

Storage and Retrieval

Тимофей (@timofey256)

26.03.2026

What the chapter is about?

- **LSM-tree**
- B-tree
- **Column-oriented DB internals**
- Multidimensional and Full-text indexes

Core trade-off

Reads want structure, Writes want to avoid reorganisation

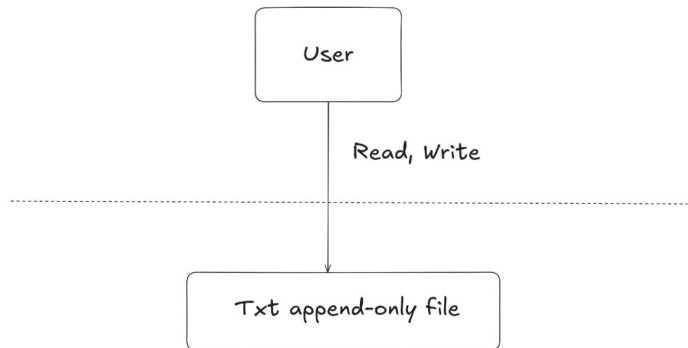
World's simplest database

```
#!/bin/bash

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

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

$O(1)$ writes, $O(n)$ reads



I want faster reads...

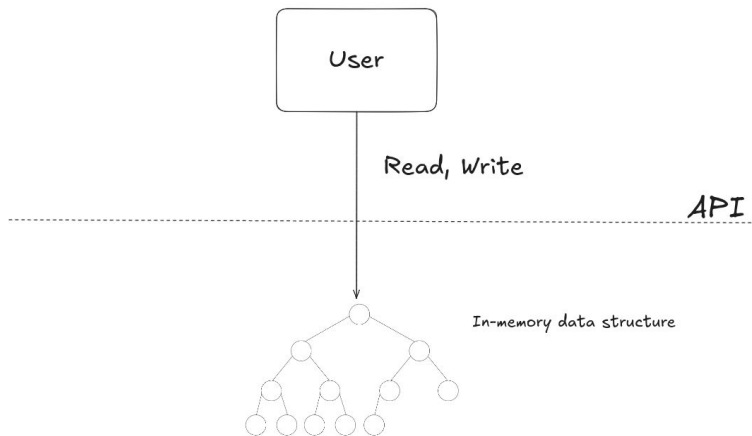
I want faster reads

Solution: keep in-memory DS.

Implemented as RB-tree, AVL-tree, trie, skip-list, ...

Still in-memory DB.

$O(\log n)$ writes, $O(\log n)$ reads.

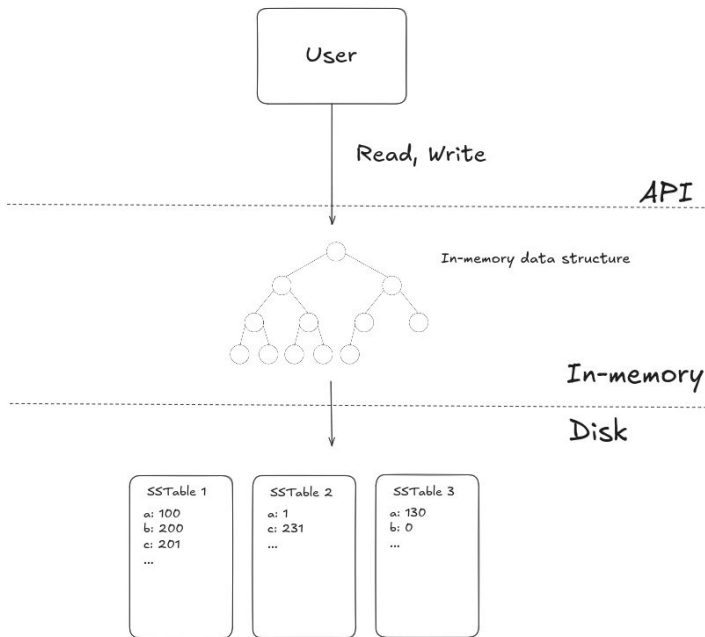


I don't have enough RAM...

I don't have enough RAM (or I want durability)

Solution: keep in-memory Memtable. When it grows past a certain threshold (~few MBs), flush as SSTable on disk.

$O(\log n)$ writes, $O(\log n)$ reads for in-memory, $O(?)$ reads from disk



How to search in SSTable?...

How to search in SSTable?

Solution: add sparse index to SSTables

$O(\log n)$ writes, $O(\log n)$ reads.

