NovKV: Efficient Garbage Collection for Key-Value Separated LSM-Stores

Chen Shen, Youyou Lu, Fei Li
Weidong Liu, Jiwu Shu

Tsinghua University

Outline

- Background
- Motivation
- Design
- Evaluation
- Summary

Background

KV-Stores are everywhere

- LevelDB, RocksDB, Ceph, HBase, TiKV...

LEVELDB

RocksDB

The most popular index structure in KV-Stores

- Buffers incoming writes in memory & flushes to disk sequentially
- Almost has no random write
- Makes good use of the disk bandwidth
- Maximizes write performance

ceph

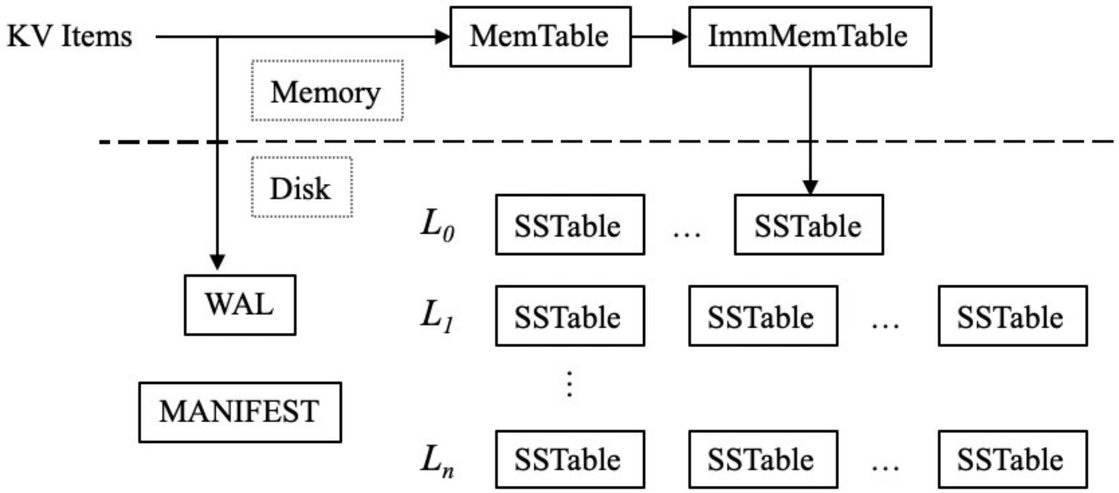
APACHE HBASE

TiKV

Background

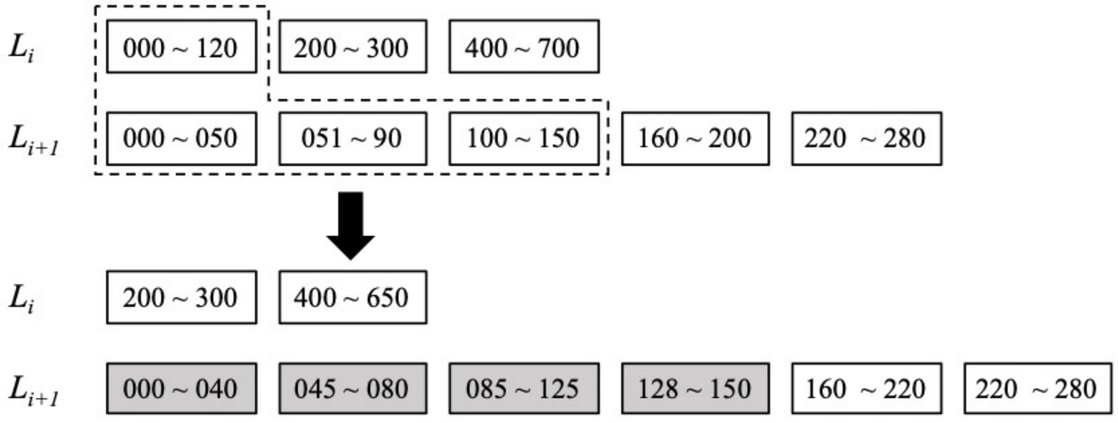
The LSM-Tree basics

- Write Path
  - Writes the WAL → Inserts to the MemTable
- Read Path
  - MemTable → ImmMemTable → SSTables



The LSM-Tree compaction

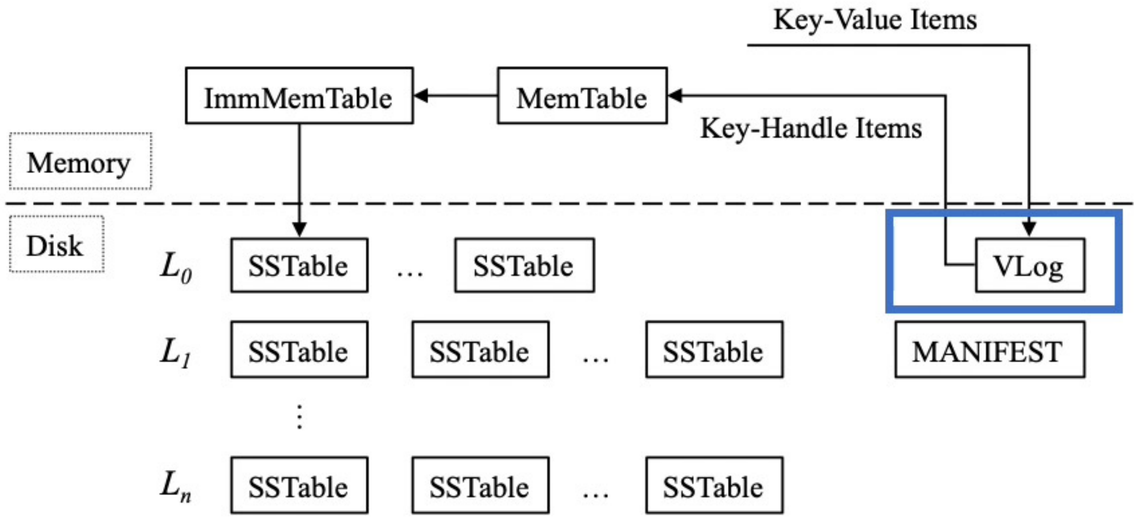
- Key ranges of SSTables maybe overlapped
- Introduces severe overheads in reads
- Key-overlapped → non-key-overlapped
- Write amplification, resource contention...



