Priority-Driven Differentiated Performance for NoSQL Database-as-a-Service

Remo Andreoli, Tommaso Cucinotta, and Daniel Bristot De Oliveira

***Abstract*—Designing data stores for native Cloud Computing services brings a number of challenges, especially if the Cloud Provider wants to offer database services capable of controlling the response time for specific customers. These requests may come from heterogeneous data-driven applications with conflicting responsiveness requirements. For instance, a batch processing workload does not require the same level of responsiveness as a time-sensitive one. Their coexistence may interfere with the responsiveness of the time-sensitive workload, such as online video gaming, virtual reality, and cloud-based machine learning. This article presents a modification to the popular MongoDB NoSQL database to enable differentiated per-user/request performance on a priority basis by leveraging CPU scheduling and synchronization mechanisms available within the Operating System. This is achieved with minimally invasive changes to the source code and without affecting the performance and behavior of the database when the new feature is not in use. The proposed extension has been integrated with the access-control model of MongoDB for secure and controlled access to the new capability. Extensive experimentation with realistic workloads demonstrates how the proposed solution is able to reduce the response times for high-priority users/requests, with respect to lower-priority ones, in scenarios with mixed-priority clients accessing the data store.**

***Index Terms*—Cloud computing, differentiated performance, NoSQL, cloud storage, MongoDB.**

I. Introduction

OVER the past decade, Cloud Computing proved to be a cost-effective paradigm for businesses looking for ways to ease the development, deployment, monitoring, and operations of monolithic or distributed applications featuring continuous availability and reliability. In this regard, the Infrastructure-as-a-Service (IaaS), Software-as-a-Service (SaaS), and Platform-as-a-Service (PaaS) cloud paradigms allow customers to avail of a number of services with on-demand capabilities, avoiding the capital investment and maintenance burden of on-premise infrastructures [1]. One of the key advantages of the cloud computing paradigm is the decoupling between *providers*, who manage the physical infrastructure and offer cloud services fully managed on a 24/7 basis, and *customers* (or *tenants*), who use them. Cloud-native software [2] makes use of many techniques to efficiently operate at a global scale on multiple geo-distributed, fault-independent sites, such as data replication and sharding, horizontal scalability, load balancing, and efficient orchestration of virtual machines and containers.

At the heart of the cloud-native distributed software ecosystem, there are storage services, which must address the stringent performance, reliability, and scalability requirements of today's database-driven web applications. These often perform massive big-data processing, as in extreme-scale simulations [3] or application scenarios making use of machine learning and artificial intelligence. This has led to a new generation of data stores, named NoSQL to highlight the difference compared to the relational counterpart, which employs more relaxed design choices, departing from traditional ACID guarantees to embrace schema-less and weak-consistency data models and simpler APIs, gaining in efficiency and scalability.

Since the success of a cloud-based application is directly correlated to the quality of the user experience, the so-called Quality-of-Service (QoS), it is fundamental for a cloud provider (CP) to consistently provide performance levels as expected by its customers, whether implicit or formally defined in an in-place Service Level Agreement (SLA). A well-known [4], [5], [6] performance-related challenge in Cloud Computing is to provide predictable response times for time-sensitive, low-latency, and interactive applications, such as video streaming, online gaming [7], social networks in virtual and augmented reality [8] and Internet-of-Things (IoT) network systems, which all intrinsically possess tight responsiveness requirements.

In this context, a CP should expose managed virtualized infrastructure elements (e.g., virtual machines) and cloud services (e.g., storage solutions) that exhibit stable and predictable performance to build reliable applications that satisfy the timing constraints necessary for high-quality interactivity with minimum degradation [4], [9]. However, multi-tenant architectures are well-known to suffer from the "noisy neighbor" effect. This can be tackled by deploying dedicated hardware, but it is quite expensive, whilst normally CPs save on operational costs and energy consumption by applying resource consolidation techniques [10], [11], [12] to maximize the infrastructure utilization and help running a sustainable and cost-effective service. For instance, multiple VMs can be hosted on the same physical machine, or data from multiple independent tenants may be stored within a single scalable data store, to reduce the overall software footprint. As a practical example, DynamoDB [13] is

Manuscript received 12 August 2022; revised 19 May 2023; accepted 24 June 2023. Date of publication 4 July 2023; date of current version 6 December 2023. Recommended for acceptance by M. Singhal. *(Corresponding author: Remo Andreoli.)*

Remo Andreoli and Tommaso Cucinotta are with the Institute of Communication, Information and Photonics Technologies, Scuola Superiore Sant'Anna, 56127 Pisa, Italy (e-mail: remo.andreoli@santannapisa.it; tommaso.cucinotta@santannapisa.it).

Daniel Bristot De Oliveira is with RedHat Inc., 56127 Pisa, Italy (e-mail: bristot@redhat.com).

Digital Object Identifier 10.1109/TCC.2023.3292031

© 2023 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

an infinitely scalable and reliable fully-managed NoSQL service with performance guarantees that stores data from different customers on the same physical machines to ensure higher utilization, and therefore saving on costs. Clearly, this allows for heterogeneous workload patterns with competing requirements, such as a combination of heavyweight requests for batch or high-performance applications, which generally need to process high volumes of data at maximum throughput, and lightweight requests coming from time-sensitive applications, which generally need to process a small amount of data with tight response times. This turns into a risk of unstable performance for multi-tenant cloud services, therefore a cloud platform may not be able to meet pre-specified temporal requirements without a mechanism to reduce and/or control the interferences [14] among the different co-located workloads. The problem is non-trivial and does not revolve around maximizing overall throughput [4], as with general-purpose applications: although the response time for an activity tends to decrease as the overall throughput increases, the trade-off becomes evident when the cloud service has to balance the need for maximizing throughput in ongoing batch requests, with the urgent need to serve time-sensitive requests. The more resources are dedicated to the latter request types, the larger is the impact on the throughput; conversely, the fewer resources time-sensitive activities are granted, the longer it takes for them to complete, while the overall system throughput grows.

A. Contributions

This paper tackles the above-mentioned performance challenge in the domain of storage services, addressing the need of designing evolved fully-managed NoSQL data stores that support highly heterogeneous workloads. We present a modification to the popular open-source MongoDB data store that extends it, adding the ability to differentiate response times on a per-request or per-client basis, according to a simple mechanism that enforces a priority-driven request ordering. This is achieved by exploiting a combination of features and locking primitives within the Operating System (OS). The final result is a NoSQL data store that enables *higher*-priority users to pre-empt or postpone access to the data store for *lower*-priority users, for the time needed by higher priority requests to be served. The proposed solution acts as a "building block" towards fully-managed storage services with predictable performance (if coupled with a real-time admission control mechanism) or for use-cases where there is a need to *differentiate* between multiple performance provisioning offers. A CP hosting our modified version of MongoDB can offer a subscription-based model with different fees in which *high*-priority, "gold" users are served before *low*-priority, "bronze" users. For instance, a tenant requesting data access with high responsiveness will be assigned the *high*-priority status.

This paper follows up on, and provides significant extensions to, our prior research [15], [16]. Compared to that, in this work we present improvements to the internal design of the prioritization mechanism, measuring the additional computational overhead of the proposal. Moreover, each component of the proposal is coupled with pseudocode describing the implementation logic. We present a workaround for an "unsafe" instance of busy-waiting in MongoDB, which went unnoticed in our previous papers due to the experimentation setup. This caused very high response time spikes in many-core, many-clients scenarios. We integrated the proposal within the MongoDB access-control model; in this way, database administrators can restrict the use of the differentiated performance feature to a subset of "privileged" users. We provide new experimental results with more realistic, many-clients interference scenarios using the well-known Yahoo! Cloud Serving Benchmark (YCSB) framework [17]. Finally, we provide a more detailed and up-to-date discussion of related research.

B. Paper Overview

The rest of this paper is organized as follows: Section II introduces background concepts about MongoDB and CPU scheduling in Linux, which are useful for a better understanding of what follows; Section III describes how the proposed modifications have been integrated into the internals of MongoDB; Section IV provides experimental data from executions of the modified MongoDB, demonstrating the effectiveness of the proposal and highlighting what trade-offs between throughput and individual response times are achievable. Section V briefly discusses related industrial and academic works. Finally, Section VI provides concluding remarks, and it addresses possible directions for future research on the topic.

II. Background

This section introduces basic concepts about MongoDB and its internals, and some details on the default scheduler within the Linux kernel. The goal of this section is to provide the necessary background information to understand the reasoning behind our proposed modifications to MongoDB, which are described in Section III.