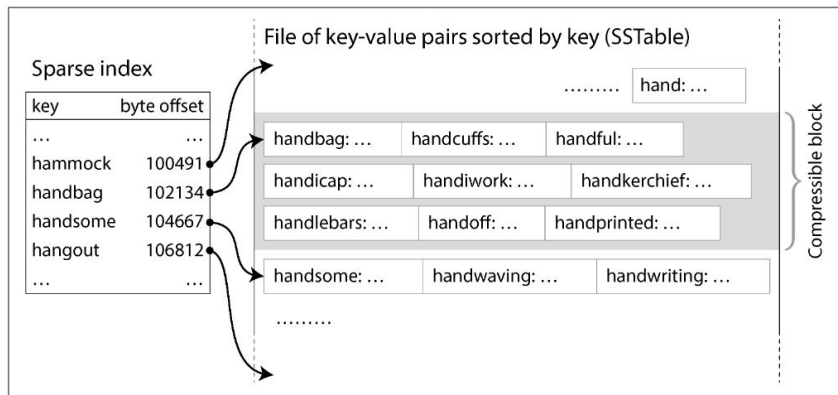
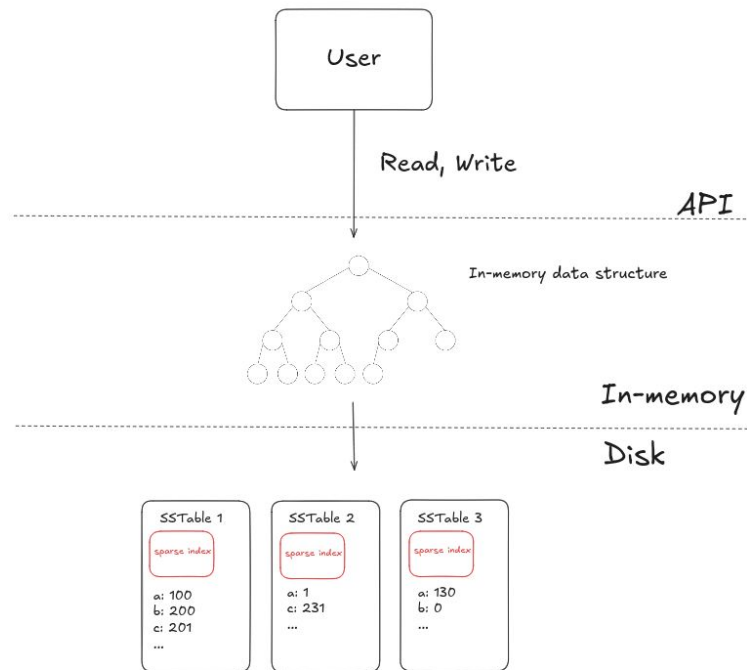


Figure 4-2. An SSTable with a sparse index, allowing queries to jump to the right block



How to search faster?...

How to search faster?

Solution: add a Bloom filter into SSTable

Skip most disk I/O.

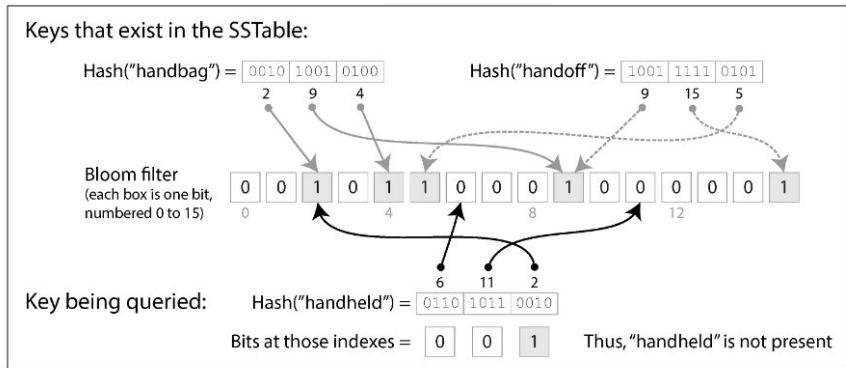
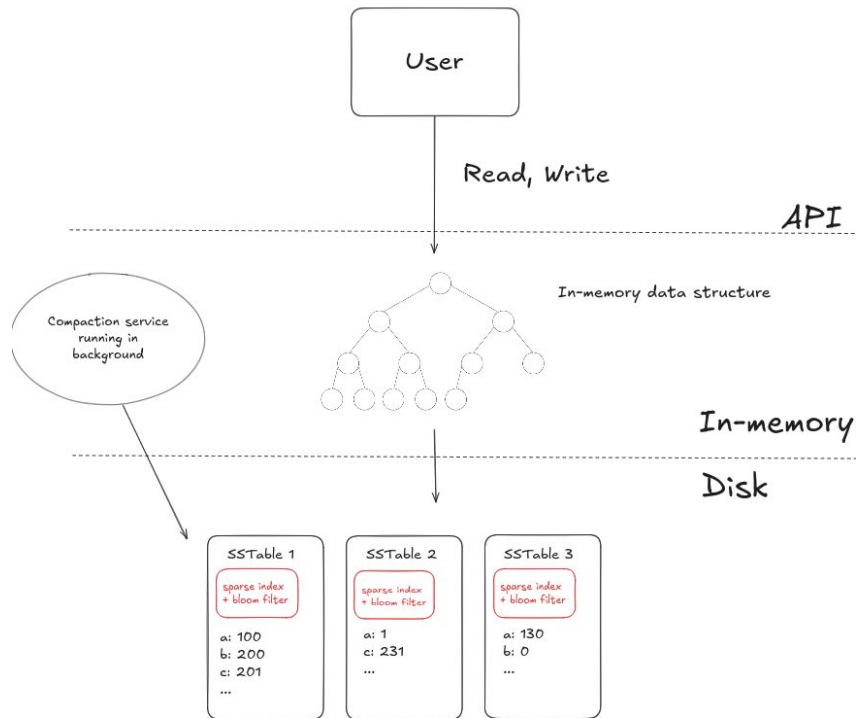
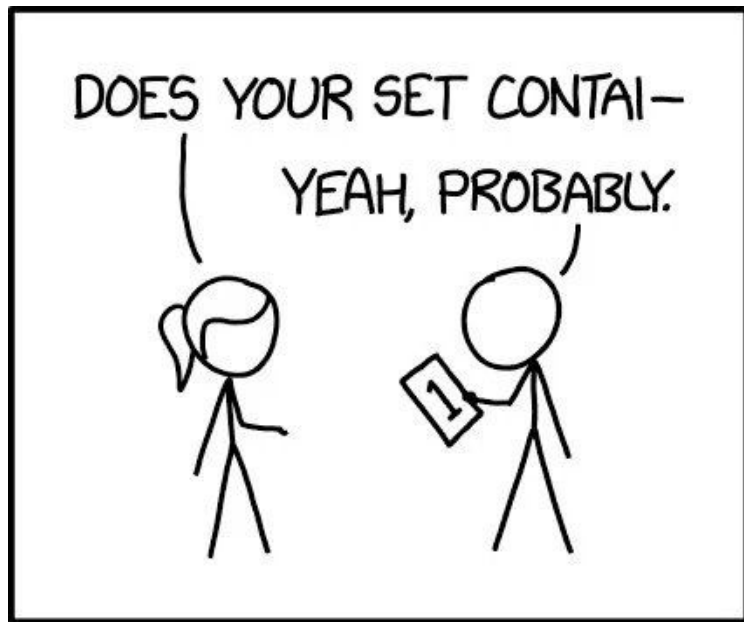


Figure 4-4. A Bloom filter provides a fast, probabilistic check on whether a particular key exists in a particular SSTable.



