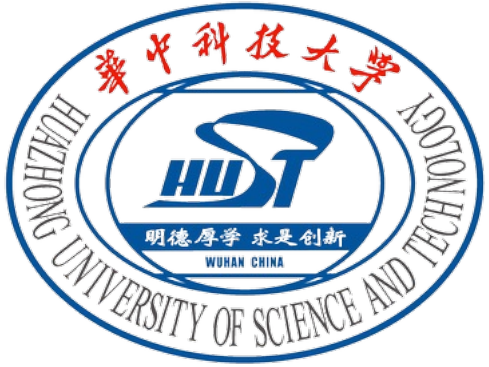


Accelerating Range Queries of Primary and Secondary Indices for Key-Value Separation

Chenlei Tang, Jiguang Wan, Zhihu Tan, Guokuan Li

Huazhong University of Science and Technology

Outline

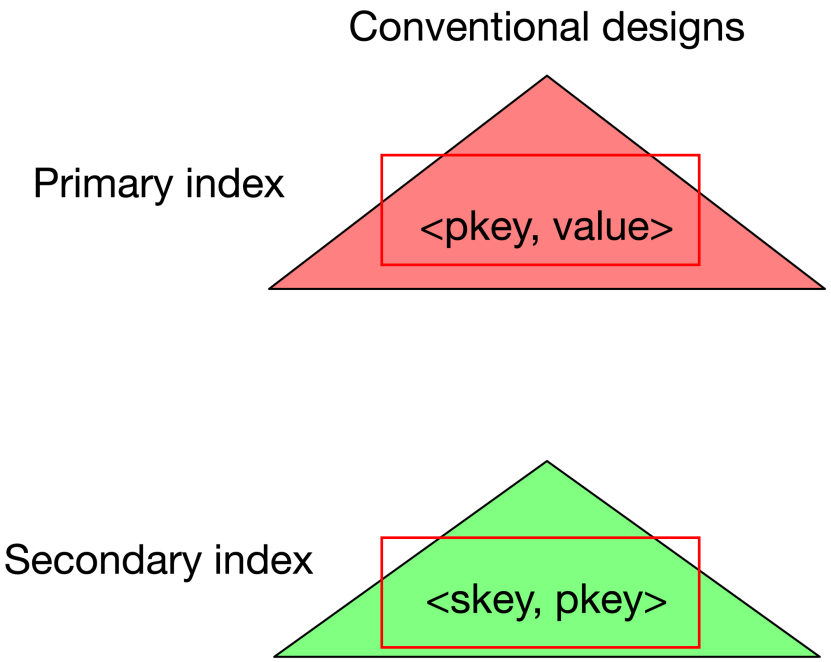
- **Background and Motivation**
- Design
- Evaluation
- Conclusion

Key-Value Store

- LSM-tree based key-value (KV) stores become the backbone of data centers
  - Primary index: support operations (e.g., Put, Get, Delete) on the primary key
  - Secondary index: allow queries on non-primary key attributes