Background

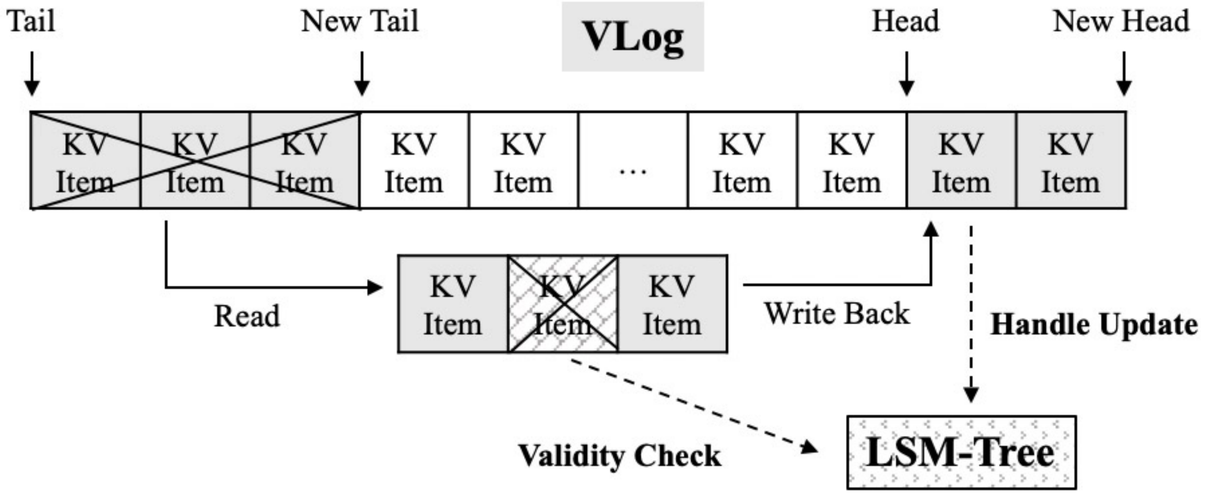
The key–value separated LSM–Tree

- VStore (VLog): Actual KV items
- KStore (LSM–Tree): Keys and value handles
- A garbage collection process



The garbage collection

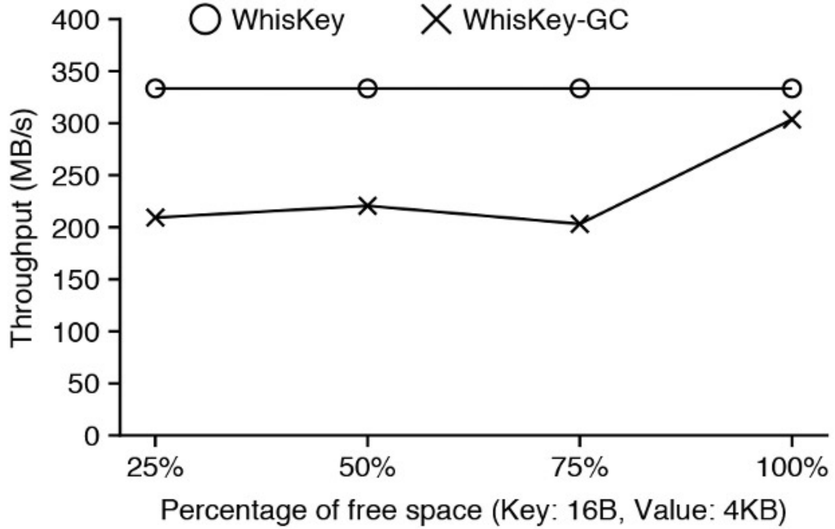
- Reads a chunk of KV items from the VLog
- Finds valid values (Validity Check)
- Writes back valid values and updates new value handles (Handle Update)



Motivation

What's the problem

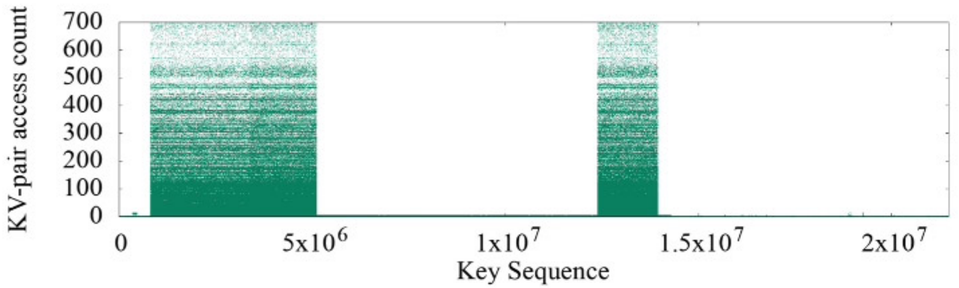
- The garbage collection introduces another query and insert overheads



* WiscKey: Separating Keys from Values in SSD-conscious Storage

Observations

- The compaction and the garbage collection run independently
- Keys in the KStore are only dropped during compaction, termed DropKeys
- Only a part of KV items will be read
- Many handle updates are unnecessary



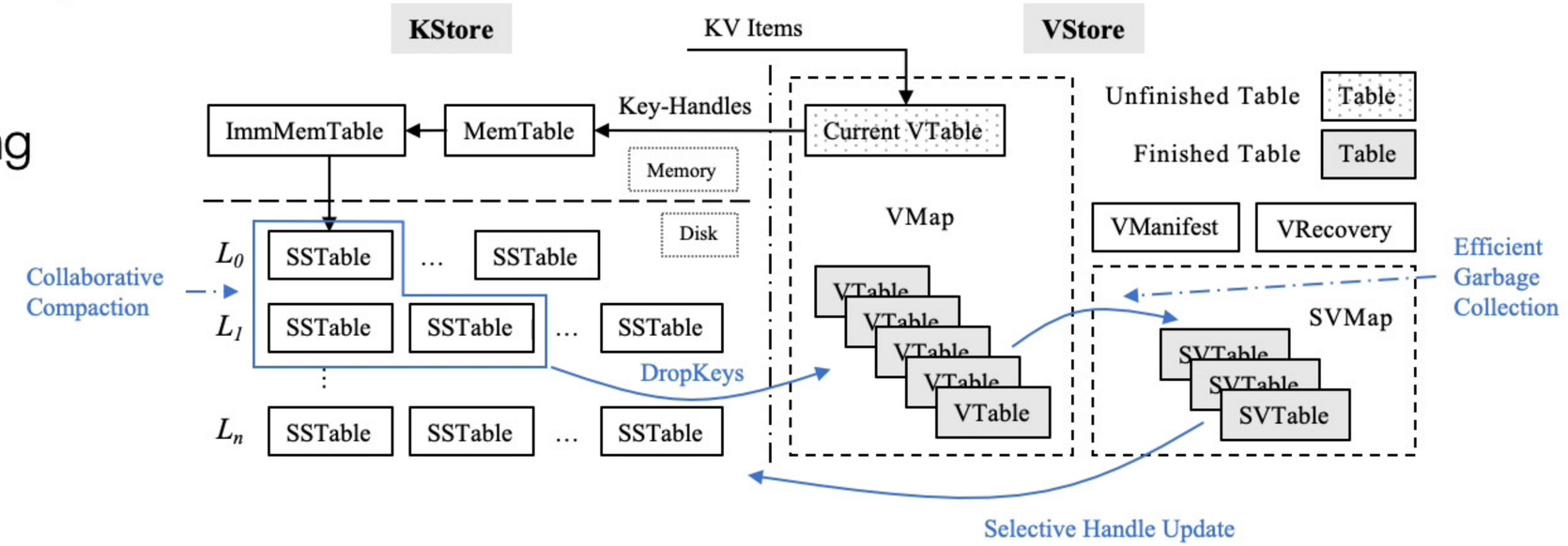
Heat-map of KV-pairs accessed by Get in ZippyDB

* Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook

Design – Architecture

Key techniques

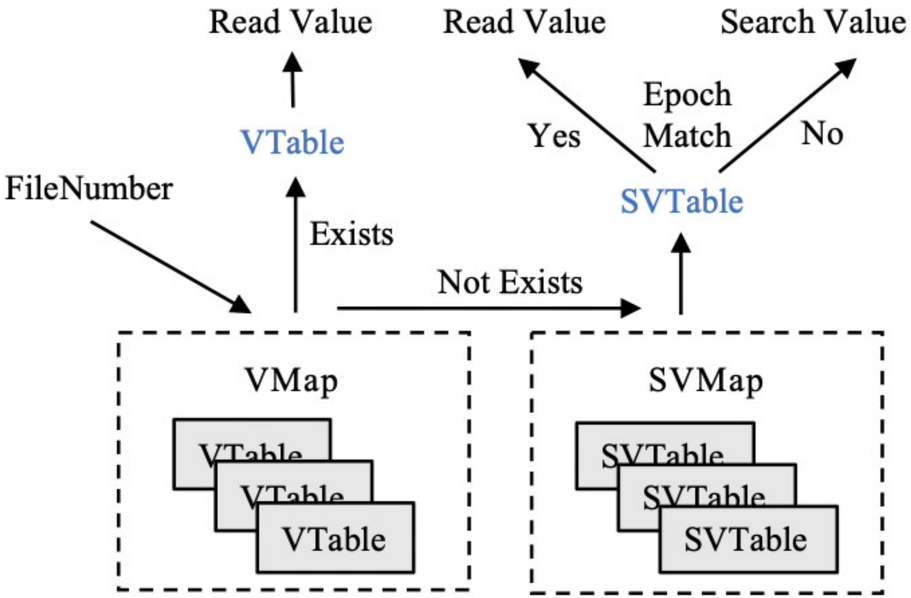
- Collaborative Compaction
  - Collects DropKeys and saves them to VStore
- Efficient Garbage Collection
  - Checks validity of KV items without querying the KStore
  - Avoids value handle updating
- Selective Handle Updating
  - Delays the handle updating



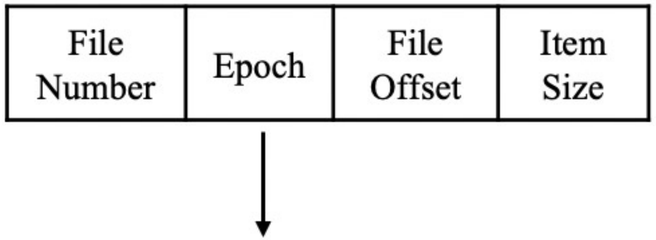
Design – Architecture

Read Path

- Takes the table from VMap, and reads the value
- If not exists, takes the table from SVMap
- Checks whether the table's epoch matches the handle's epoch
- If matches, reads the value
- Otherwise, searches the value by the key



Value Handle

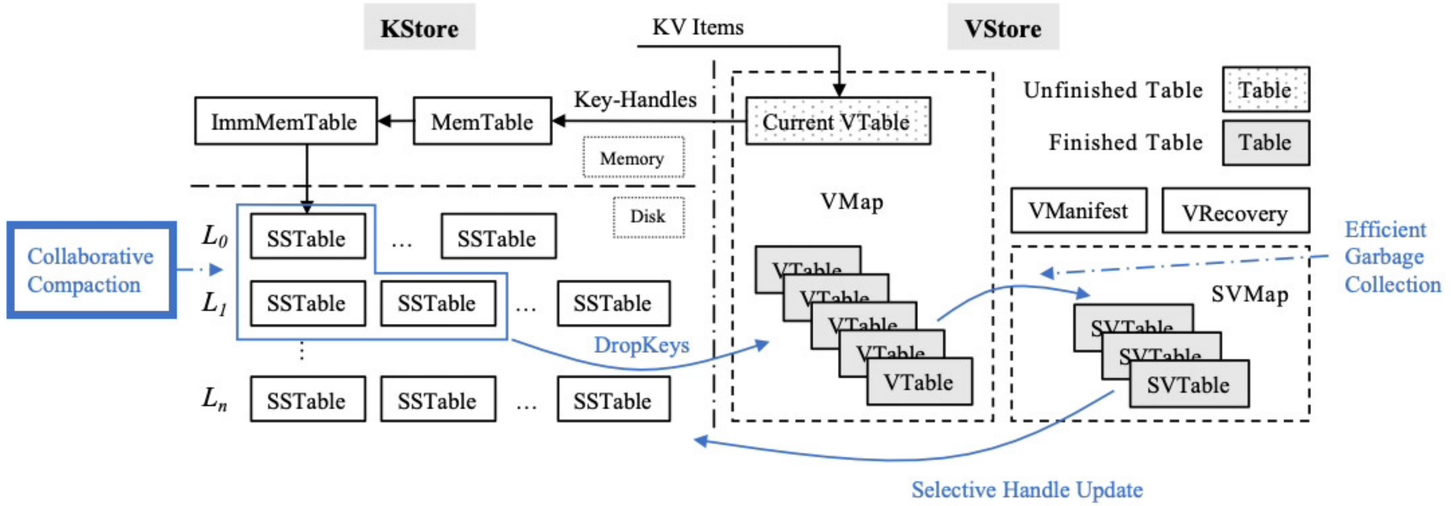


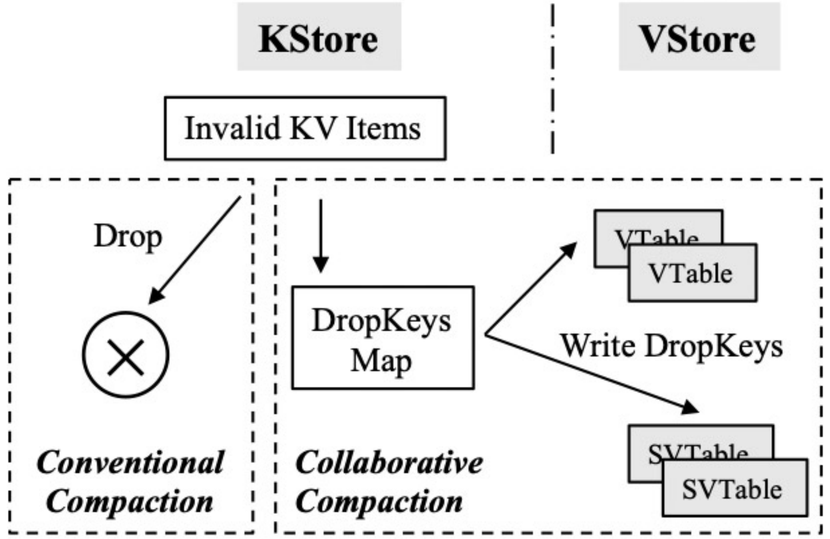
the number of times a table has been rewritten

