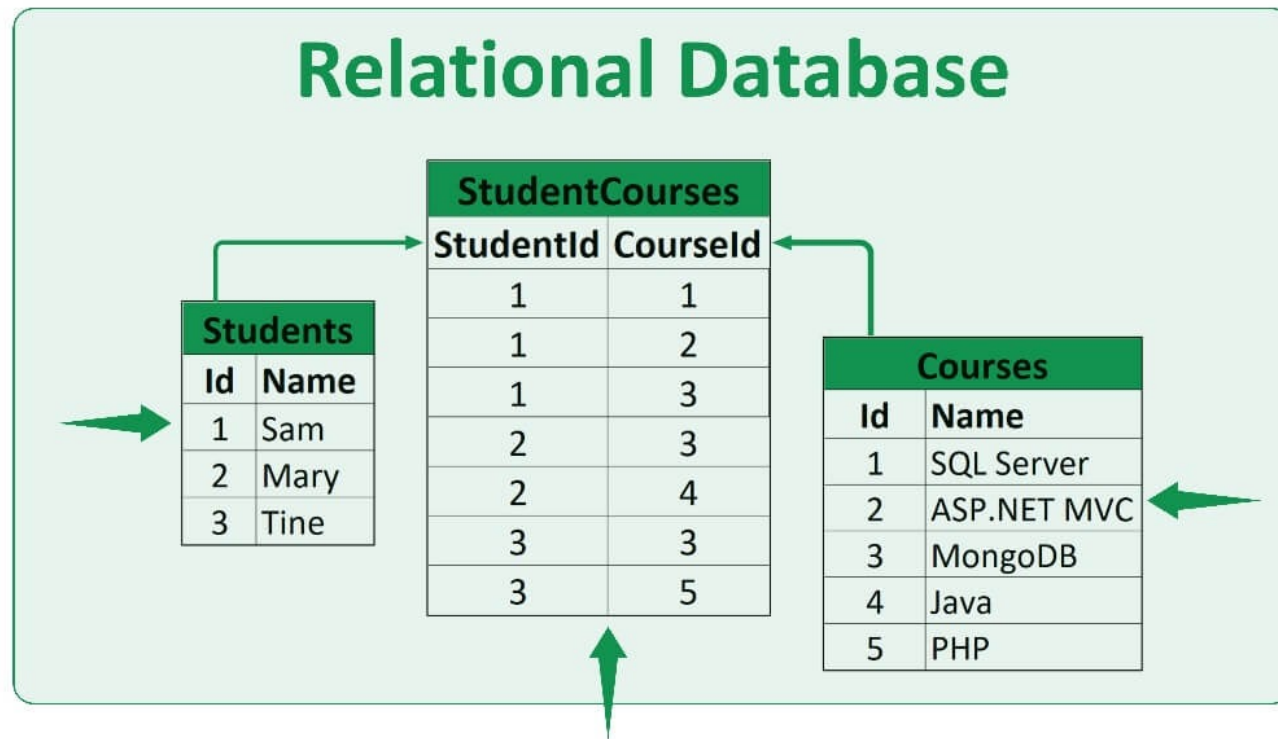


Figure 1-1. One possible architecture for a data system that combines several components.

Databases

- Database is an organized collection of data
- Relational databases
 - Organizes data into rows and columns with a well-defined structure (schema)
 - Tables are related to each other
 - Built on relational algebra
 - Required SQL queries to retrieve information
- NoSQL databases
 - Everything else! But typically, key-value store (database)

Example: Relational Databases



Key-Value Database

- Non-relational database that stores a collection of key-value pairs
- Keys and values can be anything (strings, arrays)
 - Keys are typically strings
 - Values can be strings or data structures
 - RocksDB, MemCached, Redis
- Suitable for modern services
 - Ease of scaling to billions of users
 - Ease of adding new “types” of data
 - No strict adherence to a pre-defined schema

Key-Value Database: Example

Phone directory

Key	Value
Paul	(091) 9786453778
Greg	(091) 9686154559
Marco	(091) 9868564334

MAC table

Key	Value
10.94.214.172	3c:22:fb:86:c1:b1
10.94.214.173	00:0a:95:9d:68:16
10.94.214.174	3c:1b:fb:45:c4:b1

source: <https://redis.com/nosql/key-value-databases/>

Hash Index

- Suppose our data storage consists of only appending updates to a file (it is called a append-only **sequential log**)
- Indexing strategy
 - Keep an in-memory hash table (map) where every key is mapped to a byte offset in the data file (the location at which the value can be found)
- Writes/Updates: Append the database file and update the hash entry
- Reads: ?

