ChameleonDB: a Key-value Store for Optane Persistent memory

Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang

University of Texas at 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. This property makes Optane Pmem different from what was assumed about persistent memory (cacheline as access unit). 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 **Optane Pmem**, whose latency is ~3x DRAM’s, the time to checking filters for multiple levels becomes **significant**.

![Graph showing Read Latency Breakdown (Optane Pmem)]

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?

Tree or Hash table as index in DRAM while items are batch written to storage.

Avoid checking multiple levels.

Avoid small random writes to storage.

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

- HIGH write performance
- LOW read latency
- SMALL DRAM footprint
- SHORT restart time
Structure of ChameleonDB

Key-value items are written to the storage log according to their arrival order.

Querying Key-value items in the storage log by the index.
Each shard is a multi-level structure. The last level contains only one hash table, while other levels contain multiple hash tables.

Multi-shard structure. Keys are hashed to a unique shard.

Index Items are gathered in MemTable before being flushed to Pmem as a table in $L_0$. 

Table 00
Table 01
Table 02
Table 03
Table 10
Table 11
Table 12
Table 20
Table 21
Table 22
Last-level Table
Size-tiered compaction is used to compact tables in levels except the last level. Tables in $L_0$ are merged to a new table in $L_1$. **Size-tiered** compaction is used to compact tables in levels except the last level. Tables in $L_0$ are merged to a new table in $L_1$. Merging the tables in $L_0$ into $L_1$ is done to reduce the size of the tables and optimize the database's performance. Further operations like inserting and updating data will be handled in the $L_0$ level.
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. $L_0$ and $L_1$ are merged together to a new table in $L_2$. The overhead to generate and read Table 13 is avoided.
Leveled compaction is used for compactions to the last level, so as to maintain only one table in the last level.

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

Get(key)

MemTable

Auxiliary Bypass Index (ABI)

Table 00
Table 01
Table 02

Table 10
Table 11
Table 12

Table 20
Table 21
Table 22

Last Level
Last-level Table

Check tables in multiple levels one-by-one, unacceptable long latency!
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

The MemTable should be recovered during a restart.

The ABI is not necessary to be recovered immediately during a restart as its items can be found by checking upper level tables that are persisted in Pmem.
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

Good Write & Read performance, medium DRAM footprint.

Poor Write & Read performance.

Poor Read performance.

High DRAM footprint.
Experiment results

- Restart in 1~2 seconds.
- Restart needs nearly 2 minutes.
Experiment results

Restart in 1~2 seconds.

Restart needs nearly 2 minutes.

Much more experiment results are on the paper!
Thank You!

wenhui.zhang@uta.edu