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

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

Department of Computer Science and Technology, Tsinghua University and Beijing National Research Center for Information Science and Technology (BNRist)

In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023).

2024. 09. 11

Presentation by Yeongyu Choi

choiyg@dankook.ac.kr

Contents

Introduction

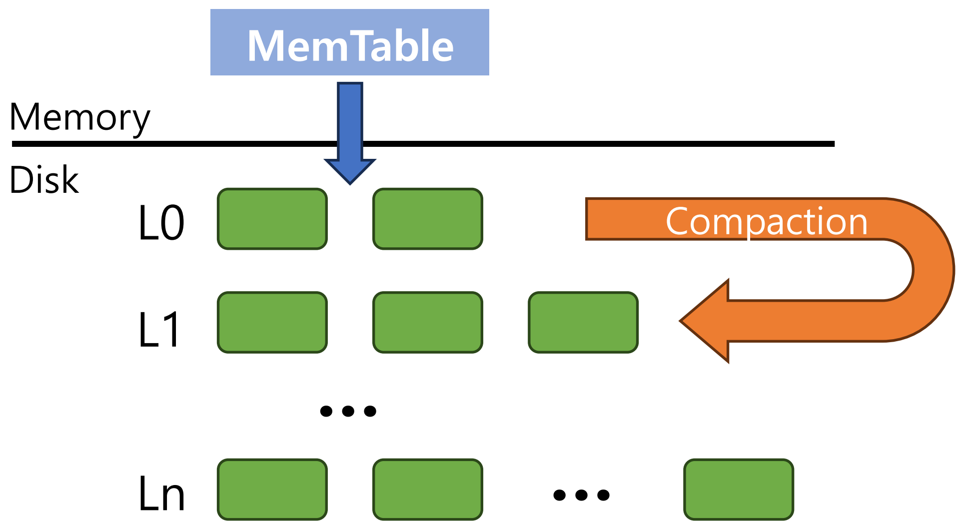
RocksDB

levelDB

cassandra

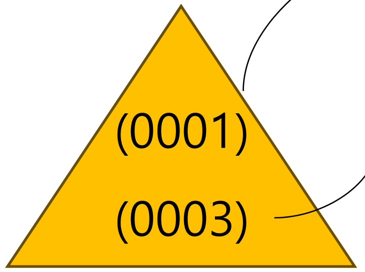
etc…

LSM-tree



- **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

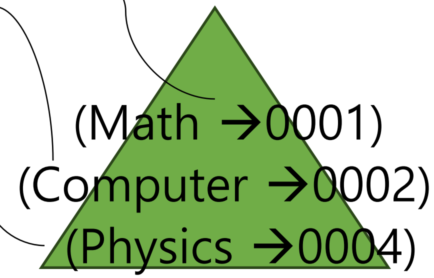
Background: Secondary Index



Primary Index

StudentID	Name	Major	EnrollmentYear	...
0001	가단국	Computer	2021	
0002	나단국	Math	2021	
0003	다단국	Music	2022	
0004	라단국	Physics	2022	
0005	마단국	Computer	2023	

Primary Table



Secondary Index

- **Primary Index(StudentID)**: Indexed by **primary key**
- 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 system

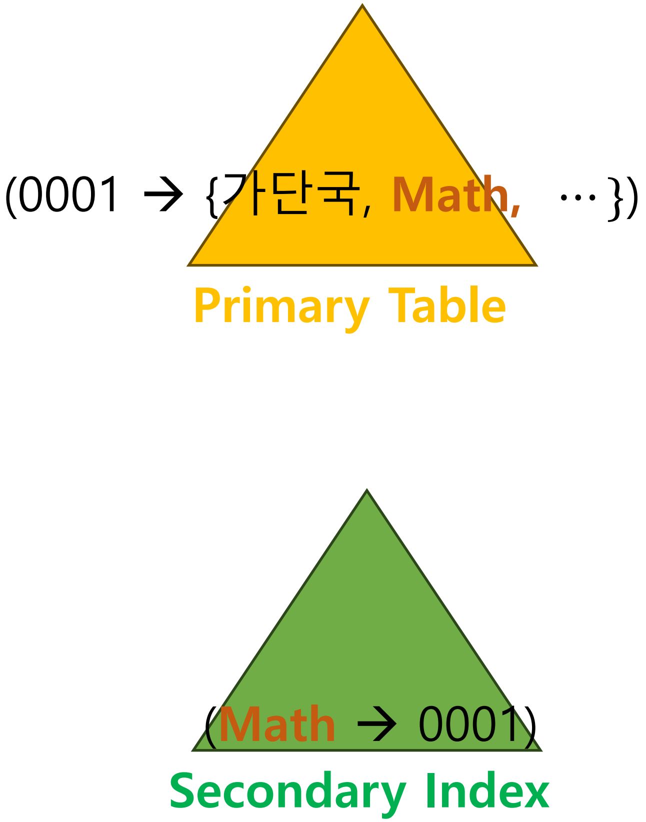
Motivation: Secondary Index

- Secondary indexing is inefficient with LSM-tree

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

E.g., update 가단국(0001)'s major **Math** → **Computer**
PUT: {0001→ 가단국, **Computer**, …} in LSM-tree

In secondary Index:
1. Insert new entry {**Computer** → 0001}



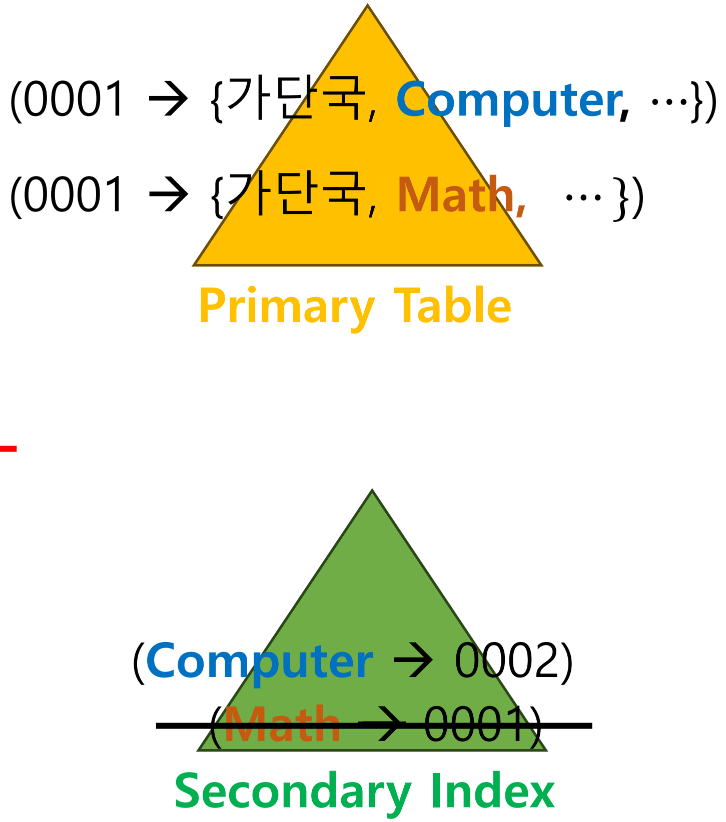
Motivation: Secondary Index

- Secondary indexing is inefficient with LSM-tree

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

E.g., update 가단국(0001)'s major **Math** → **Computer**
PUT: {0001→ 가단국, **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**

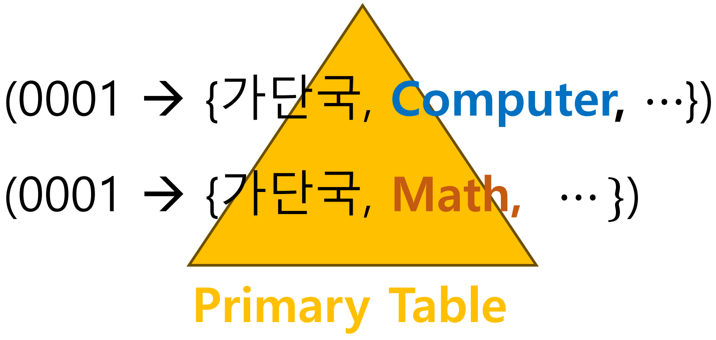
Motivation: Secondary Index

- Secondary indexing is inefficient with LSM-tree

1. Consistency among indexes are troublesome due to blind-write

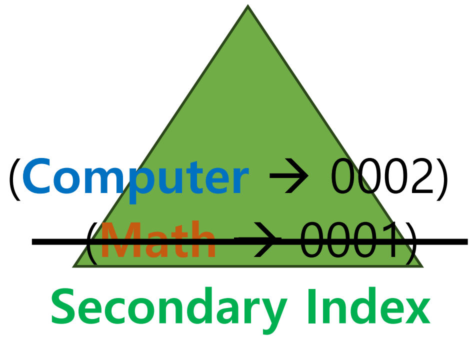
E.g., update 가단국(0001)'s major **Math** → **Computer**
PUT: {0001→ 가단국, **Computer**, …} in LSM-tree

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**

- **Synchronous**
  - READ old record to get old secondary key **Math** and then delete in secondary index
    → BUT, discard blind-write, low write performance
- **Validation**
  - Keep old entry {**Math** → 0001}, but at query, fetch record of '0001' in primary table for validation
    → BUT, low query performance



Motivation: Secondary Index

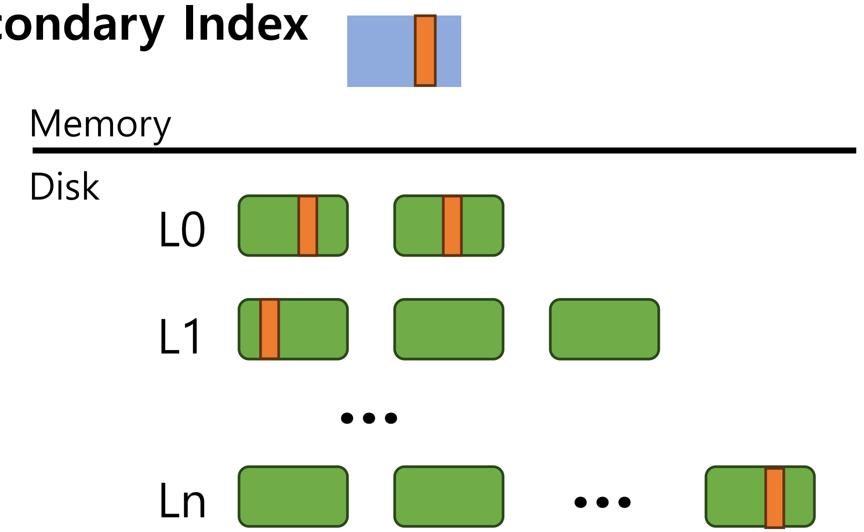
- Secondary indexing is inefficient with LSM-tree

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)
- **LSM-tree:**
  - Disk & Block based
  - Multi-level

Mismatch!

