Differentiated Performance in NoSQL Database Access for Hybrid Cloud-HPC Workloads

Remo Andreoli(✉) and Tommaso Cucinotta

Scuola Superiore Sant'Anna, Pisa, Italy
{r.andreoli,t.cucinotta}@santannapisa.it

Abstract. In recent years, the demand for cloud-based high-performance computing applications and services has grown in order to sustain the computational and statistical challenges of big-data analytics scenarios. In this context, there is a growing need for reliable large-scale NoSQL data stores capable of efficiently serving mixed high-performance and interactive cloud workloads. This paper deals with the problem of designing such NoSQL database service: to this purpose, a set of modifications to the popular MongoDB software are presented. The modified MongoDB lets clients submit individual requests or even carry out whole sessions at different priority levels, so that the higher-priority requests are served with shorter response times that exhibit less variance, with respect to lower-priority requests. Experimental results carried out on two big multi-core servers using synthetic workload scenarios demonstrate the effectiveness of the proposed approach in providing differentiated performance levels, highlighting what trade-offs are available between maximum achievable throughput for the platform, and the response-time reduction for higher-priority requests.

1 Introduction

Cloud Computing has become increasingly popular over the past decade as an affordable solution for all size businesses, thanks to virtualization technologies (and containerization, more recently) that add flexibility in the management of the physical infrastructure. For instance, public cloud providers are able to maximize resource utilization by multiplexing its access across a wide number of tenants that can deploy widely heterogeneous workloads. With the increasingly distributed nature of applications and services, cloud providers have been playing a key role in providing reliable storage solutions, thanks to their ability to replicate data on multiple sites and fault-independent availability zones. Storage services are at the heart of distributed cloud-native and big-data processing applications, where the always increasing need for higher and higher capacity, performance and scalability requirements pushed towards departing from traditional relational data-base architectures to embrace more lightweight, less feature-rich, NoSQL architectures. The selling points of NoSQL data stores is massive scalability and the ability to deal with arbitrary table sizes and numbers of concurrent clients. The historical work in [8] discussed how such data stores can be effectively and productively used in cloud services, without any need for relying on expensive reliable hardware, but achieving reliability by distribution of the workload across various inexpensive nodes. Later, those principles were reused for engineering AWS DynamoDB, the industrial real-time data-base offering in AWS.

In cloud providers, it has become commonplace to find fully managed 24/7 data store services which can be directly and conveniently used by a plethora of clients submitting widely heterogeneous workload patterns. On one hand, the elasticity of the Cloud has brought the traditionally owner-centric High Performance Computing (HPC) community to explore, in a number of cases, cloud-based solutions in order to deal with the emergence of extreme-scale simulations and big-data processing [11,25]. On the other hand, time-sensitive applications are being increasingly hosted in cloud environments, where a great component of the end-to-end latency (and its variability) is due to the time needed to access one or more data stores. Therefore, a cloud provider must be capable of designing evolved NoSQL data store services for virtualized/containerized applications exhibiting highly heterogeneous workloads. These include a mix of high-performance applications and services, that need to process high volumes of data at the maximum average throughput, as well as (soft) real-time ones that need to process relatively smaller amounts of data, but with tight timing constraints for individual requests. A widely adopted mechanism in real-time systems is the use of priorities, so that higher priority workloads may be served earlier than lower-priority ones. In this work, we propose a modification to the popular MongoDB [7] NoSQL data store in order to achieve differentiated per-client and per-request performance using the aforementioned prioritized access principle.

1.1 Contributions

This paper extends our prior work, RT-MongoDB [2], which exploits *UNIX nice level* combining them with two design choices of MongoDB, the per-client threading model and optimistic concurrency control, in order to achieve differentiated performance. The additional modifications we introduced pay special care to avoid penalizing the achievable throughput in presence of mixed-priority workloads, keeping effectiveness in separating performance among different-priority requests. This is crucial when serving both HPC and interactive cloud workloads from the same MongoDB instance. We present for the first time results performed under a heavy-stress for a multi-core MongoDB instance deployed over a pair of 20-cores Xeon-based servers, with a variety of mixed HPC/interactive workload scenarios, showing the effectiveness of the proposed approach.

2 Related Work

In the research literature, a number of proposals can be found dealing with optimizing the performance of databases, including NoSQL data stores for cloud computing, so to support scenarios with mixed workload types and requirements.

