# A Case for Dynamically Programmable Storage Background Tasks

Ricardo Macedo
*INESC TEC and U. Minho*
ricardo.g.macedo@inesctec.pt

Alberto Faria
*INESC TEC and U. Minho*
alberto.c.faria@inesctec.pt

João Paulo
*INESC TEC and U. Minho*
joao.t.paulo@inesctec.pt

José Pereira
*INESC TEC and U. Minho*
jop@di.uminho.pt

**Appears in the Proceedings of the 2019 38th International Symposium on Reliable Distributed Systems Workshops**

***Abstract*—Modern storage infrastructures feature long and complicated I/O paths composed of several layers, each employing their own optimizations to serve varied applications with fluctuating requirements. However, as these layers do not have global infrastructure visibility, they are unable to optimally tune their performance. Background storage tasks, in particular, can rapidly overload shared resources, but are executed either periodically or when a certain threshold is hit regardless of the overall load on the system.**

**In this paper, we argue that to achieve optimum holistic performance, these tasks should be dynamically programmable and handled by a controller with global visibility. To support this claim, we evaluate the impact on performance of compaction and checkpointing in the context of HBase and PostgreSQL. We find that these tasks can respectively increase 99th percentile latencies by 955.2% and 61.9%. We also identify future research directions to achieve programmable background tasks.**

***Index Terms*—Storage Systems, Background Tasks, I/O Interference, Programmable Storage, Software-Defined Storage**

## I. Introduction

A massive, ever-growing amount of digital information is continuously generated, stored, and processed in both public and private premises [1]. The consequent continual need for greater storage and processing capacity has significantly increased the complexity of the underlying infrastructures. These feature long and complicated I/O paths composed of several layers (*e.g.*, hypervisors, operating systems, device drivers, I/O schedulers, file systems, databases), each employing its own optimizations (*e.g.*, caching [2], I/O sequentialization [3], task prioritization [4]) to serve a myriad of applications with requirements that fluctuate over time.

However, as these layers do not have global infrastructure visibility, they are unable to tune their behavior in order to achieve optimal system-wide performance. Thus, the configuration of each layer is currently done on an individual basis. This design means that ensuring optimal end-to-end performance is hard, and if not correctly assessed can lead to high levels of I/O interference and performance degradation [4, 5]. This effect becomes further amplified when concurrent I/O services operate over the same shared resources, either coming from other layers of the I/O stack or even from internal background activities of a given layer that are competing for resources with the corresponding foreground tasks.

In particular, background tasks, such as compaction [6], checkpointing [7], and replication [8], are predefined I/O-intensive activities that can rapidly overload shared resources, introducing significant I/O interference and workload burstiness, ultimately impacting overall throughput and latency. Currently, in order to minimize interference with foreground activities, background tasks are processed in a best-effort manner, being executed either periodically or when a certain threshold is hit. In many cases, these tasks are also rate-limited to avoid impacting other foreground activities. Again, the decision on when and how to execute such operations is defined by the layer itself regardless of the overall load on the infrastructure at the time.

In this paper, we argue that in order to achieve optimum holistic performance, storage background tasks should be dynamically programmable and their execution should be handled by a control module with system-wide visibility and holistic I/O control. This means that each layer should expose the set of supported background tasks and corresponding configurations to such a controller, which in turn would provide the building blocks for executing each task with optimal I/O performance. Such a programmable environment is aligned with the one proposed by current Software-Defined Storage (SDS) solutions, which ensure holistic control of the I/O stack by breaking the vertical alignment of conventional infrastructures and separating storage policies from the control mechanisms that employ them, thus ensuring Quality of Service (QoS) provisioning, performance isolation, and resource fairness [5, 9].

To support this claim, we evaluate the impact of different background tasks, namely compaction and checkpointing, and corresponding configurations on the performance of two data stores: *HBase*, a highly-available distributed key-value store, and *PostgreSQL*, a traditional SQL database. Under *HBase* deployments, results show that background tasks introduce a performance degradation of up to 87.3% and 955.2% for mean and 99th percentile latencies, respectively. Under *PostgreSQL* deployments, mean and 99th percentile latencies experienced a performance degradation of at most 20.7% and 61.9% respectively, when executed under varying background task settings.

Through the observed results we further support the previous claim and provide future research directions to achieve the envisioned design. The full evaluation results are publicly available at https://rgmacedo.github.io/drss19-website.

