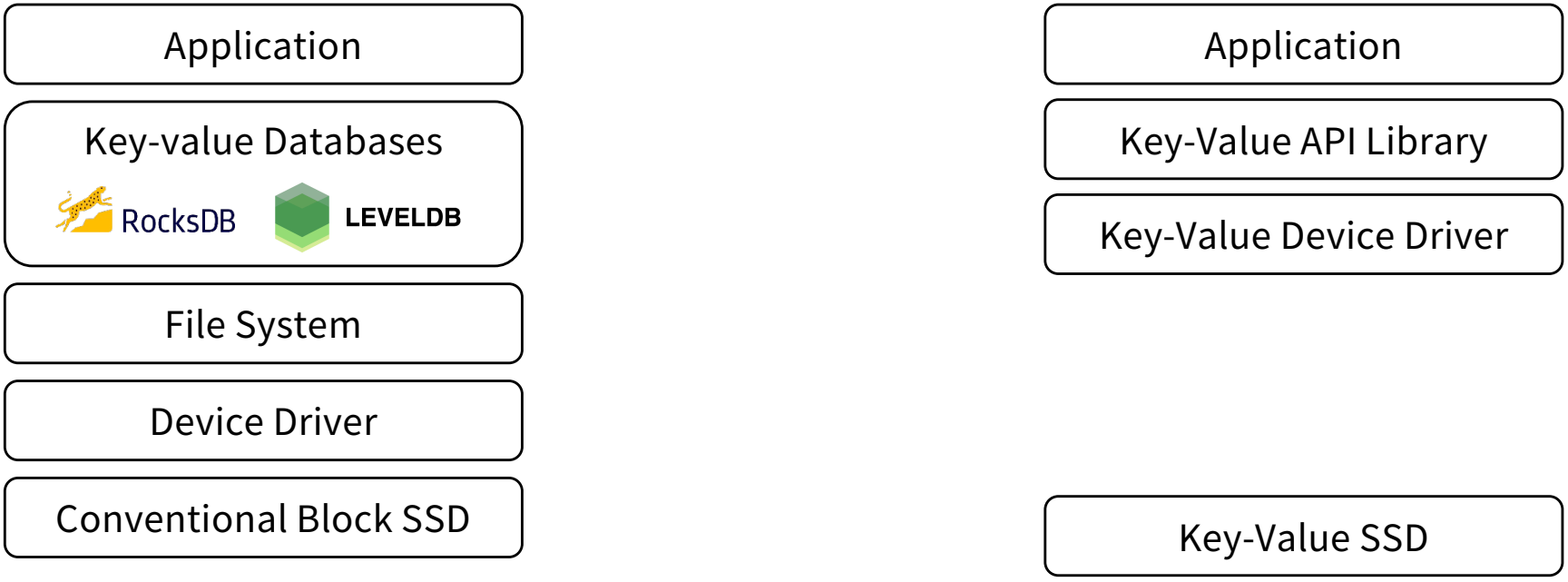
Iterator Interface Extended LSM-tree-based KVSSD for Range Queries

Seungjin Lee[1], Chang-Gyu Lee[1], Donghyun Min[1], Inhyuk Park[2], Woosuk Chung[2], Anand Sivasubramaniam[3], Youngjae Kim[1]

[1] Sogang University, [2] SK hynix, [3] The Pennsylvania State University

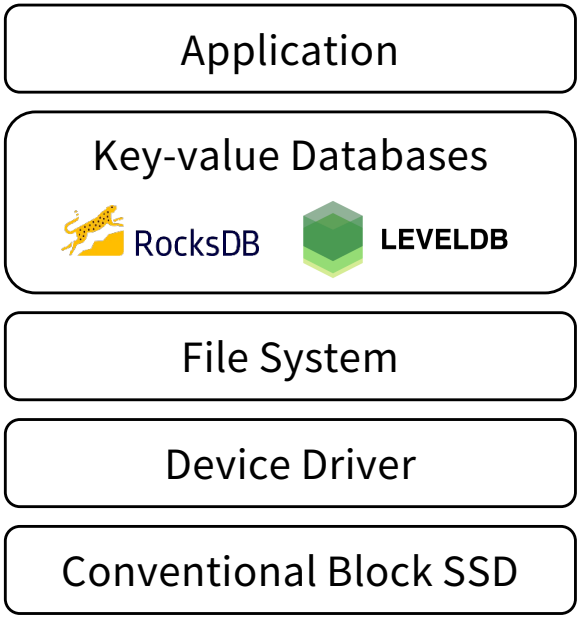
Key-Value SSD

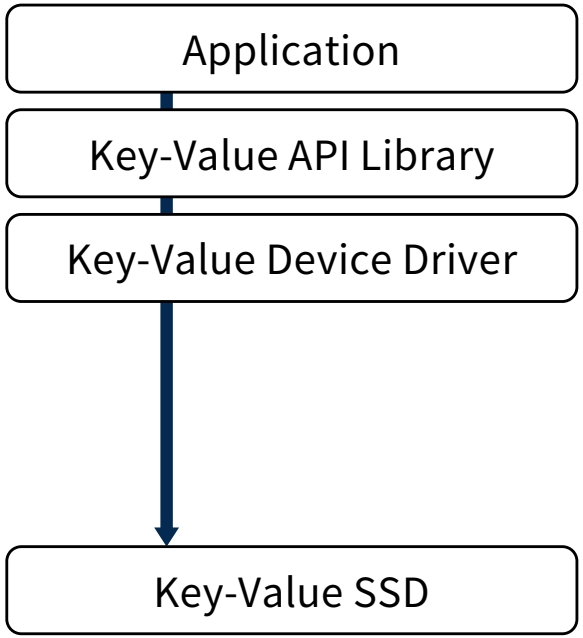
- **Key-Value SSD: Removing the host software I/O stack**
  - By removing the existing deep software I/O stack,



Key-Value SSD

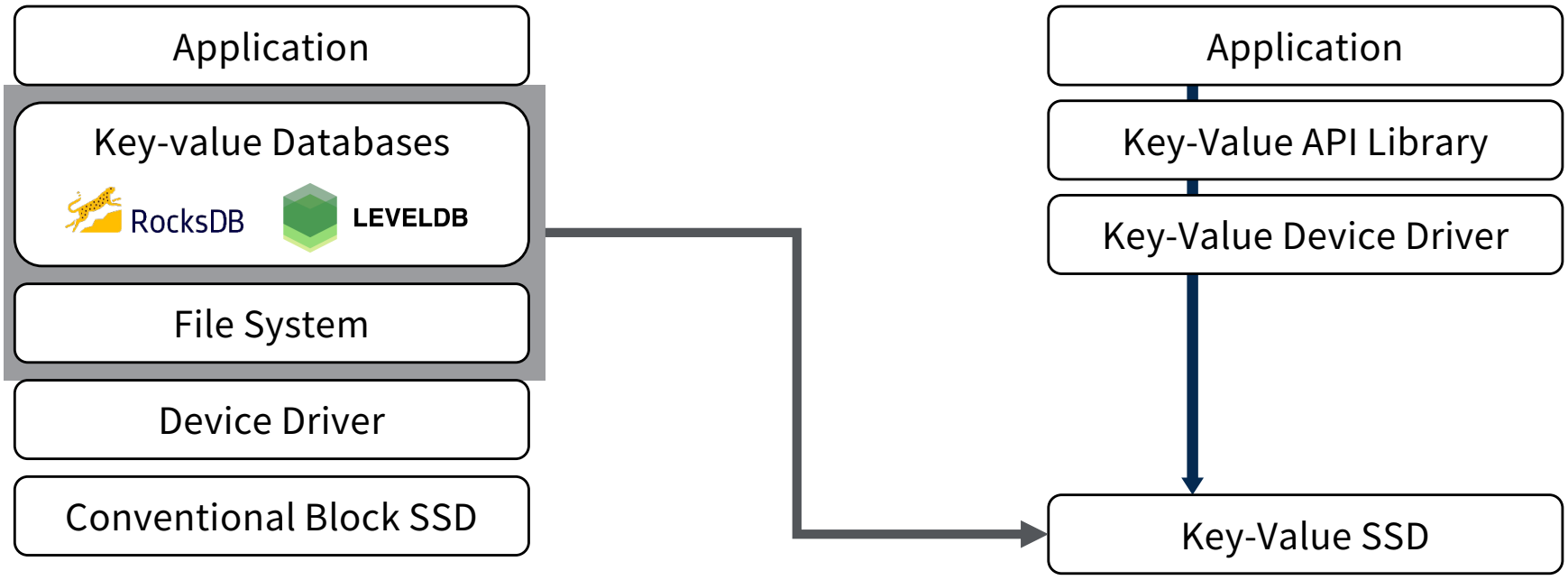
- **Key-Value SSD: Removing the host software I/O stack**
  - By removing the existing deep software I/O stack,





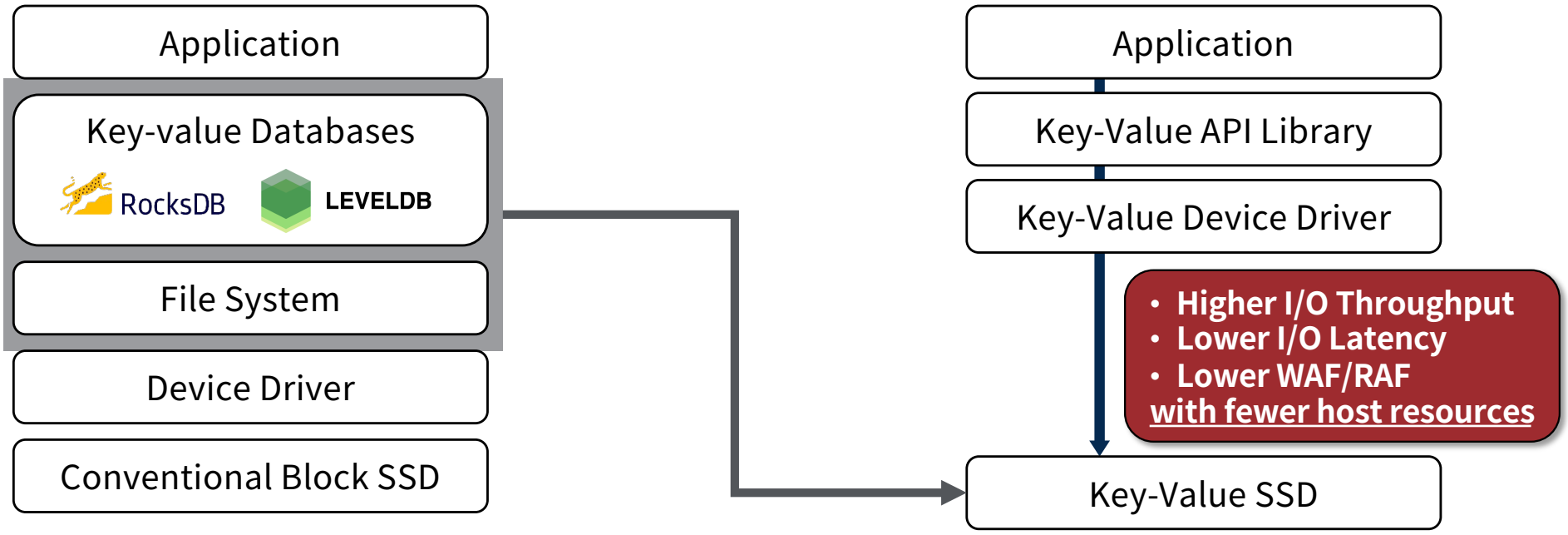
Key-Value SSD

- **Key-Value SSD: Removing the host software I/O stack**
  - By removing the existing deep software I/O stack,



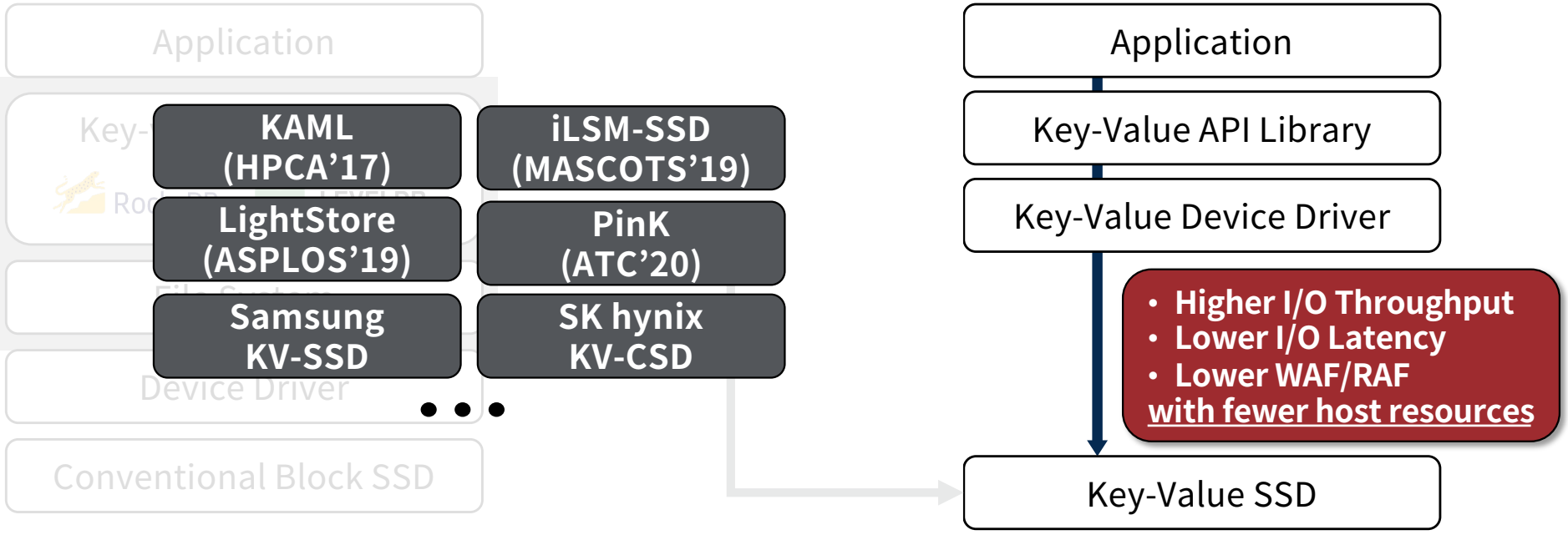
Key-Value SSD

- **Key-Value SSD: Removing the host software I/O stack**
  - By removing the existing deep software I/O stack,



Key-Value SSD

- **Key-Value SSD: Removing the host software I/O stack**
  - By removing the existing deep software I/O stack,



Range Query for KVSSD

- **Existing KVSSDs mostly focus on point queries (Put, Get)**
  - Index Pinning[1], Index Compression[2], H/W Accelerator for Compaction[1]
  - …

[1] PinK: High-speed In-storage Key-value Store with Bounded Tails, USENIX ATC 2020
[2] Modernizing File System through In-Storage Indexing, USENIX OSDI 2021

Range Query for KVSSD

- **Existing KVSSDs mostly focus on point queries (Put, Get)**
  - Index Pinning[1], Index Compression[2], H/W Accelerator for Compaction[1]
  - …
- **What about range query?**

[1] PinK: High-speed In-storage Key-value Store with Bounded Tails, USENIX ATC 2020
[2] Modernizing File System through In-Storage Indexing, USENIX OSDI 2021

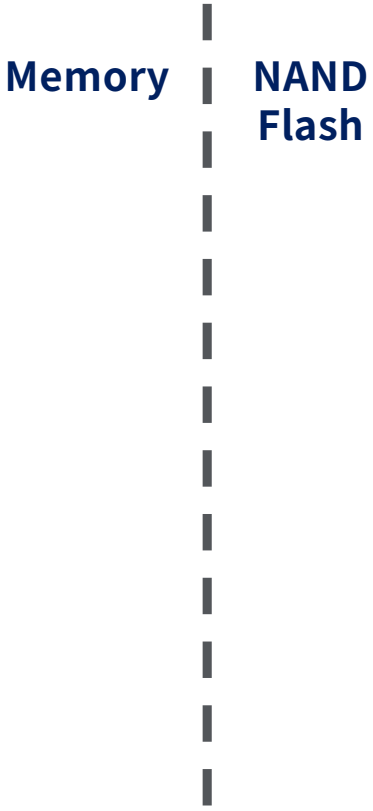
Range Query for KVSSD

- **Existing KVSSDs mostly focus on point queries (Put, Get)**
  - Index Pinning[1], Index Compression[2], H/W Accelerator for Compaction[1]
  - …
- **What about range query?**
  - With ordered data structure, it is often considered simple to implement
  - In previous studies, the design detail of range queries is not covered
  - We claim that there are more things to consider for range query

[1] PinK: High-speed In-storage Key-value Store with Bounded Tails, USENIX ATC 2020
[2] Modernizing File System through In-Storage Indexing, USENIX OSDI 2021