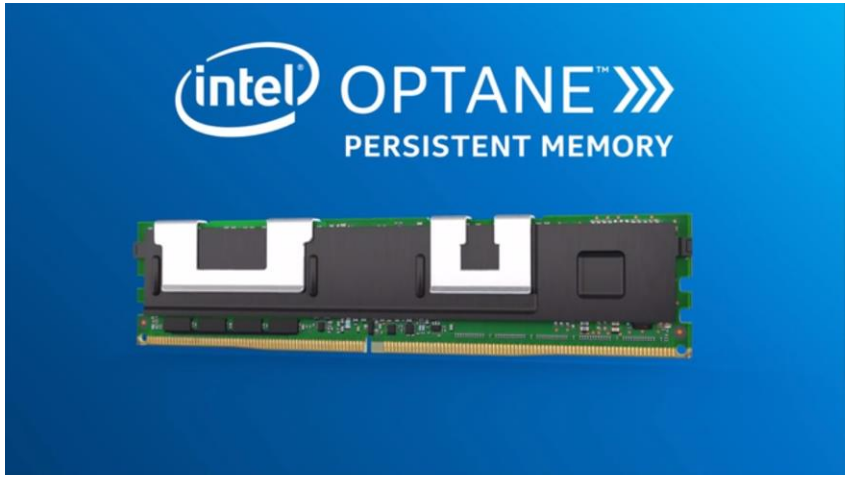
- **LSM Secondary Index**



Attributes of secondary indexes
and
LSM-tree
are **mismatched**

Background: Persistent Memory

- Using Persistent Memory(PM) for secondary indexing is promising
  - Byte-addressability
  - DRAM comparable latency
  - Data persistency



- PM-based indexes

wB+Trees[VLDB'15]	FPTree[SIGMOD'16]	WORT[FAST'17]	FAST&FAIR[FAST'18]
Recipe[SOSP'19]	LB+Tress[VLDB'20]	DPTree[VLDB'20]	ROART[FAST'21]
Nap[OSDI'21]	TIPS[ATC'21]	PACTree[SOSP'21]	NBTree[VLDB'22]

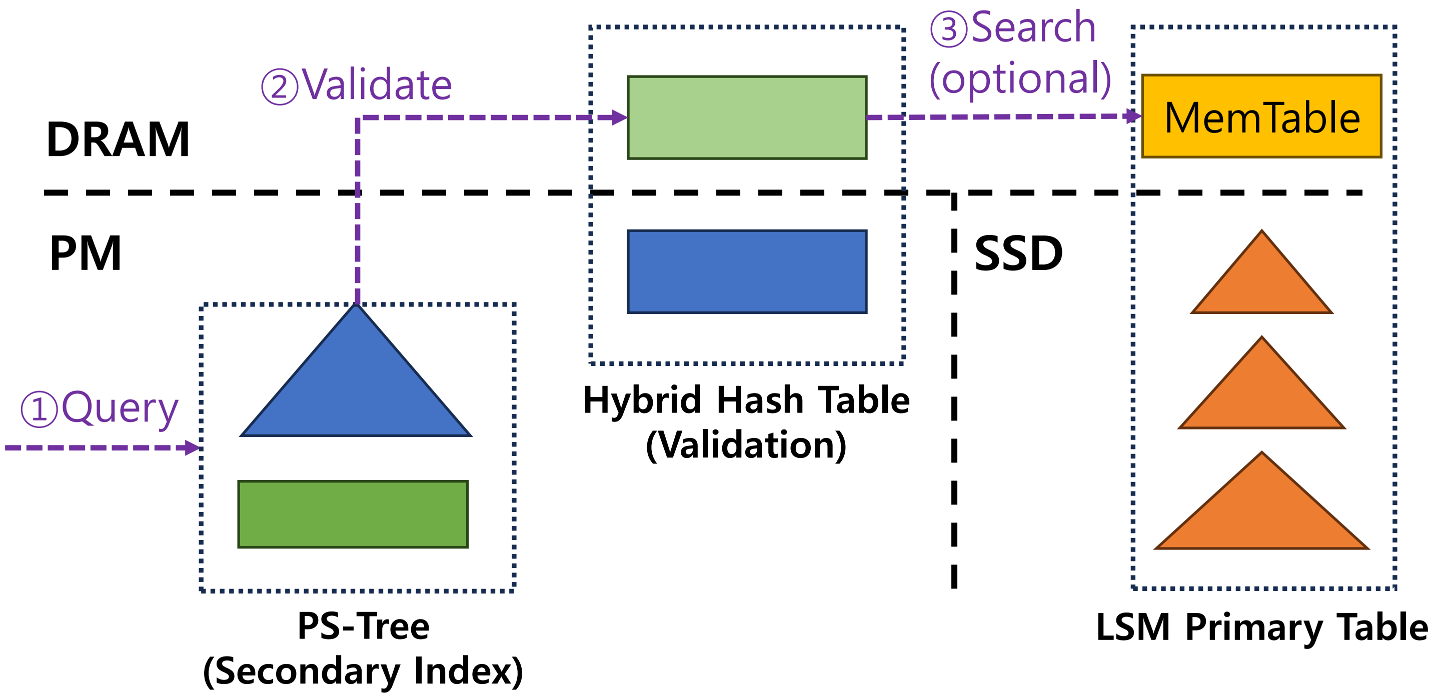
…

Motivation: PM-based Indexes for Secondary Index

- **Directly adopting existing PM indexes for secondary indexing is inefficient**
- How to handle the feature of **non-unique** ? (**Computer** → {A, B, E, …})
  - Allocate 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 allocate 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
  - Expanding the number of KV pairs → larger index → **degraded performance**