How to search faster?



ONE-BIT BLOOM FILTER

“As a rule of thumb, you need to allocate **10 bits** of Bloom filter space for every key in the SSTable **to get a false-positive probability of 1%**, and the probability is reduced tenfold for every 5 additional bits you allocate per key.”

I don't have enough disk space...

I don't have enough disk space

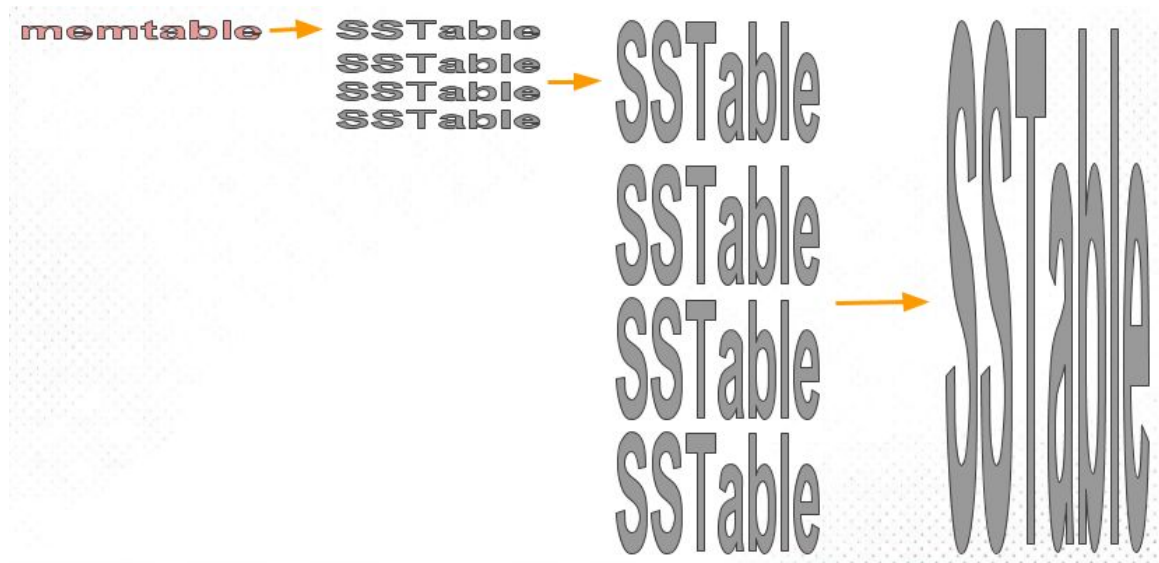
Solution: compact SSTables using MergeSort

Size-tiered compaction =>

Good: fast writes

Bad: slow reads, space amplification

Used in ScyllaDB, Cassandra [4], ...



