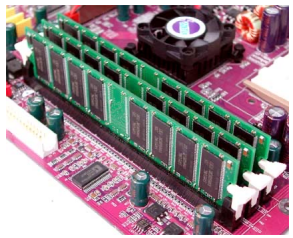
COMP2310/COMP6310
Systems, Networks, & Concurrency

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 do Amazon has? How many web pages do Google manages?
  - 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

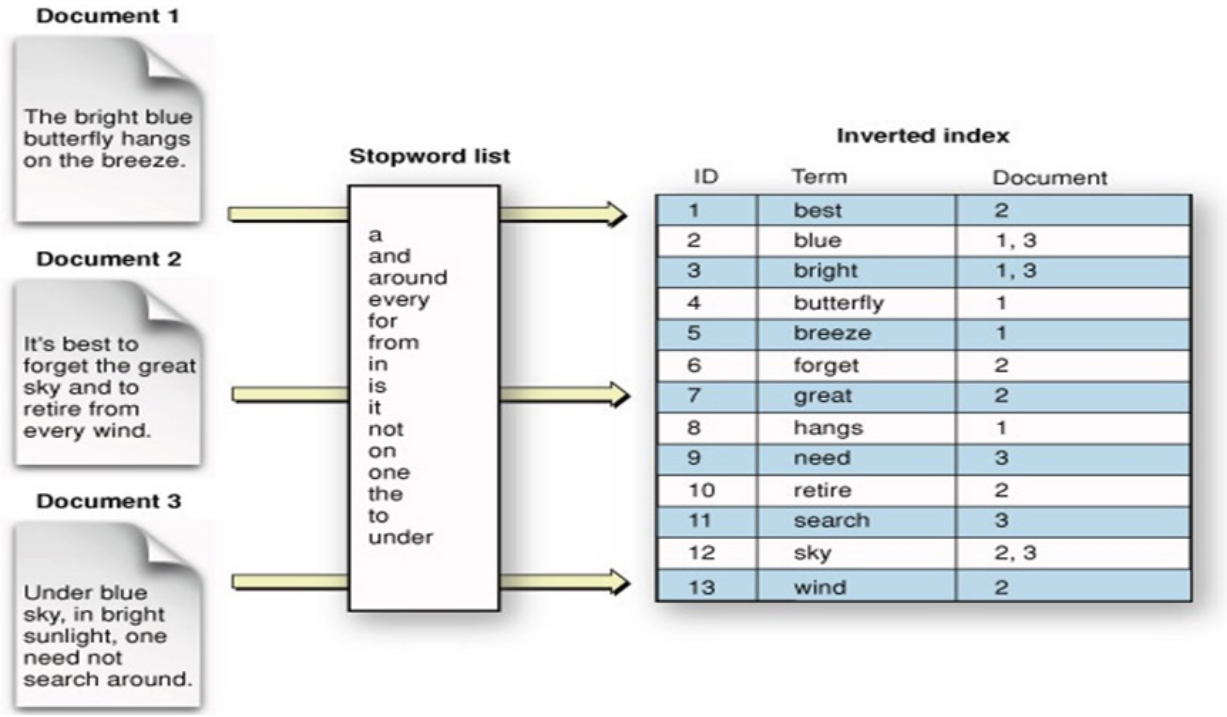
- Searching for “land rover”

- 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

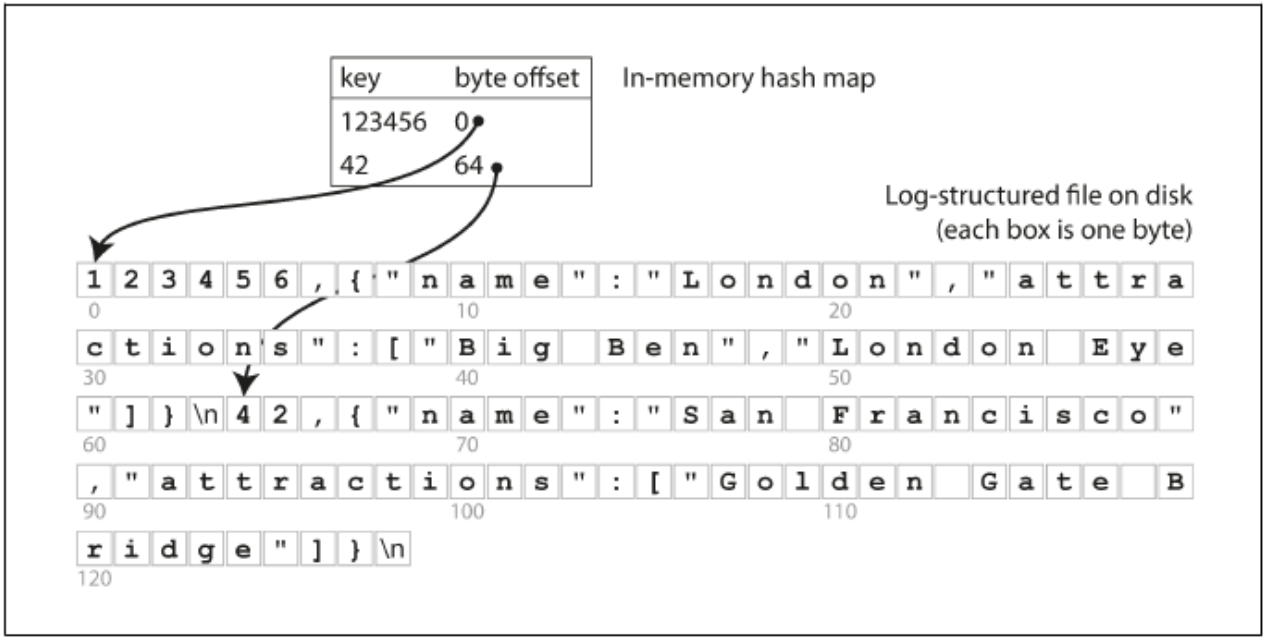


Figure 3-1. Storing a log of key-value pairs in a CSV-like format, indexed with an in-memory hash map.

