IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

Single-node key-value store/storage

IPADS, Shanghai Jiao Tong University

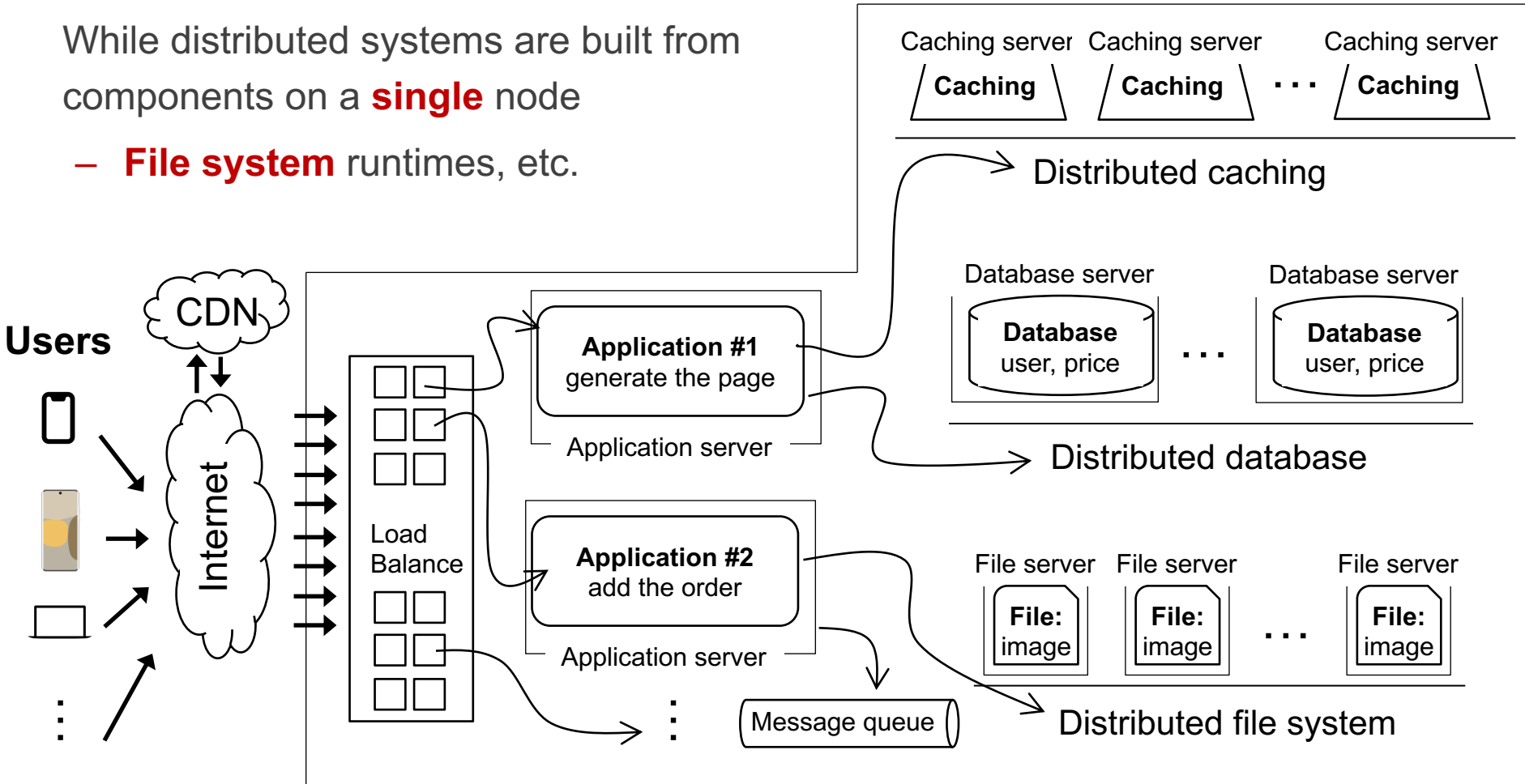
https://www.sjtu.edu.cn

Credits: Niv Dayan , JiFei@IPADS

Review: Large-scale websites on distributed systems

While distributed systems are built from components on a **single** node

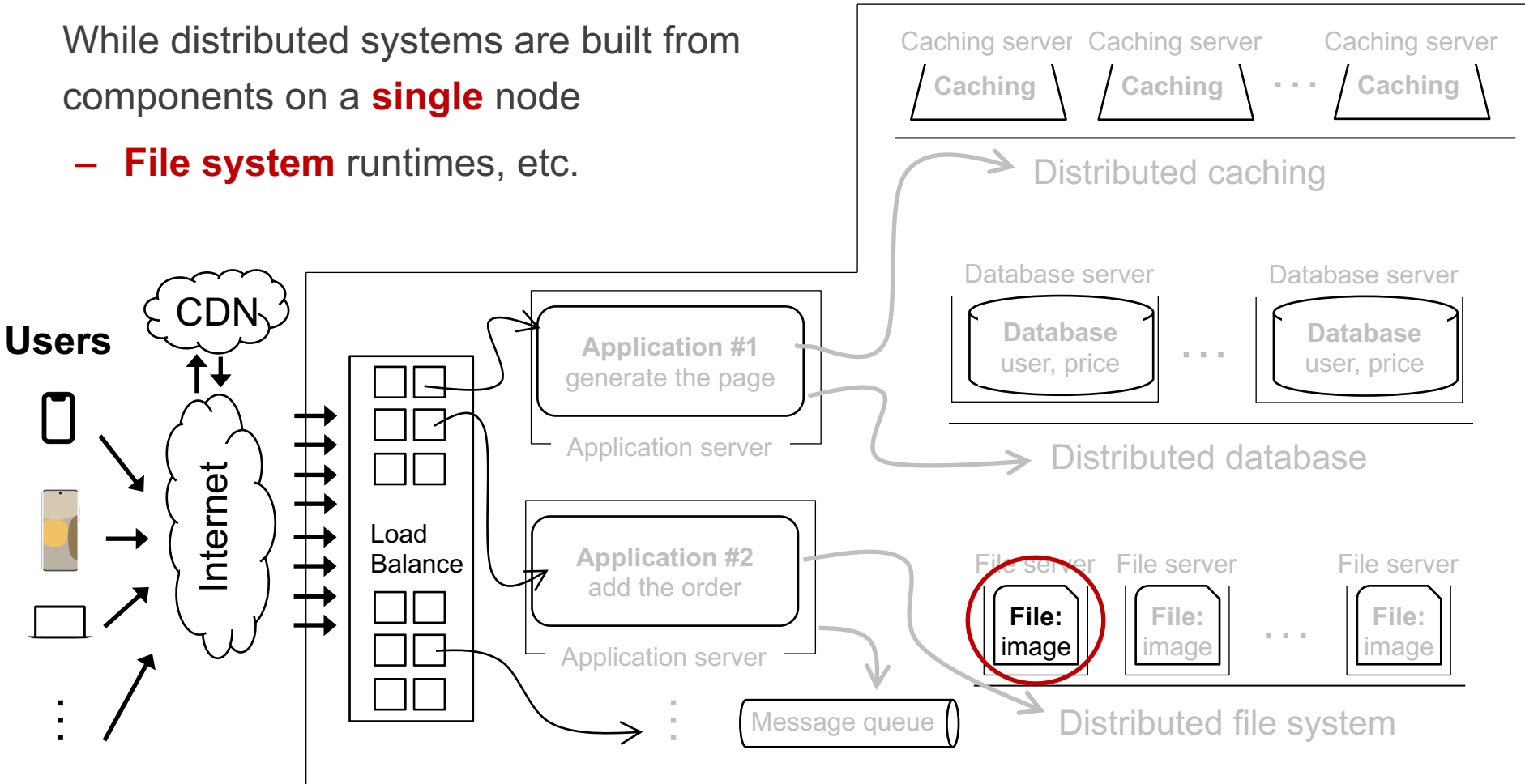
- **File system** runtimes, etc.



Review: Large-scale websites on distributed systems

While distributed systems are built from components on a **single** node

- **File system** runtimes, etc.



Review: The filesystem API

API

- `CHDIR, MKDIR`
- `CREAT, LINK, UNLINK, RENAME`
- `SYMLINK`
- `MOUNT, UNMOUNT`
- `OPEN, READ, WRITE, APPEND, CLOSE`
- `SYNC`

Implemented as system calls to user applications

- Kernel has many sets of function pointers implementing the API
- Each set is specific to a FS (chose at mount point)

Does filesystem sufficient?

Different storage systems to support **different type of data**

不同的存储系统对于存储的数据类型的需求不同，在进行数据访问时的需求也不同，所以不同的数据可能需要不同的存储系统。如下图，图片数据较大，且大部分操作为read，所以使用文件系统较合适，而价格数据可能会被频繁的修改并且会产生新的记录，所以可以使用数据库来存储。而对于访问数量数据，用户并不会关心具体的完整的数量，即对准确性要求不是很高，并且往往会被频繁修改，所以可以使用KV存储。

File system

Database

Key-value store

Key-value storage: simple yet widely used

Storage layout:

透明存储：指的是对于存储系统而言，数据是没有具体含义和类型，只是一段字节码

- Each data (**V**alue) is opaque to the underlying storage/database
  - The K and V can be arbitrary byte-sequence (e.g., JSON, int, string)
- Indexed by a key (**K**), which itself is also a data
- Stored on disk (tolerate failure & support a large capacity)

Interface

- Get(K) -> V, Scan(K,N)
- Update(K,V), Insert(K,V), Delete(K,V)

LEVELDB

RocksDB

redis

Key-value storage is similar to file system

Storage layout:

- Key → The file name
  - Assume the key is not so long
- Value → The file content
- So we can store each K,V as a file ☺

Interface

- Get(K) -> V is similar to OPEN(…) + READ(…)
- Insert(K,V) -> is similar to CREATE(…) + WRITE(…)
- Etc.

Why not implement the key-value storage as file system?

Treating File System as Key-Value storage is inefficient

Redundant system call caused by mismatched interface

- Example
  - E.g., Get(K,V) needs OPEN(…) + READ(…)
  - Ideally, we want to query the data in one system call, better **0**

Space amplification

- File data are stored in fixed-sized blocks (e.g., 512B – 4096B)
- The value in key-value store can be small (e.g., 8B)

KV存储最终没有完全实现为文件系统的原因：
1.文件系统的OPEN READ等操作涉及到大量的syscall以及与磁盘的直接交互，会直接的影响KV系统的性能。
2.文件系统的文件都是分成等大的block进行存储的，对于一些较小的KV对，可能会出现严重的碎片化，导致空间利用率很低。

Treating File System as Key-Value storage is inefficient

Redundant system call caused by mismatch interface

- Example
  - E.g., Get(K,V) needs OPEN(…) + READ(…)
  - Ideally, we want to query the data in one system call, better **0**

Space amplification

- File data are stored in fixed-sized blocks (e.g., 512B – 4096B)
- The value in key-value store can be small (e.g., 8B)

Idea: using one or few file to store key-value store data

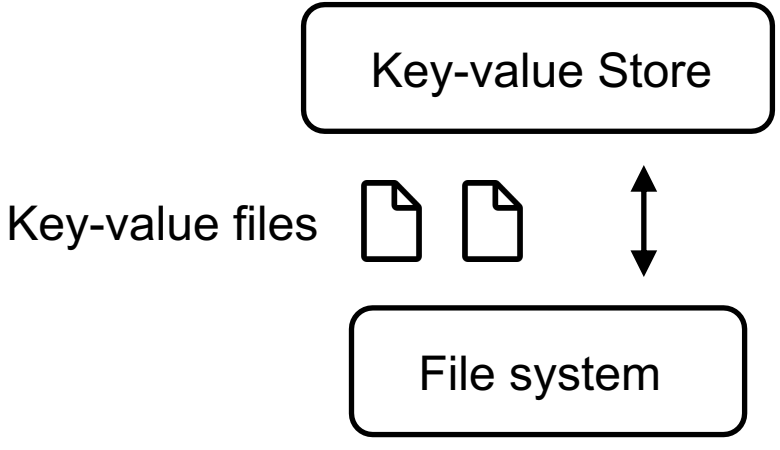
- Different key-values can pack into the same disk block
- Reduce system call overhead: open file once, serve all the requests then

Building Key-value Storage on File System

Why builds upon the file system?

– We still system to manage disk hardware

> ***"All problems in computer science can be solved by another level of indirection"*** -- often attributed to Butler Lampson, who attributes it to David Wheeler



A naïve Key-Value Storage (KVS)

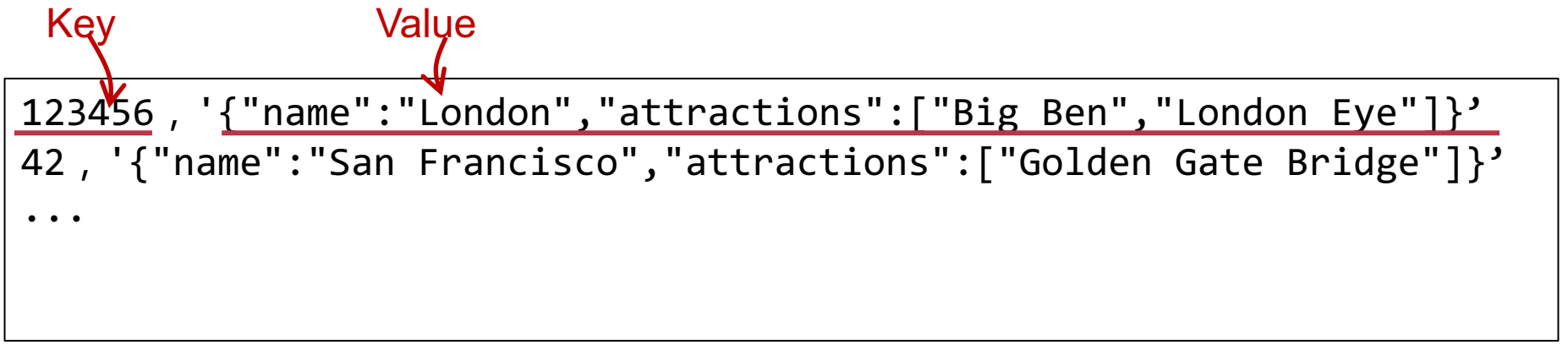
Suppose we store all the KV **in a single file** in the following format:

Key Value

```
123456 , '{"name":"London","attractions":["Big Ben","London Eye"]}’
42 , '{"name":"San Francisco","attractions":["Golden Gate Bridge"]}’
...
```

Update(K,V) in the naïve KVS

Suppose we store all the KV **in a single file** in the following format:



We use a simple update rule:

- Directly **append** update to the **end of the file**

```
procedure UPDATE(integer fd, character[] &key, character[] &value)
```

Update(K,V) in the naïve KVS

Suppose we store all the KV **in a single file** in the following format:

Key Value

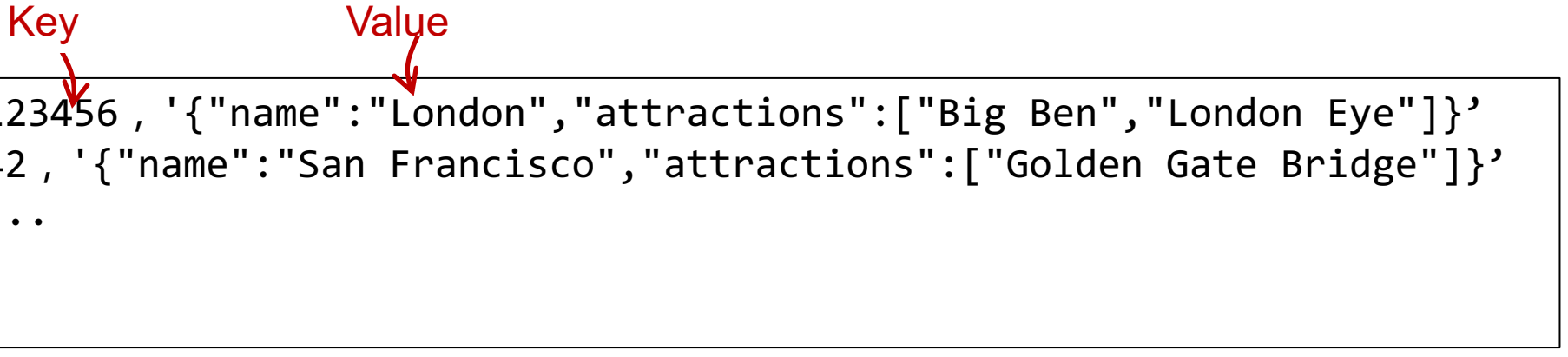
```
123456，'{"name":"London","attractions":["Big Ben","London Eye"]}’
42，'{"name":"San Francisco","attractions":["Golden Gate Bridge"]}’
...
```

We use a simple update rule:

- Directly **append** update to the **end of the file**
- **Benefits:** update the file is efficient, if there are **many updates,** because disks are good at **sequential writes** 因为顺序写时磁盘的arm不需要不停地旋转，只需要停在一个地方一直写，减少了旋转寻址的时间，所以比random write要快。
- **Overwrite the original content** introduce **slow random accesses**

Insert(K,V) in the naïve KVS

Suppose we store all the KV **in a single file** in the following format:



Insert is the same as update

- Question: what about deletion?
  - Append a NULL entry and later garbage collected (will talk later)

BTW: File format of naïve KVS & log file

log-structured file：指的是类似于LOG一样每次都将新的记录添加到文件末尾的文件。

The file format is some kind of log file

- We called **Log-structured file**: an **append-only** sequence of records

Widely used in computer systems in various scenarios

- Suits the performance of underlying hardware (e.g., disk)
- E.g., ensure consistency under failure, etc
- We will see more examples in later lectures

Hardware speed matters!
In our case: disk.

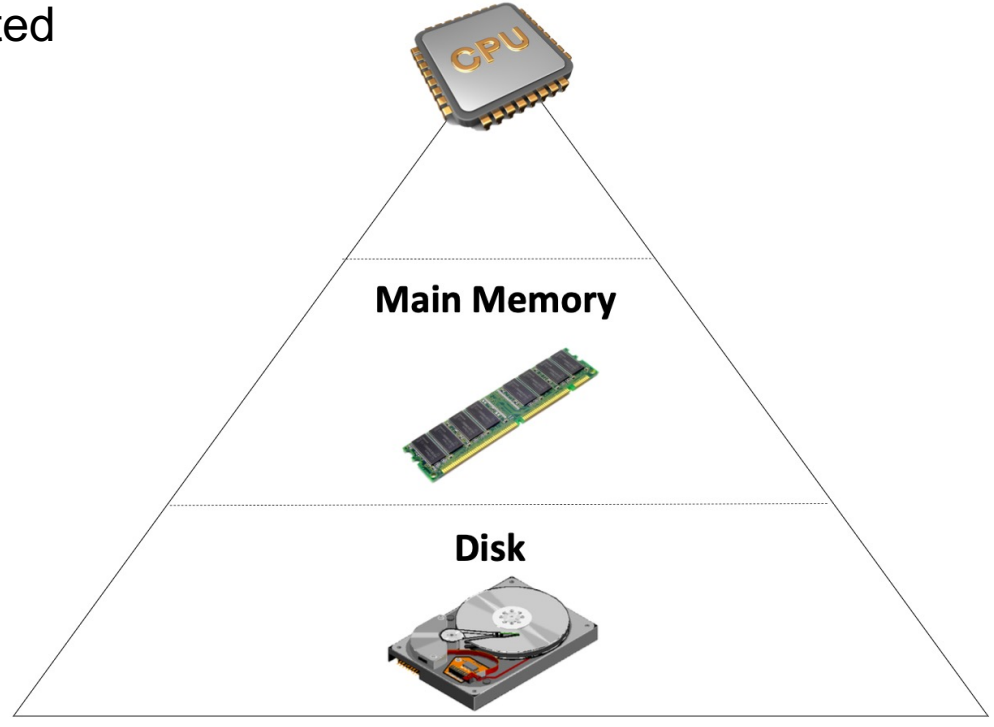
Storage Hierarchy

Fast device (e.g., Main Memory)

- Fast, but capacity is limited
- Expensive
- Volatile

Disk has huge capacity

- Cheap & slow
- Non-volatile



Storage Hierarchy: asymmetric performance of disk

CPU <-> Memory

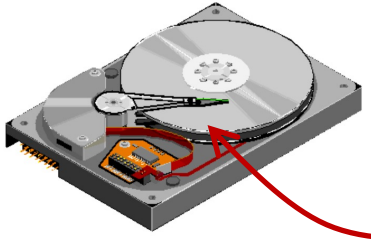
- ~= 100ns

Memory <-> Disk

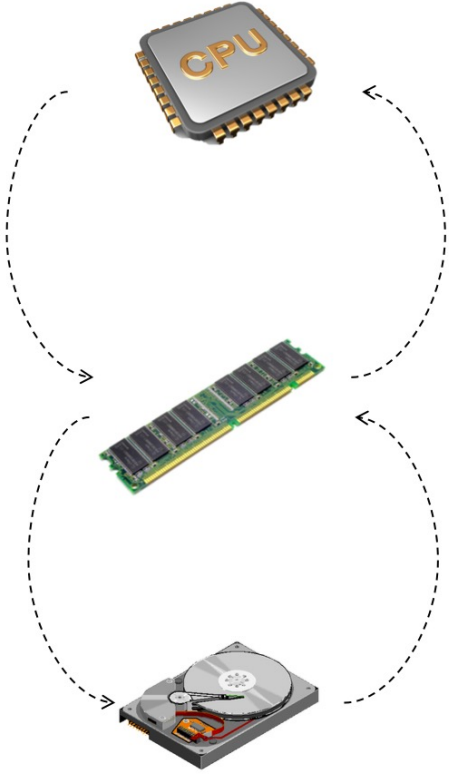
- ~= 10ms

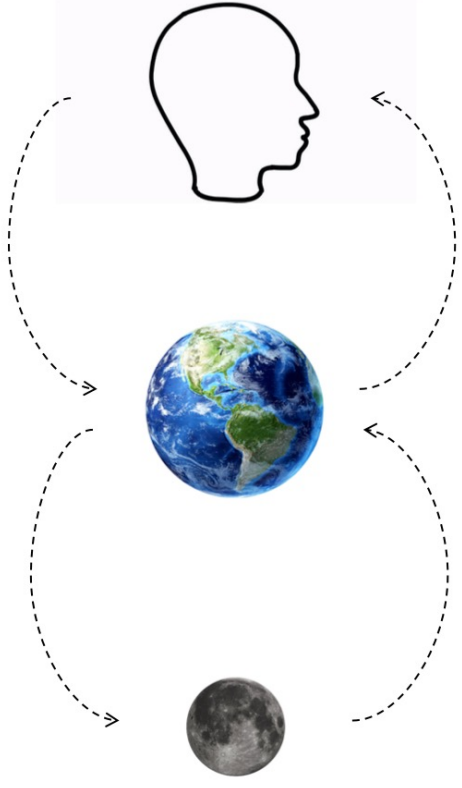
Why disk is slow at random access?

- **Random**: move disk header
- **Sequential**: disk just spins



Disk Header





Updates work Well, What about Get(K,V)?