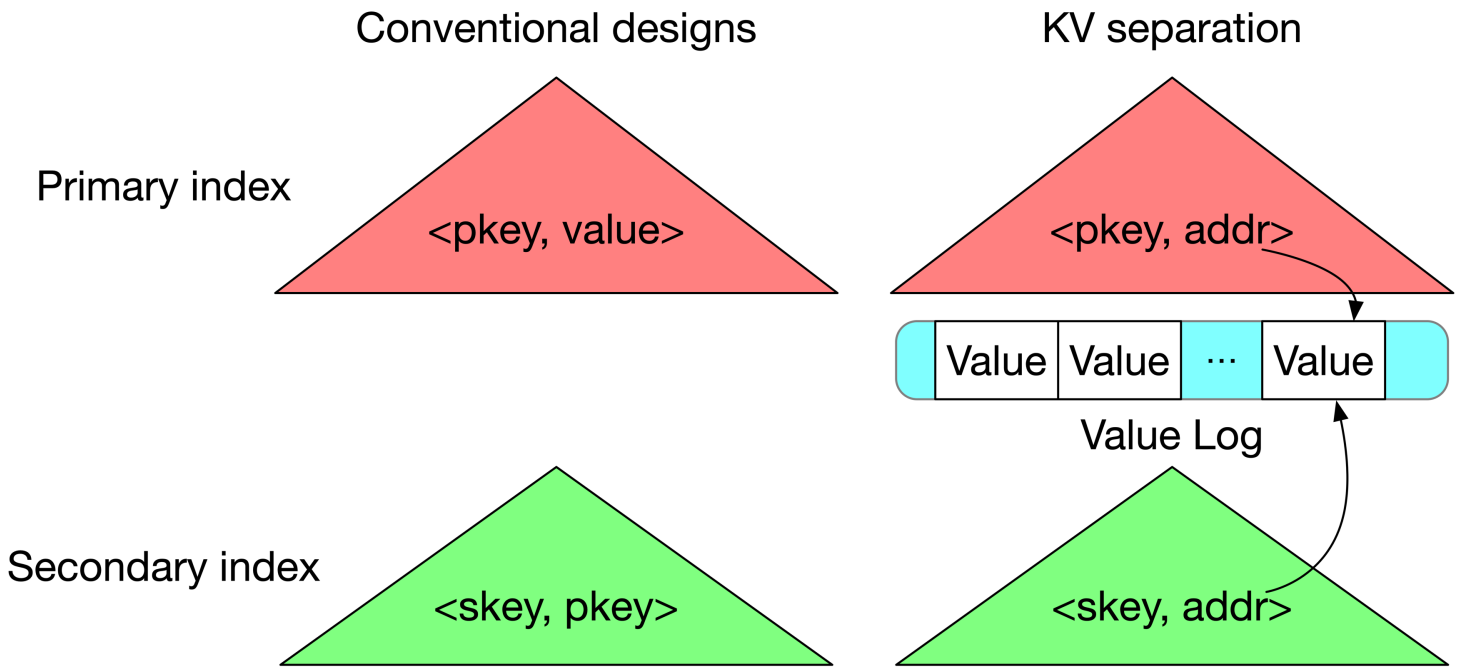
Primary and Secondary Indices

- Conventional LSM-tree based KV Store
  - Primary index: pkey -> value
  - Secondary index: skey -> pkey
- Index Navigation Problem
  - A query on a secondary key has extra lookup overhead



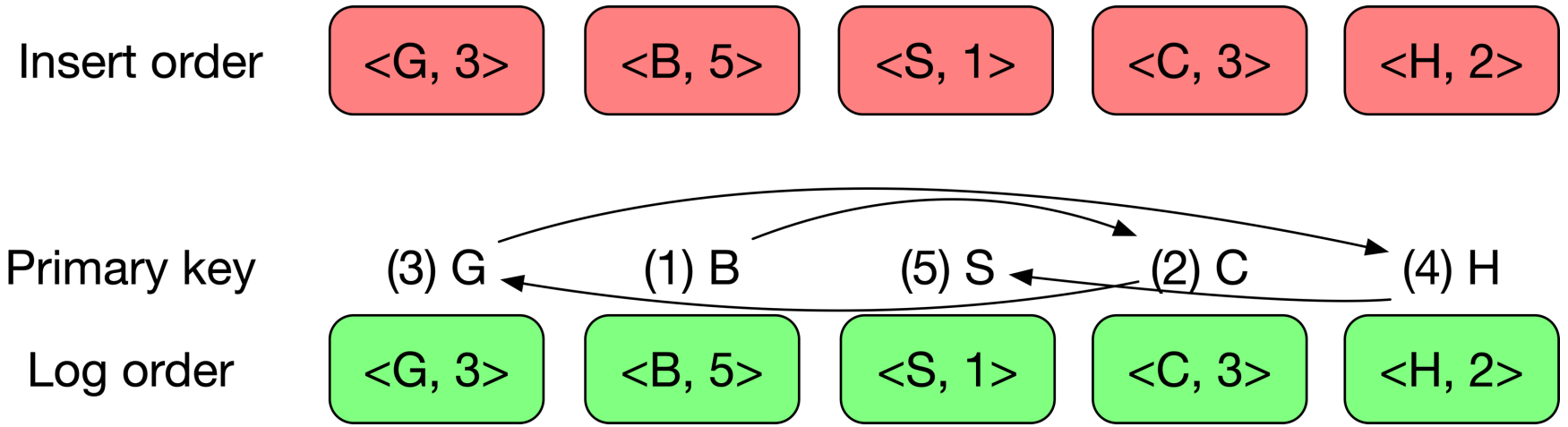
Primary and Secondary Indices

- KV Separation by SineKV
  - Primary index: pkey -> addr
  - Secondary index: skey -> addr
  - Values are stored in a separate value log
  - A query on a secondary key can directly get its value



