Lecture 7

Type	Lecture

NoSQL Databases

LSM-trees are commonly used in NoSQL databases like Cassandra, LevelDB, and RocksDB. These databases favor write throughput and eventual consistency, suiting LSM's design.

Heap-File / B+ Tree Storage Model

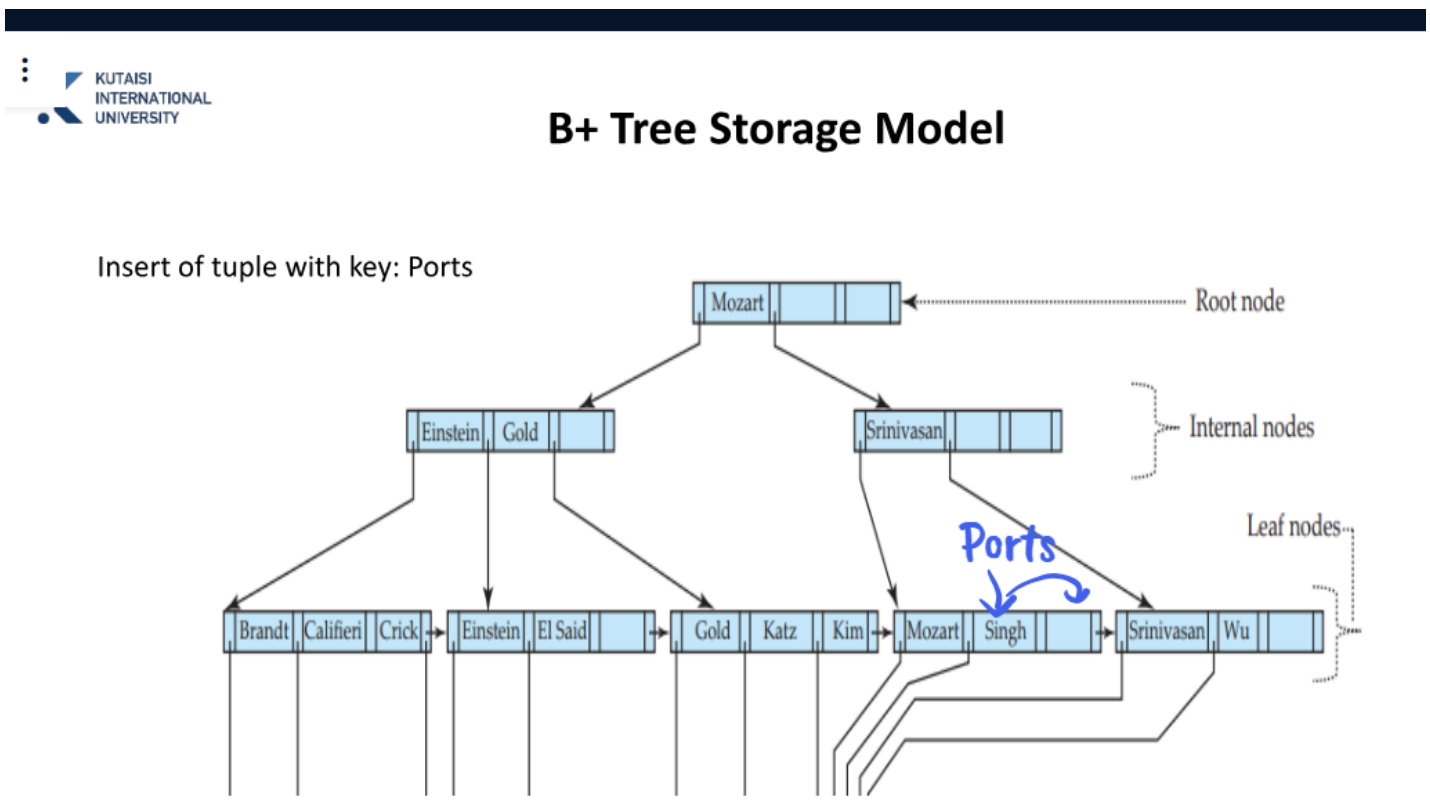
Advantages (properties) of the B+ Tree storage model:

balanced, sorted, page-oriented, optimized for disk I/O.

