TSRL: Boosting IoT Edge Time Series Database Scalability through Lightweight Rate Limiter

Ahmed El-Naggar
Computer and Systems Engineering Department, Alexandria University
Egypt
ahmed.nagar@alexu.edu.eg

Magdy Nagi
Computer and Systems Engineering Department, Alexandria University
Egypt
magdy.nagi@ieee.org

Sahar Ghanem
Computer and Systems Engineering Department, Alexandria University
Egypt
sahar.ghanem@alexu.edu.eg

Abstract: *In the Internet of Things, edge computing decreases the latency between things and the central computing service commonly deployed on the cloud. However, as edge devices are located at the edge and closer to things, the type and scale of hardware capabilities are below the commodity hardware. This resulted in limited, in terms of computational power, deployments of Time-Series Databases (TSDB). Consequently, these TSDB do not have enough buffer space to handle an increase in data ingestion rates, causing a high loss rate of data points and sometimes database crashes. In this work, we introduce a lightweight Time-Series Rate Limiter (TSRL) deployed between the clients and the TSDB to boost its scalability and decrease the loss rate due to higher ingestion rates. A buffering of requests and a re-buffering of failed requests increase the chances of data points being successfully inserted. The proposed TSRL is modular, meaning it can be deployed without any changes to the existing TSDB engine. To evaluate the proposed solution, the IoTBenchmark tool is used to compare the performance of InfluxDB with and without a TSRL. The results demonstrate significant performance improvements across multiple metrics. It shows an increase in successfully ingested points and overall throughput, alongside a decrease in both median latency and peak memory usage. The system's scalability also improved with more allocated TSDB memory and with a higher number of concurrent clients. Additionally, an enhancement in query performance is observed.*

Keywords: *IoT, edge computing, time series, database, rate limiter.*

Received August 13, 2025; accepted December 18, 2025
https://doi.org/10.34028/iajit/23/4/15

1. Introduction

The Internet of Things (IoT) has become ubiquitous, permeating numerous aspects of our lives. The sheer volume of data generated by these connected devices, primarily in the form of time series, presents significant challenges for traditional database systems. While SQL and NoSQL databases like MySQL and MongoDB, respectively, are widely used, their inherent design makes them less than ideal for efficiently handling the unique characteristics of time series data [20]. Time Series Databases (TSDBs), such as Influx DB and Timescale DB [22] have emerged as specialized solutions, addressing these limitations by providing optimized storage, retrieval, and analysis capabilities for time-stamped data. In data-intensive environments such as energy markets, TSDBs are essential for managing time-sensitive operations like rapid ingestion and real-time querying, offering enhanced scalability and superior performance over traditional relational systems, achieving quantifiable improvements that demonstrate operations up to 100 times faster [12].

However, IoT data's increasing volume and velocity necessitate a shift towards edge computing [27]. Edge computing is a distributed computing paradigm that brings data processing and storage closer to the source of the data generation, rather than relying solely on a centralized cloud or a remote data center. The inherent challenges of managing massive data streams, including potential data spikes that can overwhelm system resources and lead to crashes or data loss, are amplified at the edge, particularly with limited resources [5]. Simply scaling server resources is often prohibitively expensive, especially for large-scale deployments. While cloud-based solutions often employ message queues like Kafka to buffer and manage data streams, these solutions can be resource-intensive and may not be optimally suited for the limited resources available at the edge [33].

Communicating the readings to the ultimate persistence warehouse in the cloud is not feasible as well [3]. In an IoT configuration, the network delay between the Thing and the cloud is too long for a Thing to handle such a task. The Thing is, most of the time, a low-power device. Performing such an IO-intensive task on the Thing would be inefficient. Moreover, the Thing should always be ready to sample the next reading and should not be busy performing long-distance IO [18].

Lost readings can be problematic as well. The sensor sampling rate choice is central to capturing some patterns that are only observable at a specific granularity [34]. However, when the data received and processed at the edge is communicated to time-series databases deployed on the cloud, these databases often fail to

ingest data under high workloads [15]. This may force the loss of the acceptable granularity as some data have been lost and, thus, it looks as if the sampling rate has decreased [19].

Therefore, an edge-optimized TSDB requires a lightweight rate-limiter [24] to manage incoming data and prevent resource exhaustion effectively. This rate limiter, deployed alongside the TSDB on the same edge device or a nearby one, acts as a gatekeeper, controlling the rate of data ingestion. This paper proposes a lightweight rate limiter designed specifically for TSDBs deployed on resource-constrained edge devices. We evaluate the performance of the proposed rate limiter using InfluxDB, a popular open-source TSDB, as the target database. The evaluation is conducted using IoTBenchmark [20], a benchmarking tool designed for IoT workloads against time series databases. Through a series of carefully designed experiments, we investigate the impact of various rate limiter parameters on InfluxDB's performance. Our results demonstrate the significant performance enhancements achieved by integrating the proposed lightweight rate limiter with InfluxDB.

This paper presents a standalone extension to our work [8] where we introduced a novel lightweight layer that utilized rate limiting, buffering, and retries to regulate the ingestion traffic directed at a time-series database.

The remainder of this paper is structured as follows: Section 2 provides background on rate-limiting algorithms and approaches, along with a review of existing performance studies and benchmarks for TSDBs. Section 3 presents a high-level design of the proposed rate limiter. Section 4 details the algorithms employed by the different components of the rate limiter. Section 5 describes the experimental setup and methodology. Section 6 presents and analyzes the experimental results. Finally, section 7 concludes the paper and outlines potential future work.

2. Related Work

2.1. Rate Limiting

Rate limiters control request rates, ensuring fair resource allocation and system stability. In addition, it helps in preventing resource abuse as Denial of Service attacks (DoS). There are five key rate-limiting algorithms: token bucket, leaky bucket, fixed window counter, sliding window log, and sliding window counter [1].

Such algorithms are mainly used for traffic burst handling and smoothing traffic spikes. The Token Bucket is ideal when bursts need to be allowed but controlled over time. The Leaky Bucket ensures a steady request rate, making it useful for traffic shaping. The Fixed Window Counter is simpler but may lead to spikes on window edges. The Sliding Window Log offers the most precise control but requires a high memory footprint. The Sliding Window Counter is a more efficient alternative to the log approach, balancing accuracy and memory usage. Rate limiting is different than load shedding that drops low-priority requests under high traffic.

Rate limiting can be applied at the server level or per-user basis. In addition, it can be applied on the network edge to limit several users. Implementing rate limiting on the edge improves resilience, scalability, and service quality. The proposed rate limiting is applied particularly in a TSDB data ingestion pipeline to prevent the database overload and protect it by capping write requests, preventing system crashes during spikes. In addition, it buffers and reschedules excess requests instead of discarding them, maintaining data integrity.

