Justin: Hybrid CPU/Memory Elastic Scaling for Distributed Stream Processing*

Donatien Schmitz[1], Guillaume Rosinosky[2], and Etienne Rivière[1]

[1] ICTEAM, UCLouvain, Belgium, {first.last}@uclouvain.be
[2] IMT Atlantique, Nantes Université, École Centrale Nantes, CNRS, Inria, LS2N - UMR 6004, France, guillaume.rosinosky@inria.fr

Abstract. Distributed Stream Processing (DSP) engines analyze continuous data via queries expressed as a graph of operators. Auto-scalers adjust the number of parallel instances of these operators to support a target rate. Current auto-scalers couple CPU and memory scaling, allocating resources as one-size-fits-all packages. This contrasts with operators' high diversity of requirements.
We present Justin, an auto-scaler that enables hybrid CPU and memory scaling of DSP operators. Justin monitors both CPU usage and the performance of operators' storage operations. Its mechanisms enable fine-grain memory allocation for tasks upon a query reconfiguration. The Justin policy identifies individual operators' memory pressure and decides between adjusting parallelism and/or memory assignment. We implement Justin in Apache Flink, extending the Flink Kubernetes Operator and the DS2 CPU-only auto-scaler. Using the Nexmark benchmark, our evaluation shows that Justin identifies suitable resource allocation in as many or fewer reconfiguration steps as DS2 and supports a target rate with significantly fewer CPU and memory resources.

Keywords: Distributed Stream Processing · Resource Management · Elastic Scaling · Hybrid Scaling · Apache Flink

1 Introduction

Stream Processing allows continuous data analysis in a large variety of applications [14]. Supporting stream processing over large volumes of incoming data requires distributing computation over multiple machines. Distributed Stream Processing (DSP) engines, such as Apache Storm [2], Spark Streaming [38], or Apache Flink [1], emerged to address the many challenges associated with the distribution and orchestration of parallel stream processing. This includes fault tolerance [9], connection to input sources and destinations, and support for elastic scaling, i.e., the ability to dynamically adapt the amount of computational resources to sustainably support a target *rate* of input events.

A stream processing query is a directed graph of *operators* performing computation on incoming events. Elastic scaling assigns, at runtime, each operator

* Artifacts available in https://doi.org/10.5281/zenodo.15209785

to several *tasks*, i.e., individual threads of execution. The flow of events is distributed to these tasks to be processed in parallel. Numerous auto-scalers have been proposed in the literature [10,28]. Apache Flink recently integrated a variant of the DS2 [17] auto-scaler in its Kubernetes Operator [3]. DS2 determines the processing capacity of tasks based on a measure of their *busyness*, the fraction of CPU time effectively spent processing events. It derives a new configuration mapping operators to a required number of tasks to sustain a target input rate while considering cascade effects between operators' loads.

Motivation. Existing auto-scalers couple CPU and memory allocation [9]. Scaling out an operator (i.e., adding more tasks) adds a proportional amount of memory. This coupled, one-size-fits-all allocation clashes with the heterogeneous memory requirements of operators. Some operators, such as a *filter* or a *map*, are stateless and do not require more than a minimal amount of memory to process events efficiently. In contrast, other operators, such as *joins* or *group by* over long windows, maintain state across the processing of many events [36].

Intuitively, stateful operators with under-allocating memory perform poorly: The necessary state may not fit in the main memory, and accesses may resort to on-disk storage. In contrast, ample memory allocation (e.g., as the operator has been scaled out to multiple tasks) should not yield better performance than a smaller but sufficient allocation.

In reality, the relation between event processing performance, state access latency, and memory requirements is more subtle than this simple intuition. State-of-the-art state backends for stream processing use a Log-Structured-Merge tree (LSM) [21, 25]. An LSM combines tables and caches in memory with bulk storage on disk. They use optimizations to favor write performance and minimize the wear of modern SSD drives. For instance, Flink uses the LSM-based RocksDB [4, 12] for production deployments. The performance in response to the available memory of RocksDB depends heavily on the nature of the workload [7]. Workloads formed mainly of *write* operations do not benefit from a large memory allocation. In contrast, workloads dependent on *read* operations do, in proportion to the size of their working set. The nature of an operator's access to its state is not predictable, as it often depends on the characteristics of its input events.

Contributions. We present Justin, a hybrid auto-scaler that considers *both* CPU and memory allocation when reconfiguring DSP queries. In contrast with previous approaches, Justin's auto-scaling policy can decide to scale *up* (i.e., adding more memory to each task) or scale *out* (adding more tasks) depending on runtime indicators on state access performance. We integrate Justin in Apache Flink and the Flink Kubernetes Operator [3], extending the current elastic scaling engine based on DS2 [17].

Our contributions and the outline of the remainder of this paper are as follows. We start with the necessary background and terminology about Flink (§2). We discuss state management and RocksDB and analyze, using microbenchmarks, the impact of workload characteristics on state lookup performance and response to memory availability (§3).

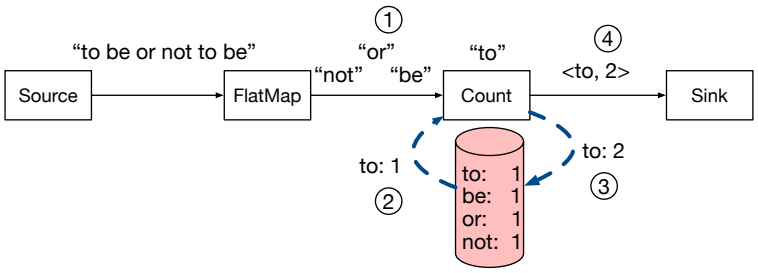


Fig. 1: A WordCount query using four operators.

We then present Justin policies and mechanisms (§4). We detail DS2, Flink's current elastic scalers. We present how Justin collects metrics about a running query's resource usage and storage performance in RocksDB. Based on these metrics, Justin's auto-scaling policy adjusts DS2 decisions and derives CPU and memory allocations. These allocations are enacted by mechanisms for fine-grain resource allocation in Flink and the Flink Kubernetes Operator [3].

