Clip slide

RocksDB

Embedded Key-Value Store for Flash and RAM

안미진

RocksDB

Clip slide

Contents

Overview

1. RocksDB Introduction
2. RocksDB Architecture
3. LSM DB
4. RocksDB Compaction

RocksDB Introduction

A persistent key-value store for fast storage environments

- **Open source** based on *LevelDB 1.5*, written in C++
- Key-Value persistent store
- Embedded Library
- Pluggable database
- Optimized for fast storage (flash or RAM)
- Optimized for server workloads
- Get(), Put(), Delete()

Three Basic Constructs

of RocksDB

- **Memtable**
  - in-memory data structure
  - A buffer, temporarily host the incoming writes
- **Logfile**
  - Sequentially-written file
  - On storage

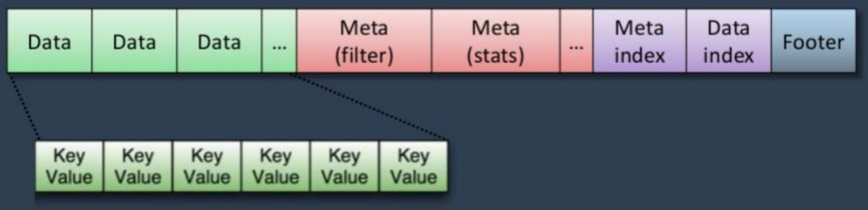
Three Basic Constructs

of RocksDB

- **SSTable(=SSTfile)**
  - Sorted Static Table on storage
  - A file which contains a set of arbitrary, sorted key-value pairs inside
  - Organized in levels
  - Immutable in its life time
  - sorted data → to facilitate easy lookup of keys
  - Storage of the entire database