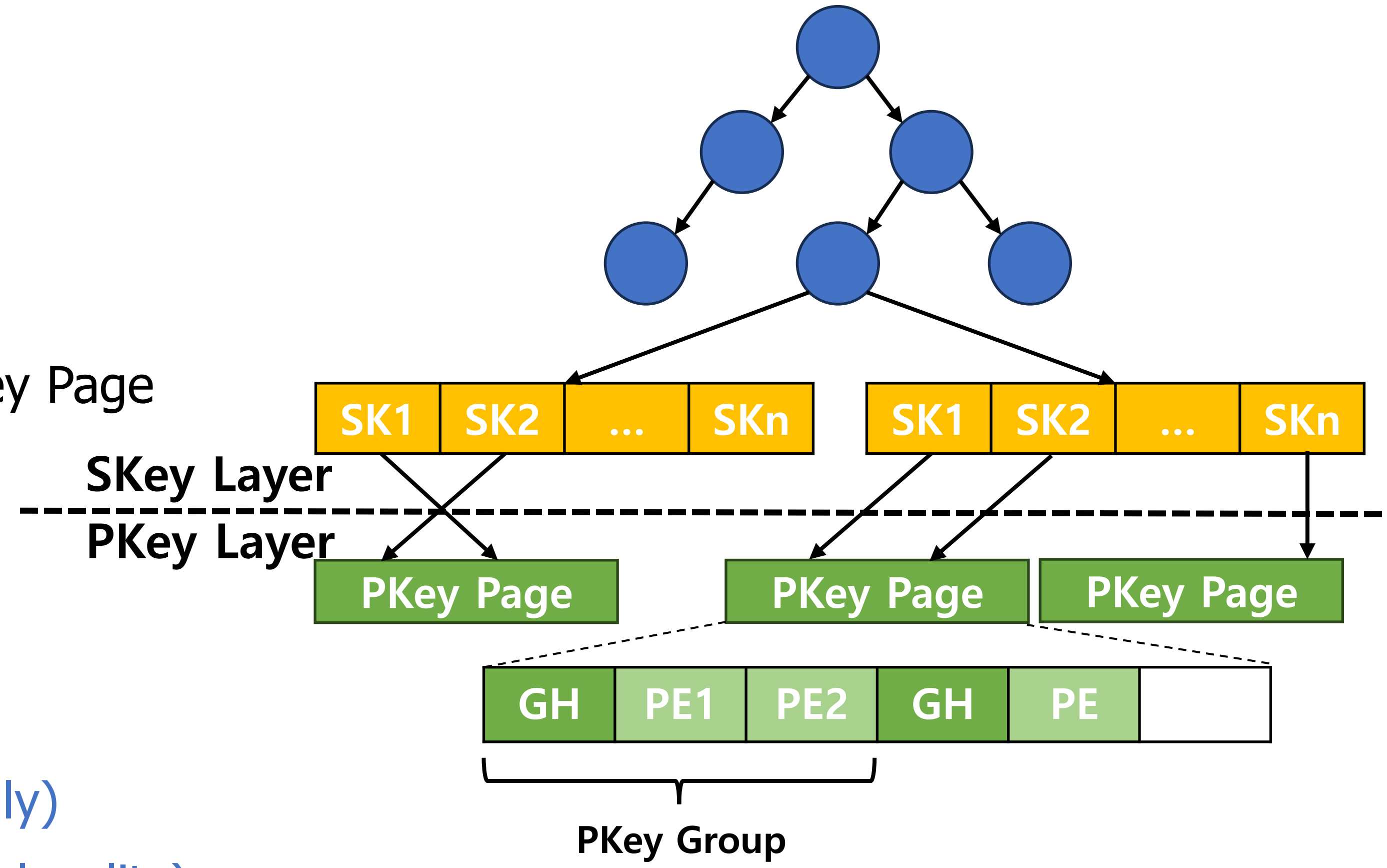
PERSEID: Overview

- **PS-Tree**
  - 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



PERSEID: PS-Tree

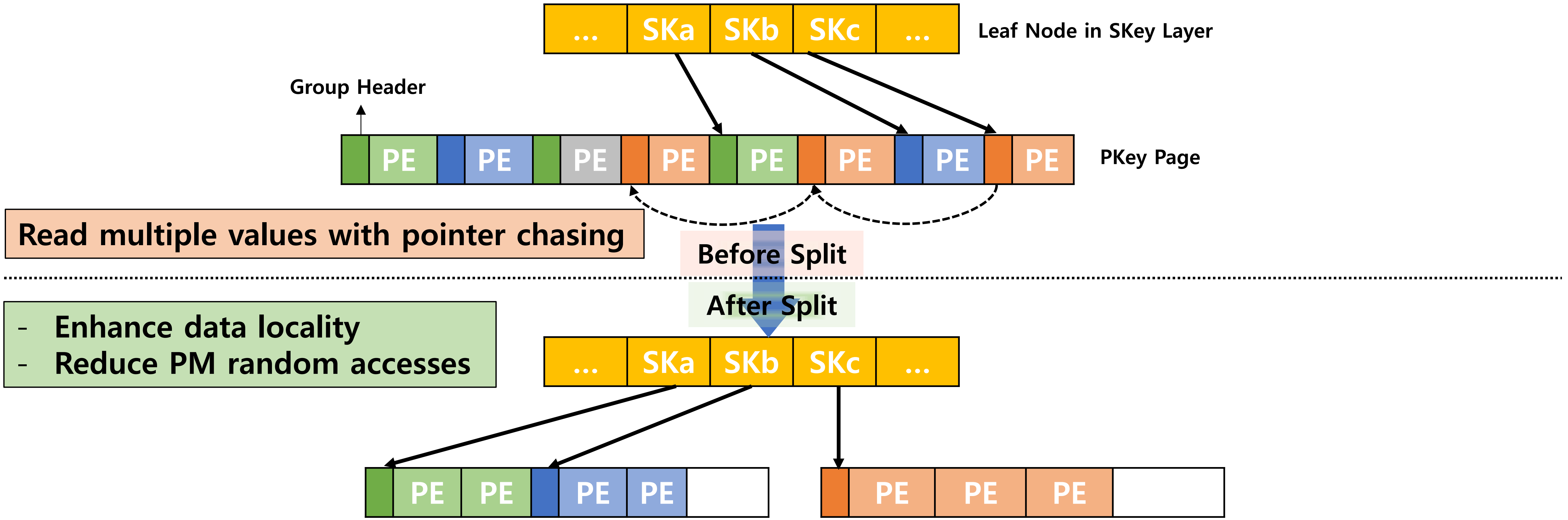
- **PM-based secondary index**
- **SKey Layer**
  - Index for secondary key to values in PKey Page
  - Leverage existing PM index
- **PKey Layer**
  - Store multiple values for Skeys
  - Append entries in PKey Pages (PM friendly)
  - Adjacent SKeys share PKeys Pages (data locality)
  - Rearrange entries 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

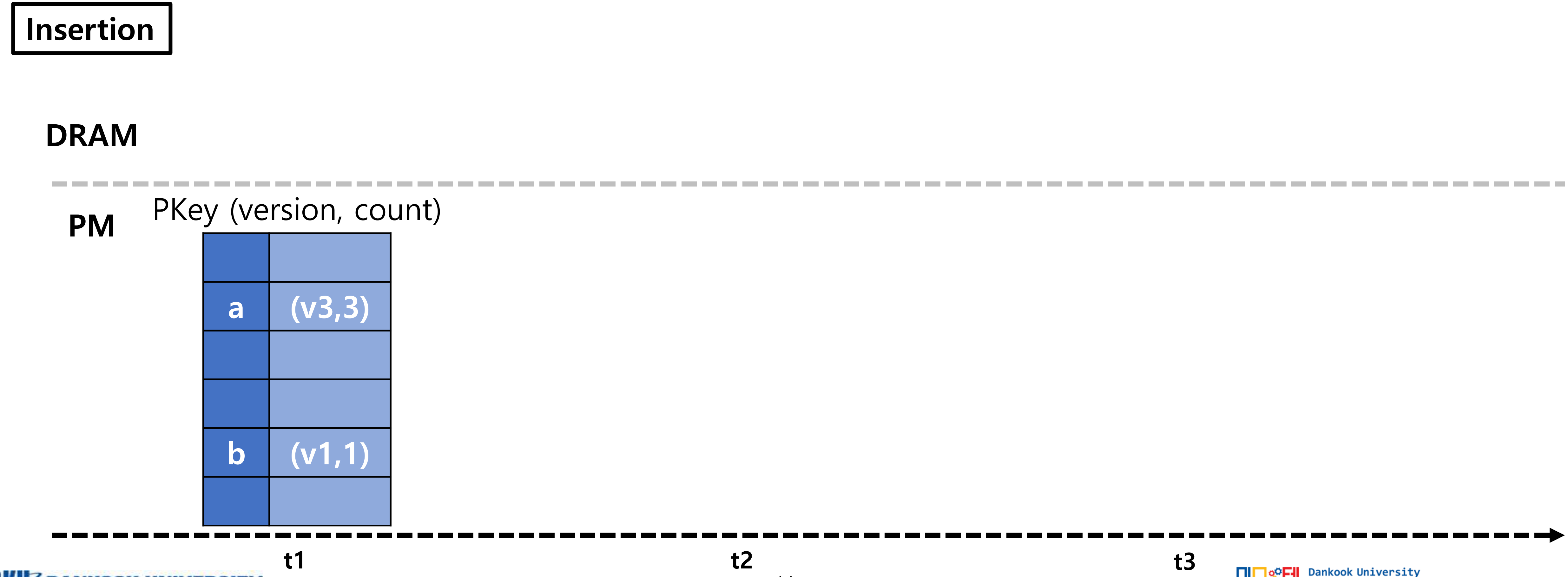
PERSEID: PS-Tree

- Rearrangement and garbage collection at splitting



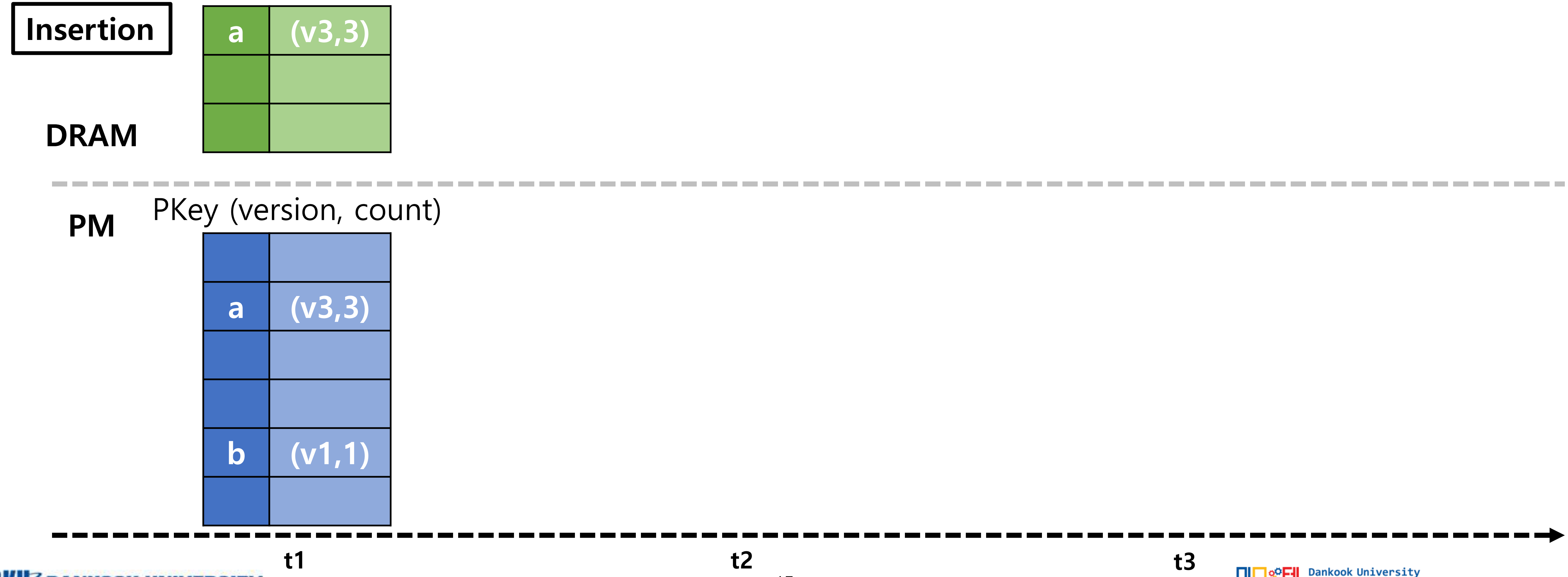
PERSEID: Hybrid PM-DRAM Hash-based Validation

