IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 12 October 2023, accepted 29 October 2023, date of publication 2 November 2023, date of current version 21 November 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3329474

RESEARCH ARTICLE

Analyzing I/O Characteristics of Time-Series Data Using High Performance Storage Devices

SANGMYUNG LEE[1], YONGSEOK SON[2], AND SUNGGON KIM[1]
[1]Department of Computer Science and Engineering, Seoul National University of Science and Technology, Gongneung-ro, Nowon-gu, Seoul 01811, Republic of Korea
[2]Department of Computer Science and Engineering, Chung-Ang University, Seoul 06974, Republic of Korea

Corresponding author: Sunggon Kim (sunggonkim@seoultech.ac.kr)

This work was supported by the Seoul National University of Science and Technology.

ABSTRACT As the importance of data increases, data is continuously collected from diverse sources such as sensors, IoT devices, and edge computing devices. To manage these continuously monitored data, it is often organized chronologically with time which is referred as time-series data. By managing the data using time, data from different streams can be analyzed in a comprehensive manner with an identical index which is time. However, due to the unique characteristics of time-series data, it is essential for the underlying database systems to understand the characteristics of the time-series data. To handle this, time-series database systems, which specially target time-series data, are emerging. These database systems have different performance characteristics due to the unique characteristics of the data which should be investigated to efficiently store and analyze the data. In this paper, we analyze the time-series database from the perspective of I/O using various storage devices from HDD, SATA and NVMe SSD. First, we analyze the I/O characteristics such as runtime, throughput and size of total requests using various storage devices. In addition, we analyze the effect of unique time-series database features such as data chunk interval, compression and number of workers. Our analysis results show that adapting high-performance devices can greatly improve the performance of the database by up to 33.22×.

INDEX TERMS Performance analysis, NVMe SSD, SATA SSD, time-series data, benchmark, database.

I. INTRODUCTION

In modern computing, as the importance of data increases, collecting and analyzing data to produce new insights is becoming more and more important [1], [2], [3], [4]. For a detailed analysis of the problem, it is becoming common in both industry and academia to collect data from many types of devices such as IoT and sensor data [1], [5]. These devices are widely used to collect various types of data such as manufacturing data from the factory and environmental data from smart farms [6]. For example, various analyses can be conducted by analyzing time-series data such as gas sensor data, CPU data to identify trends or predict future values.

When collecting data from these IoT and sensor devices, the data is often organized based on time and referred to as time-series data. Time-series data has many unique characteristics compared with traditional data. First, the data is continuously collected at a uniform time interval and stored in an append-only manner. Second, since the primary purpose of time-series data is analysis, update or delete operations occur relatively less frequently compared to other general workloads. Finally, time-series data is generated from a large number of data sources, so the size of data stored in the database is usually large-scale data. These characteristics of time series data need to be carefully considered and require a unique approach when managing the data to improve performance and reduce capacity overhead.

To handle such time-series data, time-series databases like TimescaleDB [12] and InfluxDB [7] are widely adopted. These databases are specifically designed to store and access time-series data. They process data by taking advantage of the time-based nature of time-series data and offer unique

The associate editor coordinating the review of this manuscript and approving it for publication was Genoveffa Tortora.

features such as data compression engines to efficiently utilize disk space. For example, TimescaleDB, which operates as an extension of PostgreSQL, divides the tables into small-sized chunks for manageability and processes data in a column-oriented format, allowing data from each data source (column) to be managed independently. In addition, each chunk of data is compressed by time range and then stored in the storage, reducing the storage overhead. This append-only nature and compression result in unique I/O characteristics of time-series data that differ from traditional relational database systems. Thus, to efficiently handle large-scale time-series data with these unique characteristics, it is important to investigate the I/O characteristics of the database, which is a major factor in the overall database performance.

There have been many studies that analyze time-series data with its characteristics. Wang et al. [8] proposed a method for clustering time-series data based on structural characteristics of the data from different data sources. In addition, there have been many studies that analyze performance using different storage devices. Xu et al. [9] analyzed the performance and I/O characteristics of both relational and NoSQL databases using emerging storage devices such as SATA and NVMe SSD. Iwata et al. [10] proposed a method to maximize overall system throughput by writing to SATA SSD and NVMe SSD disks concurrently with RocksDB. Previous studies analyzed the characteristics of time-series data or examined the impact of NoSQL databases on emerging storage devices. In contrast to these previous studies, our study focuses on time-series data from the perspective of I/O characteristics using various storage devices, from HDDs and SATA SSDs to NVMe SSDs. To the best of our knowledge, this is the first research that analyzes the I/O characteristics of time-series data using a variety of storage devices.

In this paper, we analyze and evaluate the time-series data from the perspective of I/O using various storage devices such as HDD, SATA SSD, and NVMe SSD. We target these devices as they are widely adapted and cover a wide range of devices, from cost-efficient and large-capacity HDD to high-performance NVMe SSD. We focus on time-series databases and workloads because they are write-dominant and append-dominant workloads that require in-depth investigation using various storage devices. With these unique characteristics, we aim to answer two research questions: 1) How do the distinct functionalities (e.g., chunking and compression) of time-series databases impact I/O performance and behaviors? 2) How does adopting various storage devices (e.g., HDD, SATA SSD, and NVMe SSD) affect the performance and I/O behaviors of the underlying storage layers? To do this, we used a widely used time-series data benchmark (Time Series Benchmark Suite (TSBS) [11]) and a database (TimescaleDB [12]). With this setup, we performed an analysis from the perspectives of runtime, throughput, data and request size, I/O type and processes, and scalability.

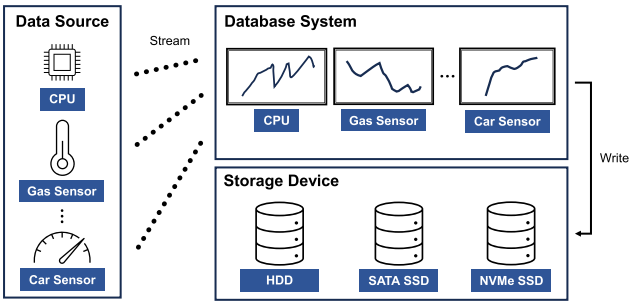


FIGURE 1. Data flows of time-series data.

Through analysis, we present key observations as follows:

- The performance of the database is significantly influenced by the underlying storage device, while chunk size and compression have minimal impact on the performance.
- While adopting high-performance devices reduces the runtime, time-series database is unable to fully utilize the available bandwidth of the underlying storage devices across all devices, with utilization ranging from 3.1% to 6.3%.
- Compression is efficient in reducing the total data size, but it does not alter the total size of I/O requests to the storage.
- I/O requests are mostly random and primarily dominated by write synchronization and direct I/O, indicating potential areas for future improvement to optimize overall performance.
- Increasing scalability can positively impact high-performance devices, but it may lead to reduced performance in slow-storage devices.