Suppose we store all the KV **in a single file** in the following format:

Key Value

```
123456  '{"name":"London","attractions":["Big Ben","London Eye"]}’
42 '{"name":"San Francisco","attractions":["Golden Gate Bridge"]}’
...
```

Get is much tricky to implement

- We need to iterate the file line by line **starting at the end of file**, and then finds the first line whose key matches K
- **Terrible performance**! Complexity: O(n)

此时只能迭代！因为在这种实现下，不遍历到最后不知道当前独到的KV对是否是最新的一条。

Accelerate Get with index

Index

- An **additional structure** that is derived from the primary data (e.g., log file)
- Can be added/removed without affecting the primary data
  - Only affects the **performance** of interface (e.g., Get)

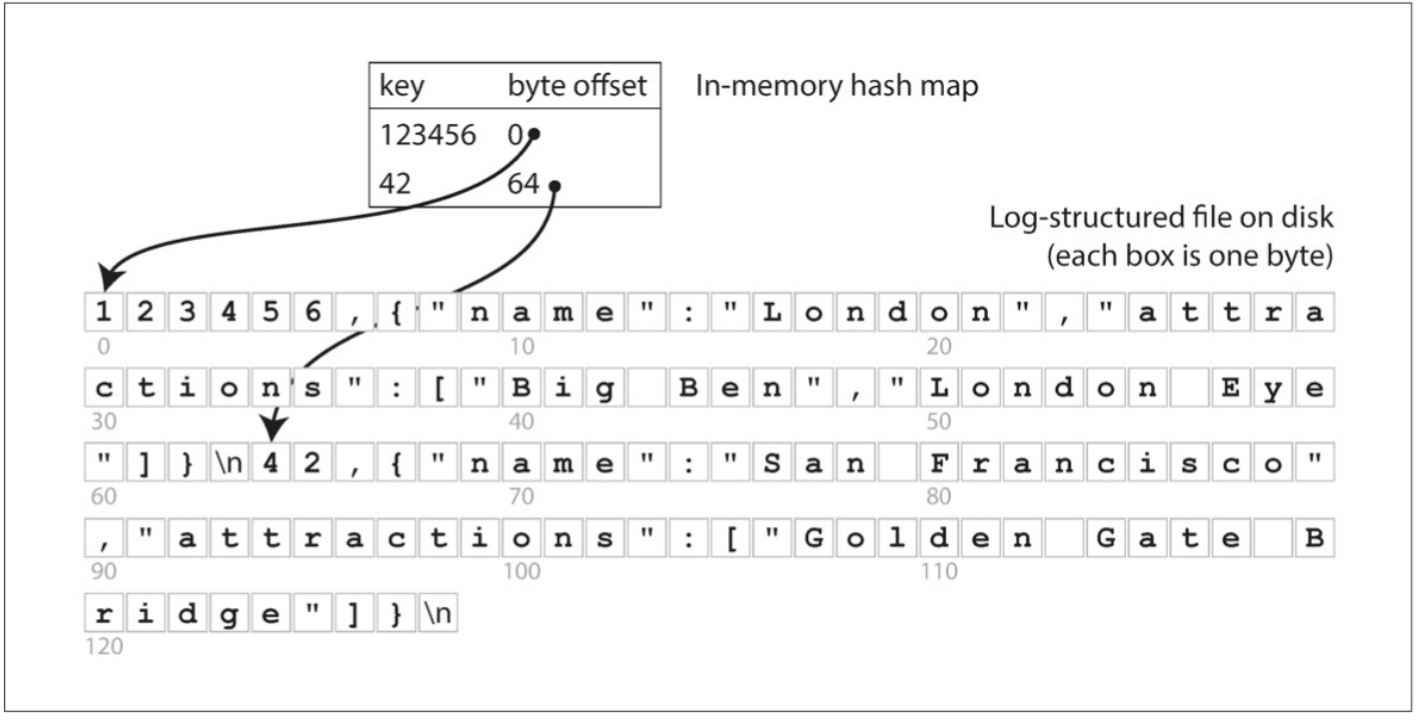
Example:

- B+Tree, HashTable, etc
- **Question**: how do you implement the lookup in the Lab1? Can we accelerate it with index?

In-memory Hash Index

The simplest possible indexing strategy

– Keep an in-memory hash map that map key -> byte offset in the log file



Hash index & log-structured data file

The simplest possible indexing strategy

- Keep an in-memory hash map that map key -> byte offset in the log file

A viable solution

- Adopted by commercial systems， e.g., Bitcask in **riak**KV 

Benefits

- High-performance of reads/writes

Restrictions?

Hash index & log-structured data file

The simplest possible indexing strategy

- Keep an in-memory hash map that map key -> byte offset in the log file

A viable solution

- Adopted by commercial systems， e.g., Bitcask in riakKV

优点：读写快，读快是因为用offset直接跳过不必要的数据，write快是read快，然后直接在read到的V上修改就行了。
缺点：in-memory，当KV对过多时可能会超出内存容量而崩溃
所以上述的HASH Index更加是用于读取多插入少的情况。

Benefits

- High-performance of reads/writes

Restrictions

- Index must **fit into the RAM**
- Only good at: when workloads have many updates but not insertions
  - Too many insertions will exhaust the RAM

Put hash index on the disk

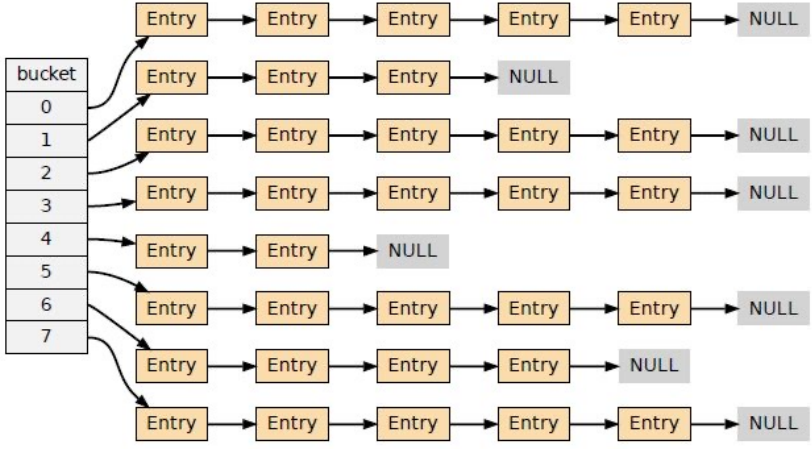
Usually not efficient

Hash indexes have different **performance trade-offs** when stored on disk

- Linked-list based hash index
- Cuckoo hashing
- Etc.

Linked-list based hash index

即使用散列表法进行重新散列



Pros:

– Easy for implementation

Cons:

– Bad read performance under hash collision

(因为链表相邻节点的地址是不连续的)

  • Requires traverse the linked buckets in random I/O

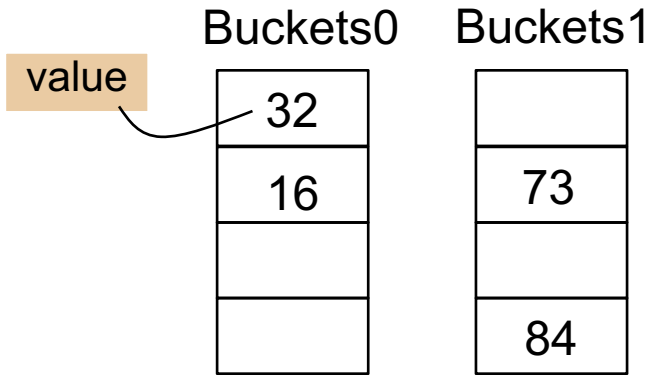
– Bad insert performance under hash collision

  • Similar problem as the read

Cuckoo hashing

Cuckoo hashing

- Maintain two hash buckets



Cuckoo hashing

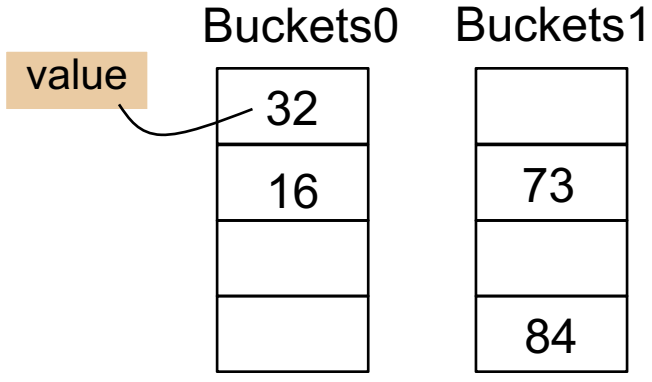
Cuckoo hashing

– Maintain two hash buckets

Get()

– Using two hash functions H0 & H1

– The **key** must be in **`Buckets0[H0(key)]`** or **`Buckets1[H1(key)]`**



Cuckoo hashing

Cuckoo hashing

– Maintain two hash buckets

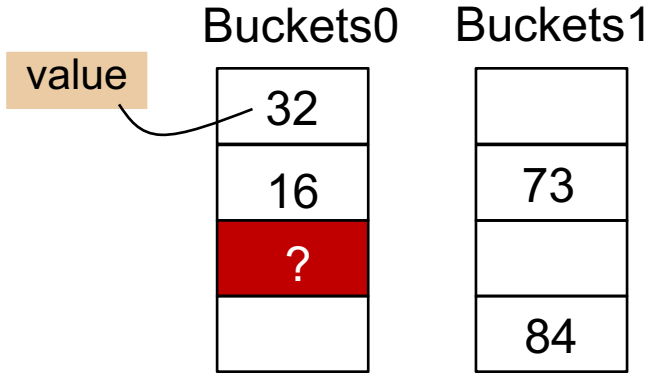
Get()

– Using two hash functions H0 & H1

– The **key** must be in **`Buckets0[H0(key)]`** or **`Buckets1[H1(key)]`**

Example: `Get(73)`

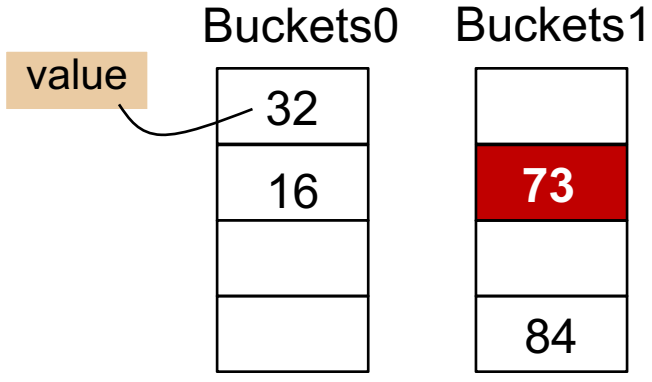
– **`Buckets0[H0(73)] = 2`**



Cuckoo hashing

Cuckoo hashing

- Maintain two hash buckets



Get()

- Using two hash functions H0 & H1
- The **key** must be in **`Buckets0[H0(key)]`** or **`Buckets1[H1(key)]`**

Example: `Get(73)`

- **`Buckets0[H0(73)] = 2 & Buckets1[H1(k73)] = 1`**

Cuckoo hashing

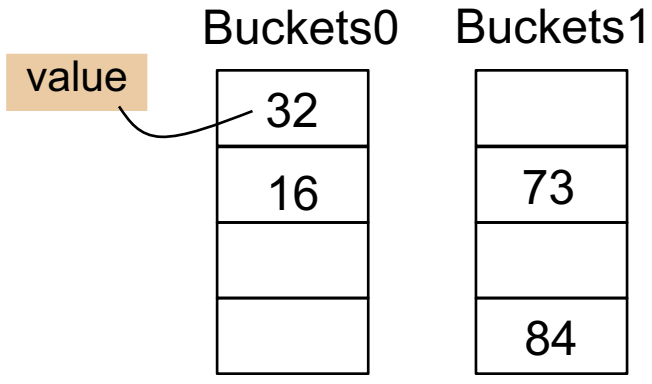
Cuckoo hashing

– Maintain two hash buckets

Update()

– Similar as `Get()`

- 1. find the bucket
- 2. update the value



What about insert?

Cuckoo hashing

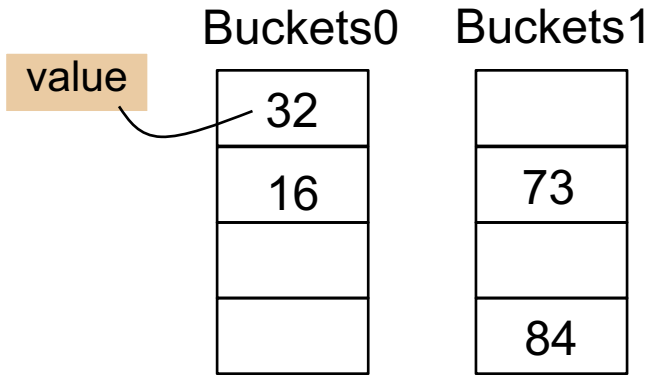
Cuckoo hashing

– Maintain two hash buckets

Insert()

– Much complicated: should insert to `Buckets0[H0(key)]` or `Buckets1[H1(key)]`

  • Question: what if no available space?



Cuckoo hashing

Cuckoo hashing

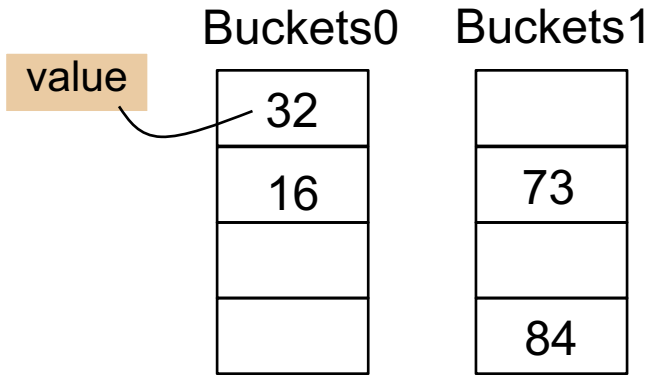
– Maintain two hash buckets

Insert()

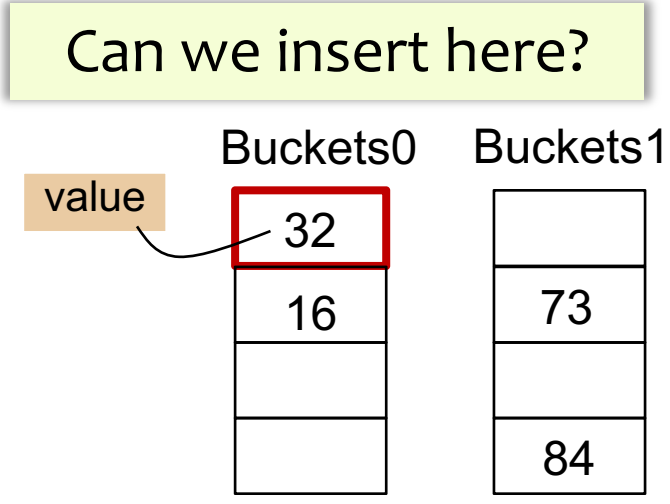
– Much complicated: should insert to `Buckets0[H0(key)]` or `Buckets1[H1(key)]`

Example: Insert(2)

– `Buckets0[H0(2)] = 0 & Buckets1[H1(2)] = 0`



Cuckoo hashing



Cuckoo hashing

- Maintain two hash buckets

Insert()

- Much complicated: should insert to `Buckets0[H0(key)]` or `Buckets1[H1(key)]`

Example: Insert(2)

- `Buckets0[H0(2)] = 0 & Buckets1[H1(2)] = 0`

Cuckoo hashing

Cuckoo hashing

– Maintain two hash buckets

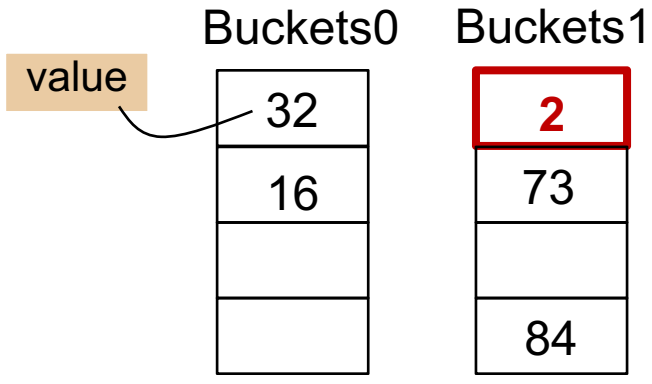
Insert()

– Much complicated: should insert to `Buckets0[H0(key)]` or `Buckets1[H1(key)]`

Example: Insert(2)

– `Buckets0[H0(2)] = 0 & Buckets1[H1(2)] = 0`

– Insert here!



Cuckoo hashing

Cuckoo hashing

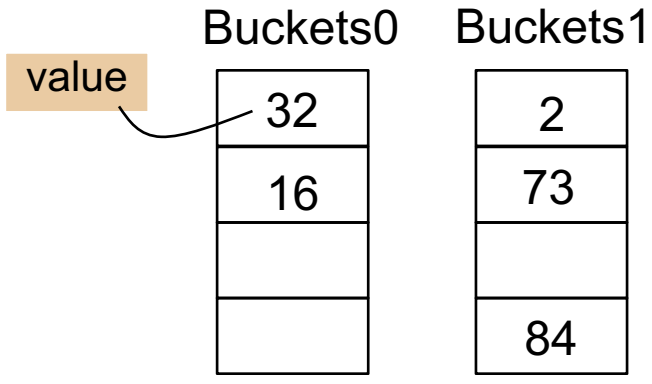
– Maintain two hash buckets

Insert()

– Much complicated: should insert to `Buckets0[H0(key)]` or `Buckets1[H1(key)]`

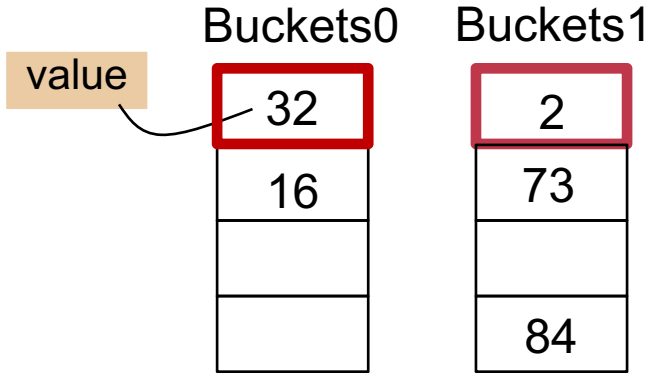
Example: Insert(9)

– `Buckets0[H0(9)] = 0 & Buckets1[H1(9)] = 0`



Cuckoo hashing

Both slots are not empty!



Cuckoo hashing

- Maintain two hash buckets

Insert()

- Much complicated: should insert to `Buckets0[H0(key)]` or `Buckets1[H1(key)]`

Example: Insert(9)

- `Buckets0[H0(9)] = 0 & Buckets1[H1(9)] = 0`

Cuckoo hashing

Cuckoo hashing

– Maintain two hash buckets

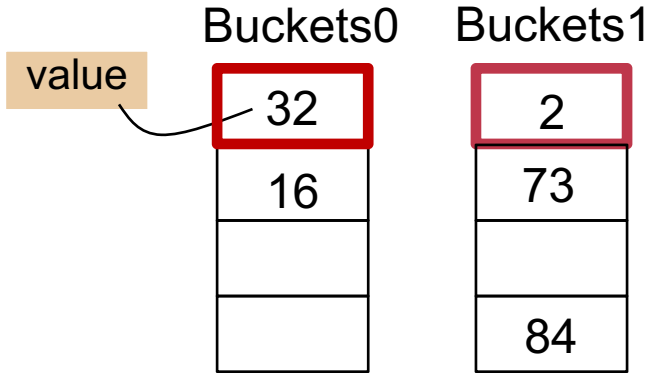
Insert()

– Much complicated: should insert to `Buckets0[H0(key)]` or `Buckets1[H1(key)]`

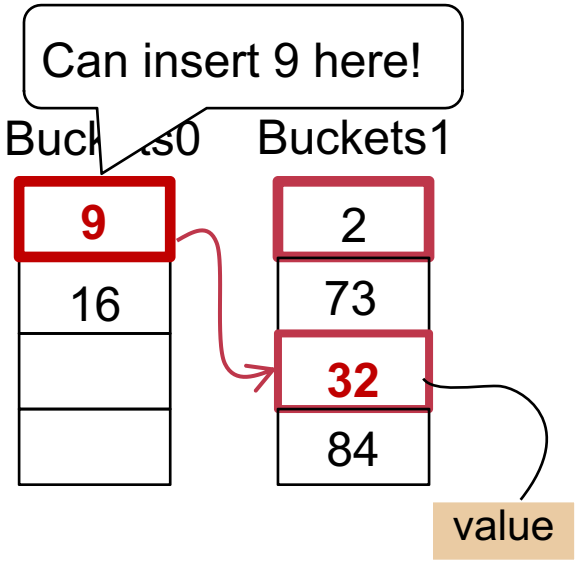
Example: Insert(9)

– `Buckets0[H0(9)] = 0 & Buckets1[H1(9)] = 0`

– Cuckoo hashing will kick others to different place to create space for the new insertions



Cuckoo hashing



Cuckoo hashing

- Maintain two hash buckets

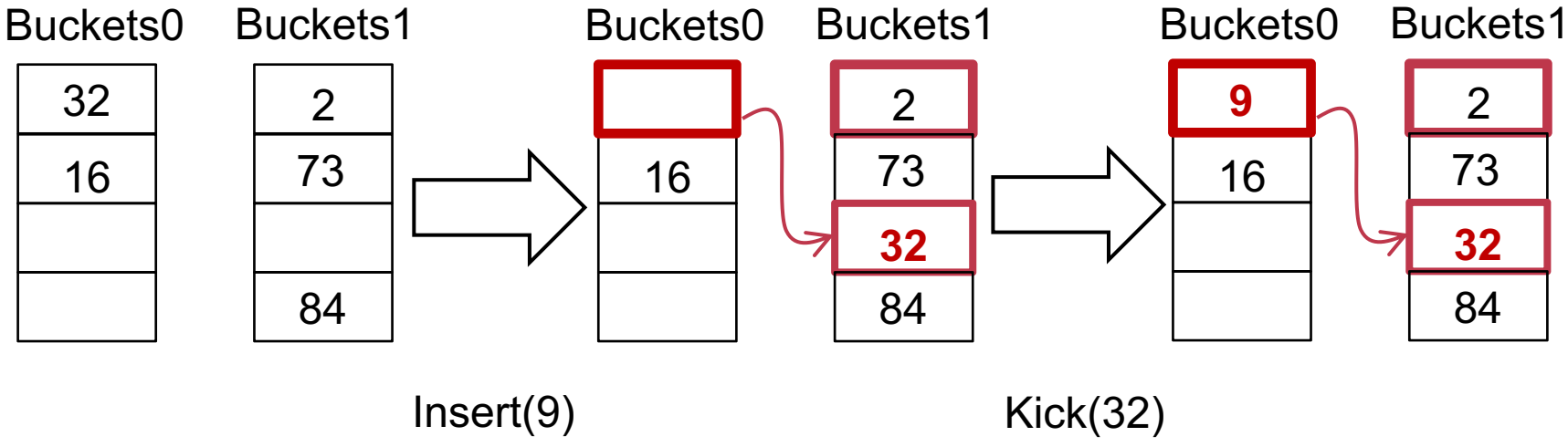
Insert()

