2025 Winter RocksDB Study
2nd week

Hojin Shin, Guangxun Zhao

http://sslab.dankook.ac.kr/, https://sslab.dankook.ac.kr/~choijm

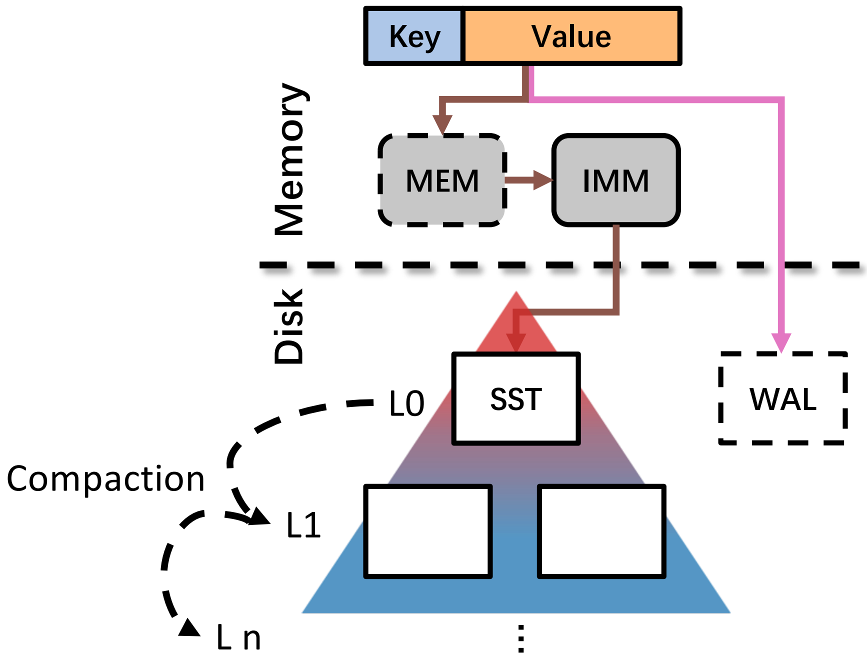
Presentation by Guangxun Zhao

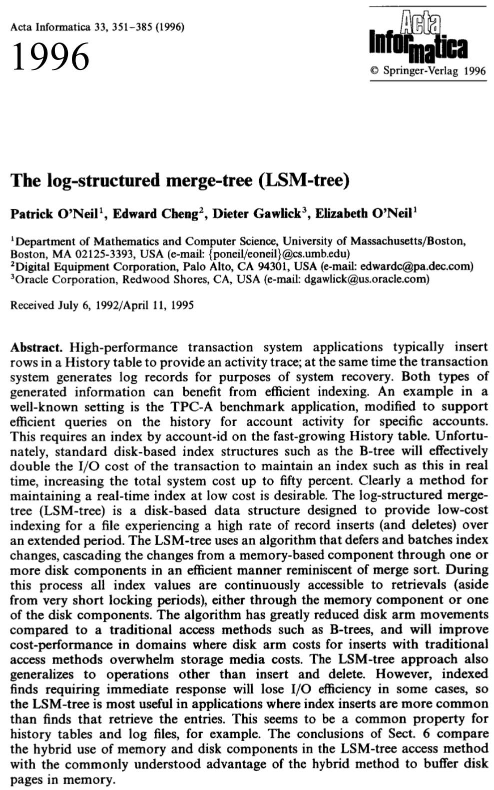
guangxunzhao@dankook.ac.kr

Contents

Log Structured Merge-Tree

- Systems with Memory-Disk
- Write optimized data structure
- Use Out-of-place updates





Acta Informatica 33, 351–385 (1996)
1996
Acta Informatica

The log-structured merge-tree (LSM-tree)

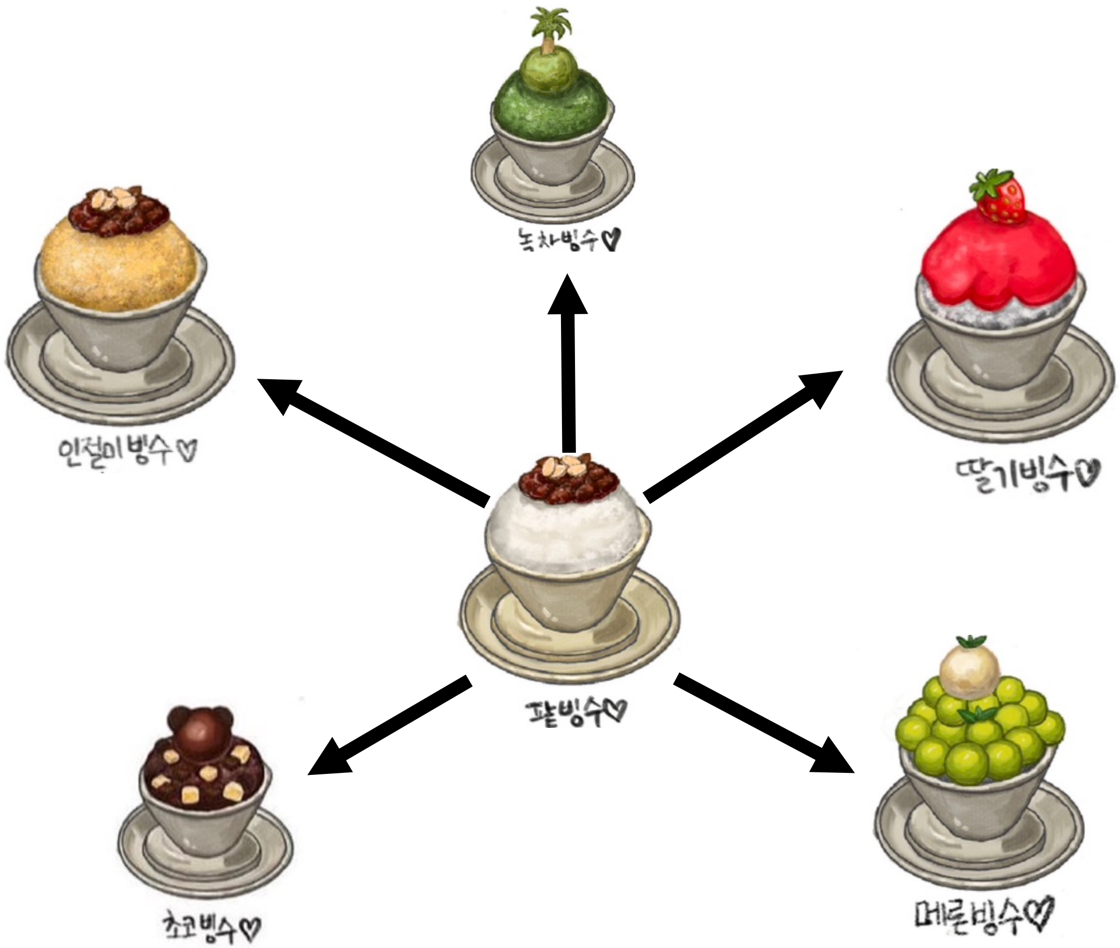
Patrick O'Neil[1], Edward Cheng[2], Dieter Gawlick[3], Elizabeth O'Neil[1]

[1]Department of Mathematics and Computer Science, University of Massachusetts/Boston, Boston, MA 02125-3393, USA (e-mail: {poneil/eoneil}@cs.umb.edu)
[2]Digital Equipment Corporation, Palo Alto, CA 94301, USA (e-mail: edwardc@pa.dec.com)
[3]Oracle Corporation, Redwood Shores, CA, USA (e-mail: dgawlick@us.oracle.com)

Received July 6, 1992/April 11, 1995

Abstract. High-performance transaction system applications typically insert rows in a History table to provide an activity trace; at the same time the transaction system generates log records for purposes of system recovery. Both types of generated information can benefit from efficient indexing. An example in a well-known setting is the TPC-A benchmark application, modified to support efficient queries on the history for account activity for specific accounts. This requires an index by account-id on the fast-growing History table. Unfortunately, standard disk-based index structures such as the B-tree will effectively double the I/O cost of the transaction to maintain an index such as this in real time, increasing the total system cost up to fifty percent. Clearly a method for maintaining a real-time index at low cost is desirable. The log-structured merge-tree (LSM-tree) is a disk-based data structure designed to provide low-cost indexing for a file experiencing a high rate of record inserts (and deletes) over an extended period. The LSM-tree uses an algorithm that defers and batches index changes, cascading the changes from a memory-based component through one or more disk components in an efficient manner reminiscent of merge sort. During this process all index values are continuously accessible to retrievals (aside from very short locking periods), either through the memory component or one of the disk components. The algorithm has greatly reduced disk arm movements compared to a traditional access methods such as B-trees, and will improve cost-performance in domains where disk arm costs for inserts with traditional access methods overwhelm storage media costs. The LSM-tree approach also generalizes to operations other than insert and delete. However, indexed finds requiring immediate response will lose I/O efficiency in some cases, so the LSM-tree is most useful in applications where index inserts are more common than finds that retrieve the entries. This seems to be a common property for history tables and log files, for example. The conclusions of Sect. 6 compare the hybrid use of memory and disk components in the LSM-tree access method with the commonly understood advantage of the hybrid method to buffer disk pages in memory.