The remainder of the paper is structured as follows. §II surveys related work. §III depicts the general organization of

the I/O stacks used. §IV describes the evaluation methodology, while §V presents the obtained results. §VI discusses the results and current challenges, and concludes the paper.

## II. Related Work

Recent works on the SDS paradigm have been proposed to ensure holistic control of the I/O stack by breaking the vertical alignment of conventional infrastructures and separating storage policies from the mechanisms that employ them [5, 9]. However, current solutions do not consider the programmability and holistic control of background tasks, thus limiting their ability to enforce storage policies at higher levels of performance and QoS provisioning.

The impact of mixed background and foreground tasks in storage I/O performance has already been contemplated in the literature [4, 10]. These studies are mainly focused on the I/O stack of modern operating systems, *i.e.*, kernel caching, file system, and block device layers, and propose novel scheduling solutions for orchestrating I/O resources across the different tasks according to their priority. Similarly, several studies have investigated the root causes of tail latency for different I/O layers (*e.g.*, Linux schedulers, file systems) while taking into account the performance impact and I/O interference generated from background processes, such as garbage collection and file system initialization [11, 12]. For the *HBase* software stack, previous studies have drawn conclusions on the overall I/O impact induced by background compaction processes when performed in local and distributed settings [13].

This paper aims at showing that background tasks are not limited to the above systems and that, in many cases, these must be efficiently managed at the user space level. The conducted study and the corresponding observations allow us to argue two main points that are not contemplated by previous studies. First, for complex applications, background and foreground tasks need to be considered in a holistic fashion, considering both kernel and user space components that may have an impact in the shared I/O resources. Secondly, simply applying scheduling and prioritization mechanisms may not be sufficient to ensure complex storage policies for applications and users (*e.g.*, enforcing $X^{th}$ percentile latency under high I/O interference). We argue that this will only be possible by increasing the programmability of critical I/O components. To the best of our knowledge, these points are not addressed by previous work.

## III. Scope

Modern storage infrastructures comprehend multiple independent layers throughout the I/O path. Typical I/O stack settings of these infrastructures can be composed by applications, databases, caches, file systems, and storage devices. Figure 1 illustrates two examples of such stacks. The left side depicts a typical *Hadoop* deployment, composed by different I/O applications that submit read and write requests to *HBase*, a distributed, scalable, open-source NoSQL data store, which in turn reads/writes data from/to *HDFS*, a highly available distributed file system, then backed by a local file system.

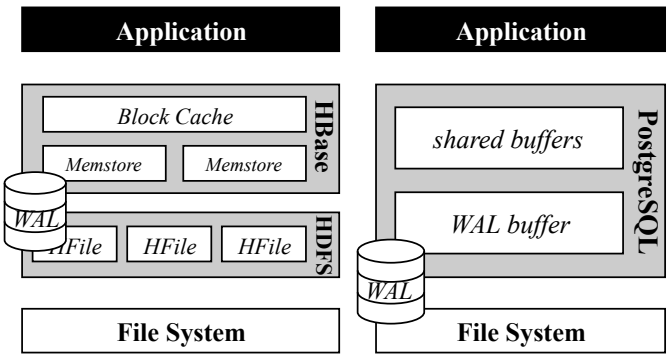


Fig. 1. I/O stack of both *HBase* and *PostgreSQL* deployments.

On the right side, a *PostgreSQL* relational database serves incoming requests of different applications, and is directly backed by a local file system.

### A. HBase

A typical *HBase* deployment is composed by two main components, an *HMaster* and several *RegionServers*. *RegionServers* are the data units of the database, being responsible for serving read and write requests to applications, while the *HMaster* is responsible for coordinating the overall database infrastructure and redirecting clients to the *RegionServer* that holds their data. Further, to improve performance, database tables are horizontally partitioned by row key range into *Regions*, which are then assigned to *RegionServers*.

For write requests, applications submit a key-value pair to the *RegionServer*, being first written to a Write Ahead Log (WAL) file that is used to persist all data on permanent storage and assist on data recovery in case a server fails. Once written in the WAL, data is then written to a write-oriented caching instance, known as *Memstore*, returning back to the application an acknowledgment message. The *Memstore* holds sorted key-value pairs of write requests. Moreover, each *Region* comprises several *Memstores*, each for different parts of the table. When the *Memstore* reaches a predefined size (*e.g.*, 128 MiB), *Memstore*'s data will be flushed to *HDFS* as a new *HFile*, which will then be persistently written at the underlying file system. Internally, similarly to other data stores [6, 14], *HBase* stores its data in a Log-Structured Merge (LSM) tree [15].