Design – Collaborative Compaction

Differences between conventional & collaborative compaction

- Conventional compaction discards invalid KV items directly
- Collaborative compaction collects these items' keys and writes them to corresponding tables





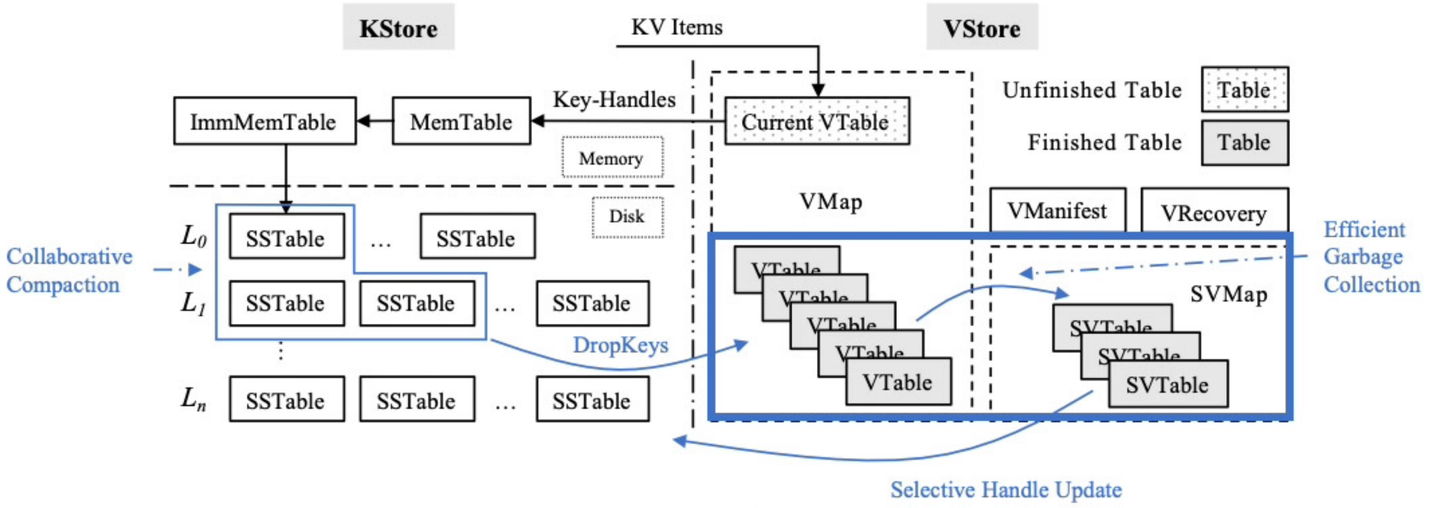
Design – VTable & SVTable

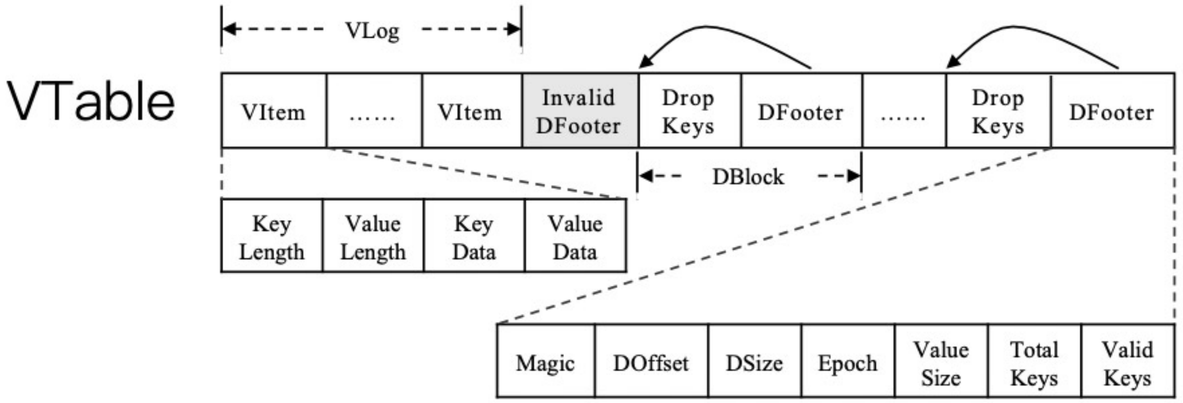
How to store DropKeys efficiently?

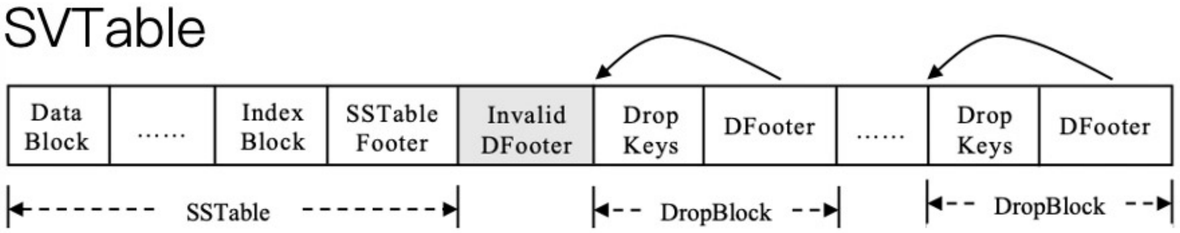
- KStore stores different DropKeys multiple times
- Recods DropKeys and table metadata as a block each time

- DOffset and DSize: the offset and the size of this DropBlock
- Epoch: how many times this table has been rewritten.
- Total and valid keys: the numbers of total keys and currently valid keys