Concept Map



Source:https://blog.naver.com/smc1143/223342199436

역사 [편집]

가장 오래된 역사 기록은 기원전 400년 페르시아이다.[1]

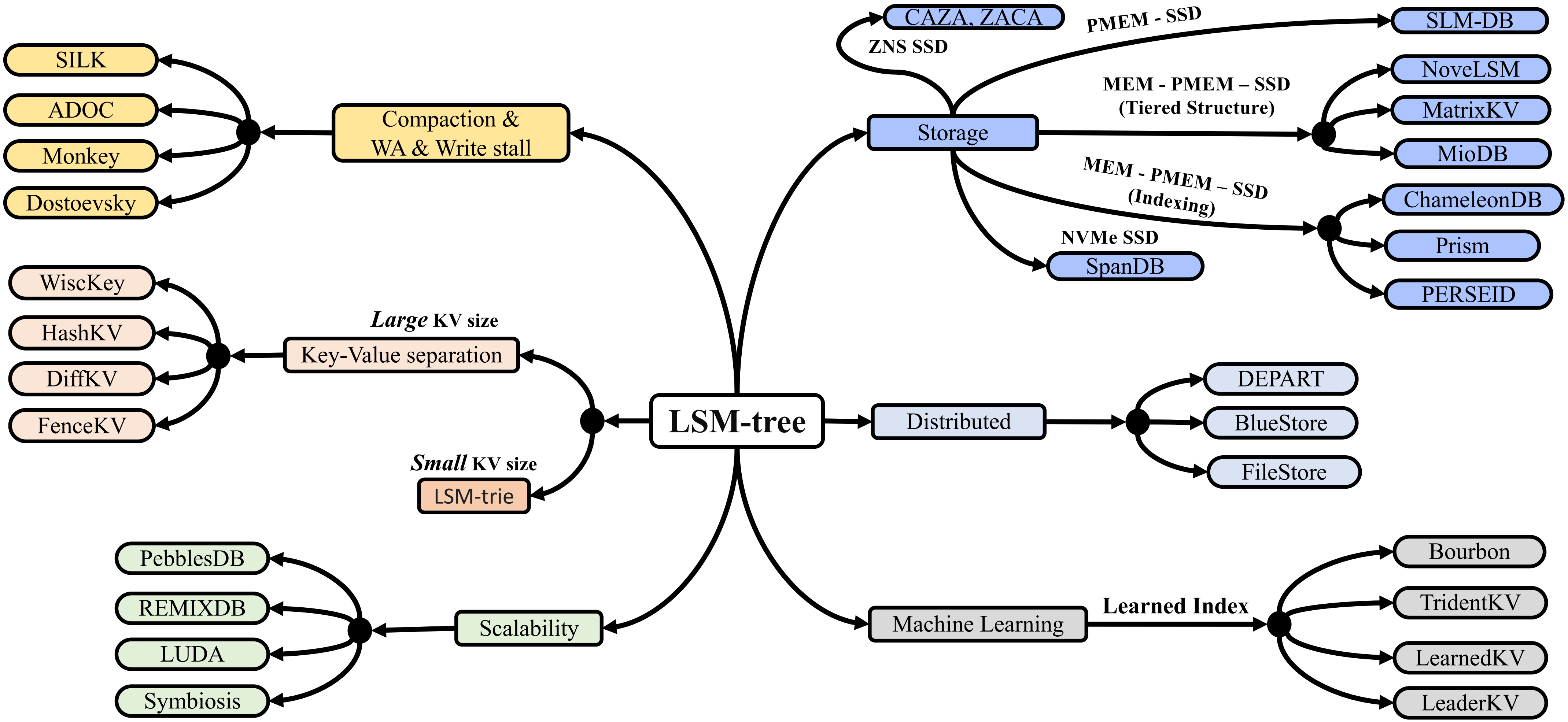
서양에서는 기원전 300년경 마케도니아 왕국의 알렉산더 대왕이 페르시아 제국을 점령할 때 만들어 먹었다는 설도 있는데, 병사들이 더위와 피로에 지쳐 쓰러지자 높은 산에 쌓인 눈에 꿀과 과일즙 등을 넣어 먹었다고 한다. 또 로마의 정치가이자 장군인 카이사르는 알프스에서 가져온 얼음과 눈으로 술과 우유를 차게 해서 마셨다고 한다.

11세기 일본의 마쿠라노소시(枕草子)에는 얼음을 칼로 갈아 그 위에 칡즙을 뿌려서 먹었다는 기록이 나온다. 11세기 송나라 역사를 기록한 송사(宋史)에는 밀사빙(蜜沙氷)을 황제가 조정 대신에게 하사했다는 기록이 나온다. 밀사(蜜沙)는 꿀모래가 아니라 꿀에 버무린 팔소(豆沙)를 의미하고, 빙(氷)은 얼음을 의미한다.[2]

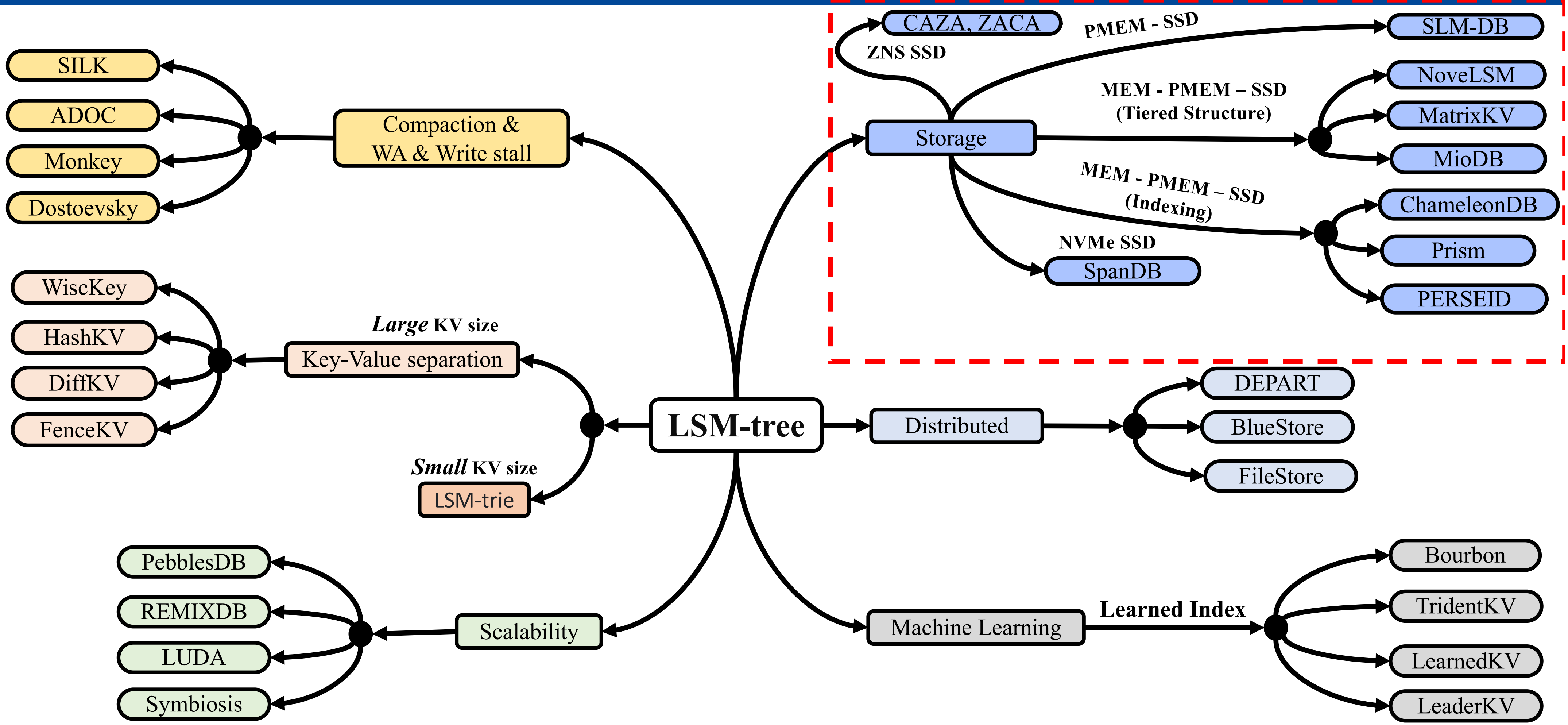
이탈리아의 마르코 폴로가 쓴 《동방견문록》에는 중국 베이징에서 즐겨 먹던 'frozen milk'의 제조법을 베네치아로 가져가 전했다는 기록이 있다.