Hash Index

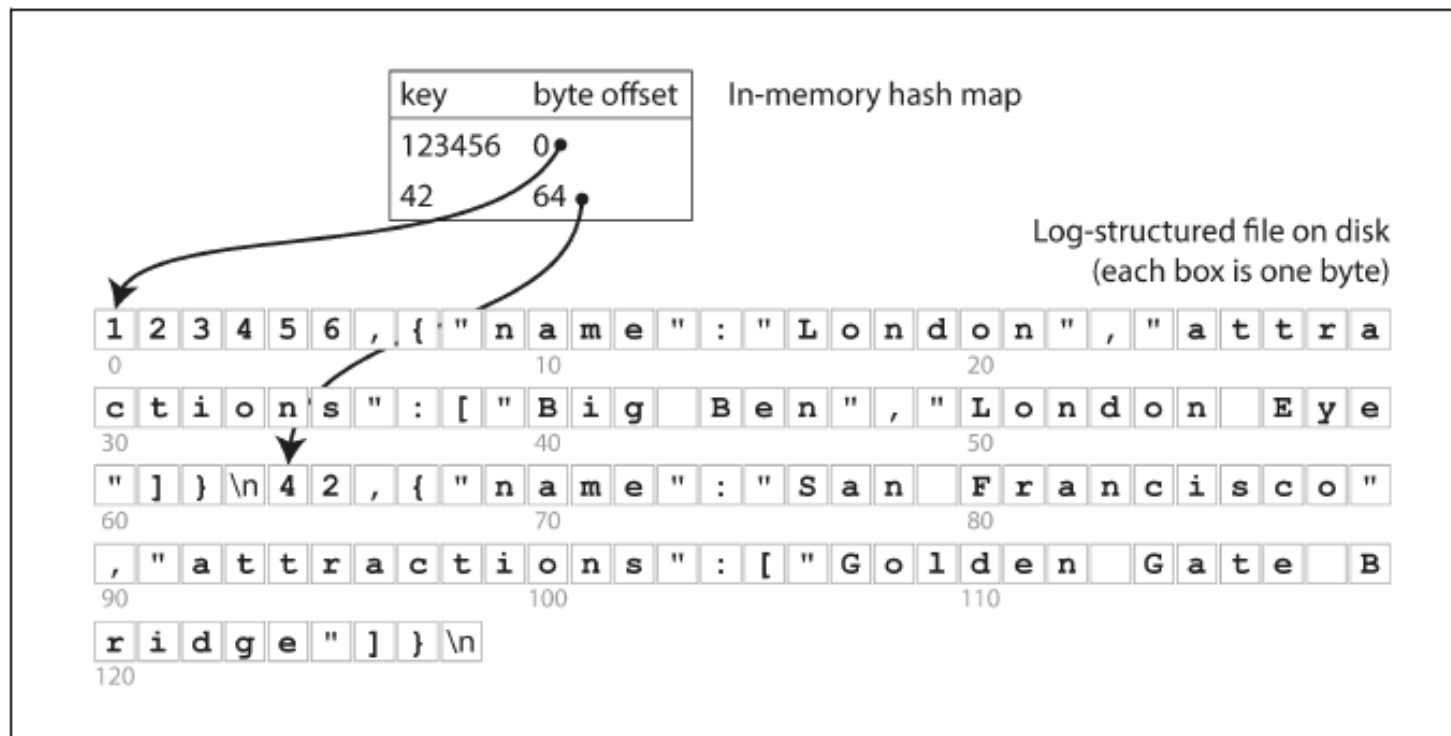


Figure 3-1. Storing a log of key-value pairs in a CSV-like format, indexed with an in-memory hash map.

