2026 Winter RocksDB Study
2nd week

Dayeon Wee, Yongmin Lee
http://sslab.dankook.ac.kr/, https://sslab.dankook.ac.kr/~choijm

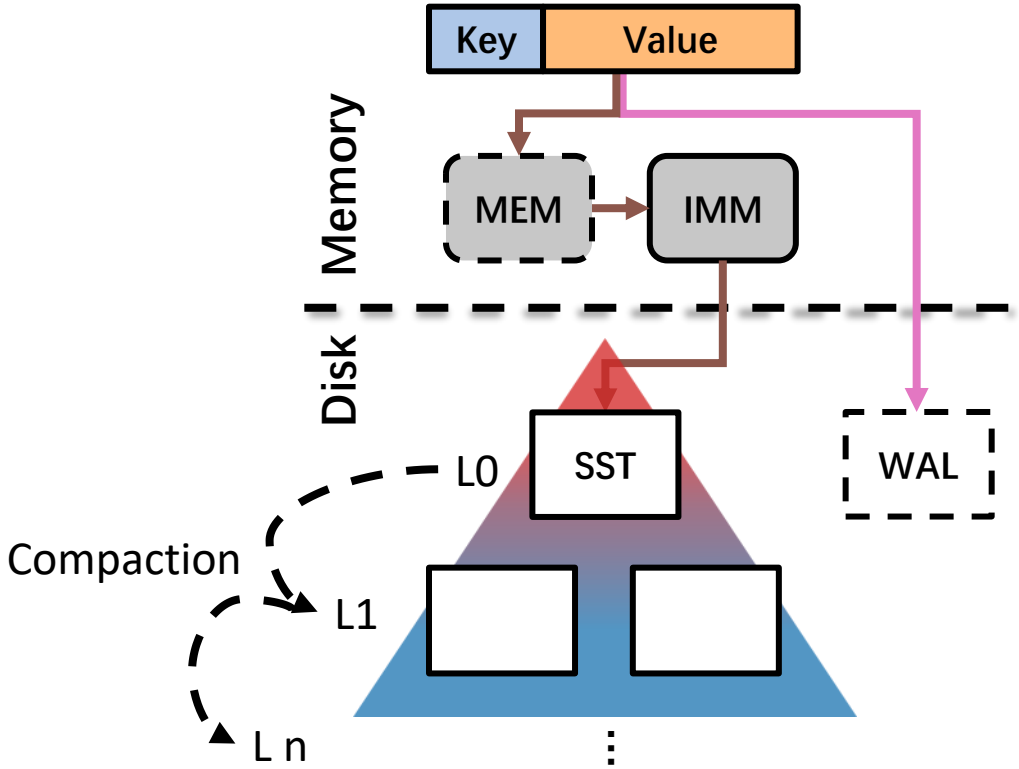
Presentation by Dayeon Wee
wida10@dankook.ac.kr

Contents

1. LSM-tree's Concept map
2. LSM-tree with Tiered Storage
   - NoveLSM
3. LSM-tree: How to Handle Large KV Sizes
   - WiscKey
     - LSM-tree's W/R amplification
     - WiscKey structure
4. LSM-tree with Machine Learning
   - Bourbon
5. Research Topics

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
© Springer-Verlag 1996

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

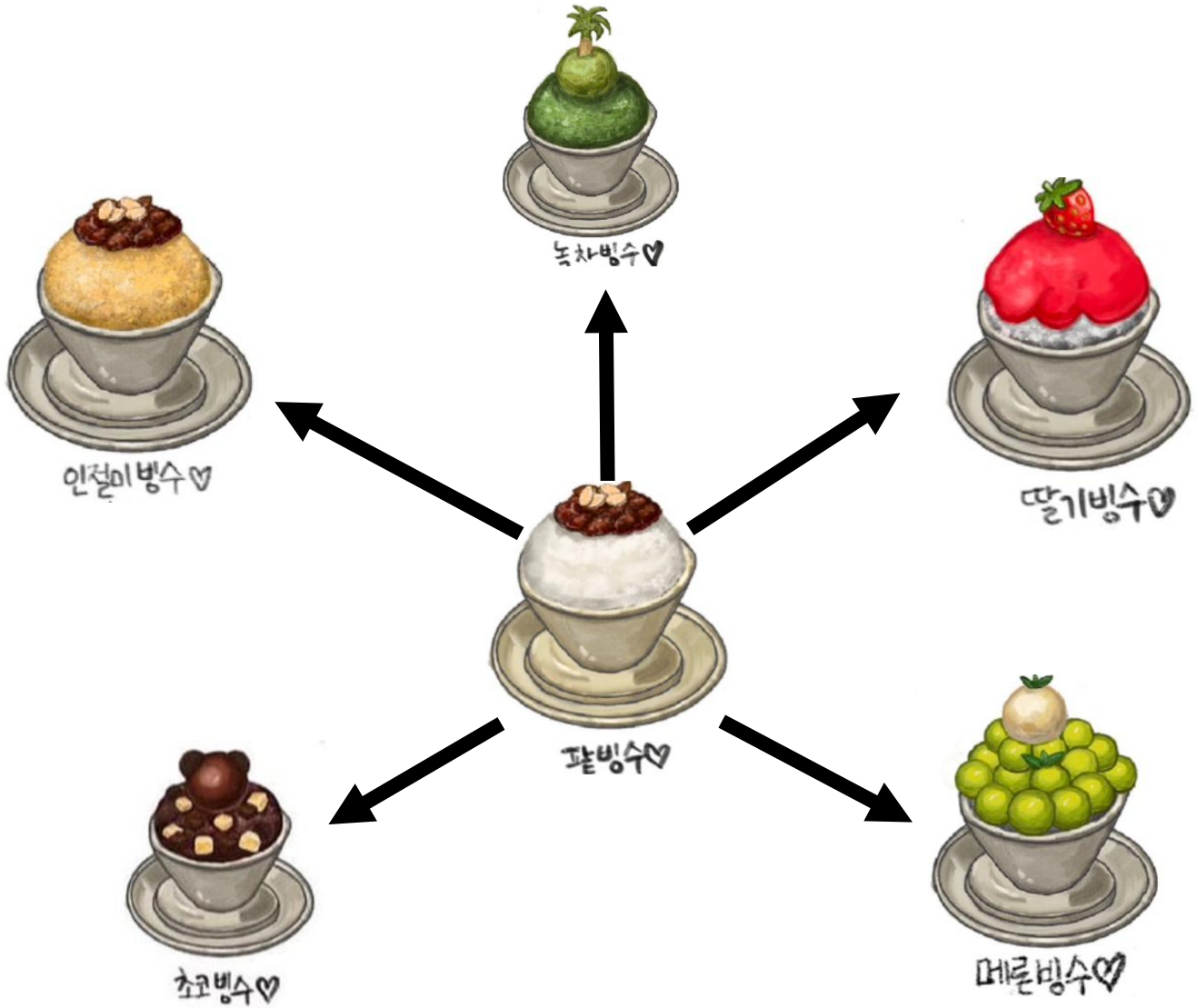
Patrick O'Neil[1], Edward Cheng[2], Dieter Gawlick[3], Elizabeth O'Neil[1]

[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



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.

