Storing and retreiving large data volumes

Storage and Retrieval

TE 20/02/25

ResQ Club

- Zero food waste problem
- 120-160 million Kg of food waste in Finland

- ResQ Club
  - mobile app to buy surplus food by restaurants, cafes, hotels, grocery stores
  - food is discounted from 30 to 100%
  - pay and pick up the meal
  - 30.000 active users

How ResQ Works?

With the ResQ Club service you can easily bring in more customers and turn your surplus into profit.

1. **Easily announce surplus portions**

   Make offers of your unsold portions with our simple browser interface on any device.

2. **Customers purchase directly via ResQ app**

   ResQ Club notifies nearby users with a fitting diet. They can purchase portions directly via the app.

3. **Customers pick up orders in person**

   Customers retrieve purchased portions from your venue within the time frame you have specified.

Everyone **Benefits!**

For **You**	For **Customers**
Extra * Income	Quality Food
Sustainable Brand	Great Deals
New Clients	Discovering Local Food
Fresher Offering	ResQ the Planet

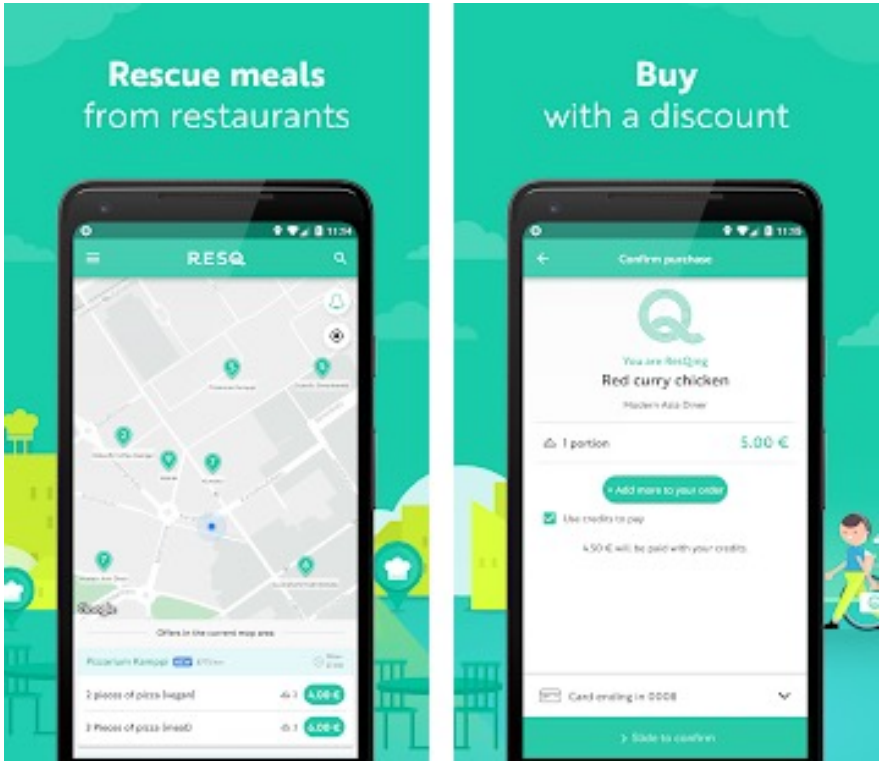
A Risk-free Opportunity

There are no subscription fees - we only deduct a commission of realized sales. You receive your ResQ income once per month - we cover all the card fees.

* 6 portions at $ 7.5 sold per day = **£ 7 500** extra annually.

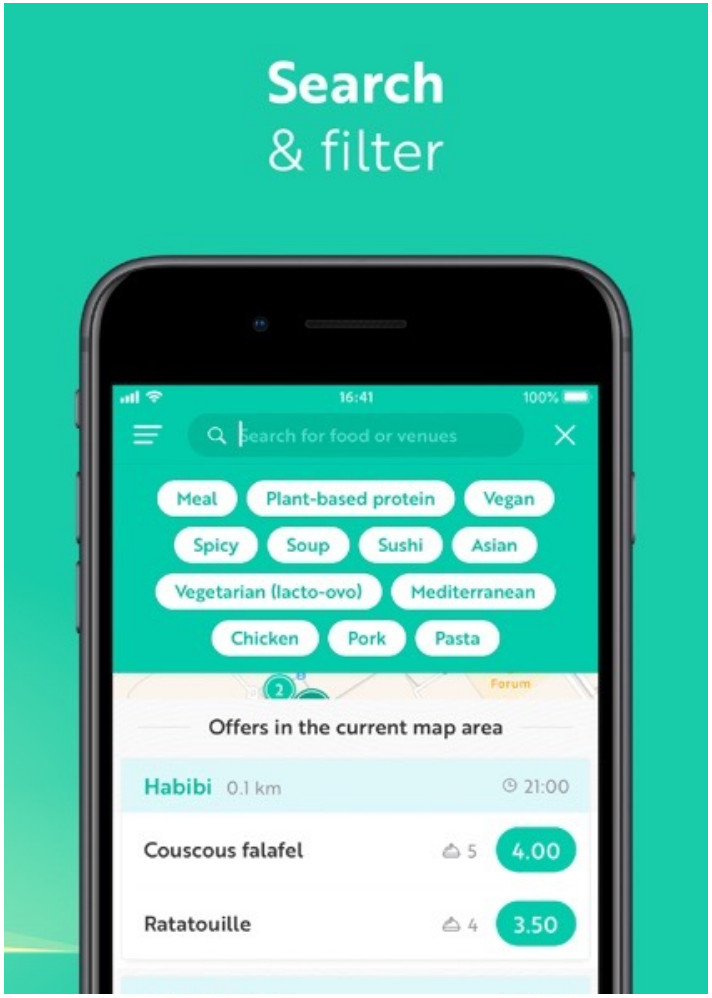
How does it work?

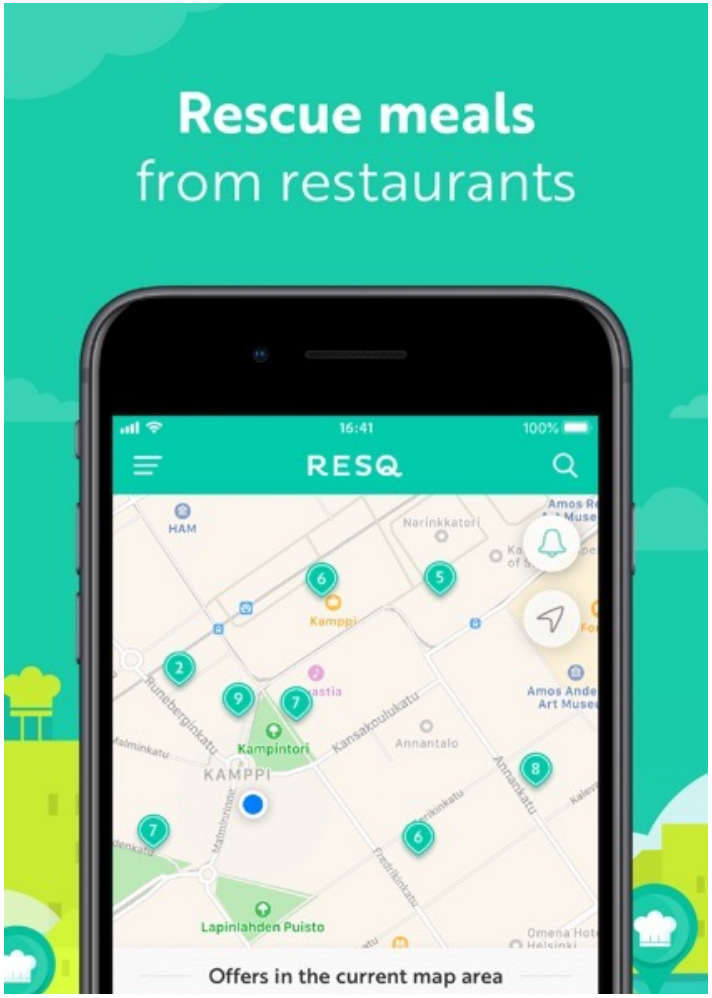
- Relevant elements?
- How elements are related?



Data view

- Which information do we need for the project?
- Which entities are relevant?
- Which features define these entities?
- How do these entities are related?





Relevant entities?

restaurants

Data ingestion problems

How can we store the information that we receive?

How can we restore this information quickly?

What if we have a pattern of large amount of writes and few reads?

Example

- Simplest possible data base:
- Write a pair of {key, value}: (1234, Sushi restaurant)
- If we need a value, we request its key: get(1234)

```
123, Pizza parlor
101, Bakery
361, Soup central
778, Salad hut
```

DB data structures (python-like)

```
db_set(){
    //append at the end of file database
    database.write(line_to_append + '\n')
}
db_get(){
    //look up word at database file content
    pattern = r'\b' + word_to_find + r'\b’
    found = re.findall(pattern, database)
}
```

How to write and read from DB?

```
db_set 123456 '{"name":"super pizzeria","food":["pizza"]}'

db_set 42 '{"name":"pastafiore","food":["spaghetti","Lasagna"]}'

db_get 42

{
   "name":"pastafiore",
   "food":["spaghetti","Lasagna"]
}
```

simple DB internal structure

DB is a text file with a (key, value) pair at each line

- db_set(key,value): append (k,v) **at the end of file**
- db_get(key): **return the value** of a specific key

