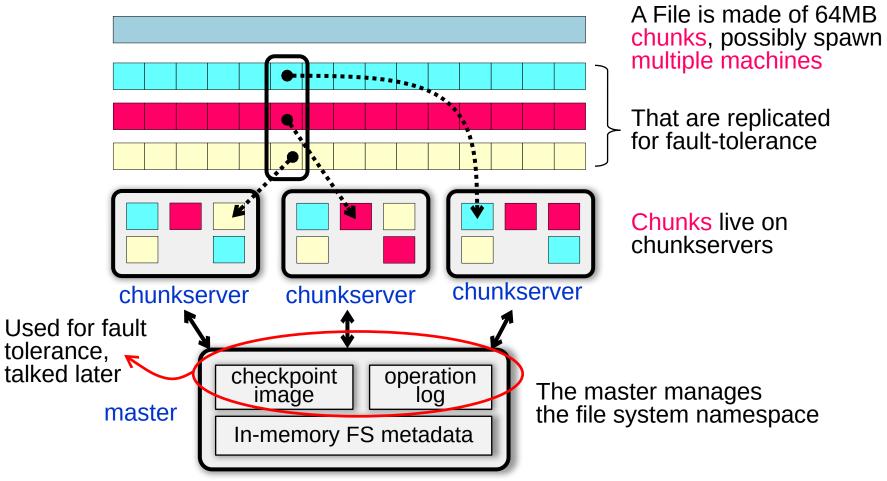# Designing efficient
key-value store/storage (KVS)
on a single node

IPADS, Shanghai Jiao Tong University

https://www.sjtu.edu.cn

Credits: Niv Dayan ， JiFei@IPADS

Review: GFS

A File is made of 64MB chunks, possibly spawn multiple machines

That are replicated for fault-tolerance

Chunks live on chunkservers

chunkserver   chunkserver   chunkserver

Used for fault tolerance, talked later

master

| checkpoint image | operation log |

In-memory FS metadata

The master manages the file system namespace

Reading a file in GFS (very similar to the naïve DFS)

Reading a file is simple in GFS

1. Contact the **master**

2. Get file's **metadata**: chunk handles

3. Get the **location** of each of the chunk handles
   – Multiple replicated chunkservers per chunk

4. Contact any **available** chunkserver for chunk

Writing a File in GFS (More complicated due to replication)

Less frequent than reading

- But is more complex, because we need to deal with the consistency issues
- GFS adopts a **relaxed consistency model (see later lectures)**
- E.g, may have inconsistency state, but work well for their apps
- Benefits: simple & efficient to implement

Master grants a chunk lease to one of the replicas

- This replica will be the **primary** chunkserver
  - The only one that can modify the chunk
- Primary can request extensions (of lease), if needed
  - Master increases the chunk version number and informs replicas

Naming in GFS: simple flat naming

No per-directory data structure like most file systems

– E.g., directory file contains names of all files in the directory
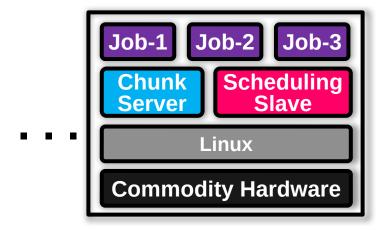
No aliases (i.e., no hard or symbolic links)

Namespace is a single lookup table

– Maps pathnames to metadata

Review: GFS in Google cluster

Google cluster environment

- Core services: GFS + cluster scheduling system
- Typically, 100s to 1000s of active jobs
- 200+ clusters, many with 1000s of machines
- Pools of 1000s of clients
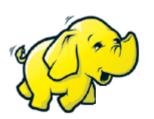- 4+ PB filesystems, 40GB/s read/write loads

Job-1	Job-2	Job-3
Chunk Server	**Scheduling Slave**	
Linux		
Commodity Hardware		

GFS Master — File system Master

Scheduling Master — Job Scheduler

HDFS: another popular (open-source) DFS

Hadoop **D**istributed **FS**

Primary storage system for **Hadoop** apps

A framework that allows for the distributed processing of large data sets across clusters of computers
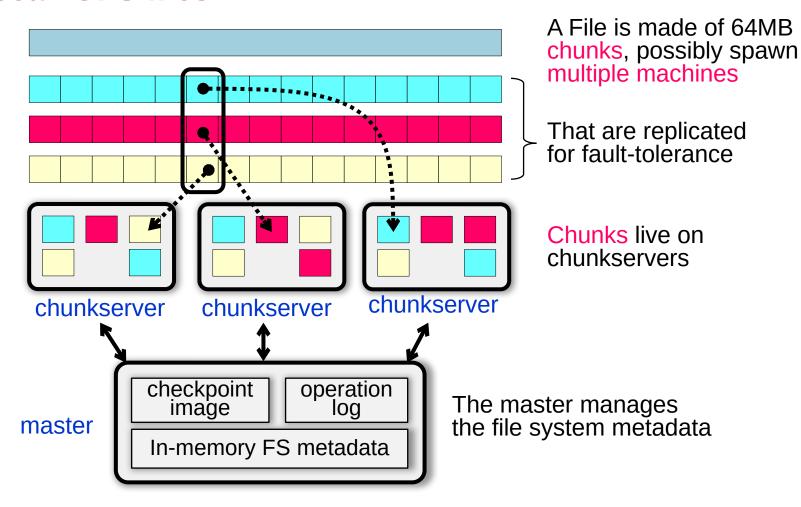
Design Goals & Assumptions of HDFS

HDFS is an open source (Apache) implementation inspired by GFS design
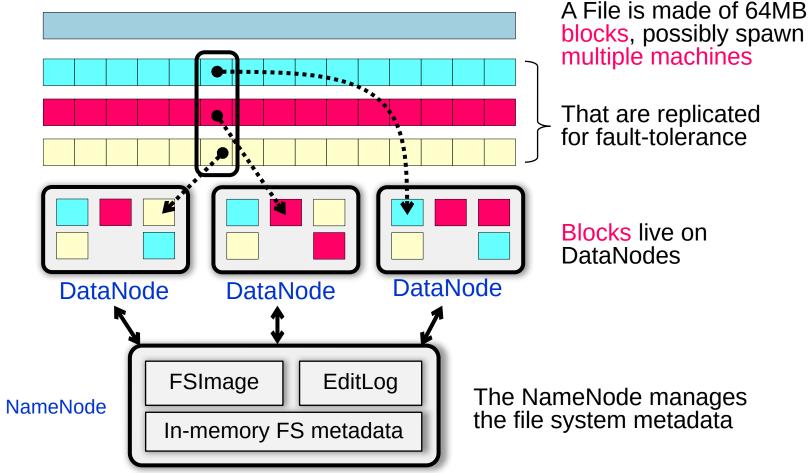
Similar goals as GFS

– Run on commodity hardware

– Highly fault tolerant

– High throughput & large-scale deployments

– …

Recall GFS files

A File is made of 64MB chunks, possibly spawn multiple machines

That are replicated for fault-tolerance

Chunks live on chunkservers

chunkserver   chunkserver   chunkserver

checkpoint image

operation log

master

In-memory FS metadata

The master manages the file system metadata

HDFS files

A File is made of 64MB blocks, possibly spawn multiple machines

That are replicated for fault-tolerance

Blocks live on DataNodes

DataNode   DataNode   DataNode

NameNode

FSImage   EditLog

In-memory FS metadata

The NameNode manages the file system metadata

Summary

Designing distributed file system (DFS) is not simply

– Single-node file system + RPC

Many design choices for performance, consistency model & failure handling

– Interface

– Caching

– Access model

Two case studies

– **NFS**: transparent access files on a remote server

– **GFS**: highly-scalable & fault-tolerant DFS optimized for Google's workload

GFS or NFS are not Perfect

NFS

- Can **not scale**
- Is **not fault-tolerant**
- But is well-enough for many workloads, e.g., sharing the data for experiments in our lab ◀◀

GFS

- **Relaxed consistency model**: the results of concurrent mutations are undefined
- Single-node master: single point of failures (though next-generation of GFS refines this, with more advanced techniques developed later)
- Work well in Google's datacenter workloads

GFS or NFS are not Perfect

NFS

- Can **not scale**
- Is **not fault-tolerant**
- But is well-enough for many workloads, e.g., sharing the data for experiments in the lab

GFS

- **Relaxed consistency model**: the results of concurrent mutations are undefined

  We will see system principles to cope with them in future lectures.
- Single-node master: single point of failures (though next-generation of GFS refines this)
- Work well in Google's datacenter workloads

  For other workloads (e.g., Database) may not sufficient.

