intel XEON

White Paper

Data Center and Network
4th Gen Intel® Xeon® Scalable Processors

Accelerating RocksDB Compression with Intel® QuickAssist Technology

An overview of software optimizations that are enabled by 4th Gen Intel® Xeon® Scalable processors.

Authors

Mohamed Issa
Cloud Software Development Engineer

Luca Giacchino
Cloud Software Development Engineer

Smita Kumar
Silicon Architecture Engineer

David Qian
Cloud Software Development Engineer

Brian Will
Principal Engineer

Table of Contents

Abstract 1
Introduction 2
RocksDB: High-performance data management and storage 2
Workload architecture of RocksDB 2
Test system configuration 4
Hardware setup 4
Software setup 5
Benchmarks 6
Read-only 8
Read/write 11
Write-only 13
Conclusion 15

Abstract

RocksDB is an open source embedded persistent key-value storage engine maintained by Meta. This engine is used in many popular applications, such as MongoDB, Apache Samza, and Redis. It is written in C++ for maximum performance with fast storage (such as solid-state drives [SSDs]) while also being adaptable to different workloads. Additionally, it offers users the option to compress storage data using different approaches, such as Zstd and LZ4. However, adding a new compression method to RocksDB can require a non-trivial amount of work because it only supports a predefined set of algorithms with no interface for extension.

To solve this problem, Intel has designed a new pluggable compression framework that makes existing compression components much more modular. This framework also allows new components to be added as plugins hosted separately from the core RocksDB repository in GitHub. With this approach, developers can avoid the extra time and effort required to modify RocksDB whenever a new compression format is needed, instead focusing on optimizing the developed plugin. With this new framework, hardware-accelerated compression opportunities are now easily available to RocksDB applications with a performance uplift over pure software compute implementations.

This paper discusses the RocksDB integration with hardware-based Intel QuickAssist Technology (Intel® QAT) and QAT-DEFLATE-based compression performance improvements over software-based Zstd compression (note that QAT-Zstd-based compression is another option not discussed in this paper). Intel QAT is an accelerator on 4th Gen Intel Xeon Scalable processors and newer Intel Xeon processors, which integrate multiple built-in accelerators to deliver enhanced performance across various established and emerging workloads. Depending on the CPU model, the number and types of accelerators might vary, cost-effectively matching diverse workload needs at data centers. Intel QAT provides hardware acceleration for compression/decompression and common cryptographic operations, enabling higher queries per second, memory compaction, and efficient database scans.

All results described in this paper were captured from benchmark testing on 4th Gen Intel Xeon Scalable processors with the built-in Intel QAT accelerator. The tests collected performance data across three key workloads: read-only, read/write, and write-only.

- In the read-only workload, throughput increased by as much as 23 percent, and P99 latency dropped by as much as 24 percent.
- In the read/write (80 percent read, 20 percent write) workload, throughput increased by as much as 27 percent, and P99 latency dropped by as much as 21 percent.
- Finally, in the write-only workload, throughput increased by as much as 77 percent.

These performance benefits have a data-size variance of ±2 percent.

Introduction

Persistent key-value stores (KV-stores) are used by many applications that manage data in production environments. High performance, flexibility, and ease of use have largely driven the widespread adoption of KV-stores by applications that retrieve, process, and store data at scale. In each of these applications, a storage engine orchestrates data transformations in hardware.[1] A database management system (DBMS) is one such application type that follows a client/server model, in which server nodes contain the database system, and client nodes make requests to retrieve or modify data. Each DBMS storage engine subsystem typically contains access methods for managing and organizing data on storage devices (such as disk, flash, memory, and cache).[2]

It follows then that the foundation of any KV-store is the access method used. Furthermore, the performance characteristics of different access methods will vary due to the underlying algorithms and data structures. There are three types of overhead associated with a storage engine's access methods: read overhead (R), update (or write) overhead (U), and memory (or storage) overhead (M). Collectively, these are referred to as "RUM overheads," and optimizing them requires careful consideration of the hardware and workloads involved.[3]

Given these considerations, the most desirable access methods should simultaneously minimize each aspect of the RUM overheads. Put another way, an ideal access method would have the lowest read cost, the lowest update cost, and no extra memory footprint over that of the original dataset. In this context, "cost" refers to the time needed to complete the operation. Unfortunately, minimizing all three parameters is impossible because of their interdependence. This situation is known as the "RUM conjecture," which states that an access method that sets ceiling values for two RUM overheads automatically sets a floor value for the remaining overhead. Additionally, finding the lowest value for an access method's read, update, or memory cost parameter means that the remaining two cost parameters cannot be optimized. For example, an access method could minimize the required memory footprint by storing data without applying an ordering scheme. However, on average, this approach would lead to sub-optimal read and update overheads because of the increased work required to find the right locations in memory upon which to operate.[4]

RocksDB: High-performance data management and storage

RocksDB is an open source persistent KV-store developed and maintained by Meta (formerly Facebook). To achieve high-performance, it is written in C++ while representing keys and values as arbitrarily-sized byte streams. With fast, low-latency storage, such as flash drives and high-speed disk drives, an application using RocksDB as a storage engine can achieve the maximum read/write rates available to those devices by using certain data structures and deferring expensive background operations, so they are less likely to affect user-visible performance. RocksDB is also highly adaptable to different workloads, and it can provide access to many types of operations (such as read, write, and delete) through its public APIs.[5]

Higher-level applications that embed RocksDB as a library component can access its operations, managing data on storage devices for a single node. This means RocksDB does not handle transfers between nodes, replication, load-balancing, and other high-level operations. However,due to its easily customizable nature, RocksDB is adopted by many production-quality applications, including mixed-use database systems, write-heavy stream-processing services, write-heavy logging/queuing services, read-heavy indexing services, and write-heavy caching services. Table 1 lists several services (or applications) that use RocksDB as a storage engine.[6]

Table 1. Services that use RocksDB as a storage engine[6]

Type	Service	Description
Databases	MongoDB	A NoSQL implementation written in C++, where every database contains a set of collections. Unlike structures in relational implementations (such as MySQL), a collection has no predefined schema and instead stores data as BSON documents (binary-encoded JSON-like objects).[7]
	TiDB (PingCAP)	An open source implementation that supports hybrid transactional and analytical processing (HTAP) workloads. It is compatible with MySQL and provides horizontal scalability, strong consistency, and high availability.[8]
	UDB (Meta)[9]	The main Facebook user database that serves social activities such as likes, comments, and shares. It uses a wrapper layer called MyRocks over RocksDB for compatibility with the MySQL pluggable storage engine architecture.[10]
	ZippyDB (Meta)	A distributed KV-store that uses RocksDB as the storage node to achieve data persistence and reliability. Typically, ZippyDB is used by Facebook services to store photo metadata and object metadata.[1]
	UP2X (Meta)	A special distributed KV-store that's based on RocksDB. It stores the profile data (such as counters and statistics) used for the prediction and inferencing of several artificial intelligence (AI)/machine learning (ML) services in Facebook.[1]
Stream processing	Apache Flink	A framework and distributed processing engine for stateful computations over unbounded (defined start but no defined end) and bounded (defined start and defined end) data streams. RocksDB serves as one possible back end for accessing and storing data.[11]
	Apache Kafka Streams	A library built for Apache Kafka, an open source distributed event-streaming platform used for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications. A built-in library that simplifies development and uses RocksDB as the default KV-store.[12]

Type	Service	Description
Stream processing	Apache Samza	A distributed system for stateful and fault-tolerant stream processing. It is capable of processing infinite streams of events and finite datasets from streaming sources (such as Kafka), database snapshots (such as Databus), or file systems (such as Apache HDFS).[13]
	Stylus (Meta)	A low-level real-time framework written in C++ that processes data from web and mobile applications in Facebook. It can operate in a stateful or stateless manner.[14]
Logging/ queues	LogDevice (Meta)	A distributed data store designed specifically for logs, which means it is highly available and provides durable record storage. It is written in C++ and used by several Facebook services.[15]
Indexing	Dragon (Meta)	A distributed graph query engine used in Facebook. It creates several different types of indices that improve the efficiency of fetching, filtering, and reordering data to reduce network usage, improve latency, and free CPU cycles for other tasks.[16]
	Rockset	A real-time analytics database service (accessible through a RESTful interface) for serving low-latency, high-concurrency analytical queries at scale. It builds a converged index on structured and semi-structured data from online transaction processing (OLTP) databases, streams, and lakes in real time.[17]
Caching	Pika (Qihoo)	A persistent storage service designed to handle large amounts of data while also being compatible with Redis interfaces. There are many commercial users of this service, including Meituan and Xiaomi.[18]
	Redis	This remote dictionary server is a popular in-memory KV-store that allows keys to be complex data types such as hashes, lists, sets, and sorted sets. Furthermore, it enables complex atomic operations on these data types (for example, enqueuing and dequeuing from a list or inserting a new value with a given score to a sorted set).[19]

More than a third of the services listed in Table 1 are internal to Meta (except LogDevice), which initially developed and maintained these services to manage data on fast storage devices. It is important to note that Meta's in-house services have different requirements and usage characteristics, which underscores the highly adaptable and tunable nature of RocksDB as a storage engine. As other companies and organizations continue to adopt and customize RocksDB, every optimization opportunity that improves performance or reduces data size becomes increasingly valuable.