Inserts involve multiple disk operations:

1. search the B+Tree to find where to insert → random disk read.
2. write the data into a **heap file** → random disk write.
3. update the **index (B+Tree)** → random write.
4. write to the **WAL (Write-Ahead Log)** → sequential write.
5. if updated **secondary indexes** → more random writes.

▼ Example



Insert of tuple with key: Ports

→ traverse PK BTREE → locate correct leaf node → load leaf into buffer
1 random disk access, assuming that root node and internal nodes of PK BTREE are in buffer cache.
→ with the help of the BTREE pointer, locate heap file page → load heap file page into buffer
1 random disk access (heap file is fragmented)
→ insert tuple into heap file page and insert PK correctly into BTREE leaf node
→ write insert into WAL and flush WAL to disk
sequential write
→ write modified heap file page back to disk
1 random disk access
→ write modified PK BTREE leaf back to disk
1 random disk access

For each further secondary index, add two more random disk access.

On spinning disks (HDDS), there is mechnical movement. SSDs erase the page and then re-write.

Heap-File & BTREE storage supprts read operations better than write operations

Downside: Random disk I/O is costly, especially on HDDs.

Random I/O versus Sequential I/O

Random I/O:

Positions arm each time

Latency time each time

$1000 \times (5ms + 3ms)$

$8000ms \rightarrow 8s + transfer\ time$

8s for 1000 4KB blocks (very slow)

Sequential I/O (Chained I/O**):**

Position once, then “scrape off the disk”

$5ms + 3ms \rightarrow 8ms + transfer\ time$

8ms (much faster)

LSM-trees leverage this by writing **sequentially to disk**.

Heap-File / B-Tree Storage vs LSM Storage

Heap-File / B-Tree Storage

Bottleneck is the write-throughput (velocity)

The system is slow because writing data to disk is the slowest part — that's the bottleneck.

slow:

- random writes on disk
- writes-in-place

Read operations are supported by indexes.

LSM Storage (Log-Structured-Merge TREE Storage)

logs only sequential writes on disk and files on disk are immutable.

Tree: sorted keys

LSM trees **batch and sort in memory**, then write sequentially.

memory access is much quicker than a disk access, which is why LSM is faster then b-tree since it does most of the operations in memory.

LSM-Storage

Uses **two-tiered storage**:

1. **Memory Structure (in memory):**

   Memtable: sorted data structure like a red-black tree or skip list. sorts incoming random write operations according to the key.

2. **Disk Structure (on disk):**

   SSTables (Sorted String tables): content of red-black-trees are flushed to disk sequentially into SSTs in a sorted format from memory. The disk files (SSTables) are immutable(once written, never changed).

When new data arrives (e.g. an update), it's written again in a new SSTable — old versions are cleaned up later by compaction.

Compaction: merges multiple SSTables into fewer files.

LSM-Storage: Memtable: Sorting Incoming Random Writes

Writes come in random and need to be sorted, they are sorted into an in-memory (only) tree-structure. The tree structure usually is a self-balancing binary search tree (e.g. a red-black-binary tree). The depth (or height) of the tree that we are so concerned about in B-Trees is not an important aspect because Tree only exists in memory and tree does not get large.

memtable is Cassandra terminology. The term memtable does not tell the exact implementation. This could be any self-balancing tree-structure or even a B-tree or yet another structure used to sort random incoming writes.

Red-Black-Tree as memtable

These 5 rules must always be true:

1. Each node is either red or black.
2. The root is always black.
3. Every leaf (NIL pointer/null) is black.
4. Red nodes cannot have red children (no two red nodes in a row — this is called the "No Red-Red" rule, !RR).
5. Every path from a node to its descendant leaves has the same number of black nodes.