SSTable-BlockBasedTable

of RocksDB



The default SSTable format in RocksDB

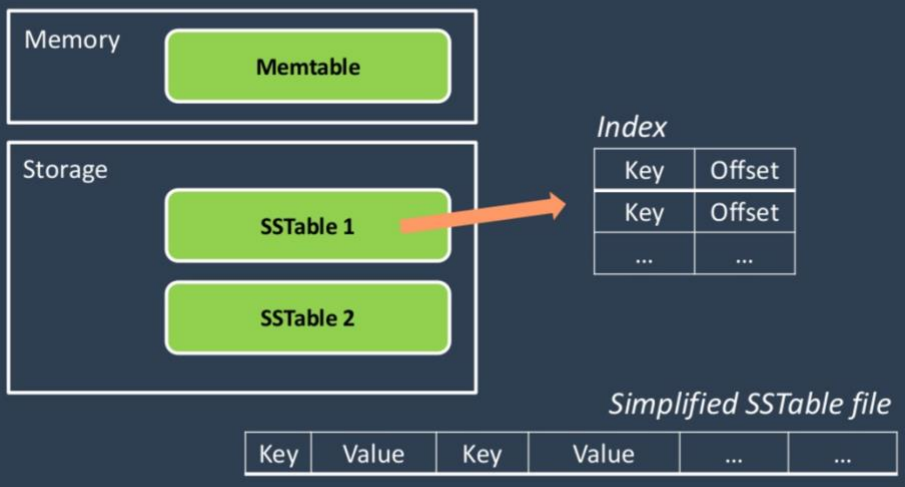
SSTable & Memtable

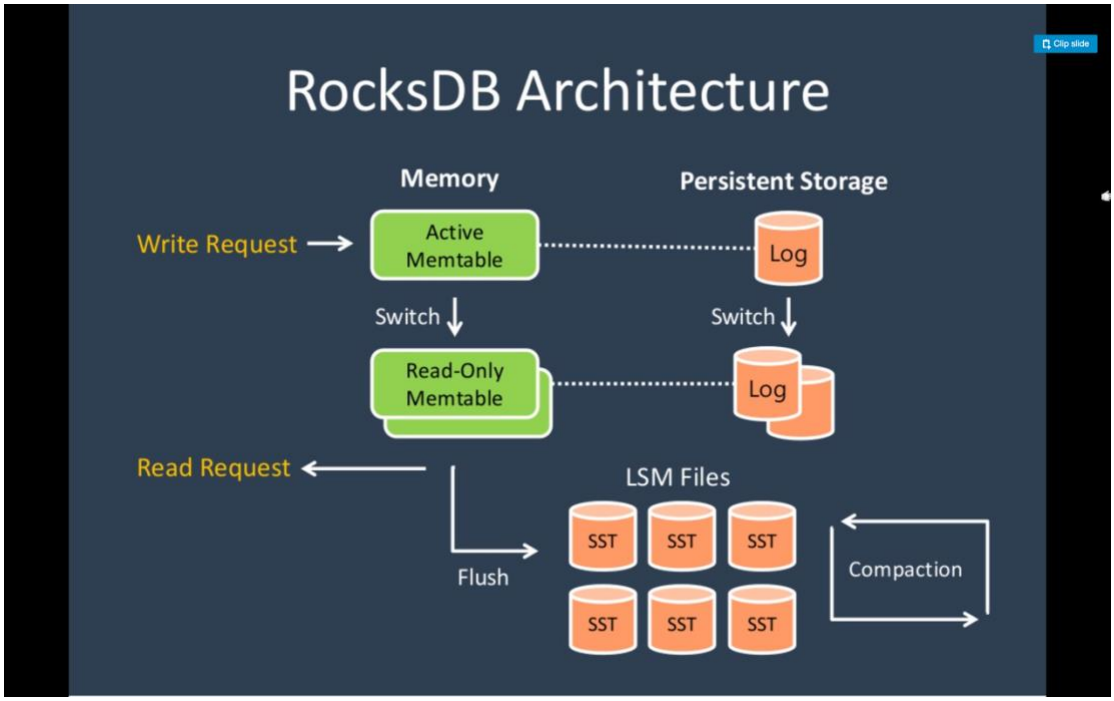
of RocksDB

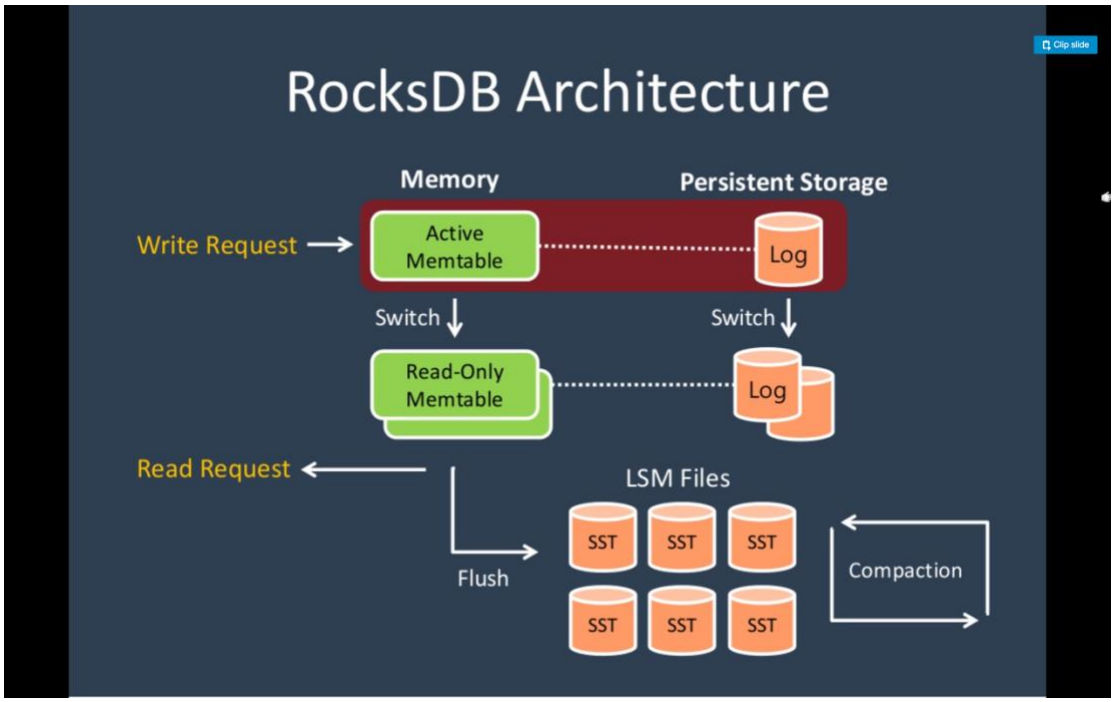
- **On-disk** SSTable indexes are always loaded into memory
- **All writes** go directly to the *Memtable* index
- **Reads** check the Memtable first → the SSTable indexes
- Periodically, the Memtable is flushed to disk as an SSTable
- Periodically, on-disk SSTables are merged