Workload architecture of RocksDB

All aspects of RocksDB are based on three key design principles. First, it is performant for fast storage and server workloads. Point lookups (single key-value pair accesses) and range scans (ordered multi-key-value pair accesses) are efficient, while high random read, high update, or mixed scenarios are accommodated with configuration options. Second, there are built-in tools and utilities that help users deploy or debug in production environments. Those looking to adjust performance can use the available tuning parameters. Third, RocksDB is backwards-compatible with previous versions, simplifying upgrades.[20]

With these fundamental motivations in place, it now makes sense to discuss how RocksDB is organized. RocksDB offers many features; however, this paper focuses on the simplified view in Figure 1.

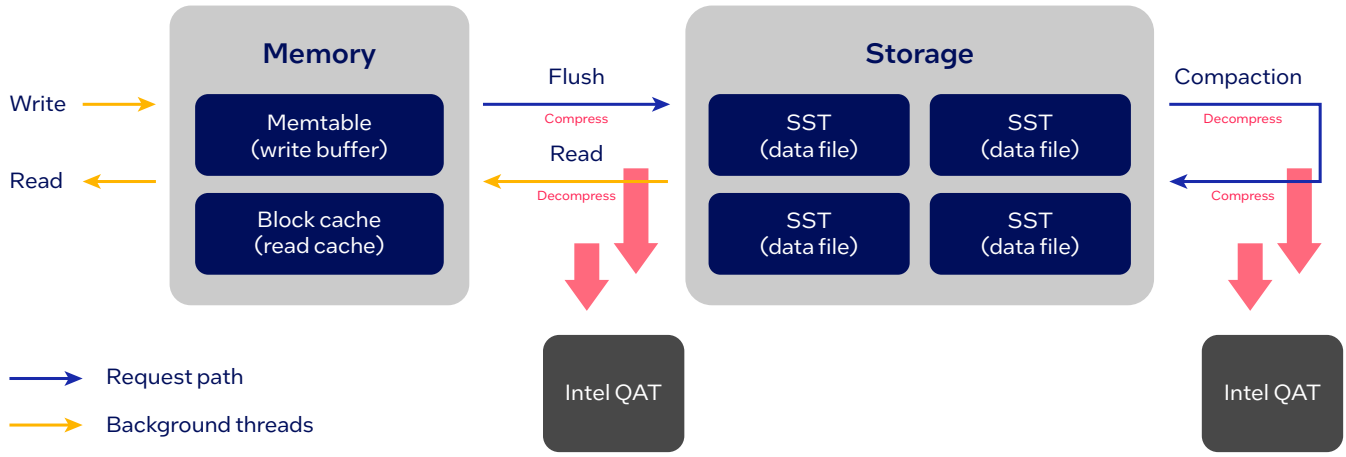


Figure 1. Simplified view of RocksDB architecture

The first step in analyzing the RocksDB architecture is differentiating data at rest from data in flight. In system memory, the memtable data structure handles writes until it becomes full, triggering a flush into a sorted string table (SST) file on storage.[21] Because the set of all SST files is stored as a log-structured merge (LSM) tree, this could trigger the compaction process to kick in.[22] Although several different forms of compaction (such as tiered and universal) are used, the process essentially involves merging file records at a certain level of the LSM tree and potentially pushing data down into the next (lower) level of the tree (for example, from L0 to L1). This action can cause a cascade of lower-level merge and push-down operations. Also, if the stored data is compressed, decompression and compression operations will be required on the data that needs to be moved around.

The process is simpler when a read request is initiated because no compactions are required. The desired portion of data is retrieved after decompression (if necessary) from the correct level in the LSM tree. If the block cache is enabled, the key-value pair(s) will be stored uncompressed for future read requests. In Figure 1, the data paths are distinguished from one another. The "request path" operations are run with high-priority user threads, whereas the "background" operations run with low-priority threads that are triggered when certain conditions are met.

Test system configuration

This section describes key hardware and software settings to explain Intel's performance test results.

Hardware setup

Table 2 provides relevant specifications for the system.

Table 2. Specifications of the test machine

Field	Value
CPU model	Intel Xeon Platinum 8488C processor
Sockets	2
Cores per socket	48
Intel® Hyper-Threading Technology (Intel® HT Technology)	Enabled
Logical cores	192
Intel® Turbo Boost Technology	Enabled
Base frequency	2.4 GHz
All-core maximum frequency	3.2 GHz
Maximum frequency	3.8 GHz
NUMA nodes	2
Accelerators (for 4th Gen Intel Xeon Scalable processors only)	Intel QAT: 8, Intel® Data Streaming Accelerator (Intel® DSA): 8, Intel® In-Memory Analytics Accelerator (Intel® IAA): 8, and Intel® Dynamic Load Balancer (Intel® DLB): 8
Installed memory	512 GB (16 x 32 GB DDR5 4,800 megatransfers per second [MT/s])
Network interface controller (NIC)	1 x Intel® Ethernet Controller I225-LM
Disk	1 x 1024M vmDisk-CD, 2 x 3.7 TB Intel® SSDPE2KX040T8, 1 x 223.6 GB Intel SSDSC2KB24
Microcode	0xab000350
Operating system (OS)	CentOS Stream 8
Thermal design power (TDP)	385 watts
Power and performance policy	Performance
Results collection date	Tested by Intel as of 11/17/2022

To avoid any side effects caused by cross-socket communication, all processes were pinned to the first socket (with only one NUMA node) during each test. Without this constraint, the workload would scale across both sockets, but only if the application is managed so that each process accesses resources only on the socket on which it is running. Using the single-socket approach, the number of logical cores is 96 (48 x 2) because hyper-threading is enabled, and all tests had access to four of eight Intel QAT accelerators. Intel DSA, Intel IAA, and Intel DLB accelerators were not used in test scenarios. Regarding system memory, 256 of the 512 GB was made available to the benchmark tool because of the one-socket limit. Finally, it is important to note that the power/performance policy in place targeted the maximum CPU frequency for the best possible throughput and latency values on the Intel Xeon Platinum 8488C processor.

Software setup

In its current form, RocksDB has built-in components that access compression libraries installed on the system. These include Zstd, LZ4, Zlib, and Snappy. Within each SST file, all blocks are compressible except for filter blocks used to query membership in the KV-store. Furthermore, different levels of the LSM tree can use different compression types.[23] Unfortunately, the existing implementation makes adding new compression components to the main RocksDB GitHub repository difficult because functionality is diffused across the code base. This problem also causes increased maintenance complexity for both new and existing compression types. To resolve this problem, Intel developed a new pluggable compression framework (see Figure 2).