2.2. Edge TSDB

Edge computing is critical for IoT applications as it enables real-time data processing closer to the data source, reducing latency and bandwidth usage. However, edge systems face challenges such as resource constraints, network instability, and device failures. TSDBs are available for deployment in various environments, including cloud, on-premise, and edge computing setups. Comparing TSDB in usually performed in terms of:

- Data Model
- Ingestion Performance (the speed at which the database can ingest time-series data)
- Query Performance
- Resource Utilization (CPU, memory)
- Storage Efficiency (disk usage)
- Scalability (the ability to scale horizontally and vertically to handle increasing data volumes)
- Ease of Use
- Ecosystem and Integration [4, 21, 28].

Different studies show that there are tradeoffs; some databases are optimized for high-speed data ingestion, while others perform better for complex queries or large-scale data storage [25]. Edge TSDBs must support simultaneous data ingestion from numerous sources and in high-concurrency scenarios [30]. Scaling an existing TSDB is challenging due to the growing number of sensors and network issues that cause data to arrive late or out-of-order. Horizontal scaling usually adds more nodes in distributed systems.

Recently, the work in [35] shows that InfluxDB suffers from write stalls (delays during high-pressure writes) and write amplification (excessive data rewriting during compaction) under high-pressure writes. They proposed solution is an optimized InfluxDB engine (referred to as Z-Influx DB). The work in [15]. addresses the memory consumption issues in InfluxDB when handling high-cardinality time-series data and proposes a memory optimization method to reduce it. Both the work in [15, 35] modifies InfluxDB engine.

MatrixGate [32] implemented an external buffer layer

that utilized efficient policies to utilize slots in batch insertion SQL statements. IoTDB [31] used TsFile, a specifically tailored file format for time-series data, and engine for handling the case of delayed data arrivals. Dhanagari [7] [7]examines how MongoDB addresses the difficulty traditional relational databases face when processing high velocity, high volume, and heterogeneous data due to rigid schemas and vertical scale limitations; the solutions presented rely on MongoDB's document-oriented architecture to provide flexibility and good horizontal scaling via sharding, which, combined with features like dynamic shard rebalancing, inbuilt replication, and automated failover, achieves high availability and low latency in real-time environments. IntDB [29], on the other hand, optimized ingestion and query performance for spatiotemporal data.

The main motivation of our work is to develop a modular solution to the scalability problem that can be easily adopted by different scenarios and sizes without modifying the DB engine.

IoTDB Benchmark frameworks measure standard performance indicators like data ingestion rates and query response times, and monitors system resource usage, including CPU and memory consumption [20, 13].

3. The Rate Limiting Approach

Accurate and complete data is crucial for many applications, especially those where critical decisions depend on insights from ingested time series data [11]. To address the challenge of data loss during ingestion into an edge TSDB, the proposed Time Series Rate Limiter (TSRL) implements a robust strategy.

The TSRL's core design to reduce data loss relies on three key components:

1. Respecting Server Limits: it actively monitors and adheres to the server's designated request rate to avoid overwhelming the database.
2. Intelligent Buffering: when requests fail to transmit, the TSRL temporarily stores them, preventing immediate data loss.
3. Strategic Retries: it then attempts to resend these buffered requests at a later time, maximizing the chance of successful delivery as network or server conditions improve.

We will elaborate on each of these strategies in the following sections.

3.1. Rate Limiting

Edge TSDBs often operate with constrained resources like memory, disk space, network capacity, and CPU cores. These limitations directly define the maximum workload a TSDB can manage.

This workload limit is frequently expressed as a maximum number of client requests per time window. For example, InfluxDB Cloud's free tier allows only 1 MB of data ingestion per minute [16]. Other data-intensive applications, like Google Firebase, restrict both the maximum data size per request (1 MB) and overall data per second (10 MB) [10].

Ultimately, an edge TSDB's maximum capacity is a function of its resource efficiency and the server's ability to spawn individual client handlers. Exceeding these limits leads to critical issues: the TSDB might throw errors to prevent a full crash, crash entirely (e.g., from an OutOfMemoryError), or exhibit unexpected behavior like data inconsistencies.

To combat these problems, the proposed TSRL introduces a buffering layer. This layer tracks requests and enforces the TSDB's announced rate limit, effectively preventing system crashes and significantly reducing data point insertion failures.

Scaling a TSDB usually involves two main strategies: vertical or horizontal scaling.

Vertical scaling means upgrading a server's resources or replacing it entirely. However, this isn't always feasible, especially with cloud deployments where higher-tier machines might be scarce (e.g., Google Cloud Platform doesn't always stock them as readily as lower tiers [9]). Plus, migrating data to a new database adds extra cost and effort.

Horizontal scaling, conversely, involves adding more servers to distribute the workload and reduce data loss. The downside, in this case, is the need for an often-expensive load balancer to manage this distribution.

In contrast, a rate limiter offers a more flexible solution, deployable on existing Edge or cloud infrastructure. The proposed TSRL, with its integrated buffering layer, specifically adheres to the TSDB's set rate limit. This significantly reduces the frequency of data point insertion failures, offering an efficient alternative to traditional scaling methods.

3.2. Buffering Requests

Buffering is a core strategy for preventing data loss, traditionally holding incoming requests until a system can process them. Our approach enhances this by also buffering requests that fail to send to the TSDB, whether due to network problems or exceeding the rate limit. In both scenarios, these requests are temporarily stored for later transmission.

To reduce latency, we employ in-memory buffering. While this buffering layer, situated between clients and the TSDB, introduces a minor delay from rerouting requests, it provides significantly faster overall performance compared to file-system buffering. File-system buffering, though excellent for ensuring data durability by safeguarding requests from loss, comes with a substantial performance penalty.

3.3. Retry Sending

Retrying failed requests is a simple, reliable way to

prevent data loss. Failures often happen due to temporary issues like network congestion or brief outages. But these problems are usually short-lived; after a slight delay, the request can typically be resent and successfully delivered to the server.

3.4. Limitations

The TSRL effectively reduces data loss and improves accuracy, though it does introduce a slight increase in tail latency. Tail latency is formally defined as the latency experienced at high-percentile values of a system's response time distribution, reflecting the slowest transactions or data packets. It often results from resource contention, background maintenance tasks (such as garbage collection or disk defragmentation), and internal queueing effects within distributed or complex computing systems. This makes the TSRL particularly well-suited for applications where a small rise in tail latency is acceptable. Additionally, it reshapes application traffic, smoothing out occasional spikes to ensure the overall load remains within the TSDB's capacity.

3.5. Deployment

The TSRL offers flexible deployment options, varying in their proximity to the TSDB.

Deployment Scenarios:

- Cases 1 and 2: Edge-Centric Deployments
- Case 1: The TSDB and TSRL are co-located on a single edge device.
- Case 2: The TSDB is on one edge device, and the TSRL resides on a nearby edge device.

In both these edge-centric scenarios, the delay between the TSRL and TSDB is very small. The primary source of latency comes from the connection between the TSRL and the client/IoT device. These two setups are ideal for rate-limiting devices within a local area network (LAN).