We evaluate Justin in Flink on a 7-node cluster under high loads and compare it to DS2 using the Nexmark benchmark [35] (§5). Our results show Justin produces configurations with 48% less CPU and 27-28% less memory for `q8` and `q11`, two complex stateful queries while requiring the same or fewer reconfiguration steps than DS2.

We review related work on hybrid vertical/horizontal scaling (§6) and conclude the paper by identifying future directions (§7).

2 Background

We provide background information on stream processing. We use Flink's terminology but note that concepts are similar in other stream processing engines. We detail state management and elastic scaling in Sections 3 and 4.

Stream processing. A stream processing query is a dataflow graph where vertices are *operators* $o_i \in O$ and edges represent flows of events between these operators. Operators can support arbitrary computation but are often selected from a library of classical transformations or compiled from SQL [26].

Operators can be single-input, such as a *filter* (i.e., letting through a subset of events), a *map* (individual transformation of events), or a *group by* followed by aggregation. Others consume multiple streams, e.g., *joins*. Commonly, operators such as aggregates and *joins* compute over a *window* defined as a count of events or a period [36]. These operators are *stateful*: they must keep track of information related to events over this window. In contrast, operators that process events in isolation (*map*, *filter*, etc.) are *stateless*.

Word count example. Figure 1 presents *word count*, a classic example query implemented using three operators. This query counts occurrences of individual words in a stream of sentences. The *Source* is a specific operator injecting sentences as new events, e.g., from disk or a stream store such as Apache Kafka [18]. A *FlatMap* operator splits these sentences into multiple events, one for each word

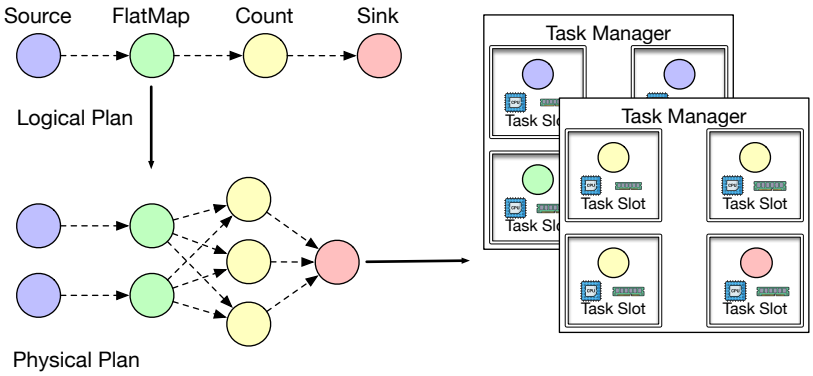


Fig. 2: Word count query deployed on two Task Managers and eight Task Slots.

(①). The *Count* operator combines a *group by* and a *sum* aggregate to count occurrences of each word over a time window. The processing of each event by this operator requires fetching the current count from the storage backend (②) and updating it (③). Finally, a *Sink* operator receives and stores outputs, e.g., to Kafka (④). In this example, the *Count* operator is stateful; others are stateless.

Query execution. A query's logical plan is executed by a physical plan where each operator can be supported by several processing *tasks*, as illustrated by Figure 2. Each task is a single thread of execution. The number of tasks of an operator is also denoted as its *parallelism*. In Flink, the execution of tasks is supported by a fleet of processes (Java Virtual Machines) called Task Managers (TMs) orchestrated by a centralized Job Manager (JM). Each TM has a predefined number of identical Task Slots (TSs). Each TS can run one task. CPU resources allocated to a TM are divided between its TSs. In this paper, we consider the standard one-core-per-task model. We discuss the allocation of TM memory in the next section and evaluate its impact on performance.

3 Impact of Memory on Operators' Performance

We study in this section the relationship between memory allocation and task performance in Flink. The takeaways of this section guide the design of our hybrid CPU/memory auto-scaler, Justin, detailed in the next section.

RocksDB. The state of stateful operators' tasks must be persisted across the processing of different events. The storage must allow for reconfiguration and restoration after failures [9]. Furthermore, this state may be arbitrarily large and not fit into memory. While an in-memory storage backend is available for testing in Flink, production deployments use RocksDB [4, 12], a state backend using a Log Structured Merge (LSM) tree and offering a key/value interface.

An LSM tree, illustrated on the right of Figure 3, is a data structure optimized for write-heavy workloads and reducing wear on the storage device, typically an SSD. Writes are buffered in memory (in a MemTable) and later flushed to disk in sorted order, reducing random I/O. A hierarchy of sorted SSTables

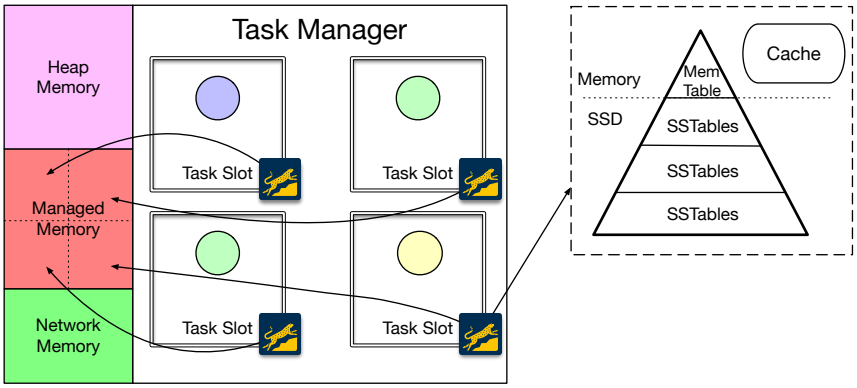


Fig. 3: Memory allocation between tasks and RocksDB LSM tree.

(Sorted String Tables) files is periodically merged into larger ones, reducing fragmentation and read amplification. Data is stored across multiple levels of this hierarchy, with newer data at higher levels and older data compacted into lower levels. As a result, reads may require searching SSTables across multiple levels, increasing their latency. To mask these costs, RocksDB uses an in-memory *cache* of recently accessed data.