These rules ensure that the tree doesn't become unbalanced (i.e., too tall or skewed).

Insertion Algorithm Red-Black-Tree

Let x be the node to insert

- Perform standard BST insertion and color newly inserted nodes as RED, if x is not the root.
- If x is the root, color x BLACK
- Do the following if the color of x parent is Red and x is not the root.

a) If x uncle is RED
  (i) Recolor parent and uncle as BLACK.
  (ii) Recolor grandparent as RED if grandparent is not root.

b) If x uncle is BLACK, then there can be four configurations for x, parent (p) and grandparent (gp)
  (i) Left Left Case (line) rotate gp right (in the opposite direction of x),
    recolor: former p (now root of subtree): black, former gp (now right child: red)
  (ii) Left Right Case (triangle) – rotate parent left (in the opposite direction of x)
  (iii) Right Right Case (line) - rotate gp left ((in the opposite direction of x),
    recolor: former p (now root of subtree): black, former gp (now left child: red)
  (iv) Right Left Case (triangle) – rotate parent right (in the opposite direction of x)

Memtable Node vs B+Tree Node

Memtable Node Structure

```
rb_node_structure {
    key: string or number used for ordering
    value: payload like (docID, frequency)
    color: RED | BLACK
    parent: pointer to parent node
    left: pointer to left child
```

BTREE Node Structure

```
btree_leaf_node_structure {
 num_keys: number of keys
 keys[array_of_keys]
 values[array_of_values]
```

```
    right: pointer to right child
}
```

small node size.
inserts: simple, modifications to some pointers according to rb-tree rules.

```
    pointer_next_leaf
}
```

large node size, many keys to keep the tree shallow.
inserts: need shifting the keys array,
potential need for node-splitting.

LSM-Storage: SSTables Writing memtables to persistent disk structure

The memtable holds data **in memory** temporarily. When it gets full (reaches a **size threshold**), it's time to **flush** it to disk. This flushed file becomes an **SSTable (Sorted String Table)**.

The memtable is written to disk in **sorted order**. Since the write is **sequential**, it's **very fast** (especially on HDDs, and also efficient on SSDs). No random I/O is needed.

As soon as flushing starts:

- A **new empty memtable** is created.
- All **new writes** go to this new memtable.
- Meanwhile, the old one is still being flushed.

As long as the flushing process continues, both memtables need to be accessible in memory, Because:

- **New data** is in the new memtable.
- **Old data not yet flushed** may still be queried or referenced.

So both exist in memory during the transition.

Once the old memtable is fully written to disk and confirmed, it can be safely deleted from memory.

This is a cyclical process:

- Memtables fill → flushed → become immutable SSTables on disk.
- New memtables begin, and the cycle continues.

SSTables are immutable - Once an SSTable is written, it is never changed. Updates or deletes are handled by writing new records and later compacting SSTables (merging/removing duplicates).

LSM-Storage: Updates

Any new update is written into the current memtable (not to disk directly). This allows updates to be fast and efficient — no disk I/O involved initially.

If the key is in the current memtable

update is done directly in memtable, the key already exists in memory, so you just change its value in place, no need to look on disk or do anything extra. This is fast and avoids any write amplification.

If the key is not in the current memtable

The key was likely flushed to disk already (in a previous SSTable). In this case, the update is appended as a new entry in the memtable. So now there are multiple versions of the same key: **Older version** in SSTable and a **New version** in memtable or a newer SSTable later.

As time passes, more SSTables are created, and duplicate keys with different values pile up. This is a natural result of LSM-tree design and will be resolved by compaction (merging & deduplication).

How is this different from B-Trees?

> In B-Trees, a key exists in only one location, and any update replaces the original value in place. In LSM-Trees, you don’t overwrite the old value. Instead, you add a new value for the same key. This leads to multiple versions across SSTables — which are later cleaned up during compaction.

