USENIX
ATC '23

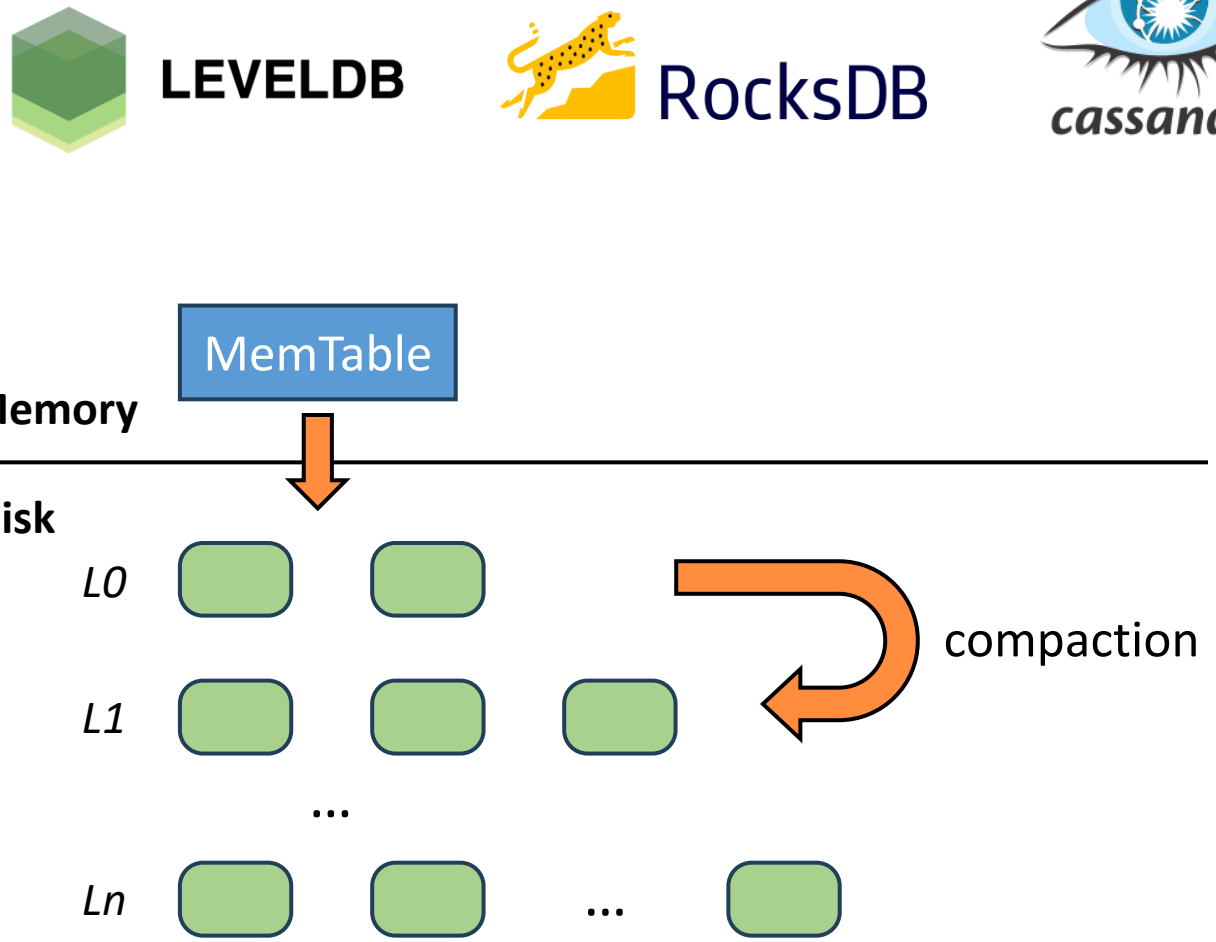
Revisiting Secondary Indexing in LSM-based Storage Systems with Persistent Memory

Jing Wang, Youyou Lu, Qing Wang, Yuhao Zhang, Jiwu Shu

Tsinghua University

LSM-tree based KV Stores / Databases

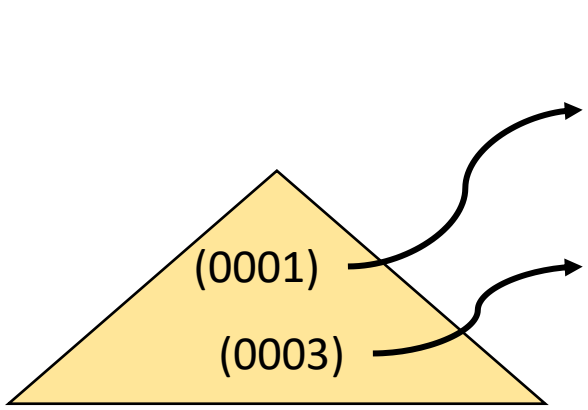
Log-structured merge tree (LSM-tree) is widely adopted



- ❖ High write performance
  - Blind-write (write without read)
  - Buffer writes in memory
- ❖ Inferior read performance
  - Multi-level structure
  - Computing overheads of indexing and Bloom filters

Secondary Indexing

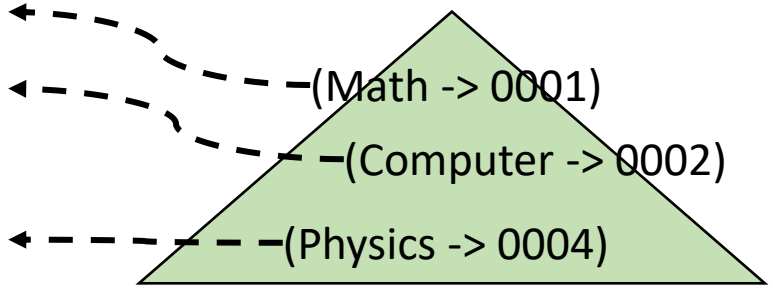
- Primary Index (or integrated primary table): indexed by primary key (StudentID)
- Querying by non-primary-key is common. E.g., find students whose major is *Computer*
- Secondary Index:
  - Additional index maintaining mappings of other fields to primary key. E.g, { Major → StudentID }
  - Besides the main index based on primary key, all other indexes are secondary indexes
  - Indispensable technique in database systems



Primary Index

StudentID	Name	Major	EnrollmentYear	...
0001	Alice	Math	2021	
0002	Bob	Computer	2021	
0003	Carol	Music	2022	
0004	Dave	Physics	2022	
0005	Ellen	Computer	2023	

Primary Table



Secondary Index

Secondary Index in LSM-based Systems

Secondary indexing is inefficient with LSM-tree

1. Consistency among indexes is troublesome due to *blind-write*

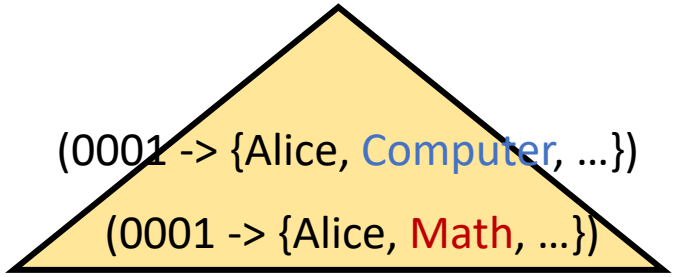
   E.g., update Alice(0001)'s major from Math to Computer.