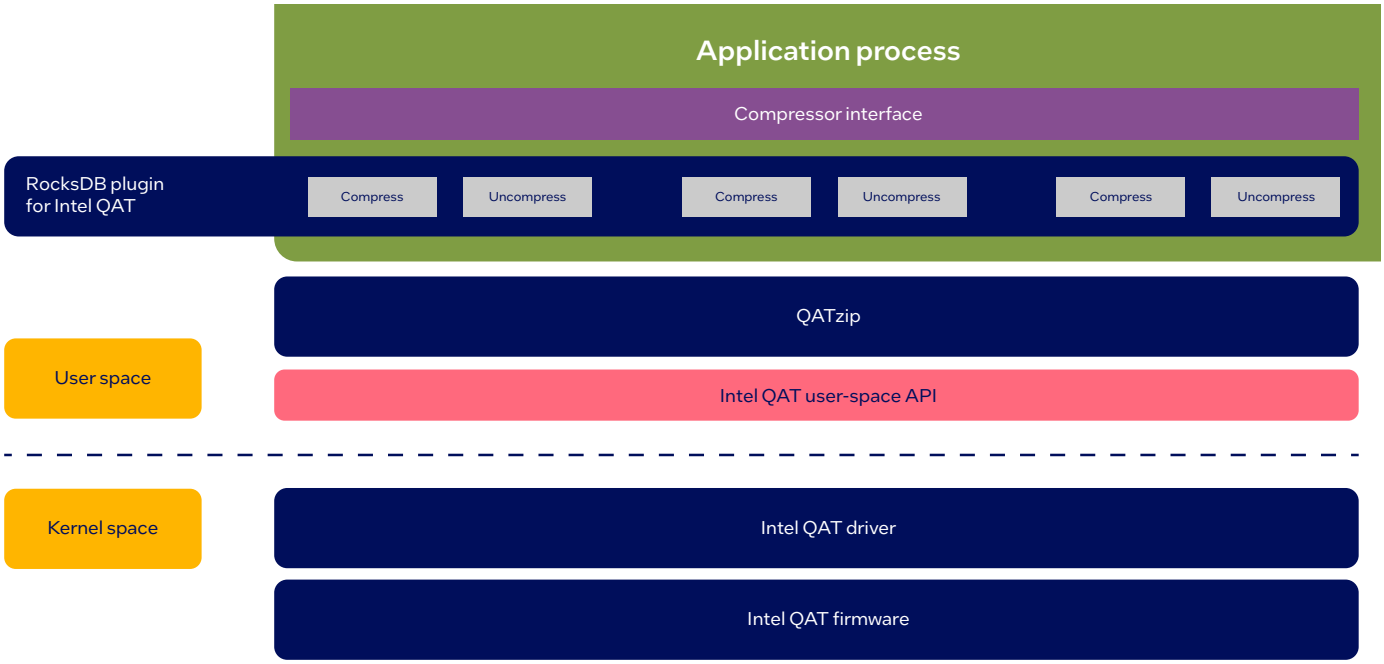


Figure 2. RocksDB pluggable compression framework and Intel QAT software stack

This new framework modularizes RocksDB compression functionality and creates a clean interface for new plugins to use. Existing components (Zstd, LZ4, and Snappy) remain inside the core GitHub repository, while new ones are hosted separately.[24] As compress/decompress calls are triggered by higher-level operations (see Figure 1), the underlying component libraries are accessed. In the case of each Intel QAT plugin object, these requests are passed into the QATzip library, where important pre-/post-processing is done before invoking driver and firmware functionality with the user-space APIs.

Because compression libraries like QATzip offer so many configuration options, the compressor interface allows plugins to define any custom parameters that might be required. For the Intel QAT plugin, this means exposing the formats (such as DEFLATE and LZ4), compression levels, and various runtime options provided by QATzip to higher-level RocksDB applications. The parameter values associated with each plugin object are independent from all others, so the risk of unwanted side effects is minimal. Furthermore, each application can compress/decompress data across multiple threads, so every plugin must properly manage data owned by each thread context.

Key components of the software stack are listed in Table 3. For Zstd, the default RocksDB configuration with compression level 3 is used. With Intel QAT, measurements are gathered with both level 1 (weakest) and level 9 (strongest). Note that Zstd compression levels are not related to Intel QAT compression levels. For example, Zstd L6 is not equivalent to Intel QAT L6.

Table 3. Configuration details for key software components

	Zstd	LZ4	Intel QAT
RocksDB origin	v7.9.0 with pluggable compression	v7.9.0 with pluggable compression	v7.9.0 with pluggable compression
Tools/compilers	GCC 8.5.0	GCC 8.5.0	GCC 8.5.0
Libraries	Zstd v1.5.2	LZ4 v1.8.3	QATzip v1.0.9
Drivers	Not applicable (N/A)	N/A	Intel QAT v20.L.0.9.6-00024

The pluggable compression framework is currently not integrated into the main RocksDB GitHub repository, which is why the foundational piece of the software stack was built from a custom repository. The LZ4 compression library was installed using the built-in package manager; in contrast, Zstd was compiled and installed from source because the desired 1.5.2 version does not exist by default as an installable package in CentOS Stream 8. In addition, all Intel QAT results are based on the DEFLATE compression algorithm. Finally, the QATzip library and prerelease Intel QAT driver were compiled and installed from the source to get all the needed features during testing. However, when the plugin is publicly available, it will rely on the in-tree versions of these Intel QAT components because they contain all the required features and provide a better setup experience for the user.[25]

Benchmarks

RocksDB contains a utility called `db_bench` that provides many benchmarks with various options. Initially, `db_bench` was inherited from LevelDB and enhanced to support additional parameters.[26] However, one change was needed to generate data from a corpus file instead of randomized strings.[27] With this modified version of `db_bench`, each resulting test has two distinct phases: setup and execution. The setup phase prefills corpus data into multiple KV-stores split evenly between both NVM Express (NVMe) drives, and the execution phase runs multiple instances of the read-only, read/write, or write-only benchmarks on each KV-store created during the setup phase. Command 1 contains the general form of `db_bench` prefill invocations used by every test.

```
numactl --physcpubind=<integer> --membind=<integer> /path/to/rocksdb/db_bench
--benchmarks=<string> --statistics --db=<string> --key_size=<integer> --value_size=<integer>
--value_src_data_type=<enum> --value_src_data_file=<string> --block_size=<integer>
--num=<integer> --bloom_bits=<integer> --compression_type=<string> --threads=<integer>
--disable_wal=<boolean> --cache_size=<integer> --use_cache_memkind_kmem_allocator=<boolean>
--cache_index_and_filter_blocks=<boolean> --cache_numshardbits=<integer> --compressed_cache_
size=<integer> --row_cache_size=<integer> --use_direct_reads=<boolean> --use_direct_io_for_
flush_and_compaction=<boolean> --compressor_options=<string> --max_background_jobs=<integer>
--subcompactions=<integer> --max_write_buffer_number=<integer> --min_write_buffer_number_to_
merge=<integer> --level0_file_num_compaction_trigger=<integer> --level0_slowdown_writes_
trigger=<integer> --level0_stop_writes_trigger=<integer> --max_bytes_for_level_base=<integer>
--stats_level=<integer> --stats_interval_seconds=<integer> --stats_per_interval=<integer>
```

Command 1. General form of db_bench sequential fill used to create a KV-store

Each `numactl` command pins its corresponding sequential fill process to a socket and NUMA node. This approach is applied to all execution-phase benchmarks described later in this section. After creating a KV-store, the sequential fill process generates key-value pairs where the keys are monotonically increasing integers. Some options are not used in every scenario; these can be identified by viewing the settings listed in Table 4. For further details on each option, consult the RocksDB wiki or source code.[26]

Table 4. S-templates of sequential fill fields used in all cases

Field	S-template
--physcpubind	{[(all)]}
--membind	{[(0)]}
--benchmarks	{[("fillseq, stats"), ("readrandom,stats")], [("fillseq,stats"), ("readrandomwriterandom,stats")], [("fillseq,stats"), ("overwrite,stats")]}
--db	{[(/path/to/data)]}
--key_size	{[(16)]}

Field	S-template
--value_size	{[(32 | 256)]}
--value_src_data_type	{[(file_direct)]}
--value_src_data_file	{[(/path/to/10hresho_corpus)]}
--block_size	{[(4096)]}, {[(16384)]}
--num	{[(550000000 | 100000000)], [(275000000 | 50000000)], [(7500000 | 5000000)]}
--bloom_bits	{[(10)]}
--compression_type	{[(zstd, lz4, com.intel.qat_compressor_rocksdb)]}
--threads	{[(1)]}
--disable_wal	{[(true)]}
--cache_size	{[(-1)]}
--use_cache_memkind_kmem_allocator	{[(false)]}
--cache_index_and_filter_blocks	{[(false)]}
--cache_numshardbits	{[(6)]}
--compressed_cache_size	{[(-1)]}
--row_cache_size	{[(0)]}
--use_direct_reads	{[(false)]}
--use_direct_io_for_flush_and_compaction	{[(false)]}
--compressor_options	{[(N/A, N/A, 10hresho_hdr=dynamic;data_fmt=deflate_gzip_ext;comp_lvl=1|9;comp_algorithm=deflate;sw_back-up=disable;input_sz_thrshold=1024;polling_mode=busy ;retry=true)]}
--max_background_jobs	{[(60), (N/A)], [(30), (10)], [(30), (20)]}
--subcompactions	{[(10), (N/A)], [(5)], [(5)]}
--max_write_buffer_number	{[(N/A)], [(20)], [(20)]}
--min_write_buffer_number_to_merge	{[(N/A)], [(1)], [(1)]}
--level0_file_num_compaction_trigger	{[(N/A)], [(10)], [(10)]}
--level0_slowdown_writes_trigger	{[(N/A)], [(60)], [(40)]}
--level0_stop_writes_trigger	{[(N/A)], [(120)], [(80)]}
--max_bytes_for_level_base	{[(N/A)], [(671088640)], [(671088640)]}
--stats_level	{[(4)]}
--stats_interval_seconds	{[(60)]}
--stats_per_interval	{[(1)]}

The three enclosure types used in Table 4 determine which parameters and scenarios apply to the units (elements) of each setting template (S-template):

- Parentheses cover different compression methods (such as Zstd, LZ4, and Intel QAT).
- Brackets cover different test phases.
- Braces cover different test types.