- 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



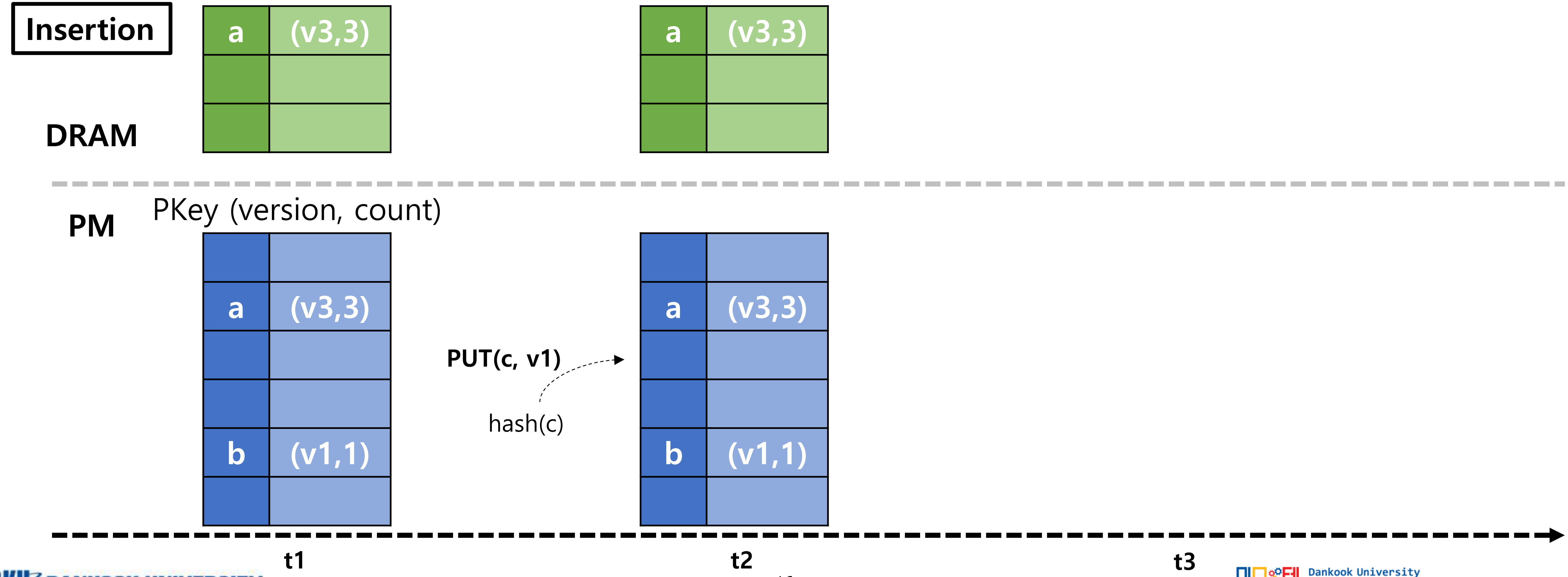
PERSEID: Hybrid PM-DRAM Hash-based Validation

- 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



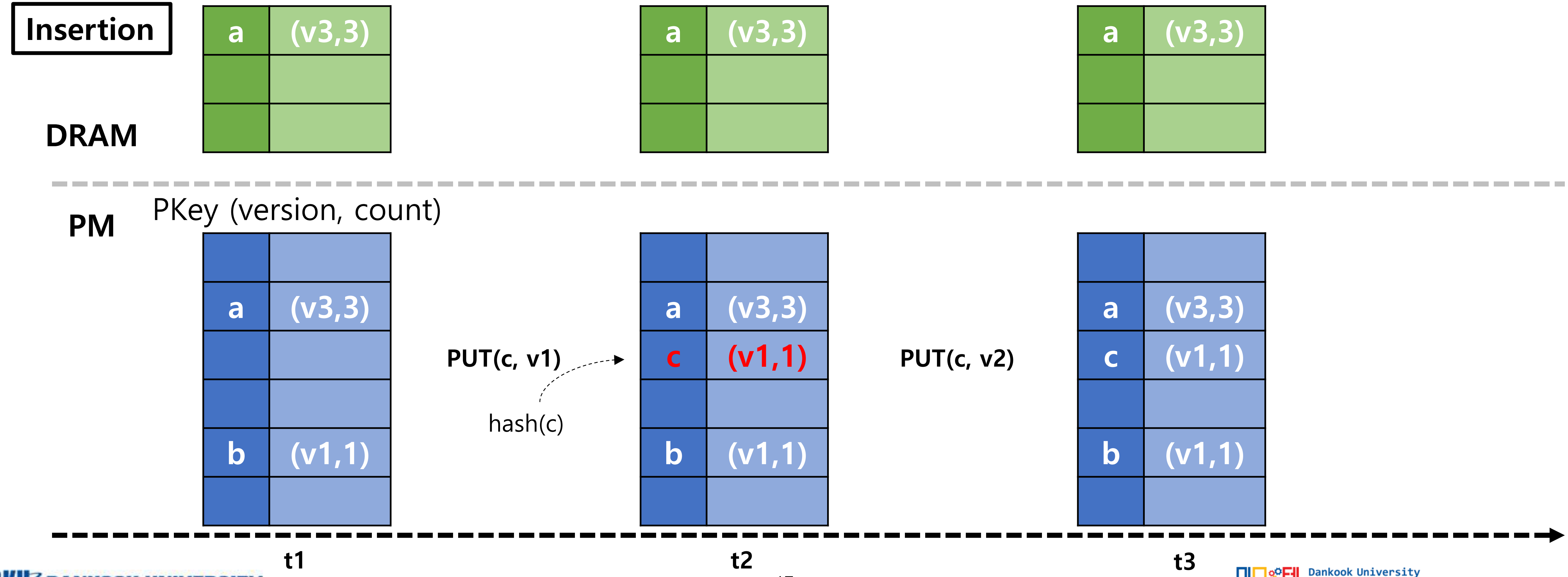
PERSEID: Hybrid PM-DRAM Hash-based Validation

- 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



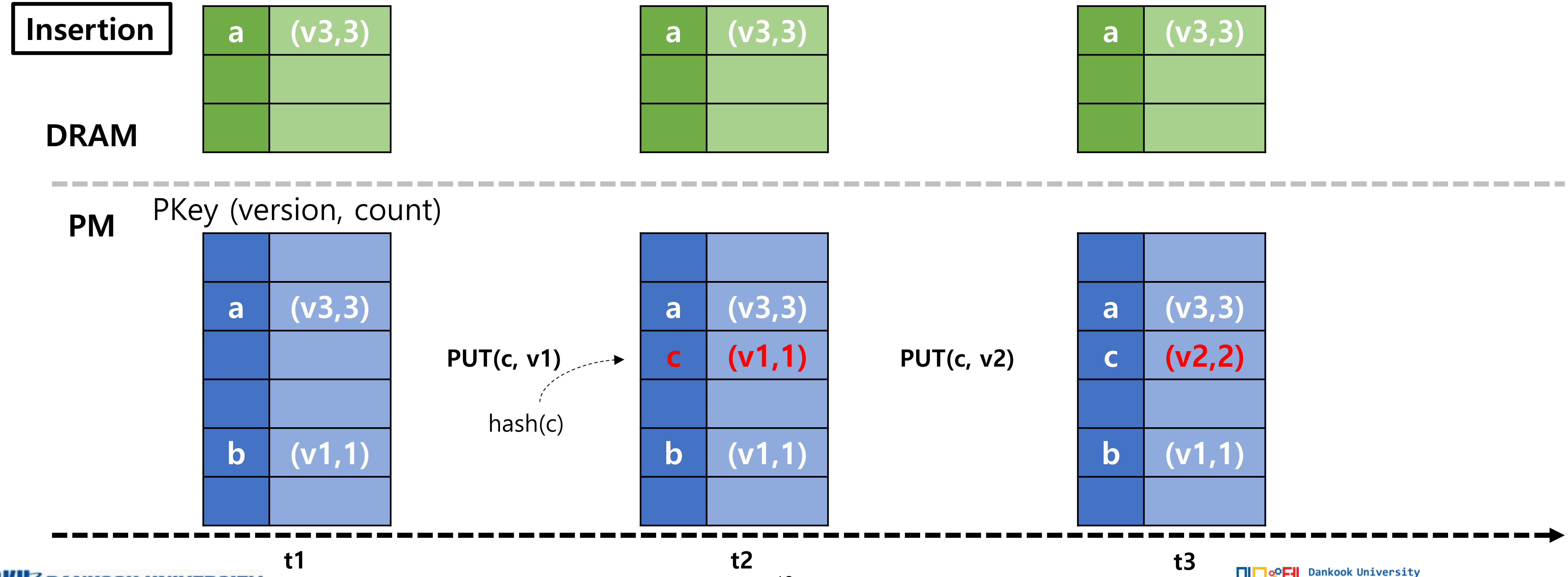
PERSEID: Hybrid PM-DRAM Hash-based Validation