Review: Large-scale websites on distributed systems

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

– **File system** runtimes, etc.

Caching server Caching server   Caching server
Caching   **Caching**   · · ·   **Caching**

Distributed caching

Users

CDN

Internet

Load Balance

Application #1
generate the page

Application server

Application #2
add the order

Application server

Database server   Database server
Database
user, price   · · ·   **Database**
user, price

Distributed database

File server File server   File server
File:
image   **File:**
image   · · ·   **File:**
image

Distributed file system

Message queue

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 API sufficient?

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

Database

File system

Key-value store

Key-value storage (KVS): system w/ a simpler API

Storage abstraction:

– 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)

Application-level Interface (API)

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

Why key-value storage instead of filesystem?

1. Filesystem is designed for bulk storage

– E.g., images, videos KV

For small data, we need a way to store them in one file to improve storage efficiency

– But question remains: how can we find data in a file efficiently?

2. Developers need a way to map abstract name to the concrete data

– E.g., "item name" -> number of views

– They don't need to take care of directory, etc.

Key-value store

Naïve solution to implement a KVS atop of filesystem

Storage abstraction remapping:

– 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 ◀◀

Application-level Interface (API)

– Get(K) -> V is similar to OPEN(…) + READ(…)

– Insert(K,V) -> is similar to CREATE(…) + WRITE(…)

– Etc.

Naïve implement a KVS on the filesystem 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)

  - Why? To match the underlying device block size

- On the other hand, the value in key-value store can be small (e.g., 64B)

Naïve implement a KVS on the filesystem 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)

Idea: using one or few file to store key-value store data
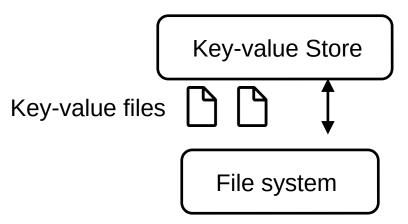
- Different key-values can pack into the same disk block
- Reduce system call overhead: open file once, serve all the requests then

Building Key-value Storage on File System

Why builds upon the file system?

– We still need a system to interact with the disk hardware!

– Though modern KVS may also bypass the file system, but is uncommon

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

Key-value Store

Key-value files

File system

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:

Key   Value

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

How to implement the update? It is simple:

– 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"]}'
...
```

How to implement the update? It is simple:

– 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**

– **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:

Key   Value

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

How about the insertion? The same s the update

– Question: what about deletion?

- Append a NULL entry and later garbage collected (will talk later)

BTW: File format of naïve KVS & log file

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.
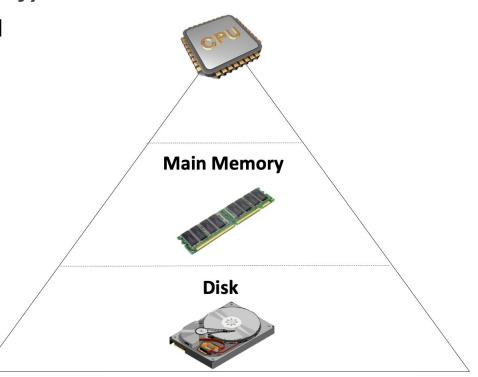
Storage Hierarchy

Fast device (e.g., Main Memory)

- Fast, but capacity is limited
- Expensive
- Volatile

Disk has huge capacity

- Cheap & slow
- Non-volatile

CPU

Main Memory

Disk

Storage Hierarchy: asymmetric performance of disk

CPU <-> Memory

– ~= 100ns
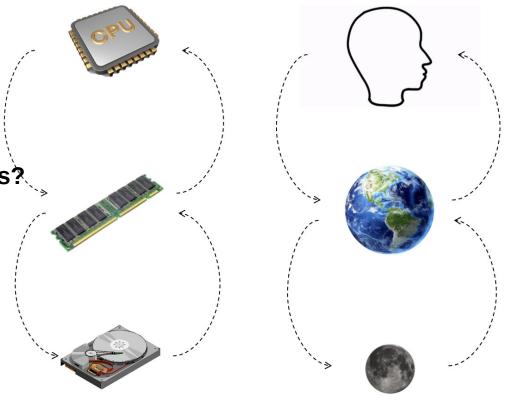
Memory <-> Disk

– ~= 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)

Principle: 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** instead of correctness
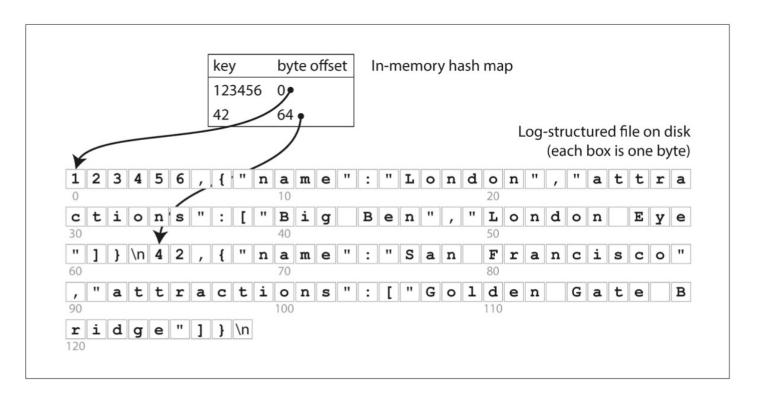
Example of index data structure:

- B+Tree, HashTable, etc
- The inode table in Lab1 is essentially an index; but is extremely simple.
  What are the trade-offs of an inode table?

First try of index: an 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 **riak**KV

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

Hash Index++: Put it onto the disk

Question: how to choose the appropriate hash index data structure?

Many hash data structures exist

- Their performance differences are not so different in an in-memory setting for KVS

- But on disk, the things are different, because accessing the disk is orders of magnitude slower than the memory

Different hash indexes have different trade-offs when stored on disk

- For a specific index, we need make some optimization to make it suit for the disk

Linked-list based hash index

The simplest data structures

Pros:
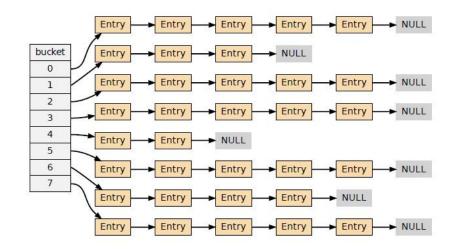
– Easy for implementation

Cons (of the easiest implementation):

– Bad read performance under hash collision

- Requires traverse the linked buckets in random I/O

– Bad insert performance under hash collision
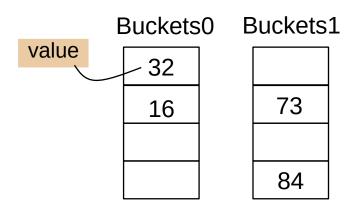
- Similar problem as the read

Question: can you make it better?

– Simple idea: put the entries in the same file block to reduce the chain length
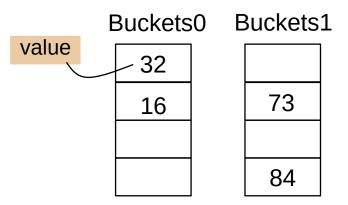
Cuckoo hashing

Cuckoo hashing

– Maintain two hash buckets

Buckets0   Buckets1

| value |

| 32 |
| 16 |
| |
| |

| |
| 73 |
| |
| 84 |

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)]`
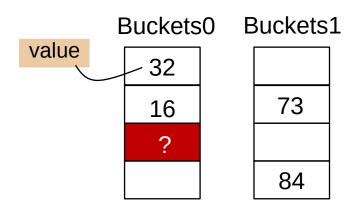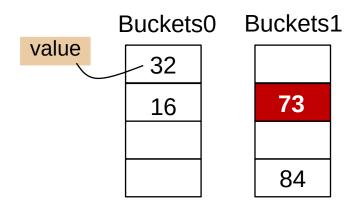
Buckets0   Buckets1

value → 32

16   73

84

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)
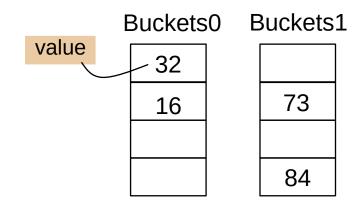
– **Buckets0[H0(73)] = 2**

Buckets0   Buckets1

value

| 32 |
| 16 | 73 |
| ? |
| | 84 |

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**

Buckets0   Buckets1

value

| 32 |
| 16 |
| |
| |

| |
| **73** |
| |
| 84 |

Cuckoo hashing

Cuckoo hashing

– Maintain two hash buckets

Update()

– Similar as `Get()`

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

Buckets0   Buckets1

value

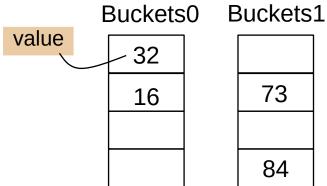| 32 |
16	73
	84

What about insert?

Cuckoo hashing

Cuckoo hashing

– Maintain two hash buckets

Insert()

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

