HashKV: Enabling Efficient Updates in KV Storage via Hashing

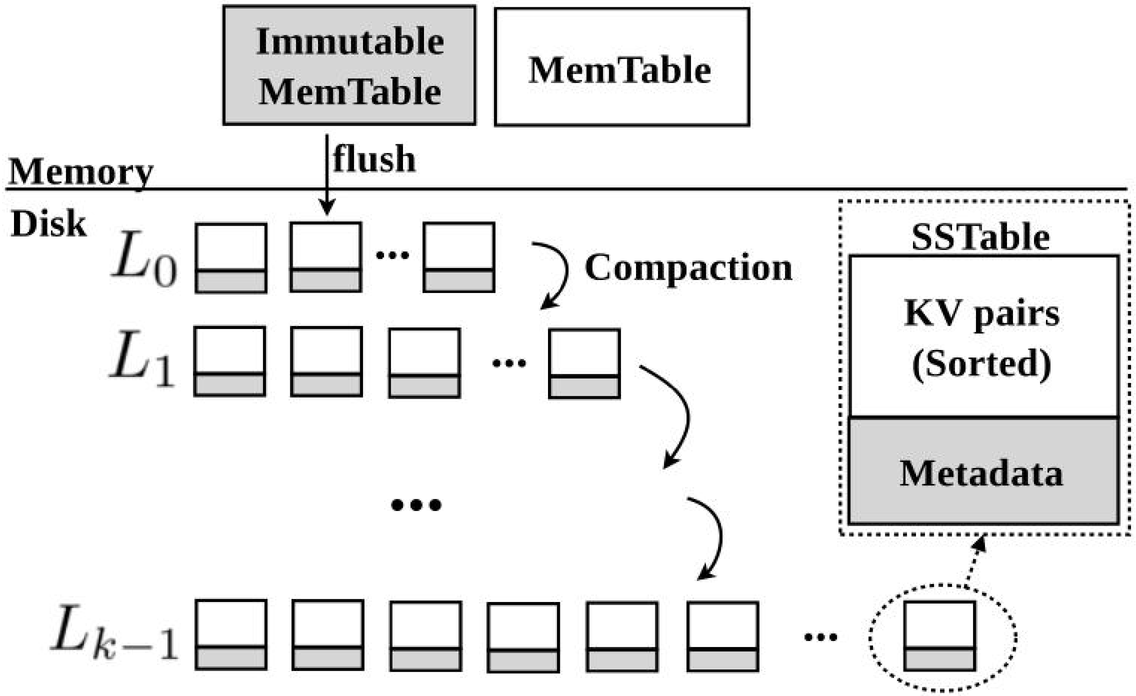
Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee Yinlong Xu,

slide made by wgl

Background

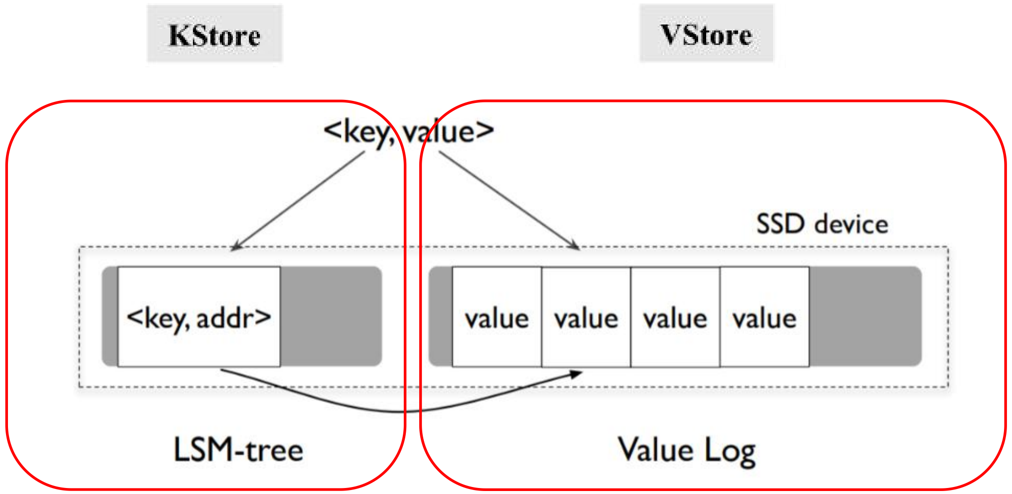
- **LSM-Tree**
  - Log Structured Merge Tree

