CS 564: Database Management Systems

Lecture 18: Advanced Indexing

Xiangyao Yu

3/4/2024

Module B2 Indexes

Hash index

B+ tree index

LSM tree index

External sort

Outline

Discussion of B+ tree

- **Duplicate search keys**
- Calculating B+ tree parameters
- Key compression

Clustered index

Log-Structured Merge (LSM) tree

Duplicates

<u>Duplicate keys</u>: many index entries having the same key value

Solution #1

- All entries with a given key value reside on the same set of pages
- Use overflow pages

Solution #2

- Allow duplicate key values in data entries
- May consider <key, rid> as the new unique key

Outline

- Discussion of B+ tree
  - Duplicate search keys
  - **Calculating B+ tree parameters**
  - Key compression
- Clustered index
- Log-Structured Merge (LSM) tree

B+ Tree – Fan-out (f)

<u>Fan-out</u> ***f***: the number of pointers to child nodes coming out of a non-leaf node

- Compared to binary trees (fan-out =*2*), B+ trees have a high fan-out ($d+1 \leq f \leq 2d+1$)

- The fan-out of B+ trees is dynamic and depends on the key/record size, but we will typically assume it is constant for our cost model

B+ Tree – Fill Factor (F)

Fill-factor ***F***: the percent of available slots in the B+ tree that are filled (0.5 < F < 1)

- It is usually < 1 to leave slack for (quicker) insertions!
- Typical fill factor F = 2/3

B+ Tree – Height

<u>**Height**</u> ***h***: the number of levels of the non-leaf nodes

- B+ tree with only the root has height 0
- I/O requirement for each search = *h*
- High fan-out -> smaller height-> less I/O per search
- Typical heights of B+ trees is 3 or 4

B+ Tree – Example

page size **P** = 4096 bytes

search key size = 30 bytes

address size = 10 bytes

fill-factor **F** = 2/3

number of records = 2,000,000

We assume that the index entries store only the search key and the address of tuple

We assume no duplicate entries

B+ Tree – Example

What is the order **d** and fan-out **f** ?

B+ Tree – Example

What is the order **d** and fan-out **f** ?

– Each non-leaf node stores up to 2**d** keys + (2**d**+1) addresses
– To fit this into a single page, we must have:

$$2d \cdot 30 + (2d + 1) \cdot 10 \leq 4096$$
$$\boldsymbol{d} \leq 51$$

B+ Tree – Example

What is the order **d** and fan-out **f** ?

- Each non-leaf node stores up to 2**d** keys + (2**d**+1) addresses
- To fit this into a single page, we must have:

$$2d \cdot 30 \; + (2d + 1) \cdot 10 \; \leq \; 4096$$
$$\boldsymbol{d} \leq 51$$

Since a maximum capacity node has (2**d**+1) = 103 children, and the fill-factor is 2/3, the equivalent fan-out is $\boldsymbol{f} = 103 * \frac{2}{3} = 68$

B+ Tree – Example

How many leaf pages are in the B+ tree?

B+ Tree – Example

How many leaf pages are in the B+ tree?

- We assume for simplicity that each leaf page stores only pairs of (key, address)
- Each pair needs 30+10 = 40 bytes
- To store 2,000,000 such pairs with fill-factor **F** = 2/3, we need

$$\#leaves\ = (2{,}000{,}000 * 40)/(4{,}096 * \boldsymbol{F}) = 29{,}297 \approx 30{,}000$$

B+ Tree – Example

What is the height **h** of the B+ tree?

B+ Tree – Example

What is the height **h** of the B+ tree?

We calculated that we need to index **N** = 30,000 pages

- **h** = 1 -> indexes **f** pages
- **h** = 2 -> indexes **f**^{2} pages
- …
- **h** = k -> indexes **f**^{k} pages

For our example, $\boldsymbol{h} = \lceil log_{68} 30{,}000 \rceil = 3$

B+ Tree – Example

What is the total size of the tree?

B+ Tree – Example

What is the total size of the tree?

#pages = 1 + 68 + 68^2 + 29,297 = 33,990

B+ Tree – Example

What is the total size of the tree?

#pages = 1 + 68 + 68^2 + 29,297 = 33,990

The top levels of the B+ tree do not take much space and can be kept in the buffer pool

- *level 0* = 1 page ~ 4 KB
- *level 1* = 68 pages ~ 272 KB
- *level 2* = 4,624 pages ~ 18.5 MB
- *level 3* = 30,000 pages ~ 120MB