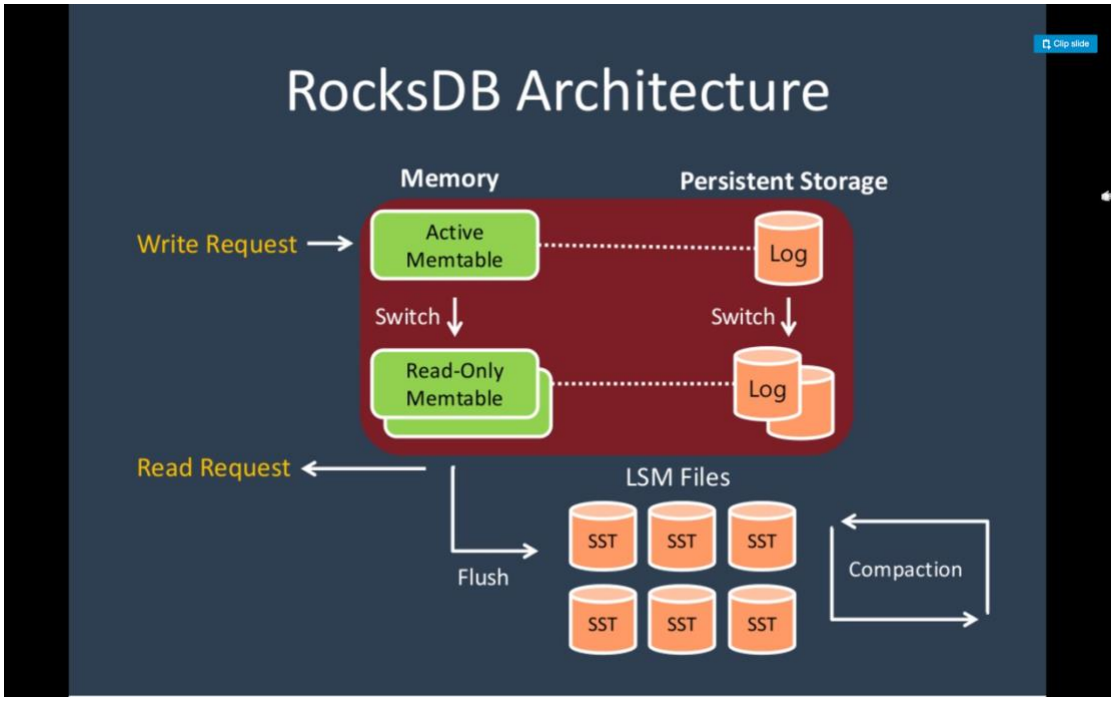
  → update/delete records will overwrite/remove the older data