- Much complicated: should insert to `Buckets0[H0(key)]` or `Buckets1[H1(key)]`

Example: Insert(9)

- `Buckets0[H0(9)] = 0 & Buckets1[H1(9)] = 0`
- Cuckoo hashing will kick others to different place to create space for the new insertions
  - Kick 32 (Buckets0[0]) to another place
  - Assume: `H1(32) = 2`

Cuckoo hashing




```
Buckets0[H0(9)] = 0
& Buckets1[H1(9)] = 0
```

```
H1(32) = 2
```

Question: what if H1(32) = 3? (which is taken by 84)

Cuckoo hashing

Cuckoo hashing

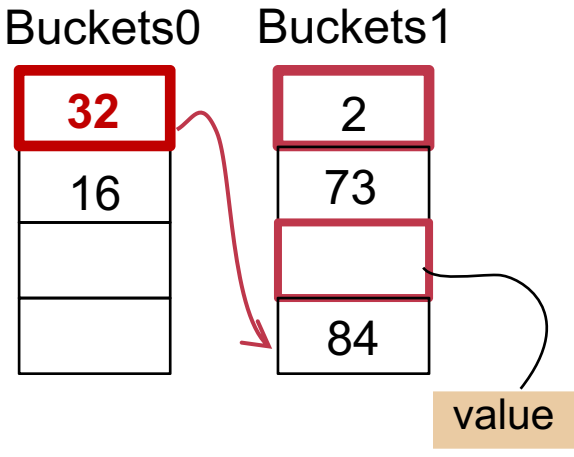
- Maintain two hash buckets

Insert()

- Much complicated: should insert to `Buckets0[H0(key)]` or `Buckets1[H1(key)]`

Example: Insert(9)

- `Buckets0[H0(9)] = 0 & Buckets1[H1(9)] = 0`
- Cuckoo hashing will kick others to different place to empty
  - Kick 32 (Buckets0[0]) to another place
  - **Question: what if `H1(32)` ~~= 2~~ = 3?**



Cuckoo hashing

Cuckoo hashing

– Maintain two hash buckets

Insert()

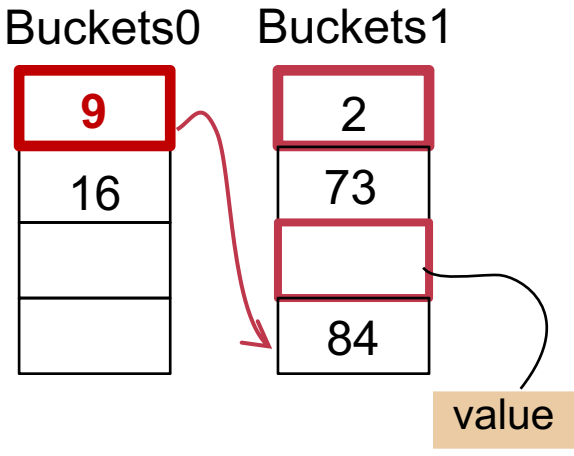
– Much complicated: should insert to `Buckets0[H0(key)]` or `Buckets1[H1(key)]`

Example: Insert(9)

– `Buckets0[H0(9)] = 0 & Buckets1[H1(9)] = 0`

– Cuckoo hashing will kick others to different place to empty

- ~~Kick 32 (Buckets0[0]) to another place~~
- Kick 2 (Buckets1[0]) to another place



Cuckoo hashing

Cuckoo hashing

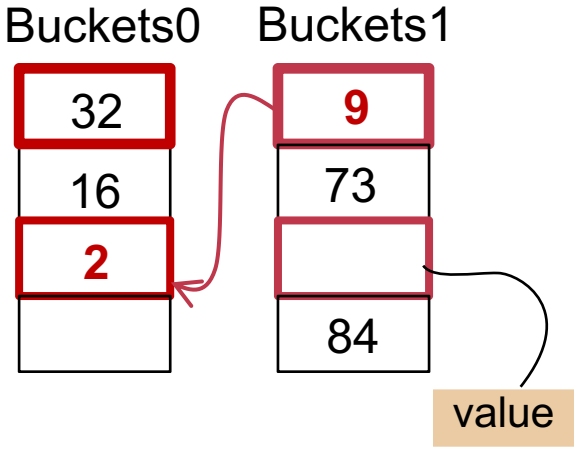
- Maintain two hash buckets

Insert()

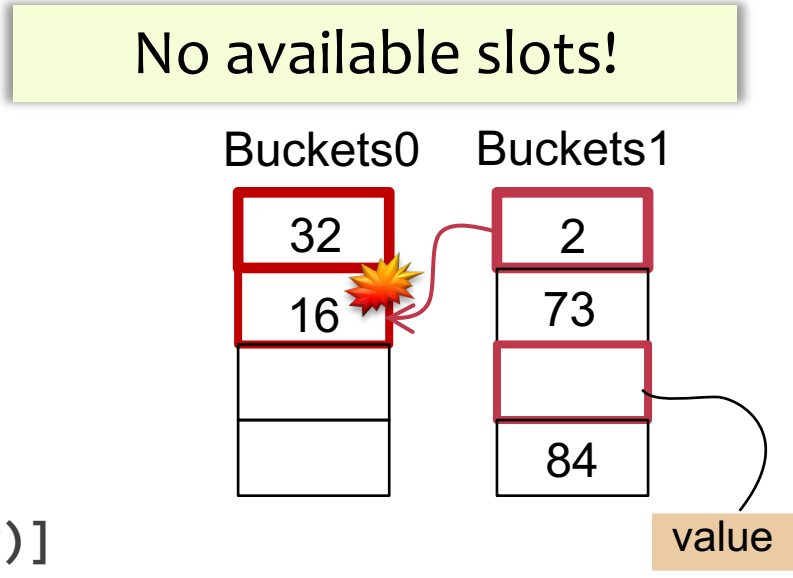
- Much complicated: should insert to `Buckets0[H0(key)]` or `Buckets1[H1(key)]`

Example: Insert(9)

- `Buckets0[H0(9)] = 0 & Buckets1[H1(9)] = 0`
- Cuckoo hashing will kick others to different place to empty
  - Kick 2 (Buckets1[0]) to another place
  - Assume: `H0(2) = 2`



Cuckoo hashing



Cuckoo hashing

- Maintain two hash buckets

Insert()

- Much complicated: should insert to `Buckets0[H0(key)]` or `Buckets1[H1(key)]`

Example: Insert(9)

- `Buckets0[H0(9)] = 0 & Buckets1[H1(9)] = 0`
- Cuckoo hashing will kick others to different place to empty
  - Kick 2 (Buckets1[0]) to another place
  - Assume: `H0(2) = ~~2~~ 1` What if that happens?

Cuckoo hashing

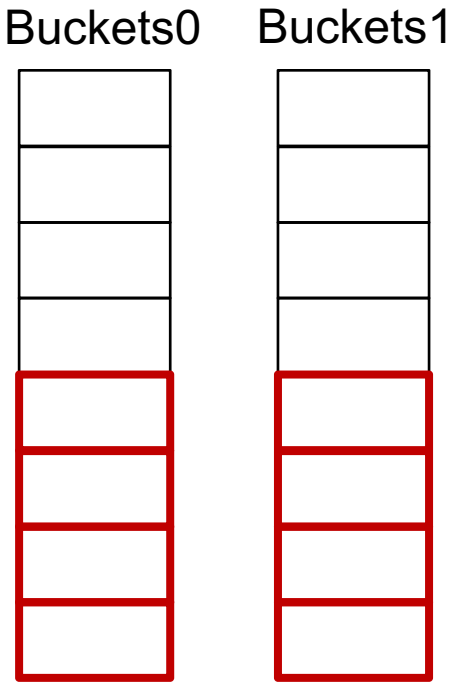
Cuckoo hashing

- Maintain two hash buckets

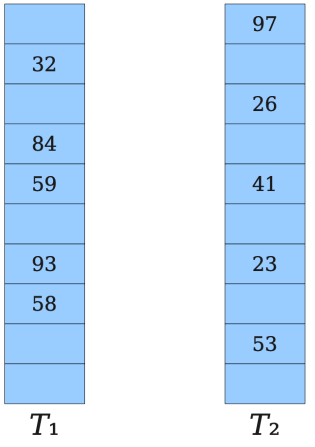
Insert()

- Much complicated: should insert to `Buckets0[H0(key)]` or `Buckets1[H1(key)]`
- If no available slots, rehashing & repeat the above insert process
- Rehashing: enlarge the buckets (typically **X2**)

rehash之后还需要对原有数据进行一次重新得插入操作，因为哈希函数一般都会和hash Table的size有一定关系。



Cuckoo hashing



Cuckoo hashing

- Two tables & Two hash functions (H_1, H_2)
- The key can only at $T_1[H_2(key)]$ or $T_2[H_2(key)]$

Pros

- Good for read: at most two 2 random I/O for get

Cons

- Insertion is complex and can issue many random disk I/O
- **Rehashing** is required if there are many insertions, which is extremely **slow**
  - Expanding the table, and re-insert all the keys in the new table (!)

Selecting a proper index for KV store is non-trivial

Not just choosing a data structure

Trade-offs of selecting indexes （ The RUM Conjecture ）

- **Read** performance vs. **update** performance
  - An index needs to be updated upon inserts/deletions
- Whether index **is friendly to** the storage?
  - E.g., random vs. sequential
- Others (e.g., storage overhead)

Issues of using hash index & log file

Index limited by the RAM size

- Can put it on the disk but is slow. Why?

Log file is growing forever -> running out of disk space

Lack range query support

- E.g., scan over all keys between kitty00000 & kitty99999

Issues of using hash index & log file

Index limited by the RAM size

- Can put it on the disk but is slow. Why?

Log file is growing forever -> running out of disk space

Lack range query support

- E.g., scan over all keys between kitty00000 & kitty99999

How to prevent a file from growing forever?

Log-structured data file makes updates fast

- Only append to the file

However, we will eventually **run out of disk space**

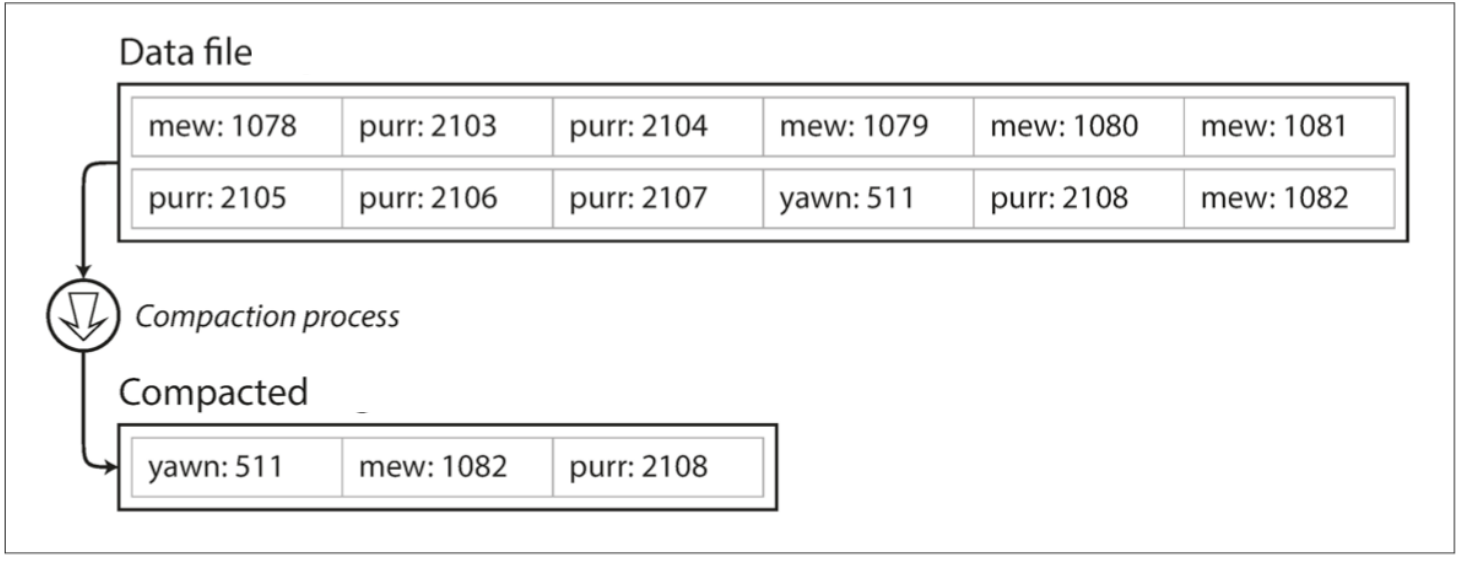
- Since many **duplicate records** in the log file, we should have space to store more records

How to prevent a file from growing forever?

Solution: compaction 合并操作

(最终保留相同key的最新的(最后的)一条记录)

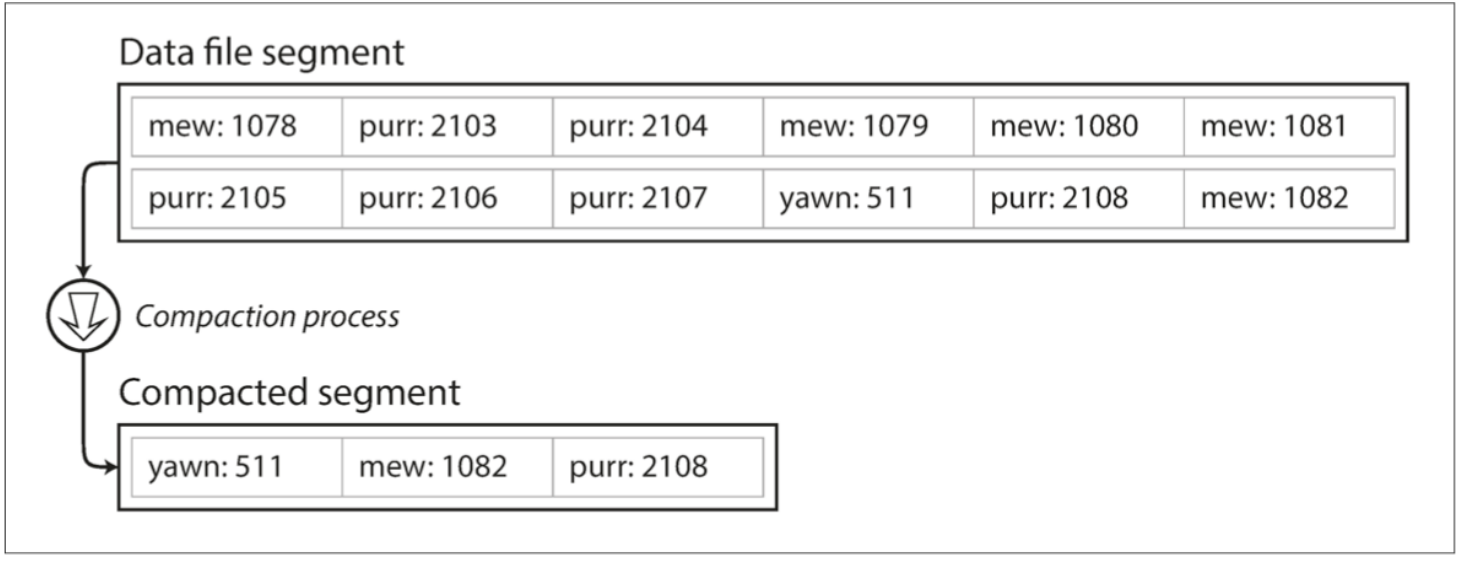
- Scan the whole file, delete duplicate records and write to a new file
- **Question**: what if the file is too large?



How to prevent a file from growing forever?

Solution: compaction & **segmentation**

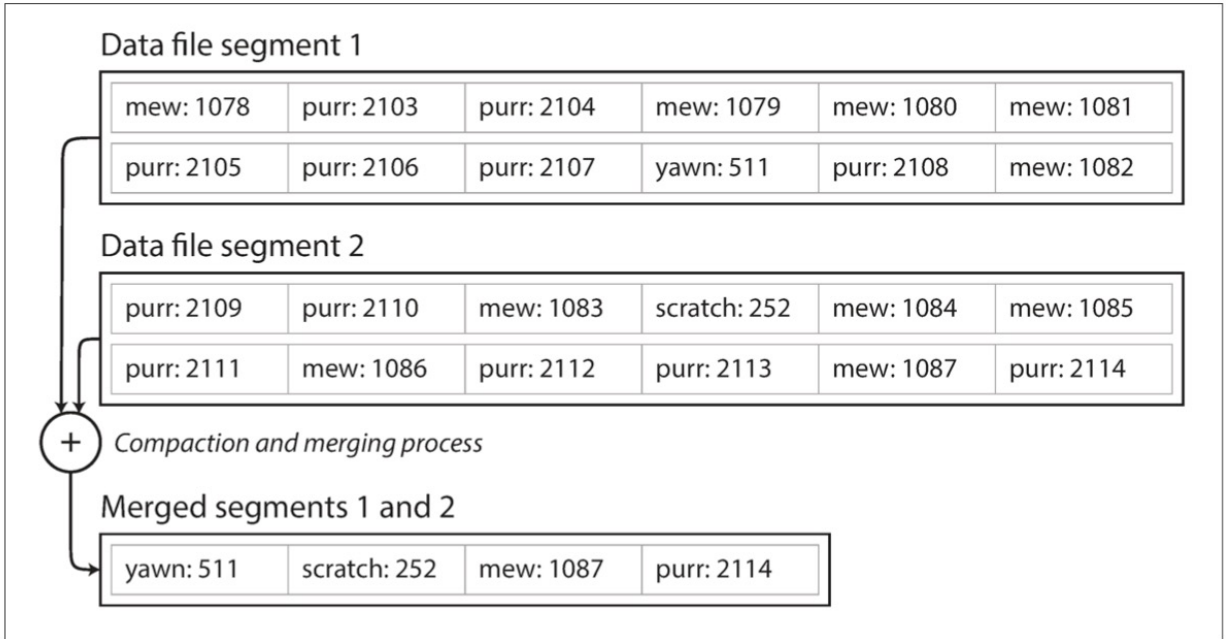
- Break a file into **fixed-sized** segments
- If a segment is full, then create a new file for insertions
- Can control the compaction **granularity** with the segment size



How to prevent a file from growing forever?

Solution: compaction & segmentation & merging 分块&合并&多路归并排序

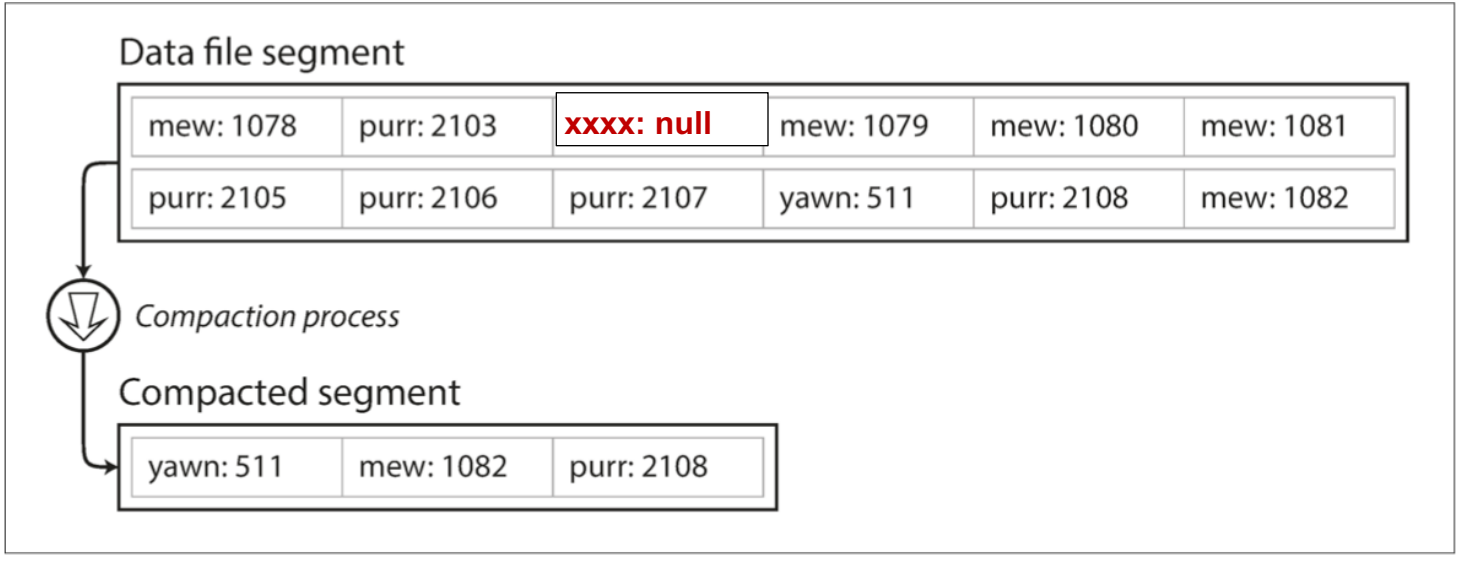
- **Compact**: remove duplicated records in a segmented file
- **Merge**: merge two compacted segments into a new one
  - Why? Because compaction makes the segment smaller



Recall: how to handle delete() for log-structed file?

1. **Append a NULL entry**
2. **Garbage-collected during compaction!**

合并时的“垃圾回收”是指：
在合并过程中发现一个key有一条最新的NULL记录时说明此KV对已经被删除了，所以直接把对应K记录清除即可。



Issues of using hash index & segmented log file

Index limited by the RAM size

- Can put it on the disk but is slow. Why?