- 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



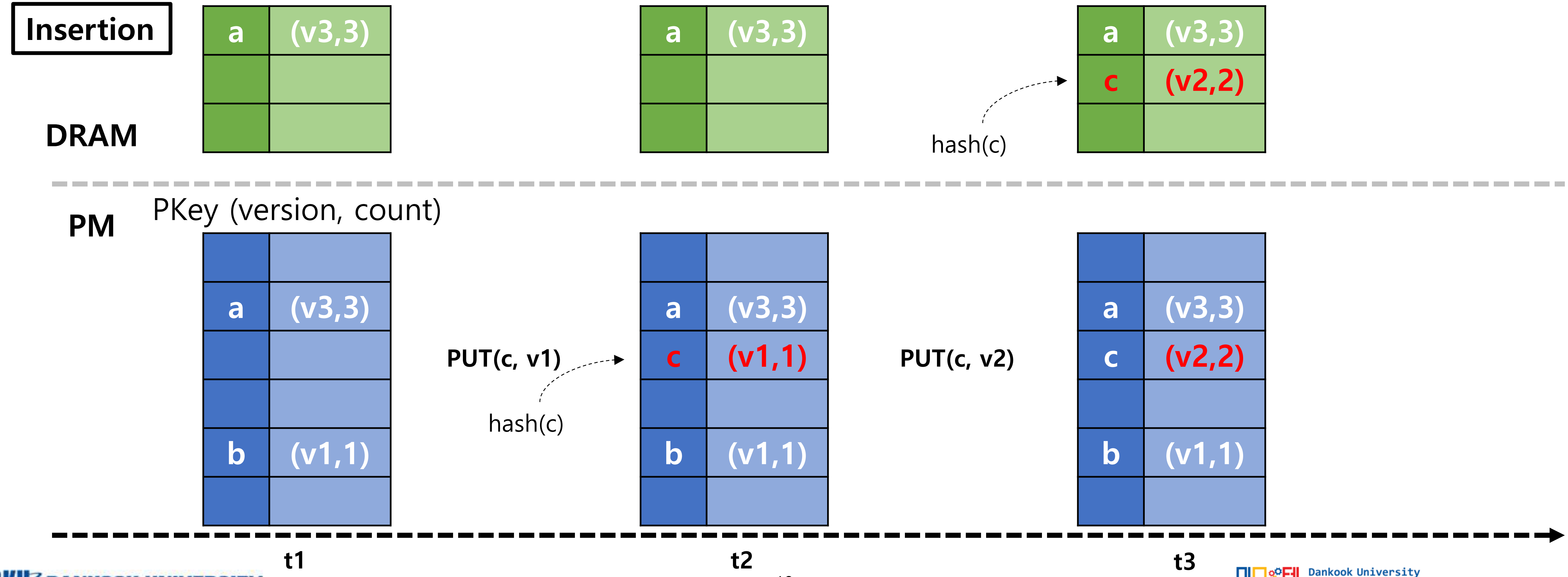
PERSEID: Hybrid PM-DRAM Hash-based Validation

- 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



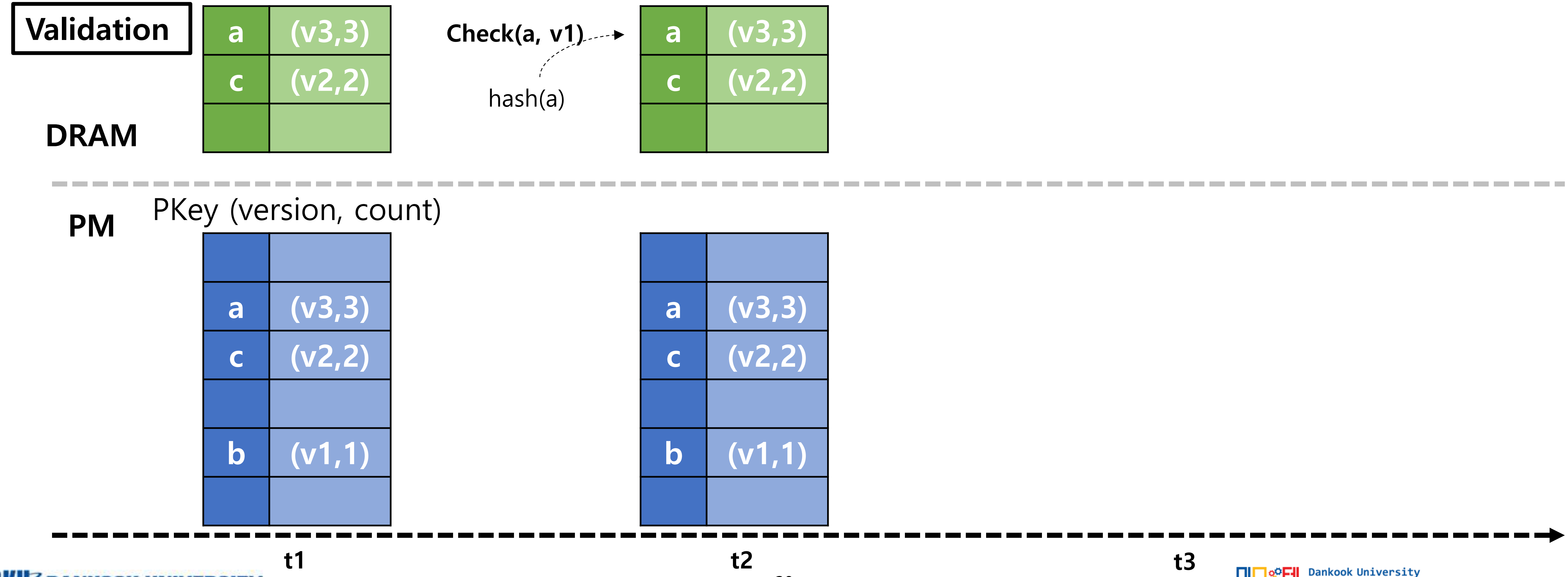
PERSEID: Hybrid PM-DRAM Hash-based Validation

- 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



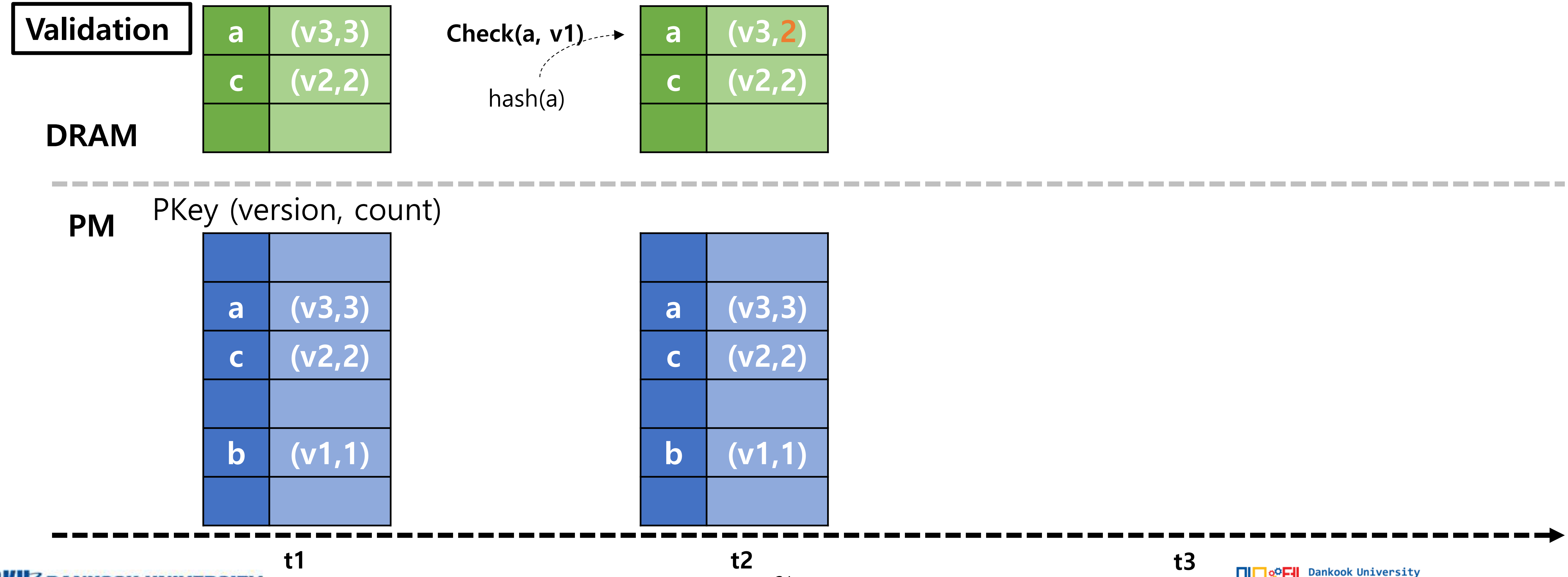
PERSEID: Hybrid PM-DRAM Hash-based Validation