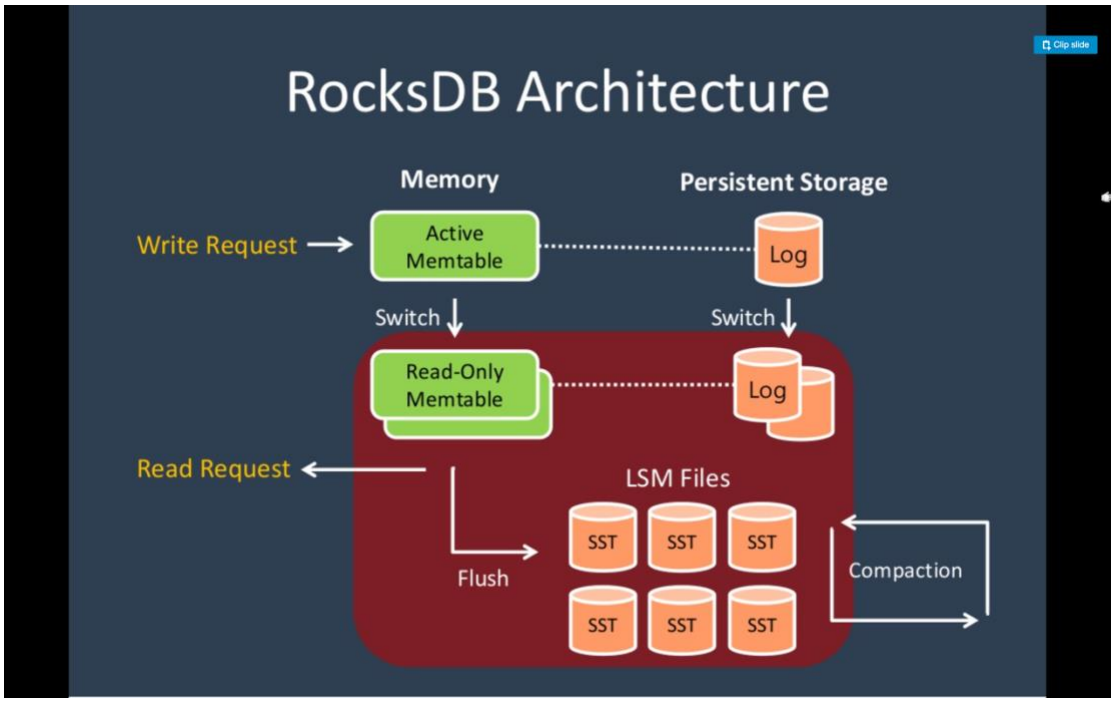
Simplified RocksDB

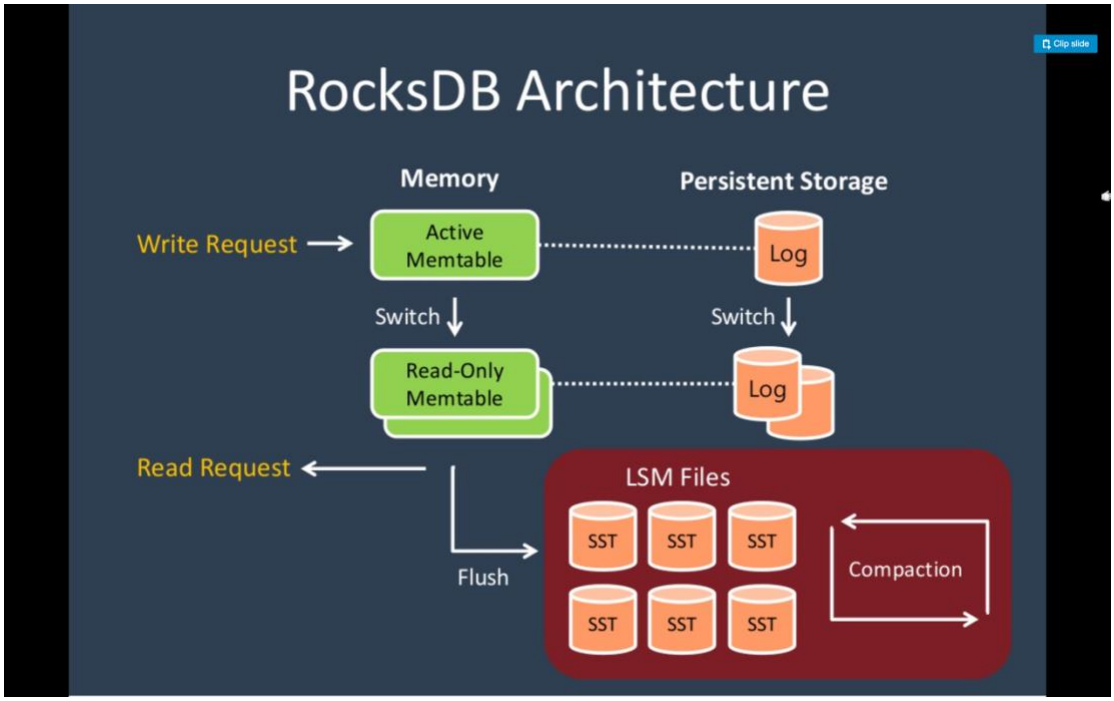


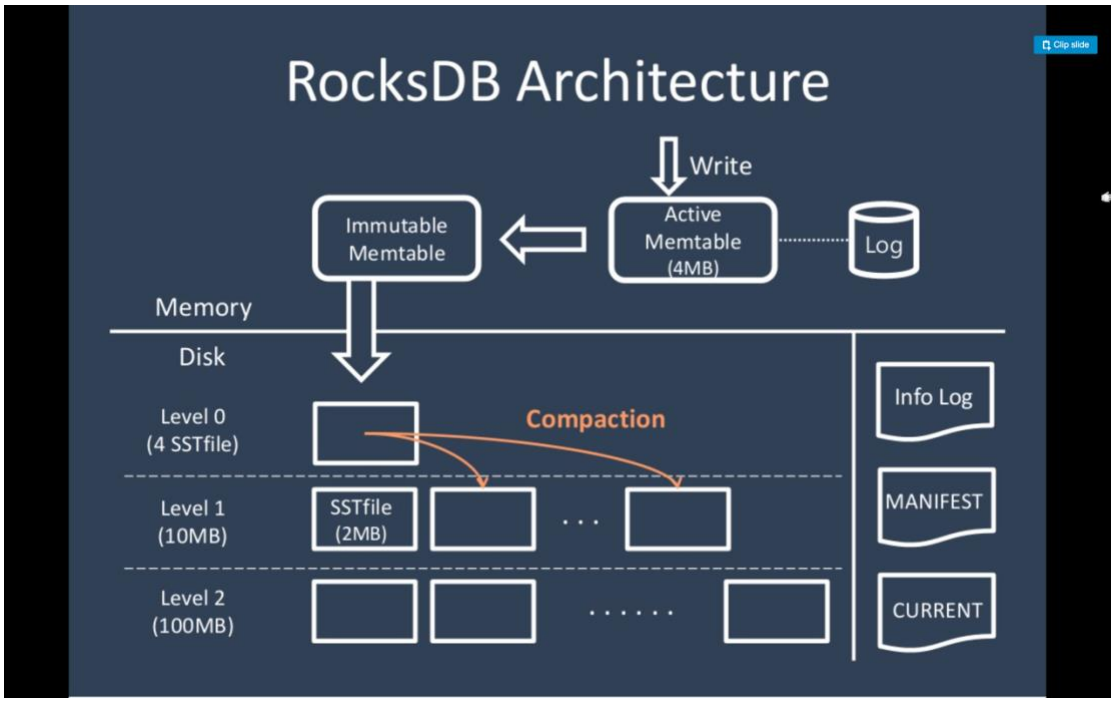

Clip slide
RocksDB Architecture
Memory
Persistent Storage
Write Request
Active Memtable
Log
Switch
Switch
Read-Only Memtable
Log
Read Request
LSM Files
SST
SST
SST
SST
SST
SST
Flush
Compaction