This paper is organized as follows. First, in Section II, we explain time-series data and databases for managing time-series data. In Section III, we present the results of the benchmark, covering aspects such as runtime, throughput, data and request size, I/O type, processes, and scalability. Section IV describes related work, and in Section V, we conclude the paper.

II. BACKGROUND

A. TIME-SERIES DATA

As the importance of data grows, many devices such as IoT and sensors are used to collect data from various sources. These data are usually used to record the state of each device at a regular time interval. Through analyzing these data, new insights can be generated such as forecasting future values, detecting anomalies or patterns, understanding trends and seasonality, and making data-driven decisions based on historical patterns and correlations [13], [14], [15]. Since these data are recorded at regular time intervals, they share common time-related properties and are categorized as time-series data. Time-series data is continuous, as time is represented by a continuous value. Additionally, the values in time-series data change gradually, as they depend on previous

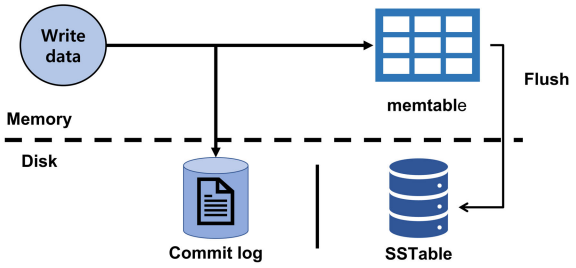


FIGURE 2. Cassandra write process.

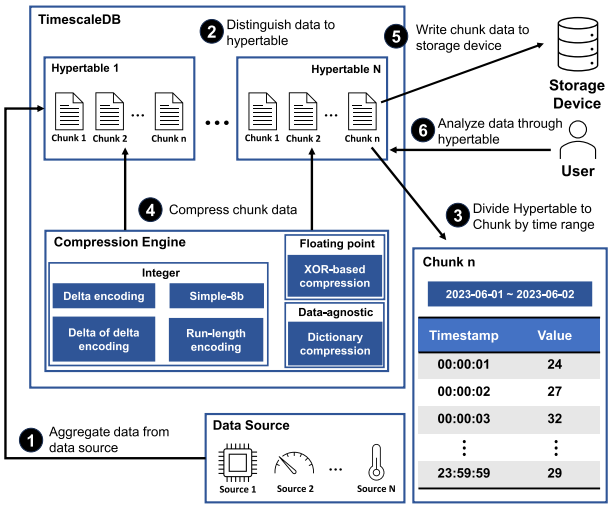


FIGURE 3. TimescaleDB architecture.

data points. Moreover, because the data is continuously recorded, time-series data exhibits an append-dominated workload with infrequent updates and deletions.

Figure 1 shows the overall data flow of time-series data collected using multiple devices. As shown in the figure, there are different hardware devices generating time-series data such as CPU, gas sensors, and car sensors. Data generated from numerous data sources is first sent to the database system. Then, the received data is aggregated in chronological order based on each stream with a single timestamp. Also, the aggregated data are distinguished by data sources and stored in various storage devices such as HDD, SATA SSD, and NVMe SSD. As depicted in the figure, the overall data flow of time-series data is similar to other database systems. However, since the data from multiple streams should be managed independently with chronological information, it requires unique approaches to handling time-series data. Thus, having a unique approach to storing and analyzing time-series data is important for improving performance.

B. MANAGING TIME SERIES DATA

1) NOSQL DATABASES

NoSQL databases such as Cassandra [16], MongoDB [17], and RocksDB [18] are emerging to store unstructured data with various types. In addition, it can be used to store time-series data which also have different data streams and types. Figure 2 shows the overall process of write operations in one of the most popular NoSQL databases, Cassandra. While we present Cassandra, the figure we provided is also applicable to other NoSQL systems such as LevelDB and RocksDB. Cassandra writes three important data to the storage which are: commitlog, memtable, and SSTable.

The commit log is an append-only record of all changes made to a node for data persistence. This allows the database system to prepare for unexpected shutdowns and recover any changes that were not yet flushed to disk. After recording changes into the commit log, the data is first stored in the memtable. The memtable is an in-memory space where the database buffer processes write operations. Data is first written to the memtable, which resides in memory with the key as the timestamp and the column as the value. When a memtable is full with key-value pairs, it is flushed to an immutable SSTable on disk to free up commitlog space when the memory usage exceeds the set threshold or the commit-log reaches its max size.

However, when using NoSQL to store time-series data, it can be challenging to traverse multiple time-series data that are stored in different SSTables since the data is organized as key-value pairs. For example, when storing data from multiple data sources, multiple NoSQL databases should be created for each source. This is because the key in a database is unique and cannot be shared among multiple values. Since multiple sensors record the data at a single timestamp, they cannot share a single key (timestamp) to store each of their data. In addition, when analyzing the time-series data from multiple sources, multiple get operations from multiple databases are required, significantly increasing the analysis overhead. Thus, when storing and analyzing time-series data, an efficient data management scheme is required.

2) TIME SERIES DATABASES

To support time-series data, time-series database such as IoTDB [19], InfluxDB [7], OpenTSDB [20], and KairosDB [21] are emerging. Among these time-series databases, TimescaleDB [12] is a widely used database as it supports traditional SQL queries. It is implemented as the extension to the existing popular Relational DBMS (RDBMS), PostgreSQL [22]. Its goal is to reflect the characteristics of time-series data, such as append-only, continuous, and large-scale, while maintaining the existing data management mechanism of the RDBMS. Figure 3 shows the overall architecture of PostgreSQL with TimescaleDB. First, when data is generated from the different data sources, TimescaleDB continuously aggregates data ❶. After data are aggregated, TimescaleDB categorizes each data by the stream and records the data into an abstract table called Hypertable ❷. A Hypertable is composed of multiple chunks, which are subsets of data from a stream. Each chunk holds data from a specific time range

of the stream. If a chunk for a specific time range does not exist when inserting data, Timescale automatically generates a new chunk for that time range and stores the data within it. As a result, by creating chunks based on time ranges and data streams and effectively managing multiple chunks within the Hypertable structure, time-series data can be analyzed with both temporal and stream-specific information. This is achieved while retaining support for the traditional RDBMS interface.