   PUT: { 0001 -> Alice, Computer, …} in LSM-tree

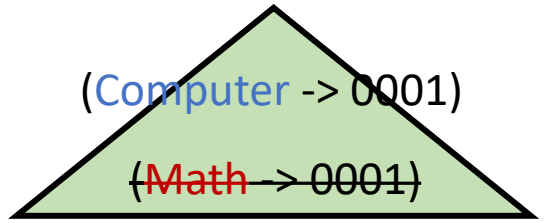
   In secondary index:

   1. insert new entry { Computer → 0001 }
   2. delete old entry { Math → 0001 }

   Problem: Do not know the old secondary key Math due to blind-write !



Primary Table



Secondary Index

Two strategies for this issue:

1. Synchronous: READ old record to get old secondary key Math, and then delete in secondary index ----> discard blind-write, low write performance

2. Validation: keep old entry { Math → 0001 }, but at query, fetch record of '0001' in primary table for validation ----> low query performance

Secondary Index in LSM-based Systems

Secondary indexing is inefficient with LSM-tree

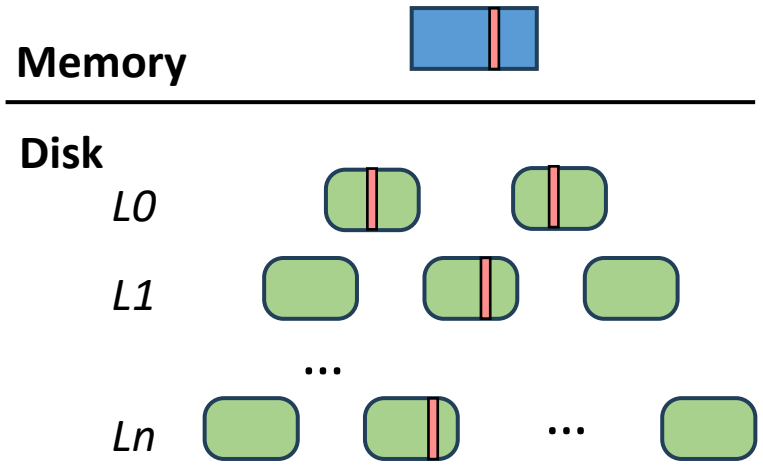
2. Inferior read performance is not friendly to secondary indexing

❖ Secondary index:
- ➢ KV pairs are small (value is just *primary key*)
- ➢ Non-unique (multiple values)

Mismatch!

❖ LSM-tree:
- ➢ Disk & Block based
- ➢ Multi-level

LSM Secondary Index



Attributes of secondary indexes and LSM-tree are mismatched!

Persistent Memory

Using persistent memory (PM) for secondary indexing is promising

- Byte-addressability
- DRAM comparable latency
- Data persistency

Plenty of PM-based indexes (ordered)

wB+Trees [VLDB'15] FPTree [SIGMOD'16] WORT [FAST'17] FAST&FAIR [FAST'18]

Recipe [SOSP'19] LB+Trees [VLDB'20] DPTree [VLDB'20] ROART [FAST'21]

Nap [OSDI'21] TIPS [ATC'21] PACTree [SOSP'21] NBTree [VLDB'22] …

PM-based Indexes for Secondary Indexing

Directly adopting existing PM indexes for secondary indexing is inefficient

How to handle the feature of *non-unique* ? (Computer → {A, B, E, …})

- Alloc space for all values {A, B, E, …} with allocator (e.g., slab-based)
  - Add/Remove mapping → value changes with size → frequent reallocation
  - Heavy persistence overheads
- Only alloc for new value, and link all values. {A} -> {B} -> {E}
  - Scatters values → low data locality, query performance
- Composite index

  Divided (Computer → {A, B, E, …}) into (Computer_A → {}), (Computer_B → {}), …
  - Values update → heavier insert/delete operations in PM index
  - Point query → heavier scan
  - Expanding the number of KV pairs → larger index → degraded performance

