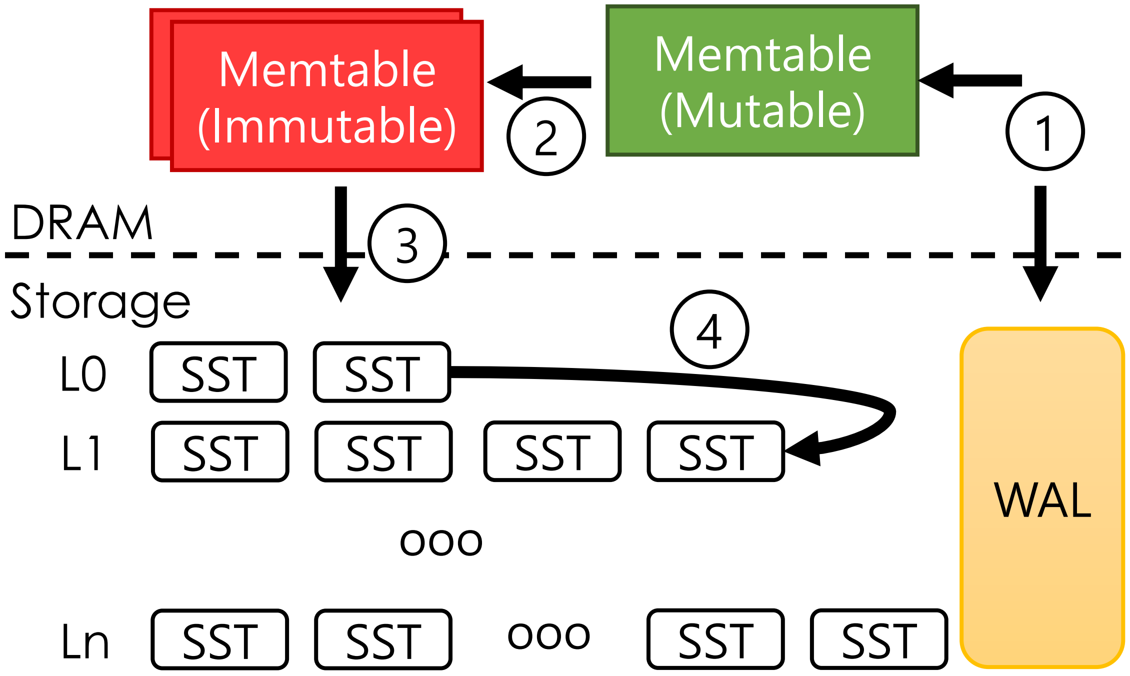
Exploring Data Deduplication in LSM-Tree Based Key-Value Stores

Safdar Jamil[1], Awais Khan[2], Youngjae Kim[1]

[1]Sogang University, Seoul, South Korea

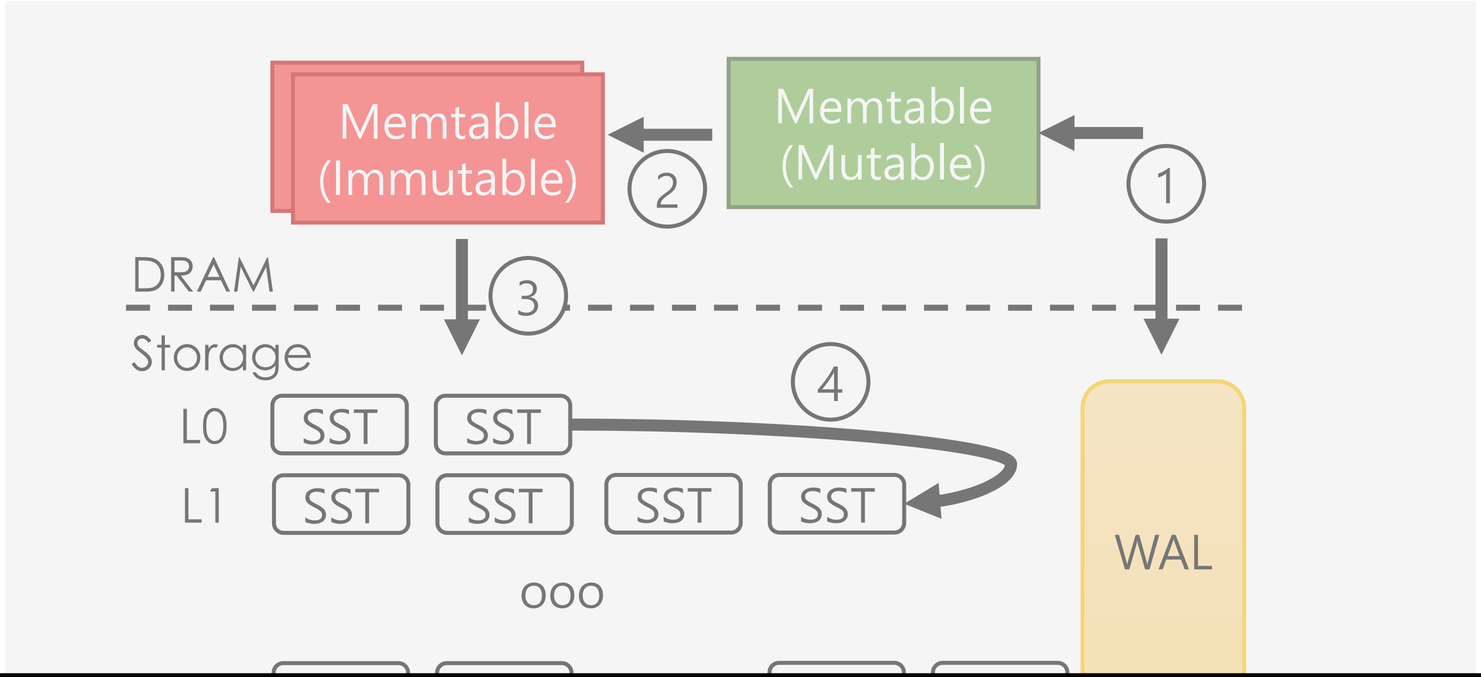
[2]Oak Ridge National Lab, TN, USA

LSM Tree-based KV-stores Architecture



LSM-based KV-stores architecture

LSM Tree-based KV-stores Architecture



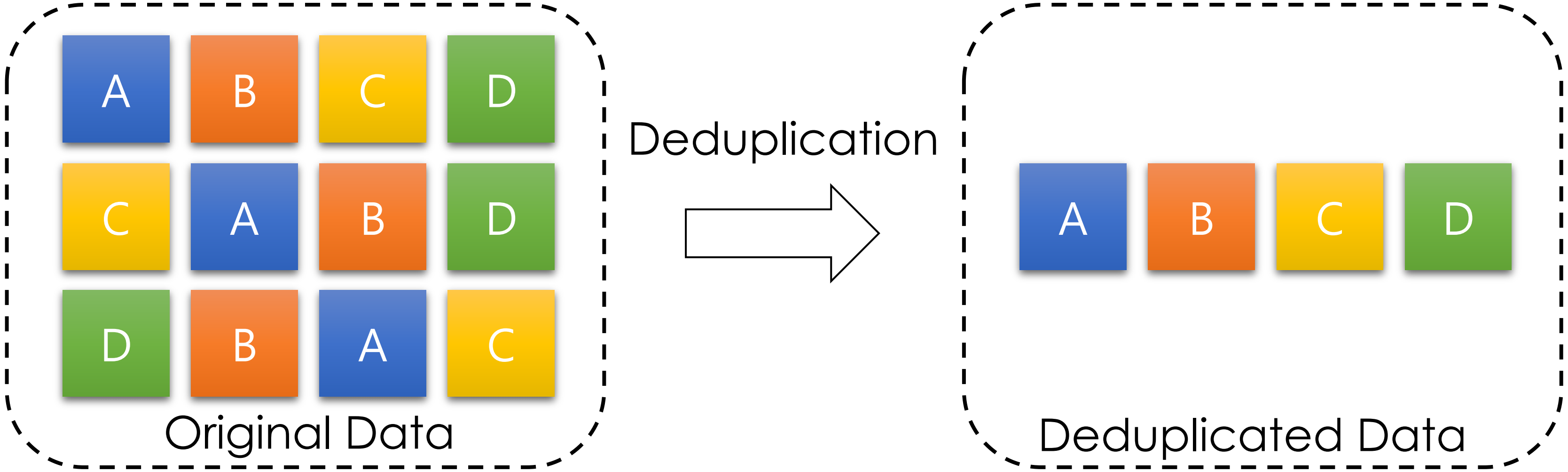
Limitations of LSM tree

- ❑ High write amplification (WA)
- ❑ High space amplification (SA)

Deduplication can be adopted to minimize the WA and SA

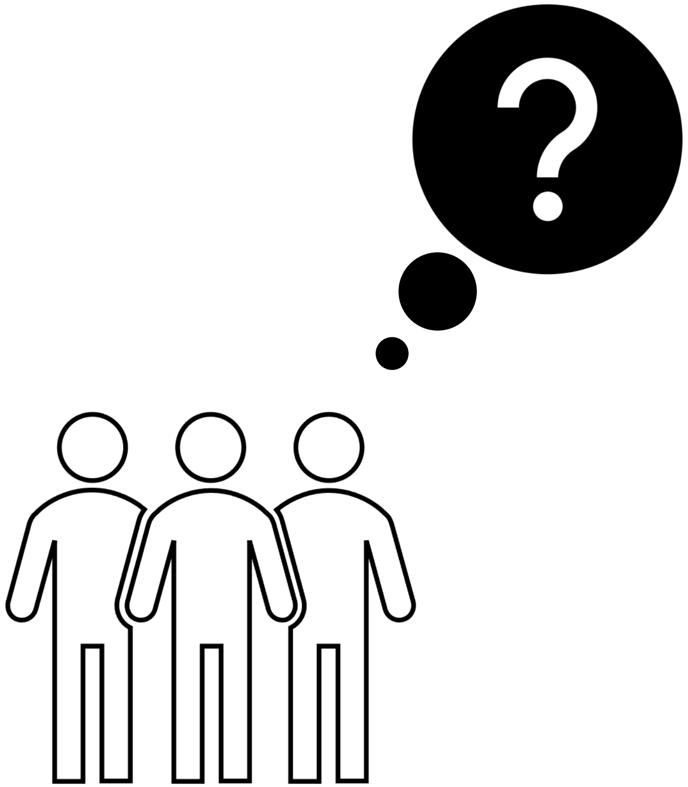
Deduplication in LSM

- Novel way to minimize WA and SA
  - SSTable can be a complete duplicate
  - A partition of SSTable can be duplicate
- Incorporating value-based deduplication
  - Can help reduce the actual size of KV-store



Basic Questions for Deduplication in LSM

- How to manage deduplication overhead?
  - Performance and storage overhead
- What deduplication technique?
  - Inline deduplication or offline deduplication?
- Where to apply deduplication?
  - Memtable or Immutable Memtables or SSTables?



