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

ATC’2023

Background: LSM-Tree

- **WAL Log**

  A sequential log that records all writes before they are applied, enabling crash recovery.

- **Memtable**

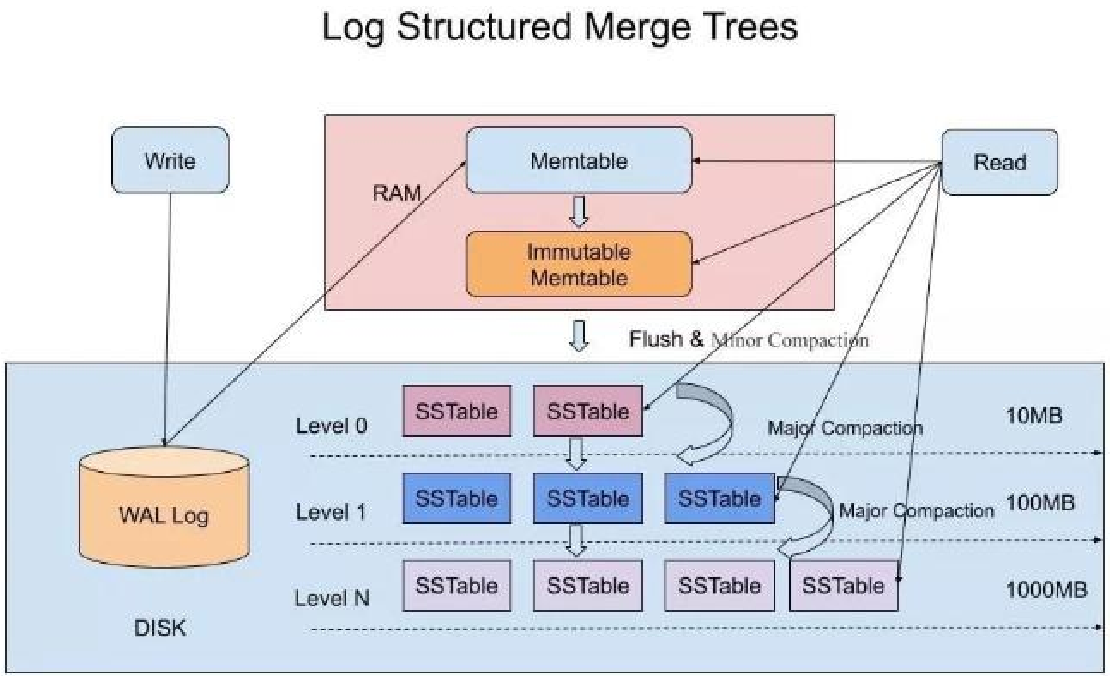
  An in-memory balanced tree that buffers incoming writes.

- **Immutable Memtable**

  A memtable that is made read-only and pending flush to disk.

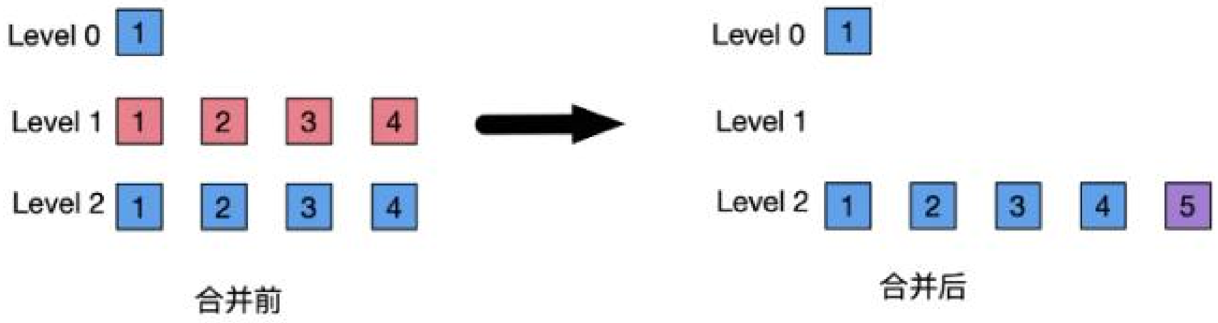
- **SSTable**

  An immutable sorted file on disk containing key-value pairs.



