StreamBed: Capacity Planning for Stream Processing

Guillaume Rosinosky, Donatien Schmitz, Etienne Rivière

To cite this version:

Guillaume Rosinosky, Donatien Schmitz, Etienne Rivière. StreamBed: Capacity Planning for Stream Processing. DEBS 2024 - 18th ACM International Conference on Distributed and Event-based Systems, Jun 2024, Lyon, France. pp.90-102, 10.1145/3629104.3666034 . hal-04708354

HAL Id: hal-04708354
https://hal.science/hal-04708354v1

Submitted on 24 Sep 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

StreamBed: Capacity Planning for Stream Processing

Guillaume Rosinosky*
guillaume.rosinosky@inria.fr
IMT Atlantique, Nantes Université,
École Centrale Nantes, CNRS,
Inria, LS2N - UMR 6004
F-44000 Nantes, France

Donatien Schmitz
donatien.schmitz@uclouvain.be
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium

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

ABSTRACT

We present StreamBed, a capacity planning system for stream processing. StreamBed predicts, ahead of any production deployment, the resources that a query will require to process an incoming data rate sustainably, and the appropriate configuration of these resources. For this purpose, StreamBed builds a capacity planning model by piloting a series of runs of the target query in a small-scale, controlled testbed. We implement StreamBed for Apache Flink. Our evaluation with large-scale queries of the Nexmark benchmark demonstrates that StreamBed can accurately predict capacity requirements for jobs spanning more than 1,000 cores using a model built with a 48-core testbed.

CCS CONCEPTS

• **Computer systems organization → Data flow architectures**; • **General and reference → Performance**; **Estimation**; • **Information systems → Data streams**.

KEYWORDS

capacity planning, stream processing, Flink

ACM Reference Format:
Guillaume Rosinosky, Donatien Schmitz, and Etienne Rivière. 2024. StreamBed: Capacity Planning for Stream Processing. In *The 18th ACM International Conference on Distributed and Event-based Systems (DEBS'24), June 24–28, 2024, Villeurbanne, France.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3629104.3666034

*Work performed while at UCLouvain.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
DEBS'24, June 24–28, 2024, Villeurbanne, France
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0443-7/24/06
https://doi.org/10.1145/3629104.3666034

1 INTRODUCTION

Distributed Stream Processing (DSP) is a key component of analytics and decision-support systems. DSP engines such as Twitter Heron or Spark Streaming support the efficient execution of stream processing queries. Apache Flink [9] is one of the most popular open-source engines. Like other engines, Flink leverages massively parallel processing. A user query is expressed as a graph of processing *operators*, each supported by several *tasks*. Elastic scaling policies [6, 8, 10, 36] can adapt at runtime the parallelism of each operator to support increasing event rates.

Motivation. It is hard to know, before the full-scale deployment of a new query, the resources budget that it will *eventually* require after elastic scaling, or how these resources will have to be configured to *sustainably* ingest a target workload (i.e., achieve a given *capacity*). The scaling behavior of queries is often sub-linear, disallowing simple proportionality-based predictions. They often exhibit non-trivial resource usage and performance patterns, such as load spikes due to stragglers [15] or imbalance between tasks' loads due to skew [19, 24, 48].

Uncertainty in the resource requirements of queries often leads practitioners to exercise caution and over-provision their infrastructure. An infrastructure that cannot scale out to the needs of a query could, indeed, result in cascading failures and query termination [32]. When DSP jobs are deployed in a public cloud, the problem becomes that of mastering costs, as uncontrolled scale-out may lead to paying for a large amount of on-demand resources.

Contributions. We present StreamBed, a system for determining, ahead of any production deployment, the resources budget needed for supporting sustainably a target DSP query. In addition, StreamBed can predict the appropriate *configuration* of these resources, without requiring a long and costly adaptation process using elastic scaling. StreamBed is the first system to enable *capacity planning* for stream processing, an approach that has so far mostly targeted relational databases [23, 47].

In contrast with previous approaches, based on costly benchmarking at production scale [22, 25] or pre-execution modeling of queries [1, 2, 21, 27, 44], StreamBed analyzes

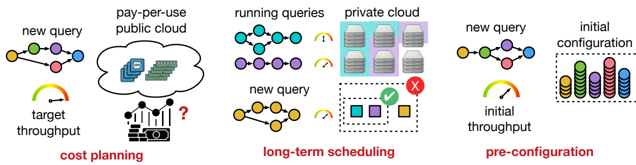

Figure 1: StreamBed enables capacity planning for stream processing. On the left, it helps to determine the cost of running a large query at a certain throughput and for a certain duration in the cloud. In the middle, it allows for determining if a new query can be safely deployed alongside others in a private cloud with limited resources. On the right, it returns an initial configuration for a query, avoiding the costs and service interruptions of many scale-out reconfigurations.

the behavior of a submitted query running in a controlled, small-scale testbed, taking into account the impact of the workload in actual runs of the query. The testbed used for capacity planning is much smaller than the target production system, with typically one to two orders of magnitude less CPU cores and memory.

Using the small testbed, StreamBed builds a *capacity planning model* using a Bayesian Optimization method that explores different bounded resource budgets and decides on the best fitting model using regression methods. For each budget, StreamBed identifies its optimal configuration and its maximum sustainable throughput [26], i.e., the volume of data that can be ingested without provoking instabilities and, eventually, crashes. These measurements allow training and choosing the most appropriate scaling and configuration model from different candidate regression models. The resulting capacity planning model is used to predict, for large target rates, the amount of necessary resources and how these resources should be configured, enabling the use cases illustrated by Figure 1.

We evaluate StreamBed in an 85-node cluster, featuring a total of 1,344 cores and 7.5 TB of RAM. Out of these, StreamBed uses 48 cores and 192 GB of RAM for controlled runs; the rest supports production deployments and data storage and replay. We use 5 representative queries from the Nexmark benchmarking suite [45]. Our results show that StreamBed can accurately predict the volume of necessary resources with a low cores.hours budget, for queries reaching more than 1,000 cores or ingesting up to 190 million events per second. Our evaluation of StreamBed predictions at a production scale shows that the system avoids over- or under-provisioning and derives configuration that can sustainably inject the targeted loads, even for complex, stateful queries with non-linear scaling profiles.

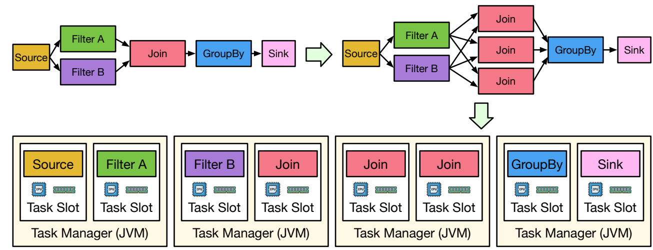

Figure 2: A Flink job with 4 operators (top left). The Join operator is scaled-out to a parallelism of three (top right). The resulting eight tasks are distributed to available task slots on four task managers (bottom).

Outline. We present the background on Flink and elastic scaling in Section 2. We detail our operational assumptions and give an overview of StreamBed in Section 3, before detailing its components, namely the Capacity Estimator (Section 4), the Configuration Optimizer (Section 5), and the Resource Explorer (Section 6). We report implementation details in Section 7 and our evaluation of StreamBed in Section 8. We follow with related work in Section 9 and a conclusion in Section 10.

2 BACKGROUND

Apache Flink. A DSP engine supports queries over continuous data, or *jobs*, implemented as an oriented graph of interconnected *operators* as illustrated by the top-most example in Figure 2. Each operator implements a basic operation on the data flow(s) it receives as its input, and outputs a stream of events to be consumed by downstream operators. Flink supports a variety of operators, selected from a standard library, or compiled from SQL [35]. While some operators are stateless (e.g., map or filter), others need to persist state across events (e.g., join or group by) such as data structures over a window of time or events. Sources and Sinks are specific operators injecting and outputting data from/to the external world.

