RT-MongoDB: a NoSQL database with differentiated performance

Remo Andreoli[1,2][a], Tommaso Cucinotta[1][b], Dino Pedreschi[2][c]
[1] *Scuola Superiore Sant'Anna, Pisa, Italy*
[2] *University of Pisa, Pisa, Italy*
remo.andreoli@santannapisa.it, tommaso.cucinotta@santannapisa.it, dino.pedreschi@unipi.it

Keywords: Cloud Computing, NoSQL DataBases, Real-Time Systems

Abstract: The advent of Cloud Computing and Big Data brought several changes and innovations in the landscape of database management systems. Nowadays, a cloud-friendly storage system is required to reliably support data that is in continuous motion and of previously unthinkable magnitude, while guaranteeing high availability and optimal performance to thousands of clients. In particular, NoSQL database services are taking momentum as a key technology thanks to their relaxed requirements with respect to their relational counterparts, that are not designed to scale massively on distributed systems. Most research papers on performance of cloud storage systems propose solutions that aim to achieve the highest possible throughput, while neglecting the problem of controlling the response latency for specific users or queries. The latter research topic is particularly important for distributed real-time applications, where task completion is bounded by precise timing constraints.
In this paper, the popular MongoDB NoSQL database software is modified introducing a per-client/request prioritization mechanism within the request processing engine, allowing for a better control of the temporal interference among competing requests with different priorities. Extensive experimentation with synthetic stress workloads demonstrates that the proposed solution is able to assure differentiated per-client/request performance in a shared MongoDB instance. Namely, requests with higher priorities achieve reduced and significantly more stable response times, with respect to lower priorities ones. This constitutes a basic but fundamental brick in providing assured performance to distributed real-time applications making use of NoSQL database services.

1 Introduction

The importance and disruptive capabilities of Cloud Computing have become increasingly evident throughout the last decade for the development of modern web applications. Indeed, it is a technology that offers to any-size business organizations the tremendous benefits of relieving them from the burden of investing into dedicated datacenters. Albeit a number of studies (Ara et al., 2019; Lettieri et al., 2017) apply high-performance computing techniques to distributed and cloud applications for the purpose of achieving the highest possible performance, only a few works focus on providing customizable end-to-end service-level objectives with differentiated QoS offered to different users/customers. While increasing the overall throughput of a system brings naturally to a decrease of its *average* response-time, the trade-off between throughput and response-time becomes evident when designing services that need to balance throughput maximization and quick reaction to asynchronous *real-time* requests: the former calls for approaches based on aggregating many requests into big batches to be processed at once with minimum overheads, and the use of long intermediate buffers so that worker threads minimize their idle times; whilst the latter requires preemptible short-lived activities that can be suspended if higher-priority ones arrive, and the use of small buffers so to minimize the processing latency of individual requests. Designing distributed software systems seeking the right balance between these requirements is all but trivial.

Scenarios that would benefit the most from a design supporting differentiated per-user/request performance are those where highly heterogeneous types of applications need to submit requests to the same component(s). For example, a database system in a cloud datacenter that needs to serve heavyweight requests for batch or high-performance applications, alongside lightweight requests coming from *soft real-time* appli-

[a] https://orcid.org/0000-0002-3268-4289
[b] https://orcid.org/0000-0002-0362-0657
[c] https://orcid.org/0000-0003-4801-3225

cations that need to react promptly to user interaction or asynchronous conditions, such as on-line gaming or collaborative editing of documents.

In this paper, focusing on the field of storage systems, the problem is addressed by using the prioritized access principle, a trade-off widely adopted in the design of real-time systems: higher priority activities are allowed to take resources earlier than lower priority ones, and sometimes even able to preempt them by revoking their resource access in the middle of a task, or to starve them for arbitrarily long time windows. While the priority-based access technique by itself may not be enough to achieve predictable performance, when coupled with strong real-time design principles and analysis it is possible to ensure correct operation and sufficient resources to all hosted real-time activities (Buttazzo et al., 2005). In this context, NoSQL database services are taking momentum as a key technology for replicated and infinitely scalable data storage services, thanks to their often reduced consistency and functionality requirements with respect to their relational counterparts.

1.1 Contributions

In this paper, an approach is presented to support differentiated per-client/request performance in MongoDB, a well-known open-source NoSQL database widely used in cloud applications. A special API is introduced to let clients declare their needed service priority, and modifications to the MongoDB backend daemon allow for prioritizing requests with higher priorities. The result is a set of modifications to MongoDB allowing for a differentiated performance mechanism that creates an on-demand prioritized access channel with reduced response times. These changes integrate well with the internal architecture of the base software without affecting the expected performance and reliability capabilities when not in use. In order to become usable in a production environment, the proposed approach has to be completed with a formal performance analysis and an appropriate multi-user access-control model, which are subjects of future work.

1.2 Paper Organization

This paper is organized as follows. In Section 2, related work in the research literature is briefly discussed. In Section 3, a few background concepts and terminology useful for a better understanding of the paper are recalled. In Section 4, our proposed approach to support differentiated per-client/request performance levels in MongoDB is presented. In Section 5, experimental results are discussed, gathered on a real platform running our modified MongoDB on Linux, demonstrating the effectiveness of the approach in tackling the problem introduced above. Finally, conclusions are drawn in Section 6, along with a brief discussion of possible further work on the topic.