Vertical bars indicate that the left element maps to 32-byte values and the right element maps to 256-byte values. If an S-template is missing elements, that means the visible elements are reused for all other scenarios. For example, the {[(1 | 2), (3 | 4), (5 | 6)]} S-template means elements 1, 3, and 5 are used only with 32-byte values, while elements 2, 4, and 6 are used only with 256-byte values. All elements are reused with Zstd, LZ4, and Intel QAT across the setup and execution phases of the read-only, read/write, and write-only tests. When an element is labeled N/A, or not applicable (such as `--subcompactions`), that is because the corresponding field does not affect that specific execution phase and test type. Additionally, an S-template might contain a placeholder (such as `--db`) that is user configurable and machine filesystem dependent. These placeholders are highlighted in bold for easy reference. Finally, please note that all other parts of this section also use this notation.

As mentioned earlier, the KV-stores created during the setup phase are operated on during the execution phase. The number of KV-stores created matches the number of db_bench processes launched, which varies between test types. Furthermore, the location of these KV-stores is determined by the order of process creation. For example, if the first process's KV-store maps to /mnt/nvme0/rockstest0, then the second one will map to /mnt/nvme1/rockstest1, and so on. With this approach, the data is almost equally split between the two NVMe drives if there are an even number of processes. Also, the values generated for every KV-store originate from a binary formed by concatenating all files in the standard Calgary Corpus. This corpus is widely used for evaluating lossless data-compression techniques, so it makes sense to use it for the benchmarks in this paper.[28]

Before presenting the benchmarks, it is important to explain how Intel QAT accelerators are configured. In its default mode, the out-of-tree Intel QAT driver has eight configuration files installed into the /etc folder. Each configuration file maps to exactly one of the eight physical Intel QAT accelerators on the system on chip (SoC), and each socket is associated with four Intel QAT accelerators. The settings inside a configuration file can be modified to adjust how the file's corresponding Intel QAT accelerator behaves. Some of these adjustments include enabling/disabling services, specifying data-compression instances, and limiting/relaxing restrictions on which devices processes can access. After modifying the relevant configuration files, the Intel QAT driver service is reloaded so that the new settings take effect. These changes are described in Table 5.

Table 5. Intel QAT configuration file changes

Section	Setting(s)
[General]	ServicesEnabled = dc
[SSL]	NumberCyInstances = 0 NumberDcInstances = 0 NumProcesses = 0

[SHIM]	NumberCyInstances = 0
	NumberDcInstances = X
	NumProcesses = Y
	LimitDevAccess = 0
	
	#Data Compression – User instance #0
	Dc0Name = "Dc0"
	Dc0IsPolled = 1
	
	...

Note: The ellipsis in this table indicates that the other 63 Dc logical instances follow a similar format to those shown.

Because all tests only rely on Intel QAT compression acceleration, the cryptographic services are disabled. This change (ServicesEnabled = dc) is highlighted in the first row of Table 5. Next, there are no process and logical instance allocations in the Secure Sockets Layer [SSL] section to maximize the number available to the QATzip library. This is where the added [SHIM] section comes into play. In this area, only compression/decompression services are active, and the product of logical instances (X) and processes (Y) equals the maximum number of logical instances (64) in an Intel QAT accelerator powered by 4th Gen Intel Xeon Scalable processors. For QATzip to load-balance across the four Intel QAT accelerators, the process affinity option is disabled (LimitDevAccess = 0). This means a process can use available logical instances from all Intel QAT devices. Finally, each of the 64 logical instances in the [SHIM] section is clearly labeled (for example, Dc0Name = "Dc0") and specified, so the driver can provide access to applications enabled by QATzip.

Read-only

In the execution phase of this test, multiple db_bench processes perform Get operations on the KV-stores created during the setup phase. Every Get operation's key is a randomly generated integer that is less than the total number of entries in the corresponding KV-store. Command 2 contains the general form of db_bench invocations used in every read-only benchmark.

```
numactl --physcpubind=<integer> --membind=<integer> /path/to/rocksdb/db_bench
--benchmarks=<string> --statistics --db=<string> --use_existing_db --key_size=<integer>
--value_size=<integer> --value_src_data_type=<enum> --value_src_data_file=<string> --block_
size=<integer> --num=<integer> --bloom_bits=<integer> --compression_type=<string>
--duration=<integer> --reads=<integer> --threads=<integer> --disable_wal=<boolean> --cache_
size=<integer> --use_cache_memkind_kmem_allocator=<boolean> --cache_index_and_filter_
blocks=<boolean> --cache_numshardbits=<integer> --compressed_cache_size=<integer> --row_cache_
size=<integer> --use_direct_reads=<boolean> --use_direct_io_for_flush_and_compaction=<boolean>
--compressor_options=<string> --stats_level=<integer> --stats_interval_seconds=<integer>
--stats_per_interval=<integer>
```

Command 2. General form of db_bench random read used to query a KV-store

Much like sequential fill, each `numactl` command pins its corresponding random read process to a socket and NUMA node. Also, many of the options presented in Command 1 are reused with the settings described in Table 4. Table 6 provides a list of options (apart from `--use_existing_db`) specific to the execution phase of the testing.

Table 6. S-templates of random read fields

Field	S-template
--duration	500
--reads	6250000
--threads	16

The S-templates in Table 6 are much simpler than those of sequential fill because they only apply to the execution phase in read-only tests. If all data were uncompressed, it would be about 100 GB in size. Overall, each read-only benchmark runs for 500 seconds on a total of 64 threads that break down into 16 threads per `db_bench` process. This means each configuration file has `NumProcesses` set to 4 and `NumberDcInstances` set to 16. The `--reads` field also specifies the maximum number of operations per thread in case `--duration` is either zero or not provided. Generally, throughput (measured in operations per second) increases as the number of threads increases in this benchmark. However, this is not the case for the write-only prefill operations in the setup phase because the probability of contention between write operations is high. Fortunately, this does not matter in practice because performance is only relevant while running in the execution phase.

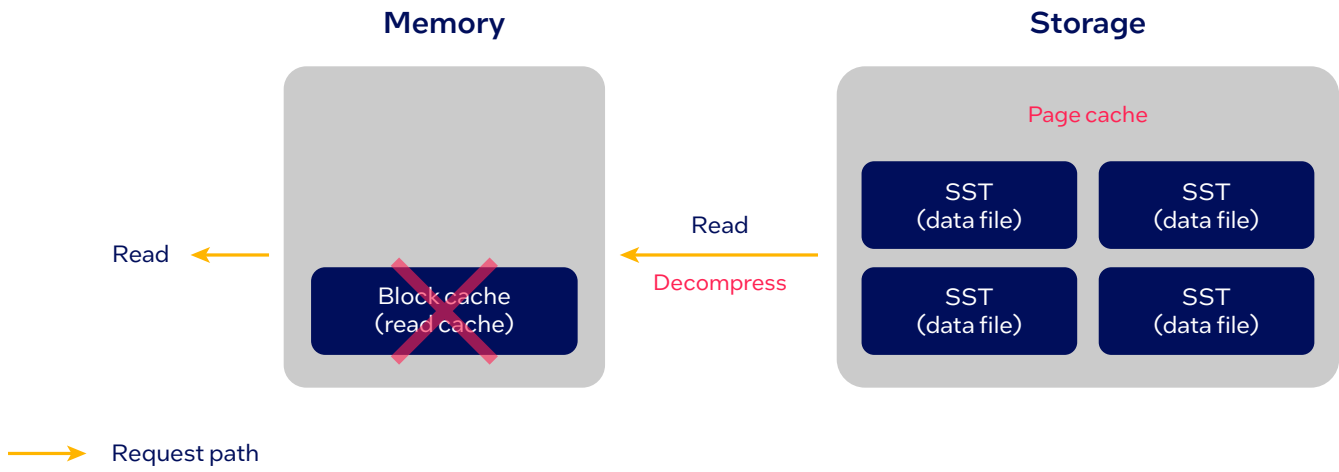


Figure 3. Simplified view of a read-only data flow

Because the read-only benchmark does not have any write operations, there are no background operations, such as flush and compaction. This means data blocks in the LSM tree only deal with decompression as the compute-heavy task when reads are triggered on the request path that is given the highest priority in RocksDB. Moreover, compressed data is stored in the operating system (OS) page cache to avoid the limitations caused by disk retrievals, and the block cache is disabled to ensure decompression always occurs during read requests. All of this is illustrated in Figure 3, which is a modified form of the architecture shown in Figure 1.