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

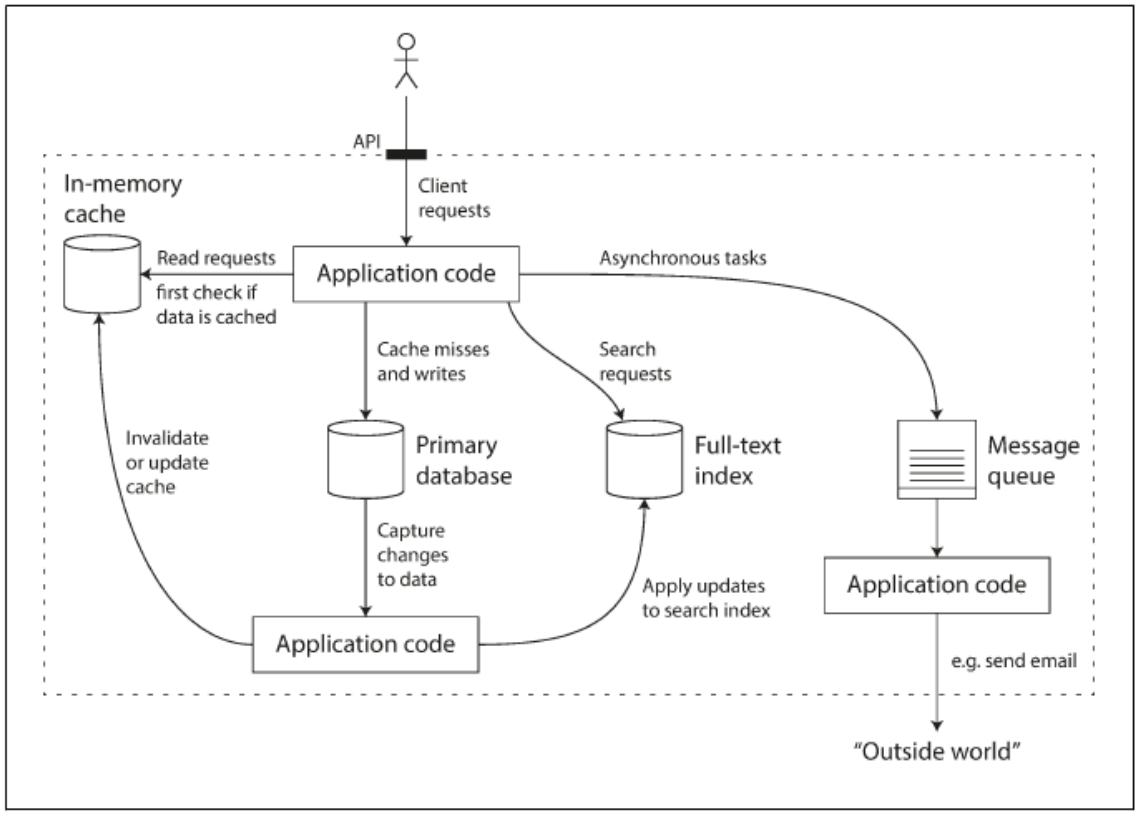


Figure 1-1. One possible architecture for a data system that combines several components.