Clip slide
RocksDB Architecture
Memory
Persistent Storage
Write Request
Active Memtable
Log
Switch
Switch
Read-Only Memtable
Log
Read Request
LSM Files
SST
SST
SST
SST
SST
SST
Flush
Compaction

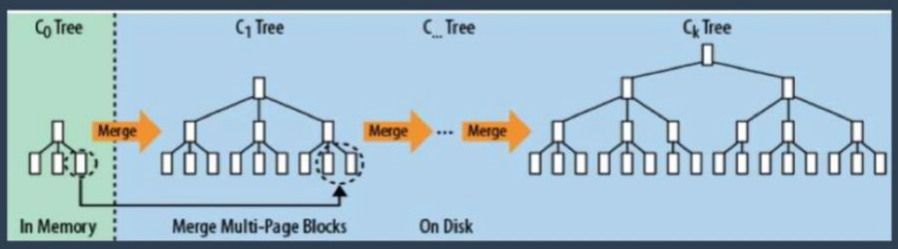

Clip slide
RocksDB Architecture
Memory
Persistent Storage
Write Request
Active Memtable
Log
Switch
Switch
Read-Only Memtable
Log
Read Request
LSM Files
SST
SST
SST
SST
SST
SST
Flush
Compaction


Clip slide
RocksDB Architecture
Memory
Persistent Storage
Write Request
Active Memtable
Log
Switch
Switch
Read-Only Memtable
Log
Read Request
LSM Files
SST
SST
SST
SST
SST
SST
Flush
Compaction


Clip slide
RocksDB Architecture
Memory
Persistent Storage
Write Request
Active Memtable
Log
Switch
Switch
Read-Only Memtable
Log
Read Request
LSM Files
SST
SST
SST
SST
SST
SST
Compaction
Flush


Clip slide
RocksDB Architecture
Write
Immutable Memtable
Active Memtable (4MB)
Log
Memory
Disk
Level 0 (4 SSTfile)
Compaction
Level 1 (10MB)
SSTfile (2MB)
Level 2 (100MB)
Info Log
MANIFEST
CURRENT

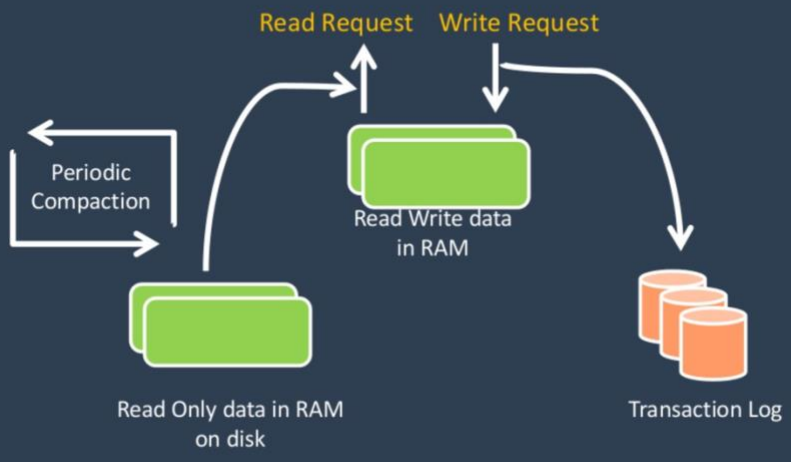
Log-Structured Merge Tree



- LSM-tree
  - N-level merge trees
  - Splitting **a logical tree** into several **physical pieces**
  - So that *the most-recently-updated portion of data* is in a tree in memory
  - Transform **random writes into sequential writes** using *logfile* & in-memory store(*Memtable*)