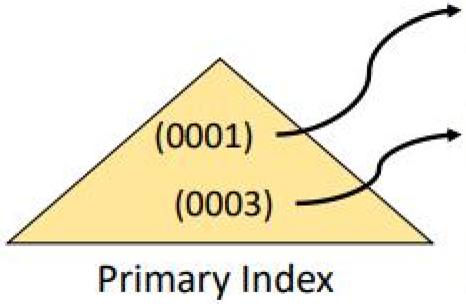
Background: LSM-Tree

- **Tier Compaction**
  - **Number of SSTables in a level Reach Threshold conduct Tier Compaction**
  - **Multiple SSTables in that level will be merged into a new SSTable and placed into the next higher level**



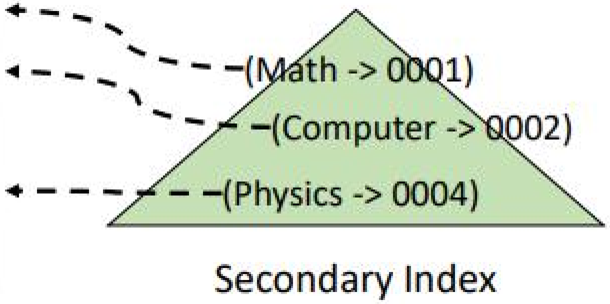
Background: Secondary Indexing

- **Primary Key Index**: indexed by primary key.(StudentID)
- Querying by **non-primay-key** is common. E.g., find student whose major is Computer.
- Second Index
  - Additional index maintaining mappings of **other field to primary key**. E.g., {Major -> StudentID}
  - Besides the main index based on primary key, all other indexes are **secondary index**
  - Indispensable technique in database system



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



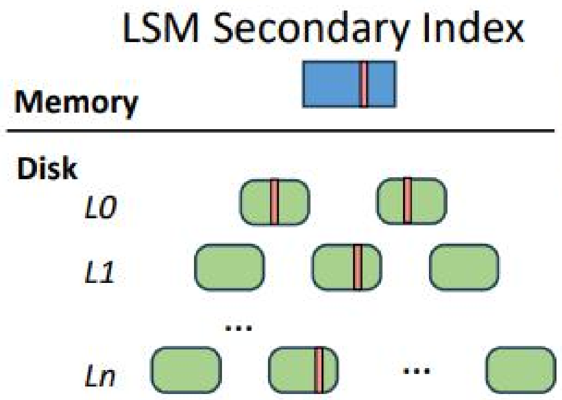
Challenge: Inefficient Secondary Indexing in LSM-based Systems

- **Secondary Indexing is inefficient with LSM-Tree**
  - 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



Attributes of secondary indexes and LSM-tree are mismatched!

Challenge: Consistency Among Indexes

- **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:
    - Insert new entry {Computer -> 0001}
    - Delete old entry {Math -> 0001}
  - Problem: Do not know old secondary key Math due to blind-write

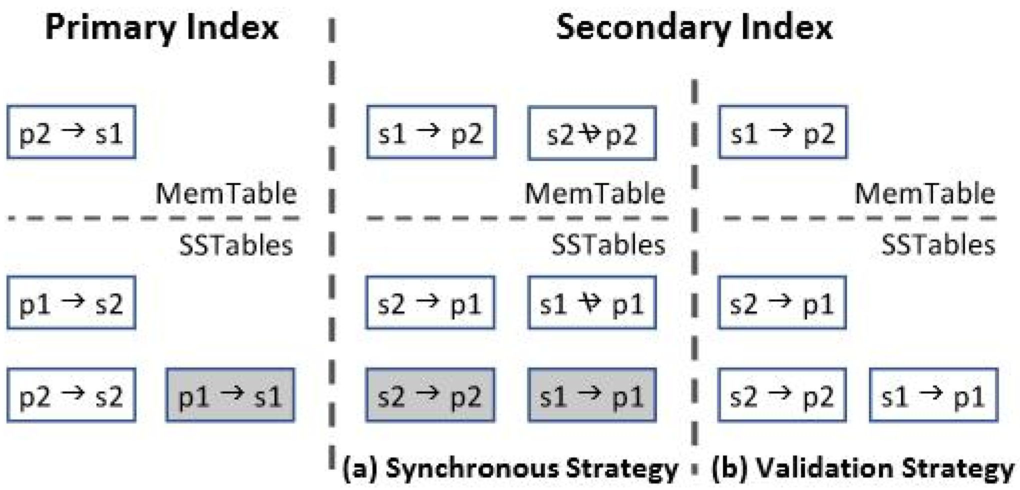
StudentID	Name	Major	EnrollmentYear	...
0001	Alice	Math	2021	
0002	Bob	Computer	2021	
0003	Carol	Music	2022	
0004	Dave	Physics	2022	
0005	Ellen	Computer	2023	