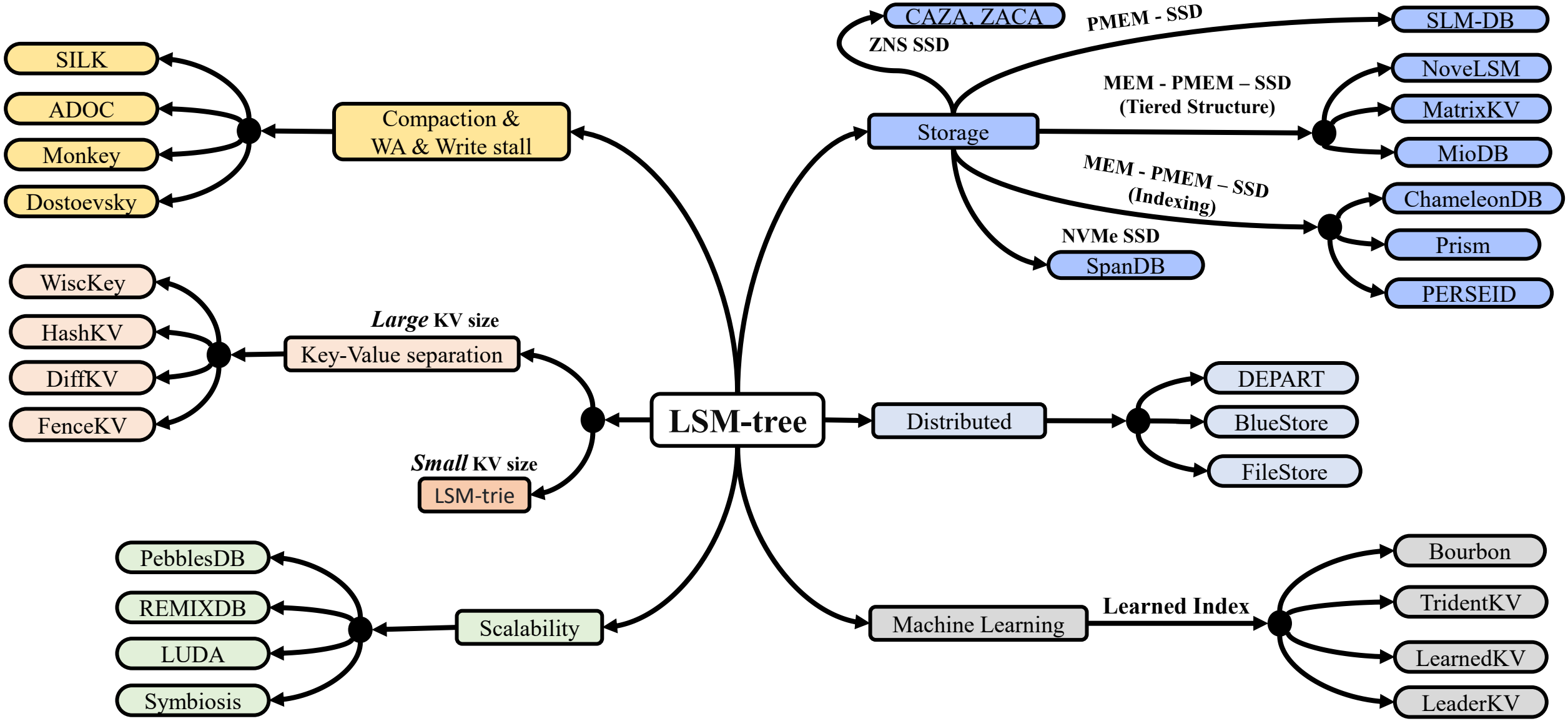
11세기 일본의 마쿠라노소시(枕草子)에는 얼음을 칼로 갈아 그 위에 칡즙을 뿌려서 먹었다는 기록이 나온다. 11세기 송나라 역사를 기록한 송사(宋史)에는 밀사빙(蜜沙氷)을 황제가 조정 대신에게 하사했다는 기록이 나온다. 밀사(蜜沙)는 꿀모래가 아니라 꿀에 버무린 팥소(豆沙)를 의미하고, 빙(氷)은 얼음을 의미한다.[2]

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

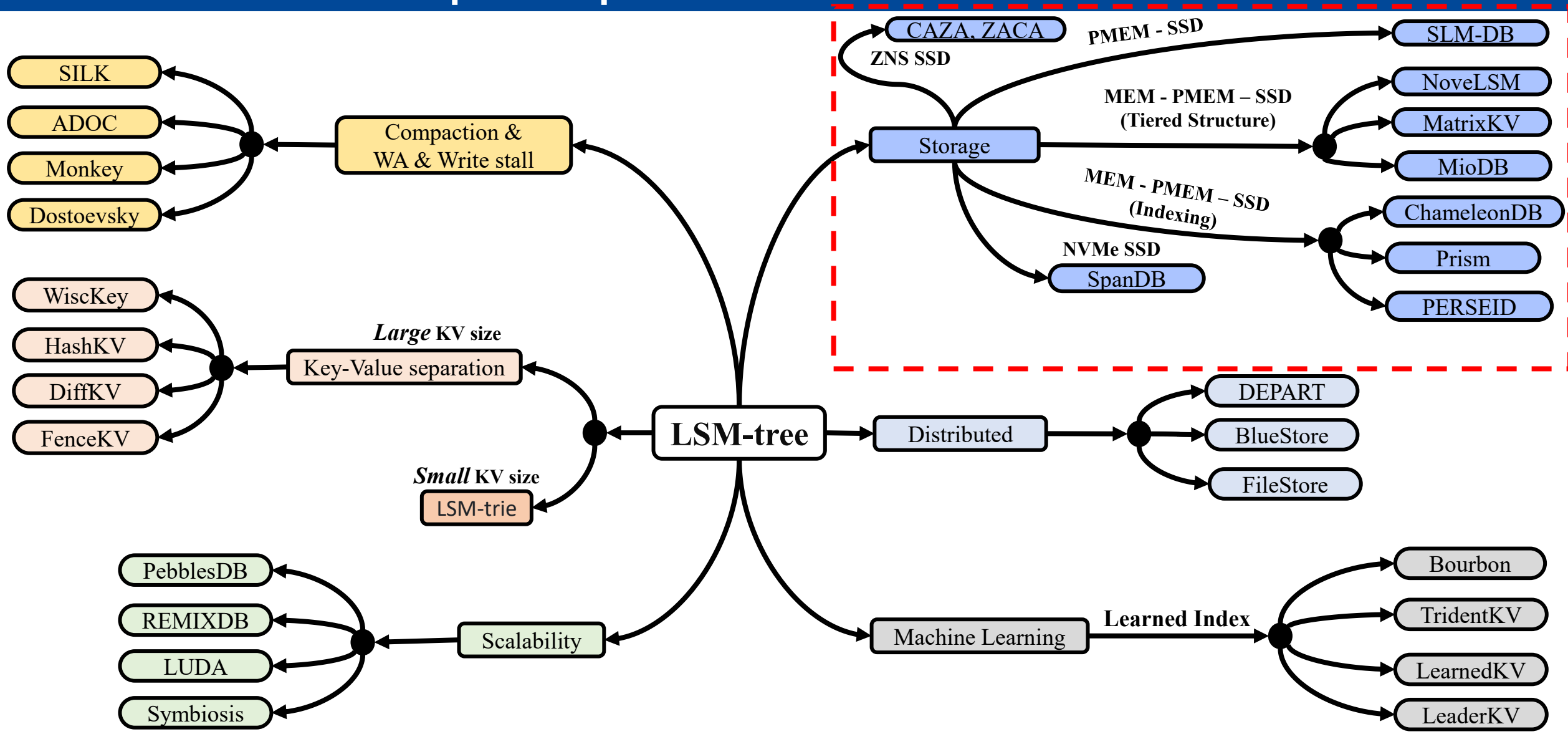
빙수는 기원전 3000년경 중국에서 눈이나 얼음에 꿀과 과일즙을 섞어 먹은 것에서 비롯됐다.[3]