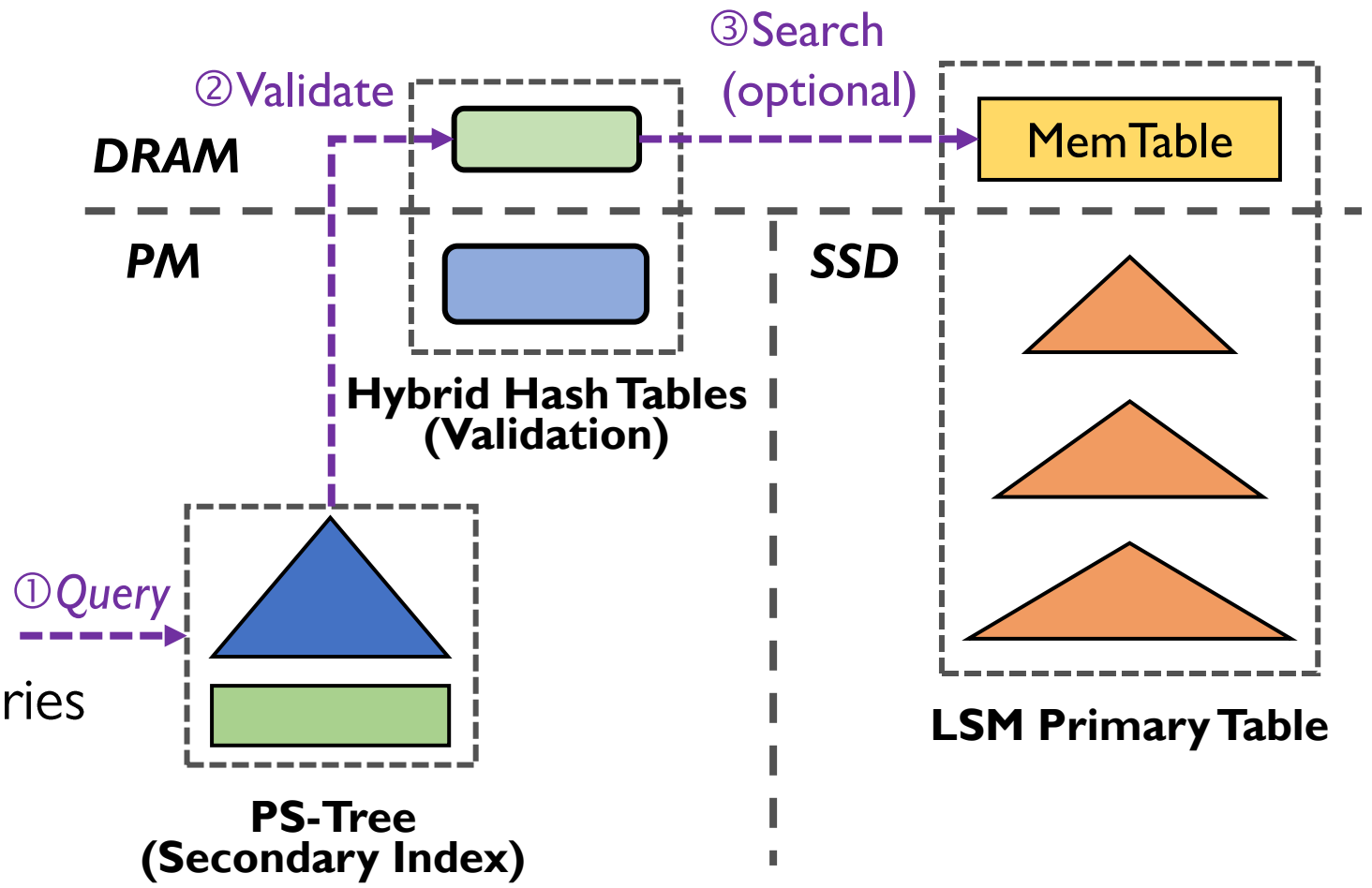
Outline

- Background & Motivation
- Perseid – a PM-based secondary indexing mechanism
- Results
- Conclusion

Overall

Perseid overview

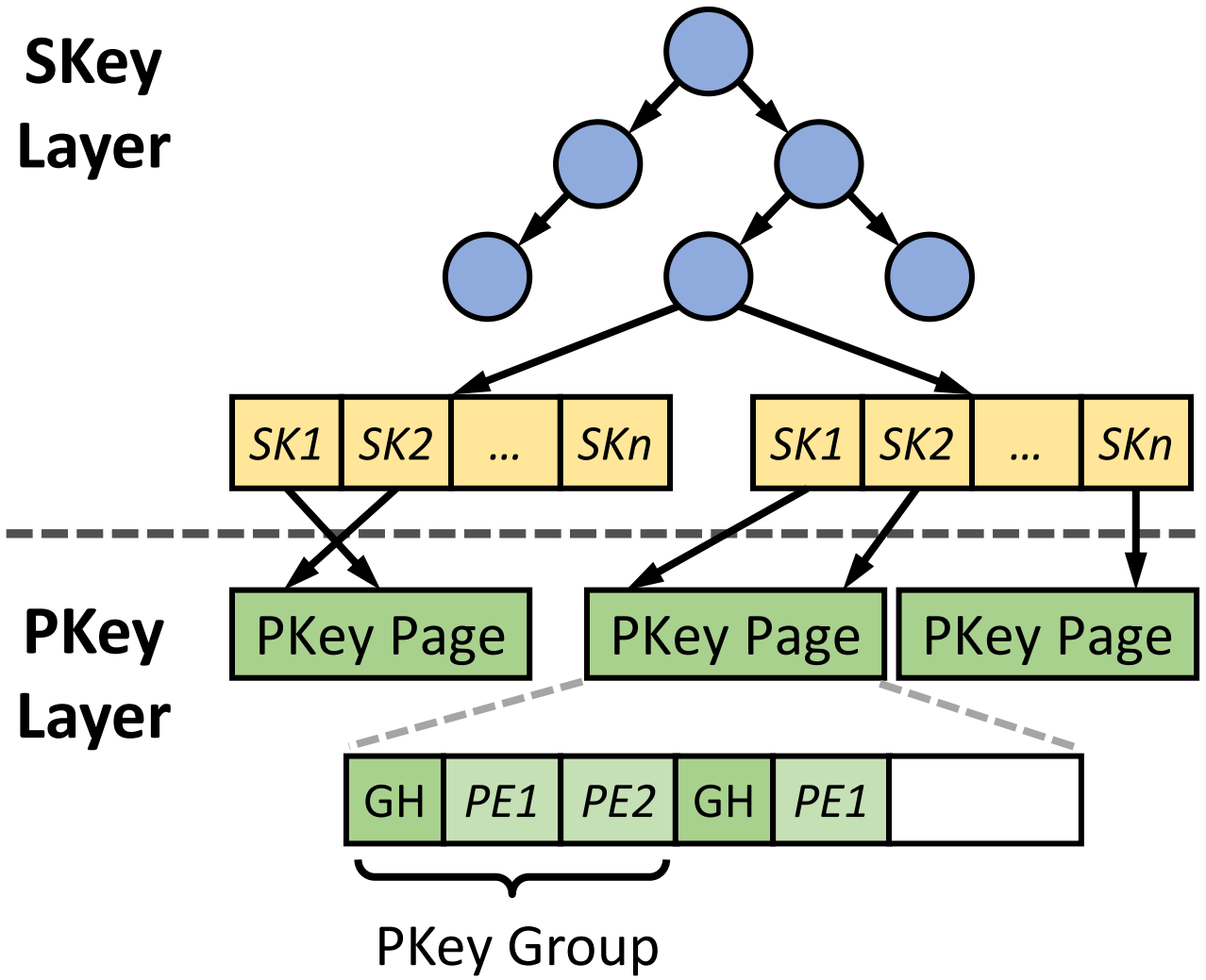
- PS-Tree (PM secondary index)
  - Specific layer for secondary values
  - PM-friendly log-structured insertion
  - Arranges entries with good locality
- Hybrid Hash Table
  - Retains blind-write of LSM
  - Lightweight validation on DRAM
- Optimizations for non-index-only queries
  - 1) filters out irrelevant component
  - 2) parallelizes primary table searching



PS-Tree

PM-based secondary index

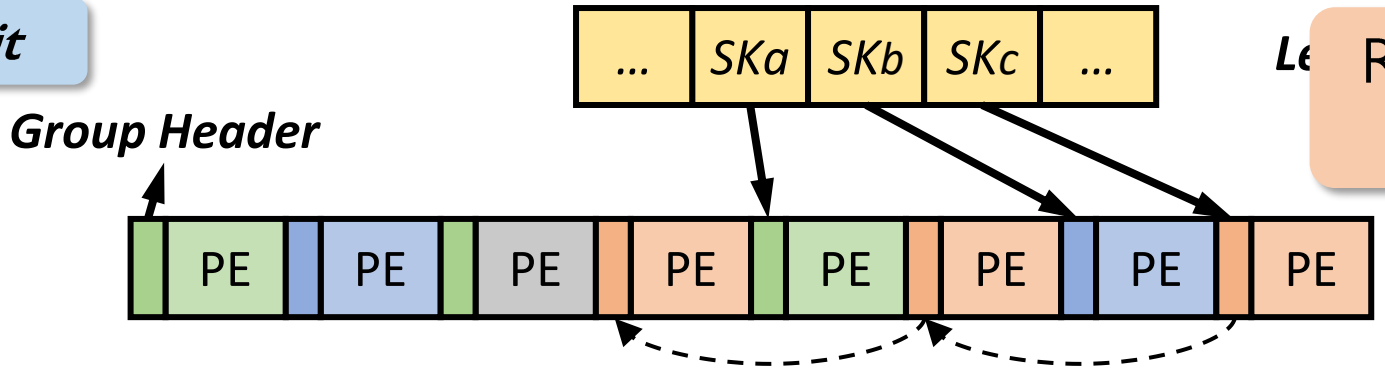
- SKey Layer
  - Index for secondary key to values in PKey Page
  - Leverage existing PM index
- PKey Layer
  - Store multiple values for Skeys
  - Append entries in PKey Pages (PM friendly)
  - Adjacent Skeys share PKey Pages (data locality)
  - Rearrange entries at splitting (data locality)
  - PKey Group
    - Contain a group header (GH) and multiple PKeys (PE) of the same Skey
    - PE contains PKey and its version
    - SKey points to latest PKey Group
    - Groups belong to one SKey are linked