Log file is growing forever -> running out of disk space

Lack range query support

- E.g., scan over all keys between kitty00000 & kitty99999

B-Tree (B+Tree) indexes: a form of ordered index

Introduced in 1970, the “ubiquitous” index 10 years later

– Standard data structures of many databases and key-value stores

“It could be said that the world’s information is at our fingertips because of B-trees”

Goetz Graefe Microsoft, HP Fellow, now Google
ACM Software System Award

B+Tree indexes

A B-tree is a tree data structure

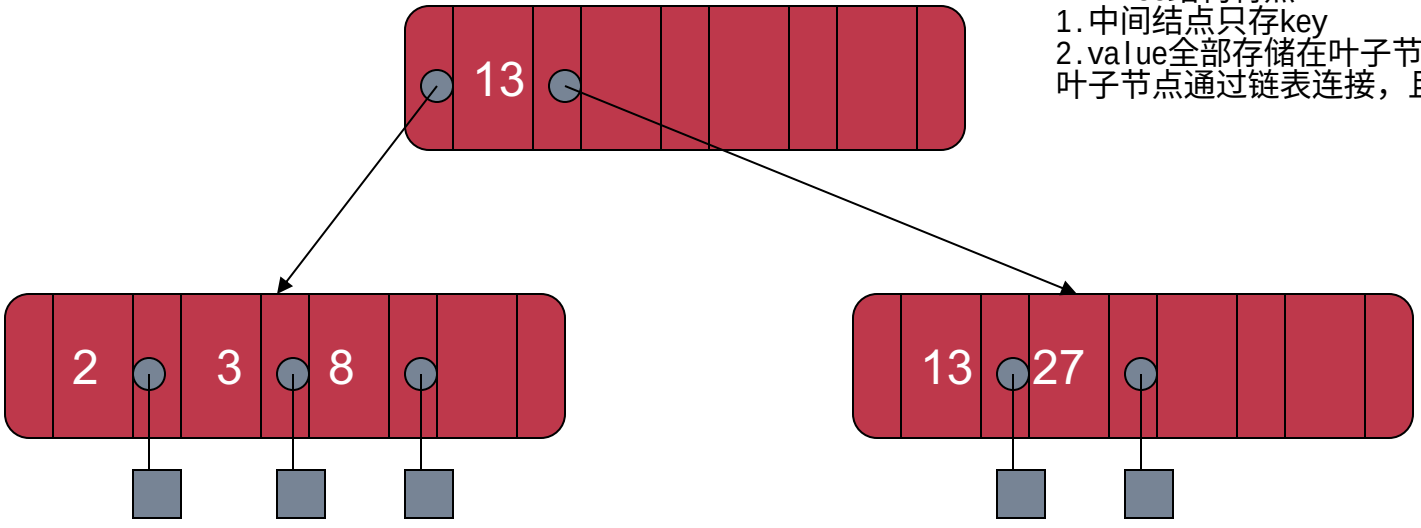
- Each node is **fixed-sized**, can store multiple keys, and keys are **sorted**
- Support **efficient range** operations (e.g., Scan)
- Optimized for **large** read/write blocks of data

B+Tree

- All the leaf nodes of the B-tree must be at the same level
  - Simper to link leave nodes to support range queries

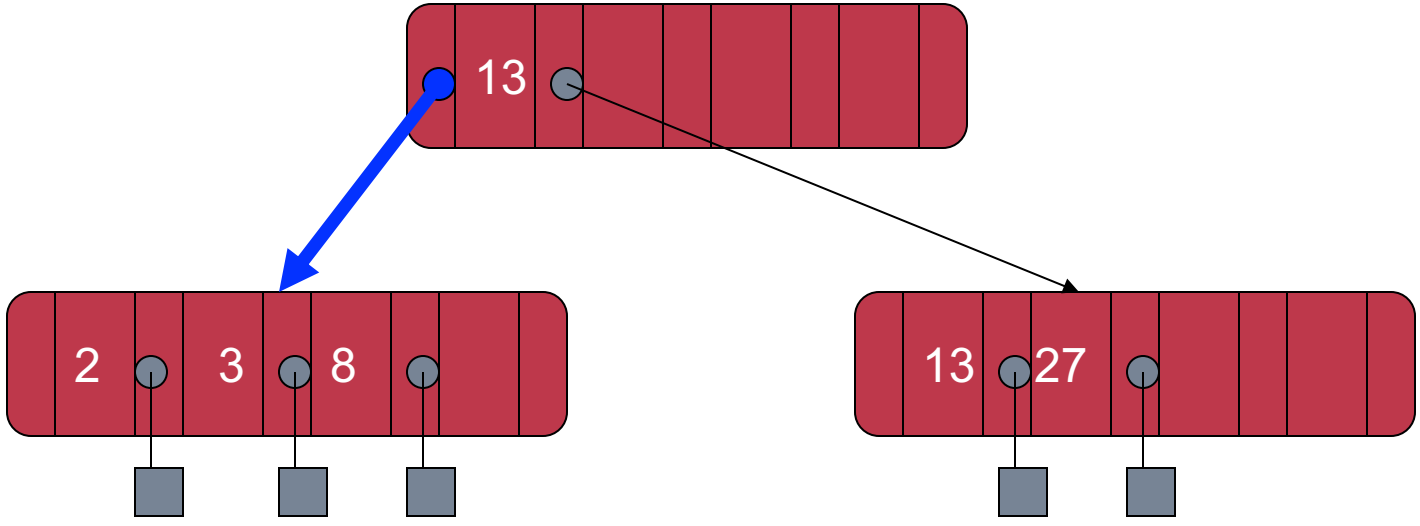
B+Tree indexes example (fanout = 5)

fanout：指的是每个节点最大的分支个数。



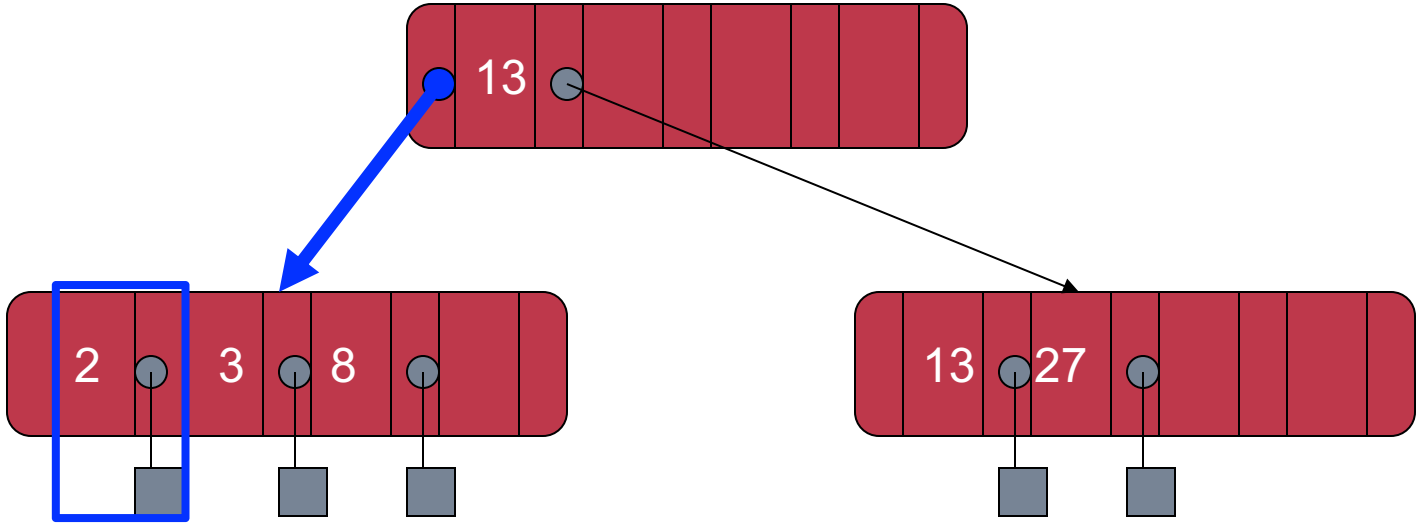
B+ Tree结构特点：
1.中间结点只存key
2.value全部存储在叶子节点中，并且叶子节点通过链表连接，且深度都相等。

B+Tree indexes example: Get(2)



B+Tree indexes example: Get(2)

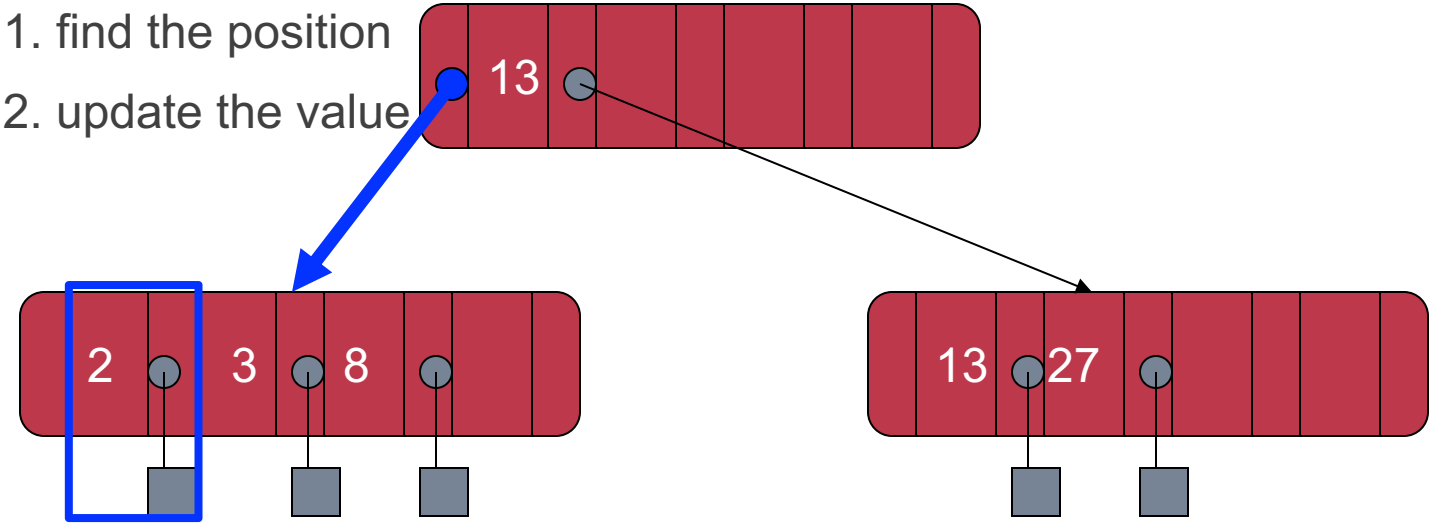
可能需要一次二分查找



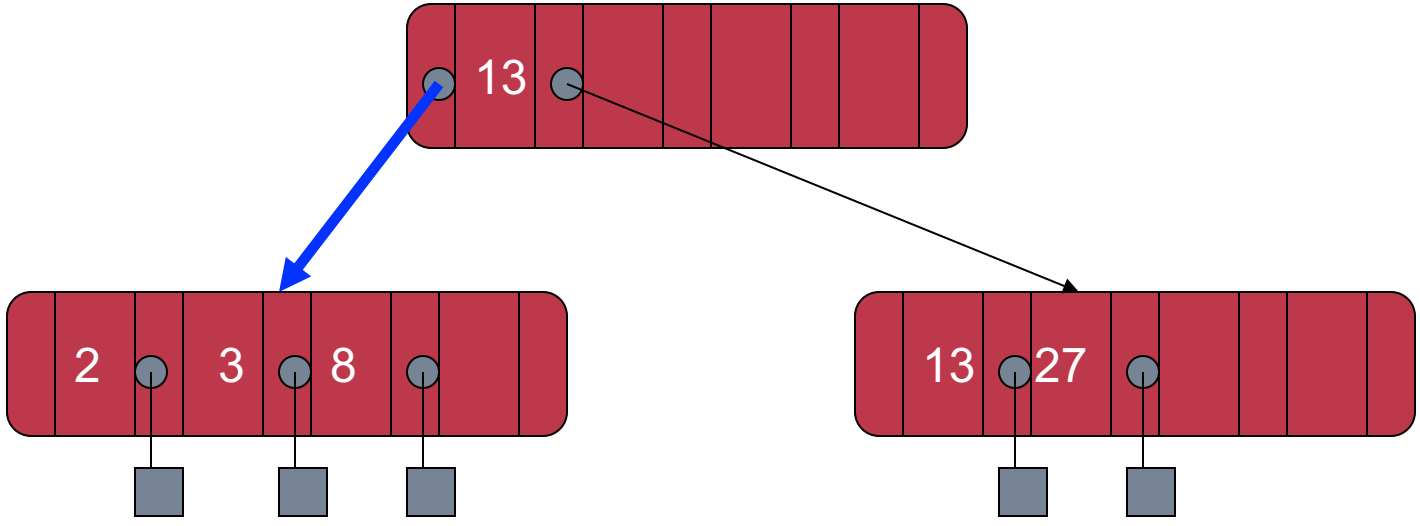
B+Tree indexes example: Update(2)

Similar to Get()

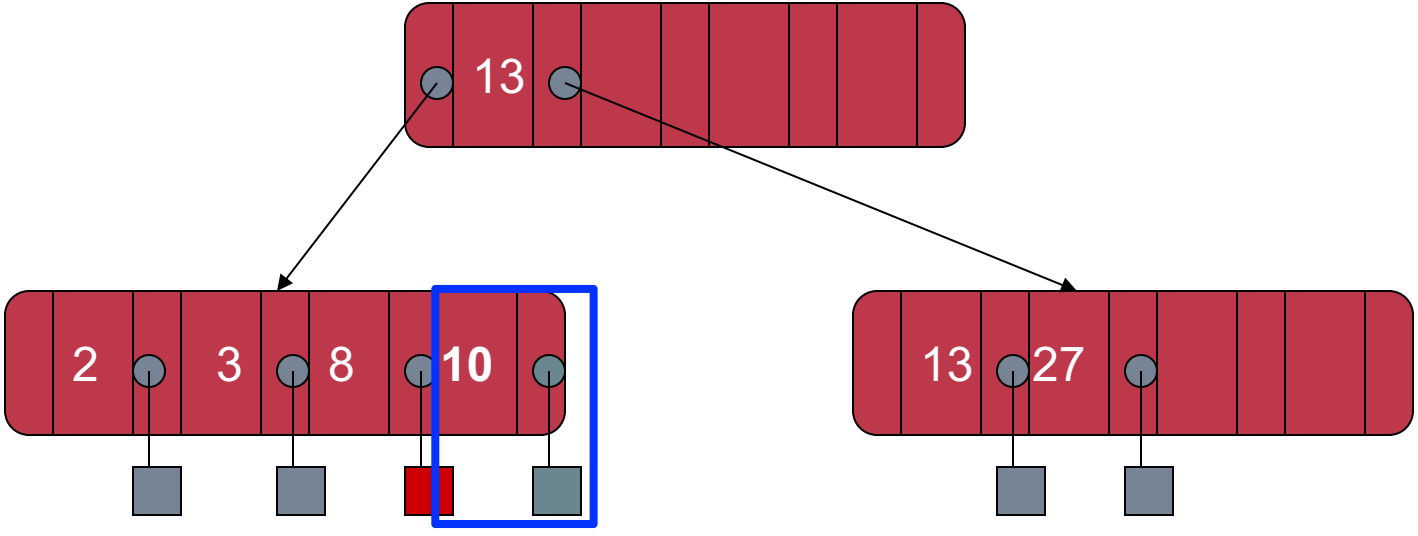
- 1. find the position
- 2. update the value



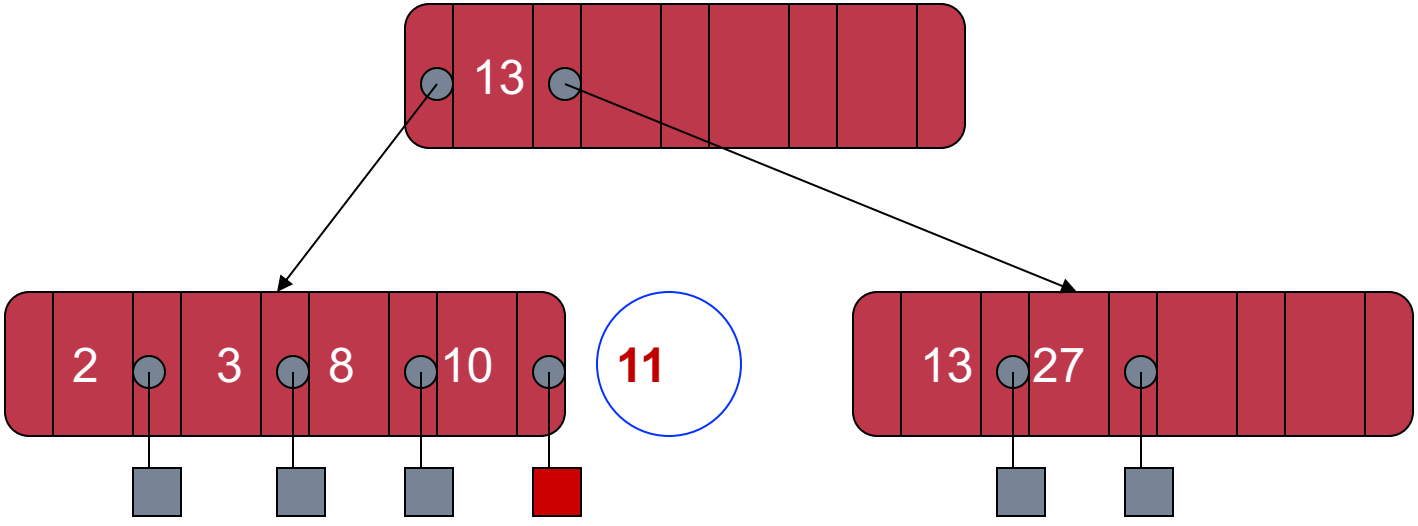
B+Tree indexes example: insert(10)



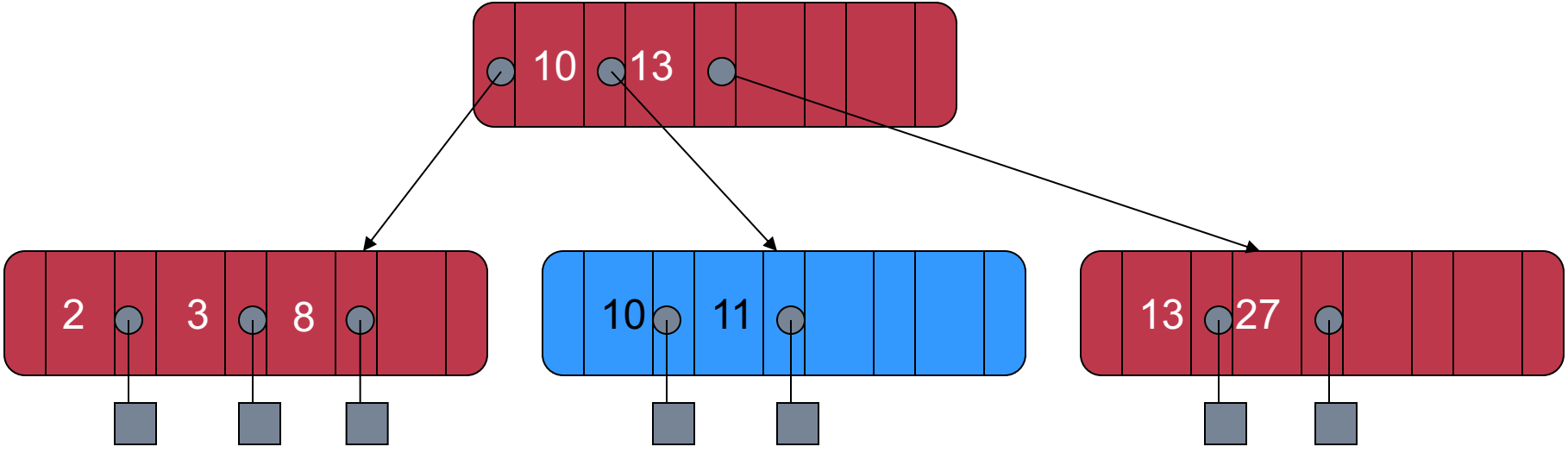
B+Tree indexes example: insert(10)



B+Tree indexes example: insert(11)



B+Tree indexes example: insert(11)



Split the nodes!

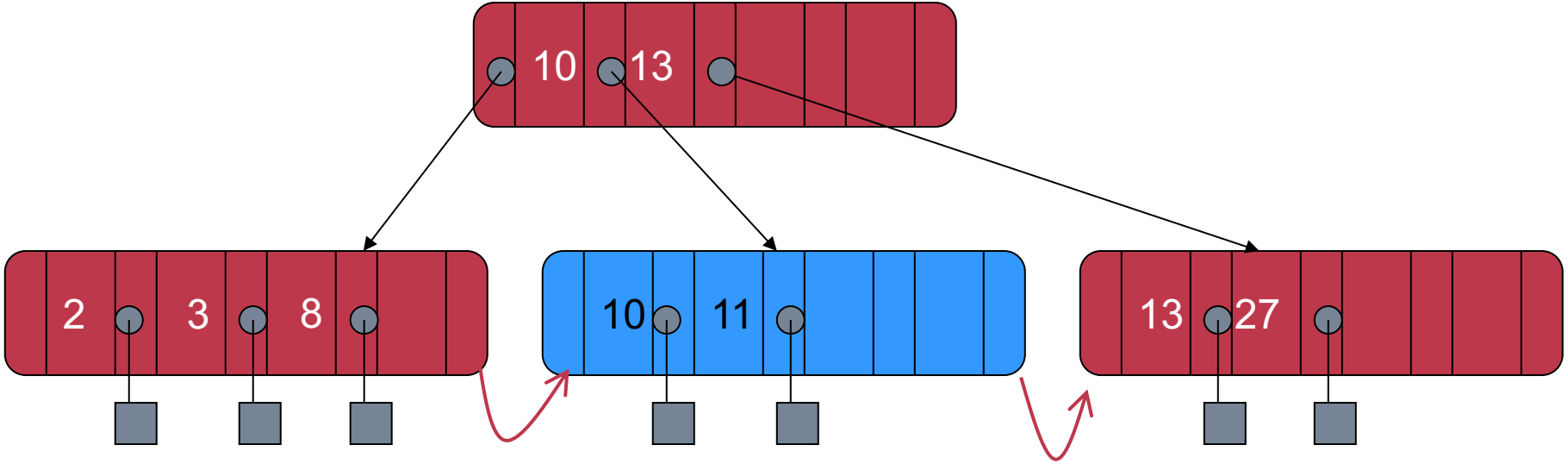
B+Tree can support efficient range operations

Add a sibling link between the leaves

range operations：
取某个特定范围内的数据。

e.g., SCAN(3,5)

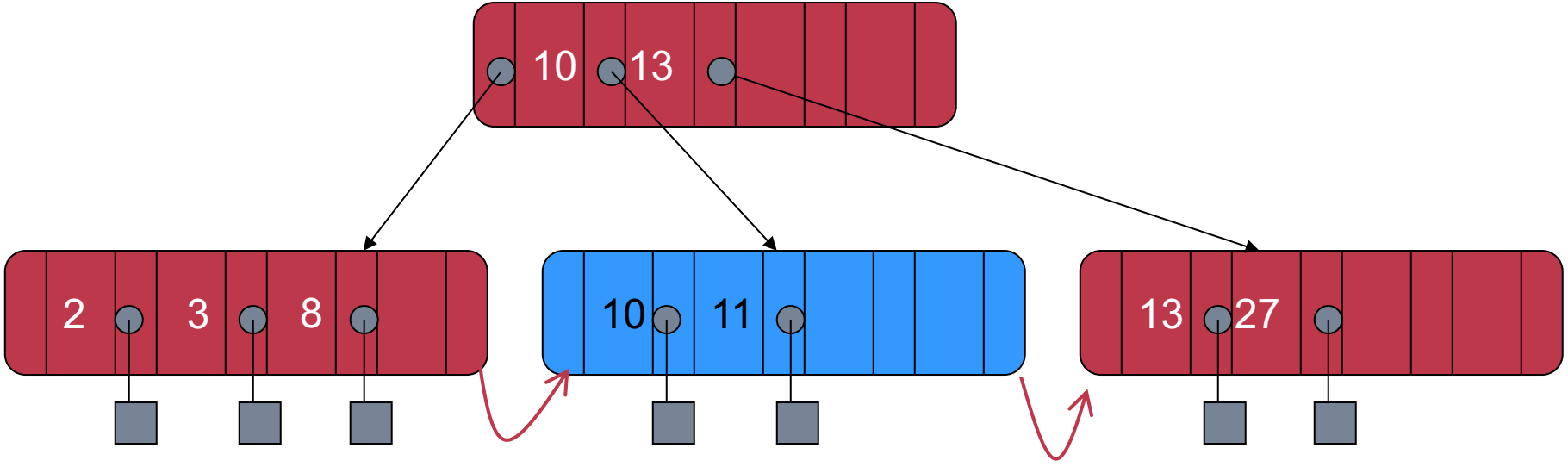
取出来的是3，8，10，11， 13几个节点对应的KV对。



Note that Btree does not use log-structured file

It can store the values in its leave node

- Which does not need compaction & merge



B-Tree indexes on the disk?