- **Problem**
  - Read amplification
  - Write amplification
  - Garbage Collection



Background

WiscKey (FAST'16)

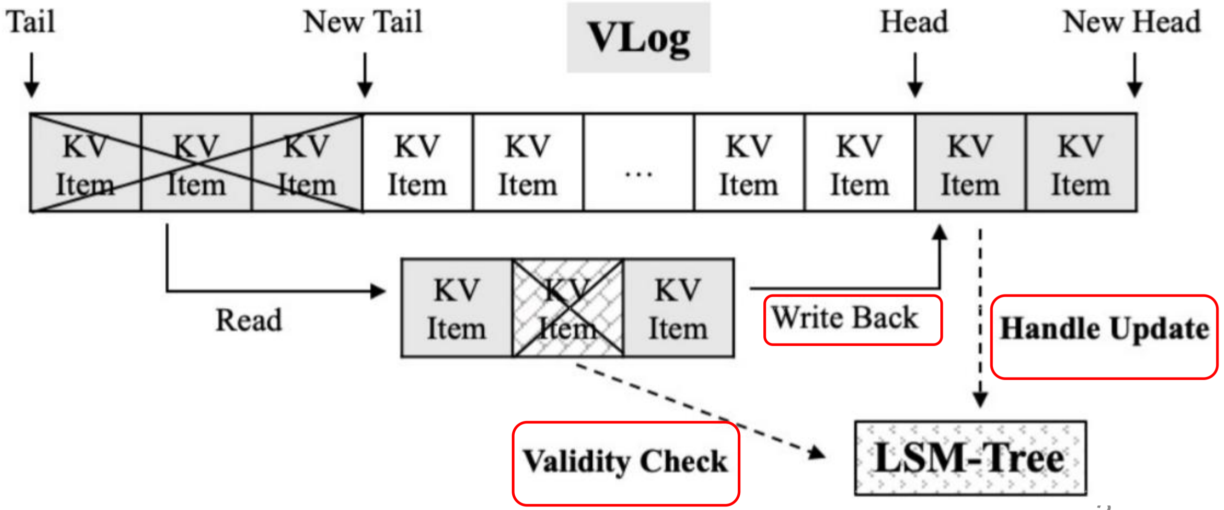


➢LSM-tree Optimizations

- Key-Value separation
- KStore->Key + Value Handle
- VStore->KVItem

➢Problem

- Garbage Collection
- query and insert overheads

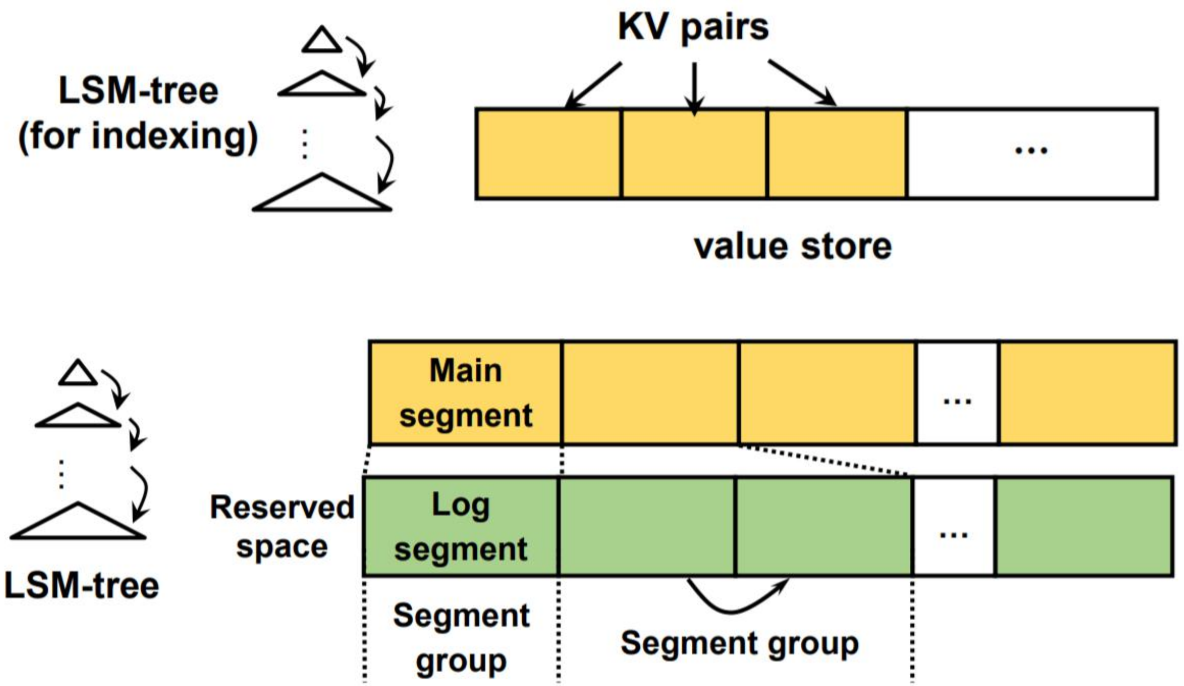


