Hector: A Framework to Design and Evaluate Scheduling Strategies in Persistent Key-Value Stores

Louis-Claude Canon
FEMTO-ST, Univ. Franche-Comté
Besançon, France
louis-claude.canon@femto-st.fr

Anthony Dugois
LIP, ENS Lyon, Inria
Lyon, France
anthony.dugois@ens-lyon.fr

Loris Marchal
LIP, ENS Lyon, CNRS, Inria
Lyon, France
loris.marchal@ens-lyon.fr

Etienne Rivière
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium
etienne.riviere@uclouvain.be

ABSTRACT

Key-value stores distribute data across several storage nodes to handle large amounts of parallel requests. Proper scheduling of these requests impacts the quality of service, as measured by achievable throughput and (tail) latencies. In addition to scheduling, performance heavily depends on the nature of the workload and the deployment environment. It is, unfortunately, difficult to evaluate different scheduling strategies consistently under the same operational conditions. Moreover, such strategies are often hard-coded in the system, limiting flexibility. We present Hector, a modular framework for implementing and evaluating scheduling policies in Apache Cassandra. Hector enables users to select among several options for key components of the scheduling workflow, from the request propagation via replica selection to the local ordering of incoming requests at a storage node. We demonstrate the capabilities of Hector by comparing strategies in various settings. For example, we find that leveraging cache locality effects may be of particular interest: we propose a new replica selection strategy, called Popularity-Aware, that can support 6 times the maximum throughput of the default algorithm under specific key access patterns. We also show that local scheduling policies have a significant effect when parallelism at each storage node is limited.

KEYWORDS

Scheduling, Key-Value Stores, Replica, Modularity, Performance

ACM Reference Format:
Louis-Claude Canon, Anthony Dugois, Loris Marchal, and Etienne Rivière. 2023. Hector: A Framework to Design and Evaluate Scheduling Strategies in Persistent Key-Value Stores. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Storage systems are particularly important in modern cloud applications. Among these systems, key-value stores found their way as central components in many cloud backends, as they offer a simple API, excellent horizontal scalability, and data availability through replication. The performance of key-value stores is of paramount importance for cloud-based applications. Achievable throughput and service latencies play a significant role in their scalability and response times. It is often observed, for instance, that even if most requests are successfully handled with low latency, some of them suffer from high delays globally impacting responsiveness: this is the famous *tail latency* problem [10]. The optimization of these metrics in persistent key-value stores has been the subject of extensive research, from the maximization of throughput through careful optimization of internal data structures [25] or CPU overhead minimization [21], to tail latency mitigation through redundant requests [26], popularity-aware data replication [8], or internal I/O scheduling [6]. Among these optimization techniques, the proper scheduling of client requests has received significant attention. Distributed key-value stores replicate data over multiple storage nodes in the cluster to ensure fault tolerance. This unlocks the possibility to select which replica should process a given request. In this direction, prior work proposed monitoring servers and redirecting each request to the replica that is supposed to perform the best at a given time [1, 13, 23].

We observe that the variety of scheduling algorithms previously proposed in the literature all bring some form of improvement, but often for a specific situation. However, the authors of these different proposals make nonidentical assumptions about the workload and the testing environment, which prevent any fair comparison between different solutions. For instance, on the one hand, C3 is a replica selection strategy that outperforms the default algorithm of Apache Cassandra [23]. On the other hand, the authors of the Héron strategy demonstrate that C3 may remain inefficient when the stored data items are heterogeneous in size [13]. Another example is when we consider different key access patterns. Some data items may be more requested than others [4] and we show that it has direct implications on performance. Furthermore, environment settings such as hardware and software configurations also have a significant impact.

Another difficulty comes from the complexity of key-value store implementations. This is especially true in widely-used systems

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

that have matured over the years and that became particularly sophisticated, as is the case of Apache Cassandra. This open-source project has a large codebase (about 500 000 lines of Java code) that is now hard to understand and extend. In particular, scheduling-related components are dispatched across the system and were not designed and implemented for easy extensibility. Testing new scheduling policies is thus challenging for newcomers.

Contributions. We introduce Hector, a framework extending Apache Cassandra that enables experimenters to implement and evaluate a large variety of scheduling algorithms. Hector provides highly modular components that interact seamlessly with each other. They are based on high-level principles, extracted from what we observed in existing proposals on scheduling in key-value stores: replica selection, local scheduling, state propagation, and workload oracles. The goal of Hector is to remove the friction experienced when implementing new strategies, i.e., to allow the user to focus on their implementation and delegate integration details to the framework. It also facilitates evaluation, in order to let the user compare experimentally and fairly different policies, under the same assumptions on the environment and on the workload. Hector should, ultimately, enable key-value store users to make better decisions on which scheduling strategy to adopt for a specific use case.

We begin by showing how the scheduling of requests works in a persistent key-value store such as Apache Cassandra (Section 2) and we also present the challenges one may encounter when introducing new strategies. Then, we present the components that we implemented in Hector and we provide implementation examples of scheduling policies (Section 3). We demonstrate in our evaluation that Hector does not exhibit any additional performance overhead compared to unmodified Apache Cassandra, and we show how Hector enables easy comparisons of different policies by illustrating some use-cases through two experiments (Section 4). In particular, we introduce a new replica selection strategy, called Popularity-Aware, that leverages cache locality and can achieve a throughput that is more than 6 times higher than the one of the default strategy in Apache Cassandra, under moderately biased key popularities. We also show that local scheduling may improve throughput and latency when parallelism is limited and when the dataset is heterogeneous. Finally, we review related work (Section 5) and conclude the paper (Section 6).

2 SCHEDULING IN PERSISTENT AND DISTRIBUTED KEY-VALUE STORES

A key-value store is a simple NoSQL database where each data item is bound to a unique key. The simple API (i.e., without secondary indexes) and lack of integrated support for complex queries (e.g., no joins) allow implementations of key-value stores that scale remarkably well horizontally, i.e., they are easily able to include more nodes if the workload requires it. In distributed key-value stores, keys are dispatched on servers using a partitioning scheme based on consistent hashing. As the same hash function is used across the cluster, each server can know where a data item associated with a given key is stored by simply hashing its key. In addition, data items are replicated according to a replication factor to preserve availability in the presence of faults. The typical replication factor in large-scale key-value stores is 3, which means that each data item is stored on 3 different servers.

