# ChameleonDB: a Key-value Store for Optane Persistent memory

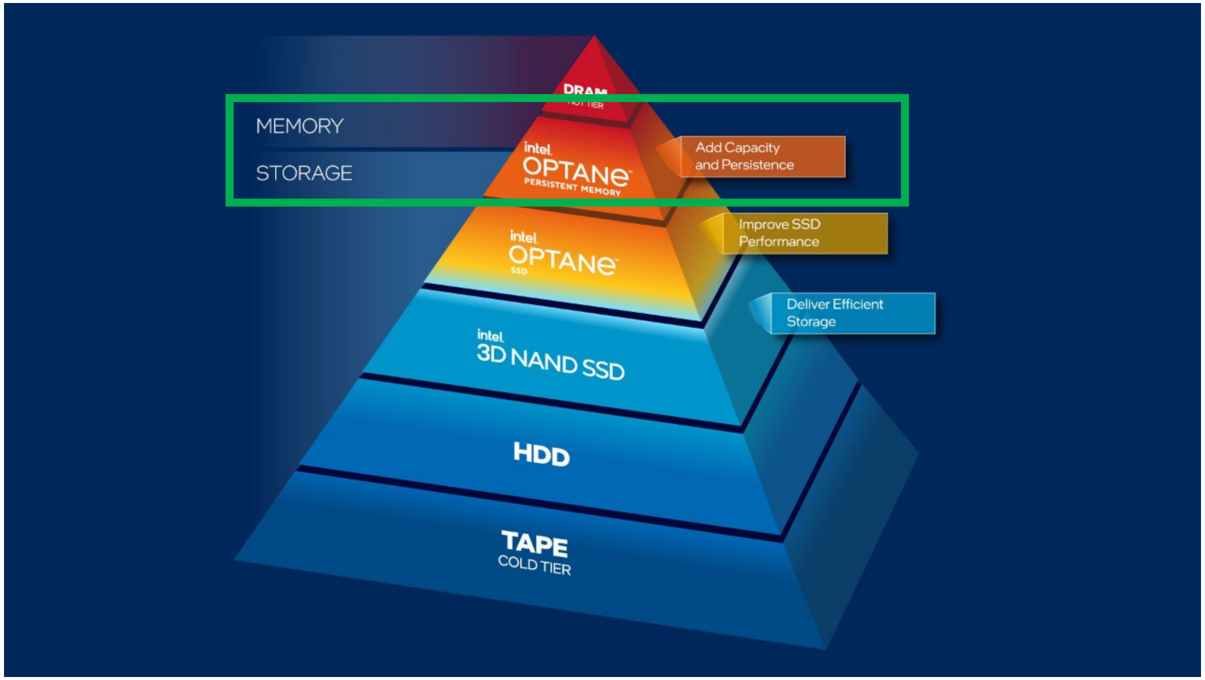
Wenhui Zhang, Xingsheng Zhao,

Song Jiang, and Hong Jiang

University of Texas at Arlington

UNIVERSITY OF TEXAS ARLINGTON

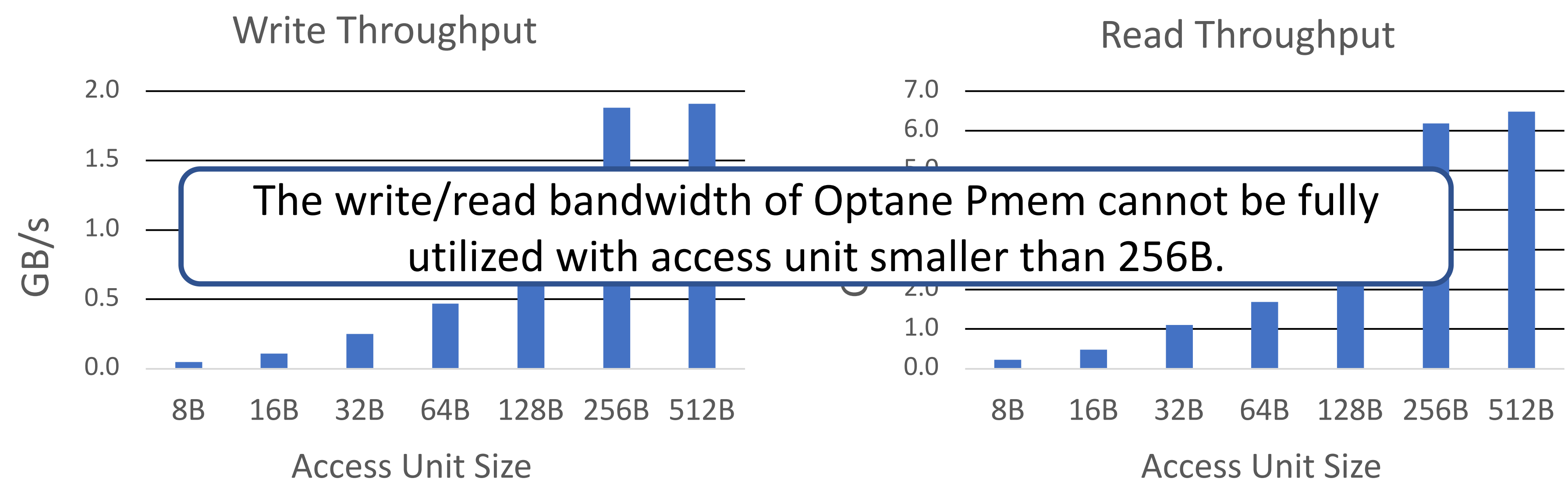
# Why do we need a key-value store design for Optane persistent memory?




Because Optane Persistent Memory (Pmem) is very **different.**

It's different from **DRAM**, different from **traditional block devices**, and even different from **what was assumed about persistent memory** (slower, persistent DRAM).

# Optane Pmem is a block device with access unit as 256B.



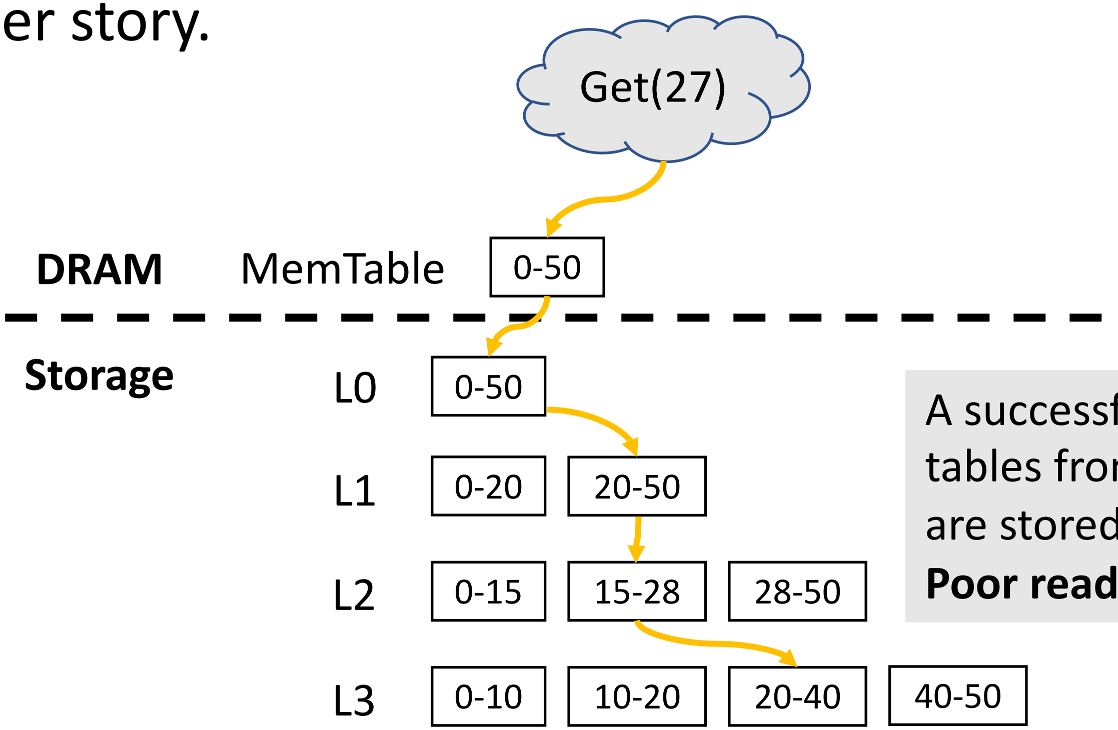

The write/read bandwidth of Optane Pmem cannot be fully utilized with access unit smaller than 256B.

This property makes Optane Pmem different from what was assumed about persistent memory (cacheline as access unit).

KV store designs **employing small random writes** of persistent memory, including Level Hashing, CCEH, and FAST&FAIR, are **unable** to provide high **write performance** on Optane Pmem.