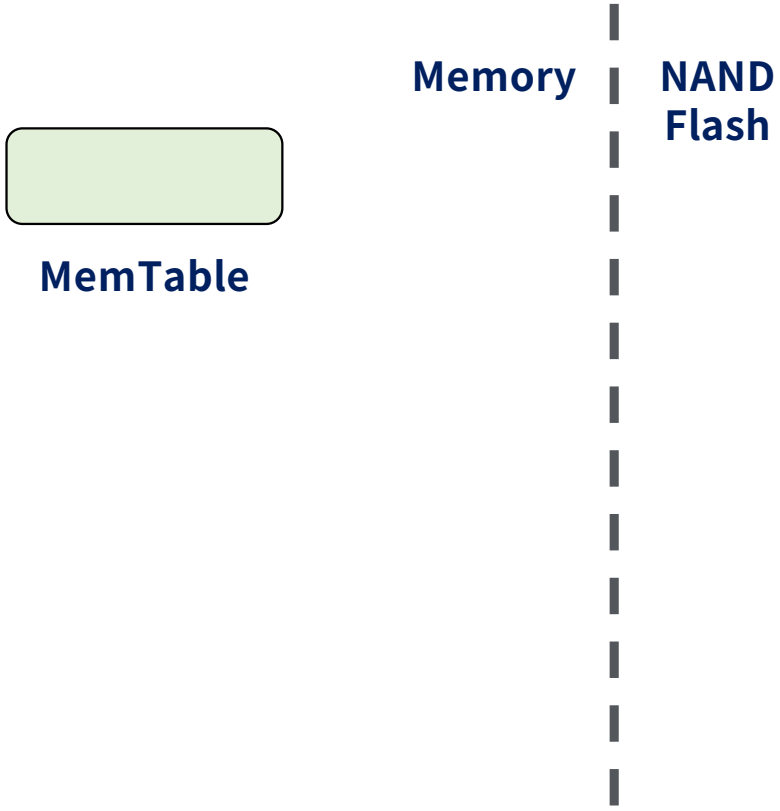
Key-Value SSD Internals

- **LSM-tree-based Key-Value SSD**



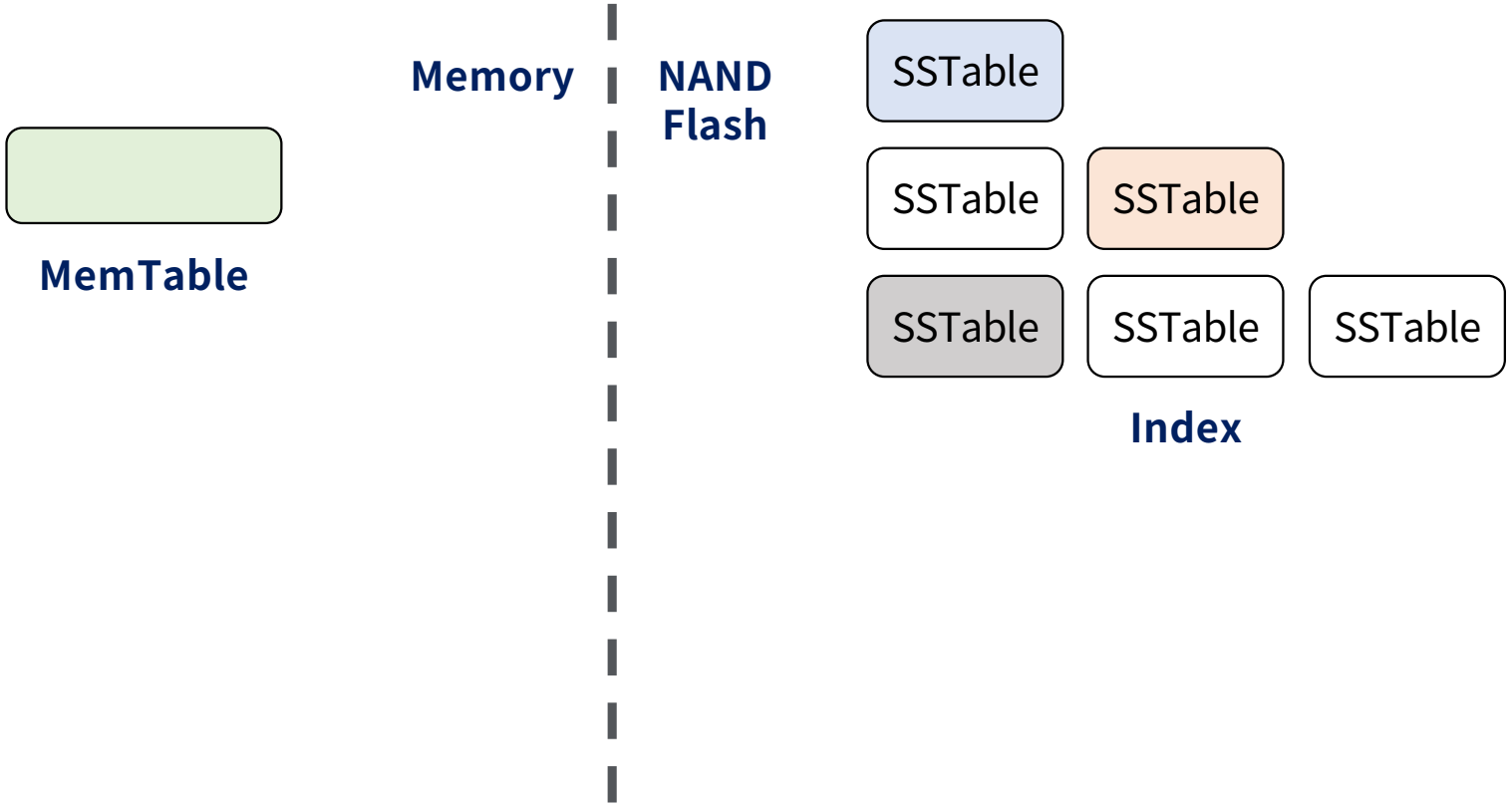
Key-Value SSD Internals

- **LSM-tree-based Key-Value SSD**



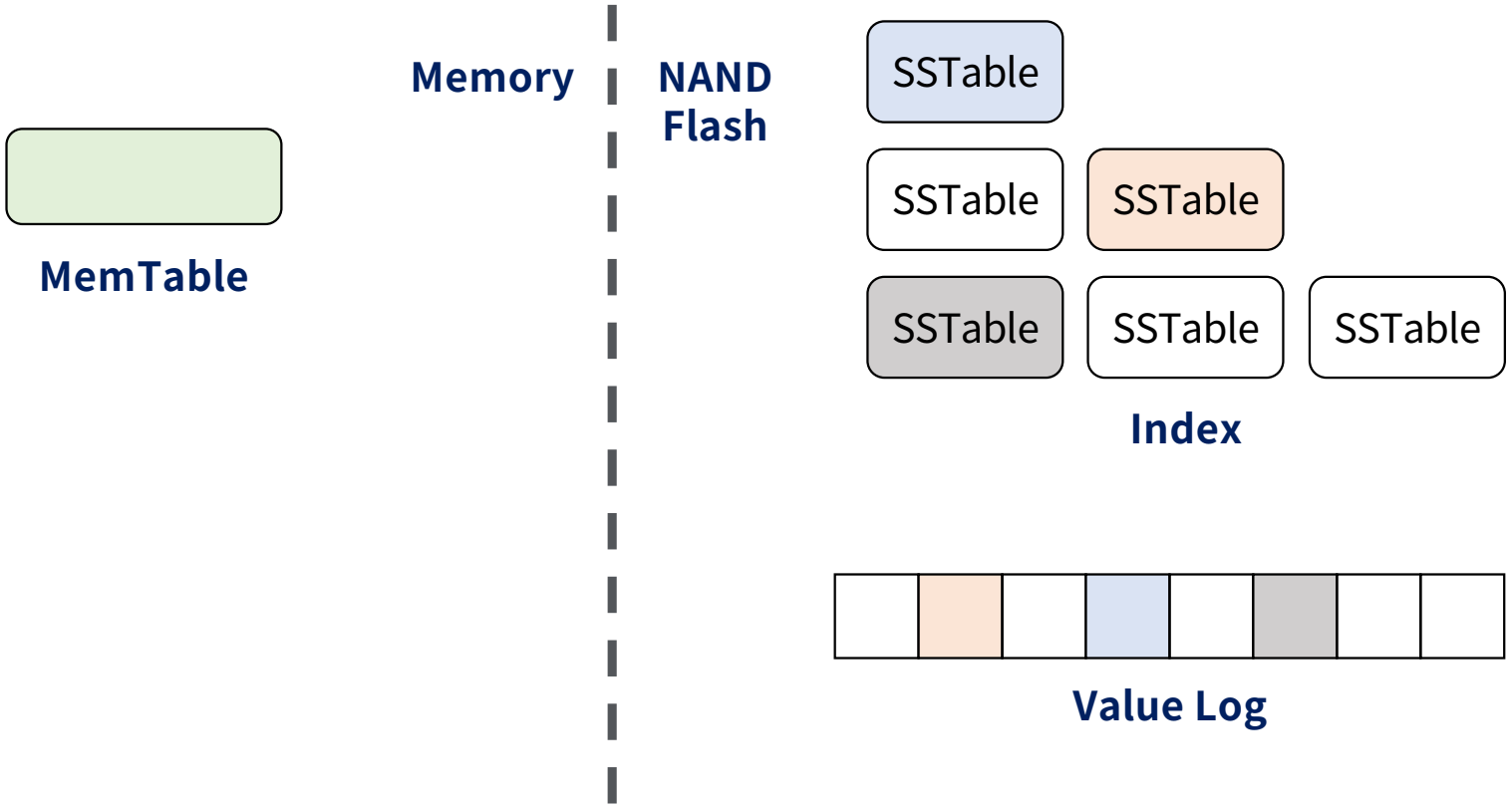
Key-Value SSD Internals

- **LSM-tree-based Key-Value SSD**



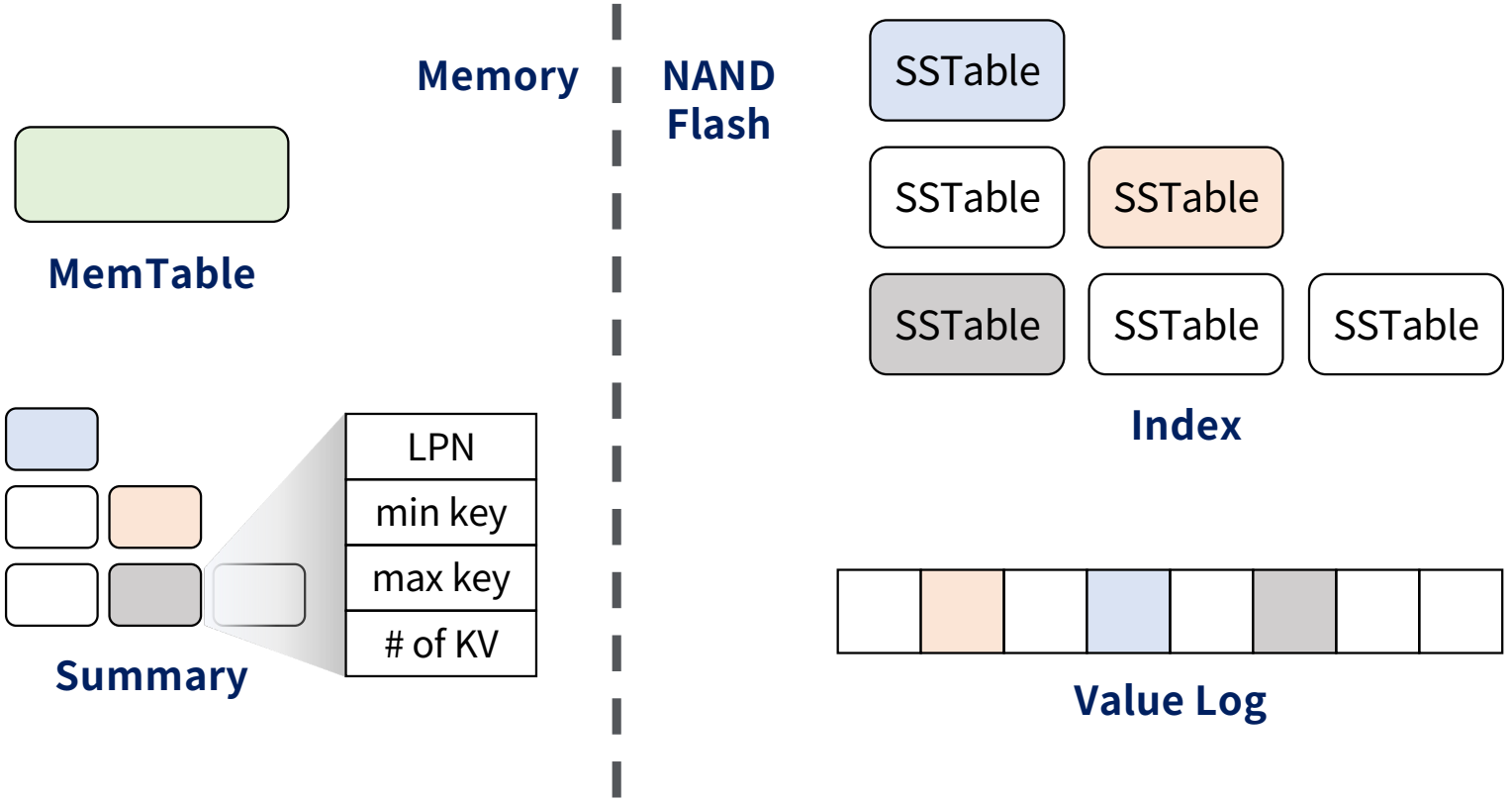
Key-Value SSD Internals

- **LSM-tree-based Key-Value SSD**



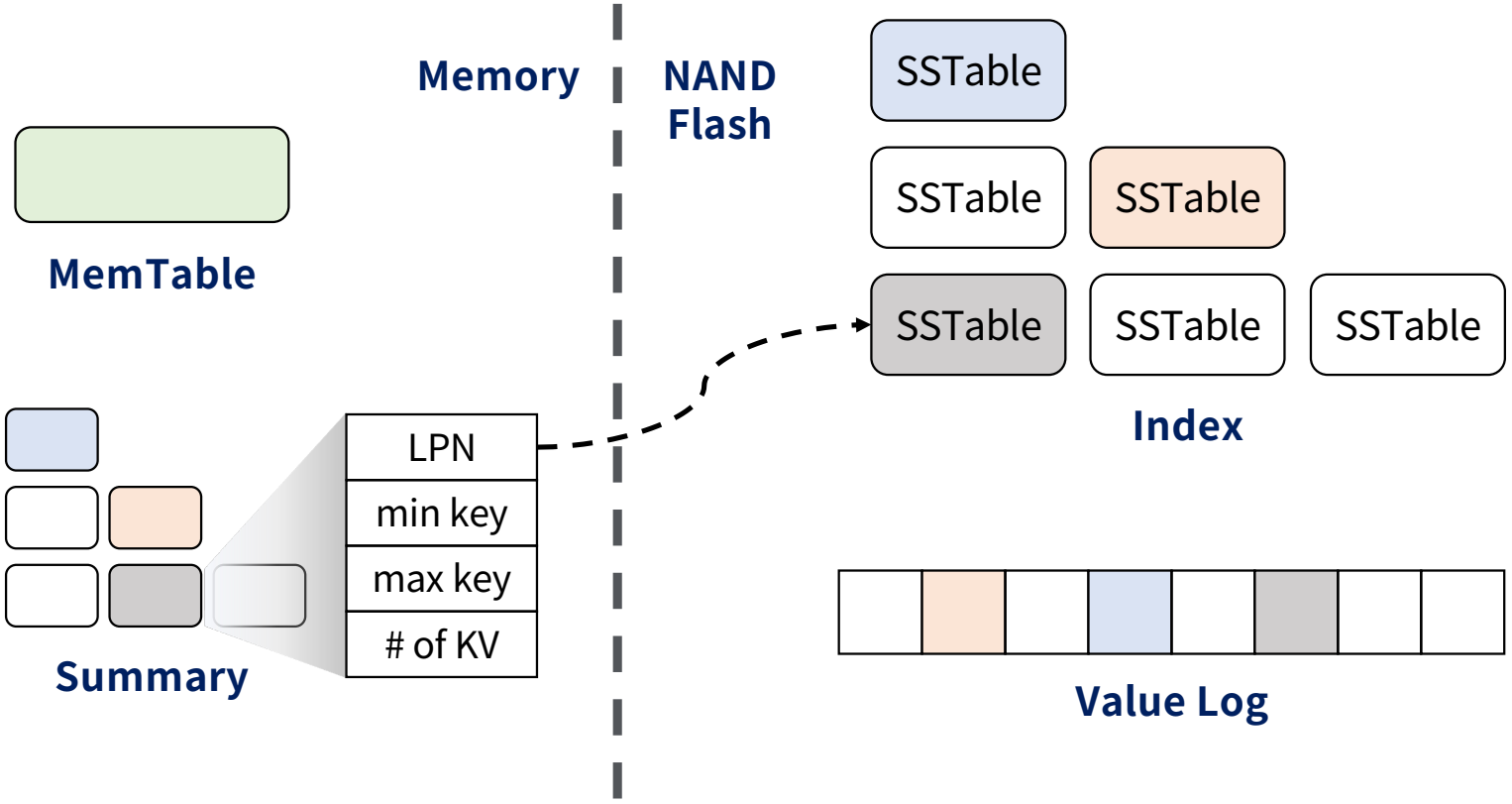
Key-Value SSD Internals

- **LSM-tree-based Key-Value SSD**



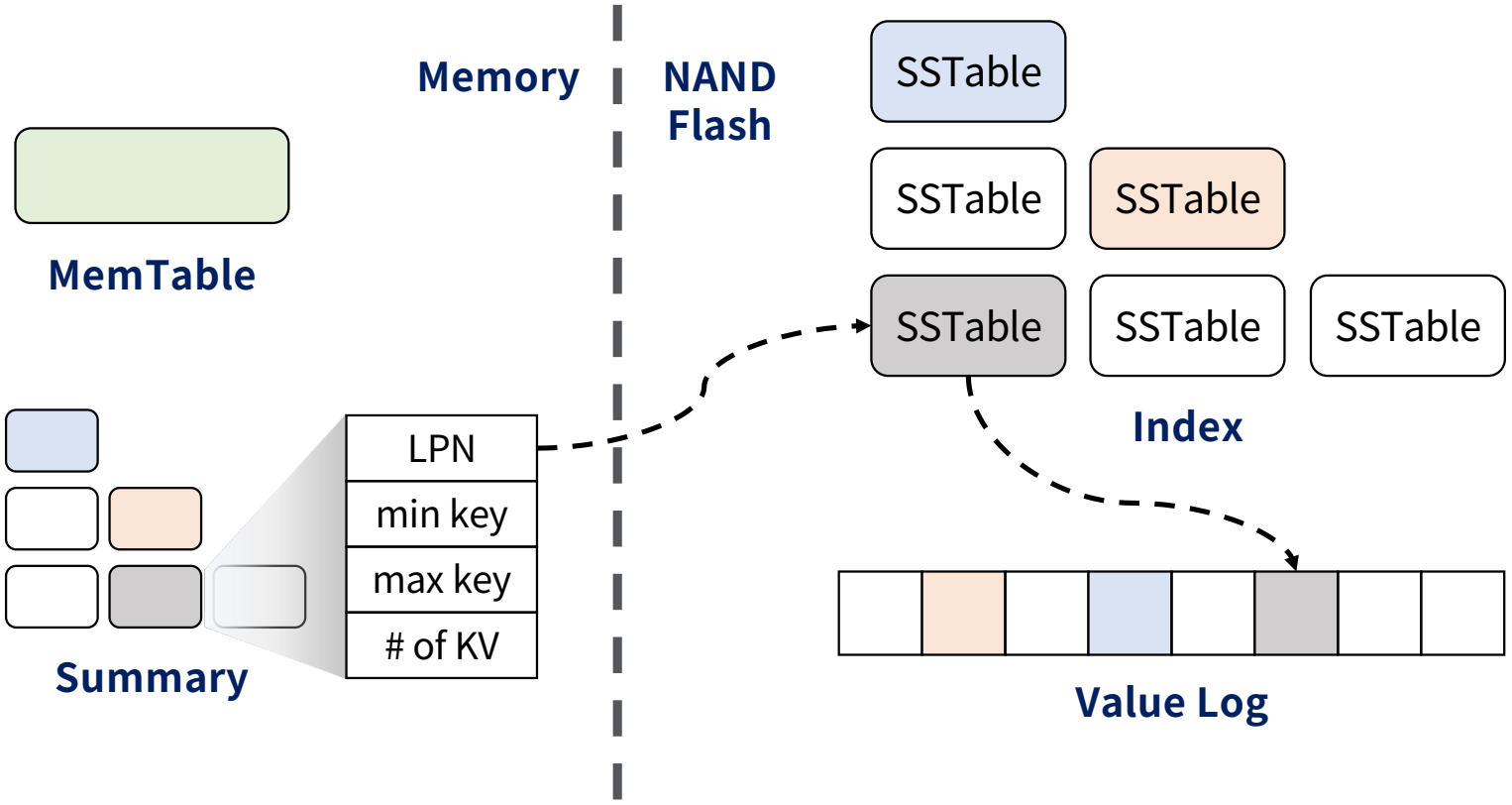
Key-Value SSD Internals

- **LSM-tree-based Key-Value SSD**

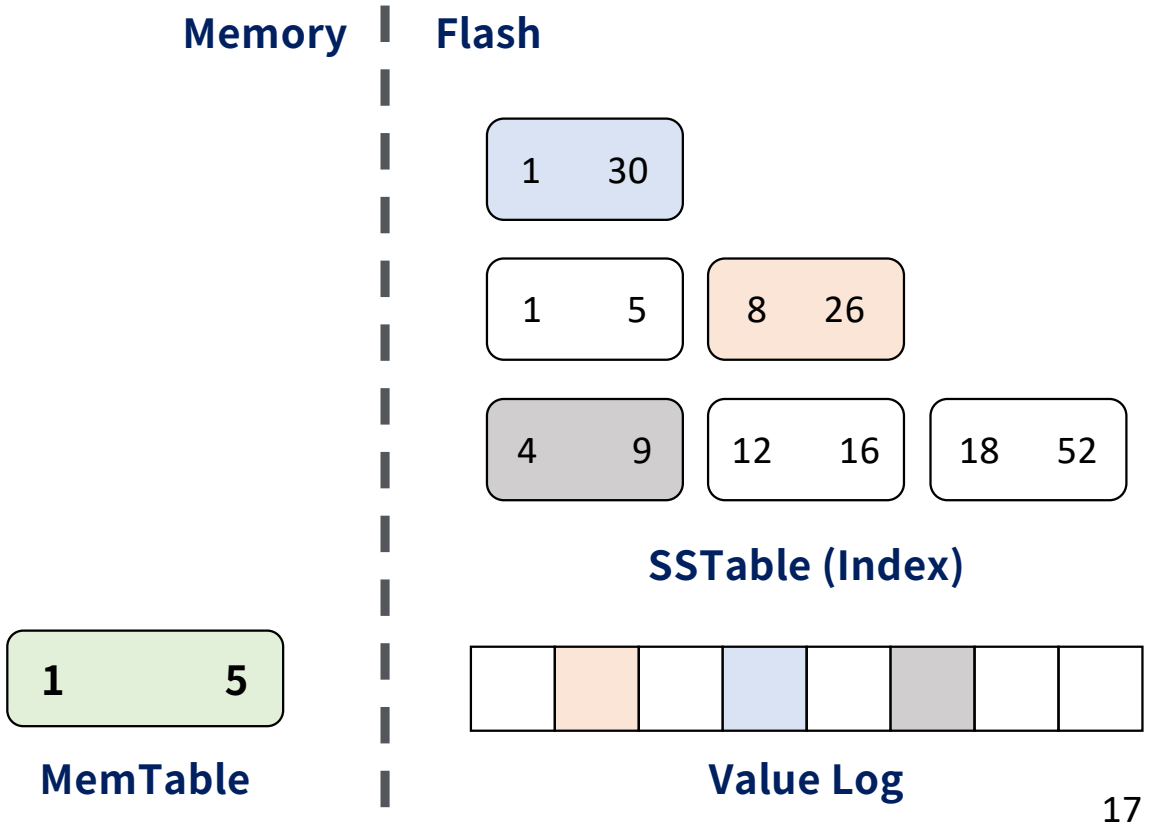


Key-Value SSD Internals

- **LSM-tree-based Key-Value SSD**

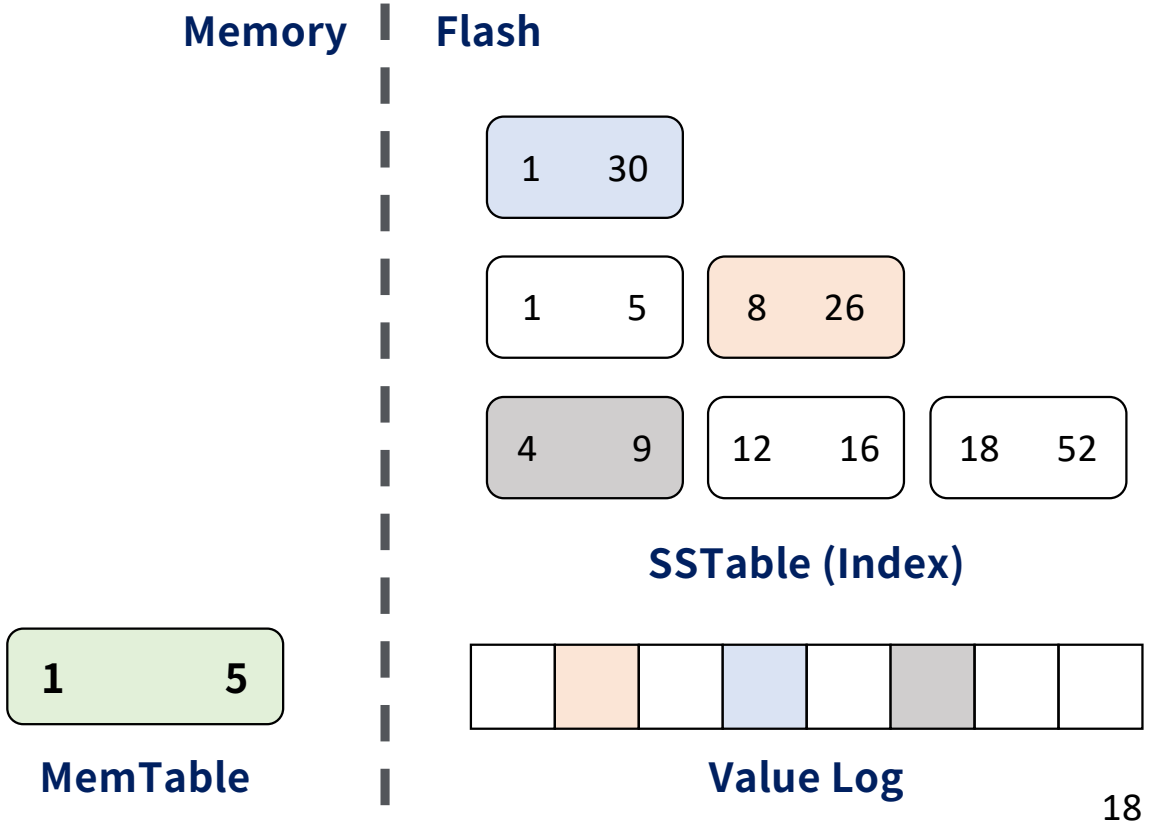


Range Query in LSM-tree-based KVSSD



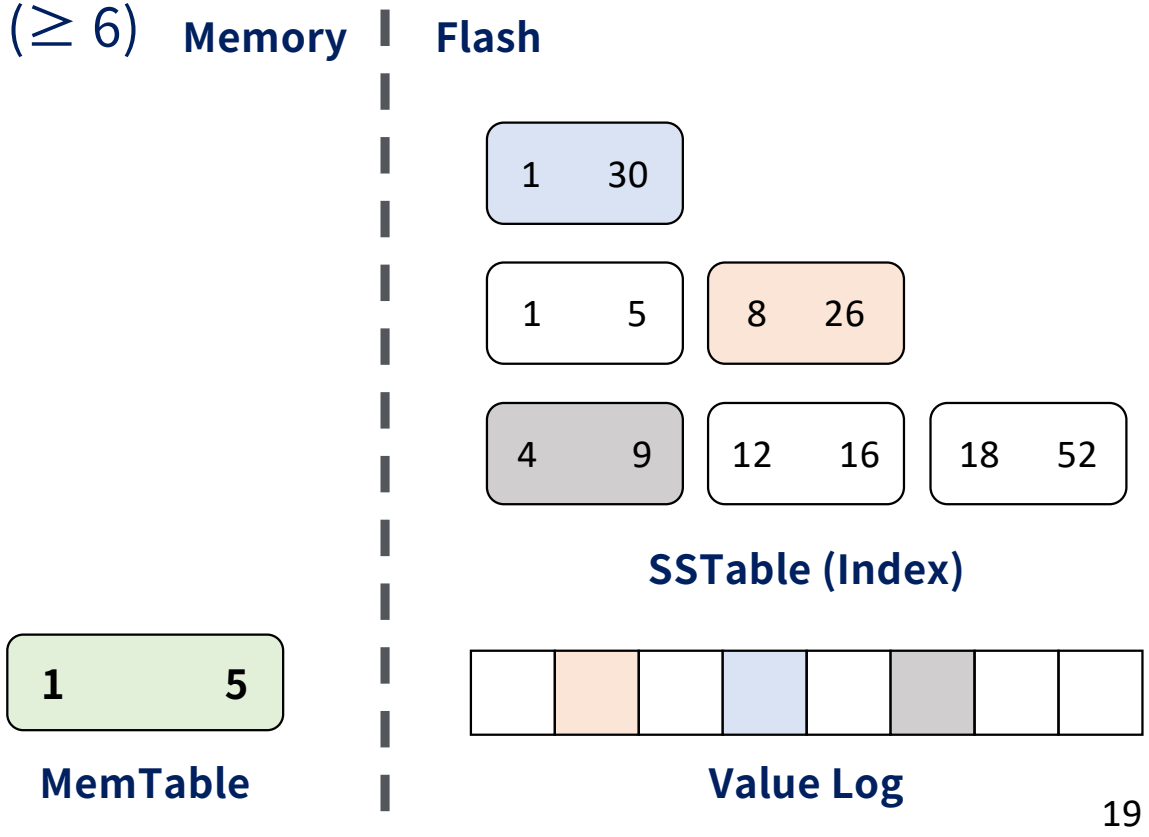
Range Query in LSM-tree-based KVSSD

- **Range Queries are often served as Iterator Interface**
  - Seek() and Next()



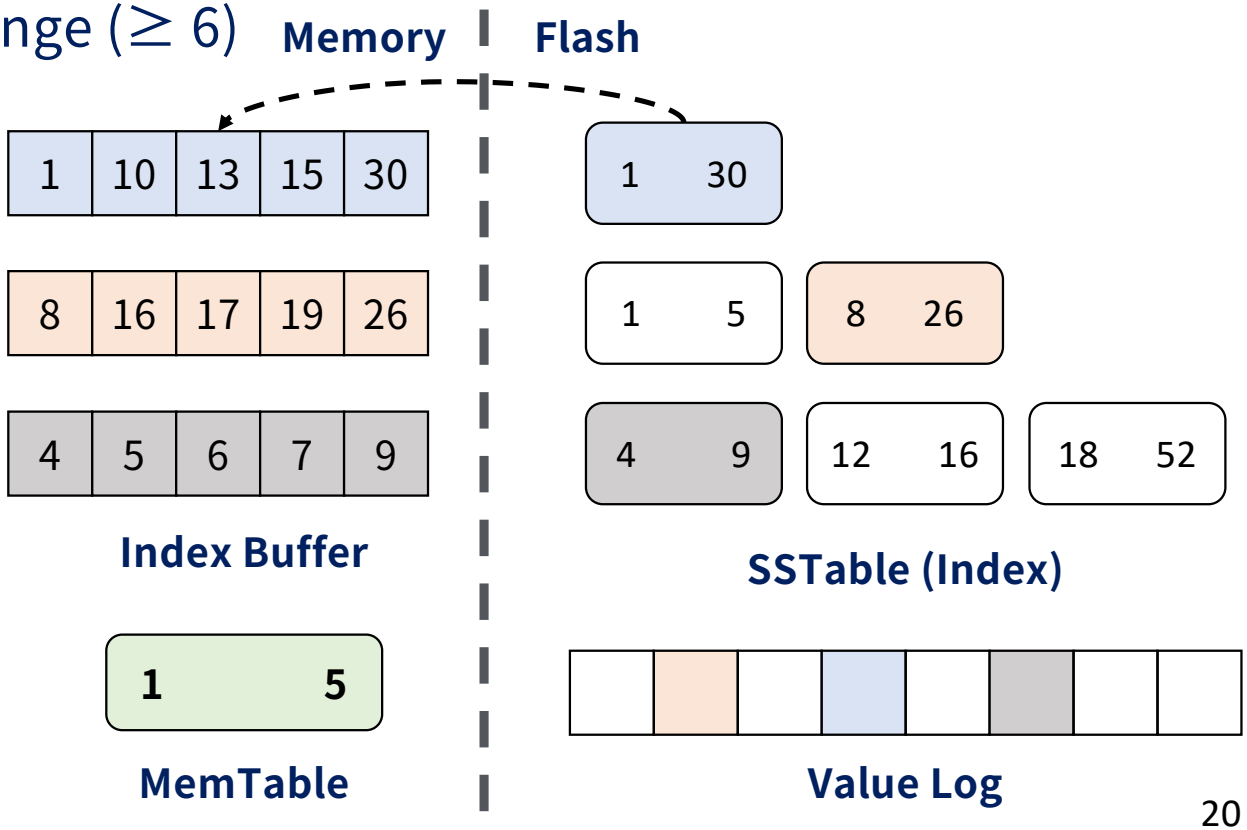
Range Query in LSM-tree-based KVSSD

- **On Seek with start_key=6**
  - Find Index in the search range (≥ 6)