In *persistent* key-value stores, in contrast with *in-memory* key-value stores, data is eventually saved on disk. This adds a level of complexity: as random I/O operations are slow, key-value stores typically employ special data structures, called Log-Structured Merge (LSM) trees, that temporarily perform write operations in memory and periodically flush data to disk [19]. This allows bulk write operations that reduce wear on disks, in particular SSDs, and significantly improves write throughput. Read operations, on the other hand, may either result in a cache hit (when the key is still represented in memory) or a cache miss, in which case costly disk reads are necessary. In this paper, we focus on one such persistent and distributed key-value store, namely Apache Cassandra, where each server plays two roles:

A. It receives client requests. For a given client request that is received by a server, we say that this server is the *coordinator* for this request.
B. It executes requests. When a coordinator receives a client request, it computes the set of servers able to execute the request. These servers are called *replicas*. The coordinator (i) decides which subset of these replicas will execute the request, (ii) forwards the request, (iii) awaits the response, and (iv) replies to the client.

The number of chosen replicas may vary according to the nature of the request and its *consistency level*, which corresponds to the number of replicas that must acknowledge this request before it is considered successful. A write operation is always processed on all replicas even if its corresponding consistency level is lower than the replication factor (however, we do not necessarily wait for the response of all replicas). This makes scheduling in general less imperative for write requests, as they must be executed by all replicas anyway, and each write operation is very fast thanks to the LSM tree structure that absorbs most I/O bottlenecks. A read operation, on the other hand, is executed on the exact number of replicas that corresponds to its consistency level and is considered unsuccessful if the responses are inconsistent. In this work, we set the consistency level to 1, which is arguably the most common setting for read-dominated workloads. Moreover, we focus on a single-datacenter cluster of static size: all servers are geographically located in the same place, they are linked by a high-performance local network (in the same rack), and there is no dynamic reconfiguration of scale-out/scale-in operations.

Within each server in Apache Cassandra, the parallel execution of request coordination and execution using multiple cores relies on a Staged Event-Driven Architecture (SEDA) [24]. Each type of operation is processed by a different pool of worker threads (also called a *stage*). For instance, the reception and handling of client requests (as part of the coordinator role) are processed by the Native-Transport-Requests thread pool, whereas read request reception and execution (replica role) are processed by the ReadStage thread pool. Additional stages are dedicated to the execution of write requests, internode messaging protocol, data migration, tracing, etc. Scheduling requests in Apache Cassandra is a two-step operation. First, the coordinator must choose which replica the request should be sent to (then, the request is sent to this replica that inserts it into

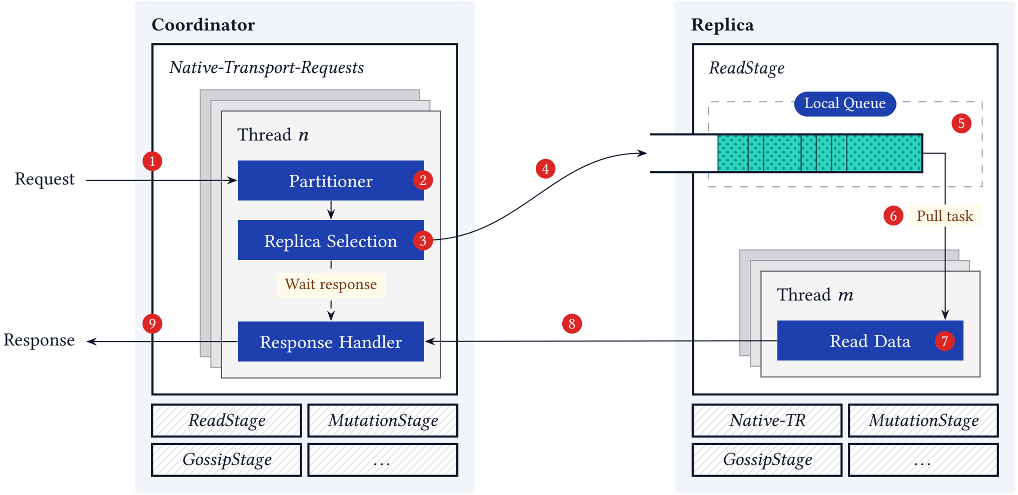

Figure 1: Scheduling of a read request in Apache Cassandra.

its local operation queue). This step is called *replica selection*. Second, the chosen replica must decide in which order its pending read operations should be processed. This step is called *local scheduling*. Note that, for a given request, the coordinator and the replica may sometimes be the same server. Figure 1 presents a more thorough breakdown of the steps involved in the scheduling of a read request:

(1) A client request reaches a server, which becomes the coordinator for this request.
(2) A worker thread from the Native-Transport-Requests thread pool picks the request and infers the set of replicas that are able to execute it.
(3) **The coordinator chooses a replica according to a replica selection strategy.**
(4) The coordinator forwards the request to the replica.
(5) The request reaches the replica, which pushes it into its local queue dedicated to read operations.
(6) **Worker threads from the ReadStage thread pool process the local queue in a specific order according to a local scheduling strategy.**
(7) The request is executed by a worker thread, which reads data (in memory, if present in the cache, or on disk).
(8) The replica sends data to the coordinator.
(9) The coordinator responds to the client.

Steps 1-4 and 9 happen on the coordinator, whereas steps 5-8 happen on the replica itself. We highlight in boldface the key scheduling operations: replica selection at step 3 and local scheduling at step 6.

Challenges. Various scheduling strategies have been proposed in the literature to improve the overall performance of Apache Cassandra [12–14, 22, 23]. By studying and comparing these papers, we observe that designing and implementing new solutions poses several challenges.

First, scheduling strategies often need information (e.g., the current sizes of the request queues or the current service rates at all replicas) on the cluster state in order to compute a score for each replica. The more accurate this information is, the more representative the score will be of the server health, and therefore the better the scheduling decisions will be. Unfortunately, key-value stores are subject to the usual constraints of distributed systems, namely that each server cannot know the exact state of other machines, as measurements must take place at a bounded pace and information takes time to propagate over the network. This means that algorithmic decisions (such as scheduling of read requests) can only be made with partial and out-of-date knowledge of the cluster condition. Efficiently leveraging such stale data is challenging. For example, the default replica selection algorithm of Apache Cassandra frequently causes *herd behaviors*, i.e., situations where all coordinators periodically select the same, supposedly most-suited server, leading to load oscillations [23].