I don't have enough disk space (+ I don't want to read too much.)

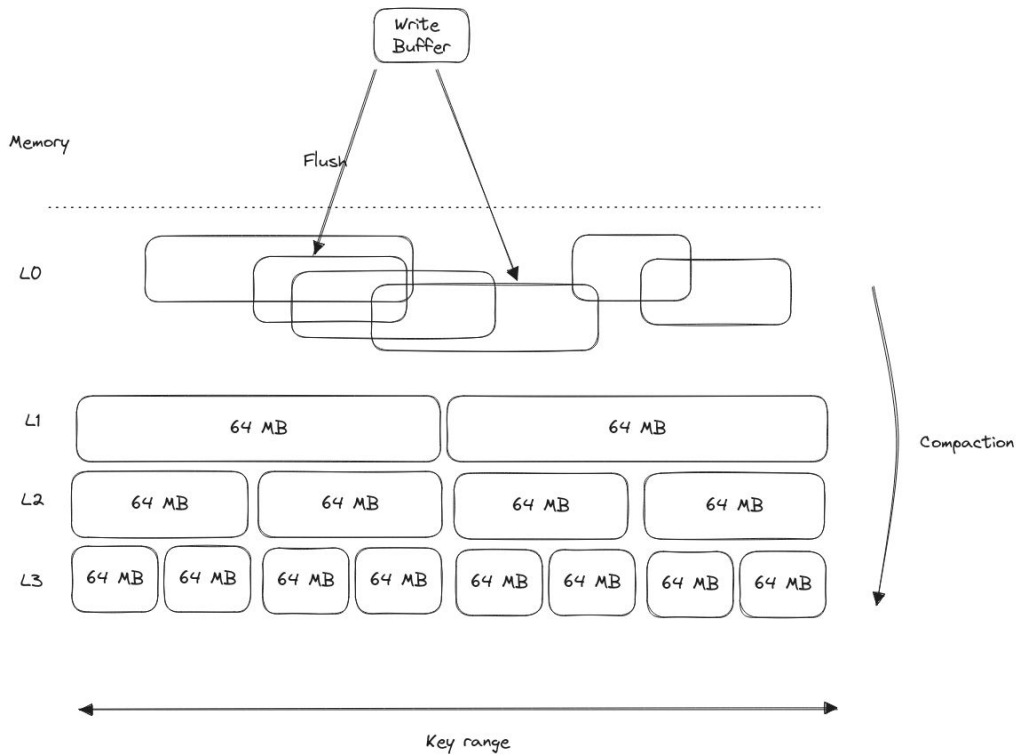
Solution: leveled compaction

The worst-case disk I/O ops is
 $L0_segments + (level_count - 1)$.

Good: fast reads

Bad: slower writes, write-amplification

Used in LevelDB, RocksDB.



Remember, it's all about trade-offs [5]

	Write amplification	Files checked per read
Size-tiering	$O(L)$	$O(T \times L)$
Leveling	$O(T \times L)$	$O(L)$

T = number of segments in a level

L = number of levels

Remember, it's all about trade-offs [5]

Parameters

K : bound on runs at Levels 1 to $L - 1$

Z : bound on runs at Level L

T : size ratio

Systems

○ Dostoevsky

△ Monkey

◇ Well-Tuned RocksDB

□ Default RocksDB

Existing Design Space

† Tiering ($T = 10$)

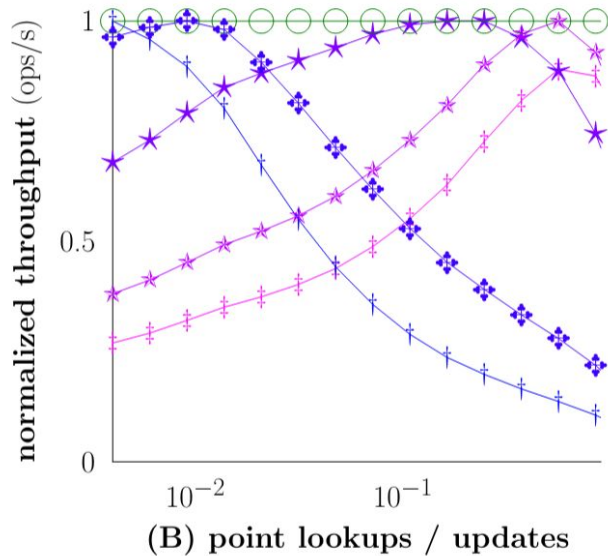
‡ Leveling ($T = 10$)

Fluid LSM-Tree New Design Space

