Lecture 8

Type	Lecture

LSM Storage (Log-Structured-Merge TREE Storage)

why writes in LSM-trees are fast and efficient:

- **Keys sorted in memory using a Red-Black Tree:**

  A Red-Black Tree ensures keys are always sorted, enabling efficient insertion/search in O(log n) time.

- **Sequential writes to disk:**

  Instead of random writes (slow on disks), the in-memory tree (called memtable) is flushed as a Sorted String Table (SSTable) when it's full.

- **SSTables are immutable:**

  Once written, SSTables are never modified (no random writes to disk, no writes-in-place on disk)

- **Updates = inserts:**

  Any update is treated as adding a new key-value pair; it overrides older entries logically (no random writes to disk, no writes-in-place on disk).

- **Deletions = tombstone inserts:**

  Instead of removing a key deletion is treated as an insert, a "tombstone" marker is added → signals deletion during later compaction (no random writes to disk, no writes-in-place on disk).

LSM prioritizes write throughput over immediate read performance. Reads need to check multiple SSTables, hence the need for optimizations like compaction, sparse indexes, Bloom Filters.

Downside:

- **Obsolete key-value pairs accumulate:**

  Since updates/deletes add new entries instead of modifying/deleting in place, multiple outdated versions exist → wasted space.

- **Reads slow down over time:**

  Since keys may exist in multiple SSTables, a read may need to search every SSTable until it finds the latest valid record.

Compaction

compaction is the solution for the problems caused by the LSM design

- Merges multiple SSTables → discards obsolete key-value pairs → reduces number of files to search (reduces disk access).
- Improves read performance by reducing duplication and SSTable count.

Without compaction → LSM would keep degrading → reads get slower → more disk usage.

LSM-Storage: Read Operations

1. load (1st page of) most recent file (segment n) from disk and check smallest key to see whether key COULD in in the file.
    - If yes → search whole segment n for key → if found: return key-value pair
    - If no or if key not found in segment n, move to segment (n - 1)
2. Repeat until:
    - key is found, or
    - reached the oldest segment without success.

*Without optimization → every read could degrade into a **linear search across multiple files**.*

LSM Storage Read Operations: Sparse Indexes

__Sparse index__

A sparse index stores only one key per page, typically the smallest key in the page in memory:

Segment 1:

```
[Page 0]: keys = a, b, c
[Page 1]: keys = d, e, f
[Page 2]: keys = g, h, i
[Page 3]: keys = j, k, l
```