Get(K) and Insert(K,V) costs

- O(log(n)) random disk accesses
- Typically, is small, since B-Tree indexes uses large node size

B+Tree indexes are not good enough for insertion-intensive workloads

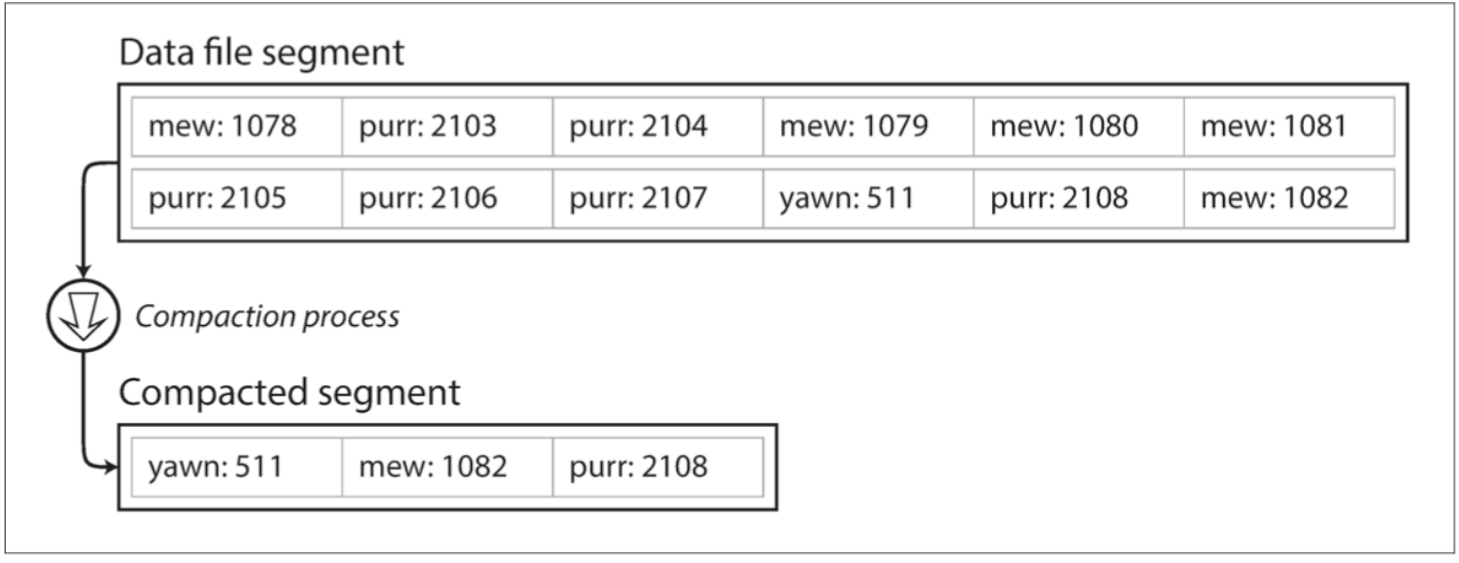
- Still **many random disk I/O** involved
- Insertions are common for on-disk storages, because reads are served by other system component
    - E.g., check lecture 02, step#2 for scaling a website

Can we avoid random access for index updates ?

Recall: segmented log file

Compaction & **segmentation**

- Break a file into **fixed-sized** segments
- If a segment is full, then create a new file for insertions
- Can control the compaction **granularity** with the segment size



LSM Trees: SSTables (Sorted String Table)

SSTable is based on the **segmented log file** described in the hash index

- Break a file into **fixed-sized** segments
- If a segment is full, then create a new file for insertion
- Old segment files are **immutable** and can be **compacted** to save spaces

immutable：不变的

LSM Trees: SSTables (Sorted String Table)

SSTable is based on the **segmented log file** described in the hash index

- Break a file into **fixed-sized** segments
- If a segment is full, then create a new file for insertion
- Old segment files are **immutable** and can be **compacted** to save spaces

With one extension

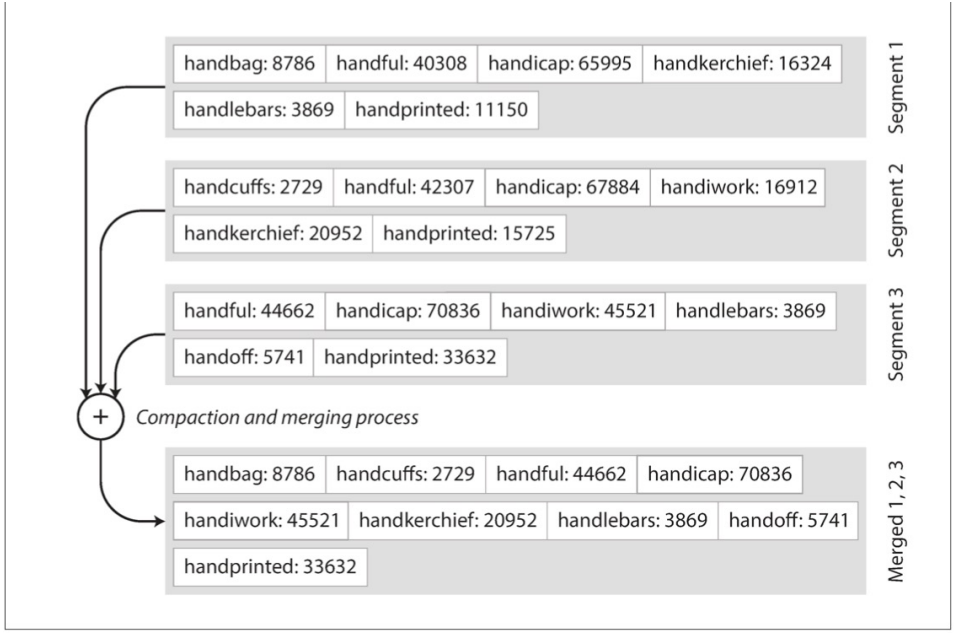
- Key-values in a segment are **sorted** by the key
- Each segmented file is called Sorted String Table (SSTable)

Inserting in a SSTable is **not append-only**
We will come back to this topic later

Benefits of SSTable (compared to original log file)

#1. Merging two SSTables is efficient

- Even if the two tables are much larger than the available memory
- Similar to the `mergesort`

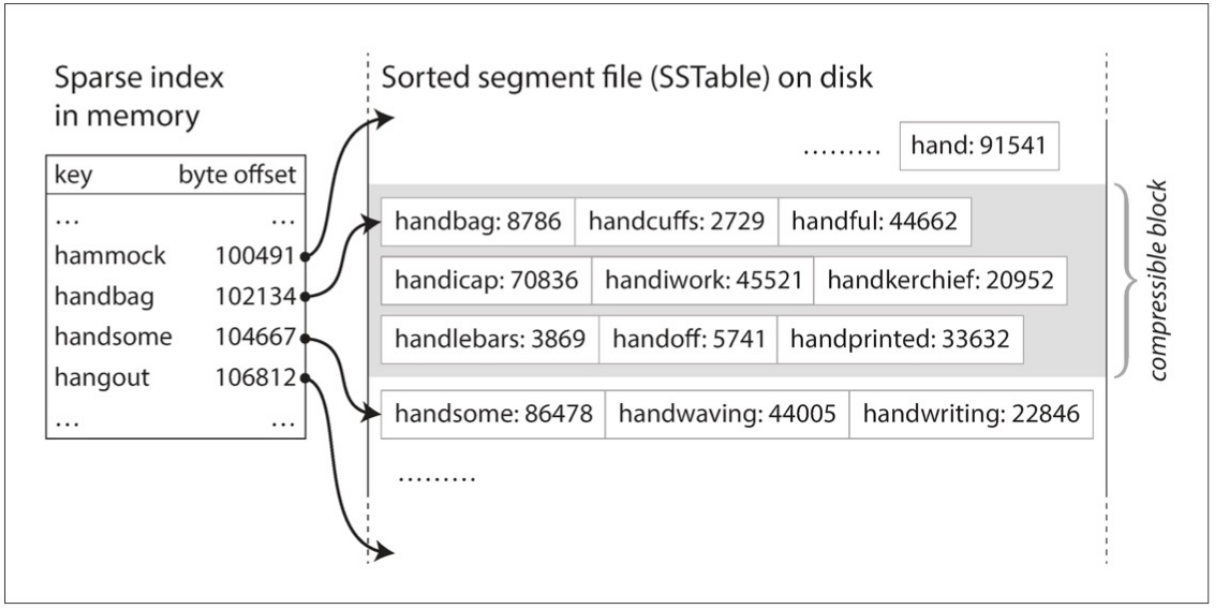


Benefits of SSTable (compared to original log file)

#2. Searching a given key in a SSTable is efficient

- Only needs some sparse index for the acceleration, one for each SSTable
- E.g., one key in index for every few KB of SSTable is sufficient, since scanning them is quick

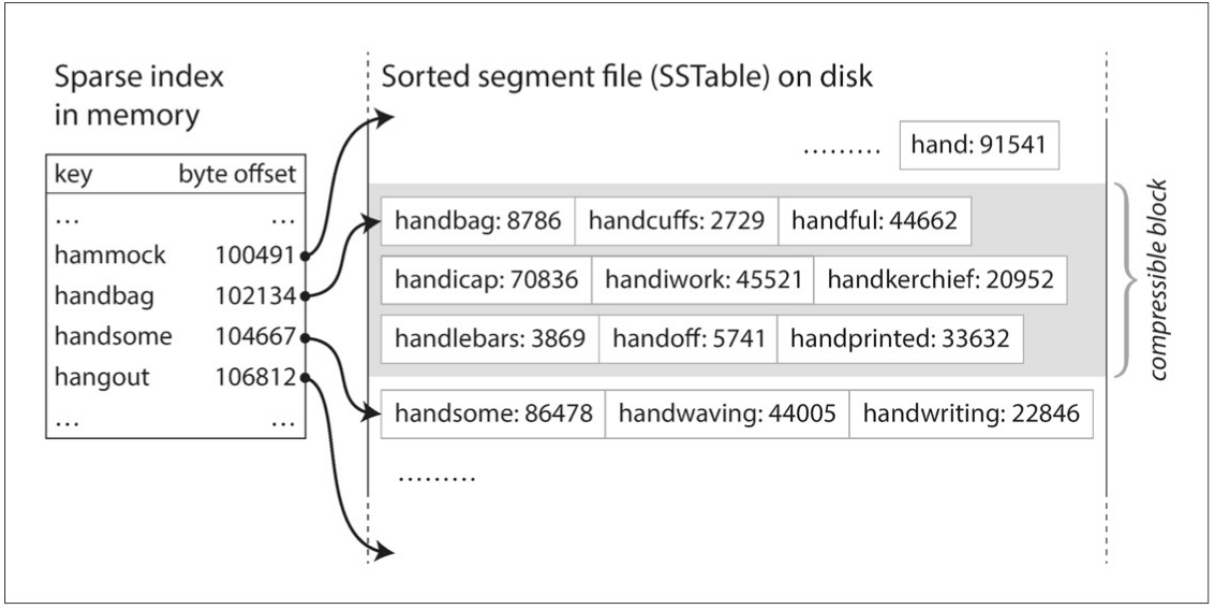
spare index：
稀疏索引
指的是这个索引
并不是存储这个
SST中所有key，
而是只记录一部分
，如之前LSM实现
的只记录最小和
最大值的样子。



Benefits of SSTable (compared to original log file)

#3. Support range operations

– Since the KVS are sorted in the file



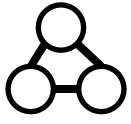
How to maintain SSTable with sequential writes?

Use a MemTable: an in-memory sorted data structure (e.g., red-black Tree)

- Question: why not using hash table?

或者是SkipList

HashTable为冲突时性能很好，但是在数据量很大时发生重散列之后读写性能很差。

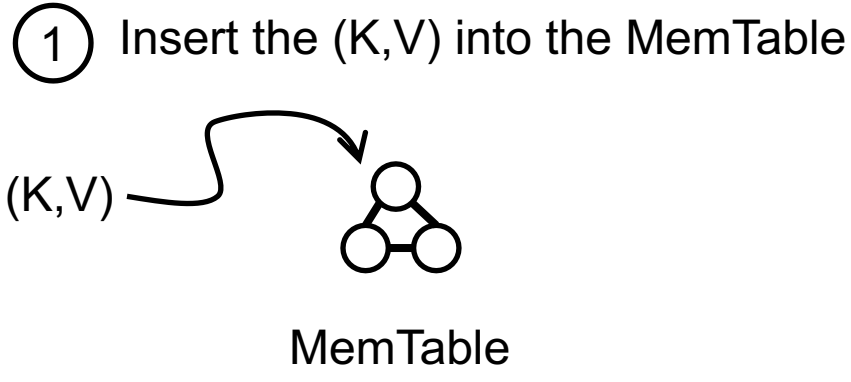


MemTable

How to maintain SSTable with sequential writes?

Use a MemTable: an in-memory sorted data structure (e.g., red-black Tree)

Example: insert a (K, V) into SSTable

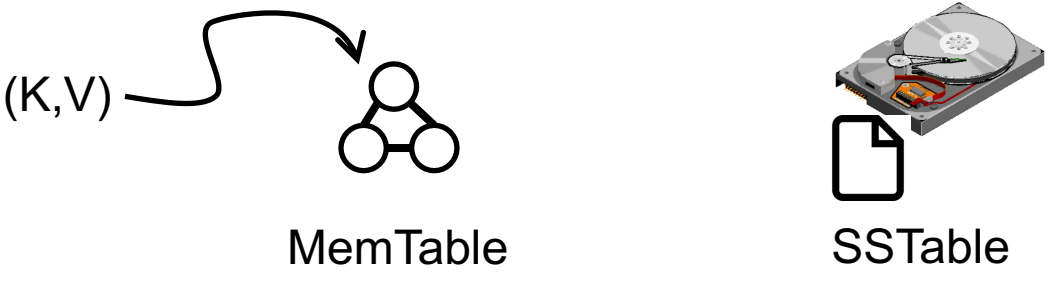


How to maintain SSTable with sequential writes?

Use a MemTable: an in-memory sorted data structure (e.g., red-black Tree)

Example: insert a (K, V) into SSTable

1 Insert the (K,V) into the MemTable



Question: does writing to SSTable sequential?

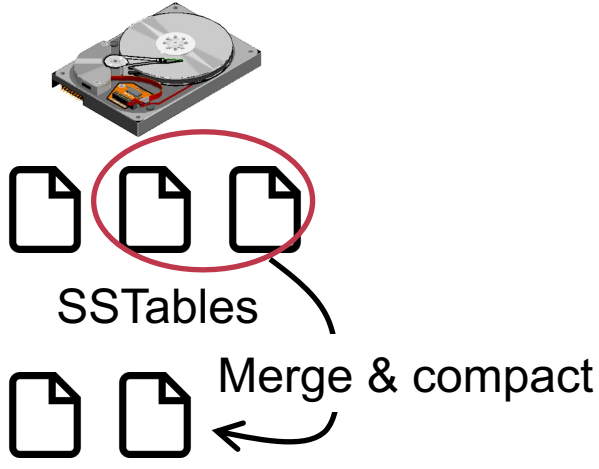
2 check whether MemTable exceeds some threshold
If so, flush it to a new SSTable

How to maintain SSTable with sequential writes?

Use a MemTable: an in-memory sorted data structure (e.g., red-black Tree)

Example: insert a (K, V) into SSTable

3 What if there are too many SSTables? We can **merge & compact** them as we have done before with segmented log files

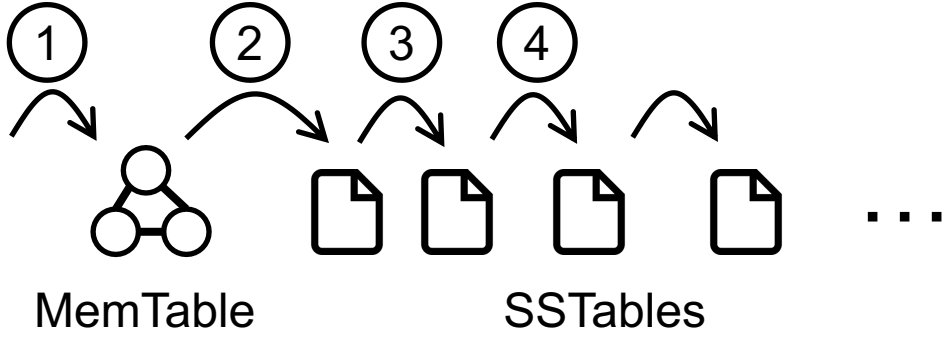


Reads with MemTable + SSTable

1. Checks the MemTable
2. If misses, checks the latest SSTable
3. If still misses, checks the next older SSTable
4. …

Optimization: keep a sparse index on each SSTable to accelerate the lookup

通过sparse index来快速的判断要查找的key是否位于某个SST中。

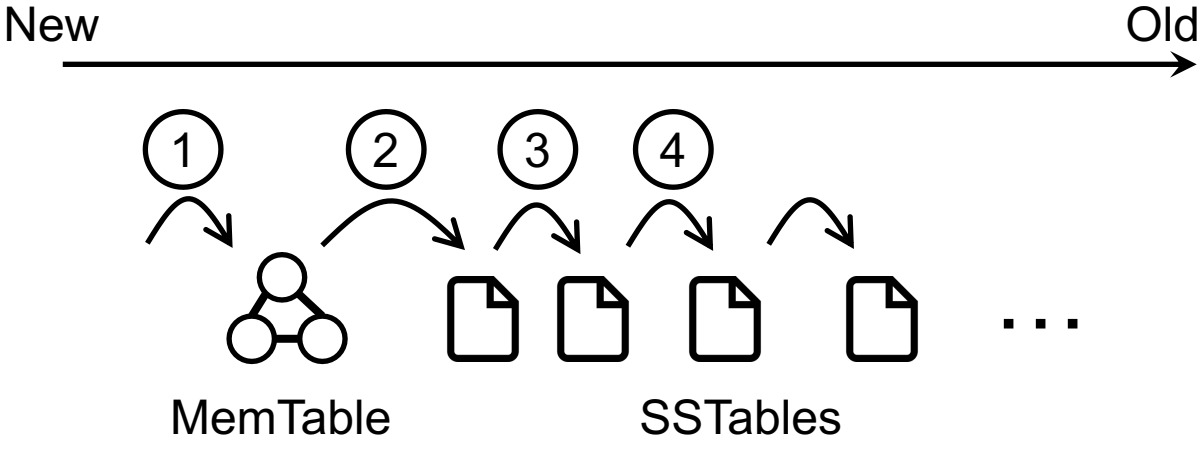


Question: what are the drawbacks of such design?