Another difficulty is that the workload, i.e., the flow of requests reaching the key-value store system at runtime, is generally unknown beforehand. This is a problem, as various workload characteristics have direct implications on the behavior of a scheduling strategy. For example, one such characteristic is the distribution of data item sizes. Existing workloads may be homogeneous, where data items are all of a similar size, or heterogeneous, e.g., with sizes exhibiting a power law or a bimodal distribution. Another example is the distribution of key popularities: this is the statistical distribution of key access frequencies in client requests. Many more characteristics could be extracted from real traces, such as temporal patterns, the correlation between size and popularity, and reuse periods between keys, among others [4].

We also identify a reproducibility concern, coming from the lack of a common baseline on which strategies may be properly compared [5]. Existing proposals typically implement new algorithms in different versions of Apache Cassandra. When code artifacts are publicly available (which is not systematic), this requires transferring the implementation of a prior solution in the more recent codebase, which is a cumbersome task and implies potential incompatibilities with newer components. When code artifacts are not publicly available, this requires building state-of-the-art strategies from their general description, which is far from being ideal and prone to errors, as implementation details are often overlooked in scientific papers. Moreover, the software configuration is usually not indicated, and the hardware configuration is rarely similar. This makes directly comparing published performance figures particularly unreliable.

Finally, diving into the Apache Cassandra codebase may be intimidating (it contains almost 500 000 lines of Java code), and scheduling-related code is dispatched across many different packages. Testing new solutions is, as a result, a time-consuming task for newcomers, especially when they want to determine if a particular idea is efficient and worth paying the associated communication, storage, or complexity cost.

3 DESIGN OF HECTOR

We unify the scheduling-related components of Apache Cassandra (version 4.2) into a coherent framework called Hector[1]. The API of this framework is simple enough to let any user implement new scheduling strategies without knowing the details of the entire codebase. Moreover, we introduce features that are not present by default in Apache Cassandra and that help designing more powerful and sophisticated policies. Our approach enhances the workflow of comparing strategies under specific assumptions by providing a common baseline. As each component is configurable from a single control point, this also enables users to more quickly identify the best setting for their use-case. We present in this section the general components of Hector and we give some examples of implemented scheduling strategies.

3.1 Modular Components

The API of Hector is a set of general interfaces. The user implements these interfaces and specifies with which parameters they must be loaded using a configuration file. Hector takes the responsibility of instantiating the components in the correct order and connecting them together when starting the system. We describe the different components in what follows.

Replica selection. This is the step in which the coordinator chooses the set of replicas that are considered to be most suited for the current request. In Apache Cassandra, this module has access to the full list of replicas and the mapping between keys and replicas. When presented with a target key, the module must return a list of servers in decreasing order of priority. The n first servers from this list are contacted, where n is the consistency level. When reading from a single replica (i.e., the consistency level is 1), the module typically returns a list of replicas and only the first one is contacted. In a nutshell, the ordering step constitutes the essence of replica selection.

We find that the programming interfaces of Apache Cassandra lack two essential features to make better scheduling choices, as illustrated by Figure 2. First, sorting is made without knowing any characteristics of the current request. This makes fine-granularity decisions, e.g., workload-aware choices, nearly impossible. Second, it is not possible to include additional data in the routed request to guide subsequent steps such as local scheduling, for example by specifying a priority score calculated by the coordinator node. In Hector, defining a new replica selection strategy simply consists in extending an abstract class and implementing a sorting function.

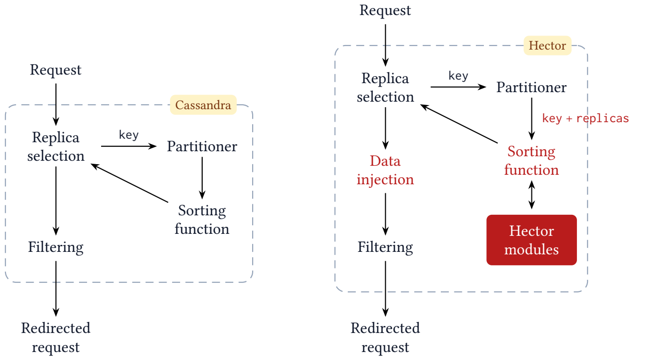

Figure 2: Replica selection on coordinators in Apache Cassandra (left) and Hector (right). Additional features of Hector are highlighted in red.

This function takes the unordered list of replicas and the current request as parameters and must return an ordered list of replicas. Of course, it may also leverage external information (being built by other modules of Hector), as well as internal information (being built in the replica selector instance itself). In addition to request identification, Hector provides interfaces to include custom data in the request to be transferred over the network and retrieved later on replicas.

Local scheduling. As explained in Section 2, the system is highly concurrent and divides its execution model into various stages. Stages are a general abstraction in the execution model, meaning that each stage simply handles a linked queue of self-contained runnable objects and thus does not know about the nature of the operations it is responsible for. Moreover, the queue instantiation is hard-coded, which makes it impossible to associate different local scheduling policies to different stages.

In Hector, we generalize the stage concept by setting the queue as a parameter in the class definition. We also augment the runnable operation objects to include various information about the current request, making any queue implementation aware of the current request characteristics. Moreover, any data added by the replica selection component (on the coordinator) may be retrieved to help taking local scheduling decisions. The local scheduler instance can also rely on external (e.g., data that it gets from replica selection) and internal information to make better ordering decisions.

State propagation. Getting the instantaneous state of remote servers is unfortunately not possible in distributed systems. However, even an out-of-date view can be of interest when talking scheduling decisions. This is why some existing proposals monitor server state characteristics (e.g., queue sizes, average service time, or number of I/Os). This data often forms the basis of replica scoring decisions [23]. The challenge comes from the channels by which we retrieve information: one must periodically transmit values of interest at a sufficiently high rate to take advantage of fairly recent information.

In Hector, each server holds its own copy of a *cluster state* data structure. This cluster state consists of a list of *endpoint state* entries. Each endpoint state corresponds to a specific server in the cluster (including the current host) and maps a value to a property of

[1] https://anonymous.4open.science/r/hector

interest, which we call a *fact*. Let us describe the building process for the cluster state, which is summarized in Figure 3. The definition of a fact i consists of 4 functions:

- The measurement M_i defines how to retrieve the value to send over the network.
- The serializer S_i (resp. deserializer D_i) encodes (resp. decodes) a value to a byte buffer.
- The aggregator A_i combines received values into a unique value to save in the endpoint state. This component is useful to define custom aggregation operators, e.g., (weighted) moving averages.