Primary Table

Primary Table   Secondary Index

Challenge: Consistency Among Indexes

- **Two strategies for this issue:**
  - **Synchronous**: READ old record to get old secondary key Math, and then delete in secondary index. → Discard blind-write, low write performance
  - **Validation:** keep old entry {Math -> 0001}, but at query, fetch record of '0001' in primary table for validation → Low query performance



Challenges

- **LSM-Tree Not suitable for secondary indexing:**
  - Optimize query efficiency in LSM-based secondary index (consider multiple values & small KV pairs)
  - Retain blind-write attribute and consistency of secondary indexes

⬇

Find a better solution for secondary indexes in LSM-based storage systems

Persistent Memory

- **leveraging persistent memory (PM) to provide a new solution for secondary indexing is promising.**
  - Byte-addressability
  - DRAM comparable latency
  - Data persistency
  - high random access latency
  - write amplification for small random writes

- **Though there are many state-of-the-art PM-based indexes, none of them are designed for secondary indexing**
  - Use Composite Index
  - Use a conventional allocator

→ **Overshadow their performance!**

Perseid Design Overview

- **PS-Tree**
  - PKey layer for storing secondary values
  - log-structured approaches insertions
  - Arranges entries with good locality
- **Hybrid PM-DRAM Validation**
  - Retains blind-write of LSM
  - Lightweight validation on DRAM
- **Non-Index-Only Query Optimizations**
  - Filters out irrelevant component
  - Parallelizes primary table searching

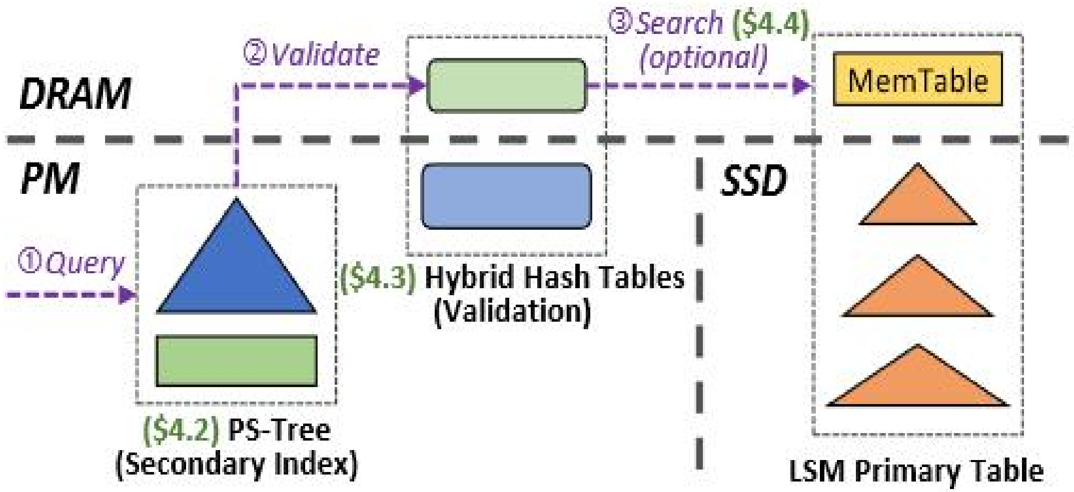
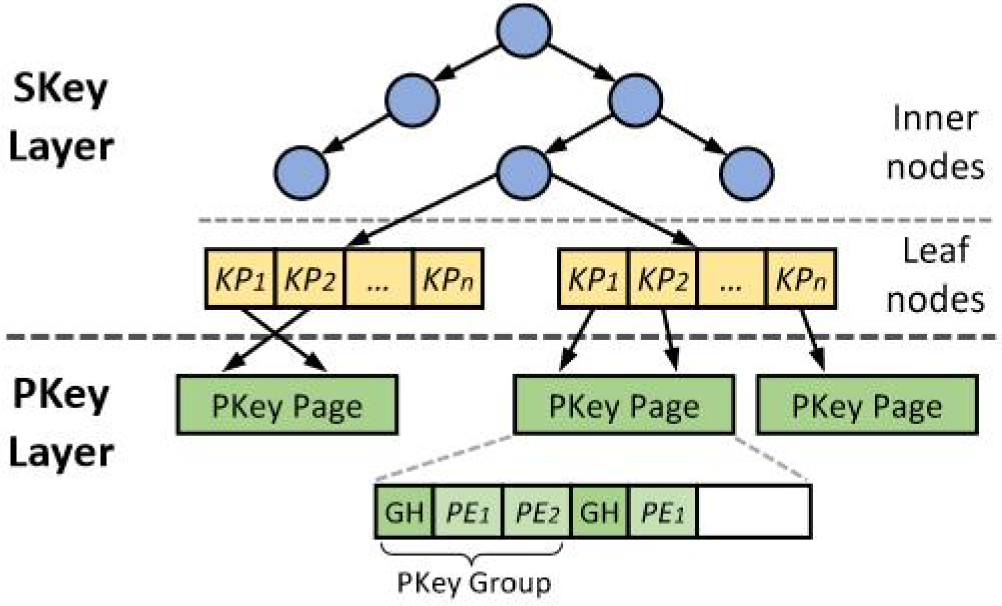


