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

L.-C. Canon\textsuperscript{1}, A. Dugois\textsuperscript{1,2}, L. Marchal\textsuperscript{2} and E. Rivière\textsuperscript{3}

\textsuperscript{1} FEMTO-ST, Univ. Franche-Comté, Besançon, France
\textsuperscript{2} LIP, ENS Lyon, Inria, Lyon, France
\textsuperscript{3} ICTEAM, UCLouvain, Louvain-la-Neuve, Belgium

August 9, 2023 – 52nd International Conference on Parallel Processing (ICPP 2023)
Key-value store

NoSQL database where each data item is identified by a unique key.

Examples: Dynamo, Cassandra

OPERATIONS

get(k) put(k,v) scan(k1,k2)
Key-value store
NoSQL database where each data item is identified by a unique key.
Examples: Dynamo, Cassandra

OPERATIONS
get(k) put(k,v) scan(k1,k2)
**Introduction to Key-Value Stores**

**Must be**
- Highly scalable
- Highly available
- Blazing fast

---

**Tail latency problem**

1 service request = many read operations.

< 1% slow ops = degraded QoS for most users.

---

Scheduling in Apache Cassandra

Request Execution

1. A node receives a client request
2. The read operation is forwarded to appropriate storage server
3. The server performs the read
4. The result is sent back to the client
Scheduling in Apache Cassandra

Request Execution

Coordinator

Native-Transport-Requests

Thread n

Partitioner

Replica Selection

Wait response

Response Handler

Request

Response

Replica

ReadStage

Local Queue

5

Pull task

6

Thread m

Read Data

Native-TR

MutationStage

GossipStage

...
Scheduling in Apache Cassandra

Prior Work


Jaiman et al. *TailX: Scheduling heterogeneous multiget queries to improve tail latencies in key-value stores.* (2020)
## Observations & Challenges

### Observations

<table>
<thead>
<tr>
<th>Feature-related</th>
<th>Challenges</th>
</tr>
</thead>
<tbody>
<tr>
<td>• 2 critical steps</td>
<td>• Poor replica selection</td>
</tr>
<tr>
<td>• Need info on cluster</td>
<td>• Poor local scheduling</td>
</tr>
<tr>
<td>• Need info on workload</td>
<td>• No info provider</td>
</tr>
</tbody>
</table>

### API-related

### Evaluation-related
### Observations & Challenges

<table>
<thead>
<tr>
<th>Observations</th>
<th>Challenges</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Feature-related</strong></td>
<td><strong>API-related</strong></td>
</tr>
<tr>
<td>• 2 critical steps</td>
<td>• Huge codebase</td>
</tr>
<tr>
<td>• Need info on cluster</td>
<td>• No unified API</td>
</tr>
<tr>
<td>• Need info on workload</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Error-prone</td>
</tr>
<tr>
<td><strong>Evaluation-related</strong></td>
<td><strong>No common baseline</strong></td>
</tr>
<tr>
<td><strong>No easy comparison</strong></td>
<td><strong>Different assumptions</strong></td>
</tr>
<tr>
<td><strong>No easy reproducibility</strong></td>
<td></td>
</tr>
</tbody>
</table>
## Observations & Challenges

<table>
<thead>
<tr>
<th>Observations</th>
<th>Challenges</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Feature-related</strong></td>
<td>• 2 critical steps</td>
</tr>
<tr>
<td></td>
<td>• Need info on cluster</td>
</tr>
<tr>
<td></td>
<td>• Need info on workload</td>
</tr>
<tr>
<td><strong>API-related</strong></td>
<td>• Poor replica selection</td>
</tr>
<tr>
<td></td>
<td>• Poor local scheduling</td>
</tr>
<tr>
<td></td>
<td>• No info provider</td>
</tr>
<tr>
<td><strong>Evaluation-related</strong></td>
<td>• Huge codebase</td>
</tr>
<tr>
<td></td>
<td>• No unified API</td>
</tr>
<tr>
<td></td>
<td>• Difficult to extend</td>
</tr>
<tr>
<td></td>
<td>• Error-prone</td>
</tr>
<tr>
<td></td>
<td>• No common baseline</td>
</tr>
<tr>
<td></td>
<td>• Different assumptions</td>
</tr>
<tr>
<td></td>
<td>• No easy comparison</td>
</tr>
<tr>
<td></td>
<td>• No easy reproducibility</td>
</tr>
</tbody>
</table>
Introducing Hector

Overview

Scheduling Framework Hector

Hector is a fully-integrated scheduling framework built in Apache Cassandra.

https://github.com/anthonydugois/hector

Apache Cassandra 4.2

Modular components

No conflict

Simple API

Single config file

...
Introducing Hector

Modular Components

- **Replica Selection**
  Select the “best” replica to serve a request

- **Local Scheduling**
  Schedule local operations in optimized order

- **Cluster State**
  Propagate info on the cluster state to help scheduling

- **Workload Oracle**
  Gather information on the workload to help scheduling

---

Introducing Hector
Workflow

Evaluation

1. Setup environment
2. Define scheduling settings
   ▶ Optional: adapt implementations
3. Run experiments
4. Go to Step 2

Example of config file

```yaml
replica_selector:
  - class_name: hector.C3ScoringSelector
    parameters:
      - concurrency_weight: '5.0'
local_read_queue:
  - class_name: hector.FIFOReadQueue
state_feedback:
  - PENDING_READS
  - SERVICE_TIME
```