빙수는 기원전 3000년경 중국에서 눈이나 얼음에 꿀과 과일즙을 섞어 먹은 것에서 비롯됐다.[3]

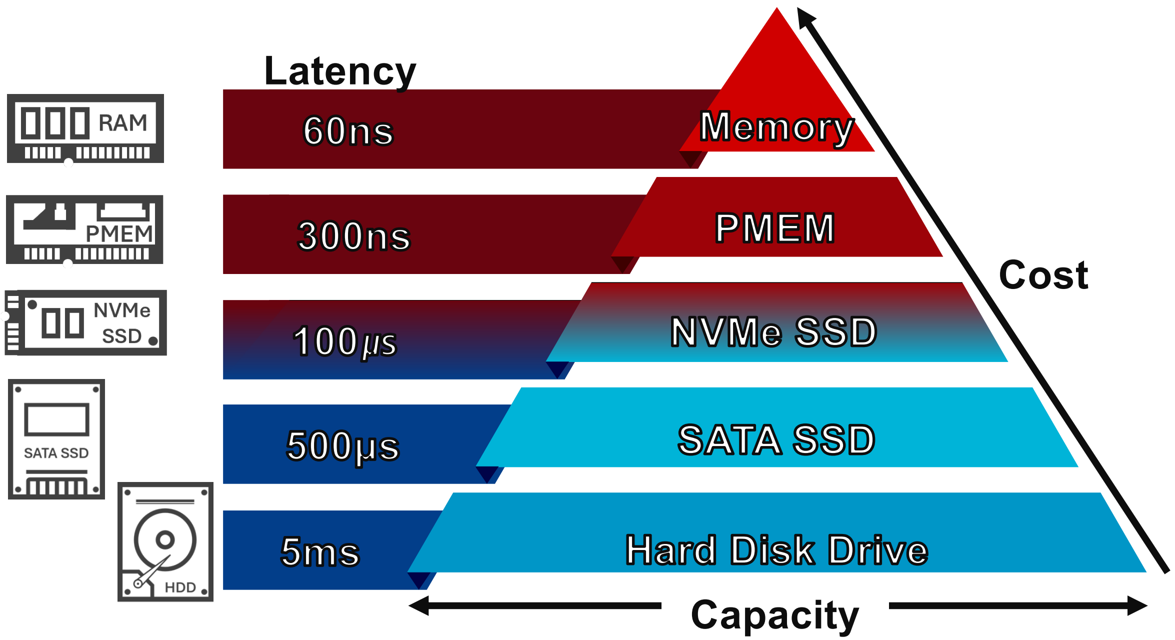
LSM-tree's Concept Map



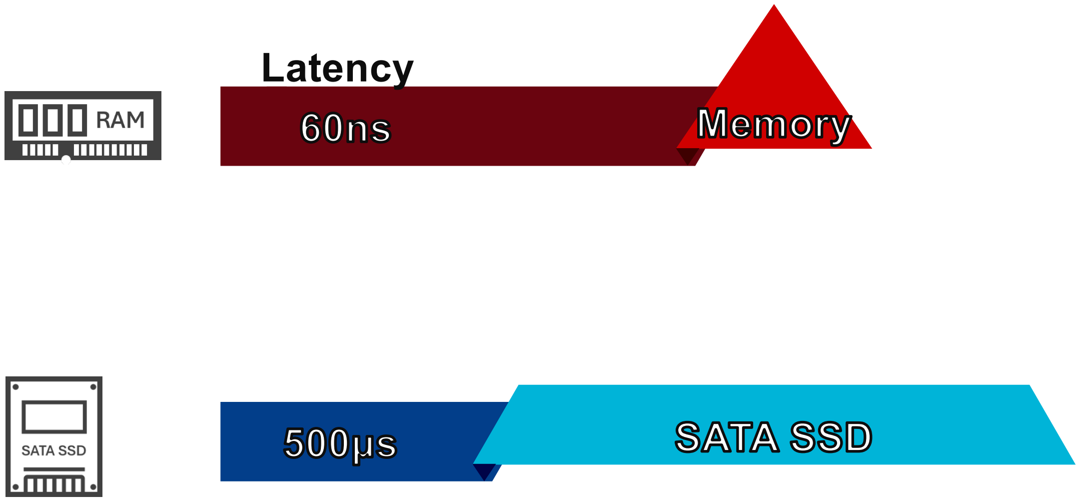
LSM-tree's Concept Map



LSM-tree with Tiered Storage



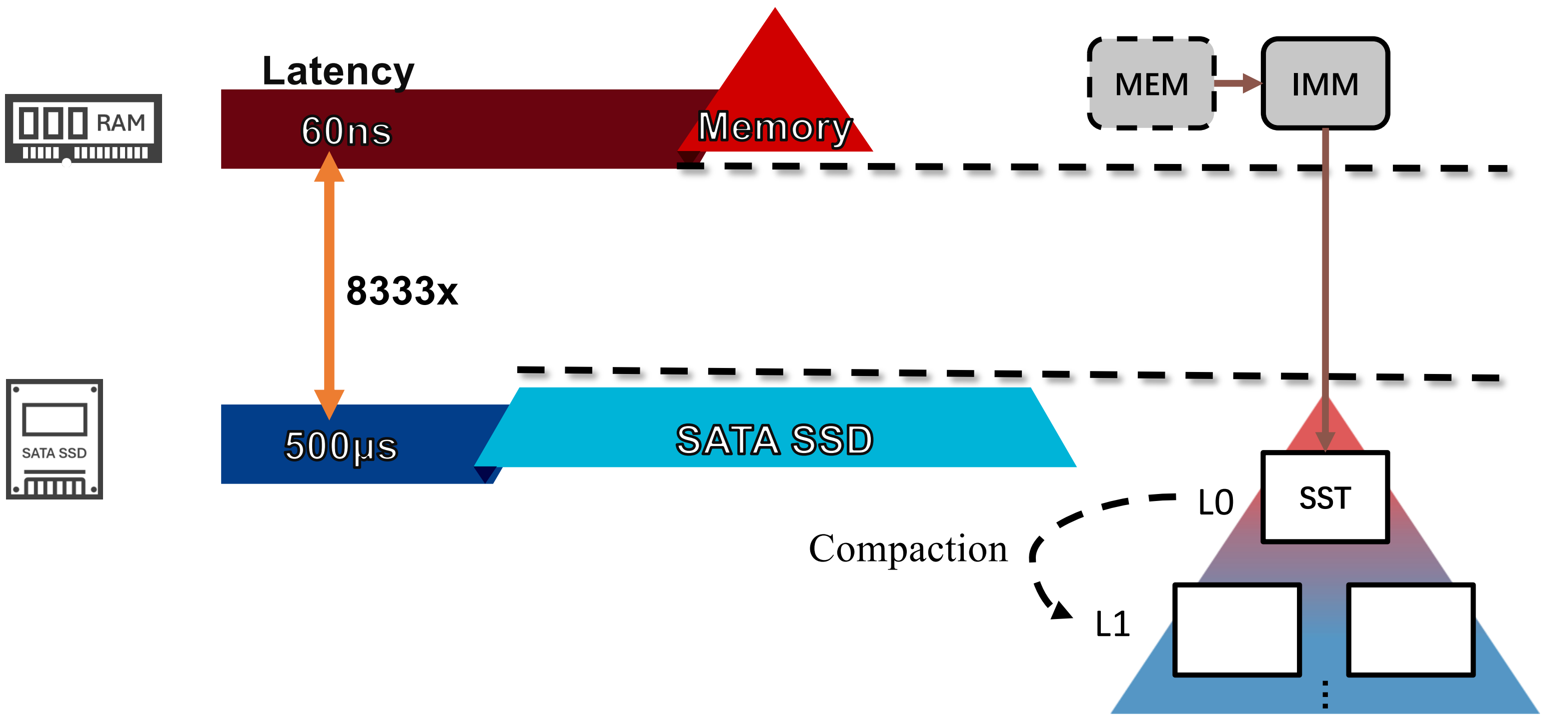
LSM-tree with Tiered Storage



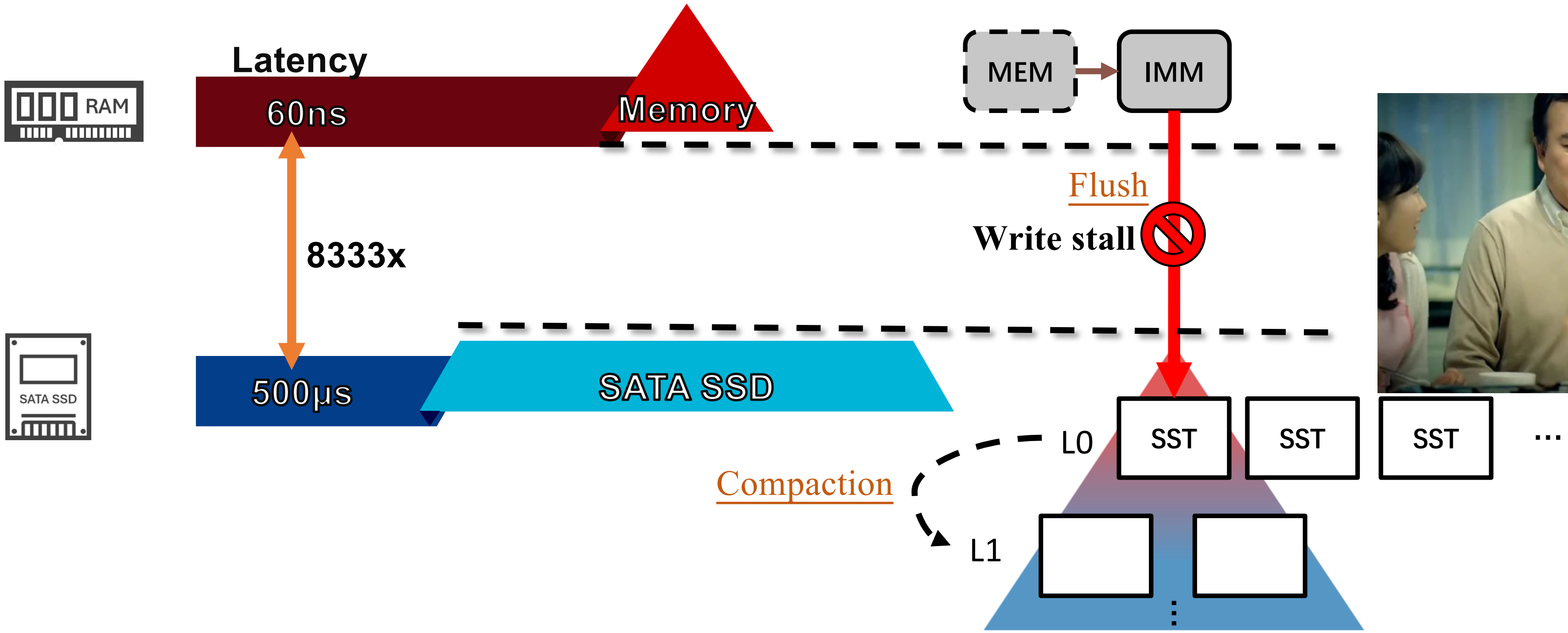
- Memory
  - Volatility
  - Low latency
  - Byte-addressability