The state propagation module extends the internode messaging service of Apache Cassandra. This means that the user may include state values in any message, being a message purposely built for state propagation, or an already-existing message (piggybacking). Hector examines the previously-defined facts and executes the corresponding measurement functions to get raw state values on the host, which are then gathered as *state feedback*. When the state feedback is ready, a timestamp is added and data is transformed into a byte buffer through the fact serializer. Moreover, the byte buffer is prefixed by the fact identifier to know how to deserialize it in the future. Then, the state feedback bytes are added to the message before being sent over the network. When the packet is received, the message handler proceeds to decode the byte buffer and retrieves the state feedback. Each included value is added to the local endpoint state that corresponds to the message sender by applying the aggregation operation of the fact. Note that we must be careful when dealing with ordered values: the high concurrency of Apache Cassandra implies that some feedback f_a from a given endpoint may arrive before feedback f_b from the same endpoint, whereas f_a contains values measured *after* values of f_b. This is why we associate a timestamp with each feedback. As we only compare feedback coming from the same peer, timestamp values stay comparable, and we may choose to discard values that are older than the most recent processed feedback.

Workload oracle. Although they are generally unknown (or known with little precision), workload characteristics such as the distribution of data item sizes or key access frequencies have a direct influence on the efficiency of scheduling strategies. Being able to predict these characteristics is a clear advantage when designing policies.

In Hector, oracles are the components that give information about these characteristics to other modules. According to the use-cases, there are various ways an oracle can build this information. Of course, the simplest situation is when the characteristics are known beforehand: the oracle may for example load data in memory from static files describing the workload. However, in more common situations, the oracle will have to learn the workload at runtime. This can be done through machine learning techniques, statistical inference, probabilistic data structures such as Bloom filters [13], etc. In order to ease the evaluation process, each oracle instance is assigned a unique identifier. In this way, other components such as replica selection or local scheduling are not tied to a specific oracle definition, and the user may switch between different oracle implementations without modifying the calling component.

3.2 Implemented Schedulers

We demonstrate in this section the flexibility of Hector components by implementing state-of-the-art policies such as Dynamic Snitching [1] and C3 [23], and we show that it is also easy to test new ideas by introducing two novel replica selection/local scheduling strategies that we named *Popularity-Aware* and *Random Multi-Level*.

Dynamic Snitching. This is the default replica selection strategy in Apache Cassandra [1]. Dynamic Snitching is based on replica scoring. Each coordinator measures service time for each sent request and maintains a history of these measurements for each server in the cluster. The coordinator assigns a score to each replica by computing an Exponentially Weighted Moving Average (EWMA) of recorded latencies. When processing a request, it selects the replica with the current lowest score. A parallel process updates scores every 100 milliseconds to avoid being in the critical path of query service, and another process resets scores every 10 minutes to allow slow servers to recover. We reimplement Dynamic Snitching as a replica selection module in Hector.

C3 (scoring-only). The C3 replica selection algorithm was proposed to overcome some weaknesses of Dynamic Snitching [23]. This strategy aggregates information on the cluster state by including values of interest in the responses of each replica, such as the read operation queue size q and the average service rate μ. We easily reimplement this process using the state propagation module of Hector: we measure the queue size q and the number of completed operations w in the ReadStage thread pool and we transmit this information. In the aggregation step, we compute the average service rate as $\mu = \frac{w-w'}{t}$, where w' and t are respectively the previously-received value of w and the time elapsed since the last update, and we add q and μ in corresponding moving averages.

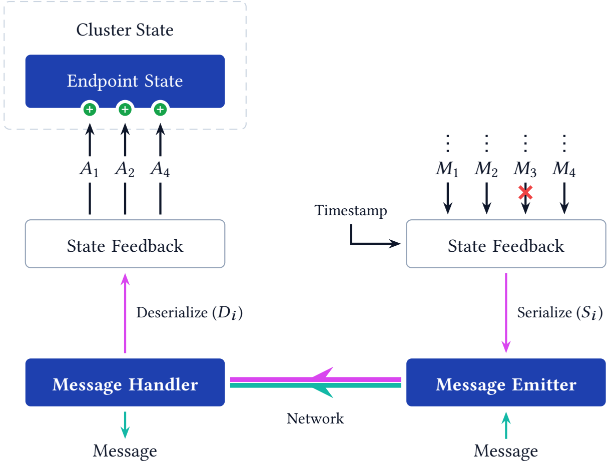

Figure 3: State propagation module of Hector. M_i, A_i, S_i, and D_i are respectively the measurement component, the aggregation component, the serializer, and the deserializer of fact i. The message emitter and message handler represent any internode communication, e.g., a periodic broadcast or a request response. The user may filter the transmitted values. In this example, the fact 3 is not included in the packet.

C3 also maintains the current count c of remote pending requests for each replica, and an history of observed latencies R, which are finally used to compute a score according to the cubic function $\bar{R} - \bar{\mu}^{-1} + \bar{\mu}^{-1}(1 + cm + \bar{q})^3$, where $\bar{R}$, $\bar{\mu}^{-1}$ and $\bar{q}$ are respectively the EWMAs of observed latencies, service times, and queue sizes, while m is the number of servers. Here, using a cubic term aims to penalize servers with longer queues, which is expected to lead to better balancing. In the original proposal, the replica scoring is coupled to a rate limiting process, which monitors the health of each replica and limits the sending rate towards a replica when it is suspected to be overloaded. For the sake of simplicity, we do not implement this part in this example, although it would be easy to integrate in Hector without any conflict with existing components.

Popularity-Aware. Workloads often exhibit biased popularity distribution on partition keys. We design a new replica selection strategy that is able to learn and leverage this distribution. The popularity of a key (at a given time) is defined as the ratio between the number of accesses for this key and the total number of requests. We implement a workload oracle that maintains a histogram of key accesses. When a request is received by the coordinator, it asks the oracle to record a new access, which is done by first hashing the key to a positive long integer and incrementing the corresponding access count in a map. By hashing the key, we limit the memory footprint of the data structure (at a small cost on the precision of the distribution). For example, recording accesses to a dataset that comprises 1 billion keys requires at most 16 GB of memory. We also ensure that the map implementation guarantees atomic, lock-free increments to enable efficient concurrent mutations.

We schedule requests according to the popularity value. The idea is to maximize the benefit we obtain from caching at the different servers, while balancing the load of serving popular content over multiple servers. When a key is considered popular, the probability to find the corresponding data item in the cache of any of the replicas holding it is high. Therefore, all replicas are expected to be able to respond without performing costly disk-read operations, and the best choice is to spread the load over these replicas. On the other hand, requests for unpopular keys will take advantage of the cache memory more if they are always executed on the same server, in order to avoid the eviction of the corresponding data items. In summary, if the popularity of a key is above a user-defined threshold, we schedule the request according to a round-robin strategy; otherwise, we always schedule the request on the same (first) replica. For example, with 1 million keys and a popularity distribution following a Zipf law with parameter 1.5, setting a threshold of 10^{-5} leads to 0.1% of keys marked as popular when the learning process has converged.