A. MongoDB Overview

MongoDB[1] is an open-source, document-oriented NoSQL data store [18]. In what follows, the discussion refers to version 4.4.[2] Data is stored in the form of *document collections*, which are analogous to tables in relational databases, but since MongoDB is schema-less, a collection can accommodate heterogeneous documents with different attributes. Documents are defined using the JSON format, and they are transmitted and stored in a binary-encoded serialized format for efficiency in storage space and scan speed. A user interacts with a MongoDB system using libraries called drivers, available for various programming languages, that implement an application-programming interface (API) exposing a JSON-based query language. The simplest MongoDB deployment, which does not allow for redundancy nor scalability, is called a *standalone* deployment and comprises a single instance of the *mongod* daemon. This is a multi-threaded C++ program implementing

[1]See: https://www.mongodb.com/

[2]See: https://github.com/mongodb/mongo/tree/v4.4

the core database activities: management of user connections, query planning and all the "background" activities, such as replication, monitoring and throttling. MongoDB supports data redundancy by deploying a *replica set*, a group of *mongod* instances residing on different physical machines that store replicated versions of the same data set: the so-called *primary node* receives all the write requests and tracks the changes to the data set, whereas the *secondary nodes* replicate the state of the primary to reflect the changes. Read operations are usually also directed to the primary node, but it is possible to tweak availability by changing the *read preference* request parameter, at the cost of sacrificing full consistency and data freshness, because a secondary node may return stale data. A replicated MongoDB deployment allows leveraging data durability and throughput on a per-request or per-client basis through the *write concern* and *read concern* options, which respectively define the number of replica set members that must acknowledge a write operation before returning a positive response to the client, and the number of replica nodes that have acknowledged and persisted the data requested by a read operation: high values lead to high reliability but poor system throughput, whereas low values lead to quicker responses from the database but also to a higher risk of data loss in case of hardware failures. The combination of the two concern options enables different causal consistency guarantees [19]: for instance, the write concern value to guarantee that a write operation on a given document is completed before any successive write operation (i.e., monotonic write) is $\lfloor \frac{\#nodes}{2} \rfloor + 1$, the *majority* of the replica set.

MongoDB supports also data *sharding*, which allows partitioning a document collection into smaller fragments (shards) distributed across a cluster of many machines according to user-defined criteria. The resulting *sharded cluster* requires a set of *mongod* instances handling the individual shards, possibly arranged in replica sets. A *sharded cluster* interfaces with the clients through a router component called *mongos*, which behaves like a regular *mongod* instance, but it uses internally a local shard metadata database to look up which *mongod* instances can actually handle each request.

B. MongoDB Internals

There are mainly two technical points regarding the internal architecture of MongoDB that interest the present proposal: how concurrent user connections are handled and how the replication internal activities are carried out. MongoDB uses a client-server architecture: a client communicates with the database server (i.e., a stand-alone *mongod* instance, a replica set member, or a *mongos* instance) using a simple socket-based, request-response protocol called *MongoDB Wire Protocol*. A client, which could be either a secondary node fetching data changes or a user application performing a database operation, establishes a connection with the database using an IP address and a port. For the sake of simplicity, the term *external* client refers to a user connection, whereas *internal* client refers to a replica member connection.

The server manages each individual connection context (the so-called session) synchronously using a thread pool: a unique *client worker thread* is reserved for every connection to handle

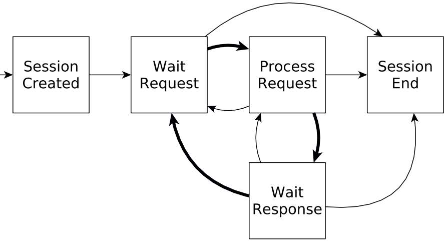


Fig. 1. The FSM modeling the session life-cycle of a client connection.

the server-side activities that determine the life-cycle of the interaction. Therefore, the underlying worker thread does not change for the duration of the client session. Fig. 1 presents the workflow of interaction as a finite-state machine (FSM), where each state corresponds to a set of activities performed by the underlying client worker thread. First, a client thread creates the context of the connection (*Session-Created* state), then it waits for incoming messages from the client (*Wait-Request* state). Upon receiving a message, the worker thread processes the operation enveloped into the request message (*Process-Request* state) and handles possible interactions with other *mongod instances*, waiting for their responses (*Wait-Response* state), e.g., in case of replication with write concern higher than 1. Then, the client thread waits for another request from the client. Eventually, the connection to the database is terminated, or a failure occurs, thus causing the session to end (*Session-End* state). The underlying worker thread then performs clean-up operations, becoming ready to be destroyed, or being reused for another client connection, depending on the server configuration. The bold path depicted in Fig. 1 corresponds to the so-called *standard transition path*, which models the traditional client-server interaction: wait for a client request, process it, and send a response back.

MongoDB is capable of achieving high throughput for concurrent read and write operations thanks to the WiredTiger[3] storage engine. This implements an optimistic version of the classic Multi-Version Concurrency Control (MVCC) mechanism [20] at document-level, allowing for multiple write operations to different documents to occur at the same time. The term "optimistic" refers to how write conflicts are handled: a write conflict occurs whenever simultaneous update operations affect the same document, and it is resolved by accepting only a single operation as valid, while transparently retrying the others. Data consistency is enforced by periodic point-in-time snapshots of the data to present a consistent view to the clients. Therefore, new changes are visible for read operations only after all write conflicts are resolved and a new snapshot of the data set is taken.

C. Replication Internals

Regarding the internals of the replication mechanism, the primary node keeps track of the changes to the data set in the *operation log* (*oplog* in short), a fixed-size MongoDB collection. Changes to an oplog entry are described in an idempotent

[3]See: https://source.wiredtiger.com/

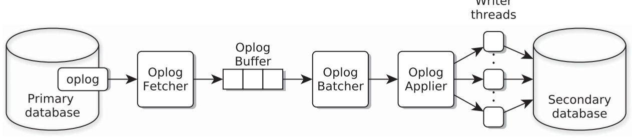


Fig. 2. The workflow of the replication process from the secondary node point of view: fetch oplog entries, batch them, apply in parallel to the local copy of the data set.

format: each oplog operation produces the same results whether it is applied to the given dataset once or multiple times. In order to enforce data consistency among replica members, each oplog entry is paired with a timestamp, which is also used by WiredTiger to return the correct view of the data to a read operation. A secondary node periodically connects to a source node, which could be either the primary node or an up-to-date secondary node, then copies and applies asynchronously these operations in order to reflect the changes to its local copy of the data set. More specifically, the replication process is performed by the following components, depicted in Fig. 2: the *OplogFetcher* establishes the connection to the primary node, retrieves the oplog entries in several batches (i.e., multiple runs of the *standard transition path*) and stores them in the *OplogBuffer*; the *OplogBatcher* pulls the fetched entries from the buffer and creates the next batch to be applied to the local data set; lastly, the *OplogApplier* distributes the newly created batch to a pool of parallel writer threads that apply them. As the chronological order of oplog entries within a batch cannot be controlled, some database operations like removing a collection require a singleton batch. After the batch has been applied, the secondary node notifies the primary, which in turn sends a successful response to the pending user operations that took place *before* the timestamp of the last applied oplog entry, if the number of received notifications matches with the write concern requirement.

D. Linux Scheduler / POSIX Niceness

A multi-tasking OS has to serve multiple concurrently running threads or processes, often generically referred to as tasks [21], assigning them time slices over the available CPU(s). The component responsible for granting CPU time to the tasks is the scheduler, which chooses the execution order depending on the scheduling policy and per-task scheduling parameters. For instance, the Linux kernel provides a framework that comprises three scheduling policies, each suitable for specific use cases: fair scheduling for general-purpose applications, fixed-priority, and reservation/deadline-based scheduling for real-time scenarios. The last two categories are deterministic policies used for embedded real-time scenarios where the total real-time workload is known upfront: failing a proper analysis of the requirements causes problems that could compromise the functioning of the entire system. However, there are studies [22] exploring the applicability of these scheduling strategies to deploy highly time-sensitive applications in Cloud infrastructures. Notice that these scheduling policies are associated with a static ordering, so that deadline-based tasks run before, and preempt, any priority-based and general-purpose tasks which determine the tasks to run next. Given the nature of real-time tasks, they are always scheduled before general-purpose tasks.

The default Linux scheduler for general-purpose applications, the *Completely Fair Scheduler* (CFS) [23], provides a weighted-fair partitioning of the time available on each CPU among the ready-to-run tasks in its ready queue. CFS employs UNIX *nice* levels to manipulate the weight associated with a task in the weighted-fair share scheduling algorithm: more specifically, a numerically large nice value increases the willingness of a thread to give precedence to others. The valid range of nice level values is between -20 (highest priority) and 19 (lowest priority), where each nice value increment/decrement corresponds roughly to a relative modification of the task weight of 10%.[4] Negative nice values are usually only available to privileged tasks, but it is possible to make them accessible to unprivileged ones as well, by proper configuration of the permissions in `limits.conf`.[5]

III. Proposed Approach

This section explains how the internal MongoDB architecture design is leveraged to achieve differentiated performance on a per-request/client basis without compromising the correct functioning of the data store, with near zero overhead when the prioritization mechanism is not in use. Our modifications can be summarized in 5 main points, which are further detailed in the subsections that follow:

1) UNIX *nice levels* are exploited to prioritize higher-priority client threads over lower-priority ones;
2) An instance of *busy-waiting* in WiredTiger is modified to avoid priority inversion and starvation of client threads in *mongod*;
3) when replication is used, *request batches are truncated* earlier when containing mixed-priority requests, to avoid a large number of lower-priority requests to slow down the time needed to complete higher-priority ones;
4) a custom *semaphore* allows for containing the volume of lower-priority requests that hit the disk while higher-priority requests are pending;
5) the *access control model* of MongoDB is extended to allow an administrator to configure what users are allowed access to higher-priority requests.