Figure 2: The overall architecture with Perseid.

PS-Tree



- **SKey Layer**
  - Indexing secondary- keys
  - Using existing high-performance PM-based index
  - Each pointer stores a pointer to Pkey page and offset within a Pkey page
- **PKey Layer**
  - Each PKey entry has an 8-byte metadata header and a primary key.
  - Metadata Header contains SQN Number
  - Inserts PKey Entries into PKey Pages in a log-structured manner to reduce the write overhead
  - Stores PKey Entries of contiguous SKeys in the same PKey Page
  - Rearrange entries Pkey entries that belongs to same secondary keys to store as a Pkey Group

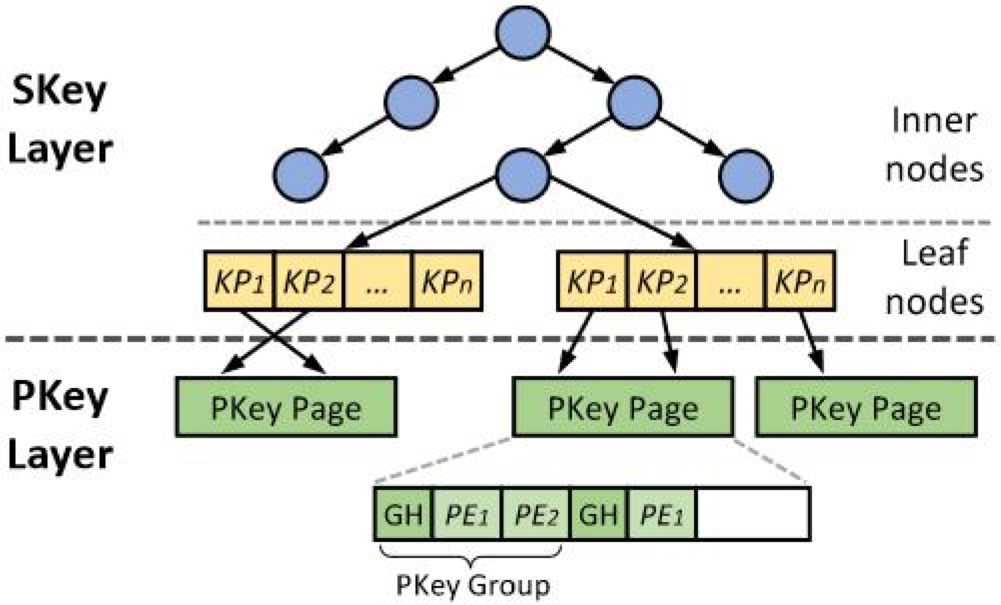
PS-Tree

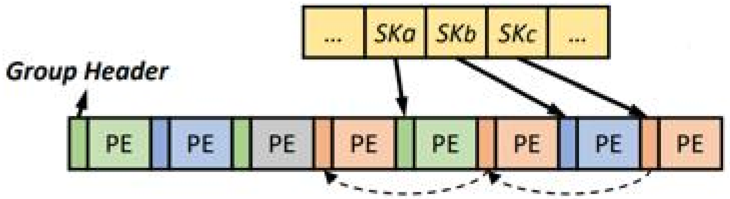
- **PKey Group**

    - Contain a group header(GH) and multiple Pkeys(PE) of the same Skeys

    - Skey point to latest Pkey group

    - Groups belongs to one Skey are linked





PS-Tree Basic Operations

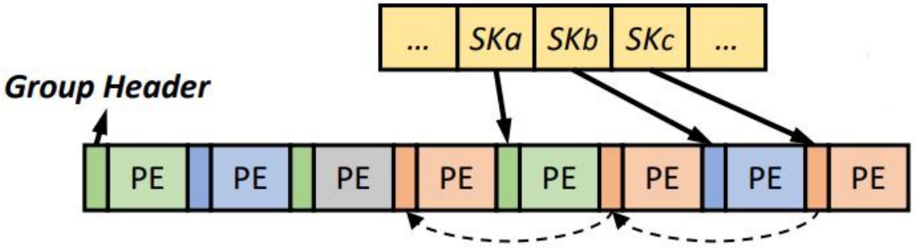
- **Log-Structured Insertion**
  - First search for the Skey to get corresponding Pkey Group