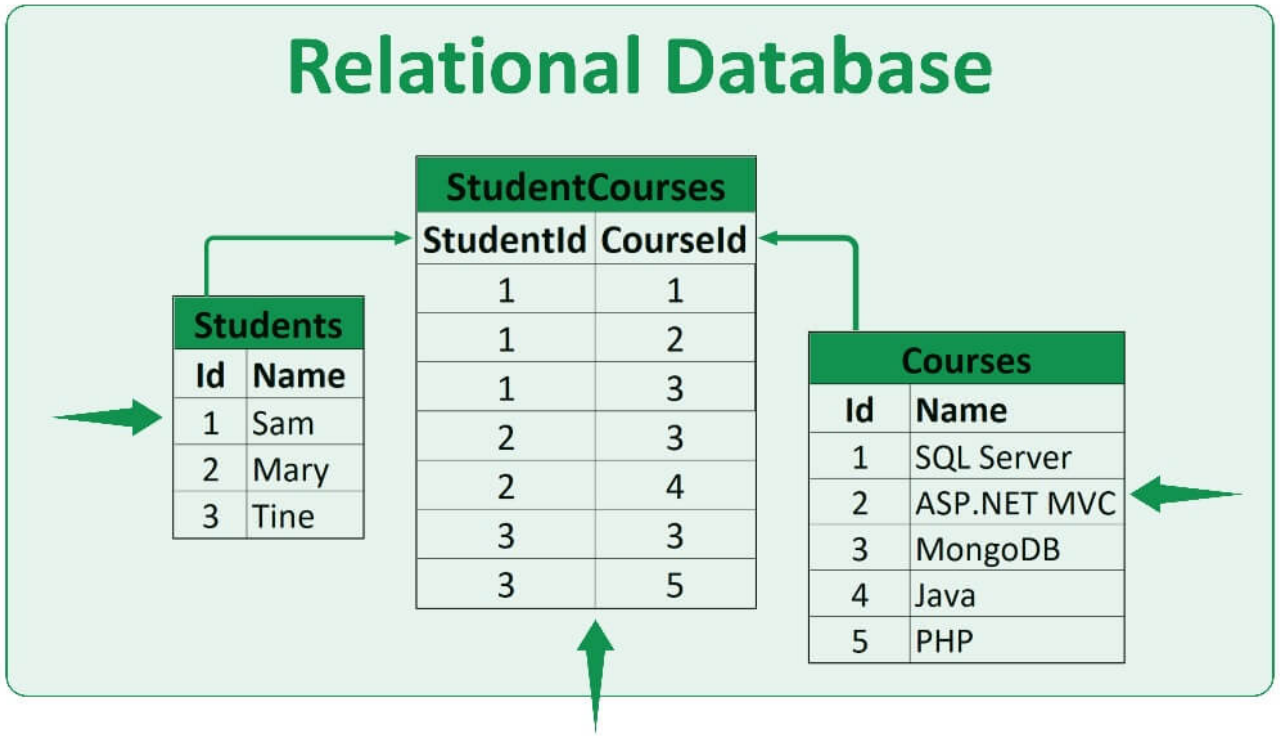
Databases

- 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

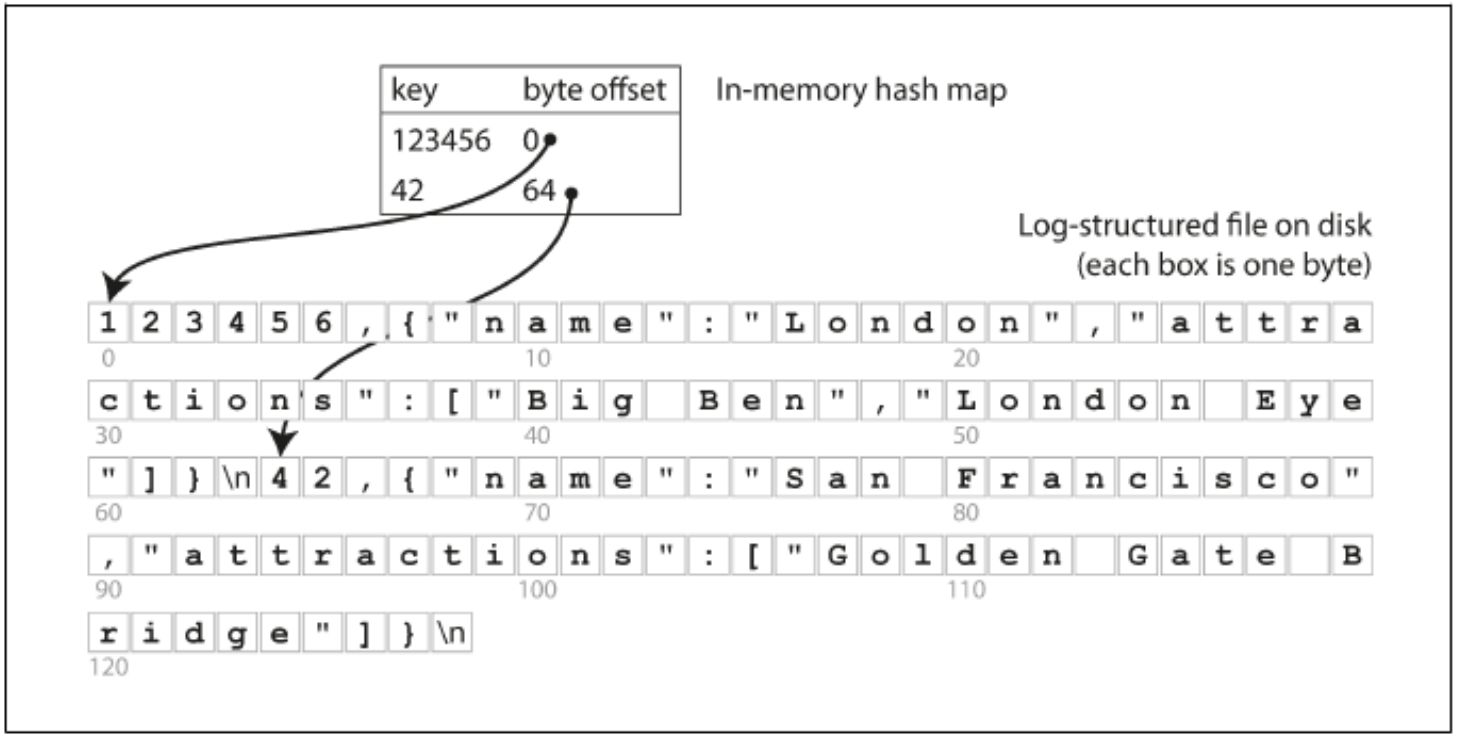


Figure 3-1. Storing a log of key-value pairs in a CSV-like format, indexed with an in-memory hash map.