- 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



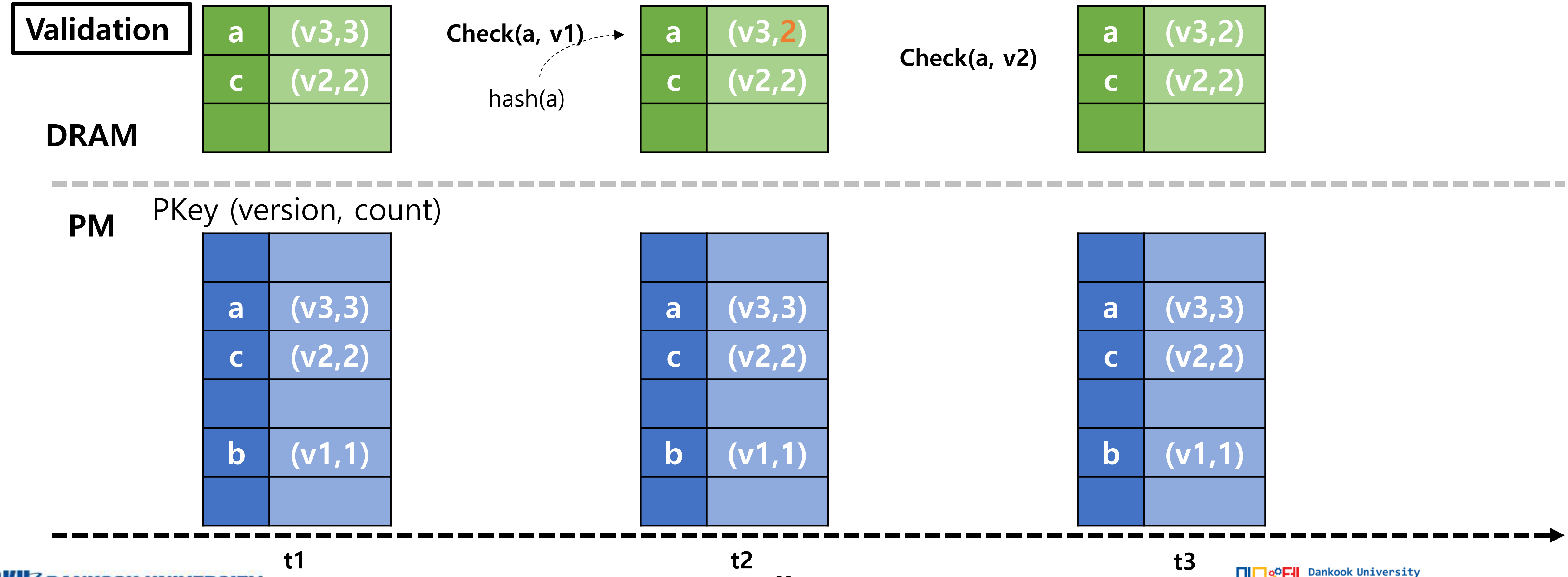
PERSEID: Hybrid PM-DRAM Hash-based Validation

- 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



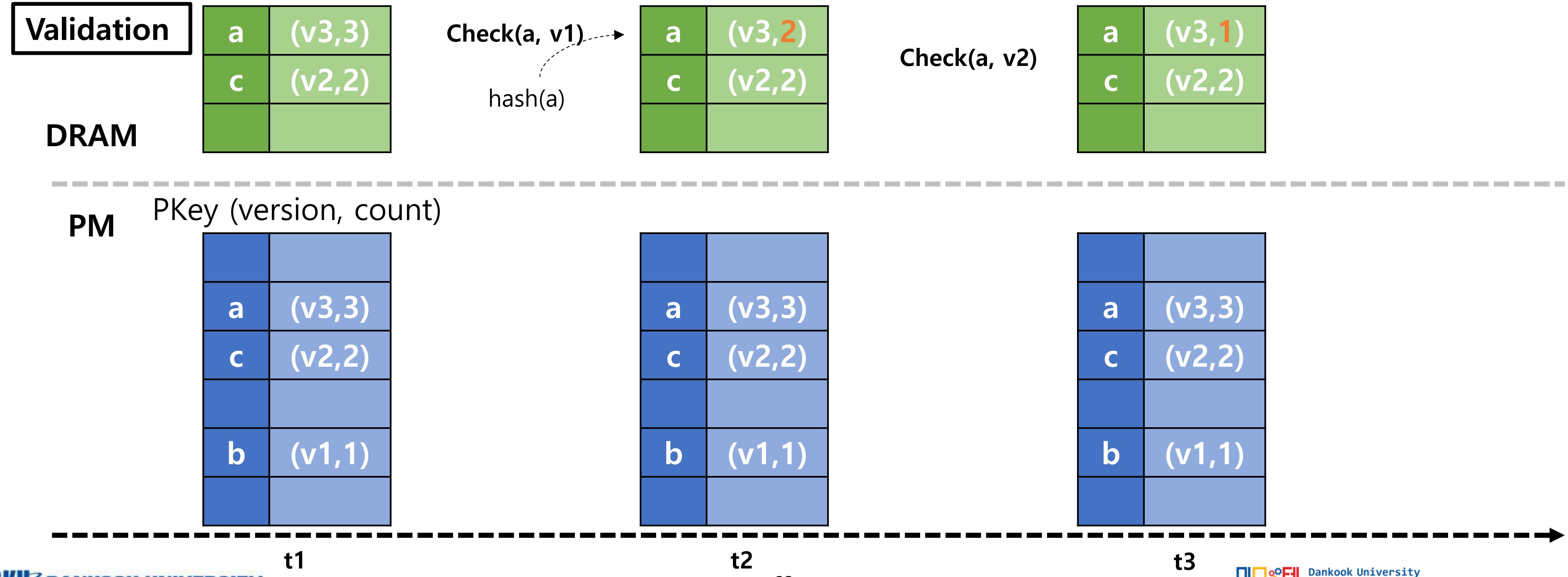
PERSEID: Hybrid PM-DRAM Hash-based Validation

- 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



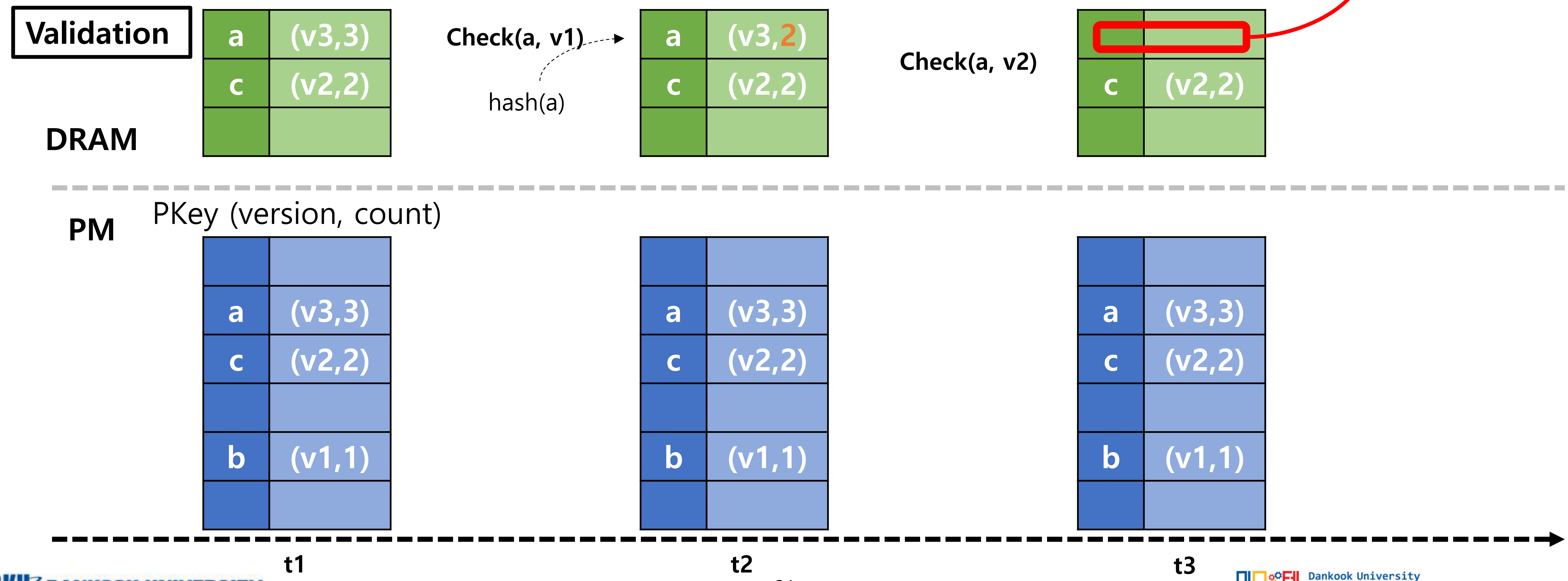
PERSEID: Hybrid PM-DRAM Hash-based Validation

- 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



PERSEID: 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



PERSEID: Non-Index-Only Query Optimization

- 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.,

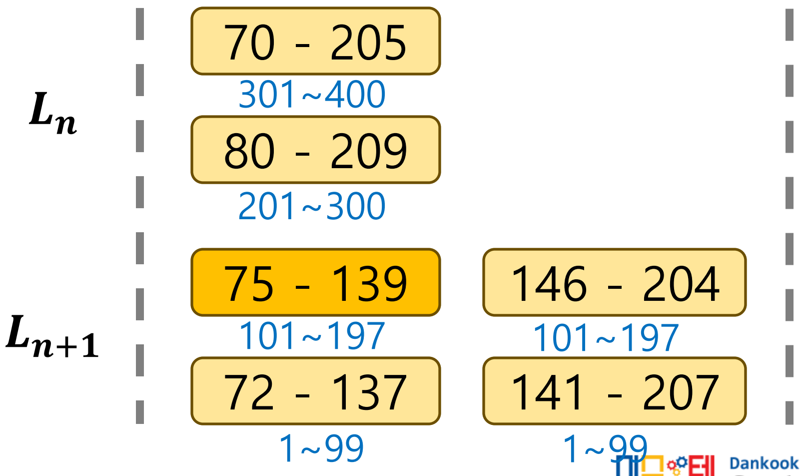
    SELECT * FROM table WHERE Major = Computer