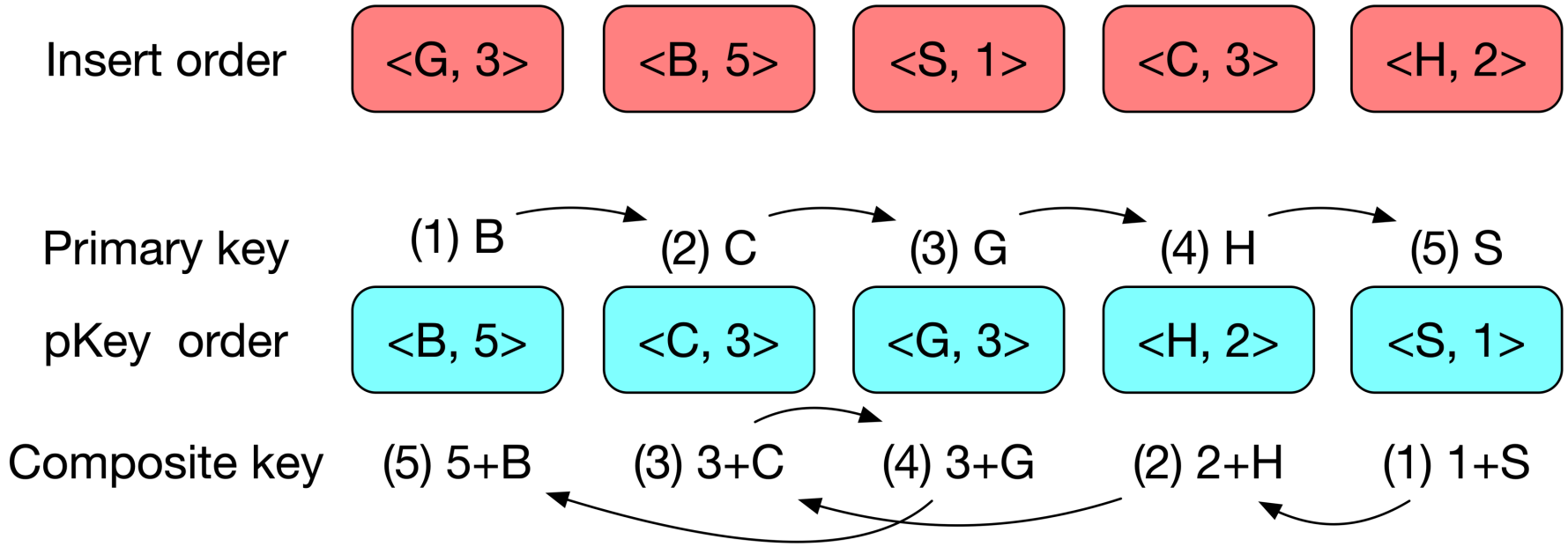
Challenges of Range Query Optimization

- Challenge 1: Insert/Update vs. Range Query
  - KV separation appends KV pairs in a log-structured manner for higher throughput
  - A range query has to issue multiple random reads (B->C->G->H->S)
  - Sequential reads are much faster than random reads on SSD devices



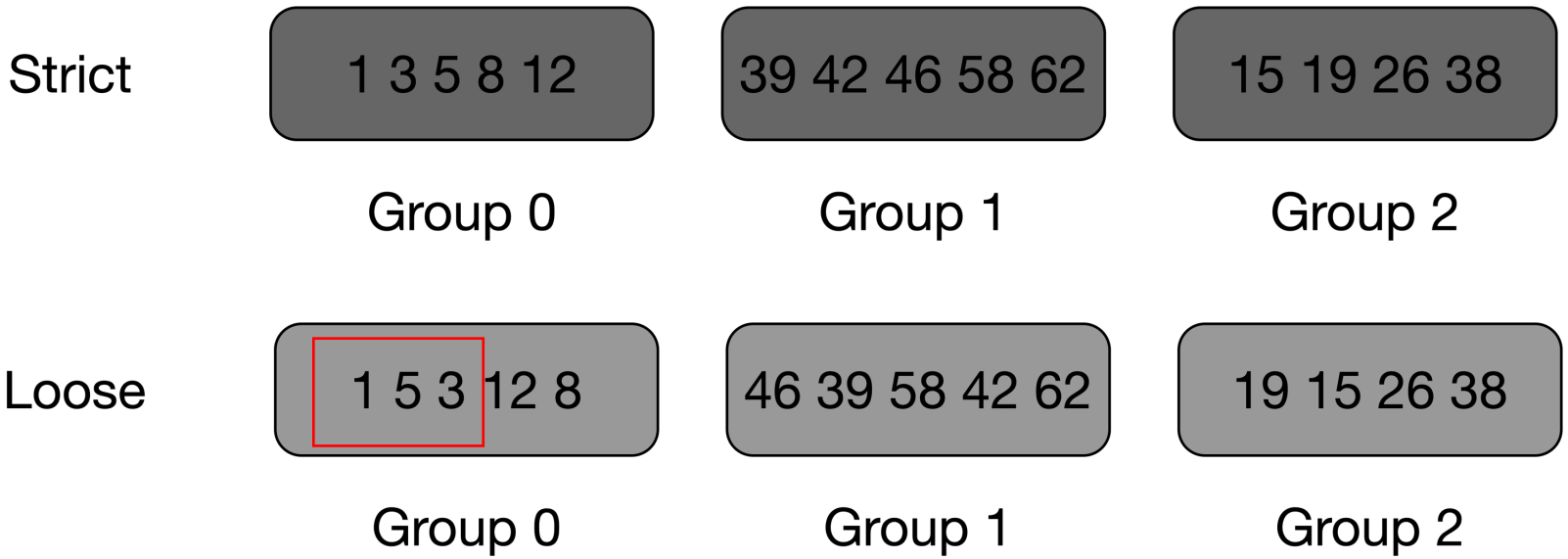
Challenges of Range Query Optimization

- Challenge 2: Primary Index vs. Secondary Index
  - The secondary index adopts composite-key (i.e., skey + pkey) method
  - When KV pairs are stored in primary key order, it issues random reads for the secondary index