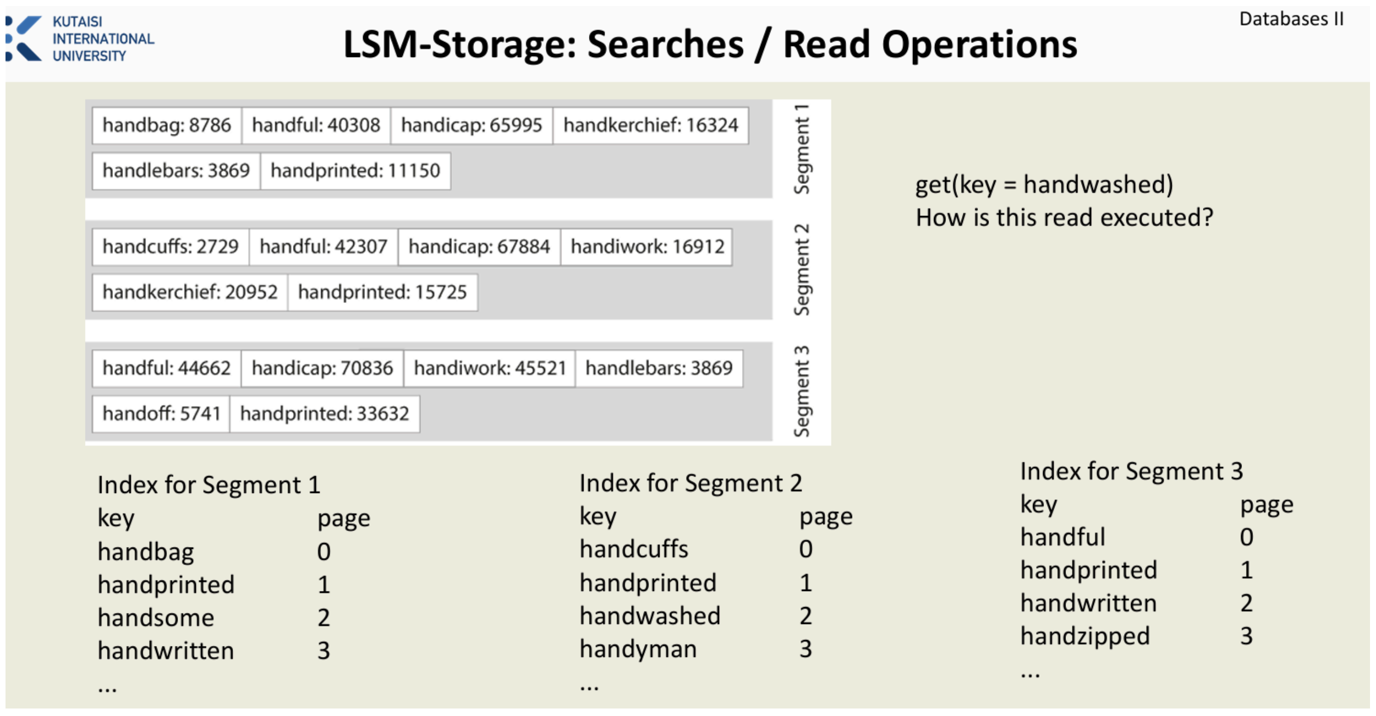
Sparse index for segment 1:

```
a → page 0
d → page 1
g → page 2
j → page 3
```

Instead of indexing every key in memory (too large), Store the smallest key for each disk page in memory.

This allows Binary search in sparse index to locate correct page quickly and then load that single page from disk instead of scanning all (loads 1 page instead of the whole file). This reduces disk reads and speeds up key location.

▼ Example



KUTAISI INTERNATIONAL UNIVERSITY

Databases II

LSM-Storage: Searches / Read Operations

Segment 1: handbag: 8786 | handful: 40308 | handicap: 65995 | handkerchief: 16324 | handlebars: 3869 | handprinted: 11150

Segment 2: handcuffs: 2729 | handful: 42307 | handicap: 67884 | handiwork: 16912 | handkerchief: 20952 | handprinted: 15725

Segment 3: handful: 44662 | handicap: 70836 | handiwork: 45521 | handlebars: 3869 | handoff: 5741 | handprinted: 33632

get(key = handwashed)
How is this read executed?

Index for Segment 1

key	page
handbag	0
handprinted	1
handsome	2
handwritten	3
...	

Index for Segment 2

key	page
handcuffs	0
handprinted	1
handwashed	2
handyman	3
...	

Index for Segment 3

key	page
handful	0
handprinted	1
handwritten	2
handzipped	3
...	

1. Search handwashed in the segment 3, handprinted < handwashed < handwritten, so if handwashed is in this segment it must be on page 1, if handwashed is not on this page,
2. then go to segment 2, we find the handwashed key on page 2, go to page 2.