PERSEID: Non-Index-Only Query Optimization

- **1. Filtering components with sequence number(SEQ)**
  - Many LSM-trees adopt <u>tiering strategy for compaction</u> (also L0 in most of LSM-trees)
    - Multiple sorted runs per level; No rewriting SSTables in higher level
    - Small 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

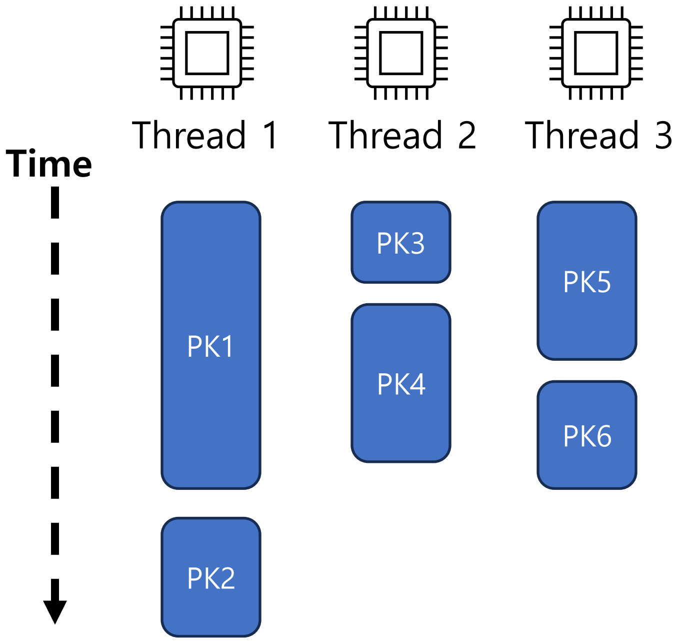
E.g., searching PKey=100 with SEQ=150

Reduce most component probing overhead with tiering strategy



PERSEID: Non-Index-Only Query Optimization

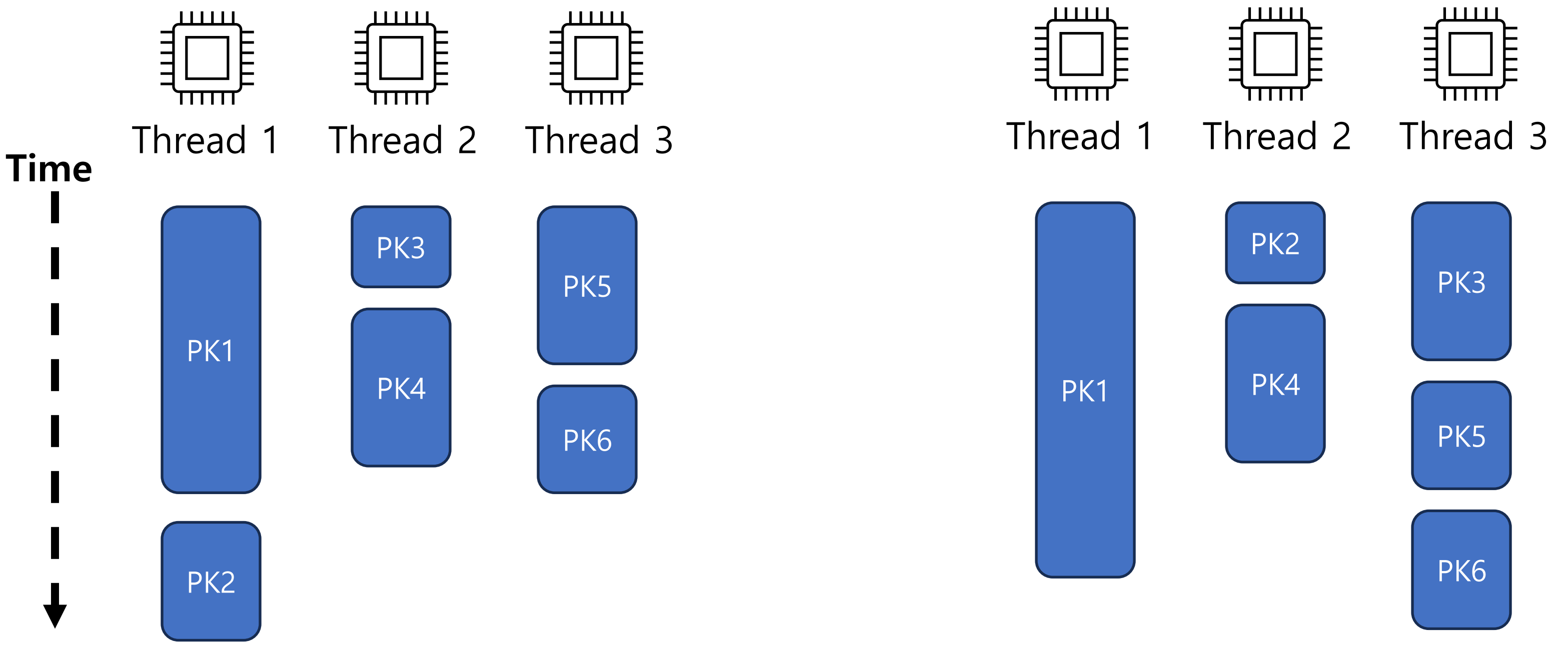
- **2. Parallel Primary Table Searching(PAR)**
  - Searching a key in LSM can have varied latencies
  - Simply assigning tasks evenly results in load imbalance



(a) Equal Distribution

PERSEID: Non-Index-Only Query Optimization

- **2. Parallel Primary Table Searching(PAR)**
  - 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



(a) Equal Distribution

(b) Worker-Active

Evaluation: Experiment Setup

- **Hardware Platform**

CPU	18-core Intel Xeon Gold 5220 CPU
PM	2 * 128GB Intel Optane DC PMMs
DRAM	64GB DDR4 DIMMs
SSD	480GB 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
  - 100M primary keys, 4M secondary keys, record size 1KB

Evaluation: Results

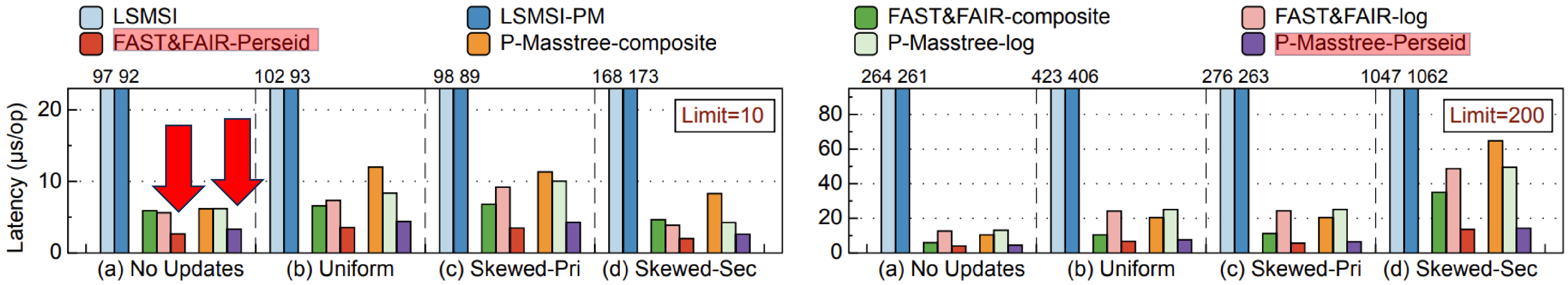


Figure 7: Index-only query performance.

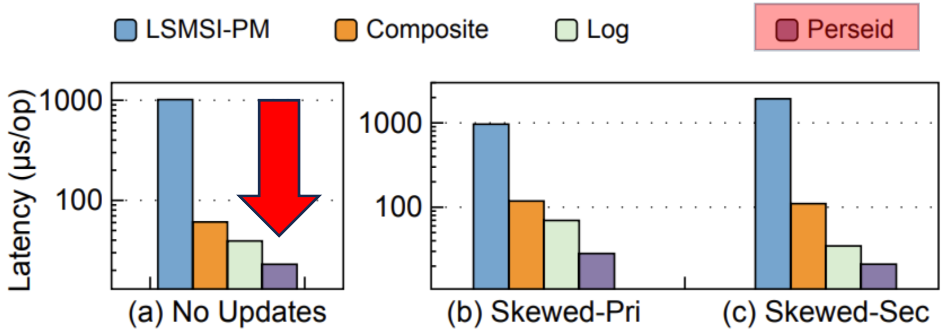


Figure 8: Index-only range query performance.