# Are LSM-tree based KV stores designed for block devices efficient for Optane Pmem?

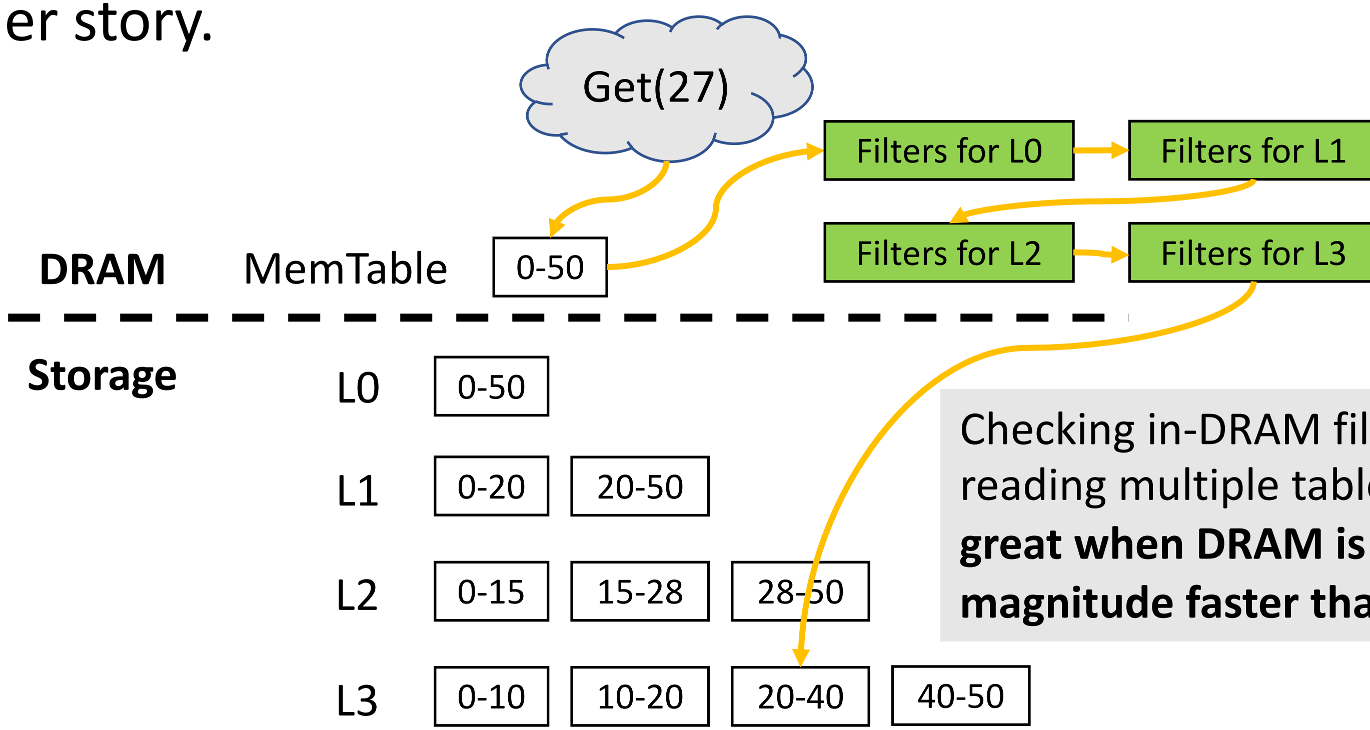
They are efficient on write performance, but **read performance** is another story.




A successful Get may need to read tables from multiple levels, which are stored in the slow block device. **Poor read performance.**

# Are LSM-tree based KV stores designed for block devices efficient for Optane Pmem?

They are efficient on write performance, but **read performance** is another story.

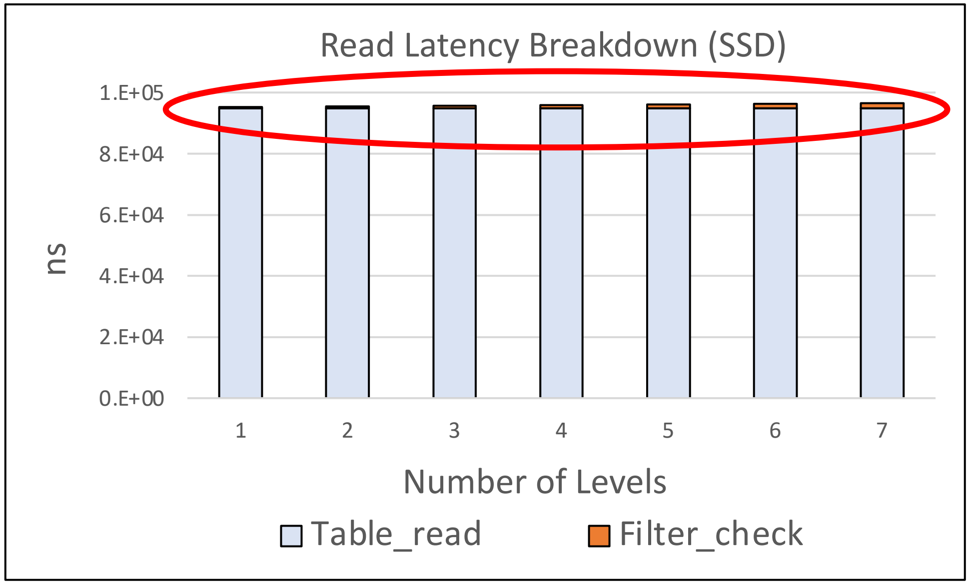



Checking in-DRAM filters to bypass reading multiple tables. **Works great when DRAM is orders of magnitude faster than the storage.**

# Are LSM-tree based KV stores designed for block devices efficient for Optane Pmem?

They are efficient on write performance, but **read performance** is another story.

When use **SSD** as storage, the time to checking filters in DRAM for multiple levels is **negligible**.



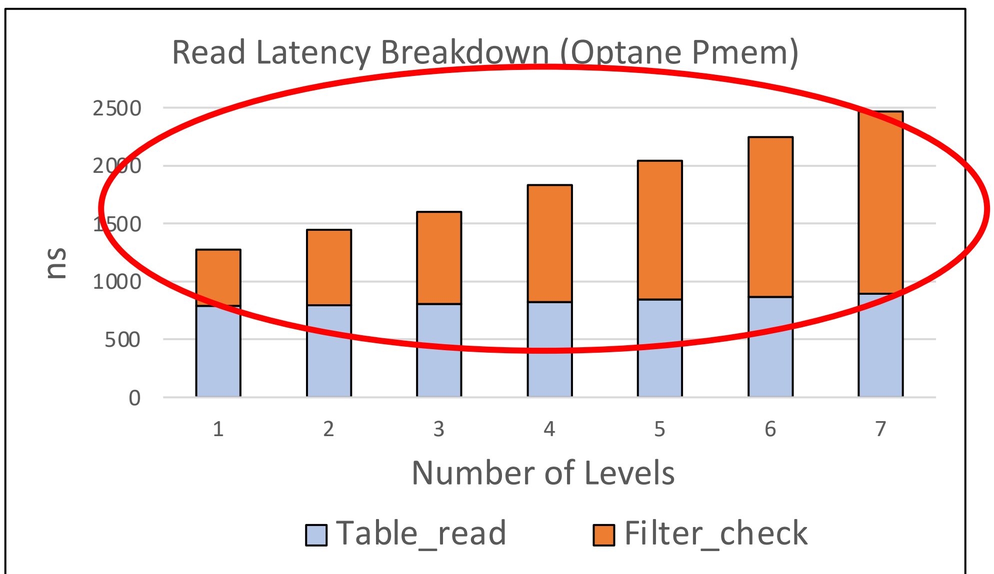

Reading table from the slow storage (SSD) contributes to 99% of the read latency, while checking multiple in-DRAM filters is nearly negligible.

Read latency is stable with the multi-level structure.

# Are LSM-tree based KV stores designed for block devices efficient for Optane Pmem?

They are efficient on write performance, but **read performance** is another story.

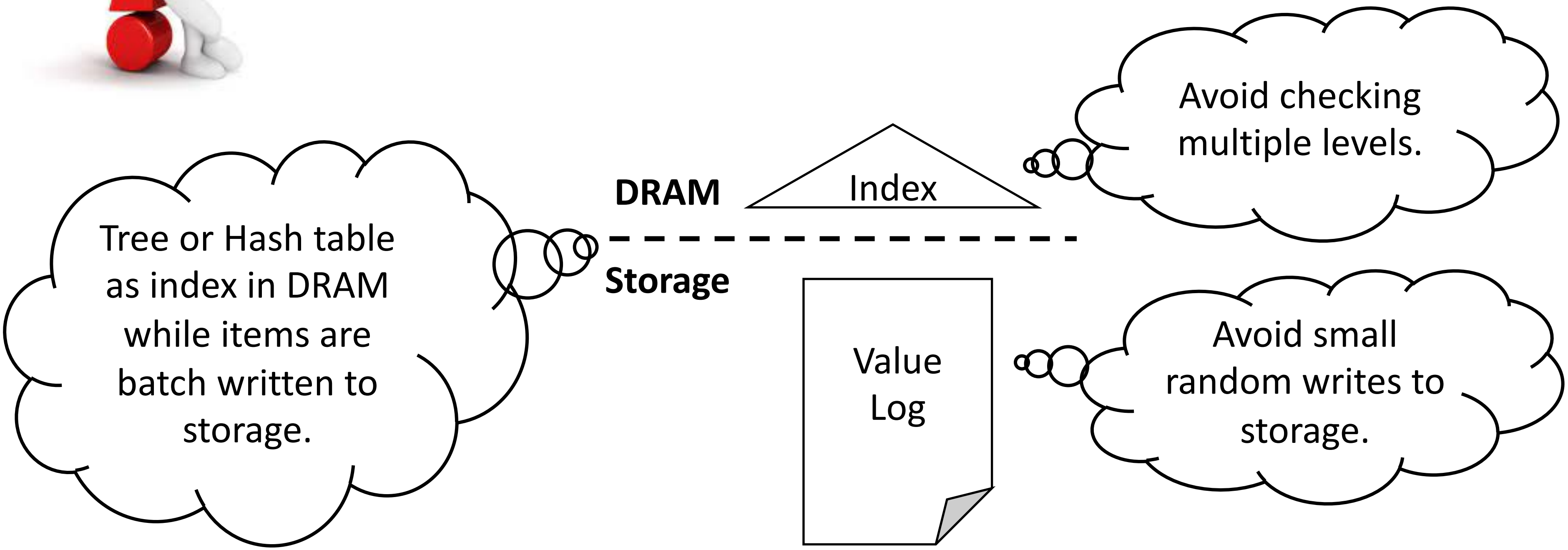
When use **<u>Optane Pmem</u>**, whose latency is **~3x DRAM's**, the time to checking filters for multiple levels becomes **<u>significant</u>**.