Organizing data sequentially has several drawbacks

#1. Finding the old value is slow

#2. Cannot support efficient scan



LSM Tree organizes LSM Tree as sorted Tree

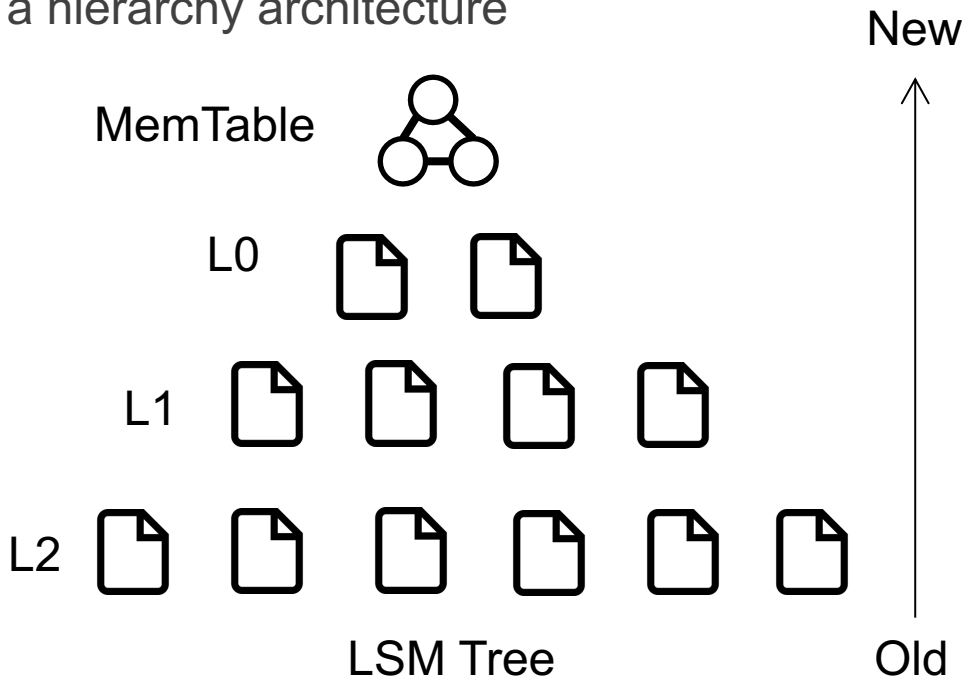
The SSTable & operations are essentially LSM Tree

- **L**og-**s**tructured **m**erge tree

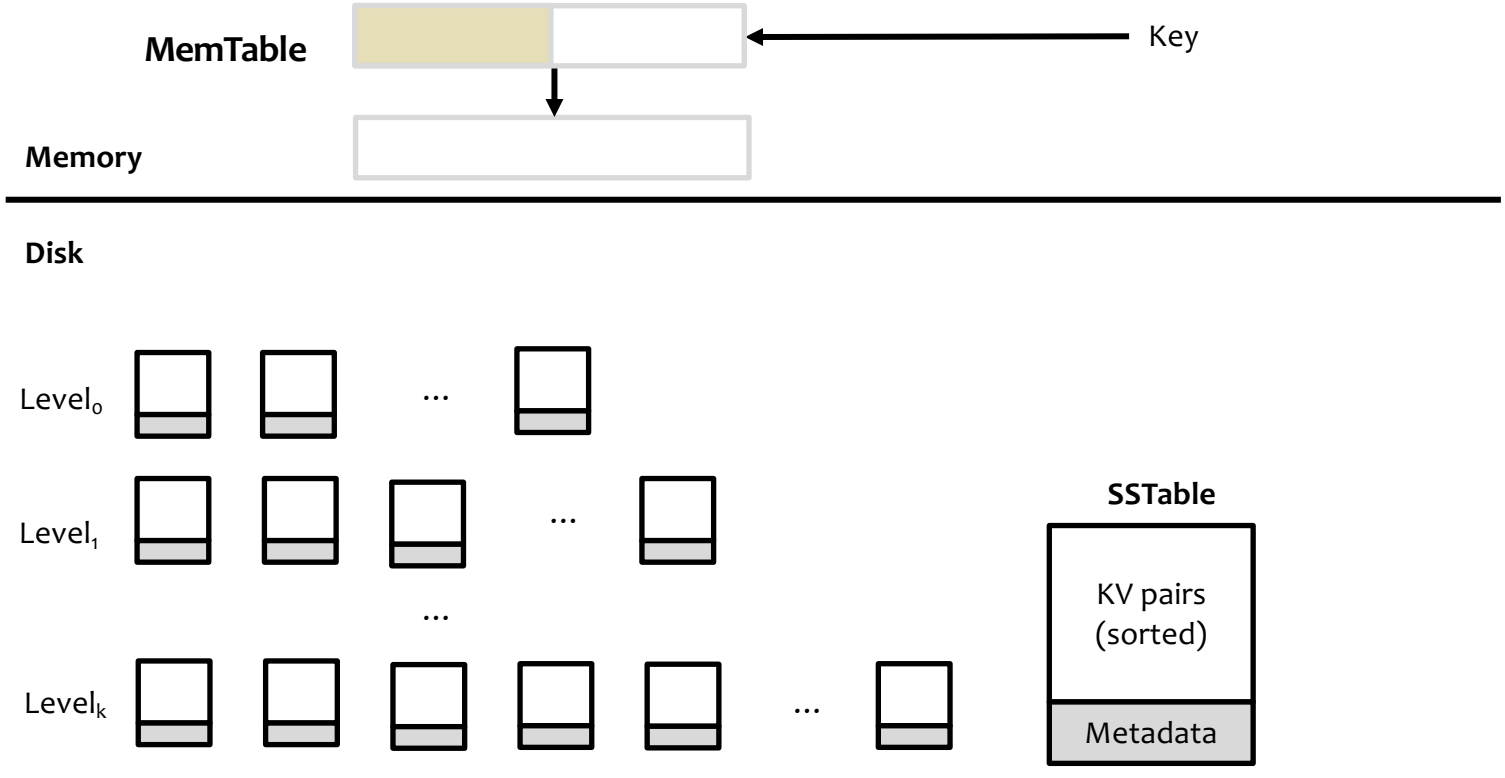
LSM Tree organizes SSTables as a hierarchy architecture

- Each layer has maximum size
- Except L0, all files in layers are **sorted**

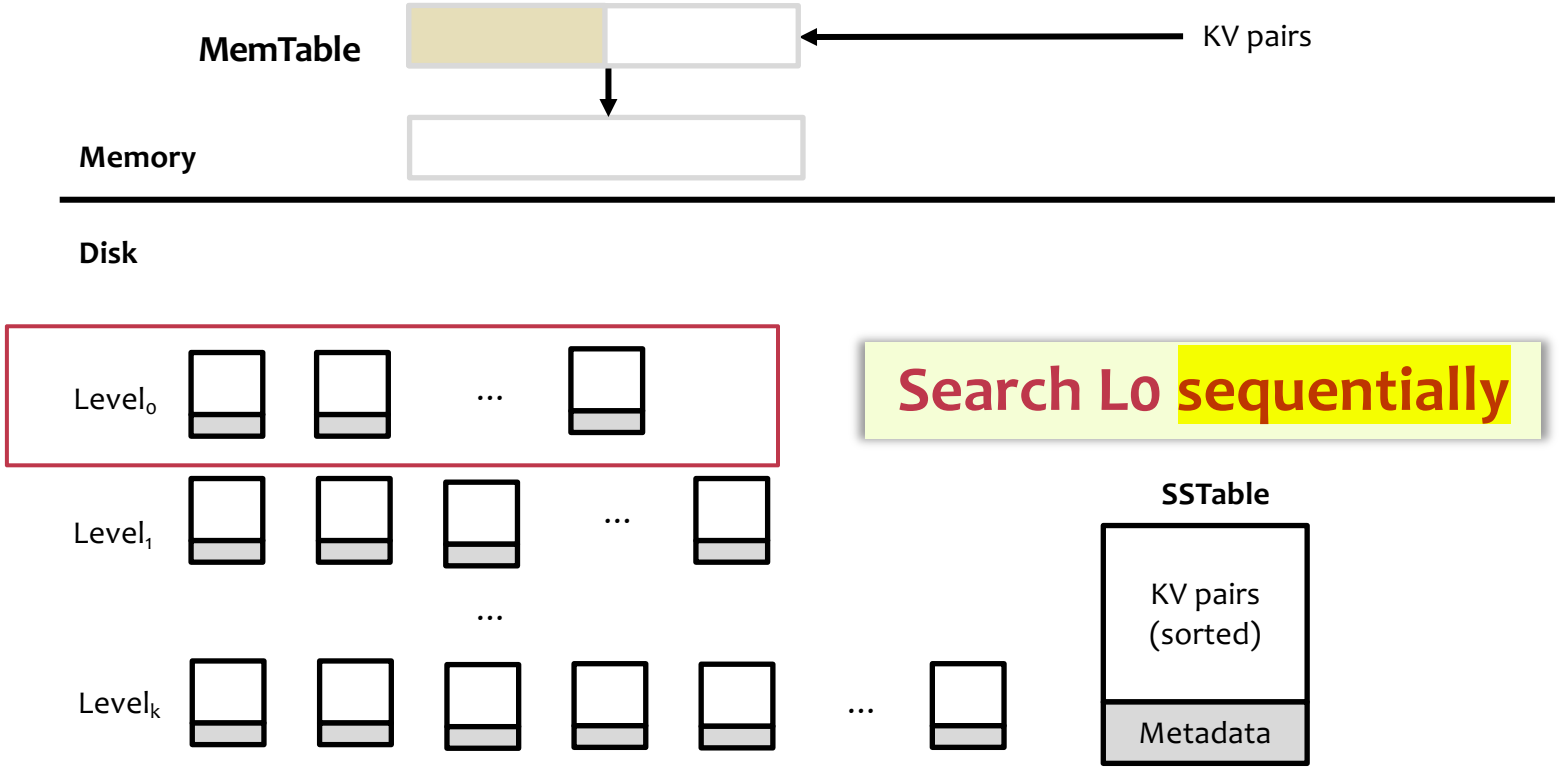
L0层不排序原因：
1.L0层文件较少，直接遍历不会很慢
2.L0层是新的SSTable的写入的位置，不排序也有利于提高MemTable写入disk的效率。



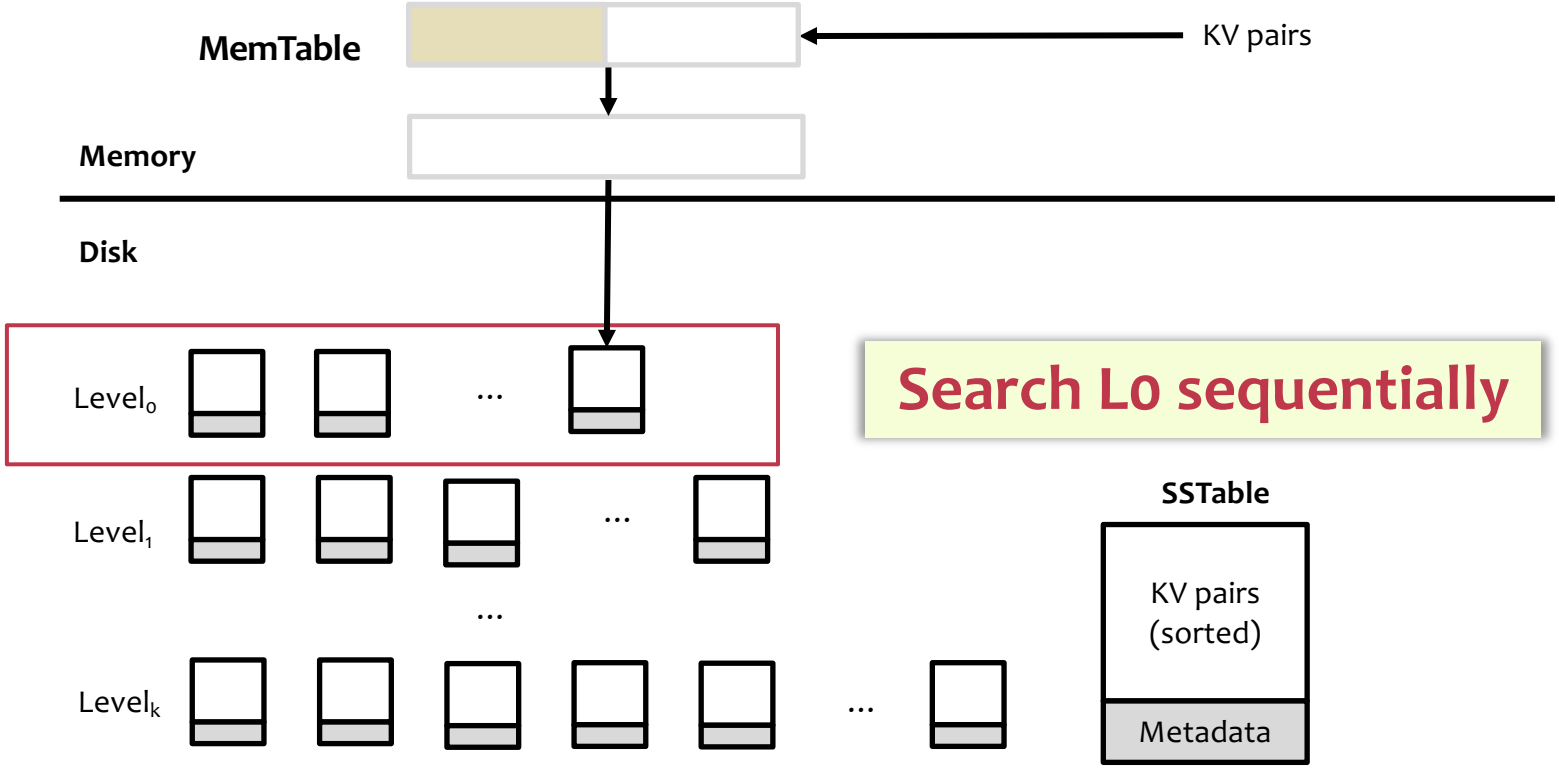
Example: Read in LSM-Tree



Example: Read in LSM-Tree

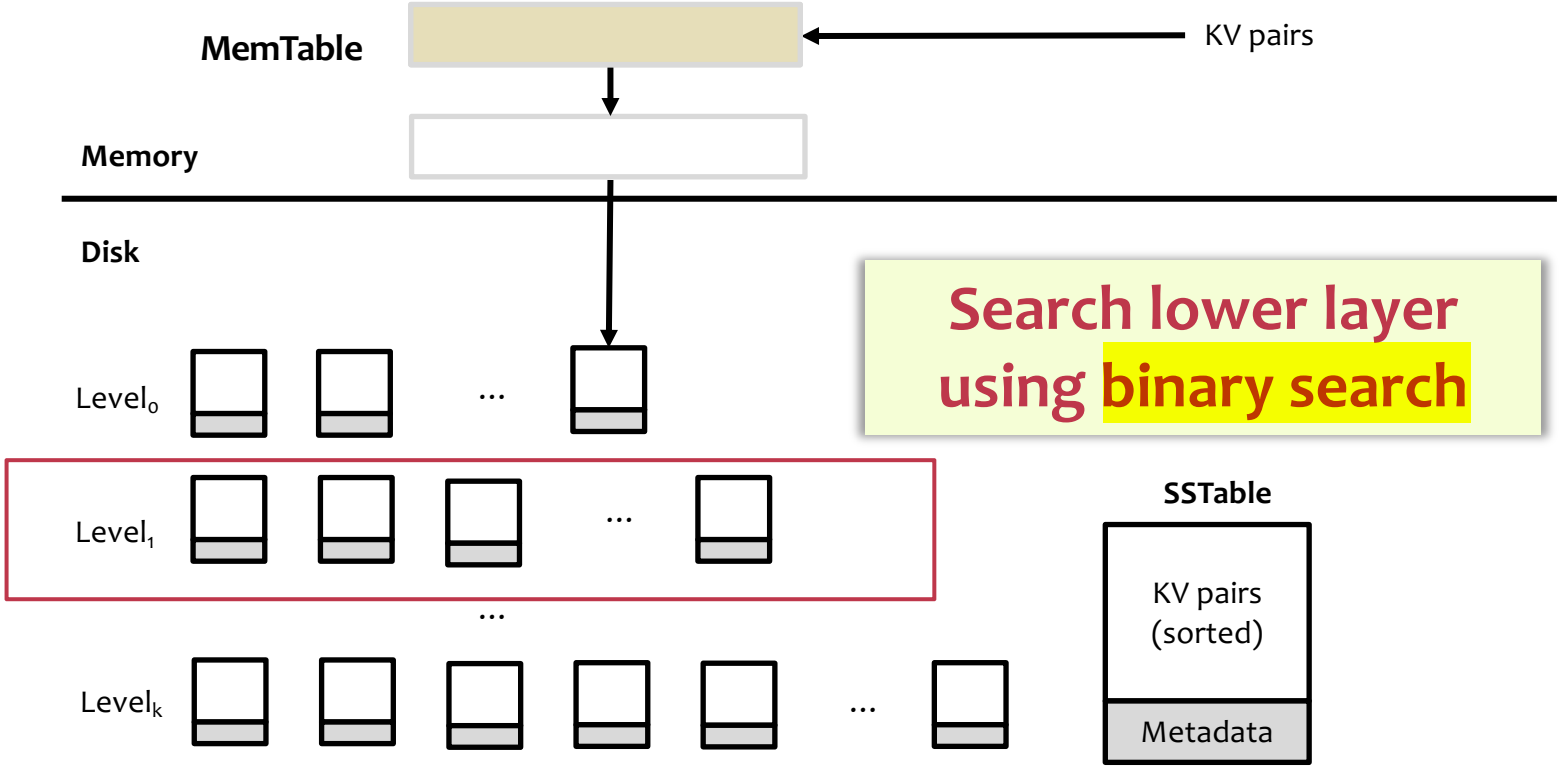


Example: Read in LSM-Tree

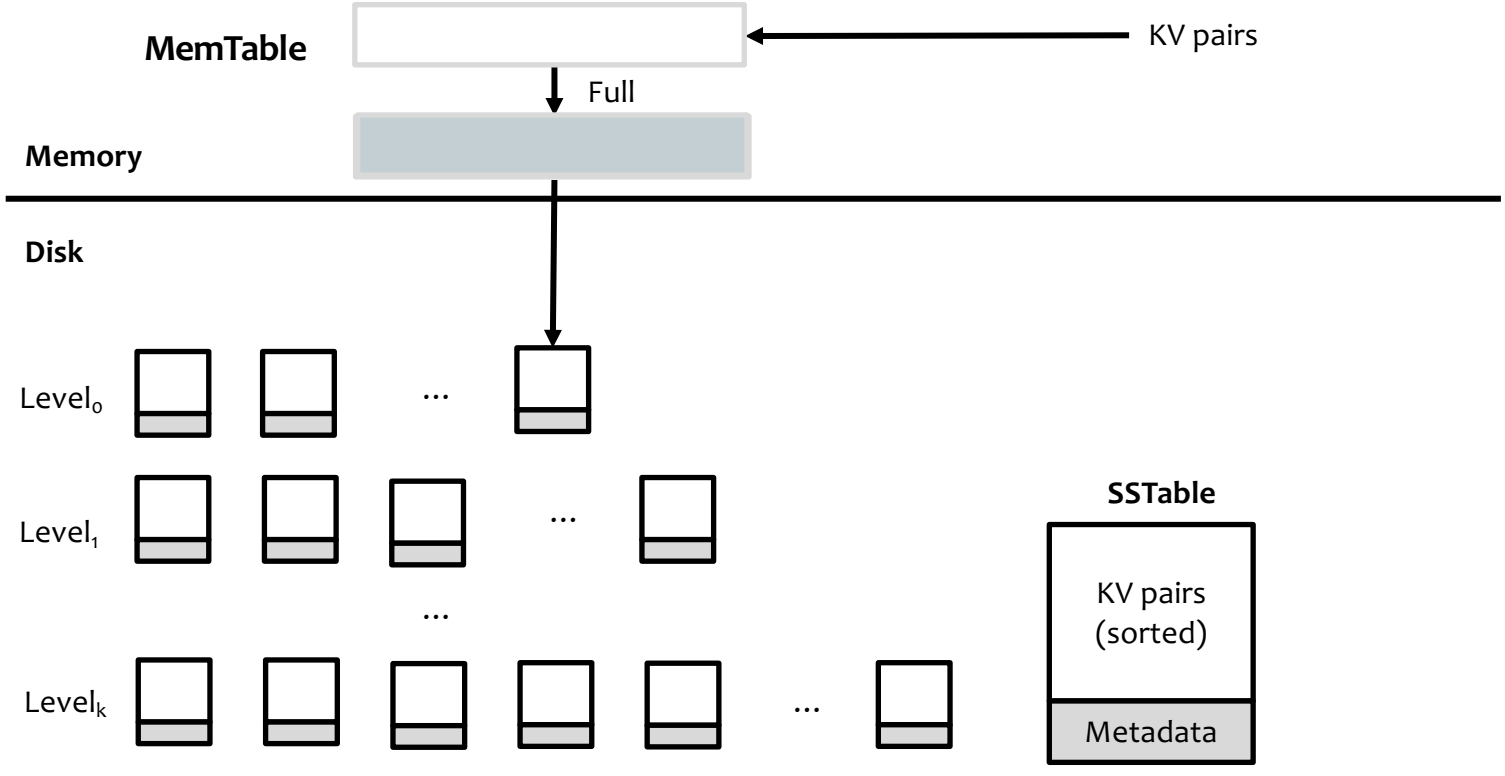


Example: Read in LSM-Tree

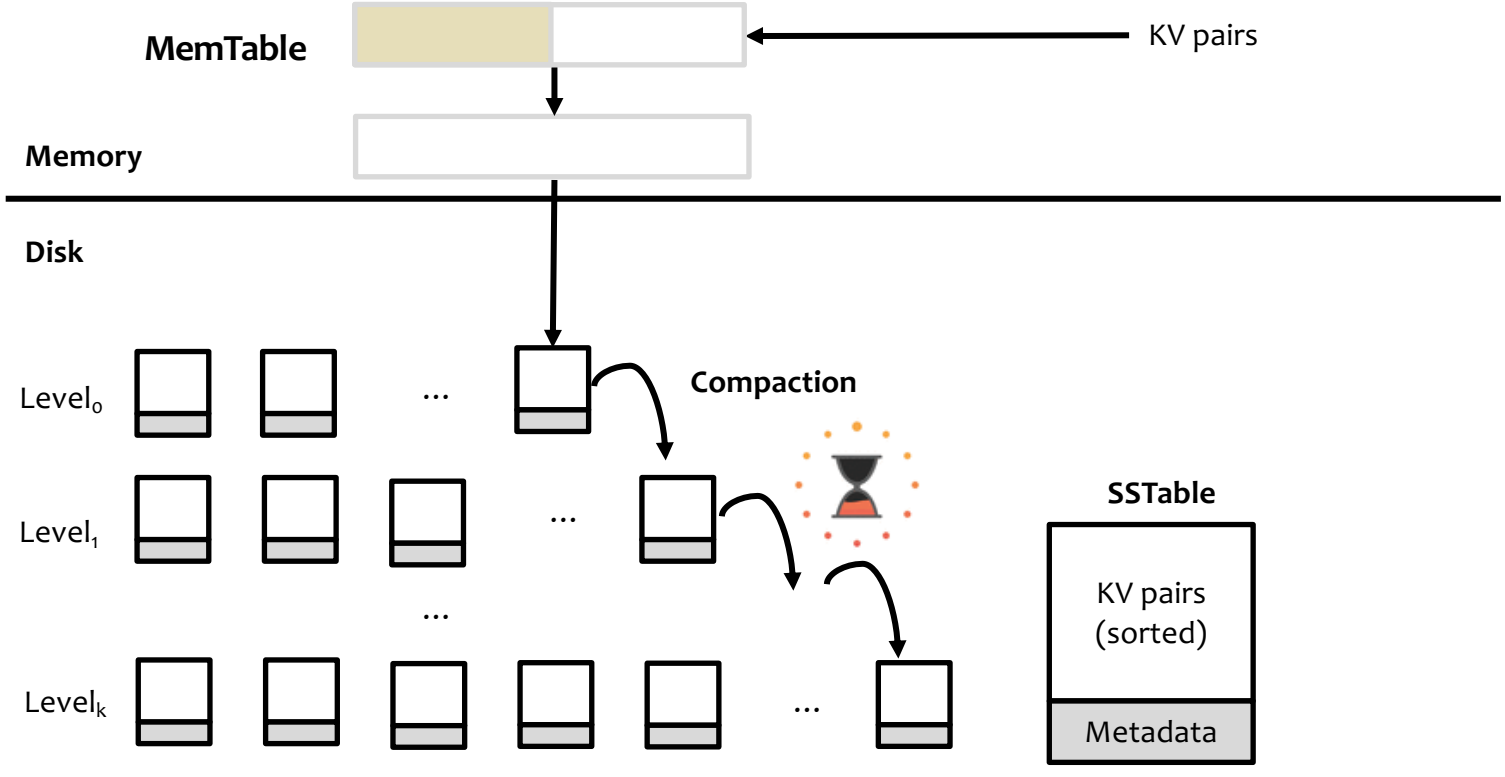
顺序数组查找-->二分查找



Example: insert/update in LSM-Tree



Example: insert/update in LSM-Tree



What about crash?