Discussion

- Read performance
 - Good if there is only a single log segment
- Write performance
 - Append is the fastest way to perform updates in systems
- Drawbacks
 - Log segments incur a space overhead due to duplicates

Compaction

- Limit the size of each log segment
 - Make the segment read-only (immutable) once it reaches a threshold size
- Periodically compact the segments
- **Idea:** Can compact and merge multiple segments at a time
 - Eliminate internal and external fragmentation
- Such compaction can happen in the background by a different CPU core (thread or process)

Compaction

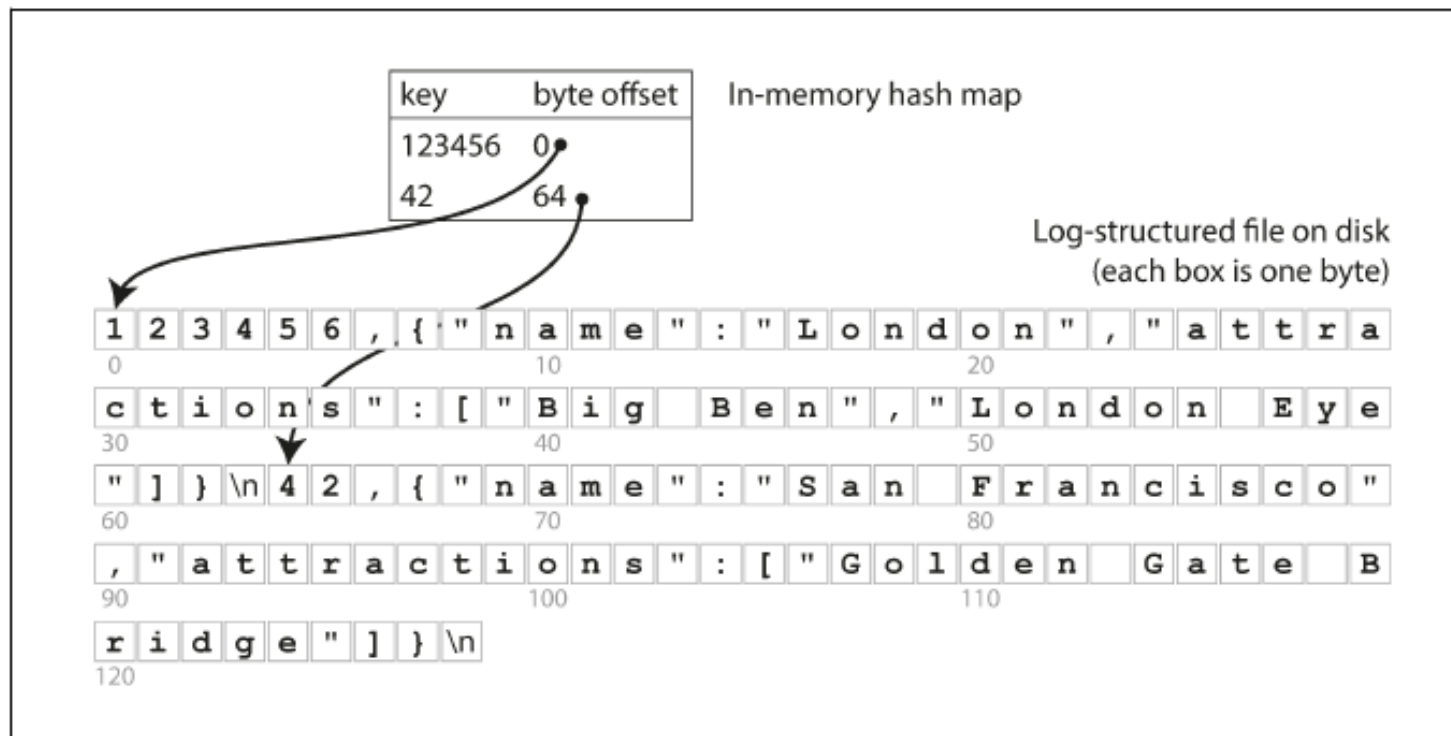


Figure 3-1. Storing a log of key-value pairs in a CSV-like format, indexed with an in-memory hash map.