- Case 3: Hybrid Edge-Cloud Deployment
- The TSDB is cloud-hosted, while the TSRL is deployed on an edge device. Here, network delay between the edge device and the cloud becomes the dominant factor in latency.
- Case 4: Full Cloud Deployment
- Both the TSDB and TSRL are located in the cloud. This configuration is designed to manage traffic from clients distributed across the internet.

In every scenario, the TSRL can be replicated behind a load balancer to enhance its horizontal scalability.

4. The Rate Limiter Design

The TSRL acts as an intermediary, sitting between the TSDB and the clients/ IoT devices. It intercepts incoming requests, deciding whether to immediately forward them to the TSDB or to buffer them for later (re)transmission. Requests are buffered either because of an active rate limit or if an initial insertion attempt fails.

Figure 1 illustrates a request's life cycle through the TSRL.

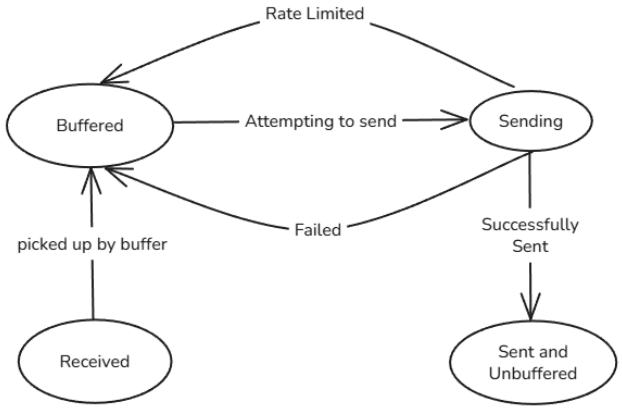


Figure 1. Request life cycle.

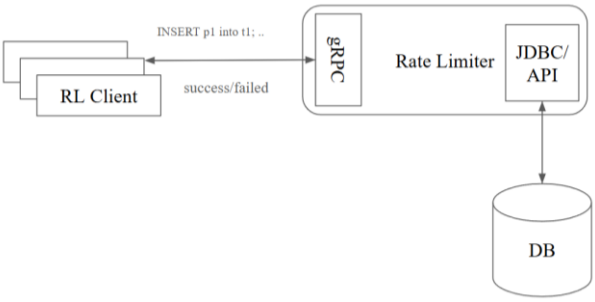


Figure 2. The rate limiter apis and communication model.

1. A request is first received and placed in a temporary holding area.
2. It then moves into a thread-specific buffer, entering a buffered state.
3. From this buffer, an attempt is made to send the request to the database, transitioning it to a sending state.
4. If the delivery fails-either due to exceeding the rate limit or a transmission error-the request is sent back to the buffered state for another attempt.
5. Once the request is successfully processed by the database, it's removed from the buffer and enters an unbuffered state, signifying completion. The request might get processed by the database, and the database may respond with a failure. In this case, this response is returned to the caller.

4.1. The Rate Limiter APIs

0Figure 2 illustrates a system where clients/ IoT devices interact with the TSRL via a Rate Limiter Client (RLClient). This integration requires modifying the client's code to embed the RLClient. The TSRL itself functions as a gRPC server, leveraging this efficient,

low-latency, cross-platform RPC framework [26]. Whereas gRPC is Google Remote Procedure Call framework that allows a client application to directly call methods on a server application in a distributed network as if it were calling a local function.

As shown, an RLClient sends SQL INSERT requests, containing data batches, to the TSRL and awaits a response. The RLClient manages the request execution through the TSRL, receiving a Boolean value indicating the success or failure of the insertion. Should the request time out, the RLClient returns False; if an error occurs, it raises an exception based on the TSRL's response.

The TSRL then forwards these requests to the TSDB, utilizing either a REST API or JDBC. For example, it communicates with IoTDB via its JDBC client library, while using a REST API for InfluxDB [2].

4.2. The Rate Limiter Components

Figure 3 offers a detailed look at the TSRL's internal architecture.

At its core, the gRPC layer acts as the API interface, taking in requests from clients. These incoming requests are then managed by multi-threaded *gRPCHandlers*, which push them to *RequestPools*. A *RequestPool* temporarily holds these requests until a *RequestBuffer* picks them up (pull requests).

From the RequestBuffer, a Limiter processes the buffered requests after pulling a batch, attempting to execute them via a DBClient. The DBClient is a simple client interface that uses the destination DB API (REST, JDBC, etc.).

If a request fails, it's returned to the RequestBuffer for a retry. The TSRL is designed to be highly scalable, supporting multiple instances of this "worker pipeline" (comprising the gRPCHandler, RequestBuffer, and Limiter), with the number of workers configured based on specific system needs.

4.2.1. The gRPC Handler and the Request Pool

A RequestPool temporarily holds incoming requests until a RequestBuffer retrieves them.

Algorithm (1) illustrates the pseudo-code for the steps a gRPCHandler performs.

For every incoming request, a RequestListener is created (Line 2) and registered in the ResponseRepository (Line 3). This repository, implemented as a hash map, acts as a crucial communication link between the RequestListener and a Limiter. Each incoming request receives a Globally Unique Identifier (GUID), which serves as the key in this map, with the corresponding listener (callback function) as its value.

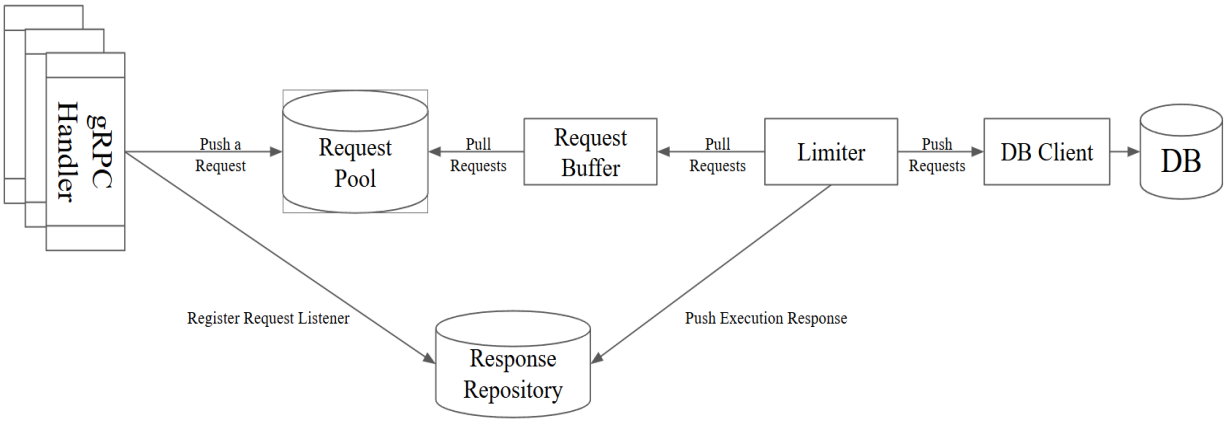


Figure 3. The TSRL worker components.

