Understanding RocksDB and the Log-Structured Merge-tree (LSM Tree)

1 Introduction

RocksDB is an embeddable, high-performance key-value store developed by Facebook, based on Google's LevelDB. It is optimized for fast, low-latency storage, especially on flash and SSD devices. A defining feature of RocksDB is its use of the Log-Structured Merge-tree (LSM tree) as the underlying data structure for efficient writes, reads, and space management.

2 What is an LSM Tree?

An LSM tree is a write-optimized data structure that transforms random writes into sequential disk operations. It consists of multiple levels of sorted key-value data, with newer data in memory and older data in immutable, sorted files on disk.

2.1 Core Components

- **MemTable**: An in-memory write buffer implemented as a sorted skip list. It temporarily holds incoming writes before they are flushed to disk.
- **Immutable MemTable**: Once full, the MemTable becomes immutable and is scheduled for flush to disk.
- **SSTable (Sorted String Table)**: An immutable file on disk storing key-value pairs in sorted order.
- **Write-Ahead Log (WAL)**: Ensures durability by logging updates before they enter the MemTable.
- **Compaction**: Merges and reorganizes SSTables to reduce read amplification, reclaim space, and maintain sorted order.

3 Write Path in RocksDB

1. **Write**: A key-value pair is first appended to the WAL and inserted into the MemTable.
2. **Flush**: When the MemTable is full, it is flushed to disk as an SSTable at Level 0.
3. **Compaction**: Periodically, RocksDB merges SSTables across levels, preserving key order and resolving overwritten or deleted keys.

4 Multi-Level Example: Inserts and Compactions

Suppose we insert keys `D`, `B`, `A`, `C`, `E`, `F`, `G`, `H`, `I`, `J` sequentially.

4.1 Initial State: MemTable and Flush to Level 0

```
MemTable:
A -> vA
B -> vB
C -> vC
D -> vD
E -> vE
```

Flushed to SSTable:

```
Level 0:
SSTable 1: A -> vA, B -> vB, C -> vC, D -> vD, E -> vE
```

Next batch:

```
MemTable:
F -> vF
G -> vG
H -> vH
I -> vI
J -> vJ
```

Flushed to:

```
Level 0:
SSTable 2: F -> vF, G -> vG, H -> vH, I -> vI, J -> vJ
```

4.2 Compaction to Level 1

Both SSTables in Level 0 are merged into a single Level 1 SSTable:

```
Level 1:
A -> vA
B -> vB
C -> vC
D -> vD
E -> vE
F -> vF
G -> vG
H -> vH
I -> vI
J -> vJ
```

Now suppose we insert: `B -> vB2`, `D -> vD2`, `K -> vK`, `L -> vL`, `M -> vM`

```
MemTable:
B -> vB2
D -> vD2
K -> vK
L -> vL
M -> vM
```

Flushed to new Level 0 SSTable:

```
Level 0:
SSTable 3: B -> vB2, D -> vD2, K -> vK, L -> vL, M -> vM
```

Compaction merges into Level 1:

```
Level 1:
A -> vA
B -> vB2
C -> vC
D -> vD2
E -> vE
F -> vF
G -> vG
H -> vH
I -> vI
J -> vJ
K -> vK
L -> vL
M -> vM
```

5 Read Path: Lookups and Scans

5.1 Point Lookups

When reading a key (e.g., `D`):

1. Check the MemTable.
2. Check the Immutable MemTable (if exists).
3. Search SSTables in Level 0 (unordered, may need multiple files).
4. Search SSTables in Level 1 and beyond (non-overlapping, binary search by key range).

To speed this up:

- RocksDB uses **Bloom Filters** to skip SSTables that do not contain the key.
- **Index blocks** and **filter blocks** in SSTables accelerate key-location.
- SSTables are divided into blocks, and a block index enables binary search within a file.