Compaction & Merging

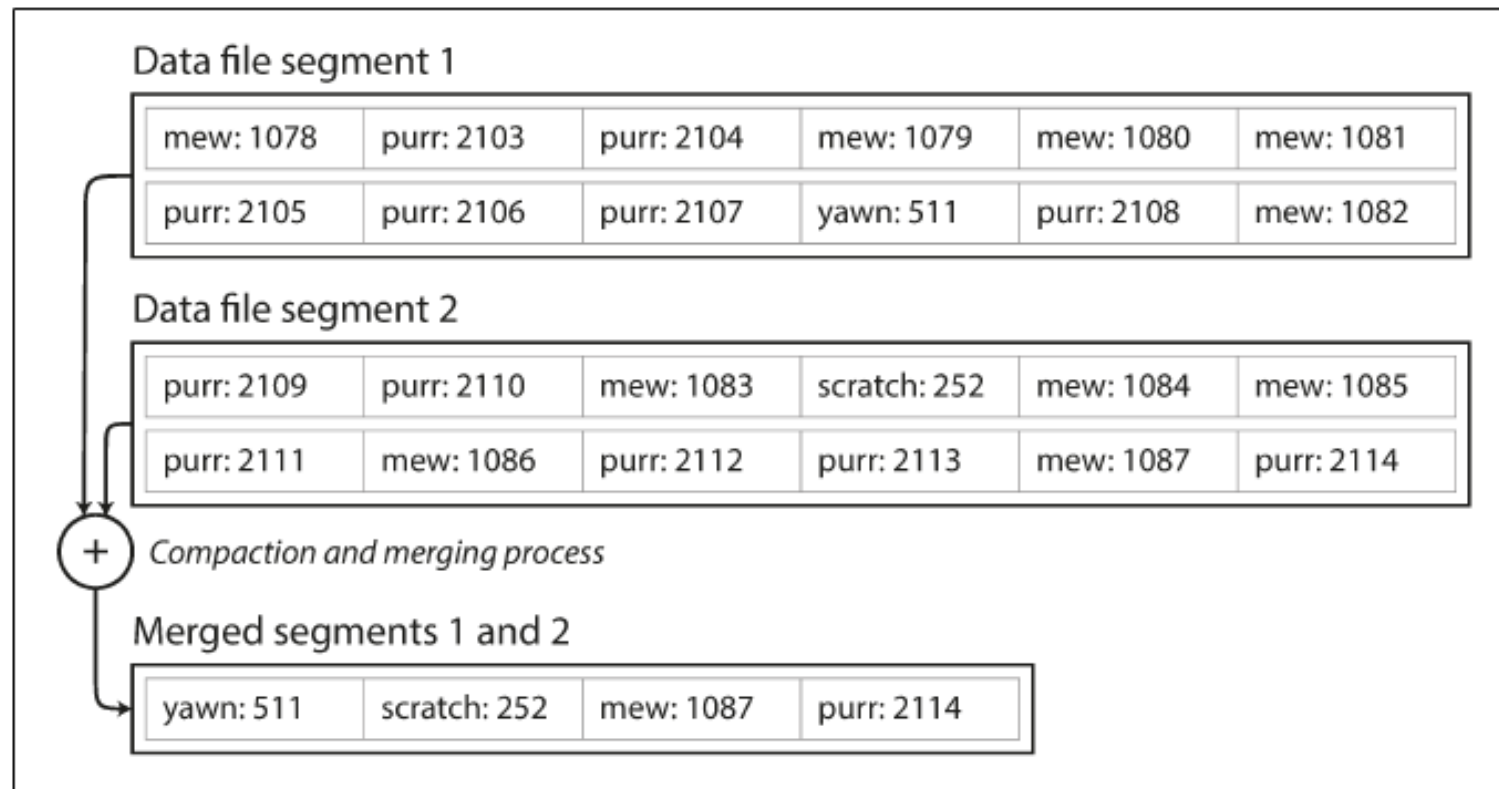


Figure 3-3. Performing compaction and segment merging simultaneously.

Limitations of Hash Indices

- Hash table must fit in memory
- As data grows on disk, the size of the hash table grows proportionally
 - Memory requirements proportional to data growth is a disaster
- Range queries are not efficient
 - Searching all keys between kitty0000 and kitty9999 requires looking up each individual key in the hash map

SSTables and LSM-Trees

- So far, each log segment is a sequence of key-value pairs
- These pairs appear in the order they are written
 - Later values for the same key are more important
 - Otherwise, there is not order
- Let's change the format of our segment files
 - sequence of KV pairs are sorted by key
 - can we still do sequential writes?
 - This format is called Sorted String Table or SSTable

Think!