2 Related Work

This section includes a summary of the research literature related to real-time and scalable data stores, and well-known mechanisms for optimizating their performance in practical cloud deployments.

2.1 Real-Time Database Systems

In the research literature, a real-time database system (RTDBS) (Kao and Garcia-Molina, 1994; Bestavros et al., 1997; Lindström, 2008) is a data storage and retrieval system designed in a way that it is possible to rely on a predictable timing of the operations requested by clients. This in turn enables the possibility to design real-time applications or services with precise guarantees regarding their execution times and their ability to respect their timing constraints, like deadlines. Research efforts in this area complemented research on scheduling of processes on the CPU (Buttazzo et al., 2005; Baruah et al., 2015), with investigations on scheduling of transactions. This is a complex area where different requirements may have to be considered: from maximization of throughput and efficient execution vs fairness trade-offs, as generally required in general-purpose systems, to predictable execution and prioritization of clients, as needed in real-time systems. Real-time database systems found applications in a number of traditional hard real-time application domains, ranging from mission control in aerospace to process control in industrial plants, telecommunication systems and stock trading (Kang et al., 2007). A number of challenges was typically due to the high seek latency of traditional rotational disks, a problem either tackled using special disk access scheduling techniques coupled with sufficiently pessimistic analysis techniques, or avoided using memory-only database systems (Garcia-Molina and Salem, 1992). In the last decade, the introduction in industry of SSD drives caused a major shift of the panorama, getting rid of the high seek latency problem, albeit predictability is impaired by the often non-disclosed sector allocation, prefetch and caching logic within the drive controllers. Another problem is the presence of dynamic workload conditions, that pushed towards the adoption of adaptive feedback-based scheduling techniques (Amirijoo et al., 2006; Kang et al., 2007). Unfortunately, the great focus of real-time database research on hard real-time systems, and the necessarily pessimistic analysis accompanying their design, causing poor utilization at run-time, caused these systems to remain of interest only in a restricted domain area.

2.2 Scalable Real-Time Data Stores

Recent developments of highly scalable cloud infrastructures, with sophisticated and highly adaptive performance monitoring and control techniques that are deployed in cloud systems, raised a certain interest in cloud-hosted database systems also for real-time applications, particularly soft real-time ones. Here, the focus on big-data workloads that do not focus on a single system, coupled with high-performance and high-reliability requirements forcing the adoption of data sharding and replication techniques, caused an increasing interest in dropping the typical feature-richness of relational databases, in favour of NoSQL architectures (Jing Han et al., 2011; Li and Manoharan, 2013) that implement essentially reliable/replicated distributed hash tables with the ability to ingest arbitrarily high volumes of data, scaling at will on several nodes. A commercial database that is almost unique in this context is DynamoDB[1] from AWS, a fully managed 24/7 NoSQL data store for AWS customers. Its major selling point is the ability to declare a desired per-table read and write operations per second, and the system is able to guarantee these requirements with a per-request latency lower than 10ms with a high probability.

More recently, in the area of distributed computing in cloud infrastructures, an increasing attention has been given by researchers and industry practitioners to real-time stream processing services (Wingerath et al., 2016; Basanta-Val et al., 2017; Kulkarni et al., 2015). These are services designed for immediate processing of data as it comes from sources like IoT devices or data stores, in a way that the computation results are made available with a short latency from when new data becomes available. These systems are developed using specialized platforms on top of which complex analytics performing *continuous queries* (Babu and Widom, 2001) can be realized in the form of arbitrary topology of processing functions to be applied to each incoming data item, distributed throughout the network. Cloud technologies let these systems scale easily to tens or hundreds of nodes. Often, performance and latency control is heuristically achieved by applying elastic scaling to individual functions in the topology. A few works (Theeten et al., 2014; Li et al., 2016) tried to build empirical performance models of these system, so to employ more precise control logics for the end-to-end performance.

[1]See: `https://aws.amazon.com/dynamodb/`.

2.3 Performance Optimizations

Database practitioners are used to a number of tricks to fine-tune the configuration of the runtime environment to maximize performance (Navrátil et al., 2020). For example, for databases running on the Linux operating system (OS), it is typical to drop the default Ext4 file-system (FS) used by most distributions, in favour of faster solutions like XFS[2], a high-performance, scalable file-system originated on SGI IRIX. This comes at the very acceptable cost of losing some flexibility in FS management operations that may not be needed on production servers (RHE, 2020; Navrátil et al., 2020). Another useful configuration of servers that need to host memory-intensive workloads (as needed to host several virtual machines or database systems), is the one of reserving in the OS a portion of the main memory to huge pages[3]. This is a hardware mechanism generally available in modern architectures, that allows for the management of memory pages of different sizes, often 2MB pages (and more rarely also 1GB pages) in addition to the standard 4KB ones. This allows for decreasing the pressure on the memory pages descriptor cache, and it becomes particularly effective when there are memory-hungry processes that need to allocate big amounts of RAM, like in the case of KVM processes, each hosting an entire virtual machine, or database processes, that need to optimize read performance by caching large quantities of data in RAM. As a drawback, huge pages reduce the granularity at which memory can be allocated and assigned by the OS to the running processes, thus it is used on big multi-core servers with plenty of RAM available.