Algrithm 1. gRPC Handler

1: ***for*** *each request* ***do***
2: *Create a Request Listener to await request execution response*
3: *Register Request Listener at the Response Repository*
4: *Hand over request to Request Pool*
5: *Await execution response*
6: *De-register Request Listener from the Response Repository*
7: ***Return*** *execution response*
8: ***end for***

Algrithm 2. Request Buffer

1: ***while*** *true* ***do***
2: *requestPoolCount = number of requests in Request Pool*
3: *requestsCount = min(requestPoolCount, R_BATCH)*
4: *Pull requestsCount requests from Request Pool*
5: *Enqueue requests into memory buffer*
6: *Sleep for BUFFER_SLEEP_TIME*
7: ***end while***

A gRPCHandler attaches this callback to the request. This ensures that once a Limiter processes the request, it can return the result by invoking the callback using the GUID. The gRPCHandler then passes the request to the RequestPool and pauses, waiting for the listener to

be notified via the ResponseRepository (Lines 4-5). Once the result is ready, it's sent back to the client as a gRPC response.
In essence, the RequestPool functions as an intermediary buffer, holding requests until a RequestBuffer takes them for execution.

4.2.2. The Request Buffer

0Algorithm (2) details the operations of a RequestBuffer thread.

The process starts with the RequestBuffer pulling requests from a RequestPool, up to a maximum limit, R_BATCH (defaulting to one million requests). Once retrieved, these requests are added to a concurrent linked priority queue. This is a thread-safe, linked-list-based data structure designed for simultaneous access [6].

Concurrency is crucial here because a Limiter can dequeue requests at the same time new ones are being added. The priority queue ensures older requests are processed first (FIFO), providing a fair system when a Limiter selects requests.

After enqueuing, the RequestBuffer thread pauses for a configurable interval, BUFFER_SLEEP_TIME (Line 6). This "sleep" period acts as a back-off mechanism during high load, helping to conserve CPU usage.

Algrithm 3. Rate Limit and Re-buffer

```
1:  while true do
2:      bufferedCount = number of requests in RequestBuffer
3:      requestsCount = min(bufferedCount, L_BATCH)
4:      Pull requestsCount requests from Request Buffer
5:      for request = 1,2,… do
6:          if Rate Limit is reached then
7:              Re-buffer request again
8:              Continue
9:          end if
10:         Send request to DB
11:         if request failed to send then
12:             Re-buffer request again
13:         end if
14:         if request succeeds then
15:             Notify request listener waiting at the Result Repsitory
16:         end if
17:     end for
18:     Sleep for LIMITER_SLEEP_TIME
19: end while
```

4.2.3. The Limiter

0Algorithm (3) illustrates the operational flow of a Limiter thread.

A Limiter thread begins by retrieving a set number of requests, capped by the max L_BATCH value (Lines 1–2), from its associated RequestBuffer. It then processes these requests sequentially (Line 5).

For each request, the Limiter first checks if the rate limit has been reached (Line 6), employing a sliding-window counter algorithm (detailed in section 4.2.4). If the limit is exceeded, the request is immediately re-buffered for later processing (Line 7). While this process might alter the execution order (due to prioritization based on arrival time), time-series requests inherently contain timestamps that allow the TSDB to restore the correct sequence.

If the rate limit permits, the Limiter proceeds to execute the request via a DBClient. Should this execution fail, the request is again re-buffered (Line 12). Conversely, if the request is successfully executed, its result is sent to the ResultRepository using the request's GUID. This enables the identification and notification of the corresponding listener and callback function (Line 15). After the Limiter sends the requests and processes the response, it sleeps for the configured time by the LIMITER_SLEEP_TIME.

The sleep time configurations are introduced to prevent the Limiter from consuming CPU cycles waiting for requests when the arrival rate is low. For high arrival rates, setting the sleep times to zero (or small value) ensures CPU cycles are utilized processing requests.

4.2.4. Sliding Window Counter Algorithm

The sliding window counter Algorithm [1] operates by tracking the number of requests within the current time window. When a new request comes in, the current window's start time is set if it has not been already. If the window is active, the algorithm calculates the elapsed time. Should this duration exceed the configured window size, it then determines if the request should be allowed based on the total count.

To improve accuracy near window transitions, the algorithm also considers the previous window's request count and the overlap percentage between the two windows. This method offers enhanced accuracy compared to a basic fixed-window counter, which often struggles with requests at window boundaries. It also uses less memory than a full sliding-window log, making it a more efficient choice for the proposed rate limiter.
In Figure 4, we assume the rate limiter allows a maximum of 10 requests per minute, and there are 7 requests in the previous minute and 1 in the current minute. For a new request that arrives at a 20% position in the current minute, the number of requests in the rolling window is calculated using the following formula:

$$sliding\_window\_requests = current\_window\_requests + previous\_window\_requests * overlap\_percentage \quad (1)$$

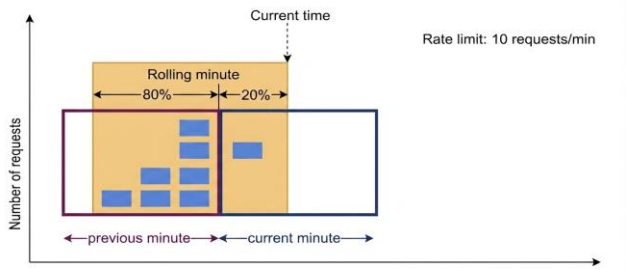


Figure 4. Sliding window counter example.

This means we add requests in current window to the requests in the previous window and multiply this by the overlap percentage of the rolling window and previous window. Applying (1) on the example, we get 1+7 * 0.8% = 6.6 requests.

Depending on the use case, the number can either be rounded up or down. In our example, it is rounded down to 6. Since the rate limiter allows a maximum of 10 requests per minute, the current request can go through.

5. Experimental Methodology

InfluxDB, recognized as the most widely adopted TSDB system [23], is used in version 2.4 for this study. In addition, the IoTBenchmark tool is employed [20] to generate workloads and collect performance metrics to compare InfluxDB without and with integrating the proposed rate limiter, TSRL.

This section presents the experimental methodology for examining how various factors influence system performance. The next section presents the experimental results.

5.1. Test Machine Specifications

The experiments were conducted on a machine with the following specifications:

- Operating System: Ubuntu 22.04 LTS
- RAM: 16 GB DDR5-4800 MHz
- Processor: 13th Gen Intel Core i7-13650HX (14 cores, 24 MB cache) (2.60 GHz)
- Storage:
  - 512 GB M.2 PCIe NVMe SSD
  - 1 TB M.2 PCIe NVMe SSD

During each experiment, no additional applications were running except for the terminal used to execute the benchmark and the systems under evaluation (InfluxDB and the TSRL). The TSRL process was re-initialized for each run.