Inline Deduplication at MemTable (MemDedup)

Deduplication at Memtable-level (MemDedup)

- Inline deduplication
  - Intercepts PUT request and performs deduplication before storing to Memtable
  - With inline dedup, less data is written to the Memtable
    - Leads to less data at all levels of LSM
  - Maintains deduplication metadata in Chunk Information Table

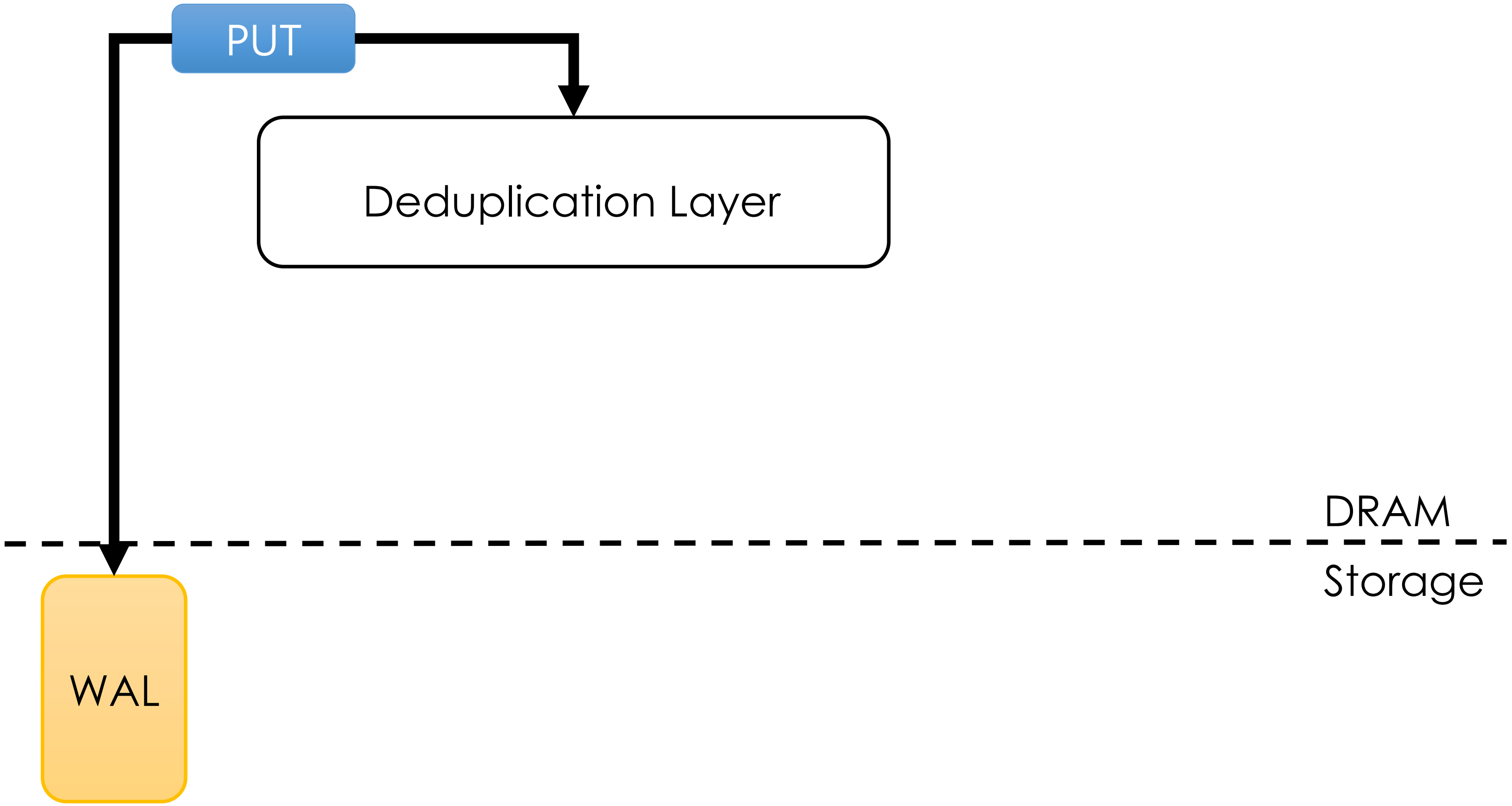
Deduplication at Memtable-level (MemDedup)

- Inline deduplication
  - Intercepts PUT request and performs deduplication before storing to Memtable
  - With inline dedup, less data is written to the Memtable
    - Leads to less data at all levels of LSM
  - Maintains deduplication metadata in Chunk Information Table

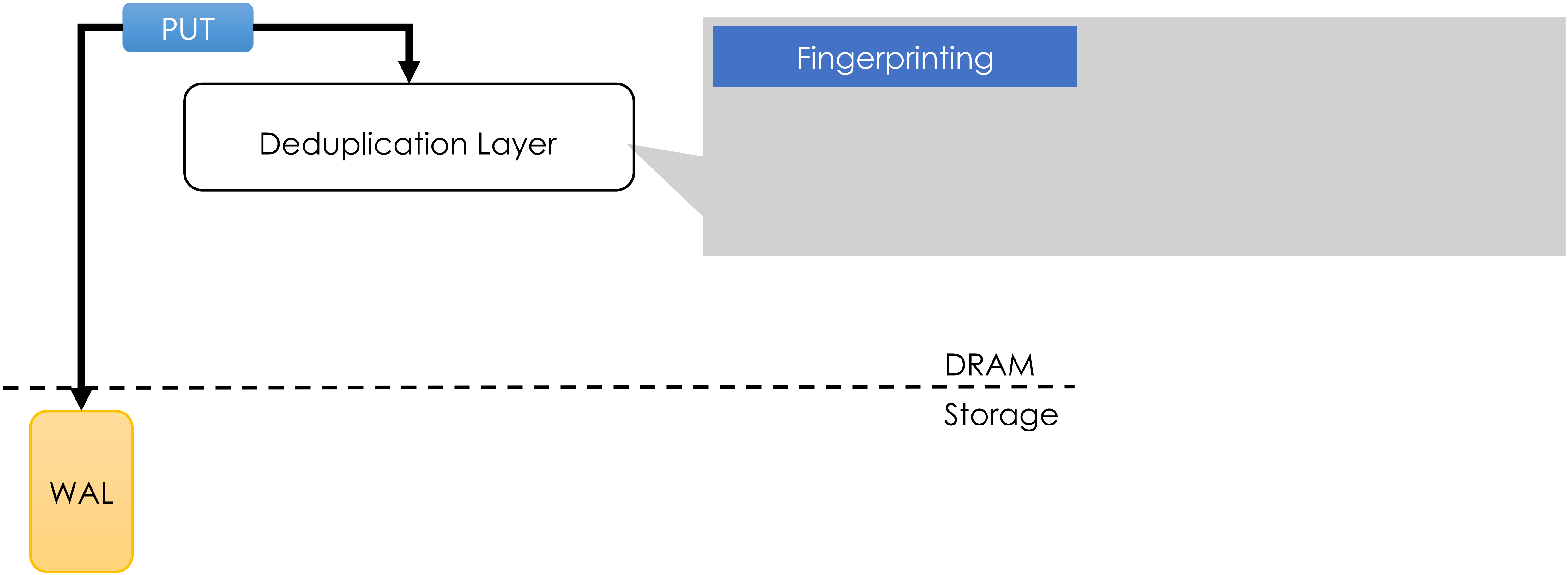
Fingerprint of value FP(V)	List of parent keys (PK[])	Reference count (RC)	Value location (Offset)
FP(12)	K1	1	0xf8
FP(34)	K2, K3	2	0xff

Chunk Information Table

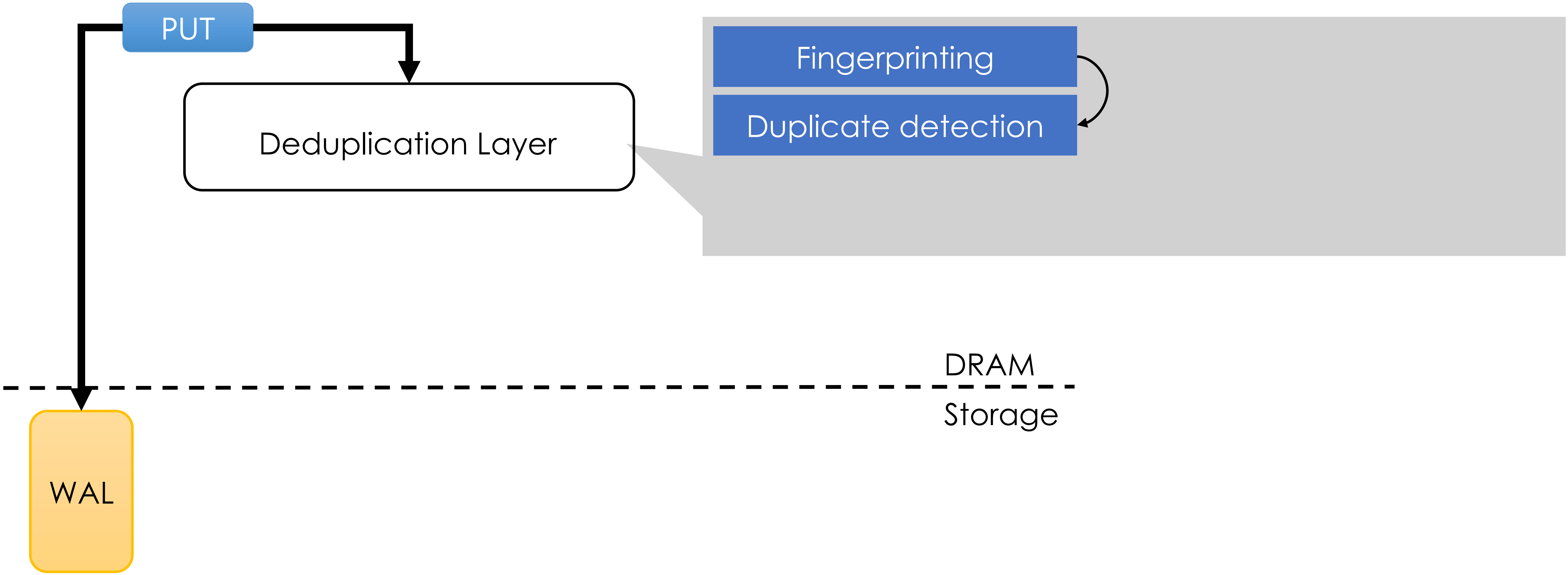
Deduplication at Memtable-level (MemDedup)