In practice, the #IO = # of levels that is not cached in buffer pool

Outline

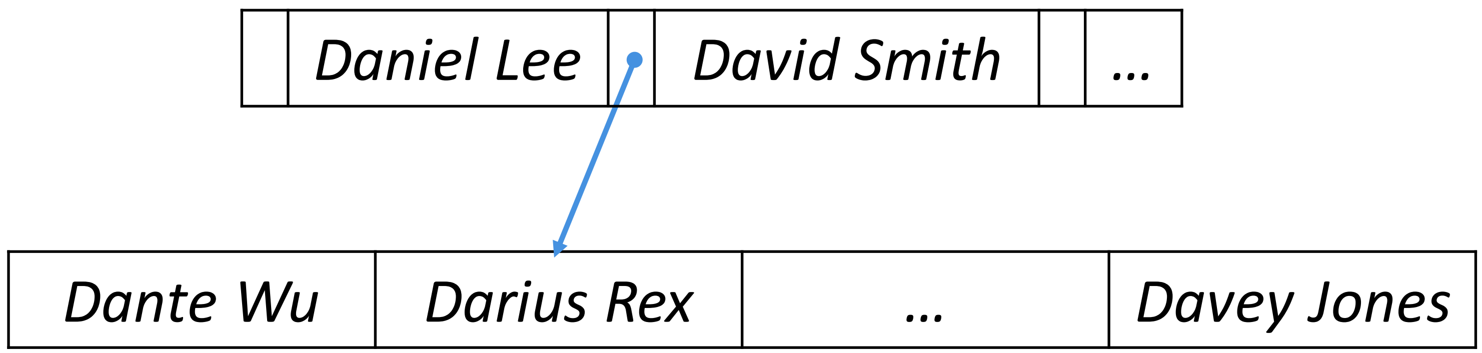
Discussion of B+ tree

- Duplicate search keys
- Calculating B+ tree parameters
- **Key compression**

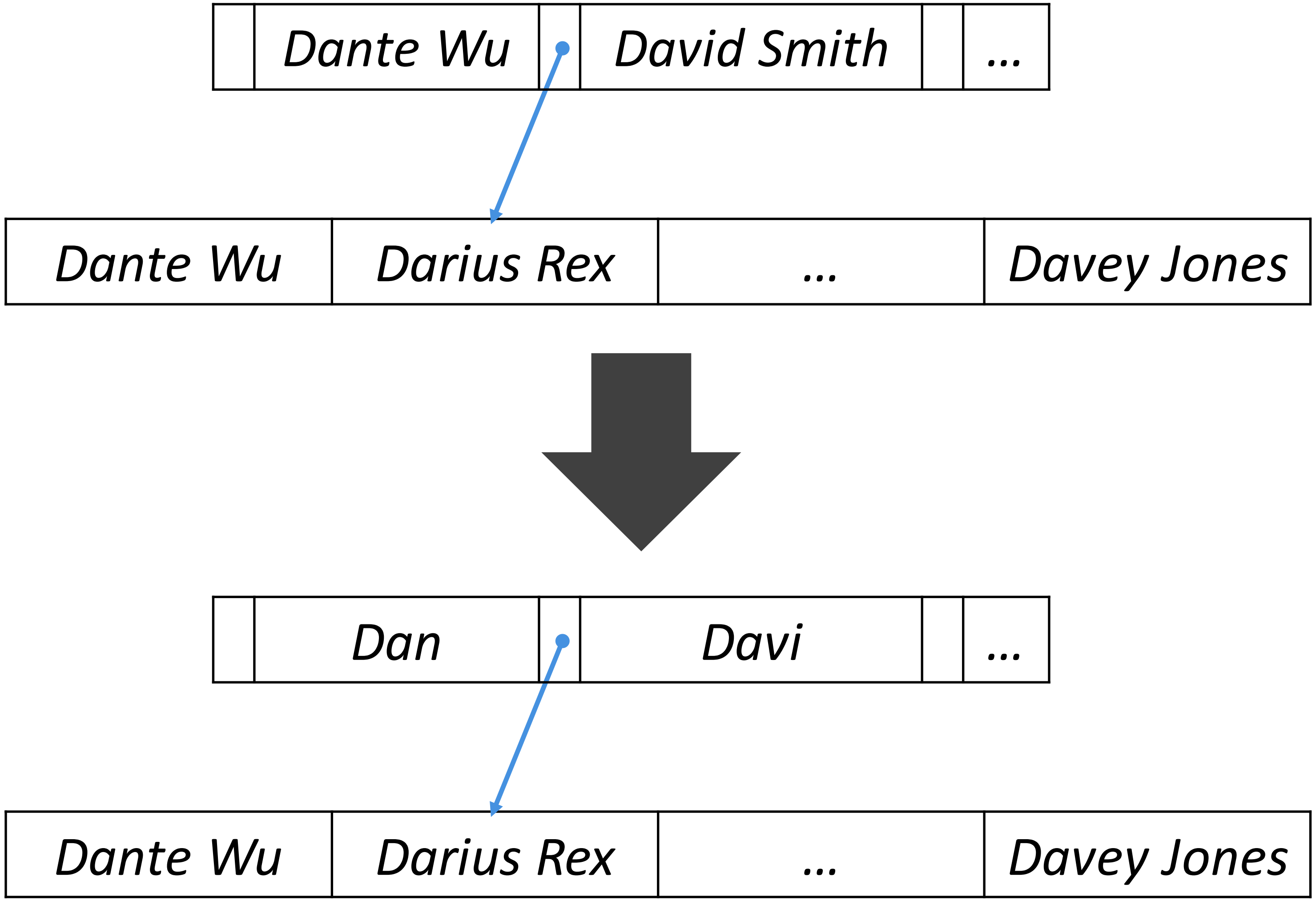
Clustered index

Log-Structured Merge (LSM) tree

Key Compression



Key Compression



Search keys in non-leaf nodes are used only to direct traffic to the appropriate leaf

Need not store search key values in their entirety in non-leaf nodes

Outline

Discussion of B+ tree
- Duplicate search keys
- Calculating B+ tree parameters
- Key compression

Clustered index

Log-Structured Merge (LSM) tree

Clustered Indexes

Clustered index: The order of records in the data pages corresponds to the order of records in the index

- A table can have **only one clustered index**
- An index that stores records as index entries is clustered, by definition
- An index that stores RIDs as index entries may or may not be clustered

Usually, the primary index is implemented as a clustered index

Clustered index is most useful for columns that have range predicates

Outline

- Discussion of B+ tree
  - Duplicate search keys
  - Calculating B+ tree parameters
  - Key compression
- Clustered index
- Log-Structured Merge (LSM) tree
  - **Sequential vs. Random IO**
  - Two-level LSM tree
  - LSM tree

Complexity of Accessing Data

	B+ tree
Write (Insert or update)	**O(log n)** => random IO
Search	**O(log n)**

Can we do better than B+ tree for write-intensive workload?

Complexity of Accessing Data

	B+ tree	Log file
Write (Insert or update)	**O(log n)** => random IO	**O(1)** => sequential IO
Search	**O(log n)**	**O(n)**

Can we do better than B+ tree for write-intensive workload?

Complexity of Accessing Data

	Log file	B+ tree
Write (Insert or update)	**O(1)** => sequential IO	**O(log n)** => random IO
Search	**O(n)**	**O(log n)**

Can we do better than B+ tree for write-intensive workload?

Log-Structured Merge (LSM) Tree [1]

LSM tree: An index that optimizes for writes (by adopting a log structure) while maintaining good search performance

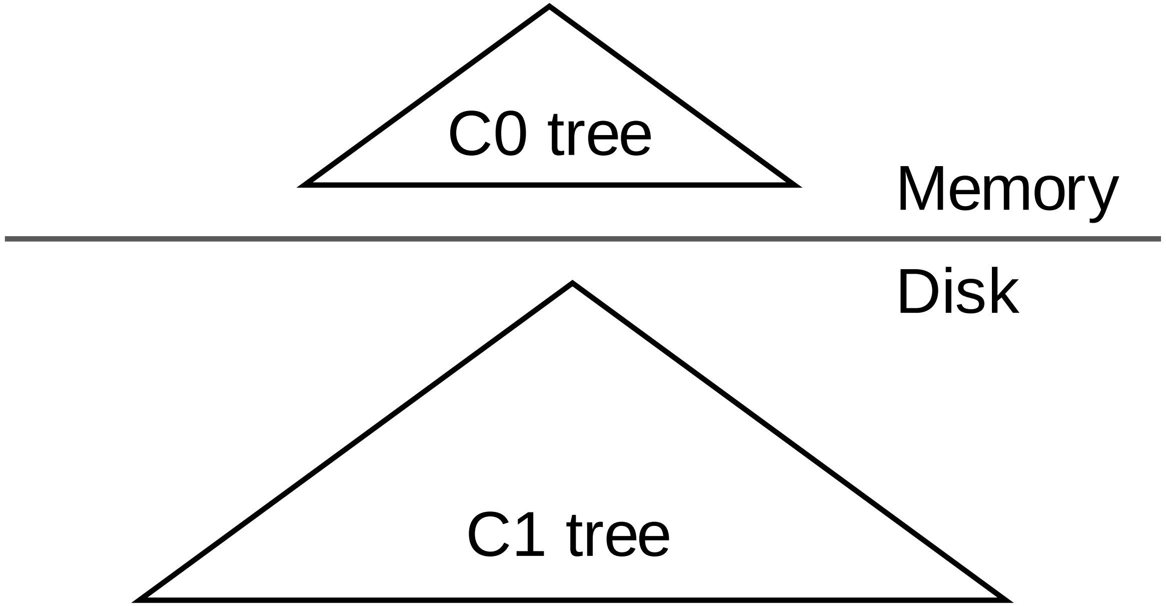
Widely implemented in modern database systems

[1] O'Neil, Patrick, et al. "The log-structured merge-tree (LSM-tree)." *Acta Informatica* 33 (1996): 351-385.

Outline

- Discussion of B+ tree
  - Duplicate search keys
  - Calculating B+ tree parameters
  - Key compression
- Clustered index
- Log-Structured Merge (LSM) tree
  - Sequential vs. Random IO
  - **Two-level LSM tree**
  - LSM tree

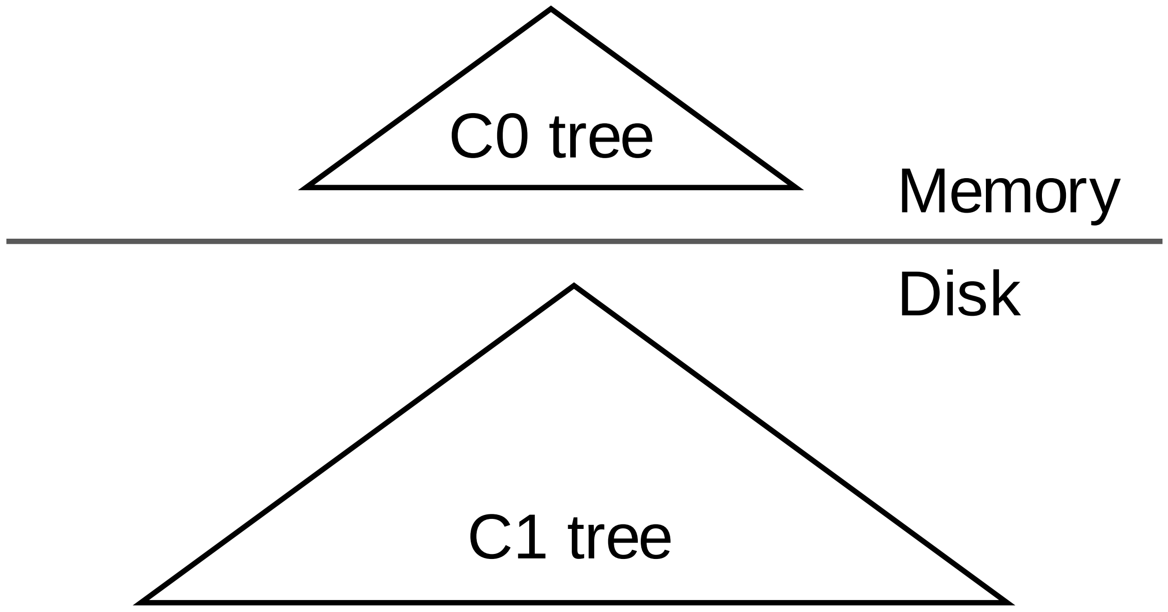
Simple Version: Two-Level LSM Tree



C0 tree (e.g., B-tree) is completely memory resident

C1 tree (e.g., B-tree) is disk resident but some hot pages can be cached in memory