5.2. Benchmarking Tool

The IoTBenchmark tool offers built-in support for InfluxDB, provides extensive data ingestion configuration options, and tracks various performance metrics. Moreover, its extensibility allows for the integration of custom components, such as the TSRL.

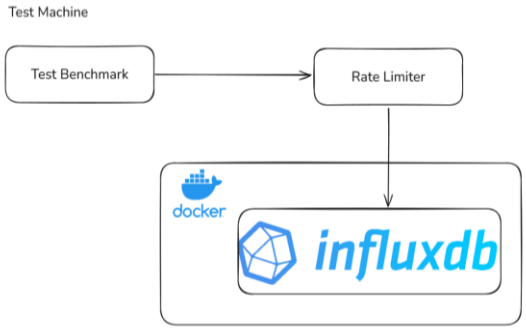


Figure 5. Test machine setup.

The benchmark simplifies database performance testing by enabling test execution and result collection through straightforward commands. It is open-source, written in Java, and supports periodic time-series data generation based on user-defined settings, along with capabilities for direct data insertion and querying.

5.3. Test Setup

0 illustrates the test environment, where the IoTBenchmark tool, the TSRL, and InfluxDB are all executed on a single machine. InfluxDB runs inside a Docker container to emulate the resource constraints of an edge device. The container is allocated 2 GB of memory and 50 GB of disk storage. Similarly, the TSRL is also assigned 2 GB of memory.

This memory allocation reflects typical hardware configurations expected in edge deployments. Since the proposed solution is designed to operate on commodity hardware, a 2 GB memory limit was deemed sufficient for evaluating its feasibility in resource-limited environments.

5.4. Test Workload

Two workload configurations were used: small and large. The small workload includes 5 billion data points, while the large one contains 12 billion data points. Additionally, each experiment defines a data distribution strategy, with the default being an even distribution across all types. The various data distribution patterns used are illustrated in Figure 6.

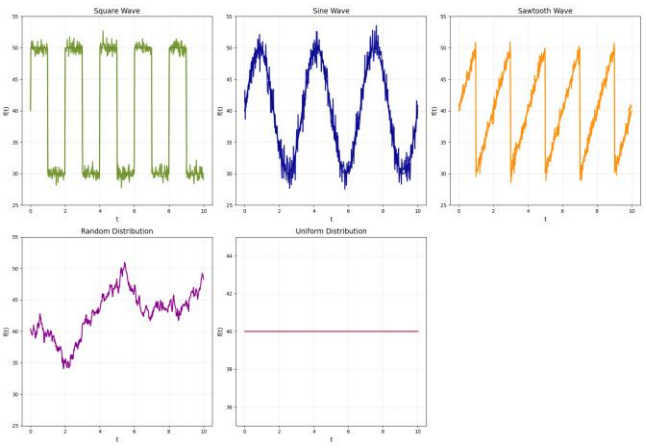


Figure 6. Time series data distributions.

Table 1. Factors Considered In 2^K design.

No.	Factor	Description	Values
A	Workers Count	The number of threads the TSRL spawns to serve the workload	1 and 100
B	R_BATCH	The maximum number of requests the Request Buffer thread can fetch from the Request Pool	1 and 1000
C	L_BATCH	The maximum number of requests the Limiter thread can fetch from the Request Buffer thread	1 and 1000
D	Rate Limit	The maximum number of requests the system should handle in 5 minutes window	256 MB and 1 TB
E	BUFFER_SLEEP_TIME	The period in milliseconds the Request Buffer threads waits before pulling the requests from the Request Pool	10 and 1000 milliseconds
F	LIMITER_SLEEP_TIME	The period in milliseconds the Limiter waits after sending requests to the destination DB	10 and 1000 milliseconds

5.5. Factors and Metrics

Table 1 outlines the factors were configured for both the IoTBenchmark and the TSRL, employing a 2^K factorial design to analyze their effects. We repeated these experiments across various data distributions generated by the benchmark.

Factor A in Table 1 represents the number of workers, which directly controls the level of multi-threading in our setup. Additionally, we considered the impact of the memory allocated to the InfluxDB Docker container.

The primary metrics (dependent variables) that are measured, all sourced from the IoTBenchmark, include: OKPoints (The count of time-series points successfully ingested); Throughput; Median latency; 90th percentile latency.

5.6. Experiments Phases

The experiments are conducted in two phases. In phase 1, the 2^K factorial design is used to identify significant factors. In phase 2, the performance comparison between InfluxDB with and without a TSRL is considered.

5.6.1. Phase 1: Significant Factors

A 2^K factorial design was used to quantify the effects of the factors to the metrics under study. The 2^K design involves conducting 2^K experiments. Each factor is experimented with 2 values (low and high). The effect of each factor is analyzed using statistical methods such that the total effect of all factors sums to 100%. The workload used is the small workload and the data distribution is the default.

5.6.2. Phase 2: InfluxDB Performance

To assess how the TSRL impacts InfluxDB, we ran a series of experiments using diverse workloads and configurations. First, we tested a large ingestion workload on both setups to measure the increase in successfully ingested data points (OKPoints). Next, we executed a mixed workload-combining data ingestion with various query types-to confirm that query latencies didn't worsen. Finally, we analyzed InfluxDB's memory usage, monitoring consumption within its Docker container using Prometheus and coAdvisor [14].

6. Experimental Results

This section discusses the experimental results corresponding to the two evaluation phases previously introduced. In Phase 1, the impact of key system configuration parameters is analyzed, demonstrating how factors like worker count and sleep time settings directly influence system performance. In Phase 2, a comprehensive performance evaluation of InfluxDB under various workloads and memory allocations is provided, comparing its efficiency and resource consumption with and without the integration of the proposed TSRL.

6.1. Phase 1: Significant Factors

The worker count is the most significant factor, directly influencing the level of multithreading. Generally, more workers lead to higher throughput. However, increasing the worker count beyond the host machine's physical CPU cores will not provide any significant further throughput gains. In addition, increasing the number of workers directly impacts latency by reducing queueing, which in turn lowers the overall request latency.

The sleep time settings for both the BUFFER and LIMITER had a noticeable impact on throughput. This is due to the factors controlling how idle the system is. Essentially, longer sleep times result in lower throughput. Similarly, these sleep times also affect latency; the more time the system spends idle, the longer requests remain within the system before being processed.

Figure 7 illustrates how increasing the worker count impacts ingestion throughput. Figure 8 demonstrates the effect of increasing LIMITER_SLEEP_TIME (Factor F) on ingestion throughput, clearly showing a decline as this value rises. As a logical consequence, Figure 9 reveals that increasing this same fact or significantly increases the median latency of ingestion requests.