Motivation

- **Challenge**
  - Address the mixture of hot and cold data
  - hot-cold data grouping
- **Focus**
  - **Reduce write amplification**
  - **Reduce query overhead**

Design

➢ Storage Management



➢ Hash-based Data Grouping

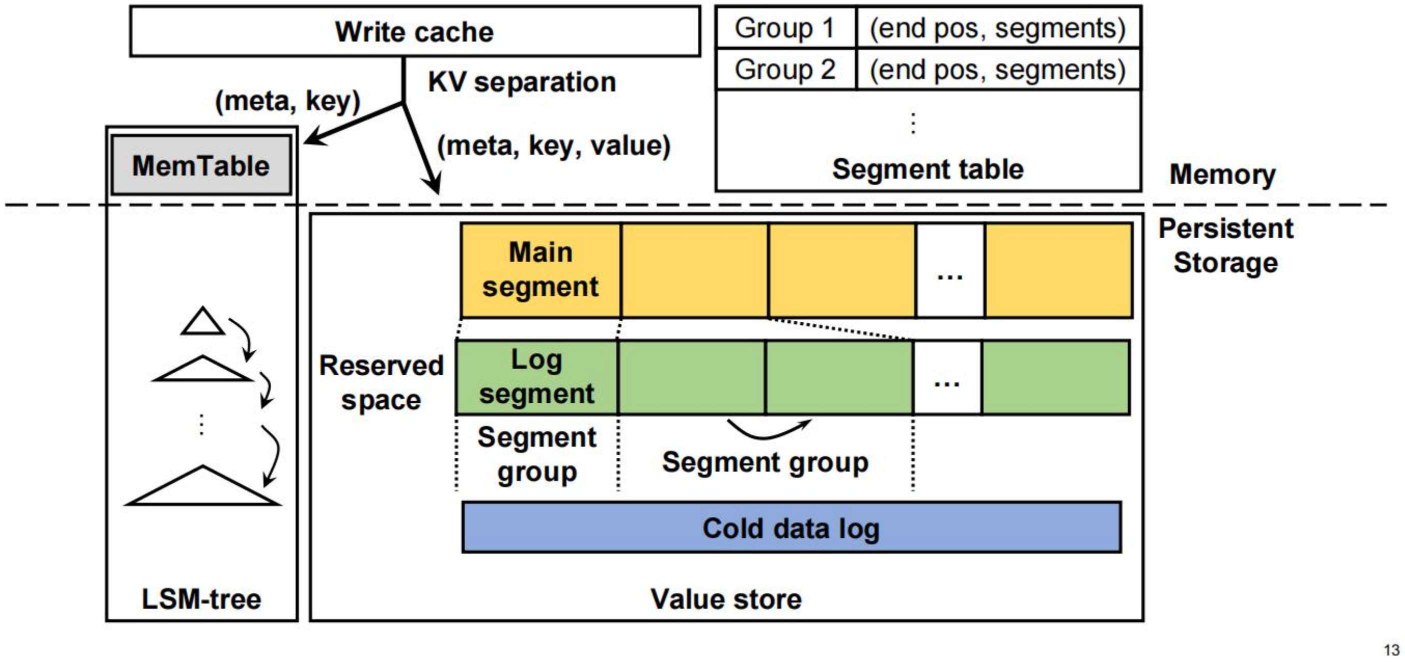
- Partition isolation: hash(Key) -> Partition
- Deterministic grouping