For reads, requests are first sent to a read-oriented caching instance, known as *Block Cache*. If the records are not found, it will then check the *Memstore*. If records are held in neither of these caching instances, *HBase* will then read a number of *HFiles* until records are found. After successfully reading the value, it will be inserted into the *Block Cache* and sent back to the application.

As observed, the generation of different *HFiles* per *Memstore* leads to multiple files to be examined during read requests, thus experiencing read amplification effects. To address this problem, *HBase* executes a background compaction process that merges a number of small-sized *HFiles* into

fewer large-sized ones — *minor compaction*. Moreover, *HBase* provides an additional compaction process, namely *major compaction*, that merges and rewrites all *HFiles* comprehended on a *Region* into a single large file, removing in the process all deleted entries. While significantly improving read performance, compaction processes may severely overload I/O resources, namely disk and network, introducing increased I/O interference and burstiness.

### B. PostgreSQL

Regarding the *PostgreSQL* I/O stack, a typical deployment is composed by a database server that efficiently handles requests from multiple applications with different workload profiles. To access the database, a client connects to a running *postmaster* that establishes a communication channel between the application and the database.

For write operations (*e.g.*, INSERT/UPDATE/DELETE statements), the database server maps the corresponding blocks in the *shared buffers* memory area and directly modifies them. The changes are written to a WAL buffer, usually as logical deltas, but in some cases by fully copying the complete physical 8KiB block (*i.e.*, immediately after a checkpoint).

On commit, changes are then persistently written to a WAL file on disk. Directly writing data blocks to the respective data files would lead to significant performance overhead due to the random writes experienced at the disk, and endanger recovery as writing 8KiB blocks cannot be done atomically with common file system semantics. Instead, writes at the WAL file are sequential and can be truncated to include only complete records. Dirty blocks in shared memory are then eventually written to the respective data files by a separate background process, becoming available again for other uses.

For read operations (*e.g.*, SELECT statements), required file blocks are also mapped to *shared buffers*, if not already present. They are thus fetched from the kernel caching layer or read from disk. When the block is no longer needed, and if it is not dirty as a consequence of concurrent write operations, it can be immediately removed from *shared buffers* and remains only in the operating system cache.

In order to truncate the log and allow a fast recovery, *PostgreSQL* periodically performs *checkpoints*, a synchronization event that flushes all dirty data pages in the shared buffer to disk. Such an event guarantees that in case of a failure, a crash recovery procedure seeks for the latest checkpoint record to determine from which point it should start the REDO operation. Checkpointing tasks are conducted in the background, and begin either when the WAL file is about to exceed a certain size (1 GiB by default), or upon a checkpoint timeout (5 minutes by default). Setting these values to a lower bound will lead *PostgreSQL* to conduct checkpoints more often, allowing the database to perform faster recovery upon failure. Interestingly, in order to avoid high I/O interference and workload burstiness, *PostgreSQL* throttles the write performance of checkpoints, leading to dirty buffers to be written over a predefined period of time. To balance these trade-offs, *PostgreSQL* provides a *checkpoint completion target* parameter that allows database administrators to adjust the throughput at which checkpoints are made. When setting a smaller bound (*e.g.*, 0.1), checkpoints will be made faster, creating I/O burstiness. On the other hand, setting a higher bound (*e.g.*, 0.9) will ensure sustained performance and reduced I/O interference for foreground activities, leading however to lower recovery times upon a failure.

## IV. Methodology

In order to illustrate the impact of background tasks on storage systems performance and underline the importance of making them programmable, we evaluated the overhead imposed by each of the two types of background task described above. Here, we describe the adopted evaluation methodology, which in particular aims to answer the following questions:

- *How much overhead do these background tasks impose?*
- *How does their overhead vary across operation types?*
- *How does their overhead vary across time?*
- *How do these tasks impact tail latencies?*
- *How do background task configuration parameters impact performance?*

*a)* ***Evaluated deployments****:* To quantify the overhead imposed by the compaction process, we considered a local deployment consisting of an *HBase* v2.0.5 instance (1 *HMaster*, 1 *RegionServer*, and 1 *Zookeeper* instance, all in the same machine; dedicated heap size = 4 GiB, *Memstore* size = 0.4, block cache size = 0.25), backed by an *HDFS* v2.9.2 instance (1 *NameNode* and 1 *DataNode*, both in the same machine as *HBase*; replication factor = 1; block size = 128 MiB; dedicated heap size = 1GiB), in turn backed by an ext4 file system with default configurations.