For example, the old concept of a real-time database system (RTDBS) [6,15,24] refers to a data management system with predictable timing of the operations requested by clients. Research efforts in this area complemented research on scheduling of processes on the CPU [3], with investigations on scheduling of on-disk transactions. Cumbersome issues that have been tackled in the last decade include: how to work around the high seek latencies of traditional rotational disk drives, dealt with by switching to memory-only data management systems [12], or recently introducing solid-state drives; the presence of dynamic workload conditions, that pushed towards the adoption of adaptive feedback-based scheduling techniques [1,14]. Thanks to their capability of guaranteeing predictable access times, real-time database systems found applications in traditional hard real-time application domains, like mission control in aerospace, process control in industrial plants, telecommunication systems and stock trading [14]. 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.

In contrast, research on distributed and cloud systems has a traditional focus on maximizing average-case performance and overall throughput, neglecting predictability for individual requests. Here, the typical feature-richness and ideal consistency model of relational databases has been progressively dropped, in favour of NoSQL architectures [13,22] with relaxed consistency models that implement essentially reliable/replicated distributed hash tables with the ability to ingest arbitrarily high volumes of data, scaling at will on several nodes.

However, the growing interest in deploying time-sensitive web-based applications and services in cloud infrastructures, led to a return of interest in enriching these lightweight NoSQL databases with predictability features, albeit the focus is not on guaranteeing every single request, but rather to control a sufficiently high percentile of the response-time distributions. For example, the DynamoDB[1] solution from AWS, designed around lessons learnt from the Dynamo project [9], has been among the first solutions with the capability of providing guaranteed levels of read and write operations per second for each table, as required by customers, keeping also a per-request latency (99th percentile) lower than 10ms.

More recently, real-time stream processing [4,17,28] solutions have gained momentum. These process data as soon as it comes from various sources, without necessarily storing data on disk, using arbitrarily complex topologies of processing functions. This way, the results of the computation are made available with a very short latency from when new data arrives. Thanks to cloud technologies, it is possible to let these systems scale to several nodes, so that end-to-end latency

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

can be controlled by applying elastic scaling to individual processing functions. A few works [20,27] tried to build empirical performance models of these system, so to employ more precise control logics for the end-to-end performance.

ZHT [21] is a key-value data store for extreme-scale system services hosted on clouds and supercomputers, designed as a zero-hop distributed hash table. SILT [23] is a high-performance key-value store that uses new fast and compact indexing data structures to balance the use of memory, storage and computations. Xyza [26] proposes an extension of MongoDB that combines multiple concurrency control techniques to achieve high performance and scalability. Moreover, several studies combine MongoDB and Hadoop to build reliable storage solutions for HPC-Cloud environments [10,16].

Along with MongoDB, another NoSQL data store that gained significant popularity is Cassandra [18]. This has become also an interesting target for researchers willing to demonstrate applicability of differentiated performance methods. For example, a quality-of-service aware allocator (AQUAS) [29] has been proposed, with the capability of allocating physical resources in a Cassandra deployment to satisfy individual clients' performance requirements.

3 Proposed Approach

This paper extends the modified version of MongoDB we presented in [2], improving the flexibility in replicated scenarios where higher data durability is required. The key to ensure reliability and data consistency in MongoDB is to deploy a *replica set*, a group of multiple database instances residing on different physical machines that share the same data set, and then reply to a user write request only after a number of such instances have locally replicated the operation. In this system configuration, the write operations are all issued to the same database instance, called the *primary* node, which logs them in the *oplog*, a MongoDB capped collection that stores the history of logical writes to the database. Each oplog entry is paired with a timestamp in order to assert the operations order and avoid data corruptions. The other MongoDB instances, namely the *secondary* nodes, rely on the oplog to replicate the state of the data set. For the sake of performance, the replication internals elaborate new oplog entries in batches, carefully crafted by the so-called *OplogBatcher* to be applicable in parallel to the local storage unit. Thus each batch represents a "limbo" state where chronological order is not enforced. While this certainly leads to a mismatch between how history is depicted by each replica node, the underlying storage unit, WiredTiger, stores multiple versions of the data in a tree-based structure [5] and thus is able to return the correct state of the data set to the user, based on the query timestamp.