➢ Dynamic reserved space

- Main segments 64 MiB
- Log segments 1 MiB
- Segment group:main segment + multiple log segments

Architecture

➢HashKV



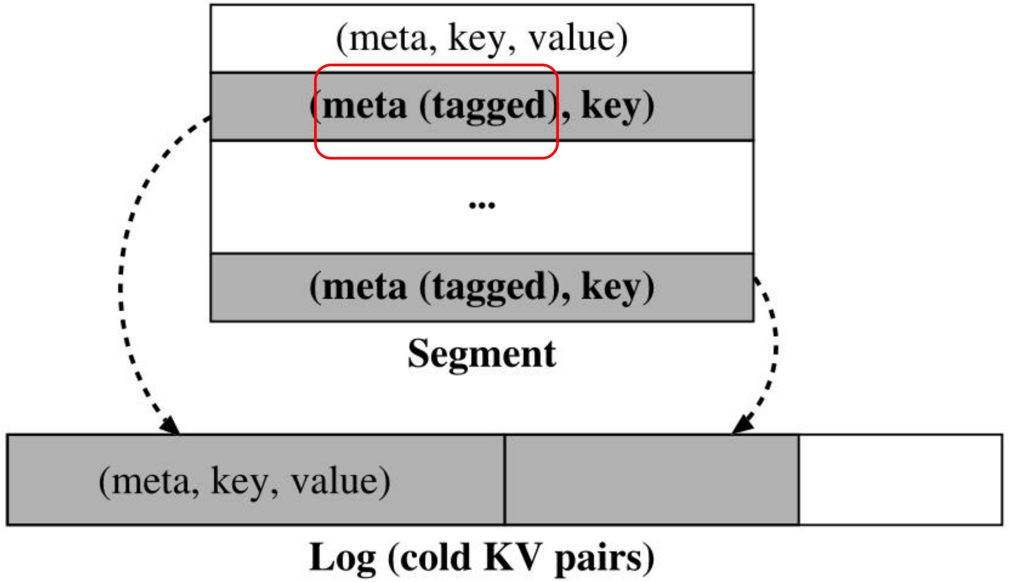
- ➢LSM-tree
  - key + meta
- ➢Reserved space
  - hash(key) -> key + meta +value
- ➢Segment table
  - hash(key) -> segment end pos
- ➢Write cache
  - 64 MiB.

Design

➢ **Group-Based Garbage Collection**
  - Select the segment group with the largest amount of writes
  - Using hash table (No LSM-tree queries required)
  - Write all valid KV pairs to new segments
  - Update LSM-tree

Design

➢ Hotness Awareness



- ➢ Unnecessary rewrites for cold KV pairs
- ➢ Tagging for Hot-cold value separation
  - Tag to indicate presence of cold values
  - Cold values are separately stored
- ➢ Update cold KV pairs
  - cold -> hot
  - GC rewrites small tags instead of values

