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

Safdar Jamil¹, Awais Khan², Youngjae Kim¹

¹Sogang University, Seoul, South Korea
²Oak 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

![Deduplication Diagram]

Original Data

Deduplicated Data
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)
Approach I: Inline Deduplication at Memtable

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
**Approach I: Inline Deduplication at Memtable**

**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

<table>
<thead>
<tr>
<th>Fingerprint of value FP(V)</th>
<th>List of parent keys (PK[])</th>
<th>Reference count (RC)</th>
<th>Value location (Offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP(12)</td>
<td>K1</td>
<td>1</td>
<td>0xf8</td>
</tr>
<tr>
<td>FP(34)</td>
<td>K2, K3</td>
<td>2</td>
<td>0xff</td>
</tr>
</tbody>
</table>

Chunk Information Table
Approach I: Inline Deduplication at Memtable

Deduplication at Memtable-level (MemDedup)

WAL

Deduplication Layer

DRAM Storage
Approach I: Inline Deduplication at Memtable

Deduplication at Memtable-level (MemDedup)

- PUT
- Deduplication Layer
- Fingerprinting
- WAL
- DRAM Storage
Approach I: Inline Deduplication at Memtable

Deduplication at Memtable-level (MemDedup)

PUT

Deduplication Layer

Fingerprinting
Duplicate detection

DRAM Storage

WAL
Approach I: Inline Deduplication at Memtable

Deduplication at Memtable-level (MemDedup)

- **PUT**
  - Deduplication Layer
  - Fingerprinting
  - Duplicate detection
  - Metadata update
  - CIT

<table>
<thead>
<tr>
<th>FP(V)</th>
<th>PK[]</th>
<th>RC</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP(12)</td>
<td>K1</td>
<td>1</td>
<td>0xf8</td>
</tr>
</tbody>
</table>

- WAL
- DRAM Storage
Approach I: Inline Deduplication at Memtable

Deduplication at Memtable-level (MemDedup)

- PUT
- Deduplication Layer
  - Memtable (Mutable)
- WAL
- DRAM Storage
- Fingerprinting
- Duplicate detection
- Metadata update

<table>
<thead>
<tr>
<th>FP(V)</th>
<th>PK[]</th>
<th>RC</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP(12)</td>
<td>K1</td>
<td>1</td>
<td>0xf8</td>
</tr>
</tbody>
</table>

CIT
Approach I: Inline Deduplication at Memtable

Deduplication at Memtable-level (MemDedup)

- PUT
- Deduplication Layer
- Memtable (Mutable)
- Memtable (Immutable)
- Fingerprinting
- Duplicate detection
- Metadata update

<table>
<thead>
<tr>
<th>FP(V)</th>
<th>PK[]</th>
<th>RC</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP(12)</td>
<td>K1</td>
<td>1</td>
<td>0xf8</td>
</tr>
</tbody>
</table>

CIT

DRAM Storage

WAL
Deduplication at Memtable-level (MemDedup)

**Approach I: Inline Deduplication at Memtable**

- **Deduplication Layer**
  - **Memtable (Mutable)**
  - **Memtable (Immutable)**
  - **DRAM Storage**
  - **WAL**

- **Fingerprinting**
- **Duplicate detection**
- **Metadata update**

<table>
<thead>
<tr>
<th>FP(V)</th>
<th>PK[]</th>
<th>RC</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP(12)</td>
<td>K1</td>
<td>1</td>
<td>0xf8</td>
</tr>
</tbody>
</table>

**CIT**
Approach I: Inline Deduplication at Memtable

Deduplication at Memtable-level (MemDedup)

Deduplication Layer

Memtable (Mutable) → Memtable (Immutable)

Fingerprinting
Duplicate detection
Metadata update

Table:
<table>
<thead>
<tr>
<th>FP(V)</th>
<th>PK[]</th>
<th>RC</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP(12)</td>
<td>K1</td>
<td>1</td>
<td>0xf8</td>
</tr>
</tbody>
</table>

CIT

WAL

L0  SST  SST
L1  SST  SST  SST  SST

DRAM
Storage
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

<table>
<thead>
<tr>
<th></th>
<th>PK[]</th>
<th>RC</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP(12)</td>
<td>K1, K2</td>
<td>1</td>
<td>0xf8</td>
</tr>
</tbody>
</table>

Approach I: Inline Deduplication at Memtable
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

Drum

<table>
<thead>
<tr>
<th>FP(V)</th>
<th>PK[]</th>
<th>RC</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP(12)</td>
<td>K1, K2</td>
<td>1</td>
<td>0xf8</td>
</tr>
<tr>
<td>FP(34)</td>
<td>K1</td>
<td>1</td>
<td>0xf8</td>
</tr>
</tbody>
</table>

CIT
Deduplication at Memtable-level (MemDedup)