First-Come First-Served. This is the default local scheduling strategy in Apache Cassandra. Read operations are simply stored in a wait-free concurrent linked queue, and there is no priority mechanism. This is strictly equivalent to the standard First-Come First-Served algorithm.

Random Multi-Level. Priority queues are a common solution to execute operations in a statically-defined order. However, existing standard implementations need thread synchronization when used in a highly concurrent environment, which may degrade overall throughput. Inspired by the work on Rein [22], we emulate a priority queue, while avoiding any related thread contention, with the following randomized process. We define a Random Multi-Level (RML) queue as an ordered list of n wait-free concurrent linked queues. Each sub-queue q is associated a weight $w_q = \alpha^{n-q+1}$, i.e., the first queue has weight α^n, the second queue has weight α^{n-1}, and so on, where α is a user-defined positive coefficient. A given operation entering a sub-queue q has a priority value equal to $n - q + 1$. In other words, operations entering the first sub-queue will have the highest priority, whereas operations entering the n-th sub-queue will have the lowest priority. The RML queue is processed by generating a random integer x between 0 and the sum of weights $\sum_{q=1}^{n} w_q$, and dequeuing the first sub-queue q' such that $x \leq \sum_{q=1}^{q'} w_q$ (if q' is empty, we generate another random number and repeat the process). For example, with $n = 2$ and $\alpha = 2$, the first sub-queue would have priority 4 and the second sub-queue would have priority 2. In other words, the first sub-queue would be treated first with probability 2/3. For reasonable values of n and α, the probability to treat each sub-queue is high enough to ensure that no starvation occurs.

We implement a priority-based local scheduling policy by retrieving a priority value from each request and pushing read operations in an RML queue. This priority value is defined during the replica selection step on the coordinator server by leveraging the data injection and transmission mechanism of Hector. In this manner, we improve the flexibility of the scheduling process, as requests are locally executed according to a custom order that is directly decided by the coordinator. For example, we may consider that each priority value depends on the size of the requested data item, i.e., requests for data items whose size is below a given threshold must be processed with higher priority. However, this implies that we must be able to estimate the size of requested data items. In this paper, for illustration purposes, we use a simple workload oracle that is able to extract the size of a data item from the corresponding partition key, but a real use-case would require a more sophisticated approach (e.g., Bloom filters [13]).

4 EXPERIMENTAL EVALUATION

We evaluate Hector through several experiments on a real cluster. We compare the cost incurred by the generalization of scheduling components and newly-introduced features in the framework to the unmodified system, in particular in terms of overhead on throughput and latencies. We also illustrate the possibilities of Hector by showing how it enables easy comparisons of different approaches in scheduling requests with various assumptions on the workload and the environment.

4.1 Platform and Workload Setup

We run all experiments on the large-scale experiment platform Grid'5000 [7]. We use 15 identical servers located in the same geographic cluster, each equipped with a 18-core Intel Xeon Gold 5220 (2.20 GHz) CPU and 96 GiB of RAM. Data is stored on each machine on a 480 GiB SATA SSD device. Servers are interconnected by 25 Gbps Ethernet, and they all run Debian 11 GNU/Linux. The system runs on Java 11 with the CMS garbage collector. The replication

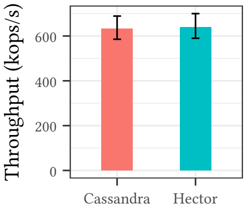

Figure 4: Maximum attainable throughput ($\times 10^3$ operations per second) for Apache Cassandra and Hector. Keys are accessed according to a Zipf law with parameter 0.9 and data item sizes range from 1 to 100 kB. The replica selection strategy is Dynamic Snitching and the local scheduling policy is First-Come First-Served.

Table 1: Absolute and relative (average) differences on the observed maximum attainable throughput of vanilla Apache Cassandra and Hector.

	Cassandra	Hector	Abs. diff.	Rel. diff.
Throughput	632 911 ops/s	640 205 ops/s	7294 ops/s	1.15%

factor is set to 3: each data item is written on 3 replicas, but each read request is processed by only one of these 3 replicas.

As we do not have access to real datasets and traces, we evaluate the system with synthetic workloads. Yahoo's Cloud Serving Benchmark (YCSB) is a commonly used tool to generate such workloads [9]. However, we find that YCSB lacks several features for evaluating modern database systems (e.g., advanced workload customization, modular architecture, and reproducibility), and falls behind more recently developed tools. NoSQLBench is one such tool, which permits to tune advanced characteristics of the workload and which takes care of many pitfalls related to benchmarking practice [2]. We run NoSQLBench on additional nodes, located in the same rack as the Hector cluster. We make sure that we bring enough concurrency, and we systematically check that we do not overload the CPU on client nodes to ensure that they are not the bottlenecks in the experiments. In each run, we select a number of keys to store at least 150 GiB of data at each server, a dataset that does not fully fit in memory.

4.2 Results

The first set of results is dedicated to the evaluation of Hector itself, as we want to make sure that it behaves identically to Apache Cassandra in the nominal use-case. Then, we illustrate how Hector enables comparing replica selection and local scheduling strategies under various assumptions.

No overhead. We check that the additional features of Hector do not introduce performance overhead compared to the unmodified Apache Cassandra (noted as "vanilla" in what follows). We make sure that both systems are identically configured, and we run them in the same conditions. The replica selection strategy is Dynamic Snitching, and the local scheduling strategy is First-Come First-Served. Key access distribution follow a Zipf law with parameter 0.9, corresponding to a biased popularity. For instance, with 100 keys, the first key is requested with probability 15.5%, the second key with probability 8.3%, and so on, until the 100th key with probability 0.2%. We also bring some heterogeneity: 1/2 of the values in the dataset have a size of 1 kB, 1/3 have a size of 10 kB, and the last 1/6 have a size of 100 kB. This setting simulates a common type of read-only workload to benchmark Apache Cassandra. Moreover, no additional data is transmitted through the state propagation module in Hector, and there is no workload oracle.