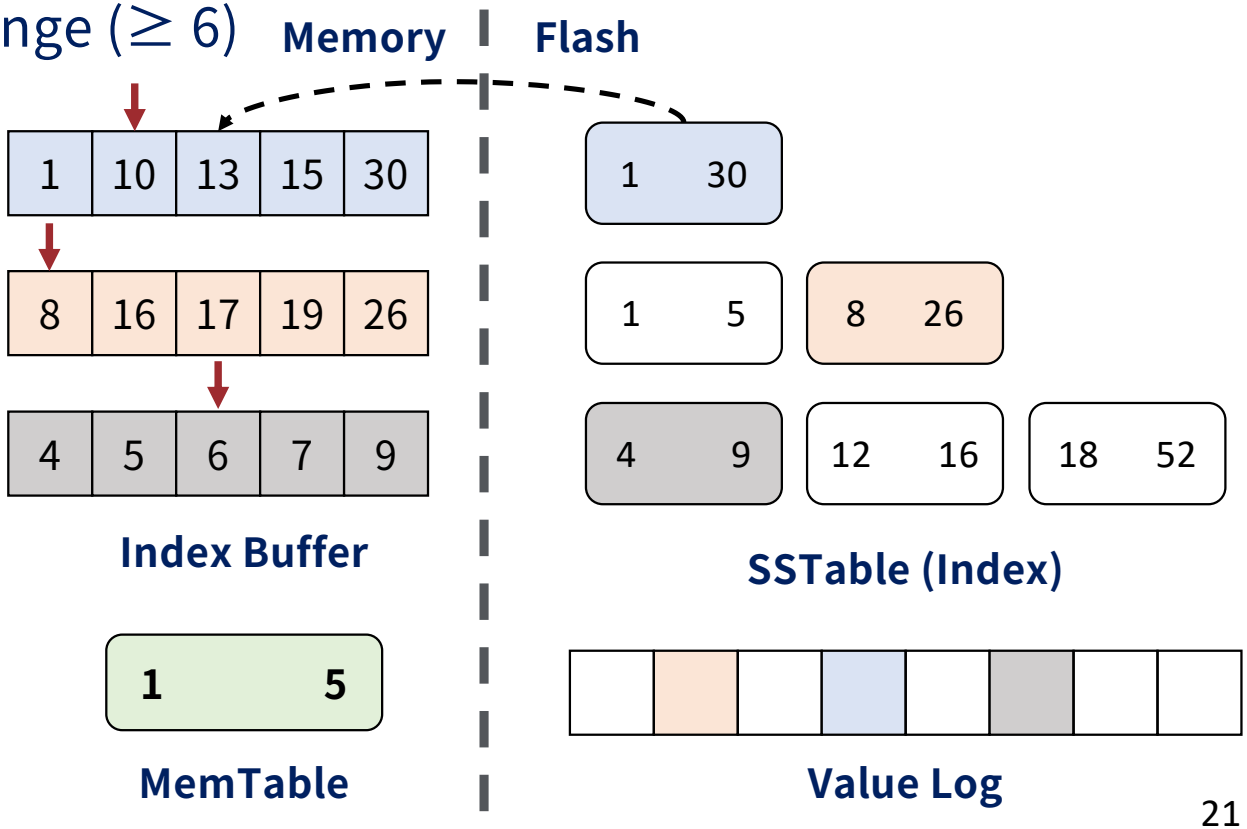
Range Query in LSM-tree-based KVSSD

- **On Seek with start_key=6**
  - Find Index in the search range (≥ 6)



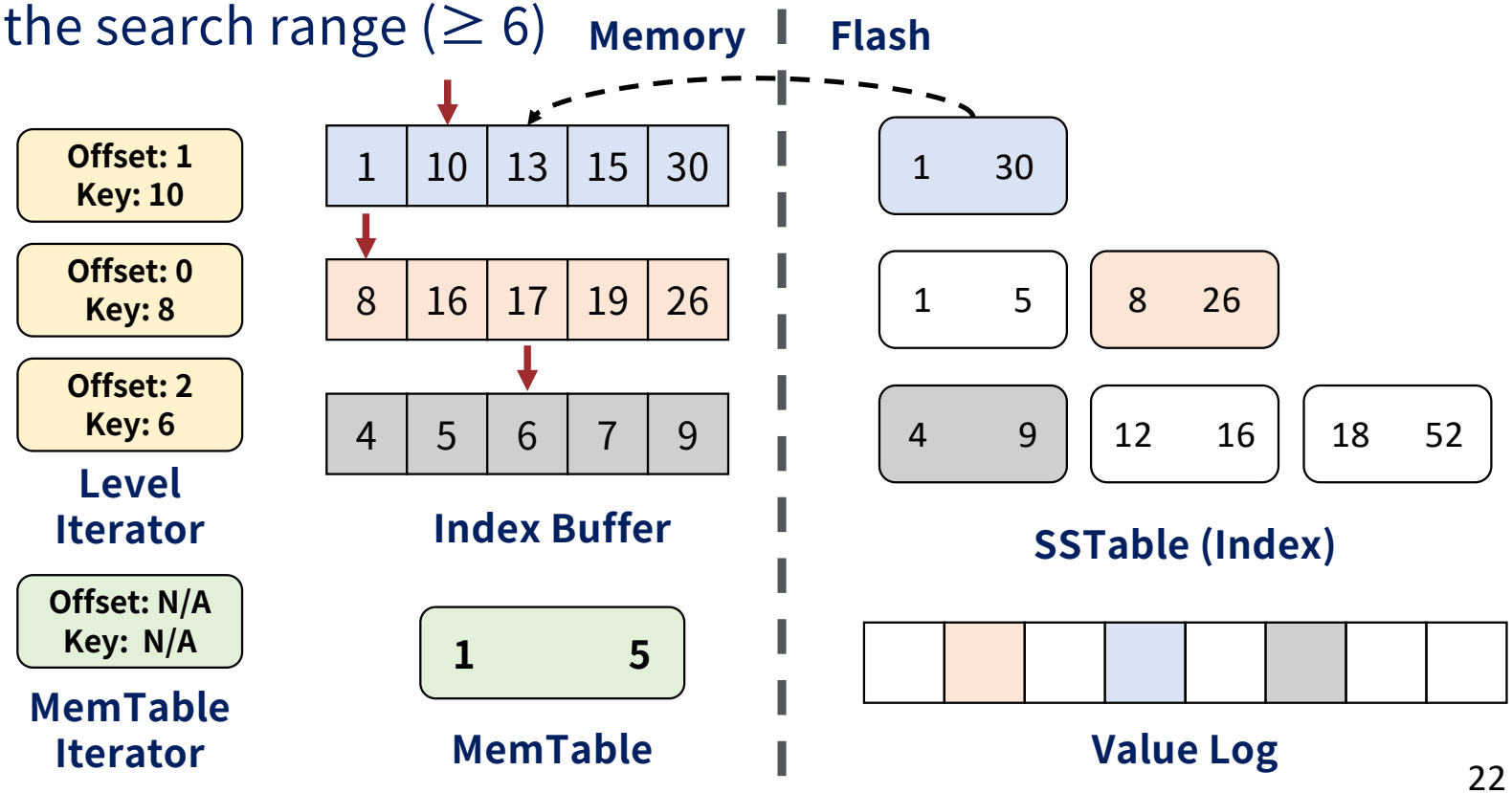
Range Query in LSM-tree-based KVSSD

- **On Seek with start_key=6**
  - Find Index in the search range (≥ 6)



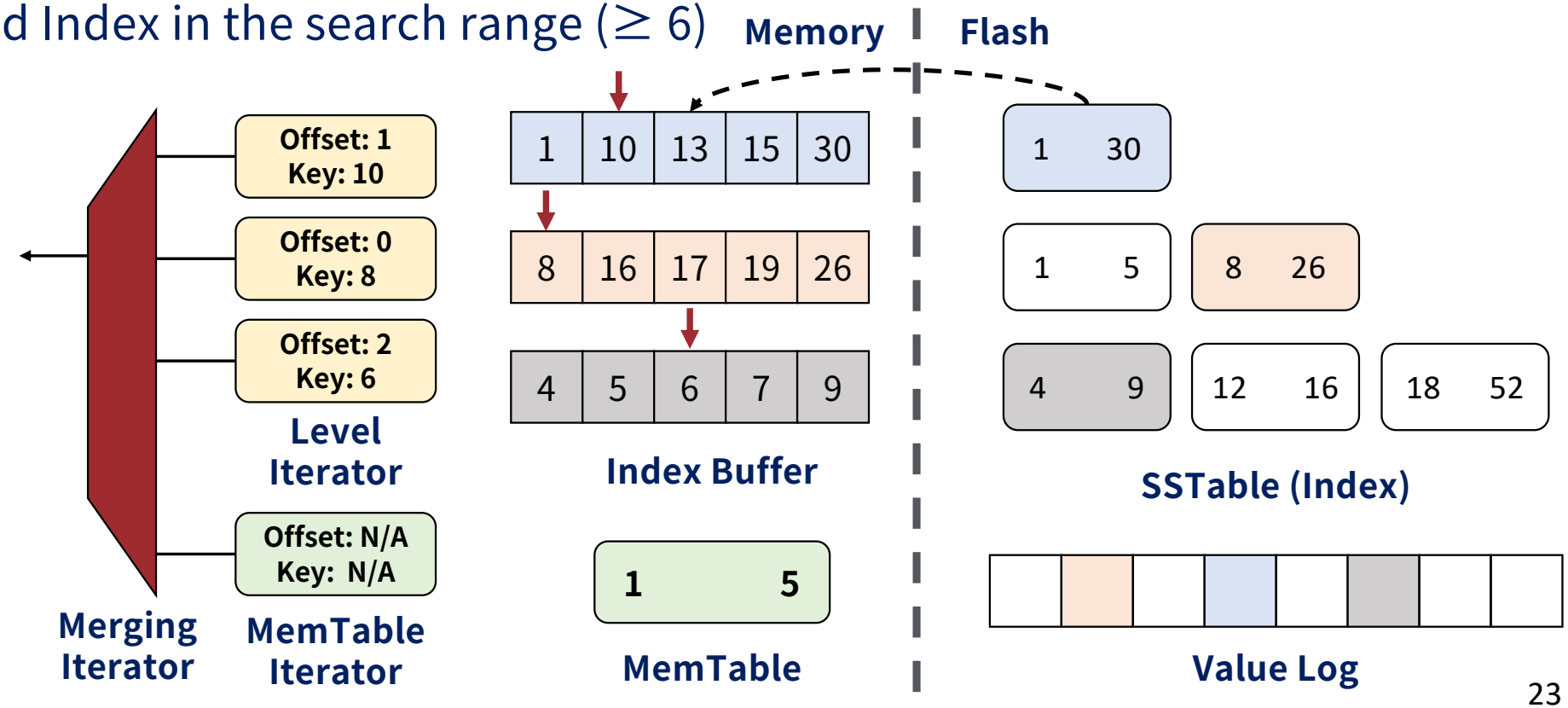
Range Query in LSM-tree-based KVSSD

- **On Seek with start_key=6**
  - Find Index in the search range (≥ 6)



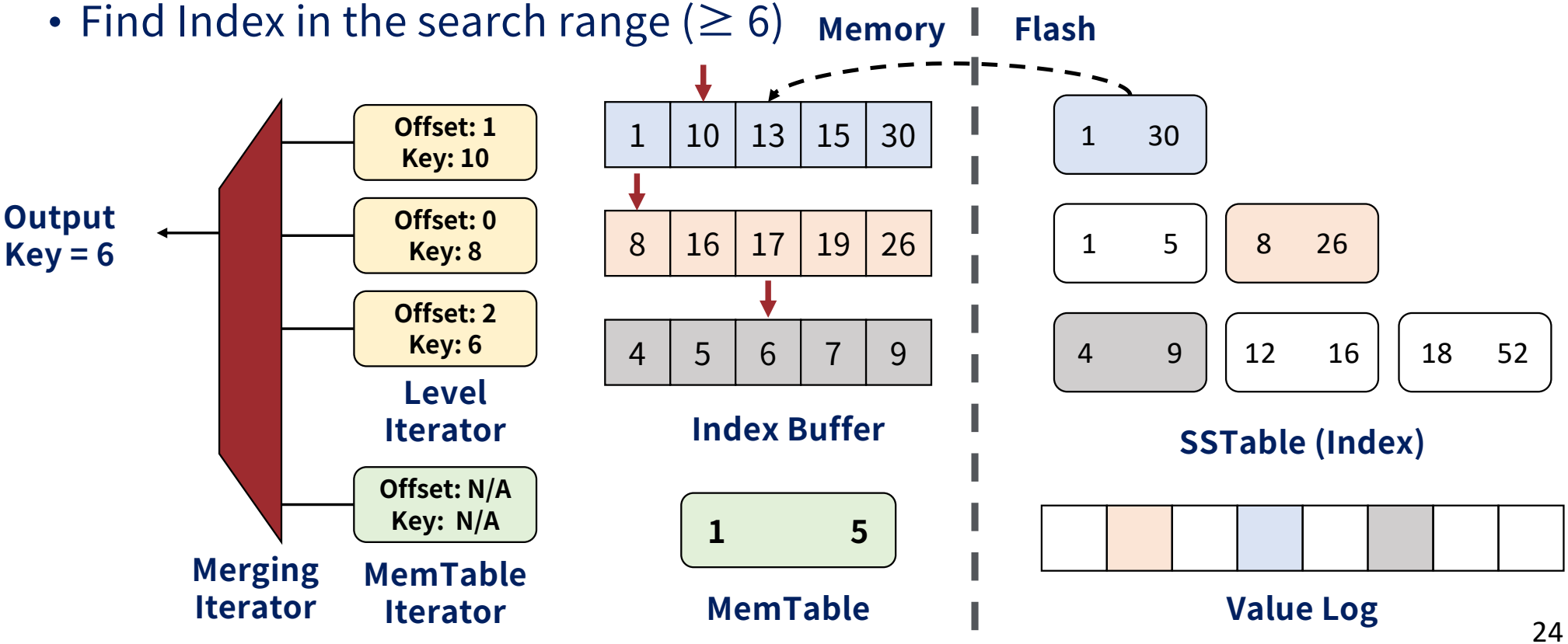
Range Query in LSM-tree-based KVSSD

- **On Seek with start_key=6**
  - Find Index in the search range (≥ 6)



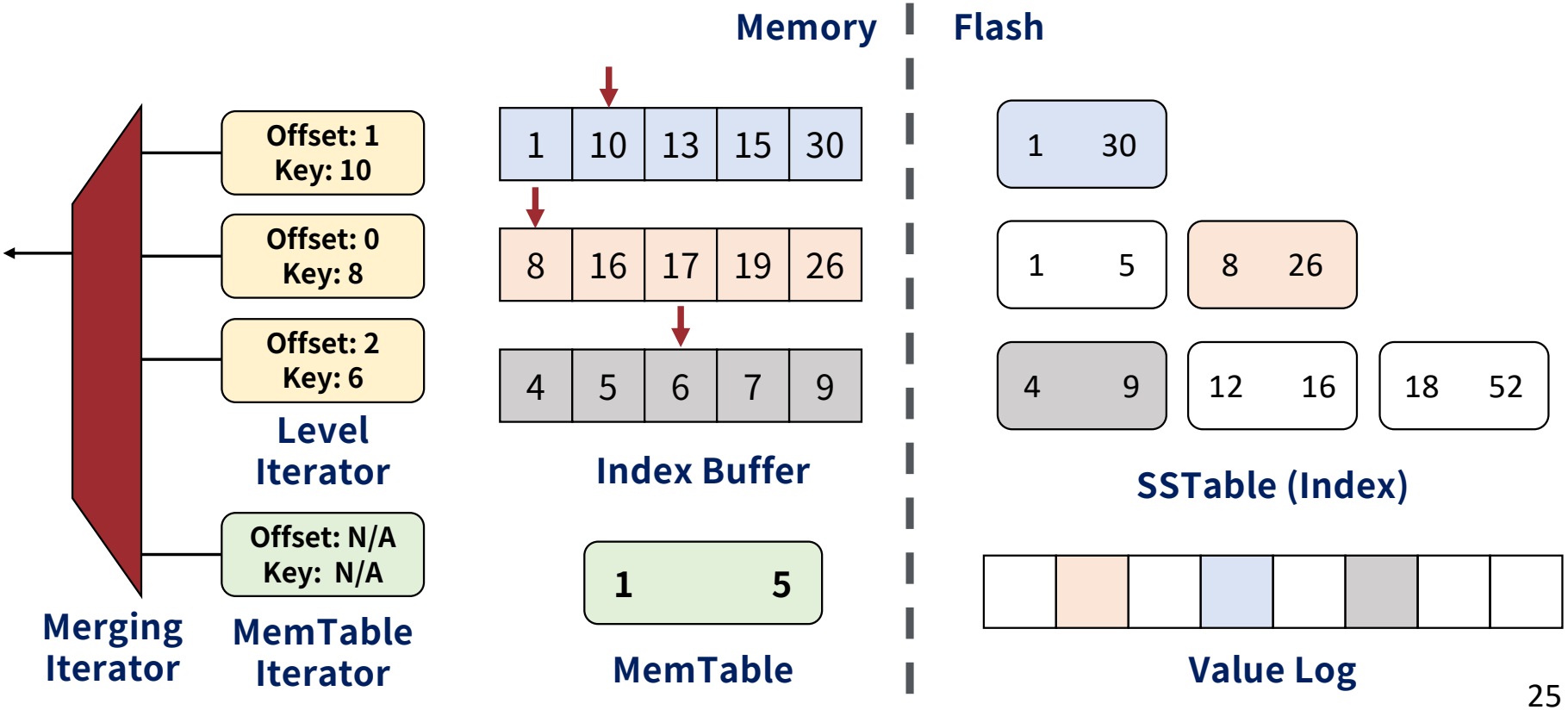
Range Query in LSM-tree-based KVSSD

- **On Seek with start_key=6**
  - Find Index in the search range (≥ 6)



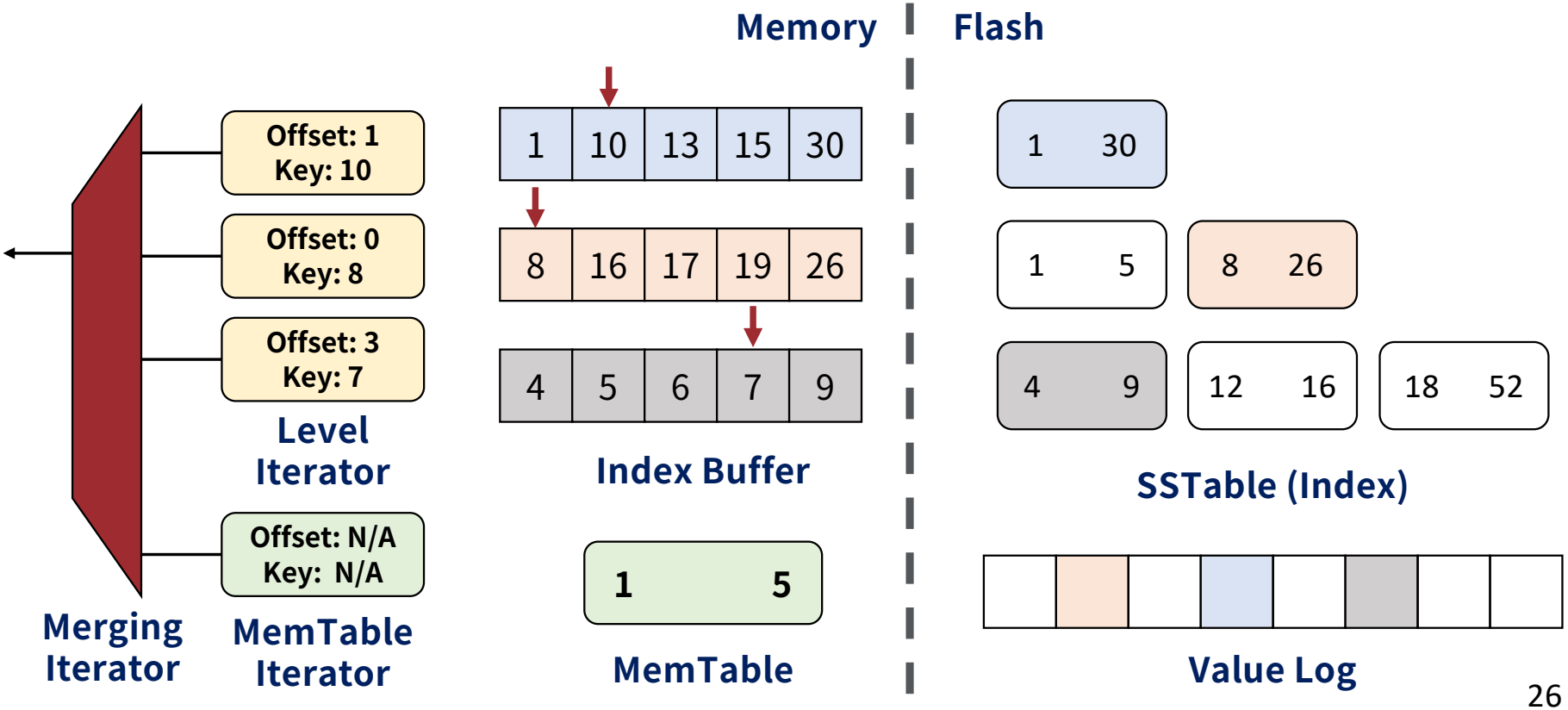
Range Query in LSM-tree-based KVSSD

- **On Next(),**



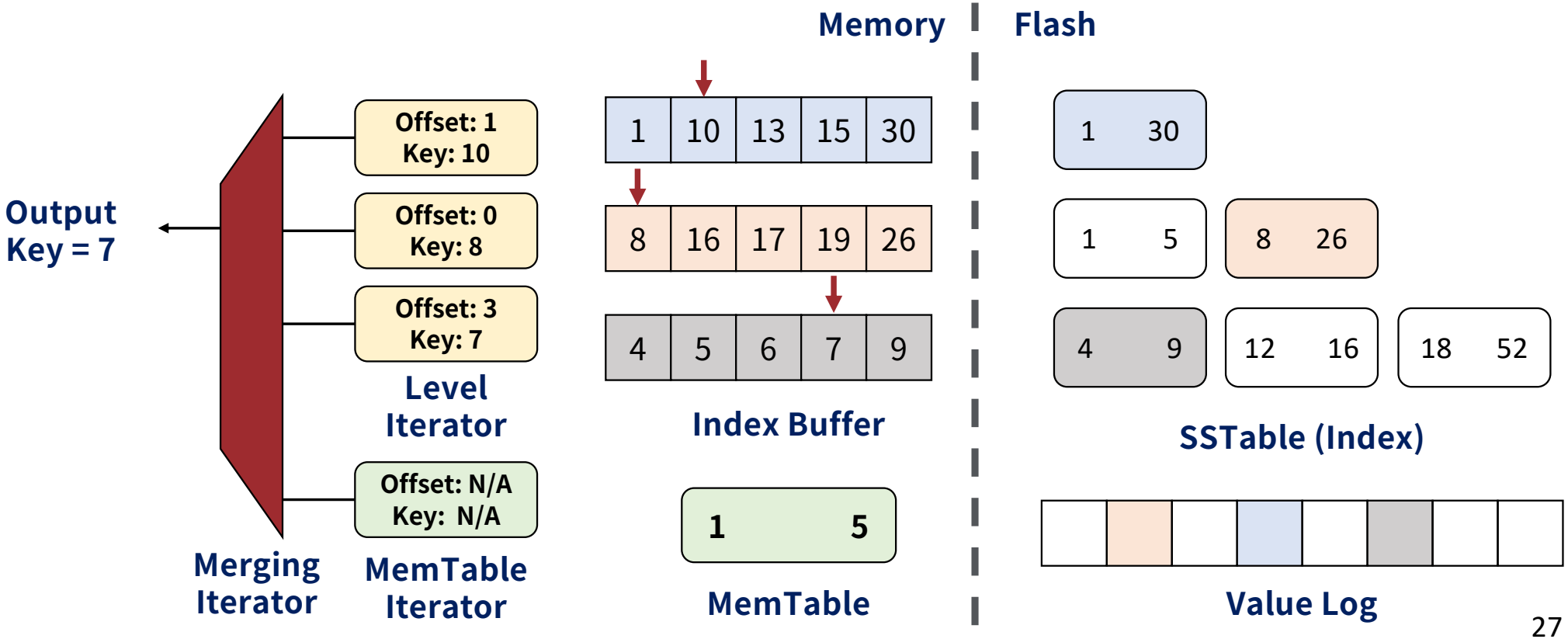
Range Query in LSM-tree-based KVSSD

- **On Next(),**



Range Query in LSM-tree-based KVSSD

- **On Next(),**



Problems of Current Iterator

Problems of Current Iterator

- **Problem #1 – Inconsistent Range Query**
  - Range queries are executed through **multiple iterator commands**
  - During range queries, LSM-tree can change by Put, Delete commands
  - How can the change in LSM-tree structure be handled?