Systems with a Non-Uniform Memory Architecture (NUMA) have a number of nodes, where each node includes a set of CPUs and a memory controller with a fastpath switching logic, so that memory from the same node can be accessed faster than the one from different nodes. On these systems, it is commonplace to configure the OS and the software so to deploy a complex software along with most of its memory pages within the same node, to maximize the throughput.

[2]More information at: `https://www.kernel.org/doc/html/latest/admin-guide/xfs.html`.

[3]More information at: `https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt`.

Traditionally, real-time systems have been using priorities to distinguish among processes with various urgency requirements, and CPU schedulers have been exposing to the user-space APIs like the standard `nice`[4] mechanism and the `setpriority()` system call of POSIX OSes, for use by general-purpose applications. However, these have been wildly customized in individual OSes with heuristics that dynamically catch and prioritize interactive workloads over batch computations, causing a somewhat unpredictable behavior. Therefore, real-time systems have been relying on much more predictable real-time scheduling extensions like the `SCHED_RT` and `SCHED_FIFO` POSIX disciplines (POS, 2004), made available through the `sched_setscheduler()` syscall. Some works (Kim et al., 2017) proposed to engineer better heuristics able to differentiate among long-running batch data-intensive activities versus short-lived interactive ones, within the Linux kernel, to dynamically tune prioritization of the disk scheduler in serving I/O requests issued by various tasks, or just adopt a different weighted fair queueing strategy (Valente and Avanzini, 2015) to differentiate among competing tasks. The Linux kernel has the `ionice`[5] subsystem to tune the priority associated to disk requests, similarly to the `nice` one mentioned above. These tricks can be useful to maximize the average performance of a database server, however in order to support differentiated levels of service depending on per-user/request requirements, the software needs to be changed, applying non-trivial modifications to the request processing and handling code path, as proposed in this paper, to be added to other certainly useful mechanisms at the disk access layer, like the just mentioned ones.

3 Background Concepts

This section provides a brief introduction to the MongoDB software, along with details of the CFS Scheduler within the Linux kernel. These are useful for a better understanding of the modifications performed to MongoDB.

3.1 MongoDB

MongoDB is an open-source document-oriented data store often appreciated in comparative studies (Palanisamy and SuvithaVani, 2020) due to its flexibility and ease to use, while offering all the capabilities needed to meet the complex requirements of modern applications. The name MongoDB derives from *humongous*, which can be literally translated to “extraordinarily large", to highlight its capability to store and serve big amounts of data. This ability to efficiently handle large-scale traffic, which is a typical use-case for content-delivery services, is something that relational database technologies are not able to easily achieve: for instance, the MongoDB access speed is 10 times higher than the MySQL one, when the data exceeds 50GB (Rubio et al., 2020). MongoDB stores documents in a format called BSON, a binary-encoded serialization of JSON designed to be efficient both in storage space and scan-speed. A BSON document contains multiple named fields and an automatically generated identifier to uniquely identify it. A field comprises three components: a name, a data type and a value. BSON supports complex JSON data types, such as arrays and nested documents, but also additional types like binary, integer, floating point or datetime. Documents are kept in schemaless tables, called collections, allowing for heterogeneous documents to coexist in the same collection (although similarity is recommended for index efficiency). The users interact with the database system using a query language expressed in JSON that is supplied via API libraries, called *driver*s, available for all the major programming languages. MongoDB supports data durability/availability and easy horizontal scaling through *replication* and *sharding*: the first consists in deploying multiple physical copies of the same database, which together form a *replica set*, while the second consists in deploying a cluster of different databases, called a *sharded cluster*, each storing a subsets of data distributed according to user-defined criteria.

3.2 Linux Scheduler / POSIX Niceness

A typical multi-tasking OS has to service multiple runnable thread or process, often generically referred to as tasks (Anderson and Dahlin, 2014), making sure that each one gets equal opportunity to execute. The component responsible for granting CPU time to the tasks is the scheduler, which chooses the execution order via a scheduling policy. For instance, the Linux kernel provides a scheduling framework that comprises three categories, each suitable for specific use cases: fair scheduling for general-purpose applications, fixed-priority and reservation/deadline-based scheduling for real-time scenarios. The two latter categories are used in typical embedded real-time scenarios where the total real-time workload is known

[4]More information at: `https://www.kernel.org/doc/html/latest/scheduler/sched-nice-design.html`.

[5]More information at: `https://www.kernel.org/doc/html/latest/block/ioprio.html`.

upfront, and failing a proper analysis on the requirements would cause problems that could compromise the functioning of the entire system. However, there are studies (Cucinotta et al., 2019) exploring applicability of these scheduling strategies to deploy highly time-sensitive applications in Cloud infrastructures.

The default Linux scheduler, the *Completely Fair Scheduler* (CFS) (Wong et al., 2008), tries to eliminate the unfairness of a real CPU by emulating an *ideal, precise multi-tasking CPU* in software: namely, a CPU where each task/thread runs in parallel and it is given an equal share of “power". This scheduler tackles the problem by giving CPU access to the task that waited the most. In particular the scheduling order is defined according to the lowest *vruntime*, a per-thread multi-factor parameter that measures the amount of “virtual" time spent on the CPU. There are several parameters taken into account when computing this value, one of these being the *nice* level. This alters the scheduling order by weighting the *vruntime* value: 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) to 19 (lowest priority). 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`[6].