Checking filters for multiple levels contributes up to 63% of the read latency.

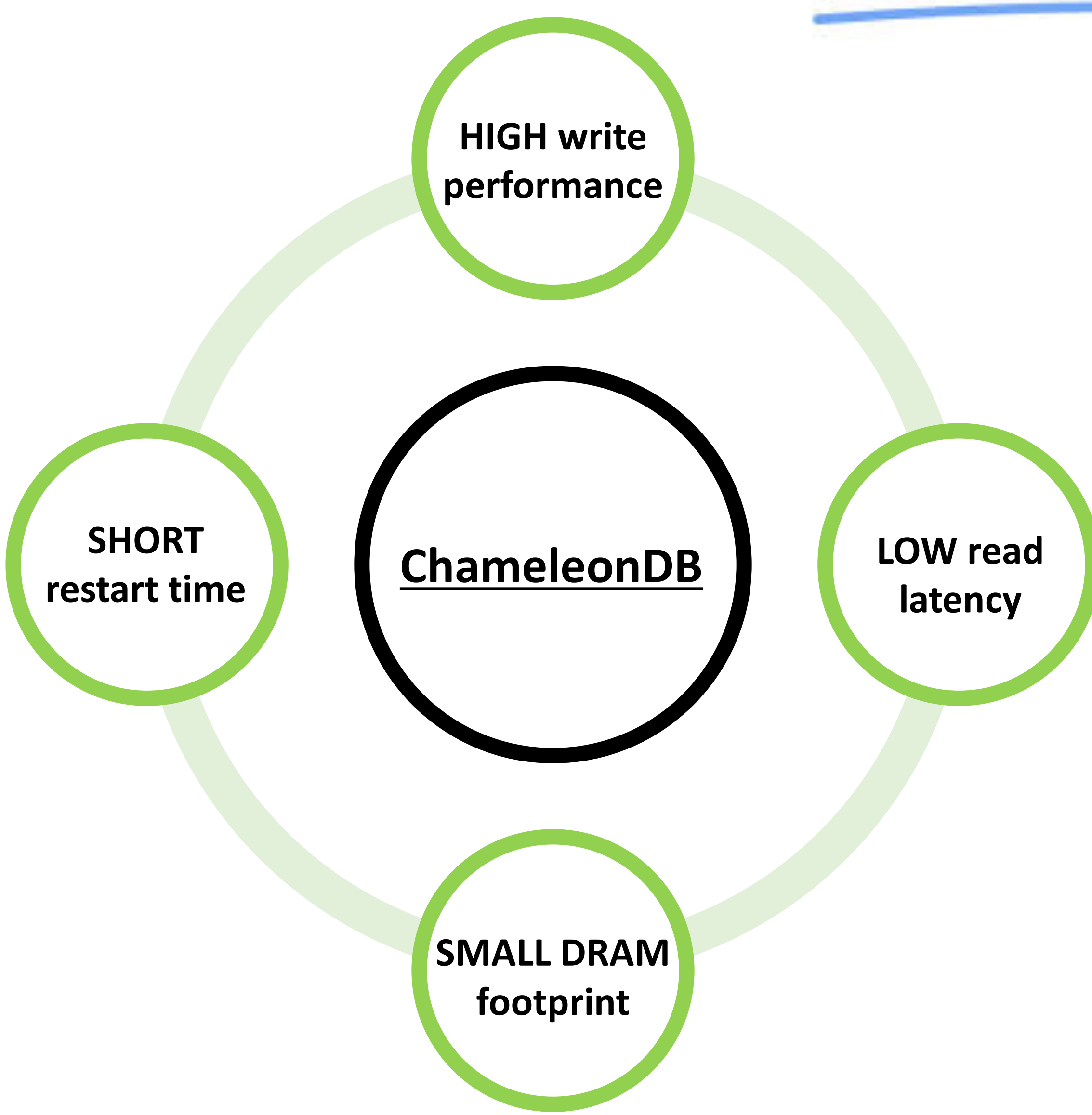
**Multi-level structure** becomes a major **barrier** to achieving consistently low **read latency**.

# How about Log structure with in-DRAM index?

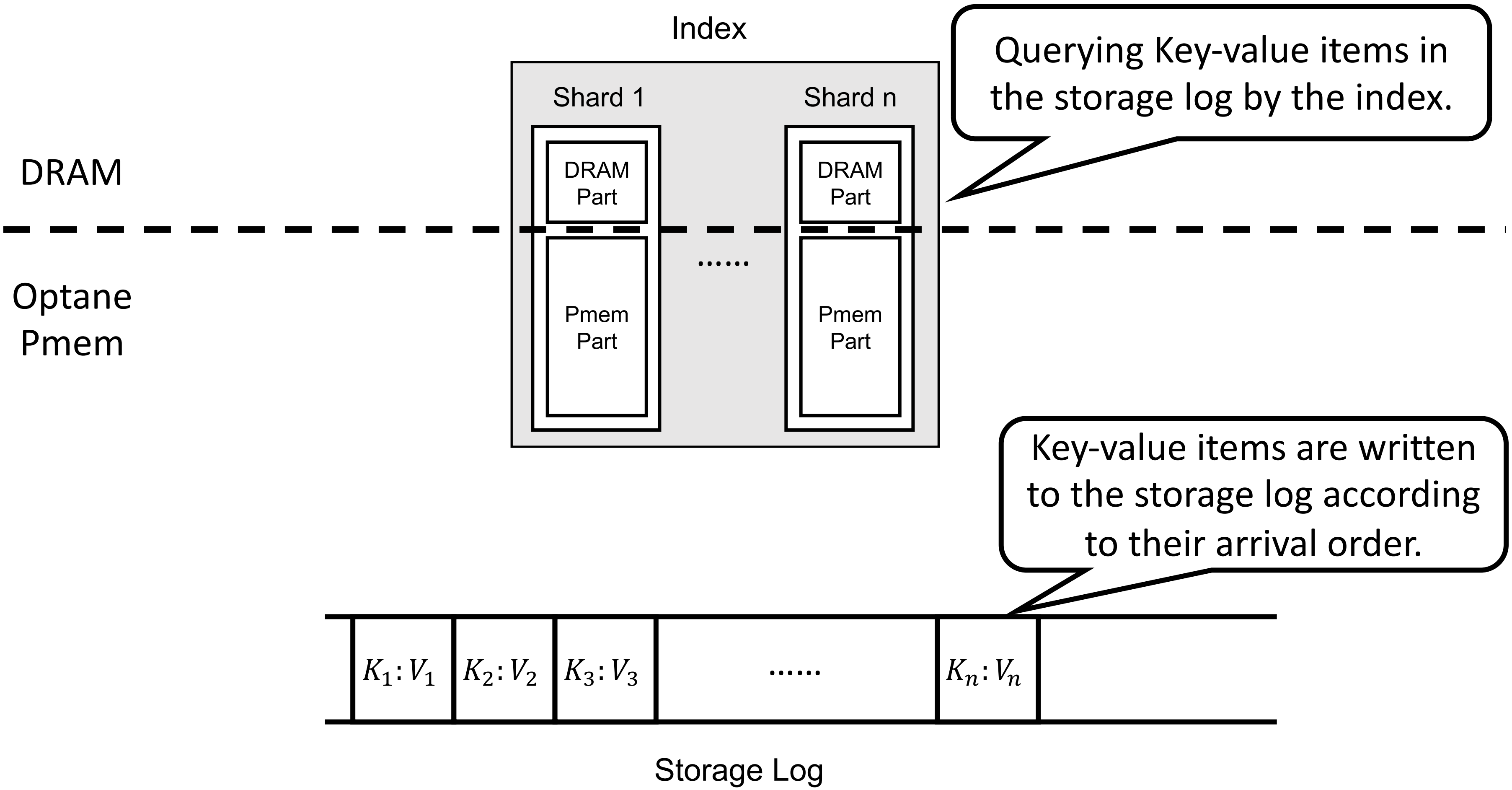



However, **DRAM footprint** for the index is considerably **large**, and recovering the index during a **restart** may take an **unacceptable long time**.