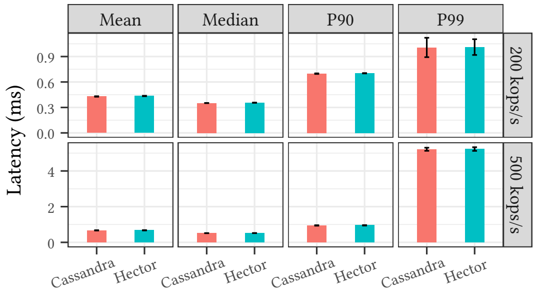

Figure 5: Latency (milliseconds) for Apache Cassandra and Hector, with two different arrival rates (200 000 and 500 000 operations per second).

Figure 4 shows the maximum attainable throughput in each system while Figure 5 presents the mean, the median, 90th and 99th percentiles of the latency when read requests arrive according to a given rate (200 000 and 500 000 operations per second, which in this case correspond respectively to 30% and 80% of the maximum capacity). Each experiment runs for 10 minutes and is repeated 10 times (each value represents the mean among the runs, and error bars indicate the standard error on the mean). Tables 1 and 2 summarize the absolute and relative (average) differences between vanilla Apache Cassandra and Hector.

We see that both systems seem to behave identically. Hector attains a slightly better throughput (in average 1.15% higher) than Apache Cassandra; the errors indicate that this difference is clearly not significant. Moreover, even if Hector shows higher latencies for the considered statistics, the absolute differences stay in the microsecond scale, with a maximum relative difference of 1.38%. Again, according to the standard error, this difference is not significant. In other words, no performance overhead is observed in Hector for the standard use-case, and we consider the framework to be a stable baseline that we can trust to evaluate and compare various scheduling strategies.

Cache-locality effects. We illustrate how different key access patterns may influence scheduling behavior. In particular, we show that a strategy that correctly leverages the Linux page cache clearly outperforms other algorithms under some assumptions on the distribution of key popularities. We compare 3 replica selection strategies: Dynamic Snitching (DS), which is the default algorithm implemented in Apache Cassandra; C3, which is a proposal coming from the literature [23]; and Popularity-Aware (PA), which is a new algorithm that we propose in this paper (see Section 3.2. We run two experiments for 100 minutes, with different assumptions on

Table 2: Absolute and relative (average) differences on the observed latency of vanilla Apache Cassandra and Hector, with two different arrival rates.

Stat.	Arrival rate	Cassandra	Hector	Abs. diff.	Rel. diff.
Mean	200 kops/s	0.429 ms	0.435 ms	0.006 ms	1.38%
	500 kops/s	0.670 ms	0.675 ms	0.005 ms	0.72%
Median	200 kops/s	0.352 ms	0.355 ms	0.003 ms	1.02%
	500 kops/s	0.515 ms	0.518 ms	0.003 ms	0.56%
P90	200 kops/s	0.698 ms	0.703 ms	0.005 ms	0.67%
	500 kops/s	0.946 ms	0.953 ms	0.007 ms	0.72%
P99	200 kops/s	1.007 ms	1.011 ms	0.004 ms	0.43%
	500 kops/s	5.214 ms	5.227 ms	0.013 ms	0.25%

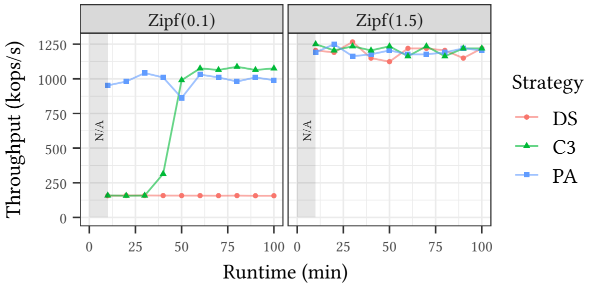

Figure 6: Maximum attainable throughput (×10^3 operations per second) over time for each replica selection strategy under two key popularity distributions. Higher is better. Zipf(0.1) **corresponds to a quasi-uniform distribution, whereas** Zipf(1.5) **corresponds to a heavily-skewed, right-tail distribution. Each data item has a fixed size of 1 kB.**

Table 3: Throughput (operations per second) of DS, C3 and PA under two key popularity distributions.

Stat.	Key popularity	DS	C3	PA
Mean	Zipf(0.1)	157 282	358 680	984 252
	Zipf(1.5)	1 193 317	1 212 121	1 194 743
Median	Zipf(0.1)	157 356	1 025 641	1 000 000
	Zipf(1.5)	1 204 819	1 219 512	1 190 476
Min	Zipf(0.1)	156 494	156 985	862 069
	Zipf(1.5)	1 123 595	1 162 790	1 162 790
Max	Zipf(0.1)	158 227	1 086 956	1 041 666
	Zipf(1.5)	1 265 822	1 250 000	1 250 000

the key popularity distribution: first, a Zipf law with parameter 0.1, which is a quasi-uniform distribution, and second, a Zipf law with parameter 1.5, which is heavily-skewed and right-tailed. All data items are 1 kB in size, and FCFS is used as the local scheduling policy.

Figure 6 measures the attainable throughput over time for each case, and Table 3 summarizes the values. This results in a significant difference between strategies when the popularity skew is small (i.e., Zipf(0.1)): for instance, PA shows a maximum throughput that is 6.58 times higher than the maximum throughput that is obtained with DS. Interestingly, we see that C3 seems to learn how to handle the workload over time, and is able to attain the same performance as PA after 50 minutes. When the skew is heavier (i.e., Zipf(1.5)), all strategies perform the same.

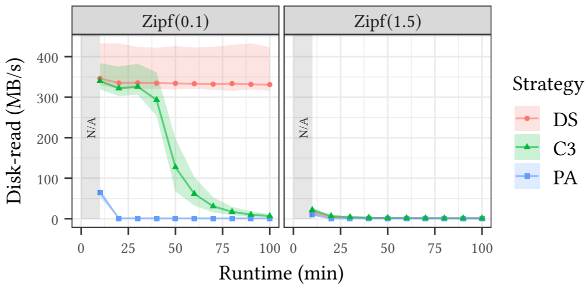

Figure 7: Amount of data read on-disk (megabytes per second) over time for each replica selection strategy under two key popularity distributions. Lower is better.

A possible explanation of these results is the cache management of the operating system that occurs on replicas. When a key is accessed, the key-value store starts by looking in the memtable, i.e., the internal data structure of the LSM tree that stores data temporarily in memory. If the key is not found, it must look for the data in the SSTable[2] files that are stored on-disk. The cache management of these files is entirely delegated to the operating system. Thus, if a key is frequently accessed, the Linux page cache keeps the corresponding SSTable file in memory, and the read operation is fast; otherwise, the file must be loaded before reading data, which is a slow operation. As Popularity-Aware tries to maximize the cache hit ratio, it rarely loads data directly from the disk, and thus performs better than other strategies. When the popularity skew is heavy, the cache hit ratio becomes naturally higher, and all strategies are able to achieve similar throughput.