Example: To find key `D`, RocksDB first checks the MemTable and WAL. If not found, it checks Level 0 SSTables (where key ranges may overlap). Using Bloom filters and key-range metadata, it narrows down to relevant files. It then uses the block index within the SSTable to locate the correct block and retrieves `vD2`.

5.2 Range Scans

To scan keys `D` to `H`:

1. Construct iterators for MemTable, Immutable MemTable, and SSTables across all levels.
2. Use a priority queue (heap) to perform a k-way merge of sorted sources.
3. Skip duplicate keys (e.g., old versions in lower levels).

Example: If `D -> vD2` is in Level 0 and `D -> vD` is in Level 1, the scan returns `vD2`. The merge iterator ensures sorted output: `D, E, F, G, H`.

6 Key Ideas for Performance

- **Sequential Writes**: Appends to WAL and MemTable are fast and sequential.
- **Write Buffering**: MemTables absorb writes before disk operations are needed.
- **Compaction Scheduling**: RocksDB schedules compactions to maintain balance between read and write cost.
- **Bloom Filters and Index Blocks**: Reduce unnecessary reads during point lookups.
- **Compression and File Layout**: RocksDB optionally compresses SSTables and uses block-level structure to reduce I/O.
- **Block Cache**: Frequently accessed data blocks are cached in memory to avoid disk reads.

7 Limitations and Comparison with B+-Trees

While LSM trees provide excellent write performance, they also introduce certain drawbacks, particularly for read-heavy workloads:

- **Read Amplification**: Since keys may exist in multiple SSTables across levels, reads may require checking multiple files before finding a match or confirming absence.
- **Latency Spikes from Compaction**: During heavy compaction events, disk I/O usage increases significantly, causing potential latency spikes for ongoing queries.
- **Overhead for Short Scans or Random Accesses**: If range scans are small or highly random, the overhead of merging iterators across multiple levels and files may outweigh the benefits.
- **Bloom Filter False Positives**: Although Bloom filters speed up lookups, false positives still result in wasted disk seeks.

7.1 Concrete Examples of Bad Cases

Example 1: High Read Amplification
Suppose key `Z` has never been written. A lookup for `Z` would need to check the MemTable, Immutable MemTable, multiple overlapping Level 0 SSTables, and potentially each file in Level 1, Level 2, and deeper, before confirming it doesn't exist. Even with Bloom filters, false positives at several levels can result in multiple unnecessary disk accesses.

Example 2: Inefficient Random Access
Suppose an application performs random lookups for 1000 different keys spread uniformly across the entire keyspace. Since LSM trees organize data by level and not by access locality, these 1000 reads may hit many different SSTables across levels. In contrast, a B+-tree would navigate through a balanced structure, fetching fewer blocks due to its strong locality of reference.

Example 3: Short Range Scans
Suppose an application frequently queries small ranges like keys `X100` to `X105`. Because RocksDB needs to perform a k-way merge over all relevant SSTables and the MemTable, the setup overhead for the iterator and merging process may dominate the actual cost of reading the five keys. A B+-tree would locate the first leaf and scan linearly, offering faster performance for small ranges.

7.2 When B+-Trees Might Be Better

B+-trees, as used in traditional relational databases, excel in the following cases:

- **Low Write Rate with Frequent Reads**: B+-trees maintain data locality, resulting in fewer disk seeks for range queries or index scans.
- **Small Updates or Deletes**: Since B+-trees update in-place, they avoid the cost of compactions and duplicate versions.
- **Predictable Latency**: B+-trees do not perform background merges, making read latencies more stable.

8 Conclusion

RocksDB combines the LSM tree's write efficiency with clever strategies for minimizing read overhead. By staging writes in memory and managing compactions efficiently, it supports high-throughput ingestion. Optimizations like Bloom filters, index caching, and non-overlapping level design enable fast lookups and scans even at scale. This makes RocksDB suitable for use cases such as metadata stores, message queues, and time-series data systems.