Deployment of a Flink job is under the responsibility of a centralized Job Manager node orchestrating several Task Managers (TM), represented at the bottom of Figure 2. Each TM runs as a process (JVM) and offers some Task Slots (TS). Flink, similarly to other DSP engines, assumes homogeneous task slots, each assigned to one CPU core but with a configurable amount of memory, forming the task *profile*. In Figure 2, the parallelism of operators is 1, except for the Join using a parallelism of 3. The resulting 8 tasks are dispatched by the Job Manager to the available 4 TMs. The flow of events to a parallelized operator, e.g., from Filter A to the Join, is partitioned based on the key associated with events [8]. Each task maintains a buffer of incoming events. *Back-pressure*

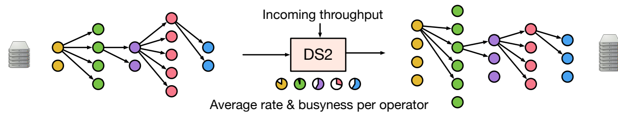

Figure 3: Illustration of a reconfiguration using DS2 [24]. Only a subset of the edges is shown in the interest of clarity.

mechanisms regulate the production of events: a task can instruct its upstream operator(s) to slow down production when its buffer goes over a certain fill rate.

Elastic Scaling. Elastic scaling allows coping with varying input rates for a DSP job, as imposed by its data sources. A configuration maps each operator to a level of parallelism, i.e., how many tasks support it. A reconfiguration adapts this parallelism and is generally triggered by indicators such as the amount of back-pressure or the average CPU consumption of TMs [8]. Röger and Mayer [36] and Carellini *et al.* [10] present comprehensive surveys of elastic scaling approaches for DSP. In what follows, we focus on the DS2 algorithm [24], used by Linkedin [41] and integrated with Flink [30] as the elastic scaler for its Kubernetes-based resource manager [18].

The key idea underlying DS2, illustrated by Figure 3, is to determine the level of parallelism for *all* operators of a job undergoing a scale-out rather than scaling out each operator independently. DS2 collects rates (processed events/s) for all operators' tasks and a measure of their *busyness*, i.e., the ratio of time spent actually processing these events. DS2 computes in one pass a new configuration taking into account the cascading loads between operators.

While DS2 improves over previous approaches based on a *cascading* reconfiguration of operators, it cannot derive the parallelism of a large-scale installment of a query from busyness rates measured over a small-scale run of that same query. *Skew* [19, 24, 48], i.e., the imbalance between the popularities of keys used to dispatch events between one operator and the next, leads, for many queries, to highly sub-linear scaling behaviors. DS2 also does not take into account the memory requirements of operators, assuming that tasks do not suffer from performance degradation with large states. In contrast, our objective is to take into account the impact of imbalance and memory pressure when performing capacity planning, while avoiding costly production-scale deployments.

3 OVERVIEW

StreamBed targets long-running queries that need to scale out to hundreds of cores for their execution. We consider a system formed of two clusters: a large one dedicated to production deployments and a much smaller one dedicated to capacity planning. The two clusters use the same hardware to make observations on one transposable onto the other.

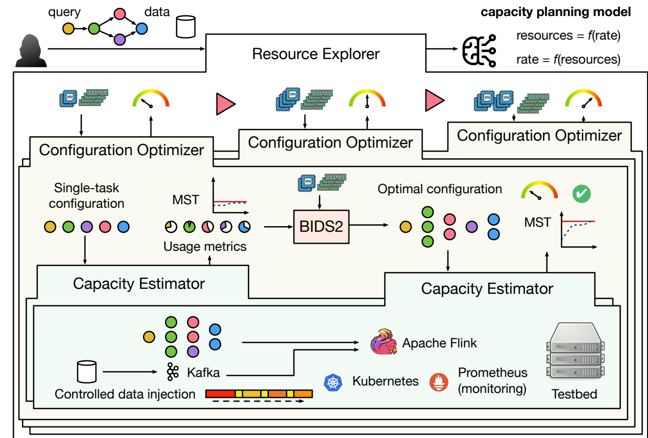

Figure 4: StreamBed workflow. For a query, the resource explorer builds a capacity planning model from capacity estimations with small resource budgets. For each budget, the configuration optimizer determines the best resource allocation and its MST via controlled benchmarking with the capacity estimator.

The goal of StreamBed is to build a capacity planning model for a *specific* DSP query provided by the user. We assume that a dataset, representative of the input of the job, is also provided by that user, e.g., historical data for the corresponding source. We do not require, however, that the query ran previously on this dataset or any other dataset. The query does not have to be modified by the user, but StreamBed needs to be informed of fields in events' schemas used to represent event time, if any. This information is needed to replace recorded, historical times with emulated times in controlled runs of the query.

We assume that the submitted DSP job uses an arbitrary combination of stateless and *windowed* stateful operators including joins, as is common for continuous queries [46]. We do not make assumptions on the length of these windows or the sizes of operators' working sets.

StreamBed builds a model allowing querying the necessary resources budget for different input rates (i.e., the number of task slots and their resource profile) as well as their appropriate configuration (parallelism for each operator). The returned configurations must be able to sustain the requested throughput without over- or under-provisioning.

StreamBed handles skew (and the resulting imbalance in the loads of tasks of a given operator) by considering the measured capacity as the maximal sustainable throughput (MST [26]) of the query under a specific resources budget, and the evolution of this MST with varying budgets.

StreamBed components. The workflow and the three nested components of StreamBed are illustrated in Figure 4.

At the top-most level, the **Resource Explorer (RE)** drives the exploration of different *resource budgets*, i.e., numbers of task slots with a given resource profile. For each budget, the RE collects the Maximal Sustainable Throughput (MST) and the associated configuration, which it uses to build the capacity planning model. The choice of profiles is driven by a Bayesian Optimization algorithm and regression techniques. In Figure 4, the RE evaluates three such resource budgets.

The determination of the best configuration and of achievable MST for a given resource budget is under the responsibility of the **Configuration Optimizer (CO)**. The CO leverages BIDS2 (Bounded-Inverse DS2), an evolution of the DS2 algorithm able to determine, in one pass and for a bounded resource budget, the configuration with the highest capacity. In Figure 4, the CO determines the MST for the smallest of the budgets submitted by the RE for evaluation.

The determination of the MST of a configuration is performed by the **Capacity Estimator (CE)** using controlled runs of the job in the small testbed. Determining the MST experimentally is a complex task due to variations of resource usage over time (e.g., ramp-up phases) and instabilities when operating above the MST. A controlled data injection strategy determines the MST of a configuration by stress-testing it and adjusting input rates using a dichotomous strategy.

In the following sections, we detail these components in a bottom-up fashion, starting with the Capacity Estimator (§4), following with the Configuration Optimizer (§5), and finishing with the Resource Explorer (§6).

4 CAPACITY ESTIMATOR

The role of the Capacity Estimator (CE) is to determine *experimentally* the Maximal Sustainable Throughput, or MST, of a job in a specific configuration, upon request of the Configuration Optimizer. The MST [26] is the highest possible average actual rate that can be processed by the configuration (i.e., no piling up of unprocessed records). The goal of the CE is to estimate the MST in a small amount of time, while guaranteeing its accuracy, in particular concerning the impact of stateful operators, skew, and stragglers. The CE must also take into account that injected rates higher than the MST often lead to chaotic behaviors such as unsteady actual processing rates.

Load injection. The CE uses controlled load injection, attaching rate-limited sources to the job. These sources can connect with a distributed storage in a data lake, allowing the use of data at rest as a representative input. Alternatively, StreamBed can employ as a source an online pseudo-random event generator provided by the user. In both cases, sources attempt to inject events at up to a fixed target rate but abide by the back-pressure received from downstream operators.