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

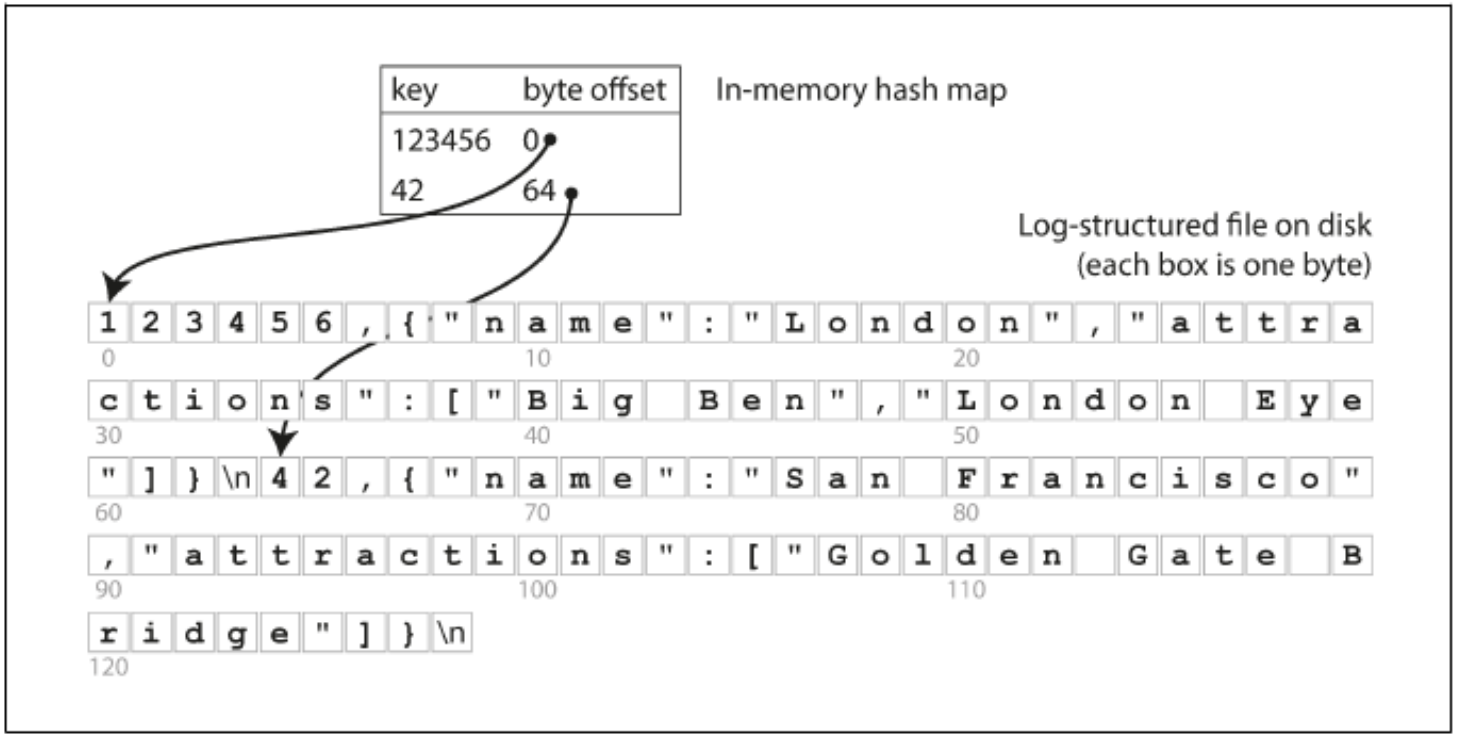


Figure 3-1. Storing a log of key-value pairs in a CSV-like format, indexed with an in-memory hash map.