The MemTable, implemented as a skip list, is used to buffer writes before consolidating them to SSTables. Using a large MemTable has no real impact on write latencies, as consolidation is performed at the granularity of the first-level SSTable (of 64 MB). In contrast, read latency is directly impacted by the size of the cache and its relation to the task's working set size (i.e., the set of keys the task frequently accesses over a period). Tasks performing frequent reads, as for the *Count* operator of the word count in Figure 1, may have to resort to costly exploration of SSTables on disk if the cache is not large enough. Tasks performing only or mostly writes are not impacted by the cache size.

TM memory allocation. Memory sharing between TSs on the same TM depends on the JVM's memory segmentation. On-heap memory, used by Flink operators for creating Java objects and subject to garbage collection, is shared among all tasks. The same applies to network memory used for communication buffers. While there is no isolation between threads for access to these segments, a TM reserves a minimum amount for each TS. In contrast, *managed memory* is specific to a thread and is not subject to garbage collection. It is used by the RocksDB instance local to each task to store its MemTable and cache.

Takeaway 1. Managed memory is reserved for all task slots in equal amounts. This memory is wasted for stateless tasks that do not use RocksDB. Stateful tasks get a one-size-fits-all allocation, regardless of their requirements.

Microbenchmarks. We evaluate the impact of memory allocation on performance using microbenchmarks. We use a single operator. The input stream is formed of events of 1,000 B. Each event includes a key generated uniformly at random between 0 and 1,000,000 and a random payload. The RocksDB state backend is pre-populated with a value associated with each key. We consider

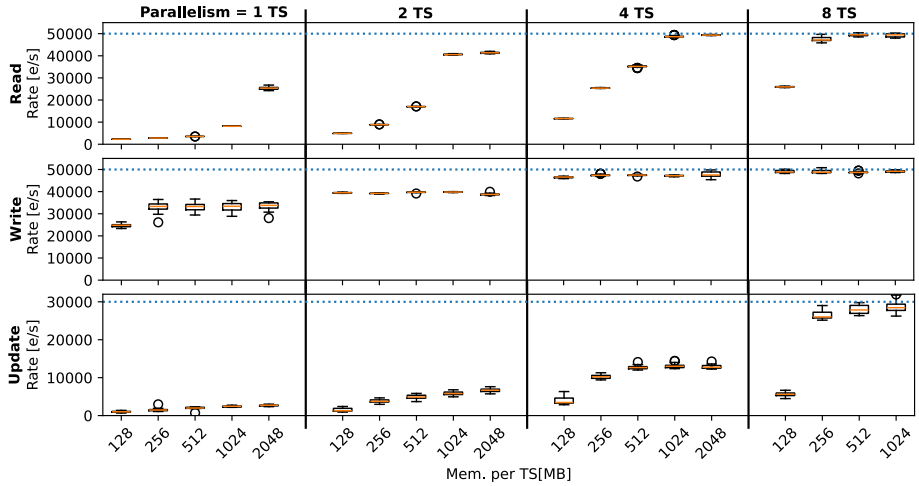


Fig. 4: Evaluation of multiple memory-cpu configurations on three different state access patterns. Benchmarks show the maximum rate reachable against a target rate (dotted line). The impact of memory on rate significantly differs depending on the workload, read-only being the more profitable.

three workloads. In the **Read** workload, the operator reads the value associated with the key in the event. In the **Write** workload, it replaces the value associated with this key without reading the previous value. In the **Update** workload, the operator reads the current value and then overrides it with the event's payload. We use a target rate of 50,000 events per second for the Read and Write workloads and 30,000 events per second for the Update workload.

Our results are presented in Figure 4. We consider 19 configurations for each workload, with an operator's parallelism ranging from 1 to 8 tasks and managed memory allocation of 128 to 2,048 MB. In any memory allocation, the MemTable size is at most 64 MB, and the rest is used for the cache. By default, Flink prioritizes the allocation of at least half of the memory to the cache, possibly reducing the size of the MemTable, which is allocated as a power-of-2 granularity. As a result, an allocation of 128 MB of memory results in a 32 MB MemTable and a 96 MB cache, but allocations of 256 MB or 512 MB result in a 64 MB Memtable and, respectively, 192 MB and 448 MB caches. We denote a configuration with t tasks and m MB of memory as $(t; m)$. We run each configuration for 10 minutes. We measure aggregate rate every 5 seconds and present the distribution as a box plot. The dotted line indicates the target rate. We use specific source operators that produce events at the maximal possible speed, subject to back pressure from the measured operator and capped by this target rate.

We observe that the target rate is sustained for the Read workload starting from (4; 1,024) or (8; 512). Below that, we observe that the cache hit rate (not shown on the plot) is very low, impacting processing time for every event. With 256 MB of memory, even the 8-task configuration is slightly below the target rate, possibly requiring further scale-out and the use of a lot of CPU resources.

Takeaway 2. Scaling out a read-intensive operator without appropriate memory allocation can lead to inefficient scale-out operations. The cache hit rate is a key metric for determining when the cache size is insufficient and scaling up.

The Write workload shows constant performance at all parallelisms with all memory configurations, except the smallest (1; 128), which is slightly below (1; 256). This lower performance is due to the smaller MemTable size. The target rate is reached with a parallelism of 8, with 4 tasks being very close.

Takeaway 3. The size of the cache does not impact write performance. Write-dominated operators should favor scale-out to scale-in.

Finally, the evaluation of the Update workload shows that only an 8-task configuration can sustain the workload. We also observe a *plateau* effect for 4 and 8 tasks, where adding memory does not result in significant gains (in contrast with the Read workload, where these gains are more linear). Configurations with insufficient memory (128 MB) cannot sustain the rate regardless of parallelism.

Takeaway 4. It is necessary to vertically scale (add memory) above a minimum threshold. If a scale-up does not improve performance significantly, it is preferable to scale out.

4 Justin: Hybrid CPU/memory Auto-Scaler

Building upon the takeaways of the previous section, we design Justin as a hybrid auto-scaler that decides on scaling *up* or scaling *out* an operator depending on its needs and avoids allocating memory to tasks that do not need or use it. Justin builds upon DS2 [17], Flink's current auto-scaler. We start with a recap of DS2 principles. Then, we detail the metrics collected by Justin. We present an auto-scaling policy that uses these metrics and explain how we enable dynamic, heterogeneous memory allocation for tasks.