4 Proposed Approach

This section describes the internal components and mechanisms of the MongoDB software that have been examined and how our proposal integrates with them. The considered version of the software is the 4.2 one, which can be found on GitHub[7].

4.1 MongoDB Internals

A MongoDB database system is designed as a configurable client-server architecture. The main component is the *mongod* service, which performs all the core database activities: it handles requests, applies changes to the storage unit and performs the management and logging processes. Our proposal relies upon two design choices regarding the default execution model and the concurrency control mechanism employed by a *mongod* instance, which are leveraged to fully integrate the modifications to the core software with minimal overhead:

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

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

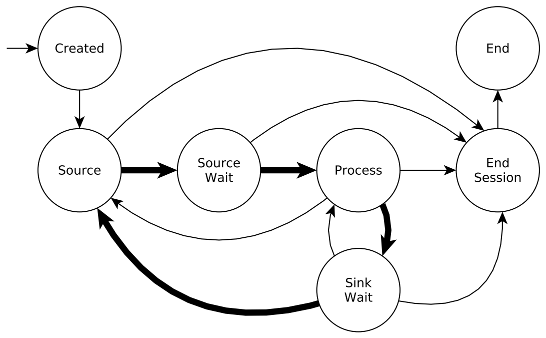


Figure 1: FSM modeling the session life-cycle of a client connection. Highlighted in bold, the *Standard RPC* transition path, which corresponds to the following events: wait for a client request, process it, and send a response back.

1) MongoDB manages networking synchronously: each incoming connection is given its own dedicated server-side *client worker thread* to perform database operations and handle the session life-cycle. The workflow of such worker threads is modeled as a finite-state machine (FSM). The main request-response interaction comprises multiple “walks" over the transition path called *Standard RPC*, which corresponds to the following events: wait for a request from the client, process it, wait for a result from the underlying storage unit, send back a response (Figure 1). The thread is reclaimed when the connection is closed. In short, as of MongoDB Version 4.2, it is safe to assume that a given incoming connection corresponds to a dedicated server-side worker thread.
2) MongoDB uses an optimistic version of the *Multiversion Concurrency Control* mechanism (Bernstein and Goodman, 1983), which allows for lock-free concurrent write operations. More specifically, data consistency is enforced by displaying to the connected users a *version* of the database at a particular instant of time, therefore any changes made by a writer will not be seen by other users until the operation has been completed without conflicts. A write conflict is the result of simultaneous updates to the same document, and is solved by accepting only one write operation as valid and transparently retrying the other ones. Multiple *mongod* services can be instantiated on different physical machines and interconnected via a simple socket-based, request-response style protocol called *MongoDBWire Protocol* to allow for more complex deployments capable of data redundancy and/or horizontal scalability. A group of independent *mongod* instances that maintain the same data set is called a *replica set*: the primary node handles all write operations and records the changes to the data set into an *operations log* (in short, *oplog*); the secondary nodes

replicate the primary's *oplog* and apply the changes asynchronously to their local copies. The members of a replica set communicate frequently via heartbeat messages to detect topology changes and react accordingly: for instance, if the primary node becomes unavailable, the replica set goes through an election process to determine a new one. Both the primary election and the *oplog* replication process are built upon a variation of the RAFT consensus algorithm (Ongaro and Ousterhout, 2016). The *oplog* is a fixed-size collection stored in the *mongod* instance itself, thus its records are handled as normal documents, but with a fixed structure. These *oplog* entries describe the changes applied to the data set in an idempotent format and are uniquely identified by the *opTime* field, a tuple consisting of two values: a timestamp and a node-specific term that identifies the primary node that serviced the write operation. Therefore, this field determines the order in which the operations are carried out. A secondary *mongod* instance performs the replication process using the following components (as depicted in Figure 2):
1) The *OplogFetcher*, that fetches *oplog* entries from the primary by issuing a number of `find` and `getMore` commands to the same endpoint of a user connection. The entries are returned in batches that are then stored in a buffer, called the *OplogBuffer*.
2) The *OplogBatcher*, which pulls the fetched batches off the *OplogBuffer* and creates the next batch of operations to be applied to the local data set replica.
3)The *OplogApplier*, that applies the batches created by the *OplogBatcher* to the local *oplog* and storage unit. In particular, it manages a thread pool of writer threads that, for the sake of performance, may not respect the chronological order of the operations within a batch by applying them in parallel (thus a few operations require singleton batches, like the `drop` one). Whenever a secondary node finishes replicating a batch, it notifies the *opTime* of the last applied entry to the primary. This is crucial for those scenarios where the users request a certain degree of data durability (the *write-concern* level, using MongoDB terminology): the primary node will wait for a user-defined number of such notifications before sending a positive response to a write operation in order to ensure that the change was replicated to a sufficient number of nodes. A high *write concern* value deteriorates throughput, while a low one leads to higher risks of data loss in case of failure. Often, one wants to set the write concern to ensure that the majority of replicas have safely stored the written data before responding to the client. An analogous feature for read operations is also available, the *read concern*, which in turn is used to control data consistency. Both options should be tuned according to the application needs.