✦ Transition 1 ($T = 10, Z = 4, K = 9$)

★ Lazy Leveling ($T = 10$)

☆ Transition 2 ($T = 10, Z = 1, K = 4$)



Remember, it's all about trade-offs [5]

Parameters

K : bound on runs at Levels 1 to $L - 1$

Z : bound on runs at Level L

T : size ratio

Systems

○ Dostoevsky

△ Monkey

◇ Well-Tuned RocksDB

□ Default RocksDB

Existing Design Space

† Tiering ($T = 10$)

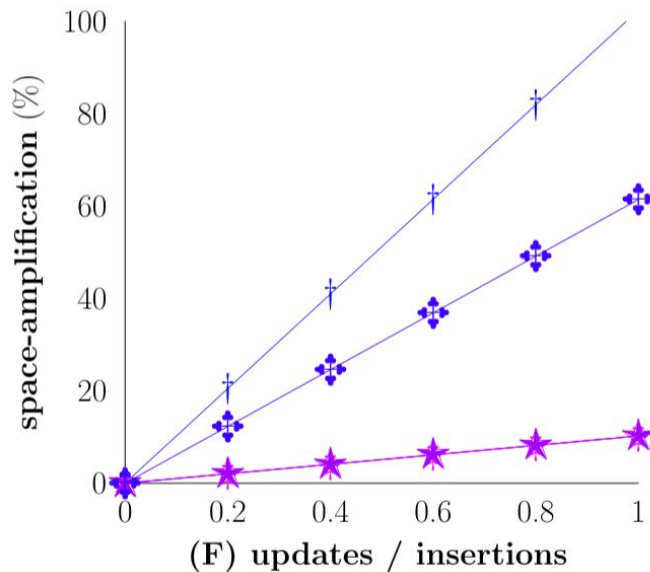
‡ Leveling ($T = 10$)

Fluid LSM-Tree New Design Space

✦ Transition 1 ($T = 10, Z = 4, K = 9$)

★ Lazy Leveling ($T = 10$)

☆ Transition 2 ($T = 10, Z = 1, K = 4$)



Remember, it's all about trade-offs [5]

Parameters

K : bound on runs at Levels 1 to $L - 1$

Z : bound on runs at Level L

T : size ratio

Systems

○ Dostoevsky

△ Monkey

◇ Well-Tuned RocksDB

□ Default RocksDB

Existing Design Space

† Tiering ($T = 10$)