- The last DFooter is kept in memory



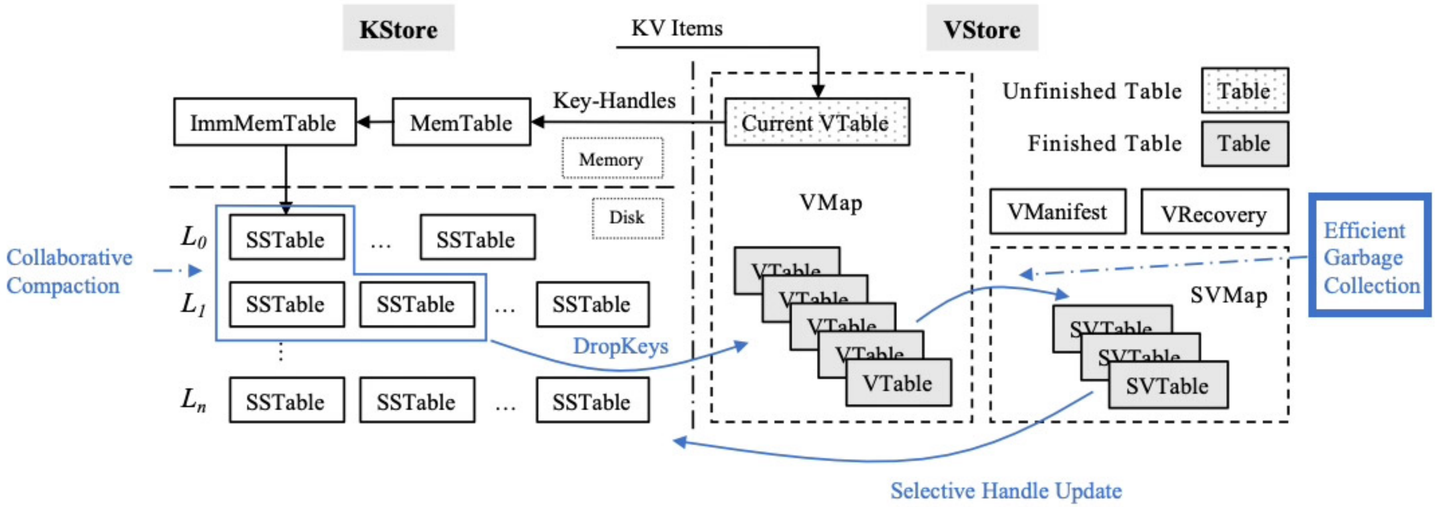


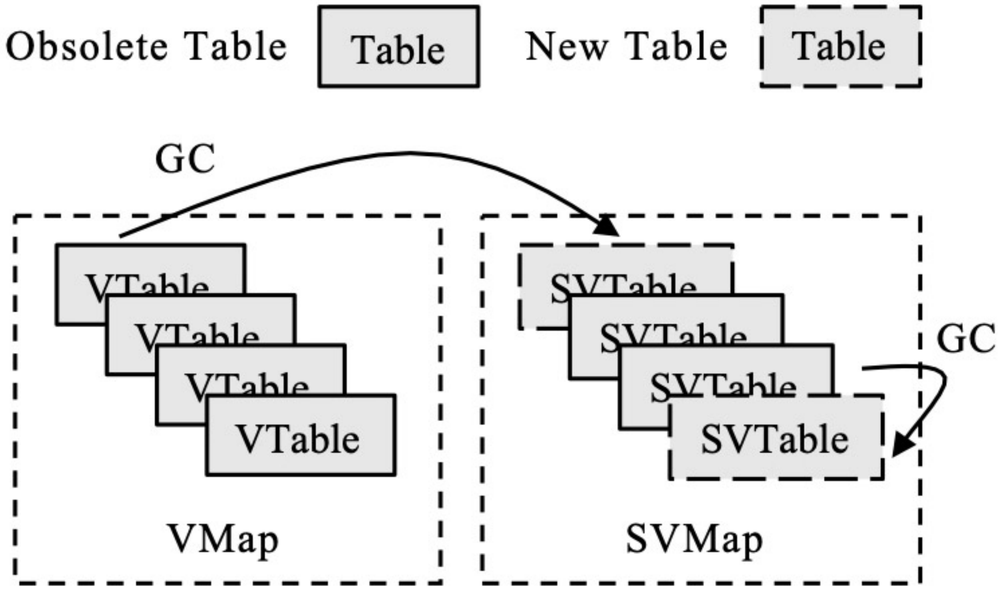


Design – Efficient Garbage Collection

High–level perspective

- Candidate VTables or SVTables are garbage collected and built as new SVTables

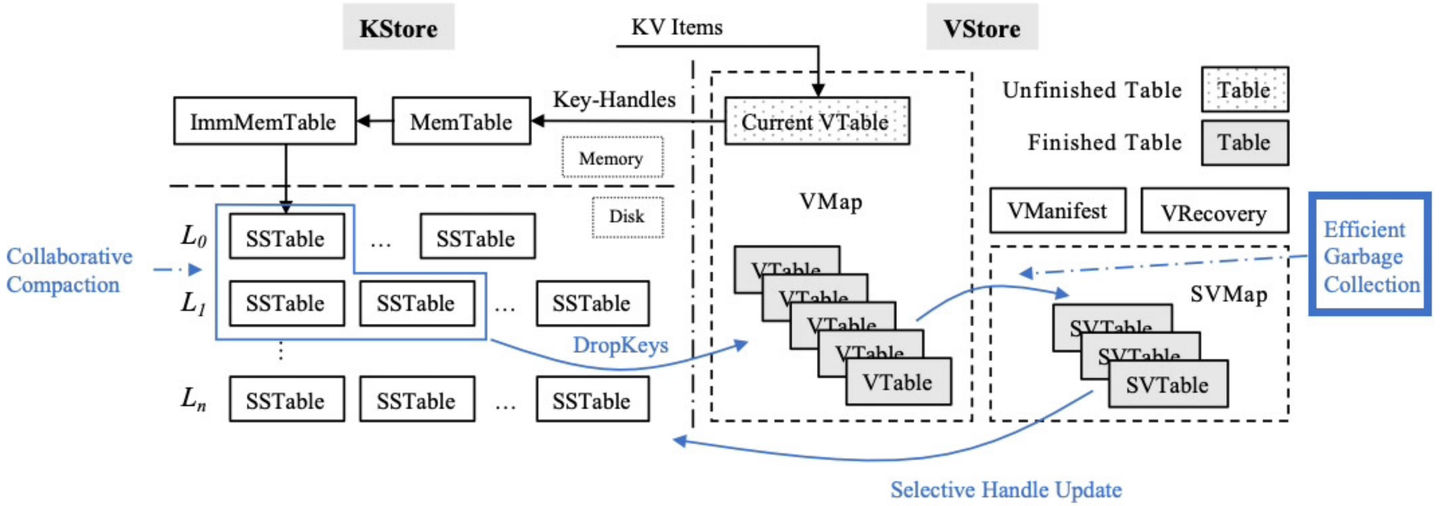




Design – Efficient Garbage Collection

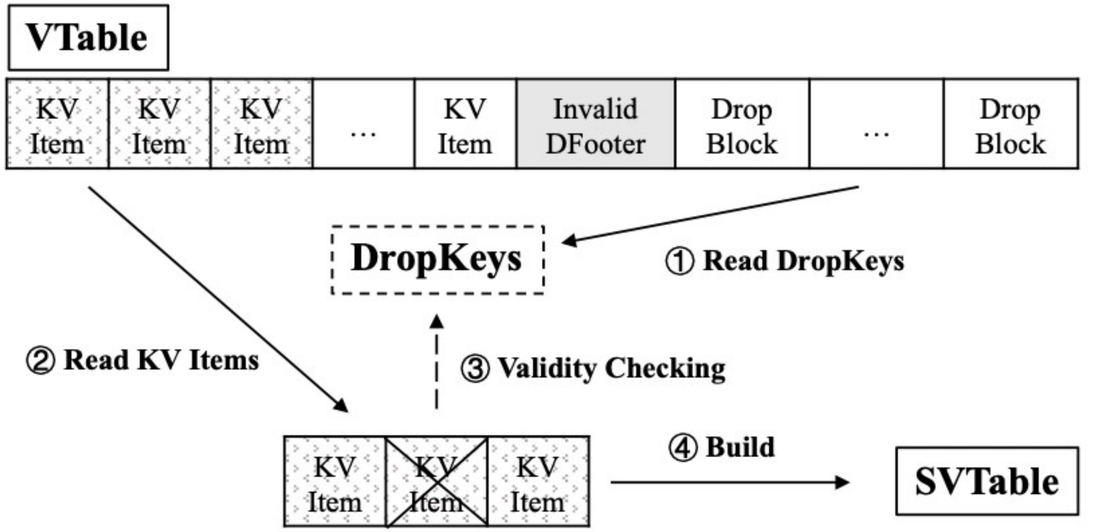
VTable GC

- Reads DropKeys from DropBlocks
- Reads a chunk of KV items
- Checks whether the KV item's key is in DropKeys
- If it is, discards this item
- If it is not, adds it to the new SVTable