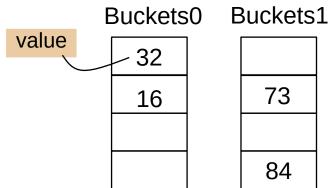 • Question: what if no available space?

Buckets0   Buckets1

value → 32
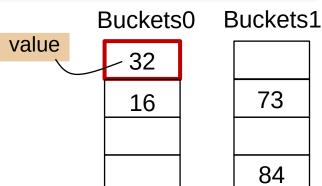
16   73

84

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

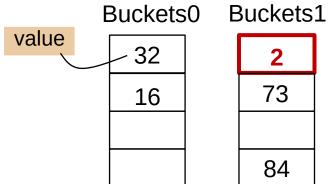– Maintain two hash buckets

Insert()

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

Example: Insert(2)

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

Can we insert here?

Buckets0   Buckets1

value

32		
16		73
		84

Cuckoo hashing

Cuckoo hashing

– Maintain two hash buckets

Insert()

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

Example: Insert(2)

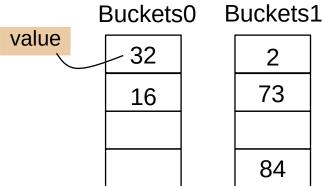– **Buckets0[H0(2)] = 0 & Buckets1[H1(2)] = 0**

– Insert here!

Buckets0   Buckets1

value

32	**2**
16	73
	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)] =
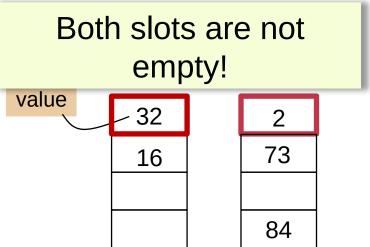 0

Buckets0   Buckets1

value

32		2
16		73
		84

Cuckoo hashing

Cuckoo hashing

– Maintain two hash buckets

Insert()

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

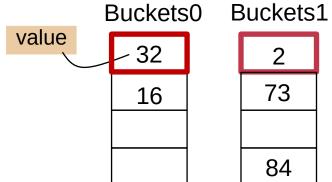**Example: Insert(9)**

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

Both slots are not empty!

value

32		2
16		73
		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
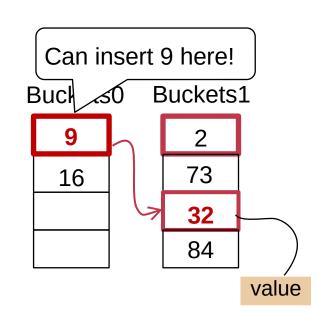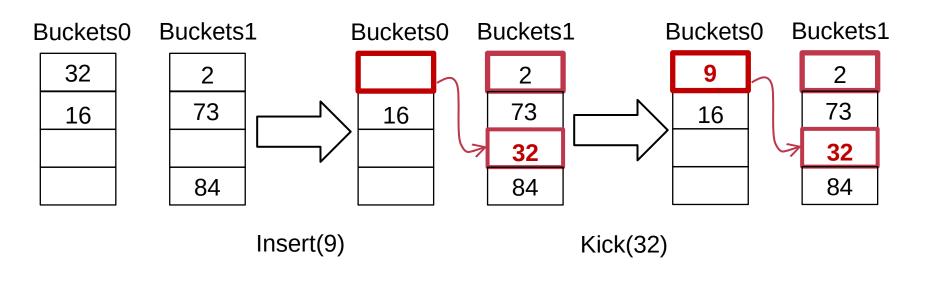 create space for the new insertions

Buckets0   Buckets1

value

32		2
16		73
		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
  create space for the new insertions

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

  - Assume: H1(32) = 2

Can insert 9 here!

Buckets0   Buckets1

9	2
16	73
	32
	84

value

Cuckoo hashing

Buckets0	Buckets1
32	2
16	73
	84

Buckets0	Buckets1
	2
16	73
	32
	84

Insert(9)

Buckets0	Buckets1
9	2
16	73
	32
	84

Kick(32)

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

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

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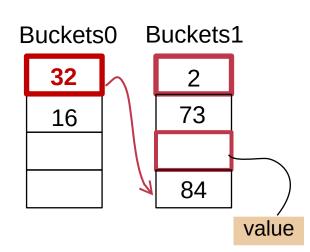
Cuckoo hashing

Buckets0　　Buckets1

value

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

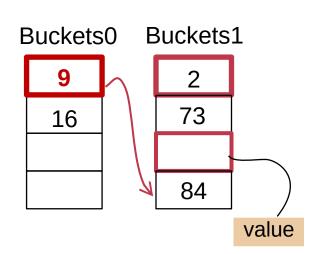　　• **Question: what if H1(32)　= 2 = 3?**

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~~
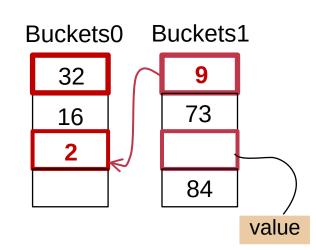
    • Kick 2 (Buckets1[0]) to another place

Buckets0   Buckets1

9		2
16		73
		84

value

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

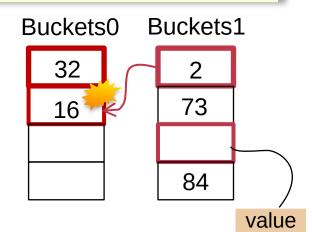 • Kick 2 (Buckets1[0]) to another place

  • Assume: H0(2) = 2

Buckets0   Buckets1

32	**9**
16	73
2	
	84

value

Cuckoo hashing

Buckets0   Buckets1

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

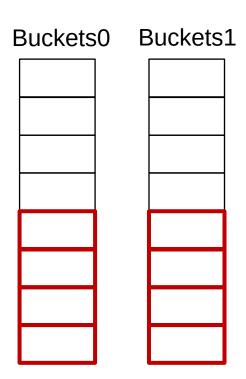- Assume: `H0(2)` = ~~2~~ **1** What if that happens?

32		2
16		73
		84

value

56

Cuckoo hashing

Buckets0   Buckets1

Cuckoo hashing

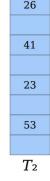– Maintain two hash buckets

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**)

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 (!)

T_1 T_2

Selecting a proper index for KV store is non-trivial

Not just choosing a data structure

Trade-offs of selecting indexes （ The RUM Conjecture[1]）

- **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

Naïve in-memory hash index has a limited capacity

– Can put it on the disk, but needs to carefully redesign the hash scheme or carefully select an appropriate hash scheme

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