PERSEID outperforms existing PM indexes by up to **4.5x**
→ Using PERSEID with PM makes to enhance data locality, effectively manage the overhead of the multi-level structure

Evaluation: Results

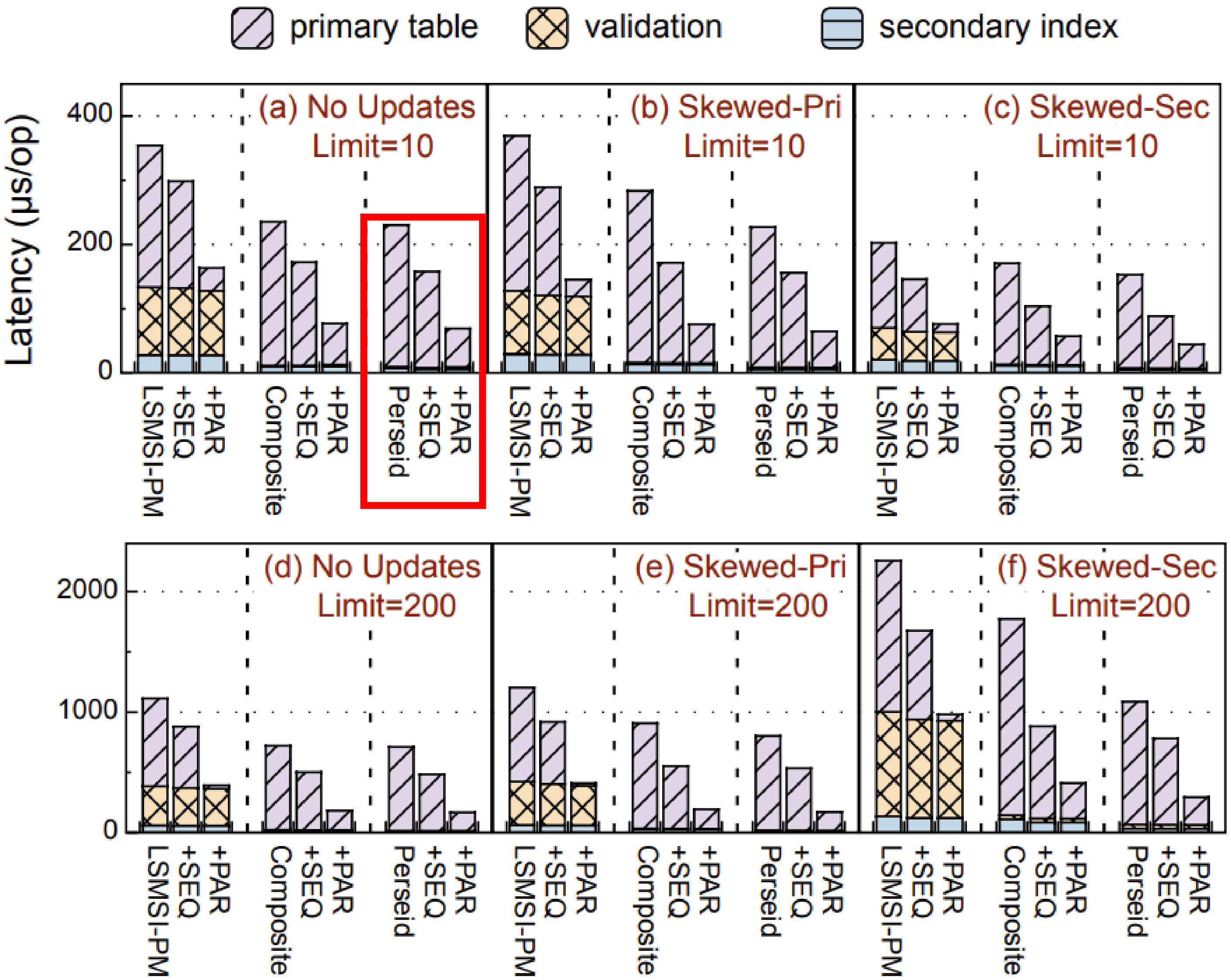


Figure 10: Non-index-only query performance.
The primary table time on +PAR only shows the time not covered by other parts.

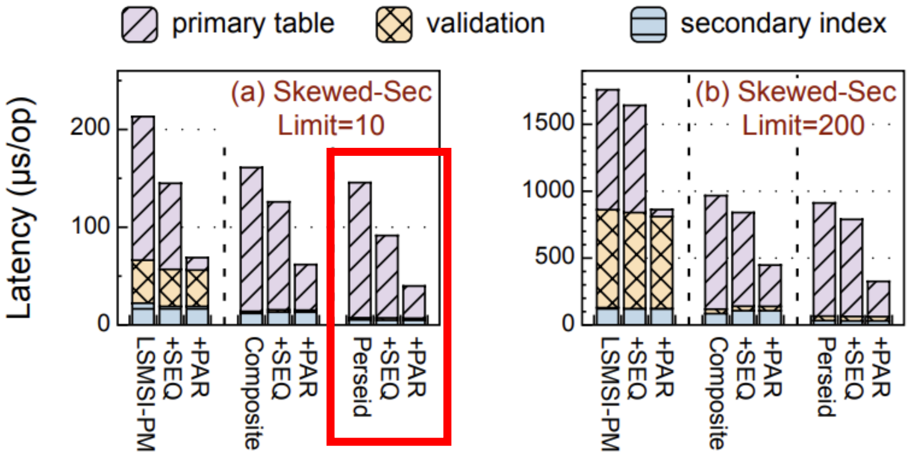


Figure 11: Non-index-only query performance on Leveling-based LSM table.

- PERSEID outperforms LSMSI by up to **2.3x**
- Our optimizations on primary table searching have significant effect, by up to **3.1x**
- Entries of a Skey in PERSEID are sorted by recency , but by Pkey in composite index

Evaluation: Results

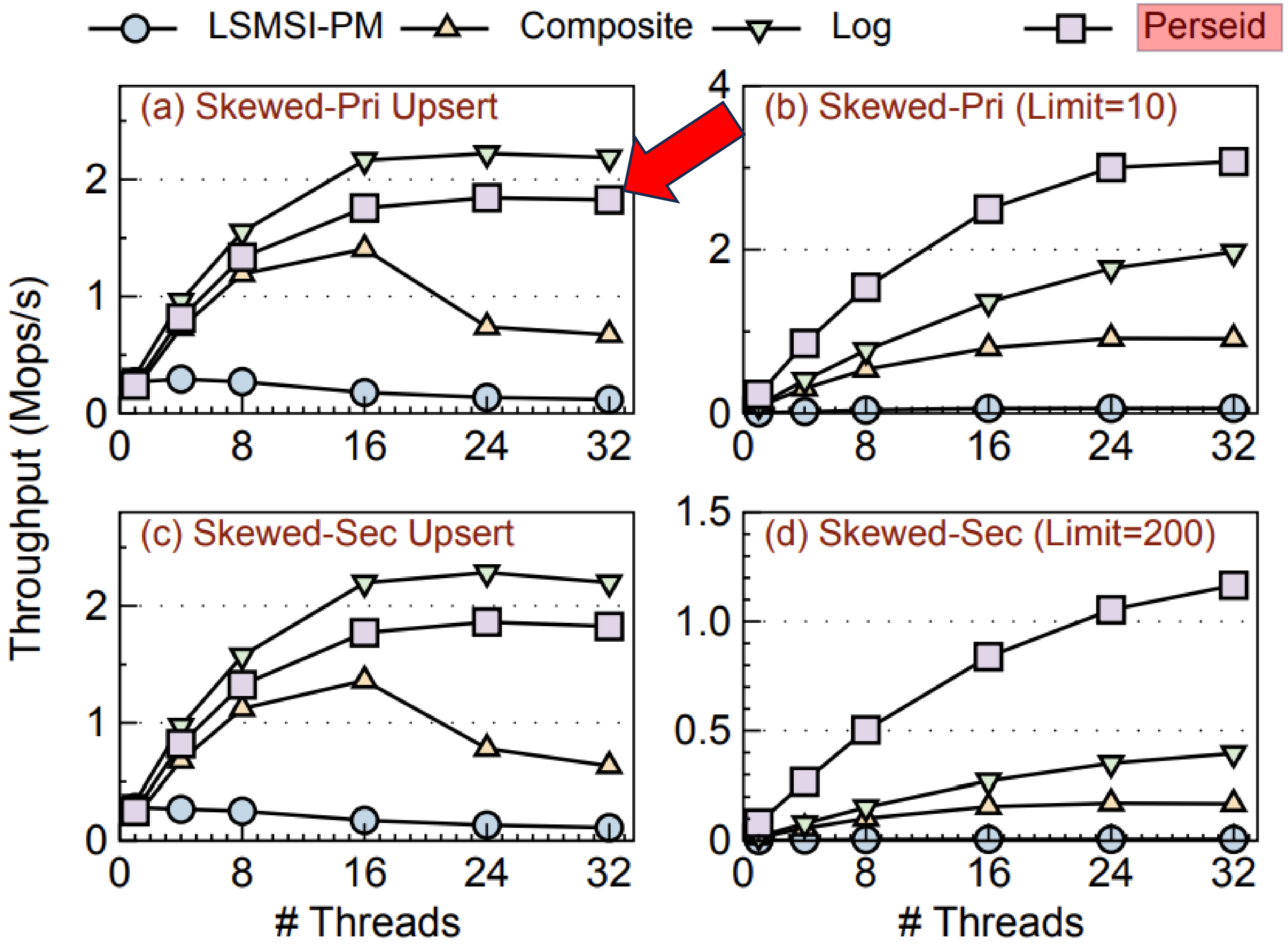


Figure 9: Multi-threaded performance.

- PERSEID:
  - has better scalability
  - achieves 3-7x query performance of other PM indexes
  - has comparable upsert performance as log-structrured approach

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

Thank you