These observations suggest that it is not possible to properly propagate priorities to the secondary nodes[2] due to the inevitable priority inversion of using batches: this is particularly noticeable when enforcing data durability, because the database does not reply to users until the secondaries finish replicating the

[2] As of MongoDB v4.4.

corresponding batch. The checkpoint system conceived in our previous work attempts to minimize this *unbiased replication problem*, by providing a prioritized channel to temporarily revoke the database access based on the user priority: *low-priority*, *normal-priority* or *high-priority*. In practice, the checkpoint system defines an entry and exit point to the processing state of the user session life-cycle through the means of two primitives, *check-in* and *check-out* (Fig. 1). The checkpoint eventually blocks the underlying worker thread in charge of serving a certain user, based on these simple rules: low priority users are blocked whenever normal or high-priority requests are being processed, while normal priority requests are blocked when high-priority ones are being processed. The major drawback of this solution is the impact on overall throughput: the lower-priority requests are completely halted at the checkpoint if there is even just 1 higher priority request under processing. This design choice effectively reduces the parallelism capabilities of the MongoDB architecture, which on HPC workloads with many-core servers is excessively penalizing, considering the underlying hardware. This paper proposes two additions to RT-MongoDB in order to tackle the issue:

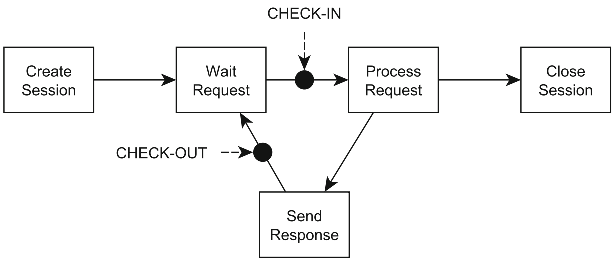

Fig. 1. A simplified view of the user session life-cycle, highlighting the positions of the checkpoint entry and exit point. Each user connection to the database is handled by a distinct worker thread

1. A customizable *checkpoint activation threshold* to specify the maximum number of high-priority requests under processing below which lower-priority requests are still brought forward at the checkpoint. This option allows to fine-tune the drop in parallelism: a high value implies a lower rate of activations, whereas a low value implies a higher rate.
2. A collection of *CPU pools*, user-defined CPU sets that specify the *working space* of the underlying threads based on user priorities, so that it is possible to restrict the number of cores available to those threads serving lower-priority users whenever higher ones are being processed. In this way, high-priority users are serviced with reduced interference, without necessarily halting completely the lower-priority ones.

Regarding the first point, it might be useful to set a high checkpoint activation threshold in scenarios where data durability is not required, since the primary node does not have to wait for the replication process to finish, and thus priority inversion does not occur. Regarding the last point, the current prototype allows for three CPU pools: the *restricted pool*, which is used by lower-priority users whenever higher ones are being processed; the *priority pool*, dedicated to high-priority users for the duration of their session; the *standard pool*, which serves the lower-priority users whenever higher priority sessions are present but not in processing state. The idea is to allocate a sufficient number of cores for high-priority worker threads and a very small restricted working space, so that lower-priority ones are slowed down (but not completely halted). Figure 2 shows a possible allocation of working spaces on a 10-core CPU. Note that this novel mechanism is integrated to the checkpoint system, thus the migration between working spaces follows the workflow already depicted in Fig. 1.

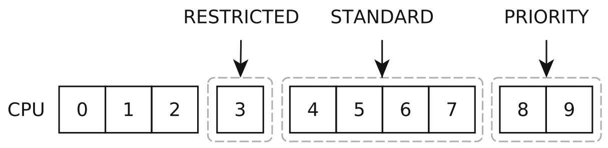

Fig. 2. A possible working space allocation on a 10-core CPU: 1 physical core for the restricted pool, 4 for the standard pool and 2 for the priority pool

In conclusion, the goal is to offer a pair of parameters to database administrators so that they can adjust the trade-off between reduced response times for priority users and total throughput, depending on the scenario. The user API remains mostly unchanged from the previous version of RT-MongoDB, with the only difference being the addition of the `setCpuPools` command to configure the working spaces. A user alters the throughput of its queries by specifying its priority via the `setClientPriority` command. Note that the setting persists for the duration of the session, or until it is changed again. In order to prioritize a single request only, the client should issue the query with `runCommand`, specifying the optional `priority` parameter.