For example, as illustrated in the figure, the chunk in Hypertable N is configured to contain data exclusively within the time range of 2023-06-01 to 2023-06-02 ❸. With this configuration, data from 2023-06-03 is stored in a separate chunk. After the chunks are created, if the compression is enabled, the TimescaleDB compression engine compresses the chunks to reduce the storage overhead ❹. TimescaleDB compression engine consists of several compression algorithms for different data types. For example, if the chunk data is an integer, it is compressed with delta encoding [23], simple-8b [24], delta of delta encoding [25], and run-length encoding [26]. Also, in the case of a floating point, it is compressed with XOR-based compression [25]. Moreover, TimescaleDB can compress data using dictionary compression [25] that works by making a list of the possible values that can appear, and then just storing an index into a dictionary containing the unique values instead of storing values directly. With these compression algorithms, TimescaleDB can reduce the size of data by 80 percent, improving the overall storage utilization. The compression process is performed in TimescaleDB at the application layer before generating disk I/O requests to the file system layer. The compressed request, with a reduced request size, is then sent to the file system. Finally, the chunks are stored in the underlying storage device such as HDD, SATA SSD, and NVMe SSD ❺. After the chunks are stored, users can analyze through Hypertable with SQL queries as they are using conventional PostgreSQL ❻.

While time-series databases utilize the traditional RDBMS interface and storage engine, they can exhibit different characteristics due to their unique features. For instance, the introduction of Hypertable can introduce complexity to DBMS software and aggregate I/O operations to the underlying storage devices, including HDDs, SATA SSDs, and NVMe SSDs. When the configured time interval of each chunk changes, it can impact the size of each request, the processing of I/O requests, and the overall runtime of the workload. Given the growing demand for storing and analyzing time-series data, it becomes crucial to assess the performance of time-series databases using emerging storage devices such as SATA and NVMe SSDs. Consequently, we evaluate the performance of a time-series database with HDDs as traditional storage devices, and SATA and NVMe SSDs as emerging storage devices, using a time-series data workload.

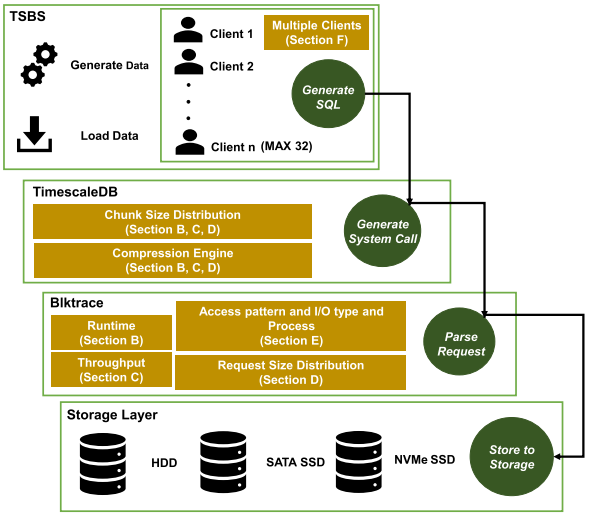


FIGURE 4. Experimental procedure.

III. ANALYSIS

Figure 4 shows our overall experimental procedure used in the paper. In this section, as depicted in the figure, we present the results of our analysis of time-series databases using various storage devices, including HDDs, SATA SSDs, and NVMe SSDs, from the perspective of various I/O characteristics, including runtime, throughput, data size, and more.

A. EXPERIMENTAL SETUP

1) SYSTEM SETUP

For the evaluation, we utilized a server equipped with an Intel i9-9900K featuring 16 physical cores and a maximum frequency of 5.0 GHz. It is equipped with DDR4 DRAM, offering a capacity of 16GB. For storage devices, we employed a Seagate BarraCuda Pro 7200 HDD with a capacity of 1TB [27], a Transcend SSD230S 2.5 SSD with 128 GB [28], and an Intel Optane 900P Internal NVMe SSD with 280 GB [29]. We selected these storage devices with diverse configurations to analyze the impact of these devices on time-series data and databases. In terms of software, the device is operating on Ubuntu 22.04.2 LTS with the 5.15.0 Linux kernel.

2) WORKLOAD CHARACTERISTICS

For the evaluation, we employed the Time Series Benchmark Suite (TSBS) [11], a benchmark specifically designed for assessing time-series databases. Within TSBS, we utilized the 'dev ops' workload, which generates, inserts, and measures time-series data related to system resources (such as CPU, memory, disk, etc.) from 9 distinct systems (streams). We used the scale flag of 1000 which generates 10GB of total data. For the database, we used TimescaleDB [12] which is a widely used time-series database that operates as an extension of PostgreSQL [22].

TABLE 1. Request types from Blktrace.

Operation	Specification
W	Write to file from the buffer.
WM	Write metadata to file.
WS	Write data from the buffer to the disk.
RA	Read a file from disk and load it to the page cache.

3) BLKTRACE

To analyze the I/O characteristics of the time-series data, we collect the trace of I/O requests when the benchmark and underlying database are running. For the collection of I/O requests, we use the Blktrace [30] tool, which is a default I/O traffic tracer for Linux, capturing block I/O information on block devices at the block layer. By capturing information at the block layer through Blktrace, we are able to obtain the I/O characteristics after the requests have been processed by other layers(e.g., memory management and file system) and understand how they are issued to underlying storage devices. For example, after parsing the collected trace using Blkparse [31], the trace shows the information of I/O requests such as process name, timestamp of request that occurred, size of the request, and the type of request. In addition, it defines various request types and we focus on the following four request types as defined in Table 1.

B. RUNTIME

To evaluate the effect of various storage devices on the time-series database, we first analyze runtime using HDDs, SATA SSDs, and NVMe SSDs.

1) WITHOUT COMPRESSION

To evaluate the impact of storage devices and compression engines on time series data insertion performance, we measured the runtimes of storage devices using a benchmark that performs write operations on the TSBS (Time Series Benchmark Suite) with pre-generated 10GB data. For the evaluation, we disabled the compression to focus solely on the performance of each device without compression. Additionally, we set different chunk sizes at 12 hours, 4 hours, 30 minutes, and one minute.

Figure 5 displays the runtime results of using HDD, SATA SSD, and NVMe SSD with different chunk sizes. The runtime is represented as a square root due to the significant variations observed among storage devices in the results. As depicted in the figure, the runtime for a one-minute chunk size on HDD is 8538 seconds, which is 1.95 times higher than the runtime of 4371 seconds for the 30-minute chunk size. In contrast, the runtime using SATA-SSD and NVMe SSD shows minimal differences when using different chunk sizes. For SATA-SSD, the result of a one-minute chunk size is 1154 seconds, only 1.05 times higher than the 30-minute chunk size runtime of 1095 seconds. Additionally, for NVMe SSD, the one-minute chunk size yields 249 seconds, which is even lower than the 30-minute chunk size runtime of 257 seconds. Comparing

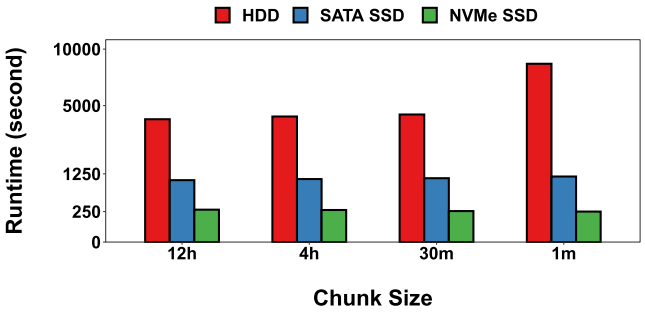


FIGURE 5. Benchmark runtime without data compression.