In addition to the data-flow model, the benchmark measurements are presented in Figure 4 and Table 7. These measurements include throughput, P99 latency, data size, and CPU utilization measurements for all read-only scenarios, in addition to combinations of 4 KB and 16 KB block sizes with 32-byte and 256-byte value sizes. In each 2D plot in Figure 4, the relative throughput (y-axis) is mapped to the relative data size (x-axis) for each compression method. Additionally, each plot point has a relative P99 latency number in parentheses. All measurements designate Zstd L3 as the baseline because of its desirable speed and data size combination. For example, Intel QAT L1 in the 4 KB/32-byte scenario is 0.02 units to the right and 0.11 units above Zstd L3, with a relative P99 latency of 0.86, which means it has 11 percent greater throughput, 14 percent lower P99 latency, and 2 percent larger data size.

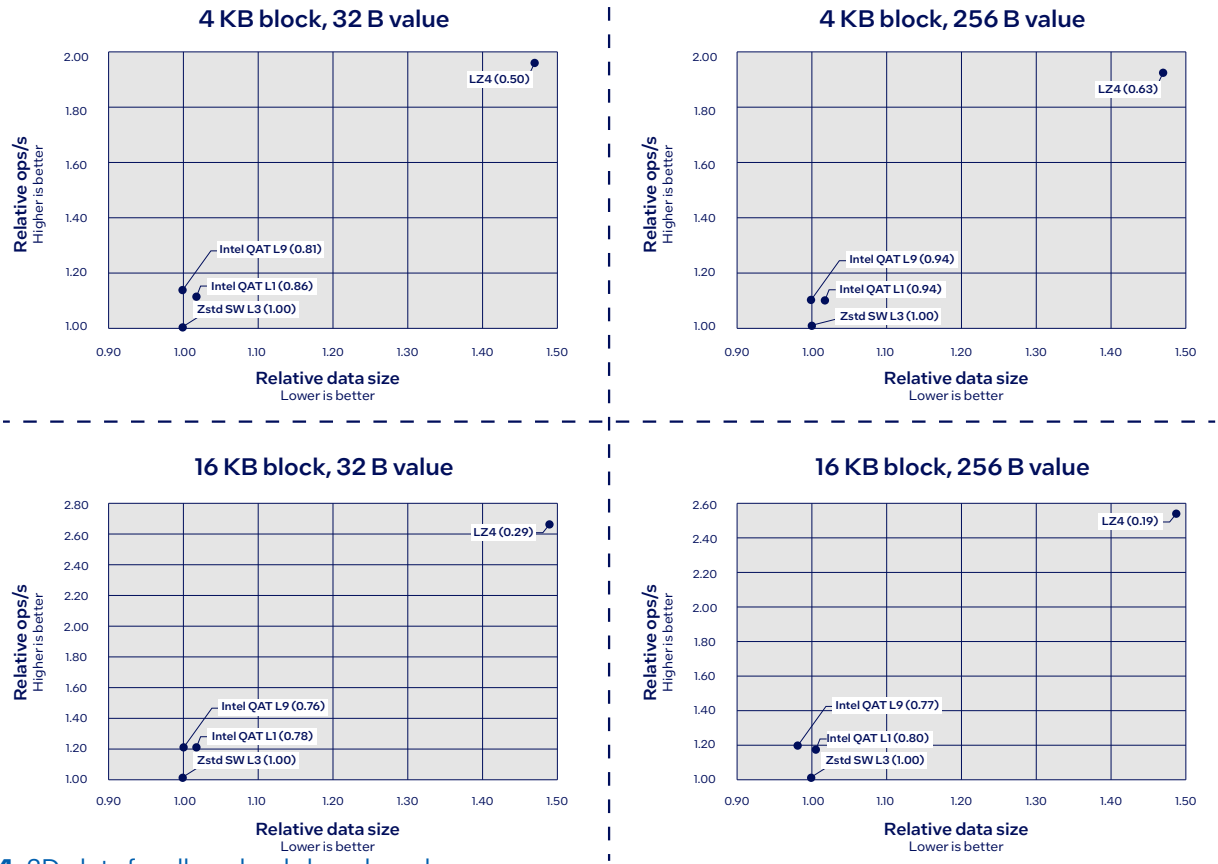


Figure 4. 2D plots for all read-only benchmarks

Table 7. Data for all read-only benchmarks

(Block size, value size)	Metric	Units	Intel QAT L1 vs. Zstd	Intel QAT L9 vs. Zstd
(4 KB, 32 B)	Throughput	ops/s	+11%	+12%
	Latency (P99)	µs	–17%	–22%
	Data size	GB	+2%	0%
(4 KB, 256 B)	Throughput	ops/s	+8%	+8%
	Latency (P99)	µs	–6%	–6%
	Data size	GB	+1%	–1%
(16 KB, 32 B)	Throughput	ops/s	+21%	+23%
	Latency (P99)	µs	–22%	–24%
	Data size	GB	+2%	0%
(16 KB, 256 B)	Throughput	ops/s	+16%	+18%
	Latency (P99)	µs	–22%	–23%
	Data size	GB	+1%	–2%

In all cases, CPU utilization is about two-thirds of a socket. While there is room to add more threads to this test, Intel QAT performance would improve marginally because all devices are mostly saturated under these conditions. In contrast, LZ4 has the highest throughput (+164 percent in 16 KB/32 B) and the lowest P99 latency (–81 percent in 16 KB/256 B) relative to Zstd, but it also has the worst data sizes (+47 percent to +49 percent) relative to Zstd. This is expected, however, because LZ4 is designed for excellent decompression performance at the cost of compression ratios. On the other hand, performance for Intel QAT and Zstd is similiar, where Intel QAT L1 has consistently larger data sizes (+1 percent to +2 percent) and Intel QAT L9 is at or below the baseline (–2 percent to 0 percent). Intel QAT L1 and Intel QAT L9 similarly outperform Zstd because there is no compression in these read-only benchmarks. However, it is important to note how Intel QAT throughput and latency improvements become more distinct when moving from 4 KB to 16 KB block sizes. This is because raw compute is more important than overhead tasks like data movement when larger block sizes are used. In contrast, Zstd performance improves relative to Intel QAT when moving from 32-byte to 256-byte values because of how much more efficient the encoding is compared to DEFLATE. Also, as block size increases (4 KB to 16 KB), data size decreases in all scenarios because of the additional string matches found, with Intel QAT L9 benefitting slightly more than other methods.

Read/write

In the execution phase of the test, multiple db_bench processes perform Get (80 percent) and Put (20 percent) operations on the KV-stores created during the setup phase. Every Get and Put operation's key is a randomly generated integer that is less than the total number of entries in the corresponding KV-store. Command 3 contains the general form of db_bench invocations used in every read/write benchmark.

```
numactl --physcpubind=<integer> --membind=<integer> /path/to/rocksdb/db_bench
--benchmarks=<string> --statistics --db=<string> --use_existing_db --key_size=<integer>
--value_size=<integer> --value_src_data_type=<enum> --value_src_data_file=<string> --block_
size=<integer> --num=<integer> --bloom_bits=<integer> --compression_type=<string>
--duration=<integer> --reads=<integer> --threads=<integer> --disable_wal=<boolean> --cache_
size=<integer> --use_cache_memkind_kmem_allocator=<boolean> --cache_index_and_filter_
blocks=<boolean> --cache_numshardbits=<integer> --compressed_cache_size=<integer> --row_cache_
size=<integer> --use_direct_reads=<boolean> --use_direct_io_for_flush_and_compaction=<boolean>
--compressor_options=<string> --max_background_jobs=<integer> --subcompactions=<integer>
--readwritepercent=<integer> --max_write_buffer_number=<integer>--min_write_buffer_number_to_
merge=<integer> --level0_file_num_compaction_trigger=<integer> --level0_slowdown_writes_
trigger=<integer> --level0_stop_writes_trigger=<integer> --max_bytes_for_level_base=<integer>
--stats_level=<integer> --stats_interval_seconds=<integer> --stats_per_interval=<integer>
```

Command 3. General form of db_bench random read/write used to operate on a KV-store

Much like sequential fill, each numactl command pins its corresponding random read/write process to a socket and NUMA node. Also, many of the options presented in Command 1 are reused with settings described in Table 4. Table 8 provides a list of options (apart from --use_existing_db) specific to the execution phase of this test.