4 Experimental Evaluation

The modifications to MongoDB described in Sect. 3 have been experimentally tested with various (synthetic) stress workloads and the resulting performance has been compared with the original MongoDB. The test environment comprises three distinct multi-core server-class NUMA machines: two 20-core servers (Dell R630 with 2 Intel Xeon E5-2640 CPUs and 64 GB of RAM) to host a 2-member replica set and one 96-core (Arm 64 server with 2 ThunderX 88XX CPUs and 64 GB of RAM) to execute the client processes concurrently. Note that of the 20 cores, 4 will be dedicated to the mandatory MongoDB activities that are not related to the management of a user connection in order to avoid unnecessary CPU contention. Moreover, hyperthreading and turbo-boost are disabled on every machine, and each user process on the 96-core server is pinned to a different physical core in order to simulate an isolated scenario with stable performance.

The stress workload consists of write operations only, since in high concurrency scenarios they put more stress than reads on the database. Every operation waits for the changes to be replicated to the secondary node, in order to simulate a high reliability scenario. Our experimental scenarios are composed of the following steps: each user declares its priority (*high-priority* or *normal-priority*, for simplicity) with `setClientPriority`, waits a randomized amount of time uniformly distributed between 0 and 300 µs and then issues 1000 insert operations, sequentially and without delays in-between. The goal of the following experiments is to show how the trade-off between reduced response times and overall throughput can be adjusted.

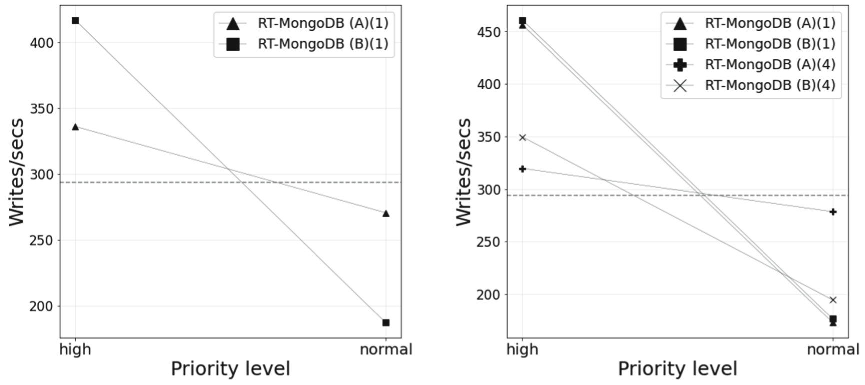

Fig. 3. Writes per second comparison in the 2 cases of: 1 high-priority and 29 normal-priority concurrent users using two different parameter configurations (left); 4 high-priority and 26 normal-priority concurrent users using four different parameter configurations (right). The dashed line corresponds to the average write throughput of the original version of MongoDB

The first two charts, depicted in Fig. 3, show how the average throughput is affected due to priorities in a 30-user scenario. Each line plot corresponds to a different parameter configuration of RT-MongoDB: (A) does not use CPU pools, (B) defines a restricted, standard and priority pool of respectively 2, 10 and 4 physical cores, leaving the remaining 4 for the auxiliary MongoDB activities. The activation threshold is specified in a similar manner: for instance, the configuration (A)(4) corresponds to RT-MongoDB with no CPU pools and checkpoint activation threshold of 4. Figure 3 (left) presents the average throughput of 29 normal-priority users concurring with 1 high-priority one. In both configurations, the checkpoint system activates as soon as a priority request is being processed, thus threshold activation is set to 1 and (A)(1) corresponds to the version of RT-MongoDB proposed in our previous work. The configuration with CPU pools

(B) is able to service the high-priority user at higher rates with respect to (A), achieving 80 more operations per second at the cost of drastic reduction in performance for the remaining users. This is due to the fact that lower-priority users are forced to compete over the 10 cores of the standard pool, while the high-priority user has is own interference-free working space. Note that in this case the restricted pool is never used due to the activation threshold configuration.