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

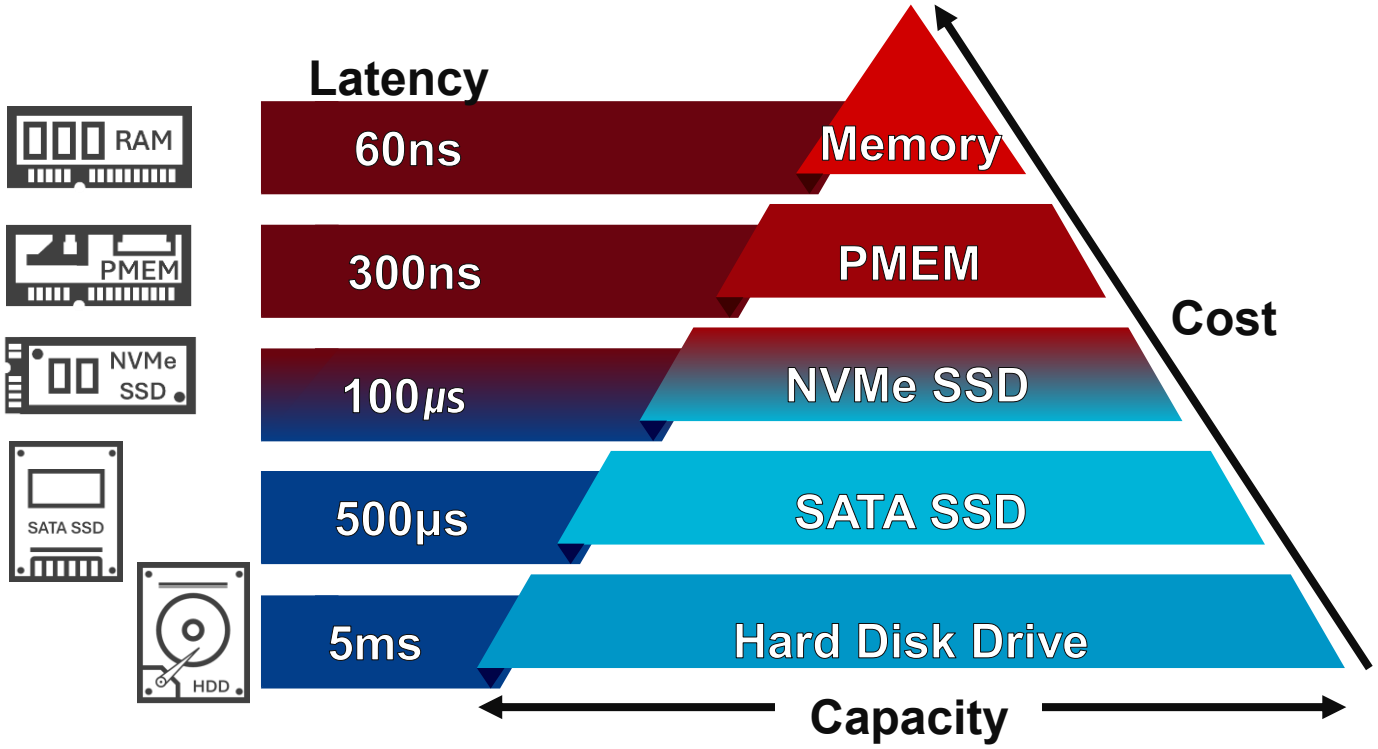
LSM-tree's Concept Map

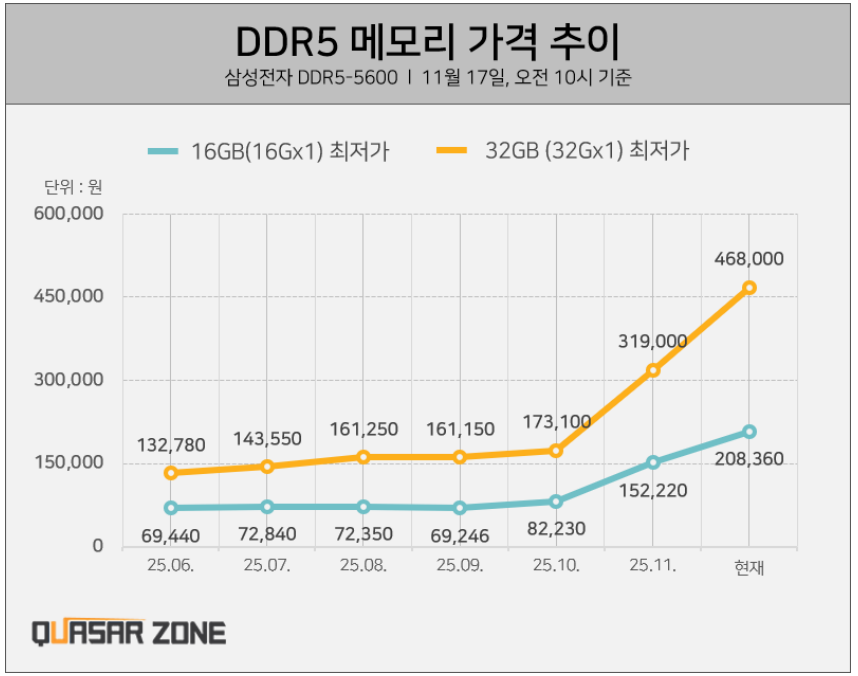


LSM-tree's Concept Map



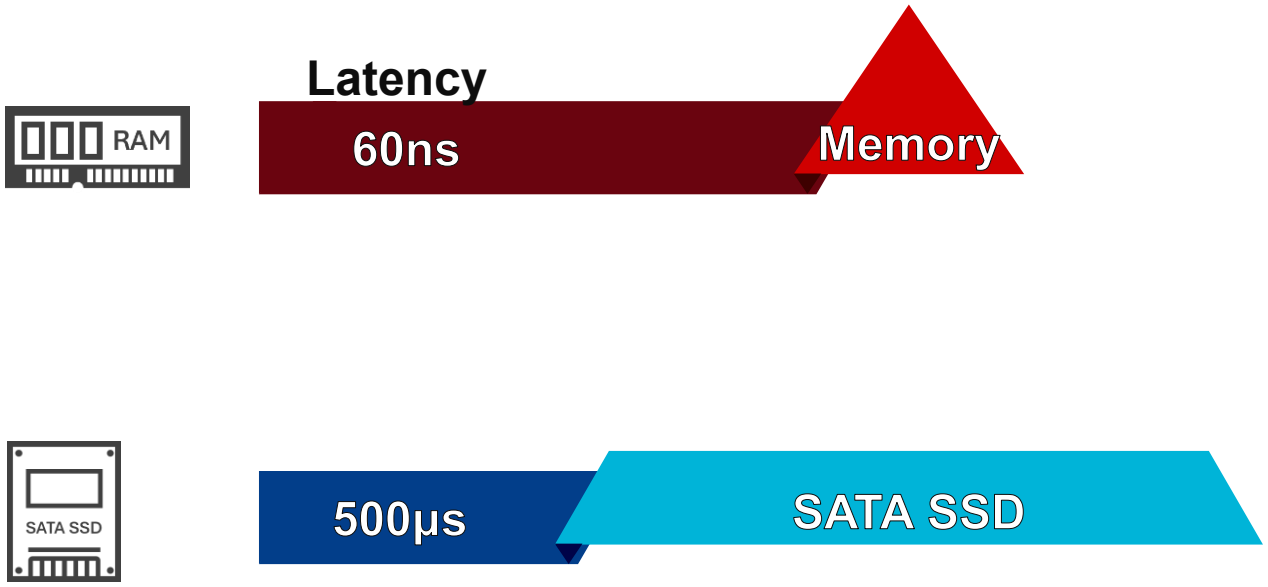
LSM-tree with Tiered Storage