Log-Structured Merge DB

to minimize "random writes"



Log-Structured Merge DB

to minimize "random writes"

① Data Write(Insert, Update)

- New puts are written to memory(Memtable) & logfile sequentially
- Memtable is filled up → flushed to a SSTable on disk
- Operated in memory, no disk access → faster than B+ tree

② Data Read

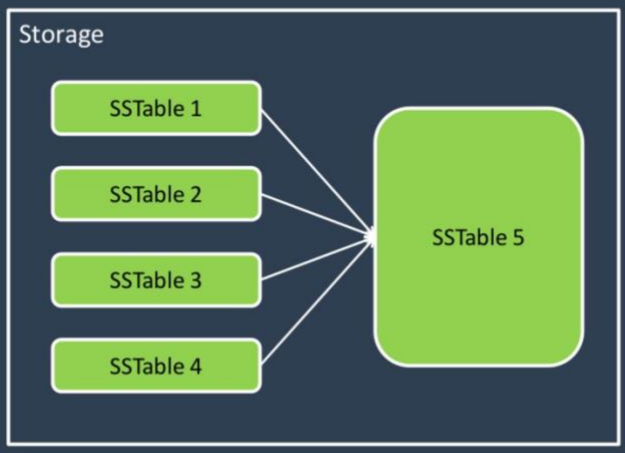
- Memtable → SSTable
- Maintain all the SSTable indexes in memory

RocksDB Compaction

Multi-threaded compactions

- Background Multi-thread
  → *periodically* do the "compaction"
  → parallel compactions on different parts of the database can occur simultaneously
- **Merge** SSTfiles to a bigger SSTfile
- **Remove** multiple copies of the same key
  – Duplicate or overwritten keys
- Process **deletions** of keys
- Supports two different styles of compaction
  – Tunable compaction to trade-off

RocksDB Compaction



1. Level Style Compaction

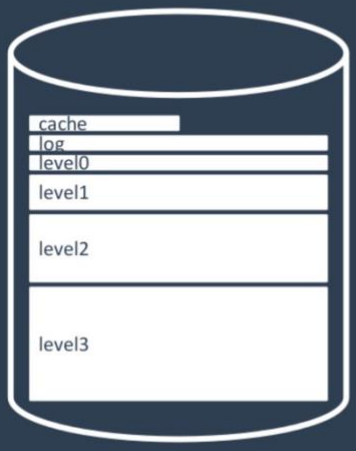
Inherited from LevelDB

- RocksDB default compaction style
- Stores data in **multiple levels** in the database
- More **recent** data → L0
  The **oldest** data → Lmax
- Files in *L0*
  - overlapping keys, sorted by flush time

  Files in *L1 and higher*
  - non-overlapping keys, sorted by key
- Each level is 10 times larger than the previous one

Level Style Compaction

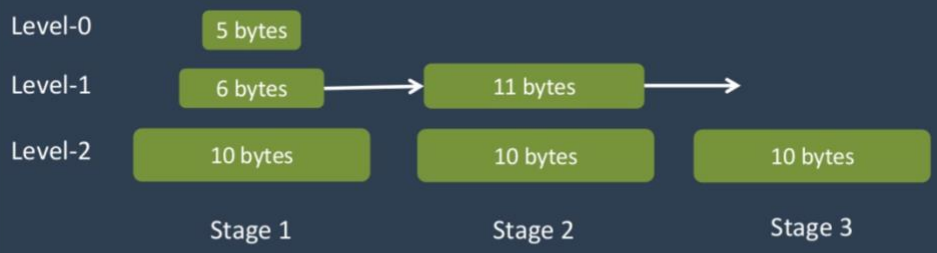
Compaction process



① Pick one file from level N
② Compact it with all its overlapping files from level N+1
③ Replace them with new files in level N+1