Problems of Current Iterator

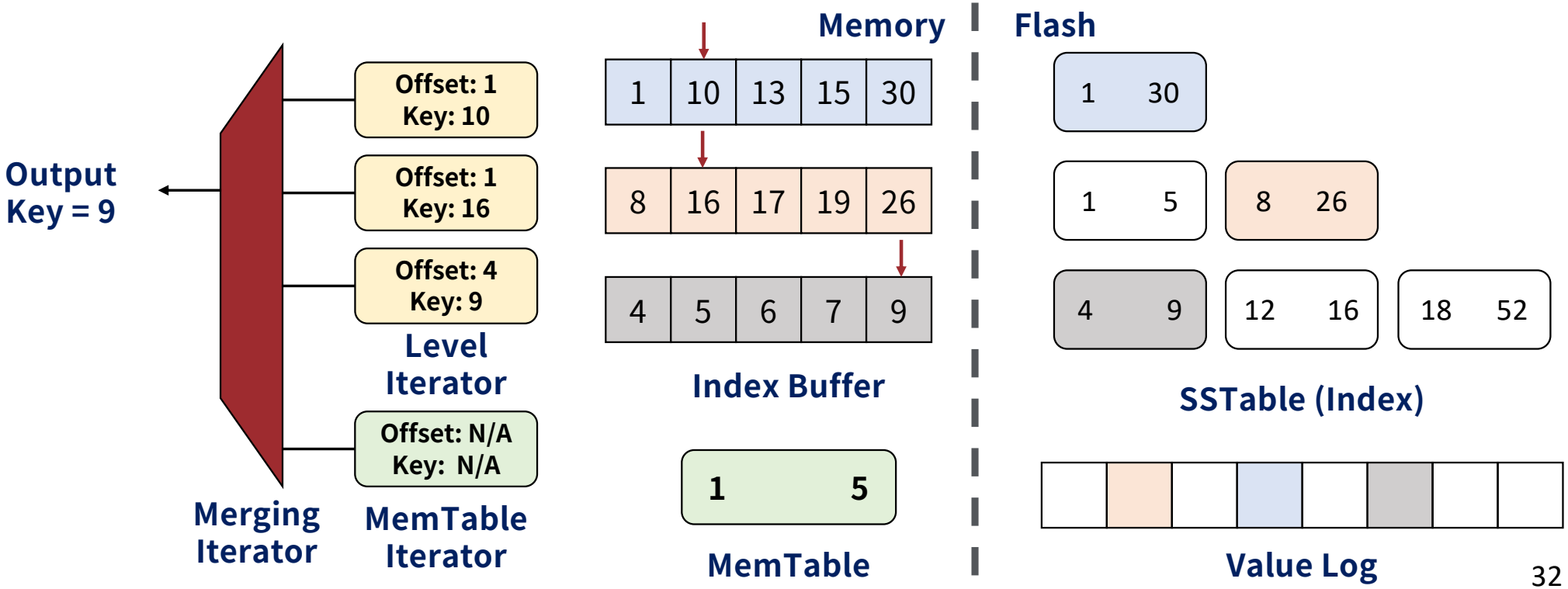
- **Problem #1 – Inconsistent Range Query**
  - Range queries are executed through **multiple iterator commands**
  - During range queries, LSM-tree can change by Put, Delete commands
  - How can the change in LSM-tree structure be handled?
- **Problem #2 – Long Tail Latency Problem**
  - During range queries, Iterator interface sometimes requires Index Read
  - This NAND access (Index Read) incurs long tail latency

Problems of Current Iterator

- **Problem #1 – Inconsistent Range Query**
  - Range queries are executed through **multiple iterator commands**
  - During range queries, LSM-tree can change by Put, Delete commands
  - How can the change in LSM-tree structure be handled?
- **Problem #2 – Long Tail Latency Problem**
  - During range queries, Iterator interface sometimes requires Index Read
  - This NAND access (Index Read) incurs long tail latency
- **Problem #3 – Poor Range Query Performance**
  - Every Seek, Next command entails Value Read from Value Log
  - This NAND Access (Value Read) incurs poor overall performance

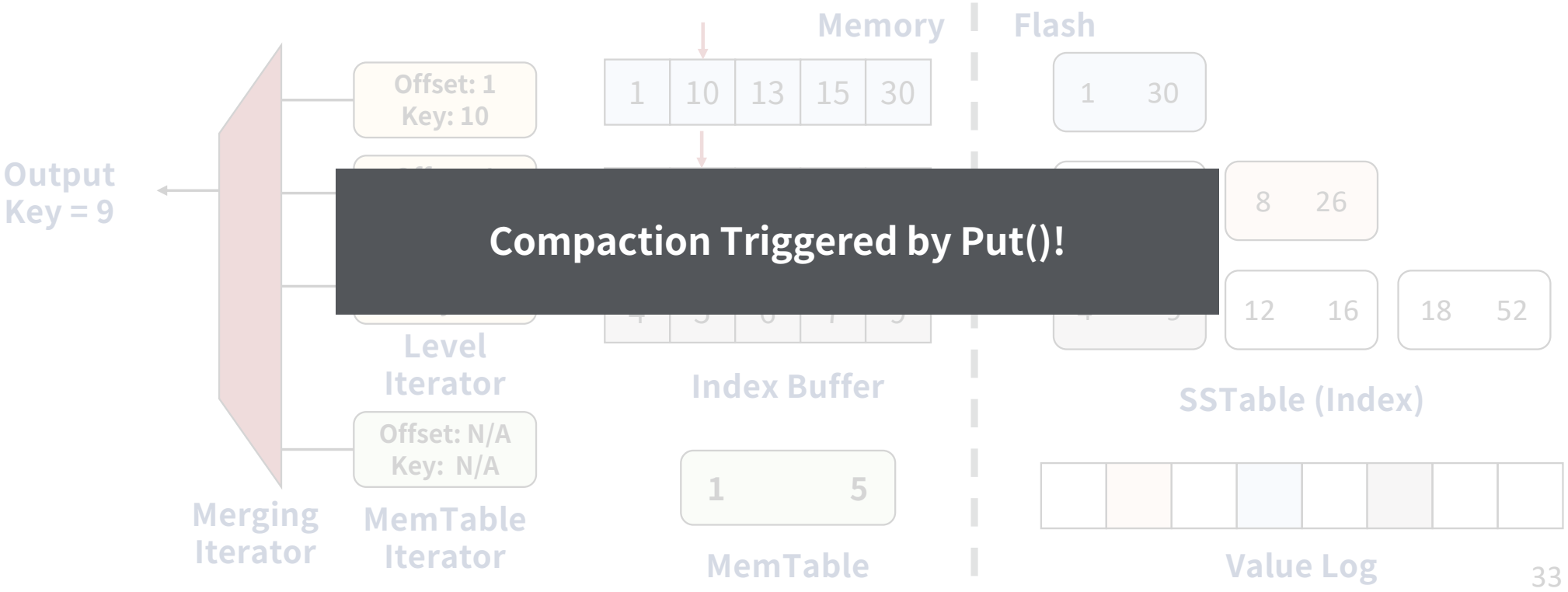
Problem #1 - Versioning

- **Versioning Problem**
  - Put(), Delete() can be issued in the middle of range query



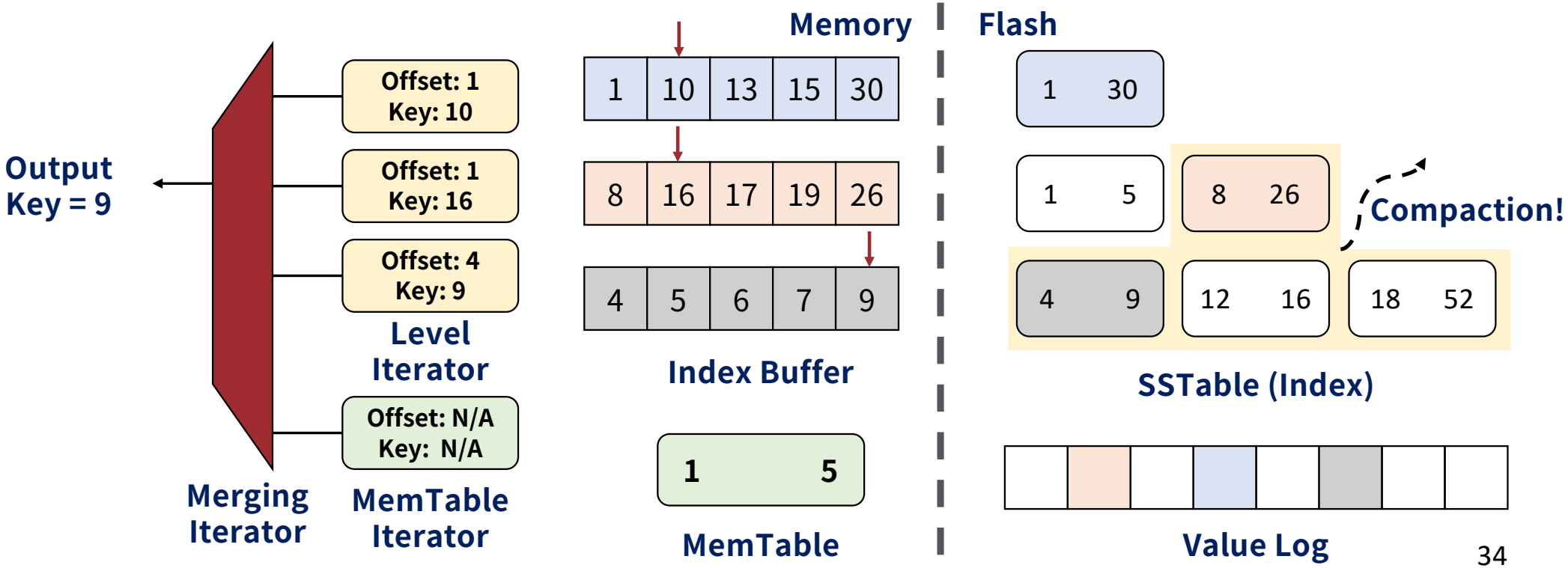
Problem #1 - Versioning

- Versioning Problem
  - Put(), Delete() can be issued in the middle of range query



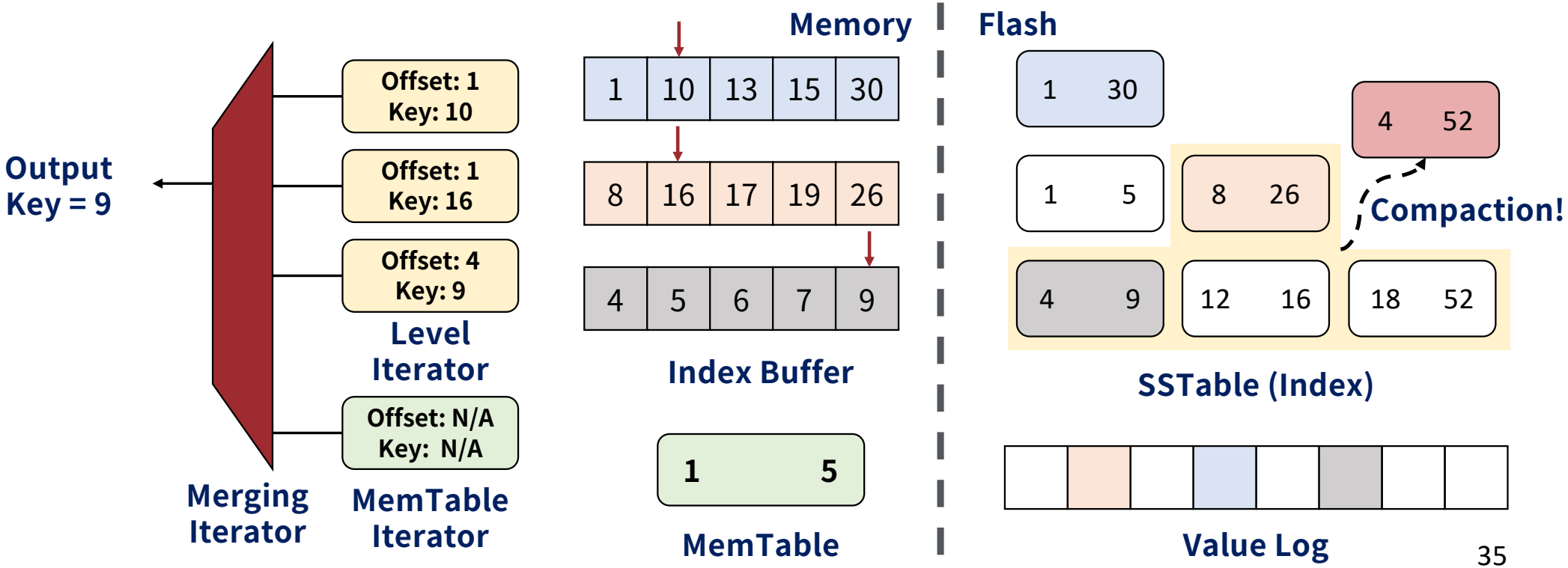
Problem #1 - Versioning

- **Versioning Problem**
  - Put(), Delete() can be issued in the middle of range query



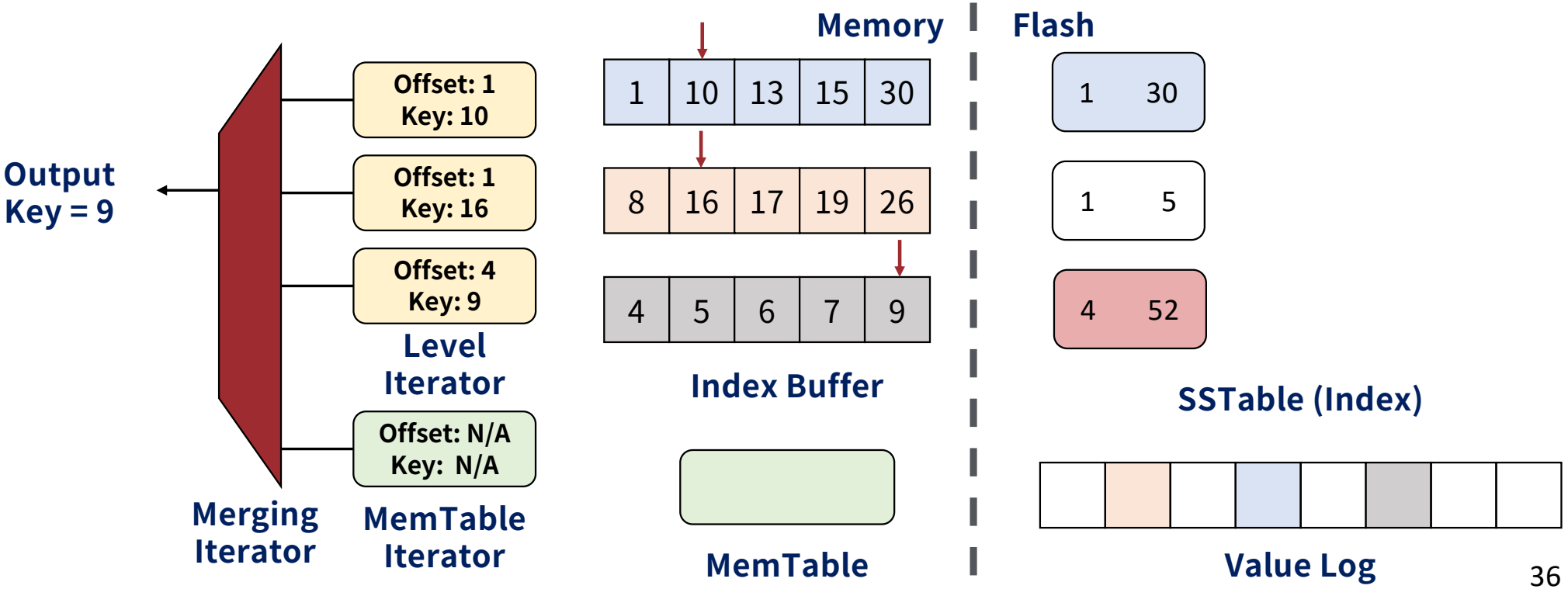
Problem #1 - Versioning

- **Versioning Problem**
  - Put(), Delete() can be issued in the middle of range query



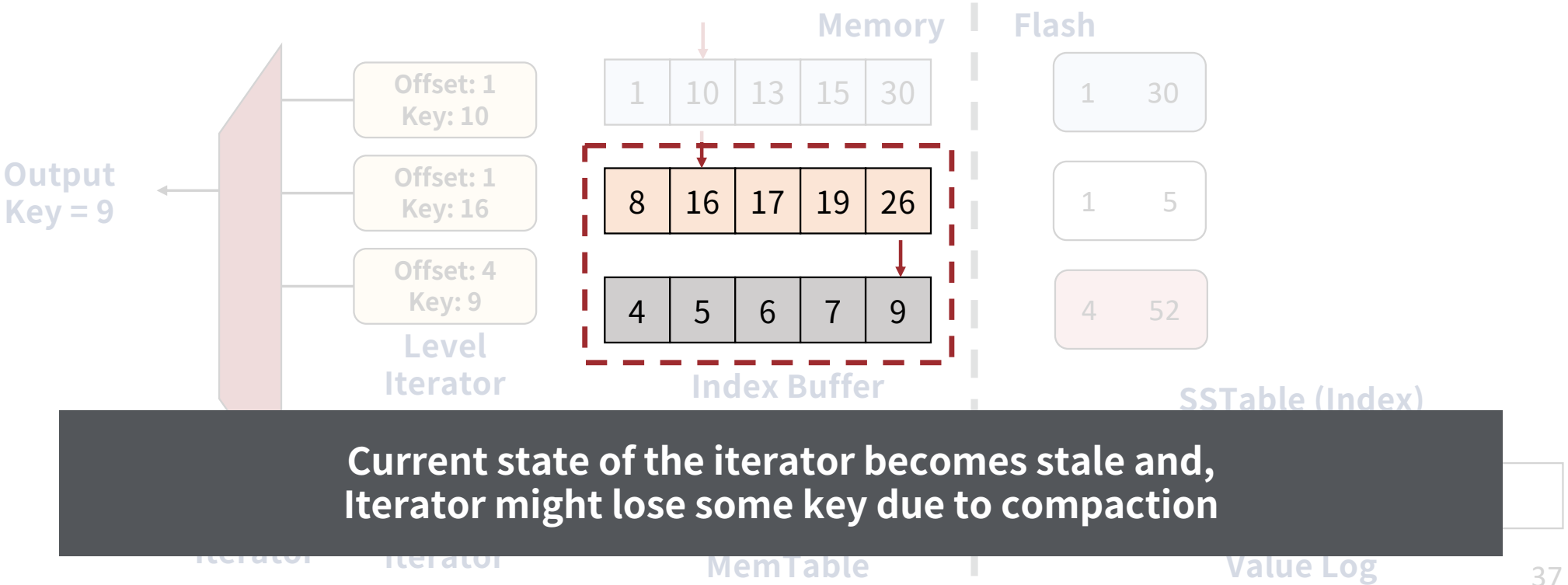
Problem #1 - Versioning

- **Versioning Problem**
  - Put(), Delete() can be issued in the middle of range query



Problem #1 - Versioning

- Versioning Problem
  - Put(), Delete() can be issued in the middle of range query



**Current state of the iterator becomes stale and,
Iterator might lose some key due to compaction**

Problem #1 - Versioning

- **Versioning Problem**
  - Put(), Delete() can be issued in the middle of range query
  - For this reason, host-side Key-Value stores support versioning in general
  - An iterator needs to see the version of the LSM-tree at its creation time.

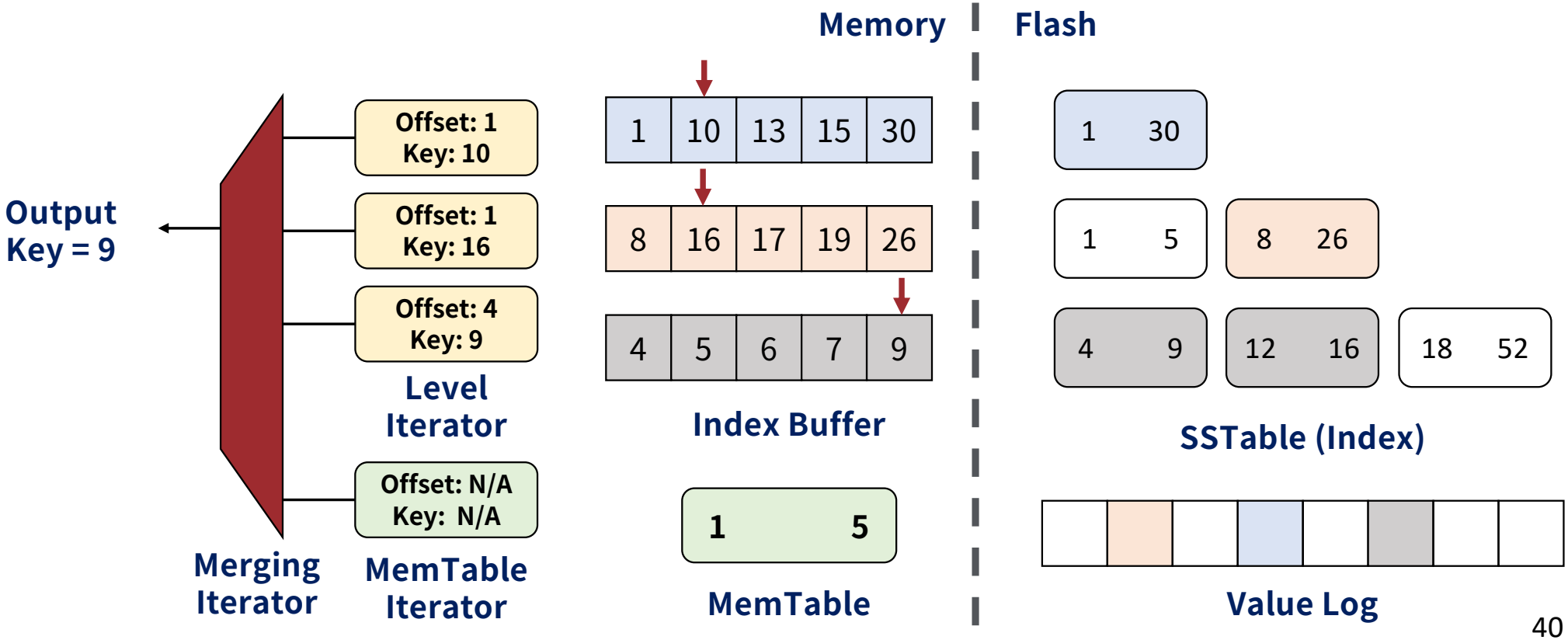
Problem #1 - Versioning

- **Versioning Problem**
  - Put(), Delete() can be issued in the middle of range query
  - For this reason, host-side Key-Value stores support versioning in general
  - An iterator needs to see the version of the LSM-tree at its creation time.

However, inside the device, memory-efficient versioning is needed!