Read Operations using Bloom Filters

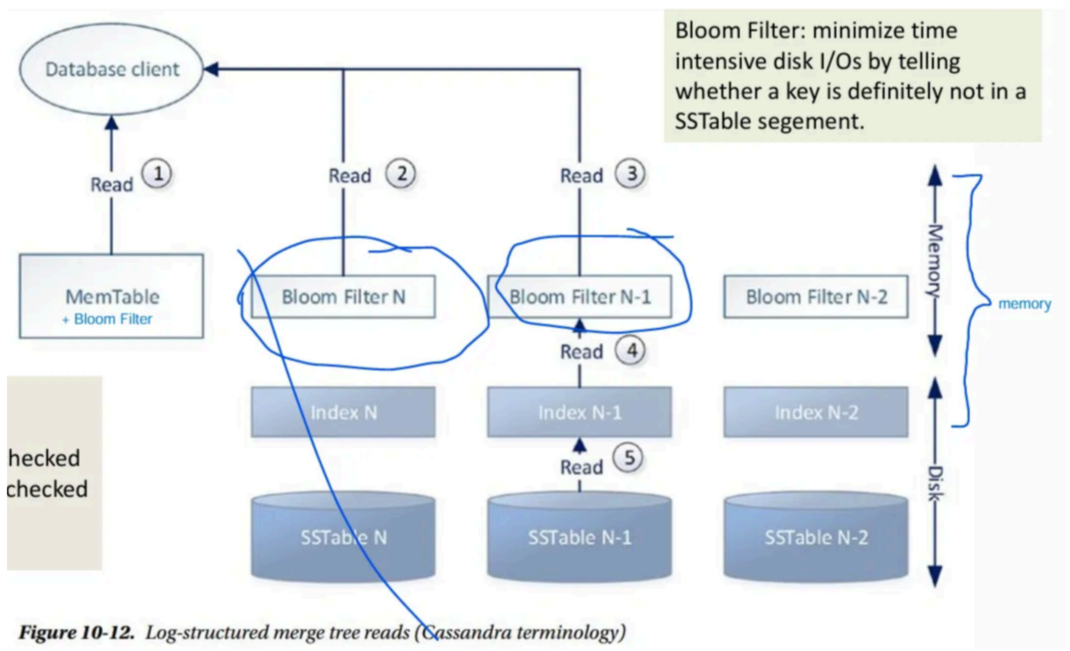
The sparse index tells whether a key could be on a certain page of a segment file. It still forces the system to load all pages on which the key could be (forces disk access even if key is not really there).

Bloom Filter adds negative filtering, which minimizes costly disk access considerably:

- Check Bloom Filter first → can tell if key is **definitely NOT in a segment.**

- If Bloom Filter says “no” → skip loading disk page.

Bloom Filter filters out unnecessary disk reads → saving ***time and resources.***



Bloom Filter: minimize time intensive disk I/Os by telling whether a key is definitely not in a SSTable segement.

Figure 10-12. Log-structured merge tree reads (Cassandra terminology)

1. **Check Bloom Filter:**

   tells if key could exist in segment.

