Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience

wu

April 22, 2024

RocksDB is a key-value store targeting large-scale distributed systems and optimized for Solid State Drives (SSDs). This paper describes how our priorities in developing RocksDB have evolved. We describe how and why RocksDB's resource optimization target migrated from write amplification, to space amplification, to CPU utilization.

Lessons from running large-scale applications taught us that:

1. resource allocation needs to be managed across different RocksDB instances,

2. data format needs to remain backward and forward compatible to allow incremental software rollout,

3. appropriate support for database replication and backups are needed.

Lessons from failure handling taught us that:

1. data corruption errors needed to be detected earlier and at every layer of the system.

1 Introduction

Each RocksDB instance manages data on storage devices of just a single server node; it does not handle any inter-host operations, such as replication and load balancing, and it does not perform high-level operations, such as checkpoints

RocksDB and its various components are highly customizable, customizations can include the write-ahead log (WAL) treatment, the compression strategy, and the compaction strategy. RocksDB may be tuned for high write throughput or high read throughput, for space efficiency, or something in between.

Used in

- **Database**:
- **Stream processing**:
- **Logging/queuing services**:
- **Index service**:
- **Caching on SSD**:

Table 1: RocksDB use cases and their workload characteristics

	Read/Write	Read Types	Special Characteristics
Databases	Mixed	Get + Iterator	Transactions and backups
Stream Processing	Write-Heavy	Get or Iterator	Time window and checkpoints
Logging/Queues	Write-Heavy	Iterator	Support on HDD too
Index Services	Read-Heavy	Iterator	Bulk loading
Cache	Write-Heavy	Get	Can drop data

Table 2: System metrics for a typical use case from each application category

	CPU	Space Util	Flash Endurance	Read Bandwidth
Stream Processing	11%	48%	16%	1.6%
Logging/Queues	46%	45%	7%	1.0%
Index Services	47%	61%	5%	10.0%
Cache	3%	78%	74%	3.5%

2 Background

2.1 Embedded storage on flash based SSDs

The high performance of the SSD, in many cases, also shifted the performance bottleneck from device I/O to the network for both of latency and throughput. It became more attractive for applications to design their architecture to store data on local SSDs rather than use a remote data storage ser- vice. This increased the demand for a key-value store engines that are embedded in applications.

2.2 RocksDB architecture

Writes. Whenever data is written to RocksDB, it is added to an in-memory write buffer called **MemTable**, as well as an on-disk **Write Ahead Log (WAL)**. Memtable is implemented as a skiplist so keep the data ordered with $O(\log n)$ insert and search overhead. The WAL is used for recovery after a failure, but is not mandatory. Once the size of the MemTable reaches a configured size, then

1. the MemTable and WAL become immutable,
2. a new MemTable and WAL are allocated for subsequent writes,
3. the contents of the MemTable are flushed to a "Sorted String Table" (SSTable) data file on disk,
4. the flushed MemTable and associated WAL are discarded.

Each SSTable stores data in sorted order, divided into uniformly-sized blocks. Each SSTable also has an index block with one index entry per SSTable block for binary search.

Compaction. Levels higher than Level-0 are created by a process called **compaction**. The size of SSTables on a given level are limited by configuration parameters. When level-L's size target is exceeded, some SSTables in level-L are selected and merged with the overlapping SSTables in level-(L+1). This process gradually migrates written data from Level-0 to the last level. Compaction I/O is efficient as it can be parallelized and only involves bulk reads and writes of entire files.

Reads. In the read path, a key lookup occurs at each successive level until the key is found or it is determined that the key is not present in the last level.

RocksDB supports multiple different types of compaction:

- **Leveled Compaction**: levels are assigned exponentially increasing size
- **Tiered Compaction** (**Universal Compaction** in RocksDB): Similar to Cassandra or HBase. Multiple sorted runs are lazily compacted together, either when there are too many sorted runs, or the ratio between total DB size over the size of the largest sorted run exceeds a configurable threshold.

- **FIFO Compaction**: discards old files once the DB hits a size limit and only performs lightweight compaction. It targets in-memory caching applications.

#+CAPTION Write amplification, overhead and read I/O for three compaction types

Compaction	Leveled	Tiered	FIFO
Write Amplification	16.07	4.8	2.14
Max Space Overhead	9.8%	94.4%	N/A
Avg Space Overhead	9.5%	45.5%	N/A
# I/O per Get() with bloom filter	0.99	1.03	1.16
# I/O per Get() without bloom filter	1.7	3.39	528
# I/O per iterator seek	1.84	4.80	967

3 Evolution of resource optimization targets

3.1 Write amplification

Write amplification emerges at two levels:

1. SSDs themselves introduct write amplification: by their observation between 1.1 and 3.

2. Storage and database software also generae write amplification; this can sometimes be as high as 100 (e.g., when an entire 4KB/8KB/16KB page is written out for changes of less than 100 bytes)

   Level Compaction in RocksDB usually exhibits write amplification between 10 and 30, which is several times better than when using B-trees in many cases.

	# keys (millions)	200	400	600	800	1000
Dynamgic Leveled	Fully compacted size (GB)	12.0	24.0	36.0	48.0	60.1
	Steady DB size (GB)	13.5	26.9	40.4	54.2	67.5
	Space overhead (%)	12.4	11.8	12.2	12.7	12.4
LevelDB-style Compaction	Fully Compacted size (GB)	12.0	24.0	36.4	48.3	60.3
	Steady DB size (GB)	15.1	26.9	42.5	57.9	73.8
	Space overhead (%)	25.6	12.2	16.9	19.7	22.4