PS-Tree

Rearrangement and garbage collection at splitting

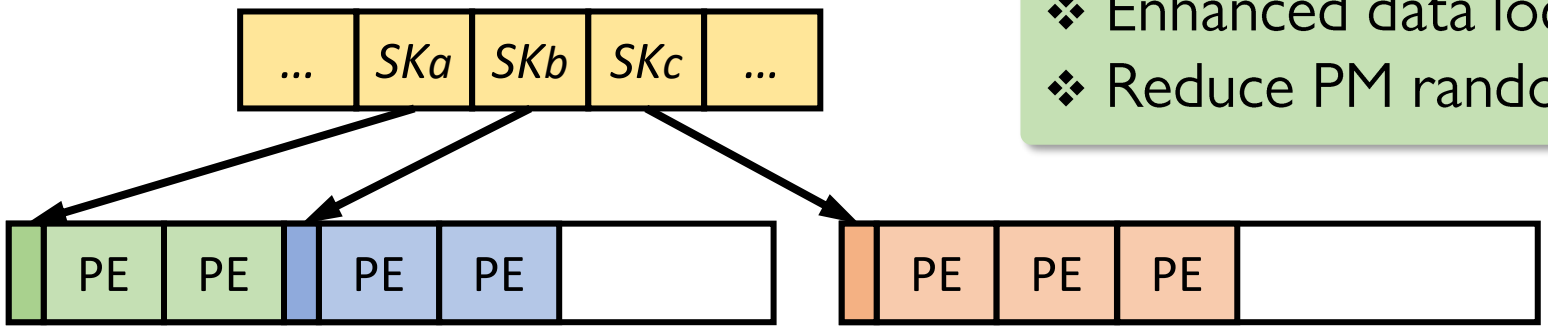
Before Split



Read multiple values with pointer chasing

PKey Page

After Split



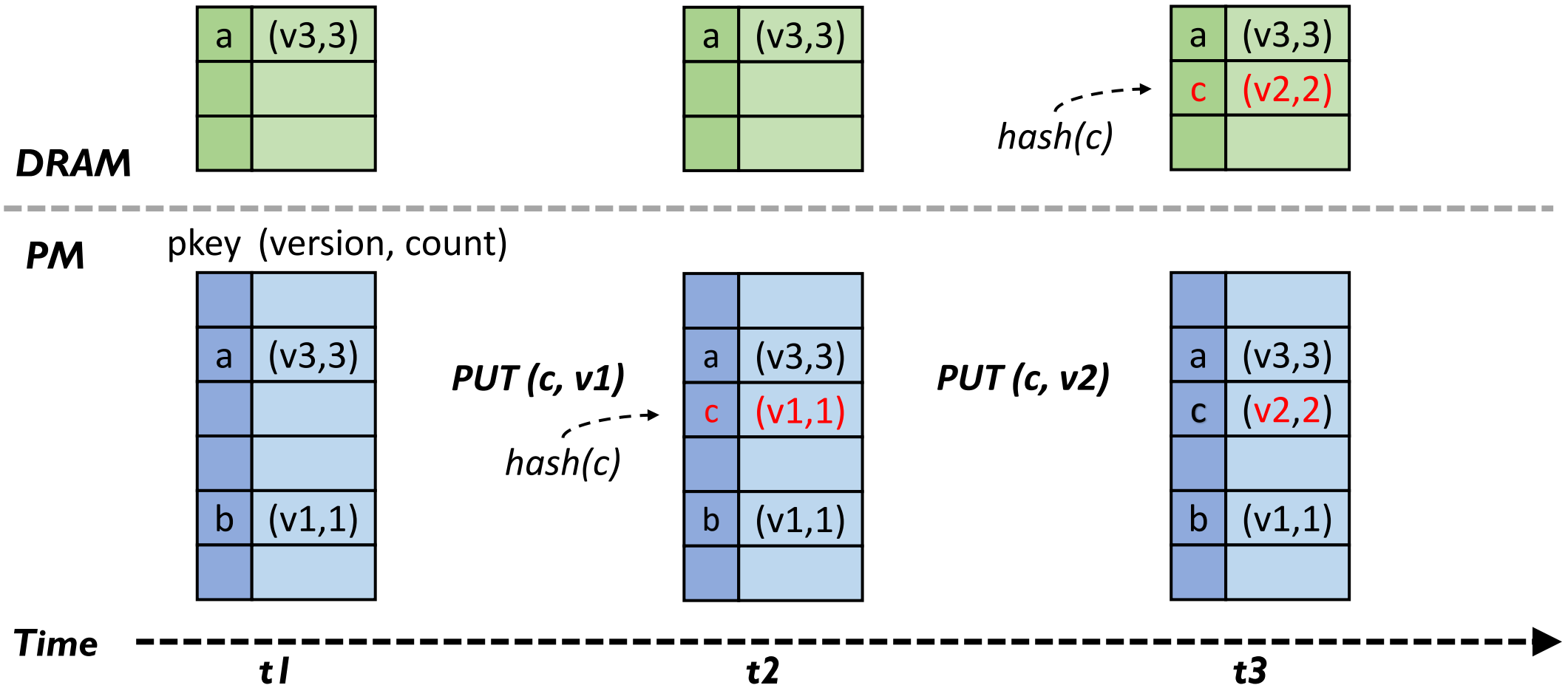
- Enhanced data locality
- Reduce PM random accesses

Hybrid PM-DRAM Hash-based Validation

- Retain *blind-write* of LSM primary table
- Maintain the latest version number for primary keys with hash table
- Validate using hash table instead of LSM primary table