Naïve in-memory hash index has a limited capacity

– Can put it on the disk, but needs to carefully redesign the hash scheme or carefully select an appropriate hash scheme

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 design makes updates fast
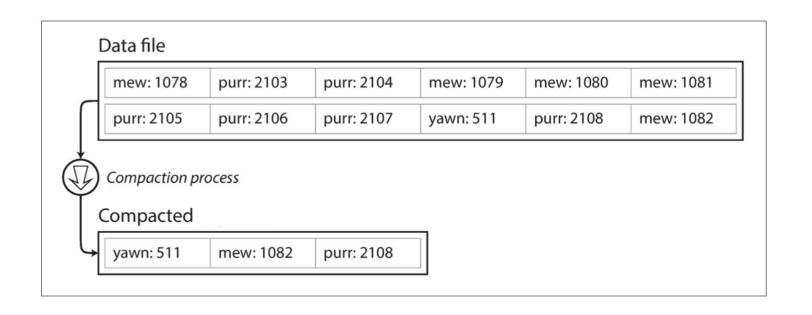
- Only append to the file

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
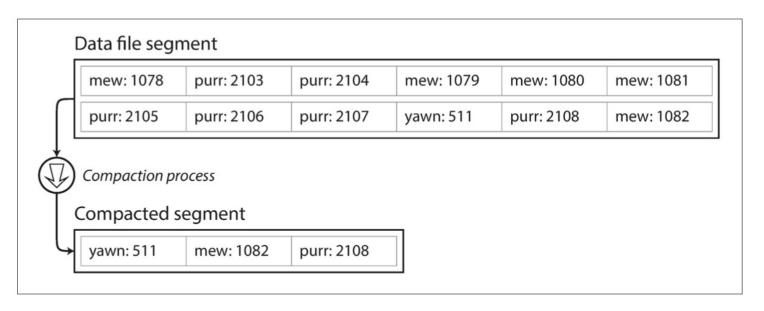
- A deleted value is also left on the file

How to prevent a file from growing forever?

Solution: compaction

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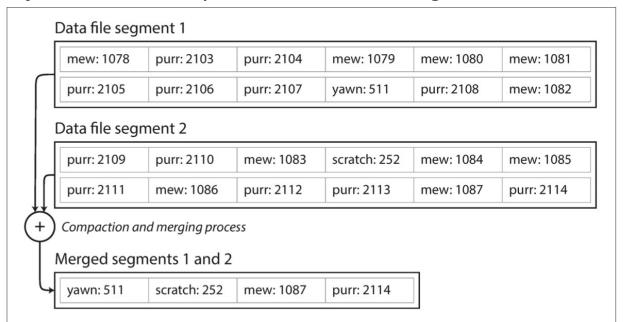- **Question**: what if the file is too large?

Data file

mew: 1078	purr: 2103	purr: 2104	mew: 1079	mew: 1080	mew: 1081
purr: 2105	purr: 2106	purr: 2107	yawn: 511	purr: 2108	mew: 1082

Compaction process

Compacted

yawn: 511	mew: 1082	purr: 2108

How to prevent a file from growing forever?

Solution: compaction with **segmentation**

– Break a file into **fixed-sized** segments

– If a segment is full, then create a new file for insertions

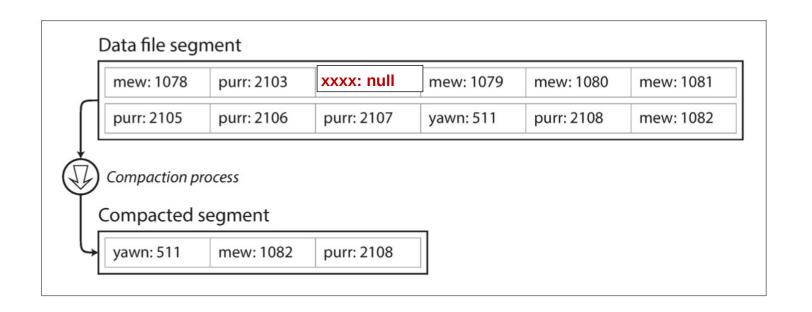– Can control the compaction **granularity** with the segment size

Data file segment

mew: 1078	purr: 2103	purr: 2104	mew: 1079	mew: 1080	mew: 1081
purr: 2105	purr: 2106	purr: 2107	yawn: 511	purr: 2108	mew: 1082

Compaction process

Compacted segment

yawn: 511	mew: 1082	purr: 2108

How to prevent a file from growing forever?

Solution: compaction with **segmentation + Merge**

– **Compact**: remove duplicated records in a segmented file

– **Merge**: merge two compacted segments into a new one

  • Why? Because compaction makes the segment smaller

Data file segment 1

mew: 1078	purr: 2103	purr: 2104	mew: 1079	mew: 1080	mew: 1081
purr: 2105	purr: 2106	purr: 2107	yawn: 511	purr: 2108	mew: 1082

Data file segment 2

purr: 2109	purr: 2110	mew: 1083	scratch: 252	mew: 1084	mew: 1085
purr: 2111	mew: 1086	purr: 2112	purr: 2113	mew: 1087	purr: 2114

(+) *Compaction and merging process*

Merged segments 1 and 2

yawn: 511	scratch: 252	mew: 1087	purr: 2114

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

1. **Append a NULL entry**

2. **Garbage-collected during compaction!**

Data file segment

| mew: 1078 | purr: 2103 | **xxxx: null** | mew: 1079 | mew: 1080 | mew: 1081 |
| purr: 2105 | purr: 2106 | purr: 2107 | yawn: 511 | purr: 2108 | mew: 1082 |

Compaction process

Compacted segment

| yawn: 511 | mew: 1082 | purr: 2108 |

Issues of using hash index & segmented log file

Naïve in-memory hash index has a limited capacity

– Can put it on the disk, but needs to carefully redesign the hash scheme or carefully select an appropriate hash scheme

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 for disk data

– 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
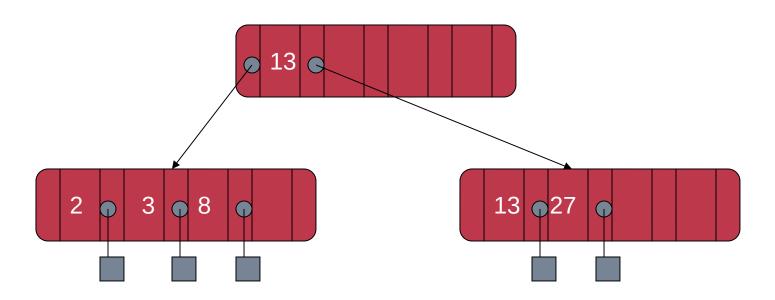
A B-tree is a tree-like data structure (one or a few files)

– 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
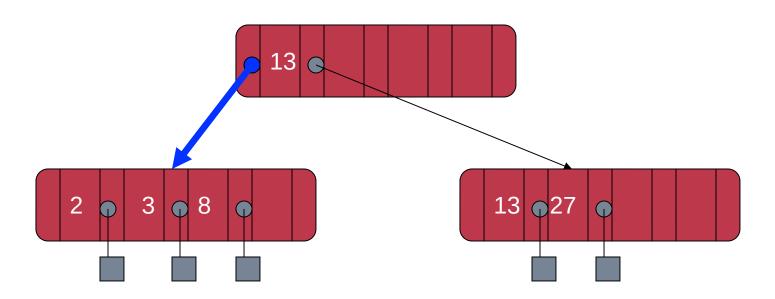
Many variants exist, e.g., B+Tree

– All the leaf nodes of the B-tree must be at the same level

  • Simper to link leaf nodes to support range queries
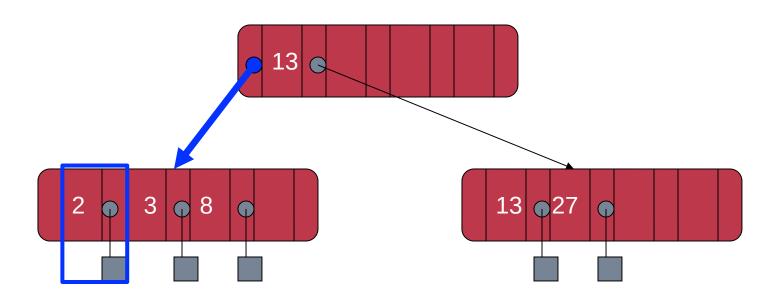
B+Tree indexes example (fanout = 5)

B+Tree indexes example: Get(2)

B+Tree indexes example: Get(2)
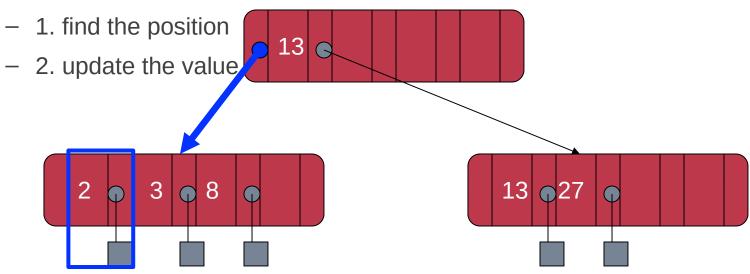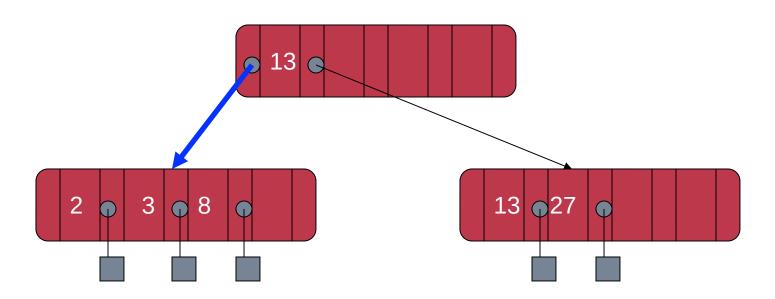
B+Tree indexes example: Update(2)
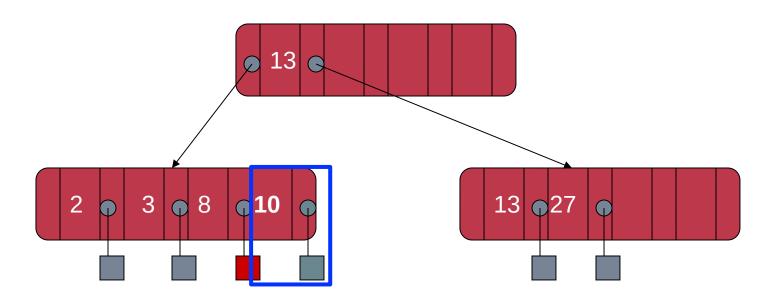
Similar to Get()
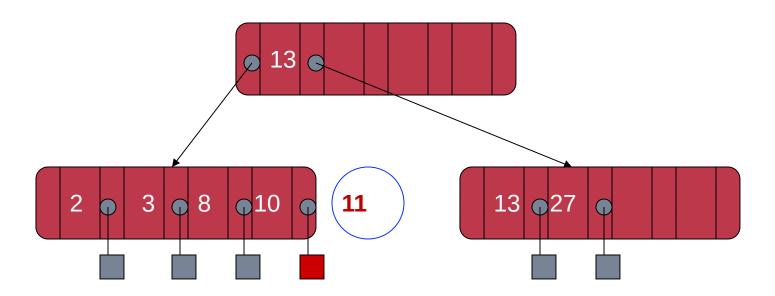
- 1. find the position