- SSD
  - Non-volatile
  - Big capacity
  - Page/Block unit

LSM-tree with Tiered Storage

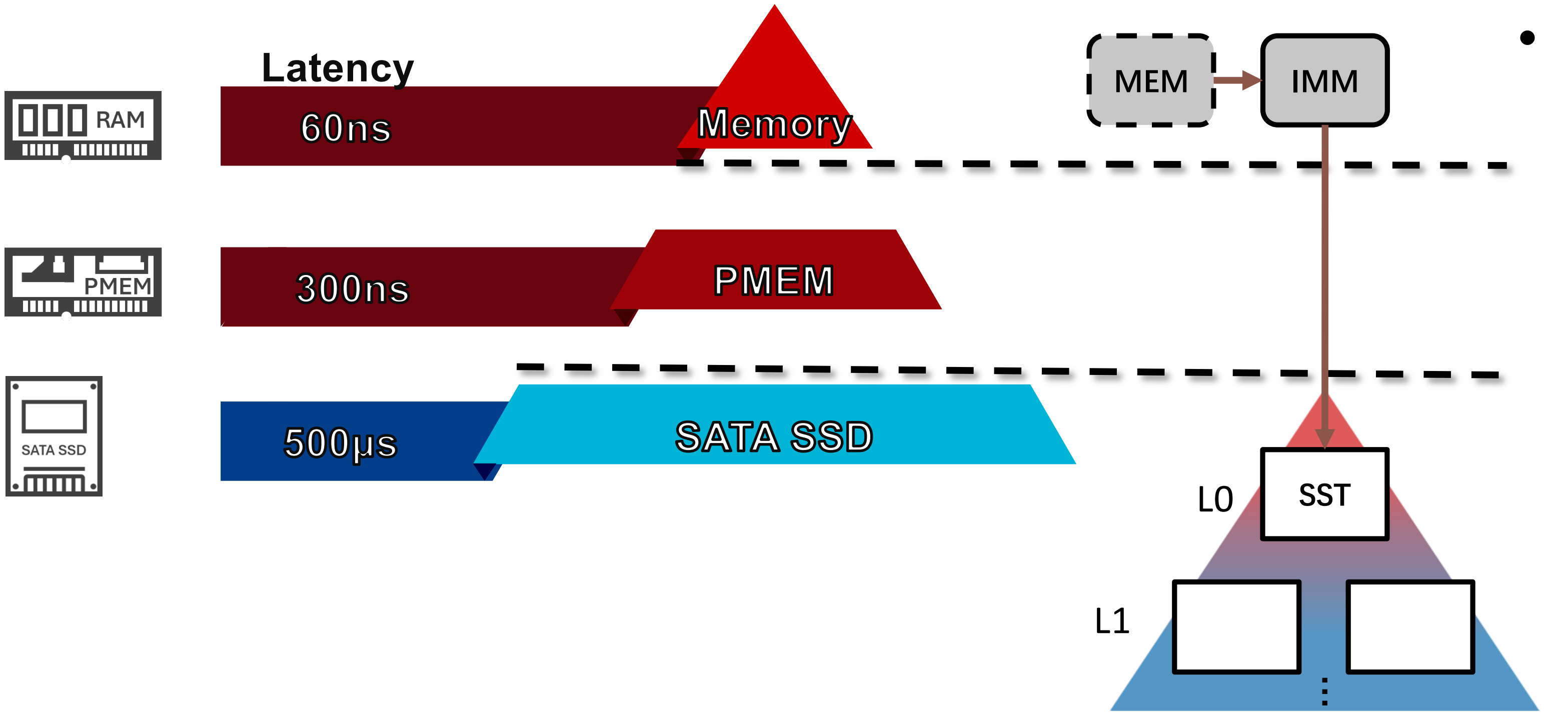


LSM-tree with Tiered Storage



Source:https://namu.wiki/w/%EA%B0%9C%EB%B9%84%EC%8A%A4%EC%BD%98?uuid=984b0c72-d574-47ea-804d-1642b7c95d28

LSM-tree with Tiered Storage

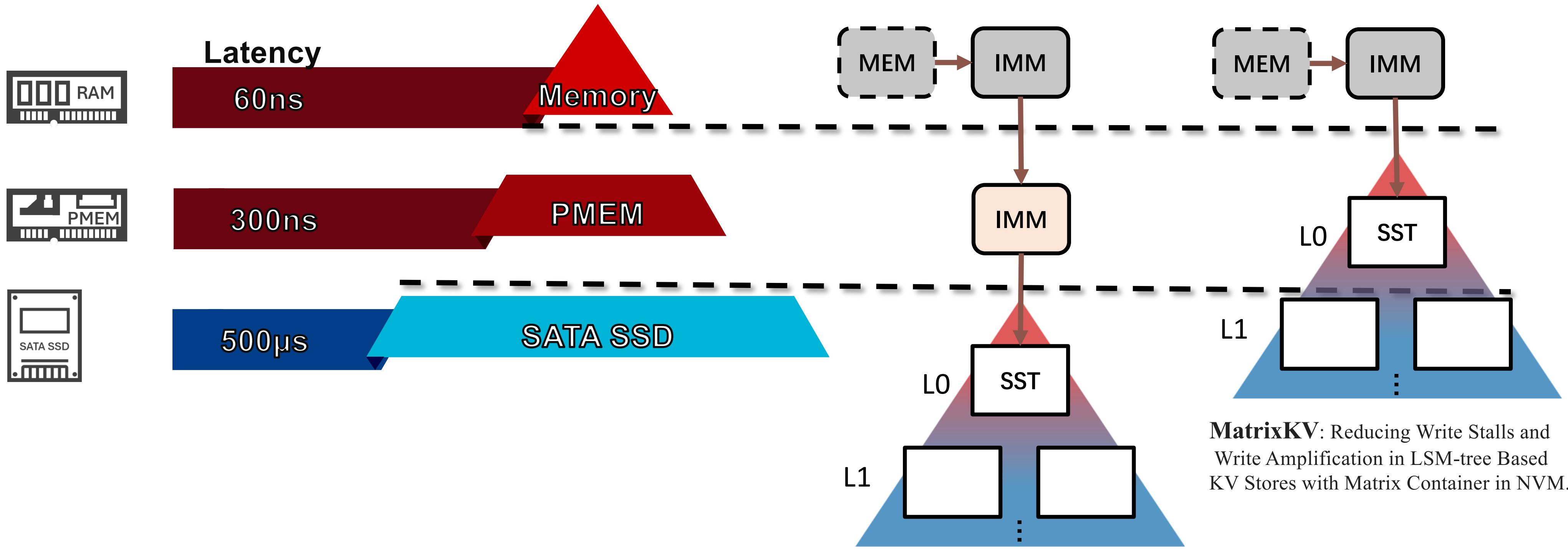


Persistent **MEM**ory

- Low latency
- Byte-addressability
- Non-volatile

Source:https://namu.wiki/w/%EA%B0%9C%EB%B9%84%EC%8A%A4%EC%BD%98?uuid=984b0c72-d574-47ea-804d-1642b7c95d28