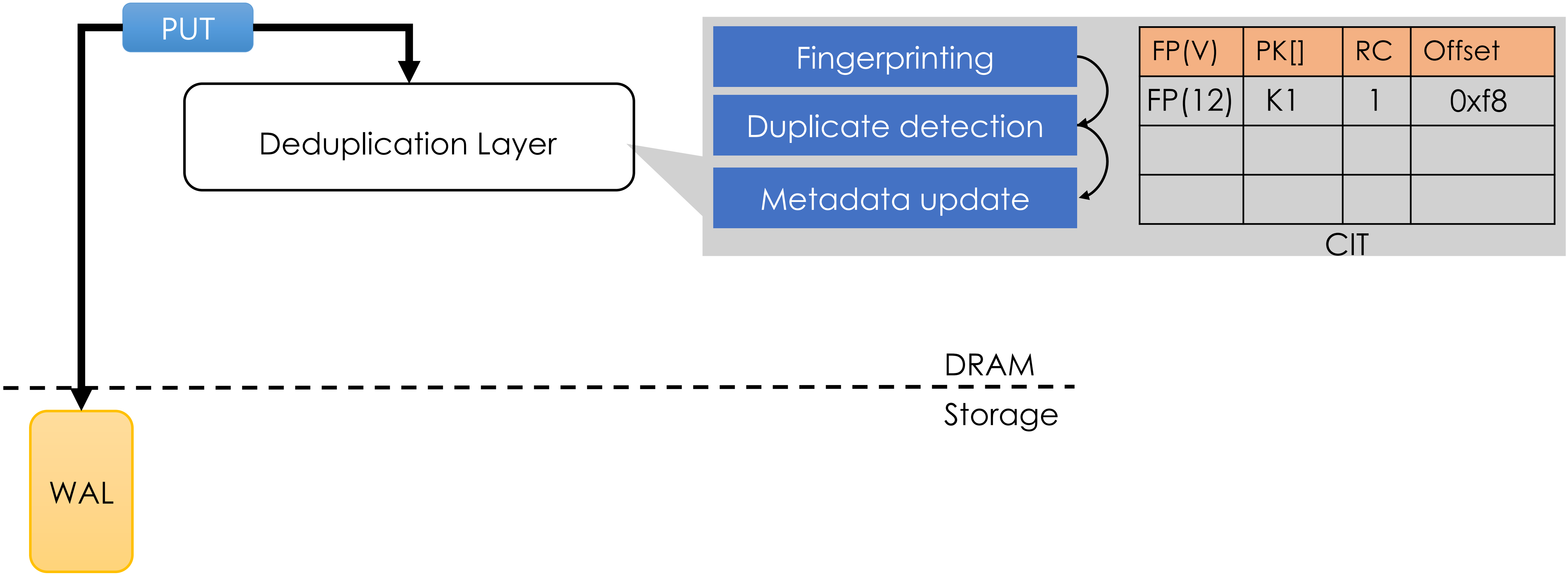
Deduplication at Memtable-level (MemDedup)



Deduplication at Memtable-level (MemDedup)

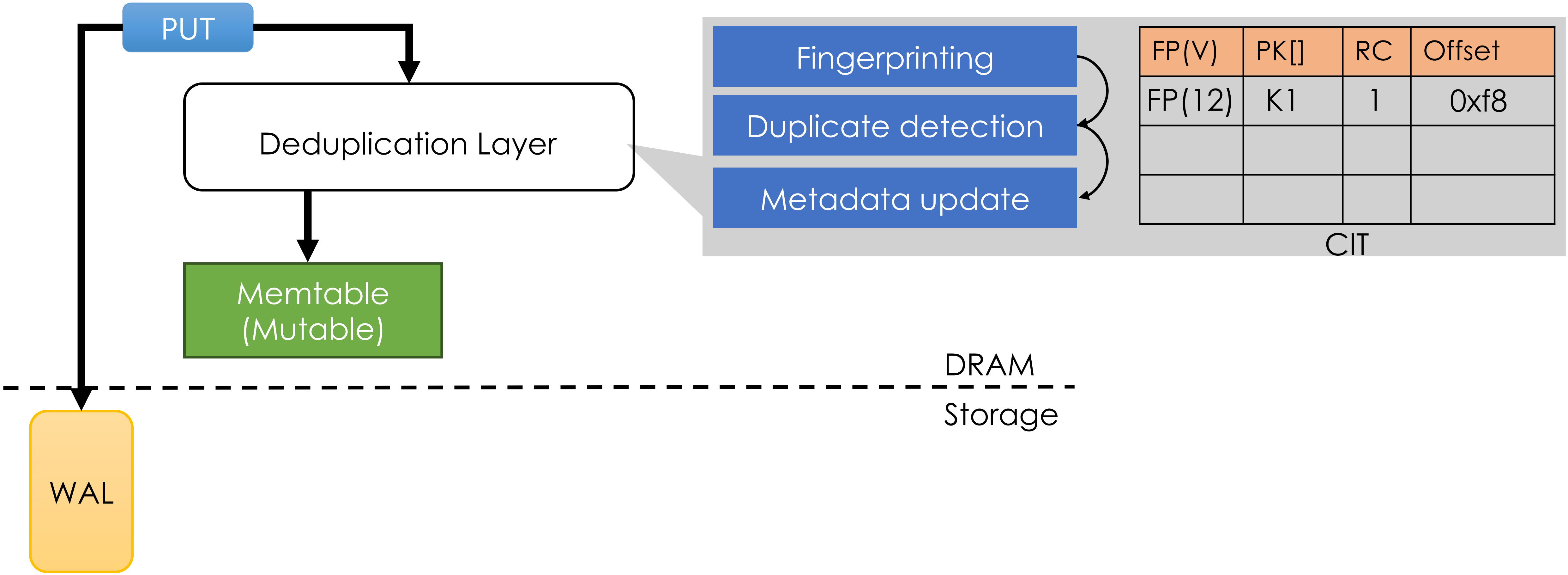


Deduplication at Memtable-level (MemDedup)

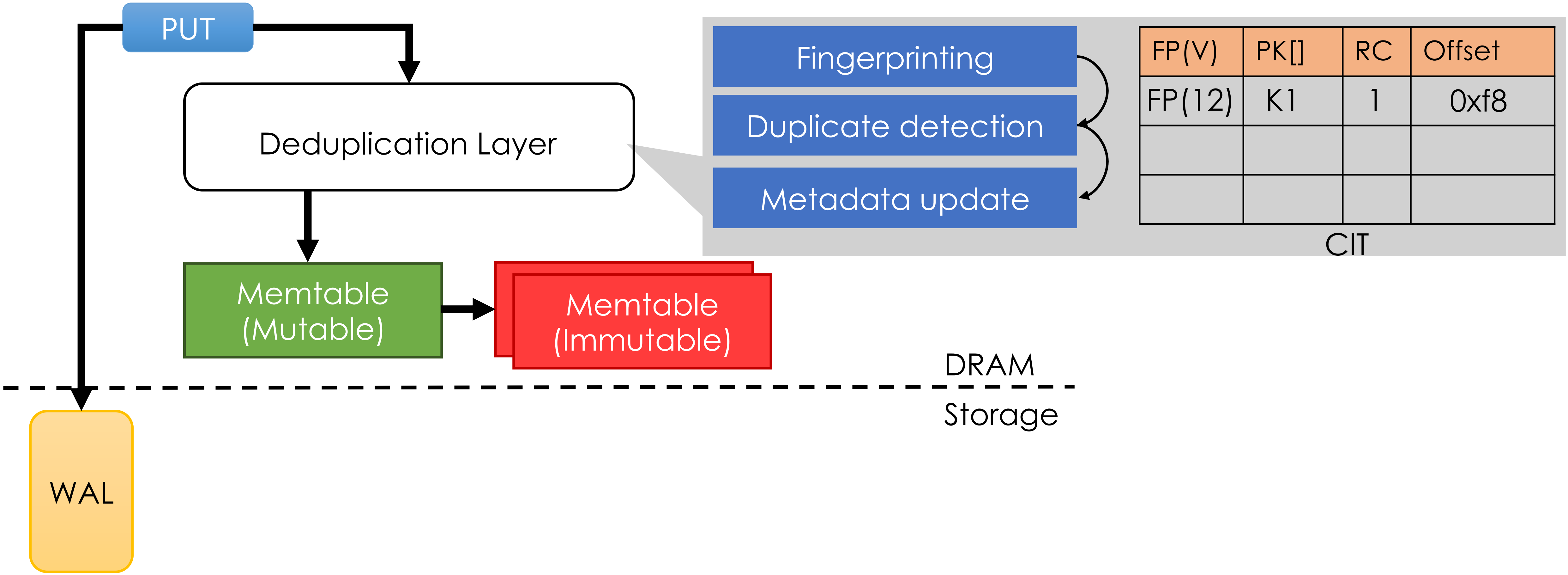


FP(V)	PK[]	RC	Offset
FP(12)	K1	1	0xf8

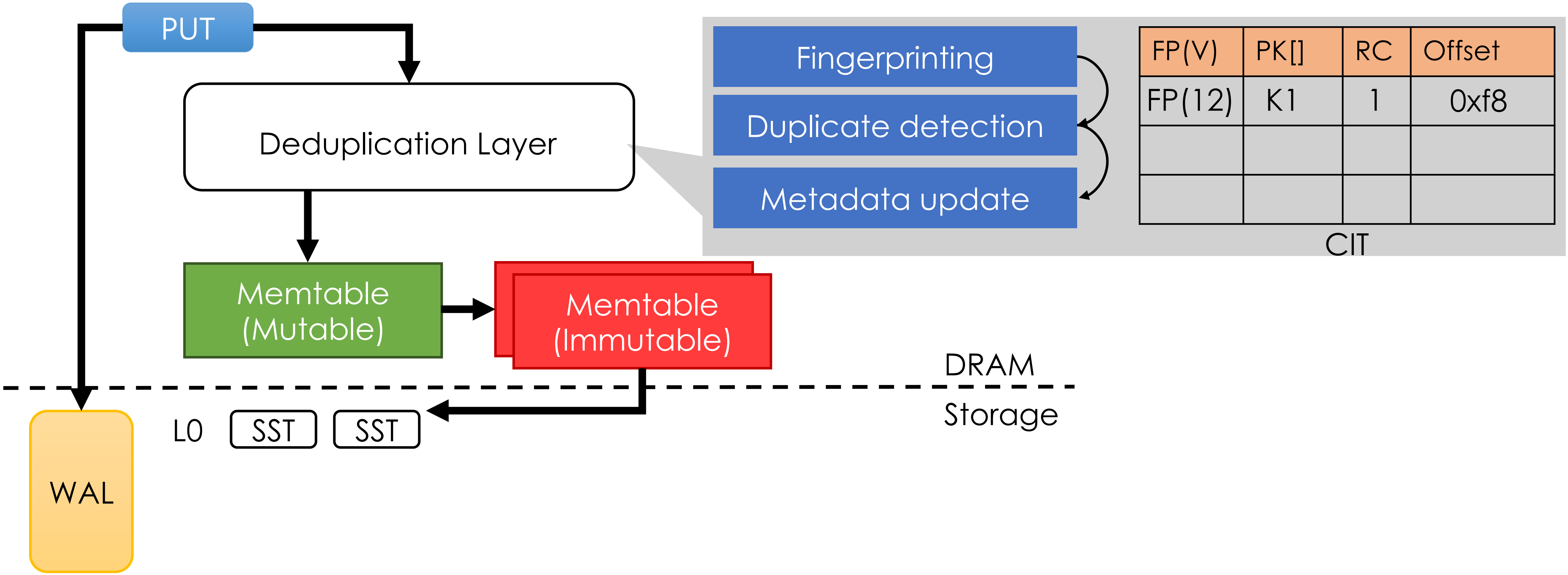
Deduplication at Memtable-level (MemDedup)