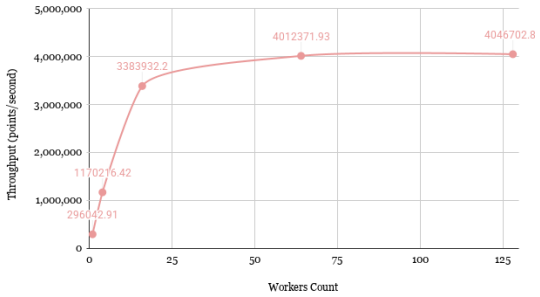


Figure 7. Workers count vs throughput.

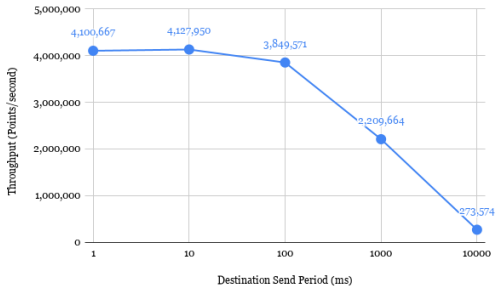


Figure 8. LIMITER_SLEEP_TIME vs throughput.

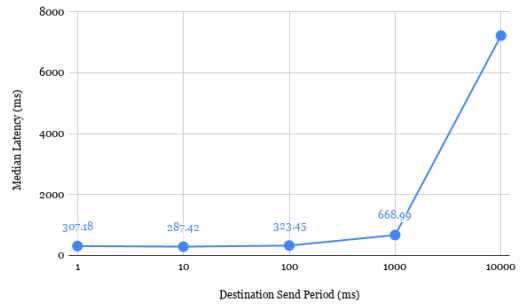


Figure 9. LIMITER_SLEEP_TIME vs median latency.

6.2. Phase 2: InfluxDB Performance

Comparing InfluxDB performance with and without a TSRL, Figure 10 demonstrates an improvement across all performance metrics: OKPoints, Throughput, median latency, P90 latency, and peak memory. Specifically, OKPoints increased by 26% and Throughput improved by approximately 20%. Meanwhile, both median and P90 latencies decreased by about 10% and 5%, respectively.

In addition, with a TSRL, InfluxDB's peak memory usage is significantly reduced by approximately 50% (peaking at 3.8 GB compared to 7.5 GB without it).

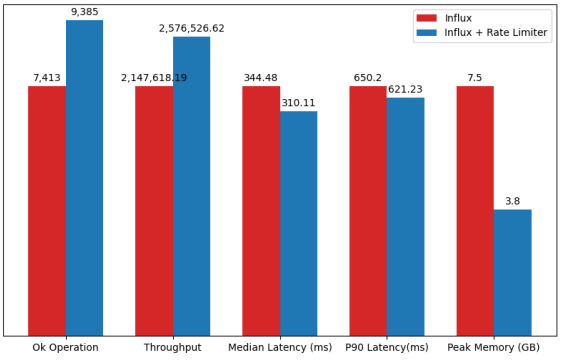


Figure 10. InfluxDB performance without and with a TSRL.

Because the memory allocated to InfluxDB significantly influences its capacity to handle ingestion requests, a series of experiments were conducted using different memory sizes, both without and with a TSRL. These experiments aimed to evaluate two aspects: the overall impact of memory size on the performance metrics and the effectiveness of the proposed solution under varying memory configurations.

Figures 11 and 12 illustrate the significant scalability pattern of increasing allocated memory, especially when a TSRL is in place.

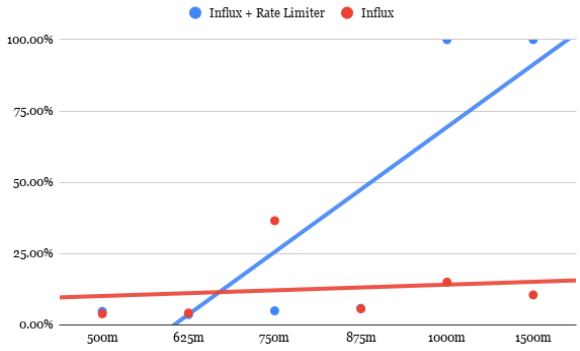


Figure 11. Memory vs success rate.

Figure 11 demonstrates that a larger memory allocation directly correlates with a higher success rate for ingestion requests in both testing configurations. However, the TSRL notably amplifies this positive effect, as indicated by a steeper trend line. This suggests that the TSRL optimizes InfluxDB's memory utilization, leading to improved success rates. The absence of request spikes, enforced by the TSRL's policy, likely contributes to this.

Similarly, Figure 12 shows that increasing Influx DB's allocated memory boosts throughput in both setups. Consistent with the previous observation, the use of TSRL leads to a more substantial increase in throughput as memory grows. This outcome is explainable, as more successfully ingested points directly translate to higher overall throughput.

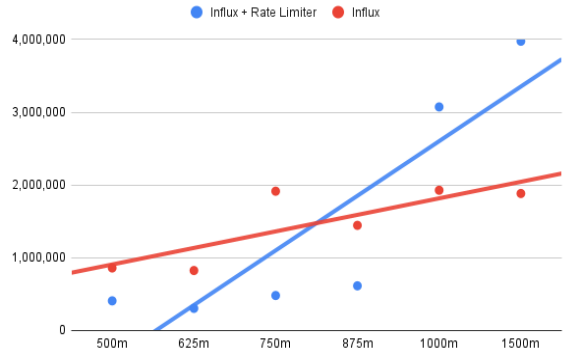


Figure 12. Memory vs throughput.

Figures 13 and 14 highlight the significant performance improvements observed when a TSRL is introduced, particularly under heavy client loads.

Figure 13 demonstrates that with 100 concurrent clients, the TSRL dramatically improved the OKPoints by 259% and throughput by 145.87%. This substantial improvement is attributed to the TSRL's ability to reduce and smooth the traffic sent to InfluxDB, preventing overwhelming bursts of requests. By buffering and rate-limiting incoming traffic, the TSRL effectively eased the concurrent pressure on the database.

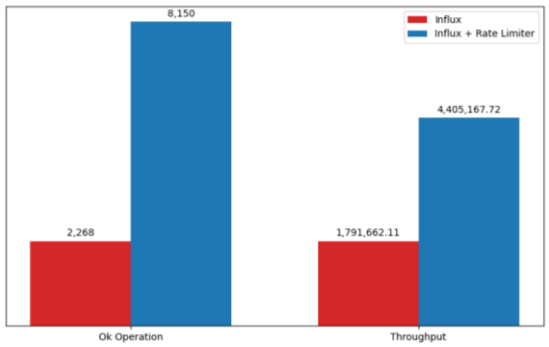


Figure 13. OkPoints and Throughput when the number of clients=100.