- 2. update the value

B+Tree indexes example: insert(10)

B+Tree indexes example: insert(10)
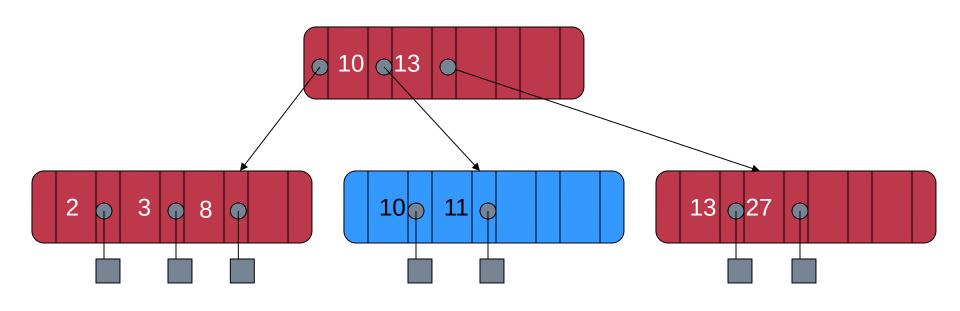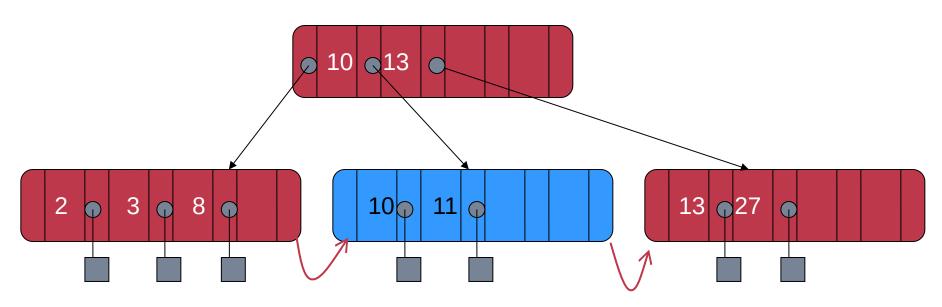
B+Tree indexes example: insert(11)

B+Tree indexes example: insert(11)

Split the nodes!

B+Tree can support efficient range operations
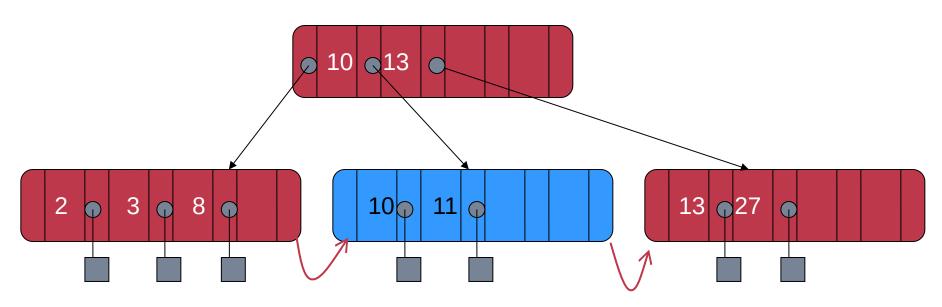
Add a sibling link between the leaves

e.g., SCAN(3,5)

Question: need a B+Tree to index the values in the log?

No. It can store the values in its leave node

– Which does not need compaction & merge

But, what are the costs?

B-Tree indexes on the disk?

Get(K), Insert(K,V) and Update(K,V) are slow

– O(log(n)) random disk accesses

– Typically, is small, since B-Tree indexes uses large node size. But is still a killer for performance

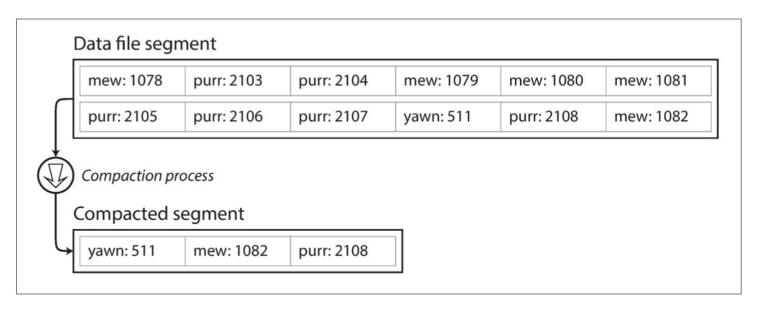**B+Tree indexes are not good enough for insertion-intensive workloads**

– Still **many random disk I/O** involved

– Insertions are common for on-disk storages, because reads are served by other system components, e.g., a cache system like Memached

  • E.g., check lecture 02, step#2 for scaling a website

Can we avoid random access for index updates and insertions ?

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

Data file segment

| mew: 1078 | purr: 2103 | purr: 2104 | mew: 1079 | mew: 1080 | mew: 1081 |
| purr: 2105 | purr: 2106 | purr: 2107 | yawn: 511 | purr: 2108 | mew: 1082 |

Compaction process

Compacted segment

| yawn: 511 | mew: 1082 | purr: 2108 |

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

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

With two extensions

– Key-values in a segment are **sorted** by the key

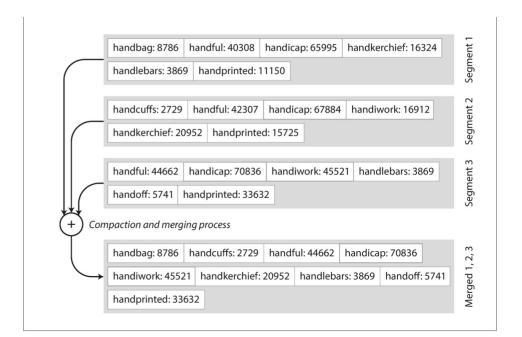– Old segment files are **immutable** and can be **compacted** to save spaces

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

Benefits of SSTable (compared to original log file)
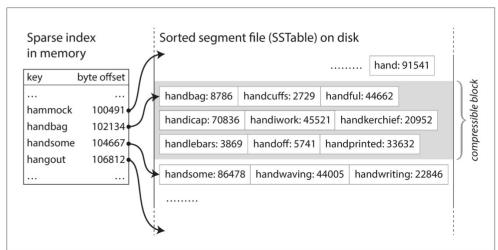
#1. Merging two SSTables is efficient

– Even if the two tables are much larger than the available memory

– Similar to the `mergesort`

Benefits of SSTable (compared to original log file)

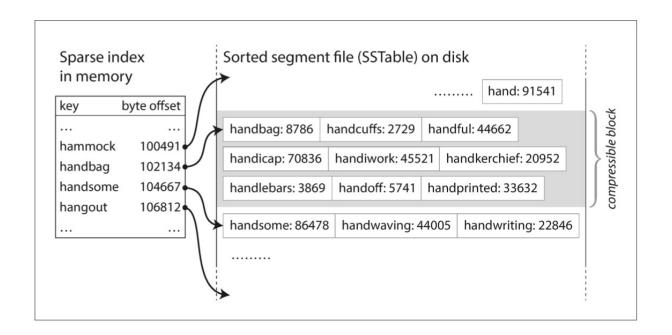**#2. Searching a given key in a SSTable is efficient**

– We can use binary search to lookup a given key

– Or we only need 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
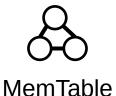
Benefits of SSTable (compared to original log file)

#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 data structure (e.g., a red-black Tree)

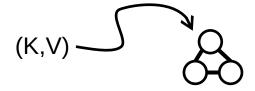– Simple quiz: why not using hash table?

MemTable

How to maintain **SSTable** with sequential writes?

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

Example: insert a (K, V) into SSTable

(1) Insert the (K,V) into the MemTable

(K,V)

MemTable

How to maintain **SSTable** with sequential writes?

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

Example: insert a (K, V) into SSTable
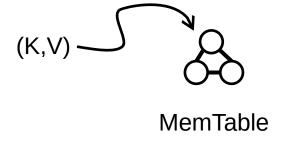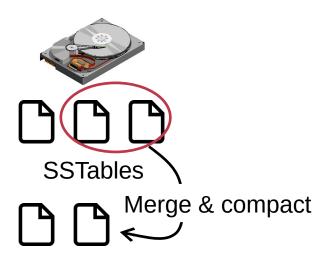
① Insert the (K,V) into the MemTable

(K,V)

MemTable

SSTable

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)
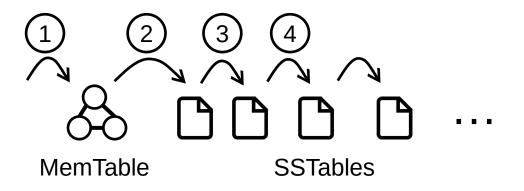
Example: insert a (K, V) into SSTable

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

SSTables

Merge & compact

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. …

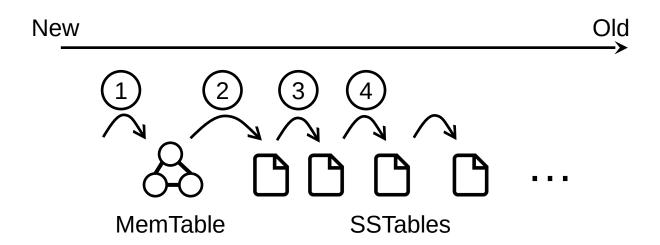**Optimization**: keep a sparse index on each SSTable to accelerate the lookup