To confirm this explanation, we plot the volume of data that is read on-disk over time on each replica in Figure 7. Light areas show the range between the minimum and maximum volume of data that are read on each replica, and the points represent the averages. First, we observe a clear visual correlation between both figures: the higher the throughput, the lower the volume of data that is read. Moreover, we see that for a quasi-uniform popularity distribution, PA rarely reads data directly on-disk (less than 1 MB/s), which indicates that it makes a very efficient use of the Linux page cache indeed. On the contrary, DS reads more than 300 MB of data per second in average, with a peak at more than 400 MB per second for one replica in the cluster.

Heterogeneous scheduling. Finally, we show how local scheduling may help improving performance in case of limited parallelism. We emulate a scenario where a slow storage device limits the number of concurrent read operations to avoid putting too much

[2] Sorted Strings Table (SSTable) is a format that maps data to keys, and these keys are kept sorted. This is the standard solution to store data in persistent key-value stores.

pressure on the disk. For illustration purposes, we set the maximum number of concurrent read operations to 4, and we compare two local scheduling policies from Section 3.2, First-Come First-Served (FCFS) and Random Multi-Level (RML), under the hypothesis that the dataset is heterogeneous, that is to say, it is composed of small data items (1 kB) and large data items (1 MB). We assume that 3/4 of the dataset is small, as it is often observed that heterogeneous datasets are mostly composed of small items in realistic workloads. For the RML strategy, we set the number of sub-queues to 2, and the priority coefficient α to 10. The first sub-queue is dedicated to read operations for small items (with priority 10^2) and the second sub-queue for large items (with priority 10). In this way, we expect that requests for large items will not block requests for small items, resulting in better overall performance. The key popularity distribution is uniform, and DS is the replica selection strategy. For each experiment, we gradually increase the arrival rate of read operations from 10 kops/s to 60 kops/s (10-minute increments), and we observe how the system handles the pressure.

Figure 8 shows the attainable throughput as a function of the arrival rate for each strategy. Horizontal lines represent the saturation throughput for each strategy. Light bars indicate that the system is saturating, and the request generator cannot reach the wanted arrival rate without causing request timeouts. Figure 9 presents the mean, the median, 90th and 99th percentiles of the latency distribution, also according to the arrival rates. The lines stop when the system reaches the saturation point.

We see from these results that RML outperforms FCFS in the considered scenario, as it is able to support a higher arrival rate (more than 50 kops/s, compared to 40 kops/s for FCFS). Moreover, we see that RML is able to achieve excellent median latencies, even when the arrival rate saturates the system (less than 5 ms with an arrival rate of 50 kops/s). However, the last percentiles are higher (almost 60 ms for the 90th percentile and 500 ms for the 99th percentile), which is due to the lower priority of requests for large items. Figure 10 compares the average latency of requests for small and large items. When saturating, the latency of requests for small items increases to about 65% of the latency of requests for large items when FCFS is the local scheduling policy. This is not the case for RML, which, when reaching saturation, is able to keep the latency of requests for small items to about 30% of the latency of requests for large items. As small items are in majority in the dataset, and RML treats them in priority, they are not awaiting in the queue and the overall performance is thus better, at the expense of slower requests for large items.

5 RELATED WORK

The problem of optimizing performance in key-value stores through appropriate scheduling has received significant attention. Dynamic Snitching [1], C3 [23], and Héron [13] are examples of scheduling strategies mixing global and local decisions to improve performance, particularly load balancing and tail latency [10]. Other systems leverage scheduling strategies to improve other metrics such as power consumption. Peafowl [3] unbalances the load of processing local requests on a storage node in a key-value store, to allow some of the cores to enter a low-power state, thereby reducing the energy footprint of the node in periods of lower activity.

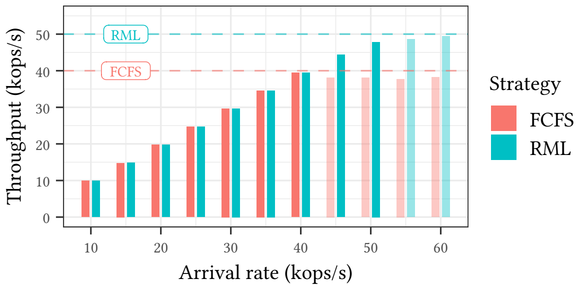

Figure 8: Attainable throughput (×10^3 operations per second) as a function of the arrival rate for each local scheduling strategy. Higher is better. Keys are accessed according to a uniform distribution. Data item sizes are small (1 kB, 75% of data items) or large (1 MB, 25% of data items). Servers have limited parallelism for read operations (up to 4 threads). Light bars indicate that the system saturates.

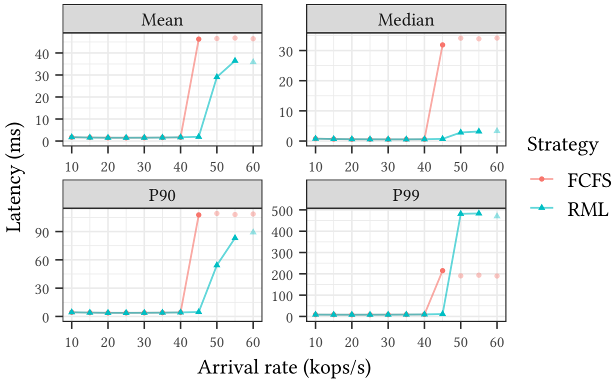

Figure 9: Latency (milliseconds) as a function of arrival rate for each local scheduling strategy. Lower is better.

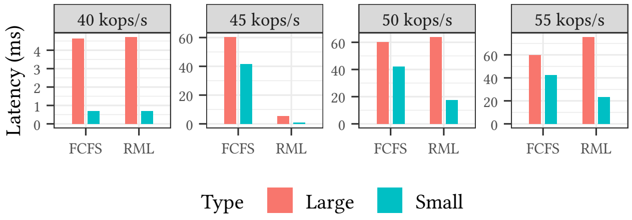

Figure 10: Average latency of requests for large (left bars) and small (right bars) items, for both strategies.