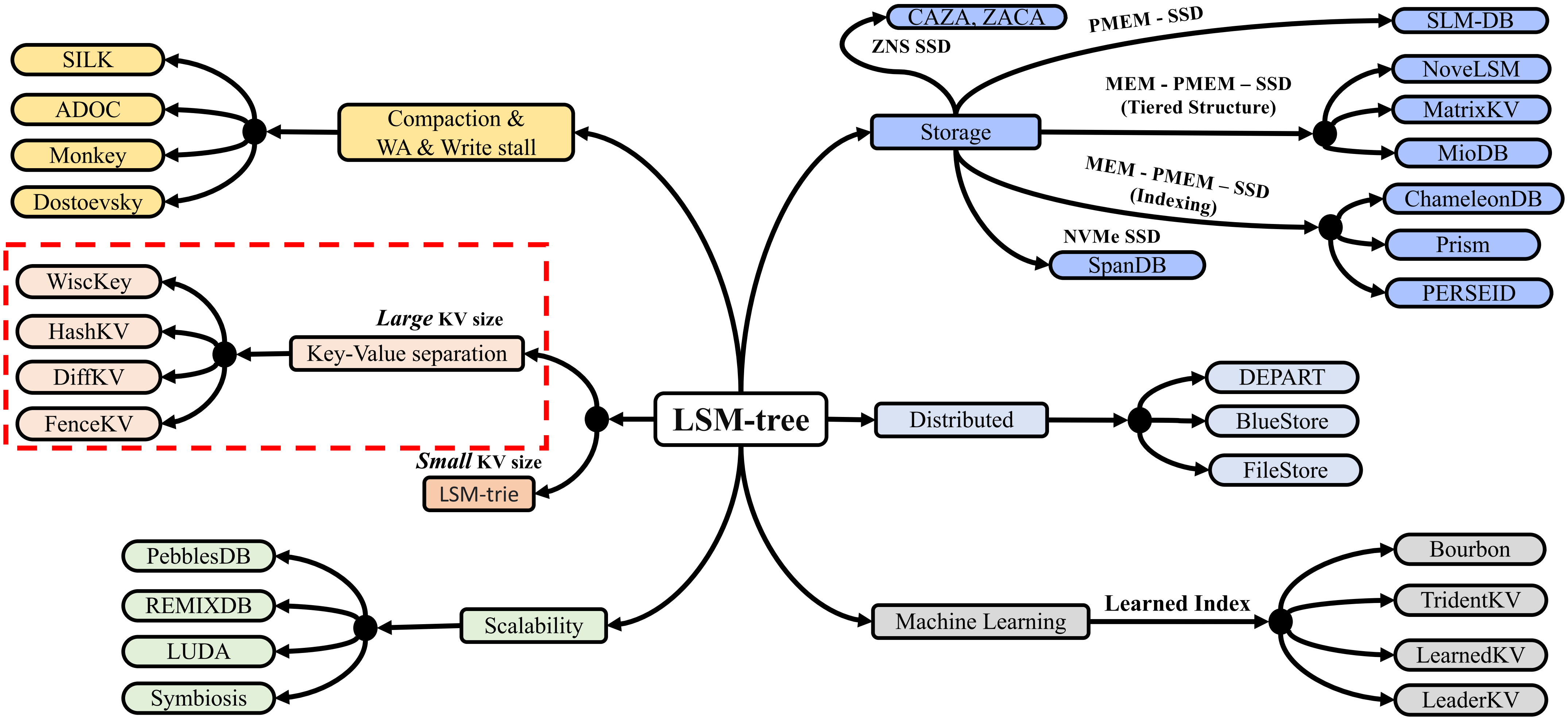
LSM-tree with Tiered Storage



MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM.

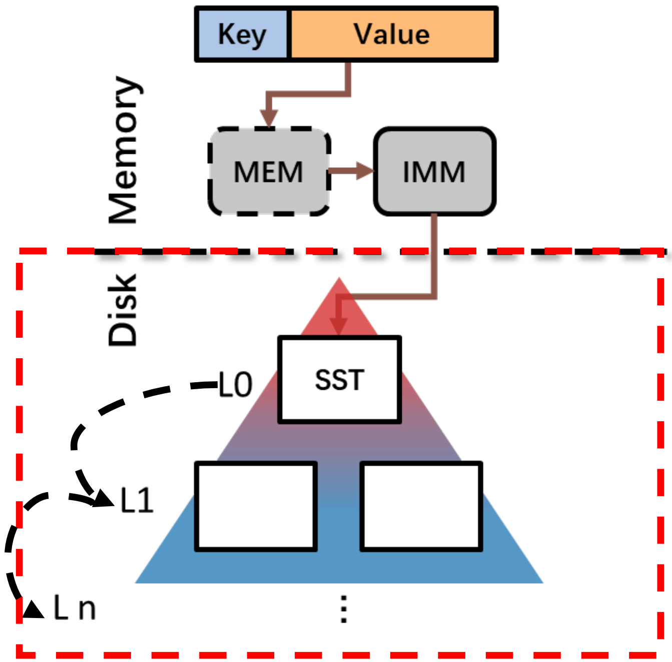
Redesigning LSMs for Nonvolatile Memory with **NoveLSM**.

LSM-tree's Concept Map



What is Write Amplification ?

- During compaction, the 10x size ratio between levels causes up to 10 files to be read and rewritten per level, leading to a cumulative write amplification can be over 50x (10 for each gap between L1 to L6).

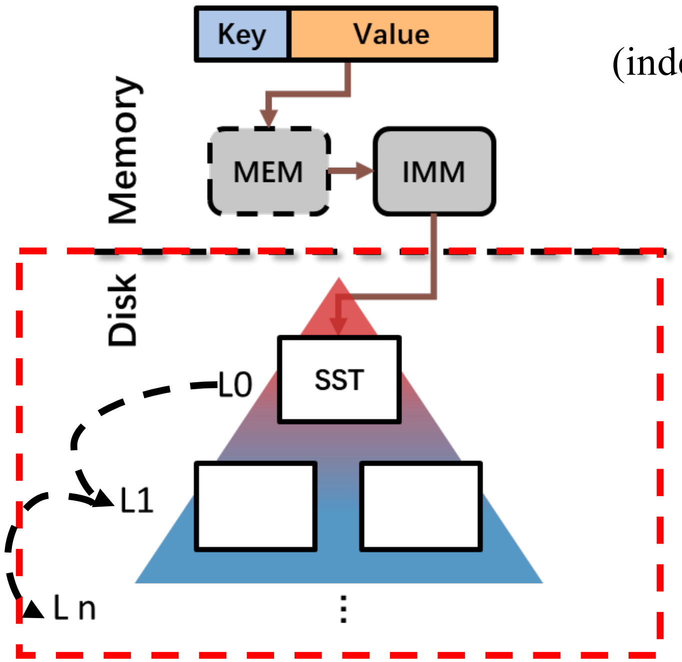


10x 10x 10x 10x 10x

Level	0	1	2	3	4	5	6
Size	8MB	10MB	100MB	1GB	10GB	100GB	1TB

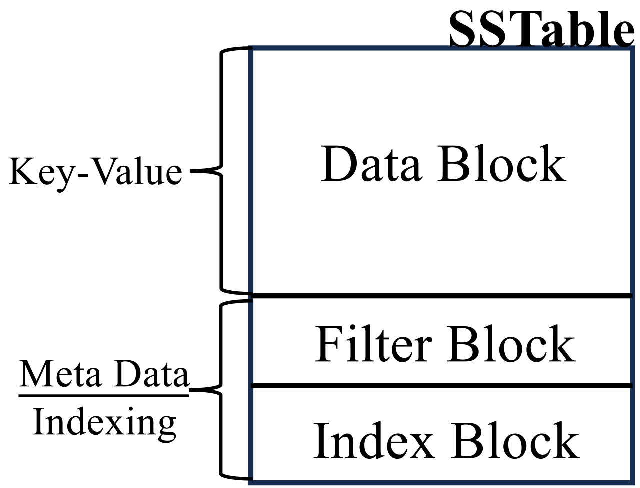
What is Read Amplification?

- Read amplification, LevelDB need to check multiple levels. LevelDB needs to searching up to 14 files across levels and reading multiple metadata blocks (index, bloom filter, and data) for each SSTable file, can reach up to 336x for a 1KB KV.



(index block + bloom-filter blocks + data block) * SSTable = Read amplification

(16 + 4 + 4) * 14 = 336

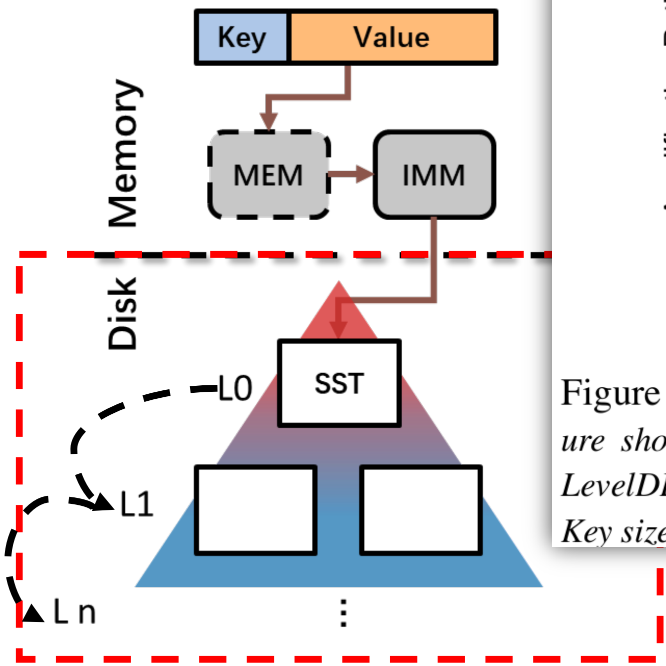


What is Read Amplification?

- Read amplification, LevelDB need to check multiple levels. LevelDB needs to searching up to 14 files across levels and reading multiple metadata blocks (index, bloom filter, and data) to 336x for a 1KB KV.