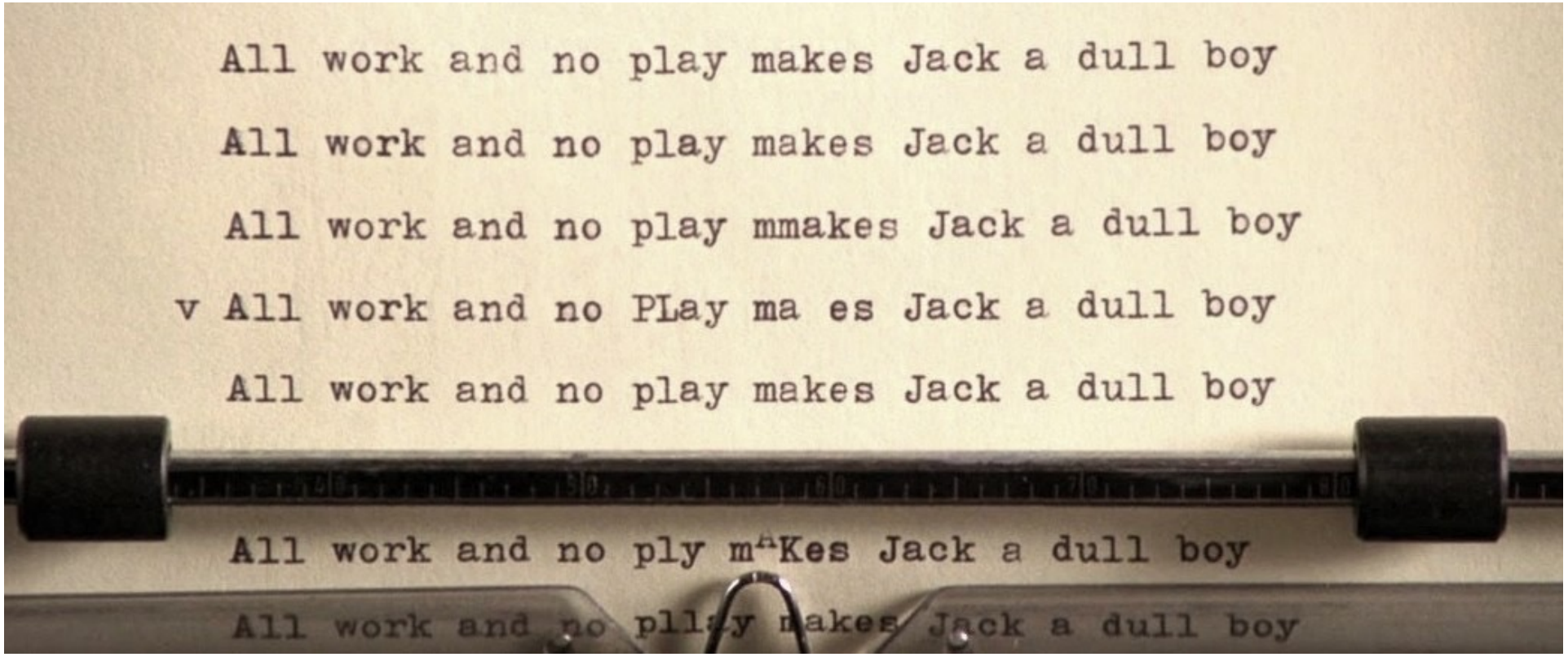
my append-only database: set and get

```
db_set 42 '{"name":"pastafiore","food":["spaghetti","Lasagna"]}'

db_get 42
{"name":"pastafiore","food":["spaghetti","Lasagna"]}

cat database
123456 '{"name":"super pizzeria","food":["pizza"]}'
42 '{"name":"pastafiore","food":["spaghetti","Lasagna"]}'
```

Log files: append-only sequence of records



Performance review

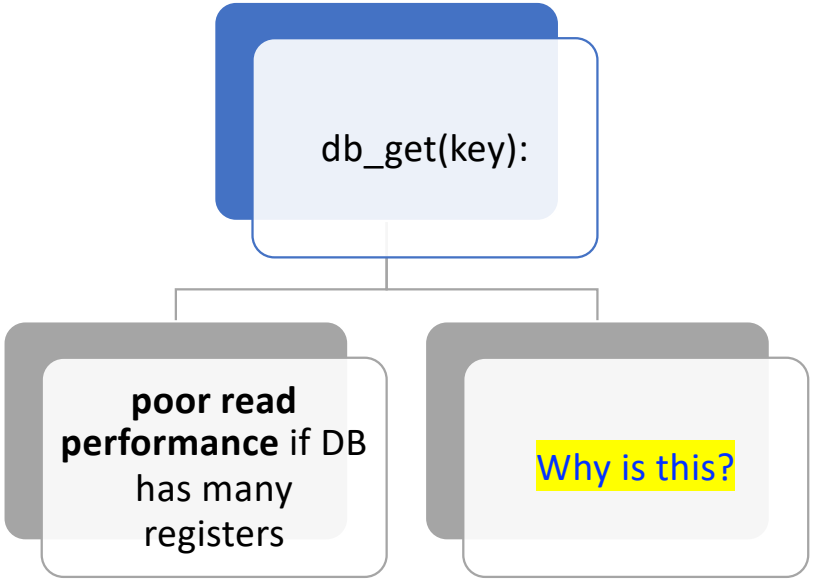
db_set(key,val)

good write performance

End file append operation is **fast**

log file: append-only sequence of records

Performance review



db_get(key) performance review

- Each query starts at file first element
- Reads each element of the file from beginning to end
  - **finds the last instance** of the key that we are looking for

- Complexity cost: **O(n)** where n is the number of registers
- Double number of registers: we will need double time

Use an **index** to accelerate reads

Additional data structure: from original data

Allows search for values efficiently <O(n) :

New problem: need to generate and store metadata to find values quickly

Costs of using an index

Does not affect data or DB content

Accelerates value **search**

Index updates affect DB write operations: **slower**

need to add data and then update index

Costs of using an index

Trade-off when using an index to big data applications

- reads are faster
- write operations are slower

Data engineer responsibility:

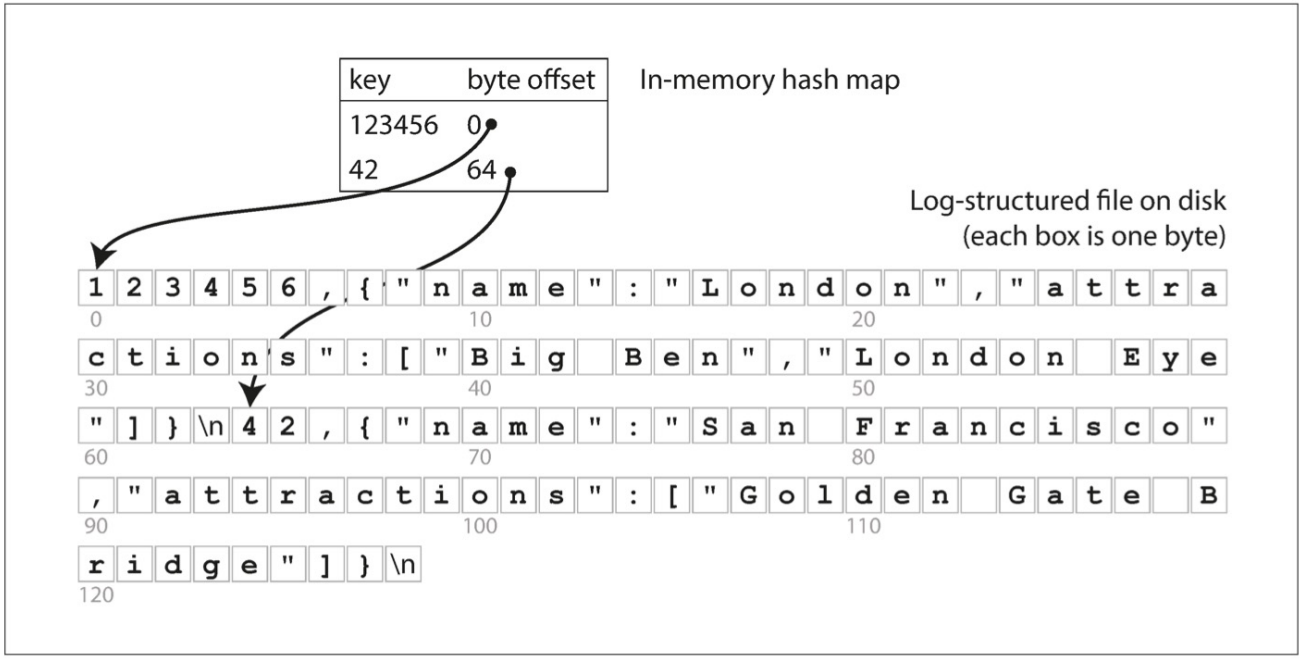
- quick reads? OK
- not-so-fast (but acceptable) write operations are good for our app?
- Answer will change during project growth

Common Index types

- Hashes
- Sorted String Tables
- Log-structured Merge Trees

Hash index to store key-values

- Python dictionary, HashMap Java, map container C++
- Hash map **in memory:** store the **position of the key** in file



Hash index to store key-value

WRITE: New value to store?

Write operation with hash index:

Append key and value **at the end of the file**

Append **new key position at hash map**

Hash index to store key-value

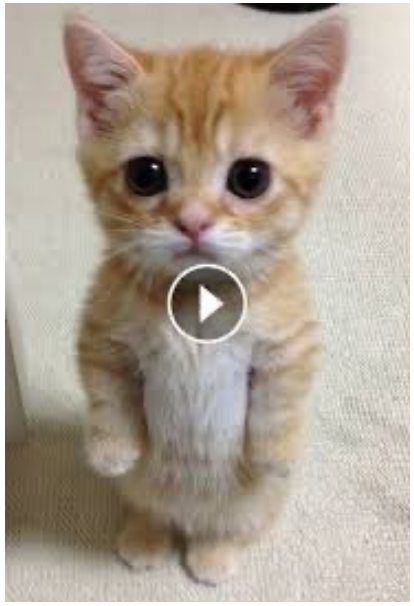
READ: Looking for pizza at “ristorante italiano”?

Hash map tells us the **position of the element in the file**

Read operations with hash index:

- Disk header has to be placed in data location
- Read operation of value from disk

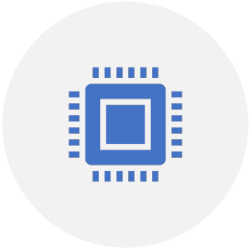
Bitcask – Riak storage engine



- 👍 Good read and write performance
- 👍 Hash-map has to be in memory
- Application:
  - Key: video URL
  - Value: number of visualizations
- Many writes per key
- Number of keys is not large
- We must store the whole hash in memory (big data?)
- https://github.com/basho/bitcask

Disk space management

How do we manage running out of disk space?