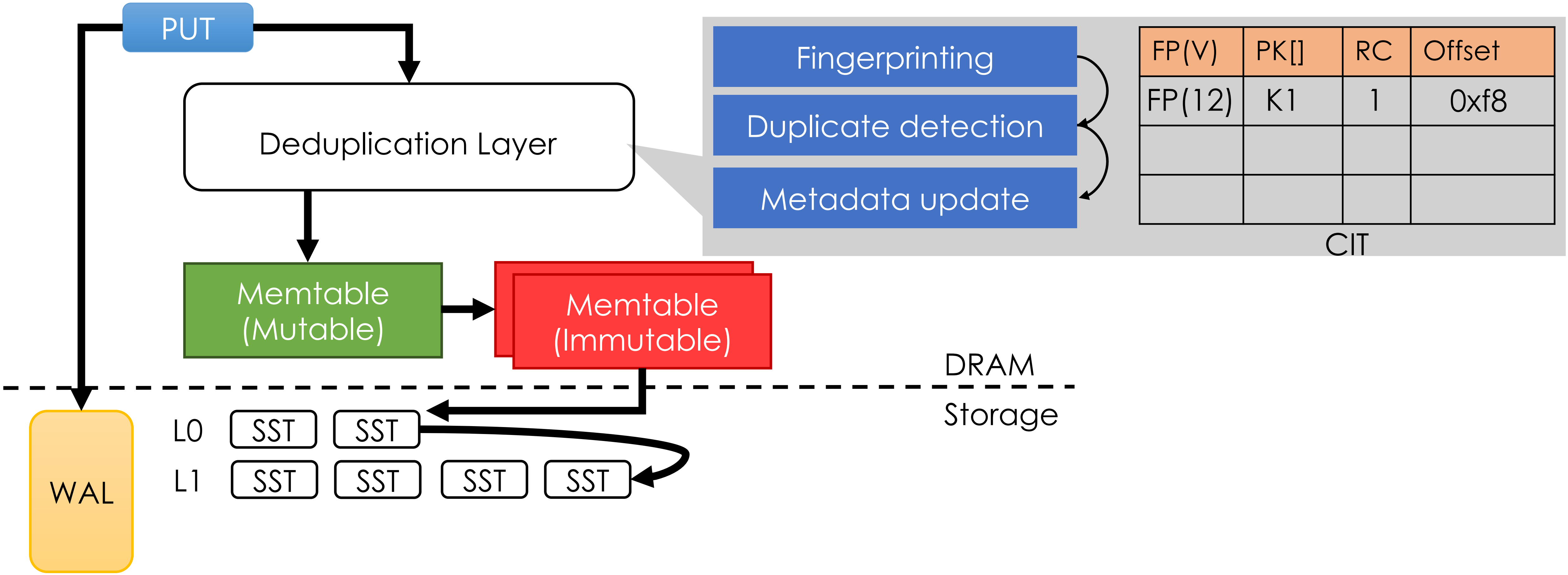
Deduplication at Memtable-level (MemDedup)



Deduplication at Memtable-level (MemDedup)

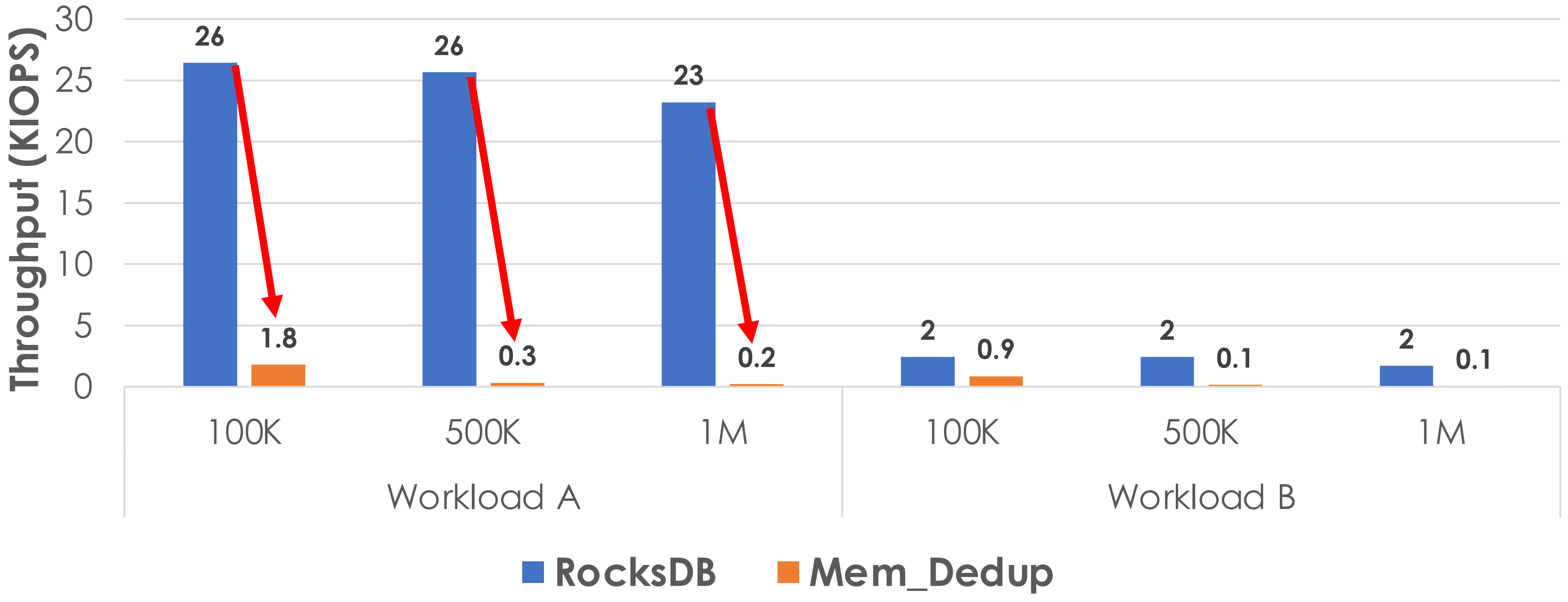


Deduplication at Memtable-level (MemDedup)



Deduplication at Memtable-level (MemDedup)

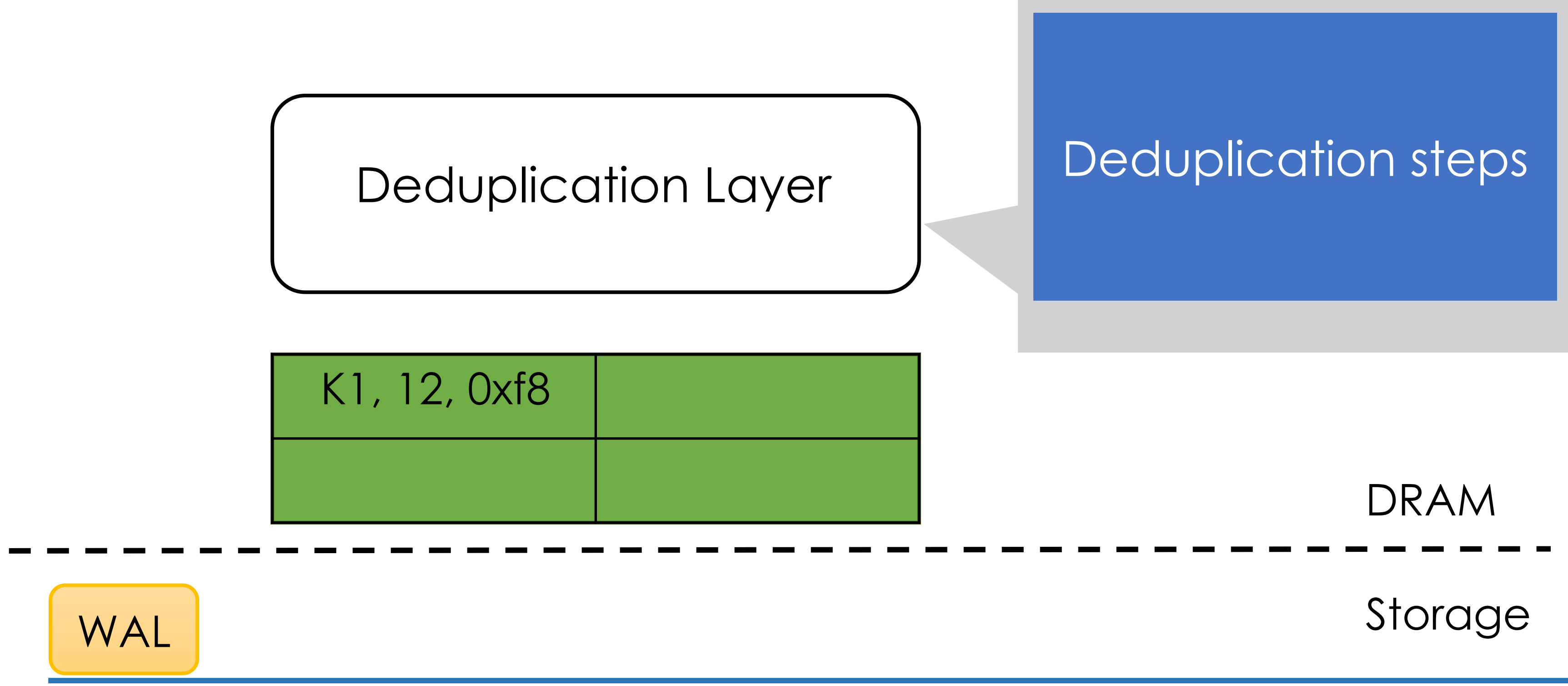
- Limitations
  1. High performance overhead
     - Dedup steps – fingerprinting
     - Frequent dedup metadata update/traversal



YCSB Benchmark | Workload A: 100% Write | Workload B: 50% write & 50% read

Deduplication at Memtable-level (MemDedup)

- Limitations
  1. High performance overhead
  2. Inconsistency issue
     - In-place update operations at Memtable