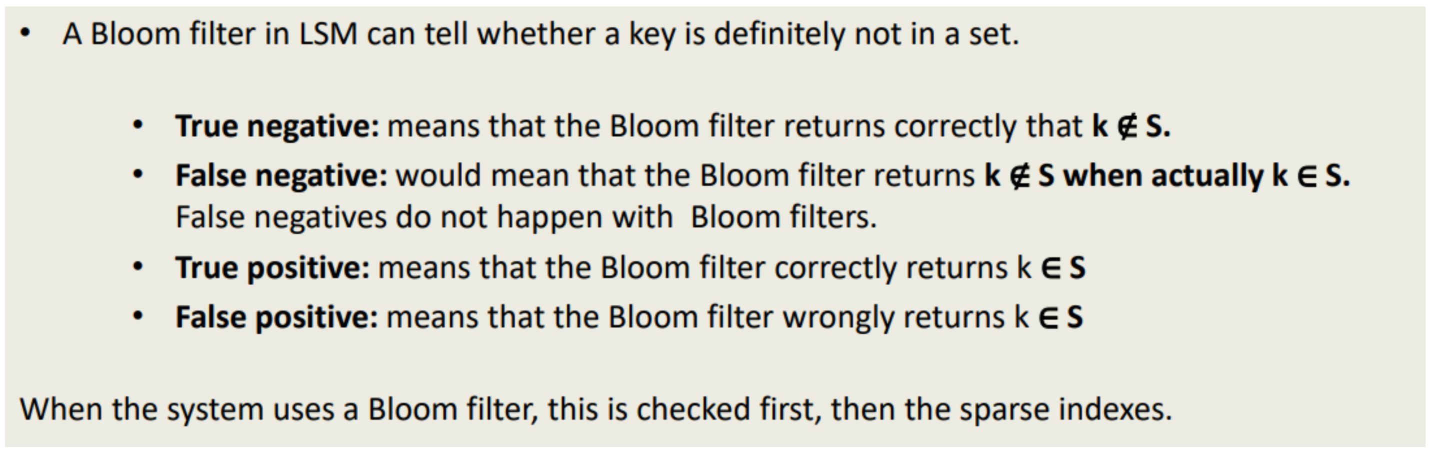
2. **Check sparse index:**

   if Bloom says “maybe” (we say maybe, because of collisions, which i will explain later).

3. **Access disk:**

   if sparse index confirms key might be inside page.

LSM-Storage: Bloom Filters



- A Bloom filter in LSM can tell whether a key is definitely not in a set.
  - **True negative:** means that the Bloom filter returns correctly that **k ∉ S.**
  - **False negative:** would mean that the Bloom filter returns **k ∉ S when actually k ∈ S.** False negatives do not happen with Bloom filters.
  - **True positive:** means that the Bloom filter correctly returns k **∈ S**
  - **False positive:** means that the Bloom filter wrongly returns k **∈ S**

When the system uses a Bloom filter, this is checked first, then the sparse indexes.

false positives (happens because of collisions) → acceptable for many database in order to achieve speed and space-efficiency (also slower read is better than wrong read).

Bloom Filter Hash Functions

Important details about hash functions:

- converts variable-length keys to fixed-length hash values.
- Hash values should be evenly distributed over the keyspace → minimizes collisions.
- Don’t need collision management → collisions are acceptable to achieve speed and space efficiency.

Each Bloom Filter for one data set (in our case: one segment):

1. has one bit array storing the hash values
2. k hash functions
   - need to be fast
   - are independent → can be computed in parallel
   - need to spread evenly over keyspace (which is the bit array)
   - k is a constant → insertions into the bloom filter have constant time complexity $O(1)$.

Collision:

when you have 1 in a bittaray on index i and hash function gives you 1 to insert on index i, you do not change it, you will still have 1 on index i, this is called a collision and it causes false positives.

Example of Bloom filter inserts

bloom filter is in memory, so the process where the bloom filter is filled happens when red-black trees are constructed (it also happens in memory).

Our bitarray has 20 bits, initially always all set to 0:

$$B = 00000000000000000000$$

We want to insert:

1: some value, 12 : some value, 7: some value

We use 3 hash functions:

$$h1(x) = x \bmod 20$$
$$h2(x) = 3x \bmod 20$$
$$h3(x) = 7x \bmod 20$$

1. insert 1:

$$h1(1) = 1 \bmod 20 = 1$$
$$h2(1) = 3 \bmod 20 = 1$$
$$h3(1) = 7 \bmod 20 = 1$$

After insert Bloom filter:

$$B = 01010001000000000000$$

2. insert 12:

$$h1(12) = 12 \bmod 20 = 12$$
$$h2(12) = 36 \bmod 20 = 16$$
$$h3(12) = 84 \bmod 20 = 4$$

After insert Bloom filter:

$$B = 0101\ 1001\ 0000\ 1000\ 1000$$

3. insert 7:

$$h1(7) = 7 \bmod 20 = 7$$
$$h2(7) = 21 \bmod 20 = 1$$
$$h3(7) = 49 \bmod 20 = 9$$

After insert Bloom filter:

$$B = 0101\ 1001\ 0100\ 1000\ 1000$$

So finally Bloom filter for this segment will be:

$$B = 0101\ 1001\ 0100\ 1000\ 1000$$

Final insert:

Once memtable is flushed:

- SSTable is written to disk.
- Memtable is discarded(typically a memtable has a fixed size, between 1 – 4 MB – and not only 3 keys).
- Bloom Filter is kept in memory (needed for fast lookups across disk SSTables).

even if the key is deleted (tombstoned) the bloom filter will still say that it is in the segment.