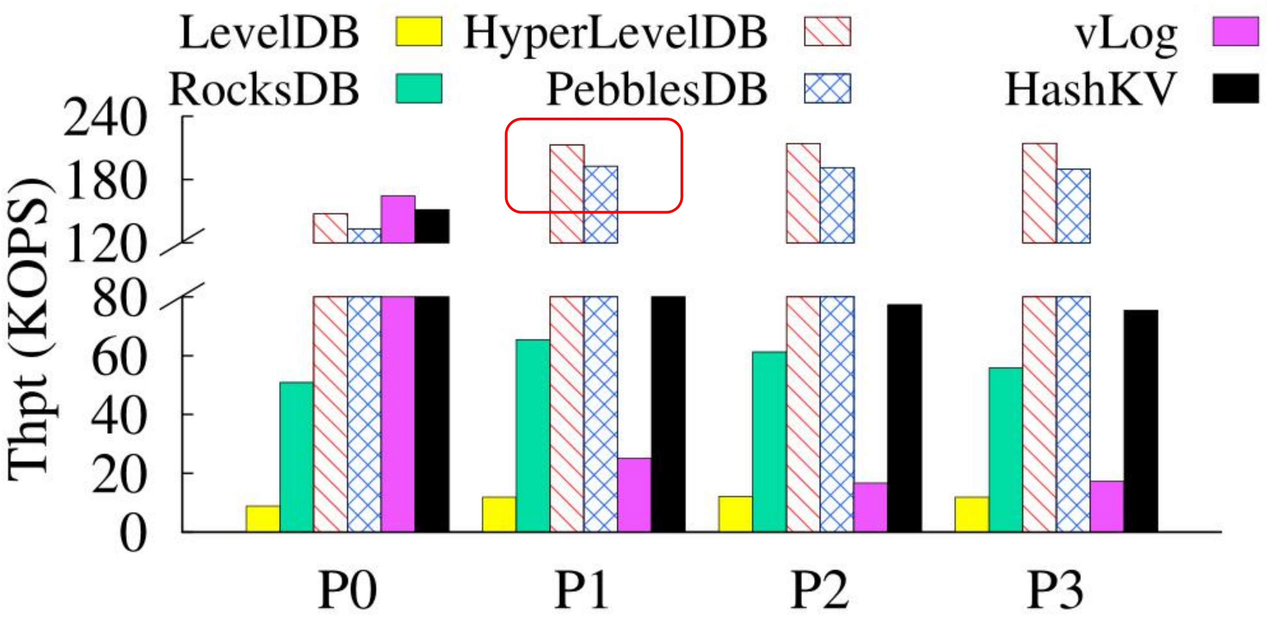
Design

- **Selective KV Separation**
  - Limited benefits for small-size KV pairs
- **Selective approach:**
  - Large values: KV separation
  - Small values: stored entirely in LSM-tree
- **How to distinguish small / large values ?**

Evaluation

24-B key / 992-B value
P0: 40GiB of KV pairs
P1~3: 40GiB of updates

- **Update Performance**

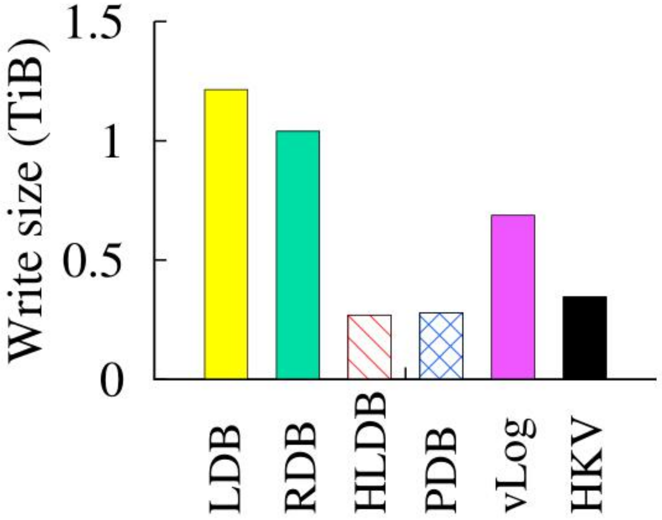


- 6.3-7.9x , 1.3-1.4x, and 3.7-4.6x throughput compared to LevelDB, RocksDB, and vLog.