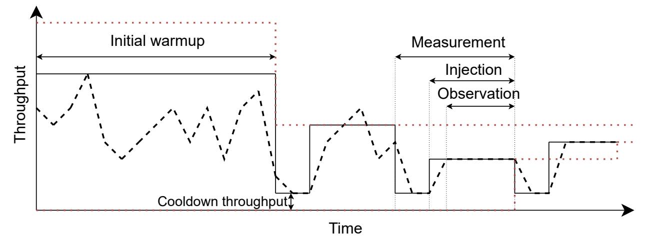

Figure 5: The Capacity Estimator uses an evaluation strategy based on fixed-rate load injection, with a dichotomous search following an initial warmup phase. Black plain lines show the target input rate, while black dashed lines show the actual, measured processing rate. Dashed, thin red lines present the maximal and minimal rates $\max_r$ and $\min_r$.

Warmup phase. A new job will typically accept a much higher throughput than its steady-state MST. Back-pressure mechanisms only kick in after a threshold capacity is reached at a specific edge, leading to a temporarily higher initial "absorption capacity". Moreover, stateful operators start with an empty state that gradually grows until it reaches a steady state (e.g., a full sliding window of events). This leads us to use a *warmup* phase before any measurement, to observe performance metrics on a stabilized job.

Evaluation strategy. A policy decides on variations of the input rate, and on when to measure the actual processing rate. A naive approach to determining the MST would be to start from a low target rate and gradually increase it until the source starts observing a certain back-pressure level. This approach is inappropriate for two reasons. First, it mixes the necessary warmup phase with the actual measurement phase, risking estimating the MST of a job that has not reached its steady state. Second, it does not take into account the inertia of the job under test: A change in input rate leads to cascading effects on buffer occupancies and back-pressure levels. These effects often take several seconds, and sometimes several minutes, to stabilize. Using an ever-increasing rate bears risks that this stabilization never occurs.

We propose a strategy, illustrated by Figure 5, that uses fixed target throughputs, includes stabilization phases, but avoids measurements during these phases. It starts with a sufficiently long warmup phase where we inject throughput at a very high rate, building up sufficient state at stateful operators and filling buffers. Reaching this steady state can require several minutes of warmup, depending on available memory and the size of the working state of operators.

The warmup phase is followed by the evaluation itself, using a dichotomous approach similar to a binary search. We consider a *maximal* and a *minimal* rate, $\max_r$ and $\min_r$, initially set to ∞ and 0. The search progresses in phases. In each phase, a target rate r is tested. If the actual processing rate being equal or very close (i.e., ≥99%[1]) to the incoming rate then $\min_r = r$. Otherwise, $\max_r = r$. The next target is the average between these two values, i.e., $r = (\max_r + \min_r)/2$. The search stops when reaching a configurable sensibility (e.g., the next value of r is within 1% of the previous) or after a maximal number of iterations.

A measurement for a target rate r starts with a *cooldown* phase, using a low (but non-zero) rate allowing operators to process events in their input buffers and recover from saturation in the previous measurement. Then, the target rate r is applied for an *injection* phase. We measure the actual processing rate after a short duration to enable a local ramp-up for the new injection. During the *observation* phase, we measure the achieved rate from the point of view of the source and collect busyness metrics and actual input rates.

The metrics for the final measurement are returned along with the MST to the Configuration Optimizer.

5 CONFIGURATION OPTIMIZER

The second component of StreamBed is the Configuration Optimizer, or CO for short. It receives from the Resource Explorer a query and a resource budget, i.e., a number of task slots with a given profile (amount of RAM). Its goal is to return a configuration fitting this budget with the best possible MST.

The CO requires usage metrics for a run of a *minimal* configuration, i.e., where each operator has a single task. These include the *actual input rate* and the level of *busyness* of each single-task operator, i.e., the percentage of time it spends actually processing events. The CO maintains a cache of these single-task configuration metrics for different resource profiles, reusing previous measurements when possible. When no data exists, the CO requests the CE to evaluate this single-task configuration with the target profile.[2]

Based on observed metrics with the single-task configuration, we formulate the problem of determining the *optimal* capacity for a target resource budget, i.e., the highest possible source input rate. The corresponding *best possible* configuration is the one where the busyness metric of all tasks, across all operators, is the highest possible while avoiding the saturation of any task. We solve this optimization problem with an algorithm we name BIDS2, for Bounded-Inverse DS2. In contrast with the original DS2 [24], BIDS2 does not determine the amount of necessary resources for a given input rate, but determines the optimal configuration for a *bounded* resource budget. The configuration resulting from the optimization is given as an input to a call to the CE to determine its MST and experienced busyness levels.

[1]Due to the use of a rate limiter, the actual rate can never exceed 100%.

[2]An exception is when the Resource Explorer explicitly requests the revaluation of a single-task configuration as part of its exploration.

Symbol	Description
$\mathcal{P}$	Task slots budget
o^i	Observed true processing rate of a task of operator i
r^i	Observed ratio of operator i's input rate over source's rate
π^i	Optimal parallelism of the operator i
λ_{src}	Optimal input rate of the source operator
λ^i	Optimal input rate of one operator instance i

Table 1: Notations used by the BIDS2 algorithm.

BIDS2 algorithm. Table 1 lists the variables used in the optimization. Our goal is to maximize the source rate:

$$\max \lambda_{\text{src}} \tag{1}$$

The level of parallelism π^i is the decision variable for each operator i (i varying between 1 and the number of operators of the job excluding the sources). As Kalavri *et al.* [24], we assume in this optimization problem a linear relation between the processing rate and the level of busyness for a specific budget and a fixed profile: we compute the true processing rate o^i of a task of operator i by dividing its actual processing rate by the average busyness level of its tasks. The non-linearity of jobs scaling due to skew will be reflected in the measured actual MST of the decided configuration, when measured by the CE, and integrated in the model built by the CO.

We compute the optimum processing rate λ^i of an operator i based on its parallelism π^i, as given by Equation 2.

$$\forall i: \ \lambda^i = \pi^i o^i \tag{2}$$

Then, we compute using Equation 3 the rate of each operator expressed as a function of the decision variable input rate λ_{src}: The ratio r_i, precomputed using the actual rates returned by the CE is used as a multiplier to estimate this proportion.

$$\forall i: \lambda_{\text{src}} \cdot r^i \leq \lambda^i \tag{3}$$

Finally, we set the constraint in Equation 4 that the sum of the parallelism for the different operators should be exactly the number of task slots $\mathcal{P}$:

$$\sum_i \pi^i = \mathcal{P}. \tag{4}$$

The configuration resulting from the optimization is given as an input to a second call to the CE, allowing it to determine its MST and to return it together with observed metrics.

6 RESOURCE EXPLORER

The goal of the Resource Explorer (RE) is to build the capacity planning model for a target unknown query. The RE drives the collection of capacity measurements for different resource budgets (number of tasks) and resource profiles (RAM per task). Based on configurations and rates received from the CO, the RE can further determine an optimal configuration for a (predicted) resource budget.

The RE models the relation between resources and capacity as a *surrogate model*, i.e., an approximation of a complex system replacing expensive simulation models. The RE identifies the most representative function that describes the relation between a number of TS Π (all used, i.e., $\sum \pi^i = \Pi$) using resource profiles with M MB of memory per TS and the resulting capacity λ_{src} as detailed in equation 5.

$$f(M, \Pi) = \lambda_{src} \tag{5}$$

While some jobs exhibit linear scaling, others subject to skew and/or containing stateful operators, aggregates, and especially joins often see their improvement in performance decrease with additional resources. We must model these sub-linear scaling profiles appropriately, i.e., using monotonically increasing functions with a derivative that decreases over time. Based on our experience and results from other researchers [19], we select in StreamBed two simple functions, logarithm and square root, matching this requirement and performing well in practice. Note that StreamBed can easily accommodate alternative functions. The three linear regressions we propose are given in equations 6, 7, and 8, with a and b the slopes of the functions applied to the quantity of memory and task slots, and c the intercept. The goal of the RE is to determine which of the three models fits best the observations, find coefficients a, b, and c, and use the chosen model.

$$aM + b\Pi + c = \lambda_{src} \tag{6}$$

$$a \log M + b \log \Pi + c = \lambda_{src} \tag{7}$$

$$a\sqrt{M} + b\sqrt{\Pi} + c = \lambda_{src} \tag{8}$$

Overview. The RE builds concurrently the three candidate models based on a succession of measurements by the CO. The model is then used to extrapolate, for high values of λ_{src}, values of Π that are outside of that search space.