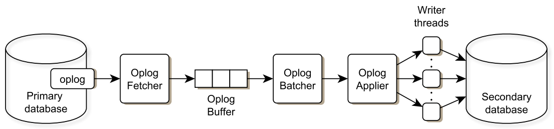


Figure 2: The replication process pipeline: a secondary node fetches the oplog entries from the primary node and stores them, in batches, into a buffer. Afterward, they are pulled and arranged in another set of batches which are applicable in parallel by the writer threads. The latter modify the local copy of the secondary's oplog and database.

4.2 RT-MongoDB

The proposed modified version of the MongoDB software, called *RT-MongoDB*, introduces an application-level notion of *priority* among concurrent requests, realized exploiting: (i) the *nice* scheduling parameter available in the underlying OS scheduler, (ii) a *checkpoint system* described later in this section.

Prioritization is achieved by giving to the user direct control over the nice value of the underlying client worker thread that services its requests. In this way, it is possible to alter the thread scheduling order according to users needs: for instance, the queries for a real-time task could increase the priority of their session by setting a lower nice value, thus reducing the response times even in the presence of an unimportant long-lasting query. We assume that the *RT-MongoDB* daemon is launched with the capability to exploit the entire range of nice values. For the sake of simplicity, the current version of *RT-MongoDB* reduces the range to three values: -20 (*high-priority*), 0 (*normal-priority*) and +19 (*low-priority*). Thanks to the synchronous execution model, it is trivial to identify the target thread without incurring in side-effects, because throughout the whole life cycle of the client session, the underlying worker thread will be always the same. For this reason, the terms “high-priority user" and “high-priority worker thread" will be used interchangeably in what follows. However, achieving differentiated performance by just lowering the nice value for high priority sessions proved to be ineffective in replicated scenarios where data durability is enforced. The key issue here is that the primary *mongod* node must wait for a number of replica nodes to finish replicating the change, before acknowledging it to the requesting user. Since the replica nodes are unaware of the nice levels declared on the primary node, because they are deployed on different physical machines, they perform an unbiased replication. To circumvent this problem, *RT-MongoDB* employs a *soft* checkpoint system to temporarily revoke CPU access to lower priority worker threads. It comprises two primitives, *check-in* and *check-out*, that have been integrated in the life cycle of the client session in order to delimit the start and end point of the *Standard RPC* transition path. The idea is to enforce a prioritized access channel for the time window required to complete a high priority request by putting to sleep the competing, lower priority worker threads. In order to do so, each worker declares their *niceness* before servicing the user request (*check-in*) to eventually get stopped by the checkpoint system, if there are higher priority requests already being processed. Whenever a worker thread completes a request, it invokes the *check-out* primitive to notify possible blocked threads and eventually wake them up, if there are no more higher priority requests (Figure 3). The adjective *soft* refers to the fact that this mechanism does not interfere with certain threads: namely those serving the users wishing to declare a different priority level, the unrelated “service" threads instantiated by the database to manage the deployment, and those worker threads serving the secondary nodes. In fact, the *oplog* fetching operation happens on the same communication channel that the end-users use to interact with the storage unit, and thus it is mandatory to distinguish between an *external*, namely an end-user, and an *internal* client connection. An accidental revocation of CPU access to a secondary node would lead to unexpected slowdowns of the entire system. In conclusion, the checkpoint system provides a prioritized channel to high priority sessions by temporarily restricting the CPU access to lower priority ones.

In terms of API, *RT-MongoDB* offers a new database command, named `setClientPriority`, and a revised version of the `runCommand` command to support differentiated performance on per-user and per-request basis, respectively. Both commands act on the same mechanisms, carry out the same activities and support every operation, but the priority declared with `runCommand` lasts for a *Standard RPC* transition only, thus allowing for a temporary prioritization of the client session, whereas the `setClientPriority` declares the priority of all subsequent requests until the session is closed, or until the priority is changed again.

5 Experimental Results

The proposed approach has been assessed with a set of synthetic stress scenarios using a testing framework built for the occasion. The test environment comprises two twin multi-core NUMA machines interconnected by a 1 gbE connection. The first one has been used to host a 2-member *replica set*, core pinned to different NUMA nodes and configured to use different independent disks, in order to emulate a distributed system. The second machine has been used to host the concurrent users interacting with the database. Each test follows the same workflow: the users connect to the *replica set*, declare their priorities with the `setClientPriority` command, and start submitting concurrently a total of 500 write operations each, with no wait time between subsequent requests from the same user.

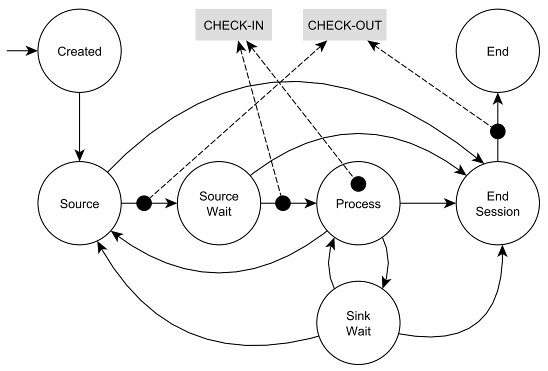
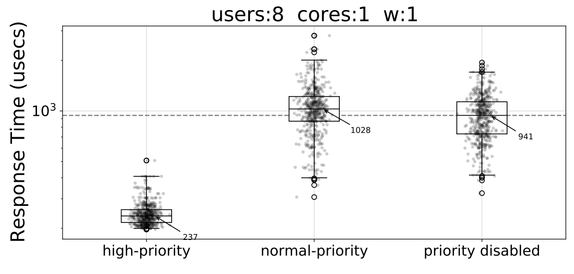