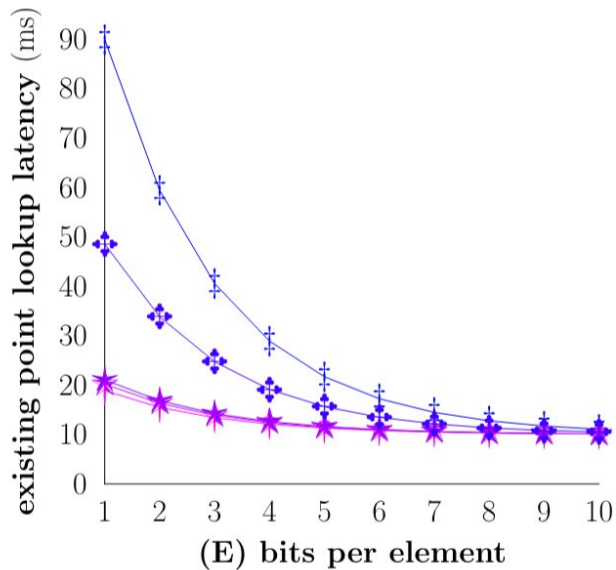
‡ Leveling ($T = 10$)

Fluid LSM-Tree New Design Space

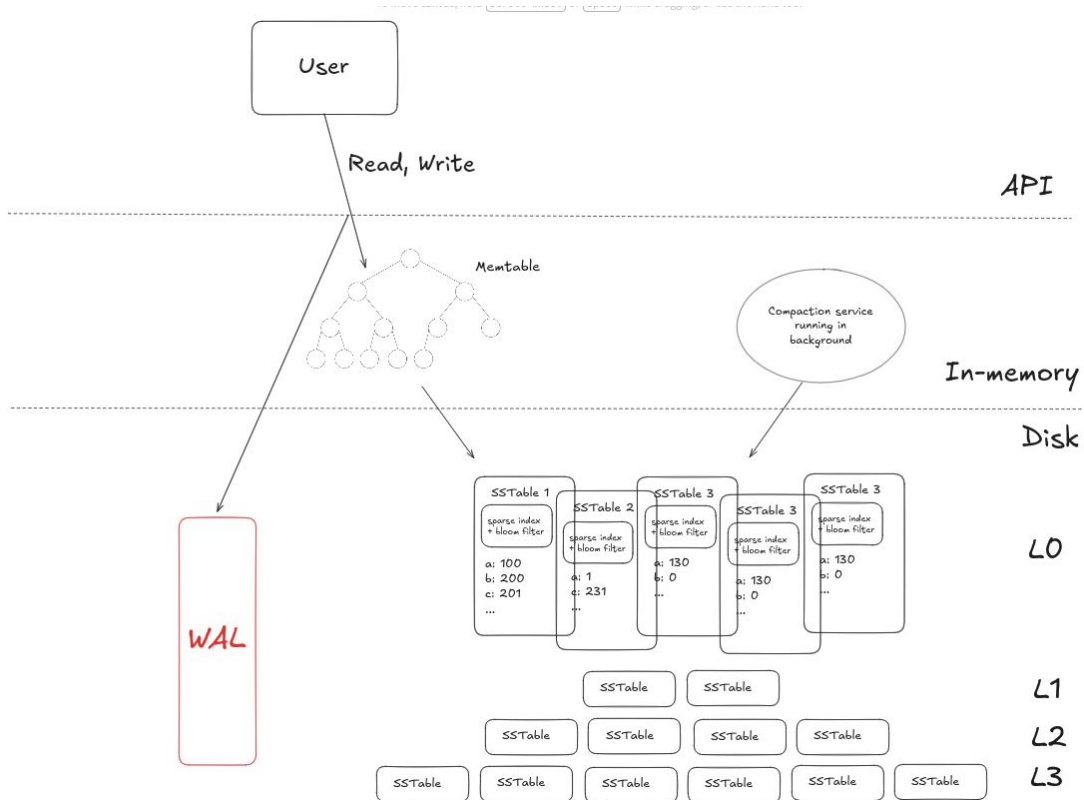
✦ Transition 1 ($T = 10, Z = 4, K = 9$)

★ Lazy Leveling ($T = 10$)

☆ Transition 2 ($T = 10, Z = 1, K = 4$)



I also want reliability



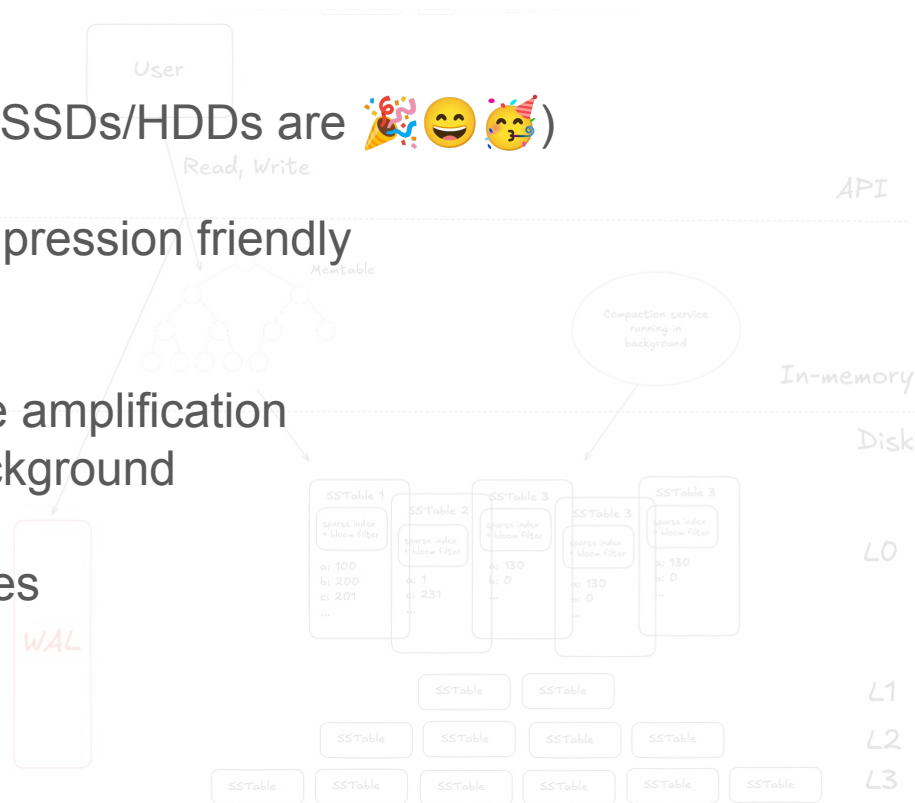
Recap: LSM-tree

Good:

- Sequential writes (SSDs/HDDs are 🎉😄🥳)
- Write throughput
- Immutable => compression friendly

Bad:

- Read-/write-/space amplification
- Compaction in background
- Range queries.
- (kinda) point queries



OLAP

OLAP

Nature of the workload:

- Read-heavy.
- Mostly range queries. Mostly aggregates.
- Few(er) but big(ger) requests compared to OLTP.
- Wide schema but few columns are accessed in requests.
- No need for transactions.

OLAP

Nature of the workload:

- Read-heavy.
- Mostly range queries. Mostly aggregates.