Introducing Hector
Schedulers

Default schedulers in Apache Cassandra

**Replica Selection**
- **Dynamic Snitching (DS)**
  Periodically compute a score based on latency history; select the replica with lowest score

**Local Scheduling**
- **First Come First Served (FCFS)**
  Process operations in order of arrival

**Replica Selection**

(+ Cluster State)

C3

Continuously compute a score based on latency history, queue size, pending operations; select the replica with lowest score
Introducing Hector

Schedulers

Idea: leverage the Linux page cache to reduce the number of disk accesses

SST = Sorted String Table file
Introducing Hector

Schedulers

Idea: leverage the Linux page cache to reduce the number of disk accesses

SST = Sorted String Table file
Introducing Hector

Schedulers

Idea: leverage the **Linux page cache** to reduce the number of disk accesses

SST = Sorted String Table file

**Canon et al. Hector: A Framework to Design Scheduling Strategies in Persistent Key-Value Stores. ICPP 2023.**
Introducing Hector

Schedulers

Idea: leverage the Linux page cache to reduce the number of disk accesses

Disk

SST a
SST b
SST c
SST d
SST e
SST f
SST g

Page cache

SST a
SST b
SST c

SST = Sorted String Table file
Introducing Hector

Schedulers

Idea: leverage the Linux page cache to reduce the number of disk accesses

SST = Sorted String Table file
Introducing Hector
Schedulers

Idea: leverage the Linux page cache to reduce the number of disk accesses

Page cache

Disk

SST a
SST b
SST c
SST d

SST e
SST f
SST g

SST = Sorted String Table file
Introducing Hector

Schedulers

Idea: leverage the **Linux page cache** to reduce the number of disk accesses

Disk

- SST a
- SST b
- SST c
- SST d
- SST e
- SST f
- SST g

Page cache

- SST e
- SST a
- SST c
- SST d

SST = Sorted String Table file
Introducing Hector

Schedulers

Idea: leverage the Linux page cache to reduce the number of disk accesses

Disk

SST a
SST b
SST c
SST d

Page cache

SST e
SST a
SST b
SST d

SST g

SST = Sorted String Table file
Introducing Hector
Schedulers

Idea: leverage the Linux page cache to reduce the number of disk accesses

Disk

Page cache

SST = Sorted String Table file

SST a
SST b
SST c
SST d
SST e
SST f
SST g
SST a
SST b
SST c
SST e
Introducing Hector
Schedulers

Replica Selection (+ Workload Oracle)

Popularity-Aware (PA)
According to popularity of keys, favor page cache hit (low popularity) or load balancing (high popularity)
Introducing Hector
Schedulers

Idea: assign priorities to operations.

Local Scheduling (+ Metadata) Random Multi-Level (RML)
Process operations in order of priority
Introducing Hector

Schedulers

Idea: assign priorities to operations.

Local Scheduling (+ Metadata)
Random Multi-Level (RML)
Process operations in order of priority

In this talk: “fast” operation = high priority
Experimental Evaluation

Settings

- Grid’5000 testbed
- 15 identical servers
  - 18-core Intel Xeon Gold 5220 + 96 GiB RAM + 480 GiB SSD
- 150 GiB of data per server
- 5 benchmark clients
- Synthetic workload
- Production settings

Hector Overhead

Cache-Locality Effects

Heterogeneous Scheduling
Experimental Evaluation
Hector Overhead

Throughput (kops/s)

<table>
<thead>
<tr>
<th></th>
<th>Cassandra</th>
<th>Hector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>200 kops/s</td>
<td></td>
</tr>
<tr>
<td>Median</td>
<td>500 kops/s</td>
<td></td>
</tr>
<tr>
<td>P90</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P99</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Latency (ms)

<table>
<thead>
<tr>
<th></th>
<th>Cassandra</th>
<th>Hector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Median</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P90</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P99</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Experimental Evaluation

Cache- Locality Effects

![Graph showing throughput vs runtime for different strategies and Zipf distributions.]

**Settings**
- Item size: 1 kB
- Zipf(0.1): quasi-uniform
- Zipf(1.5): heavily-skewed
Experimental Evaluation

Cache-Locality Effects

Zipf(0.1) and Zipf(1.5) distributions are compared for different runtime periods. The graph shows the disk-read rate (MB/s) over time (min) for three strategies: DS, C3, and PA. The performance metrics are measured in terms of disk-read rate, with DS showing the best performance followed by C3 and PA. The legend on the right indicates the strategies for each color: DS (red), C3 (green), and PA (blue). The x-axis represents the runtime in minutes, and the y-axis represents the disk-read rate in MB/s.
Experimental Evaluation
Heterogeneous Scheduling

**Settings**
- Small item size: 1 kB
- Large item size: 1 MB
- Small/large ratio: 3:1
- Uniform popularity

![Graph showing throughput vs. arrival rate for FCFS and RML strategies with different settings.](image)
Experimental Evaluation

Heterogeneous Scheduling

![Graph showing latency vs. arrival rate for mean, median, P90, and P99 metrics with two strategies: FCFS and RML.](image-url)
Conclusion

Hector benefits

- Easier implementation
- Comparisons over baseline
- Testing new ideas
- No overhead

Future work

- Support multi-get operations
- Exhaustive evaluation campaign
Thank you for your attention!

https://github.com/anthonydugois/hector