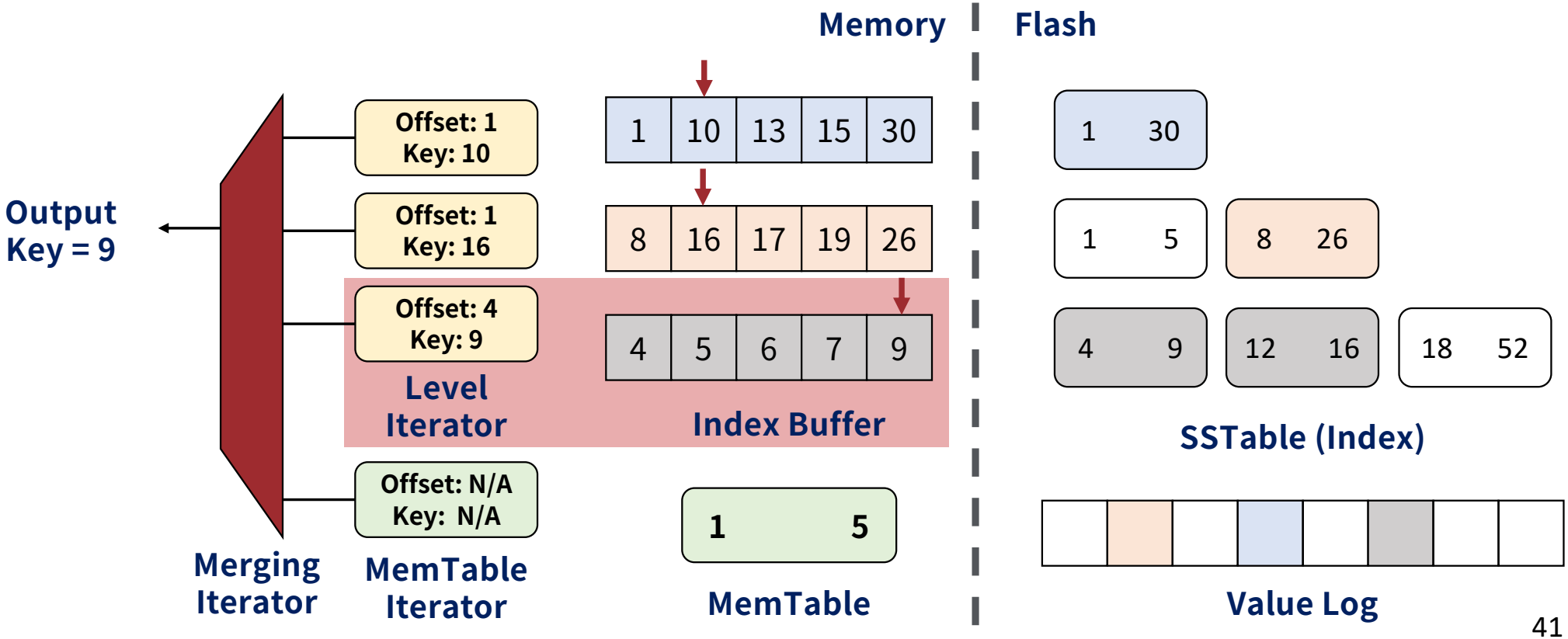
Problem #2 – Synchronous Index Read

- **Synchronous NAND Flash Access for Index Read**



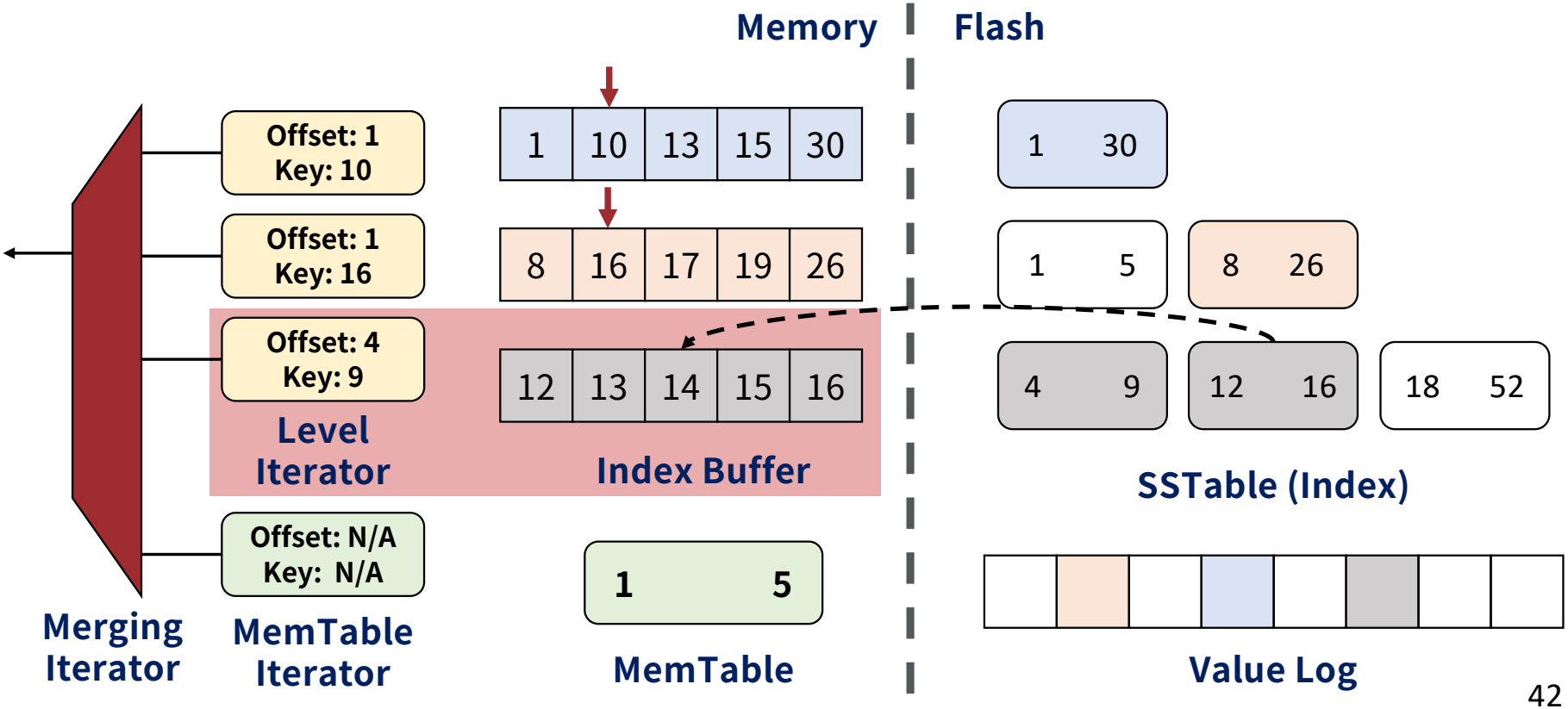
Problem #2 – Synchronous Index Read

- **Synchronous NAND Flash Access for Index Read**



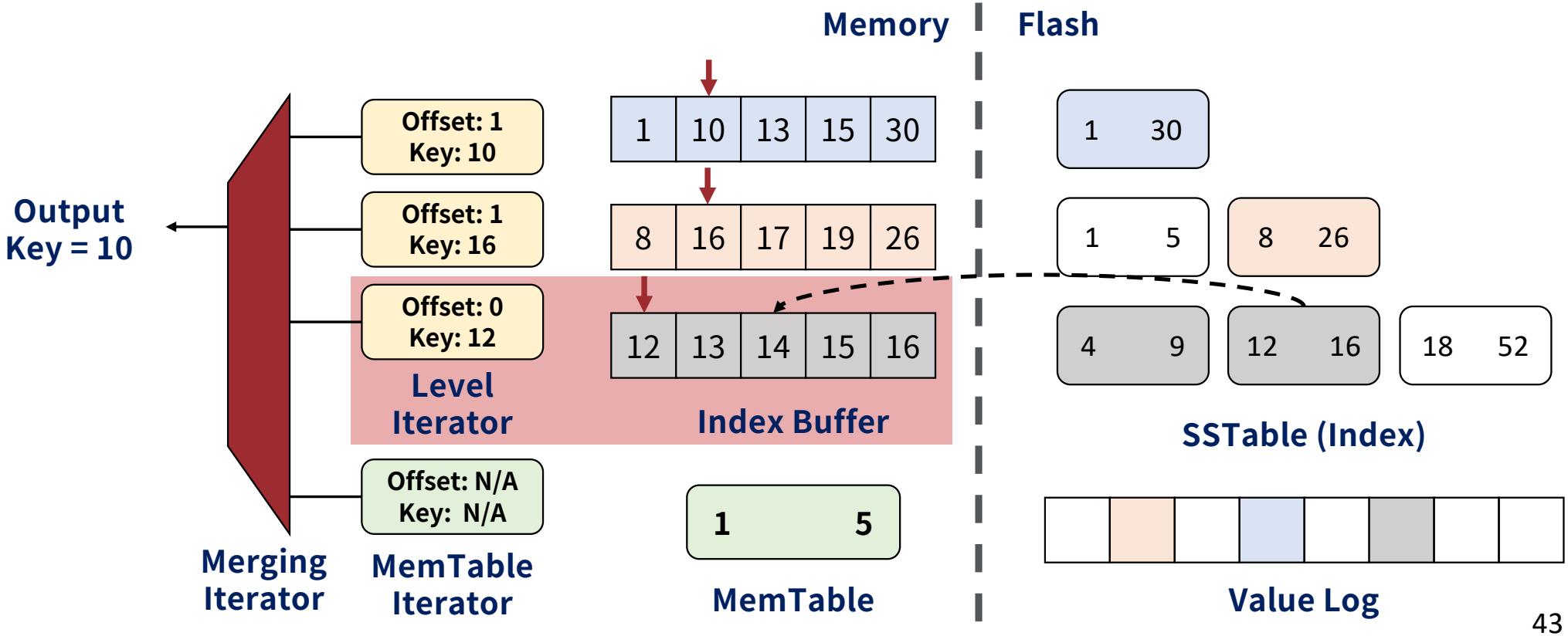
Problem #2 – Synchronous Index Read

- **Synchronous NAND Flash Access for Index Read**



Problem #2 – Synchronous Index Read

- **Synchronous NAND Flash Access for Index Read**



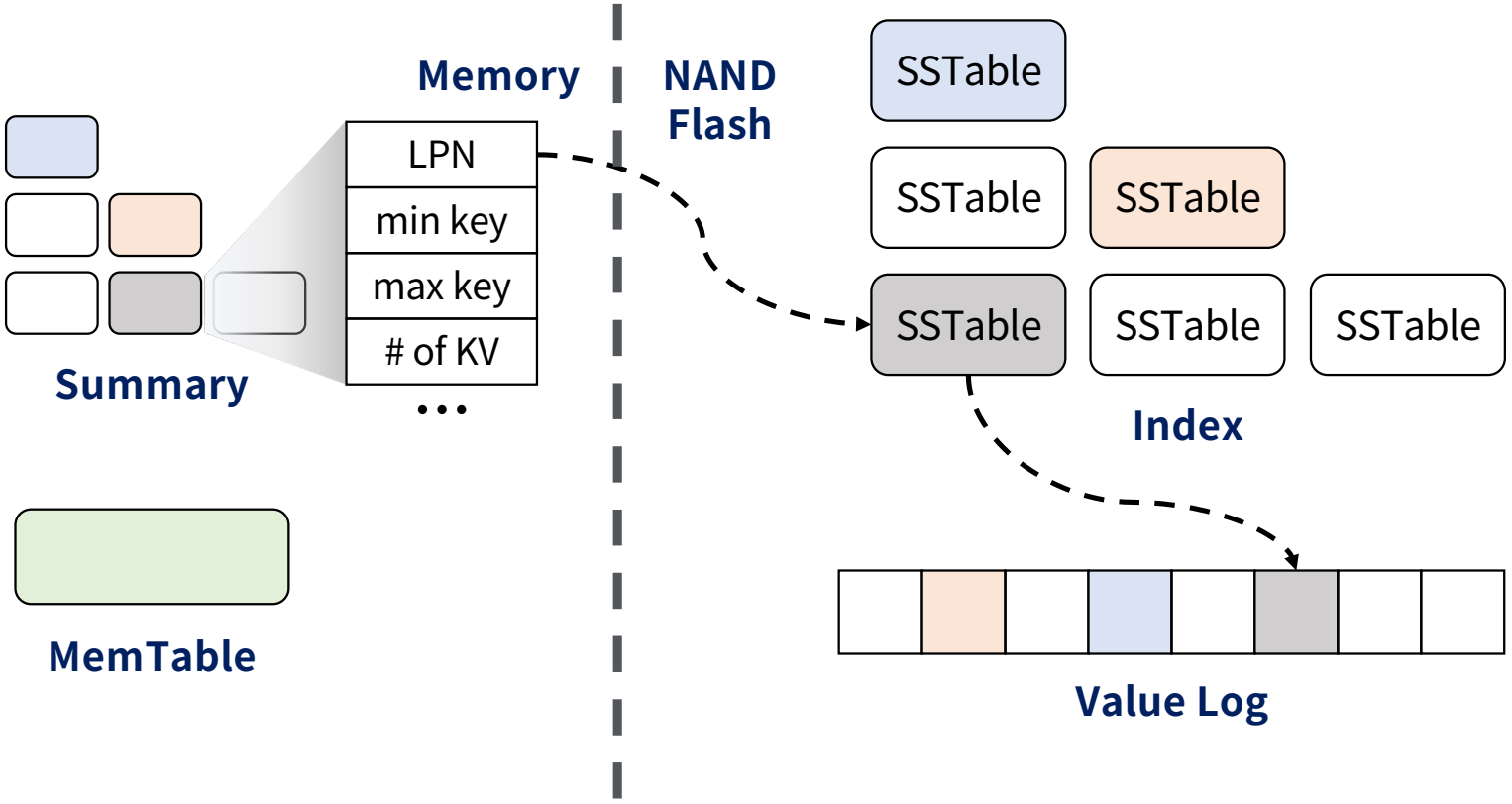
Problem #3 – Synchronous Value Read

- **Design Challenge #3 – NAND Flash Access for Value Read**
  - **Every Seek() and Next() command** requires NAND Flash Access for Value
  - Considering that NAND Flash access is much slower than the other steps, synchronous NAND Flash access for Value may woefully aggravate the overall performance

Design of IterKVSSD

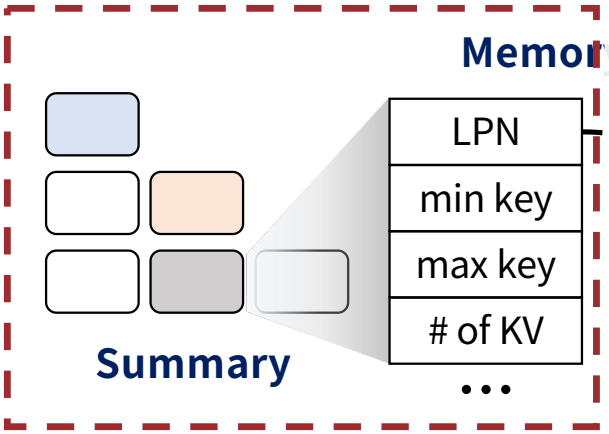
Memory Efficient Versioning Data Structure

- How to support Versioning inside the device?



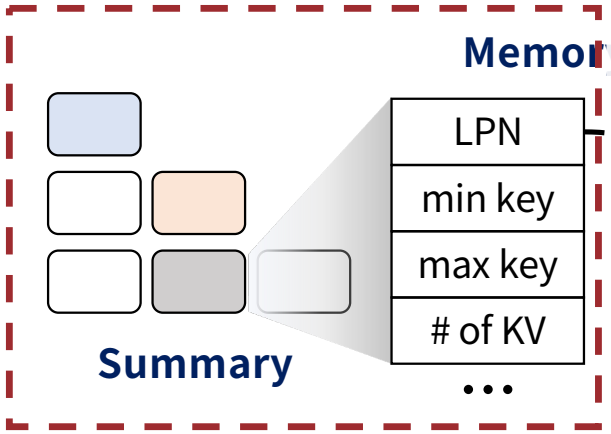
Memory Efficient Versioning Data Structure

- How to support Versioning inside the device?



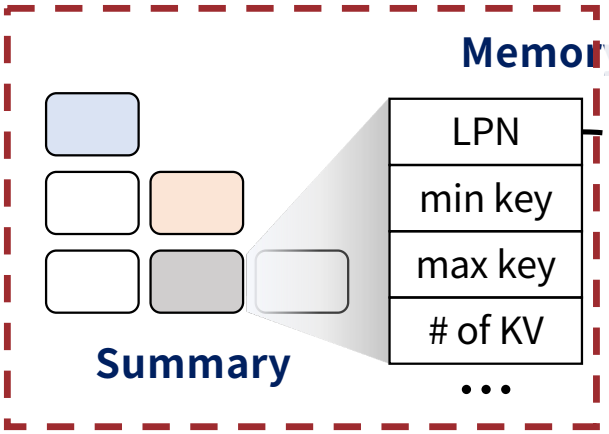
- Summary represents the state of LSM-tree

Memory Efficient Versioning Data Structure



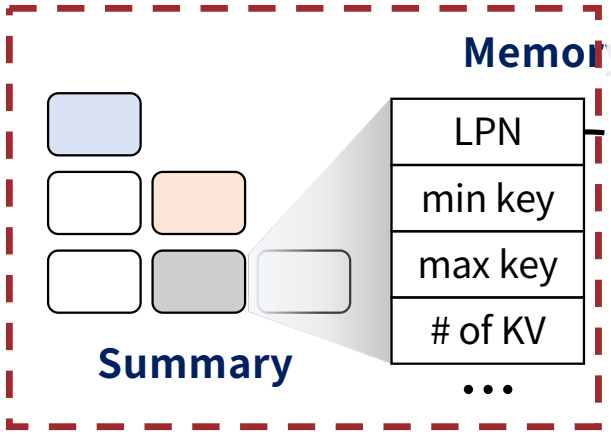
- Summary represents the state of LSM-tree
- Summary Size $\propto$ # of SSTables

Memory Efficient Versioning Data Structure



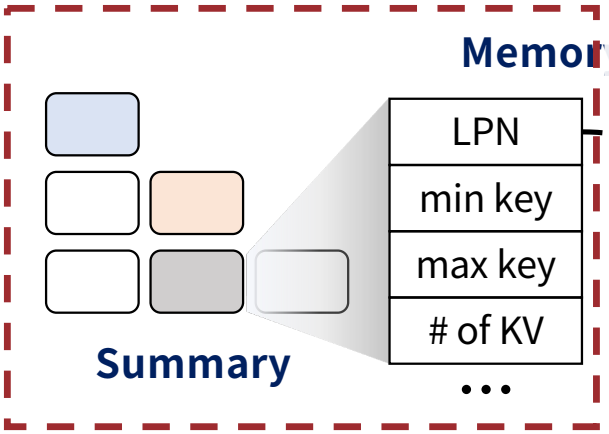
- Summary represents the state of LSM-tree
- Summary Size $\propto$ # of SSTables
- In our setup,
  1. Summary Entry Size = 44B
  2. # of SSTables $\approx$ 65,000

Memory Efficient Versioning Data Structure



- Summary represents the state of LSM-tree
- Summary Size $\propto$ # of SSTables
- In our setup,
  1. Summary Entry Size = 44B
  2. # of SSTables $\approx$ 65,000
  3. Total Size $\approx$ 2.7MB

Memory Efficient Versioning Data Structure

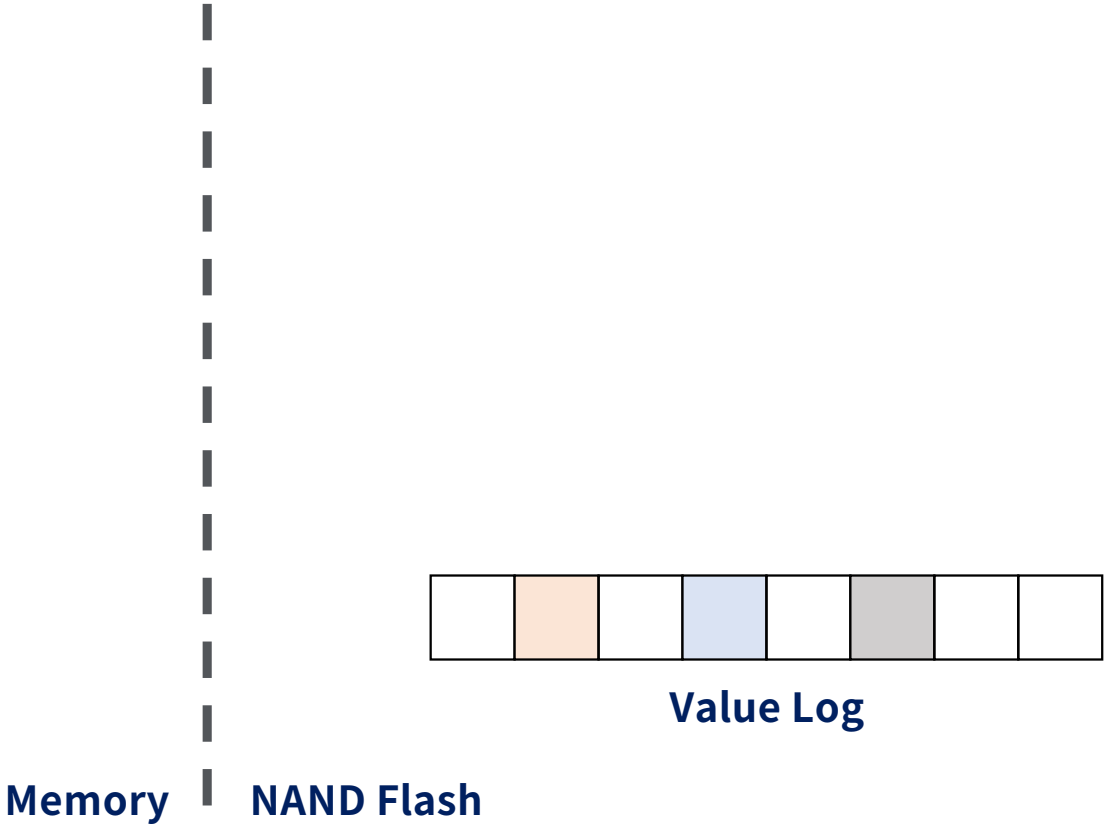


- Summary represents the state of LSM-tree
- Summary Size $\propto$ # of SSTables
- In our setup,
  1. Summary Entry Size = 44B
  2. # of SSTables $\approx$ 65,000
  3. Total Size $\approx$ 2.7MB

Keeping Summary for every Iterator is too expensive!

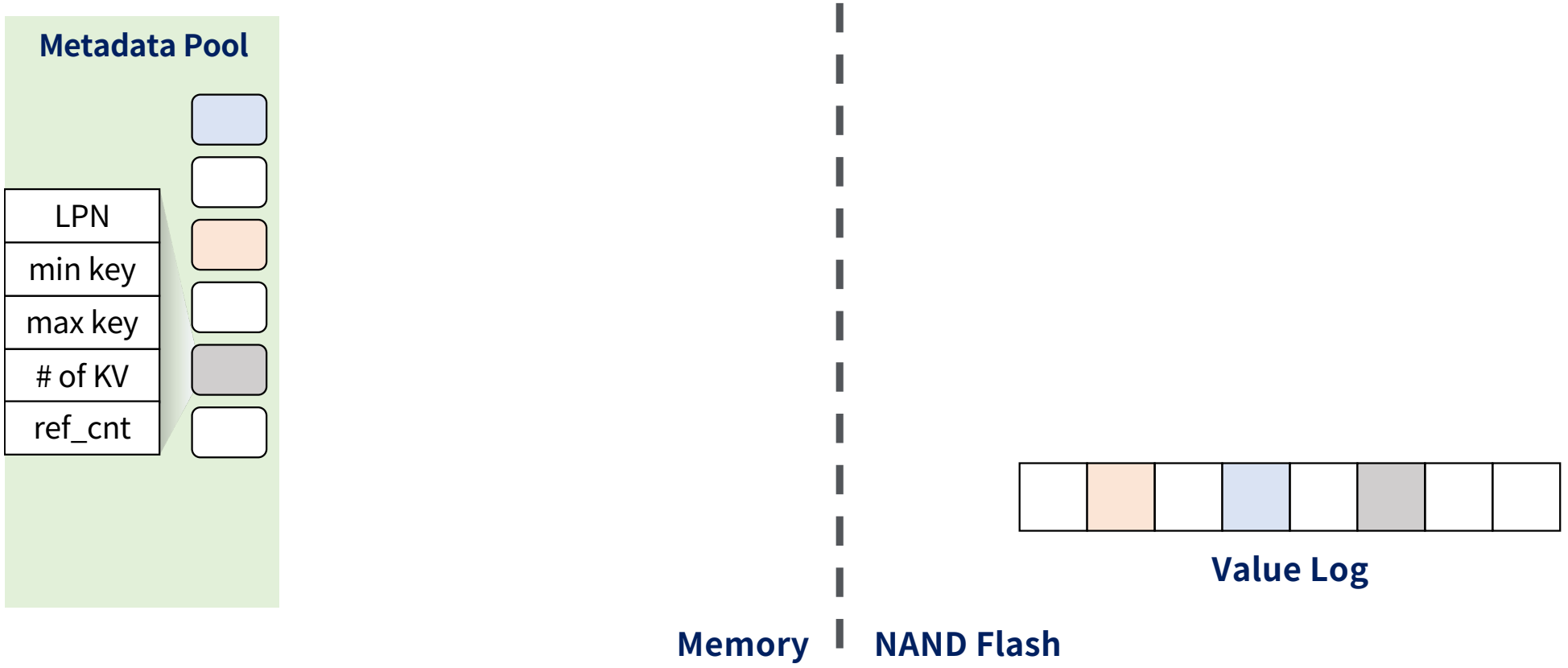
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



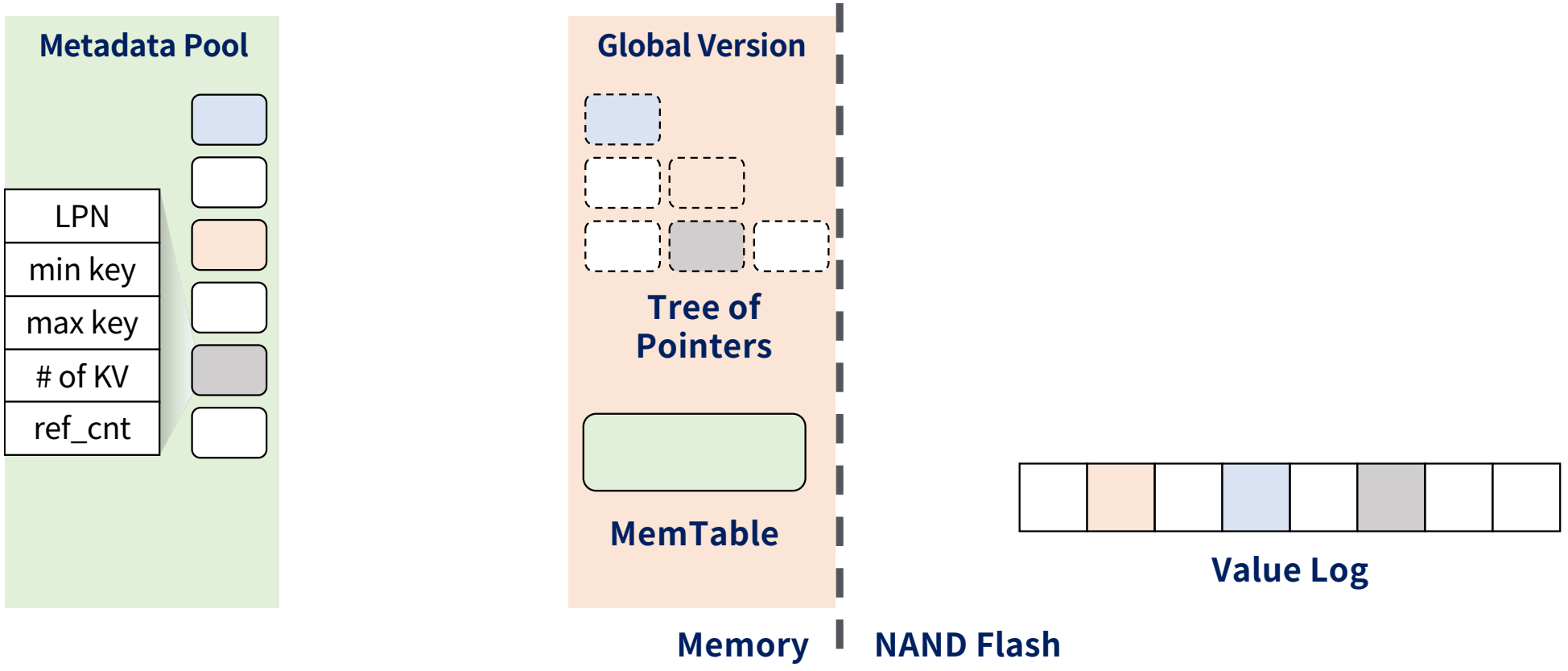
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