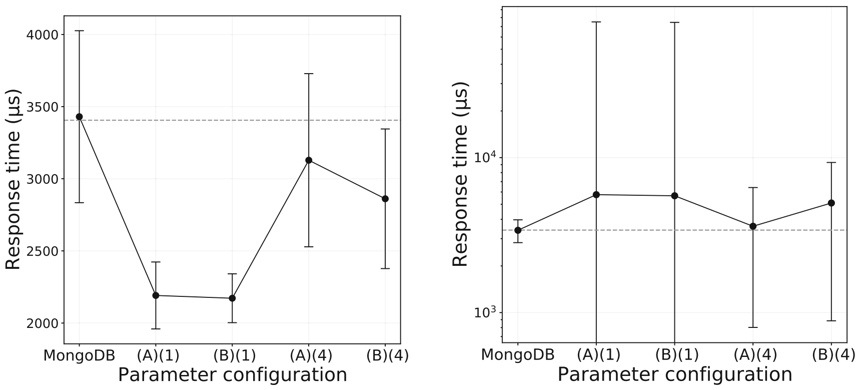

Fig. 4. Average response time and variance for a high-priority user (left) and normal-priority user (right). The scenario comprises 4 high-and 26 normal-priority users. The dashed line and the first data point corresponds to the average response time and variance of the original version of MongoDB

The following paragraph describes a more interesting scenario involving multiple concurrent high-priority users, thus allowing for a greater customization on the activation threshold. Figure 3 (right) presents a 30-user scenario similar to the previous experiment, but with 4 high-priority users. The two configurations with activation threshold (1) achieve similar results, because the checkpoint activates whenever the database receives at least one high-priority request: since this happens with high probability at any given time in the case of 4 concurrent high-priority users, the CPU pool parameter (B) is useless. Consequently, the throughput for high-priority users is very high, because they monopolize the database access. The other configurations show that it is possible to adjust the trade-off between overall throughput and responsiveness for high-priority requests: for example (A)(4) services high-priority requests with a higher rate with respect to the original MongoDB, while still being able to provide a good average throughput for the other users, because they are allowed to be serviced whenever at least one high-priority request is not being processed. It is reasonable to think that the variance for high-priority users is higher using the more "relaxed" configurations, as clearly depicted in Fig. 4, which presents the average response time and variance for high and normal priority users (left and right plots, respectively). The average variance for high-priority users is of 600 microseconds in the worst case scenario, which is equal to the variance experienced by a user using an unmodified version of MongoDB, while still perceiving better response times.

Summarizing the above results, the new version of RT-MongoDB is capable of better adapting to the requirements of HPC-Cloud workloads: one can use a very confined working space (restricted resources) for lower-priority users and 1 as activation threshold to reduce response times of higher-priority requests to a minimum, or fine-tune the available parameters to avoid an excessive degradation in the overall throughput of the system.

5 Conclusions

This paper discussed improvements to our RT-MongoDB variant of the MongoDB database, to enable differentiated per-user/request performance on a priority basis. The focus was to adapt it to the key requirements of hybrid HPC-Cloud workloads, providing the database administrator with a basic set of Quality-of-Service parameters to tune the trade-off between reduced response time for high priority queries and overall system throughput.

A future work is to couple RT-MongoDB with the use of more advanced scheduling techniques, like SCHED_DEADLINE [19], and provide a similar interface to that of DynamoDB, where the users declare the number of read/write requests so as to allow the system to set up a reasonable end-point with performance guarantees.

References

1. Amirijoo, M., Hansson, J., Son, S.H.: Specification and management of QoS in real-time databases supporting imprecise computations. IEEE Trans. Comput. **55**(3), 304–319 (2006)
2. Ferguson, D., Pahl, C., Helfert, M. (eds.): CLOSER 2020. CCIS, vol. 1399. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72369-9
3. Baruah, S., Bertogna, M., Buttazzo, G.: Multiprocessor Scheduling for Real-Time Systems. ES, Springer, Cham (2015). https://doi.org/10.1007/978-3-319-08696-5
4. Basanta-Val, P., Fernández-García, N., Sánchez-Fernández, L., Arias-Fisteus, J.: Patterns for distributed real-time stream processing. IEEE Trans. Parallel Distrib. Syst. **28**(11), 3243–3257 (2017)
5. Bernstein, P.A., Goodman, N.: Multiversion concurrency control-theory and algorithms. ACM Trans. Database Syst. **8**(4), 465–483 (1983)
6. Bestavros, A., Lin, K.J., Son, S.H.: Real-Time Database Systems: Issues and Applications. Springer, New York (1997). https://doi.org/10.1007/978-1-4615-6161-3
7. Chodorow, K.: MongoDB: The Definitive Guide: Powerful and Scalable Data Storage. O'Reilly Media, Inc., New York (2013)
8. DeCandia, G., et al.: Dynamo: Amazon's highly available key-value store. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, pp. 205–220. SOSP 2007. Association for Computing Machinery, New York, NY, USA (2007)