Table 8. S-templates of random read/write fields

Field	S-template
--duration	500
--reads	12500000
--threads	8
--readwritepercent	80

Like the read-only S-templates, the read/write S-templates are much simpler than those of sequential fill because they apply only to the execution phase in read/write tests. Overall, each read/write benchmark runs for 500 seconds on 64 threads that break down into 8 threads per db_bench process. This means each configuration file has NumProcesses set to 8 and NumberDcInstances set to 8. Note that this does not include the background threads that spawn periodically when compaction occurs. Again, if all data is uncompressed, the total size would be about 100 GB, with the --reads and --duration fields functioning the same way as those in the read-only benchmarks. The main differentiator is the --readwritepercent field, which decides which portion of entries is routed to read or write requests.

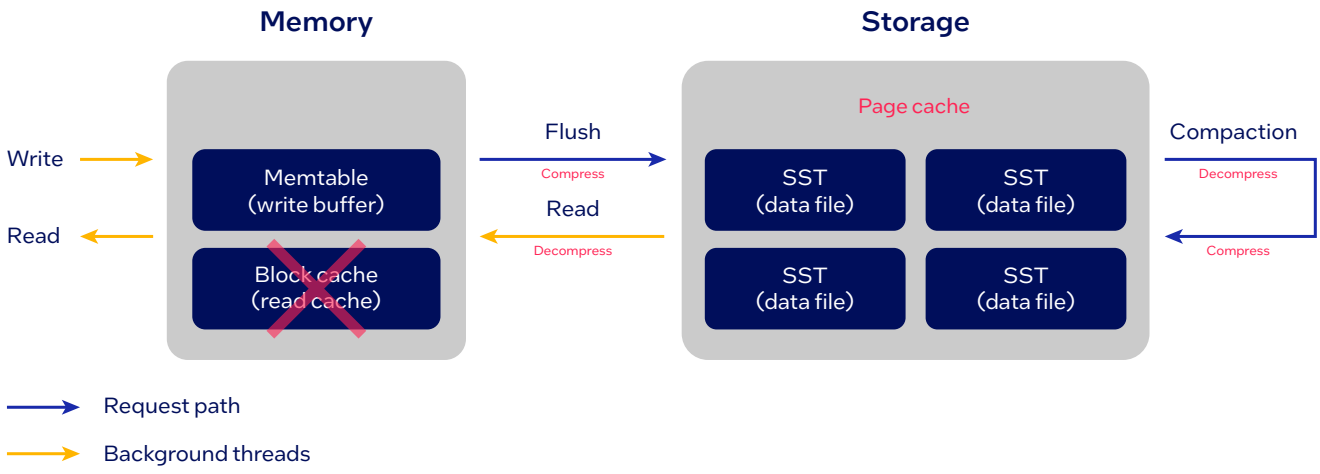


Figure 5. Simplified view of read/write data flow

Unlike the read-only benchmark, the read/write one involves flush and compaction operations in the background because periodic writes trigger updates to the SST files. This means data blocks in the LSM tree deal with both compression and decompression as compute-heavy tasks when reads or writes are triggered on the request path that is given the highest priority in RocksDB. However, the approach of including the OS page cache and disabling the block cache is identical to that of the read-only benchmark because the underlying constraints remain the same. All of this is illustrated in Figure 5, which is a modified form of the architecture shown in Figure 1. Also, the benchmark measurements are presented in Table 9 and Figure 6. To understand the format of the 2D plots in Figure 6, refer to the earlier explanation in the read-only section. Finally, while most data blocks are decompressed, the proportion depends on the block/value size combination and varies between ~90 percent and ~99 percent.

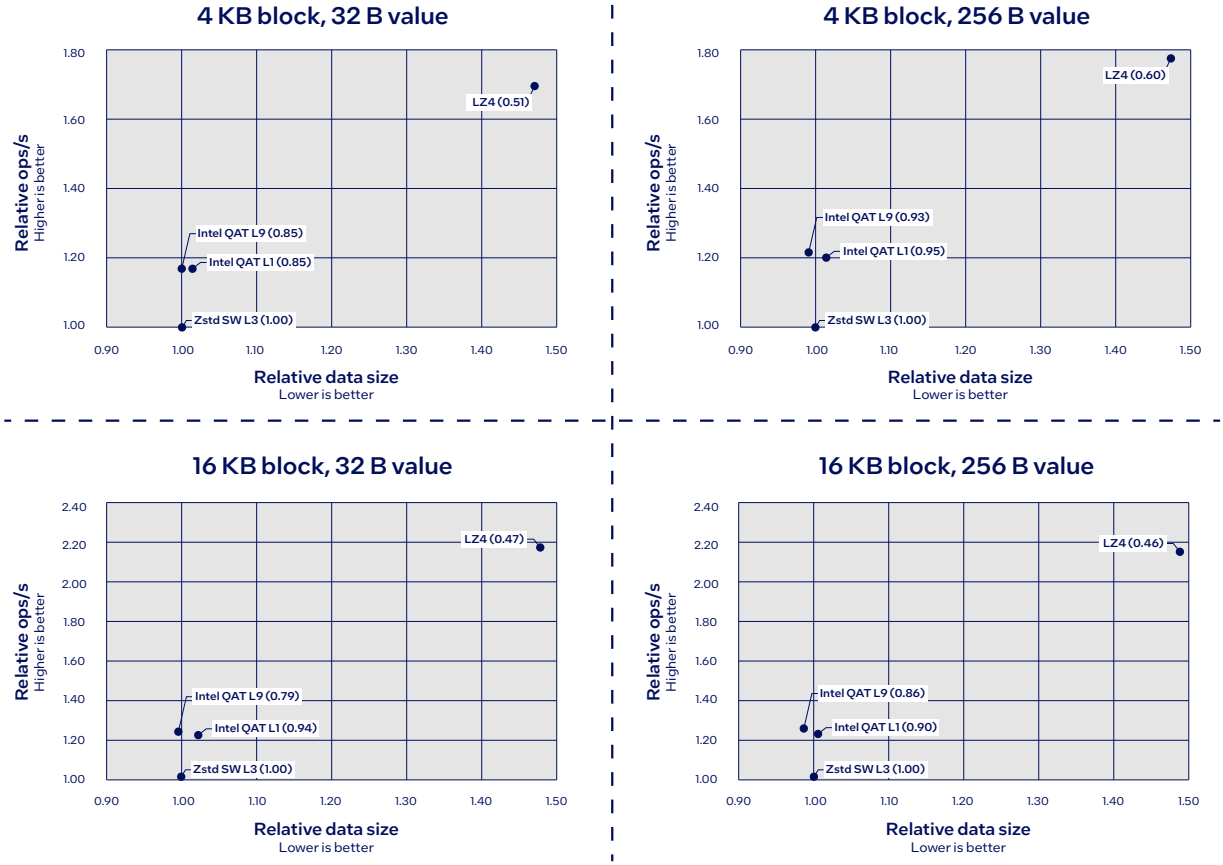


Figure 6. 2D plots for the read/write benchmarks

Table 9. Data for the read/write benchmarks

(Block size, value size)	Metric	Units	Intel QAT L1 vs. Zstd	Intel QAT L9 vs. Zstd
(4 KB, 32 B)	Throughput	ops/s	+14%	+14%
	Latency (P99)	µs	–15%	–15%
	Data size	GB	+2%	0%
(4 KB, 256 B)	Throughput	ops/s	+20%	+22%
	Latency (P99)	µs	–5%	–7%
	Data size	GB	+1%	–1%
(16 KB, 32 B)	Throughput	ops/s	+24%	+27%
	Latency (P99)	µs	–6%	–21%
	Data size	GB	+2%	0%
(16 KB, 256 B)	Throughput	ops/s	+24%	+27%
	Latency (P99)	µs	–10%	–14%
	Data size	GB	+1%	–2%

In this read/write test, CPU utilization varies between compression methods and block/value sizes. This is because of flush and compaction operations that occur at irregular time intervals with different impacts. When moving from read-only to read/write, Zstd and LZ4 see larger utilization increases relative to Intel QAT because of how expensive compression is on the CPU cores. This trend holds regardless of which block/value-size combination is used. However, the story changes when only considering read/write results. Under these constraints, utilization decreases with the 4 KB to 16 KB block-size transition due to a lower percentage of CPU front-end pipeline stalls for Zstd/LZ4 and more compression on Intel QAT hardware. The opposite occurs when changing 32-byte to 256-byte values because of the extra cost required to find and compress longer repetitions.