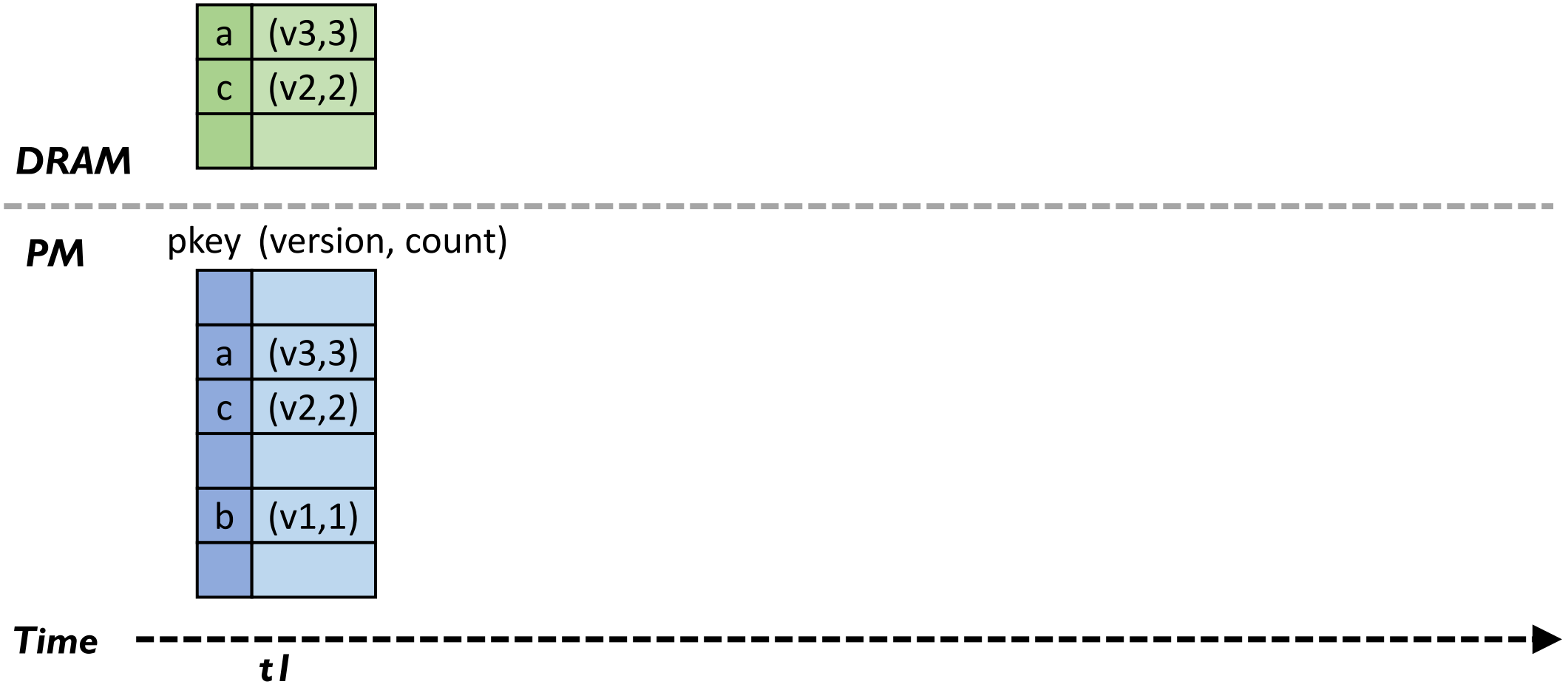
Hybrid PM-DRAM Hash-based Validation

- Retain *blind-write* of LSM primary table
- Maintain the latest version number for primary keys with hash table
- Validate using hash table instead of LSM primary table



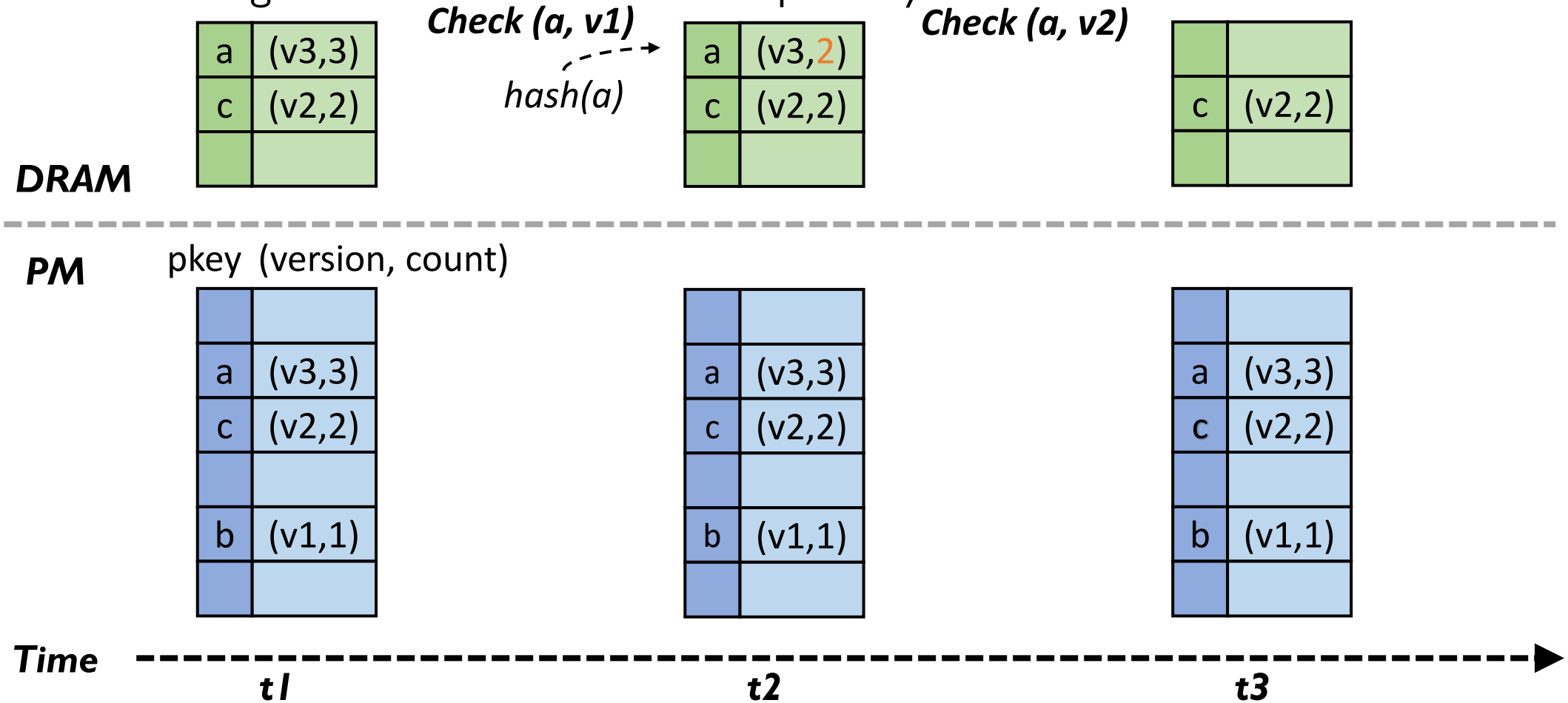
Hybrid PM-DRAM Hash-based Validation

- Retain *blind-write* of LSM primary table
- Maintain the latest version number for primary keys with hash table
- Validate using hash table instead of LSM primary table