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

### Deduplication Layer
- PUT(K1, 34)

### Deduplication steps

<table>
<thead>
<tr>
<th>FP(V)</th>
<th>PK[]</th>
<th>RC</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP(12)</td>
<td>K2</td>
<td>1</td>
<td>0xf8</td>
</tr>
<tr>
<td>FP(34)</td>
<td>K1</td>
<td>1</td>
<td>0xf8</td>
</tr>
</tbody>
</table>

CIT

**Approach I: Inline Deduplication at Memtable**
Deduplication at Memtable-level (MemDedup)

- **Limitations**
  1. High performance overhead
  2. Inconsistency issue

- In-place update operations at Memtable

---

**Deduplication Layer**

- PUT(K1, 34)

**Deduplication steps**

<table>
<thead>
<tr>
<th>FP(V)</th>
<th>PK[]</th>
<th>RC</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP(12)</td>
<td>K2</td>
<td>1</td>
<td>0xf8</td>
</tr>
<tr>
<td>FP(34)</td>
<td>K1</td>
<td>1</td>
<td>0xf8</td>
</tr>
</tbody>
</table>

**CIT**

- K2 points to invalid value

---

**Approach I: Inline Deduplication at Memtable**

- WAL
- DRAM
- Storage
Deduplication enabled LSM Tree-based Key-Value Store (DeltaKV)
DeltaKV: Deduplication enabled LSM tree-based KV-store

**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: Deduplication enabled LSM tree-based KV-store

DeltaKV – PUT example

B+-Tree node contains: Keys and their corresponding offsets.
DeltaKV: Deduplication enabled LSM tree-based KV-store

DeltaKV – PUT example

- PUT
  - Memtable (Mutable)
  - WAL
  - DRAM Storage

- Foreground IO
- Background IO

B+-Tree node contains: Keys and their corresponding offsets.
DeltaKV: Deduplication enabled LSM tree-based KV-store

DeltaKV – PUT example

- **PUT**
  - Memtable (Mutable)
  - Memtable (Immutable)

- WAL

- DRAM Storage

**Key Points:**
- **WAL**
- **Memtable (Mutable)**
- **Memtable (Immutable)**

**Diagram Notes:**
- Foreground IO
- Background IO

**Textual Notes:**
- B+-Tree node contains: Keys and their corresponding offsets.
DeltaKV: Deduplication enabled LSM tree-based KV-store

**DeltaKV – PUT example**

- **PUT**
  - Memtable (Mutable)
  - Memtable (Immutable)
  - Deduplication Layer
- **FLUSH**
  - Operation
- **WAL**
- **Deduplication steps**
- **DRAM Storage**

**B+-Tree node contains:** Keys and their corresponding offsets.
DeltaKV: Deduplication enabled LSM tree-based KV-store

DeltaKV – PUT example

<table>
<thead>
<tr>
<th>Deduplication steps</th>
<th>FP(V)</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP(V)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FP(12)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>FP(34)</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>FP(56)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B+-Tree node contains: Keys and their corresponding offsets.
DeltaKV: Deduplication enabled LSM tree-based KV-store

**DeltaKV – PUT example**

- **Memtable (Mutable)**
- **Memtable (Immutable)**
- **Deduplication Layer**
- **FLUSH Operation**
- **WAL**
- **L0**
- **SST**
- **SST**
- **Deduplication steps**
  - FP(V)
  - FP(12)
  - FP(34)
  - FP(56)
  - RC
  - 1
  - 2
  - 3
  - VIT

- **DRAM Storage**

- **B+-Tree node contains**: Keys and their corresponding offsets.
DeltaKV: Deduplication enabled LSM tree-based KV-store

**DeltaKV – PUT example**

**Deduplication Layer**

- **Memtable (Mutable)**
- **Memtable (Immutable)**

**FLUSH Operation**

**DRAM Storage**

**WAL**

**Deduplication steps**

<table>
<thead>
<tr>
<th>FP(V)</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP(12)</td>
<td>1</td>
</tr>
<tr>
<td>FP(34)</td>
<td>2</td>
</tr>
<tr>
<td>FP(56)</td>
<td>3</td>
</tr>
</tbody>
</table>

**B+-Tree**

- **M K K**

B+-Tree node contains: Keys and their corresponding offsets.
DeltaKV: Deduplication enabled LSM tree-based KV-store

**DeltaKV – PUT example**

**Memtable (Mutable)**

**Memtable (Immutable)**

**Deduplication Layer**

**FLUSH Operation**

**Deduplication steps**

<table>
<thead>
<tr>
<th>FP(V)</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP(12)</td>
<td>1</td>
</tr>
<tr>
<td>FP(34)</td>
<td>2</td>
</tr>
<tr>
<td>FP(56)</td>
<td>3</td>
</tr>
</tbody>
</table>

**B+-Tree**

**WAL**

**DRAM Storage**

**B+-Tree node contains: Keys and their corresponding offsets.**
DeltaKV: Deduplication enabled LSM tree-based KV-store

DeltaKV vs MemDedup

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

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