Break log file (our data!) in fixed sized files

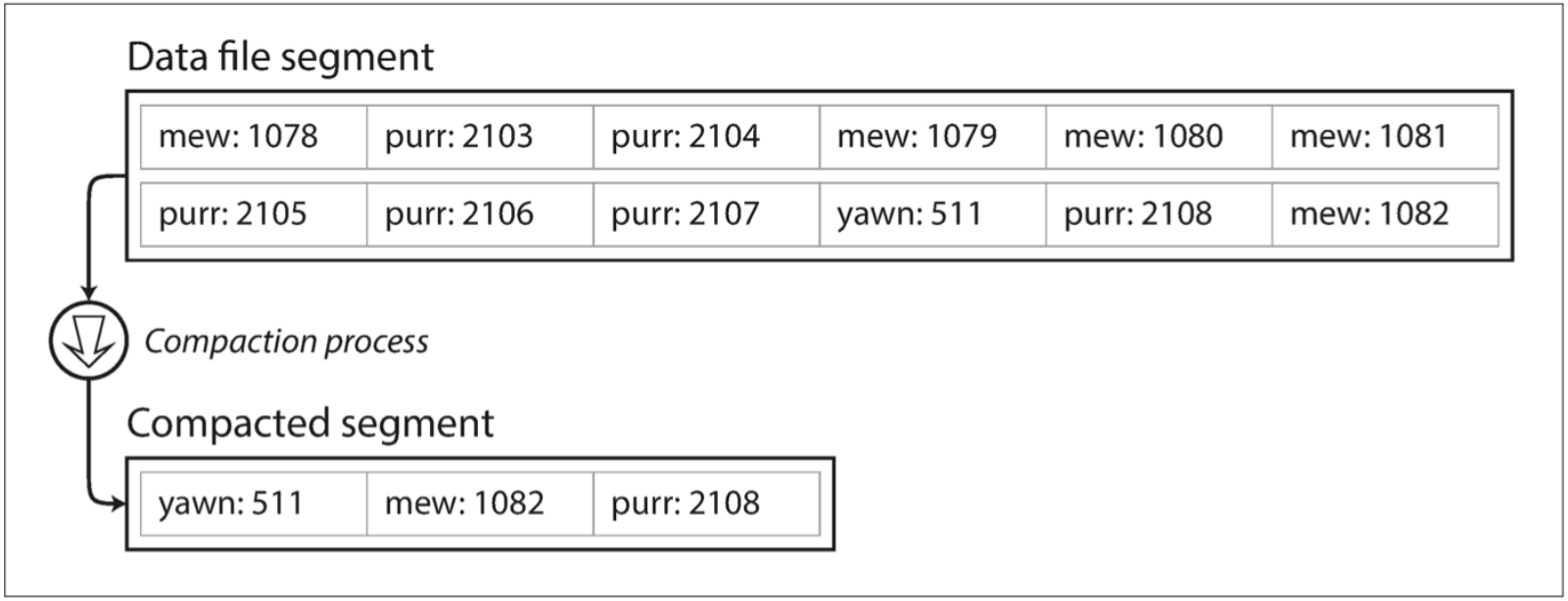
File is full?

Next write operation to new data log file

We can **compact files** old log files to save space

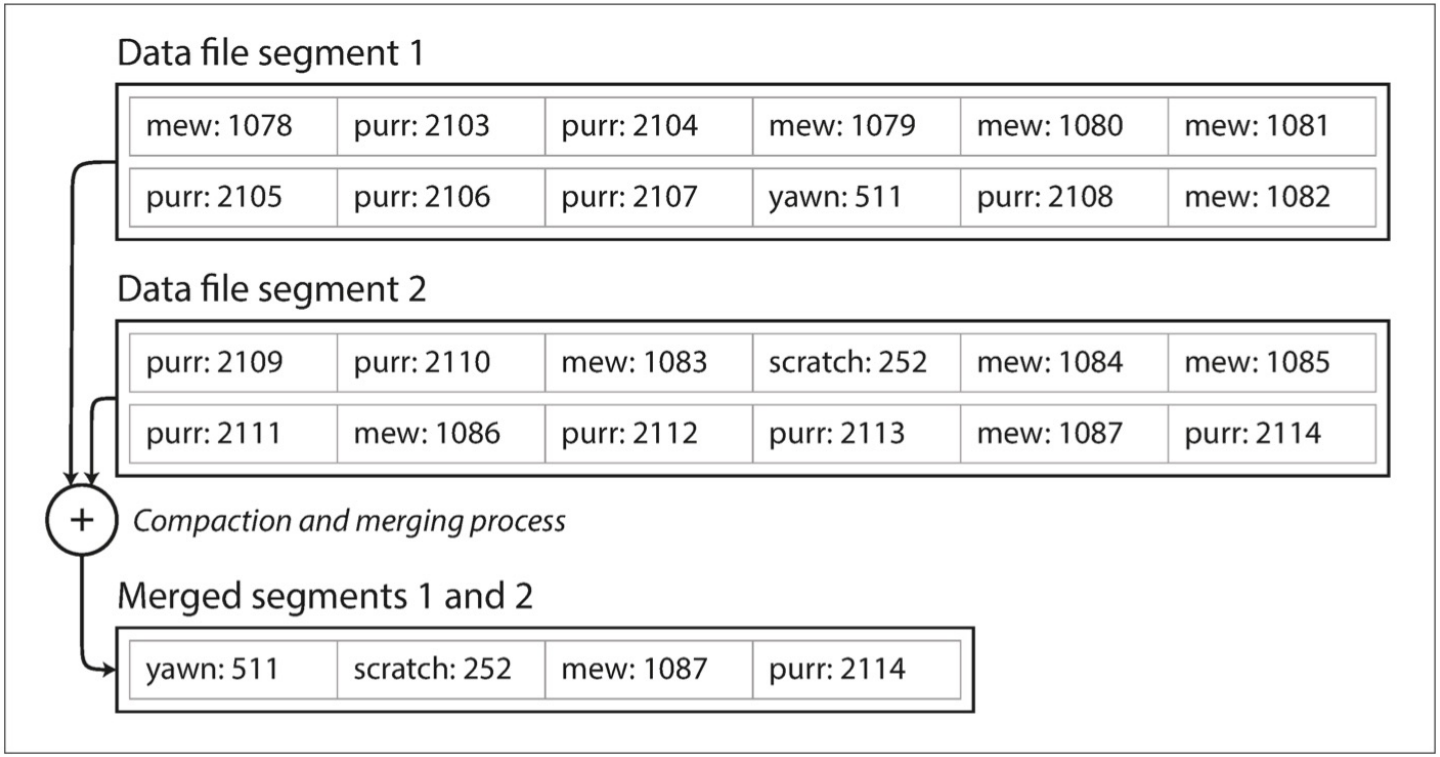
Block compact operation

- Eliminate duplicated keys and the end of log file
- Keep only the last key-value pair



Compact and aggregate blocks

Generate new blocks to substitute old blocks



Compact operations in levelDB

- levelDB implementation details

https://github.com/google/leveldb/blob/master/doc/impl.md

- BigTable: a distributed storage system for structured data

https://research.google/pubs/pub27898/

Search now becomes more complex

Each data file segment has its own hash map in memory

New search:

- search for key in most recent hash map: NOT HERE
- search for key in 2nd most recent hash map: NOT HERE
- ...
- **found** key in N-th hash map: **get the value**

We should have a limited number of segments to avoid extra hash read operations

Why not update the value of a key?

Advantages of append-only design:

Append and merge are **sequencial write** operations

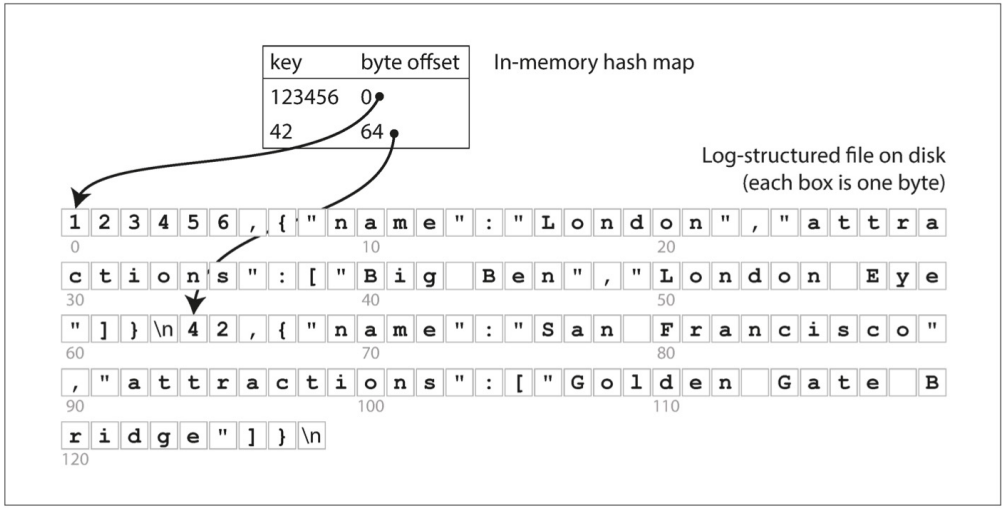
faster than random file write operations (complexity)

Error recovery and concurrency is simple to implement

But: Block merge operations have to manage fragmentation problems

Hard limitations of hash index

- Hash table has to **fit in main memory**
- Read random access pattern
- Range searches: [kitty00-kitty99]
  - no easy to solve
  - requires the individual search of each key in the range



- Other indexes?

Indexing with SST Tables

Sorted String Tables

My exam pile

Unsorted pile of exams

24 stacks of exams each of them sorted by surname initial letter:

- first stack: **A**guilar
- second stack: **B**artrina

- last stack: **Z**hu

Find exam of ”García” student

1. Unsorted list of exams: O(number of exams)
2. Go to ”G” Stack and apply binary search

Which is fastest?

Why?

German proverb

Wer Ordnung hält, ist nur zu faul zum Suchen.

If you keep things tidily ordered, you’re just too lazy to go searching.

