RESEARCH-ARTICLE

CAPSys: Contention-aware task placement for data stream processing

YUANLI WANG, Boston University, Boston, MA, United States

LEI HUANG, Boston University, Boston, MA, United States

ZIKUN WANG, Boston University, Boston, MA, United States

VASILIKI KALAVRI, Boston University, Boston, MA, United States

IBRAHIM MAHER MATTA, Boston University, Boston, MA, United States

Published: 30 March 2025

Citation in BibTeX format

EuroSys '25: Twentieth European Conference on Computer Systems
March 30 - April 3, 2025
Rotterdam, Netherlands

Conference Sponsors:
SIGOPS

CAPSys: Contention-aware task placement for data stream processing

Yuanli Wang[*]
Boston University
Boston, Massachusetts, USA
yuanliw@bu.edu

Lei Huang[*]
Boston University
Boston, Massachusetts, USA
lei@bu.edu

Zikun Wang
Boston University
Boston, Massachusetts, USA
zikunw@bu.edu

Vasiliki Kalavri
Boston University
Boston, Massachusetts, USA
vkalavri@bu.edu

Ibrahim Matta
Boston University
Boston, Massachusetts, USA
matta@bu.edu

Abstract

In the context of streaming dataflow queries, the *task placement* problem aims to identify a mapping of operator tasks to physical resources in a distributed cluster. We show that task placement not only significantly affects query performance but also the convergence and accuracy of auto-scaling controllers. We propose *CAPSys*, an adaptive resource controller for dataflow stream processors, that considers auto-scaling and task placement in concert. *CAPSys* relies on *Contention-Aware Placement Search (CAPS)*, a new placement strategy that ensures compute-intensive, I/O-intensive, and network-intensive tasks are balanced across available resources.

We integrate *CAPSys* with Apache Flink and show that it consistently achieves higher throughput and lower backpressure than Flink's strategies, while it also improves the convergence of the DS2 auto-scaling controller under variable workloads. When compared with the state-of-the-art ODRP placement strategy, *CAPSys* computes the task placement in orders of magnitude lower time and achieves up to 6× higher throughput.

CCS Concepts: • **Information systems** → **Stream management**.

Keywords: Stream Processing, Distributed systems, Resource management, Scheduling

ACM Reference Format:
Yuanli Wang, Lei Huang, Zikun Wang, Vasiliki Kalavri, and Ibrahim Matta. 2025. *CAPSys*: Contention-aware task placement for data stream processing. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689031.3696085

[*]Equal contribution.

1 Introduction

Stream processing systems (SPSs) are complex distributed frameworks that require careful expert configuration and oversight to achieve good performance [12, 21, 25, 37]. The problem of automatically tuning the configuration of SPSs has received wide attention, with recent works focusing on resource allocation [20, 22, 30, 32, 43, 49], batch size tuning [16], task scheduling [36, 50], and dynamic partitioning [24, 34, 38]. In this work, we turn our attention to automatic *task placement*: deciding how to assign streaming tasks to available workers in a distributed cluster of resources.

Stateful SPSs, like Apache Flink and Storm [1, 3], adopt the slot-oriented resource allocation model established by Hadoop, Dryad, Yarn, and Mesos [26, 28, 46]. Unlike systems that perform real-time fine-grained scheduling [39, 50], slot-based systems compute a static task assignment at job deployment time or upon reconfiguration. However, the default task assignment strategies in modern SPSs are random, under the assumption of homogeneity [2, 40]. This simplistic approach is problematic, as streaming tasks can have diverse resource requirements. A typical streaming query can include simple stateless tasks, such as lightweight filters and transformations, as well as stateful windows, joins, and machine learning inference [8, 9, 27]. As a result, the mapping of tasks to available slots is critical for query performance and resource efficiency.

Moreover, we find that task placement has major implications for streaming auto-scaling and significantly impacts the convergence and accuracy of elasticity controllers [30]. Ignoring task placement has two major implications for resource auto-scaling. First, if the original task placement is poor, the elasticity model is informed by inaccurate metrics and may underestimate the processing capacity of task slots, leading to overshooting. Second, if a sub-optimal task placement is chosen when effecting the scaling decision, the controller eventually triggers additional scaling steps, taking longer to converge.