Elastic scaling and DS2. Elastic scaling determines the appropriate level of parallelism for all operators. From a default configuration (parallelism of tasks and memory per task) at time $t = 0$, reconfigurations are triggered, based on observations, at discrete times $t > 0$. We define a configuration C^t as a map between operators $o_i \in O$ where $o_i.p^t$ is the parallelism p (number of tasks) of operator i at time t.

DS2 [17] uses a primary metric, the busyness of operators, averaged across their tasks. Busyness is the fraction of CPU time spent processing events, i.e., not idling or waiting due to back pressure. A reconfiguration trigger is a high busyness for one of its operators in addition to backpressure from its upstream operator(s), indicating that the query's capacity is insufficient to cope with the current rate. When triggered at time t, DS2 determines a new level of parallelism $o_i.p^t$ for each operator such that the resulting busy rate is below a target. For this purpose, it uses linearity assumptions and considers that the load will be equally distributed among the scaled-out operator's tasks. This computation accounts for the cascade effects between operators: scaling out an operator increases its capacity, resulting in a higher rate for downstream operators and the need to scale them out as well. The new configuration is applied via a query reconfiguration, possibly resulting in state transfers between RocksDB instances [9].

DS2 typically requires several reconfiguration steps until it reaches a stable configuration for a target rate, and setting appropriate thresholds for triggering reconfiguration and target busyness levels requires careful tuning.

Integrating memory-awareness. Justin requires memory-centric indicators in addition to CPU ones, such as busyness. We note, however, that there can be a coupling between memory and CPU indicators. A task performing high-latency state accesses may have a high busyness, as state accesses are accounted for as part of the event processing time. State access latency allows distinguishing between the two cases. High average access latency $o_i.\tau^t$ indicates that a significant fraction of accesses by o_i's tasks used the disk and that scaling out may not be the best option until the cache is appropriately dimensioned. Furthermore, RocksDB exports the number of cache hits and misses, from which we can derive an average *cache hit rate* $o_i.\theta^t$ for operator o_i's tasks. We collect these two metrics using Prometheus [11]. As for other metrics, these are collected and averaged over a period.

4.1 Justin Policy Building BLocks

The elastic scaling policy of Justin extends DS2, specifically its implementation in Flink and the associated Kubernetes support operator [3]. This choice allows building upon the complex engineering, integration, and parameter tuning already made by the open-source community and improves impact potential.

Memory allocation. Justin allocates heterogeneous memory to the tasks of different operators. $o_i.m^t$ represents in a configuration C^t the managed memory, in MB, allocated to all tasks of an operator o_i. To facilitate the mapping of tasks to task managers, we allocate memory in *levels*. Each new level corresponds to double per-task memory, from a minimum to a maximum (*maxLevel*). For $o_i.m^t = x$, we allocate 2^x times the minimum of managed memory for stateful tasks. If, for instance, the minimum managed memory is 128 MB, $o_i.m^t = 0$ means 128 MB, $o_i.m^t = 1$ means 256 MB, etc. Stateless tasks are allocated no managed memory, expressed as $o_i.m^t = \perp$.

Decisions history. DS2 does not maintain a history of scaling decisions, as it only takes these in one direction (horizontally). In contrast, Justin can choose between scaling out or up. Keeping a decision history helps determine whether they improved the query capacity. Past configurations are available in $C^0 \dots C^{t-1}$. A boolean $o_i.v^t$ indicates that, in C^t, the scaling decision involved *scaling up* (vertically) the memory of operator o_i's tasks (i.e., $o_i.v^t = \top \implies o_i.m^t > o_i.m^{t-1}$).

4.2 Justin Policy Algorithm

Algorithm 1 is called when the capacity of the query is insufficient and requires a reconfiguration. We use the unmodified DS2 trigger based on busyness and backpressure metrics.

The key principle of Justin's policy is, in appropriate situations, to prioritize a vertical scaling action to a horizontal scaling action decided by DS2. It uses memory efficiency metrics to determine if there is a gain potential and leverages

Algorithm 1: Justin's hybrid elastic scaling policy

Parameters: $\Delta_\theta \leftarrow 80\%$; $\Delta_\tau \leftarrow 1ms$; $maxLevel \leftarrow 3$
Result: A new configuration C^t

```
1  C^t ← DS2()                                   // Obtain initial configuration from DS2
2  for o_i ∈ C^t do                              // Iterate over all operators
3      if o_i is stateless                       // No recorded RocksDB access?
4          o_i.m^t ← ⊥                           // Disable managed memory for o_i
5      else
6          if o_i.p^t ≠ o_i.p^{t-1}              // o_i's capacity insufficient?
7              if o_i.v^{t-1}                    // Vertical scaling used last time?
8                  if (o_i.θ^t > o_i.θ^{t-1} ∨ o_i.τ^t < o_i.τ^{t-1})   // Did it improve?
9                      if (o_i.m^{t-1} + 1) < maxLevel                  // Can scale-up?
10                         o_i.p^t ← o_i.p^{t-1}                        // Cancel scale out
11                         o_i.m^t ← o_i.m^{t-1} + 1                    // Increase memory further
12                         o_i.v^t ← True
13                 else                                                 // Did not improve?
14                     o_i.m^t ← o_i.m^{t-1} − 1                        // Roll-back scale in
15             else
16                 if (o_i.θ^t < Δ_θ ∨ o_i.τ^t > Δ_τ) ∧ o_i.m^{t-1} + 1 < maxLevel
                                                 // Could vertical scaling be useful?
17                     o_i.p^t ← o_i.p^{t-1}                            // Cancel scale out
18                     o_i.m^t ← o_i.m^{t-1} + 1                        // Attempt increasing memory
19                     o_i.v^t ← True
20 return C^t                                    // Return the updated configuration
```

insights from past scaling decisions to evaluate whether they improved capacity. The policy starts by calling the unmodified DS2 (Line 1). It then iterates over all operators. It first identifies if the operator is stateless, which is indicated by the absence of RocksDB-specific metrics in Prometheus (Line 3). If it is, we remove its portion of managed memory (Line 4) and keep the parallelism given by DS2.