  - Second appends a new Pkey Group in that Pkey page
  - Thrid the new pointer of the Skey is updated or inserted in the SKey Layer
- **Search**
  - First Search Skey Layer for secondary key and its pointer
  - Validate the primary Key before returning

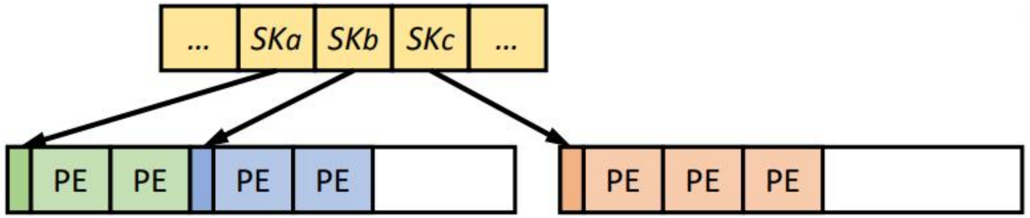
Algorithm 1: Insert(SKey sk, PKey pk, Slice val, SeqNumber seq)

```
1  search for the leaf_node and pointer ptr of sk in SKey Layer;
2  if ptr ≠ NULL then // found sk
3  |   pkey_page = pointer.pkey_page;
4  else
5  |   pkey_page = leaf_node→get_prev_pkey_page(sk);
6  end
7  if pkey_page is full then
8  |   pkey_page split;
9  |   goto Line 1;
10 end
11 construct a PKeyGroup pg with pk, val, seq, and ptr;
12 new_ptr = pkey_page→append(pg);
13 leaf_node→upsert(sk, new_ptr);
```

PS-Tree PKey Page Split & GC

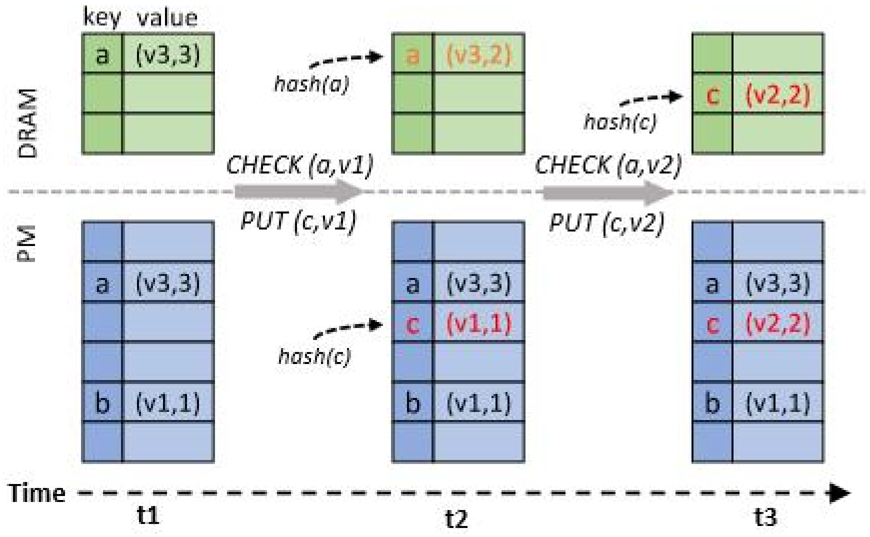
- **Pkey Page split & GC**
  - Split Pkey Page in a copy-on-write manner when Space not enough
  - rearranges PKey Entries belonging to the same SKey in one PKey Group
  - Physical remove obsolete entries and validate other