LSM-Storage: Compaction / Merge

Compaction means merging multiple SSTable segments on disk into a new segment, while removing duplicate keys and keeping only the most recent value.

If there are duplicate keys in different segments, the key with the most recent value is kept and the keys with stale values are discarded.

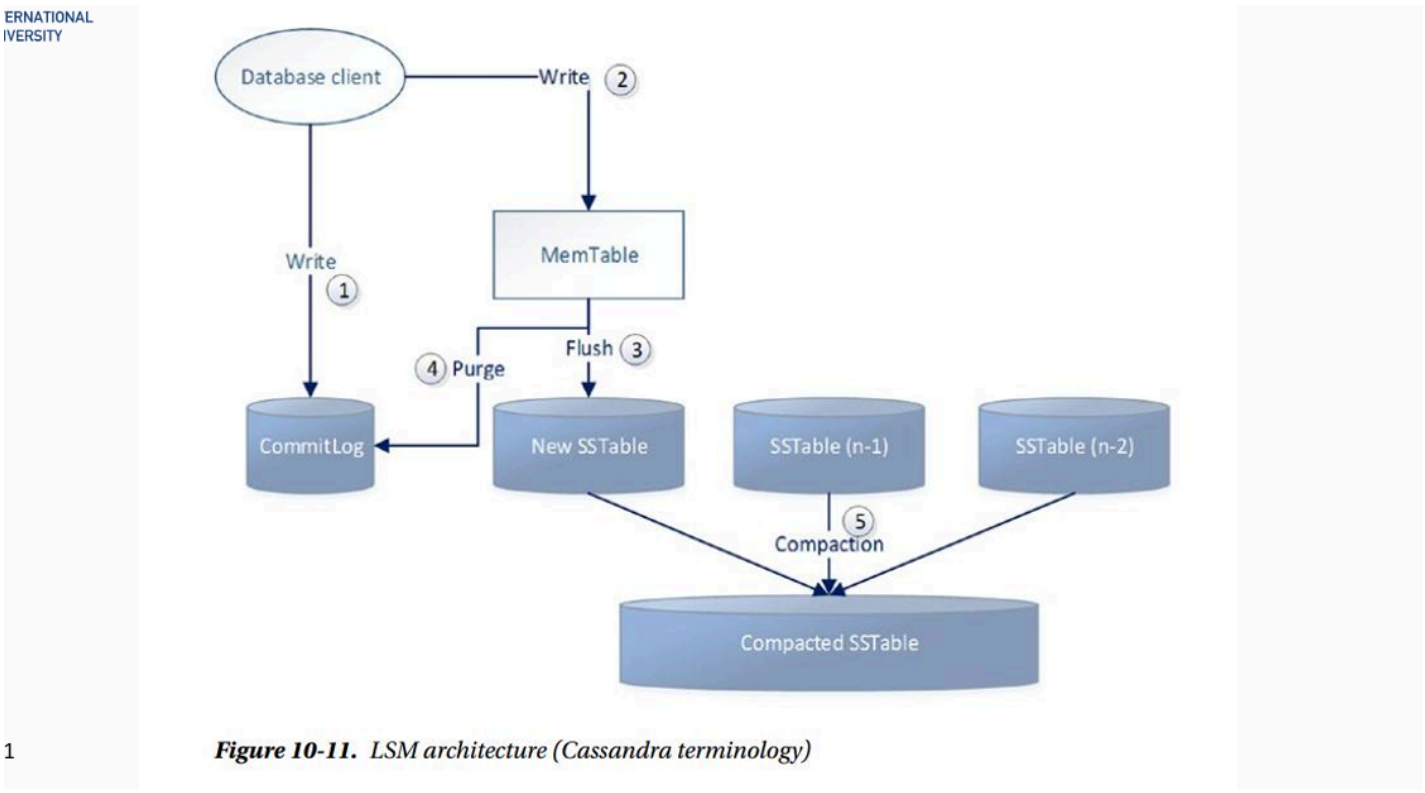
How it works:

Compaction always writes merged segments into a new file, that is created specifically to store the merged data.