MemTable is vulnerable to machine failure

Solution:

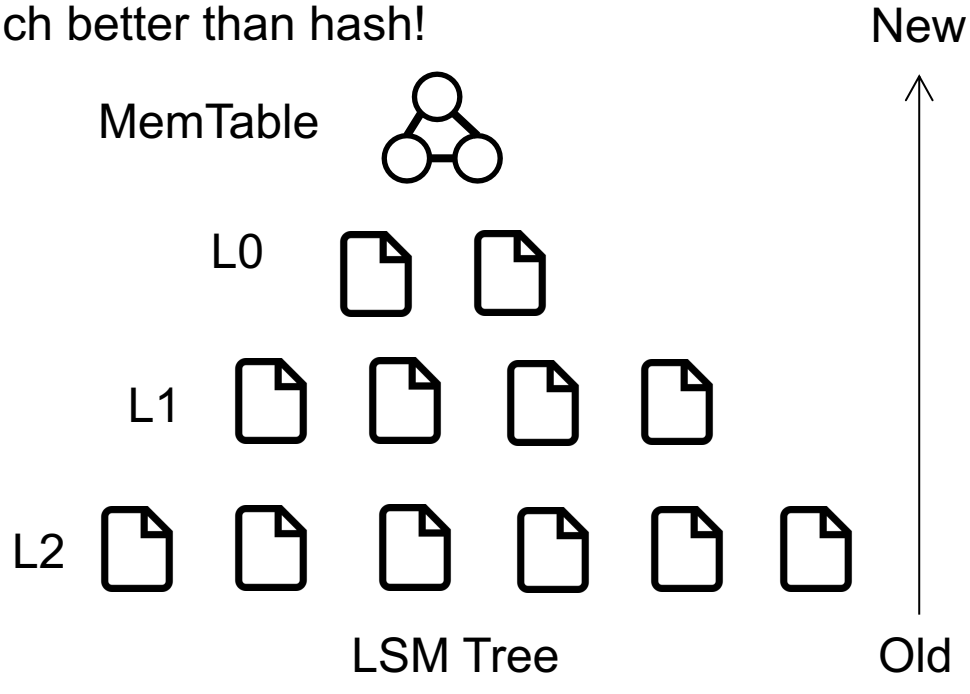
- Keep a separate log for each KV inside a MemTable (may not be sorted)
- Restore the MemTable after reboot

MemTable存储在内存中，所以发生crash时会出现数据丢失的问题。
解决方式：
1.若为正常的系统关闭，可以采用将剩余的memtable中内容写入一个新的SST中，之后重启在读出来
2.若为发生crash，则使用一个额外的log file(disk)来记录对于MemTable数据写入情况，在重启之后通过log进行恢复。

Range Query in LSM Tree

1. Search each layer using **binary search**
2. **Merge** the results of each layer

Not as good as B+Tree, but is much better than hash!



LSM Tree Summary

Good when

- Massive dataset
- Rapid updates/insertions
- Fast single-point lookup for recently updated data

Widely adopted in modern single-node key-value stores

LEVELDB

Google Cloud Bigtable

RocksDB

LSM Tree Summary

Good when

- Massive dataset
- Rapid updates/insertions
- Fast single-point lookup for recently updated data

Compared with B-Tree

- **Pros**: good write performance due to sequential writes
- **Cons**: additional compaction process, possible slow range queries, write stall caused by the compaction, slow lookup for non-existent key

write stall(写入阻塞)：当memtable满的时候，其会将存储的数据写入新的SST，但是这时候如果上一次的Compaction还未结束，则会出现SST写入失败的情况，如果在MemTable未清空时又有新的写入操作，则这个写入操作也会失败。

LSM Tree Summary

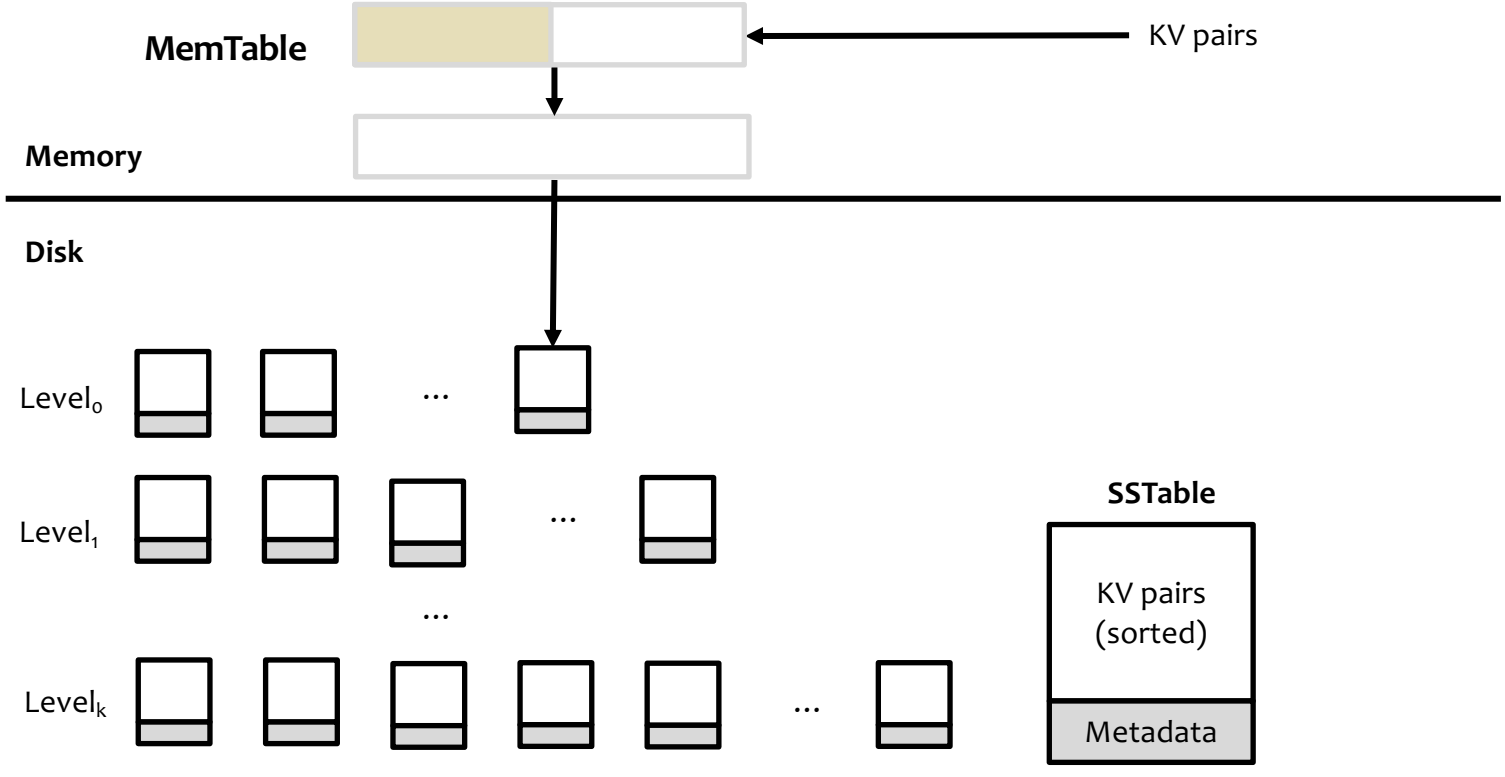
Good when

- Massive dataset
- Rapid updates/insertions
- Fast single-point lookup for recently updated data

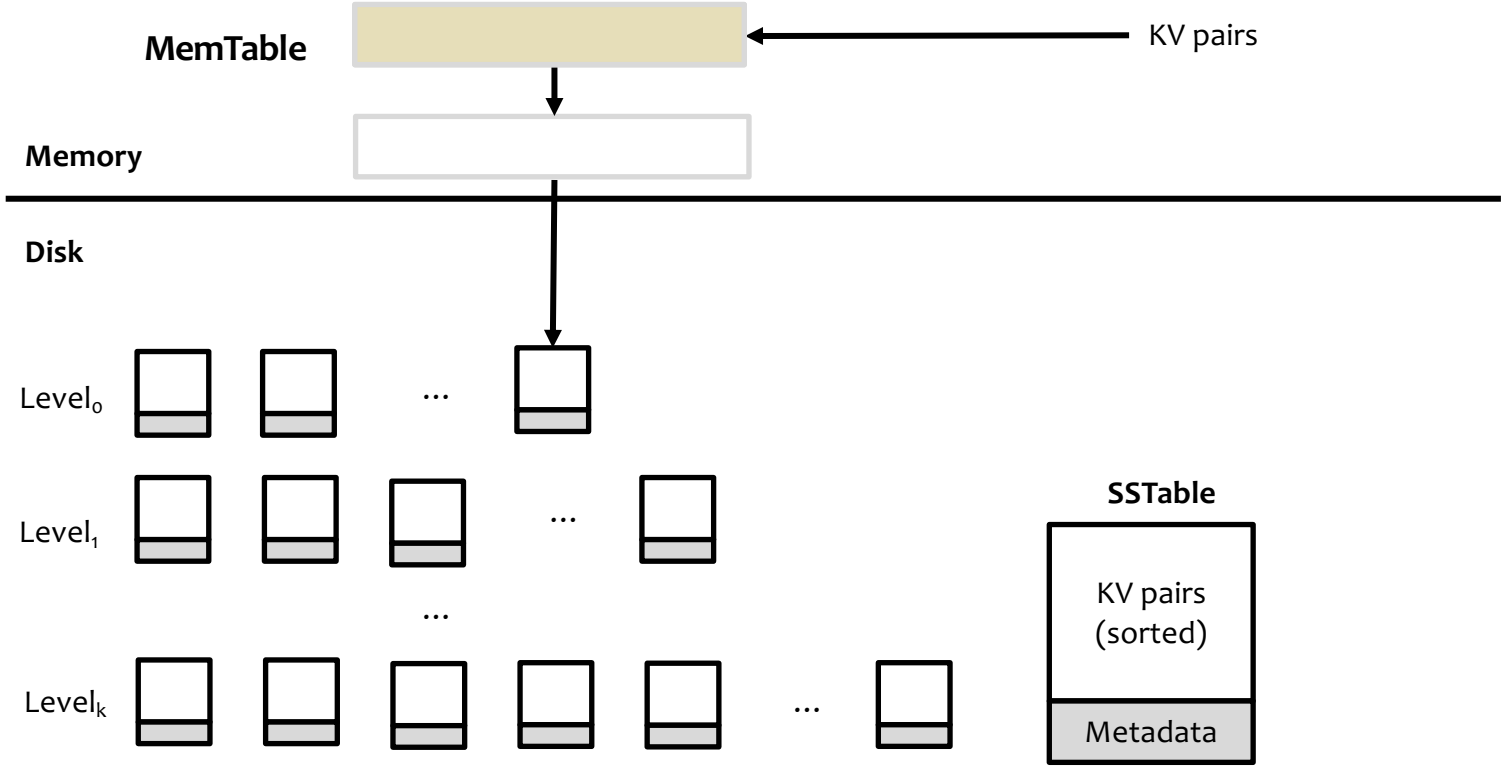
Compared with B-Tree

- **Pros**: good write performance due to sequential writes
- **Cons**: additional compaction process, possible slow range queries, **write stall** caused by the compaction, slow lookup for non-existent key

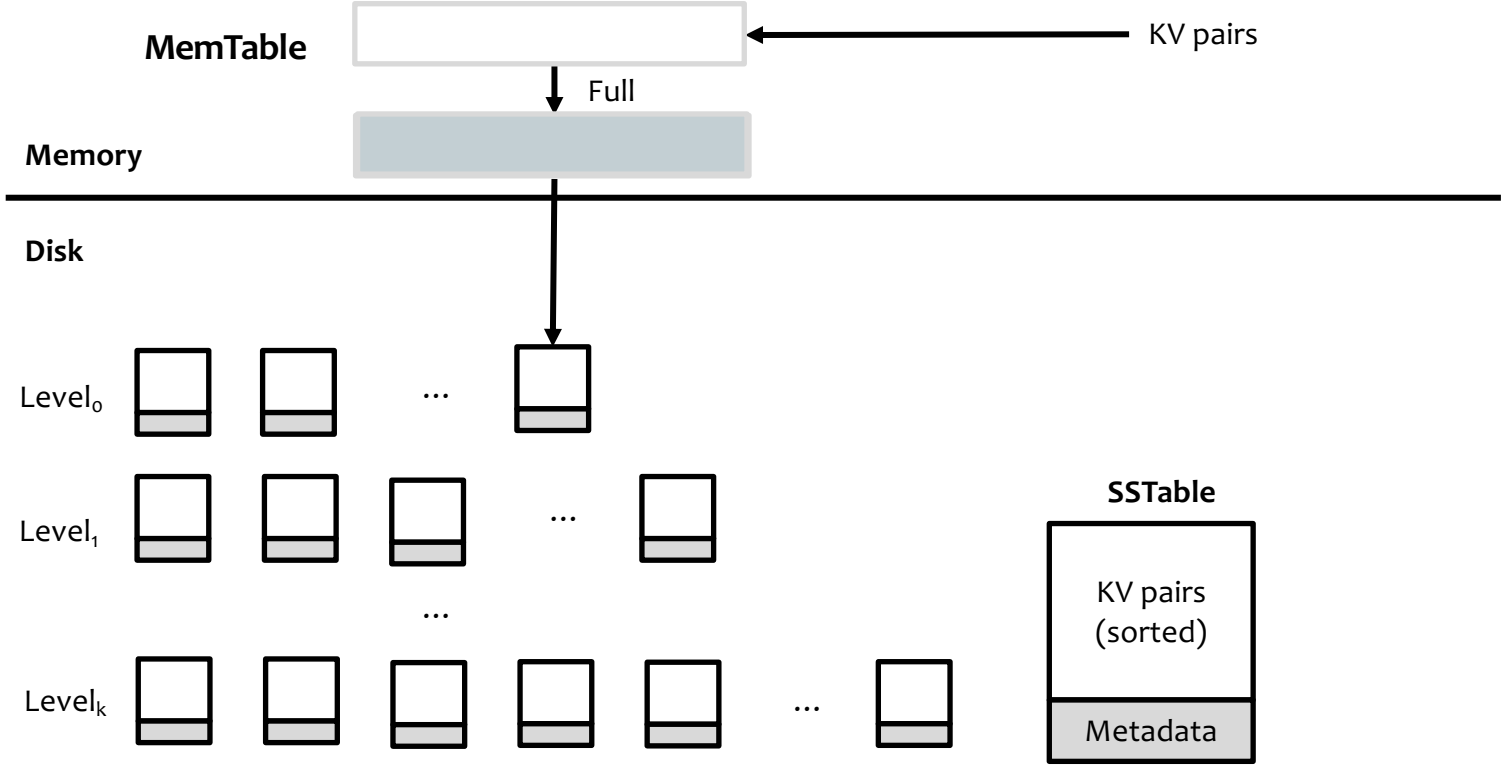
Write stall caused by compaction in LSM Tree



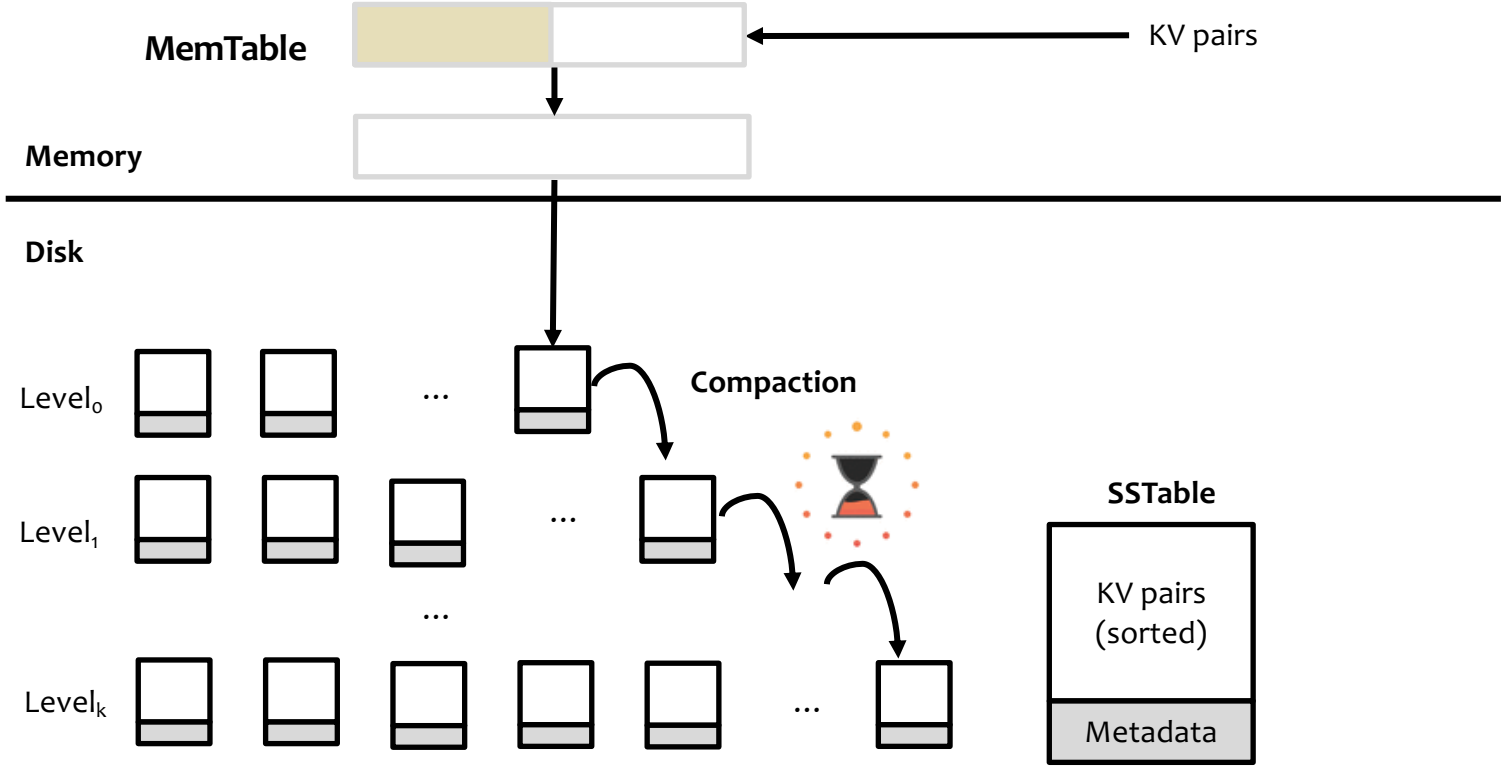
Write stall caused by compaction in LSM Tree



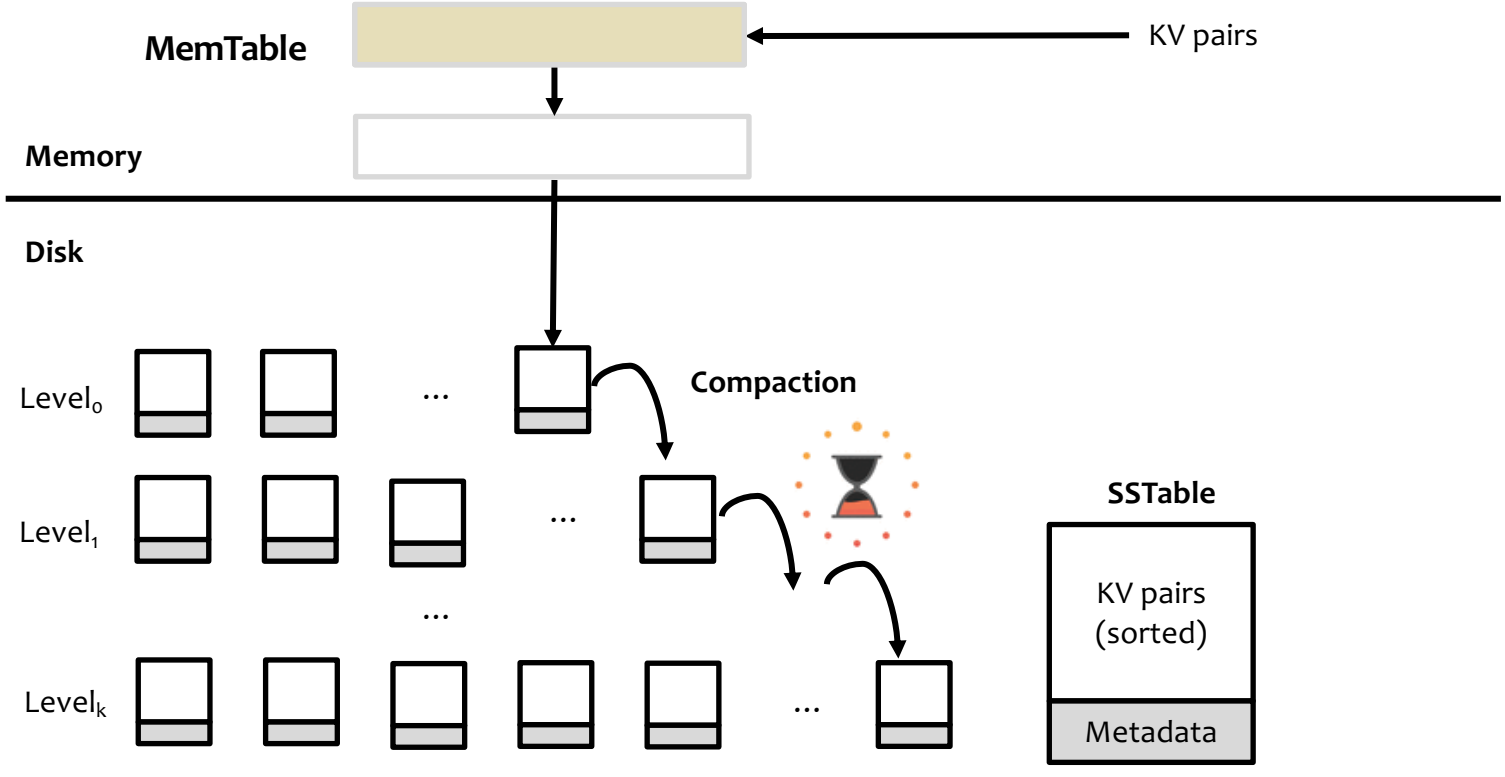
Write stall caused by compaction in LSM Tree