The construction of surrogate model candidates must balance their extrapolation capacity and the cost of collecting measurements. The collection of the MST for a given resource budget and resource profile requires running the CO and up to two instances of the CE. Due to the need for the query to stabilize to its steady state, these steps easily represent dozens of minutes of data collection in the test cluster.

The model construction happens in three phases. First, in a *candidate search* phase, we use a black-box optimization technique to determine new resource budgets and profile candidates optimizing the distance between the prediction of the current best model and the actual results of the runs. Second, in a *model selection* phase, we determine which of the three models has the best potential for extrapolation. Finally, as the best model is selected, it can be used to directly predict the MST for a given configuration.

Model performance. We measure the accuracy of the candidate models with the root mean squared error (RMSE), a quantitative measure of how well the model predicts the resource requirements by comparing predicted values with the results obtained by the CO. Lower RMSE values indicate a better-performing model.

The predictive capabilities of the models are determined using Leave-One-Out Cross-Validation (LOOCV): for each observation in the data set, we evaluate if the model trained on all other observations provides a good (low) RMSE value. LOOCV helps to minimize overfitting and provides a reliable estimate of the model's predictive capabilities, especially in cases where there are few observations [31].

Candidate search. The goal of the candidate search is to find candidate couples (M, Π) and their corresponding MST that reduce the training error for the current best model.

The search space is 2-dimensional. The number of task slots Π ranges from the number of operators to the number of cores available in the test cluster. M is discretized using a configurable level of granularity. If we consider a query with 9 operators on a cluster with 48 cores (39 possible values for Π) and memory ranging from 512 MB to 4 GB in increments of 512 MB (8 possible values) then the search space admits 312 different combinations.

An intensive evaluation of the search space (i.e., a grid search) is simply too costly. We use instead Bayesian Optimization (BO) [3], an optimization method targeting black-box functions. BO is based on a *probabilistic model* (a Gaussian Process) approximating the true function coupled to an *acquisition function* guiding the search for the optimal point for a *cost function*. The acquisition function balances exploration (searching in uncertain locations of the search space) and exploitation (searching in regions with low predicted error) of candidate points, collected in set D. Typical acquisition functions include Probability of Improvement, Expected Improvement, and Lower Confidence Bound. The RE uses Expected Improvement as it balances the search for low mean (exploitation) and high variance (exploration) candidates.

The candidate search must identify configurations that are likely to yield better RMSE values for the currently better candidate model. The cost function in Equation 9 identifies the lowest LOOCV score amongst candidate models, reducing the training error.

$$\text{BestModel}(D) = \underset{\text{model} \in \{\text{linear,log,sqrt}\}}{\arg\min} \text{LOOCV}(D, \text{model}) \quad (9)$$

The set D is bootstrapped by collecting measurements from the 4 "corners" of the search space, i.e., 4 combinations of lowest and highest values of Π and M, enabling to compute an initial LOOCV score for the three candidate models. The general behavior of the training process is that, following an initial chaotic phase, the RMSE decreases until new individuals start to worsen the score. Our stop criterion takes this behavior into account. We proceed to a minimum of 3 measurements in addition to the 4 corners. Then, we stop when the RMSE increases by more than 10% between two measurements, or after 20 measurements (by default).

Following the bootstrap phase, BO executes the candidate search phase by calling the acquisition function, a call to the CO followed by error computation using Equation 9 as the cost function, and then the adaptation of the internal probabilistic models based on the received candidate couple (M, Π). Note that, to account for the unavoidable measurement variations that happen when collecting MST from the CO and the CE, the RE may decide to re-evaluate a candidate couple that has been previously evaluated to reduce uncertainty. Our experience is that variations are common between two runs with the same budget and profile in particular for complex queries involving joins and/or windowed operations.

Model selection. Following the BO, we select the most appropriate model from the three candidates based on their predictive capability. To evaluate it we use a 50/50 split on D. Since we want to compute the extrapolation capability, we use the first half of D with observations with the lowest values of Π as the training set to predict the other half as the test set, as commonly reported in the literature [38].

We apply each candidate model and evaluate how well it predicts observations in the test set. The model with the lowest average RMSE for points in the test set is selected (obviously, we use the model trained on the full D and not only on the training set used for model selection).

Model usage. The model considers the resource budgets and their profiles as independent variables. It can be used directly to derive a capacity. Returning the necessary resource budget for a target input rate requires solving the model inversely. We adopt an iterative strategy where we consider one or several memory profiles M and we incrementally increase Π until the predicted capacity matches or exceeds the request.

StreamBed tends to estimate resource budgets for which the requested rate is *very close* to the maximum capacity. To guarantee that the requested rate will be sustainable, the RE applies a slight over-provisioning factor by interrogating the model with 110% (by default) of the requested rate.

The model returns the number of task slots but not their configuration. This configuration can be computed using a final pass of BIDS2 using the metrics collected for the largest number of cores in D.

7 IMPLEMENTATION

StreamBed uses a cloud-native stack with Kubernetes supporting Apache Kafka for data ingestion, storage, and replay and Flink for processing. We use the Strimzi Kafka Kubernetes operator [43] to use Kafka as a source of data for Flink, and Spotify's Flink Kubernetes operator [42] to be able to deploy Flink easily with different memory and CPU settings (Kubernetes operators support specific software in Kubernetes and are not Flink operators).

Capacity Estimator. The CE interacts with a running, unmodified instance of Flink v.1.14.1 [17], using Apache Zeppelin [4] notebooks. Runtime measurements (e.g., about processing rates and busyness) are collected using Prometheus [11] with a 5-second aggregation period.

We deploy a Flink instance in the test cluster without auto-scaling features, using only dedicated cores for every TS without simultaneous multi-threading. This instance is deployed by the CE on demand with a fixed, homogeneous resource profile for the TS. It is re-deployed only if the CO requests a different profile, which takes less than a minute.

The CE must be able to control precisely the rate at which the source of a query under test in the small cluster receives data. We developed a novel *rate-limited* Kafka source connector extending the regular Apache Flink Kafka source [16]. This rate-limited source obtains data from a first Kafka topic as well as rate control information from a secondary topic.

Configuration Optimizer and Resource Explorer. The RE and CO are both developed in Python. The CO uses the Python PuLP library [13] coupled to the CBC (Coin-or Branch and Cut) solver [12] both developed by the COIN-OR foundation. Metrics are retrieved from CE runs in Prometheus using the prometheus-api-client library [40]. The RE uses the scikit-optimize [20] library for Bayesian Optimization and the scikit-learn [33] library for the regression.

8 EVALUATION

We wish to answer the following research questions:

- (RQ1) Is the evaluation of the MST by the CO and CE accurate, i.e., can the RE rely on these measurements for its model training?

- (RQ2) Is the CO effective at deriving the "best possible" configuration for a given resource budget?
- (RQ3) Is StreamBed able to predict, based on small-scale runs, an accurate budget of resources, their profile, and their configuration, such that a production run based on this prediction is neither over- nor under-provisioned?
- (RQ4) How much resources and time are necessary for StreamBed to build capacity planning models?