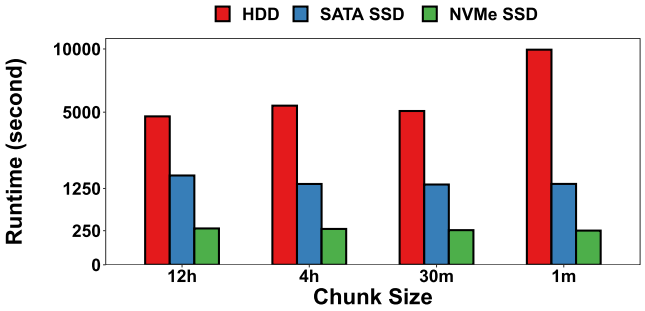


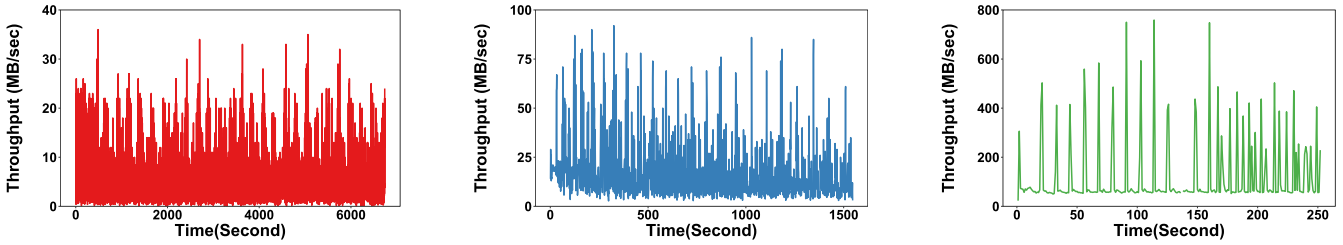
FIGURE 6. Benchmark runtime with data compression.

the slowest HDD runtime, NVMe significantly improves performance by 33.22× in the case of a one-minute chunk size.

These results demonstrate that the performance of underlying storage devices can significantly impact the overall performance. While a small chunk size has negative effects when using a slow storage device (e.g., HDD), the chunk size does not exhibit any significant impact when it increases beyond 30 minutes. Additionally, when faster storage devices such as SATA SSD and NVMe SSD are utilized, the chunk size has minimal impact on the performance.

2) COMPRESSION

Figure 6 displays the runtime results of utilizing HDD, SATA SSD, and NVMe SSD with data compression enabled. Additional experiments were conducted to analyze the effect of data compression on the overall time-series database performance. The results show that with data compression, runtime with HDD and SATA SSD increased. For HDD, the runtime of one-minute chunk size data is 9933 seconds, which is 1395 seconds longer (16.34%) than without compression. Similarly, when using a 12-hour chunk size with SATA SSD, the runtime is 1712 seconds, representing an increase of 683 seconds (39.93%) compared to the uncompressed scenario. However, in the case of NVMe SSD, the runtime difference between compression and no compression is minimal, ranging from 1 to 2 seconds. Especially in the case of a 30-minute chunk size, the runtime with the compression engine decreased by 0.3 seconds compared to the result of raw data.



(a) HDD (b) SATA SSD (c) NVMe SSD

FIGURE 7. Throughput of storage device with one-minute chunk size compressed data.

TABLE 2. Bandwidth, throughput, and average utilization of devices.

Device	Bandwidth (MB/sec)	Throughput (MB/sec)	Utilization (%)
HDD	160	5.02	3.3
SATA SSD	520	16.61	3.1
NVMe SSD	2000	126	6.3

These experiments with data compression enabled demonstrate that while compression has a significant impact on performance when utilizing slow storage devices, its effect is negligible when adapting high-performance devices such as NVMe SSDs. Therefore, since high-performance devices are widely adopted for processing time-series data, using compression can be more beneficial as it reduces the total data size while maintaining a similar runtime. These results lead to our first observation.

Observation 1. *The performance of the database is significantly influenced by the performance of the underlying storage device, resulting in differences of up to 33.22 times. While chunk size affects the performance when using HDD, it does not impact performance when using fast storage devices (SATA and NVMe SSD). Additionally, utilizing data compression can be beneficial when employing high-performance devices, as it reduces the data size while maintaining similar performance..*

C. THROUGHPUT

While previous evaluations demonstrate that using high-performance storage devices can enhance the performance of time-series databases and reduce the overall runtime, they do not indicate whether the time-series database fully utilizes the potential of the storage devices. To determine whether the database is fully exploiting the performance capabilities of the underlying storage devices, we measure the throughput of each device running identical benchmarks as the previous section. By analyzing the throughput, we can verify the extent of storage device utilization and determine whether the slow performance of devices such as HDD and SATA SSD is the primary bottleneck for the time-series databases. To gather the throughput data for each device, we employed Linux dstat [32], a real-time system resource monitoring tool capable of collecting total disk I/O utilization.

Figure 7 displays the measured throughput for HDD, SATA SSD, and NVMe SSD. The X-axis represents time in seconds, and the Y-axis represents MB per second. Data points are collected every second. Furthermore, in Table 2, we provide maximum supported bandwidth by hardware, recorded average throughput, and average utilization, for each storage device. For HDD, the average throughput is 5.01 MB per second, with an average utilization of 3.3%. Although the maximum throughput almost reaches 40 MB per second, it constitutes less than 25% of the maximum bandwidth. Regarding SATA SSD, the average throughput is 16.61 MB per second, with an average utilization of 3.1%, and the maximum throughput is less than 20% of the maximum bandwidth. Similarly, for NVMe, while the average throughput is significantly higher at 126 MB per second, with an average utilization of 6.3%, the maximum throughput reaches around 800 MB per second, which is less than 40% of the maximum bandwidth. In all cases, regardless of the storage device, the time-series database does not fully utilize the underlying storage devices. Even in the best-case scenario when storage is maximally utilized, it still utilizes less than 40% of the bandwidth of the storage devices. These results highlight the potential bottleneck within the storage software of time-series databases. Consequently, optimization is necessary to ensure the full utilization of the storage bandwidth.