Sorted String Tables

New requirements:

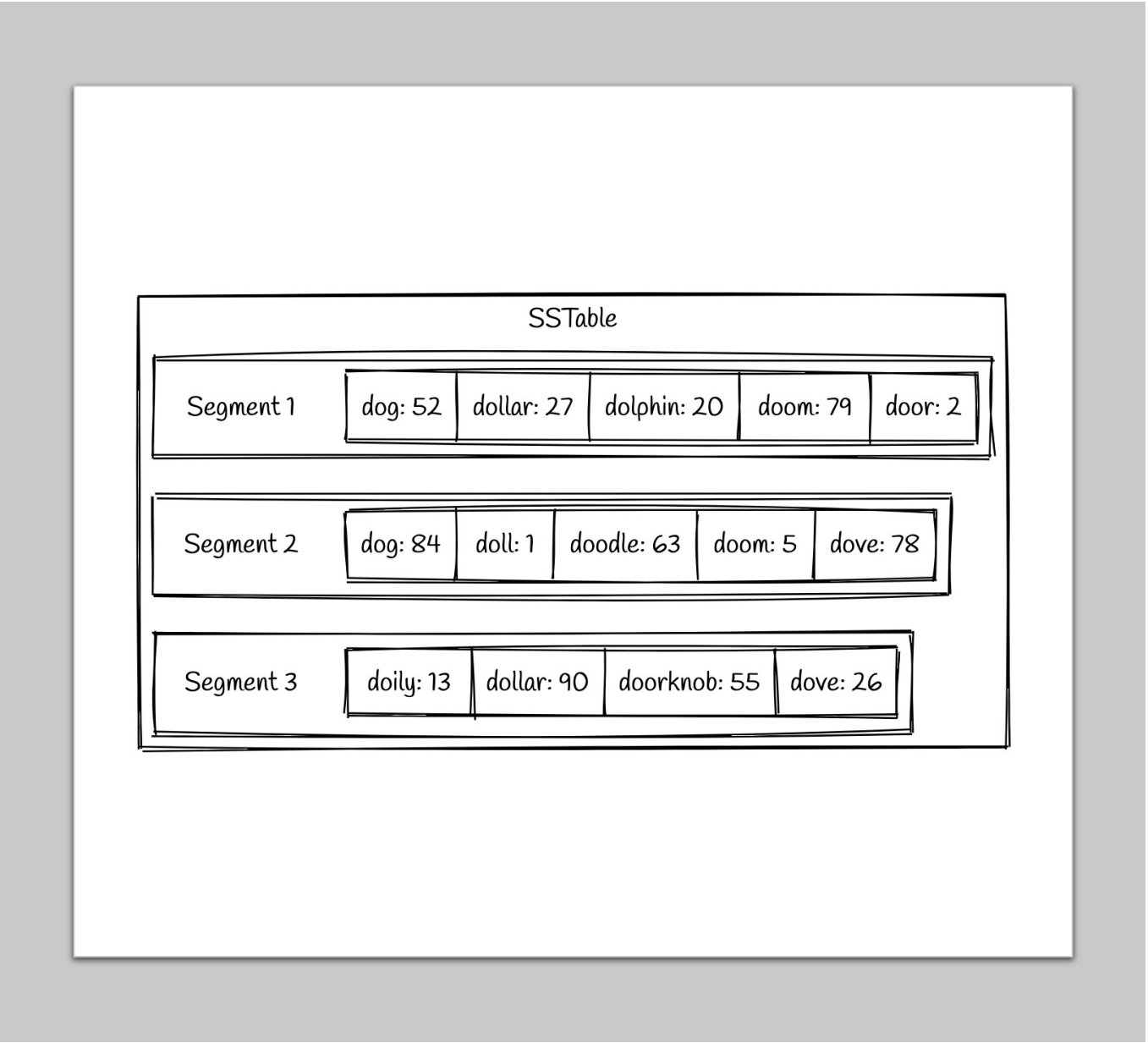
- Each data block keeps key/value pairs **sorted by key**
- Each key has **only one instance** at each data block

- Data blocks are now **Sorted String Tables (SSTs)**
- Now we need a new kind of compact operation: **merge and sort**
- merge and sort complexity: O(n * log n)

How can we read data from SSTs?

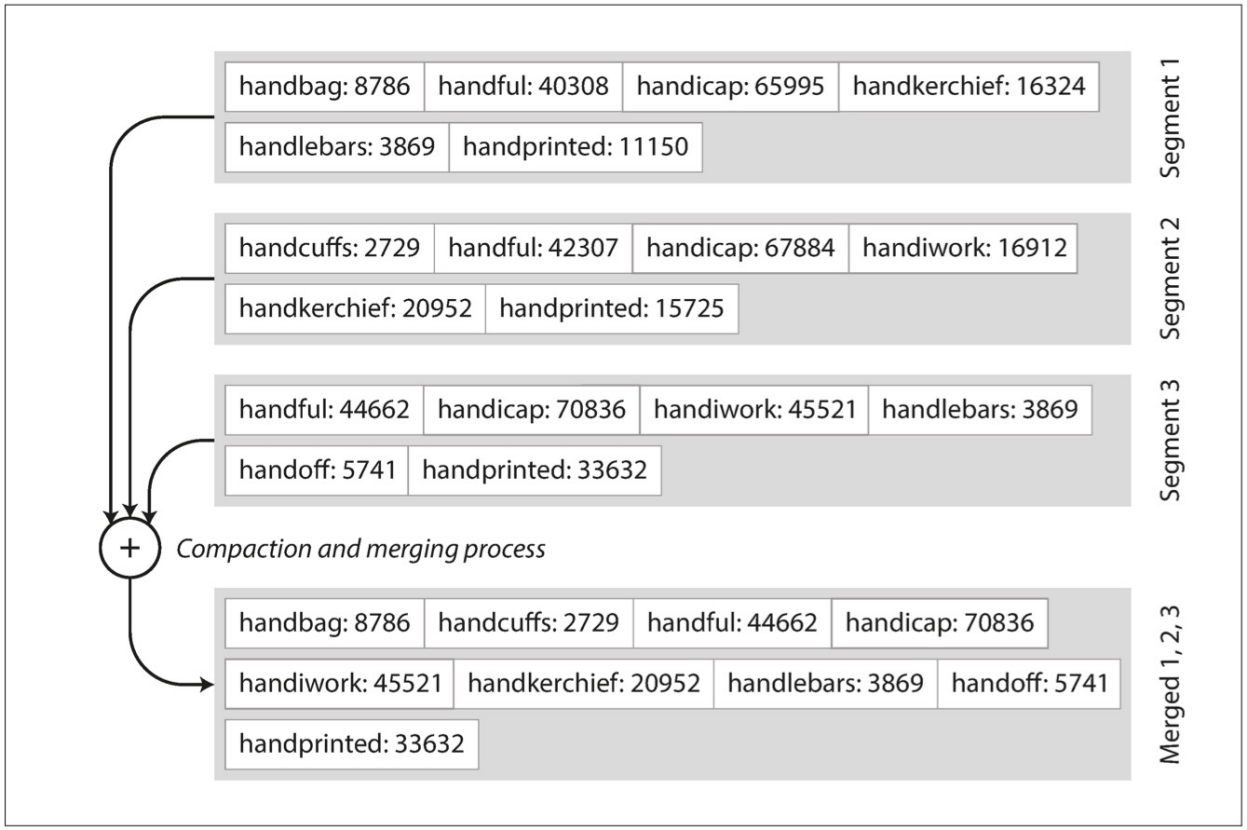
Simple approach:

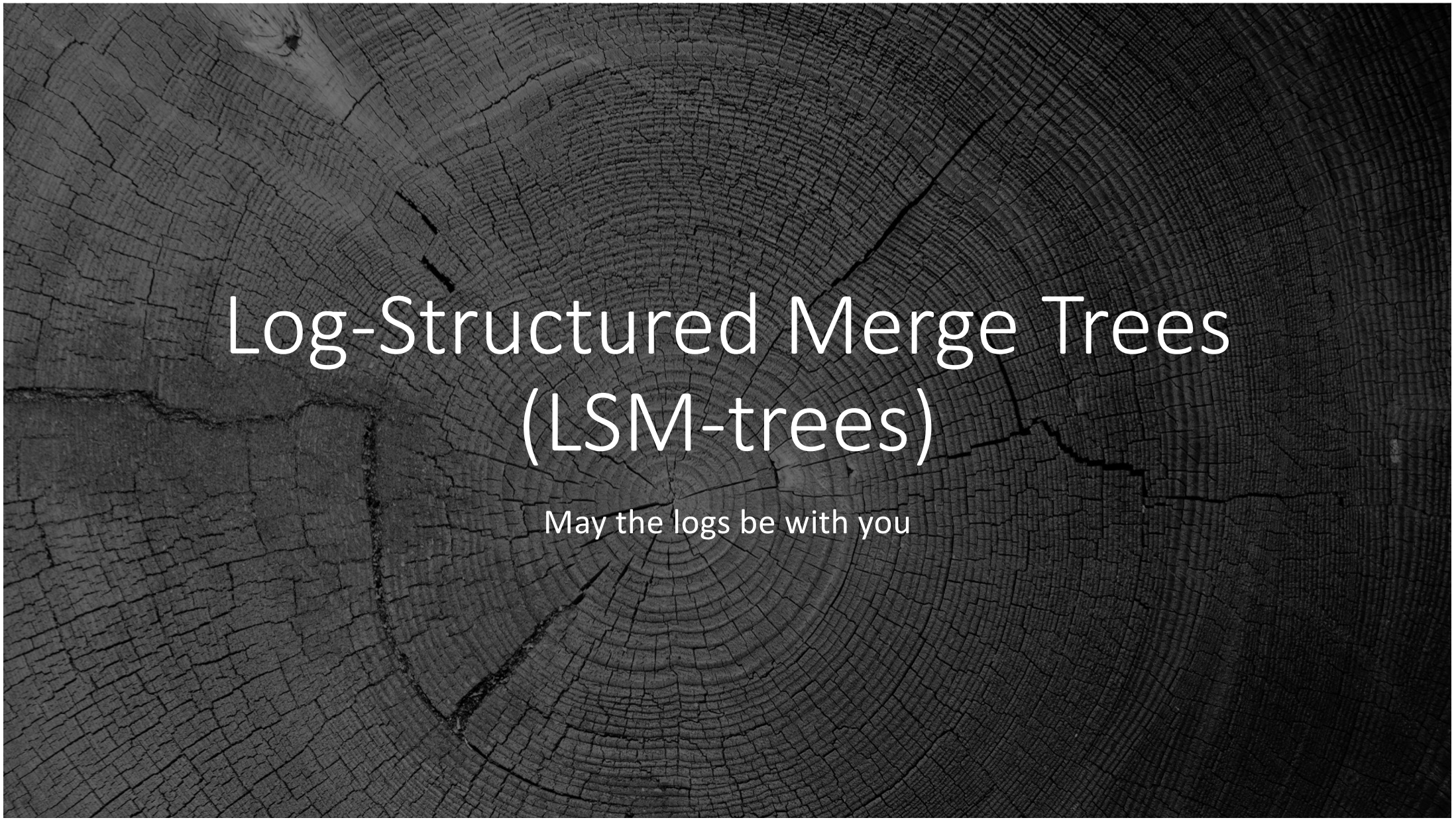
- Start reading most recent SSTable to find a key
- Go to older segments until we find the key
- No hash needed, but slow