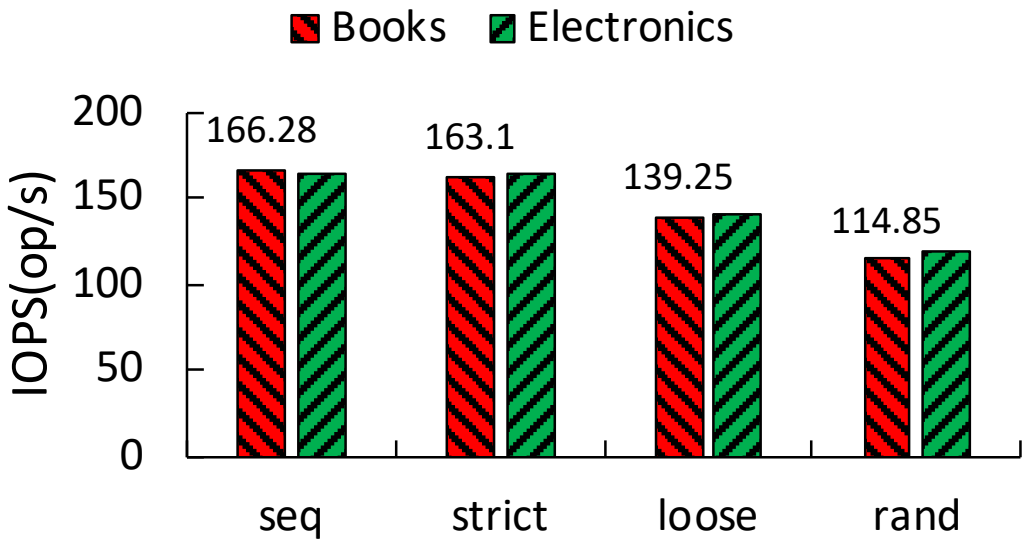
Observation

- Access characteristics of SSD devices provide new opportunities
- Strict sequentiality
  - each group in the value log covers a distinct key range
  - KV pairs are stored in key order for each group
- Loose sequentiality
  - each group in the value log covers a distinct key range
  - the internal key order can be relaxed



Observation

- Evaluate range query performance under different access patterns
  - WiscKey (a KV separation design, FAST' 16)
  - Amazon Books and Electronics review datasets
  - Access patterns: sequential, random, strict sequentiality, loose sequentiality



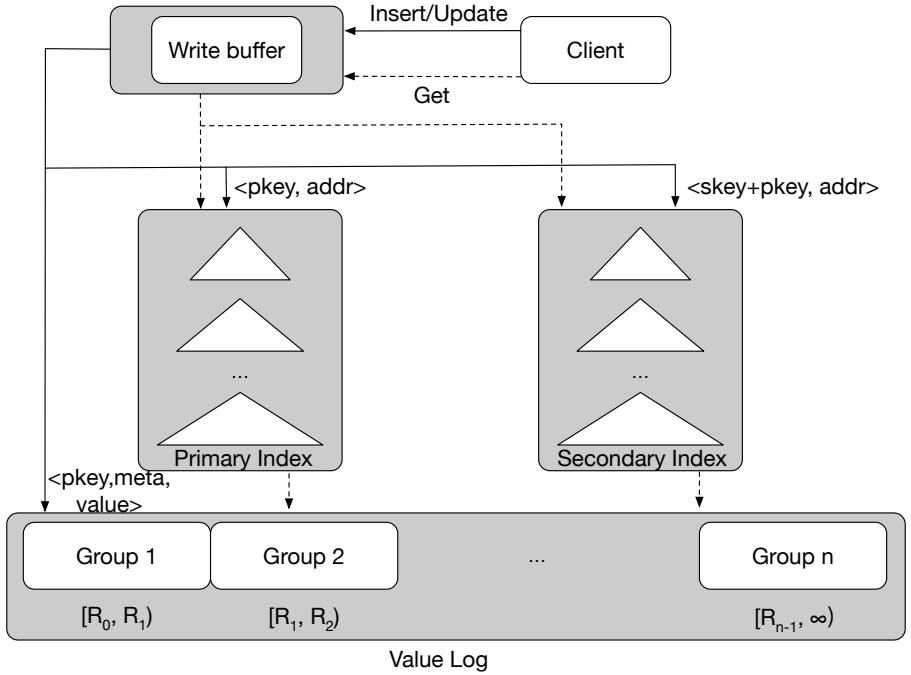
The range query performance can be improved even in loose sequentiality

Outline

- Background and Motivation
- **Design**
  - RISE Architecture
  - Key-Range Data Grouping
  - Co-Location Garbage Collection
  - Parallel Value Parsing
- Evaluation
- Conclusion