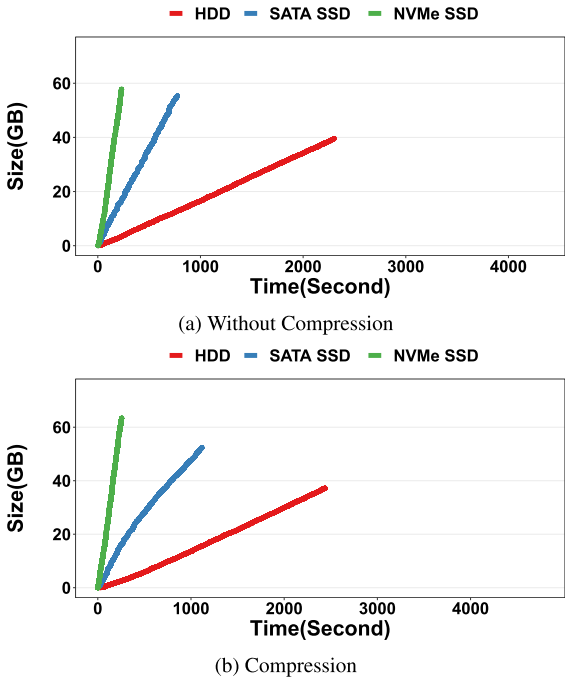
Observation 2. *As expected, the I/O performance improves when adopting high-performance devices such as NVMe SSD. However, for all the storage devices, the average utilization remains below 10% and 40%, even when the utilization is the highest. This shows that simply adopting high-performance devices cannot resolve the bottleneck and fully utilize the devices' performance. Thus, there exists significant potential for performance improvement through optimization of the I/O software layer.*

D. TOTAL DATA AND REQUEST SIZE

When storing time-series data from multiple streams, it is organized based on timestamps. As all the data shares the same timestamp, many time-series databases, such as TimescaleDB, offer a compression engine to reduce the total data size and storage capacity overhead. Apart from storage capacity considerations, data size can influence not only the

TABLE 3. Total data size in storage device.

Compression	Data size (GB)
Disabled	8.2
Enabled	1.5



(a) Without Compression

(b) Compression

FIGURE 8. Request size distribution during runtime with 12-hours chunk size.

I/O performance during execution but also the lifespan of the device [33], [34]. Thus, we analyze the impact of data compression in terms of total data size and request size during the execution of the benchmark.

Table 3 presents the total data size after the benchmark finishes executing with compression enabled and disabled. During the evaluation, the benchmark is configured to generate 10GB of time-series data. As indicated in the table, the total data size without compression is 8.2 GB. However, with the compression engine enabled, the data size is only 1.5 GB, which is 18% of the total data size without compression. These results demonstrate the effectiveness of compression in significantly reducing storage capacity overhead.

While the total data size differs significantly after the execution of the benchmark, it is also important to analyze the total request size that was issued during the execution, as it determines the I/O processing overhead. Table 4 shows the changes in total request size with and without compression. In addition, Figure 8 shows the distribution of requests using different storage devices and compression. As shown in the figure, the size of the requests gradually increases throughout the runtime, irrespective of the storage devices or compression.

TABLE 4. Changes in total request size with compression compared to without compression. The unit is in MB.

Device	12 hours	4 hours	30 minutes	1 minute
HDD	+400	+2273	+1247	+2745
SATA SSD	+889	−8088	+1260	−4310
NVMe SSD	+552	+1066	+3426	+5332

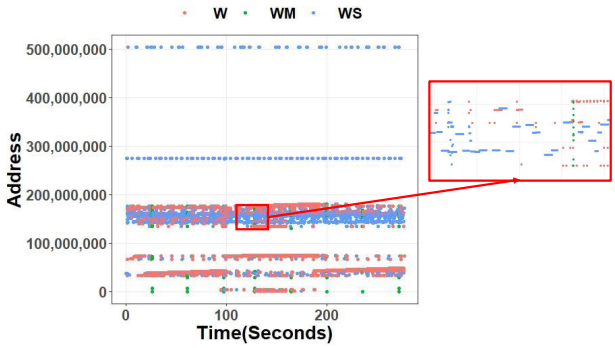


FIGURE 9. Access pattern of NVMe SSD with 4-hours chunk size.

In terms of total request size, As shown in the table, in most cases, the request size with compressed enabled is larger than that without compression. For example, in the case of an HDD with a one-minute chunk size, the total request size is 2745 MB larger with compressed data compared to without compression. Similarly, with a one-minute chunk size, an NVMe SSD has a difference of 5332 MB in favor of compression. In contrast, SATA SSD exhibits smaller differences in a compression-enabled environment for chunk sizes of 4 hours and 1 minute, showing 8088 MB and 4310 MB respectively, compared to the scenario without compression. These results are surprising because, with the compression engine at the application layer, there should be fewer requests generated with smaller sizes compared to when there is no compression. Thus, the results do not align with the common perception that compression reduces the I/O amount due to requests with compressed data size. This leads to our third observation.

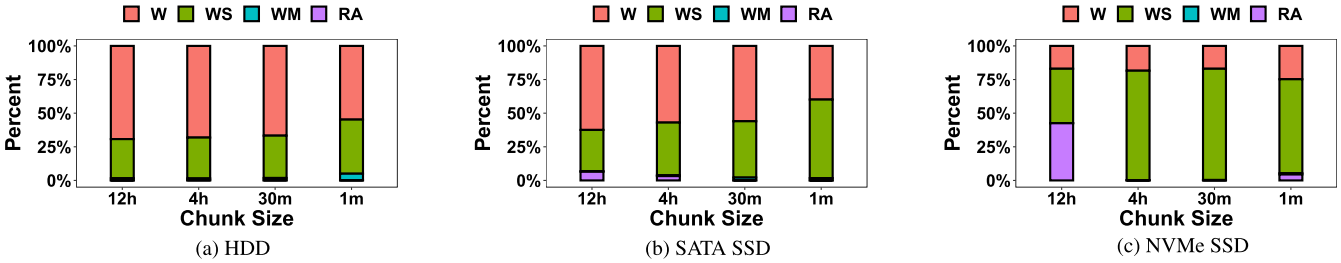
Observation 3. *Enabling compression can significantly reduce the total data size by 18%. However, the total request size during execution is similar or even larger when compression is enabled. These findings suggest that there are duplicated or unnecessary I/O requests that disrupt the performance of storage devices. Nevertheless, it is advisable to utilize compression to reduce the total data size, as the total requests do not differ significantly.*

E. ACCESS PATTERN, I/O TYPE AND PROCESSES

To further analyze the I/O characteristics of time-series databases, we present the access pattern, I/O type, and processes that perform the I/O operations.

1) ACCESS PATTERN

In the aspect of the performance of the storage device, sequential and random writes are one of the main factors.