A. Prioritization Using Nice Levels

The proposed mechanism enriches the query language of MongoDB with two new features: a new database command, called `setClientPriority`, which allows a client to change the priority for the current session, and a new option for the `run-Command` general-purpose database command, which allows prioritizing a single request. Both features allow for specifying the new priority level as either *high*, *normal*, or *low*. However,

[4]More information at: https://github.com/torvalds/linux/blob/master/kernel/sched/core.c#L11185.

[5]More information at: https://manpages.debian.org/jessie/libpam-modules/limits.conf.5.en.html.

while the `setClientPriority` command changes priority for all the following requests till the end of the session, the priority declared by a `runCommand` operation persists for a single client-server interaction only (i.e., a single loop involving transitions *Wait-Request* → *Process-Request* → *Wait-Response*, in the FSM in Fig. 1), then the subsequent requests keep being served at the configured session priority.

The main mechanism by which we provide priority-driven differentiated performance in MongoDB is to alter the CPU scheduling settings of client worker threads so that those serving *higher-priority* requests are favored. The current implementation on Linux exploits the *UNIX nice levels* through the *setpriority()* system call,[6] but an analogous Windows-based implementation could leverage the `setThreadPriority()` Win32 API.[7] The priority specified for a request, or a whole client session, is mapped to a precise nice value: the MongoDB command `setClientPriority(high-priority)` has the effect of setting the nice level of the underlying worker thread to the lowest value of -19 (if the client is authorized to do so, see Section III-E); the option *normal-priority* restores the default nice value of 0, and *low-priority* sets it to the highest value of $+20$. Since MongoDB v4.4 reserves a dedicated server-side worker thread for each incoming user connection, and these threads are concurrently running while repeatedly carrying out the actions in Fig. 1, it is clear that prioritizing some of the threads results in giving them a higher chance to run and complete earlier their pending operations when these threads are competing for being scheduled on the same CPU(s). Notice that our usage of nice levels requires a minimum of additional permissions in order to be deployed correctly on Linux. Specifically, the *mongod* server needs to be able to set negative nice values, something that can conveniently be done by configuring in `limits.conf` the allowed niceness range for the OS user through which MongoDB is launched on the server.

B. Modifications to WiredTiger Busy-Waiting

WiredTiger employs busy-waiting in a number of instances where the wait duration for a condition is expected to be shorter than the overhead of context switching and re-scheduling, which is typical of blocking synchronization primitives. For instance, WiredTiger busy-waits during the snapshot creation procedure, more specifically when initializing new transactions and allocating their transaction IDs. This is because individual completion times are expected to be short, therefore WiredTiger attempts to group together as many transactions as possible, among those being concurrently issued by different worker threads.

However, busy-waiting is generally considered an anti-pattern if: 1) interrupts are not disabled, which is not possible in user-space; 2) task-core pinning is not 1:1, which is not enforced by MongoDB nor WiredTiger – an arbitrary number of clients can be connected and issue requests to a *mongod* instance at any time, resulting in an arbitrary number of client threads concurrently using the WiredTiger API to commit transactions to disk. This "unsafe" use of busy-waiting within WiredTiger, in combination with the nice level manipulation detailed above, causes extremely high worst-case latency values for write operations when the number of client worker threads exceeds the number of free physical cores. For instance, we measured worst-case latency values of up to 1 s (300% more than the average) in a quad-core standalone deployment of *mongod* with 8 connected clients. This is due to an instance of *priority inversion:* one or more high-priority worker threads busy-wait on a transaction-related condition that only a lower-priority thread can unblock. However, if there are no free physical cores available at that time, the latter thread is not given CPU time, since the CFS keeps scheduling the higher-priority threads that keep spinning, until the exhaustion of their time-slices. At that point, the lower-priority thread exits starvation, thus it is able to finally allocate its transaction ID and unblocks the condition other spin-waiting higher-priority threads were waiting for, so they can finally proceed. Nice levels induce small variations in the scheduling evaluation (see Section II-D), without compromising the functioning of the system. Thus lower-priority threads get anyway a chance to run, albeit rarely. However, the use of real-time priorities and the POSIX real-time scheduling policy, instead of UNIX nice values, caused in some experiments a *complete stall* of the database when configured with a few physical cores. This happens because the thread with the highest real-time priority keeps running undisturbed. Note that such priority inversion problem occurs also in the regular, unmodified version of MongoDB, albeit it is far less noticeable. When all threads have the same nice value, the CFS essentially becomes a round-robin scheduler, with a time-slice between 3 ms and 24 ms, depending on the number of concurrently active threads.[8] Indeed, in such a scenario, worst-case latency peaks of 20 ms are easily observable in MongoDB deployments with a reduced number of CPU cores (e.g., quad-core).

For this reason, we integrated a simple back-off strategy to the busy-waiting logic in WiredTiger, which spins 100 times, and then sleeps for 50 microseconds. This way, the WiredTiger transactions corresponding to higher-priority requests that try to spin-wait for too long, are temporarily put to sleep so that the transactions corresponding to lower-priority requests are no longer heavily starved and the system can move forward.

C. Truncation of Oplog Batches

Solely using nice levels is not enough to assure priority-driven differentiated performance. This is because, in replica set deployments where concurrent user operations require a *write concern* higher than 1, we have an instance of what we call *unbiased replication:* since the primary node does not respond to the user until a subset of secondary nodes replicate the batch in which the operation resides, this leads to an inevitable priority inversion, as the *OplogApplier* applies the oplog entries in parallel, and thus each batch represents a "limbo" state where no chronological or priority order is enforced. A simple propagation of priorities

[6]More information at: https://man7.org/linux/man-pages/man2/setpriority.2.html.

[7]More information at https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/.

[8]The actual boundaries can be tuned via *sysctl* parameters.

Algorithm 1: Truncate Oplog Batch.

Input: $oplogBuffer$, list of to-be-applied oplog entries
1: $entries \leftarrow \varnothing$
2: **while** $\neg\, oplogBuffer.empty()$ **do**
3: $entry \leftarrow oplogBuffer.pop()$
4: **if** $entries.back().prio > entry.prio$ **then**
5: Send *entries* to *OplogApplier*
6: $entries \leftarrow \varnothing$
7: **end if**
8: $entries.append(entry)$
9: **end while**

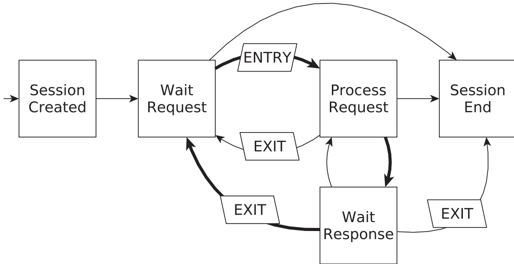


Fig. 3. The FSM modeling the session life-cycle of a client connection, integrated with the custom semaphore.

through the oplog does not solve the issue: intra-batch oplog re-ordering by priority is pointless since the next batch does not get processed until the current one is exhausted; inter-batch priority re-ordering does not comply with the data consistency model implemented by WiredTiger (see Section II), because the chronological order of the timestamps coupled to oplog entries would get mixed up between batches, causing critical internal errors. The ideal solution for oplog re-ordering would be a re-design of the replication process, but that would in turn require a complete overhaul of the database architecture and a more relaxed implementation of the data consistency model [19].

In the proposed approach herein presented, we undertake a minimally-invasive approach to the problem just mentioned above, with no changes at all in the replication protocol, and a slight modification to the batch creation logic, i.e., *batch truncation:* when the *OplogBatcher* identifies a priority fall between two continuous oplog entries in the batch, it prematurely “cuts” the batch being assembled, so to speed-up the commit to persistent storage of the prioritized entry(ies) preceding the lower-priority request(s) following in the batch. Naturally, this speeds up the secondary node acknowledgment to the primary, as described in Section II-C. Pseudocode 1 presents the batch truncation implementation logic. Note that the following priority order is assumed in the pseudocode: $HIGH > NORM > LOW$.

D. Custom Semaphore

In a typical server-class multi-core *mongod* deployment with dozens of cores, while a single high-priority request is being processed, many lower-priority ones may be concurrently handled by other client threads running on other cores, resulting in additional transactions to be performed on disk before the high-priority request can be acknowledged to the client. This still happens even in the presence of the batch truncation mechanism described above, as the system will anyway keep taking a large number of lower-priority requests that, albeit postponed by de-prioritization and/or batch truncation, will have to be synchronized on disk, and/or transferred to replicas, sooner or later, impacting on higher-priority requests, given the serializable nature of MongoDB transactions. Therefore, for guaranteeing a better service to higher-priority requests, lower-priority worker threads must be *slowed down* so that oplog entries corresponding to high-priority requests are naturally scheduled first and subsequently grouped together when processed by the *OplogBatcher*. Therefore, our modification includes a semaphore-like mechanism integrated within the client session life-cycle, that mitigates the effect of unbiased replication, by keeping track of the number of worker threads currently in the *Process-Request* state, and applying a simple priority rule: if the number of higher-priority requests being processed is beyond a tunable threshold, then temporarily block the worker threads which are processing lower-priority requests. This custom *semaphore* allows the creation of a prioritized channel to WiredTiger with close to no interferences from lower-priority requests. Whenever the number of higher-priority requests returns below the threshold, the blocked threads performing lower-priority requests are resumed and continue execution.