SSTable = Read amplification

36



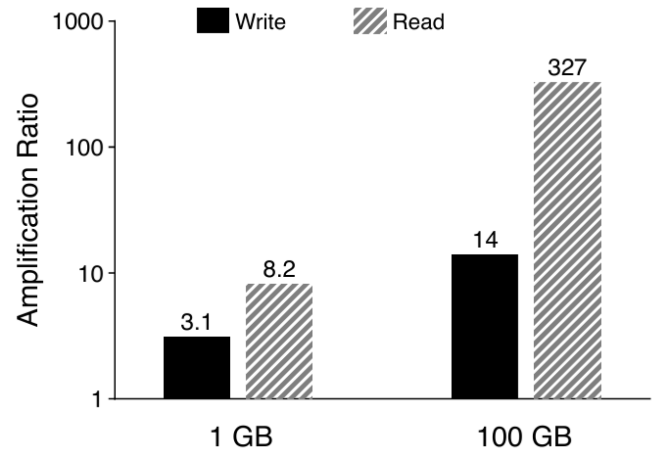
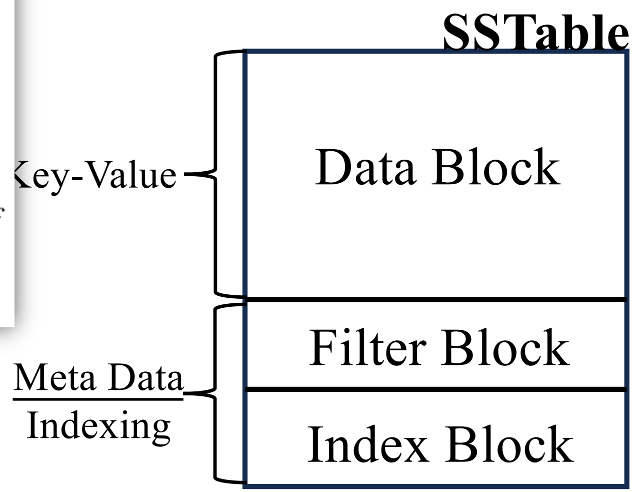
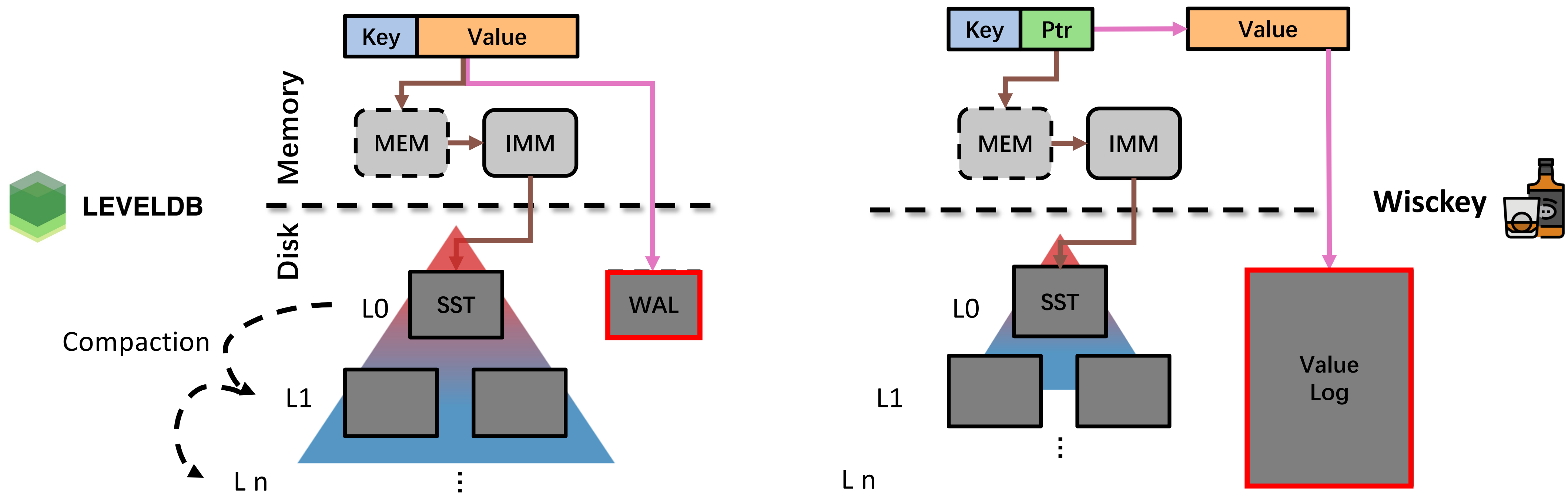


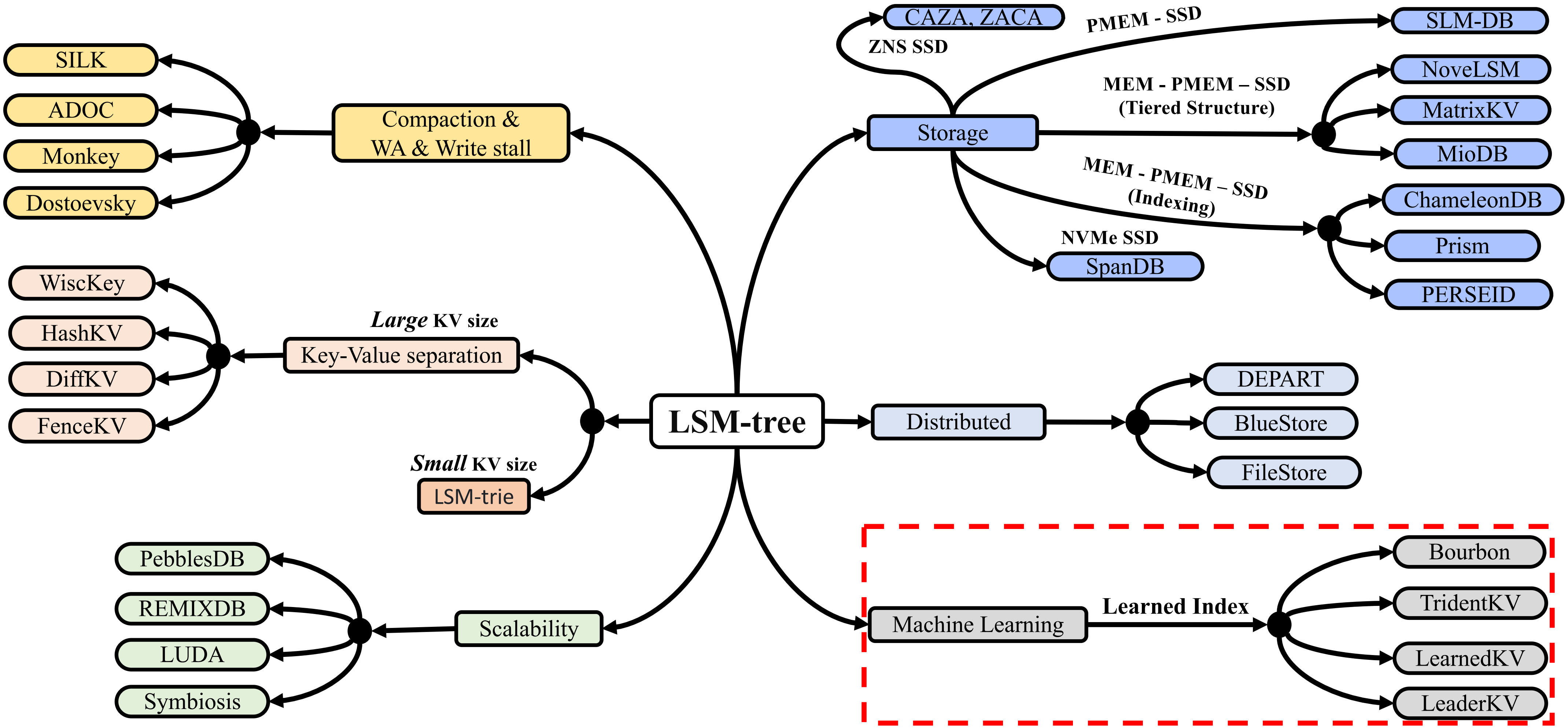
Figure 2: **Write and Read Amplification.** *This figure shows the write amplification and read amplification of LevelDB for two different database sizes, 1 GB and 100 GB. Key size is 16 B and value size is 1 KB.*



WiscKey Structure

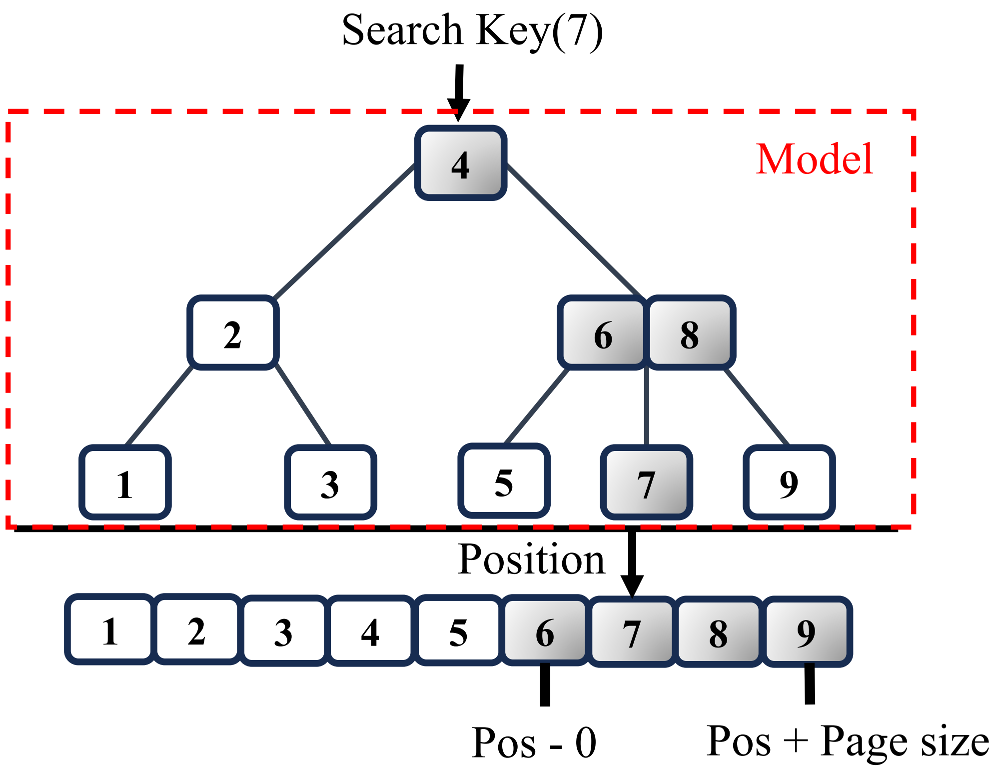


LSM-tree’s Concept Map



B-Tree

- B-tree is a data structure widely used in databases and file systems.
- The time complexity of search, insert, and delete operations is O(*log n*).