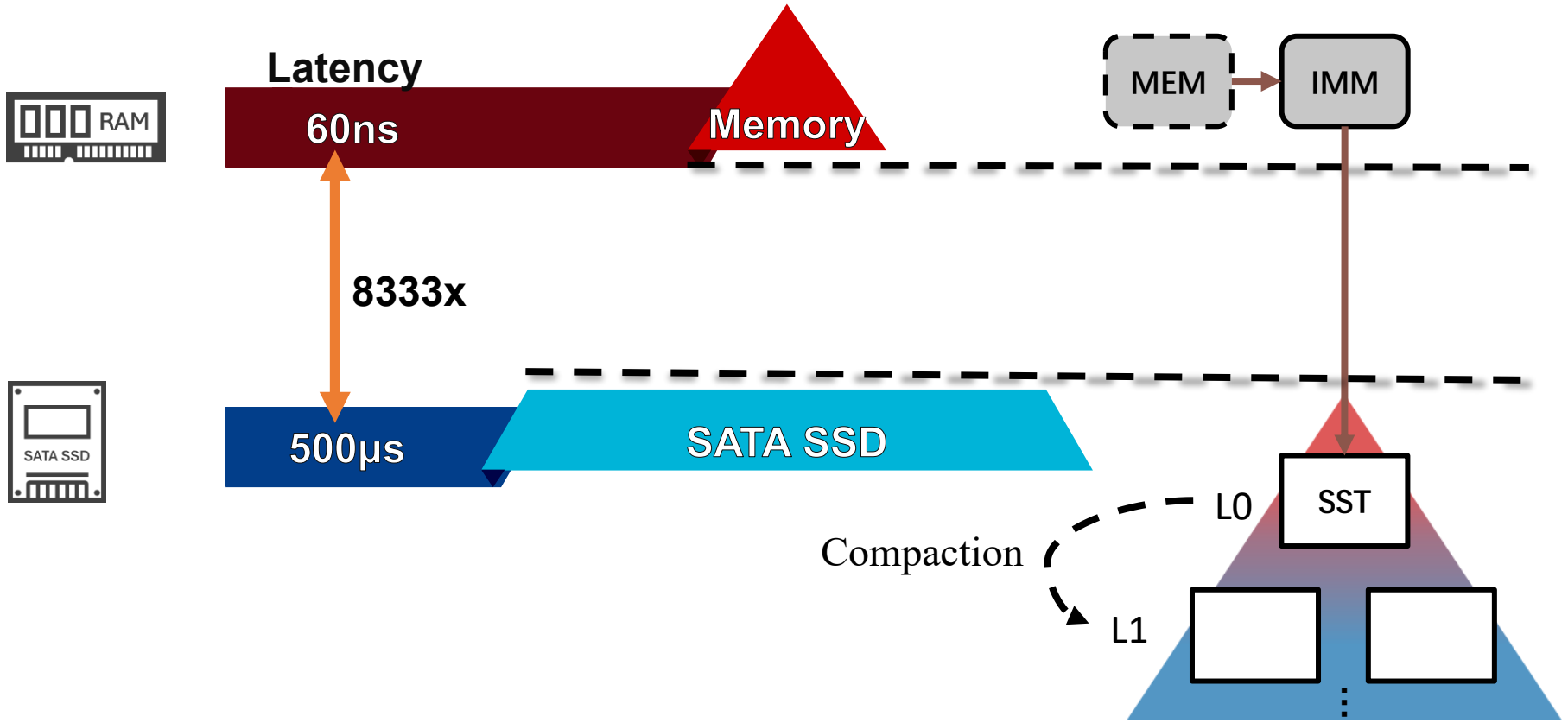
https://quasarzone.com/bbs/qc_plan/views/38707

LSM-tree with Tiered Storage

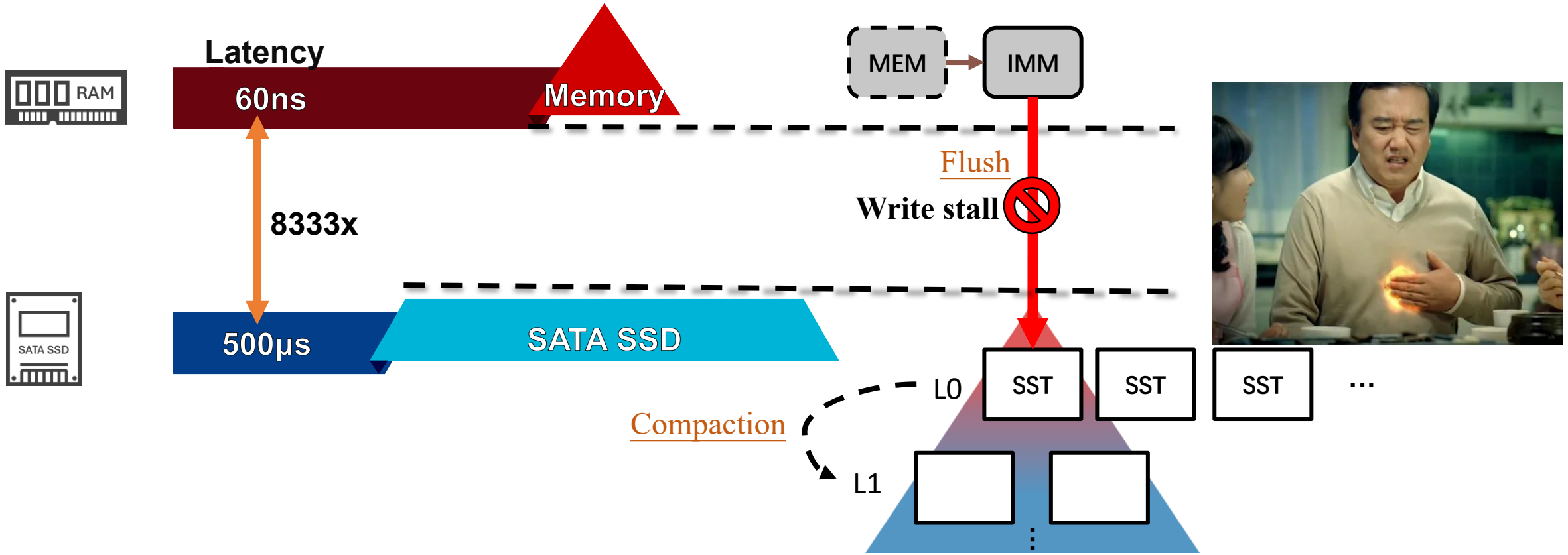


- Memory
  - Volatility
  - Low latency
  - Byte-addressability
- SSD
  - Non-volatile
  - Big capacity
  - Page/Block unit