We first present our workloads and our experimental setup. Then, we answer questions (RQ1) and (RQ2) using micro-benchmarks at the CE and CO levels. Finally, we answer questions (RQ3) and (RQ4) using macro-benchmarks involving the RE and the full StreamBed stack. StreamBed is, to the best of our knowledge, the only capacity planning system for stream processing, disallowing comparison to a system with the same goal. We validate the complete system by comparing its predictions to actual production-scale deployments.

Query	Description	Operators	Stateful	Min rate ($\times 10^3$ evt/s)
q1	**Currency conversion**: converts bid values from dollars to euros	1	-	1600
q2	**Selection**: filter bids with specific auction identifier	1	-	3600
q5	**Hot items**: determine auctions with most bids in last period	8	GB, GBW, J	50
q8	**Monitor new users**: identify active users in last period	8	GBW (×2), J	1400
q11	**User sessions**: compute number of bids each user makes while active	3	GBW	60

Table 2: Nexmark queries [45]. The number of operators excludes sources and sinks. "Stateful" lists stateful operators used by the query: GB for GroupBy, GBW for GroupBy (window), and J for Join. Minimal rates are for single-task configurations with 4-GB profiles.

8.1 Experimentation infrastructure

Target workload and queries. We use representative queries from the Flink SQL implementation of Nexmark [45], a reference benchmark also used in the evaluation of DS2 [24]. The benchmark reproduces an online auction system, with various continuous queries. According to our target assumptions, we consider queries that employ state under the windowed model, summarized in Table 2. Nexmark provides a data generator that we use with its default settings, i.e., the input event stream features 2% of events linked to persons, 6% proposing auctions, and 92% representing bids by the former to the latter. The average sizes of these events are respectively 200, 500, and 100 Bytes. We populate the data lake (Kafka) with pre-generated data streams that we use as data-at-rest input for StreamBed operations.

Queries q1 and q2 use a single, stateless operator. Other queries include stateful operators including GroupBy (window) and Joins. Queries q5 and q8 have complex graphs with 8 operators, including GroupBy and Joins. Both queries encounter skewed streams (hot items and sellers). Query q11 uses a pipeline of three operators, including a compute-heavy GroupBy (window). All stateful functions use a time window of 10 seconds. For q5, windows slide in increments of 2 seconds while they are non-overlapping for q8 and q11.

Experimental setup. We use an 85-node cluster from the Grid5000 federated testbed [7]. Nodes in our cluster have an 18-core Intel Xeon Gold 5220 and 96 GB of RAM each. Their 480-GB SSD has sequential read and write performance of 540 and 520 MB/s. They are connected with 25-Gbps Ethernet.

StreamBed needs sufficiently many nodes supporting data injection and source operators. Table 2 presents as a reference the minimal, single-task rate for each query. Queries q1, q2, and to a lesser extent q8 have a high minimal rate, while q5 and q11 process fewer events per second.

When replaying data at rest, the limiting factor are our modest SSDs, with a peak capacity per Kafka server of about 1,200,000 events/s. This represents, for our target queries, a minimal number of 8 Kafka nodes for $q5$ and $q11$ and 16 Kafka nodes for $q1$, $q2$, and $q8$ to be able to inject rates decided during the RE exploration (obviously, using machines with multiple, high-performance drives would allow reducing this number). Rate-limited source operators are CPU-bound: We must also ensure they never are the limiting factor when testing a specific configuration. We identified that 64 source tasks are necessary to support the maximal rate that the Kafka nodes can serve (4 servers).

The test cluster uses 3 servers with 48 task slots to leave 2 available cores per machine for system management. Similarly, we use up to 64 GB of RAM per machine, resulting in a maximum profile of 4 GB per task. We also dedicate 3 nodes for the Kubernetes management, Prometheus, and the Flink Job Manager.

8.2 Experimentation results

(RQ1) MST estimations. We first evaluate the accuracy of the CO and CE in identifying the MST of configurations.

A run for the CE with queries q1, q2, and q11 uses a warmup of 120 s and measurements of 75 s (30 s ramp-up, 30 s observation, and 15 s cooldown). The cooldown uses a throughput of 200 events per second and per source, for a total of 6,400 events/s. One CE run performs 8 iterations for a total duration of 645 s. For complex queries q5 and q8, our initial evaluations highlighted the need for longer measurements, due to the longer time necessary for our configurations to use disk storage (in the RocksDB backend [14])

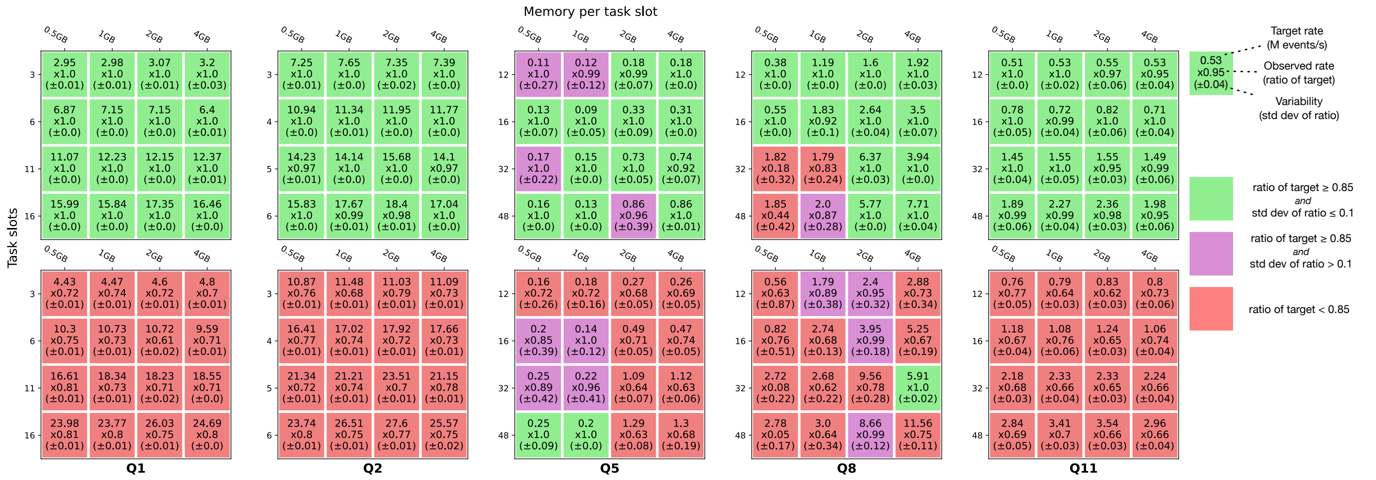

Figure 6: Accuracy of MST estimations for various resource budgets. The estimated MST is replayed at 100% (upper row) or 150% (lower row) of the prediction. Colors represent runs that sustainably process the target rate (green), approach or meet the target with instabilities (purple), or fail (red).

and not only main memory: We use a 450 s warm-up, a 900 s duration, and 7 iterations, for a total of 900 s (15 minutes); we also increase cooldown throughput to 12,800 events/s.

We query the CO for 16 combinations of resource budgets and profiles per query. We consider 3 to 16 tasks for q1, 3 to 6 for q2, and 12 to 48 for q5, q8, and q11 (the high achievable rate of q1 and q2 makes it unnecessary to test them with the full cluster). We use memory profiles of 0.5, 1, 2, and 4 GB.

We run each query with the 16 configurations returned by the CE, using their estimated MSTs as the target rate. We observe if each configuration supports the estimated rate sustainably for 10 minutes preceded by a warmup of 2 minutes. Figure 6 presents the results in its upper row. For each of the 16 configurations, a box presents first the target rate returned by the CE for this configuration (for instance, q8 with 12 TS of 2 GB has an MST of 1.92×10^{6} events/s) and the observed rate, expressed as a ratio of this target. A high level of variation in the measurements, as presented by the standard deviation of this ratio across all measurements, is a sign that the job is close or past saturation. Finally, we report in the lower row of Figure 6 results using a target rate of 150% of the estimated MST. *Passing* this test means the estimation was too conservative.