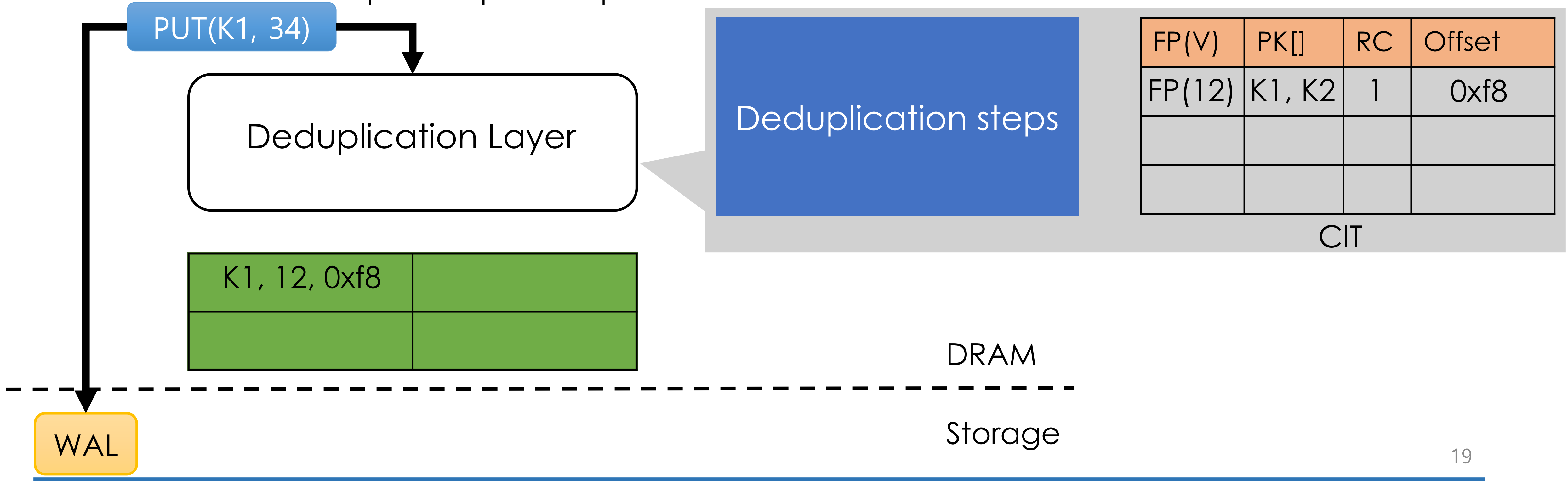


FP(V)	PK[]	RC	Offset
FP(12)	K1, K2	1	0xf8

CIT

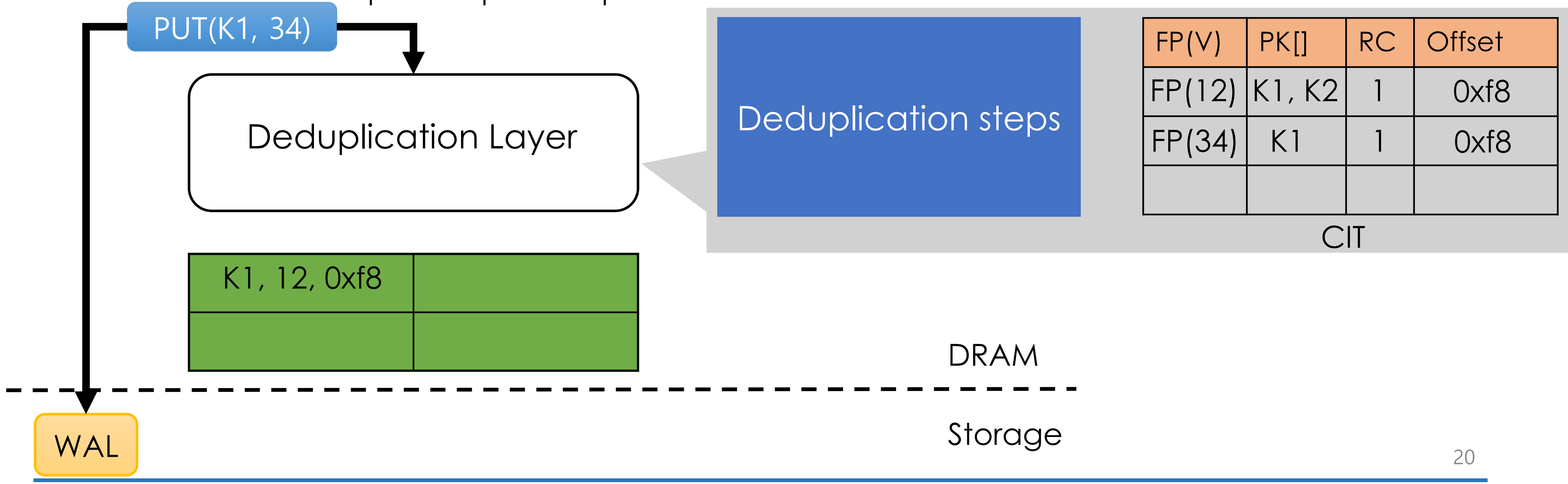
Deduplication at Memtable-level (MemDedup)

- Limitations
  1. High performance overhead
  2. Inconsistency issue
     - In-place update operations at Memtable



Deduplication at Memtable-level (MemDedup)

- Limitations
  1. High performance overhead
  2. Inconsistency issue
     - In-place update operations at Memtable

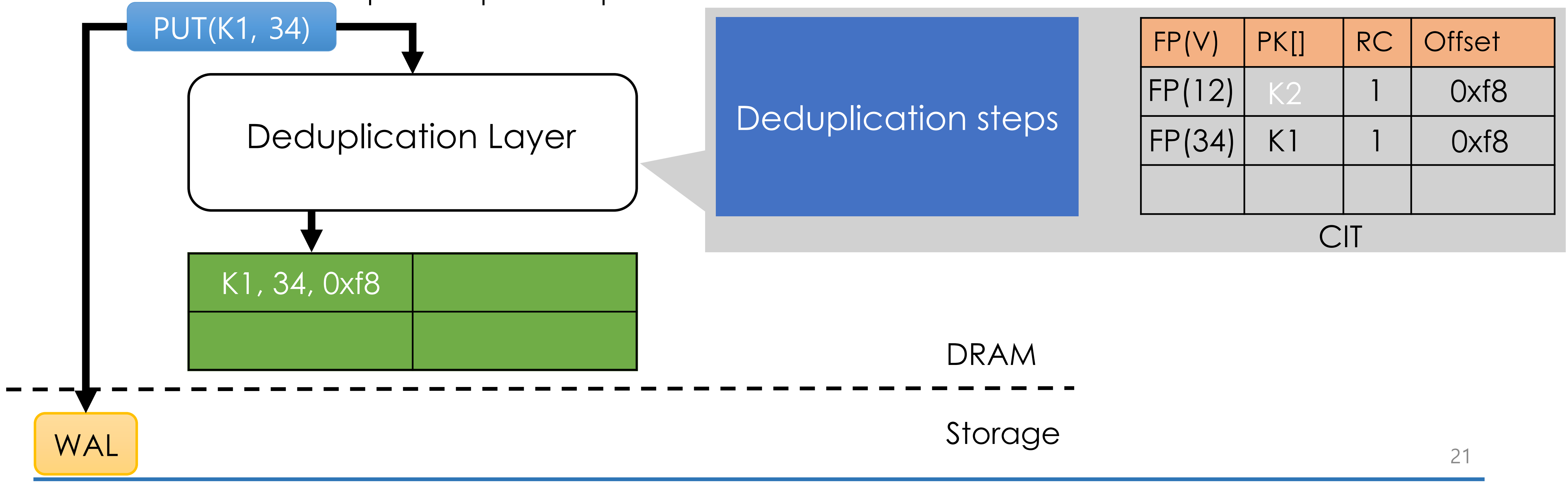


FP(V)	PK[]	RC	Offset
FP(12)	K1, K2	1	0xf8
FP(34)	K1	1	0xf8

CIT

Deduplication at Memtable-level (MemDedup)

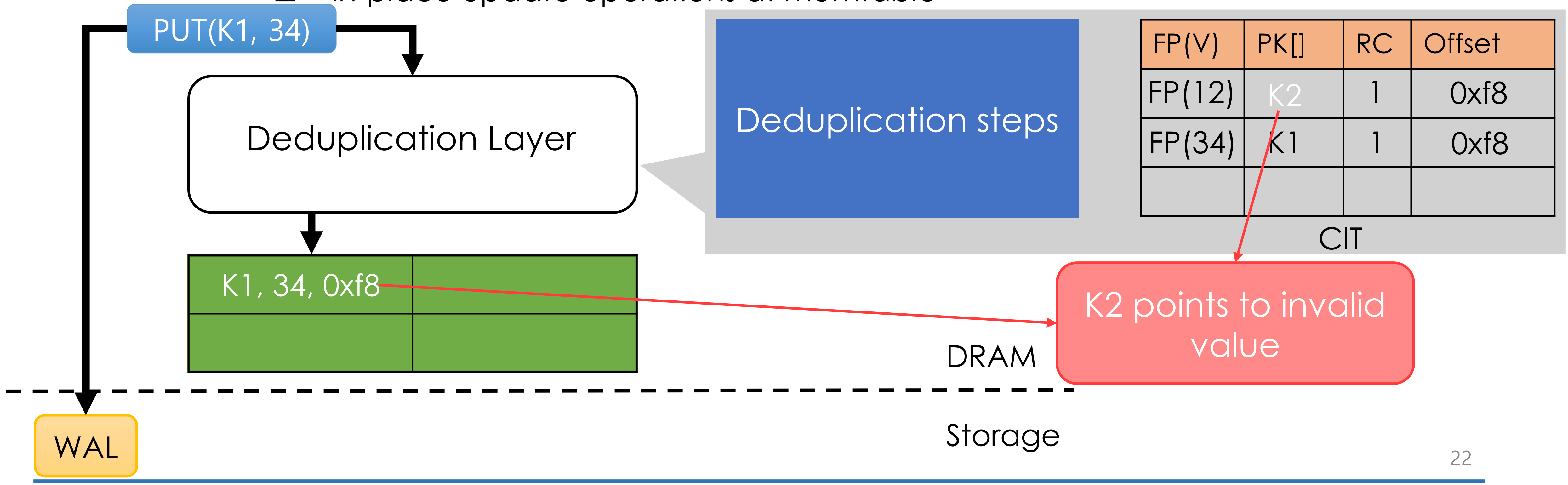
- ❑ Limitations
  1. High performance overhead
  2. Inconsistency issue
     - ❑ In-place update operations at Memtable



FP(V)	PK[]	RC	Offset
FP(12)	K2	1	0xf8
FP(34)	K1	1	0xf8

Deduplication at Memtable-level (MemDedup)

- Limitations
  1. High performance overhead
  2. Inconsistency issue
     - In-place update operations at Memtable