Compaction & Merging

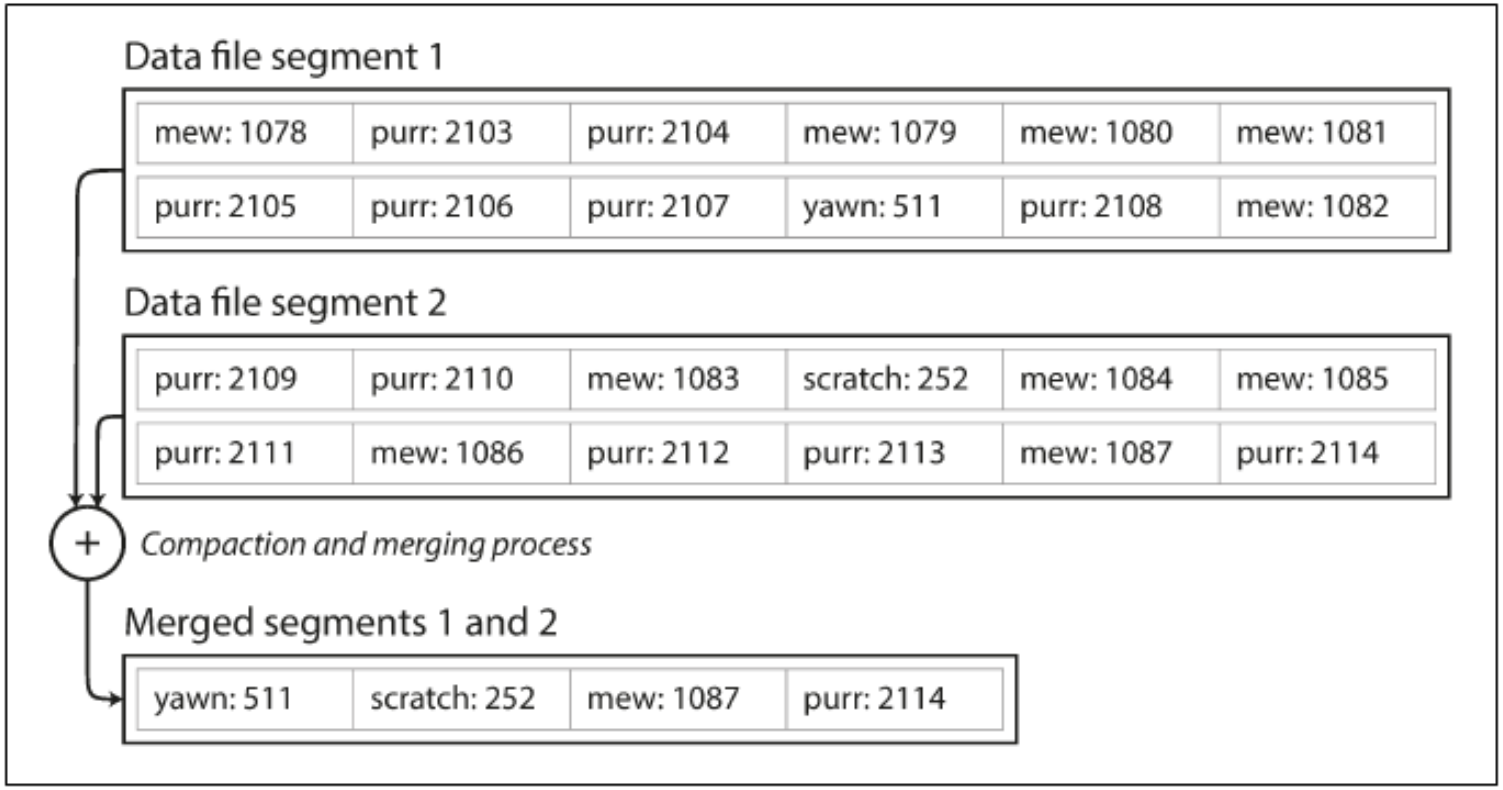


Figure 3-3. Performing compaction and segment merging simultaneously.

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 is the same key appears in several input segments?

Merging SSTables

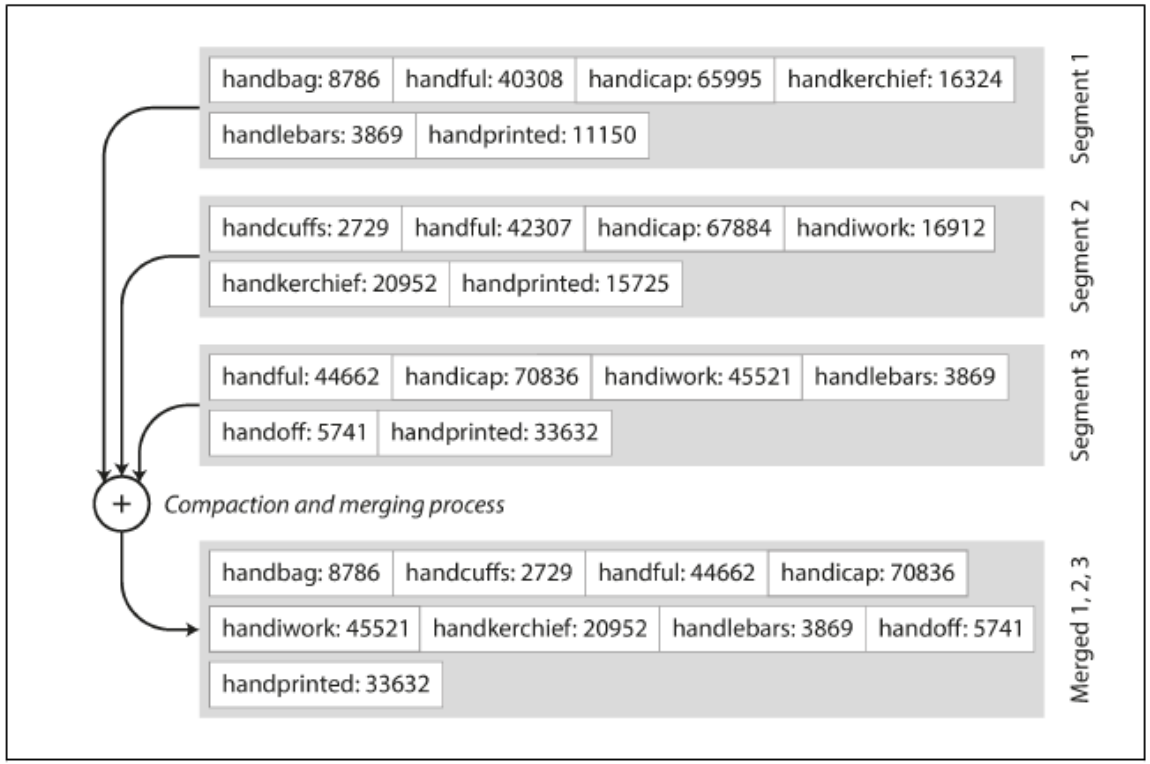


Figure 3-4. Merging several SSTable segments, retaining only the most recent value for each key.