For stateful operators, we consider those where a re-scaling decision is proposed by DS2 (Line 6). Other operators' capacities are deemed sufficient and do not need to scale. We then distinguish between two cases: if the operator was vertically scaled the previous time (Lines 7–14) or not (Lines 16–19).

If the operator was previously scaled up, we determine if this led to an improvement. As discussed in Section 3, this depends on the number of writes vs. reads, the size of the working set, among other factors. A scaled-up operator that does not give better capacity is unlikely to improve with further scale-up. We assess this improvement by comparing the cache hit ratio and state access latency of the current and previous periods (Line 8).[3] If there was an improvement and the operator is not already at the maximum memory level, we cancel the scale-out (Line 10) and replace it with scale-up (Line 11). If there was no

[3] The algorithm uses strict inequality signs $<$ and $>$ for clarity, but we can use a minimum amount of improvement as a percentage as well, implementing a hysteresis.

improvement, we cancel the previous scale-up (Line 14) to avoid wasting memory. The parallelism recommended by DS2 will thus apply using the previous memory configuration.

If an operator was not scaled up already (or was so earlier than $t-1$), we evaluate if the cache hit rate is *below* a threshold Δ_θ, indicating insufficient cache size for reads or if average state access latency is *over* a threshold Δ_τ, indicating a significant fraction of operations need costly disk/SSD accesses. In this case, we cancel the scale-out decision (Line 17) and increase the operator's memory level (Line 18). Otherwise, we apply the recommended parallelism.

4.3 Implementation

Justin is implemented in version 1.18 of Apache Flink and extends the Kubernetes Operator [3] where DS2 [17] is implemented. Our changes account for about 1,500 LoC. In addition to the policy described earlier in this section, Justin relies on mechanisms allowing the enactment of the decisions: (1) the support of Task Slots (TS) with heterogeneous memory allocations; (2) the placement of tasks to TS across different Tasks Managers (TMs), minimizing fragmentation, and (3) the creating when necessary of newer TMs.

Flink 1.18 supports an API allowing to specify custom resource configurations for operators when submitting a new query (number of cores and quantity of heap, network, and managed memory). We extend Flink's Adaptive Scheduler module to allow such changes to be enacted at runtime via a REST API. The scheduler maps heterogeneous memory demand to the available TMs using a standard multidimensional Bin-Packing algorithm [20].

The Flink policy is implemented as part of Flink's Kubernetes Operator [3]. An API allows the policy's parameters, such as trigger or decision thresholds, to be dynamically adapted at runtime. This allows control over the elastic scaling of long-running queries without redeployment. When the Bin-Packing algorithm for task placement cannot find enough space with existing TMs, the Kubernetes Operator can spawn a new TM "pod" in the cluster.

5 Evaluation

Our evaluation aims to answer the following research questions:

- For a given target rate, is Justin able to converge to efficient configurations in terms of CPU and memory usage?
- Does the integration of vertical and horizontal scaling lead to a longer convergence time compared to horizontal (i.e., CPU-only) scaling?

We compare Justin and the Flink DS2 implementation and use the obtained rate (i.e., the capacity of a configuration), the sum of assigned CPU cores, and the sum of allocated memory as metrics. We use as workload the standard Nexmark Benchmark [35]. Nexmark simulates an auction system and provides a set of representative queries for the evaluation of batch and stream processing systems.

We use the same six queries as used in the original evaluation of DS2 [17]. All queries use a single source and a single sink operator. `Q1` produces currency

conversions using one Map operator. `Q2` uses a Filter operator to select bids with a specific identifier. `Q3` classifies sales based on location and categories and maps them to specific auctions, using an incremental join operator over the complete stream (i.e., without windowing) and two filter operators. `Q5` determines the auctions with the most bids in a period and uses one stateful operator and a group-by-aggregate over a sliding window. `Q8` monitors new active users over a period and uses a tumbling-windowed join as a stateful operator. Finally, `Q11` monitors user sessions by computing the number of bids each user makes while active. It uses one operator, a stateful group-by-aggregate, over a session window.

Experimental setup. We alternatively deploy Flink with Justin or DS2 on a testbed of 7 nodes, each equipped with two 10-core Intel Xeon E5-2630L v4 CPUs, 128 GB of memory, and a 400 GB SSD. Nodes are connected with 10-Gbps Ethernet. One node hosts the Kubernetes controller, and another hosts Prometheus and Grafana. We deploy the Job Manager on another dedicated node. The four remaining nodes are used to host TMs.