Hybrid PM-DRAM Hash-based Validation

- Retain *blind-write* of LSM primary table
- Maintain the latest version number for primary keys with hash table
- Validate using hash table instead of LSM primary table



Non-Index-Only Query Optimizations

- **Index-Only Query**

  Query for specific columns

  e.g., SELECT *StudentID* FROM table WHERE Major = Computer,

  or, SELECT *COUNT(*)* FROM table WHERE Major = Computer

- **Non-Index-Only Query**

  Query for entire record

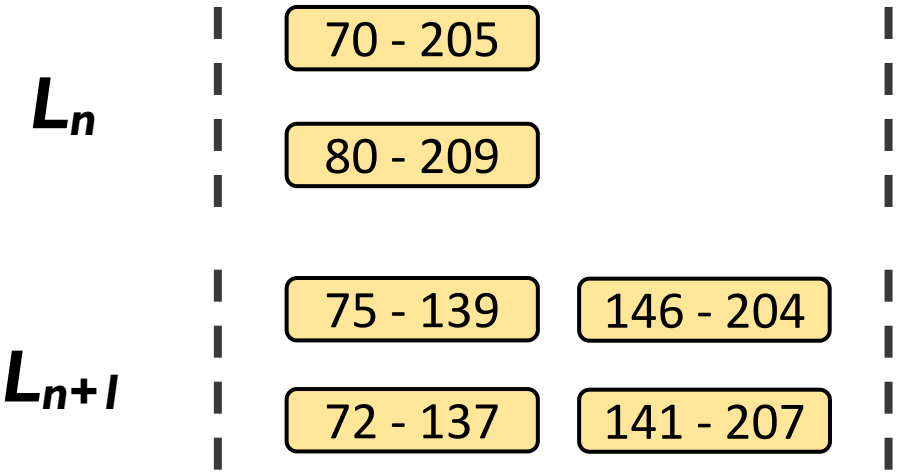
  e.g., SELECT * FROM table WHERE Major = Computer

Non-Index-Only Query Optimizations

1. Filtering components with sequence number (SEQ)

- Many LSM-trees adopt *tiering strategy* for compaction (also L0 in most of LSM-trees)
  - Multiple sorted runs per level; No rewriting SSTables in higher level
  - Smaller write amplification, but higher read amplification
- SEQ ranges of different sub-levels in the same key range are strictly divided
- Secondary query: searching PKey with a specific version (SEQ)
- Filters components with SEQ

E.g., searching PKey=100 with SEQ=150

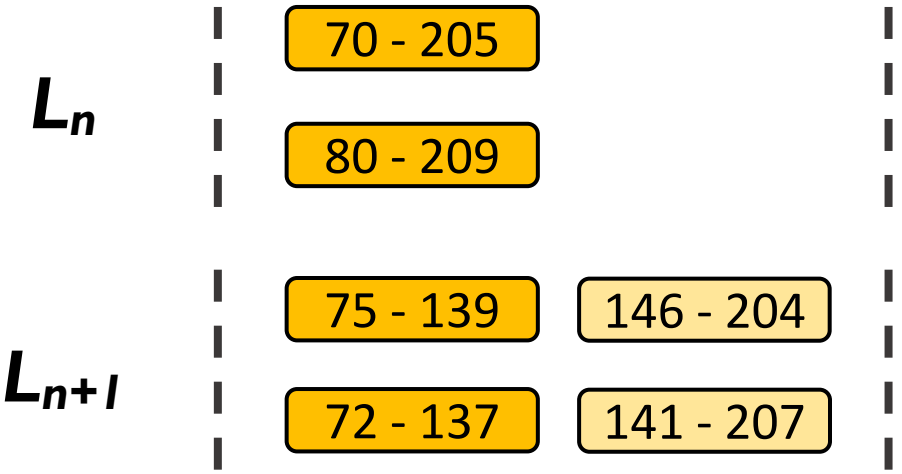


Non-Index-Only Query Optimizations

1. Filtering components with sequence number (SEQ)

- Many LSM-trees adopt *tiering strategy* for compaction (also L0 in most of LSM-trees)
  - Multiple sorted runs per level; No rewriting SSTables in higher level
  - Smaller write amplification, but higher read amplification
- SEQ ranges of different sub-levels in the same key range are strictly divided
- Secondary query: searching PKey with a specific version (SEQ)
- Filters components with SEQ

E.g., searching PKey=100 with SEQ=150

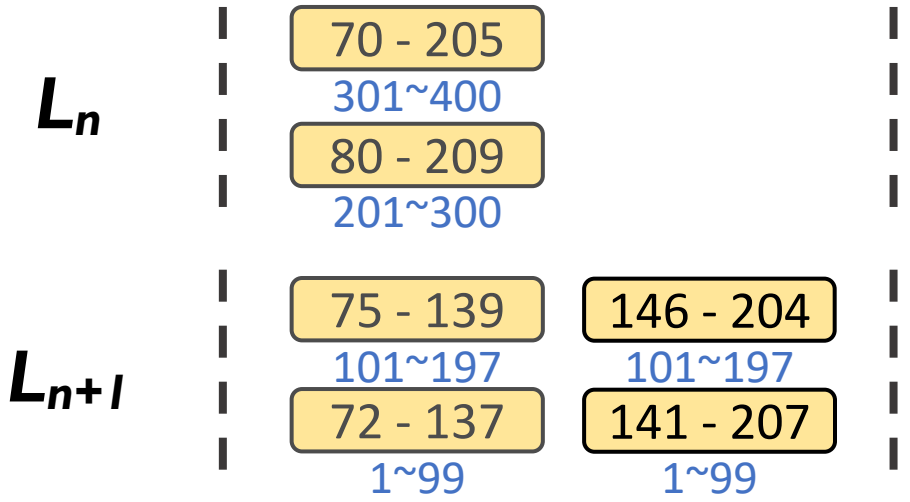


Non-Index-Only Query Optimizations

1. Filtering components with sequence number (SEQ)

- Many LSM-trees adopt *tiering strategy* for compaction (also L0 in most of LSM-trees)
  - Multiple sorted runs per level; No rewriting SSTables in higher level
  - Smaller write amplification, but higher read amplification
- SEQ ranges of different sub-levels in the same key range are strictly divided
- Secondary query: searching PKey with a specific version (SEQ)
- Filters components with SEQ