LSM-tree with Tiered Storage

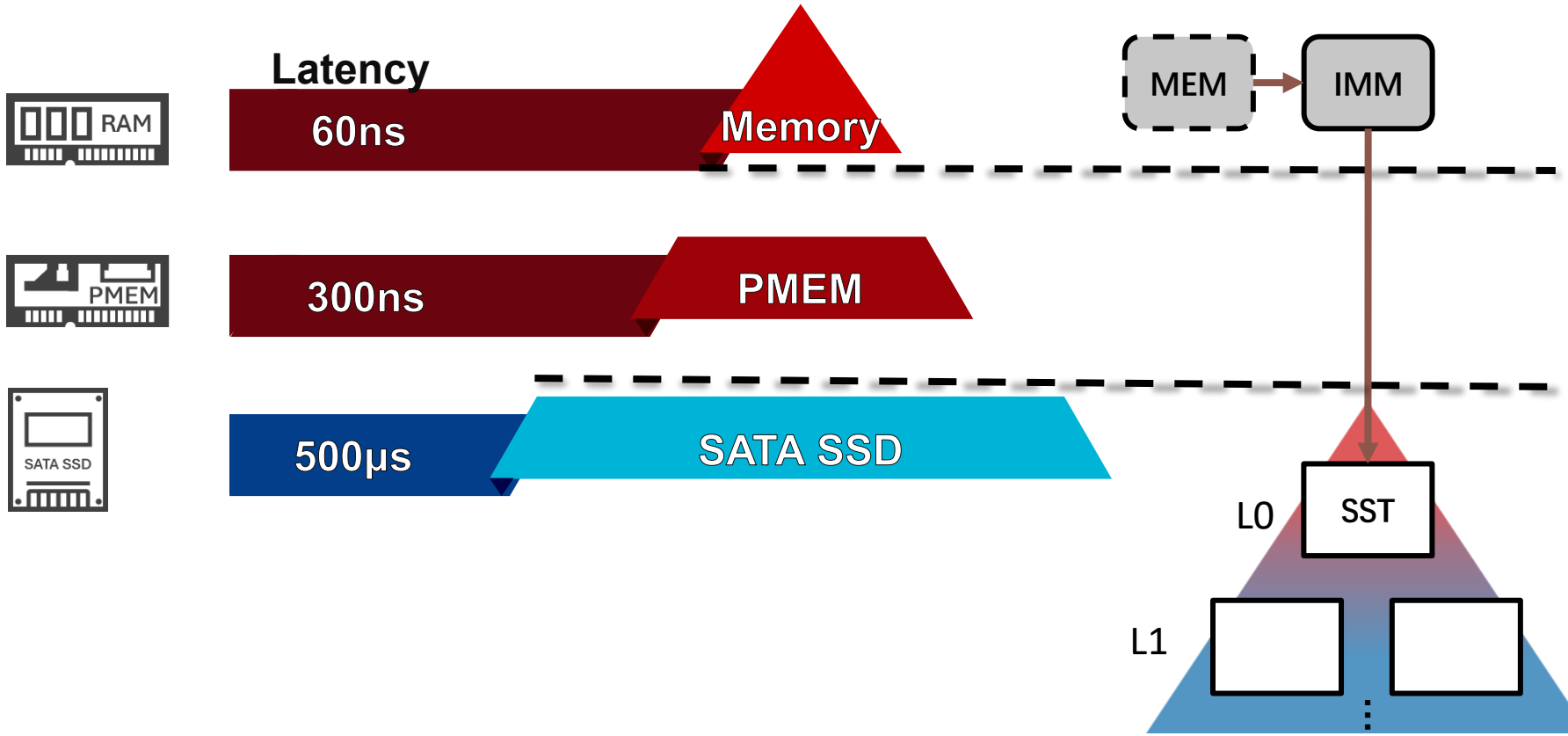


LSM-tree with Tiered Storage



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

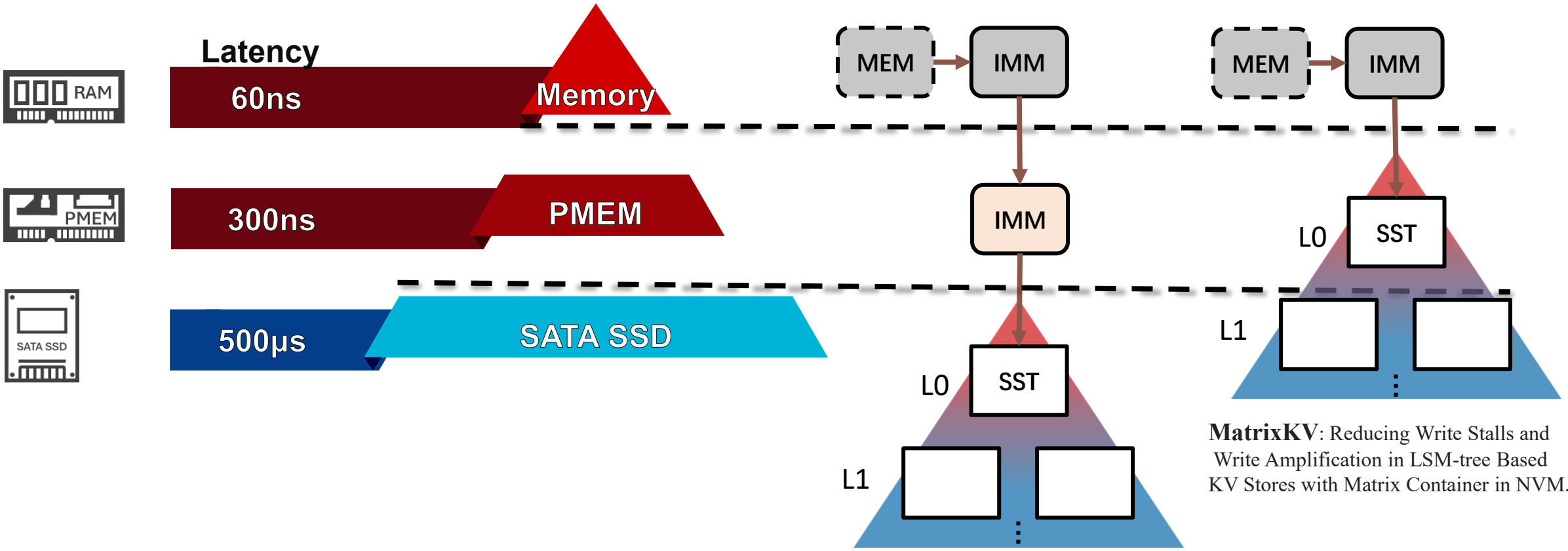


Persistent **MEM**ory

- 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

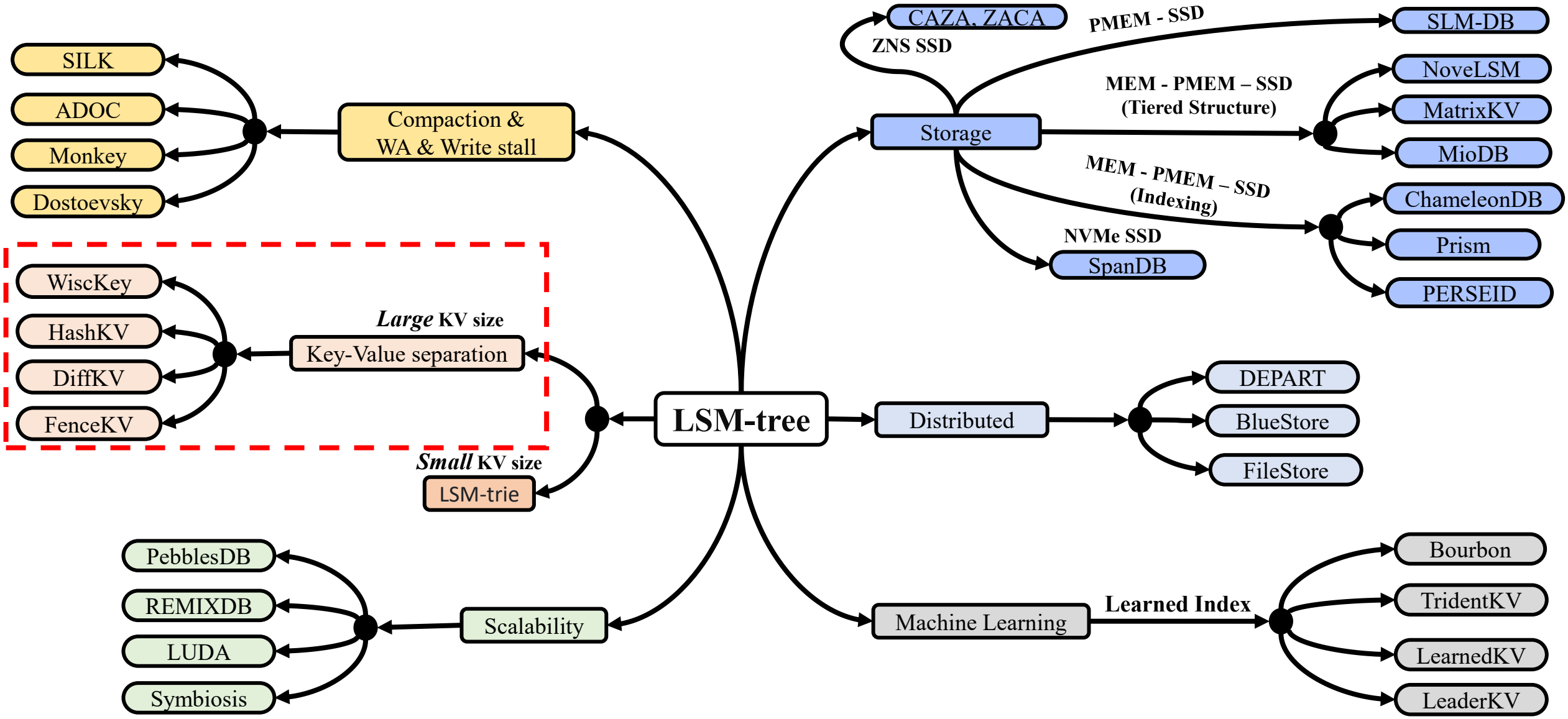