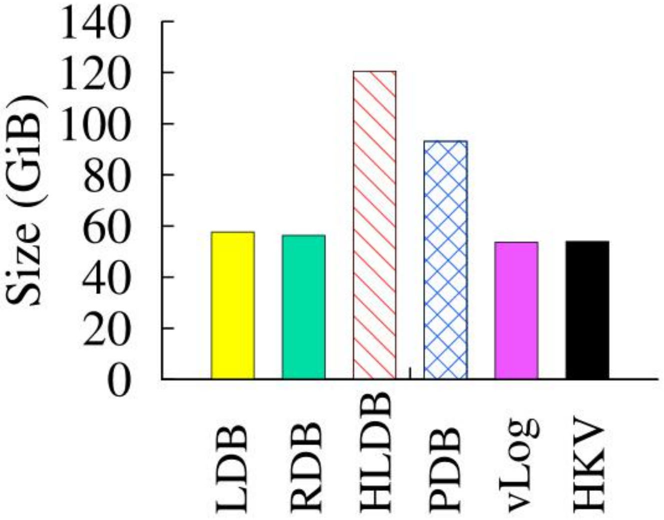
Evaluation

24-B key / 992-B value
P0: 40GiB of KV pairs
P1~3: 40GiB of updates

➢ Update Performance



(b) Total write size



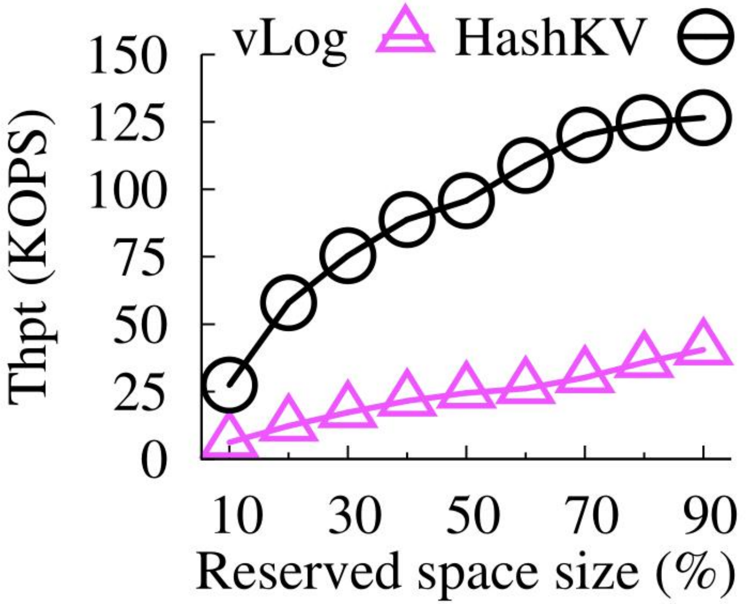
(c) KV store size

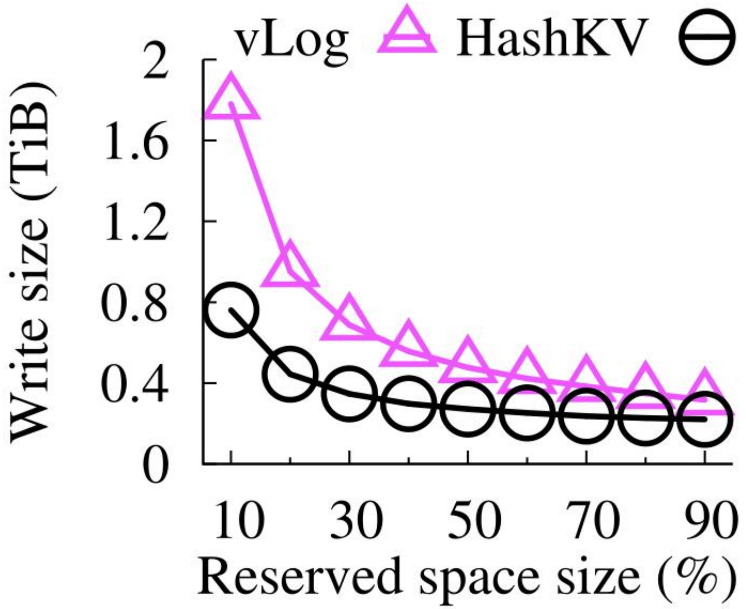
- Low write size due to Hotness Awareness
- Much lower KV store size than HyperLevelDB and PebblesDB