Simple queries q1 and q2 show little variations. All their tested configurations sustainably process the estimated MST at 100% but fail with 150%. Query q11 shows good results but slightly higher variations. This is due to the behavior of its stateful GroupBy (window) operator and resulting stragglers. For these three queries, a general linear scale behavior seems to apply with increasing numbers of TS. The impact of memory is less clear and highlights the variability of the in-situ data that the RE receives as an input (and that BO will address by repeating runs where RMSE error is important).

For complex queries q5 and q8, we observe a lower level of repeatability and lower scaling capabilities in terms of task slots. For q5, the factor of instability lies mostly in the Join operator and its very uneven load across tasks, as we also highlight in the next experiment. For q8, variability is caused by the presence of stragglers due to the windowed operators using a non-overlapping window size of 10 seconds, higher than our 5-second monitoring period, and leading to "sawtooth-like" load profiles. For both queries, results with 150% of the MST injected show that some estimations (in particular with 32 TS/4 GB and 48 TS/2 GB for q8) can be too conservative. This behavior is not predictable from one run to the next—we did not select a "good run" but the one performed in the whole series. This highlights that the RE must accommodate variations in the data collected in particular for more complex queries.

For all memory profiles, q1, q2, and q11 achieve at worst 95% of the expected rate, and often 99-100%, this showing we manage to answer positively (RQ1). For q5 and q8, performance is much lower and very volatile with 0.5 and 1 GB. This is caused by state needs that cannot be satisfied with these feeble memory values, highlighting the impact of insufficient memory for efficient scaling. In the next sections, we focus on 2- and 4-GB profiles for these two queries.

(RQ2) Configurations. We now evaluate the result of the CO optimization, zooming in the largest configuration for each query running at 100% of the estimated MST (i.e., bottom-right corners of matrices in Figure 6's first row). Figure 7 presents the distribution of the time series measurements

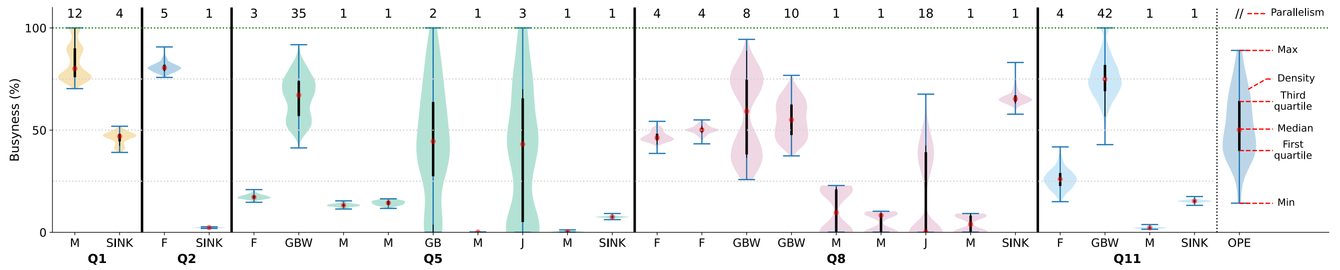

Figure 7: Distribution of task busyness levels measurements. Numbers above box plots are operators' parallelism.

Query	Requested rate	Number of TS with profile 0.5 GB	1 GB	2 GB	4 GB
q1	160 $\times 10^6$	179	179	179	178
q2	190 $\times 10^6$	69	69	69	69
q5	2.5 $\times 10^6$	-	-	1069	1079
q8	15 $\times 10^6$	-	-	179	176
q11	20 $\times 10^6$	565	564	562	559

Table 3: Capacity planning results for the five queries.

	– Min/Max –		– Runs –				— Coefficients —		
Query	TS	RAM	#CO	#CE	Duration	Model	a	b	c
q1	2/16	0.5/4	9	10	139 min.	lin	1.0	9.9E5	-7.6E5
q2	2/6	0.5/4	14	20	248 min.	lin	7.5	3.0E6	-2.7E6
q5	9/48	2/4	14	14	252 min.	log	-7.6E3	5.7E5	-1.2E6
q8	9/32	2/4	13	13	234 min.	sqrt	2.6E3	1.4E6	-3.9E6
q11	4/48	0.5/4	16	20	252 min.	lin	4.1	3.9E4	-2.1E5

Table 4: RE results: training costs, chosen model, and coefficients. #CO/#CE: number of calls to the modules.

over each 10-minute run. Most scaled-out operators reach their peak capacity at some point in time during the run, even if the median busyness is lower. This is the expected behavior: Operators must be provisioned to handle peak load for non-minimal configuration, i.e., when the parallelism of all operators is not fixed to 1 task. For Group By (window) and Joins, we observe a wide range of busyness levels, due to the skew and the stragglers resulting from upstream Group By (window) operators' uneven output rates. For the Join of q8, additional tests with a forced lower parallelism (not shown) lead to saturations, back-pressure cascades, and instabilities; the measured busyness of 60% seems to be a practical maximum. Some stateless, scaled-out operators such as the first Filter of q5, q8, and q11 have relatively low busyness levels, due to an important level of back-pressure from downstream tasks. Overall, the CO can avoid under-provisioned operators with frequent busyness levels of 100%, thus answering favorably the (RQ2).

(RQ3) Capacity planning: accuracy. Our final evaluation explores the capacity of StreamBed to build accurate capacity planning model (RQ3) and the cost of building this model (RQ4).

Table 3 presents the uses of the models for large requested rates and Table 4 presents their results and building costs. Queries q1, q2, and q11 are identified with a linear scaling model, while q5 and q8 respectively use the log and square root models. We can see that the models identify a minor impact of memory profiles for all queries (also clear in the values of coefficient a in the models). For q5, the small variation (<1%) between the 2- and 4-GB profiles is a result of the uncertainty in the CO measurements. We can also observe that the effect of increasing rates differs between tasks, e.g., with coefficient b in the linear models of q1, q2, and q11.

We validate the predictions using large-scale production runs. These runs use up to 85 nodes, using up to 69 nodes for Flink TMs (for q5) and up to 36 Kafka nodes (for q11). As Kafka in our cluster does not allow supporting the rates requested for q1 and q2 we inject for these queries data directly using sources paired with the Nexmark generator.

We adopt a similar strategy as for testing small-scale runs, to verify that the predictions of StreamBed are neither over- or under-provisioned. A prediction is not under-provisioned if it sustainably supports the requested rate over time. A prediction is not over-provisioned if, when injecting a higher rate (we use 120% and 150% of the requested rate) the query shows signs of instability or insufficient capacity. High instability in the observed rate shows that we are above the MST and should be avoided. We also observe the *pending records* metrics, i.e., how many events generated by the fixed-rate source "pile up" at the source operator due to back pressure. While a small amount of buffering is normal in a setup close to full utilization, an ever-increasing pending records metric is a clear sign of an under-provisioned system.