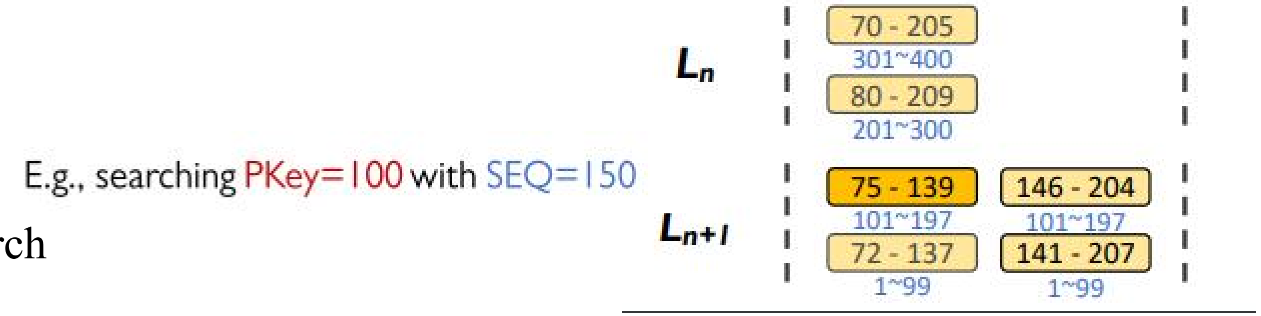
Hybrid PM-DRAM Validation

- **Perseid introduce a lightweight validation approach**
  - Retain blind-write of LSM primary table
  - Volatile hash table in DRAM, persistent hash table in PM
  - Matain the latest version number for primary keys with Persistent hash table
  - Validate using hash table instead of LSM primary table

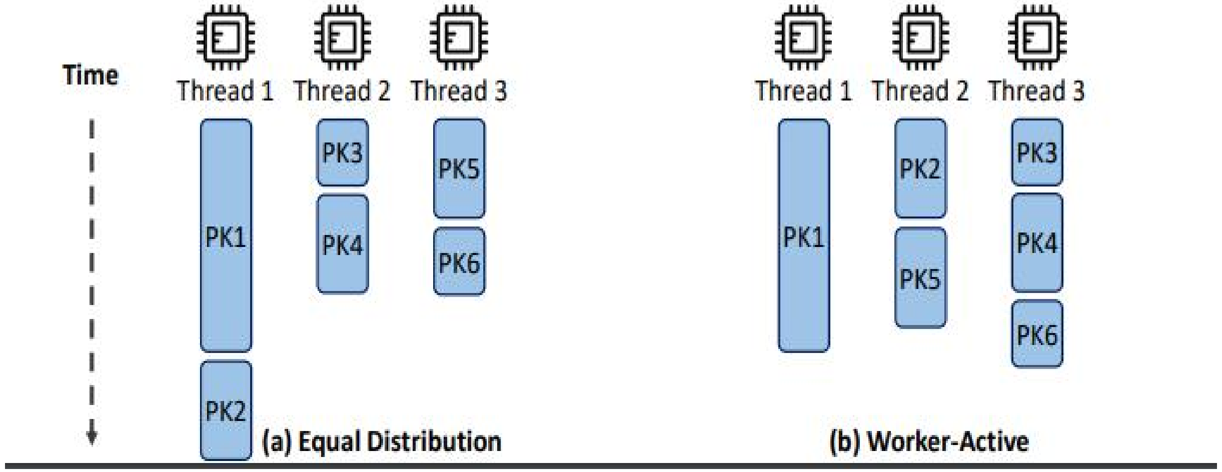


Non-Index-Only Query Optimizations

- **Locating Components with Sequence Number**
  - Build a zone map that stores SQN range for each component
  - Vertically search SQN range and horizontally search Pkey
  - Reducing most component overhead with tiering Strategy



- **Parallel Primary Table Searching**
  - using multiple threads to accelerate primary table searching
  - apply a worker-active fashion.