9. DeCandia, G., et al.: Dynamo: Amazon's highly available key-value store. SIGOPS Oper. Syst. Rev. **41**(6), 205–220 (2007)
10. Dede, E., Govindaraju, M., Gunter, D., Canon, R.S., Ramakrishnan, L.: Performance evaluation of a MongoDB and Hadoop platform for scientific data analysis. In: Proceedings of the 4th ACM Workshop on Scientific Cloud Computing, pp. 13–20. Science Cloud 2013. Association for Computing Machinery, New York, NY, USA (2013)
11. Fox, G., Qiu, J., Jha, S., Ekanayake, S., Kamburugamuve, S.: Big data, simulations and HPC convergence. In: Rabl, T., Nambiar, R., Baru, C., Bhandarkar, M., Poess, M., Pyne, S. (eds.) Big Data Benchmarking, pp. 3–17. Springer, Cham (2016)
12. Garcia-Molina, H., Salem, K.: Main memory database systems: an overview. IEEE Trans. Knowl. Data Eng. **4**(6), 509–516 (1992)
13. Han, J., Haihong, E., Le, G., Du, J.: Survey on NoSQL database. In: 6th International Conference on Pervasive Computing and Applications, pp. 363–366 (2011)
14. Kang, K., Oh, J., Son, S.H.: Chronos: feedback control of a real database system performance. In: 28th IEEE International Real-Time Systems Symposium (RTSS 2007), pp. 267–276 (Dec 2007)
15. Kao, B., Garcia-Molina, H.: An overview of real-time database systems. In: Halang, W.A., Stoyenko, A.D. (eds.) Real Time Computing, pp. 261–282. Springer, Heidelberg (1994)
16. Karim, L., Boulmakoul, A., Lbath, A.: Real time analytics of urban congestion trajectories on Hadoop-MongoDB cloud ecosystem. In: Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing, ICC 2017. Association for Computing Machinery, New York, NY, USA (2017)
17. Kulkarni, S., et al.: Twitter heron: stream processing at scale. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 239–250, SIGMOD 2015. ACM, New York, NY, USA (2015)
18. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
19. Lelli, J., Scordino, C., Abeni, L., Faggioli, D.: Deadline scheduling in the linux kernel. Softw. Pract. Exper. **46**(6), 821–839 (2016)
20. Li, T., Tang, J., Xu, J.: Performance modeling and predictive scheduling for distributed stream data processing. IEEE Trans. Big Data **2**(4), 353–364 (2016)
21. Li, T., Raicu, I.: Distributed NoSQL storage for extreme-scale system services. IEEE/ACM Supercomputing Ph.D. Showcase (2015)
22. Li, Y., Manoharan, S.: A performance comparison of SQL and NoSQL databases. In: 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), pp. 15–19, August 2013
23. Lim, H., Fan, B., Andersen, D.G., Kaminsky, M.: Silt: a memory-efficient, high-performance key-value store. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp. 1–13, SOSP 2011. Association for Computing Machinery, New York, NY, USA (2011)
24. Lindström, J.: Real Time Database Systems, pp. 1–13. American Cancer Society (2008)
25. Netto, M.A.S., Calheiros, R.N., Rodrigues, E.R., Cunha, R.L.F., Buyya, R.: HPC cloud for scientific and business applications: taxonomy, vision, and research challenges. ACM Comput. Surv. **51**(1), 1–29 (2018)

26. Patel, Y., Verma, M., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Revisiting concurrency in high-performance nosql databases. In: 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 2018). USENIX Association, Boston, MA, July 2018. https://www.usenix.org/conference/hotstorage18/presentation/patel
27. Theeten, B., Bedini, I., Cogan, P., Sala, A., Cucinotta, T.: Towards the optimization of a parallel streaming engine for telco applications. Bell Labs Tech. J. **18**(4), 181–197 (2014)
28. Wingerath, W., Gessert, F., Friedrich, S., Ritter, N.: Real-time stream processing for big data. IT - Inf. Technol. **58**(4), 186–194 (2016). https://www.degruyter.com/view/journals/itit/58/4/article-p186.xml
29. Xu, C., Xia, F., Sharaf, M.A., Zhou, M., Zhou, A.: Aquas: a quality-aware scheduler for NoSQL data stores. In: 2014 IEEE 30th International Conference on Data Engineering, pp. 1210–1213 (2014)