Figure 8 presents the results of production-scale runs using the number of TS (cores) given in Table 3. We measure rate and pending records metrics after a ramp-up period of 5

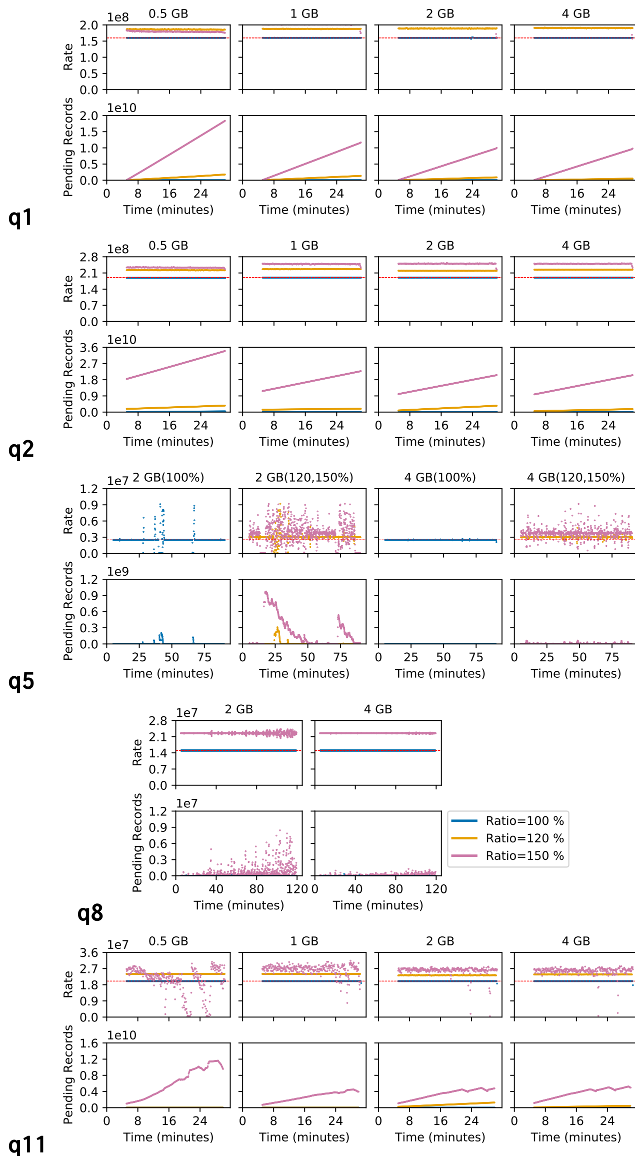

Figure 8: Large-scale production runs using capacity planning models (Table 3). We present the requested rate (thin red line) and the actual rate when injecting 100% (blue), 120% (yellow), and 150% (purple) of this rate. The second row presents accumulated pending records. We present the 100% and (120%, 150%) cases separately for the two profiles of q5 for readability.

minutes, for 30 minutes (q1, q2, and q11), 90 minutes (q5), and 120 minutes (q8). For q1 and q2, the actual rate is exactly the one requested at 100%, while 120% and 150% rates are not matched and lead to piling-up pending records. After 30 minutes, q2 with the 0.5 GB profile shows a slight delay in processing events (the equivalent of 1.6 s worth of pending records), illustrating the difficulty of running even stateless queries at large-scale and high-throughput with Flink under memory constraints. We do not observe this behavior for other setups, including stateful q11 with all profiles.

For q5, for readability, we present separately the 100% rate and the two other rates for the two considered profiles (2 GB and 4 GB). The 100% rate is sustained in both cases but with 2 GB per TS, we observe temporary instabilities. Resulting pending records are, however, later absorbed by the job indicating that the saturation point is not reached. With 4 GB the query is very stable while in both cases, we observe chaotic behavior in terms of throughput for 120% and 150%, meaning the saturation point was reached.

For q8 we consider only the 100% and 150% cases for readability. Interestingly, we can see the impact of long-term instabilities that StreamBed can account for in the model (i.e., as these show earlier on small-scale jobs than on large-scale ones with high parallelisms for the concerned operators). Here, while the query is initially able to process 150% of the requested rate for a time, we observe that pending records gradually accumulate and provoke instabilities, making the query non-sustainable. The reason is that the working set of the query fits for a time dependent on the total memory before introducing congestion. We observe this effect both with 2- and 4-GB profiles.

Overall, the predictions are accurate, with a slight over-provisioning in some cases as can be expected with our 110% over-provisioning factor, positively answering (RQ3).

(RQ4) Capacity planning: cost. Table 4 presents the results of a run of the RE for each of the five queries, with the number of calls to the CO and CE, the training time, and the resulting models. The training by the RE uses 9 (for q1) to 16 (for q11) calls to the CO. These calls result in 10 to 20 calls to the CE, as not all CO calls require evaluating a single-task configuration that is already in the cache. The complete duration of a CO call is about 24-minute long when the single-task configuration did not run, and about 13-minute when it did (for q5 and q8 a CO call always takes 18 minutes, as single-task configurations are tested first as part of the corners to bootstrap D). The CO optimization itself takes about 100 ms, while BO steps at the RE level have a negligible duration (<1 ms). In total, the total training duration ranges from 139 minutes for q1 to 252 minutes for q5. These durations are reasonable considering the costs and scale of the target production deployments, answering favorably (RQ4).

9 RELATED WORK

We review work on DSP performance analysis, modeling, and prediction, and on capacity planning for other contexts. We omit a discussion of the large body of work on elastic scaling of DSP engines but refer to existing surveys [10, 36].

DSP Benchmarking. The first class of systems targets the in-situ performance evaluation of DSP using benchmarking methodologies. In contrast with StreamBed, these systems target the deployment of a query at a production scale and do not intend to extrapolate from small-scale deployments. StreamBench [29] proposes a set of benchmark programs for Apache Storm and Apache Spark Streaming. Theodolite [22] is a benchmarking framework for Flink and Kafka Streams. Theodolite implements different search strategies piloting tests with varying CPU budgets, verifying if service-level objectives (SLO) are met for each of them, and forming a scalability profile. The binary search strategy resembles the method used in the CE, although it applies to several, independent runs. Neither works provide a solution to decide on the configuration (level of parallelism) of each operator of a query, which must be set by the user. Rafiki [34] allows determining such configurations by piloting a series of runs and gradually increasing the parallelism of each operator based on observed backpressure metrics. Gadget [5] is a benchmark suite targeting the performance of the storage subsystem in DSP, e.g., RocksDB [14] and alternatives.

DSP modeling Another class of work proposes to *model* the DSP queries, their performance, and their scalability. Truong et al. [44] use queueing theory to build a performance model of a specific query and its individual operators. Complementarily, MEAD [37] models the arrival of events at different operators using Markovian Arrival Processes to better predict performance after scale-out under bursty event patterns. The modeling of queries can enable proactive, predictive scaling and scheduling operations [28]. Twitter's Caladrius [25] models for this purpose the performance of parallel operators in Apache Heron using piecewise linear regression. These works do not take into account the sub-linear scaling behavior of complex queries.

Some authors propose to model the performance and resource usage of queries based on general characteristics, learning from a large set of queries and identifying a general model, e.g., mixture-density networks [27] or zero-shot cost models [1, 1, 21]. These approaches require initial training using a large set of queries (typically, several thousands). As no dataset of *real* queries of this size exists, the authors have to resort to synthetic, randomly-generated queries that may not represent real workloads.

Capacity planning. The need to plan the configuration or scale of computing infrastructure is obviously not limited to DSP. Higginson *et al.* [23] discuss the applicability of resource forecasting techniques based on machine learning for clustered database systems. URSA [47] is a capacity planning and scheduling system for database platforms, modeling the response of a database workload to provided resources and deriving just-sufficient resource specifications automatically. These works do not rely on controlled, small-scale testing and extrapolation as proposed by StreamBed.

10 CONCLUSION

We presented StreamBed, a capacity planning system for stream processing allowing to derive scaling and configuration decisions for large-scale Flink jobs from a series of controlled, small-scale runs of a target query. StreamBed permits to estimate accurately the needed capacity to run sustainably very large-scale queries without the need to execute them on actual production infrastructure or to build less accurate and expensive generic models. Moreover, it can identify cheap configurations able to run non-linear scaling queries that the state-of-the-art elastic scaler DS2 is not able to converge to. Our work opens several interesting perspectives that we intend to explore in our future work. First, we would like to consider the integration of synthetic data generation and upscaling mechanisms, as it exists for relational databases [39]. Second, we observed that some queries have a clearly sub-linear scaling profile, while elastic scaling solutions generally consider linear scaling assumptions. The models generated by StreamBed, or some of its methodologies, could guide the development of more accurate elastic scalers for such queries.

Artifact availability: The code of StreamBed together with all material allowing the reproduction of our experiments is available at the companion repository:
https://github.com/CloudLargeScale-UCLouvain/StreamBed/.