E.g., searching PKey=100 with SEQ=150



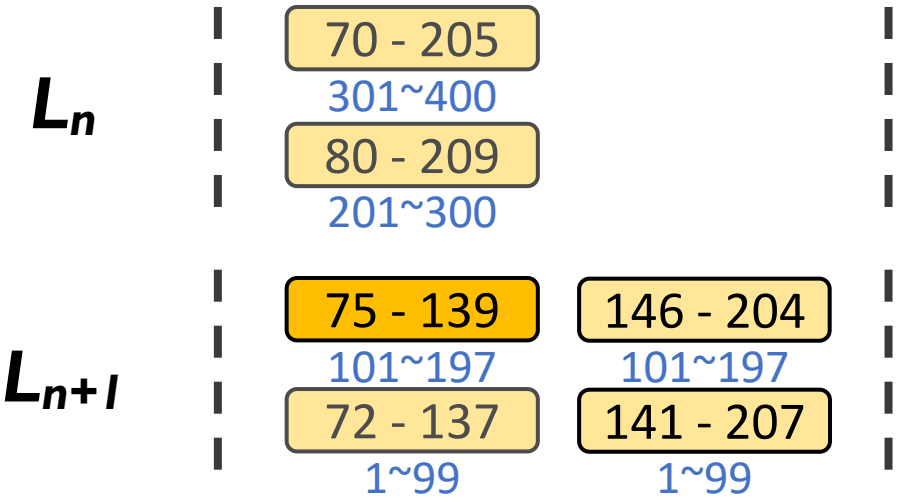
Non-Index-Only Query Optimizations

1. Filtering components with sequence number (SEQ)

- Many LSM-trees adopt *tiering strategy* for compaction (also L0 in most of LSM-trees)
  - Multiple sorted runs per level; No rewriting SSTables in higher level
  - Smaller write amplification, but higher read amplification
- SEQ ranges of different sub-levels in the same key range are strictly divided
- Secondary query: searching PKey with a specific version (SEQ)
- Filters components with SEQ

E.g., searching PKey=100 with SEQ=150

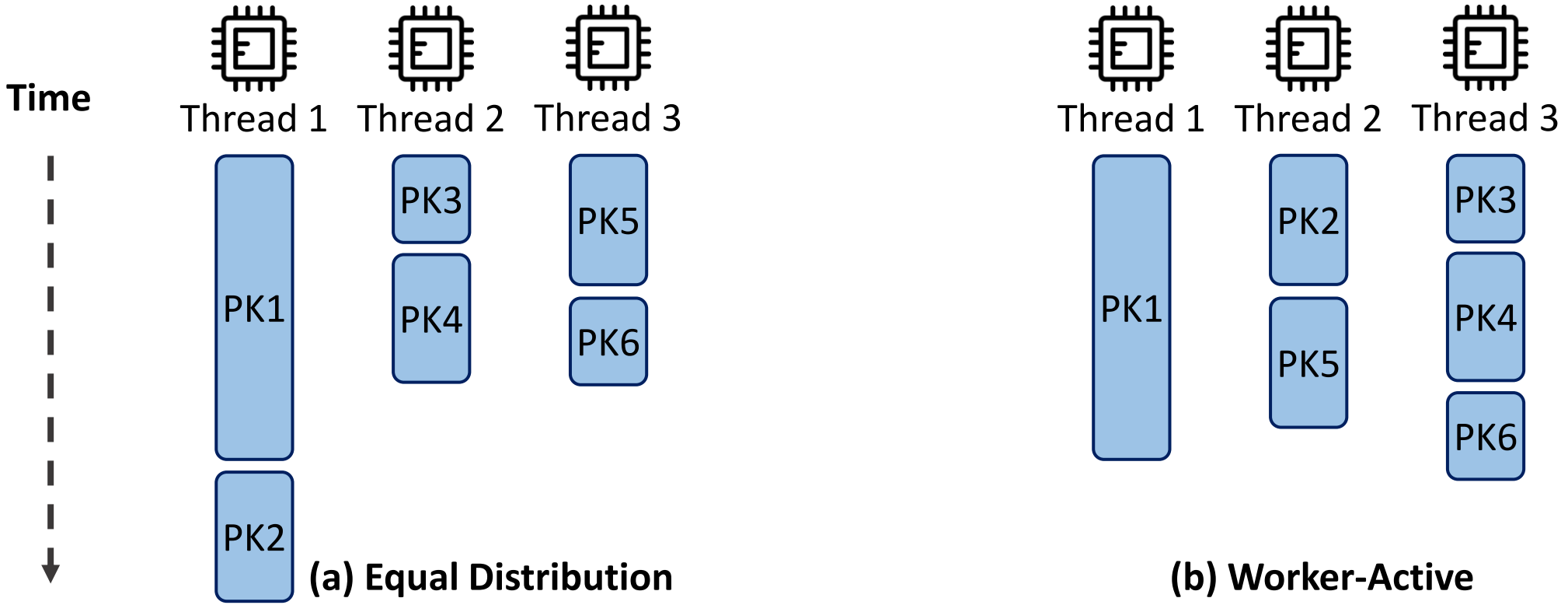
Reduce most component probing overhead with tiering strategy



Non-Index-Only Query Optimizations

2. Parallel Primary Table Searching

- Searching a key in LSM can have varied latencies
  - Simply assigning tasks evenly results in load imbalance
  - *worker-active* scheme: workers fetch tasks when they are idle



More Design Details : Check Our Paper

Garbage Collection

- GC of PKey Entries in PS-Tree and validation hash table

Data Consistency with LSM primary table

- MVCC with SEQ

Crash Consistency

- using WAL for atomic durability among the primary table, PS-Tree, and validation hash table
- Recovery of PS-Tree and validation hash table

Outline

- Background & Motivation
- Perseid – a PM-based secondary indexing mechanism
- Evaluation Results
- Conclusion

Experimental Setup

Hardware Platform

CPU	18-core Intel Xeon Gold 5220 CPU
PM	2 * 128 GB Intel Optane DC PMMs
DRAM	64 GB DDR4 DIMMs
SSD	480 GB Intel Optane 905P

Compared Systems

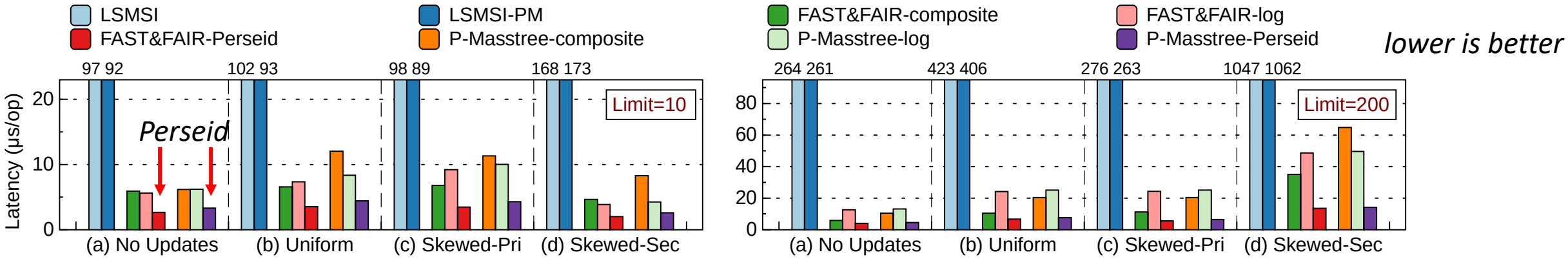
- LevelDB++ [SIGMOD'18, VLDB'19] (LSM-based secondary index, on {SSD, PM})
- PM indexes: { FAST&FAIR, P-Masstree } with { composite index, log-structured }
- LSM primary table: PebblesDB (tiering), LevelDB (leveling)

