Compas'2023 : Parallélisme/ Architecture/ Système
LISTIC/USMB - Annecy France, 4-7 juillet 2023

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

Louis-Claude Canon[1], Anthony Dugois[2], Loris Marchal[2] and Etienne Rivière[3]

[1] FEMTO-ST, Univ. Franche-Comté, Besançon, France
[2] LIP, ENS Lyon, Inria, Lyon, France
[3] ICTEAM, UCLouvain, Louvain-la-Neuve, Belgium

Abstract
Key-value stores distribute data across several storage nodes to handle large amounts of parallel requests. Proper scheduling of these requests impacts quality of service, as measured by achievable throughput and (tail) latencies. While performance heavily depends on workload characteristics, it remains difficult to evaluate different scheduling strategies consistently under the same operational conditions. Moreover, such strategies are often hard-coded in a specific system, limiting flexibility. We present Hector, a modular framework for implementing and evaluating scheduling policies in Apache Cassandra. Hector allows users to select among several options for key components of the scheduling workflow, from request propagation via replica selection to local ordering of incoming requests at a storage node. We demonstrate the capabilities of Hector by comparing replica selection strategies under various key access patterns, and we find that leveraging cache-locality effects may be of particular interest under some hypotheses. The strategy C3, as well as our new strategy Popularity-Aware, are able to support 6 times the maximum throughput of the default algorithm in specific conditions.

Keywords: Scheduling, Key-Value Stores, Replica Selection, Modularity, Performance.

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. Achievable throughput and service latencies of these systems play a significant role on the scalability and response time of cloud-based applications. 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 [9]. The optimization of these metrics in persistent key-value stores has been the subject of extensive research [25, 21, 26, 8, 6]. Among 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; hence, prior work proposed to monitor servers and redirect each request to the replica that is supposed to perform the best [1, 23, 12]. Each proposed scheduling algorithm in the literature brings some form of improvement. However, each work presents specific assumptions on the testing environment and workload, which prevent any fair comparison. For instance, on the one hand, C3 is a strategy that clearly 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 stored data items are heterogeneous in size [12]. Another example is when we consider different key access patterns; some data items may be more requested than others [5], and we show that it has direct implications on performance.

Moreover, extending key-value stores is difficult. This is especially true in widely-used systems that have matured over the years and that became particularly complex, such as 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 for easy extensibility. Implementing new scheduling policies is thus challenging for newcomers.

Contributions. We introduce Hector, a framework extending Apache Cassandra that enables experimenters to implement a large variety of scheduling algorithms. Hector provides 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 when implementing new strategies, i.e., allow the user to focus on implementation and delegate integration details to the framework. It also enables the user to perform comparisons of different policies under the same assumptions on environment and workload, which should ultimately lead to a better evaluation process.

2. Scheduling in Persistent, 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 complex queries (e.g., no joins) allow key-value stores to scale remarkably well horizontally. 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 knows where a data item associated to a given key is located, simply by hashing the key. In addition, data items are replicated to provide 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. In this paper, we focus on one such key-value store, namely Apache Cassandra, where each server plays two roles:

1. It receives client requests. For a given request that is received by a server, we say that this server is the *coordinator* for this request.
2. It executes requests. When a coordinator receives a request, it computes the set of servers able to execute it. 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. 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 only consider the setting where the consistency level is equal to 1. Moreover, we focus on single-datacenter clusters, i.e., all servers are geographically located in the same place

and linked by a high-performance local network.
To summarize, the scheduling of a read request takes place at two distinct levels. First, the coordinator must choose which replica the request should be sent to. This step is called the *replica selection*. Second, the chosen replica must decide in which order its pending read operations should be processed. This step is called the *local scheduling*. We provide a more detailed explanation of the request execution model in Appendix B.
Various scheduling strategies have been proposed in the literature to improve the overall performance of Apache Cassandra. By studying and comparing these papers, we observe that designing new solutions poses several challenges:

- Getting precise enough information on the cluster state at a given time is difficult, but enables taking better decisions.
- Authors need to compare their solution to state-of-the-art proposals, whose code artifacts are often unavailable and/or implemented in different versions of Apache Cassandra.
- Performance highly depends on the environment and on the workload characteristics.
- Diving into Apache Cassandra may be intimidating (almost 500 000 lines of Java code), and scheduling-related code is dispatched across many different packages.
- It is hard for an experimenter to determine if a particular idea is useful for scheduling 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. The API of this framework is simple enough to let any user implement new scheduling strategies without knowing the entire codebase in details. Moreover, we introduce features that are not present by default in Apache Cassandra and that help designing more powerful policies. Our approach enhances the workflow of comparing strategies under specific assumptions by providing a common baseline. By making each component configurable from a single control point (i.e., main configuration file—see an example in Appendix C), this also enables users to more quickly identify the best setting for their use-case. We present in this section the general components involved in Hector.