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] Pratyush Agnihotri, Boris Koldehofe, Carsten Binnig, and Manisha Luthra. 2023. Zero-Shot Cost Models for Parallel Stream Processing. In *6th Intl. Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM)*.

[2] Pratyush Agnihotri, Boris Koldehofe, Paul Stiegele, Roman Heinrich, Carsten Binnig, and Manisha Luthra. 2024. ZeroTune: Learned Zero-Shot Cost Models for Parallelism Tuning in Stream Processing. In *ICDE 2024*.

[3] Candelieri Antonio. 2021. Sequential model based optimization of partially defined functions under unknown constraints. *Journal of Global Optimization* 79, 2 (2021), 281–303.

[4] Apache Foundation. 2015. *Zeppelin*. https://zeppelin.apache.org

[5] Esmail Asyabi, Yuanli Wang, John Liagouris, Vasiliki Kalavri, and Azer Bestavros. 2022. A new benchmark harness for systematic and robust evaluation of streaming state stores. In *EuroSys 2022*.

[6] Raphaël Barazzutti, Thomas Heinze, André Martin, Emanuel Onica, Pascal Felber, Christof Fetzer, Zbigniew Jerzak, Marcelo Pasin, and Etienne Riviere. 2014. Elastic scaling of a high-throughput content-based publish/subscribe engine. In *ICDCS 2014*.

[7] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, et al. 2006. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *The International Journal of High Performance Computing Applications* 20, 4 (2006), 481–494.

[8] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1718–1729.

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[10] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2022. Runtime Adaptation of Data Stream Processing Systems: The State of the Art. *ACM Comp. Sur.* 54 (2022).

[11] CNCF. 2014. *Prometheus*. https://prometheus.io/

[12] COIN-OR Foundation. 2005. *CBC*. https://github.com/coin-or/Cbc

[13] COIN-OR Foundation. 2005. *PuLP*. https://coin-or.github.io/pulp/

[14] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. on Storage* 17, 4 (2021).

[15] Omar Farhat, Harsh Bindra, and Khuzaima Daudjee. 2020. Leaving stragglers at the window: Low-latency stream sampling with accuracy guarantees. In *DEBS 2020*.

[16] Flink. 2022. *Kafka Connector*. https://github.com/apache/flink-connector-kafka

[17] Apache Flink. 2023. Release 1.14.1. https://mvnrepository.com/artifact/org.apache.flink/flink-java/1.14.1.

[18] Gyula Fóra and Mattias Andersson. 2023. Flink Kubernetes Autoscaler. https://github.com/apache/flink-kubernetes-operator/tree/main/flink-kubernetes-operator-autoscaler.

[19] Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. 2016. Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join. *IEEE Trans. on Big Data* 7, 2 (2016).

[20] Tim Head et al. 2021. *Scikit-optimize*. https://zenodo.org/record/5565057

[21] Roman Heinrich, Manisha Luthra, Harald Kornmayer, and Carsten Binnig. 2022. Zero-shot cost models for distributed stream processing. In *DEBS 2022*. 85–90.

[22] Sören Henning and Wilhelm Hasselbring. 2022. A configurable method for benchmarking scalability of cloud-native applications. *Empirical Software Engineering* 27 (2022).

[23] Antony S Higginson, Mihaela Dediu, Octavian Arsene, Norman W Paton, and Suzanne M Embury. 2020. Database workload capacity planning using time series analysis and ML. In *SIGMOD 2020*.

[24] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *OSDI 2018*.

[25] Faria Kalim et al. 2019. Caladrius: A performance modelling service for distributed stream processing systems. In *ICDE 2019*.

[26] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking distributed stream data processing systems. In *ICDE 2018*.

[27] Alireza Khoshkbarforoushha, Rajiv Ranjan, Raj Gaire, Ehsan Abbasnejad, Lizhe Wang, and Albert Y Zomaya. 2016. Distribution based workload modelling of continuous queries in clouds. *IEEE Transactions on Emerging Topics in Computing* 5, 1 (2016), 120–133.

[28] Teng Li, Jian Tang, and Jielong Xu. 2016. Performance modeling and predictive scheduling for distributed stream data processing. *IEEE Transactions on Big Data* 2, 4 (2016), 353–364.

[29] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. 2014. StreamBench: Towards benchmarking modern distributed stream computing frameworks. In *UCC 2014*.

[30] Maximilian Michels. 2023. FLIP-271: Autoscaling. https://cwiki.apache.org/confluence/display/FLINK/FLIP-271%3A+Autoscaling.

[31] Andrew Y Ng. 1997. Preventing "overfitting" of cross-validation data. In *ICML 1997*.

[32] Fabian Paul. 2020. Towards a Flink Autopilot in Ververica Platform. https://www.youtube.com/watch?v=5HvGo5vRxRA.

[33] F. Pedregosa et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011).

[34] Benjamin JJ Pfister, Wolf S Lickefett, Jan Nitschke, Sumit Paul, Morgan K Geldenhuys, Dominik Scheinert, Kordian Gontarska, and Lauritz Thamsen. 2022. Rafiki: Task-Level Capacity Planning in Distributed Stream Processing Systems. In *Euro-Par 2021 workshops*.

[35] Tilmann Rabl, Jonas Traub, Asterios Katsifodimos, and Volker Markl. 2016. Apache Flink in current research. *it-Inf. Tech.* 58, 4 (2016).

[36] Henriette Röger and Ruben Mayer. 2019. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Computing Surveys (CSUR)* 52, 2 (2019), 1–37.

[37] Gabriele Russo Russo, Valeria Cardellini, Giuliano Casale, and Francesco Lo Presti. 2021. MEAD: Model-based vertical auto-scaling for data stream processing. In *CCGrid 2021*.

[38] Subham Sahoo, Christoph Lampert, and Georg Martius. 2018. Learning equations for extrapolation and control. In *ICML 2018*.

[39] Anupam Sanghi, Shadab Ahmed, and Jayant R Haritsa. 2022. Projection-compliant database generation. *Proceedings of the VLDB Endowment* 15, 5 (2022), 998–1010.

[40] Anand Sanmukhani. 2019. *Python Prometheus API client*. https://github.com/4n4nd/prometheus-api-client-python

[41] Rayman Preet Singh, Bharath Kumarasubramanian, Prateek Maheshwari, and Samarth Shetty. 2020. Auto-sizing for stream processing applications at LinkedIn. In *HotCloud 2020*.

[42] Spotify. 2023. Kubernetes Operator for Apache Flink. https://github.com/spotify/flink-on-k8s-operator.

[43] Strimzi. 2023. Strimzi provides a way to run an Apache Kafka cluster on Kubernetes in various deployment configurations. https://strimzi.io.

[44] Tri Minh Truong, Aaron Harwood, Richard O Sinnott, and Shiping Chen. 2018. Performance analysis of large-scale distributed stream processing systems on the cloud. In *IEEE Cloud 2018*.

[45] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2010. *NEXMark–a benchmark for queries over data streams*. Technical Report. OGI School of Science & Engineering at OHSU.

[46] Juliane Verwiebe, Philipp M Grulich, Jonas Traub, and Volker Markl. 2023. Survey of window types for aggregation in stream processing systems. *The VLDB Journal* (2023), 1–27.

[47] Wei Zhang, Ningxin Zheng, Quan Chen, Yong Yang, Zhuo Song, Tao Ma, Jingwen Leng, and Minyi Guo. 2020. URSA: Precise capacity planning and fair scheduling based on low-level statistics for public clouds. In *ICPP 2020*.

[48] Beiji Zou, Tao Zhang, Chengzhang Zhu, Ling Xiao, Meng Zeng, and Zhi Chen. 2022. Alps: An Adaptive Load Partitioning Scaling Solution for Stream Processing System on Skewed Stream. In *DEXA 2022*.