Level Style Compaction

Compaction example



Two compactions by Level Style Compaction

Level 0 → Level 1 Compaction

Tricky Compaction

- Level 0 → overlapping keys
- Compaction includes all files from L1
- All files from L1 are compacted with L0
- L0 → L1 compaction completion

  ➡ L1 → L2 compaction start
- **Single thread** compaction → not good throughput
- Solution : Making the size of L0 similar to size of L1

2. Universal Style Compaction

- For write-heavy workloads

  → Level Style Compaction may be bottlenecked on disk throughput
- Stores all files in L0
- All files are arranged in **time order**
- Temporarily **increase size** amplification by a factor of two
- Intended to **decrease write amplification**
- But, **increase space amplification**

Universal Style Compaction

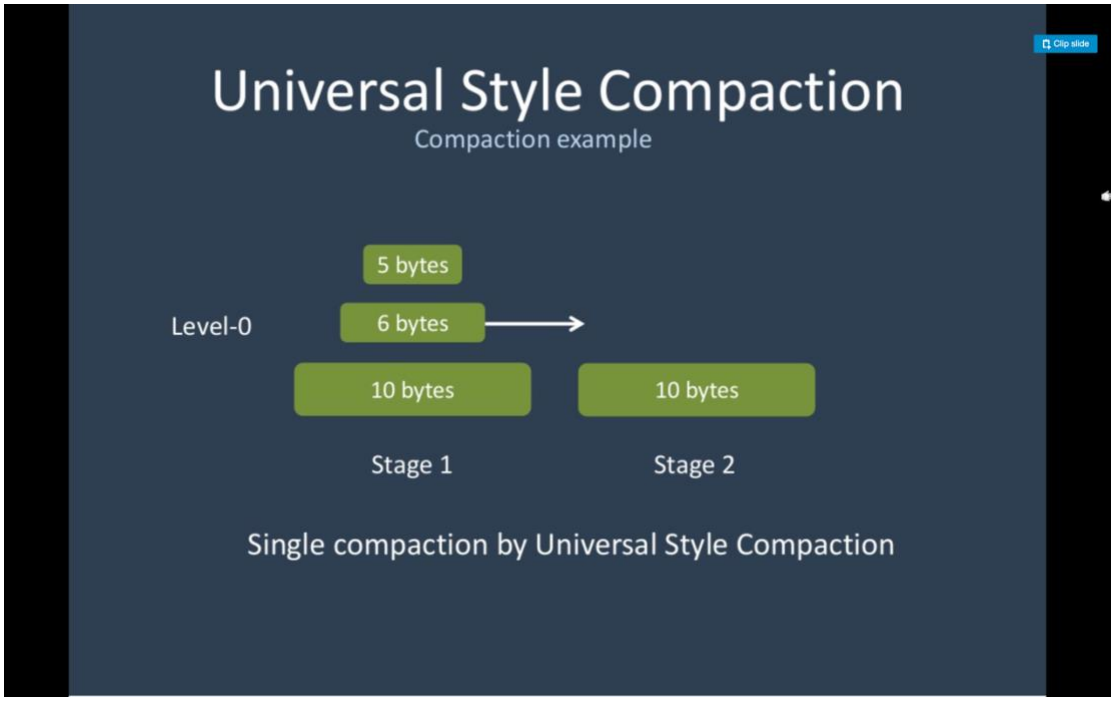
Compaction process

① Pick up a few files that are chronologically adjacent to one another
② Merge them
③ Replace them with a new file in level 0

Universal Style Compaction

Compaction options

- size_ratio
  - Percentage flexibility while comparing file size
  - Default : 1
- min_merge_width
  - The minimum number of files in a single compaction
  - Default : 2
- max_merge_width
  - The maximum number of files in a single compaction
  - Default : UINT_MAX


Clip slide
Universal Style Compaction
Compaction example
5 bytes
Level-0
6 bytes
10 bytes
10 bytes
Stage 1
Stage 2
Single compaction by Universal Style Compaction