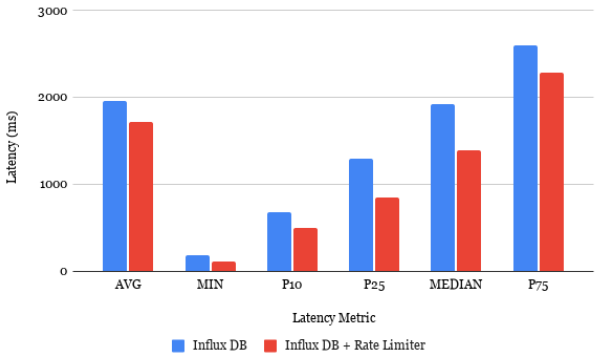


Figure 14. Latencies when the number of clients=100.

Figure 14 illustrates the TSRL's impact on latency metrics, showing a noticeable but inconsistent enhancement across different percentiles. While the minimum and 25th percentile latencies saw considerable improvement, the average, median, and higher percentiles experienced less enhancement, or in some cases, an increase. This phenomenon is due to the TSRL's mechanism of holding back requests when limits are reached or initiating retries upon failure introduces a deliberate delay, which becomes more pronounced in the higher percentiles and tail latencies. Essentially, while it protects the system and boosts overall success and throughput, this protective buffering can introduce a slight increase in the worst-case (tail) response times.

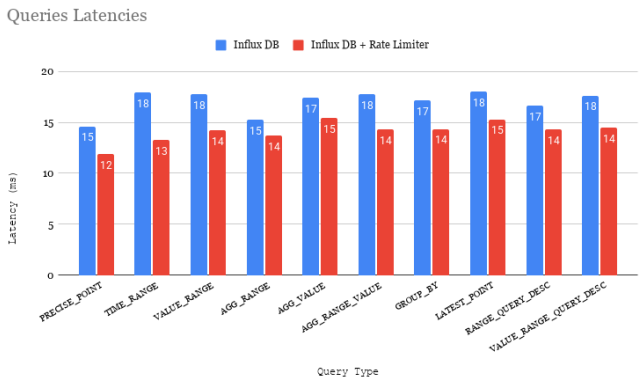


Figure 15. Queries latencies.

Lastly, Figure 15 shows that the queries' latencies were enhanced although the target workload was the ingestion. This might be rooted in the fact that InfluxDB was relieved of some pressure by the use of a TSRL and was able to respond faster to query workloads.

7. Conclusions and Future Extensions

This work introduces an extra layer to be inserted between a TSDB and clients to increase data integrity and decrease the request loss rate during an increase in data ingestion requests. The proposed solution utilized well-known resilience constructs, rate limiting, buffering, and retries, to increase the probability of successful insertion of time-series data points. The gain in loss rate compared to the increase in tail latency may be adequate for the use cases where data integrity is more important than time-dependent non-functional requirements.

We conducted a study on the system's key factors to help users tune its performance for different data distributions. We also ran experiments to compare a TSDB with and without the proposed TSRL.

The results show that the TSRL enhances the TSDB's scalability, especially with more allocated memory and a higher number of concurrent clients. It also leads to a higher success rate for ingestion requests.

Nevertheless, the reliability of in-memory has some drawbacks. The memory allocated by the rate limiter process may not be enough to handle the incoming load. Extending the in-memory buffering to disk whenever the memory is full is considered future work. Compressing the buffered requests could also prevent the rate limiter from running out of memory. Testing the rate limiter horizontal scalability by deploying more than one instance with a load balancer could also be investigated. The implementation of a distributed architecture to summarize data on the edge [17] could also be explored.

References

[1] Abendroth D., Eckel M., and Killat U., "Solving the Trade-Off Between Fairness and Throughput: Token Bucket and Leaky Bucket-Based Weighted Fair Queueing Schedulers," *International Journal of Electronics and Communications*, vol. 60, no. 5, pp. 404-407, 2006. https://doi.org/10.1016/j.aeue.2005.04.002

[2] Ahmad K. and Ansari M., *Hands-on InfluxDB, NoSQL*, Chapman and Hall/CRC, 2017.

[3] Alam T., "Cloud-Based IoT Applications and Their Roles in Smart Cities," *Smart Cities*, vol. 4, no. 3, pp. 1196-1219, 2021. DOI: 10.3390/smartcities4030064

[4] Bader A., Kopp O., and Falkenthal M., "Survey and Comparison of Open Source Time Series Databases," *in Proceedings of the Datenbanksysteme für Business, Technologie und Web Workshopband*, Bonn, pp. 249-268, 2017. https://dl.gi.de/items/1d3c3186-8d50-4b51-b097-7695027076fd

[5] Carvalho G., Cabral B., Pereira V., and Bernardino J., "Edge Computing: Current Trends, Research Challenges and Future Directions," *Computing*, vol. 103, pp. 993-1023, 2021. DOI:10.1007/s00607-020-00896-5

[6] Chalmers K. and Pedersen J., "Analysing Concurrent Queues Using CSP: Examining Java's ConcurrentLinkedQueue," *Software*, vol. 4, no. 3, pp.15, 2025. https://doi.org/10.3390/software4030015

[7] Dhanagari M., "Scaling with MongoDB: Solutions for Handling Big Data in Real-Time," *Journal of Computer Science and Technology Studies*, vol. 6, no. 5, pp. 246-264, 2024. DOI: 10.32996/jcsts.2024.6.5.20

[8] El-Naggar A., Ghanem S., and Nagi M., "Edge-Level Rate Limiting in Influx DB: Enhancing Performance and Scalability," *in Proceedings of the International Conference on Computer and Communication Systems*, Chengdu, pp. 191-196, 2025. https://ieeexplore.ieee.org/document/11069899

[9] Google Cloud, Compute Engine Machine Families, https://cloud.google.com/compute/docs/machine-resource, Last Visited, 2025.

[10] Google, Firebase Realtime Database limits, https://firebase.google.com/docs/database/usage/limits, Last Visited, 2025.

[11] Gubbi J., Buyya R., Marusic S., and Palaniswami M., "Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645-1660, 2013. DOI: 10.1016/j.future.2013.01.010

[12] Hadjichristofi C., Diochnos S., Andresakis K., and Vescoukis V., "Using Time-Series Databases for Energy Data Infrastructures," *Energies*, vol. 17, no. 21, pp. 1-23, 2004. https://doi.org/10.3390/en17215478

[13] Hao Y., Qin X., Chen Y., Li Y., and et al., "TS-Benchmark: A Benchmark for Time Series Databases," *IEEE 37th International Conference on Data Engineering*, Chania, pp. 588-599, 2021. https://doi.org/10.1109/ICDE51399.2021.00057

[14] Holopainen M., Monitoring Container Environment with Prometheus and Grafana, Bachelor's Thesis, Metropolia University of Applied Sciences, 2021. https://www.theseus.fi/bitstream/handle/10024/497467/Holopainen_Matti.pdf

[15] Hou Z., Huang Q., Song Y., Fan S., and Nie X., "A Memory Optimization Method for High Cardinality Based on InfluxDB," *in Proceedings of the 7th International Conference on Big Data Technologies*, New York, pp. 51-57, 2024. https://doi.org/10.1145/3698300.3698312

[16] InfluxData, InfluxDB Cloud Pricing and Plans, https://www.influxdata.com/pricing/, Last Visited, 2025.