MemTable   SSTables

Question: what are the drawbacks of such a design?

Organizing data sequentially has several drawbacks

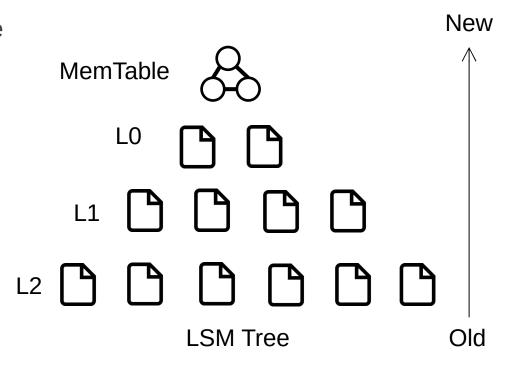**#1. Finding the old value is slow**

#2. Cannot support efficient scan

New ──────────────────────────────────────→ Old

① → ② → ③ → ④ →

MemTable   SSTables   …

LSM Tree organizes SSTables in a hierarchy

The famous Log-structured merge tree

Each layer has the entire KVS data of some time

– Each layer has maximum size
– Except L0, all files in layers are **sorted, and does not have duplicated keys**

New

MemTable

L0

L1

L2

LSM Tree   Old

Hierarchy can speed up old value lookup

Each layer has only one file that store the key (except L0)
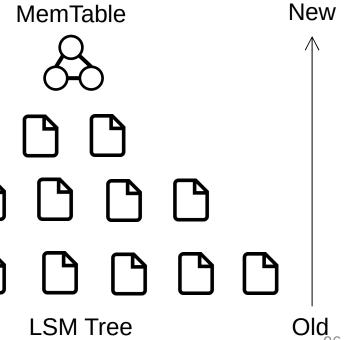
– So we can search only one file per-layer (not per-file search)

Question: how to find the key file?

– Many methods exist

– E.g., store a [min, max] per file in memory

Read upon lower layers

– First identify the file, then look it!
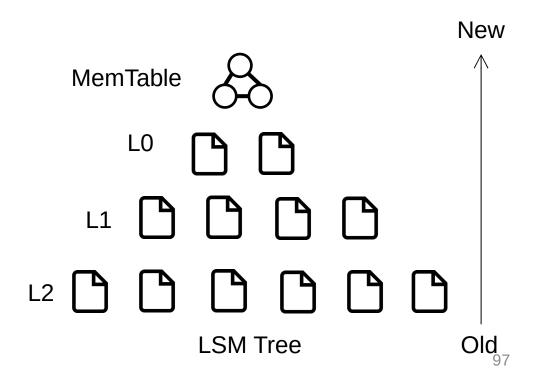
– O(1) access per layer

MemTable

New

L0

L1

L2

LSM Tree

Old

Hierarchy can speed up range query

Store a [min, max] per file

Range query

– Query layer-by-layer

– Then merge the results

New

MemTable

L0

L1

L2

LSM Tree

Old

Range Query in LSM Tree
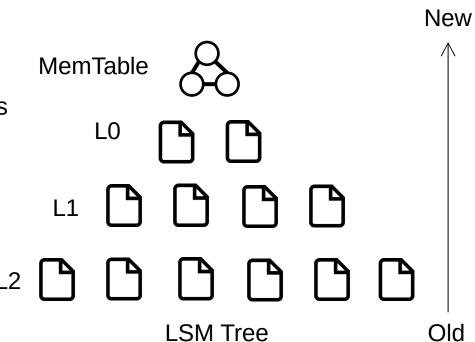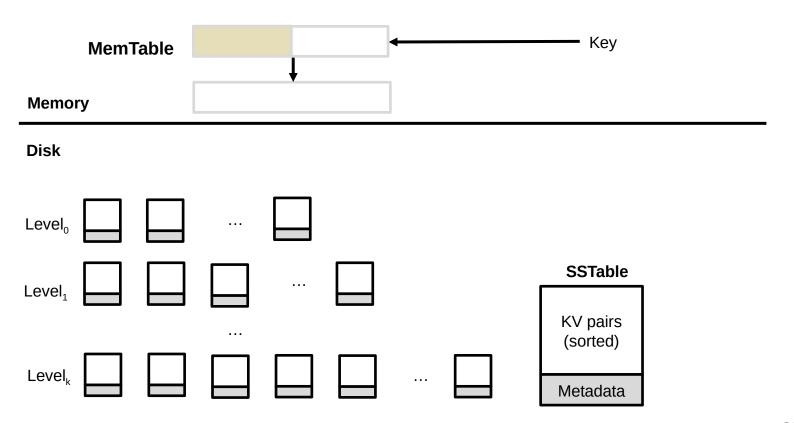
1. Search each layer using **binary search**

2. **Merge** the results of each layer

Not as good as B+Tree,
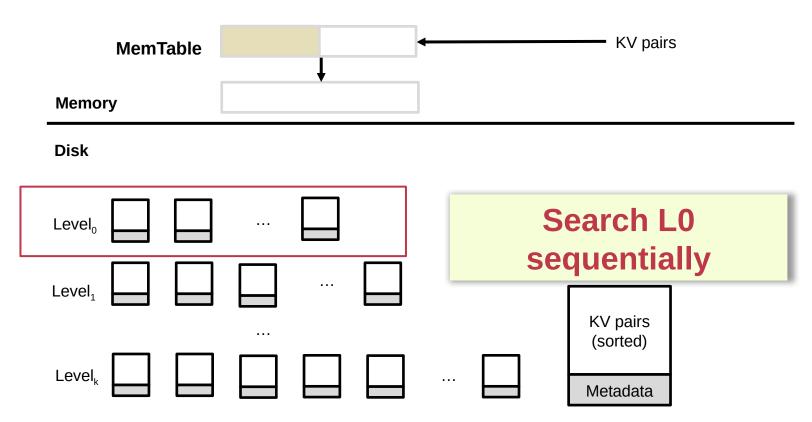
but is much better than hash & logs without hierarchy! !

New

MemTable

L0

L1

L2

LSM Tree

Old
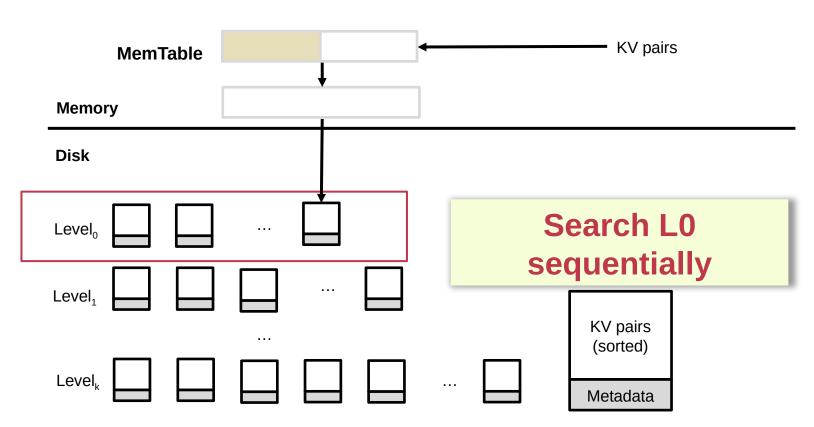
Example: Read in LSM-Tree (Similar to SSTables)

MemTable

Key

Memory

Disk

$Level_0$ ...

$Level_1$ ...

...

$Level_k$ ...

SSTable

KV pairs (sorted)

Metadata

Example: Read in LSM-Tree

MemTable

Memory

KV pairs

Disk

$Level_0$ ...

Search L0 sequentially

$Level_1$ ...

...

$Level_k$ ...

KV pairs
(sorted)

Metadata

Example: Read in LSM-Tree

MemTable

Memory

KV pairs

Disk

$Level_0$ ...

Search L0 sequentially

$Level_1$ ...

...

$Level_k$ ...

KV pairs (sorted)

Metadata

Example: Read in LSM-Tree

MemTable

KV pairs

Memory

Disk

Level$_0$

...

Search lower layer using binary search

Level$_1$

...

SSTable

...

Level$_k$

...

KV pairs
(sorted)

Metadata

Example: insert/update in LSM-Tree

MemTable

Full

Memory

KV pairs

Disk

Level$_0$ ...

Level$_1$ ...

...

Level$_k$ ...

SSTable

KV pairs
(sorted)

Metadata

Example: insert/update in LSM-Tree

MemTable

KV pairs

Memory

Disk

$Level_0$ ...

Compaction

$Level_1$ ...

SSTable

...

$Level_k$ ...

KV pairs
(sorted)

Metadata

What about crash?

MemTable is an in-memory data structure

– So it is vulnerable to machine failure

Goal: a successful insertion will store the data durably

Solution:

– Keep a separate log file the MemTable (may not be sorted)

– Before inserting to the MemTable, adding the KV to the log first (also a sequential write); reply if the log is successful

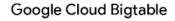– If the machine crashed, reboot it, and reconstruct the MemTable from the log

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

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

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,
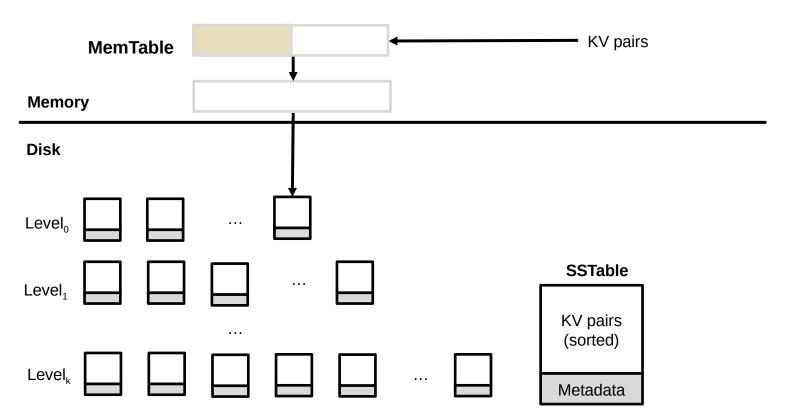 **write stall** caused by the compaction, slow lookup for non-existent key

Write stall caused by compaction in LSM Tree