- Few(er) but big(ger) requests compared to OLTP.
- Wide schema but few columns are accessed in requests.
- No need for transactions.

=> LSM-tree and B-tree *(as we know them)* don't fit

How it works in  ClickHouse ?

Idea 1: column-oriented storage

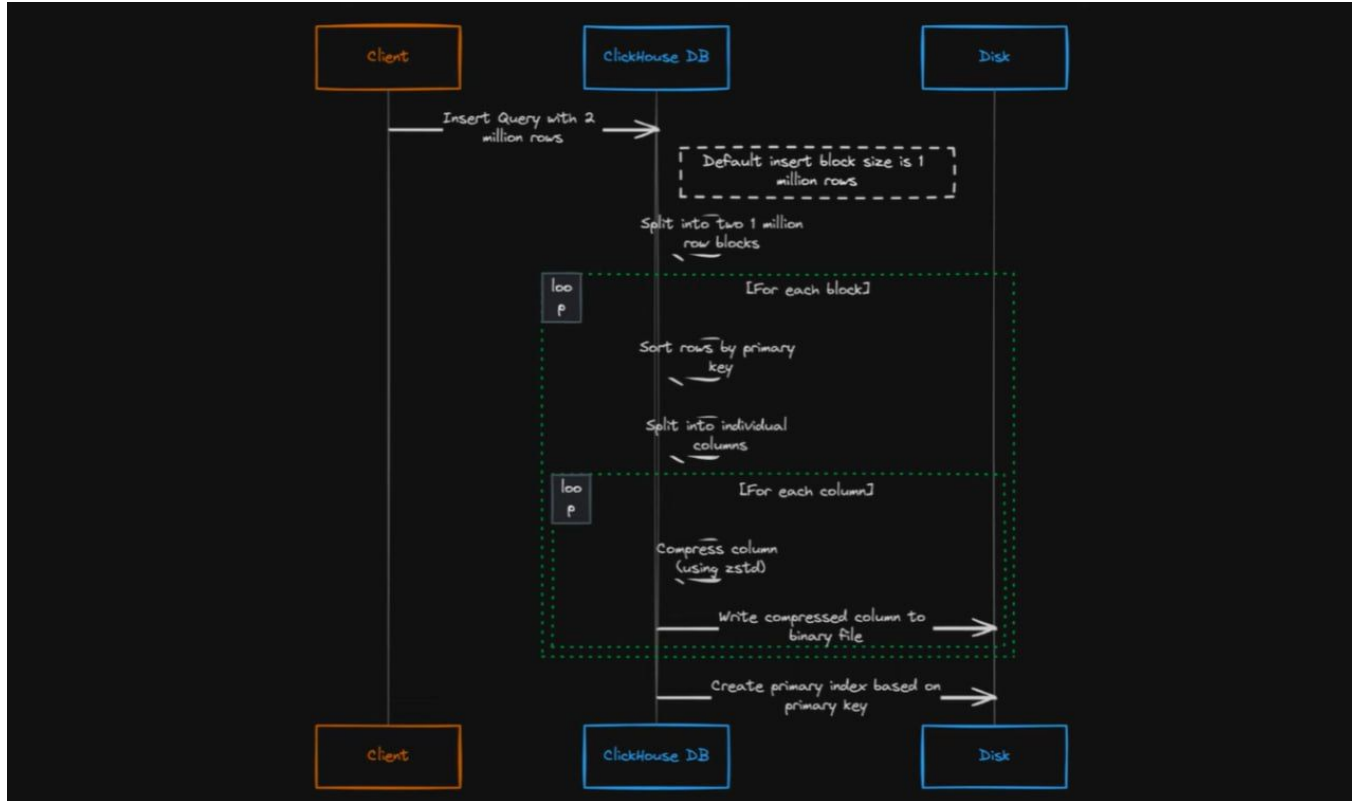
Row-based

location	ts	temp	wind_speed	humidity
Aberystwyth	2022-01-01 00:00:00	14	21	79
Blackpool	2022-01-01 00:20:00	13	9	82

Column-based

location	ts	temp	wind_speed	humidity
Aberystwyth	2022-01-01 00:00:00	14	21	79
Blackpool	2022-01-01 00:20:00	13	9	82

Idea 1: column-oriented storage w/ sparse index



Idea 1: column-oriented storage w/ sparse index

- **Primary index** is a flat uncompressed array file (primary.idx), containing index marks that are numbered starting at 0.
- **Mark file** is also a flat uncompressed array file (*.mrk) containing marks that are numbered starting at 0.

primary index

PK value -> mark ID

ID 123 -> markID 0

ID 10000 -> markID 1

ID 12345 -> markID 2

...

column1.mrk

markID -> (block_offset, granule_ofsset)

markID 0 -> (1, 2)

markID 1 -> (5, 12)

markID 2 -> (...)