In turn, to characterize the impact of checkpointing on performance, we considered a local deployment consisting of a *PostgreSQL* v11.3 instance backed by an ext4 file system, also with default configurations.

*b)* ***Workloads****:* The aforementioned deployments were evaluated under several workloads generated using YCSB v0.15.0 [16] (running locally with the deployments), with different operation type proportions and access distributions:

- *Workload A:* 50% read, 50% update, Zipfian;
- *Workload B:* 100% update, Zipfian;
- *Workload C:* 100% read, uniform;
- *Workload D:* 5% read, 95% insert, Zipfian;
- *Workload E:* 95% scan, 5% insert, Zipfian;
- *Workload F:* 50% read, 50% read-modify-write, Zipfian.

These workloads were previously used in [17], with the exception of workload C, which we modified to follow a uniform distribution, instead of the Zipfian distribution employed by the remaining workloads, as it would also be interesting to analyze a different access pattern. Workloads were executed with both 1 and 10 threads.

We performed between 3 and 9 runs per combination of deployment, configuration, workload, and number of threads. A loading phase was conducted before each run, populating the database with 12.5 M records (using approximately 16 GiB

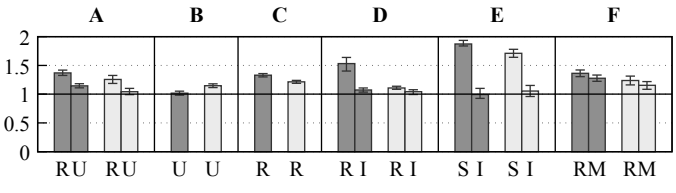


Fig. 2. *HBase* mean latency under configurations with compaction effects normalized against configurations without compaction effects, under workloads with 1 thread ( ) and 10 threads ( ). Error bars represent the 95% confidence interval. (`R` – read, `U` – update, `I` – insert, `S` – scan, `M` – read-modify-write)

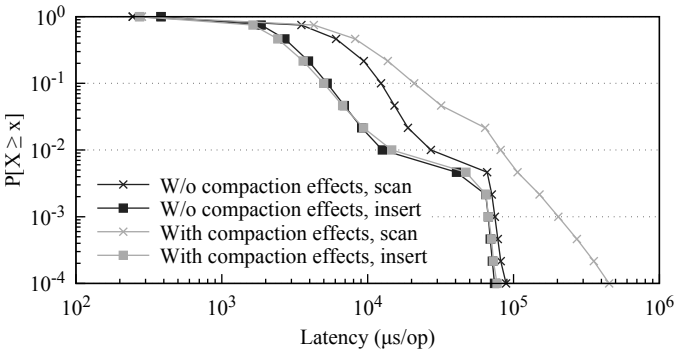


Fig. 3. Complementary cumulative distribution function for the latency of scan and insert operations attained by both *HBase* configurations under a single run of workload E with 10 threads.

of disk space). After each run, the file system was recreated, caches were purged, and a 2 minute cool down period was given. Runs were configured to end after 10 M operations were performed or 17 minutes had elapsed.

*c) **Deployment configurations**:* The *HBase* deployment was evaluated under two scenarios (which we refer to as configurations): (1) *with compaction effects*, performing each workload immediately after the corresponding loading phase is done and thus evaluating the deployment under the effect of ongoing compactions; and (2) *without compaction effects*, waiting 30 minutes after the loading phase — enough time for resulting compactions to finish. This allowed us to measure the overhead imposed by the compaction process.

To understand the performance impact of checkpointing, we evaluated 6 configurations of the *PostgreSQL* deployment, each with a different combination of values for the *maximum WAL size* parameter — either 128 or 1024 MiB — and the *checkpoint completion target* parameter — 0.1, 0.5, or 0.9. We will denote these configurations by $(x, y)$ tuples, where $x$ represents the maximum WAL size (in MiB) and $y$ is the completion target value. The default *PostgreSQL* settings correspond to the $(1024, 0.5)$ configuration.

*d) **Collected metrics**:* YCSB was configured to report the achieved mean and percentile latencies both for each run as a whole and for 10 second periods. Since YCSB reports throughput as simply the inverse of latency multiplied by the number of threads, we only present latency values.