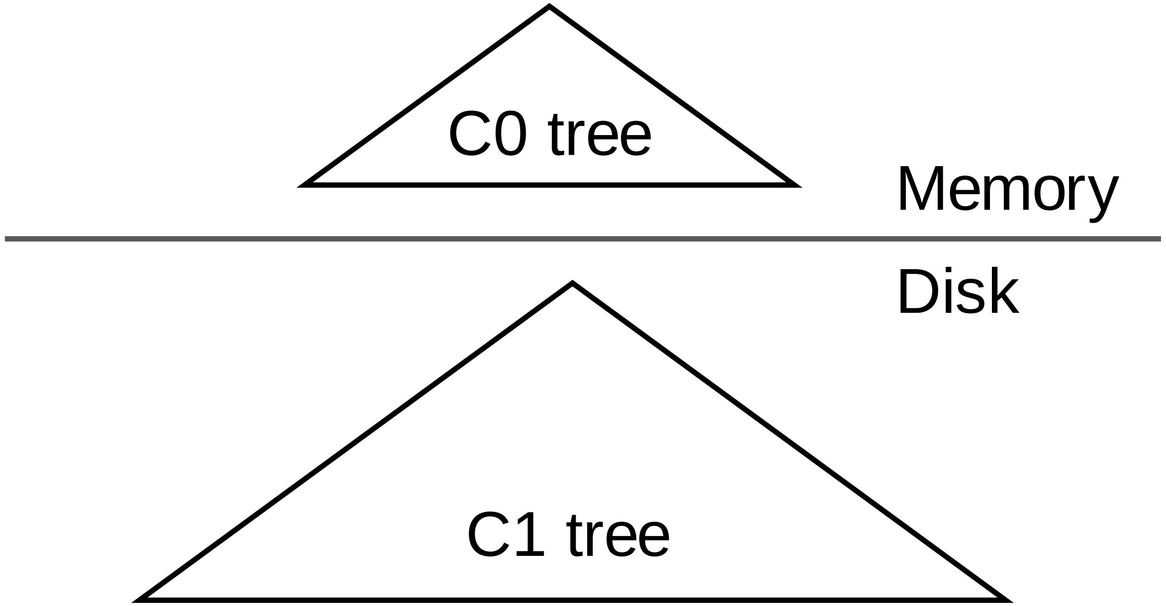
Simple Version: Two-Level LSM Tree



Lookup

- Search the key in both C0 and C1 trees

Simple Version: Two-Level LSM Tree



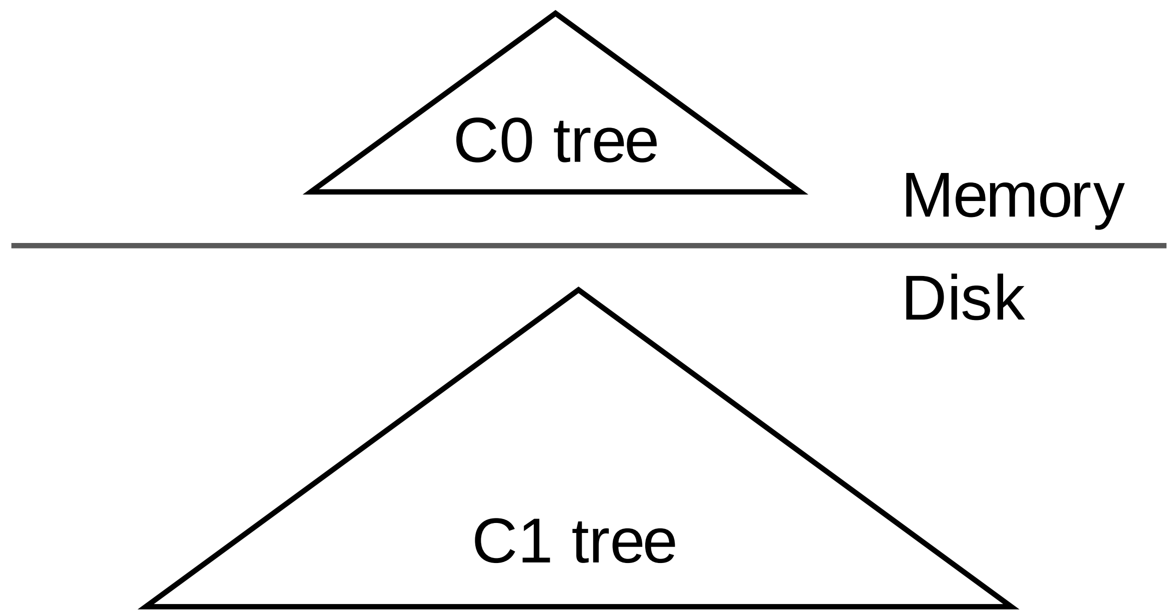
Lookup

– Search the key in both C0 and C1 trees

Insert/Update

– Insert/update the new record into C0 tree (if C0 is not full)

Simple Version: Two-Level LSM Tree



Lookup

- Search the key in both C0 and C1 trees

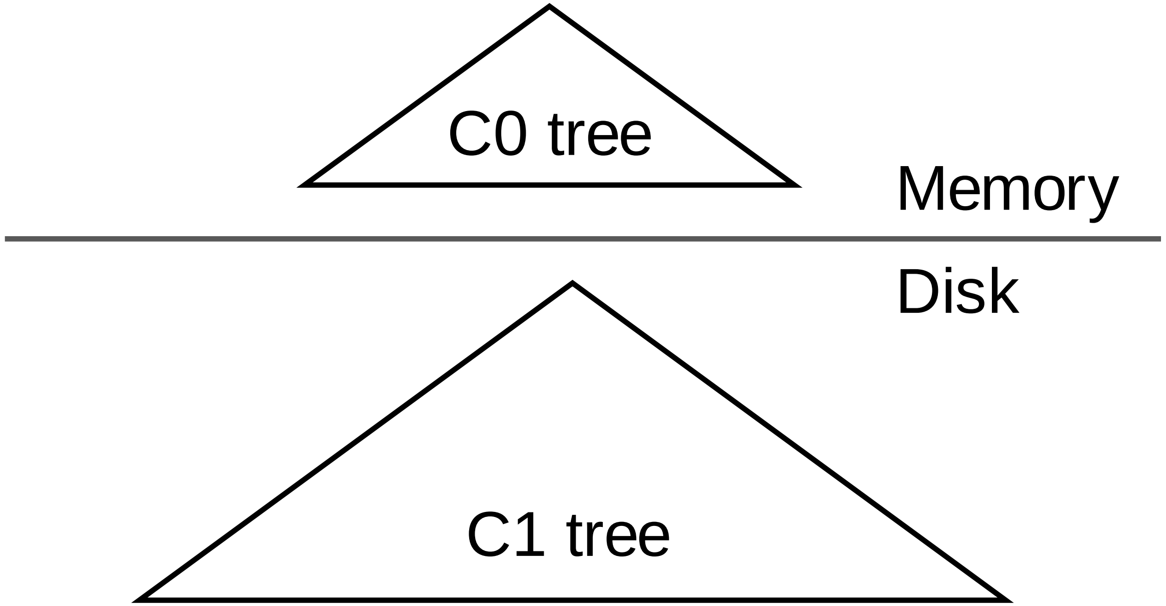
Insert/Update

- Insert/update the new record into C0 tree (if C0 is not full)