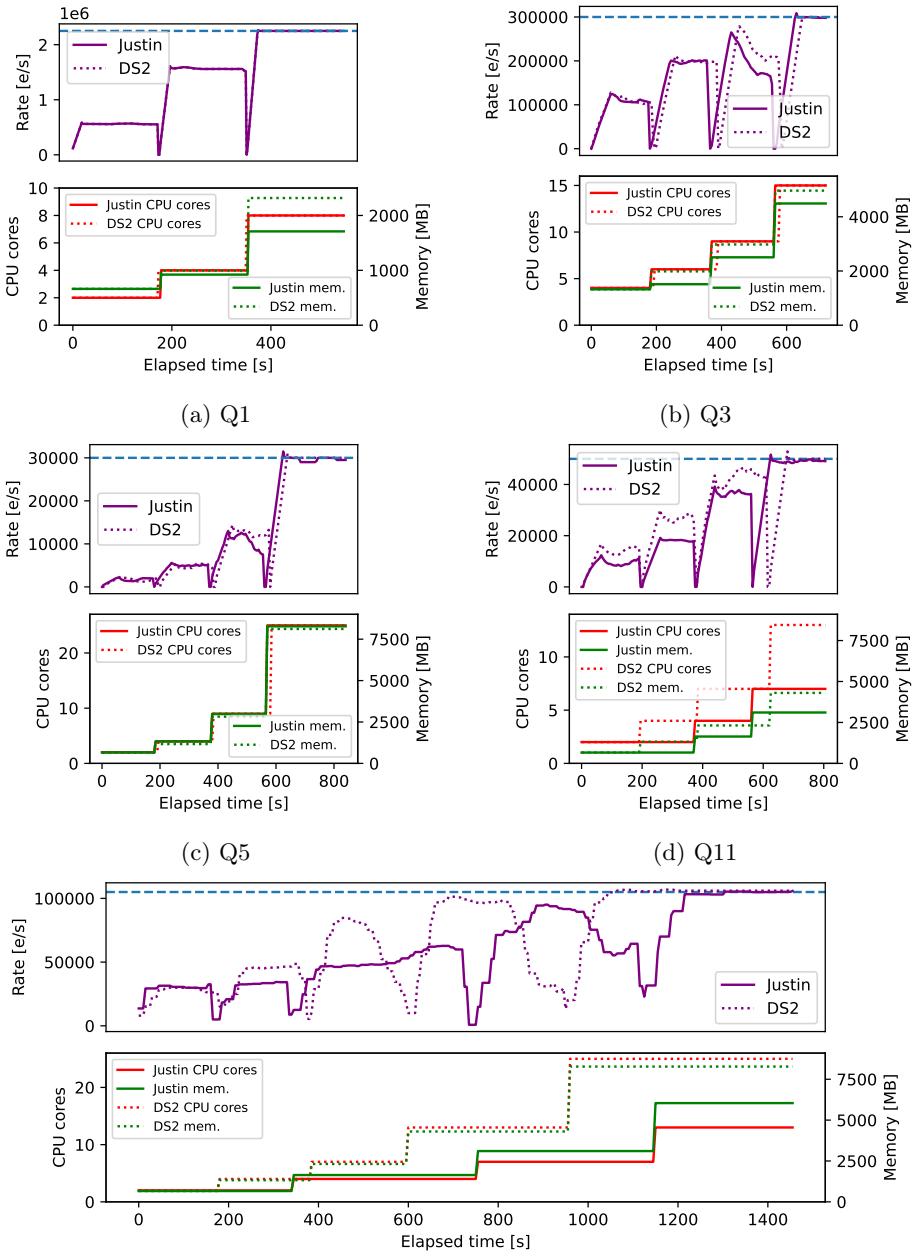
We configure each TM to use 4 CPU cores and 2 GB of memory, shared among 4 TSs. Flink reserves a portion of the available memory for framework management data and JVM-specific data. The default amount of managed memory per TS is 158 MB. With DS2, all tasks get this fixed amount. With Justin vertical scaling, a task may receive from 158 MB ($o_i.m^t = 0$) to 632 MB ($o_i.m^t = 2$, i.e., after two scale-up actions) of managed memory.

We set elastic scaling trigger parameters to keep the average busyness of operators between 20% and 80%. Through experimental analysis, we identify two thresholds that allow suitable identification of memory pressure for some operator's tasks: A cache hit rate over $\Delta_\theta = 80\%$ and an average state access latency over $\Delta_\tau = 1ms$. Similarly to DS2, we use short 2-minute decision windows and a 1-minute stabilization period. If no scaling is triggered at the end of the window, the autoscaler waits for the following full metrics window collection. Metrics are collected and aggregated with a granularity of 5 seconds.

5.1 Results

Figure 5 presents elastic scaling results for Justin and DS2. We show the achieved rate (capacity) and the resource allocation for the two auto-scalers as a function of time. The objective is for sources to reach the target rate indicated by a thin, dashed blue line after a few reconfigurations. The top plots show the achieved rate, while the bottom plots show the overall CPU and memory consumption (including heap, network, and managed memory). Note that, as in the DS2 paper [17], we exclude sources from the resource count as these are used as stateless workload injectors that would be replaced by lighter-load Flink sources connected to external sources in production. We include sink operators with a fixed parallelism of one task. Sinks have a low load across all queries and never are a bottleneck.

`Q1` and `Q2`: stateless, single-operator queries. These two queries feature a single stateless operator. As the results for the two were highly similar, we



(a) Q1

(b) Q3

(c) Q5

(d) Q11

(e) Q8

Fig. 5: Elastic scaling performance of Justin (plain lines) compared to DS2 (dotted lines), achieved capacity (purple), and resulting overall CPU cores (red) and memory consumption (green). The horizontal dashed blue line represents the target rate.

choose only to show those for `Q1`. Figure 5a shows that both auto-scalers use two steps to reach a final configuration supporting the 2,250,000 event/s target rate. Both reach a parallelism of 7 tasks for the operator, but Justin disables its managed memory. As a result, the Justin configuration uses 40% less memory, from 2,317 MB for DS2 down to 1,379 MB.

`Q3`: two stateless and one stateful operator. `Q3` performs an unbounded incremental join, but the size of this operator state converges to a relatively small value (∼8 MB), indicating vertical scaling is likely counterproductive. The two other operators are stateless. Figure 5b shows that Justin can free the managed memory of the stateless operators while avoiding unnecessary scale-up for the stateful one, using 10% less memory than DS2. Both auto-scalers converge in the same number of steps. In detail, at the first reconfiguration ($t = 1$, 180s), stateless operators get configured as (1; $\perp$) in Justin and (1; 158) in DS2; after two more scaling steps ($t = 3$) the configuration of the stateful operator becomes (12; 158) in both cases.

`Q5`: stateful sliding-window aggregation query. `Q5` performs an aggregation over a sliding window, requiring complex access patterns to state in order to recompute outputs over a new window frequently. Under the Nexmark workload, the state remains small (i.e., ∼10 MB). As for `Q3`, vertical scaling would have no impact on the performance of this query. Justin chooses only to perform horizontal scaling. Yet, it saves a small amount of memory by removing the managed memory of the sink operator, as shown by Figure 5c. For both auto-scalers, the final configuration for `Q5`'s stateful operator is (24; 158).

We discuss `Q11` before `Q8` to follow the order of presentation in Figure 5.

`Q11`: stateful session window aggregation query. `Q11` results in Figure 5d illustrate the benefit of replacing a scale-out decision with a scale-up.

We observe a first reconfiguration a little before 200 s for both auto-scalers. While DS2 increases the parallelism of the primary operator to 3, growing the memory use accordingly, the CPU and memory of Justin remain equal but for a higher resulting capacity. This is explained by the joint decisions of Justin of (1) stripping the sink operator of its unnecessary managed memory and (2) allocating the same amount of memory to the primary operator due to a scale-up decision (i.e., increasing its memory level from 0 to 1 while keeping a single task).