Theoretically, this mechanism would allow high-priority requests to undergo near-zero interference from lower-priority ones, that are paused during execution of higher-priority requests. If this is applied regardless of the number of high-priority requests being processed, then the overall throughput of MongoDB risks undergoing quite a big impact, as the parallelism capabilities of the database are effectively reduced. For this reason, the proposed mechanism allows for customizing an *activation threshold* value, which specifies the maximum number of concurrent high-priority requests in the processing state required to activate the priority channel. Below such threshold, the lower-priority requests are still served. A high value implies a lower rate of semaphore activations (i.e., fewer blocked threads), whereas a low value implies a higher rate of semaphore activations (i.e., more blocked threads). This option allows blocking lower-priority requests only under significant volumes of higher-priority traffic, thus achieving different trade-offs between response time to higher-priority requests, and drop in overall throughput of the system.

In practice, the custom semaphore is implemented by modifying the *entry* and *exit* points from the *Process-Request* state, as depicted in Fig. 3, which shows the modified FSM representing the client session life-cycle. Pseudocodes 2 and 3 present the implementation logic. The semaphore is designed to block the worker threads before entering the code sections related to the storage engine, and then resume them when the higher-priority requests have been processed. The mechanism does not interfere with the execution of requests that change the user priority, nor with sessions corresponding to internal clients (i.e., secondary node connections): this is essential for the correct

Algorithm 2: Semaphore Entry Logic.

Input:*session*, active database connection object
Input:*thr_lvl*, user-defined activation threshold
Input:*high_proc*, list of high-priority sessions in process
Input:*norm_proc*, list of normal-priority sessions in process
Input:*{norm,low}_wait*, wait queues

1: **if** $session.is_user$ **then**
2: **if** $session.prio == HIGH$ **then**
3: $high_proc.append(session)$
4: **else if** $session.prio == NORM$ **then**
5: $norm_proc.append(session)$
6: **while** $len(high_proc) \geq thr_lvl$ **do**
7: $norm_wait.push(session)$
8: $session.cond_var.wait()$
9: **end while**
10: **else**
11: **while** $len(high_proc + norm_proc) \geq thr_lvl$ **do**
12: $low_wait.push(session)$
13: $session.cond_var.wait()$
14: **end while**
15: **end if**
16: **end if**

Algorithm 3. Semaphore Exit Logic

Input:*session*, active database connection object
Input:*thr_lvl*, user-defined activation threshold
Input:*high_proc*, list of high-priority sessions in process
Input:*norm_proc*, list of normal-priority sessions in process
Input:*{norm,low}_wait*, wait queues

1: **if** $session.is_user$ **then**
2: **if** $session.prio == HIGH$ **then**
3: $high_proc.remove(session)$
4: **else if** $session.prio == NORM$ **then**
5: $norm_proc.remove(session)$
6: **end if**
7: **if** $len(high_proc) < thr_lvl$ **then**
8: **while** $\neg norm_wait.empty()$ **do**
9: $blck_session \leftarrow norm_wait.pop()$
10: $blck_session.cond_var.signal()$
11: **if** $norm_wait.empty()$ **then return**;
12: **end while**
13: **end if**
14: **if** $len(high_proc + norm_proc) < thr_lvl$ **then**
15: ▷ SAME code as lines 8-13, but for low_wait
16: **end if**
17: **end if**

functioning of the system. Note that the shown pseudocodes are conceptual – the actual implementation comprises a series of condition variables, mutexes, and atomic counters for keeping track of the high-priority, normal-priority, and low-priority users.

E. Security and Access Control

The prioritization mechanisms described above are fully integrated into the role-based access control [24] model of MongoDB to prevent "greedy" clients from draining all the throughput capacity: this allows, for instance, an administrator to provide different database accounts with properly configured permissions so that users requesting interactive workloads are allowed to submit prioritized requests, whereas users characterized by batch-type workloads cannot leverage the priority mechanism to their advantage. More precisely, the proposed modifications to MongoDB let an administrator associate each account with additional permissions that control whether an attempt to switch to a high, normal, or low priority would succeed, or fail with an unauthorized error code. Notice that authorized clients still need to explicitly call the `setClientPriority` or the generic `runCommand` commands, specifying the required priority value.

IV. Experimental Evaluation

YCSB [17] is the industry-standard benchmark tool for NoSQL data stores. It comprises a set of user-defined performance tests, called *workloads*, that define parameters such as the probability distribution of requests across the key space, the number of pre-inserted records, and the proportion of read, update, scan and insert operations to issue. YCSB evaluates the deployment's average request response time and overall throughput via a multi-threaded workload generator which issues to the targeted data store a predefined number of back-to-back operations. In this paper, the purpose of YCSB is to emulate realistic "noise" over the database instance, which could be seen as the interference created by batch applications interacting with the data store. At the same time, the individual response times of a set of client connections are monitored. These could be seen as user workloads with high responsiveness requirements (simply called "time-sensitive" from now on). The goal is to demonstrate how the proposed approach allows tuning the effect of interference between the YCSB noise and the time-sensitive workloads with respect to the unmodified version of MongoDB, thus achieving priority-based performance differentiation. Under ideal circumstances, where MongoDB is able to dedicate a physical core for each client worker thread (i.e., $\#physical_cores > \#worker_threads$), there is no need for performance differentiation, since it would efficiently handle most workloads. However, this is not often the case, especially in Cloud Infrastructures, where a multi-tenant architecture is in place and services are encapsulated in fixed-size virtual machines. For this reason, the experiments have been performed on a restricted number of physical cores such that the worker threads are forced to contend CPU time, thus emulating the resource contention scenario of a typical public Cloud Infrastructure. The workflow of each experiment does not vary: the time-sensitive clients connect to the database system while the YCSB workload

is running, declare their priority with `setClientPriority` (in the case of modified MongoDB), and then start submitting a fixed number of synchronous write operations, with no wait time between subsequent requests. For the sake of the experimental evaluation, we are interested in the response times of individual requests experienced by the time-sensitive clients, and the average throughput achieved by the YCSB noise *before*, *during* and *after* the time-sensitive workloads have been exhausted. For instance, we are not considering the total completion time of the YCSB noise, since we are assuming that it emulates a set of indefinitely long-running, batch processes issuing synchronous operations back-to-back. In our context, response time is defined as the time taken to send a query to the MongoDB deployment, execute it and receive back a response. Notice that it includes possible time spent waiting at the semaphore for the corresponding server-side worker thread. All client-related activities are hosted on a dedicated, 96-core physical system (Arm 64 server with 2 ThunderX 88XX CPUs and 64 GB of RAM) connected to the MongoDB deployment by a 1 gbE link. This is to ensure no interference with the server-side activities, which are hosted on a different 112-core physical system (x86-64 server with 2 Xeon Gold CPUs and 125 GB of RAM) with CPU frequency blocked at 2.20 GHz, and hyper-threading and turbo-boosting disabled. The latter is accompanied by two 20-core twin systems (x86-64 server with 2 Intel(R) Xeon(R) CPU E5-2640 CPUs and 64 GB of RAM) with a similar DVFS configuration, but CPU frequency blocked at 2.40 GHz. They are used as secondary nodes in replicated scenarios.

The first subsection verifies experimentally the correctness of the proposed mechanism by visually showing how the requests are ordered on a priority basis in a simple scenario with multiple priority requirements, but no YCSB noise. Subsequently, the remaining subsections demonstrate how the response times observed by different clients are differentiated on a priority basis with respect to the original version of MongoDB, in different deployment set-ups involving also the update-heavy, predefined YCSB workload (50% Updates, 50% Reads). Notice that in this case only two priority levels are used to make the plots more readable and the explanations clearer. For the sake of reproducibility, the experimental evaluation can be replicated using a small performance testing framework purposely developed for this work,[9] and a pre-configured instance of YCSB.[10]

A. Correctness of the Proposal

The first experiment aims at showing experimentally how the modified MongoDB behaves in the presence of different-priority requests hitting the system using a set of our time-sensitive clients only, without any YCSB noise workload. We deployed a 32-clients scenario in a quad-core standalone *mongod* deployment, where 4 out of the 32 clients are deployed with a high priority level, whereas the remaining ones are equally divided between normal and low priority. The CPU contention is quite high since each physical core is shared among 8 worker threads. Fig. 4 (Top) shows the response times (Y axis) over time (X axis) of 1,000 write operations issued per client: the original version of MongoDB, shown in the first plot, achieves a flat 1,233 microseconds average response time for all the clients, being unable to provide differentiated performance. The second plot shows our modified version of MongoDB with activation threshold set to 1 (default), which instead is able to transfer the requested priorities to the underlying worker threads, reducing the average response time for high-priority clients down to 511 microseconds (∼59% decrease). Naturally, this comes at the cost of some temporary unstable performance for normal-priority clients, as depicted by the average response time of 1700 microseconds with peaks of tens of milliseconds, and a complete stall of the low-priority ones. Ultimately, this implies an increase in the total duration of the test. The last timeline plot demonstrates the tuning capabilities of the activation threshold to reduce the drop in throughput: for instance, a value of 2 allows for high- and normal-priority clients to coexist at the cost of almost doubling the average response time for the high-priority clients with respect to the default activation threshold.