column2.mrk

markID -> (block_offset, granule_ofsset)

markID 0 -> (1, 2)

markID 1 -> (5, 12)

markID 2 -> (...)

column2.mrk

markID -> (block_offset, granule_ofsset)

markID 0 -> (1, 2)

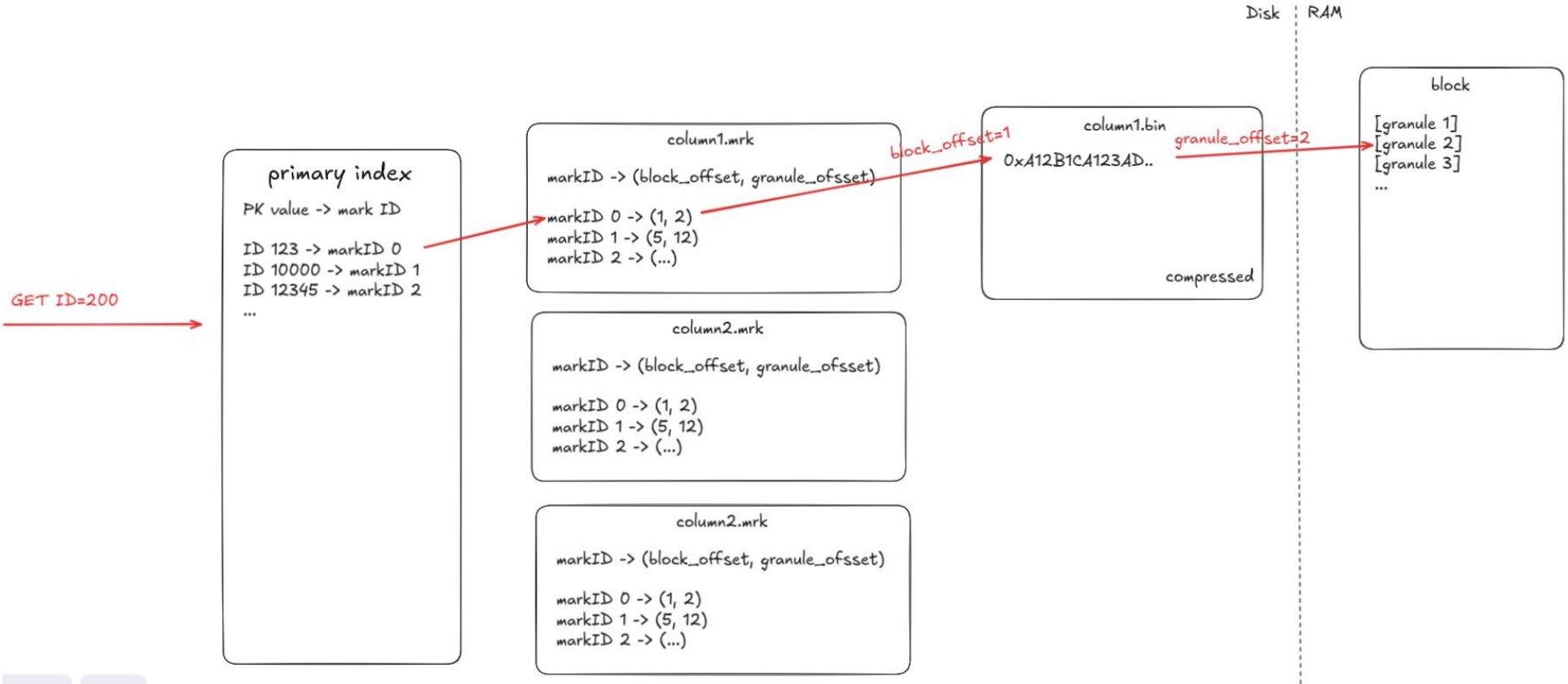
markID 1 -> (5, 12)

markID 2 -> (...)

column1.bin

0x412B1C41234D..

Idea 1: column-oriented storage w/ sparse index



Idea 1: column-oriented storage w/ sparse index

hands-on

Idea 2: compression

Less data on disk => less I/O => faster queries and inserts.

Rule of thumb: optimize **both** compressed, and uncompressed size.

How:

- Optimize your types (*encoding*).
- Optimize your codec (*compression*).

compressed_size	uncompressed_size	ratio
4.07 GiB	131.58 GiB	32.36



compressed_size	uncompressed_size	ratio
1.42 GiB	36.05 GiB	25.36

Idea N:

- Skip-indexes
- Vectorized execution
- Partitioning
- MergeTree family variants
- ...

References

- [1] DDIA Chapter 4
- [2] [An overview of Leveled Compaction in LSM-trees](#)
- [3] [Clickhouse Internals with Tom and Tyler](#)
- [4] [ScyllaDB's Compaction Strategies Series](#)
- [5] [Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging](#)
- [6] [Optimizing ClickHouse with Schemas and Codecs](#)