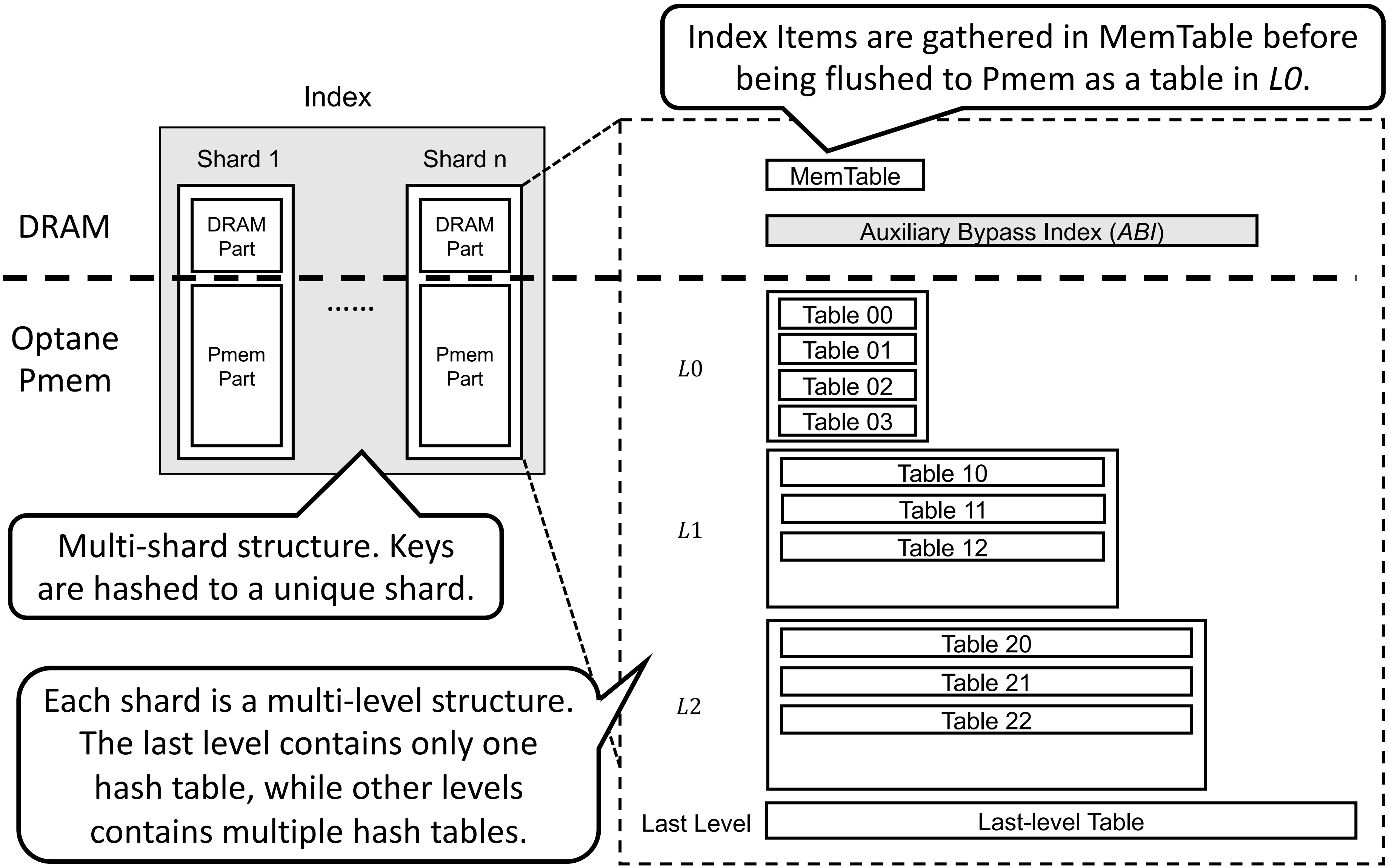
# Design Goals of ChameleonDB

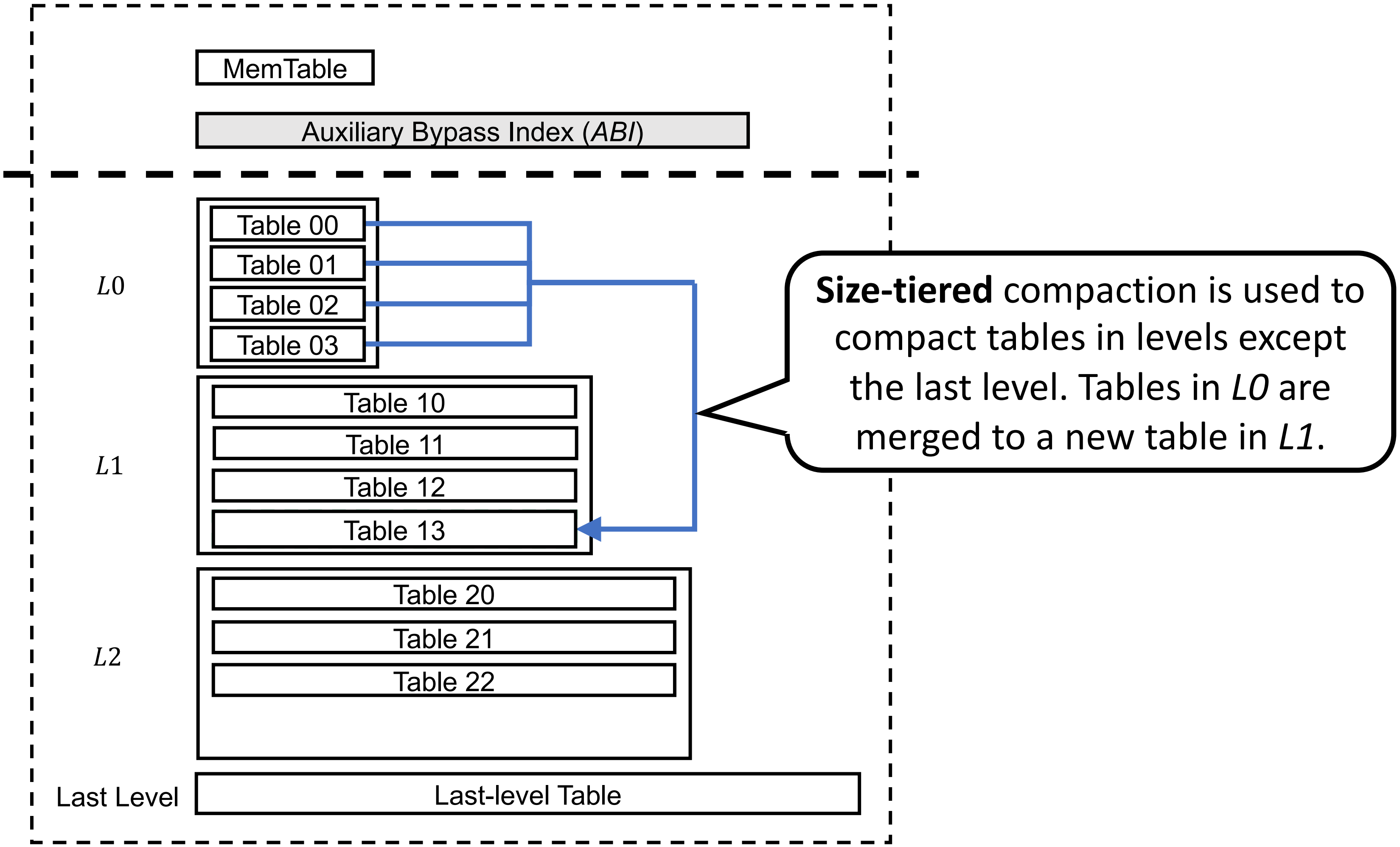
# Structure of ChameleonDB



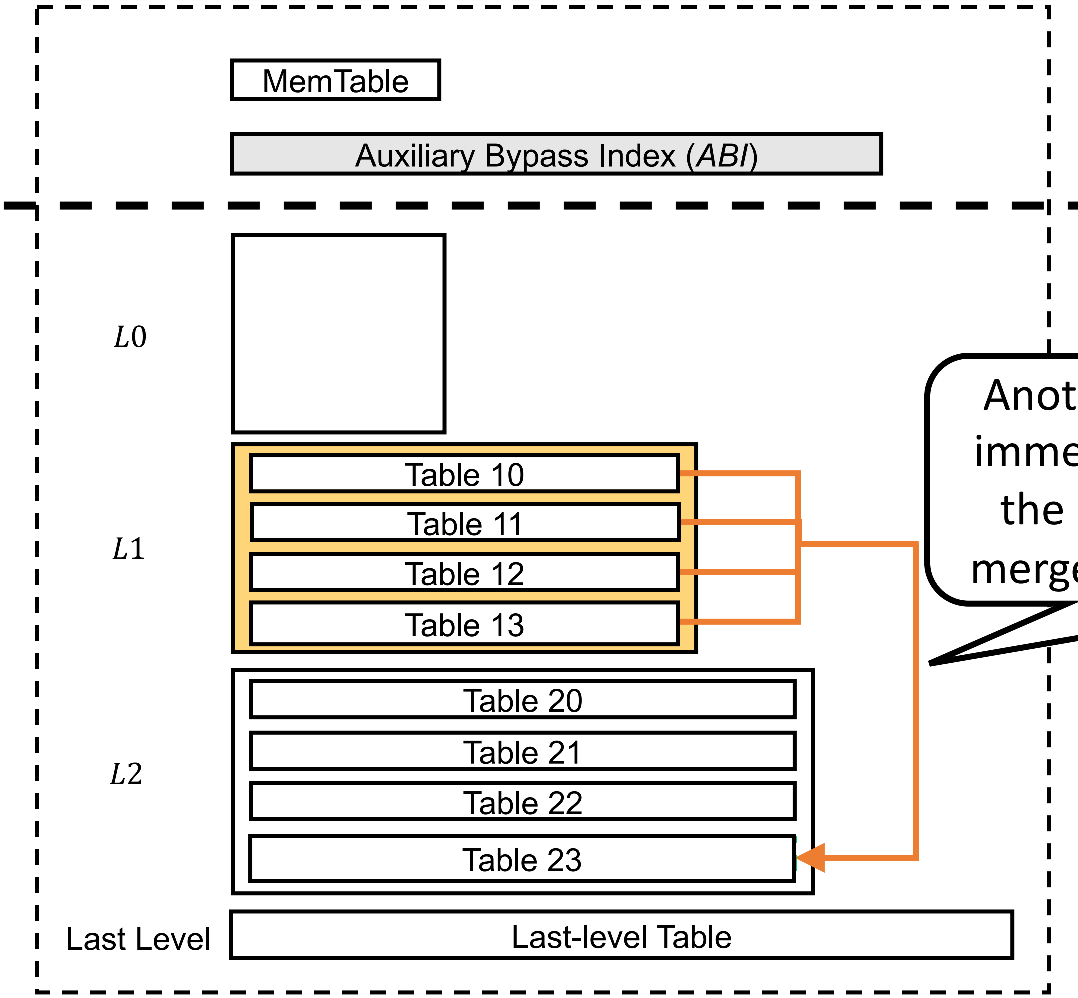
# Structure of ChameleonDB



# Size-tiered Compaction in ChameleonDB

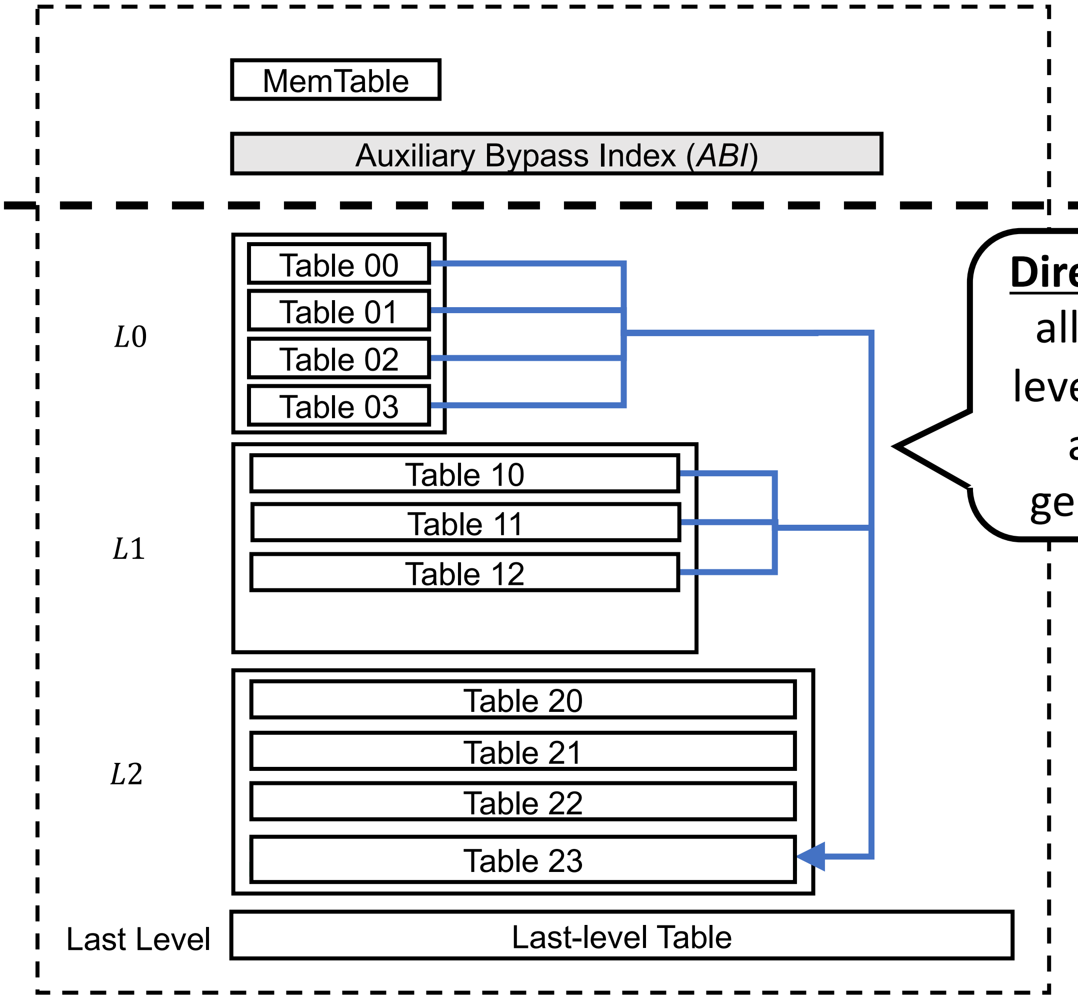


# Size-tiered Compaction in ChameleonDB



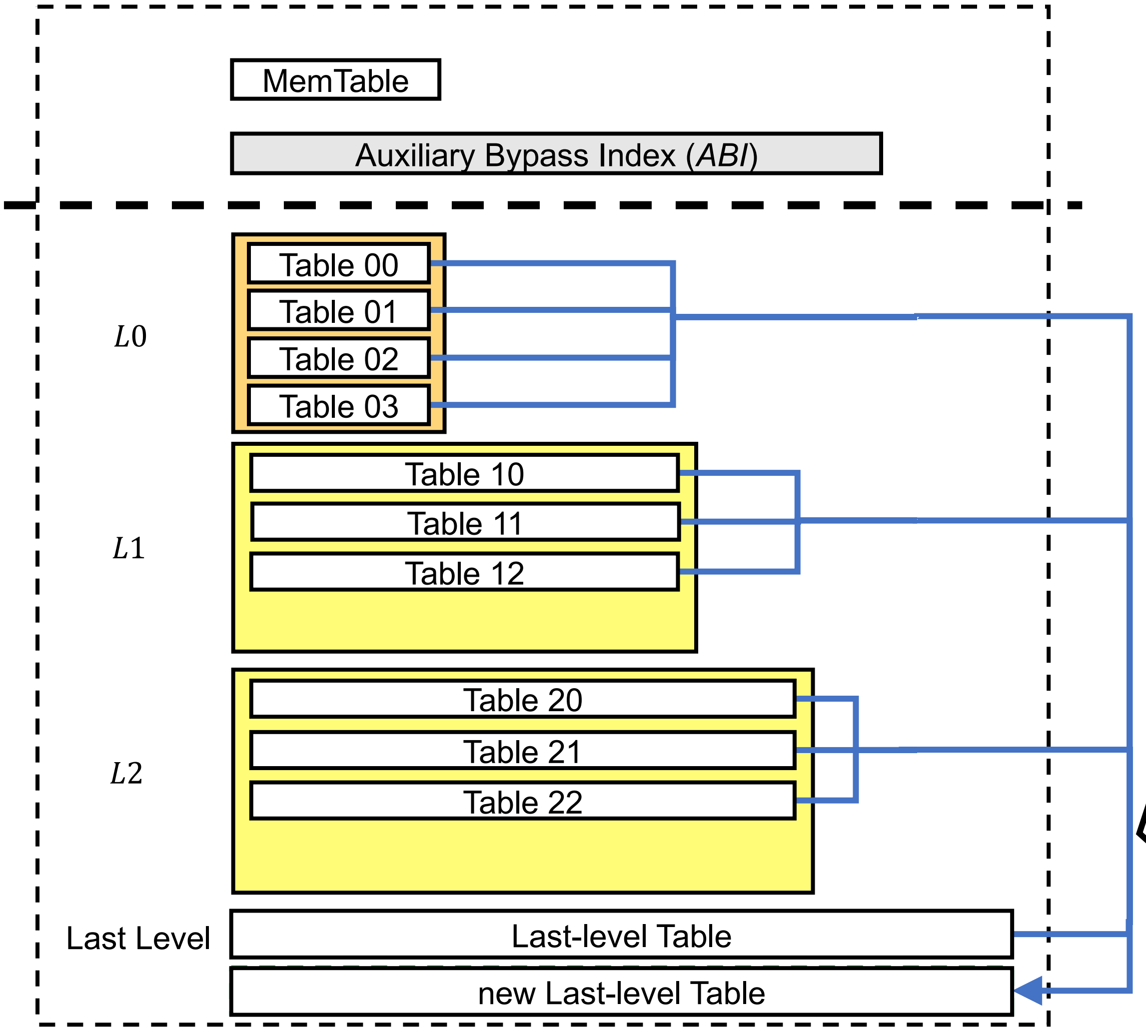

Another compaction will be triggered immediately as *L1* becomes full due to the last compaction. The Table 13 is merged right after it's been generated.

# Size-tiered Compaction in ChameleonDB




**Direct compaction** in ChameleonDB that allows a compaction involves multiple levels. *L0* and *L1* are merged together to a new table in *L2*. The overhead to generate and read Table 13 is avoided.