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

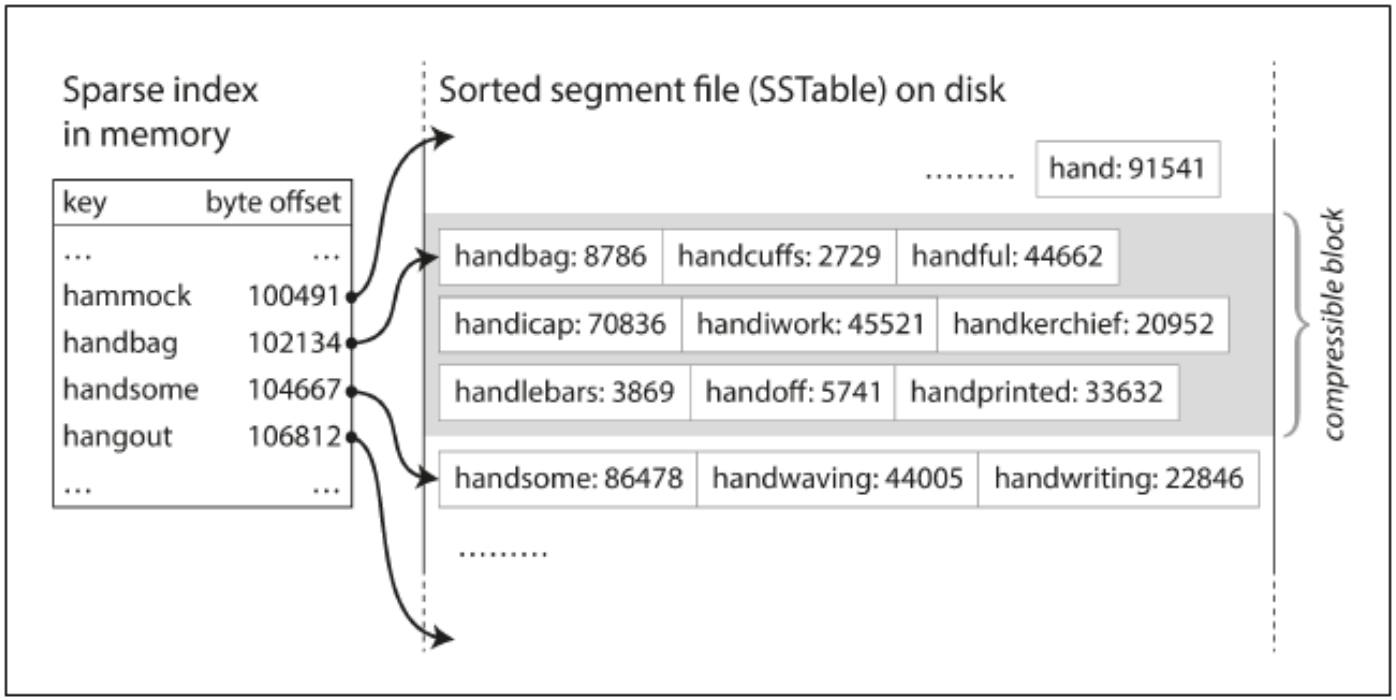


Figure 3-5. An SSTable with an in-memory index.

Constructing and Maintaining SSTables

- How do you keep the segments sorted?