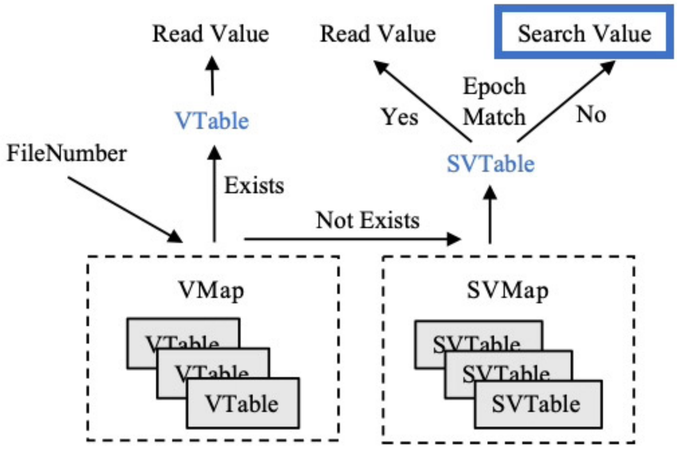
SVTable GC

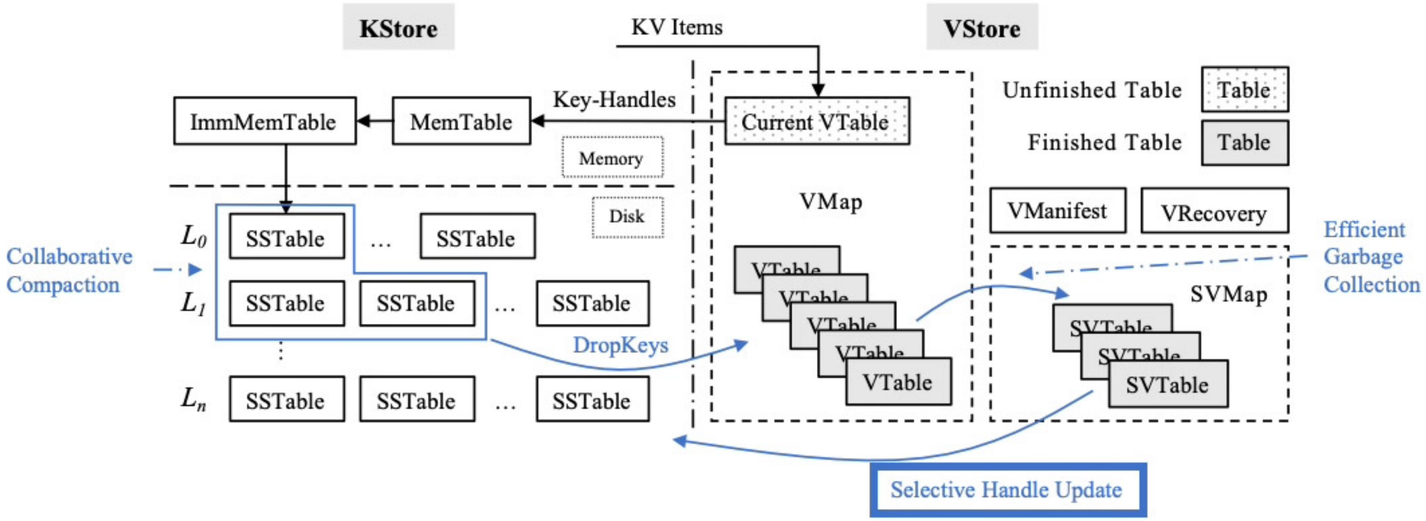
- Almost the same as VTable



Design – Selective Handle Updating

- Updates a value handle until the read request finally searches the SVTable



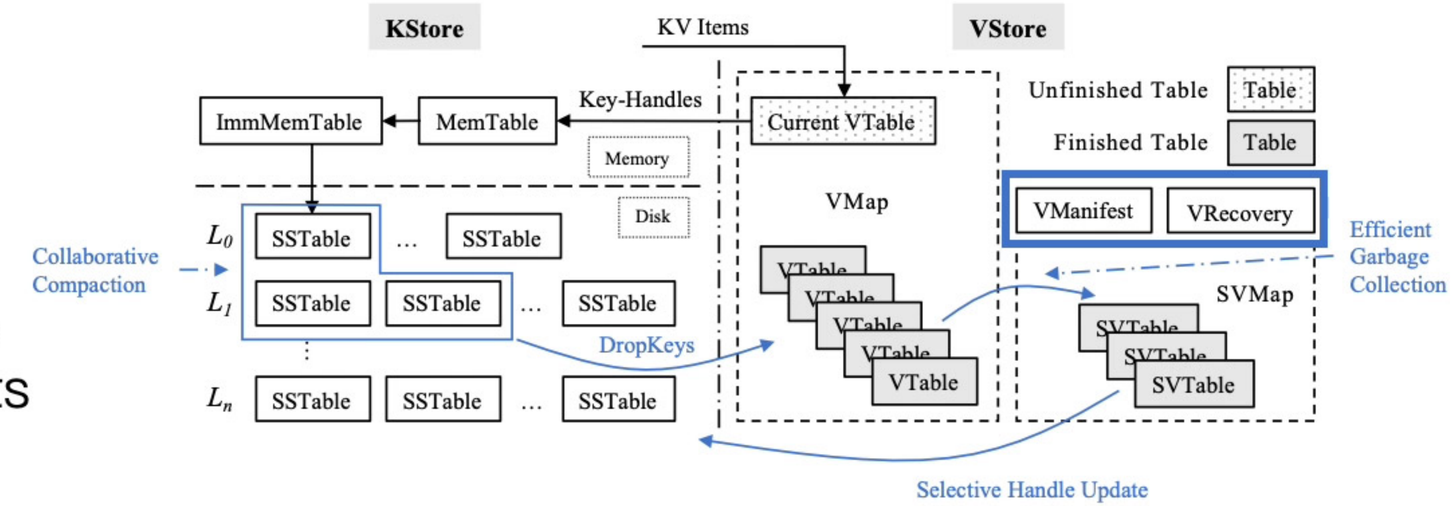


- Only needs to insert the new value handle into memory
- Do not need to write the WAL (if crash happens, just updates it again)
- A completely memory write

Design – Failure Recovery

Recovers from a failure

- VManifest: rebuilds the VMap, SVMap, and the current VTable