# Leveled Compaction in ChameleonDB

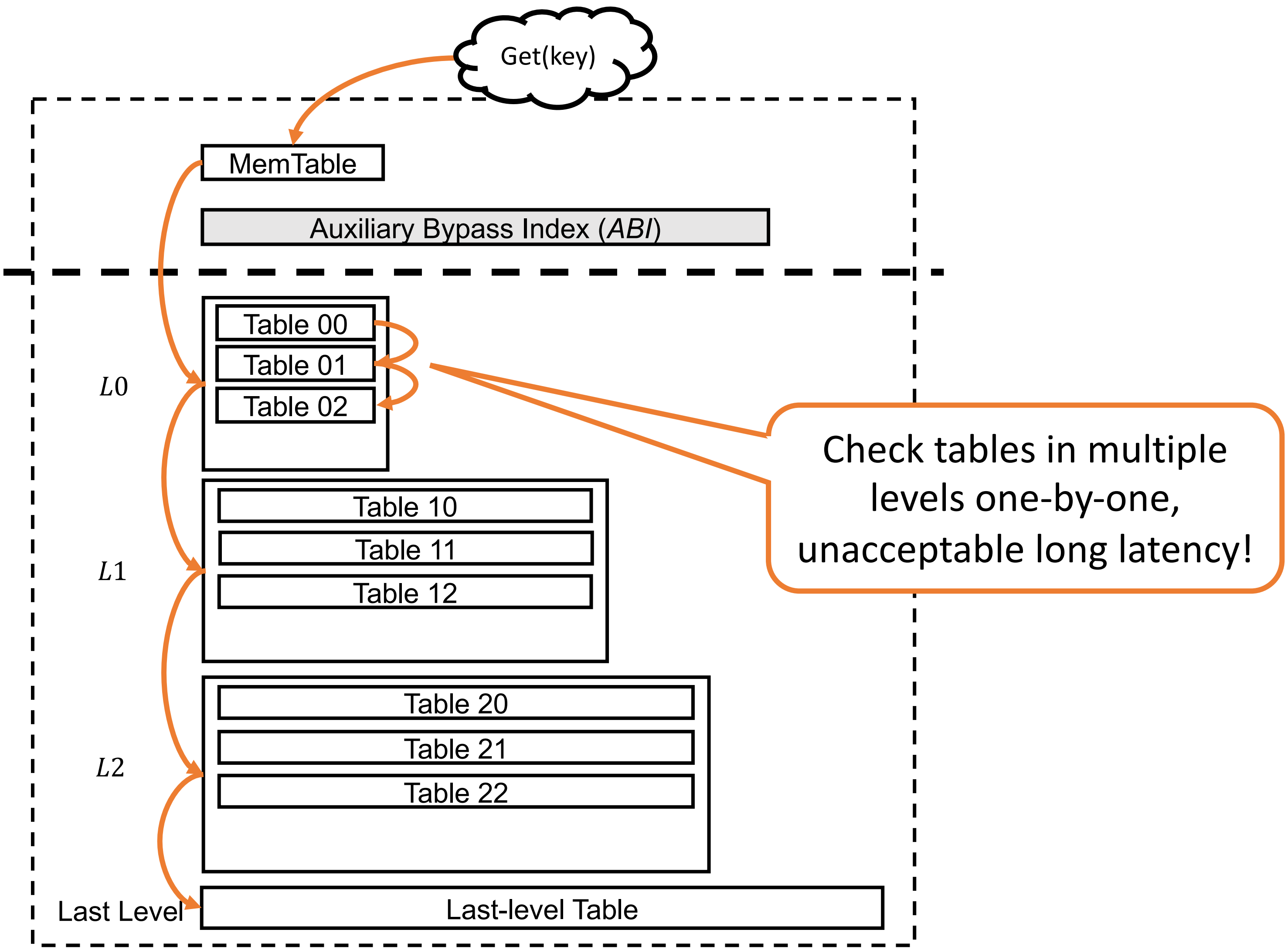



**Leveled compaction** is used for compactions to the last level, so as to maintain only one table in the last leve.

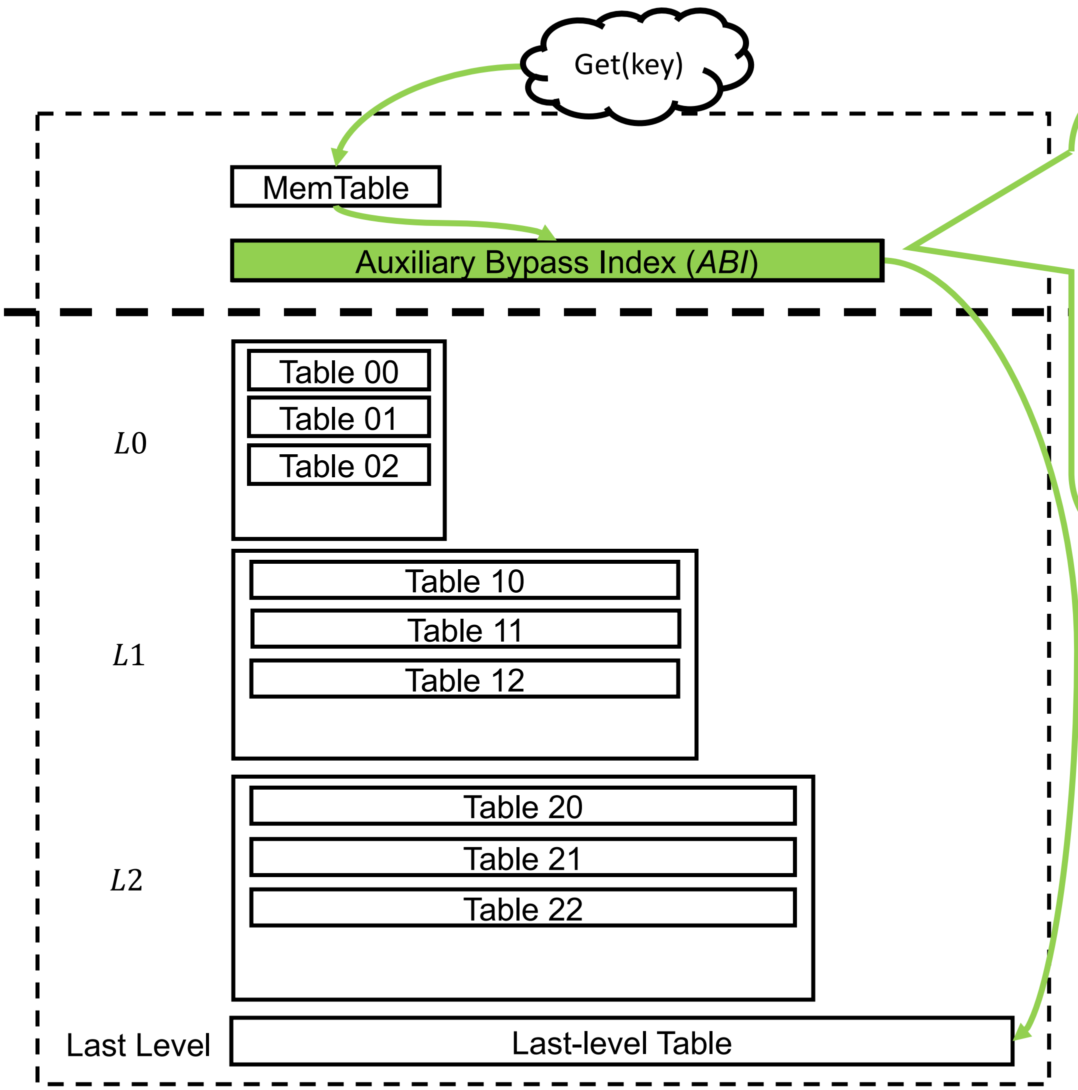
A last level compaction merges tables in all levels including the last level to a new last-level table.

After a last level compaction, all levels except the last level become empty.