Still, decompression vastly outweighs compression, so relative performance positions are like those of the read-only benchmarks. LZ4 has the highest throughput (+117 percent in 16 KB/32 B) and lowest P99 latency (–54 percent in 16 KB/256 B) relative to Zstd, but it also has the worst data sizes (+47 percent to +49 percent) relative to Zstd. In addition, Intel QAT and Zstd are still close together, where Intel QAT L1 has consistently larger data sizes (+1 percent to +2 percent), and Intel QAT L9 remains at or below the baseline (–2 percent to 0 percent). Because data size is measured in the setup phase, the extra compression in the execution phase has no effect there. However, it does amplify the Intel QAT throughput/latency advantage over Zstd found in the read-only benchmarks when increasing the block size (4 KB to 16 KB). Furthermore, the Zstd performance improvement relative to Intel QAT when moving from 32-byte to 256-byte values is nullified by the heavier load compression adds.

Write-only

In the execution phase of the test, multiple `db_bench` processes perform `Put` operations on the KV-stores created during the setup phase. Every `Put` operation's key is a randomly generated integer that is less than the total number of entries in the corresponding KV-store. Command 4 contains the general form of `db_bench` invocations used in every write-only benchmark.

```
numactl --physcpubind=<integer> --membind=<integer> /path/to/rocksdb/db_bench
--benchmarks=<string> --statistics --db=<string> --use_existing_db --key_size=<integer>
--value_size=<integer> --value_src_data_type=<enum> --value_src_data_file=<string> --block_
size=<integer> --num=<integer> --bloom_bits=<integer> --compression_type=<string>
--duration=<integer> --writes=<integer> --threads=<integer> --disable_wal=<boolean> --cache_
size=<integer> --use_cache_memkind_kmem_allocator=<boolean> --cache_index_and_filter_
blocks=<boolean> --cache_numshardbits=<integer> --compressed_cache_size=<integer> --row_cache_
size=<integer> --use_direct_reads=<boolean> --use_direct_io_for_flush_and_compaction=<boolean>
--compressor_options=<string> --max_background_jobs=<integer> --subcompactions=<integer>
--readwritepercent=<integer> --max_write_buffer_number=<integer>--min_write_buffer_number_to_
merge=<integer> --level0_file_num_compaction_trigger=<integer> --level0_slowdown_writes_
trigger=<integer> --level0_stop_writes_trigger=<integer> --max_bytes_for_level_base=<integer>
--min_level_to_compress=<integer> --stats_level=<integer> --stats_interval_seconds=<integer>
--stats_per_interval=<integer>
```

Command 4. General form of db_bench random write used to operate on a KV-store

Much like sequential fill, each `numactl` command pins its corresponding random write process to a socket and NUMA node. Also, many of the options presented in Command 1 are reused with settings described in Table 4. Table 10 provides a list of options (apart from `--use_existing_db`) specific to the execution phase of the test.

Table 10. S-templates of random read/write fields

Field	S-template
--duration	500
--writes	10000000
--threads	1
--min_level_to_compress	N/A | 1

Overall, each write-only benchmark runs for 500 seconds on a variable number of threads that break down into a single thread per `db_bench` process to avoid contention between threads. The number of processes is selected to reach approximately 90 percent CPU socket utilization. This means each configuration file has `NumProcesses` rounded up to the power of two closest to the number of processes and `NumberDcInstances` set so that the product is 64. Note that this does not include the background threads that spawn periodically when compaction occurs. In the case of uncompressed data, the total size varies between approximately 17 GB and 52 GB, with `--writes` and `--duration` fields analogous to those in the read-only and read/write benchmarks. Additionally, the `--min_level_to_compress` field keeps level 0 of the LSM tree uncompressed to avoid write stalls when 256-byte values are in effect.

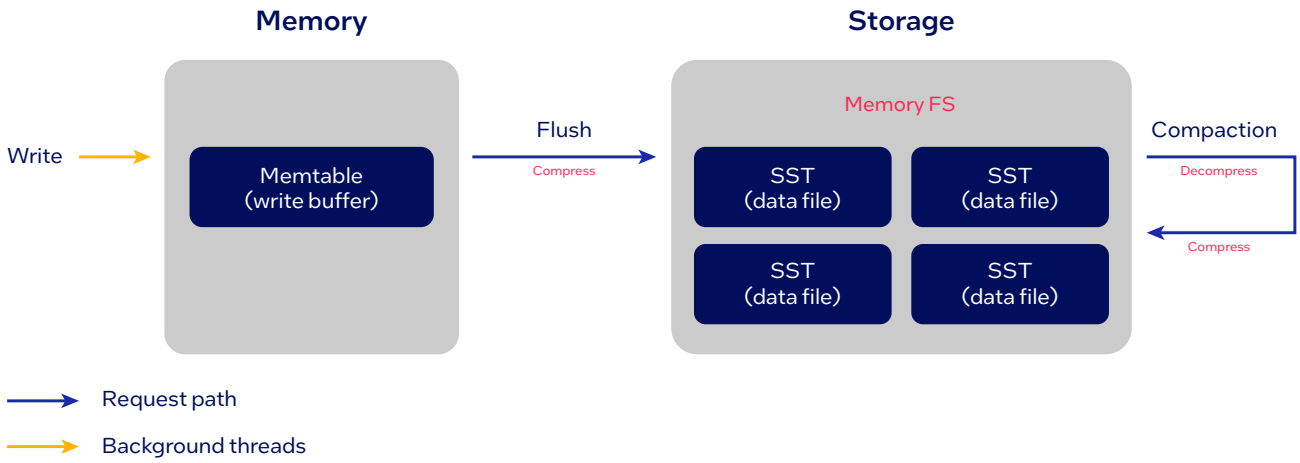


Figure 7. Simplified view of write-only data flow

Much like the read/write benchmark, this part of the test involves flush and compaction operations in the background because periodic writes trigger updates to the SST files. Unlike the read-only and read/write benchmarks, though, there are no reads, so decompression is never initiated by the application. This means data blocks in the LSM tree deal with both compression and decompression as compute-heavy tasks when writes are triggered on the request path that is given the highest priority in RocksDB. In this situation, however, the block cache is disabled, and a memory filesystem (ramfs) is used to avoid the overhead caused by OS page cache flushes. All of this is illustrated in Figure 7, which is a modified form of the architecture shown in Figure 1. Also, the benchmark measurements are presented in Figure 8 and Table 11. The format of the 2D plots is like those from the read-only benchmarks, but the parentheses contain the number of db_bench processes instead of relative P99 latency measurements. Finally, the proportion of compressed versus decompressed data blocks is about 50/50, regardless of the block/value size combination.