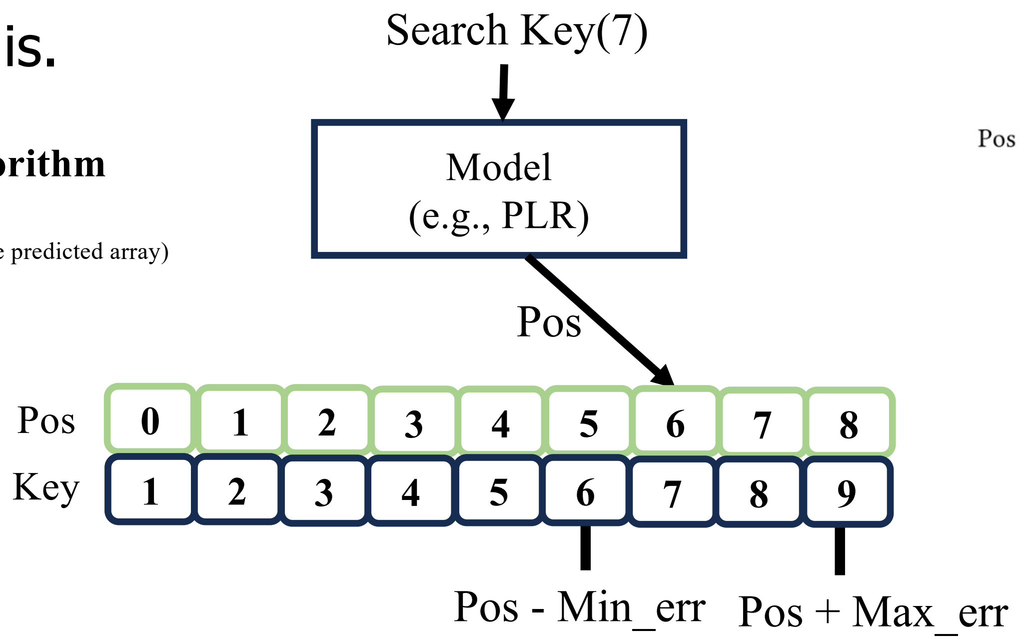
Learned Index

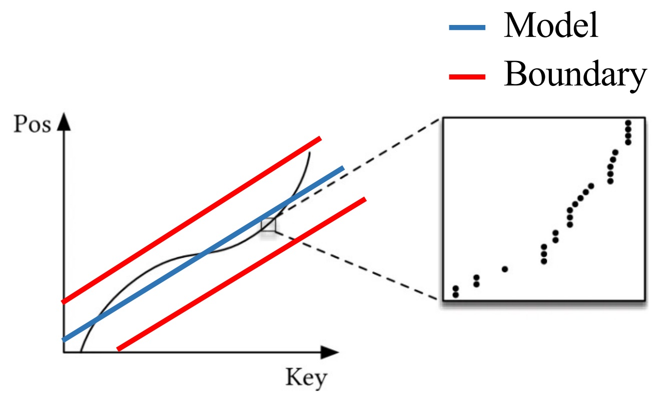
- Learned Index is a data indexing method that uses machine learning models to predict the location of data.
- Unlike traditional indexes, it learns the distribution of data and predicts the location of keys based on this.

Linear Regression Algorithm

Pos = slop*key + intercept

Pos: Position where the key is stored (index in the predicted array)
slope: Slope of the CDF
key: Key value to search for
intercept: y-intercept (bias) of the CDF



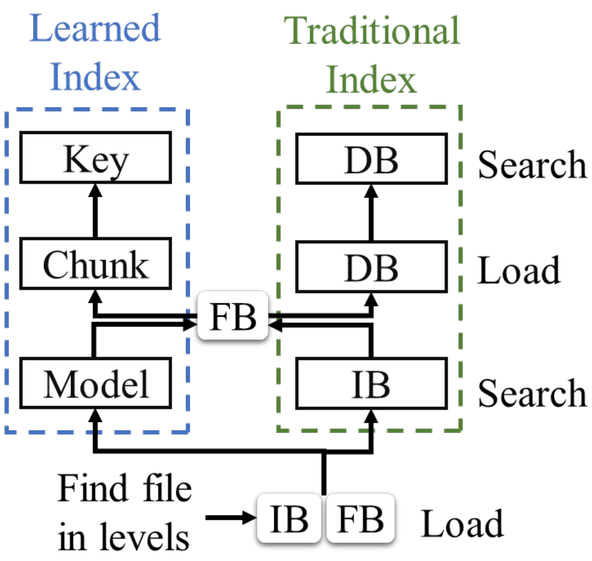


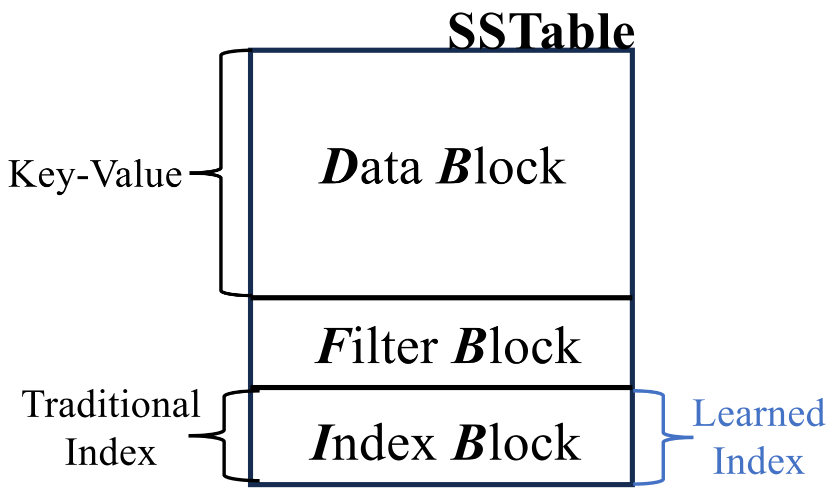
Cumulative Distribution Function

Kraska, Tim, et al. "The case for learned index structures."

Bourbon mechanism

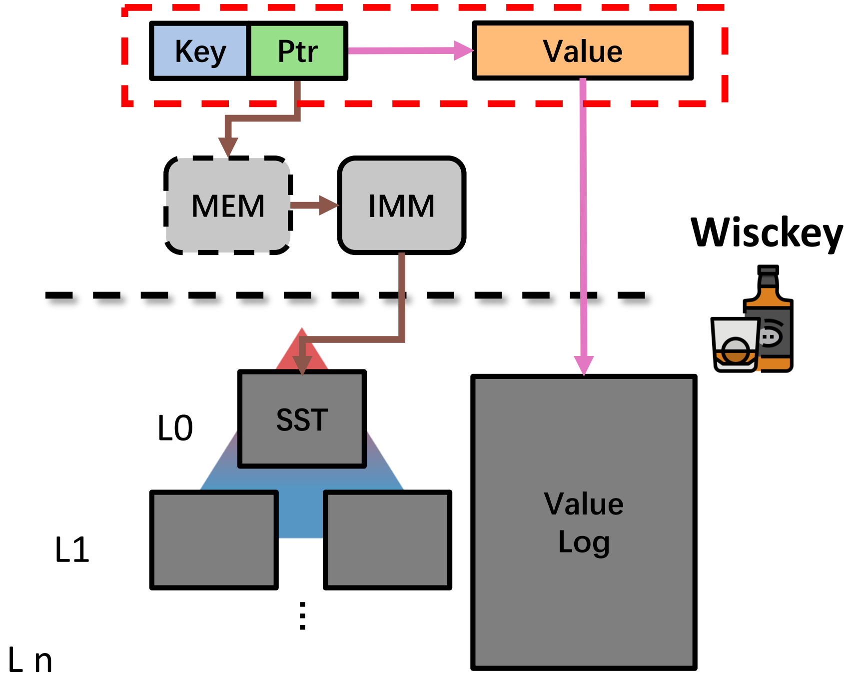
- Lookup steps in SSTable.
  - ✓Traditional Index: Load IB+FB → Search IB+FB → LoadDB → SearchDB
  - ✓Learned Index: Load IB+FB → Model lookup → Load Chunk(Boundary) → Search Chunk(Boundary)