(a) HDD (b) SATA SSD (c) NVMe SSD

FIGURE 10. I/O request types for different storage devices and chunk sizes.

Thus, we analyze the access pattern and analyze the spatial locality of the requests. Figure 9 shows the access pattern of I/O requests of NVMe SSD with a 4-hour chunk size. The X-axis represents time in seconds, and the Y-axis represents the block address of I/O requests. Each data point represents write (W), merged write (WM), and synchronized write (WS) I/O requests. As shown in the figure, the writing process is generated randomly. Many of the synchronized write requests, which incur the most overhead, target the block requests between 100 million and 200 million, which are the block addresses of the log file. This result is surprising because time-series data have strong temporal and spatial locality due to the shared timestamp, but the write requests show a random write pattern with no temporal and spatial locality.

2) I/O TYPE

Figure 10 shows the ratio of each I/O request types collected from blktrace, as previously mentioned in Table 1. Each graph bar consists of a ratio of each event.

In the case of HDD, as depicted in Figure 10a, almost no read operations occurred throughout all chuck sizes. In addition, using a smaller chunk size of one minute increased the write synchronization ratio by 11.2% compared with 12 hours. Similarly, Figure 10b demonstrates that the ratio of write synchronization increased by 27.8% when transitioning from a chunk size of 12 hours to one minute in SATA SSD. However, NVMe SSD exhibits a different pattern, as depicted in Figure 10c. For a chunk size of 12 hours, 42.5% of requests are read-ahead operations. Furthermore, the ratio of write synchronization is notably higher compared to that of HDD and SATA SSD, accounting for up to 82.7%.

TABLE 5. I/O request count.

Device	12 hours	4 hours	30 minutes	1 minute
HDD	192,095	179,750	165,312	323,108
SATA SSD	197,760	149,718	131,605	312,041
NVMe SSD	568,890	299,980	257,970	454,910

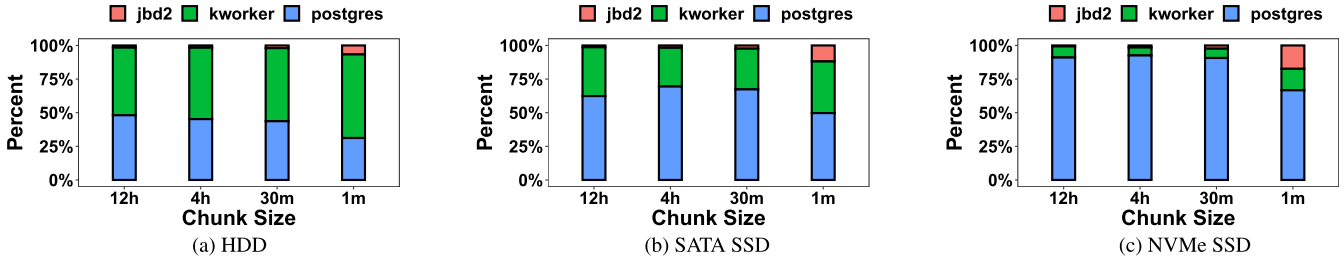
This demonstrates that as the performance of devices improves, the write requests are synchronized directly by the database rather than buffered in memory. To further investigate this, we analyzed the changes in request count using different storage devices and chunk sizes, as shown in Table 5. As depicted in the table, the number of I/O requests increases significantly in all storage devices, by up to 2.08×. This is because when using a 1-minute chunk size, the data is issued more frequently to the underlying storage devices. Thus, the data cannot be buffered in memory and merged into a few large requests. Additionally, the number of write requests is higher in general when using NVMe SSD compared to using HDD and SATA SSD. This conforms with our previous analysis that requests immediately synchronize with small request sizes rather than being buffered and merged into a large request.

3) I/O PROCESSES

In addition to analyzing the types of I/O requests, we also investigated how these requests are processed by the processes. Using blktrace, we collected data on the processes handling the I/O requests. Upon analyzing the data, we found that the majority of the requests were managed by three processes: jbd2, kworker, and postgres. In the ext4 [35] file system, jbd2 [36] is a journaling module responsible for backing up metadata and data within the file system. Kworker [37], which stands for kernel worker, is a kernel process tasked with handling I/O requests issued to the kernel. Lastly, postgres refers to the PostgreSQL database that directly performs I/O operations received from time-series databases.

Figure 11 shows the process ratio for different storage devices and chunk sizes. For HDD, as evident in Figure 11a a significant portion of the requests are managed by the kworker process, reaching up to 50.2%. In the case of SATA SSD, as shown in Figure 11b, the kworker process ratio decreases, accounting for only up to 28.8%. In contrast, the majority of requests are directly handled by postgres. Finally, Figure 11c depicts the process ratio of NVMe SSD. In contrast to HDD and SATA SSD, the kworker process ratio is only 15.8% or lower. Most of the requests are handled by postgreSQL directly by up to 92.6%.

These findings demonstrate that when the performance of the device is lower, writes are indirectly handled by the kworker process. In contrast, as high-performance devices are employed, writes are directly executed by the time-series database. These results lead to our fourth observation.



(a) HDD (b) SATA SSD (c) NVMe SSD

FIGURE 11. I/O handling process for different storage devices and chunk sizes.

Observation 4. *Time-series workloads are write-dominant and append-only workloads. However, our analysis results show that random write requests are generated in the block layer. Furthermore, as the performance of devices increases, the ratio of synchronized writes from the time-series database also increases. Therefore, it becomes crucial to optimize the write request pattern and minimize synchronization overhead in order to optimize time-series databases for high-performance devices.*

F. SCALABILITY

In real-world, time-series data is inserted by many different data sources such as sensors and devices. It is possible for multiple clients to concurrently issue requests to a single database system. Therefore, the time-series database system should be able to provide good performance when using a large number of clients. To analyze this, we performed analysis with an increasing number of clients and verified the scalability of time-series databases.

Figure 12 depicts the runtime with an increasing number of clients. We enabled compression in all cases, and each bar graph represents HDD, SATA SSD, and NVMe SSD.

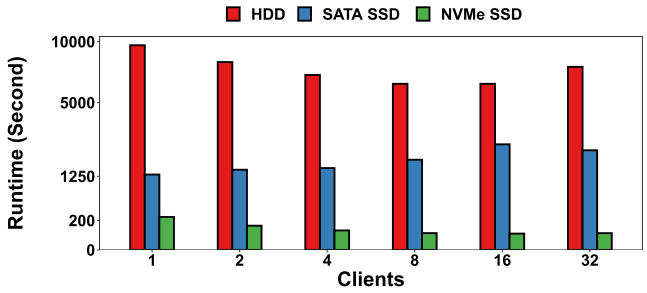


FIGURE 12. Runtime with different number of clients.