- Merging segments is simple if each segment is already sorted by key
 - Simple merge sort algorithm
 - Segments can be much bigger than memory
- Read the input files side by side, look at the first key in each file, copy the lowest key (sort order) to the output file
- This produces a new merged segment file, also sorted by key
- What if the same key appears in several input segments?

Merging SSTables

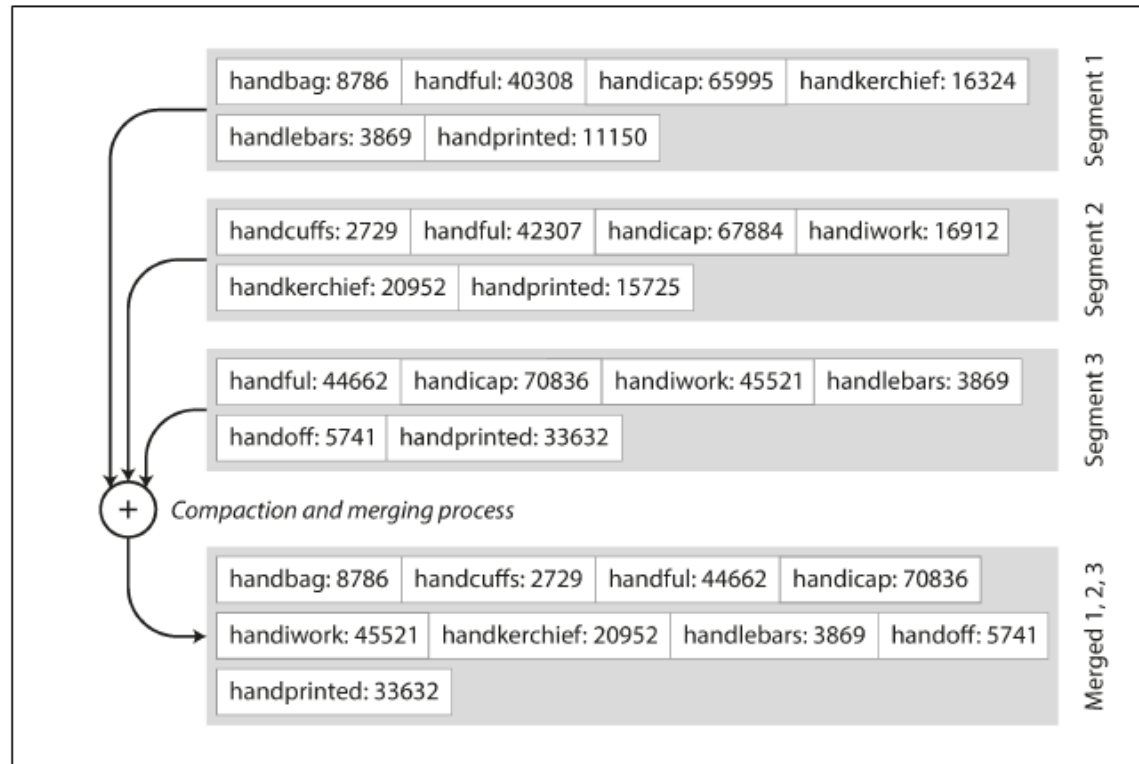


Figure 3-4. Merging several SSTable segments, retaining only the most recent value for each key.