- VRecovery: reads all KV items after the recovery point from VTables, and inserts keys & value handles into the KStore



A failure happens during compaction

- Cleaned up after failure recovery

✓ DropKeys can be collected again in the next compaction

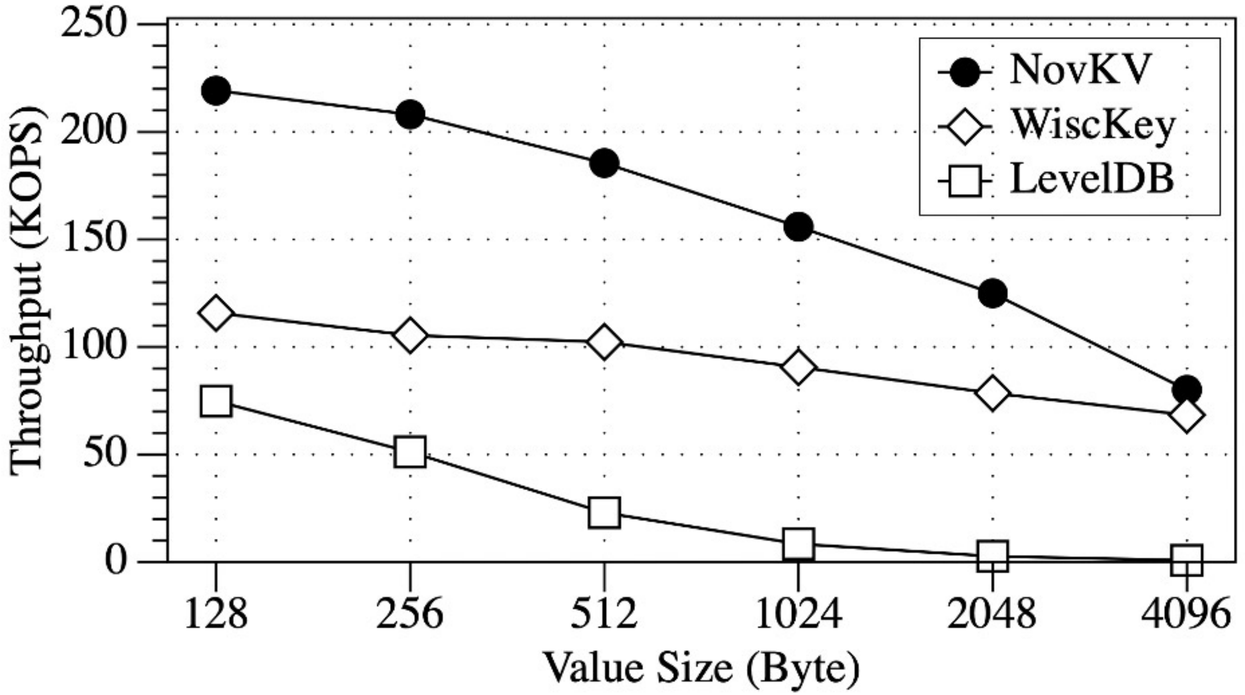
A failure happens during garbage collection

- Cleaned up after failure recovery

✓ Redoes it

Evaluation – Random Write

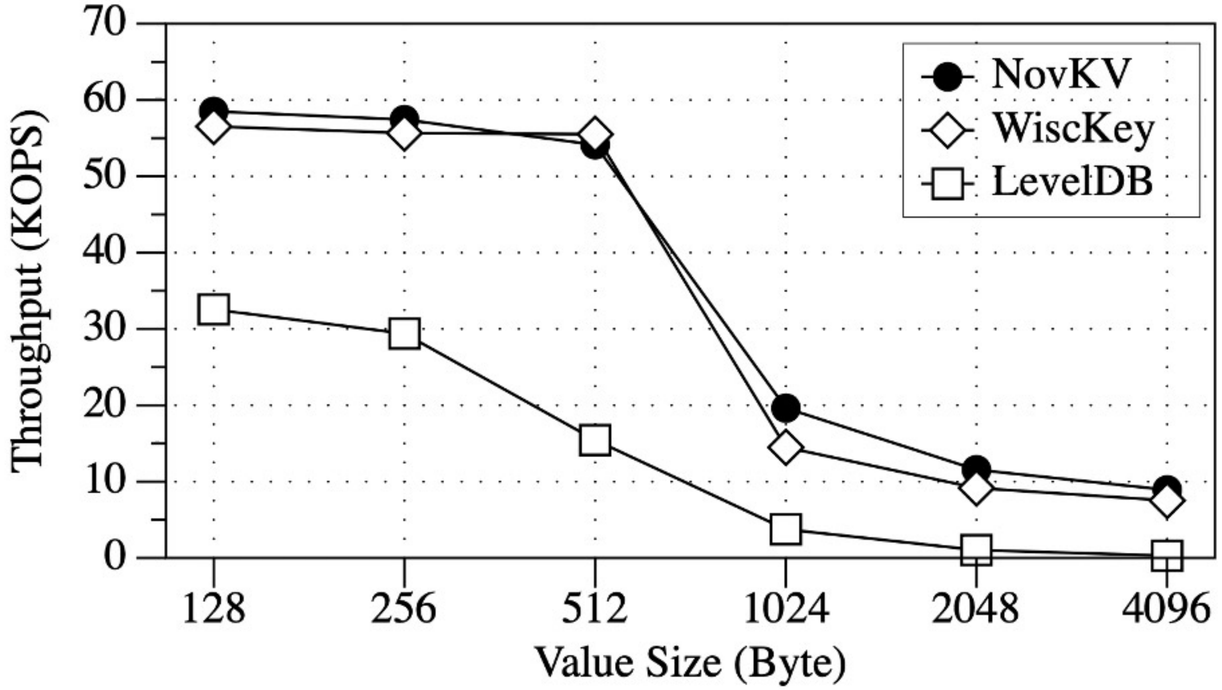
- Inserts 100M KV items in random order
- The throughput of NovKV is at most roughly 2× over WiscKey