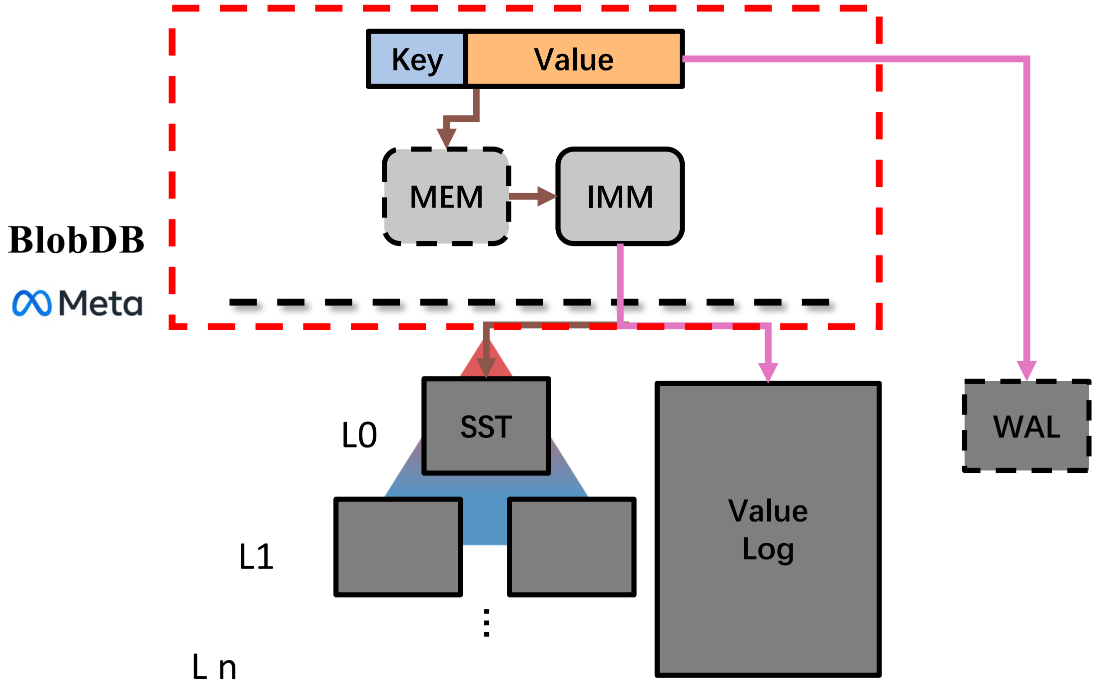




Research Topic 1

- Analysis of the advantage and disadvantage of two key-value separation methods.
- The relationship between key-value separation and compaction strategies.



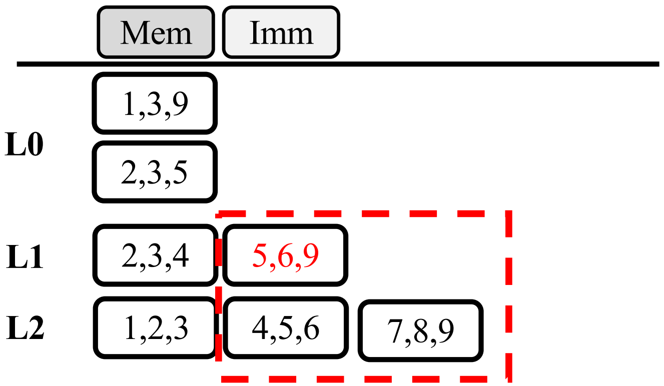


Research Topic 2

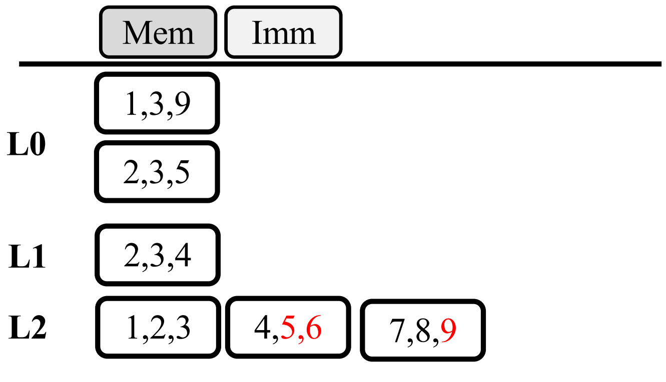
- The performance variations of different compaction strategies based on various key-value sizes.

- Compaction
  - Leveled Compaction
  - Universal compaction style
  - FIFO compaction style

```
-compaction_style (style of compaction: level-based, universal and fifo)
  type: int32 default: 0
```



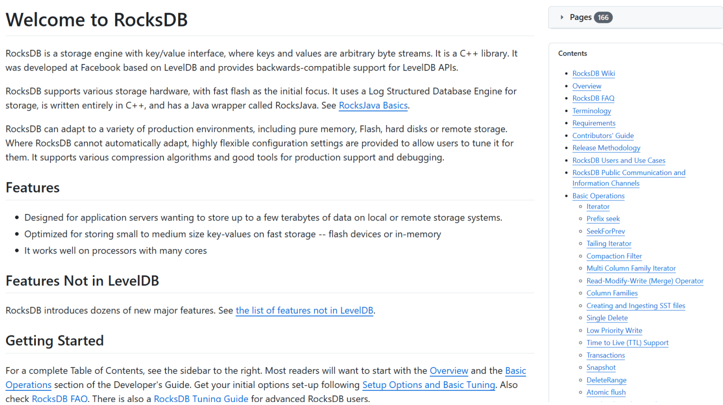
Leveled compaction



Leveled compaction

Another Topics

- Data Compression types
  - ✓ Multi-media course
- SSTable, Memtable size
- Various Key-Value size
- BlobDB (Blob file, BlobDB cache, etc.)
- Etc.



Welcome to RocksDB

RocksDB is a storage engine with key/value interface, where keys and values are arbitrary byte streams. It is a C++ library. It was developed at Facebook based on LevelDB and provides backwards-compatible support for LevelDB APIs.

RocksDB supports various storage hardware, with fast flash as the initial focus. It uses a Log Structured Database Engine for storage, is written entirely in C++, and has a Java wrapper called RocksJava. See RocksJava Basics.

RocksDB can adapt to a variety of production environments, including pure memory, Flash, hard disks or remote storage. Where RocksDB cannot automatically adapt, highly flexible configuration settings are provided to allow users to tune it for them. It supports various compression algorithms and good tools for production support and debugging.

Features

- Designed for application servers wanting to store up to a few terabytes of data on local or remote storage systems.
- Optimized for storing small to medium size key-values on fast storage -- flash devices or in-memory
- It works well on processors with many cores

Features Not in LevelDB

RocksDB introduces dozens of new major features. See the list of features not in LevelDB.

Getting Started

For a complete Table of Contents, see the sidebar to the right. Most readers will want to start with the Overview and the Basic Operations section of the Developer's Guide. Get your initial options set-up following Setup Options and Basic Tuning. Also check RocksDB FAQ. There is also a RocksDB Tuning Guide for advanced RocksDB users.

Pages 166

Contents

https://github.com/facebook/rocksdb/wiki

```
~/workspace-lsm/rocksdb$ ./db_bench --help
```

```
-block_cache_trace_sampling_frequency (Block cache trace sampling
  frequency, termed s. It uses spatial downsampling and samples accesses to
  one out of s blocks.) type: int32 default: 1
-block_protection_bytes_per_key (Enable block per key-value checksum
  protection. Supported values: 0, 1, 2, 4, 8.) type: uint32 default: 0
-block_restart_interval (Number of keys between restart points for delta
  encoding of keys in data block.) type: int32 default: 16
-block_size (Number of bytes in a block.) type: int32 default: 4096
-bloom_bits (Bloom filter bits per key. Negative means use default.Zero
  disables.) type: int32 default: -1
-bloom_locality (Control bloom filter probes locality) type: int32
  default: 0
-build_info (Print the build info via GetRocksBuildInfoAsString) type: bool
  default: false
-bytes_per_sync (Allows OS to incrementally sync SST files to disk while
  they are being written, in the background. Issue one request for every
  bytes_per_sync written. 0 turns it off.) type: uint64 default: 0
-cache_high_pri_pool_ratio (Ratio of block cache reserve for high pri
  blocks. If > 0.0, we also enable
  cache_index_and_filter_blocks_with_high_priority.) type: double
  default: 0
-cache_index_and_filter_blocks (Cache index/filter blocks in block cache.)
  type: bool default: false
-cache_low_pri_pool_ratio (Ratio of block cache reserve for low pri
  blocks.) type: double default: 0
-cache_numshardbits (Number of shards for the block cache is 2 **
  cache_numshardbits. Negative means use default settings. This is applied
  only if FLAGS_cache_size is non-negative.) type: int32 default: -1
-cache_size (Number of bytes to use as a cache of uncompressed data)
  type: int64 default: 33554432
-cache_type (Type of block cache.) type: string default: "lru_cache"
-cache_uri (Full URI for creating a custom cache object) type: string
  default: ""
```

NoSQL Database
2nd week

Hojin Shin, Guangxun Zhao
http://sslab.dankook.ac.kr/, https://sslab.dankook.ac.kr/~choijm

Thank You
Q & A ?

Presentation by Guangxun Zhao
guangxunzhao@dankook.ac.kr