Evaluation

10% to 90% of 40 GiB

➢ **Impact of Reserved Space**





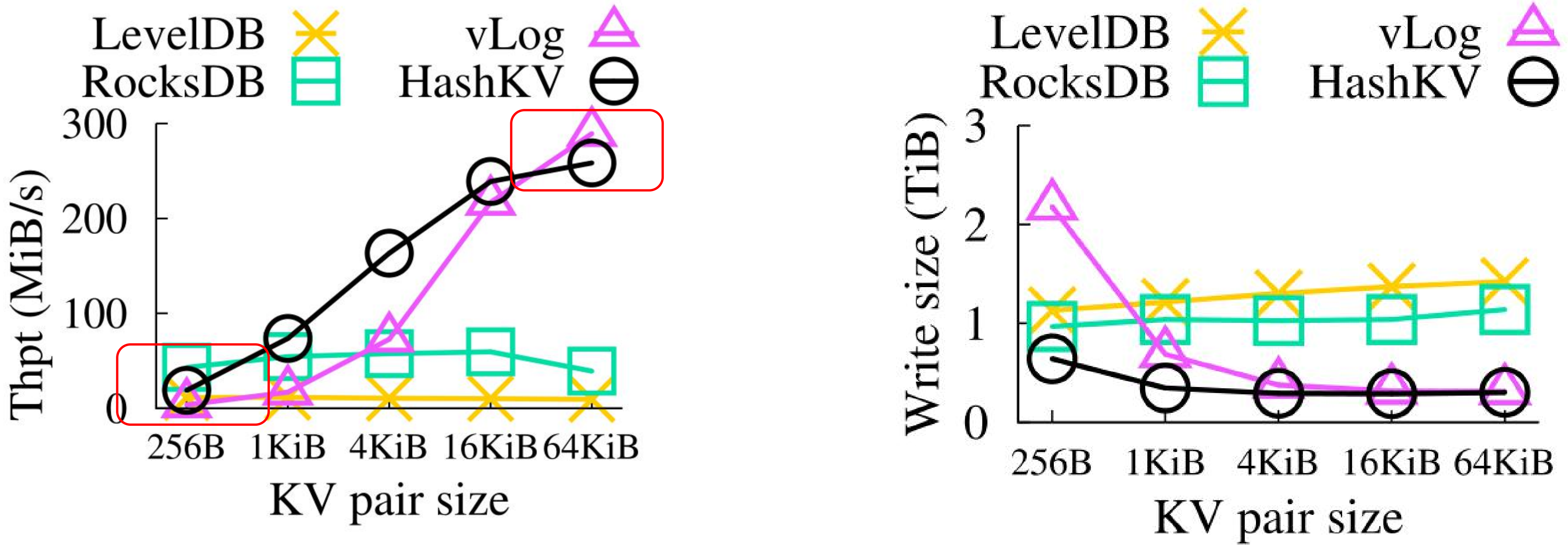
➢ 3.1 - 4.7× throughput of vLog

➢ reduces the write size of vLog

Evaluation

P3 Update

- **Different KV Size**



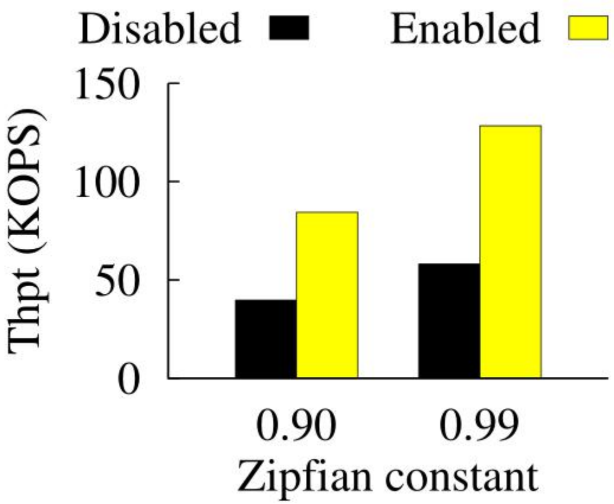
- HashKV achieves better throughput as pair size increases