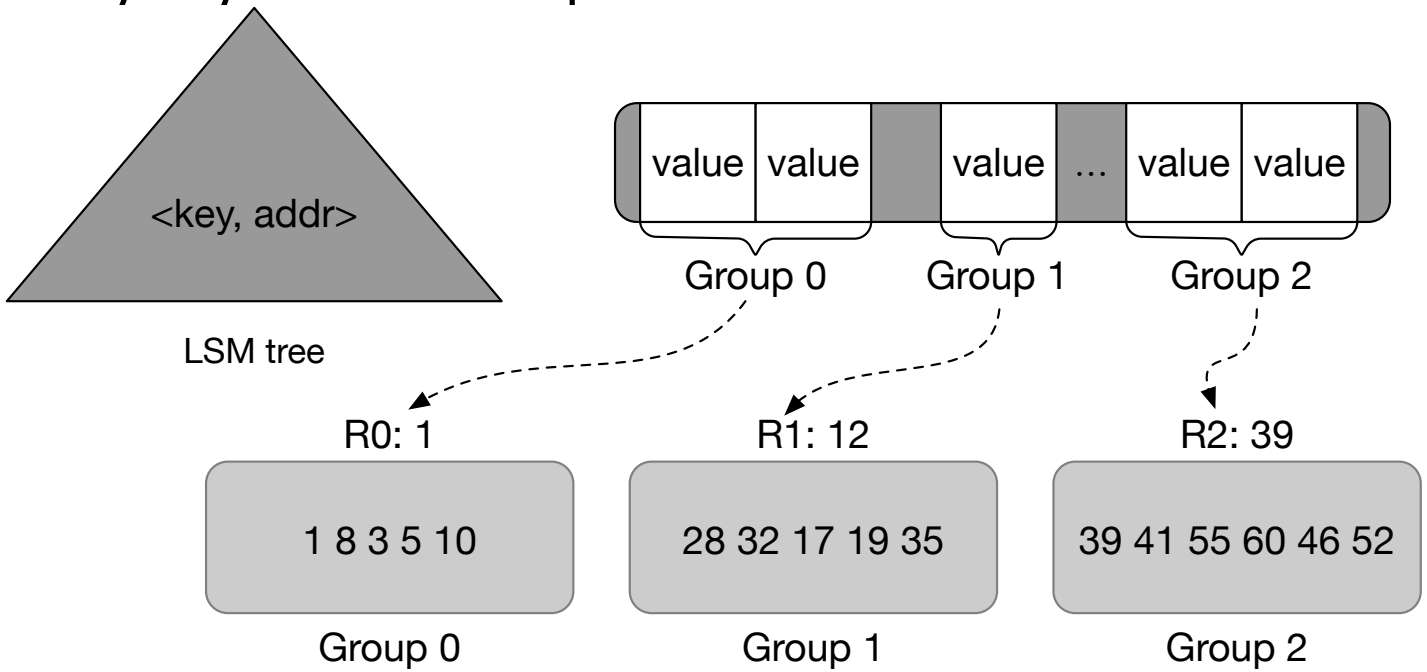
RISE Architecture

- RISE adopts the following three techniques:
  - Key-Range Data Grouping
    - Maintain loose sequentiality for the primary index
  - Co-location Garbage Collection (GC)
    - Maintain strict sequentiality for the secondary index
  - Parallel Value Parsing
    - Accelerate value parsing during GC



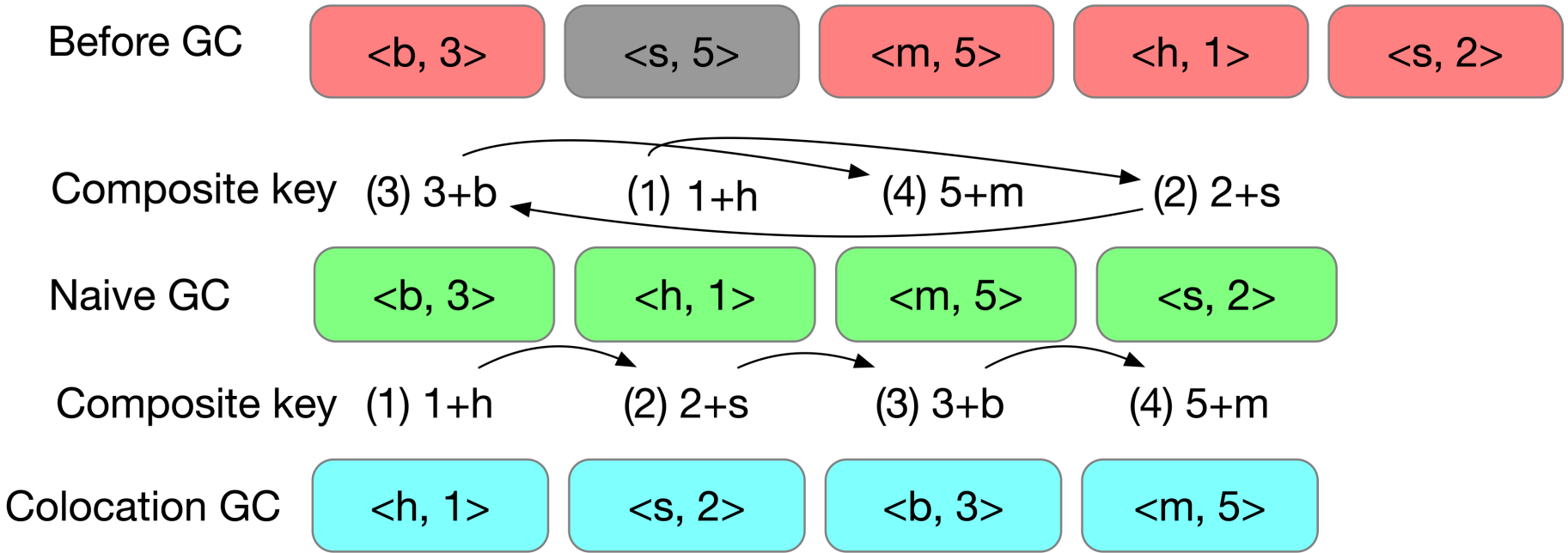
Design 1 - Key-Range Data Grouping

- Each group in the value log covers a distinct key range of primary key
  - Randomly assign a smallest key (e.g., R0) for each group to bound the key range
- Probabilistic Selection
  - Calculate a hash value for each key
  - Determine the smallest key of a group by the last n bits of the hash value
- Loose Sequentiality
  - Relax the primary key order of KV pairs



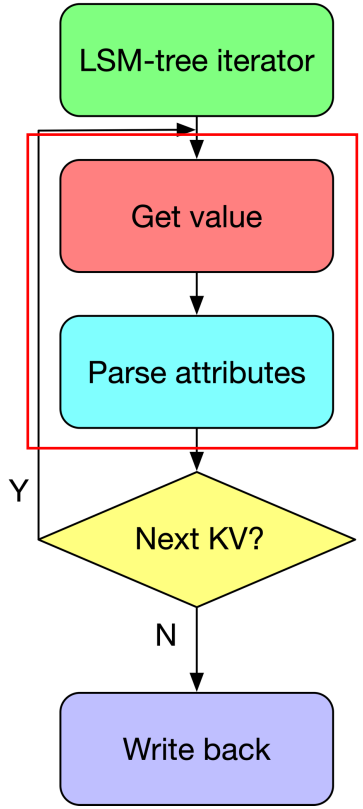
Design 2 - Co-Location Garbage Collection