Delete

- Insert a tombstone record into C0

Simple Version: Two-Level LSM Tree



Lookup

- Search the key in both C0 and C1 trees

Insert/Update

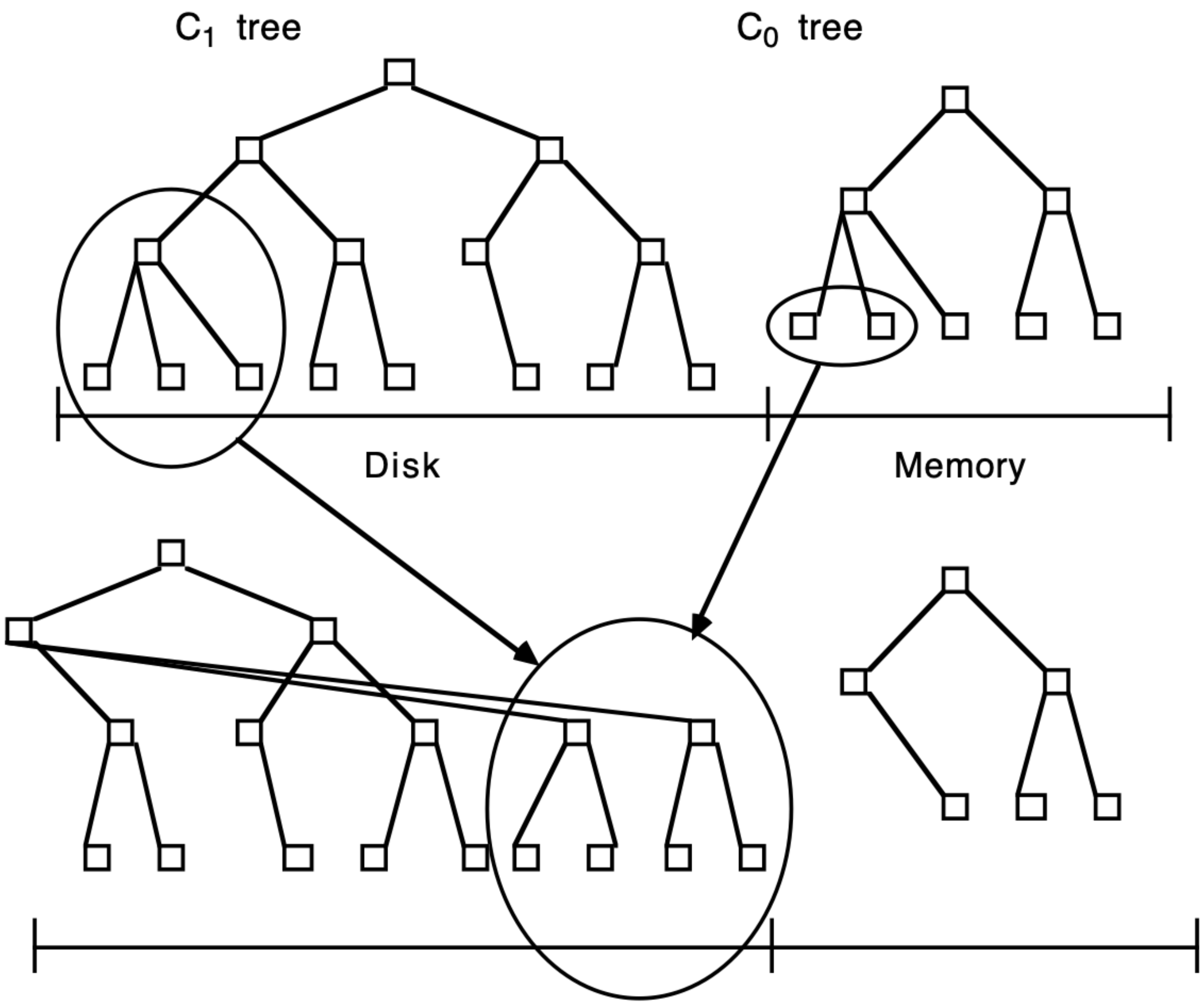
- Insert/update the new record into C0 tree (if C0 is not full)