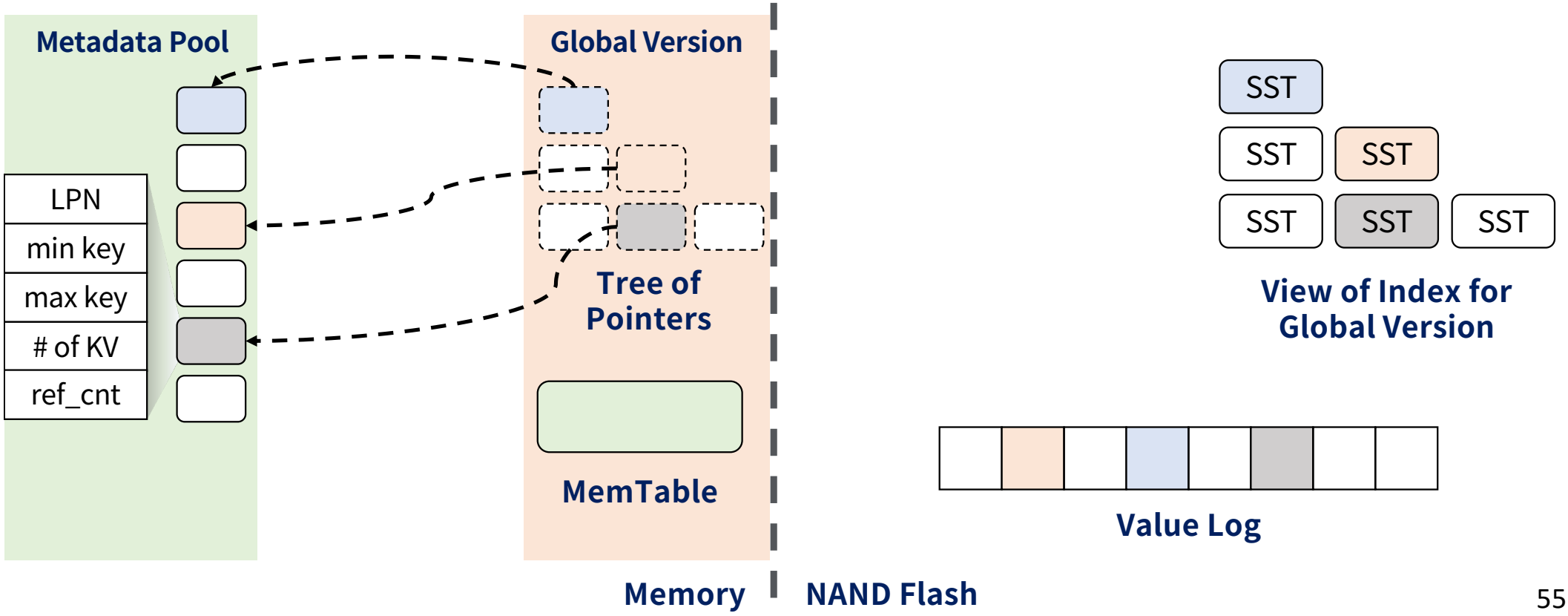
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



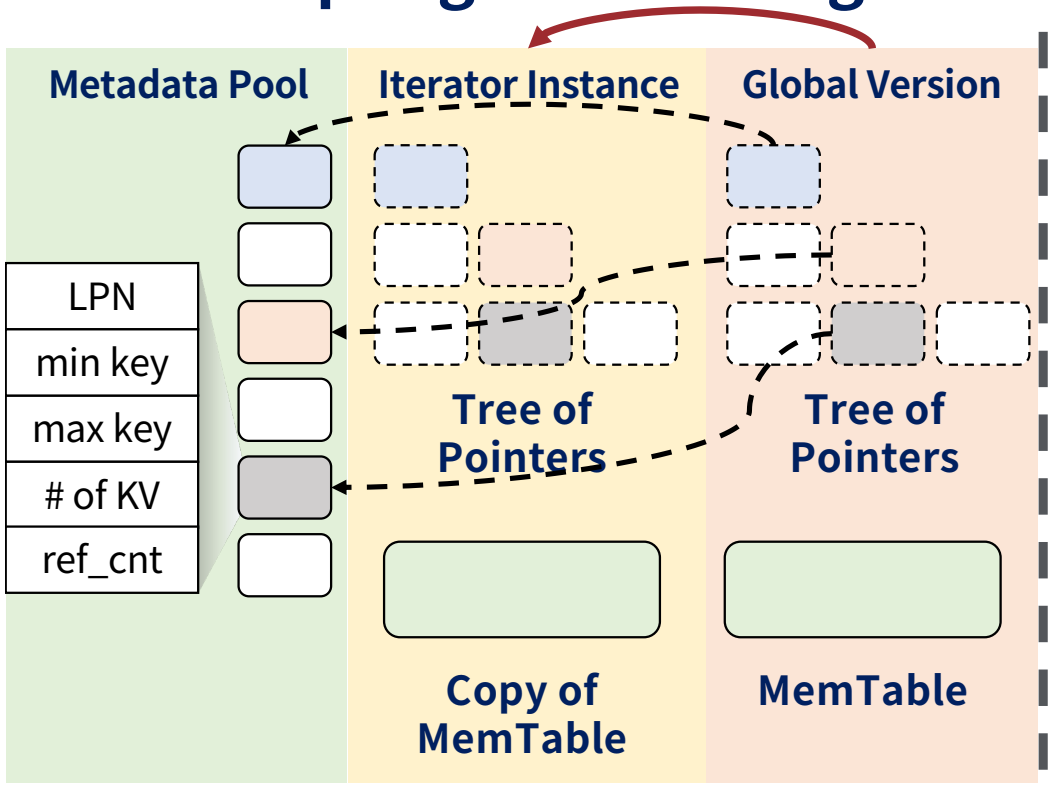
Memory Efficient Versioning Data Structure

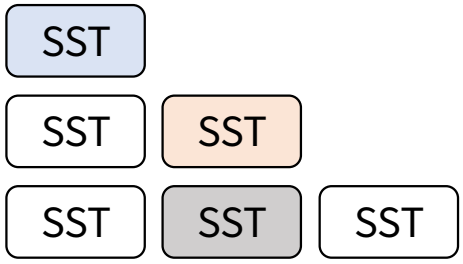
- **Decoupling and Pooling Metadata from Version Data Structure**

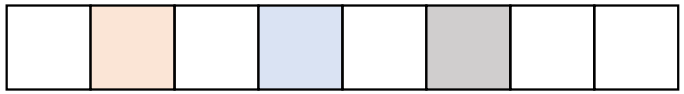


Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**

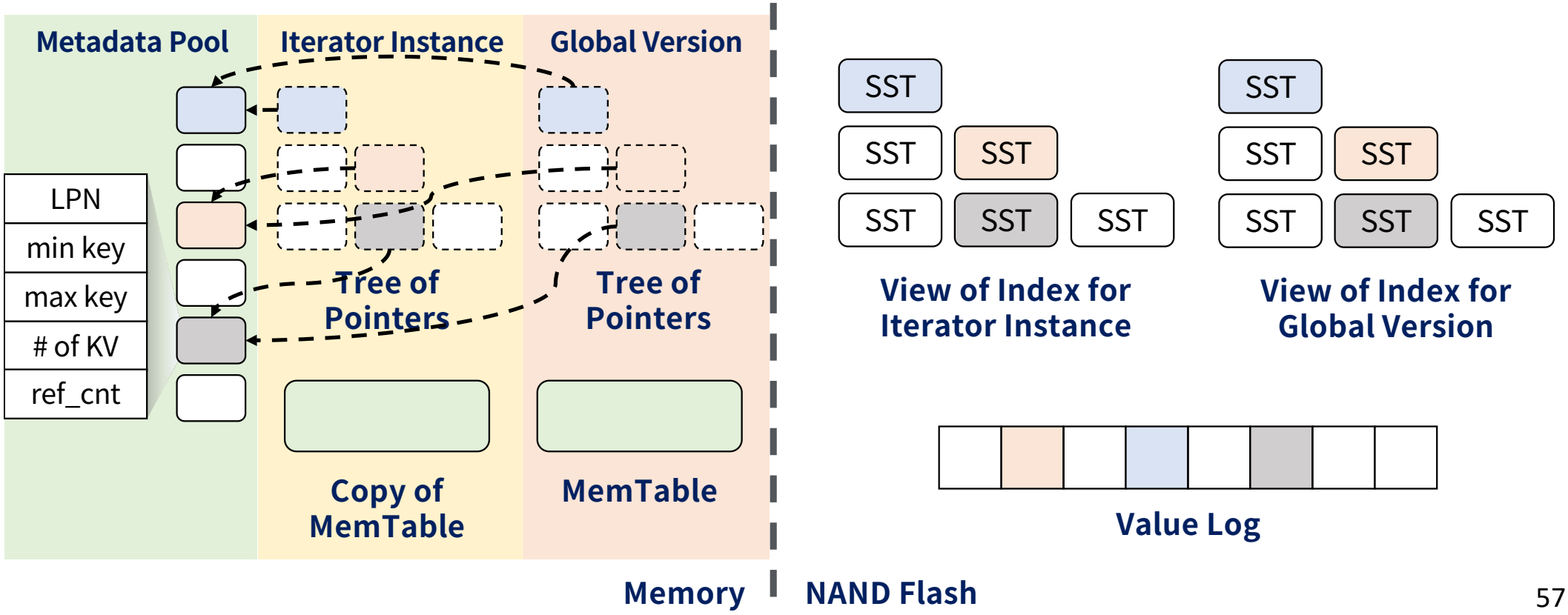






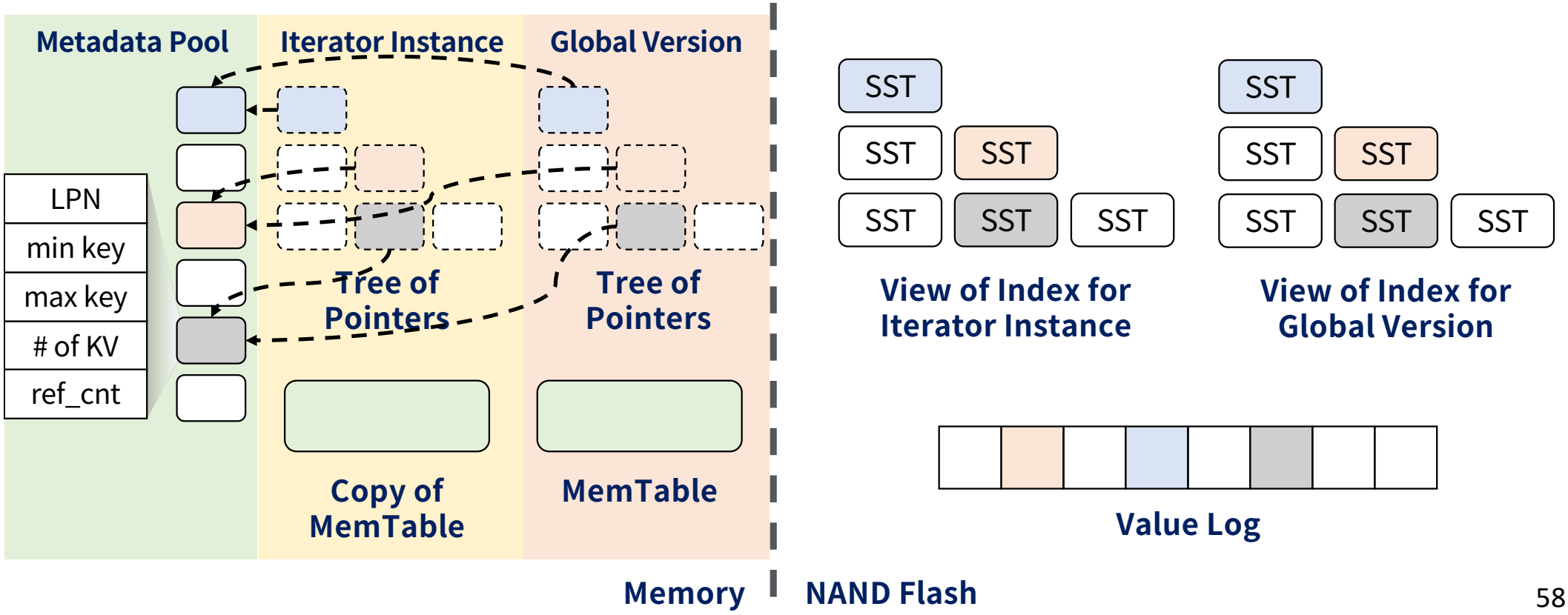
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



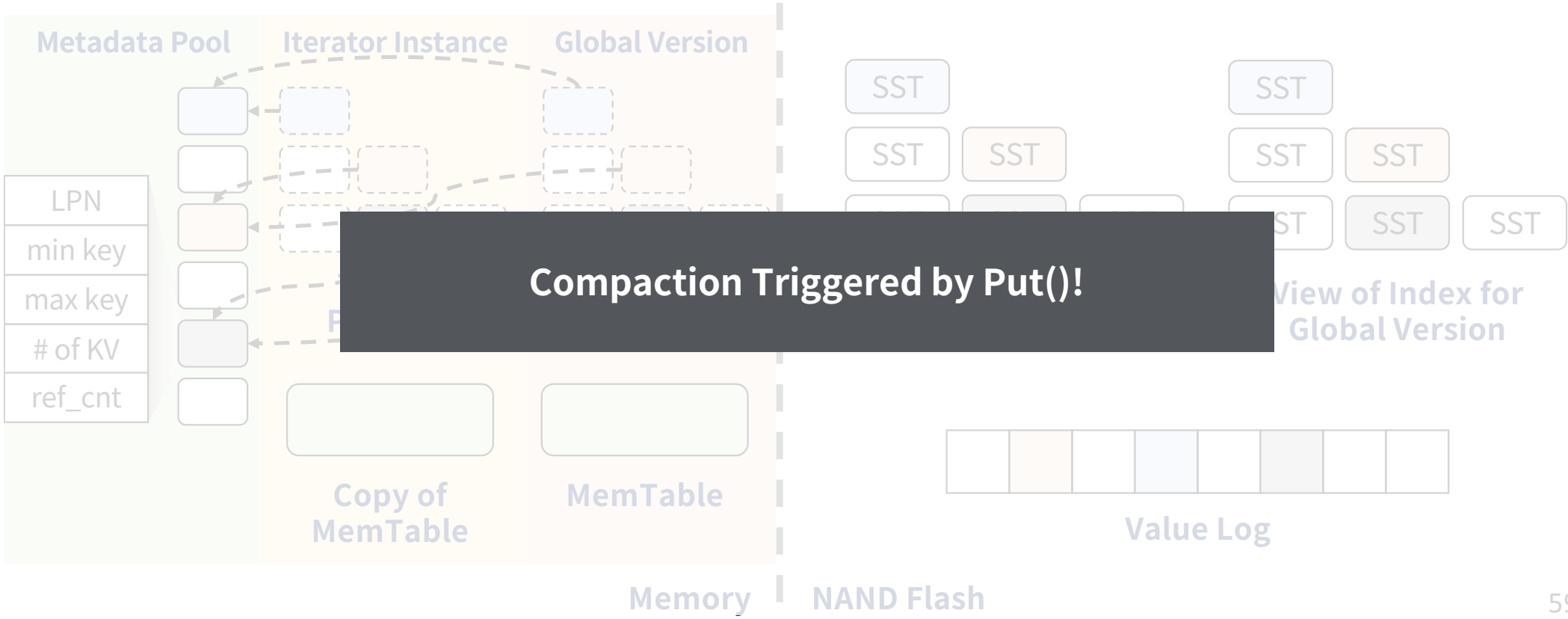
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



Memory Efficient Versioning Data Structure

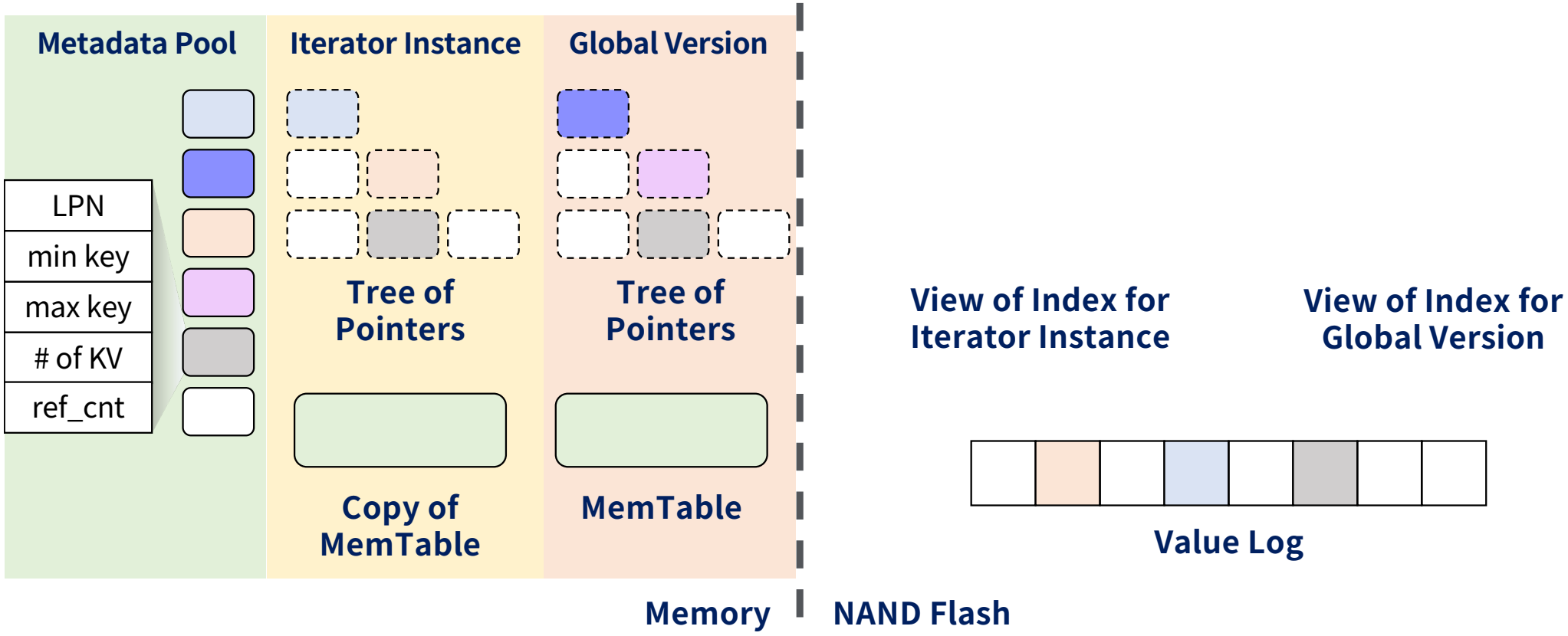
- Decoupling and Pooling Metadata from Version Data Structure



Compaction Triggered by Put()!

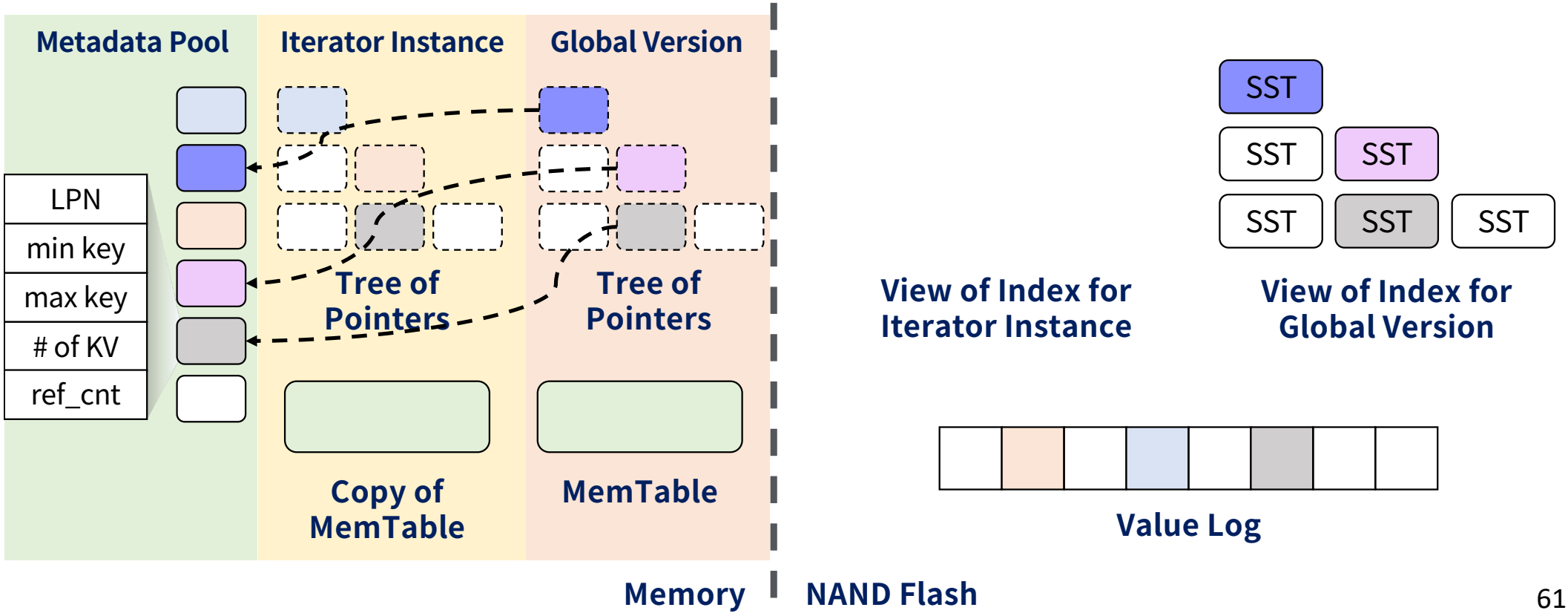
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



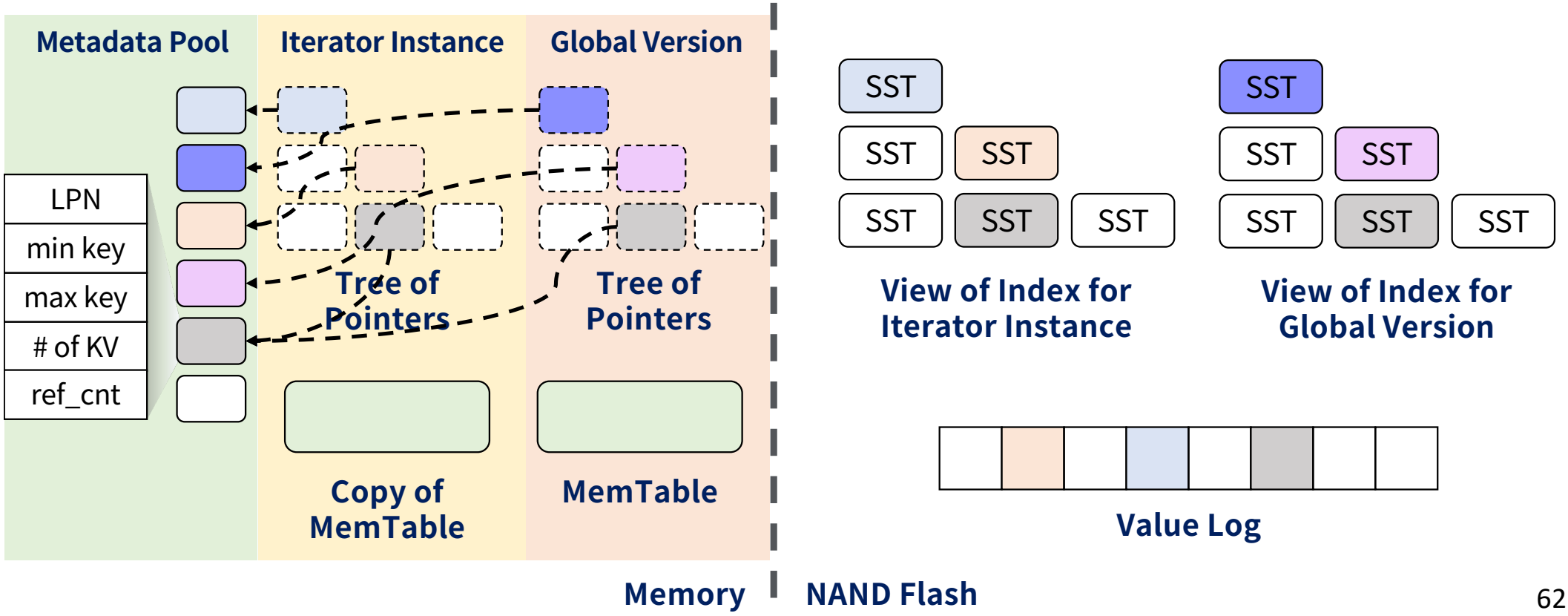
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



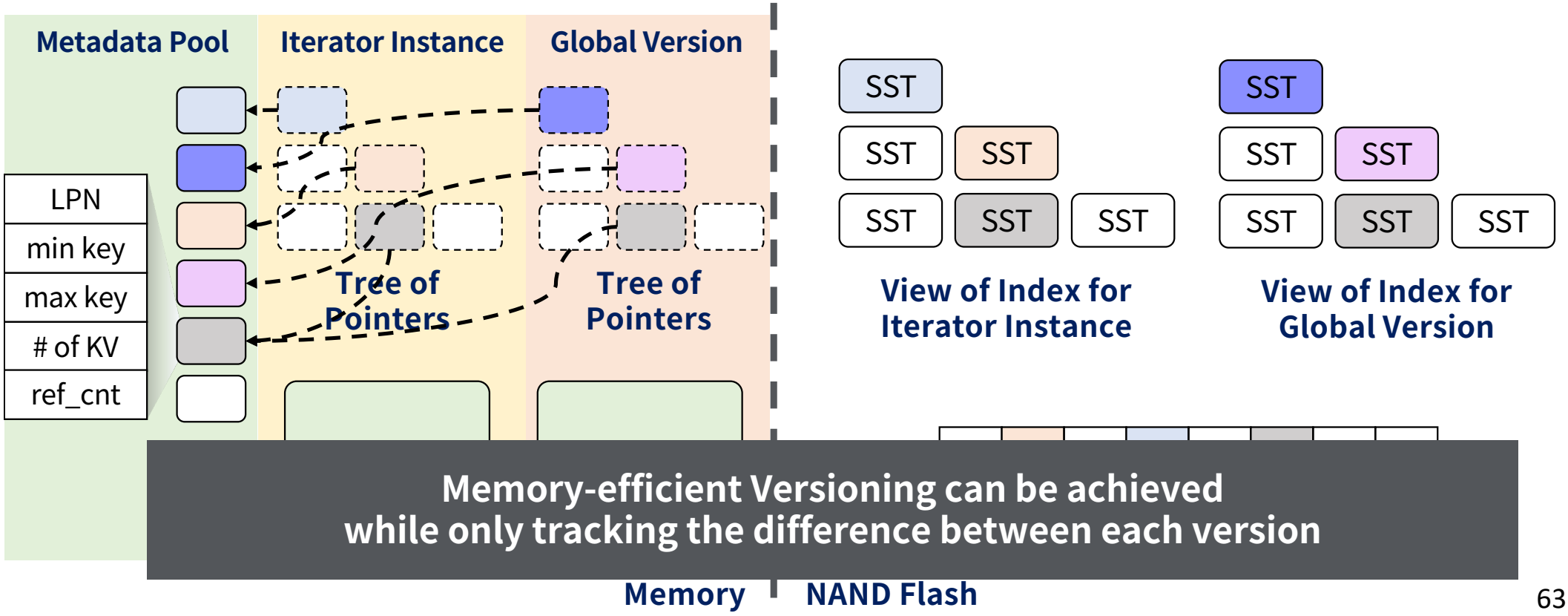
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



How to Reduce NAND Flash Access Cost?