As shown in the figure, in the case of HDD, the runtime decreases as the number of clients increases. The runtime for a single client is 9657 seconds, while for 16 clients, it is 6360 seconds, decreasing by a factor of 1.52. In the case of SATA SSD, the runtime increases as the number of clients increases. For 16 clients, the runtime is 2570 seconds, which is 1.96× higher than the 1309 seconds recorded for a single client. In the case of NVMe SSD, the runtime is 250 seconds with one worker and 61 seconds with 16 workers, resulting in a performance improvement of 4.09×.

The results indicate that the performance of HDD decreases at a certain number of clients, while that of SATA SSD decreases as the number of clients increases. We believe this is due to the limited parallelism offered by each device, and a high number of clients creating I/O request contention. In contrast, when high-performance storage devices are utilized, the increasing number of clients enhances the performance, as these devices possess high device parallelism and can support the high I/O request rates caused by a large number of clients. This leads to the fifth and final observation.

Observation 5. *When utilizing high-performance devices with parallelism, the time-series database exhibits strong scalability. Therefore, an increasing number of clients can enhance the performance of time-series databases.*

IV. RELATED WORK

A. TIME-SERIES DATA ANALYSIS

There have been many studies that analyze time-series data and its characteristics. Wang et al. [8] proposed a method for clustering time-series based on data structural characteristics. This method focuses on global features extracted from the time series, which are utilized for clustering point values without relying on distance metrics. Alhnaity et al. [38] adopt a prediction model that addresses problems arising from the financial time-series data characteristics, which include difficulties in capturing states and accurately describing financial time-series data. Lubba et al. [39] introduces a method to infer small sets of time-series features that exhibit strong classification performance across a given collection of time-series problems while minimizing redundancy.

Our study aligns with these research efforts in terms of analyzing time-series data based on their data characteristics. However, in contrast, our research focuses on analyzing the data from the perspective of I/O characteristics, utilizing emerging storage devices such as SATA SSD and NVMe SSD.

B. IMPROVING DATABASE PERFORMANCE USING EMERGING STORAGE DEVICES

There have been many studies that improved the performance of database systems using emerging storage devices such as SATA SSD and NVMe SSD. Xu et al. [9] analyzed the performance and I/O characteristics of both relational and

NoSQL databases using emerging storage devices such as SATA and NVMe SSD. Iwata et al. [10] proposed a method to maximize overall system throughput by writing to SATA SSD and NVMe SSD disks concurrently with RocksDB. Kim and Son [40] proposed a key reshaping technique to improve the performance of KV stores with high-performance storage devices such as SATA and NVMe SSD.

Our study aligns with these studies in terms of evaluating database performance using emerging storage devices. However, in contrast, our study aims to analyze the performance of the time-series database and focuses on various I/O characteristics such as runtime, throughput, total data size, total request count, request type, and request handling processes. Through this, we aim to identify areas for optimization to achieve high performance while reflecting the characteristics of time-series data when adapting high-performance devices in time-series databases.

V. CONCLUSION

In this paper, we analyze the performance of various types of storage devices based on time-series data characteristics. To achieve this, we evaluate and analyze each device across aspects such as runtime, scalability, throughput, request size, and request type. The analysis results show that while adopting high-performance devices can improve the performance of time-series databases, the utilization remains under 10% in all devices. This indicates that naive adaptation of fast storage devices has limited performance improvement and further optimization is needed to fully exploit the storage hardware. Additionally, our detailed analysis of I/O characteristics reveals that synchronization overhead is a key bottleneck, and improving the scalability of the database can enhance its performance.

Thus, in future work, we will investigate the reasons for the low utilization in each device by analyzing database or kernel-level code, with a particular focus on synchronization primitives such as the flush operation and scalability primitives such as locking. Furthermore, we will utilize new emerging storage hardware, such as Zoned Namespace SSD, which is composed of several logical zones that store related data in sequential order, to handle time-series data based on its characteristics, such as a write-dominant workload and chunk data structure.

AUTHOR CONTRIBUTIONS

Sangmyung Lee-Conceptualization, experiments, writing; Yongseok Son-Experiments, writing, review and editing; Sunggon Kim-Experiments, writing, supervision.

REFERENCES

[1] M. Marjani, F. Nasaruddin, A. Gani, A. Karim, I. A. T. Hashem, A. Siddiqa, and I. Yaqoob, "Big IoT data analytics: Architecture, opportunities, and open research challenges," *IEEE Access*, vol. 5, pp. 5247–5261, 2017.

[2] Q. Peng and X. Ye, "Research trends in social media/big data with the emphasis on data collection and data management: A bibliometric analysis," in *Empowering Human Dynamics Research With Social Media and Geospatial Data Analytics*. Cham, Switzerland: Springer, 2021, pp. 47–63.

[3] N. L. Bragazzi, H. Dai, G. Damiani, M. Behzadifar, M. Martini, and J. Wu, "How big data and artificial intelligence can help better manage the COVID-19 pandemic," *Int. J. Environ. Res. Public Health*, vol. 17, no. 9, p. 3176, 2020.

[4] A. Banerjee, C. Chakraborty, A. Kumar, and D. Biswas, "Emerging trends in IoT and big data analytics for biomedical and health care technologies," in *Handbook of Data Science Approaches for Biomedical Engineering*, 2020, pp. 121–152.

[5] R. Krishnamurthi, A. Kumar, D. Gopinathan, A. Nayyar, and B. Qureshi, "An overview of IoT sensor data processing, fusion, and analysis techniques," *Sensors*, vol. 20, no. 21, p. 6076, 2020.

[6] J. Muangprathub, N. Boonnam, S. Kajornkasirat, N. Lekbangpong, A. Wanichsombat, and P. Nillaor, "IoT and agriculture data analysis for smart farm," *Comput. Electron. Agricult.*, vol. 156, pp. 467–474, Jan. 2019.

[7] S. N. Z. Naqvi, S. Yfantidou, and E. Zimányi, "Time series databases and InfluxDB," Studienarbeit, Université Libre de Bruxelles, Bruxelles, Belgium, 2017, vol. 12.

[8] X. Wang, K. Smith, and R. Hyndman, "Characteristic-based clustering for time series data," *Data Mining Knowl. Discovery*, vol. 13, no. 3, pp. 335–364, 2006.

[9] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan, "Performance analysis of NVMe SSDs and their implication on real world databases," in *Proc. 8th ACM Int. Syst. Storage Conf.*, 2015, pp. 1–11.

[10] S. Iwata, M. Patel, P. Sarthi, and J. Shawger, "Improving performance in LSM-tree based key-value stores using NVMe," Univ. Wisconsin-Madison, Madison, WI, USA, Tech. Rep. 1, 2022.