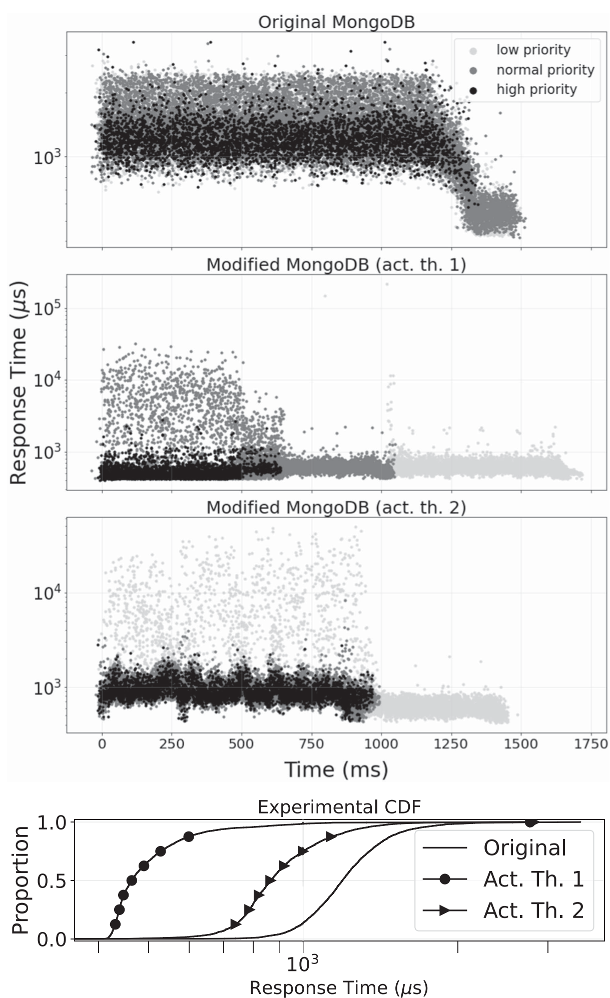


Fig. 4. Response times (Y axis) over time (X axis) in a 32-client scenario with different priorities using a quad-core deployment (Top). Three cases: unmodified MongoDB, modified MongoDB with an activation threshold of 1, and modified MongoDB with an activation threshold of 2. The corresponding experimental cumulative distribution function of the response times for high-priority clients is shown in the bottom panel.

[9]https://gitlab.retis.santannapisa.it/rtnosql/mongodb-perf
[10]https://github.com/deRemo/rt-YCSB

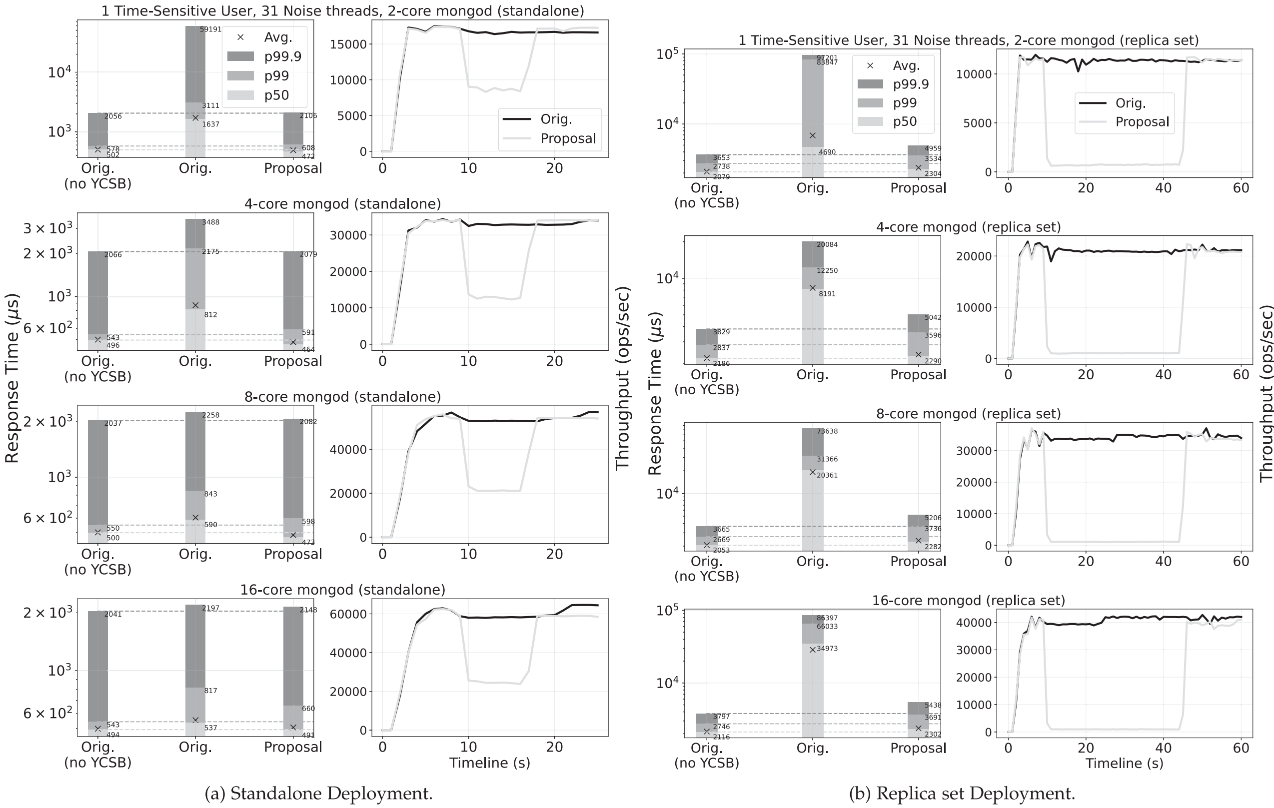


Fig. 5. Single time-sensitive client in a 32-client scenario. Each subfigure depicts 50th, 99th, 99.9th percentiles and mean response time (Left column) for the time-sensitive client; throughput per second (Right column) achieved by 31 YCSB threads while the single time-sensitive client is running. Each row corresponds to different degrees of CPU contention, from highest to lowest (Top to bottom).

Anyhow, response times for high-priority users are reduced by ∼26%, compared to original MongoDB, and the test duration is roughly the same as the first timeline plot. Fig. 4 (Bottom) shows the experimental cumulative distribution function (CDF) of the response times experienced by high-priority clients in the three considered cases. Regarding the computational overhead of the proposal, the entry logic requires 125 CPU cycles, whereas the exit logic requires 300 CPU cycles, corresponding to 52 and 125 nanoseconds, respectively, on our 2.20 GHz Xeon Gold CPU. The exit logic lasts longer because it is in charge of deciding which worker threads to re-activate based on the previously expressed priority rule. Nonetheless, both overheads are not much more expensive than an average main memory operation, making them negligible.

B. Single Time-Sensitive Client

This subsection is dedicated to the experiments performed with a *single high-priority client* in a 32-client scenario, where the remaining 31 normal-priority connections are established by YCSB. In this scenario, the activation threshold is set to the recommended value of 1 so that the semaphore is in use, otherwise the performance differentiation would be entirely dependent on nice levels, with the problems already discussed in Section III. Each time-sensitive client issues 15,000 write operations, whereas each YCSB user submits continuously a mix of 50%/50% read/write operations till termination. Fig. 5 shows the resulting response times of a time-sensitive client and the corresponding throughput of YCSB noise. Each row of plots corresponds to a different level of CPU contention in *mongod:* from the highest (2-cores, topmost plots), where each physical core is shared between 16 worker threads, to the lowest (16-cores, bottommost plots), where each physical core is shared between 2 worker threads. Notice that a lower contention scenario allows naturally for a higher throughput.

Fig. 5(a) presents the results in a standalone MongoDB deployment, thus no data durability guarantees. The proposed mechanism keeps the response times of time-sensitive users close to the scenario with no YCSB noise, hereinafter called the “baseline”, whereas the original MongoDB is unable to achieve low responsiveness, especially in a high-contention scenario. In the 2-core deployment, the proposed approach achieves a ∼71% decrease in median response time for the time-sensitive client, with respect to the original version of MongoDB. Most notably, our proposal experiences a ∼96% decrease in 99.9th percentile ($P999$) response time, closely resembling the baseline. This comes at the cost of a ∼33% decrease of the YCSB clients throughput for the duration of the time-sensitive clients, as depicted by the throughput drop in the right-column plot (roughly from time 10 to 17). In lower contention scenarios, the throughput drop is naturally higher, since more physical cores available implies more running client threads, and therefore the semaphore activates more often. Notice that in the 16-core case (bottommost plots), the decrease in average response time is

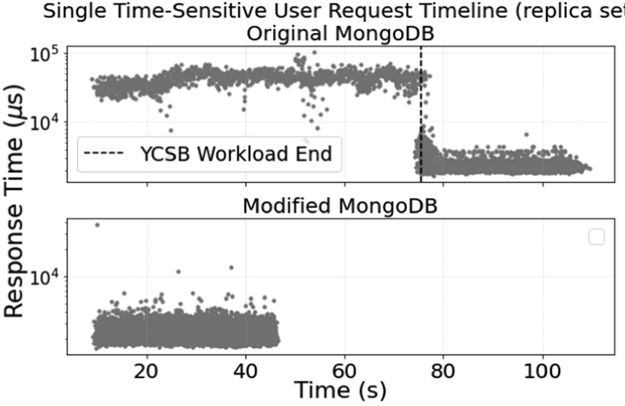


Fig. 6. Time-sensitive requests timeline in a 32-client scenario using a 16-core replica set deployment, comparing the original MongoDB with the modified one. The grey dashed line highlights the end of the YCSB stress test.

negligible with respect to the original version of MongoDB, because it is already close to the baseline (shown in the response time plot). Nonetheless, the 99th percentile ($P99$) response time is ∼20% better.