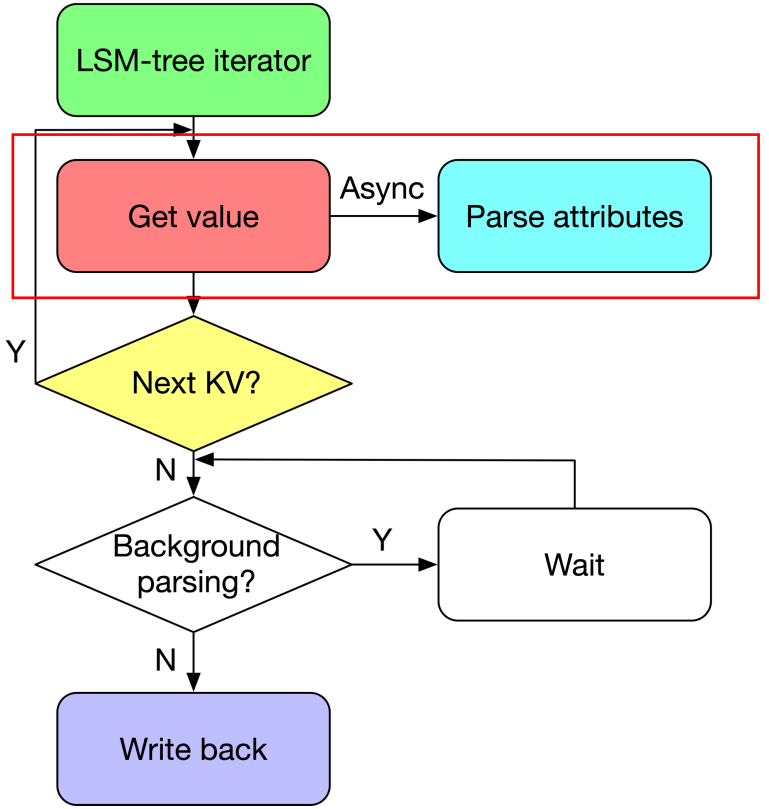
- Strict sequentiality by garbage collection
  - Group Selection: Select a group with the largest update proportion
  - Valid KV Pairs Identification: Issue a range query of pkey to get all valid KV pairs
  - Secondary Attributes Parsing: Parse KV pairs in parallel to get secondary attributes
  - Co-location and Write back: Co-locate KV pairs that have same secondary attributes, write back in composite-key order
- Example - <pkey, skey> represents a KV pair



Design 3 - Parallel Value Parsing

- It has to parse attributes to update indices during GC
  - Naive value parsing: process in serial
  - Our parallel value parsing: process in parallel





Outline

- Background and Motivation
- Design
- **Evaluation**
- Conclusion

Setup

- Baseline
  - LevelDB++ and SineKV
  - Implement SineKV atop WiscKey and HashKV
- Benchmark
  - Chirpx: workload generator derived from chirp (SIGMOD’ 18) support regular datasets
  - Amazon Books and Electronics review datasets

- Testbed

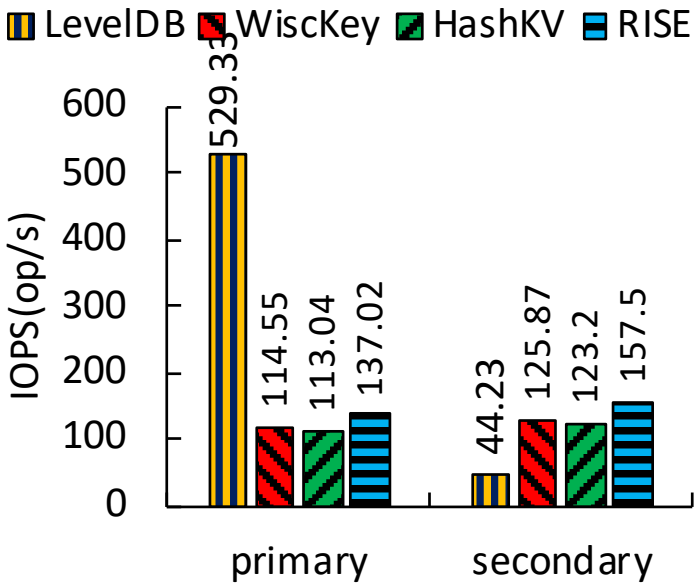
CPU	Intel Xeon E5-2660@2.2GHz
DRAM	64GB DDR4
SSD	Samsung 860 EVO
OS	Ubuntu 16.04 LTS

- Workloads

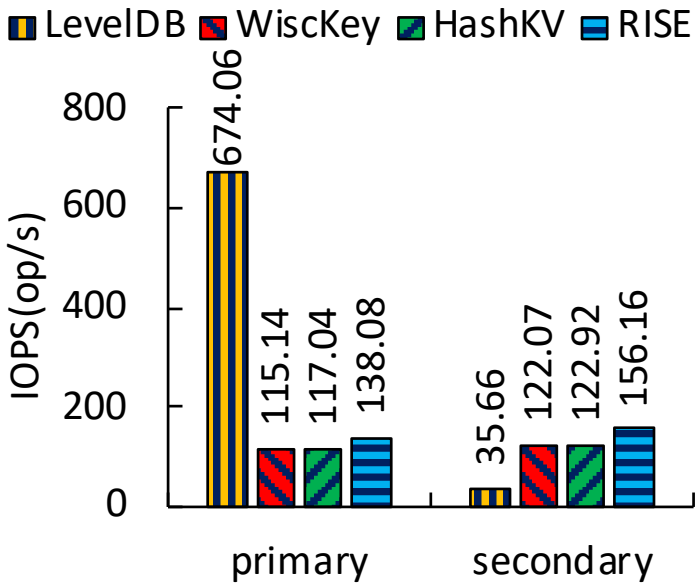
	Load	Update	Range Query
Books	9M	9M	100K
Electronics	21M	21M	100K