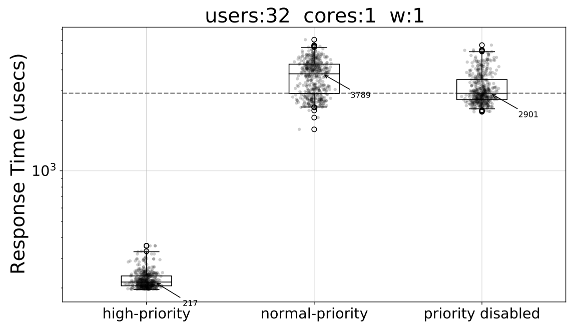
Figure 3: The same finite-state machine proposed in Figure 1, but integrated with the checkpoint system. The primitive *check-in* declares the nice value of the underlying worker threads that is about to service its client. The checkpoint system will block it, if higher priority ones are running. The primitive *check-out* notifies the conclusion of a *Standard RPC* path to the checkpoint system, which will wake up the lower priority threads, if no more higher priority ones are running

From a preliminary assessment, it has been observed that, with this test environment, a 2-member replica set can service a single client request in 200 microseconds on average when data durability is not enforced, assuming that the database has full control over the CPU cores and no other concurrent operation has been issued. Under the same conditions, but with data durability enabled, the response time rises to 3000 microseconds on average. The rest of this section presents a subset of the collected results regarding the distribution of response times over 20 re-runs of the same test during a portion of the so-called *priority window*, namely the time period where a high-priority user is being serviced. More specifically, the analyzed results concern the interval where the lower priority users have yet to be blocked, in order to highlight the moment where performance differs across user requests. The chart used for data visualization is the box-plot with whiskers set to the 5% and 95%

percentiles: each box represents the response time for a given user per priority level, including an indicative box-plot (the rightmost one) to represent the scenario where no prioritization is enforced. This way, it is easier to trace the improvements compared to the original median baseline.



(a)



(b)

Figure 4: Boxplots of the response times (on the Y axis) obtained for a single high-priority client competing with several normal-priority ones (different boxplots) during the *priority window*, compared to the one obtained in the original unmodified MongoDB (rightmost boxplot). The top and bottom subplots refer to scenarios with 8 and 32 total clients, respectively, with MongoDB configured on 1 core and write-concern 1.

The first set of box-plots shown in Figure 4 presents the results for the scenario where the primary node is pinned to one core only, a single user is declared as high-priority and data durability is not enforced. The separation between the high-priority user and the normal-priority ones is quite noticeable, especially in the crowded 32-users scenario, where the normal-priority users lose almost a millisecond with respect to the "no-priority" case and also experience higher variation, while the high-priority user is timely serviced. This is the natural course of events, as depicted in the example timelapse in Figure 5 for the 8-users scenario, which shows the response times distribution over time. The normal-priority users are frequently not allowed to run in order to give precedence to the high-priority users during their *priority window*, and to ensure fairness between same-priority

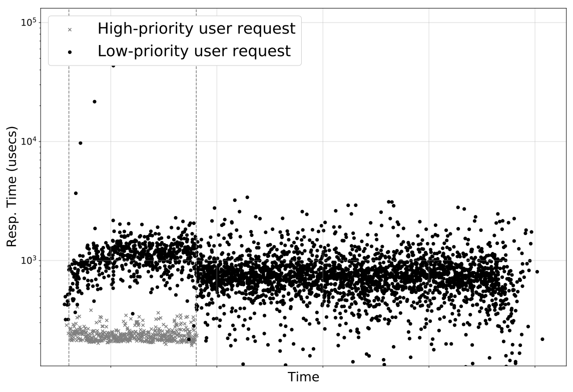
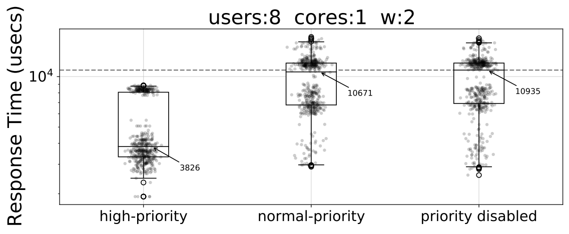


Figure 5: Example timelapse of a run with 1 high-priority client and 7 normal-priority ones with MongoDB configured on 1 core and write-concern 1. The location between dashed vertical lines is the so-called *priority window*, the time interval while the high-priority user is serviced.

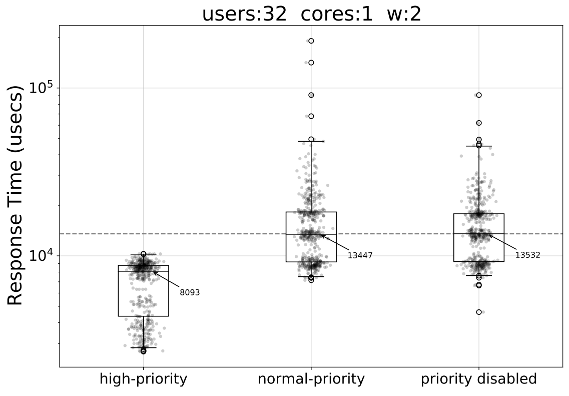
users. While the unrelated "service" threads run by the primary node will also give precedence to the high-priority clients, there are some periodic internal MongoDB tasks that delay the execution of some requests, causing the visible outliers. They could not be blocked as done with lower priority worker threads, not to compromise the database system correct operation.