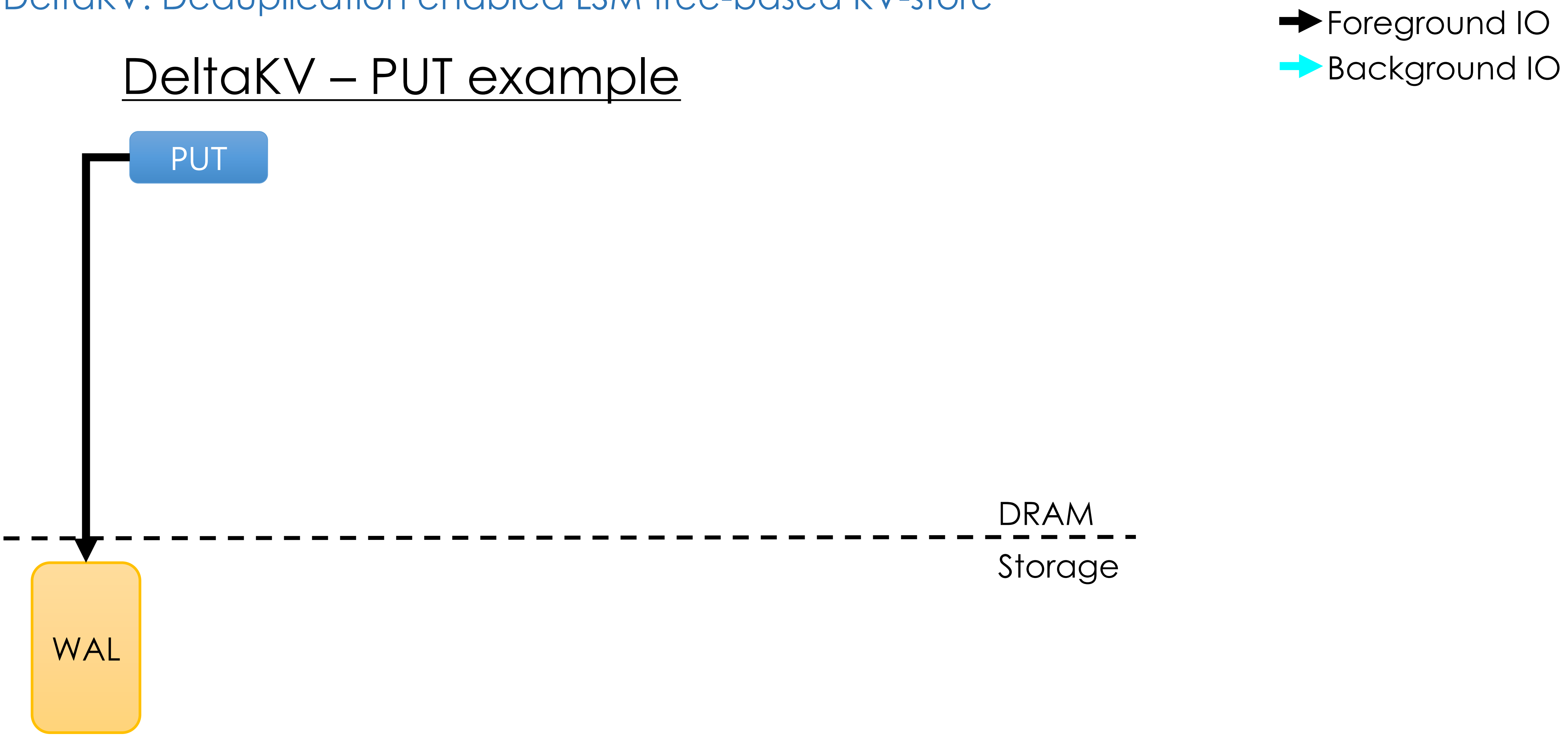


Deduplication enabled LSM Tree-based Key-Value Store (DeltaKV)

DeltaKV

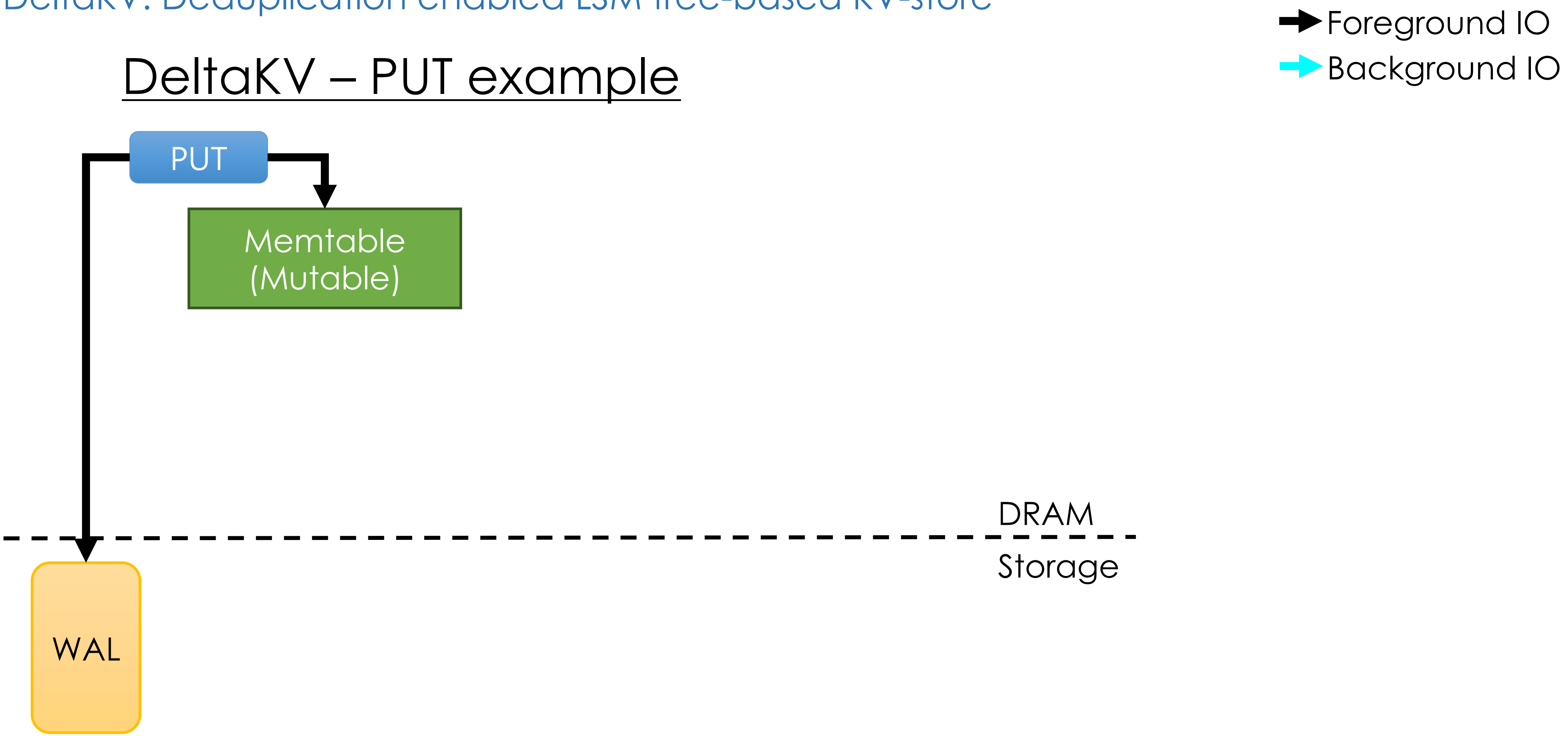
- Offline deduplication (dedup)
  - Delays dedup to FLUSH operation
    - When Immutable Memtables are written to SSTables
- Background threads perform dedup
  - No interference with foreground IOs → less performance effect
- Mitigate inconsistency issue
  - Dedup performed on immutable data
  - No direct manipulation by foreground IOs
- Further Optimization
  - Divides dedup metadata in two data structures
    - Value Information Table (VIT) – for write/update operations
    - B+-Tree – for read operations

DeltaKV – PUT example



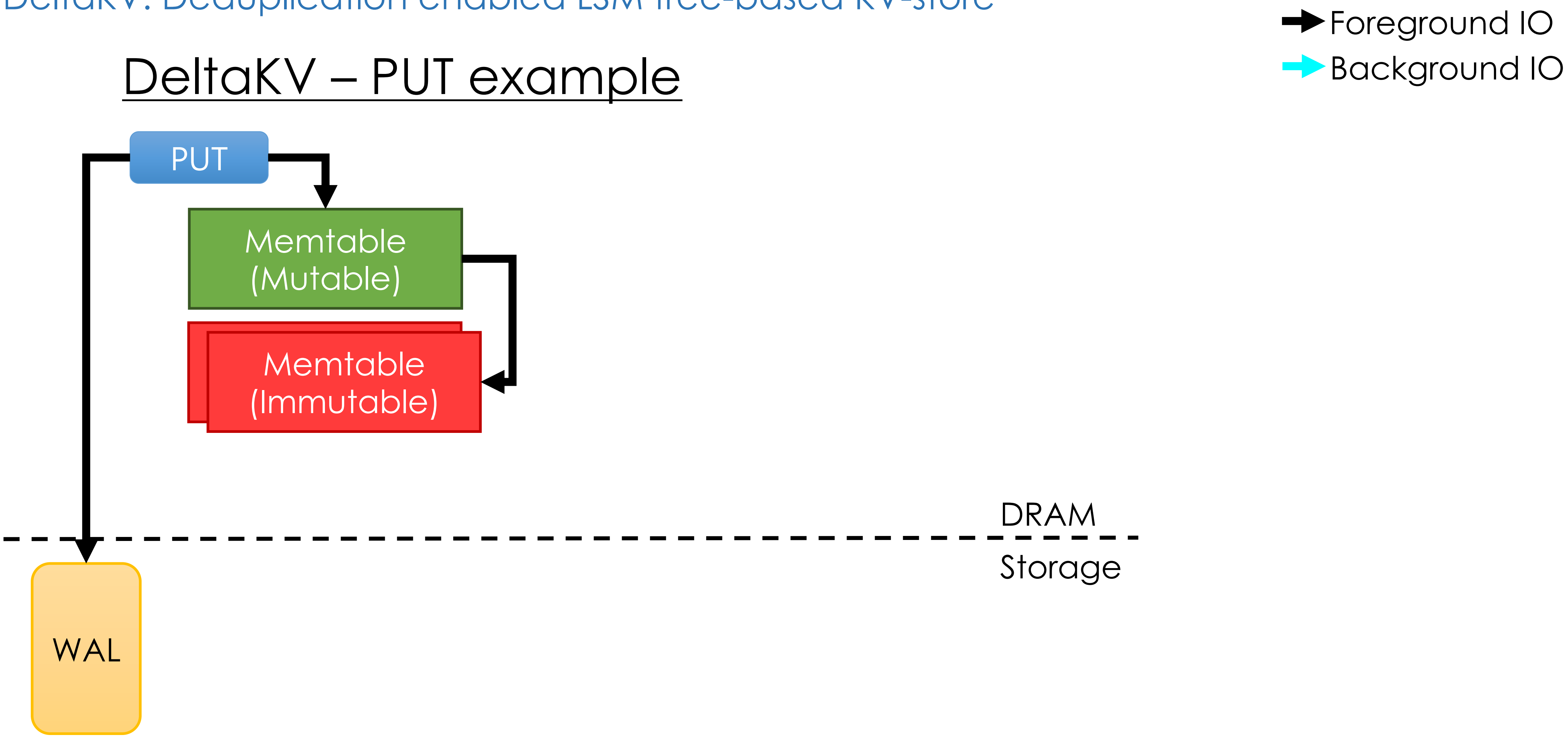
B+-Tree node contains: Keys and their corresponding offsets.