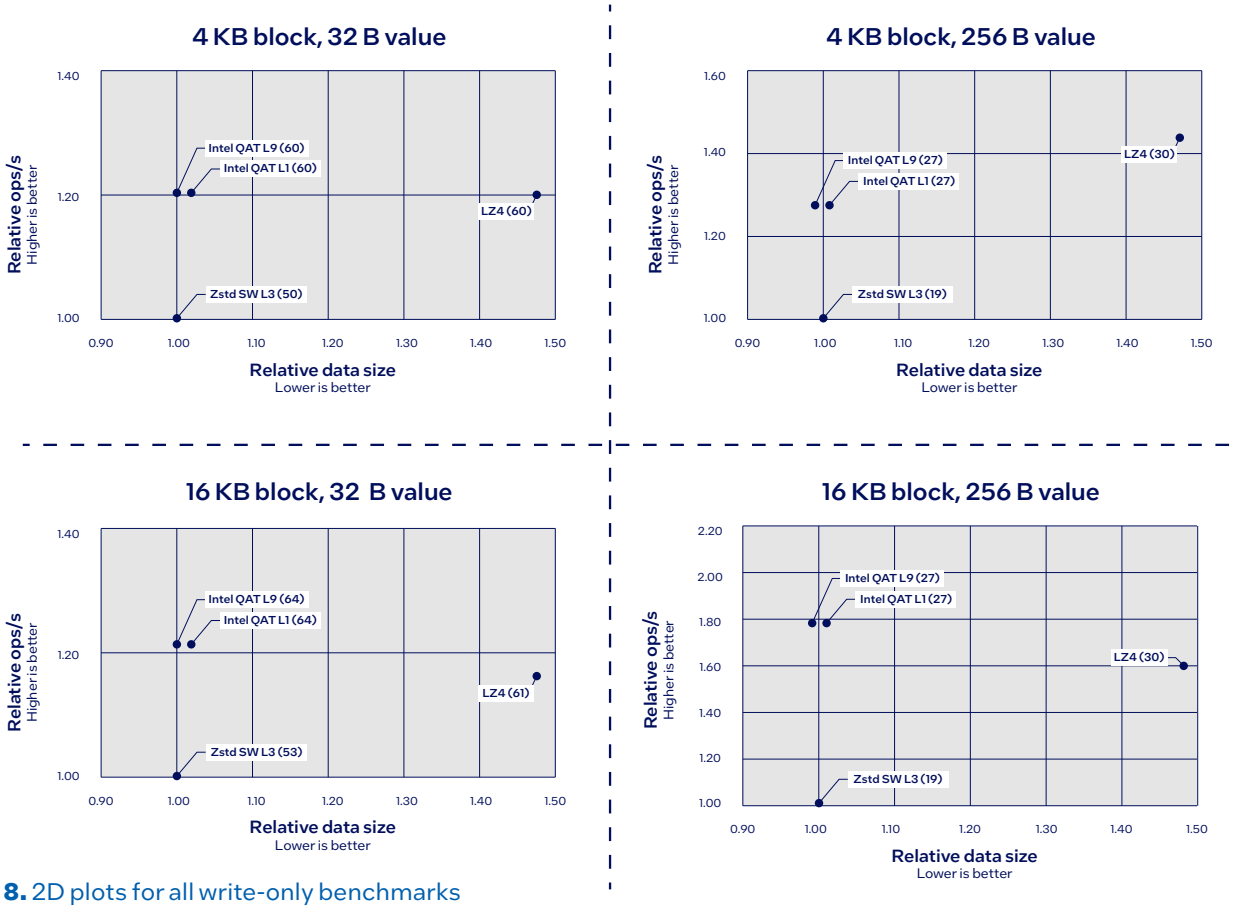


Figure 8. 2D plots for all write-only benchmarks

Table 11. Data for all write-only benchmarks

(Block size, value size)	Metric	Units	Intel QAT L1 vs. Zstd	Intel QAT L9 vs. Zstd
(4 KB, 32 B)	Throughput	ops/s	+20%	+20%
	Data size/ instance	MB	+2%	0%
(4 KB, 256 B)	Throughput	ops/s	+28%	+28%
	Data size/ instance	MB	+1%	–1%
(16 KB, 32 B)	Throughput	ops/s	+22%	+22%
	Data size/ instance	MB	+2%	0%
(16 KB, 256 B)	Throughput	ops/s	+77%	+76%
	Data size/ instance	MB	+1%	–2%

Although the CPU utilization does not exactly match the 90 percent target in each run, the results are not meaningfully affected. The fewest number of processes are always launched with Zstd, whereas LZ4 is best with 4 KB block sizes, and Intel QAT has the advantage when 16 KB block sizes are used. These differences are magnified with 256-byte values because there is so much more data handled by each process.

Because compression and decompression are roughly the same, this test's performance characteristics differ from the read-only and read/write cases. With 4 KB block sizes, LZ4 outperforms Zstd and Intel QAT, but it always has the worst data sizes. In contrast, Intel QAT has the best throughput in the 16 KB block-size scenarios while maintaining data sizes comparable to Zstd. Furthermore, the most intense compression activity occurs with the 16 KB block-size and 256-byte value-size scenario because of the sheer amount of data processed. This is where Intel QAT sees its largest throughput advantage by using previously underutilized compression bandwidth.

Conclusion

The RocksDB embedded storage engine has multiple built-in software compression methods. Unfortunately, they are not modular, so code changes across many source files are required to add a new method. To solve this problem, Intel introduced a new pluggable compression framework that allows anyone to introduce new types of compression without needing to modify core RocksDB GitHub functionality. One such type is Intel QAT, which provides hardware-accelerated compression and decompression functionality built into 4th Gen Intel Xeon Scalable processors. With a RocksDB plugin developed for the pluggable compression framework that uses Intel QAT, significant performance uplifts can be achieved compared to Zstd software compression. These improvements include increased throughput (ops/s) and reduced P99 latency (µs) while maintaining similar compressed data sizes in read-only, read/write, and write-only microbenchmarks that act as proxies for full-scale applications.

Start using RocksDB with Intel QAT today.

Visit the Intel® Developer Cloud **and learn more about** Intel QuickAssist Technology (Intel QAT)**.**



[1] Z. Cao, S. Dong, S. Vemuri, and D.H.C. Du. "Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook." *Proc. 18th USENIX Conf. File and Storage Technologies (FAST'20).* February 2020.

[2] A. Petrov. "Introduction and Overview." *Database Internals*. 2019. https://learning.oreilly.com/library/view/database-internals/9781492040330/ch01.html.

[3] M. Athanassoulis and S. Idreos. "Design Tradeoffs of Data Access Methods." *Proc. 2016 Int. Conf. Manage. of Data.* 2016.

[4] M. Athanassoulis, M.S. Kester, L.M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. "Designing access methods: The RUM conjecture." *Proc. Int. Conf. Extending Database Technology (EDBT).* 2016.

[5] RocksDB. "A persistent key-value store for fast storage environments." Accessed September 2023. https://rocksdb.org.

[6] S. Dong, A. Kryczka, Y. Jin, and M. Stumm. "Evolution of Development Priorities in Key-value stores Serving Large-scale Applications: The RocksDB Experience." *Proc. 19th USENIX Conf. File and Storage Technologies (FAST'21).* February 2021.

[7] C. Győrödi, R. Győrödi, G. Pecherle, and A. Olah. "A Comparative Study: MongoDB vs. MySQL." *Int. Conf. Eng. of Modern Elect. Syst. (EMES).* June 2015.

[8] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, and X. Tang. "TiDB: A Raft-based HTAP Database." *Proc. VLDB Endowment.* 2020.

[9] This paper replaces "Facebook" or "Facebook Inc." references in older publications with the current corporate entity name "Meta." Any mention of "Facebook" applies only to the application rather than the company.

[10] Y. Matsunobu, S. Dong, and H. Lee. "MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph." *Proc. VLDB Endowment.* 2020.

[11] Apache. "Using RocksDB State Backend in Apache Flink: When and How." January 2021. https://flink.apache.org/2021/01/18/rocksdb.html.

[12] Confluent. "Introducing Kafka Streams: Stream Processing Made Simple." March 2016. confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/.

[13] S.A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R.H. Campbell. "Samza: Stateful Stream Processing at LinkedIn." *Proc. VLDB Endowment.* 2017.

[14] G.J. Chen, J.L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz. "Realtime Data Processing at Facebook." *Proc. 2016 Int. Conf. Manage. of Data.* June 2016.

[15] Engineering at Meta. "Log Device: a distributed data store for logs." Accessed September 2023. https://engineering.fb.com/2017/08/31/core-data/logdevice-a-distributed-data-store-for-logs/.

[16] Engineering at Meta. "Dragon: distributed graph query engine." March 2016. https://engineering.fb.com/2016/03/18/data-infrastructure/dragon-a-distributed-graph-query-engine/.

[17] Rockset. "How we use RocksDB at Rockset." June 2019. https://rockset.com/blog/how-we-use-rocksdb-at-rockset/.

[18] GitHub. "Pika." OpenAtomFoundation/pika. Accessed September 2023. https://github.com/OpenAtomFoundation/pika.

[19] K. Ouaknine, O. Agra, and Z. Guz. "Optimization of RocksDB for Redis on Flash." *Int. Conf. Compute and Data Analysis.* May 2017.

[20] GitHub. "RocksDB Overview." Facebook/RocksDB wiki. Accessed September 2023. https://github.com/facebook/rocksdb/wiki/RocksDB-Overview.

[21] While the SST file is usually on flash storage media (such as an SSD), it is possible to house this portion of the data elsewhere.

[22] The LSM data structure is optimized for the write/update portion of the RUM space.

[23] GitHub. "Compression." Facebook/RocksDB wiki. Accessed September 2023. https://github.com/facebook/rocksdb/wiki/Compression.

[24] GitHub. "Add support for compressor plugins." Accessed September 2023. https://github.com/facebook/rocksdb/pull/6717.

[25] GitHub. "Intel® QuickAssist Technology Plugin for RocksDB Storage Engine." Accessed September 2023. https://github.com/intel/qat-plugin-rocksdb.

[26] GitHub. "Benchmarking tools." Facebook/RocksDB wiki. Accessed September 2023. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[27] GitHub. "Add file based data generation options to db_bench." Accessed September 2023. https://github.com/facebook/rocksdb/pull/10395.

[28] Data-Compression.info. "Calgary Corpus." Accessed September 2023. data-compression.info/Corpora/CalgaryCorpus/.

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

Printed in USA 1223/KF/PRW/PDF Please Recycle 358090-001US