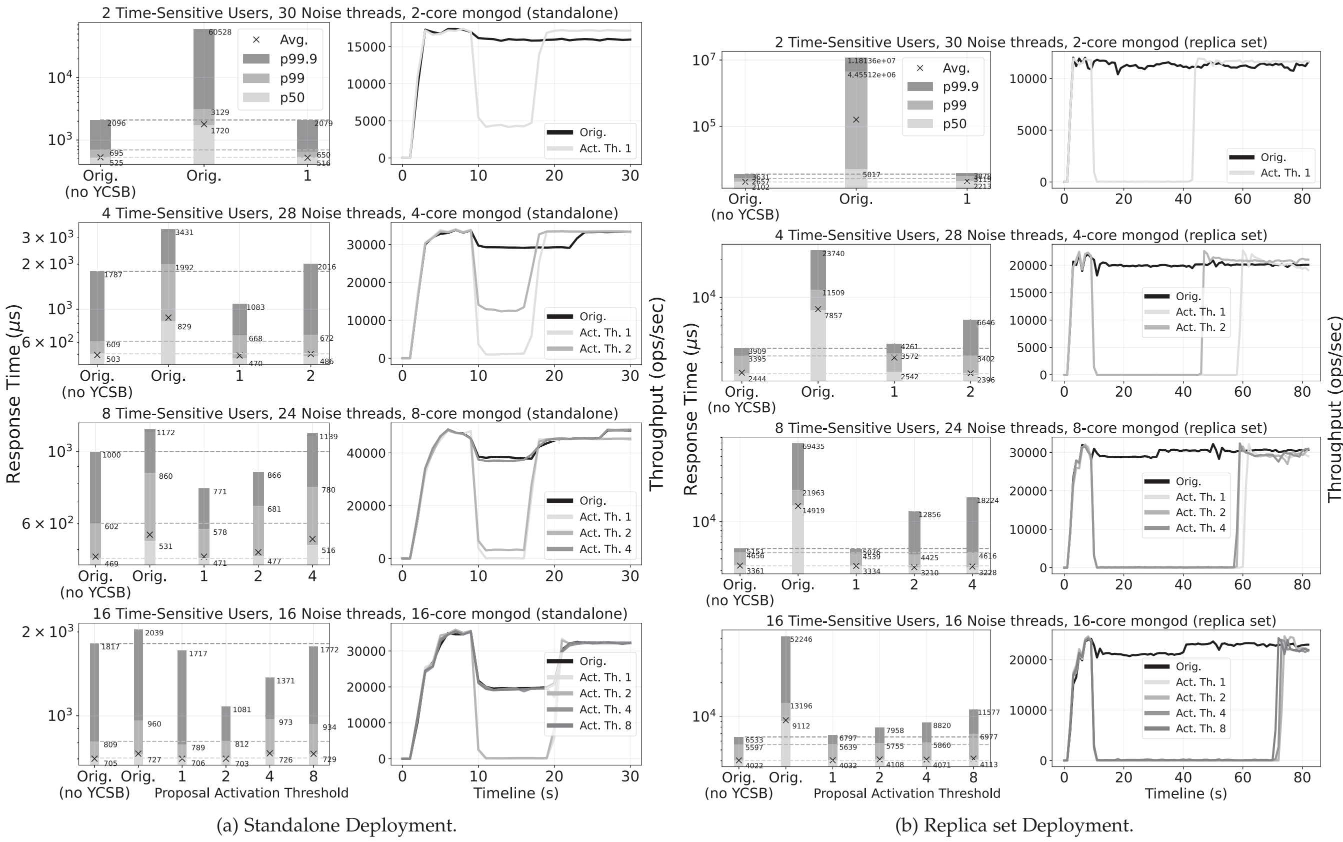
Fig. 5(b) presents the results of analogous experiments but performed in a 3-member replica set deployment with write concern set to *majority* for the time-sensitive workload. The replication process adds up a noticeable overhead to the individual latencies. As described in Section II-C, the primary node waits for a subset of secondary nodes before answering a query to ensure data consistency. The modified MongoDB provides significantly reduced response times for time-sensitive clients, w.r.t. the original MongoDB. This is especially evident in high throughput scenarios (i.e., less CPU contention): a higher throughput implies more crowded oplog batches to be processed by secondary nodes, which in turn causes extremely high latencies for write operations with data durability requirements. Therefore, the oplog entries corresponding to YCSB noise operations may backlog time-sensitive requests in the original version of MongoDB, whereas the proposed approach is able to keep the performance differentiated. This has dramatic repercussions in low CPU contention scenarios: for instance, the original version of MongoDB experiences excessively long latencies in the 16-core deployment case (corresponding to the bottommost plot in Fig. 5(b)). Our proposal beats original MongoDB by ∼93% on median response time, and by ∼94% on $P999$, closely matching the baseline response time. The effect of the backlog is further highlighted in the timeline plot in Fig. 6: the YCSB workload has been prematurely terminated, in the original version of MongoDB, to complete the experiment in a reasonable amount of time. Conversely, the modified version of MongoDB is able to almost temporarily halt the YCSB throughput, decreasing the original rate by 99% for the duration of the time-sensitive workload. Notice this rate reduction is justifiable considering the number of time-sensitive requests and the average baseline response time of 2200 microseconds: a write request with a majority write concern takes almost 6 times longer to complete, w.r.t a "normal" write request in our replica set deployment. For the majority of such a time span, the high-priority client keeps the priority channel active. This results in 33 seconds of total completion time for the time-sensitive workload, and therefore at least 33 seconds of total waiting time for the lowest priority loads with an average categorical stop period of 2,200 microseconds.

C. Multiple Time-Sensitive Clients

Additional experiments have been performed with a number of high-priority clients in a 32-client scenario, where the remaining normal-priority clients come from YCSB (Fig. 7). The testing approach is the same as in the previous subsection, although these experiments have been arranged so that: the number of time-sensitive clients is always equal to the amount of allocated physical cores; the total number of concurrent clients (i.e., time-sensitive + YCSB noise) is kept constant throughout the different levels of CPU contention to ensure a fair comparison between the various cases shown in the plots. Unlike the experiments in the previous subsection, the activation threshold can be set higher than 1 to allow for different trade-offs between response time of time-sensitive clients and overall throughput. Fig. 7(a) presents the results in a standalone MongoDB deployment, thus with no data durability guarantees. The results follow the trend of previous experiments in Fig. 5(a), especially the base case of activation threshold equal to 1. A higher activation threshold consistently implies worst performance for the time-sensitive workload but with the advantage of having a lower throughput drop in the YCSB noise. Notice that the performance degradation for the time-sensitive clients mostly impacts the $P99$ and $P999$ response time. A higher activation threshold implies worse performance for the time-sensitive workload if the number of high-priority clients is not equal to or higher than the threshold. This happens especially toward the end of an experiment since some high-priority clients exhaust their workload first. The effect is more noticeable in lower CPU contention scenarios with fewer semaphore activations (i.e., a higher threshold value), where the performance differentiation is more subtle regardless of the drop in YCSB throughput, as can be seen from plots. Fig. 7(b) presents the experimental results performed in a 3-member replica set deployment with write concern set to *majority* for the time-sensitive workload. Notice that in this case, the activation threshold parameter is almost irrelevant, due to the amount of waiting time required for a single high-priority request to be processed, as already described at the end of the previous subsection.

V. Related Work

Real-time database systems [25] were the first to move away from the conventional goal of providing fast *average* response times or overall high throughput. Unlike traditional databases [26], a real-time one must meet the timing constraints of a set of real-time applications by employing appropriate scheduling protocols for data processing, CPU, I/O, and memory access, so that there are no missed deadlines, or their number is minimized. However, real-time database system research is of interest only for the hard real-time system niche, with no recent academic literature on the topic.



(a) Standalone Deployment. (b) Replica set Deployment.

Fig. 7. Multiple time-sensitive clients in a 32-client scenario. Each subfigure depicts 50th, 99th, 99.9th percentiles and mean response time (Left column) for time-sensitive clients; throughput per second (Right column) achieved by several YCSB threads while the time-sensitive clients are running. Each row corresponds to different degrees of CPU contention, from highest to lowest (Top-Bottom). Note that the total number of concurrent clients (i.e., time-sensitive + YCSB noise) always sums up to 32.