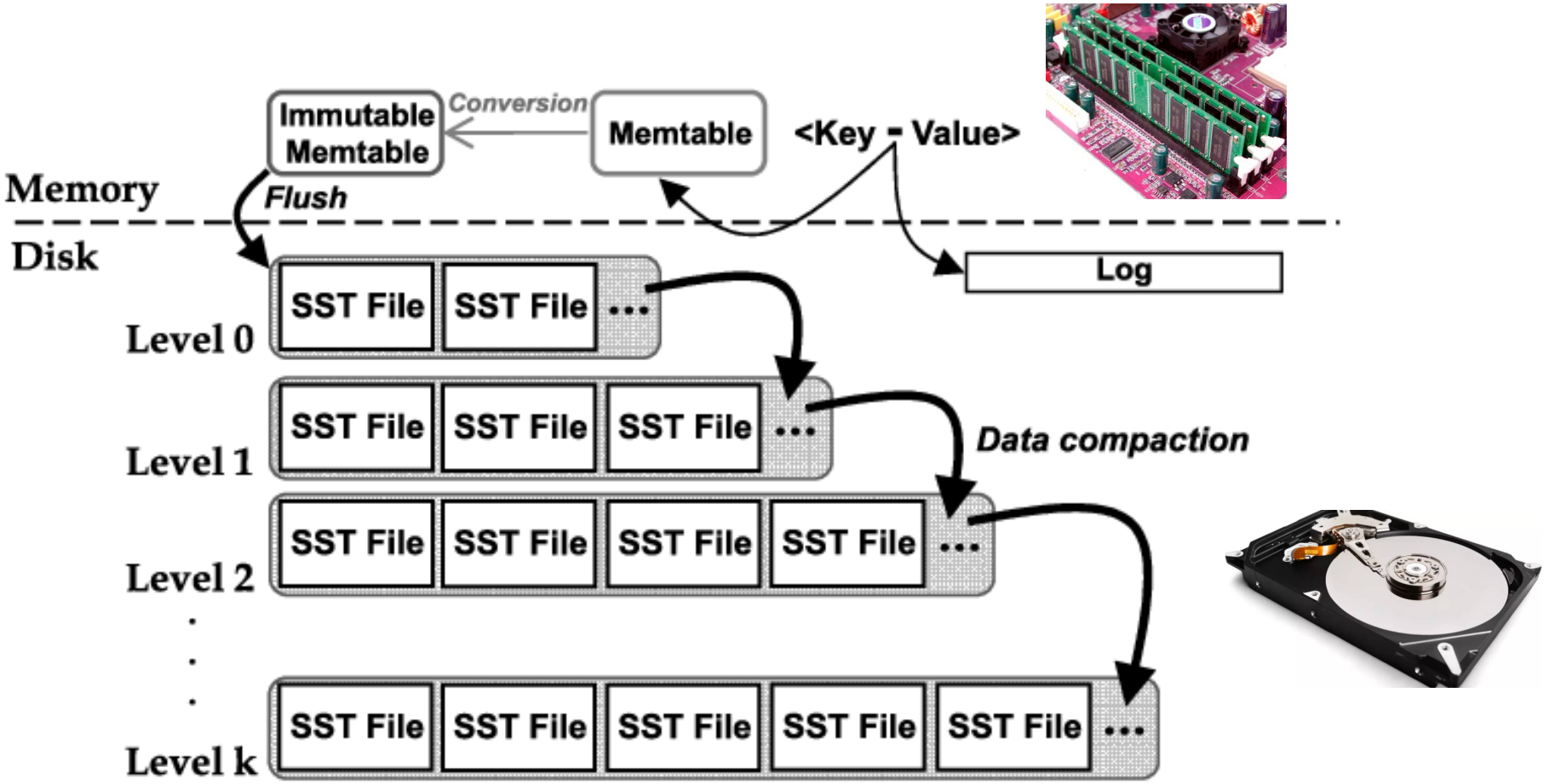
- Use an in-memory data structure for ingesting (absorbing) fresh updates
  - This data structure is called a memtable (think of it as an in-memory segment)

- 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



Compaction in LSM Engines

- Many different strategies over the years
- Out of our scope

Constructing and Analyzing the LSM Compaction Design Space

Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, Manos Athanassoulis
Boston University, MA, USA
ssarkar1@bu.edu,dstara@bu.edu,zczhu@bu.edu,mathan@bu.edu

ABSTRACT

Log-structured merge (LSM) trees offer efficient ingestion by appending incoming data, and thus, are widely used as the storage layer of production NoSQL data stores. To enable competitive read performance, LSM-trees periodically re-organize data to form a tree with levels of exponentially increasing capacity, through iterative *compactions*. Compactions fundamentally influence the performance of an LSM-engine in terms of write amplification, write throughput, point and range lookup performance, space amplification, and delete performance. Hence, choosing the *appropriate compaction strategy* is crucial and, at the same time, hard as the LSM-compaction design space is vast, largely unexplored, and has not been formally defined in the literature. As a result, most LSM-based engines use a fixed compaction strategy, typically hand-picked by an engineer, which decides *how* and *when* to compact data.

In this paper, we present the design space of LSM-compactions, and evaluate state-of-the-art compaction strategies with respect to key performance metrics. Toward this goal, our first contribution is to introduce a set of four design primitives that can formally define any compaction strategy: (i) the compaction trigger, (ii) the data layout, (iii) the compaction granularity, and (iv) the data movement policy. Together, these primitives can synthesize both existing and completely new compaction strategies. Our second contribution is to experimentally analyze 10 compaction strategies. We present 12 observations and 7 high-level takeaway messages, which show how LSM systems can navigate the compaction design space.

PVLDB Reference Format:
Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, Manos Athanassoulis. Constructing and Analyzing the LSM Compaction Design Space. PVLDB, 14(11): 2216 - 2229, 2021. doi:10.14778/3476249.3476274
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://disc.bu.edu/lsm-compaction.

1 INTRODUCTION

LSM-based Key-Value Stores. Log-structured merge (LSM) trees are widely used today as the storage layer of modern NoSQL key-value stores [36, 42, 45]. LSM-trees employ the *out-of-place* paradigm to achieve fast ingestion. Incoming key-value pairs are buffered in main memory, and are periodically flushed to persistent storage as *sorted immutable runs*. As more runs accumulate on disk, they are sort-merged to construct fewer yet longer sorted runs. This process is known as *compaction* [30, 42]. To facilitate fast *point lookups*, LSM-trees use auxiliary in-memory data structures (Bloom filters and fence pointers) that help to reduce the average number of disk I/Os performed per lookup [21, 22]. Because of these advantages, LSM-trees are adopted by several production key-value stores including LevelDB [32] and BigTable [17] at Google, RocksDB [30] at Facebook, X-Engine [34] at Alibaba, WiredTiger at MongoDB [62], CockroachDB at Cockroach Labs [18], Voldemort [40] at LinkedIn, DynamoDB [25] at Amazon, AsterixDB [3], Cassandra [8], HBase [7], Accumulo [6] at Apache, and bLSM [56] and cLSM [31] at Yahoo. Academic systems based on LSM-trees include Monkey [21], SlimDB [47], Dostoevsky [22, 23], LSM-Bush [24], Lethe [51], Silk [11, 12], LSbM-tree [58], SifrDB [44], and Leaper [63].