DeltaKV – PUT example

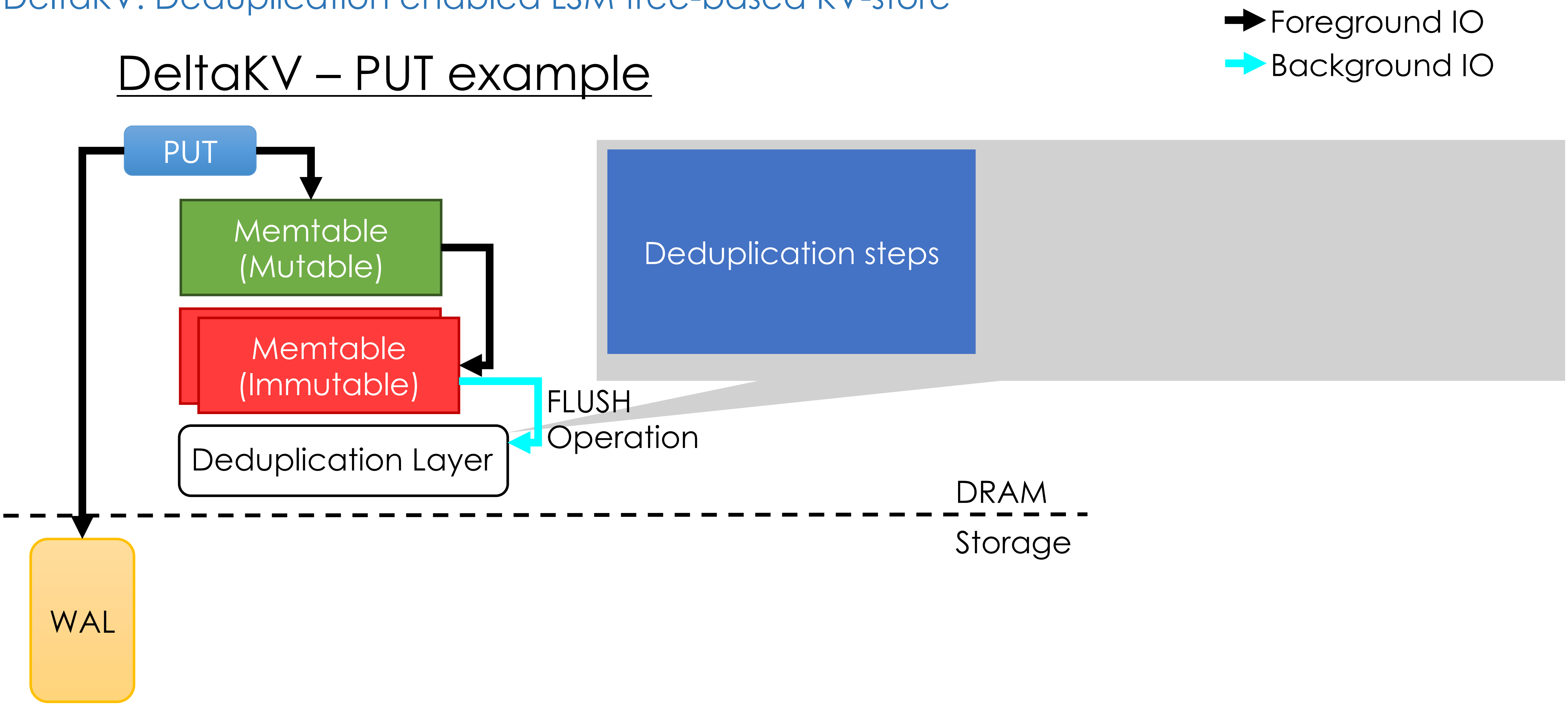


B+-Tree node contains: Keys and their corresponding offsets.

DeltaKV – PUT example

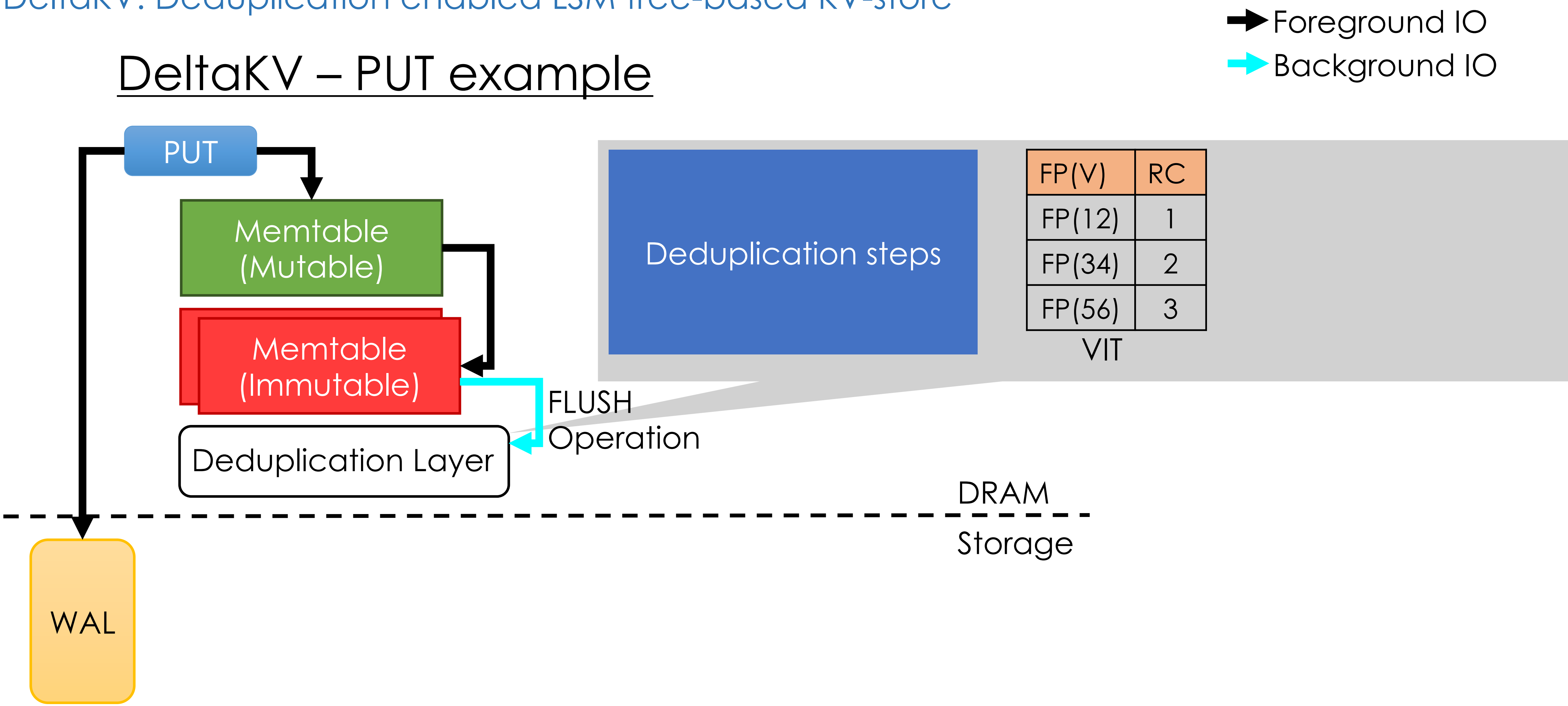


DeltaKV – PUT example



MKK B+-Tree node contains: Keys and their corresponding offsets.

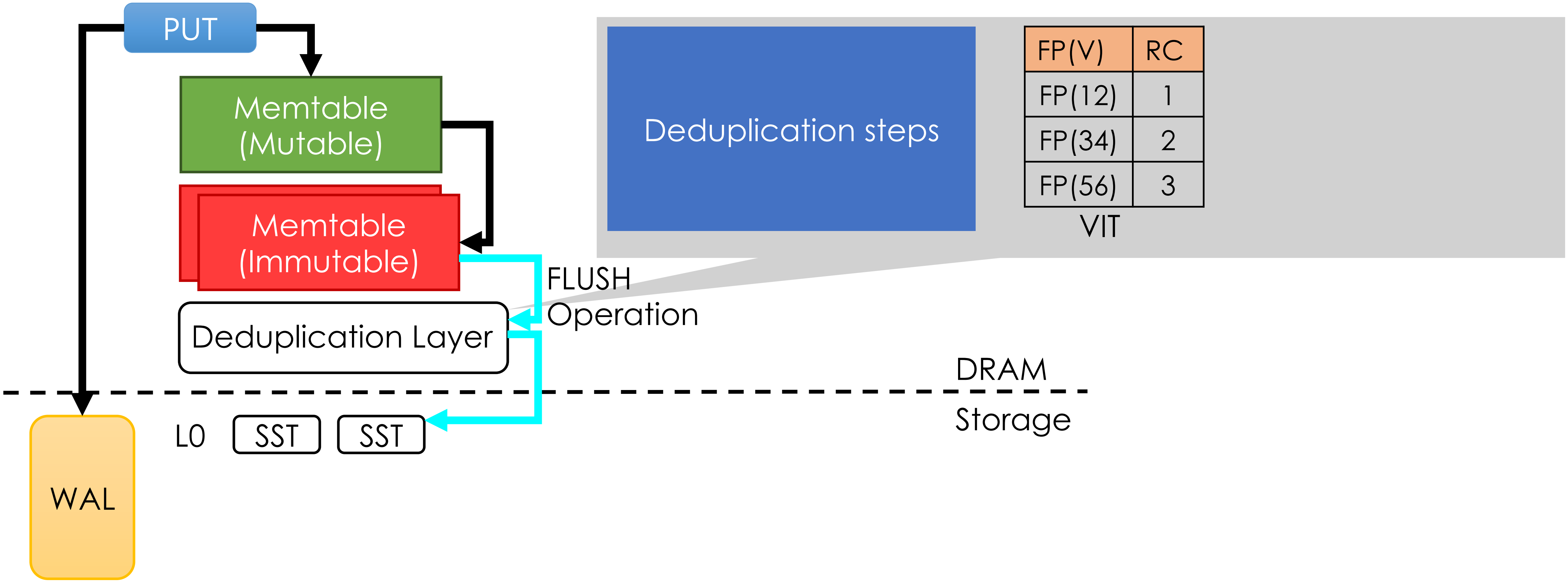
DeltaKV – PUT example



B+-Tree node contains: Keys and their corresponding offsets.