LSM-tree with Tiered Storage



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

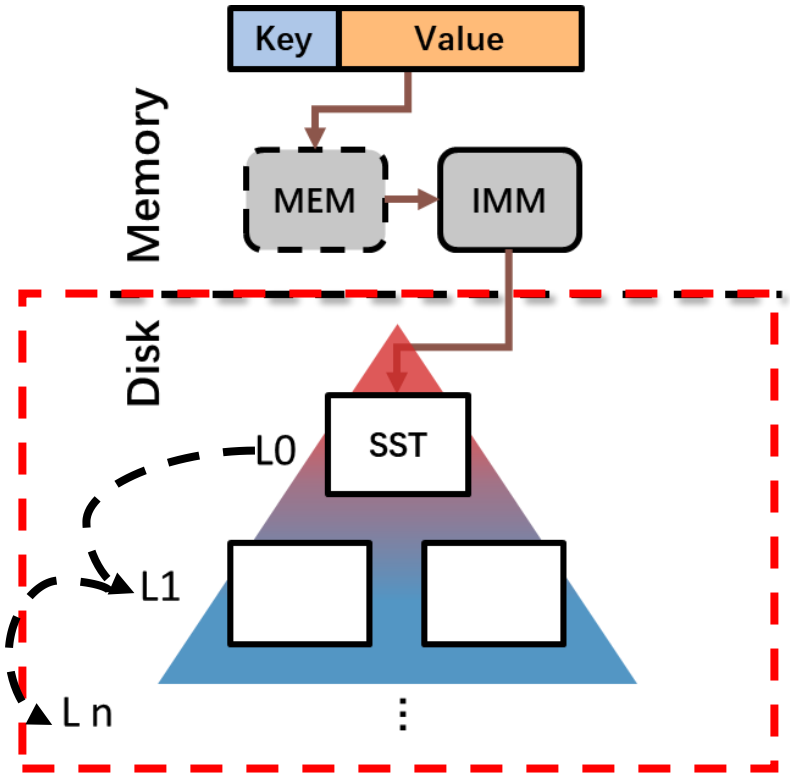
MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM.

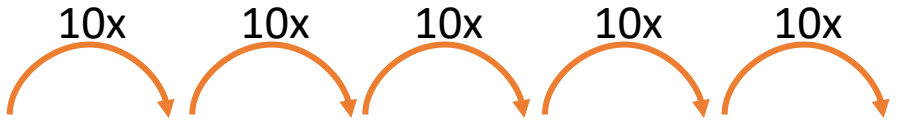
LSM-tree's Concept Map



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).