merge-sort algorithm

Copy the smallest key to the output file and repeat

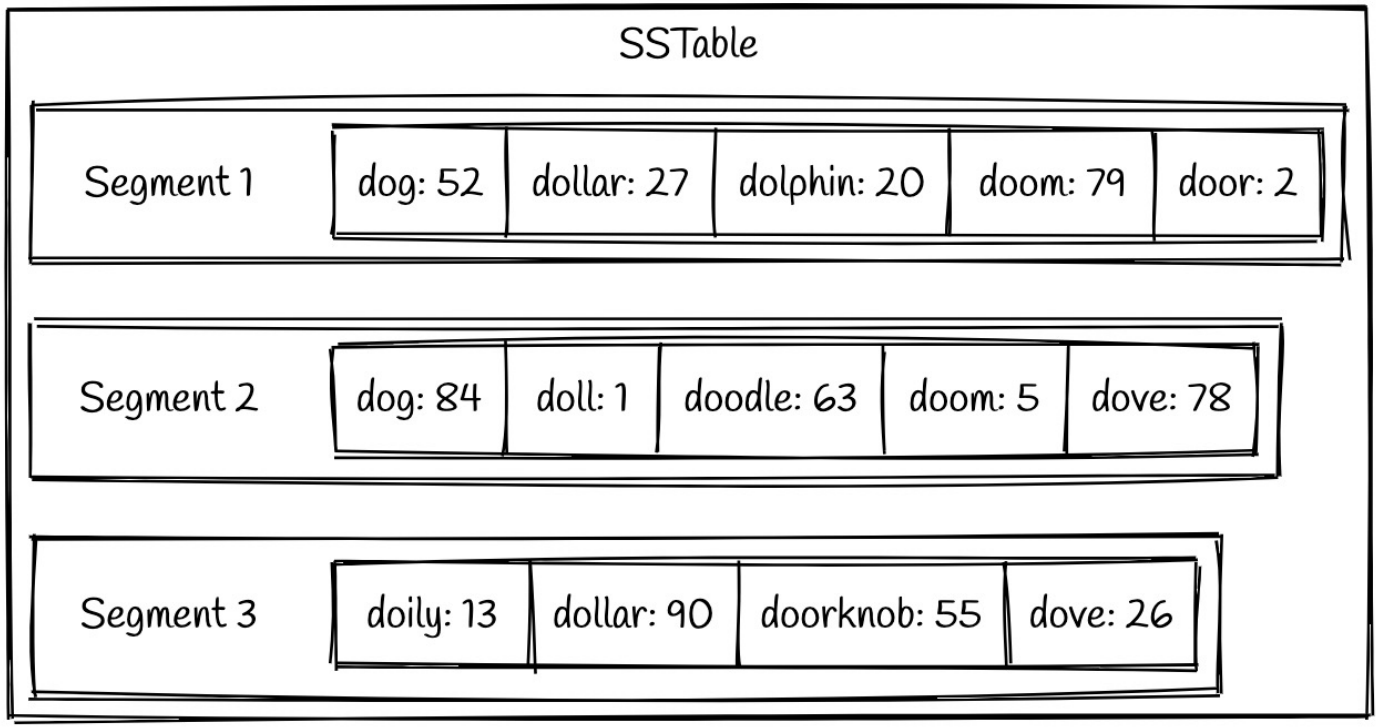



Log-Structured Merge Trees
(LSM-trees)
May the logs be with you

LSM-Tree

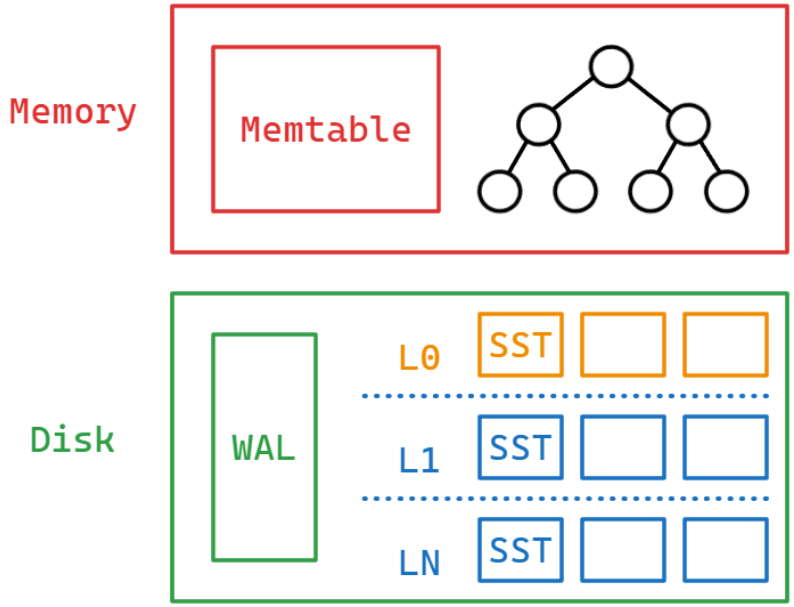
- For heavy-write workloads
- Only sequential write operations allowed
- Behind many commercial databases:
  - Google BigTable
  - AWS Cassandra
  - Scylla
  - RocksDB

Data in disk: SSTables



New structure: memtable

- in-memory sorted key/value structure
- when full: will become newest SST (Level 0)



LSM-tree Write operation

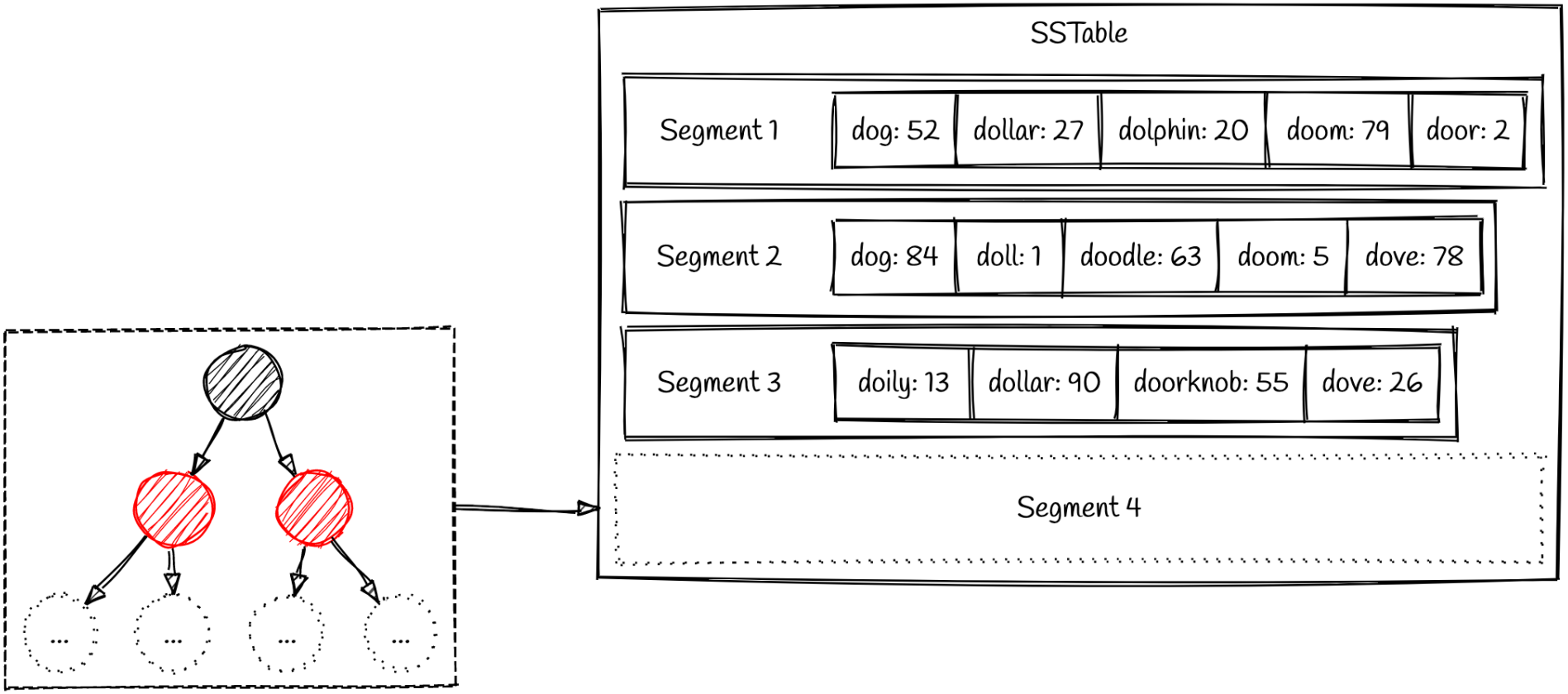
Write new value:
(doily, 42)

- add new key-value pair only to memtable

When memtable becomes larger than a limit size (few MBytes)

- Write memtable to disk becoming a SSTable: ***flush*** operation
- new SSTable is most recent DB block