Workloads

- Twitter-like workload generator for secondary indexing
- 100 million primary keys, 4 million secondary keys, record size 1KB

Evaluation Results

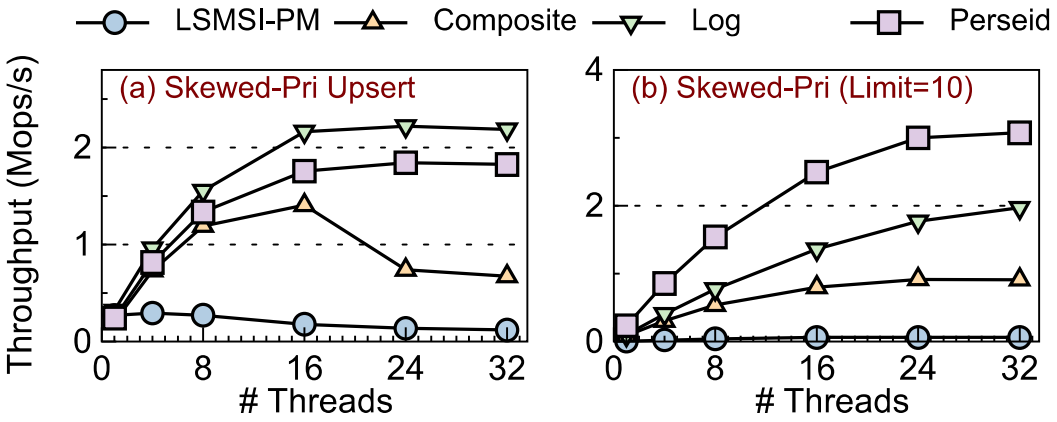
- Index-only query, 1 thread
- Limit = 10 / 200 (max number of results per query)
- Queries after:
  (a) loading, no updates; (b) uniform updates; (c) skewed primary keys (Zipfian 0.99);
  (d) skewed secondary keys (Zipfian 0.99)



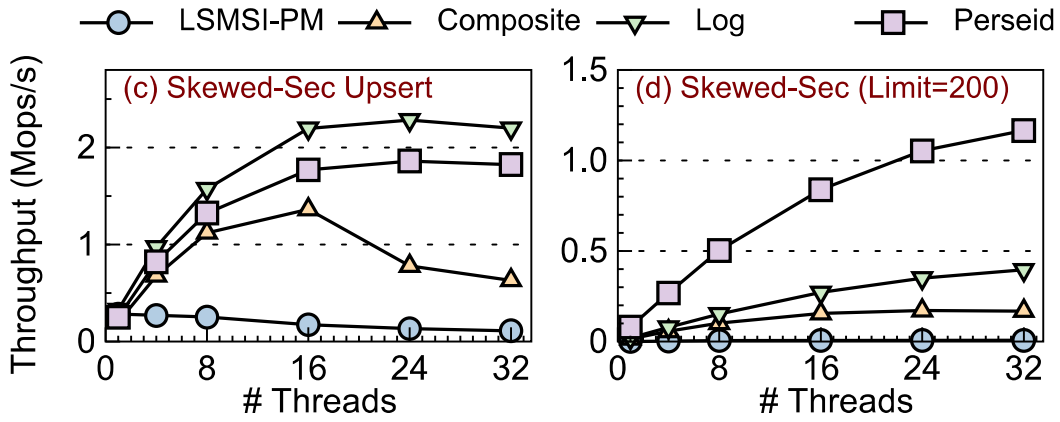
- LSMSI is quite inefficient for queries, even if on PM
- Perseid outperforms existing PM indexes by up to 4.5×

Evaluation Results

- Multi-threaded upsert and query performance



Skewed Primary Key



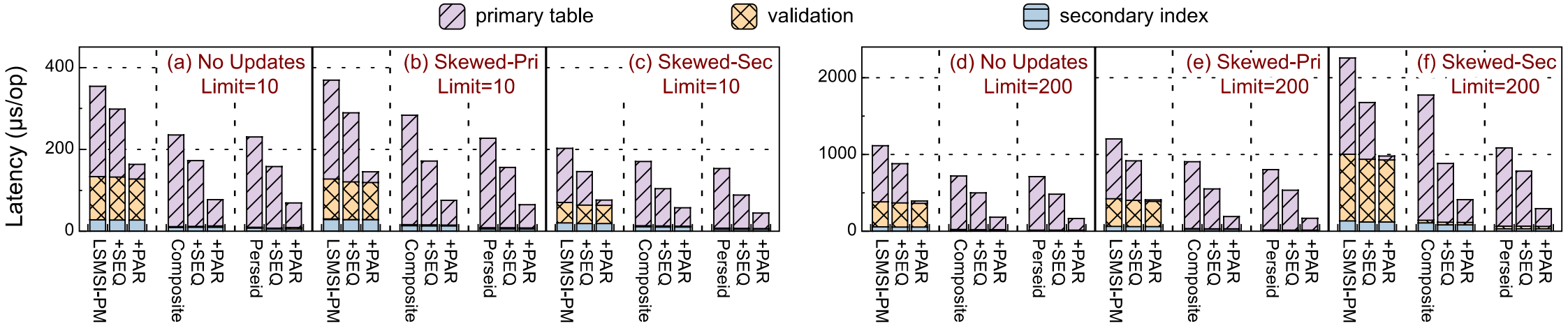
Skewed Secondary Key

Perseid:

- has better scalability
- achieves 3-7× query performance of other PM indexes
- has comparable upsert performance as log-structured approach

Evaluation Results

- Non-index-only query performance and breakdown
- Apply SEQ filter (+SEQ), parallel primary table searching (+PAR, 4 threads) sequentially



- Perseid outperforms LSMSI by up to 2.3×
- Our optimizations on primary table searching have significant effect, by up to 3.1×
- Entries of a SKey in Perseid are sorted by *recency (less likely to be invalid, more efficient for query)*, but by PKey in composite index

Conclusion

- We analyze the inefficiencies of LSM-based secondary indexing and existing PM-based general indexes as secondary indexes.
- PM is suitable for low-latency-required query operations, but still needs specific design to fully take advantage of it.
- We propose Perseid, an efficient PM-based secondary indexing mechanism for LSM-based storage engines.
- More evaluation results and analysis are in the paper

Thanks & QA

Revisiting Secondary Indexing in LSM-based Storage Systems with Persistent Memory

Open-source code: https://github.com/thustorage/perseid

Contact Information: wangjing19@mails.tsinghua.edu.cn
Thanks to *Deng Feng Fund*