Advantages of SSTables

- Merging segments that are much larger than memory is efficient due to the resulting sequential access pattern during merging
- No need to keep an index of all the keys in memory
 - Why is that?
- Still need an index to store the offsets of some of the keys but this index can be sparse

SSTable with an In-Memory Index

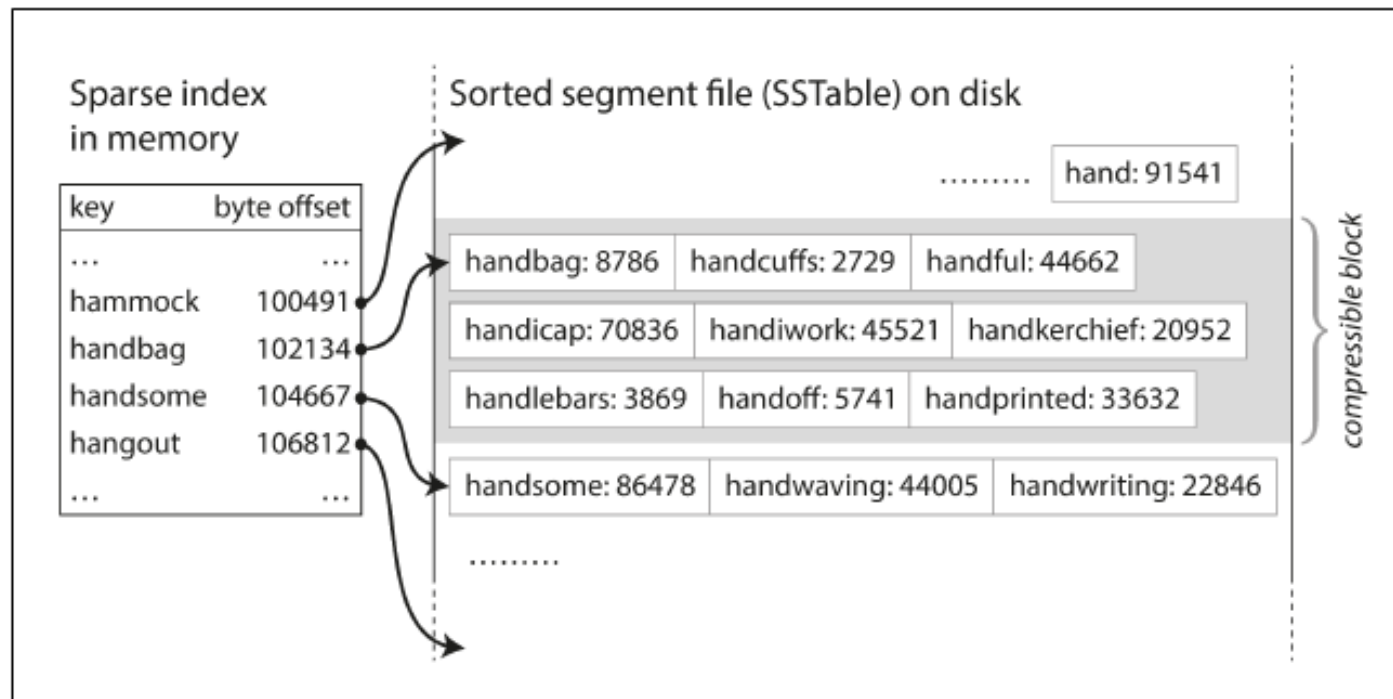


Figure 3-5. An SSTable with an in-memory index.