Range Query Evaluation

- Books dataset



- Electronics dataset

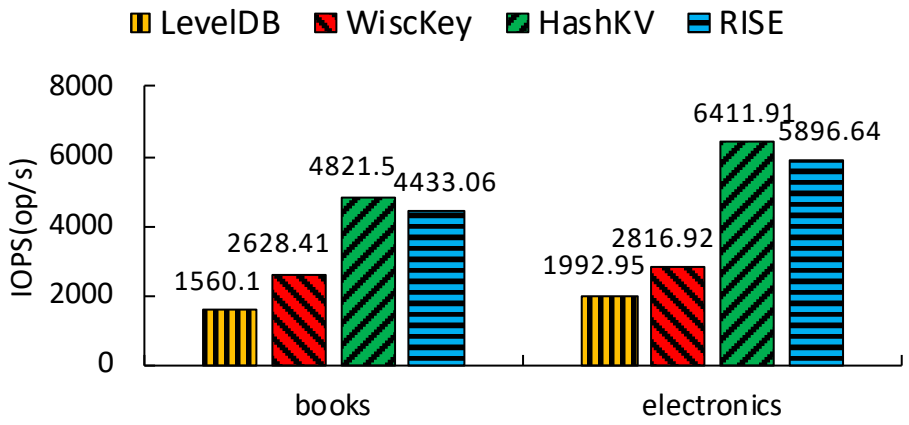


- Primary index
  - LevelDB achieves high throughput when the workload is not heavy
  - RISE outperforms WiscKey and HashKV
    - 1.21x WiscKey
    - 1.2x HashKV

- Secondary index
  - LevelDB achieves the worst throughput due to the index navigation overhead
  - RISE outperforms WiscKey and HashKV
    - 1.28x WiscKey
    - 1.28x HashKV

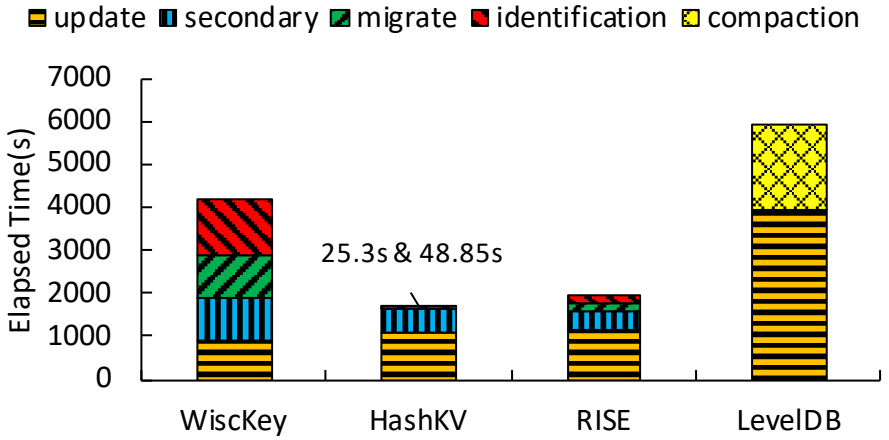
Update Evaluation

- Update Performance



- RISE outperforms LevelDB and WiscKey
  - 2.84x ~ 2.95x LevelDB
  - 1.68x ~ 2.09x WiscKey
- RISE achieves lower throughput than HashKV
  - 0.91x ~ 0.92x HashKV