- **Key ideas to minimize synchronous NAND Flash access penalty**

How to Reduce NAND Flash Access Cost?

- **Key ideas to minimize synchronous NAND Flash access penalty**

1. **Range query sequentially retrieves Key-Value pairs in a row**

How to Reduce NAND Flash Access Cost?

- **Key ideas to minimize synchronous NAND Flash access penalty**

1. **Range query sequentially retrieves Key-Value pairs in a row**
   - KVSSD knows what is the next key to read

How to Reduce NAND Flash Access Cost?

- **Key ideas to minimize synchronous NAND Flash access penalty**

1. **Range query sequentially retrieves Key-Value pairs in a row**
   - KVSSD knows what is the next key to read
2. **Key-Value semantic is enabled inside the device**

How to Reduce NAND Flash Access Cost?

- **Key ideas to minimize synchronous NAND Flash access penalty**

1. **Range query sequentially retrieves Key-Value pairs in a row**
   - KVSSD knows what is the next key to read
2. **Key-Value semantic is enabled inside the device**
   - KVSSD knows where in physical memory the next Key-Value pairs are stored

How to Reduce NAND Flash Access Cost?

- **Key ideas to minimize synchronous NAND Flash access penalty**

1. **Range query sequentially retrieves Key-Value pairs in a row**
   - KVSSD knows what is the next key to read
2. **Key-Value semantic is enabled inside the device**
   - KVSSD knows where in physical memory the next Key-Value pairs are stored
3. **Inside the SSDs, there are multiple independent channel controllers**

How to Reduce NAND Flash Access Cost?

- **Key ideas to minimize synchronous NAND Flash access penalty**

1. **Range query sequentially retrieves Key-Value pairs in a row**
   - KVSSD knows what is the next key to read
2. **Key-Value semantic is enabled inside the device**
   - KVSSD knows where in physical memory the next Key-Value pairs are stored
3. **Inside the SSDs, there are multiple independent channel controllers**
   - KVSSD can overlap NAND Flash access with processing storage protocol and the other steps in parallel

How to Reduce NAND Flash Access Cost?

- **Want to hide synchronous NAND Flash Access Penalty**

How to Reduce NAND Flash Access Cost?

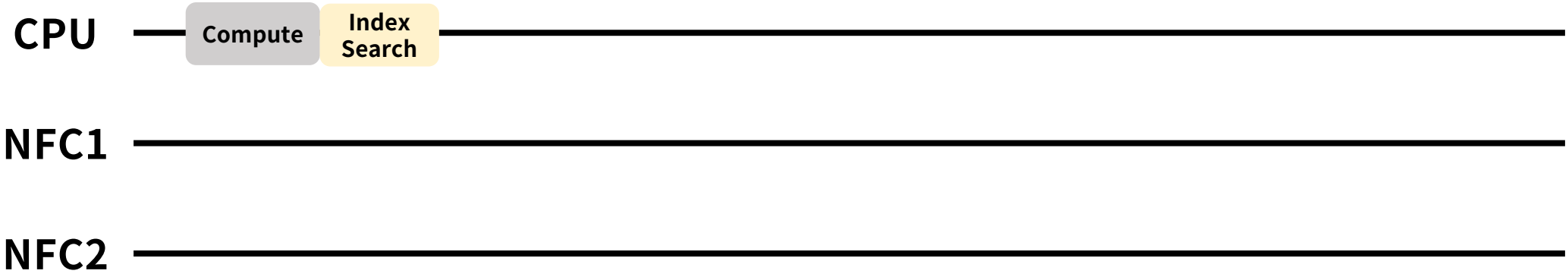
- **Want to hide synchronous NAND Flash Access Penalty**
  - On every Seek and Next commands,
    1. **Compute** for processing the storage protocol command
    2. **Read Index** from NAND Flash, **if necessary**
    3. **Search and Compare** the Index for each level
    4. **Read Value** from NAND flash
    5. **DMA transfer over PCIe** to return the result Key-Value pair

How to Reduce NAND Flash Access Cost?

- **Want to hide synchronous NAND Flash Access Penalty**
  - On every Seek and Next commands,
    1. **Compute** for processing the storage protocol command
    2. **Read Index** from NAND Flash, **if necessary**
    3. **Search and Compare** the Index for each level
    4. **Read Value** from NAND flash
    5. **DMA transfer over PCIe** to return the result Key-Value pair

Index Prefetching

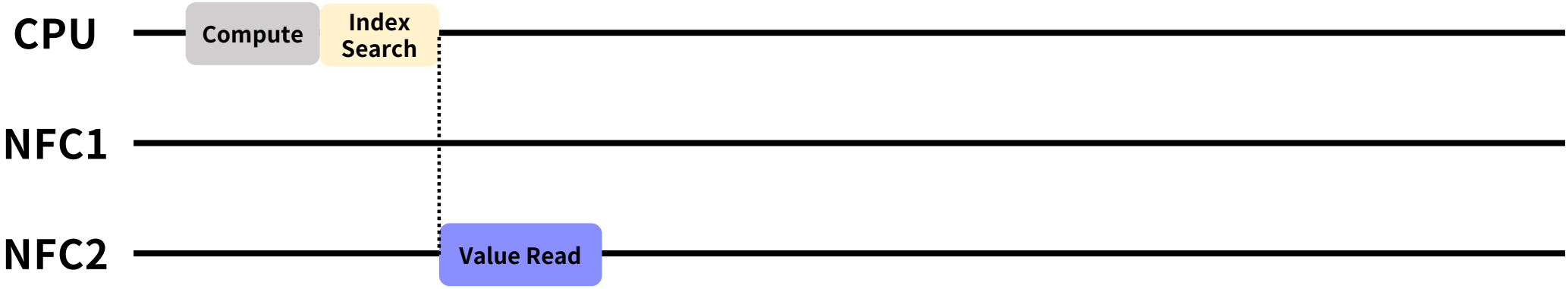
- **Index Prefetching**



- The device can calculate when it becomes needed to read index from NAND.
- Therefore, it can overlap the reading time with other process in advance by exploiting hardware (NFCs).

Index Prefetching

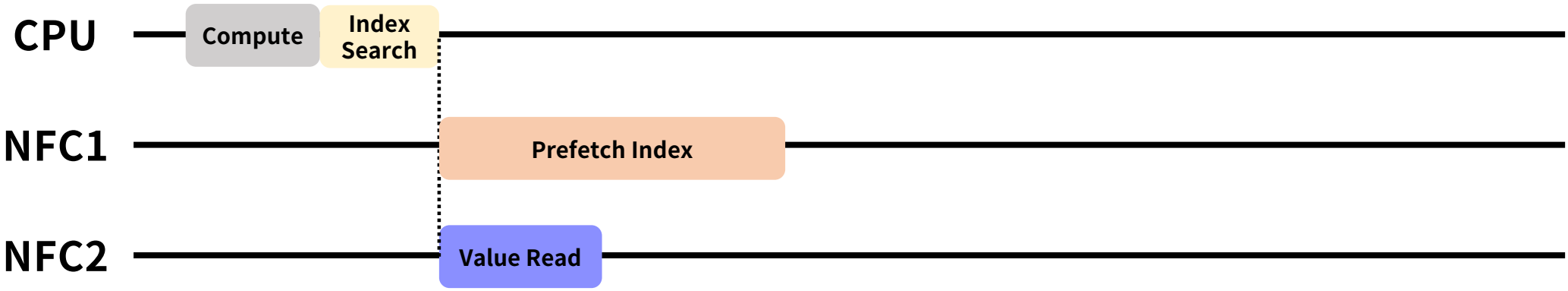
- **Index Prefetching**



- The device can calculate when it becomes needed to read index from NAND.
- Therefore, it can overlap the reading time with other process in advance by exploiting hardware (NFCs).

Index Prefetching

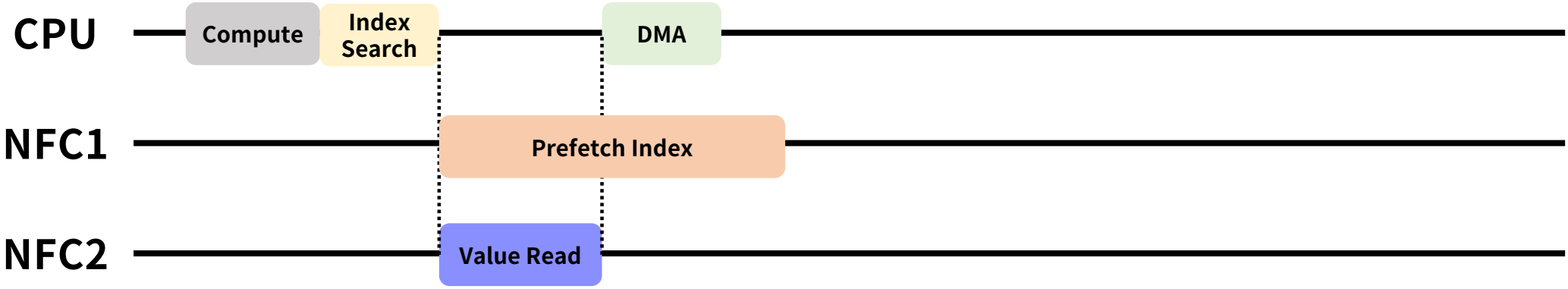
- **Index Prefetching**



- The device can calculate when it becomes needed to read index from NAND.
- Therefore, it can overlap the reading time with other process in advance by exploiting hardware (NFCs).

Index Prefetching

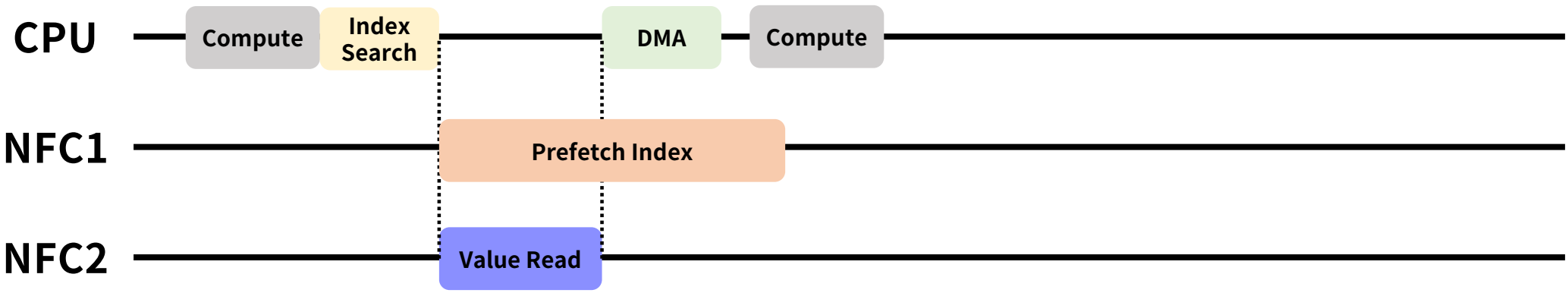
- **Index Prefetching**



- The device can calculate when it becomes needed to read index from NAND.
- Therefore, it can overlap the reading time with other process in advance by exploiting hardware (NFCs).

Index Prefetching

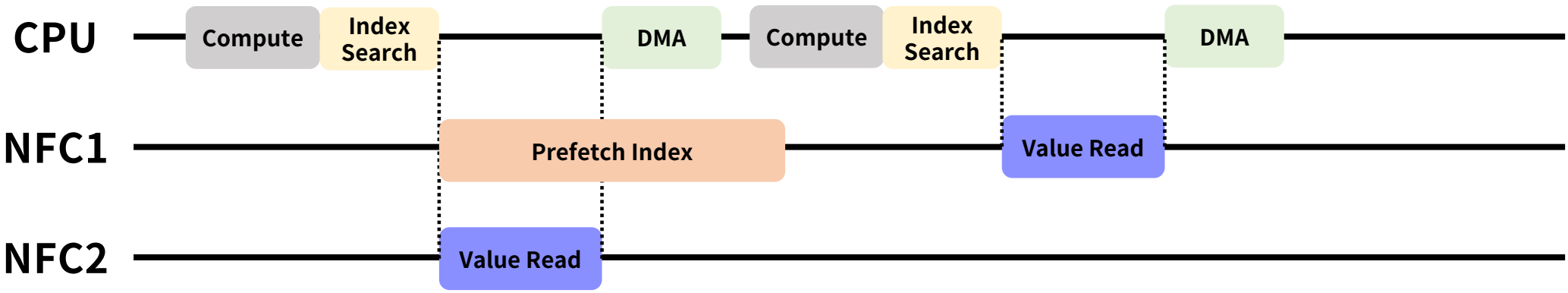
- **Index Prefetching**



- The device can calculate when it becomes needed to read index from NAND.
- Therefore, it can overlap the reading time with other process in advance by exploiting hardware (NFCs).

Index Prefetching

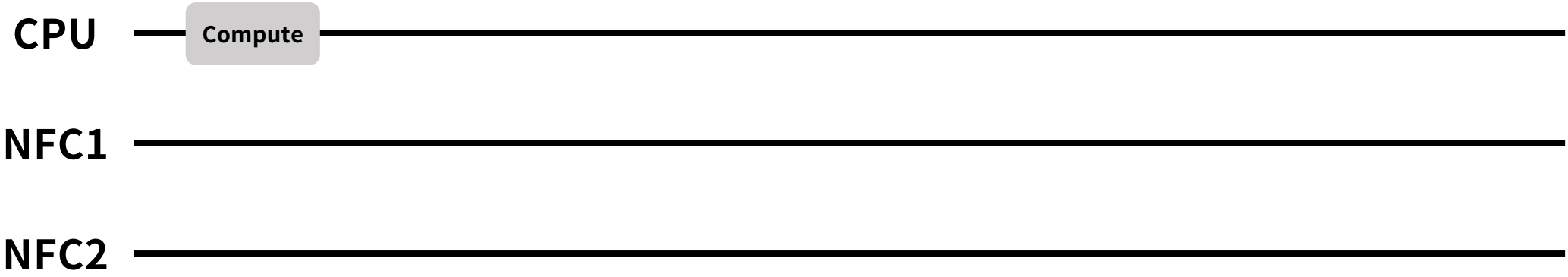
- **Index Prefetching**



- The device can calculate when it becomes needed to read index from NAND.
- Therefore, it can overlap the reading time with other process in advance by exploiting hardware (NFCs).

Value Prefetching

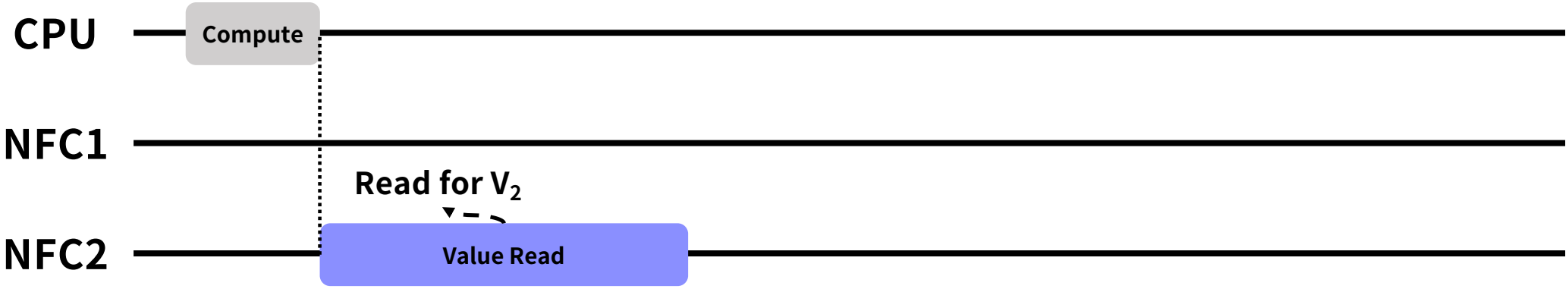
- **Value Prefetching**



- When processing **Nth** Next command, the device issues prefetch request for the **(N+D)th** Next command in the future (**D**: Prefetching Degree).
- Therefore, it can overlap the reading time with other process in advance by exploiting hardware (NFCs).

Value Prefetching

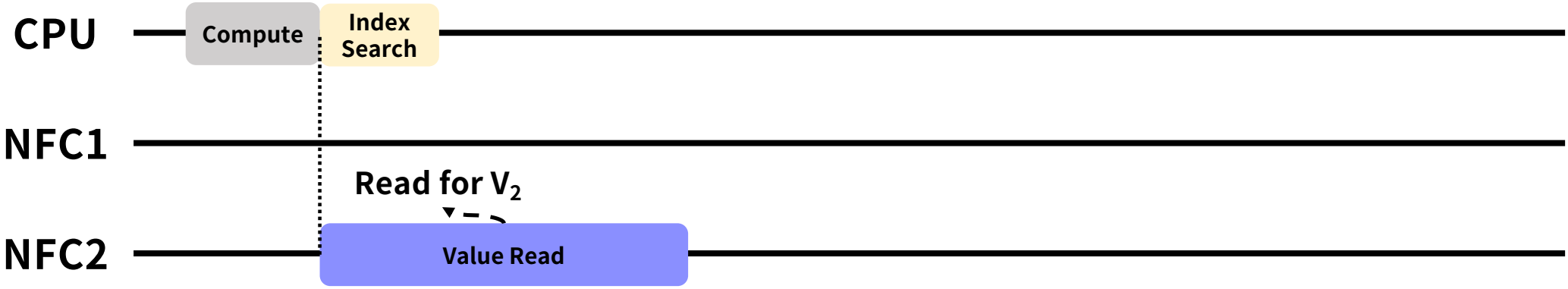
- **Value Prefetching**



- When processing **Nth** Next command, the device issues prefetch request for the **(N+D)th** Next command in the future (**D**: Prefetching Degree).
- Therefore, it can overlap the reading time with other process in advance by exploiting hardware (NFCs).

Value Prefetching

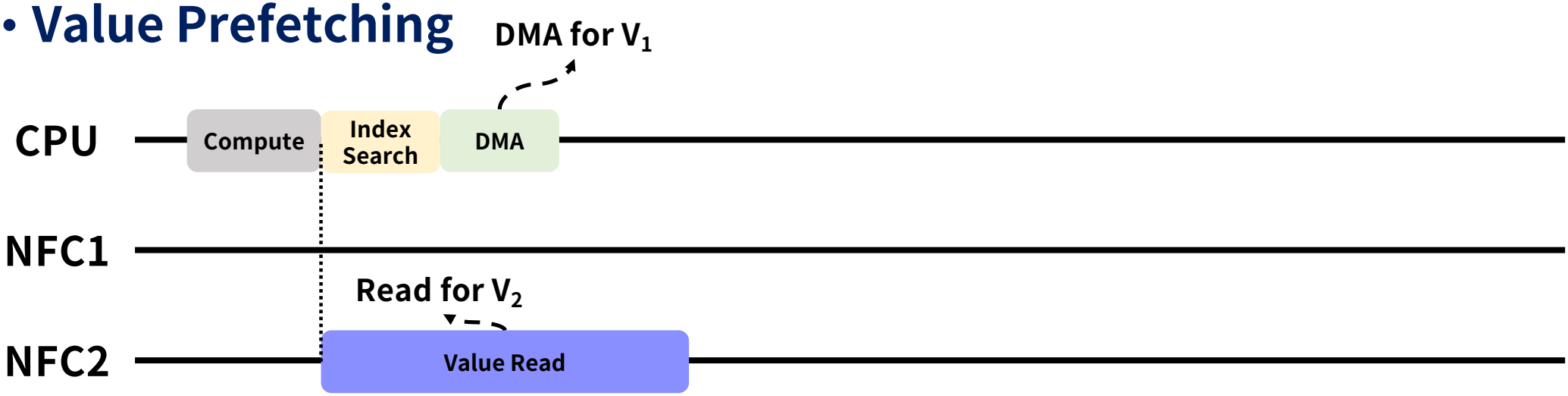
- **Value Prefetching**



- When processing **N^{th}** Next command, the device issues prefetch request for the **$(N+D)^{th}$** Next command in the future (**D**: Prefetching Degree).
- Therefore, it can overlap the reading time with other process in advance by exploiting hardware (NFCs).

Value Prefetching

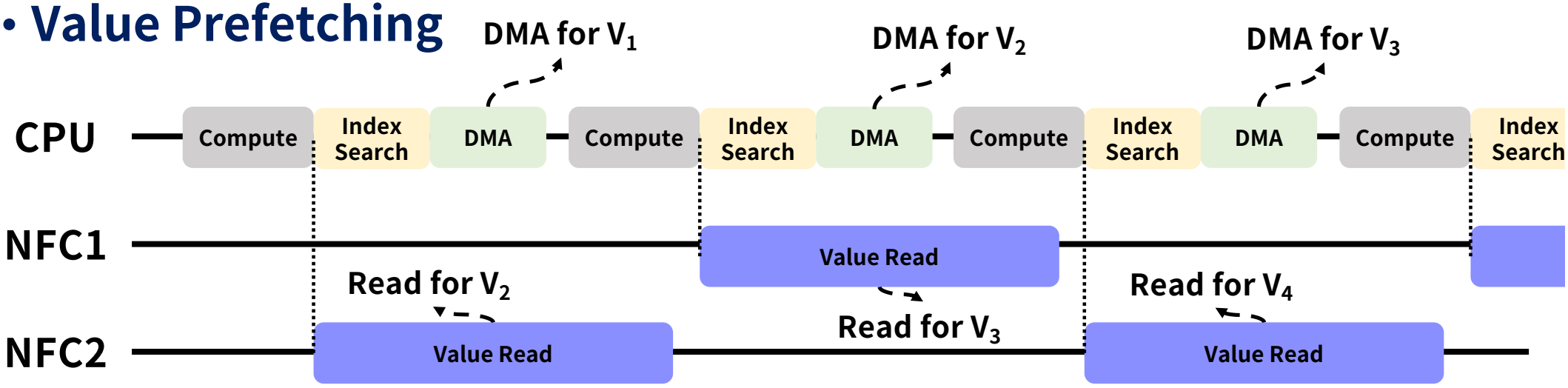
- **Value Prefetching**



- When processing **N^{th}** Next command, the device issues prefetch request for the **$(N+D)^{th}$** Next command in the future (**D**: Prefetching Degree).
- Therefore, it can overlap the reading time with other process in advance by exploiting hardware (NFCs).

Value Prefetching

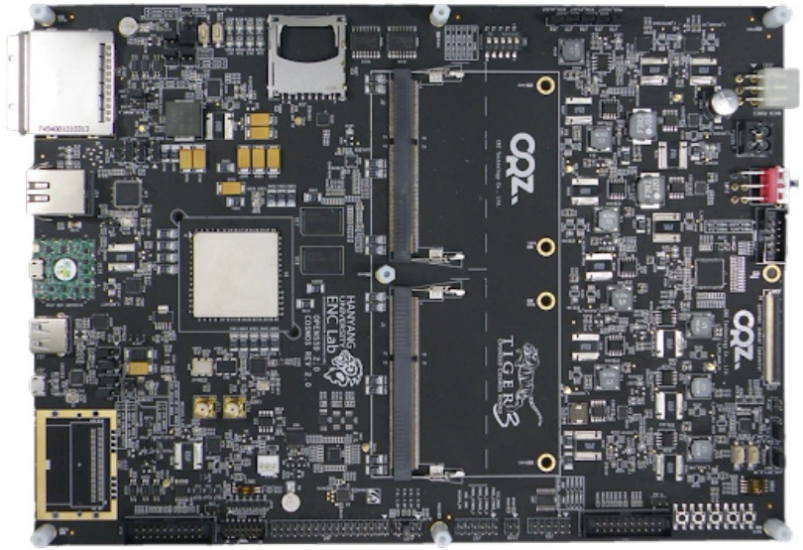
- **Value Prefetching**



- When processing **Nth** Next command, the device issues prefetch request for the **(N+D)th** Next command in the future (**D**: Prefetching Degree).
- Therefore, it can overlap the reading time with other process in advance by exploiting hardware (NFCs).