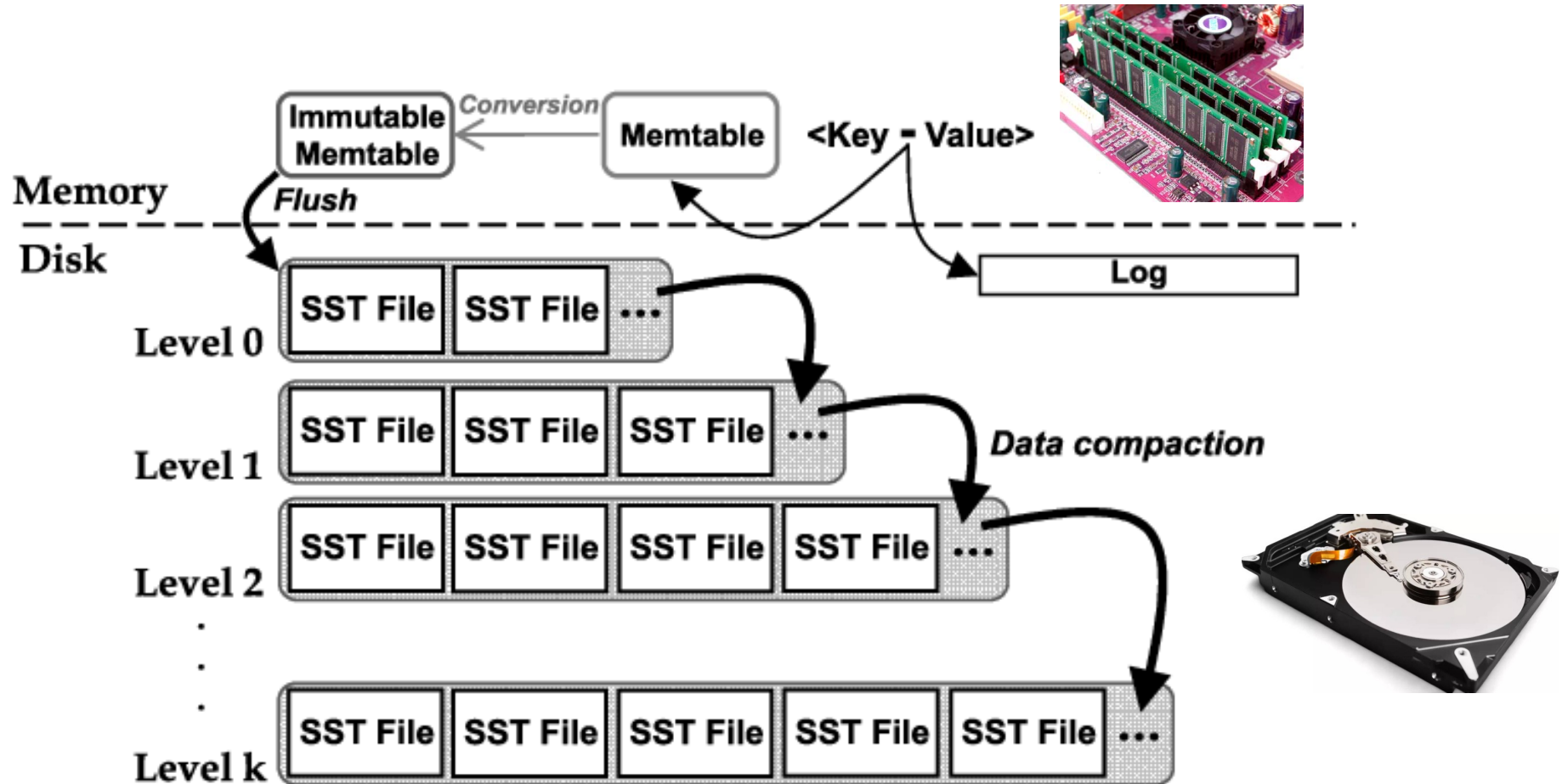
Constructing and Maintaining SSTables

- How do you keep the segments sorted?
- Use an in-memory data structure for ingesting (absorbing) fresh updates
 - This data structure is called a memtable (think of it as an in-memory segment)
- Memtable format
 - Option # 1: Red-black tree, AVL tree, skip list
 - Option # 2: Hash table
 - In this approach, memtable is sorted when it is made immutable

Working of an LSM Engine

- We can now make our storage engine work as follows:
 - **Writes**
 - When a write comes in, add it to the memtable
 - When memtable gets bigger than some threshold—typically a few megabytes, write it out to disk (**flush**) as an SSTable file
 - This can be done efficiently if the tree already maintains the key-value pairs sorted by key (otherwise sort during **flush**)
 - The new SSTable file becomes the most recent segment of the database
 - While the SSTable is being written out to disk, writes can continue to a new memtable instance
 - **Reads**
 - In order to serve a read request, first try to find the key in the memtable, then in the most recent on-disk segment, then in the next-older segment, etc.
 - From time to time, run a merging and compaction process in the background to combine segment files and to discard overwritten or deleted values

Working of an LSM Engine



Discussion

- Write performance
 - Nothing beats an update to an in-memory data structure such as memtable
- Read performance
 - Not as good as some alternatives because must perform lookups across all segments one by one
- Many optimizations to enhance read performance
 - Bloom filters to preclude segment search
 - Efficient merging strategies
- Alternative indexing structure is a B+ tree (out of scope)

Case Study: How Search Indices Work?

Document 1: Never arrive late

Document 2: Never say never