Bloom Filter Search

Bloom filter for the segment (for example):

$$B = 0101\ 1001\ 0100\ 1000\ 1000$$

True negative: means that the Bloom filter returns correctly that k ∉ S.

Example: get(key=4)

$$\begin{gathered} h1(4) = 4 \bmod 20 = 4 \rightarrow bit = 1 \\ h2(4) = 12 \bmod 20 = 12 \rightarrow bit = 1 \\ h3(4) = 28 \bmod 20 = 8 \rightarrow bit = 0 \end{gathered}$$

If at least one bit = 0, the key is NOT in the set, so key = 4 is not in the set(segment)

True positive: means that the Bloom filter correctly returns k ∈ S

Example: get(key=12)

$$\begin{gathered} h1(12) = 12 \bmod 20 = 12 \rightarrow bit = 1 \\ h2(12) = 36 \bmod 20 = 16 \rightarrow bit = 1 \\ h3(12) = 84 \bmod 20 = 4 \rightarrow bit = 1 \end{gathered}$$

all the bits are 1, so key = 12 MAY BE in the set (segment)

to get key 12 after going in the segment because of the bloom filter, we will move to the sparse index of that segment.

False positive: means that the Bloom filter wrongly returns k ∈ S (happens because of collisions)

Example: get(key=9)

$$\begin{gathered} h1(9) = 9 \bmod 20 = 9 \rightarrow bit = 1 \\ h2(9) = 21 \bmod 20 = 1 \rightarrow bit = 1 \\ h3(9) = 63 \bmod 20 = 3 \rightarrow bit = 1 \end{gathered}$$

Key = 9 MAY BE in the set (segment), but after we move to the sparse index of this segment we get to know that it is not in there so it is a false positive.

Bloom Filter Use Cases

1. Key-Value / Wide-Column Databases: Use Bloom filters to reduce costly disk access for read operations.
2. Routing (deciding where to forward)
3. Check against a "blacklist"
    - Chrome uses bloom filters to check against malicious URLs
    - Can be used exclude objects from being recommended
    - Police / Border control checks against a wanted list

Some of the use cases can tolerate a considerable false-positive rate. A false positive in a database search slows down the search but does not any other harm besides making the read inefficient. usually the rate is 10-15%, to lower it we need to adjust a parameter, that is extended bit arrays.

Bloom Filter False_Positive_Rate (fp_rate) Calculation

Shows required parameters to achieve target false positive rates:

Rate	Bits per element	Hash functions
10%	4.8	3
1%	9.6	7
0.1%	14.4	10
0.01%	19.2	13

formula → `m = - (n * ln(p)) / (ln(2)^2)` .

Cassandra Bloom Filter Tuning

- `bloom_filter_fp_chance` configures false positive rate during `CREATE TABLE` .

Rocks DB Bloom Filter Tuning

- Each SST file has its own Bloom Filter.
- Created when SST file is written.

- Default = `0.01` (1%).
- Higher value → smaller Bloom Filter → uses less memory → but more false positives.
- Lower value → more accurate → fewer unnecessary disk reads → more memory (bigger bit array) → consumes more RAM.

- Loaded into memory on file open.
- Discarded when file closed.

Uses `bits_per_key` setting to control Bloom Filter size (and thus false positive rate).

Not configurable for number of hash functions.