- Reduce the total write sizes
- Fewer KV pairs

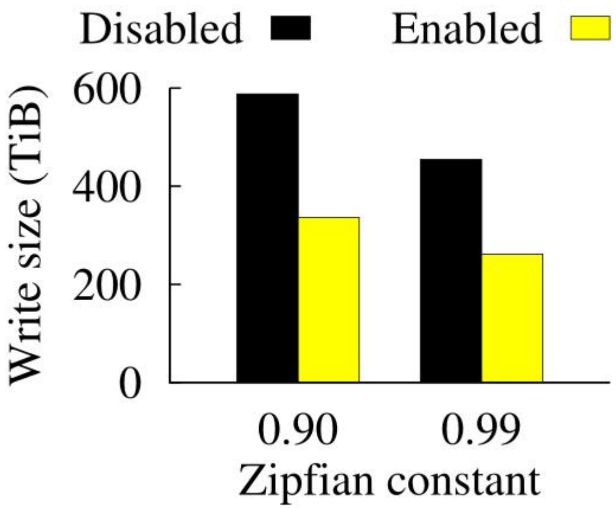
Evaluation

P3 Update

- **Hotness awareness**



(a) Throughput

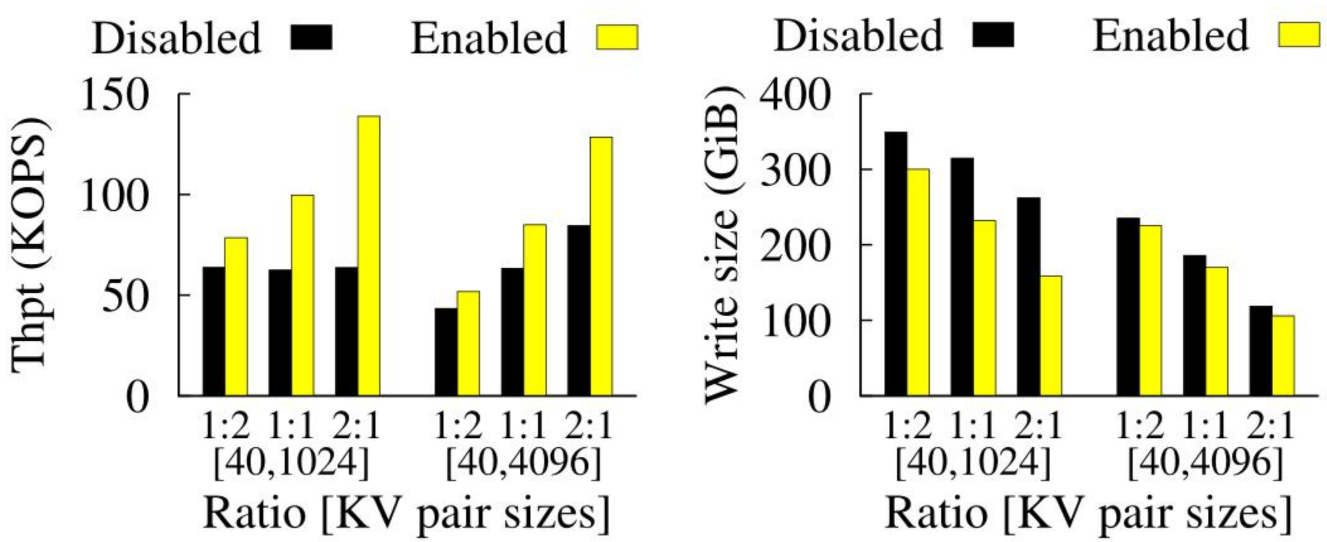


(b) Total write size

- Update throughput increases by 113.1% and 121.3%
- Write size reduces by 42.8% and 42.5%

Evaluation

- **Selective KV separation**



(a) Throughput (b) Total write size

- Throughput increases by 23.2-118.0% and 19.2-52.1%

- Reduce the total write size by 14.1-39.6% and 4.1-10.7%

Conclusion

➢ **HashKV**

- Hash-based Data Grouping
- Group-Based Garbage Collection
- Hotness Awareness
- Selective KV Separation

➢ **Evaluation**

- Elimination of mixture of hot and cold data
- No need for LSM-tree query due to hash based grouping
- Better throughput & Lower write size

Append

➢Hash-based Grouping