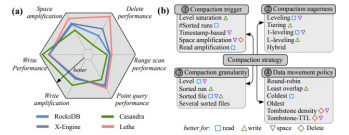


Fig. 1: (a) The different compaction strategies adopted in state-of-the-art LSM-engines lead to the diverse performances offered by the engines; (b) The taxonomy of LSM compactions in terms of the design primitives.

Compactions in LSM-Trees. Compactions in LSM-trees are employed periodically to *reduce read and space amplification at the cost of write amplification* while ensuring data consistency and query correctness [9, 10]. A compaction merges two or more sorted runs, between one or multiple levels to ensure that the LSM-tree maintains levels with exponentially increasing sizes [45]. Compactions are typically invoked when a level reaches its capacity, at which point, the compaction routine moves data from the saturated level to the next one, that has an exponentially larger capacity. Any duplicate entries (resulting from *updates*) and invalidated entries (resulting from *deletes*) are removed during a compaction, retaining only the logically correct (latest valid) version [28, 51]. Compactions dictate *how* and *when* disk-resident data is re-organized, and thereby, influence the physical data layout on the disk. Fig. 1(a) presents qualitatively the performance implications of the various compaction strategies adopted in state-of-the-art LSM-engines.

The Challenge: Hand-Picking Compaction Strategies. Despite compactions being critical to the performance of LSM-engines, the process of *choosing an appropriate compaction strategy* requires a human in the loop. In practice, decisions on *"how to (re-)organize data on disk"*, and thereby, *"which compaction strategies to implement or use"* in a production LSM-based data store are often subject

Discussion

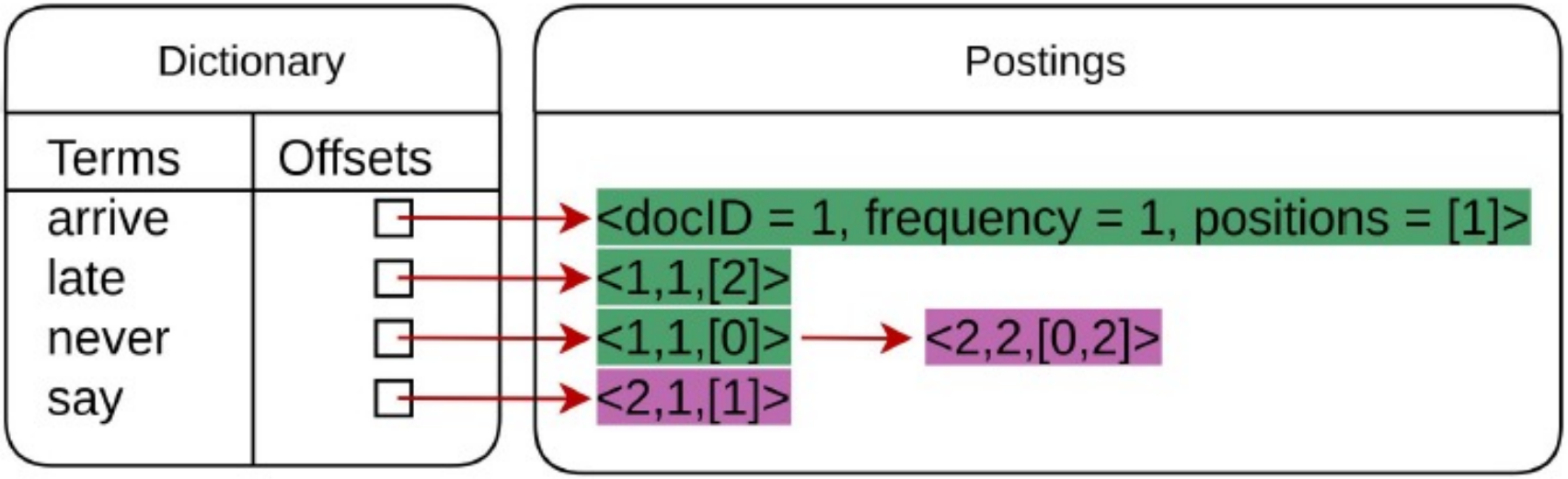
- Write performance
  - Nothing beats an update to an in-memory data structure such as memtable
- Read performance
  - Not as good as some alternatives because must perform lookups across all segments one by one
- Many optimizations to enhance read performance
  - Bloom filters to preclude segment search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

Popular Key-Value Stores

- RocksDB
- MongoDB
- MemCached
- Redis
- Apache Cassandra
- Amazon DynamoDB
    - Many are open source and you should consider exploring them!

Case Study: Indices in Search Engines

Document 1: Never arrive late
Document 2: Never say never



Some Resources in Search

- Apache Lucene
- https://github.com/twitter/the-algorithm/tree/main