Replica selection. When a request reaches the coordinator, the corresponding set of replicas is inferred from the key, according to the replication scheme. Then, replicas are ordered from most-suited to least-suited following a specific ordering policy. The request is finally executed by the n first replicas in the ordered list, where n is the consistency level (in our case, $n = 1$). In a nutshell, the ordering step constitutes the essence of replica selection. We find that Apache Cassandra lacks two essential features to make wiser scheduling choices. First, sorting is made without knowing the key of the current request. This makes fine-granularity decisions (e.g., workload-aware choices) nearly impossible. Second, we cannot include additional data in the routed request to guide local scheduling in the next step. In Hector API, defining a new replica selection strategy simply consists in extending an abstract class and implementing a replica sorting function. The requested key is identified and may be used as a parameter of the function. In addition, Hector allows including custom data in the request to be transferred over the network and to be used during the local scheduling step on the replica itself.

Local scheduling. As explained in Appendix B, the system is highly concurrent and divides its execution model in various stages. Unfortunately, each stage is completely unaware of the nature of the operations it is responsible for, and simply handles a linked queue of self-contained runnable objects. The queue instantiation is hard-coded, which makes impossible to associate different local scheduling policies to different stages. In Hector, we generalize stages by setting the queue as a parameter in the class definition. We also enhance the runnable operation objects to include various information about the current request, making any queue implementation aware of the requested key. Moreover, any added hint from the replica selection component may be retrieved to help taking local scheduling decisions.

State propagation. Getting instantaneous state of remote peers is unfortunately not possible in distributed systems. However, even an out-of-date view can be of interest in scheduling decisions. This is why some existing proposals monitor server state characteristics (e.g., queue sizes or average service time). They often constitute the basis of a replica scoring module [23]. The challenge comes from the channels by which we retrieve information; one must periodically transmit values of interest with a sufficiently high rate to take advantage of fairly recent information. In Hector, each server holds its own copy of the *cluster state* in memory, which consists in a list of *endpoint state* entries. Each endpoint state corresponds to a specific server in the cluster, and maps a value to a property of interest, which we call a *fact*. The definition of a fact i consists in 4 functions:

- The measurement M_i, which defines how to retrieve the value to send over the network.
- The serializer S_i (resp. deserializer D_i), which encodes (resp. decodes) a value to a byte buffer.
- The aggregator A_i, which combines received values 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 a *state feedback*. When a state feedback is ready, a timestamp is added and data is transformed into a byte buffer through the fact serializer. These bytes are added to the message before being sent over the network. When the packet is received, the message handler proceeds to decoding the byte buffer and retrieves the state feedback. Each received value is added to the local endpoint state that corresponds to the message sender by applying the aggregation operation of the fact.

Workload oracles. Although they are generally unknown (or known with little precision), workload characteristics, such as data item sizes or key access frequencies, have direct influence on the effectiveness of scheduling policies. In Hector, oracles provide information about such characteristics to other modules. Their advantage is to decouple the definition of scheduling policies from the management of workload characteristics. In this way, any scheduling component may leverage this knowledge.

According to the use-case, there are various ways an oracle can build 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 workload characteristics at runtime, which can be done through machine learning techniques, statistical inference, probabilistic data structures such as Bloom filters, etc. In order to ease the evaluation process, oracles are declared in the configuration file, and each oracle is assigned an identifier. Scheduling components are not directly tied to a specific oracle definition, and the user may switch between different oracle implementations without modifying the calling component.

Implementations. We present some scheduling policies that were implemented in Hector:

- Dynamic Snitching (DS) is the default replica selection strategy in Apache Cassandra [1]. Coordinators maintain a history of service times for each server in the cluster. They assign a score to replicas by computing a moving average of recorded latencies. When processing a request, the replica with the lowest score is selected.

- C3 overcomes some weaknesses of Dynamic Snitching [23]. This strategy aggregates information on the current state of the cluster, such as the read queue size q and the service rate μ. It also maintains the count c of pending requests for each replica, and an history of latencies R. Then, a score is computed according to the 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 moving averages of observed latency, service time and queue size, while m is the number of servers.
- We design a new strategy, called Popularity-Aware (PA), that learns the popularity distribution over time. The popularity of a key is defined as the ratio between the number of accesses for this key and the total number of requests. When a key is considered popular, the probability to find the corresponding data item in the cache is higher. 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 only 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 replica.