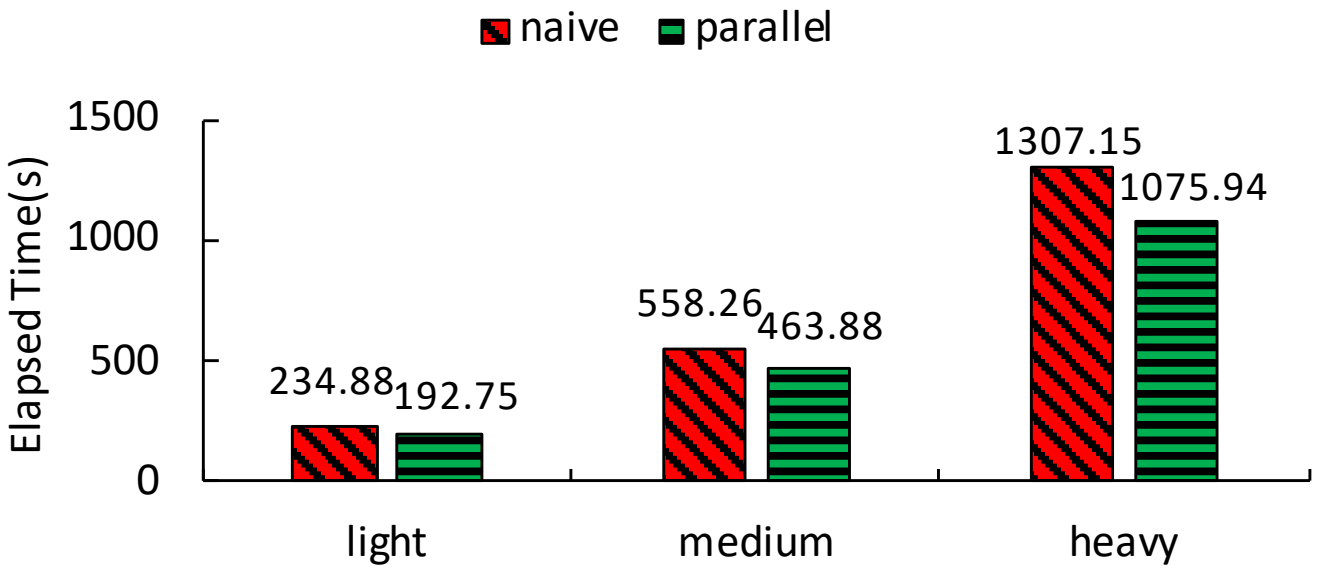
- Update Time Breakdown



- RISE costs more time for valid KV pairs identification than HashKV

Parallel Value Parsing Evaluation

- Value parsing under different update traffic
  - Vary traffic from light (4.5M) to medium (9M) and heavy (18M)

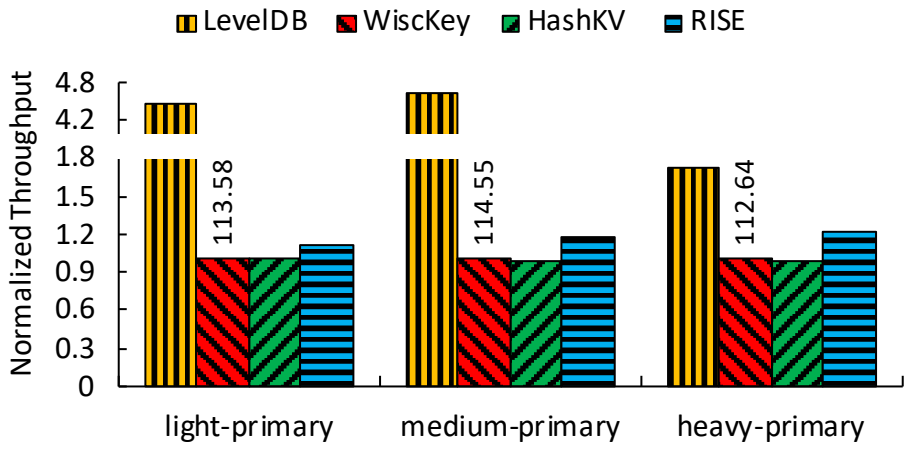


- RISE can accelerate the parsing procedure under different update traffic
  - 16.9% ~ 17.9% performance improvement

Scalability Evaluation

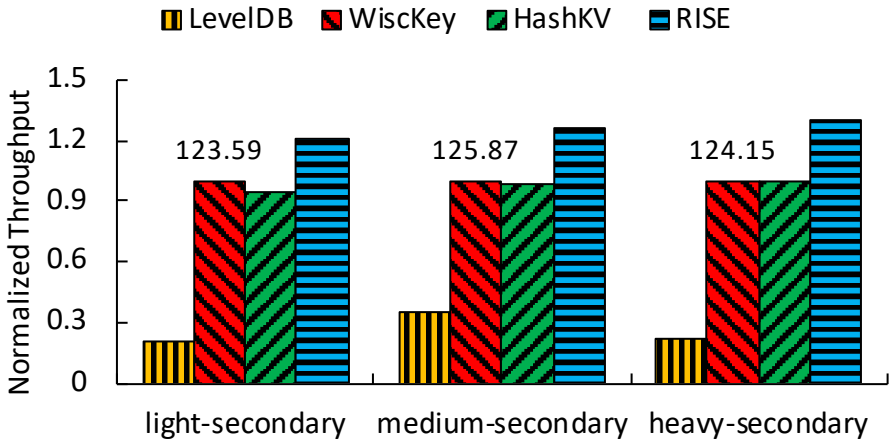
- Range queries under different update traffic
  - Vary the update traffic from 4.5M to 9M and 51M

- Primary index



- The performance of LevelDB degrades significantly with the increase of update traffic
- RISE can outperform WiscKey and HashKV
  - 1.12x ~ 1.21x WiscKey
  - 1.12x ~ 1.23x HashKV

- Secondary index



- RISE achieves the best performance
- RISE can outperform WiscKey and HashKV
  - 1.21x ~ 1.29x WiscKey
  - 1.27x ~ 1.31x HashKV

Outline

- Background and Motivation
- Design
- Evaluation
- **Conclusion**

Conclusion

- KV separation induces challenges for range queries
  - Insert/update vs. range query
  - Primary index vs. secondary index
- We propose RISE, aiming to accelerate range queries of primary and secondary indices
  - Key-range data grouping
  - Co-location garbage collection
  - Parallel value parsing
- Evaluation shows RISE can outperform existing KV separation designs
  - Range query of the primary index by up to 23%
  - Range query of the secondary index by up to 31%
  - Value parsing of GC by up to 17.9%

Thanks

Accelerating Range Queries of Primary and Secondary Indices for Key-Value Separation

Chenlei Tang, Jiguang Wan, Zhihu Tan, Guokuan Li

Huazhong University of Science and Technology