An orthogonal line of research addresses the need for improved latency by fine-tuning the underlying runtime environment, and by using optimized file systems, I/O subsystems, and cutting-edge storage devices. The first section of Table I summarizes a subset of papers in the context of "local storage optimization". Kim et al. [27] propose an I/O stack schema within the Linux kernel that takes advantage of both zero-copying and the use of the page cache for modern low-latency solid-state drives (SSDs) to reduce latency. Litz et al. [28] present an SSD device-level redundancy technique to enforce predictable low-tail latency for Flash accesses: more specifically, the mechanism provides an alternative read data path when a NAND chip is temporarily inaccessible, thus eliminating the possibility of reads being stalled by high latency operations (read-after-write serialization). Kang et al. [29] propose an optimized firmware for NVMe SSDs to enable strong physical isolation for co-located virtualized services. The device-level interferences are reduced by setting up exclusive I/O paths and cache regions, thus hugely improving the tail latency. Pine [30] is an isolation tool for storage services with differentiated performance. Pine dynamically manages the disk resource allocation and the I/O concurrency level for each service according to their latency and throughput requirements. Although this is one of the few works that closely matches our context (i.e., differentiated performance), it does not prioritize time-sensitive storage services. Moreover, it only focuses on a single physical machine. In general, these low-level mechanisms are useful for local storage optimizations, however, to support differentiated levels of service performance in a distributed environment, the database software itself needs to be changed with non-trivial modifications to the request processing and handling code paths.

The advent of web-based interactive applications with high responsiveness requirements, together with the developments of highly scalable cloud infrastructures, have fostered the growth of NoSQL architectures, capable of ingesting arbitrarily high volumes of data and scaling at will on several nodes. In the academic literature, no studies address the challenge of differentiating query response times on a priority basis. Anyhow, there are a number of works proposing whole performance-aware NoSQL database prototypes, with very few papers proposing ready-to-use mechanisms integrated into a production-grade database system such as MongoDB. These works could be coupled with strong real-time design principles to guarantee predictable response times and sufficient resources for time-sensitive activities. The second section of Table I presents a qualitative analysis of a subset of works on novel NoSQL data stores. AQUAS [31], [32] is a QoS-aware allocator that enriches the Cassandra [37] NoSQL database with several task scheduling policies and a cost estimation component to satisfy individual clients' performance requirements expressed as a set

TABLE I
RELATED WORKS

Study	Context	Software Stack	Feature	Performance Evaluation
[27]	Storage Opt.	OS	Zero-copy I/O Stack	Throughput
[28]	Storage Opt.	Firmware	Device-level resource contention control	Tail latency
[29]	Storage Opt.	Firmware	Device-level physical isolation of resources	Tail latency
[30]	Storage Opt.	OS	Dynamic storage resource allocation	• Tail latency • Throughput
[31], [32]	Cassandra Ext.	Application	QoS-aware and QoD-aware task scheduling	∅
[33]	MongoDB Ext.	Application	Concurrency control optimized for write-heavy workloads	Throughput
[34]	NoSQL Prototype	Application	Self-designing storage engine	• Latency • Throughput
[35]	NoSQL Prototype	Application	Dynamic reconfiguration	• Tail latency • Throughput
[36]	NoSQL Prototype	Application	Latency-aware storage for state externalization in stateless applications	Latency
This paper	MongoDB Ext.	OS + Application	Priority-based request processing	Diff. Performance

of user-defined Quality-of-Service (QoS) and Quality-of-Data (QoD) constraints. However, the work does not provide an exhaustive evaluation. Xyza [33] is an extension of MongoDB that combines classic and novel design techniques to overcome the scalability limits of current concurrency control mechanisms. For instance, it replaces the complex lock manager with a simpler wait-signal mechanism based on atomic primitives and partitions the system-wide journaling system into a per-client journal to avoid I/O contention. Although the paper refers to an older version of MongoDB's concurrency control and does not evaluate the proposal in replicated scenarios, the idea of per-client journaling could benefit our work too. Sophia [35] is a reconfigurable NoSQL datastore that estimates performance degradation and predicts workload pattern changes without impacting data availability during reconfiguration. Szalay et al. [36] propose a latency-aware and data access pattern-aware method in the context of stateless applications. The latency towards the most frequently accessed data entries is minimized by monitoring the performance of the underlying infrastructure, tracking the number of reads and writes, and then re-optimizing data locations across the database instances. Cosine [34] is a self-designing storage engine that introduces a unified model to produce an optimal key-value data structure given a budget, a workload, and a target performance. It spans diverse storage engine designs, such as B-trees and Log-Structured Hash-tables, and picks the one which minimizes the expected cost and latency.

Outside academia, the are several fully-managed Database-as-a-Service solutions that promise low latency and/or high throughput. The most famous one is DynamoDB [13], which allows its customers to specify the throughput requirements (read and write capacity) for a given table, and then the service allocates sufficient resources to the table to predictably achieve real-time performance and stable latency values under 10 ms at the 99th percentile. Predictability is achieved by performing granular capacity planning at individual physical infrastructure elements, like individual storage nodes and even individual single SSD drives, coupled with optimizations, adaptive load-balancing, and topology reconfigurations at the infrastructure level. DynamoDB is the only industrial-grade database service that closely matches our context: DynamoDB offers differentiated performance throughput-wise with tail latency guarantees; our proposal achieves differentiated performance between time-sensitive and throughput-first applications via prioritization. However, DynamoDB is proprietary software only available through AWS, whereas our proposal is based on an open-source code base. Scylla Cloud[11] offers a fully-managed option of their ScyllaDB NoSQL database, which employs a highly asynchronous, shared-nothing architecture (i.e., each CPU core handles a different subset of data, without sharing) that guarantees high-throughput/low-latency workloads. ScyllaDB efficiently allocates the resources to each shard of data thanks to custom-made schedulers for CPU and I/O processing, taking full advantage of low-level Linux primitives and the underlying hardware. Other fully-managed services are Google Firestore [38] and MongoDB Atlas.[12] Despite the generic claims, only DynamoDB quantifies its predictability guarantees. One of the latest breeds of database systems is in-memory data stores, which employ the main memory instead of disk storage for ultra-fast access times. They are designed for applications where huge amounts of data must be processed in real-time. A few industry-level, fully-managed examples are Google Memorystore,[13] Amazon ElastiCache[14] and Amazon MemoryDB.[15] However, none of these services offer a performance differentiation feature.

[11] See: https://www.scylladb.com/product/scylla-cloud/
[12] See: https://www.mongodb.com/atlas/database
[13] See: https://cloud.google.com/memorystore
[14] See: https://aws.amazon.com/elasticache/
[15] See: https://aws.amazon.com/memorydb/

VI. Conclusion

The paper describes a variant of the MongoDB NoSQL database which embeds priority-driven scheduling on a per-user or per-request basis so that higher-priority workloads are served first. This is achieved by instantiating a prioritized channel for higher-priority requests and revoking or restricting access to the storage unit to lower-priority ones. In practice, the priority order is enforced by a combination of nice level manipulation and a semaphore-like structure that indirectly propagates the priorities in replicated scenarios. Experimental results carried on stand-alone and a 3-replica set deployment demonstrate how higher-priority requests are consistently served with shorter response times exhibiting less variance, with respect to lower-priority requests. For instance, in most scenarios with a replica set, the $P999$ response time of the proposal is less than or equal to the median of original MongoDB. Then, it is demonstrated how to fine-tune the trade-off between reduced response time and overall system throughput when possible, by controlling the loss in parallelism caused by the semaphore system. This is especially noticeable in standalone scenarios with fewer high-priority users, where requests with mixed priority are more interleaved. Furthermore, this work highlights an issue with the synchronization mechanism implemented by the underlying storage engine (i.e., WiredTiger), and presents a workaround that proved beneficial for the case study. To the best of our knowledge, our proposal is the only open-source NoSQL data store with such query prioritization feature.

As possible future work on the topic, it might be interesting to investigate an interface that allows to specify the timing constraints for a given request/user so that the database can provide end-to-end response time guarantees. Such a feature might be implemented by adding a capacity management layer within MongoDB, and coupling the proposed mechanism with an adaptive controller that automatically adjusts the activation threshold and the priorities according to the workload requirements, or using more advanced CPU scheduling techniques, like SCHED_DEADLINE [39]. Another option is to further improve the worst-case latency values and increase the performance predictability by simplifying the synchronization logic of MongoDB and WiredTiger.

Acknowledgment

The authors would like to thank Sulabh Mahajan and the rest of the MongoDB development team for their insightful support on the internals of MongoDB and WiredTiger.

References

[1] S. Bibi, D. Katsaros, and P. Bozanis, “Business application acquisition: On-premise or SaaS-based solutions?,” *IEEE Softw.*, vol. 29, no. 3, pp. 86–93, May/Jun. 2012.

[2] D. Gannon, R. Barga, and N. Sundaresan, “Cloud-native applications,” *IEEE Cloud Comput.*, vol. 4, no. 5, pp. 16–21, Sep./Oct. 2017.

[3] M. A. S. Netto, R. N. Calheiros, E. R. Rodrigues, R. L. F. Cunha, and R. Buyya, “HPC cloud for scientific and business applications: Taxonomy, vision, and research challenges,” *ACM Comput. Surv.*, vol. 51, no. 1, Jan. 2018, Art. no. 8.

[4] M. García-Valls, T. Cucinotta, and C. Lu, “Challenges in real-time virtualization and predictable cloud computing,” *J. Syst. Archit.*, vol. 60, no. 9, pp. 726–740, 2014.