The resulting capacity is lower with Justin but higher when accounted for per expanded core. Follow-up reconfigurations at ∼380 s and ∼560 s use scale out for both Justin and DS2. However, the better per-task performance of the configuration resulting from the first reconfiguration leads Justin to require 48% lower CPU and a 28% lower memory utilization. The final configuration matching the target rate is (6; 316) for Justin and (12; 158) for DS2. Justin uses the same number of reconfigurations but converges slightly faster than DS2 with this query.

`Q8`: stateful tumbling window join. Finally, Figure 5e presents results for `Q8`. This query is more complex and requires more reconfigurations to achieve the target rate. As for `Q11`, the first reconfiguration of DS2 uses a scale-out while Justin triggers a scale-up of the primary stateful operator. Interestingly, and in

contrast with `q11`, the scale-up of Justin seems to have no real benefit, which may lead to thinking one additional round of reconfiguration will be necessary to compensate for a bad decision. The following reconfigurations show the situation is the opposite. Justin performs three scale-out reconfigurations to reach the target rate, while DS2 needs four. We note that Justin intermediate configurations take longer to stabilize than with DS2, resulting in a shorter convergence time for the latter despite the additional step. In terms of resources, however, the advantage is clearly for Justin, with 48% less CPU cores and 27% less memory than DS2 for the same capacity. The primary operator's final configuration is (12; 316) for Justin and (24; 158) for DS2.

Discussion. Our experiments with Nexmark show that Justin exploits hybrid CPU/memory scaling effectively for all queries but one. It saves memory by stripping unnecessary memory from stateless operators (in primary operators of `Q1`, `Q2`, and `Q3`, and for sink operators of all queries). It significantly reduces the overall resource consumption for a given target rate in complex stateful queries (`Q8` and `Q11`), both in terms of memory as expected and, perhaps more unexpectedly, CPU. This latter result is due to the higher efficiency of tasks not constrained by state access that, as a result, do not need to reach high parallelism for the same capacity as constrained ones. Justin results in fewer or the same number of steps as DS2 and comparable convergence time. Finally, we observe that for a query that does not really benefit from hybrid CPU/memory scaling (`Q5`), Justin does not introduce a penalty over DS2.

6 Related Work

Elastic scaling for stream processing has gathered significant interest over the last decades. Representative works include MEAD [30], hierarchical auto-scaling [31], DS2 [17], and more [8, 15, 34]. Surveys by Röger and Mayer [28] and Cardellini *et al.* [10] provide a comprehensive analysis of this field. To the best of our knowledge, no elastic scaling method combining dynamic memory allocation with horizontal scaling has been proposed by previous work on the topic.

Resource management in distributed stream processing has also attracted significant interest. A survey by Liu *et al.* [19] presents an overview of the topic. Integrating state management with stream processing operators led to advances in programmability and fault tolerance [5, 9, 22]. StreamBed [29] uses interpolation techniques from guided small-scale runs to build a model and predict resource consumption, including CPU and memory, for large-scale Flink instances. Recently, Wang *et al.* [37] proposed CAPSys, a new algorithm for assigning task slots to tasks in Flink that takes into account colocation effects between tasks and competition for resources such as memory and I/O. CAPSys reduces the unpredictability of round-robin or random task-to-TM assignments. Adapting its approach to the placement problem in Justin, where tasks have different memory granularity, is undoubtedly an interesting perspective of our work.

Understanding the performance of LSM-based storage [21, 25], including its relation to available memory, is a complex problem. Gadget [7] is a benchmark of backend storage operations in a DSP, targeting Flink and RocksDB. Tutorials

by Sarkar *et al.* [32, 33] are a good source of references on optimizations for LSM-based stores and optimization of read operations.

Memory management and allocation is an important topic in non-stream (i.e., batch) processing systems. MespaConfig [39] optimizes the memory configuration of *multiple* Spark batch processing jobs co-located on a cluster. Iorgulescu *et al.* [16] define the memory elasticity principle and show how careful memory allocation below the working set size of batch processing, in-memory applications can yield close performance to larger allocations and implement resulting scheduling policies in Apache Yarn. The specificities of write-optimized, LSM-based storage in RocksDB prevent from directly applying these techniques.

Finally, we note that the idea of combining horizontal and vertical scaling in a hybrid auto-scaler has been explored in other contexts, such as cloud workloads, e.g., with HoloScale [23], or for serving model inference [27].

7 Conclusion

We presented Justin, a hybrid CPU and memory auto-scaler for Apache Flink. In contrast with DS2, Flink current auto-scaler integrated with its Kubernetes operator, Justin arbitrates between horizontal scaling and vertical scaling based on metrics from the RocksDB storage layer. Results on the classical Nexmark benchmark show that Justin can achieve similar capacity as DS2 when reconfiguring towards a target rate while using fewer resources.

This work opens several interesting perspectives. First, and similarly to DS2, Justin makes implicit assumptions about the absence of skew, i.e., unbalanced key popularities leading to an imbalanced load between the different tasks of an operator. A solution to this problem is explicitly rebalancing keys between tasks instead of using only a hash function [13,24]. This does not address the fact that, in some cases, popular keys may be associated with a more significant state. In this case, heterogeneous memory allocation between tasks of the same operator could be an option, although it would require a significantly more complex auto-scaler. Second, a complementary path to Justin reactive auto-scaling approach would be to predict operators' response to memory availability, either by allowing programmers to provide hints or by running them in isolation and modeling their performance [6]. Finally, Justin and DS2 consider the auto-scaling and resource allocation of queries in isolation. An auto-scaler that considers the co-placement of query tasks, e.g., running memory-intensive tasks of some queries with stateless tasks of another query, may lead to better consolidation and platform resource use overall.