For each combination of deployment, configuration, workload, and number of threads, the sample mean of each collected metric was calculated, and Student's *t*-distribution was used to compute the 95% confidence intervals for the corresponding population means. The half-width of those confidence intervals for such values presented later is under 10% of the respective sample mean.

Finally, Dstat v0.7.3 was used to observe CPU, memory, and disk utilization.

*e) **Experimental environment**:* Experiments were conducted using machines with the following specifications: one Intel Core i3-4170 CPU, clocked at 3.70 GHz, with 2 physical and 4 logical cores; 8 GiB of DDR3 RAM, clocked at 1600 MHz; and one 119 GiB, SATA-III Samsung MZ7LN128 solid-state drive. Software-wise, the machines used Ubuntu Server 18.04 LTS for AMD64 with Linux kernel v4.15.0.

## V. Evaluation

Here, we present the results of the evaluation conducted on the *HBase* (§V-A) and *PostgreSQL* (§V-B) deployments. The full results are also available at https://rgmacedo.github.io/drss19-website.

### A. HBase – Compaction

Figure 2 depicts the mean latency achieved by the *HBase* deployment under the configuration with compaction effects, normalized against the configuration without compaction effects (taken as the baseline in the discussion that follows), under all workloads defined previously with both 1 and 10 threads.

Write-oriented operations, namely updates and inserts, present a performance degradation of at most 14.7% for mean latency, while read-oriented operations, namely reads and scans, exhibit a performance degradation of at most 87.3%. At the 99th percentile latency, depending on the workload, write operations experience performance degradations of at most 40%, while for reads, overhead ranges from being imperceptible to being as high as 955.2%. Regarding read-modify-write operations, mean latency results experienced overheads of at least 27.6%, while at the 99th percentile a 281% overhead is noticed.

With respect to resource utilization, effects of compaction processes do not entail significant differences. CPU utilization remains mostly unaltered, while both read and write disk throughput experiences minimal variation — a maximum absolute difference of 33 and 15 MiB/s, respectively. Interestingly, during compaction effects, this observed increase of disk throughput remains constant throughout the overall execution time (for all observations). This is due to the default throttling policy that *HBase* employs over compactions, limiting their I/O performance so as not to introduce high interference and performance overhead in incoming I/O requests [18].

As observed, write operations have a much lower impact in mean latency compared to read operations. This is due to two main factors: (1) the write operation flow at *HBase* introduces less overhead than read-oriented ones, as writes are

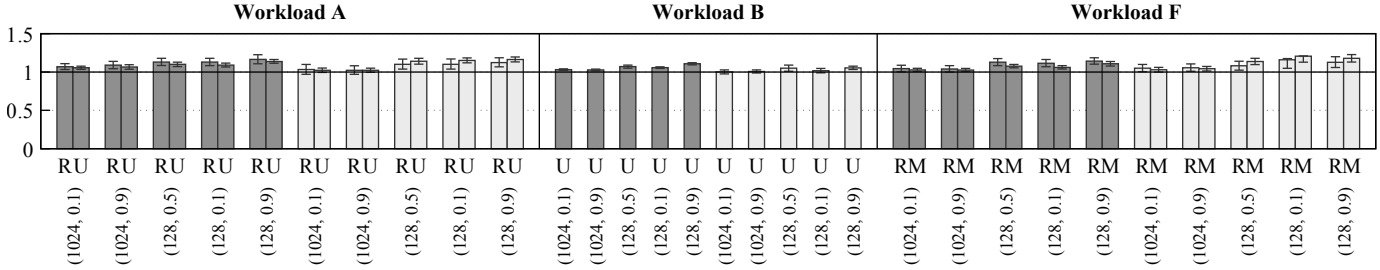


Fig. 4. *PostgreSQL* mean latency normalized against the $(1024, 0.5)$ configuration, under workloads with 1 thread ( ) and 10 threads ( ). Error bars represent the 95% confidence interval. (R – read, U – update, M – read-modify-write)