Compaction can happen in the background: While compaction goes on, writes are still served in memtable and reads are served in the old, still accessible segments.

After compaction is complete, read requests are routed to the new segment and the old segments are discarded.

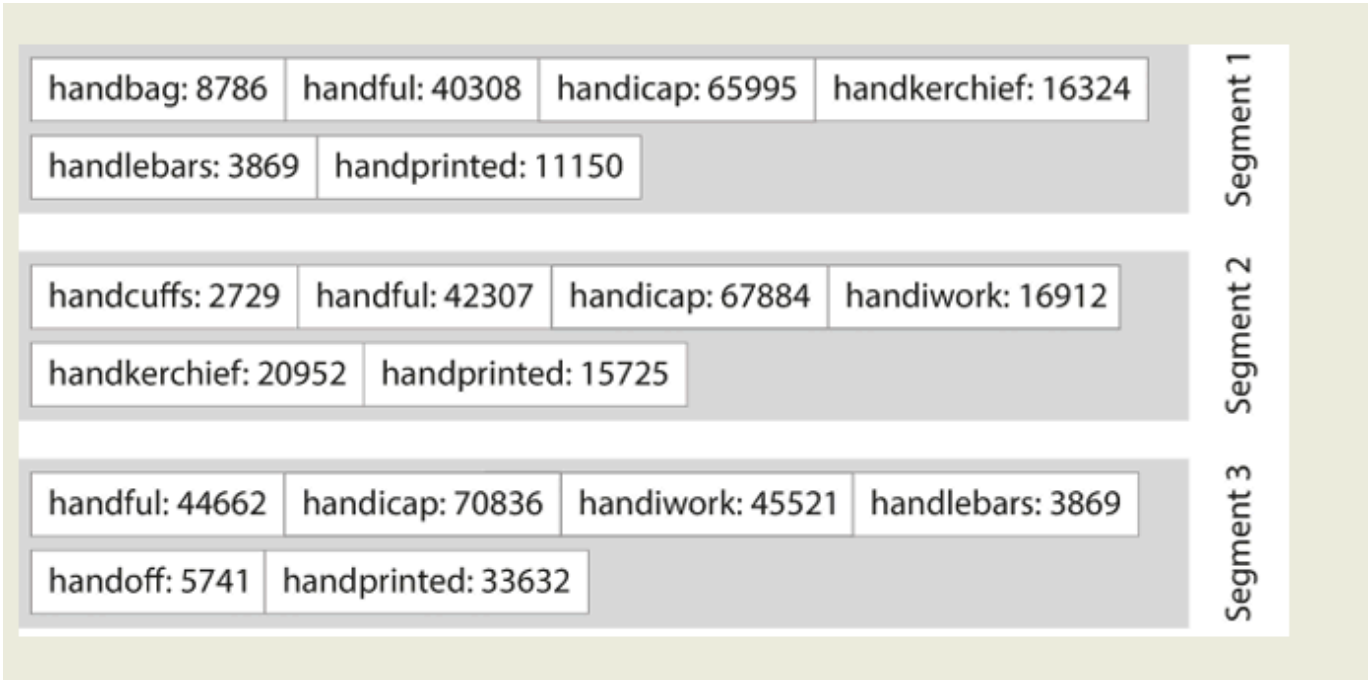
Architecture:



1 ***Figure 10-11.*** *LSM architecture (Cassandra terminology)*

▼ **Example of compaction**

Segment 1 is oldest, segment 3 is most recent.



merged segment:

handbag: 8786, handlebars: 3869, handcuffs: 2729, handful: 44662, handicap: 70836, handkerchief: 20952, handoff: 5741, handprinted: 33632, handwork: 45521

This is now written as a new SSTable, and Segments 1–3 can be deleted.