Acknowledgments. This research was funded by the Walloon region (Belgium) through the Win2Wal project "GEPICIAD" and by a gift from Eura Nova. 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. Apache flink. https://flink.apache.org/ (Feb 2025)
2. Apache storm. https://storm.apache.org/ (Feb 2025)
3. Flink kubernetes operator. https://github.com/apache/flink-kubernetes-operator (Feb 2025)
4. Rocksdb. https://rocksdb.org/ (Feb 2025)
5. Affetti, L., Margara, A., Cugola, G.: Flowdb: Integrating stream processing and consistent state management. In: DEBS 2017. pp. 134–145
6. Agnihotri, P., Koldehofe, B., Stiegele, P., Heinrich, R., Binnig, C., Luthra, M.: Zerotune: Learned zero-shot cost models for parallelism tuning in stream processing. In: ICDE 2024
7. Asyabi, E., Wang, Y., Liagouris, J., Kalavri, V., Bestavros, A.: A new benchmark harness for systematic and robust evaluation of streaming state stores. In: EuroSys 2022
8. Barazzutti, R., Heinze, T., Martin, A., Onica, E., Felber, P., Fetzer, C., Jerzak, Z., Pasin, M., Riviere, E.: Elastic scaling of a high-throughput content-based publish/subscribe engine. In: ICDCS 2014. pp. 567–576
9. Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., Tzoumas, K.: State management in apache flink®: consistent stateful distributed stream processing. Proceedings of the VLDB Endowment **10**(12), 1718–1729 (2017)
10. Cardellini, V., Lo Presti, F., Nardelli, M., Russo, G.: Runtime adaptation of data stream processing systems: The state of the art. ACM Computing Surveys **54**(11s), 1–36 (2022)
11. CNCF: Prometheus (2024), https://prometheus.io/
12. Dong, S., Kryczka, A., Jin, Y., Stumm, M.: RocksDB: Evolution of development priorities in a key-value store serving large-scale applications. ACM Trans. on Storage **17**(4) (2021)
13. Fang, J., Zhang, R., Fu, T.Z., Zhang, Z., Zhou, A., Zhu, J.: Parallel stream processing against workload skewness and variance. In: ACM HPDC. 2017 (2017)
14. Fragkoulis, M., Carbone, P., Kalavri, V., Katsifodimos, A.: A survey on the evolution of stream processing systems. The VLDB Journal **33**(2), 507–541 (2024)
15. Gedik, B., Schneider, S., Hirzel, M., Wu, K.L.: Elastic scaling for data stream processing. TPDS 2013 **25**(6), 1447–1463
16. Iorgulescu, C., Dinu, F., Raza, A., Hassan, W.U., Zwaenepoel, W.: Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling. In: USENIX ATC 2017. pp. 97–109
17. Kalavri, V., Liagouris, J., Hoffmann, M., Dimitrova, D., Forshaw, M., Roscoe, T.: Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In: OSDI 2018
18. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: A distributed messaging system for log processing. In: Proc. of the NetDB. vol. 11 (2011)
19. Liu, X., Buyya, R.: Resource management and scheduling in distributed stream processing systems: a taxonomy, review, and future directions. ACM Computing Surveys (CSUR) **53**(3), 1–41 (2020)
20. Lodi, A., Martello, S., Vigo, D.: Recent advances on two-dimensional bin packing problems. Discrete Applied Mathematics **123**(1-3), 379–396 (2002)
21. Luo, C., Carey, M.J.: Lsm-based storage techniques: a survey. The VLDB Journal **29**(1), 393–418 (2020)

22. Madsen, K.G.S., Zhou, Y.: Dynamic resource management in a massively parallel stream processing engine. In: CIKM 2025. pp. 13–22
23. Millnert, V., Eker, J.: Holoscale: Horizontal and vertical scaling of cloud resources. In: UCC 2020. pp. 196–205
24. Nasir, M.A.U., Morales, G.D.F., Kourtellis, N., Serafini, M.: When two choices are not enough: Balancing at scale in distributed stream processing. In: ICDE 2016. pp. 589–600
25. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (lsm-tree). Acta Informatica **33**, 351–385 (1996)
26. Rabl, T., Traub, J., Katsifodimos, A., Markl, V.: Apache Flink in current research. it-Inf. Tech. **58**(4) (2016)
27. Razavi, K., Salmani, M., Mühlhäuser, M., Koldehofe, B., Wang, L.: A tale of two scales: Reconciling horizontal and vertical scaling for inference serving systems. arXiv preprint arXiv:2407.14843 (2024)
28. Röger, H., Mayer, R.: A comprehensive survey on parallelization and elasticity in stream processing. ACM Computing Surveys **52**(2), 1–37 (2019)
29. Rosinosky, G., Schmitz, D., Rivière, E.: Streambed: capacity planning for stream processing. In: DEBS 2024. pp. 90–102
30. Russo, G.R., Cardellini, V., Casale, G., Presti, F.L.: MEAD: Model-based vertical auto-scaling for data stream processing. In: CCGrid 2021. pp. 314–323
31. Russo Russo, G., Cardellini, V., Lo Presti, F.: Hierarchical auto-scaling policies for data stream processing on heterogeneous resources. ACM Transactions on Autonomous and Adaptive Systems (2023)
32. Sarkar, S., Athanassoulis, M.: Dissecting, designing, and optimizing LSM-based data stores. In: SIGMOD 2022. pp. 2489–2497
33. Sarkar, S., Dayan, N., Athanassoulis, M.: The LSM design space and its read optimizations. In: ICDE 2023. pp. 3578–3584. IEEE
34. Schneider, S., Andrade, H., Gedik, B., Biem, A., Wu, K.L.: Elastic scaling of data parallel operators in stream processing. In: IPDPS 2009. pp. 1–12
35. Tucker, P., Tufte, K., Papadimos, V., Maier, D.: Nexmark–a benchmark for queries over data streams (draft). Technical report (2008)
36. Verwiebe, J., Grulich, P.M., Traub, J., Markl, V.: Survey of window types for aggregation in stream processing systems. The VLDB Journal pp. 1–27 (2023)
37. Wang, Y., Huang, L., Wang, Z., Kalavri, V., Matta, I.: CAPSys: Contention-aware task placement for data stream processing. In: EuroSys 2025
38. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: Fault-tolerant streaming computation at scale. In: SOSP 2013 (2013)
39. Zong, Z., Wen, L., Hu, X., Han, R., Qian, C., Lin, L.: Mespaconfig: Memory-sparing configuration auto-tuning for co-located in-memory cluster computing jobs. IEEE Transactions on Services Computing **15**(5), 2883–2896 (2021)