Multiget APIs allow clients to request multiple values in a single query. REIN [22], TailX [12], and AMS [14] combine local scheduling decisions at the level of the coordinator, to split the query into sub-queries for different storage nodes, and local scheduling to prioritize sub-queries that may be on the critical path for the overall query completion. For instance, the coordinator in REIN or TailX tags each sub-query by the amount of slack the local scheduler can use for its execution, allowing to prioritize sub-queries that strongly impact tail latency. Adaptive Freshness/Tardiness (AFIT) [27] is a query scheduler for fully-replicated key-value stores executing multiget queries over consistent snapshots, and that implements a configurable tradeoff between query evaluation latency and the freshness of these snapshots. Finally, Mega-KV proposes to leverage GPUs for serving local reads at the storage nodes [28]; this approach is, however, limited to situations where the entire set of values fits in memory.

OptimusCloud [17] considers the orthogonal but complementary problem of selecting appropriate resources for the deployment of a Cassandra cluster, based on a machine-learning-based characterization of the performance profile of different hardware configurations in the cloud.

Modular and compositional schedulers have been proposed in other contexts than key-value stores, such as in traditional operating systems. An example is the Completely Fair Scheduler [20] of the Linux kernel. According to use-cases and workloads, one may switch the scheduling policy of the kernel, or even extend the scheduler with new algorithms. This has been taken one step further by Bossa, an expressive domain-specific language simplifying the development of new schedulers in the Linux kernel [18], or iPanema [16] following a similar approach for formally-verified, modular schedulers.

Golatowski *et al.* [11] proposed a modular scheduling framework for RTOS, a real-time operating system. Although in a different context, their objectives bear similarities with ours, i.e., allowing to select an appropriate combination of scheduling components for a given workload and compare such combinations. Similarly, Click [15] is a modular router allowing to implement and test various packet scheduling policies in networks.

6 CONCLUSION

In this paper, we propose Hector, a framework built on top of Apache Cassandra to ease the implementation and evaluation of scheduling policies. Hector also constitutes a stable baseline to properly compare the performance of different strategies with identical assumptions. For instance, we were able to rebuild previously-proposed algorithms in Hector, and compare their performance to two novel algorithms, called Popularity-Aware and Random Multi-Level, under specific hypotheses on the workload and the environment. When key popularity is moderately biased, we found that PA is able to achieve a throughput that is more than 6 times higher than the maximum throughput of Dynamic Snitching, and converges faster than C3. In the case of limited parallelism and heterogeneous dataset, RML handles the load better than FCFS and keeps a low median latency (under 5 ms), even when the system starts to saturate. Moreover, we show that the various components introduced in Hector do not bring additional overhead on the overall performance of the system. Future work include devising a large-scale and exhaustive comparison of existing scheduling strategies under different workloads and platform configurations. We also plan to improve Hector to support customization of request partitioning in multi-get operations.

ACKNOWLEDGMENT

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

REFERENCES

[1] 2012. Dynamic snitching in Cassandra: past, present, and future. https://www.datastax.com/blog/dynamic-snitching-cassandra-past-present-and-future. Accessed on 2022-11-10.

[2] Accessed on 2022-12-11. NoSQLBench Docs. https://docs.nosqlbench.io.

[3] Esmail Asyabi, Azer Bestavros, Erfan Sharafzadeh, and Timothy Zhu. 2020. Peafowl: In-application cpu scheduling to reduce power consumption of in-memory key-value stores. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 150–164.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. 53–64.

[5] Vaibhav Bajpai, Mirja Kühlewind, Jörg Ott, Jürgen Schönwälder, Anna Sperotto, and Brian Trammell. 2017. Challenges with reproducibility. In *Proceedings of the Reproducibility Workshop*. 1–4.

[6] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 753–766.

[7] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, et al. 2012. Adding virtualization capabilities to the Grid'5000 testbed. In *International Conference on Cloud Computing and Services Science*. Springer, 3–20.

[8] Denis M Cavalcante, Victor AE de Farias, Flávio RC Sousa, Manoel Rui P Paula, Javam C Machado, and José Neuman de Souza. 2018. PopRing: A Popularity-aware Replica Placement for Distributed Key-Value Store. *CLOSER* 2018 (2018), 440–447.

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[10] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[11] Frank Golatowski, Jens Hildebrandt, Jan Blumenthal, and Dirk Timmermann. 2002. Framework for validation, test and analysis of real-time scheduling algorithms and scheduler implementations. In *13th IEEE International Workshop on Rapid System Prototyping*. IEEE, 146–152.

[12] Vikas Jaiman, Sonia Ben Mokhtar, and Etienne Rivière. 2020. TailX: Scheduling heterogeneous multiget queries to improve tail latencies in key-value stores. In *IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*. Springer, 73–92.

[13] Vikas Jaiman, Sonia Ben Mokhtar, Vivien Quéma, Lydia Y Chen, and Etienne Rivìere. 2018. Héron: taming tail latencies in key-value stores under heterogeneous workloads. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 191–200.

[14] Wanchun Jiang, Yujia Qiu, Fa Ji, Yongjia Zhang, Xiangqian Zhou, and Jianxin Wang. 2022. AMS: Adaptive Multiget Scheduling Algorithm for Distributed Key-Value Stores. *IEEE Transactions on Cloud Computing* (2022).

[15] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.

[16] Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, and Gilles Muller. 2020. Provable multicore schedulers with Ipanema: application to work conservation. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[17] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2020. OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 189–203.

[18] Gilles Muller, Julia L Lawall, and Hervé Duchesne. 2005. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In *Ninth IEEE International Symposium on High-Assurance Systems Engineering (HASE'05)*. IEEE, 56–65.

[19] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[20] Chandandeep Singh Pabla. 2009. Completely fair scheduler. *Linux Journal* 2009, 184 (2009), 4.

[21] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2016. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 537–550.

[22] Waleed Reda, Marco Canini, Lalith Suresh, Dejan Kostić, and Sean Braithwaite. 2017. Rein: Taming tail latency in key-value stores via multiget scheduling. In *Proceedings of the Twelfth European Conference on Computer Systems*. 95–110.

[23] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 513–527.

[24] Matt Welsh, David E. Culler, and Eric A. Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP*. 230–243.

[25] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 71–82.

[26] Zhe Wu, Curtis Yu, and Harsha V Madhyastha. 2015. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 543–557.

[27] Chen Xu, Mohamed A Sharaf, Minqi Zhou, Aoying Zhou, and Xiaofang Zhou. 2013. Adaptive query scheduling in key-value data stores. In *International Conference on Database Systems for Advanced Applications*. Springer, 86–100.

[28] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1226–1237.