LSM-Storage: Searches / Read Operations

To read:

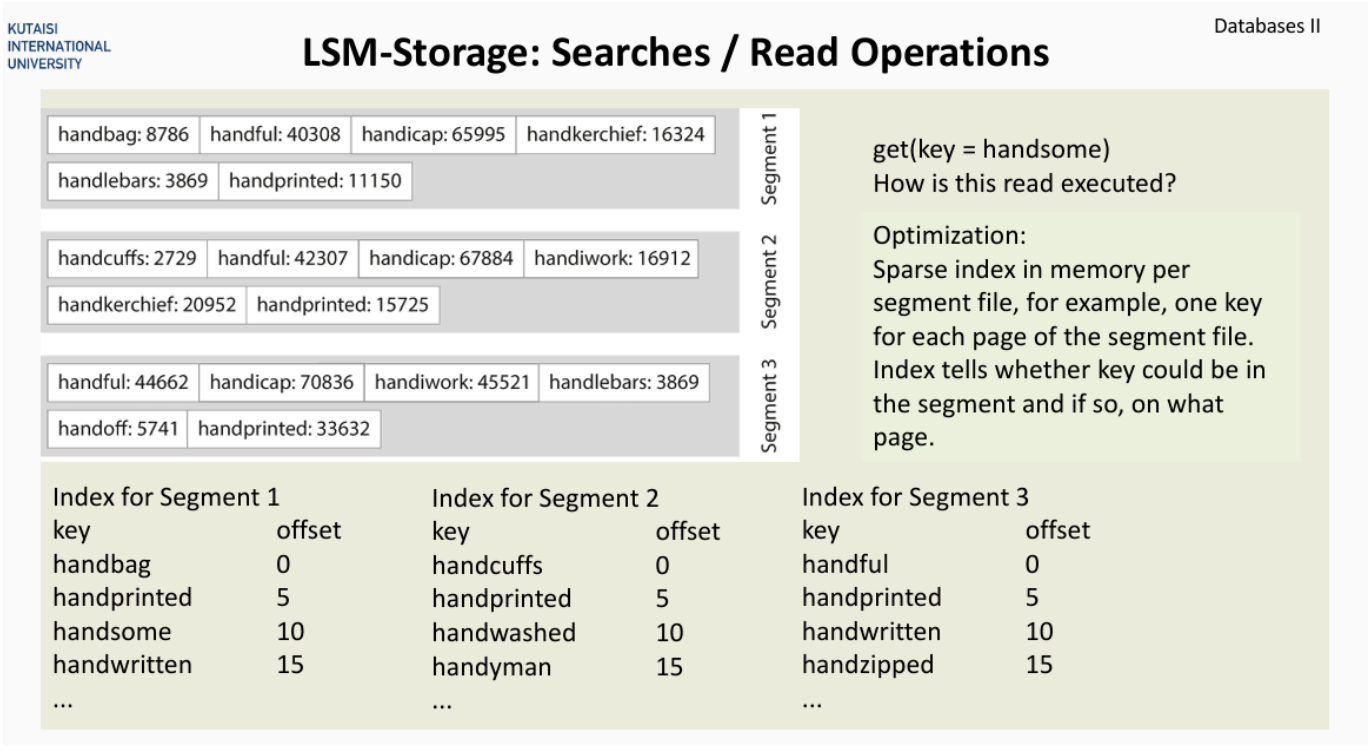
1. Check memtable
2. Search SSTables in reverse order (newest to oldest)
3. Use sparse indexes to skip irrelevant pages.

A sparse index is kept in memory (not on disk). It contains some keys and their offsets — not every key, just one key per page or block.

Benefits of Sparse Indexes:

- Saves memory (not every key stored in memory).
- Enables fast lookups without scanning entire SSTables.
- Maintains LSM-tree's high write and read efficiency.

▼ **Example of read with sparse index**



1. Look in Segment 3's index: `"handsome"` is not in the sparse index → not here.
2. Check Segment 2's index: same result → not here.
3. Check Segment 1's index:
    - `"handsome"` is listed → go to offset 10
    - Scan the block starting at that offset to find the full key.