[5] T. Li, J. Tang, and J. Xu, “Performance modeling and predictive scheduling for distributed stream data processing,” *IEEE Trans. Big Data*, vol. 2, no. 4, pp. 353–364, Dec. 2016.

[6] R. Mancini, T. Cucinotta, and L. Abeni, “Performance modeling in predictable cloud computing,” in *Proc. 10th Int. Conf. Cloud Comput. Serv. Sci.*, SciTePress, 2020, pp. 69–78.

[7] R. Shea, J. Liu, E. N. Ngai, and Y. Cui, “Cloud gaming: Architecture and performance,” *IEEE Netw.*, vol. 27, no. 4, pp. 16–21, Jul./Aug. 2013.

[8] S. M. Park and Y.-G. Kim, “A metaverse: Taxonomy, components, applications, and open challenges,” *IEEE Access*, vol. 10, pp. 4209–4251, 2022.

[9] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo, *Soft Real-Time Systems*. Berlin, Germany: Springer, 2005.

[10] R. Buyya, C. Vecchiola, and S. T. Selvi, *Mastering Cloud Computing: Foundations and Applications Programming*. Oxford, U.K.: Newnes, 2013.

[11] A. Varasteh and M. Goudarzi, “Server consolidation techniques in virtualized data centers: A survey,” *IEEE Syst. J.*, vol. 11, no. 2, pp. 772–783, Jun. 2017.

[12] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan, “Workload-aware database monitoring and consolidation,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2011, pp. 313–324. [Online]. Available: https://doi.org/10.1145/1989323.1989357

[13] S. Perianayagam et al., “Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service,” in *Proc. USENIX Annu. Tech. Conf.*, Carlsbad, CA: USENIX Association, 2022, pp. 1037–1048. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/vig

[14] T. Cucinotta, L. Abeni, M. Marinoni, R. Mancini, and C. Vitucci, “Strong temporal isolation among containers in openstack for NFV services,” *IEEE Trans. Cloud Comput.*, vol. 11, no. 1, pp. 763–778, First Quarter 2023.

[15] R. Andreoli, T. Cucinotta, and D. Pedreschi, “RT-MongoDB: A NoSQL database with differentiated performance,” in *Proc. 11th Int. Conf. Cloud Comput. Serv. Sci.*, SciTePress, 2021, pp. 77–86.

[16] R. Andreoli and T. Cucinotta, “Differentiated performance in NoSQL database access for hybrid cloud-HPC workloads,” in *Proc. Int. Conf. High Perform. Comput.*, H. Jagode, H. Anzt, H. Ltaief, and P. Luszczek, Eds. Cham: Springer International Publishing, 2021, pp. 439–449.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proc. 1st ACM Symp. Cloud Comput.*, New York, NY, USA, 2010, pp. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[18] R. Cattell, “Scalable SQL and NoSQL data stores,” *ACM SIGMOD Rec.*, vol. 39, no. 4, pp. 12–27, 2011.

[19] E. A. Brewer, “Towards robust distributed systems,” in *Proc. 19th Annu. ACM Symp. Princ. Distrib. Comput.*, Portland, OR, 2000, pp. 343477–343502.

[20] P. A. Bernstein and N. Goodman, “Multiversion concurrency control–theory and algorithms,” *ACM Trans. Database Syst.*, vol. 8, no. 4, pp. 465–483, Dec. 1983.

[21] T. Anderson and M. Dahlin, *Operating Systems: Principles and Practice*, vol. 1. Albany, CA, USA: Recursive Books, 2014.

[22] T. Cucinotta, L. Abeni, M. Marinoni, A. Balsini, and C. Vitucci, “Reducing temporal interference in private clouds through real-time containers,” in *Proc. IEEE Int. Conf. Edge Comput.*, 2019, pp. 124–131.

[23] C. S. Wong, I. Tan, R. D. Kumari, and F. Wey, “Towards achieving fairness in the Linux scheduler,” *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 34–43, Jul. 2008.

[24] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, “Proposed NIST standard for role-based access control,” *ACM Trans. Inf. Syst. Secur.*, vol. 4, no. 3, pp. 224–274, 2001.

[25] B. Kao and H. Garcia-Molina, “An overview of real-time database systems,” in *Proc. Real Time Comput.*, W. A. Halang and A. D. Stoyenko, Eds. Berlin, Heidelberg: Springer, 1994, pp. 261–282.

[26] J. M. Hellerstein, M. Stonebraker, and J. Hamilton, *Architecture of a Database System*. Boston, MA, USA: Now, 2007.

[27] S. Kim, G. Lee, J. Woo, and J. Jeong, “Zero-copying I/O stack for low-latency SSDs,” *IEEE Comput. Archit. Lett.*, vol. 20, no. 1, pp. 50–53, Jan.–Jun. 2021.

[28] H. Litz, J. Gonzalez, A. Klimovic, and C. Kozyrakis, “RAIL: Predictable, low tail latency for NVMe flash,” *ACM Trans. Storage*, vol. 18, no. 1, Jan. 2022, Art. no. 5. [Online]. Available: https://doi.org/10.1145/3465406

[29] L. Kang and B. Jacob, “Zoned FTL: Achieve resource isolation via hardware virtualization,” in *Proc. Int. Symp. Memory Syst.*, New York, NY, USA, 2022, Art. no. 7. [Online]. Available: https://doi.org/10.1145/3488423.3519326

[30] Y. Li, J. Zhang, C. Jiang, J. Wan, and Z. Ren, “PINE: Optimizing performance isolation in container environments,” *IEEE Access*, vol. 7, pp. 30410–30422, 2019.
[31] C. Xu, F. Xia, M. A. Sharaf, M. Zhou, and A. Zhou, “AQUAS: A quality-aware scheduler for NoSQL data stores,” in *Proc. IEEE 30th Int. Conf. Data Eng.*, 2014, pp. 1210–1213.
[32] C. Xu, M. A. Sharaf, X. Zhou, and A. Zhou, “Quality-aware schedulers for weak consistency key-value data stores,” *Distrib. Parallel Databases*, vol. 32, no. 4, pp. 535–581, 2014.
[33] Y. Patel, M. Verma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Revisiting concurrency in high-performance NoSQL databases,” in *Proc. 10th USENIX Workshop Hot Topics Storage File Syst.*, Boston, MA: USENIX Association, 2018, Art. no. 12.
[34] S. Chatterjee, M. Jagadeesan, W. Qin, and S. Idreos, “Cosine: A cloud-cost optimized self-designing key-value storage engine,” in *Proc. VLDB Endowment*, vol. 15, no. 1, pp. 112–126, 2021.
[35] A. Mahgoub et al., “SOPHIA: Online reconfiguration of clustered NoSQL databases for time-varying workloads,” in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 223–240.
[36] M. Szalay, P. Matray, and L. Toka, “AnnaBellaDB: Key-value store made cloud native,” in *Proc. 16th Int. Conf. Netw. Service Manage.*, 2020, pp. 1–5.
[37] A. Lakshman and P. Malik, “Cassandra: A decentralized structured storage system,” *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.
[38] R. Kesavan, D. Gay, D. Thevessen, J. Shah, and C. Mohan, “Firestore: The NoSQL serverless database for the application developer,” in *Proc. IEEE 39th Int. Conf. Data Eng.*, 2023, pp. 3367–3379.
[39] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, “Deadline scheduling in the Linux kernel,” *Softw. Pract. Exp.*, vol. 46, no. 6, pp. 821–839, 2016.

Remo Andreoli received the MSc degree with honors in computer science from the University of Pisa. He is currently working toward the PhD degree with SSSA investigating on resource management optimization techniques for cloud infrastructures. He held a research scholarship from Sant’Anna School of Advanced Studies (SSSA), during which he worked on differentiated performance mechanisms for NoSQL databases, earning a best student paper award with CLOSER 2021.

Tommaso Cucinotta received the MSc degree in computer engineering from the University of Pisa, Italy, and the PhD degree in computer engineering from Scuola Superiore Sant’Anna (SSSA) in Pisa, where he has been investigating on real-time scheduling for soft real-time and multimedia applications, and predictability in infrastructures for cloud computing and NFV. He has been MTS in Bell Labs in Dublin, Ireland, investigating on security and real-time performance of cloud services. He has been a software engineer in Amazon Web Services in Dublin, Ireland, where he worked on improving the performance and scalability of DynamoDB. Since 2016, he is an associate professor with SSSA and head of the Real-Time Systems Lab (RETIS) since 2019. He has also brought a number of funded research projects to the RETIS lab, notably EU projects, and international collaborations with private companies, including some big-tech ones. He coauthored roughly a hundred international peer-reviewed scientific publications on topics in his areas of interest, and more than 20 filed and 9 granted patents. He is in the technical program committee of a number of international workshops and conferences related to his research topics, and he also performs regularly peer-reviewing for papers submitted to prestigious international journals.

Daniel Bristot De Oliveira received the joint PhD degree in automation engineering from UFSC (BR) and embedded real-time systems from Scuola Superiore Sant’Anna (IT). Currently, he is senior principal software engineer with Red Hat, working on developing the real-time features of the Linux kernel. He helps in the maintenance of real-time related tracers and toolings for the Linux kernel and the SCHED_DEADLINE. He is an affiliate researcher with the Retis Lab, and researches real-time and formal methods.