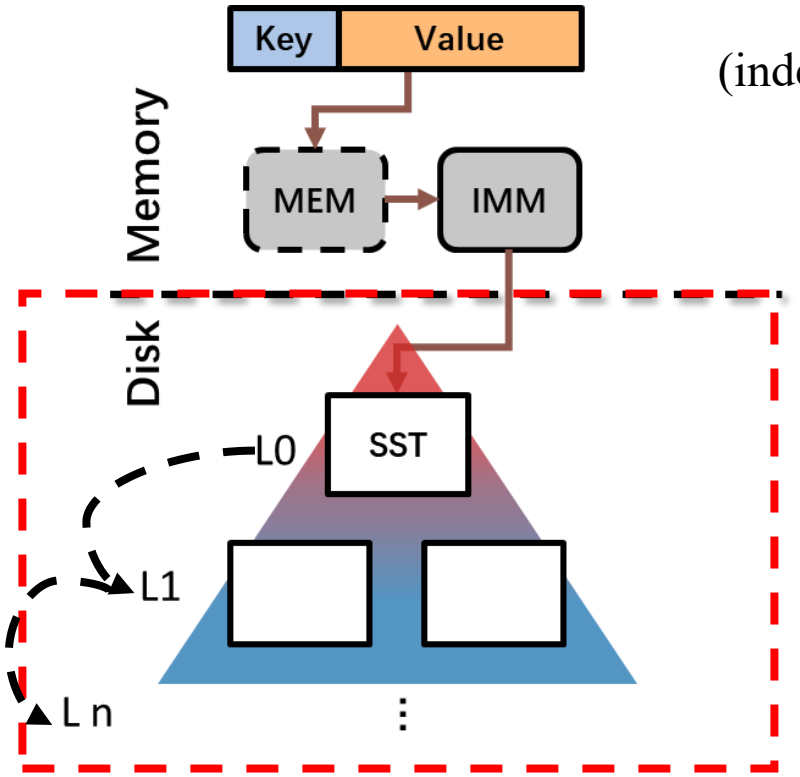




Level	0	1	2	3	4	5	6
Size	8MB	10MB	100MB	1GB	10GB	100GB	1TB

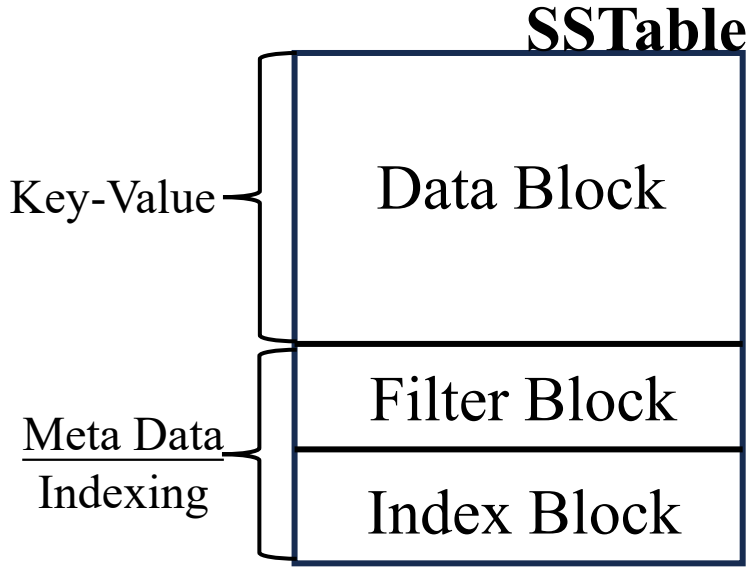
What is Read Amplification?

- 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.



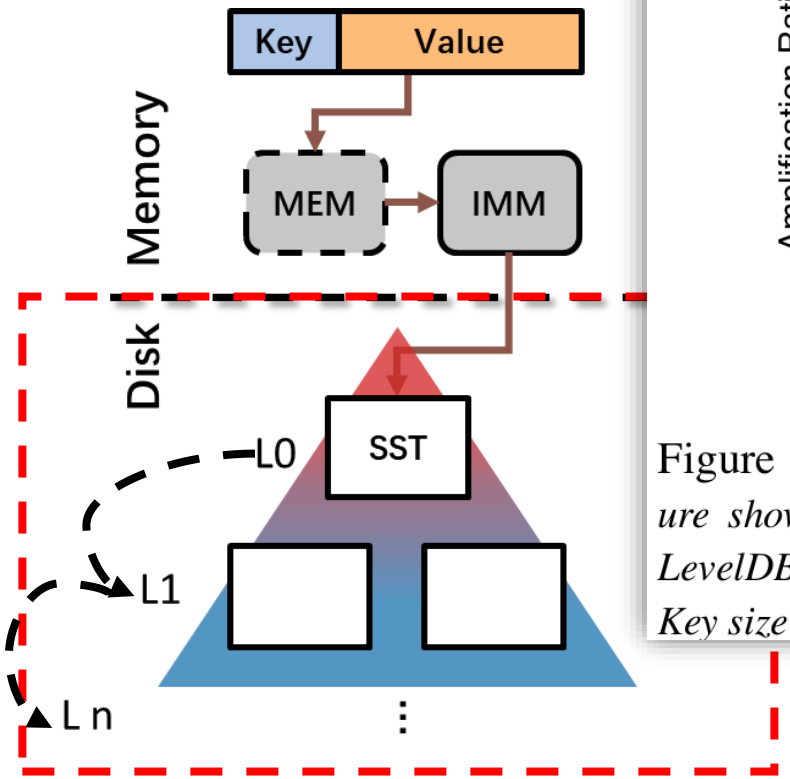
(index block + bloom-filter blocks + data block) * SSTable = Read amplification

(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.