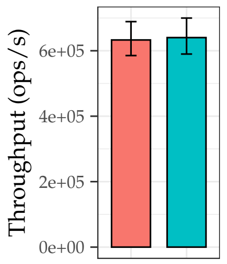
4. Experimental Evaluation

We evaluate Hector through several experiments on a real cluster. We assess that the generalization of scheduling components and newly-introduced features do not incur any significant overhead on throughput and latencies. We also illustrate the possibilities of Hector by showing how it enables easy comparisons of different algorithms in scheduling requests. Note that comparing scheduling strategies exhaustively in various contexts is left for future work.

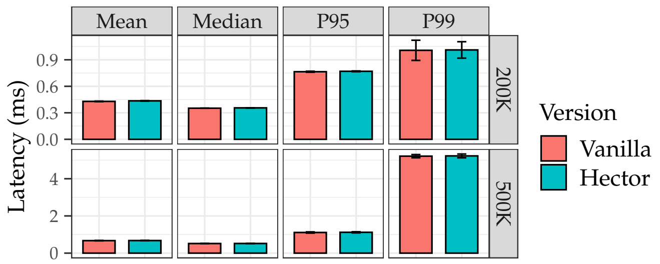
We run experiments on the large-scale experiment platform Grid'5000 [7]. We use 15 servers located in the same geographic cluster, each equipped with a 18-core Intel Xeon Gold 5220 (2.20 GHz) CPU and 96 GiB of RAM. Data is stored on each machine on a 480 GiB SATA SSD device. Servers are interconnected by a 25 Gbps network, and they all run Debian 11 GNU/Linux. The replication factor is set to 3. We evaluate the system with synthetic workloads through the industry-standard NoSQLBench, which takes care of many pitfalls related to benchmarking practice [2]. We drive benchmarking from additional nodes, located in the same rack as Hector, and we ensure that we bring enough concurrency in experiments.

Absence of overhead. We check that features of Hector do not introduce performance overhead compared to the unmodified Apache Cassandra. We make sure that both systems are identically configured, and we run them in the same conditions: key access patterns follow a Zipf law with parameter 0.9 (corresponding to a biased popularity) and data items size ranging from 1 to 100 kB. Figure 1a shows the maximum attainable throughput in each system. Figure 1b presents the mean, the median, 95th and 99th percentiles of the latency when read requests arrive according to a given throughput (200 000 and 500 000 operations per second, which in this case corresponds respectively to 30% and 80% of the maximum capacity). Each experiment is repeated 10 times (error bars indicate standard error of the mean). Tables 1-2 in Appendix D summarizes the absolute and relative differences between vanilla Apache Cassandra and Hector. For each metric, the relative difference is around 1% at most, which indicates that Hector does not incur a significant overhead on overall performance.

Cache-locality performance. We illustrate how different key access patterns may influence scheduling behavior. In particular, we show that a strategy that correctly leverages the Linux

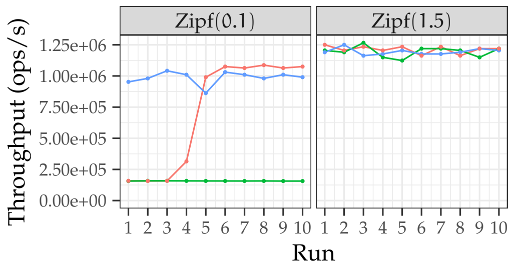


(a) Maximum attainable throughput (operations per second).

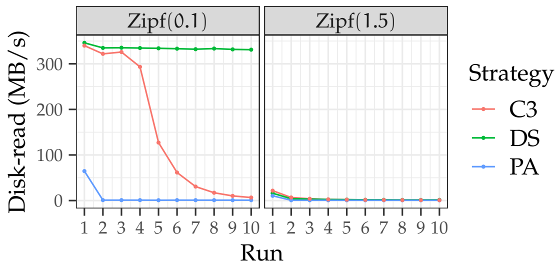


(b) Latency for two different throughputs (200 000 and 500 000 operations per second).

Figure 1: Comparison of both versions (vanilla Apache Cassandra and Hector). Keys are requested according to a Zipf law with parameter 0.9 and data items size ranges from 1 to 100 kB.



(a) Maximum attainable throughput (operations per second) for each replica selection strategy.