Delete

- Insert a tombstone record into C0

What if C0 is full? **Rolling merge**

Rolling Merge



Incrementally merge leaves from C1 and C0, and write to disk sequentially

C0 shrinks in size and C1 grows in size

Merge Sort

Array A					
	1	6	8	11	15

Array B					
	2	4	7	10	16

Merge two sorted arrays

1	2	4	6	7	8	10	11	15	16

Merge B+-Trees

Array A

1	6	8	11	15

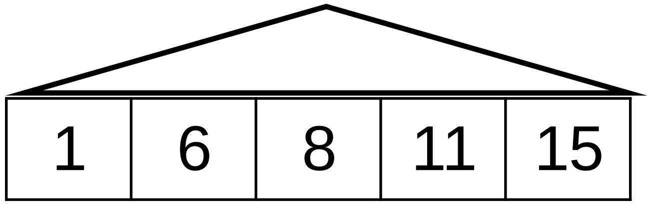
Array B

2	4	7	10	16

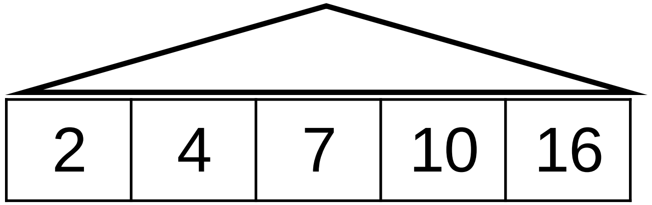
Merge two sorted arrays

1	2	4	6	7	8	10	11	15	16

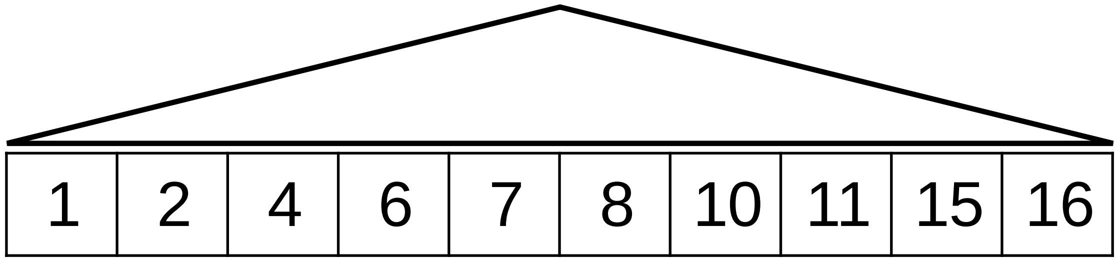
B+-Tree A



B+-Tree B

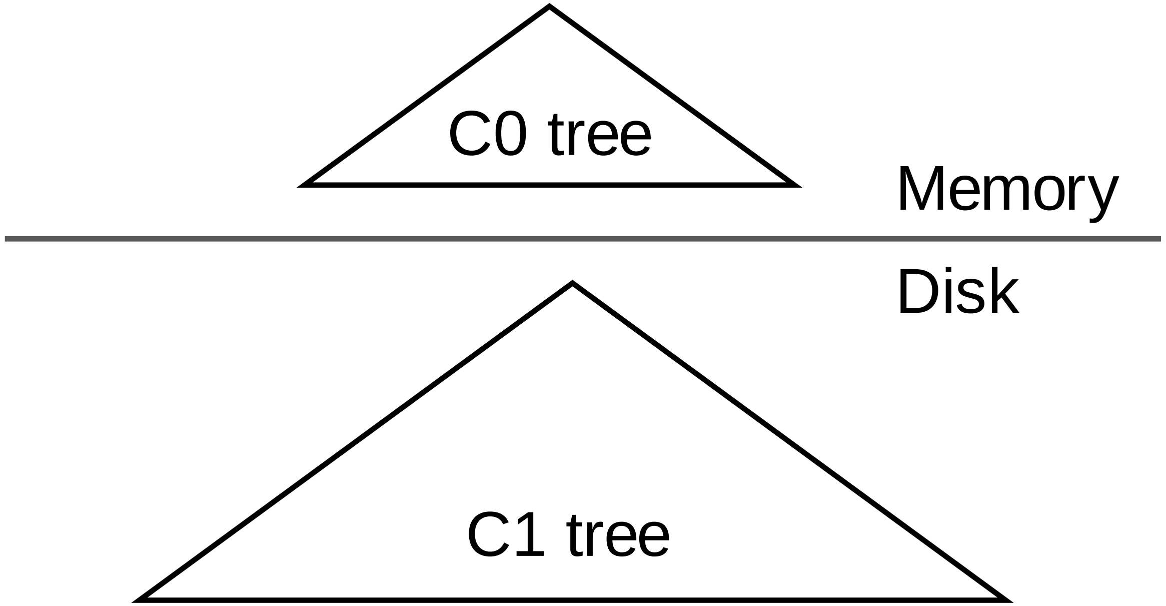


Merge two B+-trees



Merge the leaf nodes and then build the internal nodes

Simple Version: Two-Level LSM Tree



Write IOs to disk are always sequential

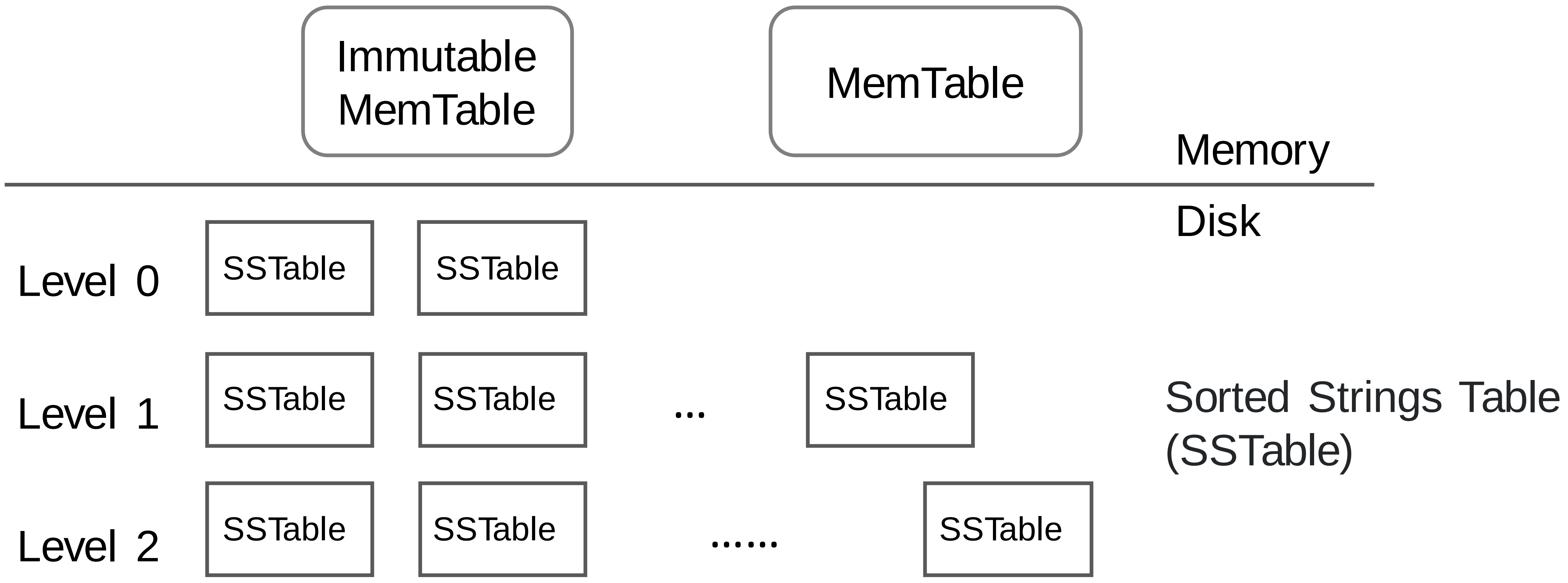
C1 seems no in-place updates

B+-tree nodes in C1 can be 100% full to be more space efficient

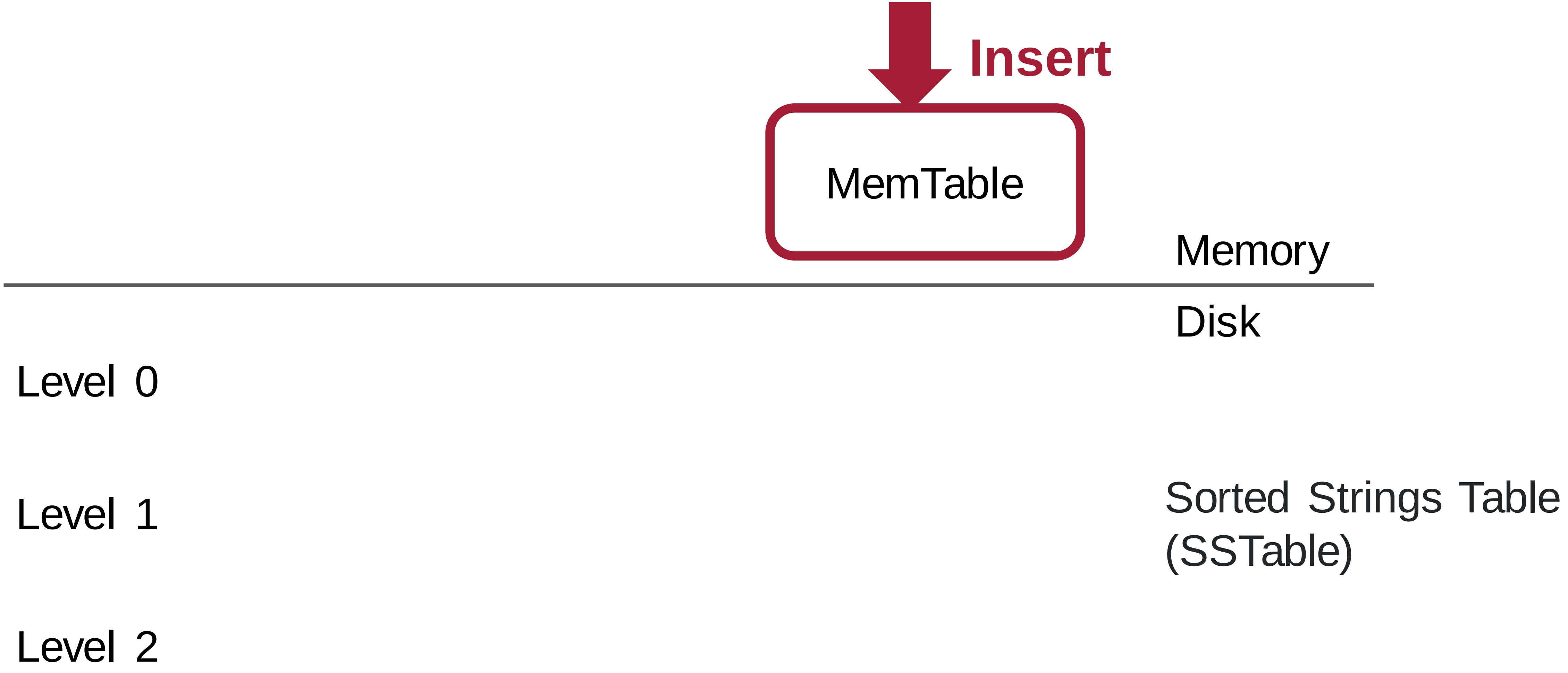
Outline

- Discussion of B+ tree
  - Duplicate search keys
  - Calculating B+ tree parameters
  - Key compression
- Clustered index
- Log-Structured Merge (LSM) tree
  - Sequential vs. Random IO
  - Two-level LSM tree
  - **LSM tree**