Memtable flush



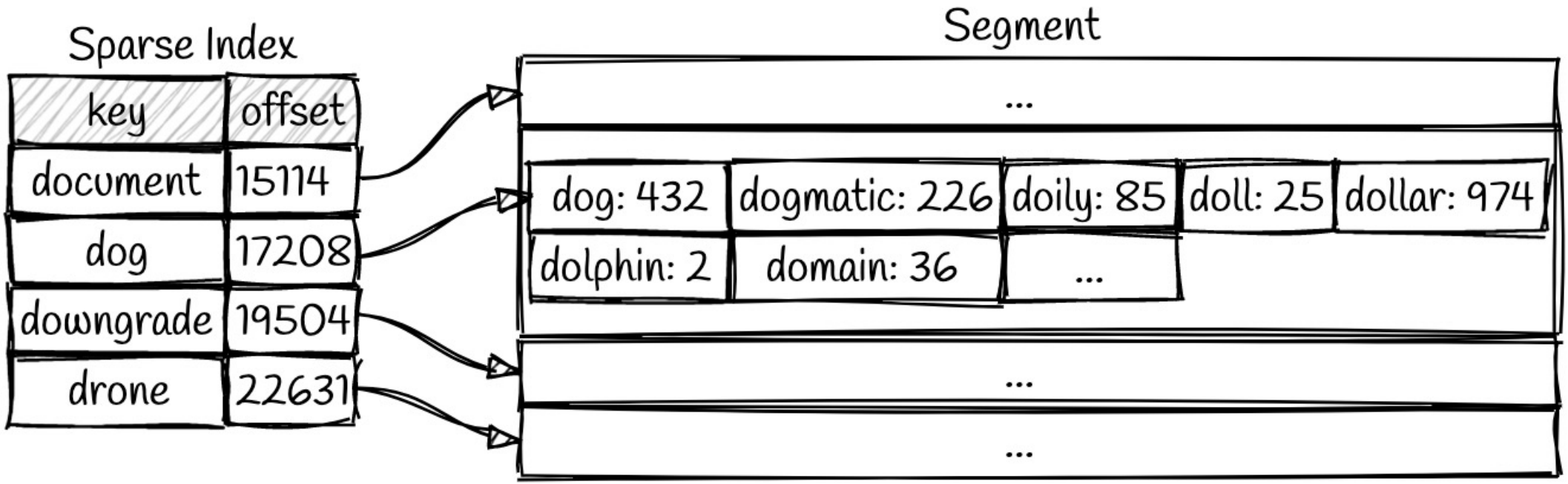
How can we read disk data faster?

New approach: use an in-memory **sparse index**

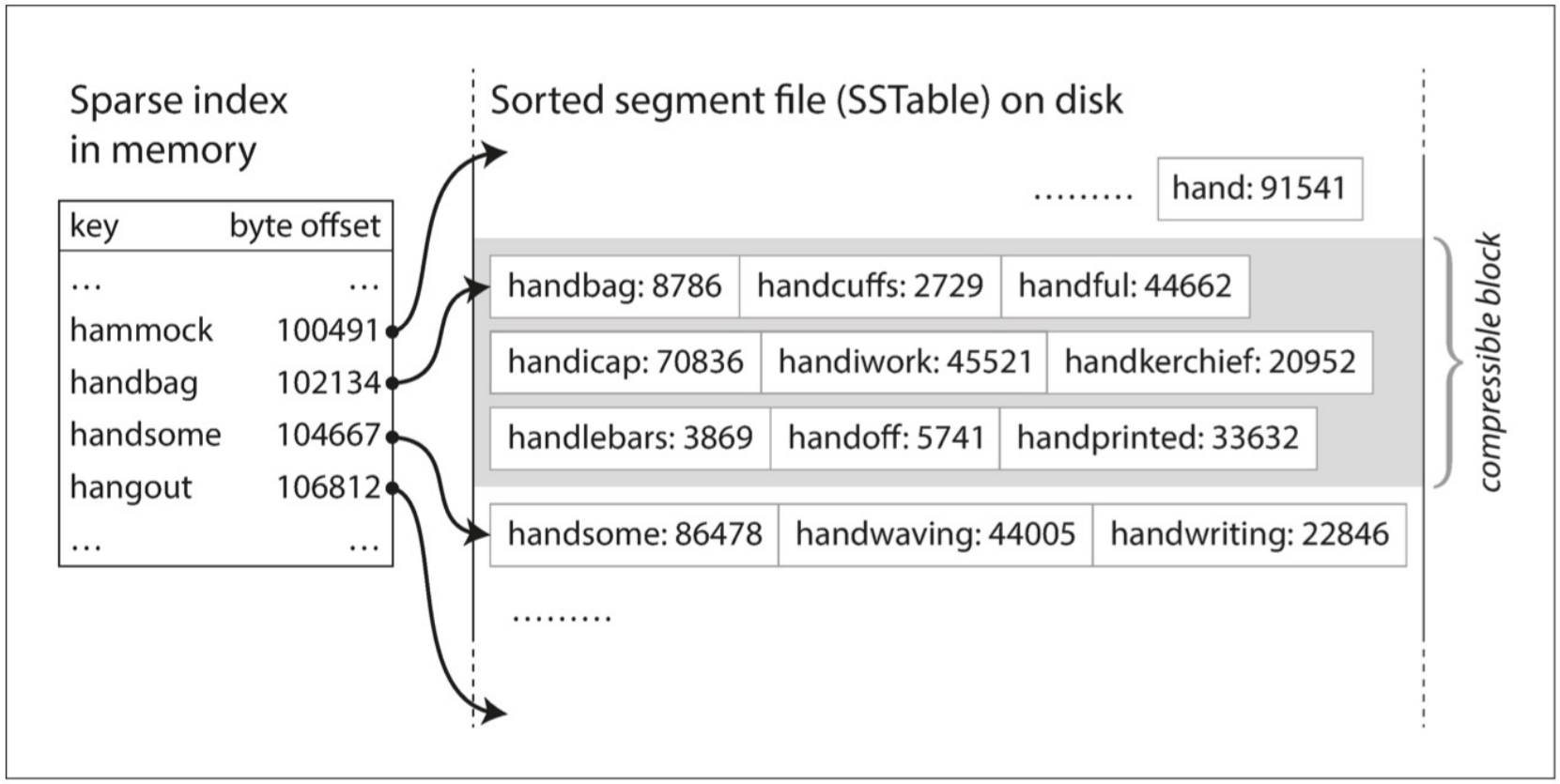
SSTs are globally sorted by key

Index contains the position of the first key in each segment

Sparse index to accelerate reads



Key search operation



LSM-tree read operation

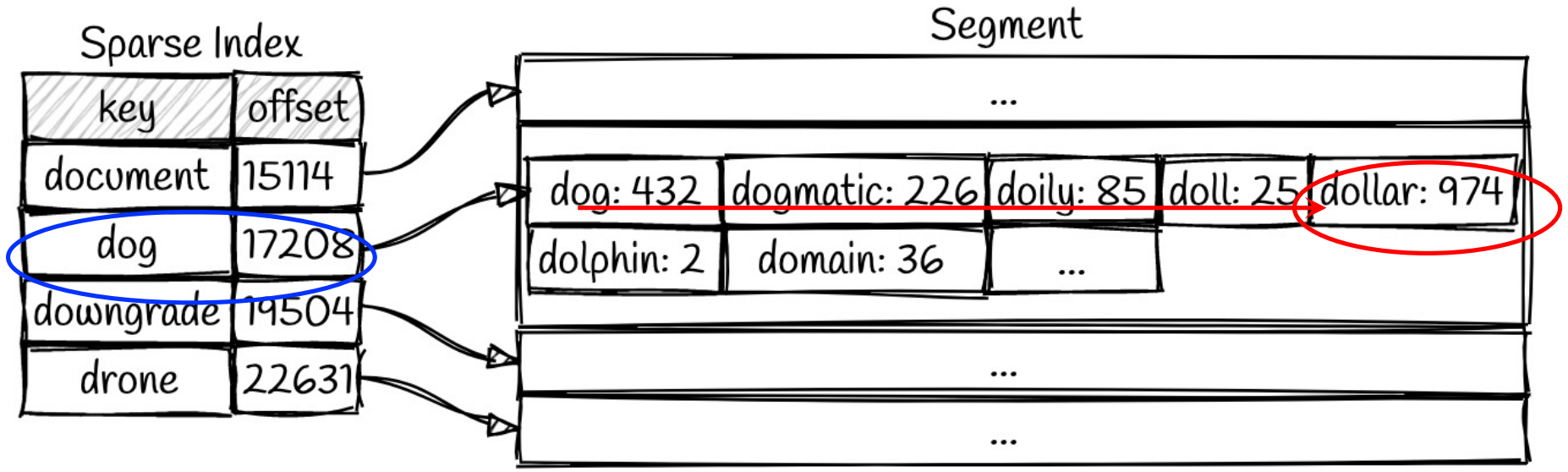
key **is not** in memtable

1- search for the key in sparse in-memory index: find a SSTable block for my key

2- scan a sorted segment file to find the key

Example: find ”dollar” (not in memtable)

1. Search in sparse index to find that dollar comes between dog and downgrade
2. Scan from offset 17208 to 19504 to find the value