A similar but smaller improvement can be seen in the second set of box-plots in Figure 6, which displays the results for the same two scenarios, but with data durability enforced. As expected, the values are higher due to the replication process to which the primary must undergo. The response times group in two distinct clusters: it appears that, since the secondary node replicates the changes in batches, a subset of unlucky requests experience higher latency because they are included in big lower-priority batches.

The last set of box-plots in Figure 7 show how *RT-MongoDB* is capable of providing differentiated performance for several users depending on the specified priority level: high, normal or low. In order to evenly service every high priority user, the primary node is allowed to use more than one core. The first scenario has 2 users for each priority level across high, normal and low priority, and the primary node is pinned to a dual-core CPU deployment. The second scenario considers 4 users for each priority level. In both cases the write operations are issued with no data durability requirement. *RT-MongoDB* is capable of making use of the different CPUs and provide almost optimal latencies for normal-priority and high-priority users, but it is evident that the high-priority ones experience less variation, especially in the second scenario.



(a)



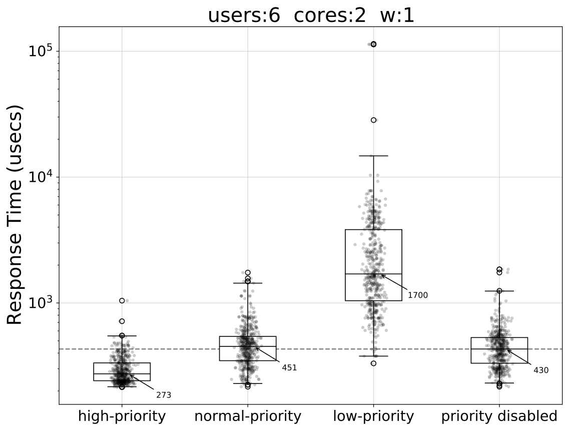
(b)

Figure 6: Boxplots of the response times (on the Y axis) obtained for a single high-priority client competing with several normal-priority ones (different boxplots) during the *priority window*, compared to the one obtained in the original unmodified MongoDB (rightmost boxplot). The top and bottom subplots refer to scenarios with 8 and 32 total clients, respectively, with MongoDB configured on 1 core and write-concern 2.

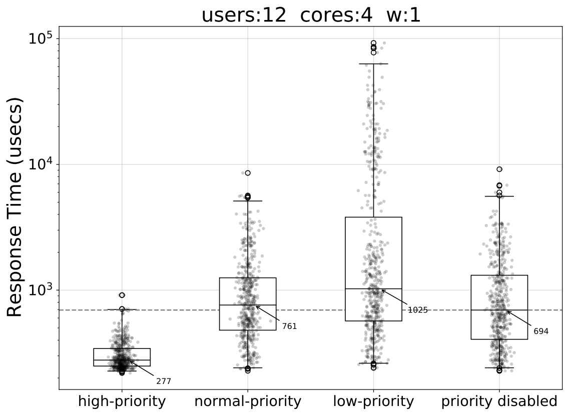
6 Conclusions and Future Work

This paper presents an extension to the MongoDB software that enables differentiated per-user/request performance among competing clients. Preliminary results suggest that the technique seems promising for obtaining a priority-based differentiation of the service level received by multiple clients when accessing a shared MongoDB instance. This is beneficial for real-time cloud applications, such online video streaming or gaming, where more important and/or critical activities need to be prioritized over less important ones.

Concerning possible future works on the topic, it might be interesting to extend the proposed modifications so to achieve a predictable performance in accessing the system, letting clients specify a custom relative deadline within which their submitted requests should be completed. This might be coupled with a formal performance analysis using real-world workloads and larger replica sets, and with the use of more advanced scheduling techniques, like SCHED_DEADLINE, that has been made recently available in the Linux kernel for real-time workloads. Other viable paths of investigations are those mixing prioritization at the CPU scheduling level with the one at the I/O disk access layer (Valente and Avanzini, 2015), tuning both CPU and block layer schedulers, trying to avoid priority inversion scenarios (Kim et al., 2017). In this kind of investigations, performance monitoring and tuning frameworks like the one presented in (Malki et al., 2018), supporting MongoDB among others, may help to gain additional insights about the points of intervention within the system. Another future work would be to address the secu-



(a)



(b)

Figure 7: Boxplots of the response times (on the Y axis) obtained in a given time window with several clients differentiated by priority levels (different boxplots), compared to the one obtained in the original unmodified MongoDB (rightmost boxplot). The top subplot refers to a scenario with 6 users, with 2 users per priority level. The bottom one refers to a 12-users scenario, with 4 users per priority level. Both deployments are configured to offer a core per high-priority client, in order to service evenly all of them. The write-concern is 1.

rity issue of giving access to the proposed feature to all users, by implementing an access-control mechanism (Sandhu, 1998). In fact, the current prototype assumes no greediness, but every user could declare himself as high-priority, making the mechanism ineffective. Therefore, the idea is to give to the MongoDB sysadmin the possibility to restrict the accessible priority levels, depending on system-wide access-control configuration settings.