LSM Tree Architecture



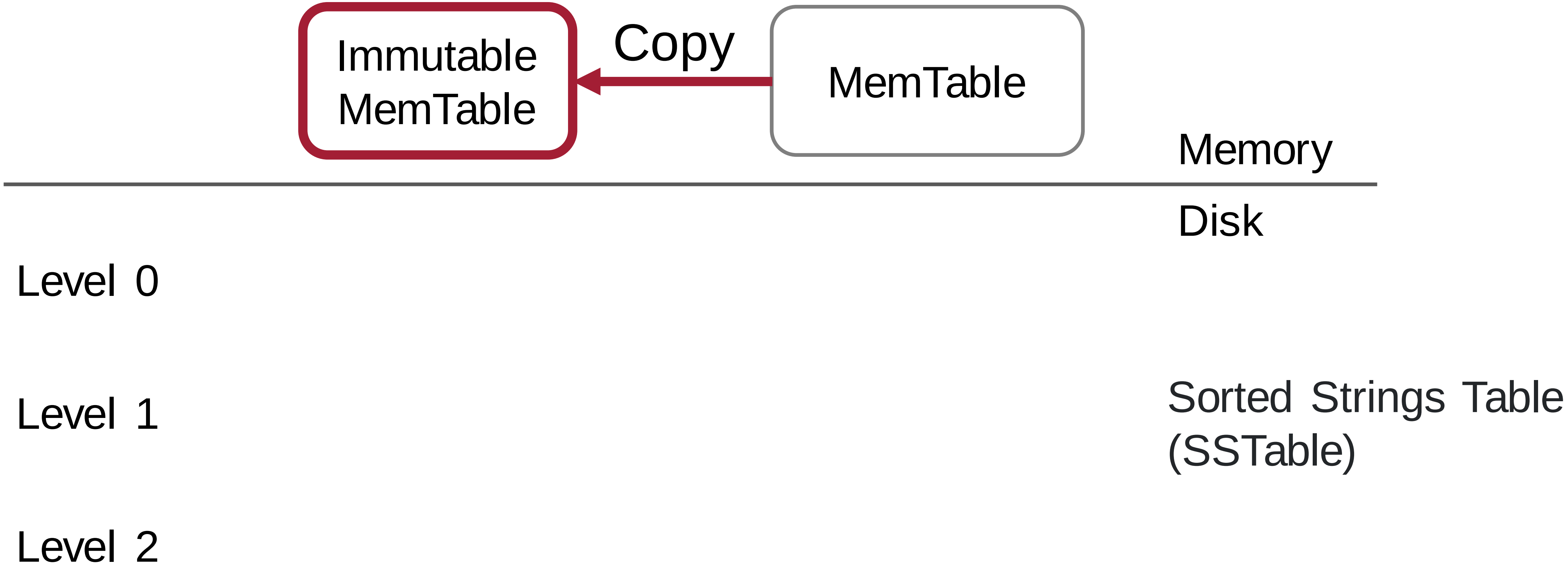
LSM Tree Example



Data insert to MemTable

- MemTable uses a regular index, e.g., B+-tree, Skiplist, etc.