DeltaKV: Deduplication enabled LSM tree-based KV-store

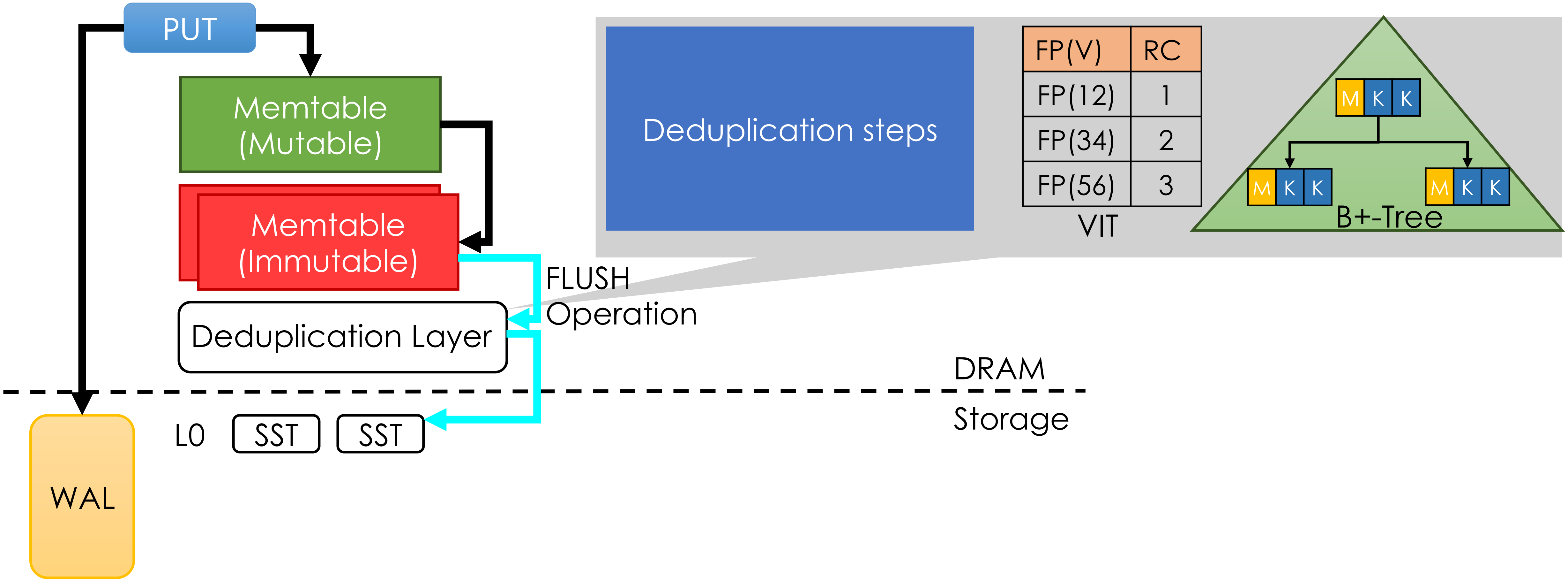
DeltaKV – PUT example



B+-Tree node contains: Keys and their corresponding offsets.

DeltaKV – PUT example

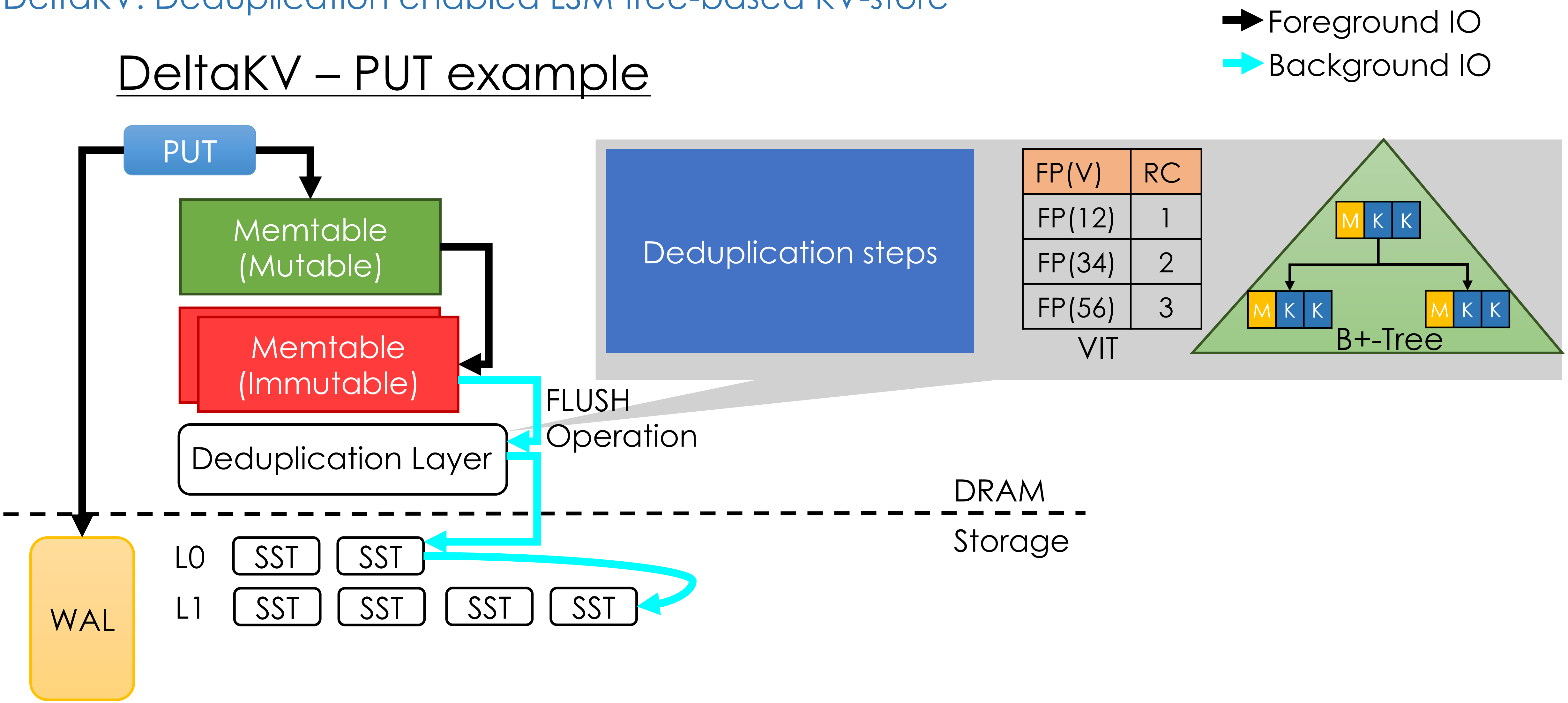
Foreground IO
Background IO



M K K B+-Tree node contains: Keys and their corresponding offsets.

DeltaKV: Deduplication enabled LSM tree-based KV-store

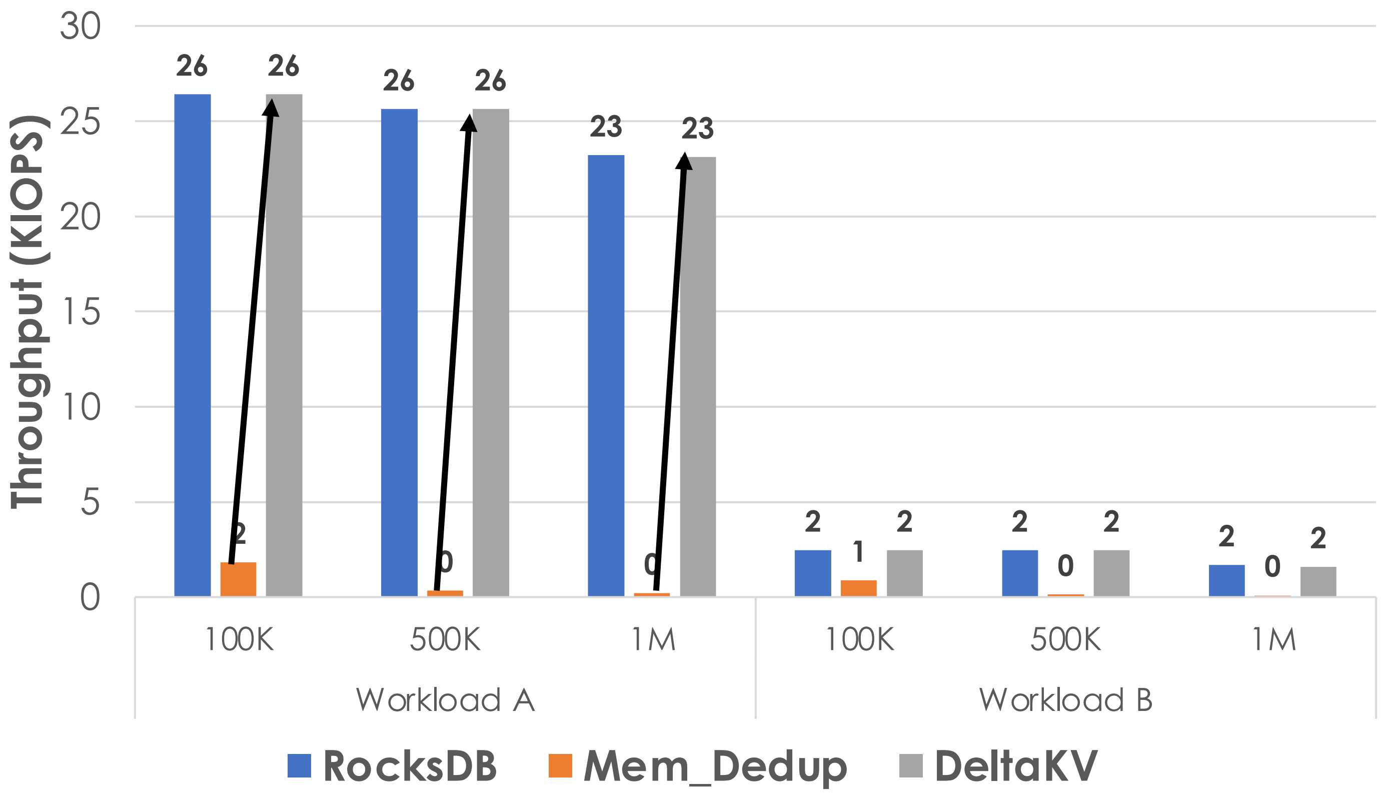
DeltaKV – PUT example



M K K B+-Tree node contains: Keys and their corresponding offsets.

DeltaKV vs MemDedup

- Preliminary evaluation
  - DeltaKV → VIT integration at FLUSH operation
  - Similar performance as baseline RocksDB



Concluding Remarks

- Explored data deduplication for LSM tree-based KV-stores
- MemDedup
  - Inline deduplication
  - Intercepts operations at the Memtable-level
  - Suffers from huge performance drop and inconsistency issue
- DeltaKV
  - Offline deduplication → FLUSH operation
  - Background thread performs dedup
  - Maintains the performance of LSM tree

Thank you

Safdar Jamil
Email: safdar@sogang.ac.kr
Site: https://sites.google.com/view/safdarjamil95