(b) Amount of data read on-disk over time for each replica selection strategy.

Figure 2: Effect of different key access patterns on scheduling behavior. Zipf(0.1) corresponds to quasi-uniform popularity, whereas Zipf(1.5) corresponds to heavily-biased popularity. Each consecutive run lasts for 10 minutes, and each requested data item has a size of 1 kB.

page cache may clearly outperform other algorithms under some hypotheses on key popularity distribution. In Figure 2a, we measure the maximum attainable throughput of 3 replica selection strategies (Dynamic Snitching, C3 and Popularity-Aware) for two different key popularity distributions. This results in a significant difference between strategies when the popularity skew is small (Zipf(0.1)): for instance, PA shows a maximum throughput that is more than 6 times the throughput that is obtained when using 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. When the skew is heavier (Zipf(1.5)), all strategies perform the same. An explanation of these results are the cache-locality effects occurring on replicas. Figure 2b plots the volume of data read on-disk over time on each replica. We see that PA rarely reads data directly on-disk, which indicates that it makes a very efficient use of the Linux page cache. On the contrary, DS reads more than 300 MB of data per second.

5. 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 under identical assumptions. For instance, we were able to rebuild previously-proposed algorithms in Hector, and compare their performance to a new strategy, called Popularity-Aware, that makes a more efficient use of the page cache when keys are accessed in an almost-uniform manner. Moreover, we assess 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).

Bibliography

1. Dynamic snitching in cassandra: past, present, and future. – https://www.datastax.com/blog/dynamic-snitching-cassandra-past-present-and-future, 2012. Accessed on 2022-11-10.
2. Nosqlbench docs. – https://docs.nosqlbench.io, Accessed on 2022-12-11.
3. Adnan (M. A.) et al. – A prediction based replica selection strategy for reducing tail latency in distributed systems. – In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pp. 522–526. IEEE, 2020.
4. Asyabi (E.), Bestavros (A.), Sharafzadeh (E.) et Zhu (T.). – 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*, pp. 150–164, 2020.
5. Atikoglu (B.), Xu (Y.), Frachtenberg (E.), Jiang (S.) et Paleczny (M.). – 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*, pp. 53–64, 2012.
6. Balmau (O.), Dinu (F.), Zwaenepoel (W.), Gupta (K.), Chandhiramoorthi (R.) et Didona (D.). – Silk: Preventing latency spikes in log-structured merge key-value stores. – In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 753–766, 2019.
7. Balouek (D.), Amarie (A. C.), Charrier (G.), Desprez (F.), Jeannot (E.), Jeanvoine (E.), Lèbre (A.), Margery (D.), Niclausse (N.), Nussbaum (L.) et al. – Adding virtualization capabilities to the grid'5000 testbed. – In *International Conference on Cloud Computing and Services Science*, pp. 3–20. Springer, 2012.
8. Cavalcante (D. M.), de Farias (V. A.), Sousa (F. R.), Paula (M. R. P.), Machado (J. C.) et de Souza (J. N.). – Popring: A popularity-aware replica placement for distributed key-value store. *CLOSER*, vol. 2018, 2018, pp. 440–447.
9. Dean (J.) et Barroso (L. A.). – The tail at scale. *Communications of the ACM*, vol. 56, n2, 2013, pp. 74–80.
10. Golatowski (F.), Hildebrandt (J.), Blumenthal (J.) et Timmermann (D.). – Framework for validation, test and analysis of real-time scheduling algorithms and scheduler implementations. – In *13th IEEE International Workshop on Rapid System Prototyping*, pp. 146–152. IEEE, 2002.