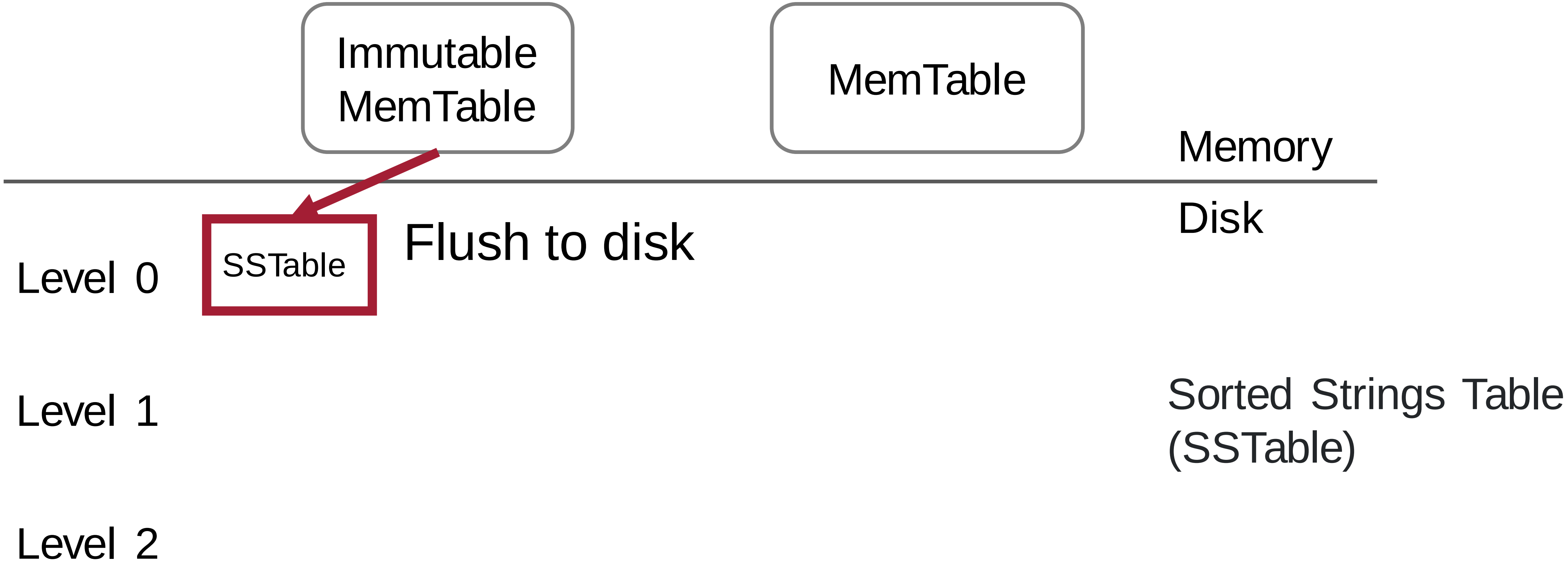
LSM Tree Example



When MemTable is full, convert to Immutable MemTable

- Immutable MemTable is organized as B+Tree

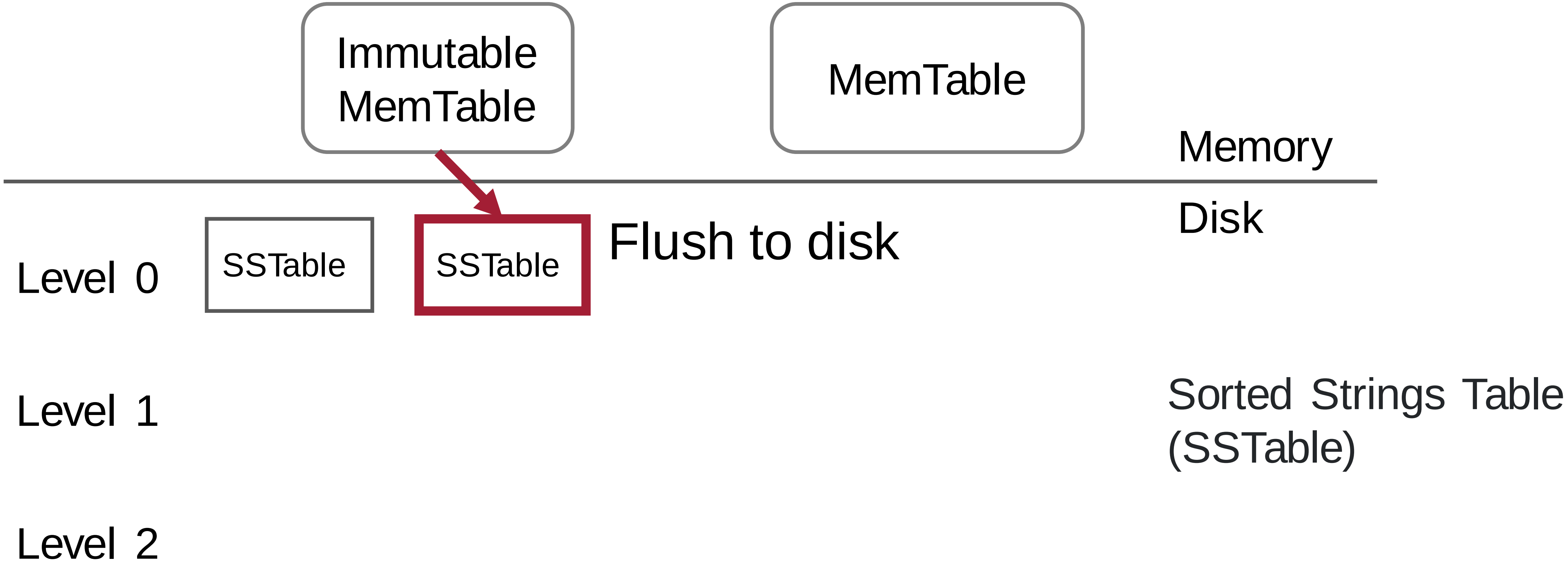
LSM Tree Example



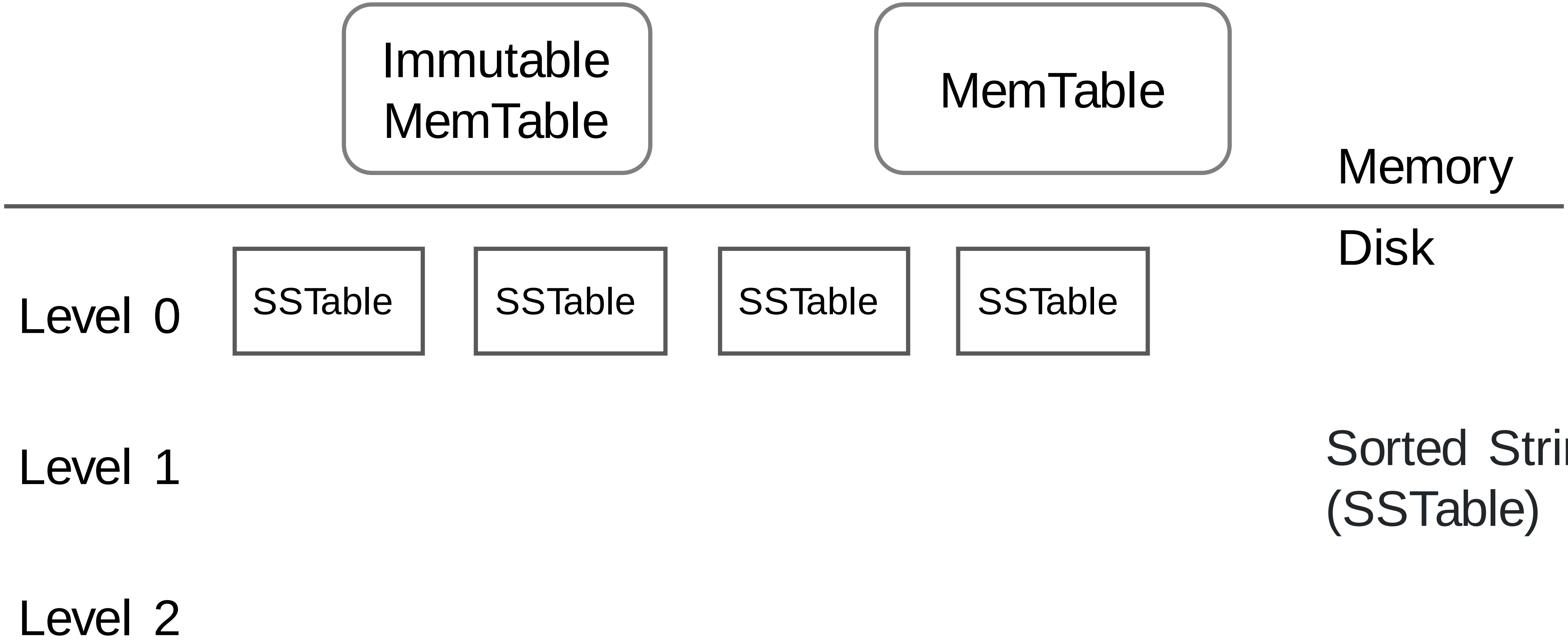
Immutable MemTable are periodically flushed to disk into SSTable

- SSTable is organized as B+ Tree

LSM Tree Example

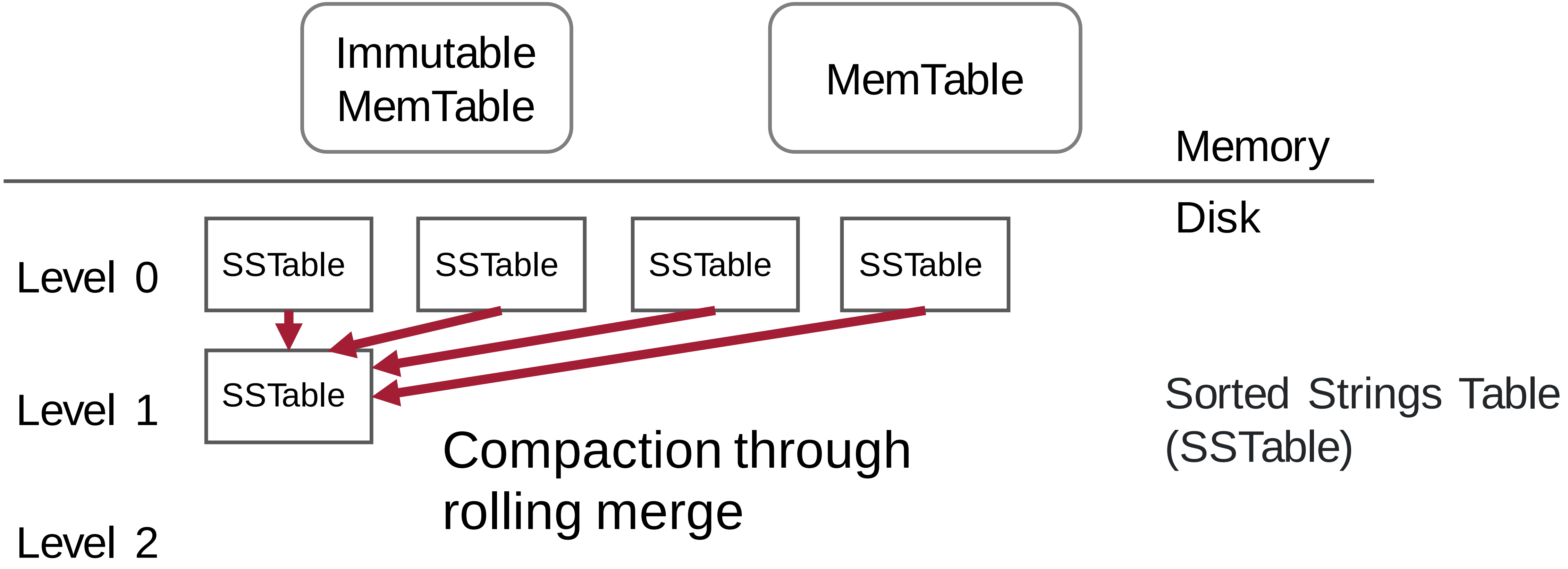


LSM Tree Example



Sorted Strings Table (SSTable)