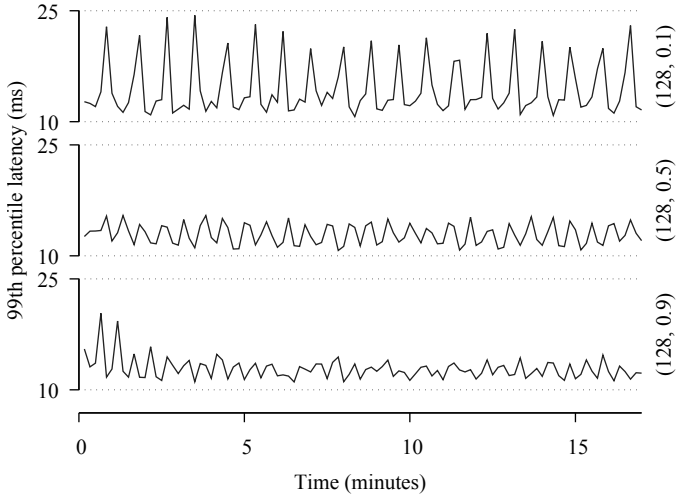


Fig. 5. 99th percentile latency variation for the update operation over the duration of a single run for each of the $(128, 0.1)$, $(128, 0.5)$, and $(128, 0.9)$ *PostgreSQL* configurations, under workload A with 1 thread.

first sequentially written to the WAL and then stored at the in-memory *Memstore*; and (2) the experienced compaction processes are comprehended on the *minor compaction* spectrum, thus not imposing major disk overload and I/O interference to write operations.

On the other hand, read operations experience significant mean latency overhead due the inherent dependency over *HFiles* currently being compacted. Specifically, compactions are mainly motivated to improve read performance, rewriting several small *HFiles* into fewer larger ones. As such, under compactions effects, read operations will be severely influenced by the concurrent merging operations being conducted. Moreover, this effect becomes further amplified at the 99th+ percentile latencies, as depicted in Figure 3, where a point $(x, y)$ means that $y$ is the fraction of requests that experience a latency of $x$ $\mu$s or higher. As observed, insert requests present similar latency performance for all latency percentiles. Contrarily, scan requests experience significant latency increase between 50th and 99.99th percentiles, achieving a 452 ms latency at the 99.99th percentile, a 509% increase compared to the baseline.

### B. PostgreSQL – Checkpointing

Figure 4 shows the mean latency achieved by the various *PostgreSQL* configurations defined previously, normalized against the default $(1024, 0.5)$ configuration, under workloads A, B, and F with both 1 and 10 threads.

Results show that completion target variations do not present significant performance differences when configured with a 1024 MiB WAL size ($(1024, y)$ configurations). Specifically, the overhead on mean latency ranges from being imperceptible to up to 8.9%. Analogously, at 99th percentile latency, performance is degraded by at most 8.7%. Contrarily, under 128 MiB WAL size, all workloads experienced a much more noticeable performance degradation in both mean and 99th percentile latencies. As opposed to the *HBase* deployments (§V-A), both read- and write-oriented operations were equally exposed to latency overheads. Specifically, experiments conducted for the $(128, 0.5)$, $(128, 0.1)$, and $(128, 0.9)$ configurations showed a performance degradation of at most 13.9%, 20.7%, and 17.8%, respectively. Likewise, this effect is further amplified at 99th percentile latencies, exposing an overhead of 33.2%, 61.9%, and 33.3% under the $(128, 0.5)$, $(128, 0.1)$, and $(128, 0.9)$ configurations, respectively. Regarding resource utilization, effects of WAL size and completion target variation do not entail significant differences.

As observed, *PostgreSQL* WAL size plays a major role in the overall performance, as flushing larger WAL files to the file system provides better end-to-end performance when compared to smaller sizes, thus ensuring sustained latency performance. On the other hand, despite the low degree of variance in mean latencies, *PostgreSQL* checkpoint completion target significantly impact latencies at the 99th+ percentiles, introducing severe I/O interference and performance variability. Figure 5 depicts a representative case, showing the performance variation under different checkpoint completion target configurations. While the achieved mean latency was 4.598 ms, 4.559 ms, and 4.755 ms for $(128, 0.5)$, $(128, 0.1)$, and $(128, 0.9)$ configurations, respectively, latency at the 99th percentile obtained 13.06 ms, 14.729 ms (+12.7%), and 12.769 ms (-2.2%) for the same configurations. When the completion target is set to a higher bound, namely 0.9, *PostgreSQL* throttles the write performance of WAL files, providing a more conservative and sustained performance for incoming I/O requests. When set to lower bounds, namely 0.1,

the system experiences increasing I/O burstiness, achieving 10.687 ms and 24.383 ms as minimum and maximum 99th percentile latencies, respectively.