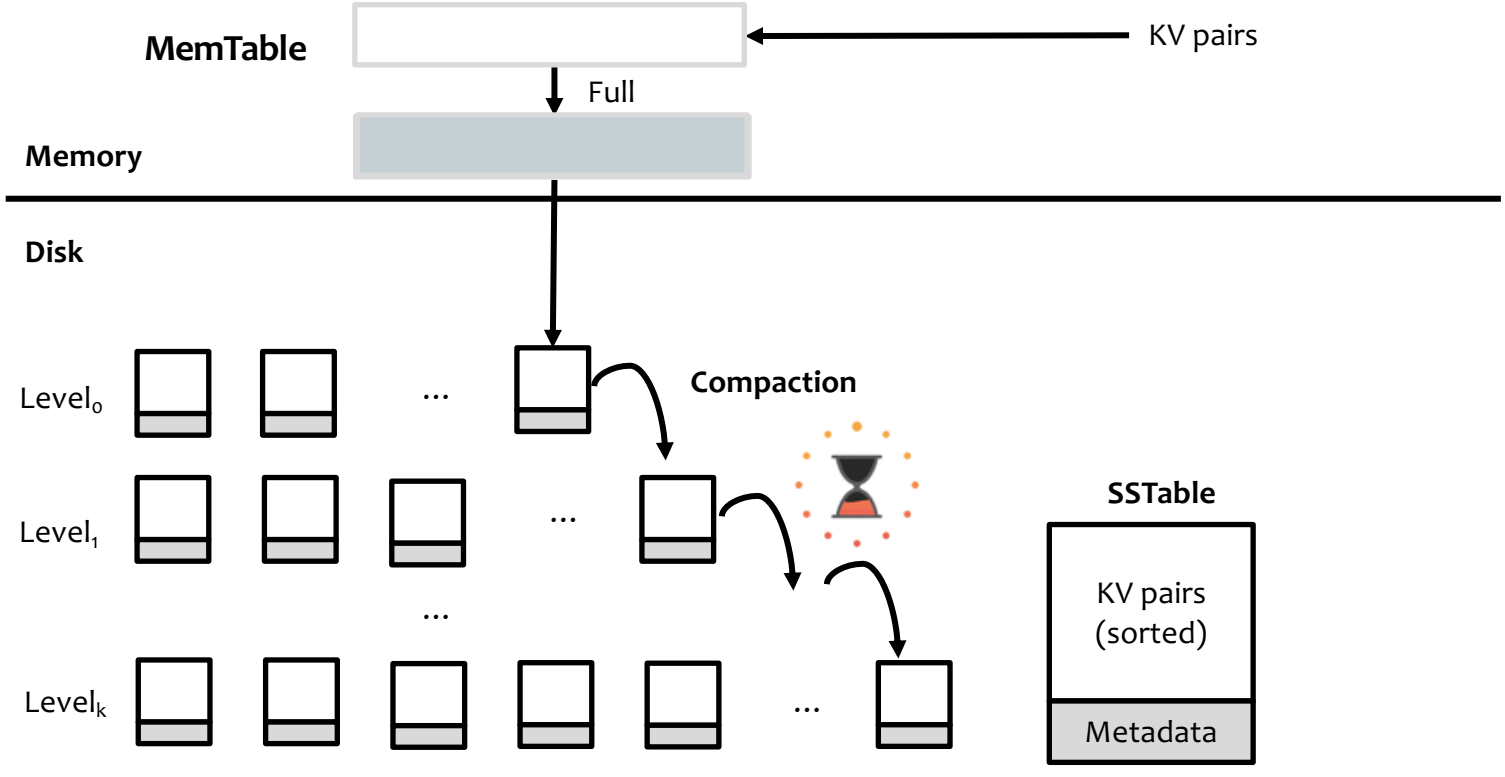
Write stall caused by compaction in LSM Tree



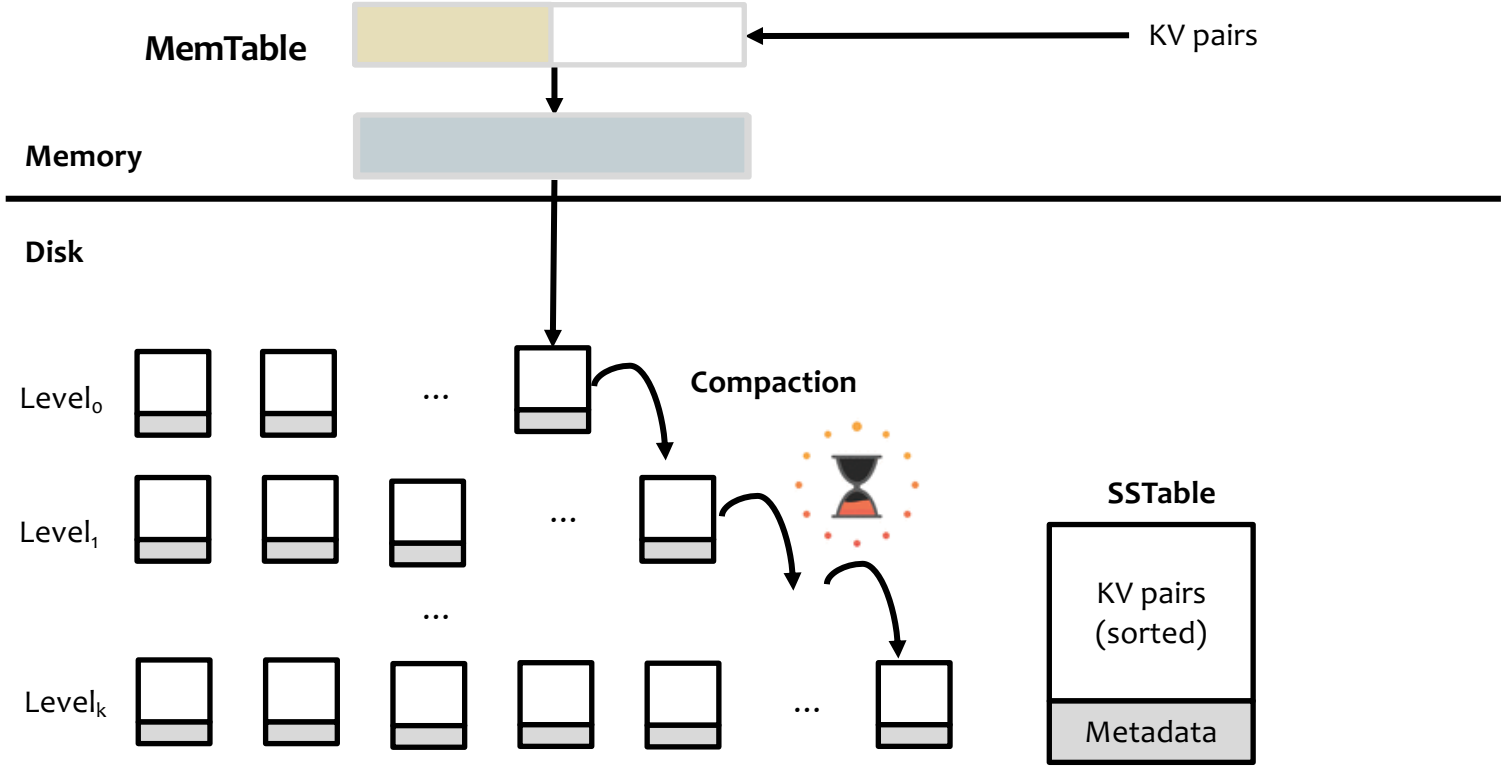
Write stall caused by compaction in LSM Tree



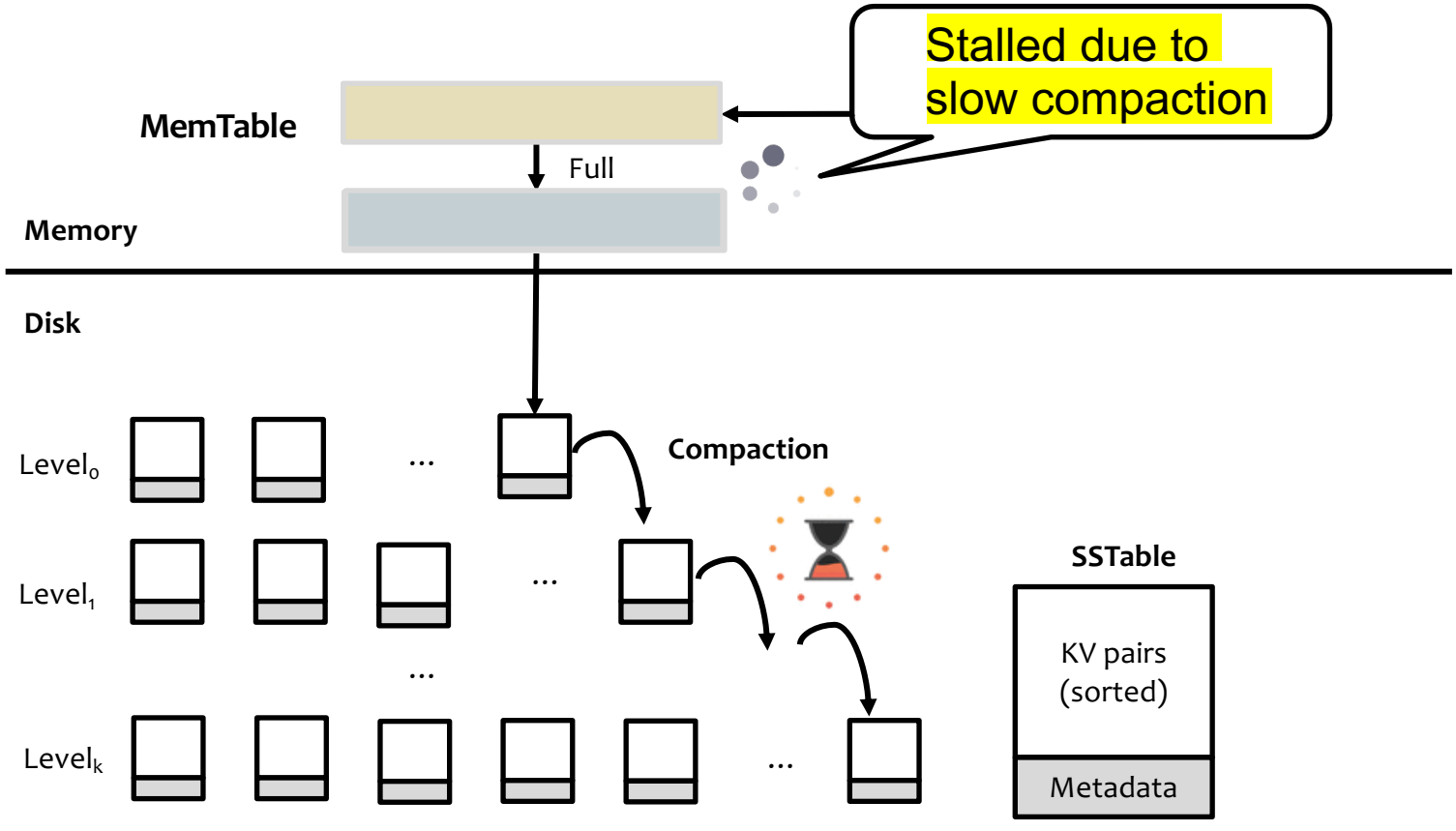
Write stall caused by compaction in LSM Tree



Write stall caused by compaction in LSM Tree



Write stall caused by compaction in LSM Tree



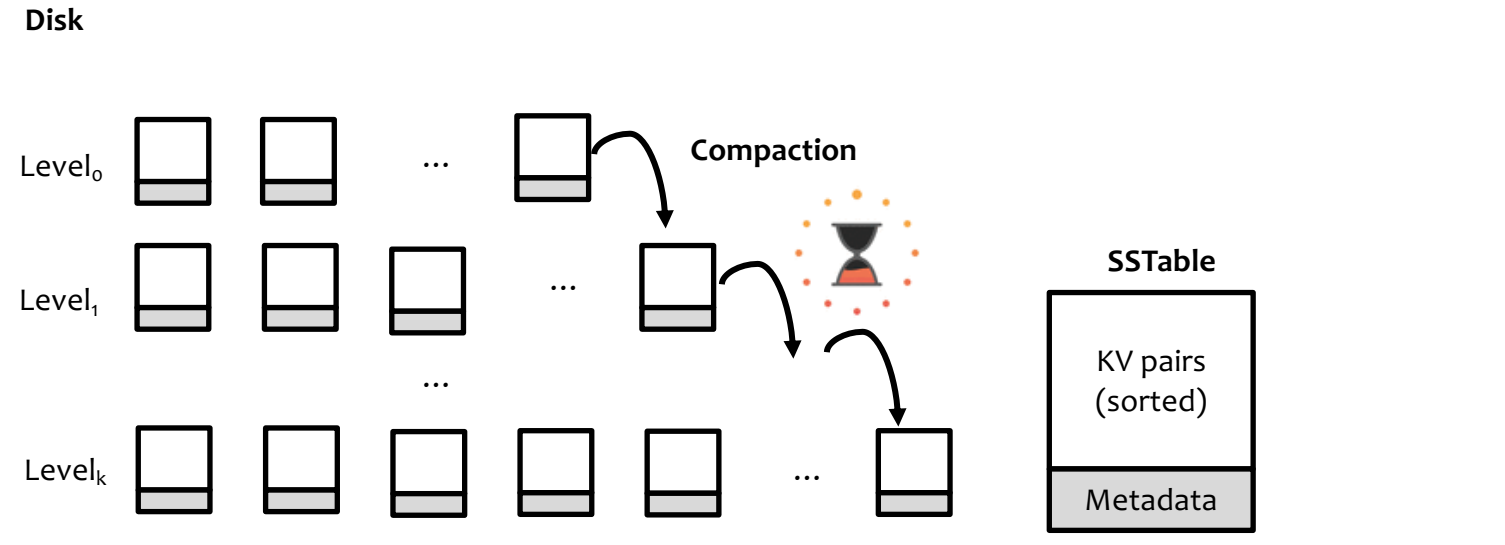
How to avoid write stall?

In principle, hard to prevent

Can only alleviate

– E.g., speed up compaction & merge process with advanced hardware

但是多路归并排序算法已经很成熟，很难在优化。

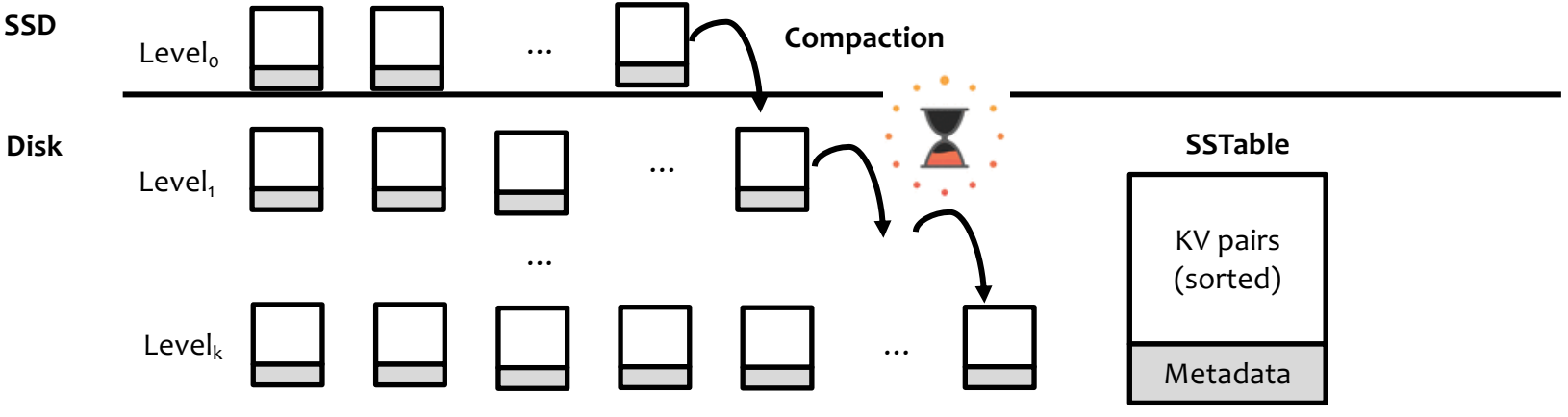


In principle, hard to prevent

Can only alleviate

- E.g., speed up compaction & merge process with **advanced hardware**
- **Observation**: SSD is much faster than disk on storage
  - Using it to store up-layer SSTables

但是SSD空间相比于Disk还是比较小的。



LSM Tree Summary

Good when

- Massive dataset
- Rapid updates/insertions
- Fast lookups

Compared with B-Tree

- **Pros**: good write performance due to sequential writes
- **Cons**: additional compaction process, possible slow range queries, write stall caused by the compaction, **slow lookup for non-existent key**

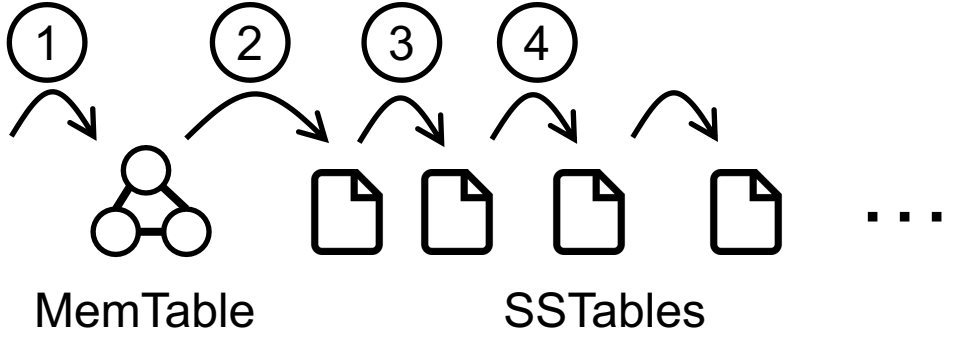
Slow lookup for non-existent key

Recall: how LSM Tree lookup keys

1. Checks the MemTable
2. If misses, checks the latest SSTable
3. If still misses, checks the next older SSTable
4. …

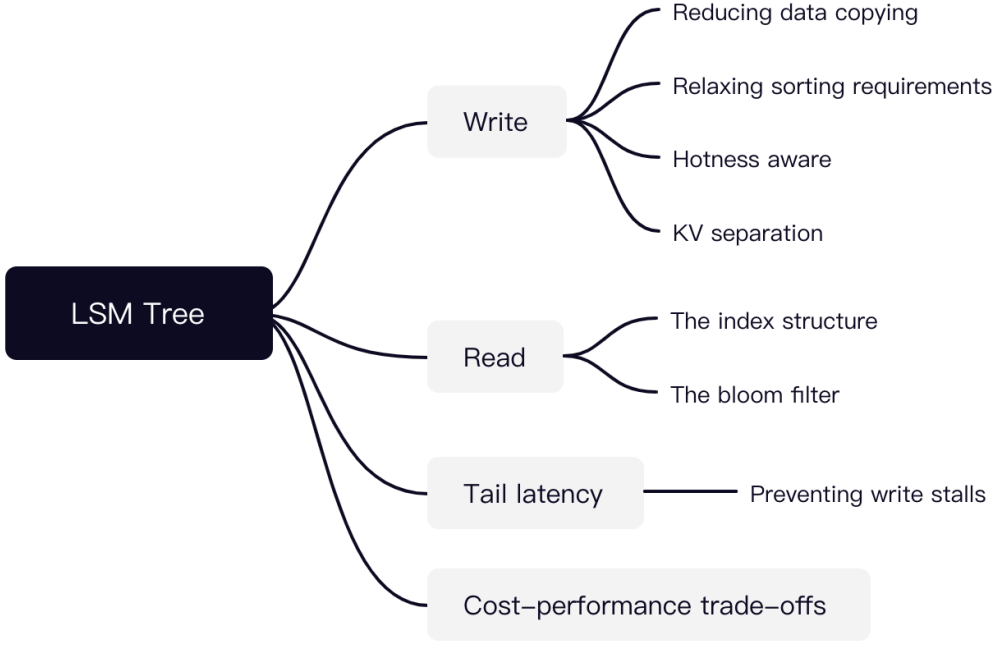
Question: what if the key non-exist?

Will lookup all the files!



LSM Tree is a hot research topic today

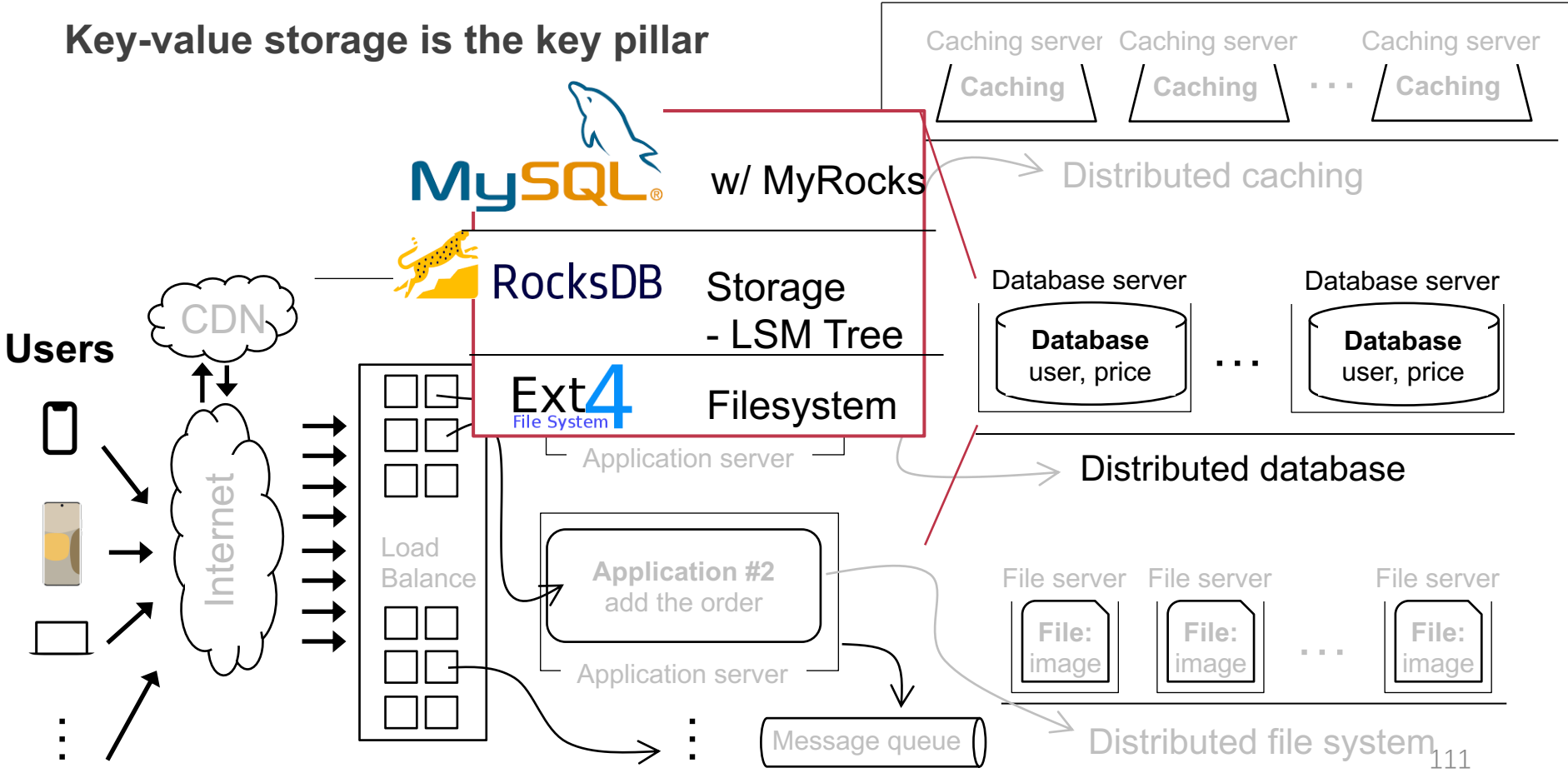
Many possible directions



Key-value storage is a key component in large-scale website

Review: large-scale websites

Key-value storage is the key pillar



Distributed key-value storage

Distributed key-value storage

Make KV store distributed (see later lectures)

- RPC + key-value storage = distributed key-value storage!
  - See the next lecture
- We can also shard the data across multiple nodes
  - i.e., high scalability

Key challenge:

- How to find the data?
  - E.g., consistent hashing (see later lectures)

Other problems:

- Fault tolerance (see later lectures)
- Availability, replication & consistency (see later lectures)

Summary of this lecture

Key-value store is an important component in computer systems

Build key-value store over file system

- Log-structured file
- Indexing
- LSM Tree