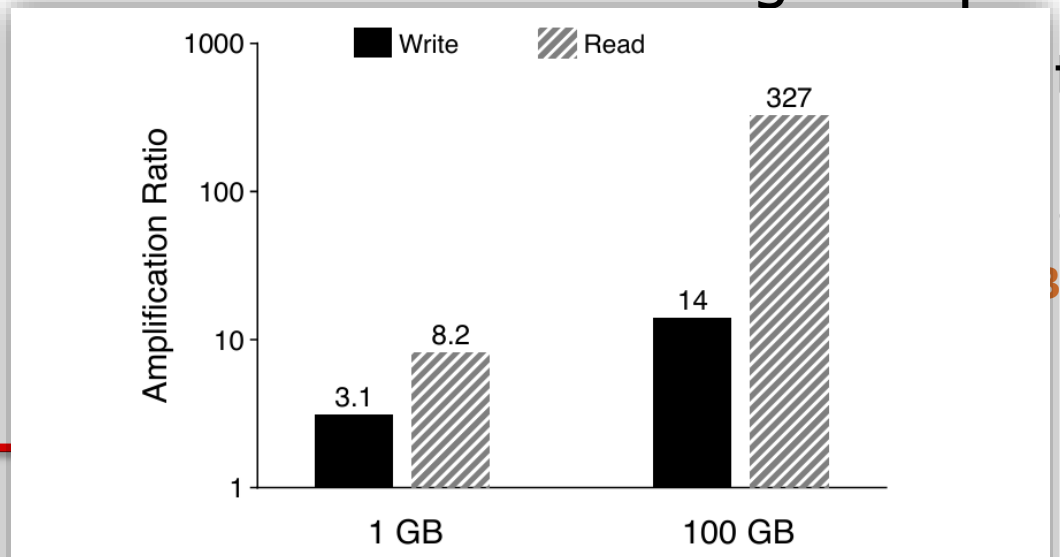
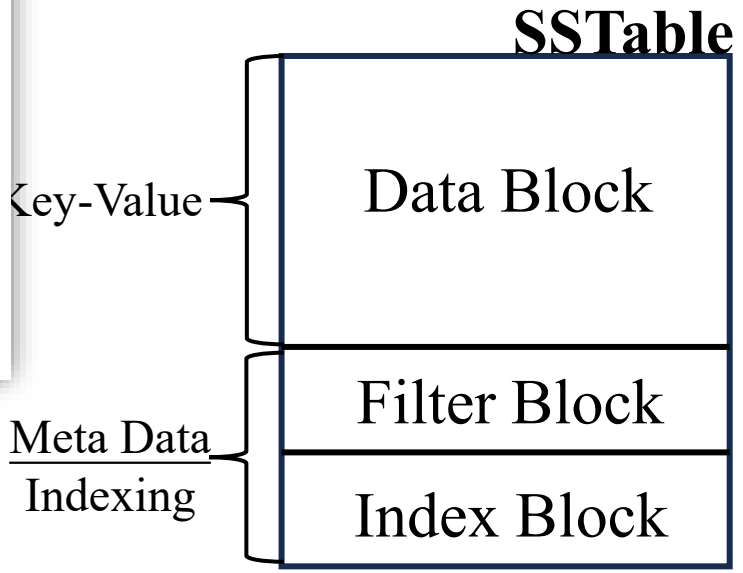
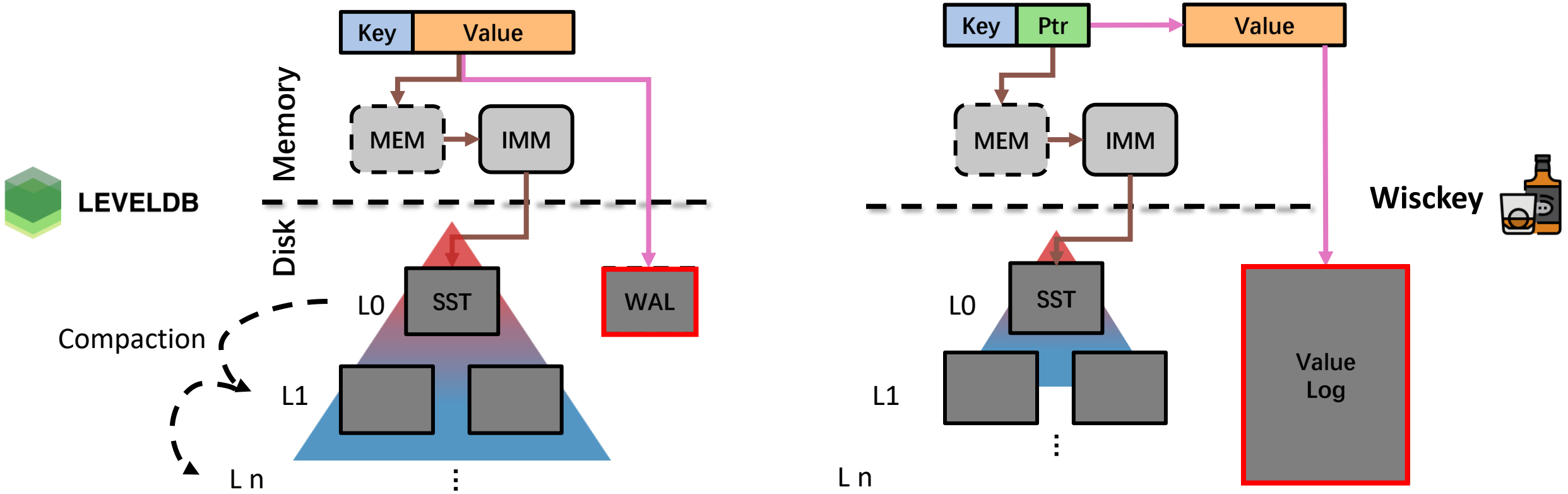


Figure 2: **Write and Read Amplification.** *This figure shows the write amplification and read amplification of LevelDB for two different database sizes, 1 GB and 100 GB. Key size is 16 B and value size is 1 KB.*

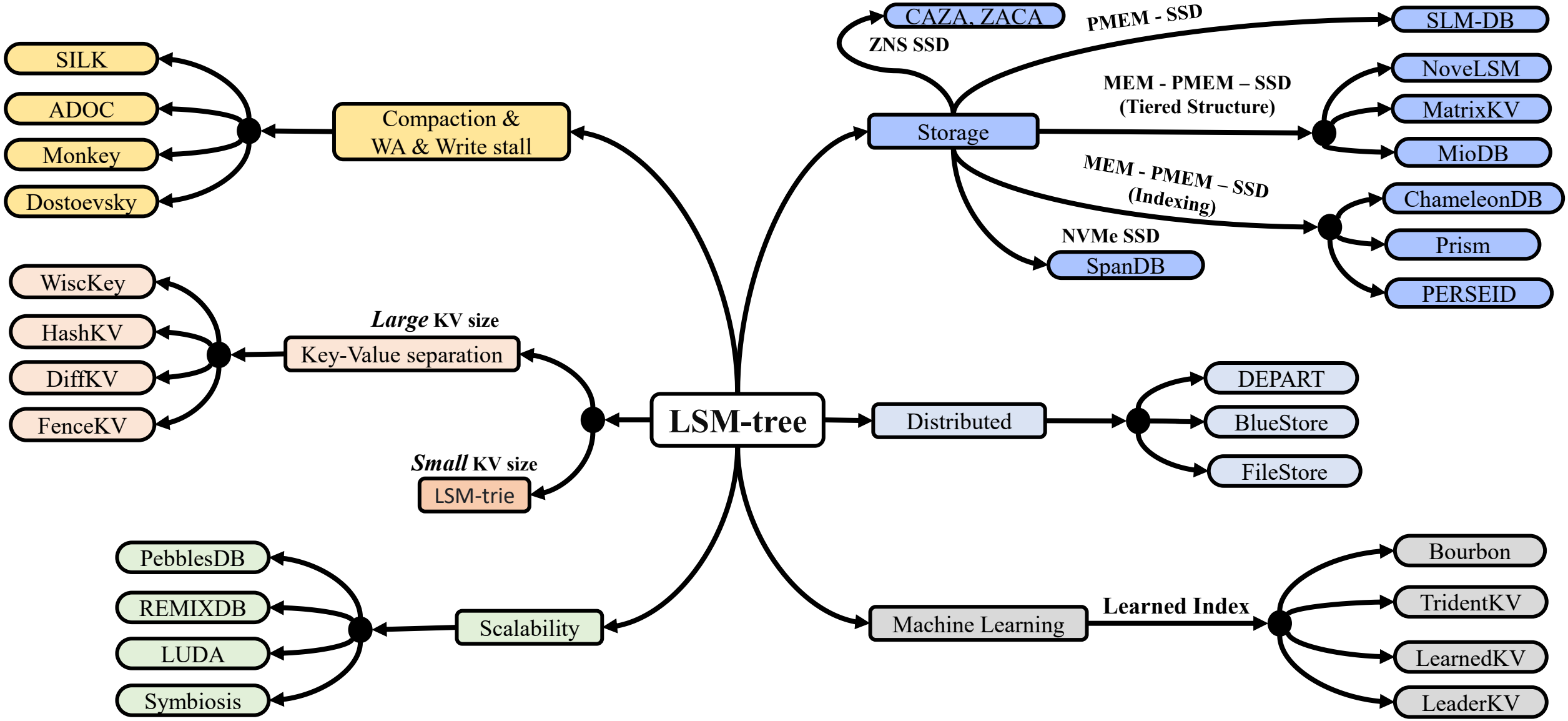
SSTable = Read amplification
36



WiscKey Structure



LSM-tree's Concept Map



2026 Winter RocksDB Study
2nd week

Dayeon Wee, Yongmin Lee
http://sslab.dankook.ac.kr/, https://sslab.dankook.ac.kr/~choijm

Thank You
Q & A ?

Presentation by Dayeon Wee
wida10@dankook.ac.kr