## VI. Discussion and Conclusion

As shown by the results presented in the previous section (§V), both compaction and checkpointing background tasks heavily impact the performance of foreground activities, introducing significant I/O interference and performance degradation, at both mean and 99th percentile latencies.

The root causes of such a performance impact can be justified by two key points. By default, *HBase* throttles both read and write performance of compaction processes, and does not provide sufficient building blocks to dynamically tune such settings, leaving applications to experience I/O interference and performance degradation for longer periods. Moreover, these background tasks should be inherently programmable, in order to adjust compaction primitives to the current overall infrastructure load, and to comply with varying storage policies submitted by client applications.

Furthermore, contrarily to *HBase*, *PostgreSQL* already provides a wide set of primitives to adjust checkpointing background processes. However, as workloads vary along time, current settings are still not enough to ensure complex storage policies for applications and users, as they cannot be dynamically tuned to the current load of the overall infrastructure.

We thus restate our argument that in order to attain optimum holistic performance, storage background tasks should be dynamically programmable and their execution handled by a control module with global infrastructure visibility and holistic I/O control.

Following SDS principles, we identify two steps that must be taken in order to achieve this. First, at layer level, storage systems should be vested with programmability primitives in order to fine-tune background parameters to any desirable setting. For instance, such a design would allow *HBase* storage applications to decide when to perform compactions or at which rate. *PostgreSQL* applications would similarly benefit from such a design, being able to dynamically configure checkpointing processing as intended.

Second, at infrastructure level, a SDS controller with system-wide visibility (*i.e.*, with holistic I/O knowledge) would be able to fully utilize this programmability to efficiently combine background and foreground tasks, and thus be able to ensure higher levels of QoS provisioning, performance isolation, and resource fairness.

## References


[1] G. Amvrosiadis, A. R. Butt, V. Tarasov, E. Zadok, M. Zhao *et al.*, "Data Storage Research Vision 2025: Report on NSF Visioning Workshop Held May 30–June 1, 2018," Tech. Rep., 2018.

[2] E. Lee and H. Bahn, "Caching Strategies for High-Performance Storage Media," *ACM Transactions on Storage*, vol. 10, no. 3, pp. 11:1–11:22, Aug. 2014.

[3] H. Kim, D. Shin, Y. H. Jeong, and K. H. Kim, "SHRD: Improving Spatial Locality in Flash Storage Accesses by Sequentializing in Host and Randomizing in Device," in *15th USENIX Conference on File and Storage Technologies*. USENIX Association, 2017, pp. 271–284.

[4] S. Kim, H. Kim, J. Lee, and J. Jeong, "Enlightening the I/O Path: A Holistic Approach for Application Performance," in *15th USENIX Conference on File and Storage Technologies*. USENIX Association, 2017, pp. 345–358.

[5] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "IOFlow: A Software-Defined Storage Architecture," in *24th ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 182–196.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.

[7] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Elsevier, 1992.

[8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *IEEE 26th Symposium on Mass Storage Systems and Technologies*. IEEE, 2010, pp. 1–10.

[9] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted Resource Management in Multi-tenant Distributed Systems," in *12th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2015, pp. 589–603.

[10] H. Jo, S. hun Kim, S. Kim, J. Jeong, and J. Lee, "Request-aware Cooperative I/O Scheduling for Scale-out Database Applications," in *9th USENIX Workshop on Hot Topics in Storage and File Systems*. USENIX Association, 2017.

[11] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, "Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency," in *5th ACM Symposium on Cloud Computing*. ACM, 2014, pp. 9:1–9:14.

[12] Z. Cao, V. Tarasov, H. Raman, D. Hildebrand, and E. Zadok, "On the Performance Variation in Modern Storage Stacks," in *15th USENIX Conference on File and Storage Technologies*. USENIX Association, 2017, pp. 329–344.

[13] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of HDFS Under HBase: A Facebook Messages Case Study," in *12th USENIX Conference on File and Storage Technologies*. USENIX Association, 2014, pp. 199–212.

[14] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[15] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil, "The Log-Structured Merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *1st ACM Symposium on Cloud Computing*. ACM, 2010, pp. 143–154.

[17] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. Paulo, J. Pereira, and R. Vilaça, "MeT: Workload aware elasticity for NoSQL," in *8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 183–196.

[18] "Limit compaction speed," Website, 2017, retrieved July 1, 2019. [Online]. Available: https://issues.apache.org/jira/browse/HBASE-8329