[17] Ishtaiwi A., Nabot A., Alzubi O., Ramadan A., and et al., "HECOCP: Hybrid Edge-Cloud Optimistic Concurrency Protocol for Sensor Data Transactional Services," *The International Arab Journal of Information Technology*, vol. 22, no. 4, pp. 756-768, 2025. https://doi.org/10.34028/iajit/22/4/10

[18] Jayakumar H., Lee K., Lee W., Raha A., and et al., "Powering the Internet of Things," *in Proceedings of the International Symposium on Low Power Electronics and Design*, La Jolla, pp. 375-380, 2014. https://doi.org/10.1145/2627369.2631644

[19] Karkouch A., Mousannif H., Moatassime H., and Noel T., "Data Quality in Internet of Things: A State-of-the-Art Survey," *Journal of Network and Computer Applications*, vol. 73, pp. 57-81, 2016. https://doi.org/10.1016/j.jnca.2016.08.002

[20] Liu R., Yuan J., and Huang X., "Benchmark Time Series Database with IoTDB-Benchmark for IoT Scenarios," *CoRR*, vol. abs/1901.08304, 2019. http://arxiv.org/abs/1901.08304

[21] McBride B. and Reynolds D., *Survey of Time Series Database Technology*, UK Centre for Ecology and Hydrology, Wallingford, 2020. https://nora.nerc.ac.uk/id/eprint/527832/

[22] Namiot D., "Time Series Databases," *in Proceedings of the 17th International Conference Data Analytics and Management in Data Intensive Domains*, Obninsk, pp. 132-137, 2015. file:///C:/Users/user/Downloads/paper20.pdf

[23] Nasar M. Abu Kausar M., "Suitability of Influxdb Database for IoT Applications," *International Journal of Innovative Technology and Exploring Engineering*, vol. 8, no. 10, pp. 1850-1857, 2019. DOI: 10.35940/ijitee.J9225.0881019

[24] Nawab I. and Kulkarni S., "Toward Next-Generation Distributed Rate-Limiters," *in Proceedings of the IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops*, Bangalore, pp. 354-356, 2023. DOI: 10.1109/CCGridW59191.2023.00083

[25] Ng M. and Huang Z., "Data-Mining Massive Time Series Astronomical Data: Challenges, Problems and Solutions," *Information and Software Technology*, vol. 41, no. 9, pp. 545-556, 1999. https://doi.org/10.1016/S0950-5849(99)00018-X

[26] Niswar M., Safruddin R., Bustamin A., and Aswad I., "Performance Evaluation of Microservices Communication with REST, GraphQL, and gRPC," *International Journal of Electronics and Telecommunication*, vol. 70, no. 2, pp. 429-436, 2024. DOI: 10.24425/ijet.2024.149562

[27] Peter O., Pradhan A., and Mbohwa C., "Industrial Internet of Things (IIoT): Opportunities, Challenges, and Requirements in Manufacturing

Businesses in Emerging Economies," *Procedia Computer Science*, vol. 217, pp. 856-865, 2023. https://doi.org/10.1016/j.procs.2022.12.282

[28] Shah B., Jat P., and Sashidhar K., "Performance Study of Time Series Databases," *arXiv Preprint*, vol. arXiv:2208.13982v1, pp. 1-14, 2022. https://arxiv.org/abs/2208.13982

[29] Tan L., Dong X., Tu N., and Hong J., "IntDB: Towards a Spatiotemporal Database for In-Band Network Telemetry," *in Proceedings of the 25th Asia-Pacific Network Operations and Management Symposium Kaohsiung*, Taiwan, pp. 1-6, 2025. DOI: 10.23919/APNOMS67058.2025.11181327

[30] Tripathi P., Miraz M., and Joshi S., "Comparative Analysis of MongoDB and InfluxDB for Time Series Data Management in IoT Environments: A Study on Performance, Scalability, and Concurrency," *in Proceedings of the International Conference on Computing, Networking, Telecommunications and Engineering Sciences Applications*, Zagreb, pp. 39-42, 2023. https://ieeexplore.ieee.org/document/10384962

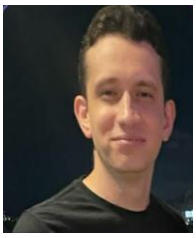
[31] Wang C., Qiao J., Huang X., Song S., and et al., "Apache IoTDB: A Time Series Database for Large Scale IoT Applications," *ACM Transactions on Database Systems*, vol. 50, no. 2, pp. 1-45, 2025. https://doi.org/10.1145/3726523

[32] Wang S., Sun Z., Hu C., Li C., and et al., "MatrixGate: A High-performance Data Ingestion Tool for Time-Series Databases," *arXiv Preprint*, vol. arXiv:2406.05462, pp. 1-13, 2024. https://arxiv.org/html/2406.05462v1

[33] Wu H., Shang Z., Peng G., and Wolter K., "A Reactive Batching Strategy of Apache Kafka for Reliable Stream Processing in Real-Time," *in Proceedings of the IEEE 31st International Symposium on Software Reliability Engineering*, Coimbra, pp. 207-217, 2020. DOI: 10.1109/ISSRE5003.2020.00028

[34] Zhang L., Jeong D., and Lee S., "Data Quality Management in the Internet of Things," *Sensors*, vol. 21, no. 17, pp. 5834, 2021. https://doi.org/10.3390/s21175834

[35] Zhu X., Nie X., and Liu J., "Time Series Database Optimization Based on InfluxDB," *in Proceedings of the International Conference on Power, Electrical Engineering, Electronics and Control*, Athens, pp. 879-885, 2023. https://doi.org/10.1109/PEEEC60561.2023.00172



Ahmed El-Naggar received her BSc and MSc in Computer Science from Faculty of Engineering in Alexandria University, Egypt, in 2019 and 2025, respectively. He is currently an assistant teacher at theFaculty of Engineering in Alexandria, Egypt. His research interests are database internals, systems, research, and the use of LLMs in data science.

Magdy Nagi received his BSc from the Faculty of Engineering Alexandria University Egypt in 1963. He received his PhD from Karlsruhe University, Karlsruhe, Germany in 1975. He is currently a Professor at the Faculty of Engineering Alexandria University. His main research interests are software engineering, DBMS, data mining and grid computing. He has published more than 35 scientific publications in international journals and conference proceedings.

Sahar Ghanem received her BSc and MSc in Computer Science from Faculty of Engineering in Alexandria University, Egypt, in 1994 and 1997, respectively, and PhD in Computer Science from Old Dominion University in VA, USA, in 2004. She is currently an Associate Professor at the Faculty of Engineering in Alexandria, Egypt. Her main research interests are computer networks, network security, performance evaluation and data mining. She has published about 20 scientific publications in international journals and conference proceedings.