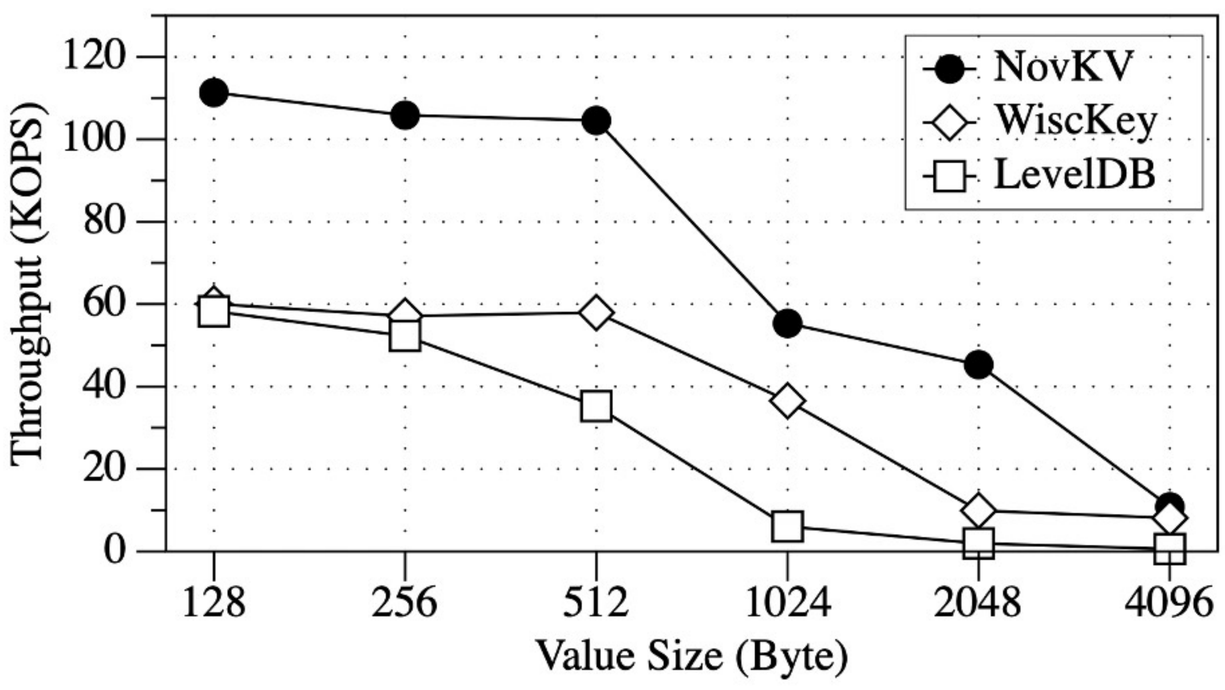
Evaluation – Random Read

- Read 10M KV items
- Almost the same as WiscKey in the first run
- Roughly 2× over WiscKey in the second run

The First Run

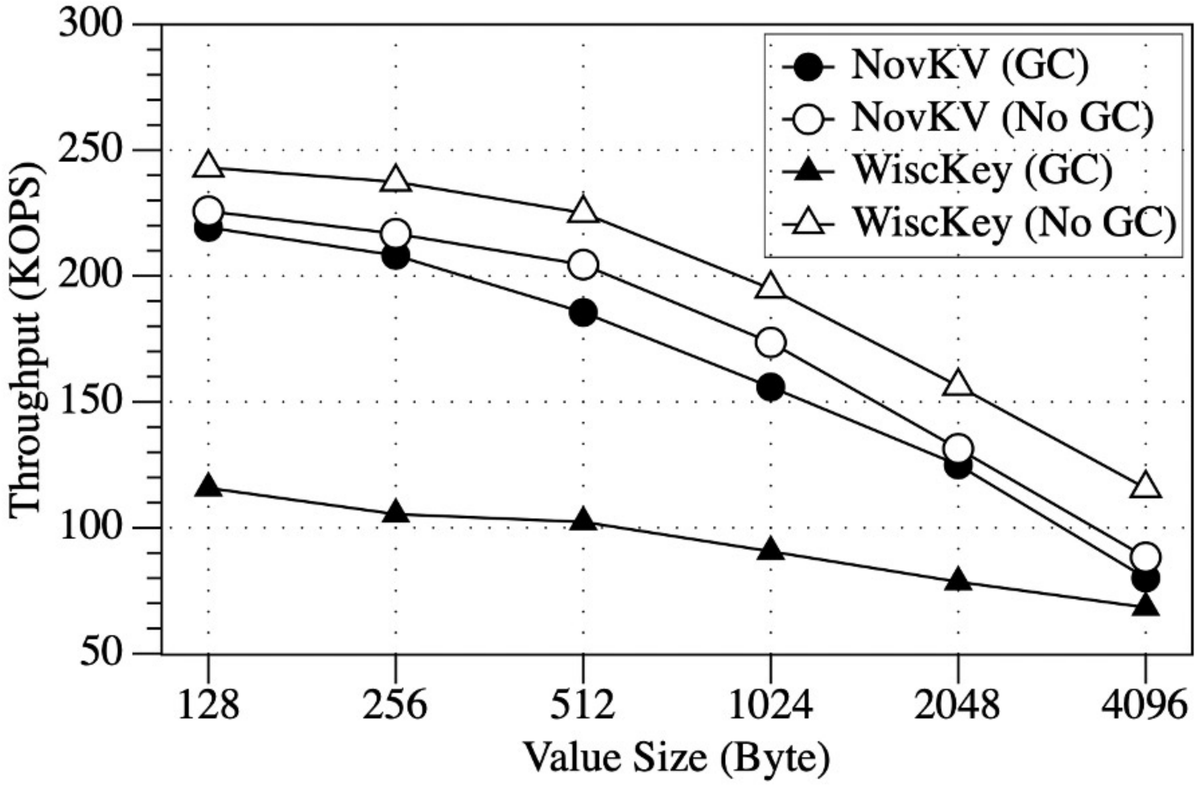


The Second Run



Evaluation – With & Without GC

- The negative impact of GC in NovKV is much lower than in WiscKey



Summary

Motivation

- Reduces the negative impact of garbage collection
- Leverages the dependencies between the KStore and VStore to eliminate the query and insertion of garbage collection

Key techniques

- Collaborative Compaction
- Efficient Garbage Collection
- Selective Handle Updating

Thanks

Q & A