All blocks must be sorted by key

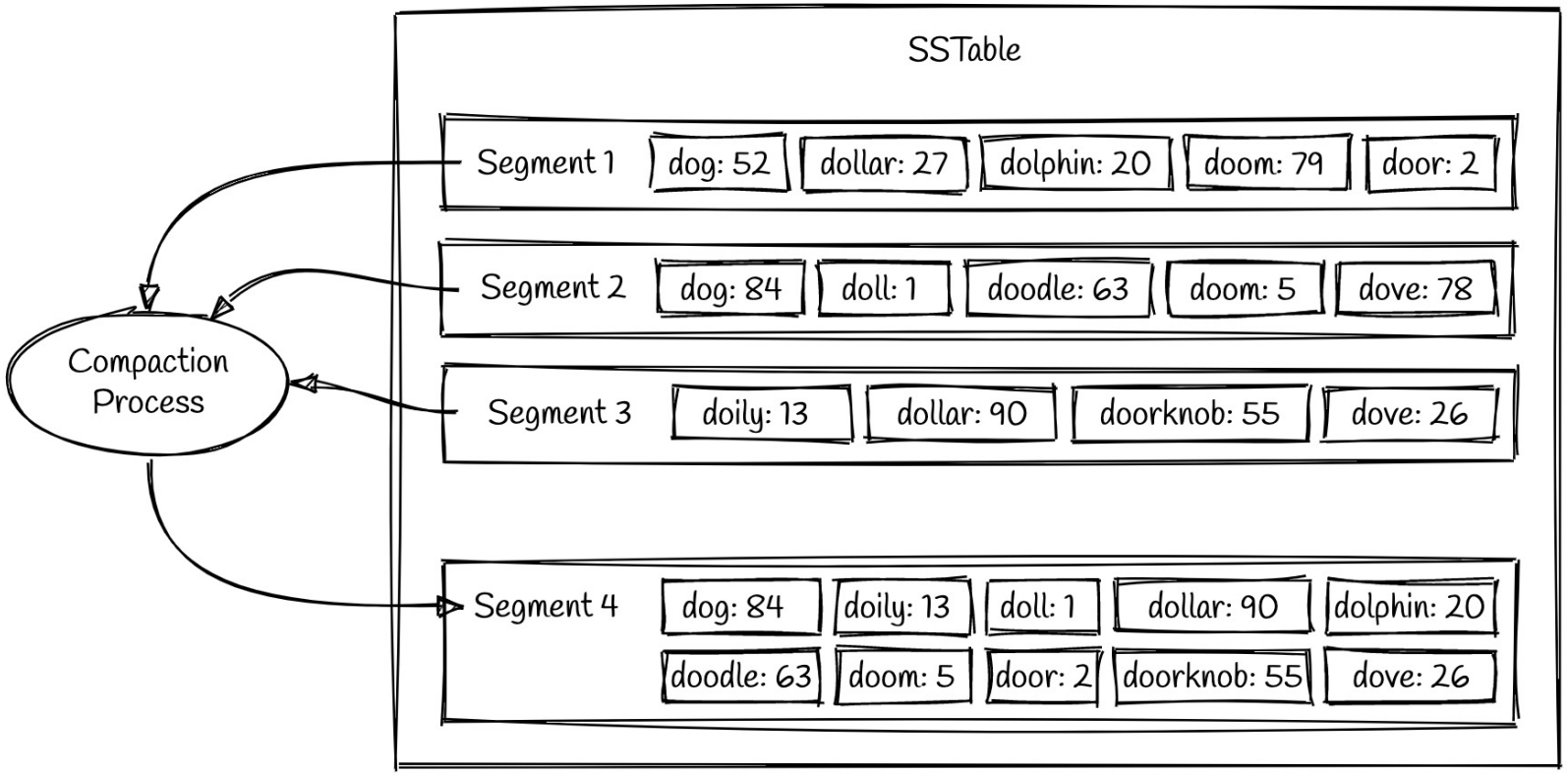
Fast access depends on blocks sorted

Problem: we need a new SST sort and merge operation

Must be global and fast!

Block compact operation

Merge process to discard old values and minimize SSTable files



LSM-tree summary

- Memtable for most recent writes
- Keep set of SSTable blocks in files
- Background SSTables merge operation
- Sparse index for read operations

- Works well when dataset is larger than memory
- Good performance for range queries (cat0000-cat9999)
- As write operations are sequential: good write throughtput

Ideal read case: few write operations

- Memtable stores the last database updates/inserts
- Sparse index
- Globally sorted SSTs

Write operation burst to memtable

- compact does not cope with block merge requests
- two kinds of SST: compacted and uncompacted
- reads become slower as we have more uncompacted blocks
- we may run out of disk space

LSM-trees limitations

Compact operation affects read and write performance

- reads must wait for a compact operation to finish

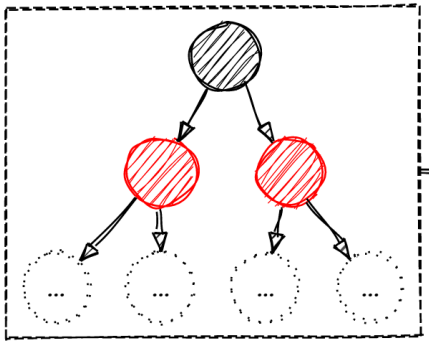
Need to share disk write capacity with three operations

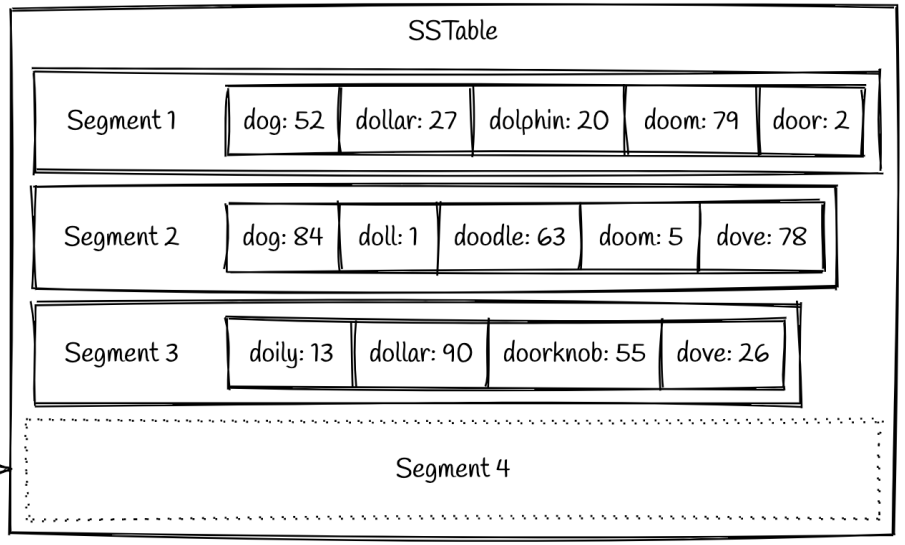
- logging
- copy memtable to a new SST file
- background compact

Write operation burst to memtable

LSM-trees main problem:

System fails: last writes in memtable can be lost





Solutions for system failure

- Keep a log file in disk with same info than memtable
- No sorting applied
- When system fails:
  - Rebuild memtable from log file (slow but safe)
  - Must add all keys and values
  - Expected to be used only in emergencies (not often)

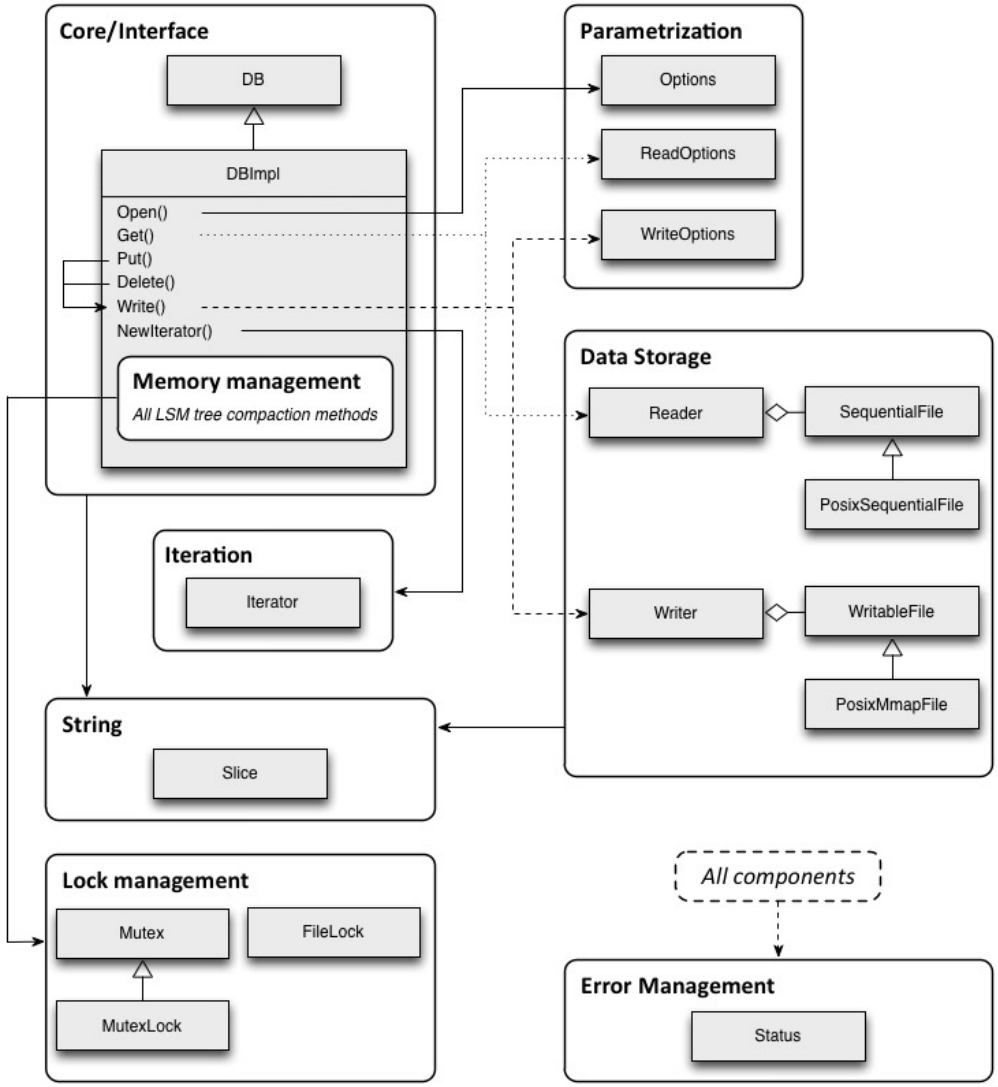
Key-Value Store architecture cases

- KingDB: log-structured storage
- LevelDB: LSM-Trees

How to implement your own key-value store:

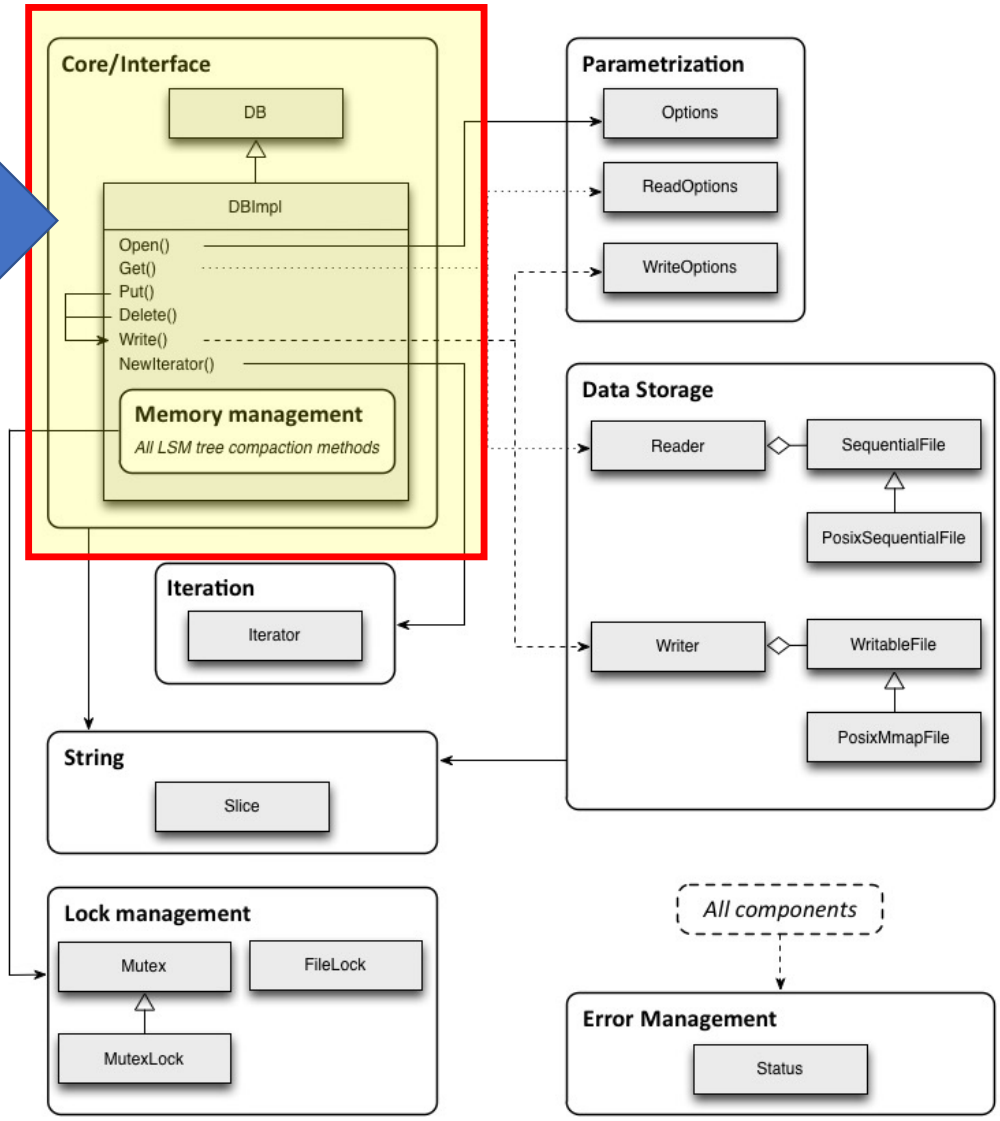
https://codecapsule.com/2012/11/07/ikvs-implementing-a-key-value-store-table-of-contents/

Architecture of LevelDB v1.7.0

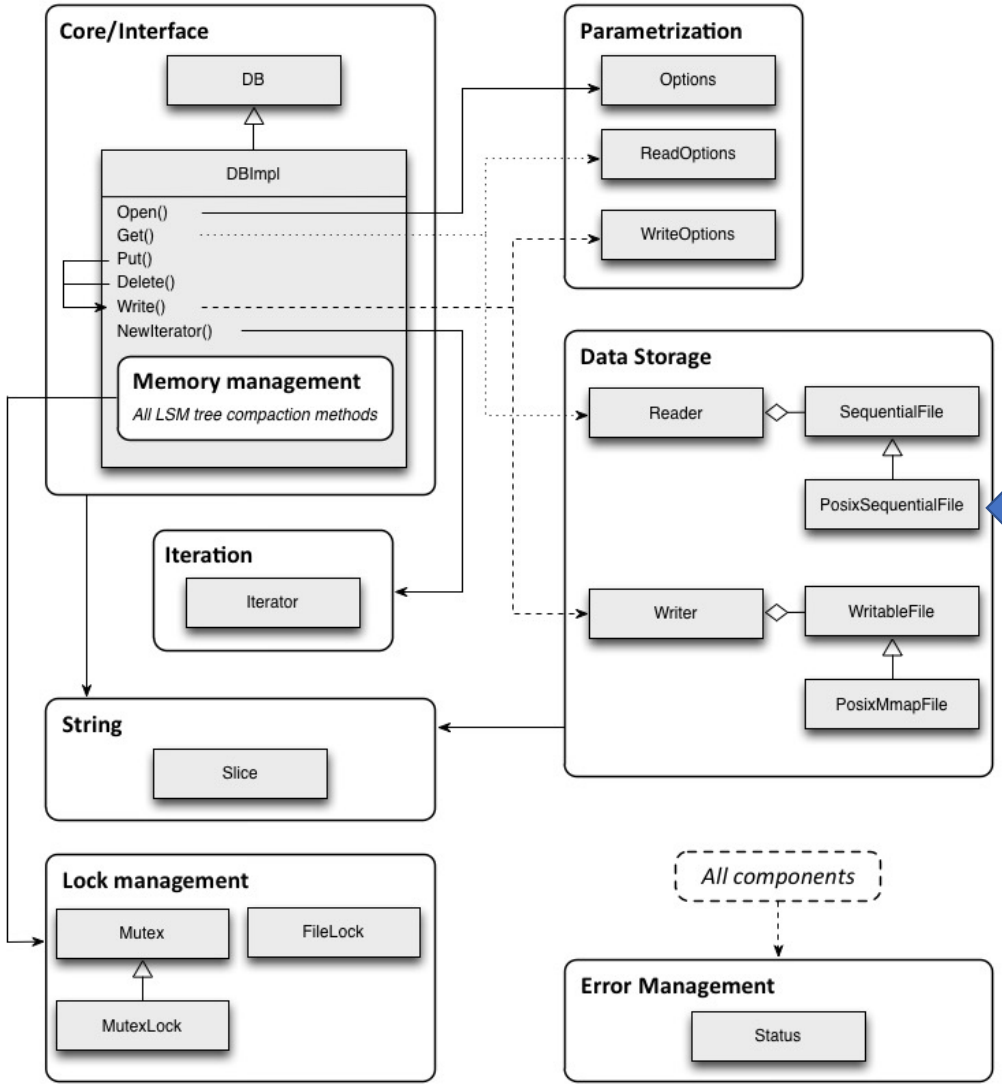


Architecture of LevelDB v1.7.0

Interface:
methods to use DB
-get()
-put()



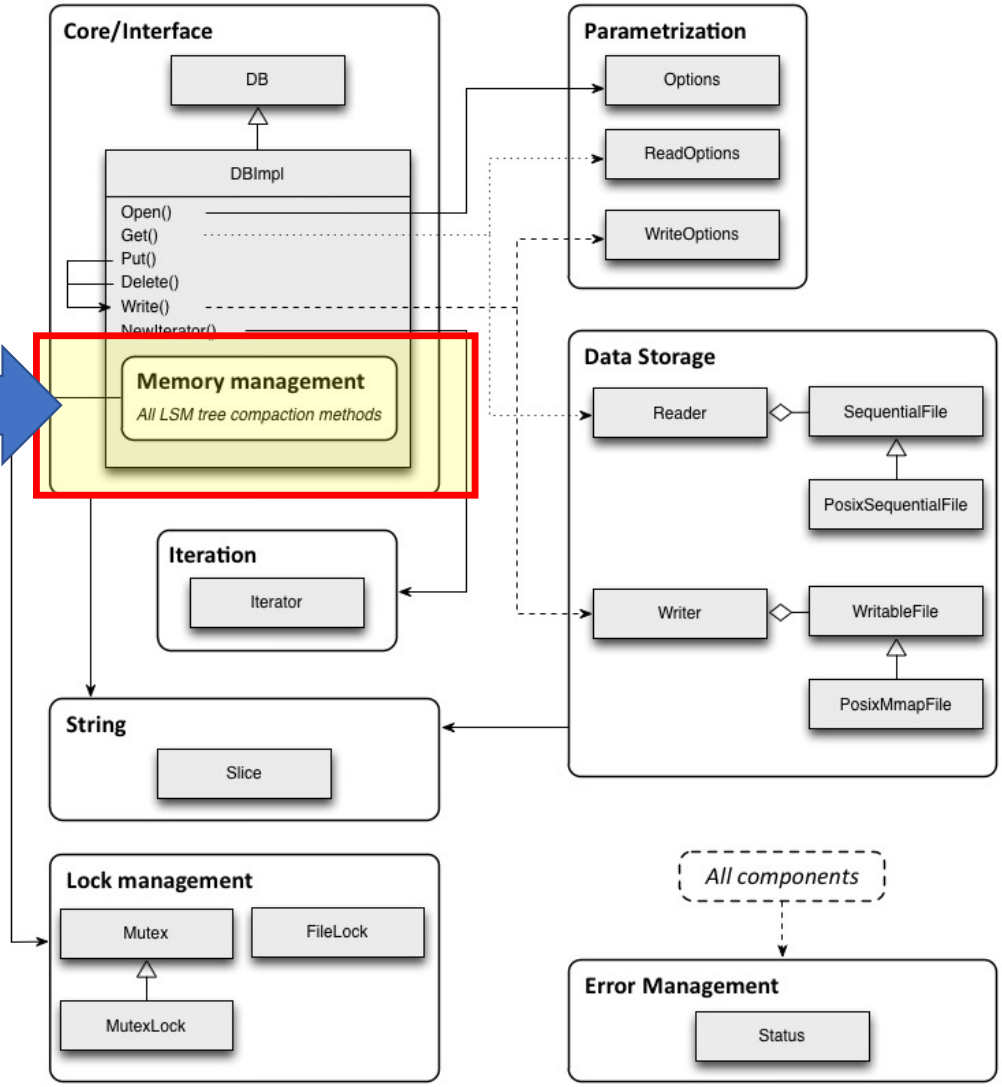
Architecture of LevelDB v1.7.0



Data Storage
interface to access memory for keys and values

Architecture of LevelDB v1.7.0

LSM-Tree:
compacting log files when limit size is exceeded



Architecture of KingDB v0.9.0