[11] InfluxData and Timescale, Inc. *Time Series Benchmark Suite (TSBS)*. Accessed: Oct. 2, 2023. [Online]. Available: https://github.com/timescale/tsbs

[12] M. Arye, "TimescaleDB: Re-architecting a SQL database for time-series data," YouTube, Uploaded TimescaleDB Oct, Tech. Rep., 2017, vol. 18. Accessed: Oct. 2, 2023. [Online]. Available: https://github.com/timescale/timescaledb

[13] C.-J. Lu, T.-S. Lee, and C.-C. Chiu, "Financial time series forecasting using independent component analysis and support vector regression," *Decis. Support Syst.*, vol. 47, no. 2, pp. 115–125, May 2009.

[14] A. Nielsen, *Practical Time Series Analysis: Prediction With Statistics and Machine Learning*. Sebastopol, CA, USA: O'Reilly Media, 2019.

[15] M. Pourahmadi, *Foundations of Time Series Analysis and Prediction Theory*, vol. 379. Hoboken, NJ, USA: Wiley, 2001.

[16] Apache Cassandra, "Apache Cassandra," Apache Softw. Found., Wakefield, MA, USA, Version 5.0, 2023. Accessed: Oct. 2, 2023. [Online]. Available: https://cassandra.apache.org/doc/latest/cassandra/architecture/overview.html

[17] *MongoDB*. Accessed: Oct. 2, 2023. [Online]. Available: https://www.mongodb.com/docs/

[18] S. Dong, "RocksDB: Evolution of development priorities in a key-value store serving large-scale applications," *ACM Trans. Storage*, vol. 17, no. 4, pp. 1–32, 2021.

[19] C. Wang et al., "Apache IoTDB: Time-series database for Internet of Things," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 2901–2904, Aug. 2020.

[20] *OpenTSDB*. Accessed: Oct. 2, 2023. [Online]. Available: http://opentsdb.net/docs/build/html/index.html

[21] KairosDB Team. *KairosDB*. Accessed: Oct. 2, 2023. [Online]. Available: http://kairosdb.github.io/

[22] B. Momjian, *PostgreSQL: Introduction and Concepts*, vol. 192. New York, NY, USA: Addison-Wesley, 2001.

[23] R. Jindal, N. Kumar, and S. Patidar, "IoT streamed data handling model using delta encoding," *Int. J. Commun. Syst.*, vol. 35, no. 13, p. e5243, 2022.

[24] V. N. Anh and A. Moffat, "Index compression using 64-bit words," *Softw., Pract. Exper.*, vol. 40, no. 2, pp. 131–147, 2010.

[25] J. Lockerman and A. Kulkarni. *TimescaleDB Compression Algorithms*. Accessed: Oct. 2, 2023. [Online]. Available: https://www.timescale.com/blog/time-series-compression-algorithms-explained/

[26] S. Golomb, "Run-length encodings (corresp.)," *IEEE Trans. Inf. Theory*, vol. IT-12, no. 3, pp. 399–401, Jul. 1966.

[27] S. BarraCuda. *Seagate Barracuda Pro 7200 White Paper*. 2018. Accessed: Oct. 2, 2023. [Online]. Available: https://www.seagate.com/www-content/datasheets/pdfs/barracuda-pro-2-5DS1966-1-1802US-en_CA.pdf

[28] I. Transcend Information. (2019). *Transcend SSD230S 2.5 SSD With 128 Gb White Paper*. Accessed: Oct. 2, 2023. [Online]. Available: https://docs.rs-online.com/0fc9/0900766b816faddf.pdf
[29] Intel Corporation. (2018). *Intel Optane 900P Internal NVMe*. Accessed: Oct. 2, 2023. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-900p-brief.pdf
[30] J. Axboe, A. D. Brunelle, and N. Scott. *Blktrace*. Accessed: Oct. 2, 2023. [Online]. Available: https://linux.die.net/man/8/blktrace
[31] J. Axboe, A. D. Brunelle, and N. Scott. *Blkparse*. Accessed: Oct. 2, 2023. [Online]. Available: https://linux.die.net/man/1/blkparse
[32] D. Wieers. *Dstat*. Accessed: Oct. 2, 2023. [Online]. Available: https://github.com/dstat-real/dstat
[33] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD lifetimes with disk-based write caches," in *Proc. FAST*, vol. 10, 2010, pp. 101–114.
[34] T. Roy and K. Kant, "Enhancing endurance of SSD based high-performance storage systems using emerging NVM technologies," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2020, pp. 1070–1079.
[35] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proc. Linux Symp.*, vol. 2, 2007, pp. 21–33.
[36] The Kernel Development Community. *Linux Ext4 Journal Jdb2*. Accessed: Oct. 2, 2023. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/ext4/journal.html
[37] The Kernel Development Community. *Linux Kernel Workqueue*. Accessed: Oct. 2, 2023. [Online]. Available: https://www.kernel.org/doc/html/next/core-api/workqueue.html
[38] B. Alhnaity and M. Abbod, "A new hybrid financial time series prediction model," *Eng. Appl. Artif. Intell.*, vol. 95, Oct. 2020, Art. no. 103873.
[39] C. H. Lubba, S. S. Sethi, P. Knaute, S. R. Schultz, B. D. Fulcher, and N. S. Jones, "catch22: Canonical time-series characteristics: Selected through highly comparative time-series analysis," *Data Mining Knowl. Discovery*, vol. 33, no. 6, pp. 1821–1852, 2019.
[40] S. Kim and Y. Son, "Optimizing key-value stores for flash-based SSDs via key reshaping," *IEEE Access*, vol. 9, pp. 115135–115144, 2021.

SANGMYUNG LEE received the B.S. degree from the Seoul National University of Science and Technology, in 2023, where he is currently pursuing the M.S. degree with the Department of Computer Science. His research interests include database systems, I/O, operating systems, and next-generation storage devices.

YONGSEOK SON received the B.S. degree from Ajou University, in 2010, and the M.S. and Ph.D. degrees from Seoul National University, in 2012 and 2018, respectively. He was a Postdoctoral Research Associate with the University of Illinois at Urbana–Champaign. He is currently an Assistant Professor with the Department of Computer Science and Engineering, Chung-Ang University. His research interests include operating, distributed, and database systems.

SUNGGON KIM received the B.S. degree in computer science from the University of Wisconsin–Madison, Madison, WI, USA, in 2015, and the Ph.D. degree in computer science and engineering from Seoul National University, in 2021. He was an Intern with the Lawrence Berkeley National Laboratory, Berkeley, CA, USA, in 2018, 2019, and 2020. He was a Postdoctoral Research Associate with Seoul National University. He is currently an Assistant Professor with the Department of Computer Science and Engineering, Seoul National University of Science and Technology. His research interests include file systems, cloud computing, big data, distributed systems, and operating systems.