# Search items without Auxiliary Bypass Index



# Search items with Auxiliary Bypass Index

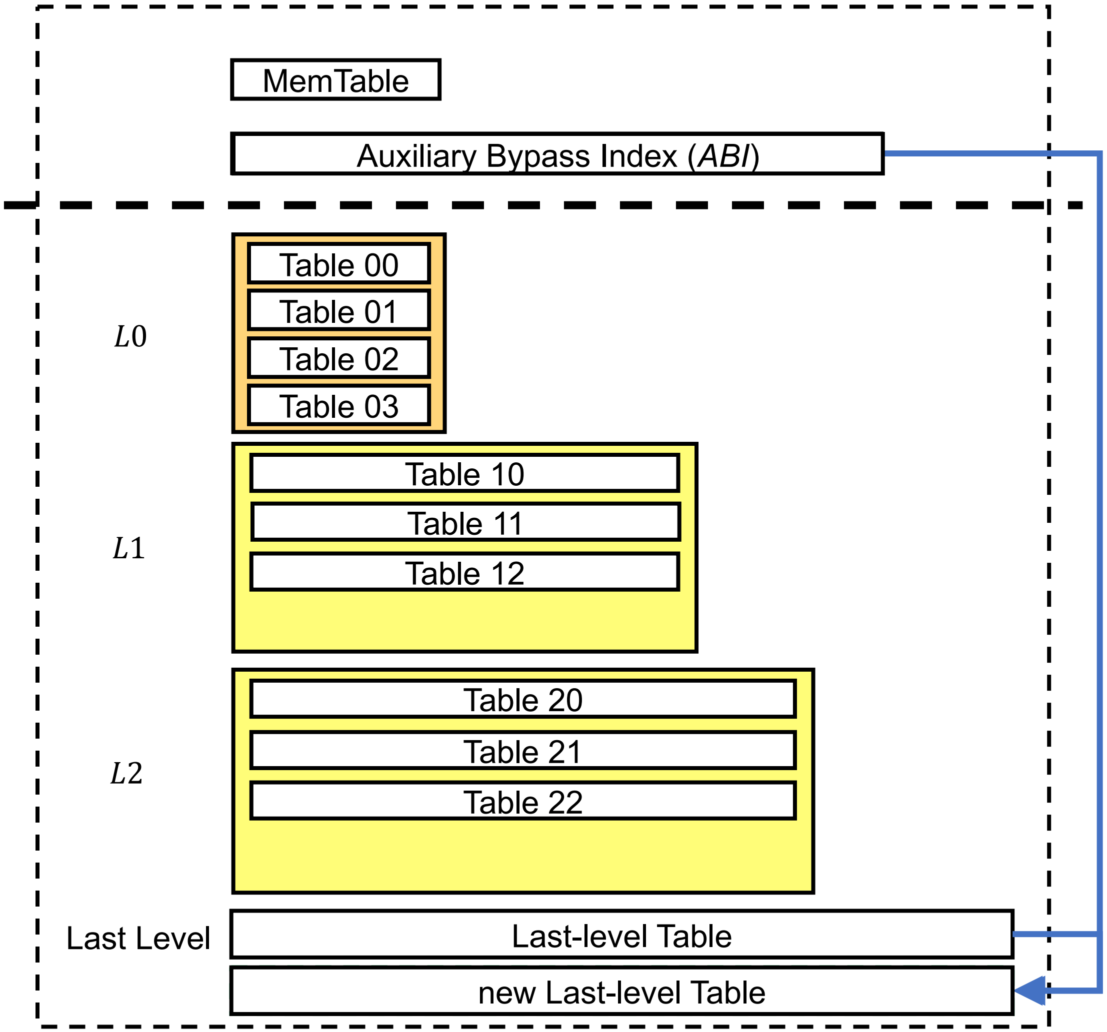



***ABI*** is an in-DRAM hash table that holds all items in the upper levels.

Everytime ChameleonDB flushes a MemTable, it also inserts the items in the MemTable to the *ABI*. *ABI* will be cleared when a last-level compaction happened.

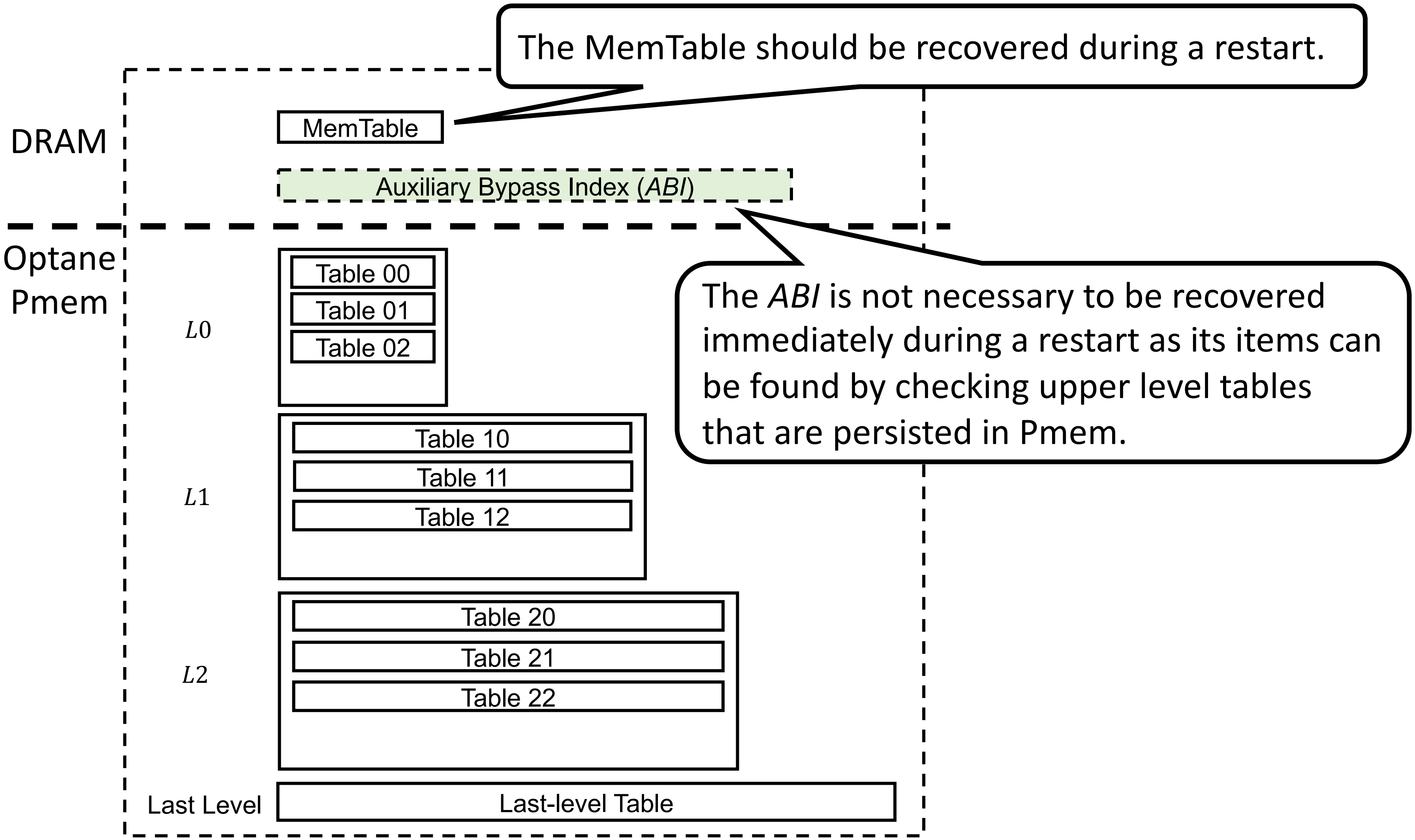
Use ***ABI*** to bypass checking upper levels one-by-one, and achieve consistently low read latency!

# Last level compacion with Auxiliary Bypass Index




Merge the *ABI* with the last-level table to generate a new last-level table. Avoid reading tables in upper levels.