(a) Equal Distribution (b) Worker-Active

Evaluation: Experiment Set Up

Hardware

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

FAST&FAIR-Perseid **P-Masstree-Perseid**

FAST&FAIR-composite **P-Masstree-composite**

FAST&FAIR-log **P-Masstree-log**

LSMSI **LSMSI-PM**

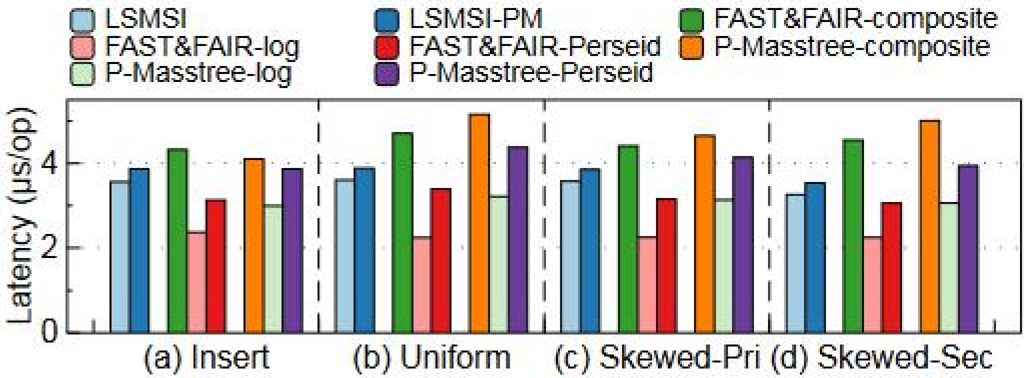
Workloads

Twitter-like workload generator for secondary indexing

100 million primary keys, 4 million secondary keys, record size 1KB

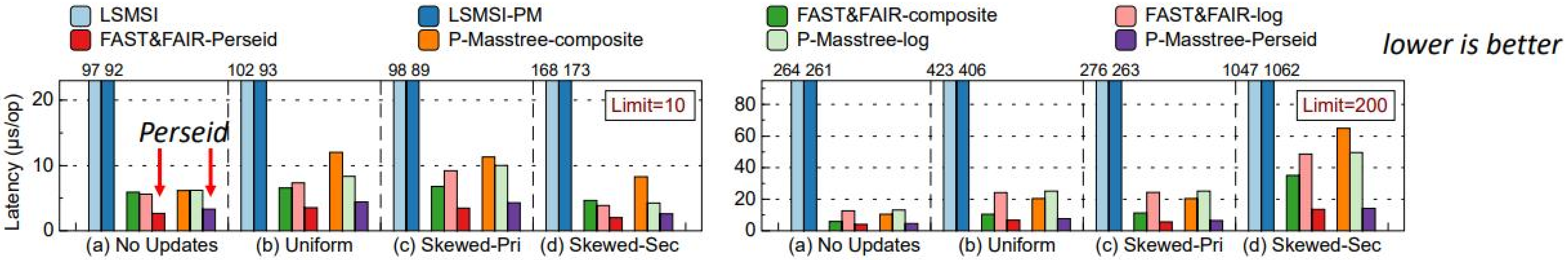
Evaluation: Insert and Update

- **Perseid performs about 10-38% faster than the corresponding composite indexes**
- **25% slower than the ideal log-structured approach without garbage collection due to the page split overhead in PS-Tree.**