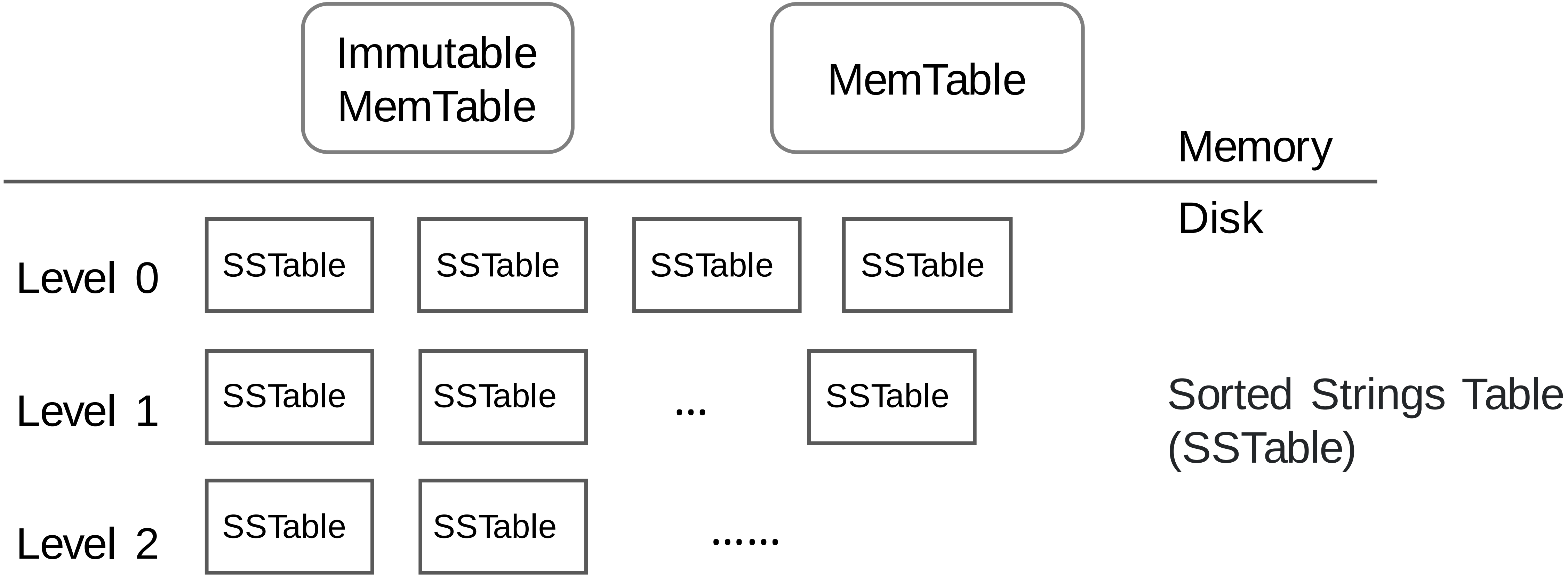
LSM Tree Example



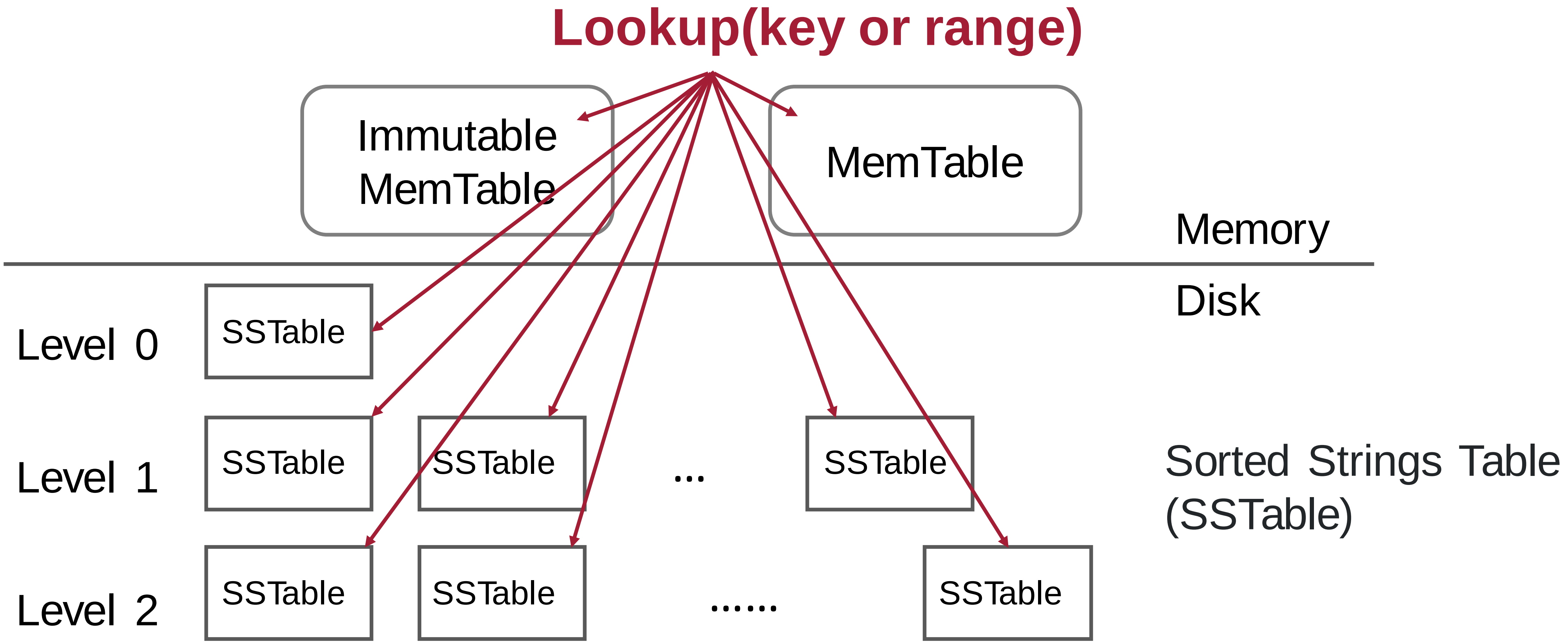
Merge SSTables in Level 0 into bigger SSTable in Level 1

- Duplicates are eliminated. Compaction reduces overall file size

LSM Tree Example



LSM Lookup



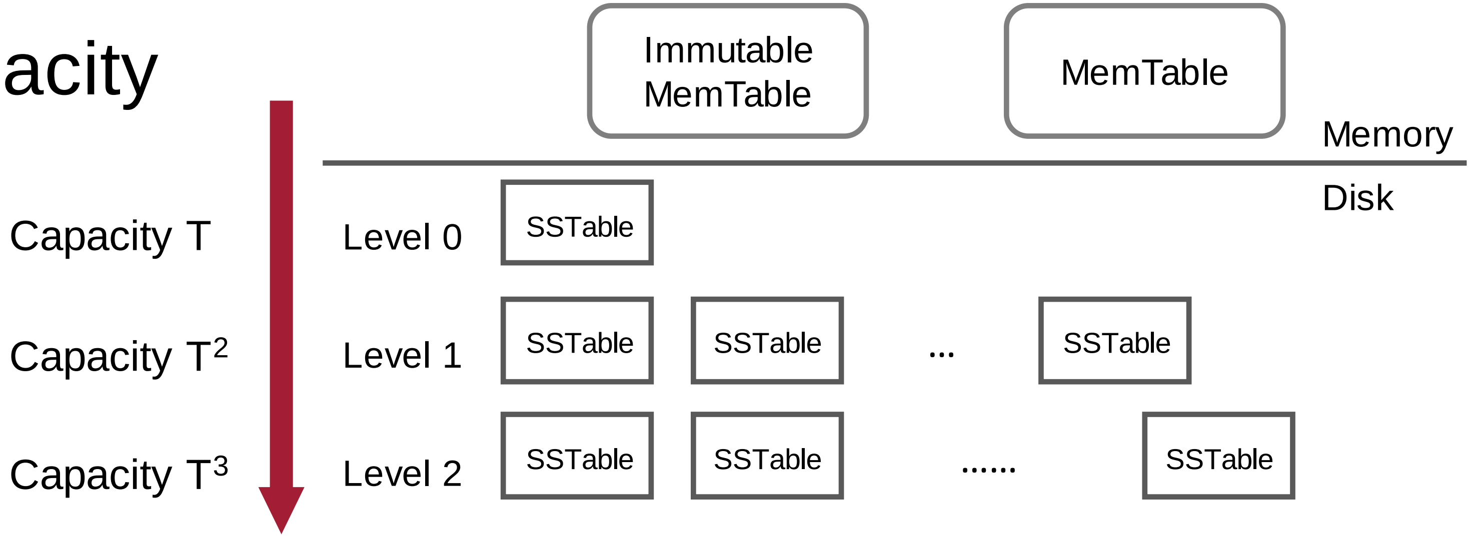
A lookup potentially accesses all tables

– Can stop the search process early if each record has a unique key

LSM Tree Properties

Lower levels have larger capacity

All write IOs are sequential



Read amplification

– A point lookup or scan needs to read multiple tables

Write amplification

– Compaction (merge) also incurs disk IOs (but sequential IOs)

Summary

- Discussion of B+ tree
  - Duplicate search keys
  - Calculating B+ tree parameters
  - Key compression
- Clustered index
- Log-Structured Merge (LSM) tree
  - Sequential vs. Random IO
  - Two-level LSM tree
  - LSM tree