HashKV: Enabling Efficient Updates in KV Storage via Hashing

Helen H. W. Chan†, Yongkun Li‡, Patrick P. C. Lee†, Yinlong Xu‡

† The Chinese University of Hong Kong
‡ University of Science and Technology of China

USENIX ATC 2018
Update-intensive workloads are common in key-value (KV) stores
- Online transaction processing (OLTP)
- Enterprise servers
- Yahoo’s workloads are shifting from reads to writes [*]

Log-structured merge (LSM) tree
- Transform random writes into sequential writes
- Support efficient range scans
- Limitation: high read and write amplifications during compaction

[*] Sears et al., “bLSM: A General Purpose Log Structured Merge Tree”, SIGMOD 2012
LSM-tree in LevelDB

1. Read SSTables
2. Merge and sort by keys
3. Split into new SSTables

High I/O amplifications!
KV Separation[*]

- Store values separately to reduce LSM-tree size
  - LSM-tree: keys and metadata for indexing
  - vLog: circular log for KV pairs

KV Separation

Does KV separation solve all problems?

- High garbage collection (GC) overhead in vLog management
  - More severe if reserved space is limited
  - Update-intensive workloads aggravate GC overhead
  - GC needs to query the LSM-tree to check if KV pairs are valid

High write amplification of vLog if reserved space is filled
Our Contributions

- **HashKV**, an efficient KV store for update-intensive workloads
  - Extend KV separation with hash-based data grouping for value storage
  - Mitigate GC overhead with smaller I/O amplifications and without LSM-tree queries

- Three extensions that adapt to workload characteristics
  - E1: Dynamic reserved space allocation
  - E2: Hotness awareness
  - E3: Selective KV separation

- Extensive prototype experiments
  - 4.6x throughput and 53.4% less write traffic over circular log
Hash-based Data Grouping

- Hash values into fixed-size partitions by keys
  - **Partition isolation**: all value updates of the same key must go to the same partition in a log-structured manner
  - **Deterministic grouping**: instantly locate the partition of a given key

- Allow flexible and lightweight GC
  - Localize all updates of a key in the same partition

- **What if a partition is full?**

![Diagram showing LSM-tree for indexing and value store for KV pairs]
E1: Dynamic Reserved Space Allocation

Layout:
- Logical address space: main segments (e.g., 64 MiB)
- Reserved space: log segments (e.g., 1 MiB)
- Segment group: 1 main segment + multiple log segments

In-memory segment table tracks all segment groups
- Checkpointed for fault tolerance
Group-Based Garbage Collection

- Select a segment group for GC
  - e.g., the one with largest amount of writes
  - Likely to have many invalid KV pairs to reclaim free space

- Identify all valid KV pairs in selected group
  - Since each group stores updates in a log-structured manner, the latest version of each key must reside at the end of the group
  - No LSM-tree queries required

- Write all valid KV pairs to new segments

- Update LSM-tree
E2: Hotness Awareness

- Problem: mix of hot and cold KV pairs
  - Unnecessary rewrites for cold KV pairs

- Tagging:
  - Add a tag in metadata to indicate presence of cold values
  - Cold values are separately stored
    - Hot-cold value separation

- GC rewrites small tags instead of values
E3: Selective KV Separation

- KV separation for small values incurs extra costs to access both LSM-tree and value store

- Selective approach:
  - Large values: KV separation
  - Small values: stored entirely in LSM-tree

- Open issue: how to distinguish between small and large values?
Other Issues

- Range scans:
  - Leverage read-ahead (via posix_fadvise) for speedup

- Metadata journaling:
  - Crash consistency for both write and GC operations

- Implementation:
  - Multi-threading for writes and GC
  - Batched writes for KV pairs in the same segment group
  - Built on SSDs
Putting It All Together: HashKV Architecture

- **Write cache**: (meta, key)
  - **KV separation**: (meta, key, value)

- **MemTable**
  - **Reserved space**
    - **LSM-tree**

- **Value store**
  - **Main segment**
  - **Log segment**
    - **Segment group**
    - **Segment group**
  - **Cold data log**

- **Group 1**: (end pos, segments)
- **Group 2**: (end pos, segments)
  - **Segment table**

- **Memory**
  - **Write journal**
  - **GC journal**

- **Persistent Storage**
Experiments

- Testbed backed with an SSD RAID array
- KV stores
  - LevelDB, RocksDB, HyperLevelDB, PebblesDB (default parameters)
  - vLog (circular log) and HashKV: built on LevelDB for KV separation
- Workloads
  - 40 GiB for main segments + 12 GiB (30%) reserved space for log segments
  - **Load**: 40 GiB of 1-KiB KV pairs (Phase P0)
  - **Update**: 40 GiB of updates for three phases (Phases P1, P2, P3)
    - P1: reserved space gradually filled up
    - P2 & P3: reserved space fully filled (stabilized performance)
Update Performance of HashKV

- Compared to LevelDB, RocksDB, and vLog:
  - 6.3-7.9x, 1.3-1.4x, and 3.7-4.6x throughput, resp.
  - 49.6-71.5% lower write size

- Much lower KV store size than HyperLevelDB and PebblesDB
Impact of Reserved Space

- HashKV’s throughput increases with reserved space size
- vLog has high LSM-tree query overhead (80% of latency)
HashKV maintains high range scan performance
Optimization Features

- Higher throughput and smaller write size with optimization features enabled

- Hotness awareness

- Selective KV separation
Conclusions

- HashKV: hash-based data grouping for efficient updates
  - Dynamic reserved space allocation
  - Hotness awareness via tagging
  - Selective KV separation

- More evaluation results and analysis in paper and technical report

- Source code: [http://adslab.cse.cuhk.edu.hk/software/hashkv](http://adslab.cse.cuhk.edu.hk/software/hashkv)