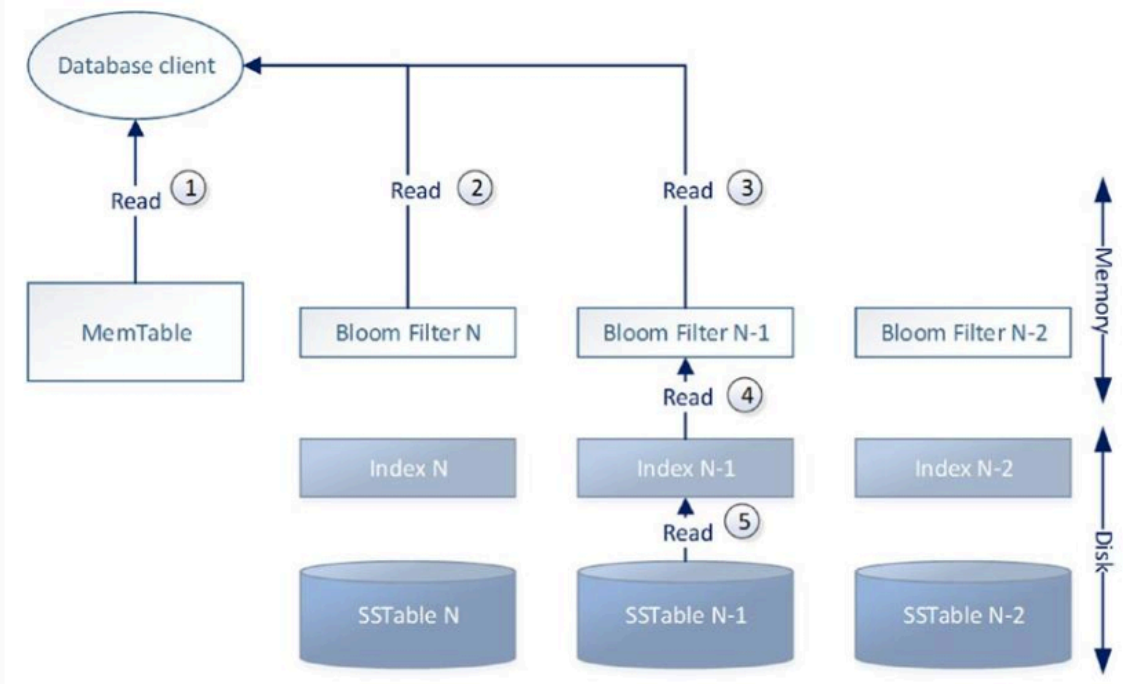
If `"handsome"` is found, return the value and stop searching further.

LSM-Storage: Bloom Filters

- The search in the SSTables could further be optimized if one could say for certain that a key is not in a segment (even though the index says that it could be in the segment).
- This is exactly what a Bloom filter does: A Bloom filter is a data structure that tells if an element is definitely not in a set. It cannot tell whether a key is positively in a set.
- LSM: A Bloom filter can tell whether a key is definitely not in a segment.
    - **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 database system uses a Bloom filter, this is checked first, then the sparse indexes.
- Bloom filters are kept in memory.

Architecture:

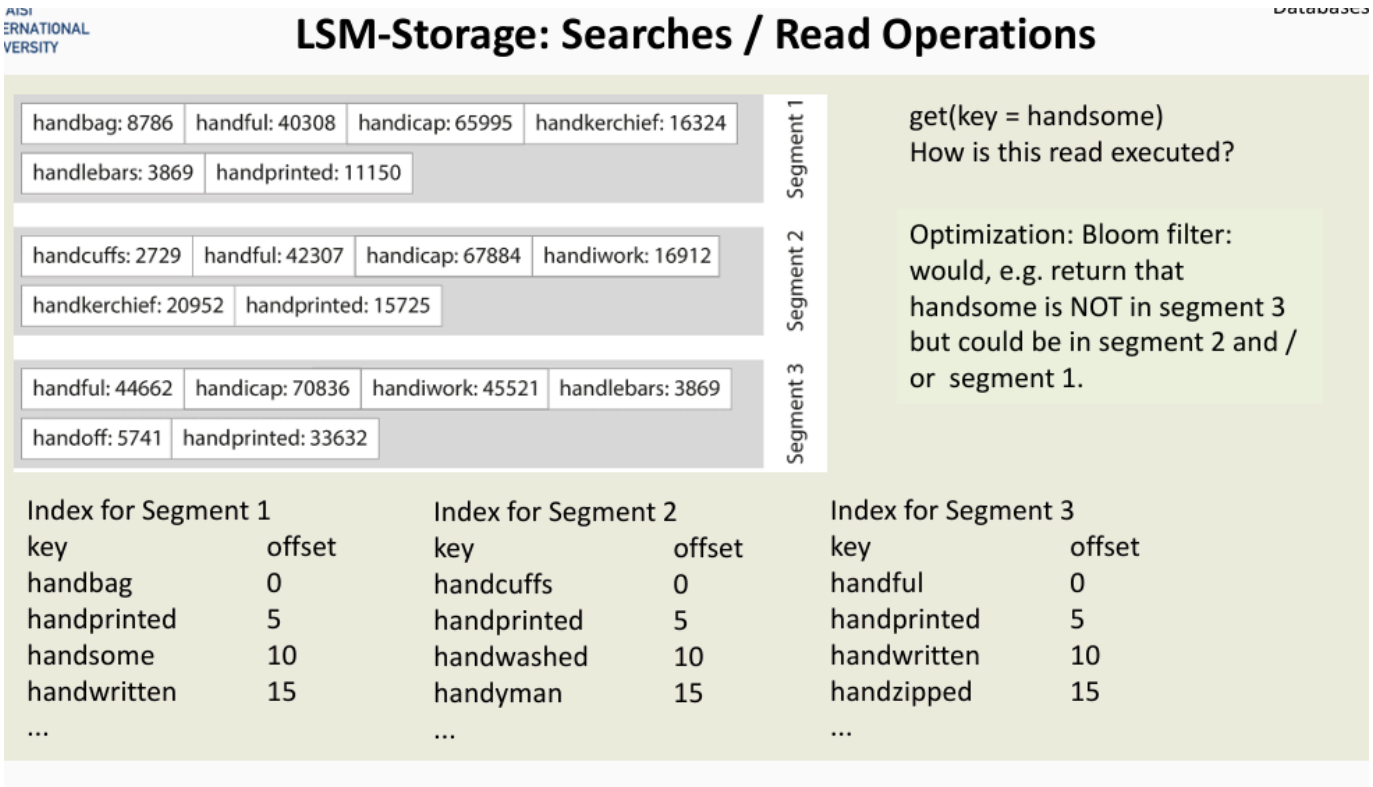
ITERNATIONAL
NIVERSITY



61

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

▼ **Example of read with bloom filter**



Now that we've **eliminated Segment 3**, we move on:

→ Check Segment 2:

- Look at the sparse index: `"handsome"` is **not in index**
- So we **probably skip** Segment 2 too

→ Check Segment 1:

- Sparse index for Segment 1 shows: handsome → offset 10

So we go to offset 10 in Segment 1, We find the actual value and return it.

Difference between sparse index and bloom filter

Bloom filter helps you skip entire files. **Sparse index** helps you jump to the right spot in the file.

Feature	Bloom Filter	Sparse Index
Helps skip segment?	✅ Yes	❌ No (must still open segment)
Helps locate inside?	❌ No	✅ Yes
False positives?	✅ Yes	❌ No
False negatives?	❌ No	❌ No

> Bloom filter = *"Should I even open this SSTable?"*
>
> Sparse index = *"If I open it, where should I look?"*