REFERENCES

(2004). *The Open Group Base Specifications Issue 6 – IEEE Std 1003.1, 2004 Edition*. The IEEE and The Open Group.

(2020). How to Choose Your Red Hat Enterprise Linux File System. [Online] Accessed November 22nd, 2020.

Amirijoo, M., Hansson, J., and Son, S. H. (2006). *IEEE Transactions on Computers, title=Specification and management of QoS in real-time databases supporting imprecise computations*, 55(3):304–319.

Anderson, T. and Dahlin, M. (2014). *Operating Systems: Principles and Practice*, volume 1: Kernel and Processes. Recursive books.

Ara, G., Abeni, L., Cucinotta, T., and Vitucci, C. (2019). On the use of kernel bypass mechanisms for high-performance inter-container communications. In *High Performance Computing*, pages 1–12. Springer International Publishing.

Babu, S. and Widom, J. (2001). Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120.

Baruah, S., Bertogna, M., and Buttazzo, G. (2015). *Multiprocessor Scheduling for Real-Time Systems*. Springer Publishing Company, Incorporated.

Basanta-Val, P., Fernández-García, N., Sánchez-Fernández, L., and Arias-Fisteus, J. (2017). Patterns for distributed real-time stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 28(11):3243–3257.

Bernstein, P. A. and Goodman, N. (1983). Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483.

Bestavros, A., Lin, K.-J., and Son, S. H. (1997). *Real-Time Database Systems: Issues and Applications*. Springer Science+Business Media, LLC.

Buttazzo, G., Lipari, G., Abeni, L., and Caccamo, M. (2005). *Soft real-time systems: Predictability vs. efficiency*. Springer US.

Cucinotta, T., Abeni, L., Marinoni, M., Balsini, A., and Vitucci, C. (2019). Reducing temporal interference in private clouds through real-time containers. In *2019 IEEE International Conference on Edge Computing (EDGE)*, pages 124–131. IEEE.

Garcia-Molina, H. and Salem, K. (1992). Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516.

Jing Han, Haihong E, Guan Le, and Jian Du (2011). Survey on NoSQL database. In *6th International Conference on Pervasive Computing and Applications*, pages 363–366.

Kang, K., Oh, J., and Son, S. H. (2007). Chronos: Feedback control of a real database system performance. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 267–276.

Kao, B. and Garcia-Molina, H. (1994). An Overview of Real-Time Database Systems.

Kim, S., Kim, H., Lee, J., and Jeong, J. (2017). Enlightening the i/o path: A holistic approach for application performance. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 345–358, Santa Clara, CA. USENIX Association.

Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J. M., Ramasamy, K., and Taneja, S. (2015). Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA. ACM.

Lettieri, G., Maffione, V., and Rizzo, L. (2017). A survey of fast packet I/O technologies for Network Function Virtualization. In *Lecture Notes in Computer Science*, pages 579–590. Springer International Publishing.

Li, T., Tang, J., and Xu, J. (2016). Performance modeling and predictive scheduling for distributed stream data processing. *IEEE Transactions on Big Data*, 2(4):353–364.

Li, Y. and Manoharan, S. (2013). A performance comparison of sql and nosql databases. In *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 15–19.

Lindström, J. (2008). *Real Time Database Systems*, pages 1–13. American Cancer Society.

Malki, M. E., Hamadou, H. B., Malki, N. E., and Kopliku, A. (2018). MPT: Suite tools to support performance tuning in noSQL systems. In *20th International Conference on Enterprise Information Systems (ICEIS 2018)*, volume 1, pages 127–134, Funchal, Madeira, PT. SciTePress.

Navrátil, M., Bailey, L., and Boyle, C. (2020). Red Hat Enterprise Linux 7 – Performance Tuning Guide – Monitoring and optimizing subsystem throughput in RHEL 7. [Online] Accessed November 22nd, 2020.

Ongaro, D. and Ousterhout, J. (2016). In search of an understandable consensus algorithm (extended version). *Retrieved July*, 20:2018.

Palanisamy, S. and SuvithaVani, P. (2020). A survey on rdbms and nosql databases mysql vs mongodb. In *2020 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–7.

Rubio, F., ., P. V., and Reyes Ch, R. P. (2020). Nosql vs. sql in big data management: An empirical study. *KnE Engineering*, 5(1):40–49.

Sandhu, R. S. (1998). Role-based access control. In *Advances in computers*, volume 46, pages 237–286. Elsevier.

Theeten, B., Bedini, I., Cogan, P., Sala, A., and Cucinotta, T. (2014). Towards the optimization of a parallel streaming engine for telco applications. *Bell Labs Technical Journal*, 18(4):181–197.

Valente, P. and Avanzini, A. (2015). Evolution of the BFQ Storage-I/O Scheduler. In *2015 Mobile Systems Technologies Workshop (MST)*, pages 15–20.

Wingerath, W., Gessert, F., Friedrich, S., and Ritter, N. (28 Aug. 2016). Real-time stream processing for big data. *it - Information Technology*, 58(4):186 – 194.

Wong, C. S., Tan, I., Kumari, R. D., and Wey, F. (2008). Towards achieving fairness in the linux scheduler. *SIGOPS Oper. Syst. Rev.*, 42(5):34–43.