11. Jaiman (V.), Ben Mokhtar (S.) et Rivière (E.). – Tailx: Scheduling heterogeneous multiget queries to improve tail latencies in key-value stores. – In *IFIP International Conference on Distributed Applications and Interoperable Systems, DAIS*, DAIS, pp. 73–92. Springer, 2020.
12. Jaiman (V.), Mokhtar (S. B.), Quéma (V.), Chen (L. Y.) et Rivìere (E.). – Héron: taming tail latencies in key-value stores under heterogeneous workloads. – In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pp. 191–200. IEEE, 2018.
13. Jiang (W.), Li (H.), Yan (Y.), Ji (F.), Jiang (M.), Wang (J.) et Zhang (T.). – Cutting the request completion time in key-value stores with distributed adaptive scheduler. – In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pp. 414–424. IEEE, 2021.
14. Jiang (W.), Qiu (Y.), Ji (F.), Zhang (Y.), Zhou (X.) et Wang (J.). – Ams: Adaptive multiget scheduling algorithm for distributed key-value stores. *IEEE Transactions on Cloud Computing*, 2022.
15. Jiang (W.), Xie (H.), Zhou (X.), Fang (L.) et Wang (J.). – Haste makes waste: The on–off algorithm for replica selection in key–value stores. *Journal of Parallel and Distributed Computing*, vol. 130, 2019, pp. 80–90.
16. Kohler (E.), Morris (R.), Chen (B.), Jannotti (J.) et Kaashoek (M. F.). – The click modular router. *ACM Transactions on Computer Systems (TOCS)*, vol. 18, n3, 2000, pp. 263–297.
17. Lepers (B.), Gouicem (R.), Carver (D.), Lozi (J.-P.), Palix (N.), Aponte (M.-V.), Zwaenepoel (W.), Sopena (J.), Lawall (J.) et Muller (G.). – Provable multicore schedulers with ipanema: application to work conservation. – In *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–16, 2020.
18. Mahgoub (A.), Medoff (A. M.), Kumar (R.), Mitra (S.), Klimovic (A.), Chaterji (S.) et Bagchi (S.). – Optimuscloud: Heterogeneous configuration optimization for distributed databases in the cloud. – In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 189–203, 2020.
19. Muller (G.), Lawall (J. L.) et Duchesne (H.). – 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)*, pp. 56–65. IEEE, 2005.
20. Pabla (C. S.). – Completely fair scheduler. *Linux Journal*, vol. 2009, n184, 2009, p. 4.
21. Papagiannis (A.), Saloustros (G.), González-Férez (P.) et Bilas (A.). – Tucana: Design and implementation of a fast and efficient scale-up key-value store. – In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pp. 537–550, 2016.
22. Reda (W.), Canini (M.), Suresh (L.), Kostić (D.) et Braithwaite (S.). – Rein: Taming tail latency in key-value stores via multiget scheduling. – In *Proceedings of the Twelfth European Conference on Computer Systems*, pp. 95–110, 2017.
23. Suresh (L.), Canini (M.), Schmid (S.) et Feldmann (A.). – C3: Cutting tail latency in cloud data stores via adaptive replica selection. – In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 513–527, 2015.
24. Welsh (M.), Culler (D. E.) et Brewer (E. A.). – Seda: An architecture for well-conditioned, scalable internet services. – In *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP*, pp. 230–243, 2001.
25. Wu (X.), Xu (Y.), Shao (Z.) et Jiang (S.). – Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. – In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 71–82, 2015.
26. Wu (Z.), Yu (C.) et Madhyastha (H. V.). – Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. – In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 543–557, 2015.

27. Xu (C.), Sharaf (M. A.), Zhou (M.), Zhou (A.) et Zhou (X.). – Adaptive query scheduling in key-value data stores. – In *International Conference on Database Systems for Advanced Applications*, pp. 86–100. Springer, 2013.
28. Zhang (K.), Wang (K.), Yuan (Y.), Guo (L.), Lee (R.) et Zhang (X.). – Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, vol. 8, n11, 2015, pp. 1226–1237.
29. Zhou (X.), Fang (L.), Xie (H.) et Jiang (W.). – Tap: Timeliness-aware predication-based replica selection algorithm for key-value stores. *Concurrency and Computation: Practice and Experience*, vol. 31, n17, 2019, p. e5171.

A. 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 [12] are examples of scheduling strategies mixing global and local decisions to improve performance, particularly load balancing and tail latency [9]. Additional proposals have been made in various contexts and evaluation methods [13, 29, 15, 3, 11, 14].

Other systems leverage scheduling strategies to improve other metrics such as power consumption. Peafowl [4] 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.

Multiget APIs allow clients to request multiple values in one query. REIN [22], TailX [11], 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 trade-off 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 [18] 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 [19], or iPanema [17] following a similar approach for formally-verified, modular schedulers.

Golatowski *et al.* [10] 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 [16] is a modular router allowing to implement and test various packet scheduling policies in networks.