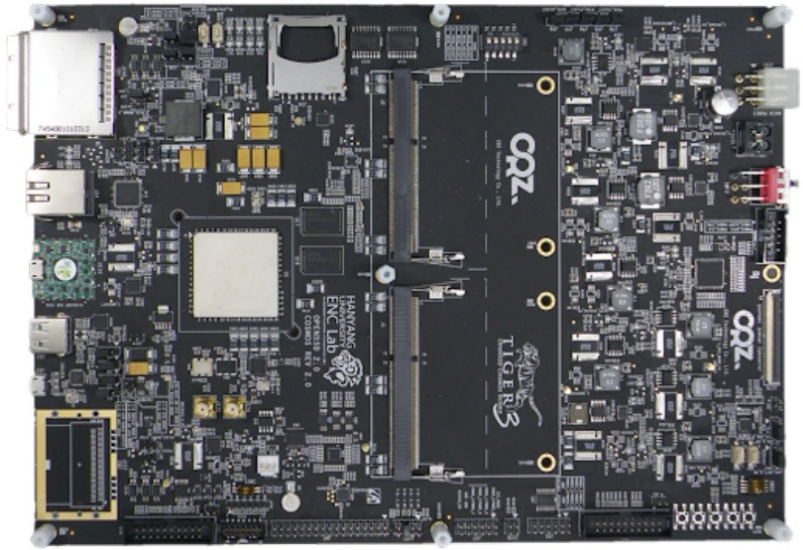
Experimental Setup

- **Implementation of IterKVSSD**
  - Prototyped on OpenSSD Cosmos+
  - Extended NVMe Protocol for Iterator Command



Experimental Setup

- **Implementation of IterKVSSD**
  - Prototyped on OpenSSD Cosmos+
  - Extended NVMe Protocol for Iterator Command

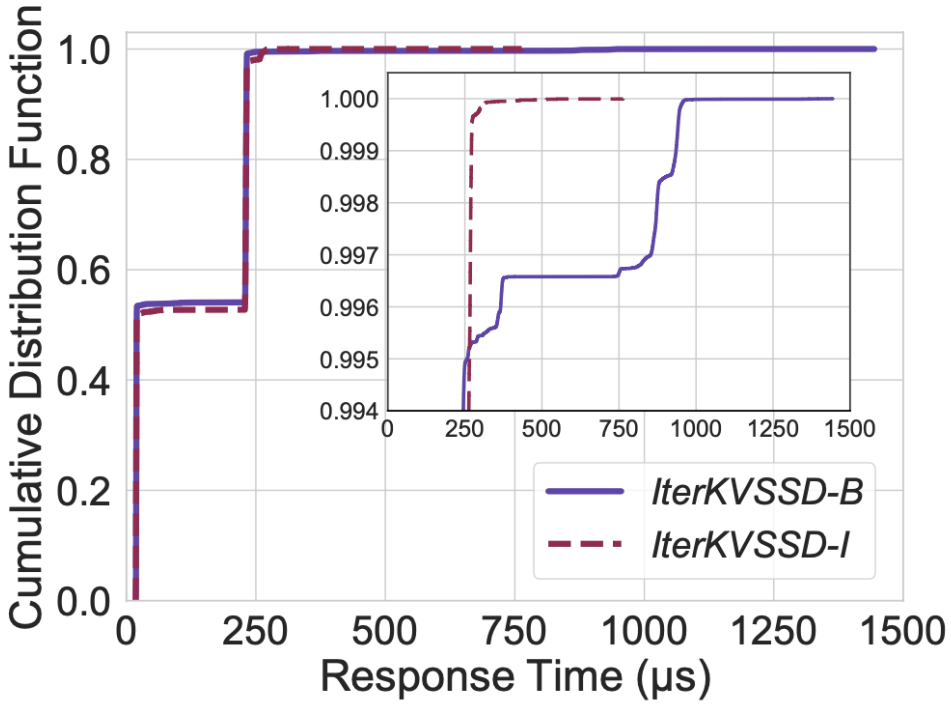


- **Experiment**
  - Evaluated with RocksDB ***db_bench*** benchmark
  - Populated with 3M Key-Value pairs with 4B Key
  - Evaluated with ***SeekRandom*** workload
  - Evaluated by varying scan length*, value size, and prefetch degree for value prefetch

scan length = the number of KV pairs retrieved during a range query

Evaluation – Index Prefetch

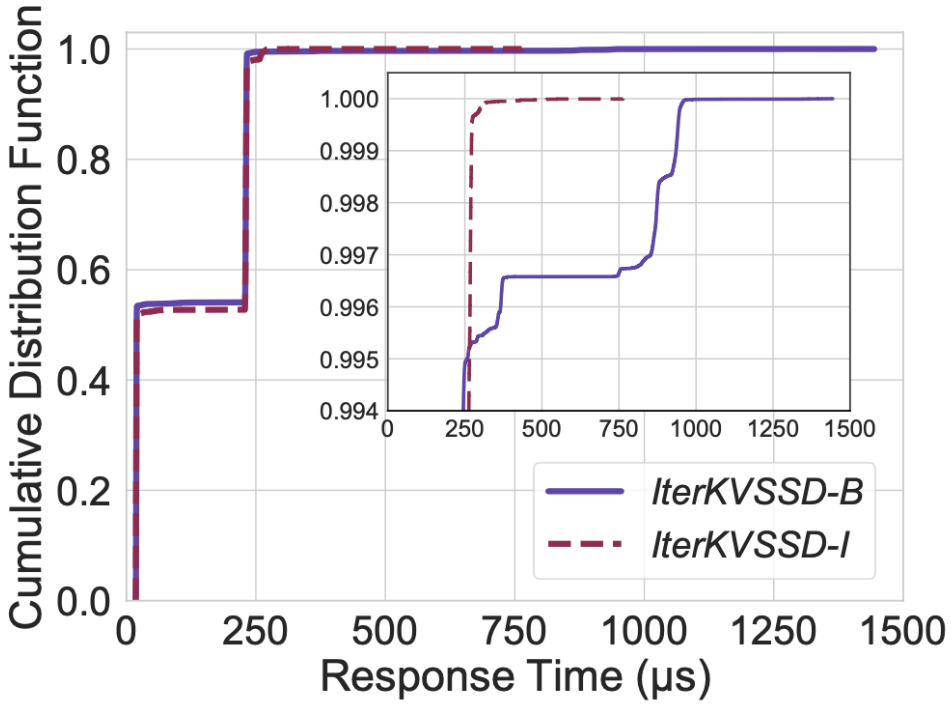
- **Effect of Index Prefetching**



- **IterKVSSD-B**: Baseline w/o prefetch
- **IterKVSSD-I**: w/ Index Prefetch + w/o Value Prefetch

Evaluation – Index Prefetch

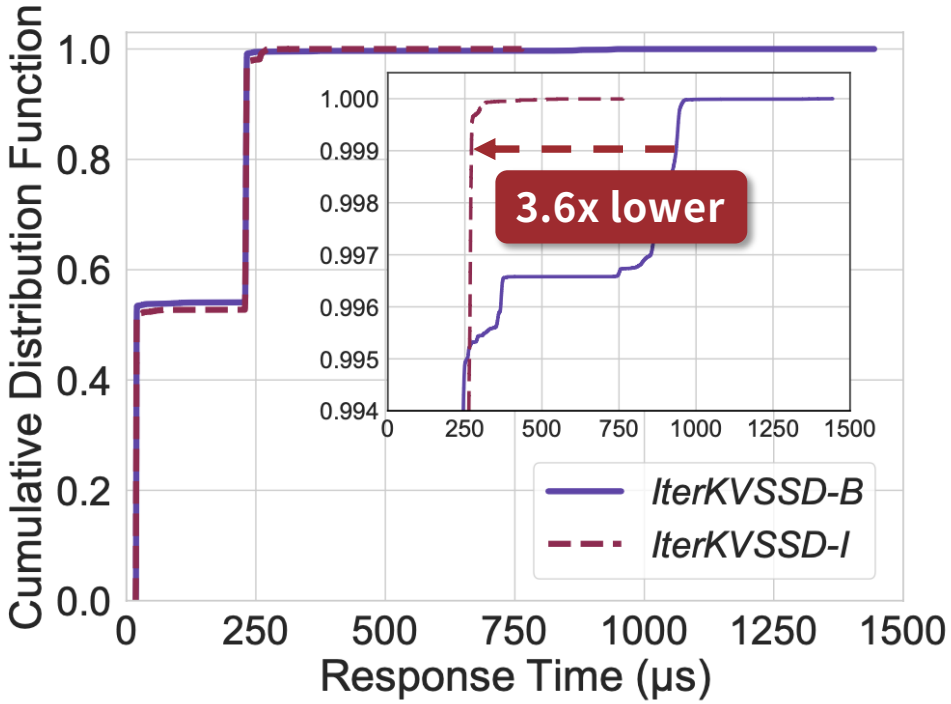
- **Effect of Index Prefetching**



- **IterKVSSD-B**: Baseline w/o prefetch
- **IterKVSSD-I**: w/ Index Prefetch + w/o Value Prefetch
- Scan Length = 200,000 which is enough to trigger Synchronous Index Read

Evaluation – Index Prefetch

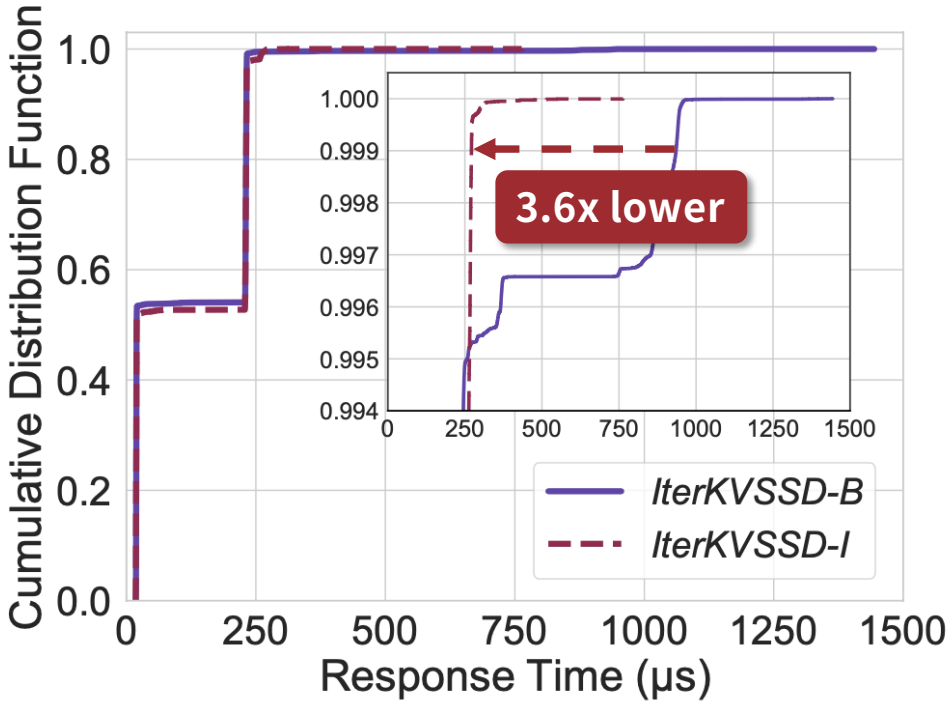
- **Effect of Index Prefetching**



- **IterKVSSD-B**: Baseline w/o prefetch
- **IterKVSSD-I**: w/ Index Prefetch + w/o Value Prefetch
- Scan Length = 200,000 which is enough to trigger Synchronous Index Read
- Show about 3.6x better P99.9 tail latency

Evaluation – Index Prefetch

- **Effect of Index Prefetching**



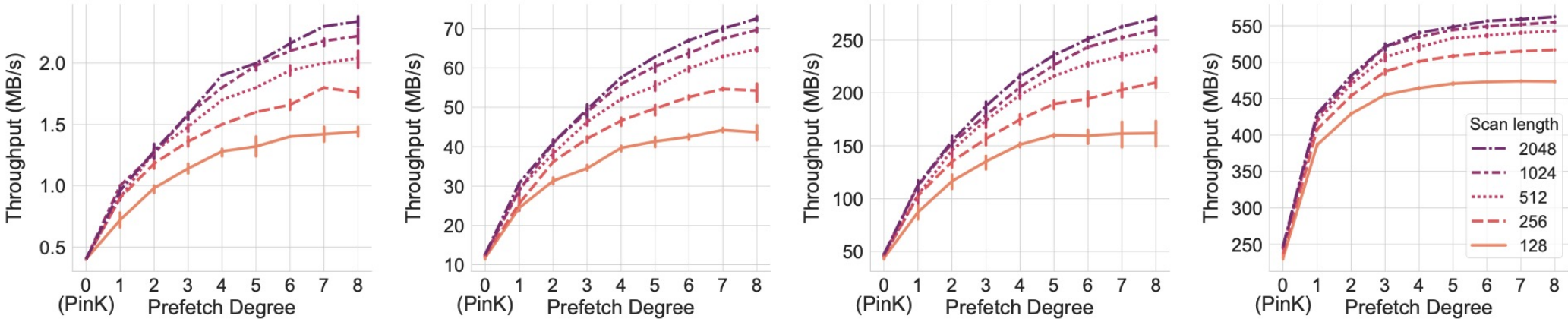
- **IterKVSSD-B**: Baseline w/o prefetch
- **IterKVSSD-I**: w/ Index Prefetch + w/o Value Prefetch
- Scan Length = 200,000 which is enough to trigger Synchronous Index Read
- Show about 3.6x better P99.9 tail latency
- Channel conflict prevents it from being removed completely

Evaluation – Value Prefetch

- **Effect of Value Prefetching**
  - Evaluated with ***SeekRandom*** workload
  - Prefetch Degree: 0 – 8
  - Value Size : 128B, 4KB, 16KB, 128KB
  - Scan Length: 128, 256, 512, 1024, 2048

Evaluation – Value Prefetch

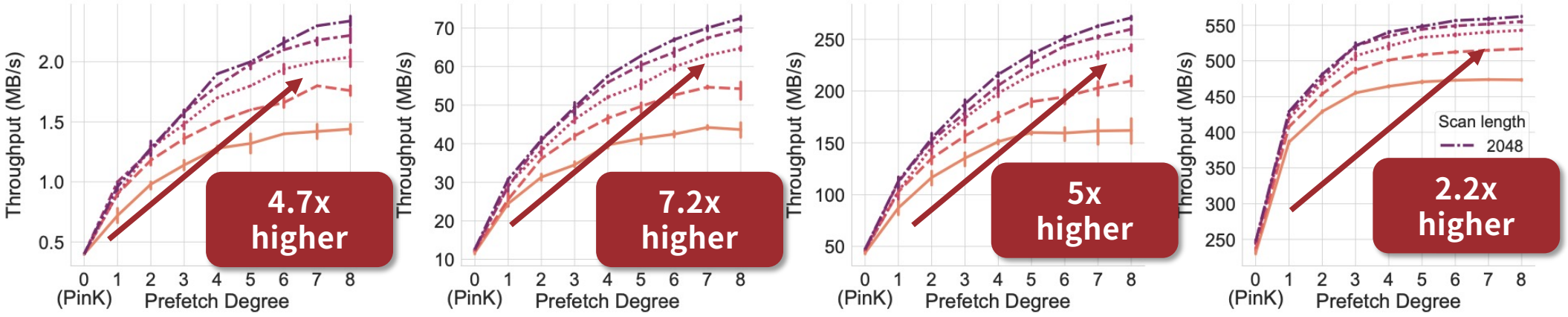
- **Effect of Prefetching Degree**



(a) 128 B (b) 4 KB (c) 16 KB (d) 128 KB

Evaluation – Value Prefetch

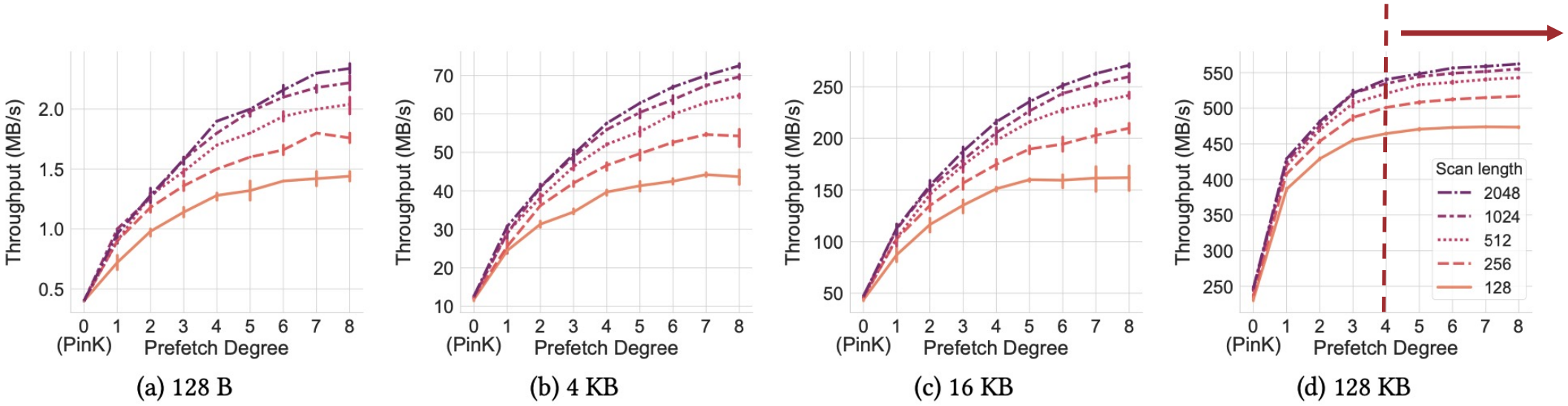
- **Effect of Prefetching Degree**
  - With higher prefetch degree, better I/O performance because small prefetch degree is not enough to hide NAND Flash access latency completely



(a) 128 B (b) 4 KB (c) 16 KB (d) 128 KB

Evaluation – Value Prefetch

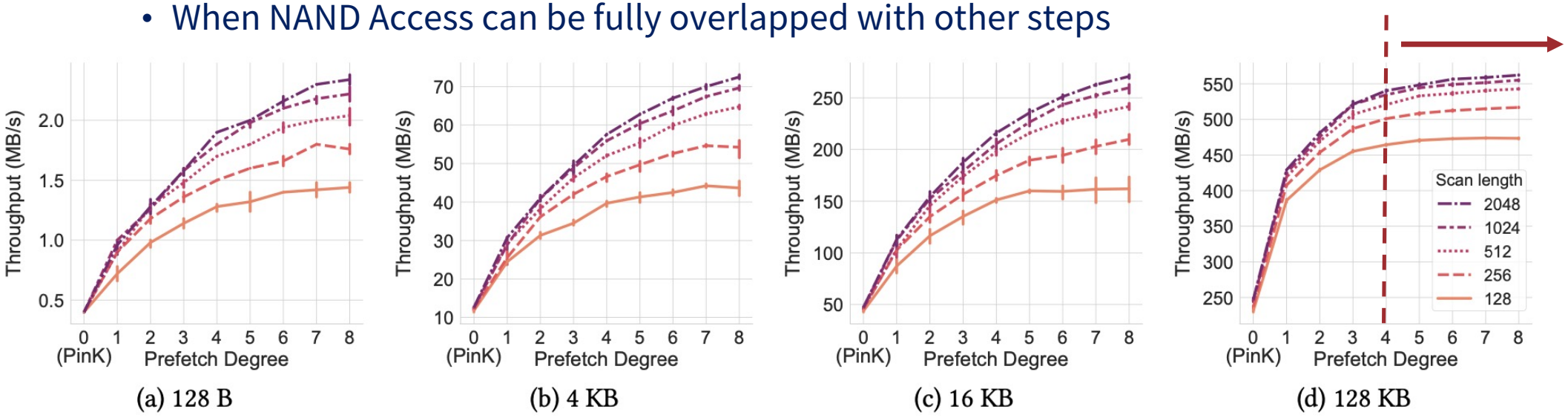
- **Effect of Prefetching Degree**
  - With higher prefetch degree, better I/O performance because small prefetch degree is not enough to hide NAND Flash access latency completely
  - However, prefetch degree over some degree will not improve performance



(a) 128 B (b) 4 KB (c) 16 KB (d) 128 KB

Evaluation – Value Prefetch

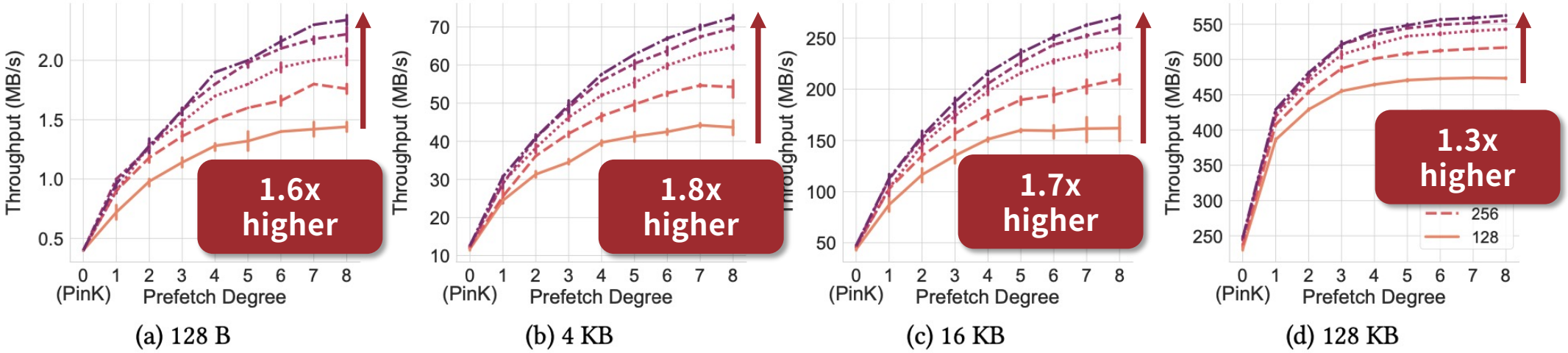
- **Effect of Prefetching Degree**
  - With higher prefetch degree, better I/O performance because small prefetch degree is not enough to hide NAND Flash access latency completely
  - However, prefetch degree over some degree will not improve performance
    - When internal bandwidth is saturated
    - When NAND Access can be fully overlapped with other steps



(a) 128 B (b) 4 KB (c) 16 KB (d) 128 KB

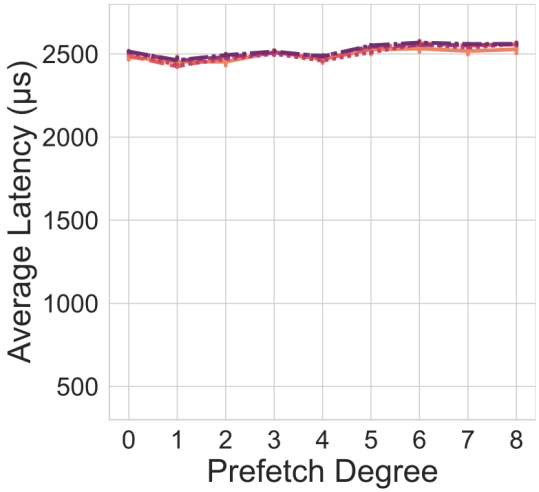
Evaluation – Value Prefetch

- **Effect of Scan Length in Range Query**
  - With higher scan length, better I/O throughput

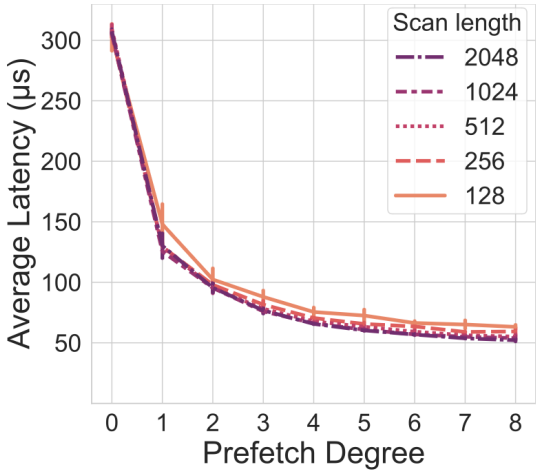


Evaluation – Value Prefetch

- **Effect of Value Prefetching**
  - 1 **Seek** + *N* **Next** (*N*: Scan Length)
  - **Seek** shows much higher latency than **Next**
  - Therefore, the higher scan length shows the better overall I/O performance



(a) Seek()



(b) Next()

Conclusion

- **Iterator Interface Extended LSM-tree-based KVSSD**
  - Explore three problems of current iterator interface
    1. Versioning Problem
    2. Synchronous Index Read Problem
    3. Synchronous Value Read Problem
- **IterKVSSD**
  - Propose solutions for the above problem by exploit the characteristics of KVSSD
  - Memory-efficient Versioning through decoupling and pooling metadata
  - Index/Value Prefetch to mitigate NAND Flash Access Penalty
  - Shows **2x** lower P99.9 tail latency, up to **7.2x** better I/O throughput

Thank you for listening! Q&A

Iterator Interface Extended LSM-tree-based KVSSD for Range Queries

Seungjin Lee[1], Chang-Gyu Lee[1], Donghyun Min[1], Inhyuk Park[2], Woosuk Chung[2], Anand Sivasubramaniam[3], Youngjae Kim[1]

[1] Sogang University, [2] SK hynix, [3] The Pennsylvania State University