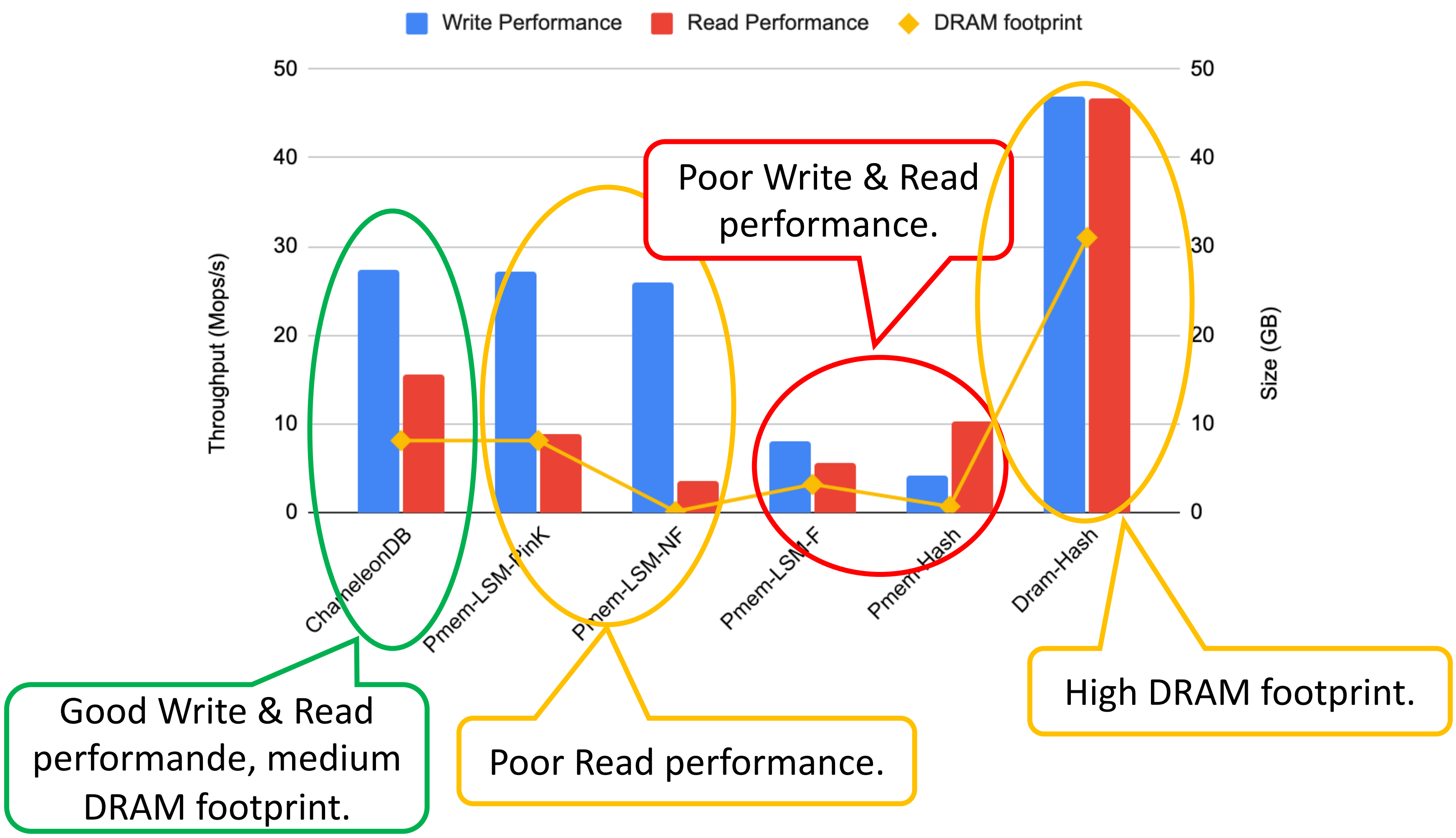
# Recovery during restart



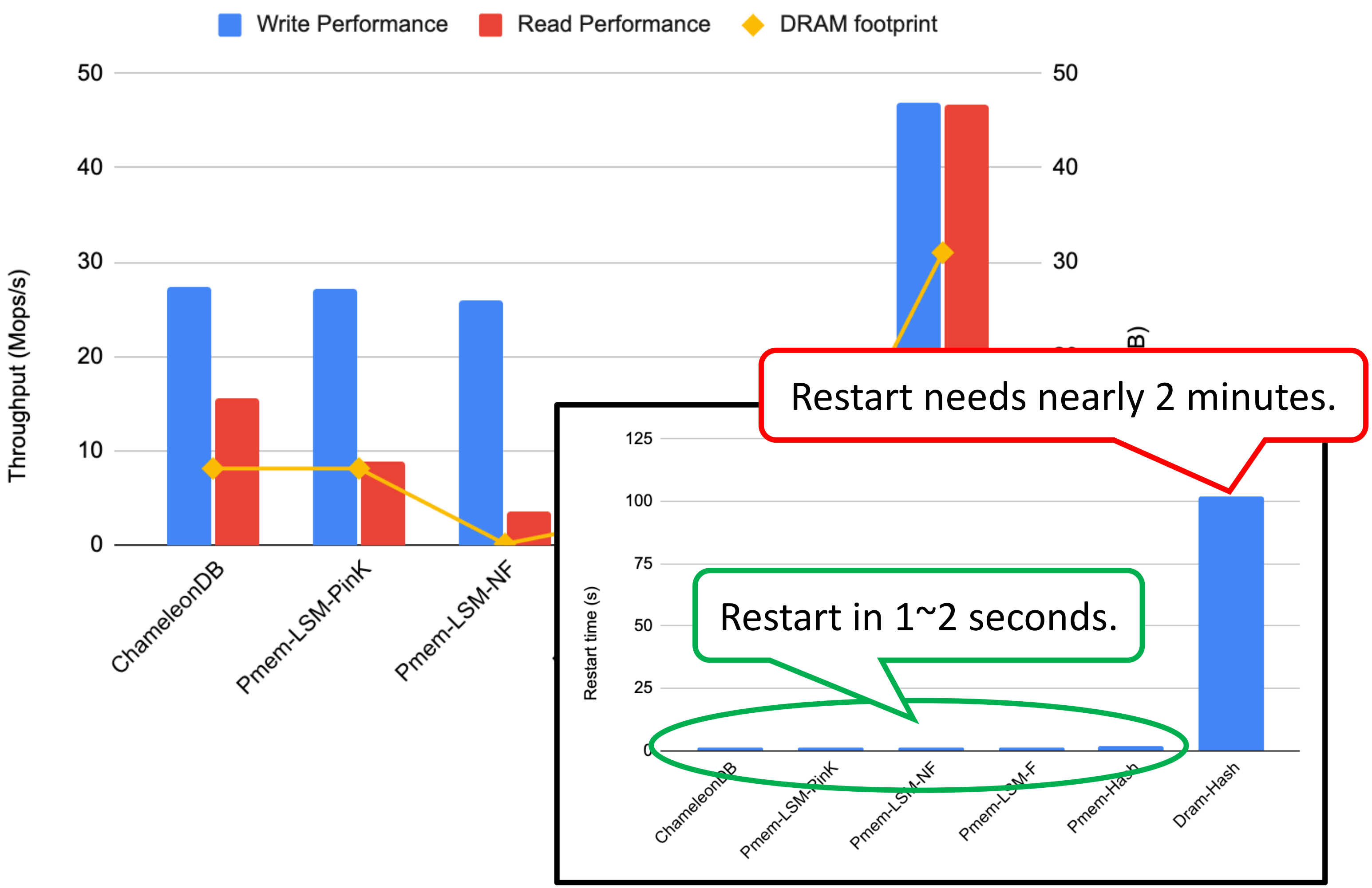
# How does ChameleonDB achieve design goals?

- Batch KV items before writing them to the storage log, batch index in MemTable before flushing them to Optane Pmem, use multi-level structure to organize index
  - ⇒ avoid small random writes to Optane Pmem
    - ⇒ **high write performance**
- Use Auxiliary Bypass Index to accelerate Get operation
  - ⇒ avoid checking multiple levels one-by-one
    - ⇒ **consistently low read latency**
- Place only a portion of the index in DRAM
  - ⇒ **small DRAM footprint**
- Recovers only MemTable during a restart
  - ⇒ **short restart time**

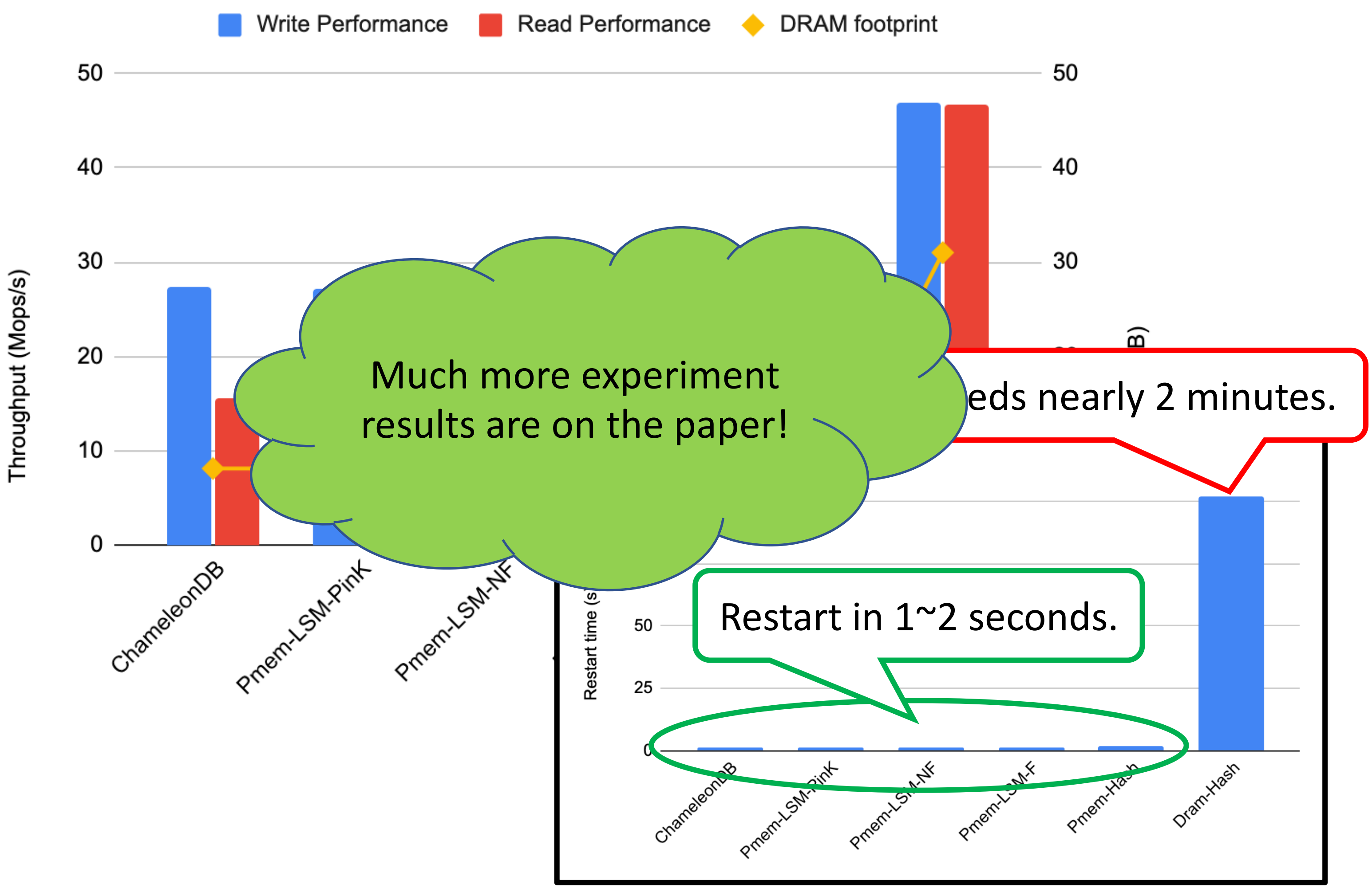
# Experiment results



# Experiment results



# Experiment results



# Thank You!

wenhui.zhang@uta.edu