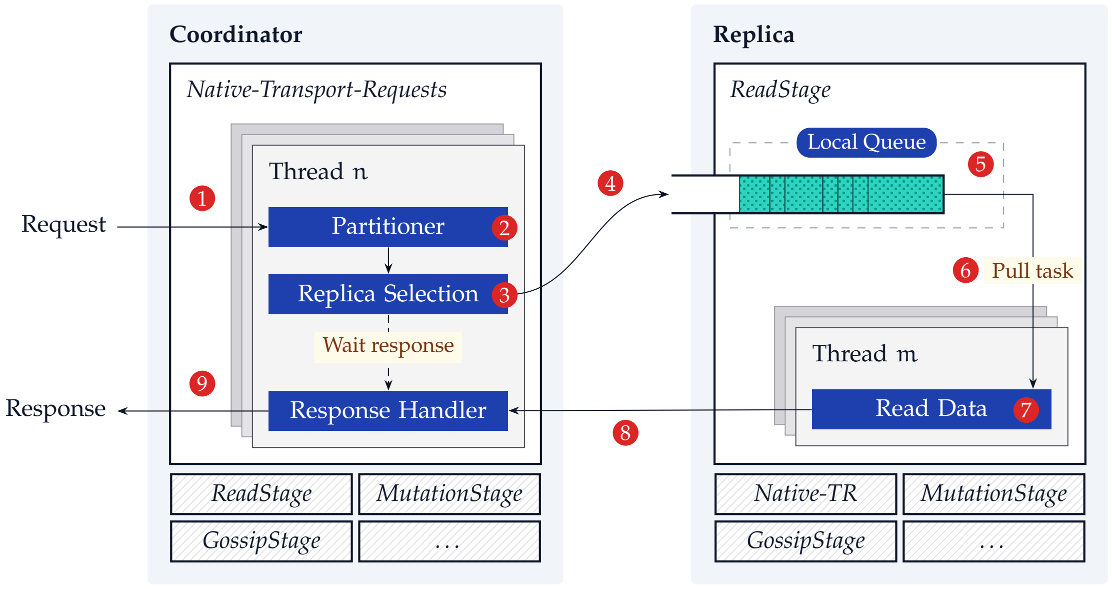


Figure 3: Detailed view of request execution in Apache Cassandra.

B. Apache Cassandra request execution model

Apache Cassandra is based on a Staged Event-Driven Architecture (SEDA) [24]. Each type of operation is processed by a different pool of worker threads (also called *stage*). For example, the reception and handling of client requests (i.e., the coordinator role) is processed by the *Native-Transport-Requests* thread pool, whereas the execution of read requests (i.e., the replica role) is processed by the *ReadStage* thread pool. Additional stages are dedicated to write request execution, internode messaging protocol, data migration, tracing, etc. Figure 3 presents a more thorough breakdown of the steps involved in the scheduling of a read request:

1. A request from a client reaches a coordinator node.
2. A worker thread from the *Native-Transport-Requests* thread pool picks this request and infers the set of replicas able to process the request from the partition key.
3. **A replica is chosen 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 the corresponding data (in memory if present or directly on disk).
8. The replica sends data to the coordinator.
9. The coordinator node 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 queue ordering and processing at step 6.

C. Example of Hector configuration

Hector components are configured through a YAML file. Each component is defined in its own section, which indicates the class to instantiate and the parameters to pass to the constructor. The state feedback module is configured through a simple list of facts to propagate.

```yaml
advanced_scheduling: true

replica_selector :
  – class_name: fr.ens.cassandra.se. selector .C3ScoringSelector
    parameters:
      – concurrency_weight: '5.0'

local_read_queue:
  – class_name: fr.ens.cassandra.se. local .read.FIFOReadQueue

oracles:
  size:
    – class_name: fr.ens.cassandra.se.oracle.KeySizeOracle

state_feedback:
  – PENDING_READS
  – SERVICE_TIME
```

D. Difference between vanilla Apache Cassandra and Hector

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

Table 1: Absolute/relative difference on the observed maximum throughput of vanilla Apache Cassandra and Hector.

Stat.	Throughput	Vanilla	Hector	Abs. diff.	Rel. diff.
mean	200K	0.429 ms	0.435 ms	0.006 ms	1.38%
	500K	0.670 ms	0.675 ms	0.005 ms	0.72%
median	200K	0.352 ms	0.355 ms	0.003 ms	1.02%
	500K	0.515 ms	0.518 ms	0.003 ms	0.56%
P95	200K	0.764 ms	0.769 ms	0.005 ms	0.62%
	500K	1.109 ms	1.119 ms	0.010 ms	0.89%
P99	200K	1.007 ms	1.011 ms	0.004 ms	0.43%
	500K	5.214 ms	5.227 ms	0.013 ms	0.25%

Table 2: Absolute/relative difference on the observed latency of vanilla Apache Cassandra and Hector, with two different throughputs.