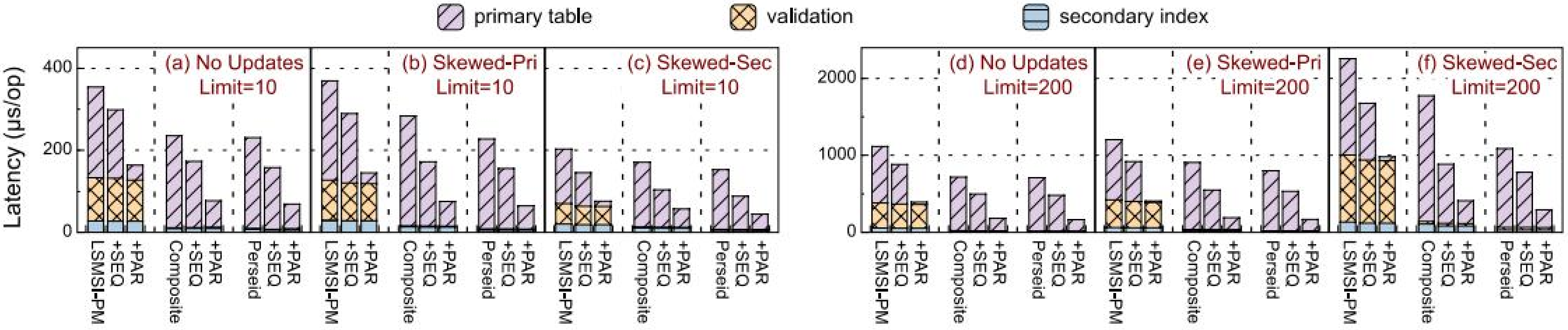
Evaluation: Index-Only Query

- **LSMSI is quite inefficient for queries, even if on PM**
- **Perseid outperforms existing PM indexes by up to 4.5x**
- **Perseid is much more stable across different workloads, owing to the locality-aware design of PS-Tree.**

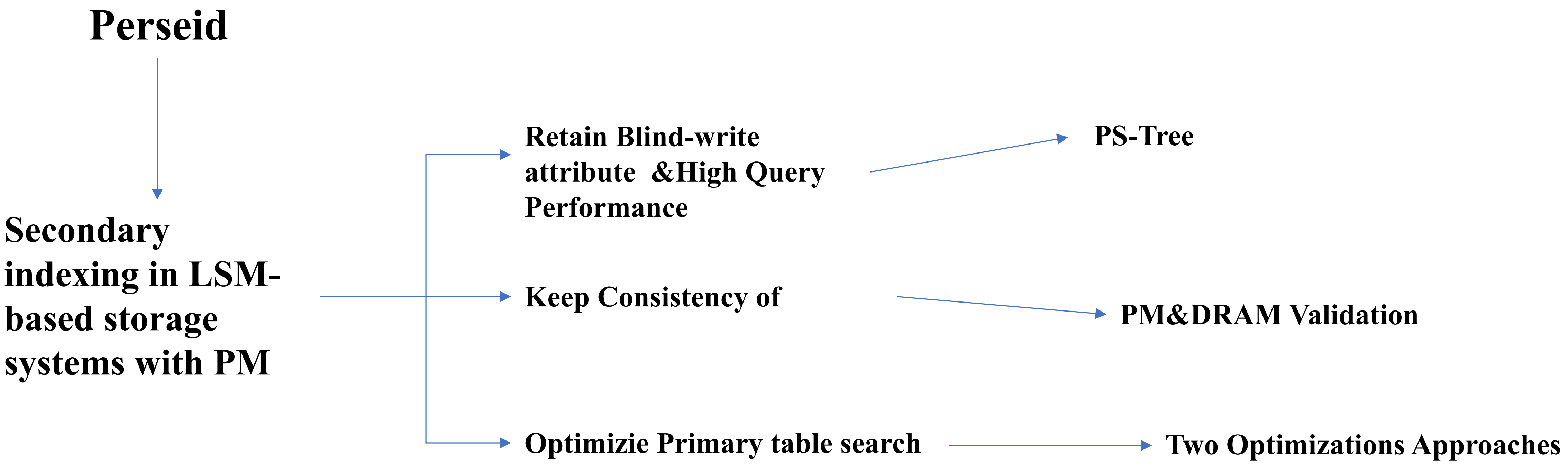


Evaluation: Non-Index-Only Query

- **Perseid outperforms LSMSI by up to 2.3x**
- **Optimizations on primary table searching have significant effect, by up to 3.1x**



Conclusion


Perseid
Secondary indexing in LSM-based storage systems with PM
Retain Blind-write attribute &High Query Performance
PS-Tree
Keep Consistency of
PM&DRAM Validation
Optimizie Primary table search
Two Optimizations Approaches