MemTable

Memory

KV pairs

Disk

Level$_0$ ...

Level$_1$ ...

...

Level$_k$ ...

SSTable

KV pairs
(sorted)

Metadata

Write stall caused by compaction in LSM Tree

MemTable

KV pairs

Memory

Disk

$Level_0$

...

$Level_1$

...

...

$Level_k$

...

SSTable

KV pairs
(sorted)

Metadata

Write stall caused by compaction in LSM Tree

Write stall caused by compaction in LSM Tree

MemTable

KV pairs

Memory

Disk

$Level_0$

…

Compaction

$Level_1$

…

SSTable

…

KV pairs
(sorted)

$Level_k$

…

Metadata

Write stall caused by compaction in LSM Tree

MemTable

Memory

KV pairs

Disk

Level$_0$ … **Compaction**

Level$_1$ … **SSTable**

… KV pairs
(sorted)

Level$_k$ … Metadata

Write stall caused by compaction in LSM Tree

MemTable

Full

Memory

KV pairs

Disk

$Level_0$ ...

Compaction

$Level_1$ ...

SSTable

... KV pairs (sorted)

$Level_k$ ... Metadata

Write stall caused by compaction in LSM Tree

Write stall caused by compaction in LSM Tree

MemTable

Full

Memory

Stalled due to slow compaction

Disk

Level$_0$ ... **Compaction**

Level$_1$ ...

...

Level$_k$ ...

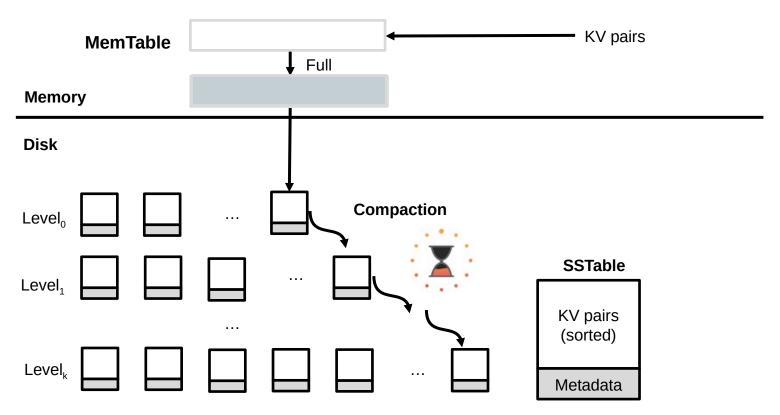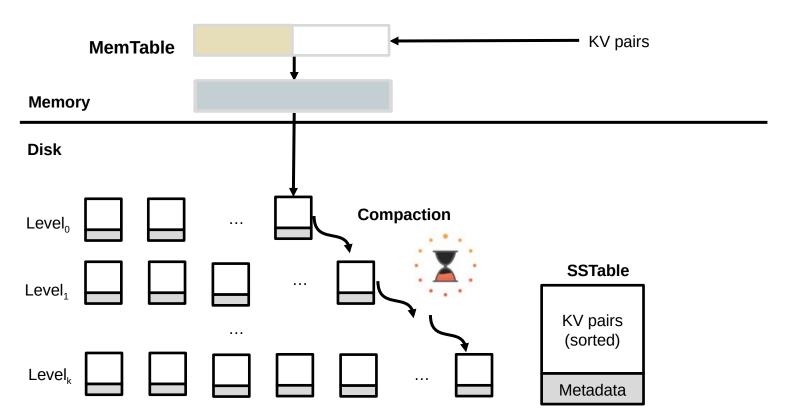**SSTable**

KV pairs (sorted)

Metadata

How to avoid write stall?

In principle, hard to prevent
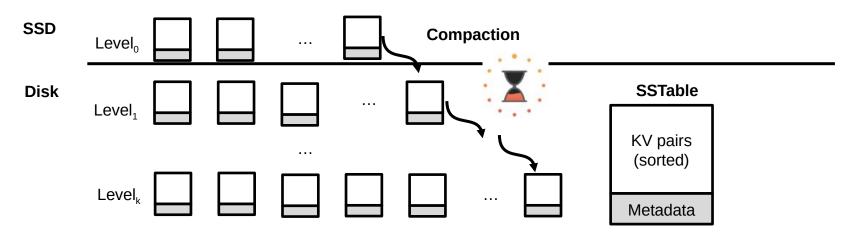
Can only alleviate

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

Disk

Level$_0$

Level$_1$

Level$_k$

…

…

…

…

…

…

Compaction

SSTable

KV pairs
(sorted)

Metadata

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

SSD

$Level_0$ ... Compaction

Disk

$Level_1$ ...

...

$Level_k$ ...

SSTable

KV pairs (sorted)

Metadata

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,

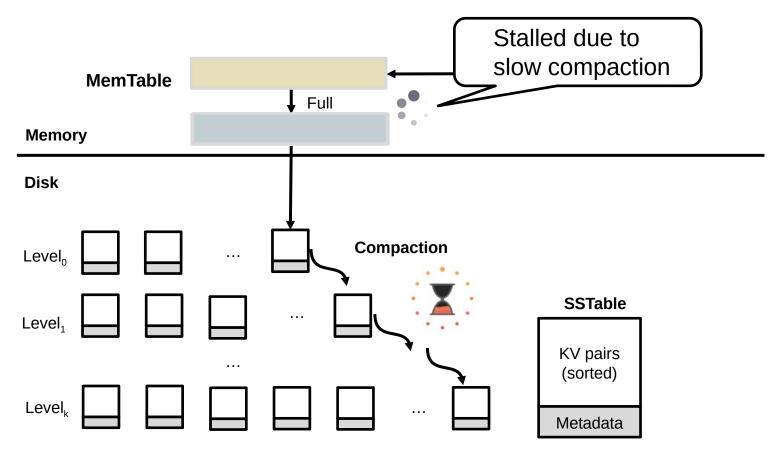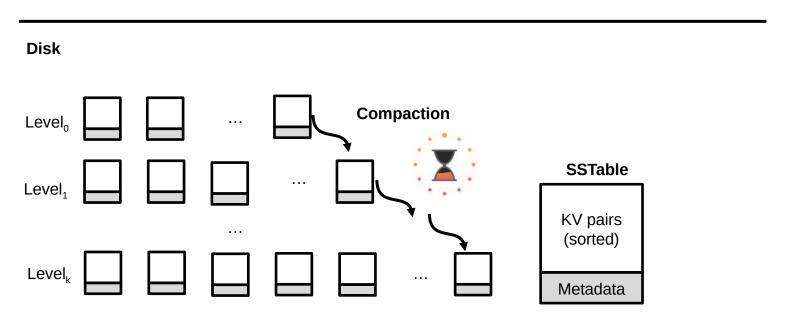write stall caused by the compaction, **slow lookup for non-existent key**
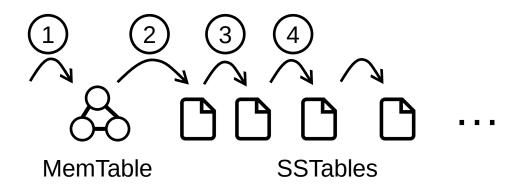
Slow lookup for non-existent key

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. …
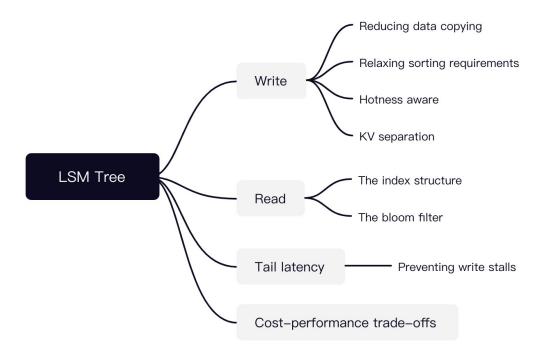
Question: what if the key non-exist?

Will lookup all the files!

1 → 2 → 3 → 4 →

MemTable   SSTables   …
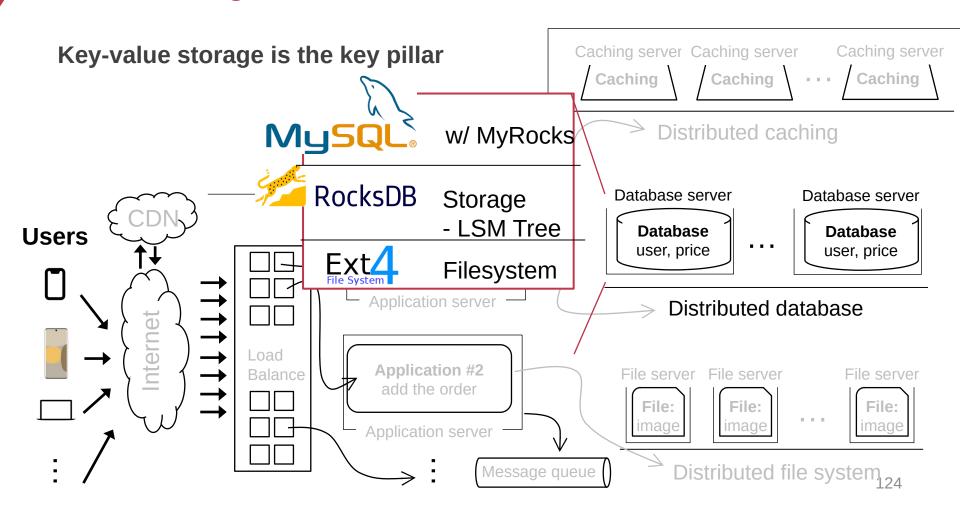
LSM Tree is a hot research topic today

Many possible directions

LSM Tree
- Write
  - Reducing data copying
  - Relaxing sorting requirements
  - Hotness aware
  - KV separation
- Read
  - The index structure
  - The bloom filter
- Tail latency
  - Preventing write stalls
- Cost–performance trade–offs

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

Review: large-scale websites

Key-value storage is the key pillar

Users

CDN

Internet

Load Balance

MySQL w/ MyRocks

RocksDB Storage - LSM Tree

Ext4 Filesystem

Application server

Application #2
add the order

Application server

Message queue

Caching server   Caching server   Caching server

Caching   Caching   ··· Caching

Distributed caching

Database server   Database server

Database user, price   ···   **Database** user, price

Distributed database

File server   File server   File server

File: image   File: image   ··· File: image

Distributed file system

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

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

– Typically built upon a filesystem to simplify disk hardware management

We show how KVS is evolved from log-structured file to the LSM tree

– Log-structured file

– Indexing

– LSM Tree

Key-value Store

Key-value files

File system

File system

Key-value store