We believe the reason behind the prevalence of random and rule-based strategies in systems like Apache Flink and Storm [1, 3] is that the optimal task placement problem is NP-hard. While sophisticated approaches have been proposed for wide-area streaming analytics [15, 29, 51], relevant research targeting data center environments is scarce. The few non-trivial task placement solutions that have been proposed in the literature are either prohibitively expensive for online settings [13, 14] or require expert user input [40].

To address these challenges, in this paper we propose *Contention-Aware Placement Search (CAPS)*, a new placement strategy that considers the effect of co-locating resource-intensive tasks on the same worker. We introduce an analytical cost model that captures compute, storage, and network usage of distributed workers and aims to minimize resource imbalance in the cluster. We further propose various optimizations and empirically-verified heuristics to prune the vast search space of alternative task assignments.

To demonstrate the practical value of *CAPS*, we implement *CAPSys*, an adaptive resource controller for Apache Flink. *CAPSys* augments Flink's auto-scaling controller with an online placement optimizer that ensures compute-intensive, I/O-intensive, and network-intensive tasks are balanced across available resources. *CAPSys* can quickly identify a set of good plans to consider on reconfiguration and it is very efficient even for large deployments and variable workloads.

Our evaluation results show that *CAPSys* reduces back-pressure and increases throughput without additional resources, when compared to the default and state-of-the-art placement policies in modern SPSs. When used jointly with DS2, *CAPSys* alleviates oscillations and requires up to 8 fewer reconfiguration steps compared to the baseline. Further, we evaluate *CAPSys* in a large-scale, multi-tenant setting with 144 task slots and show it is the only placement strategy that can achieve the target performance for all queries.

Our contributions. We review existing task placement strategies and analyze their limitations (§ 2.2). We then perform a comprehensive empirical study of task placement in SPSs and we experimentally validate the implications of task placement on query performance (§ 3). Based on the results of our study, we formulate a cost model that captures *the co-location degree* of resource-intensive tasks in alternative placement plans (§ 4.2). We formalize the task placement problem and propose a new strategy for identifying good placement plans, called *CAPS* (§ 4.3-4.4). *CAPS* automatically eliminates plausibly low-performing plans and applies various optimizations to prune the search space further and return a placement plan with minimal cost. Finally, we design and implement *CAPSys*, an adaptive resource controller for streaming dataflows powered by auto-scaling and task placement co-design (§ 5). We integrate *CAPSys* with the Apache Flink SPS and evaluate it across diverse queries and variable workload scenarios (§ 6).

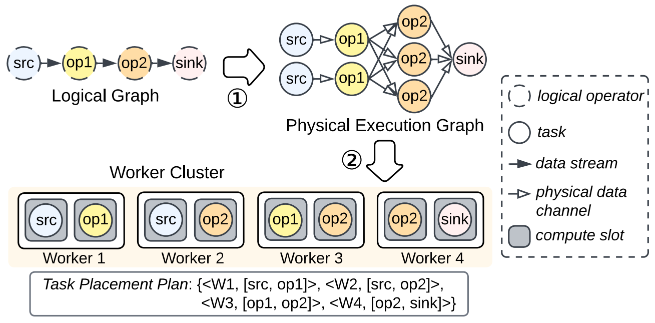

Figure 1. Streaming dataflow deployment steps

2 Preliminaries and motivation

We revisit fundamentals of dataflow stream processing, introduce necessary terminology, and describe the task placement problem. Then, we discuss existing task placement strategies and analyze their limitations.

2.1 Streaming dataflow concepts

In the streaming dataflow model [8, 21], a query is represented as a logical directed acyclic graph, where vertices denote **logical operators** and edges describe **data streams**. Upon deployment, the logical graph is translated to a physical execution graph (① in Figure 1), where each logical operator is replicated to multiple **tasks** and each data stream is instantiated to multiple **physical data channels** that connect tasks of the upstream and downstream operators. All tasks of a logical operator execute an identical processing logic on disjoint data partitions, and the number of tasks, or the **parallelism** of a logical operator, can be set manually by the user or automatically by an auto-scaling policy, such as DS2 [30]. Streaming tasks are scheduled on workers according to a **task placement strategy** and are *long-running*. Once deployed, tasks remain on their assigned worker until an explicit reconfiguration action is triggered.

Resource model. Figure 1 shows a general resource model that is adopted by popular stream processing systems (SPS), such as Apache Flink [1] and Apache Storm [3]. In this model, resources are exposed to the SPS scheduler as a set of homogeneous **workers** (e.g., virtual machines, containers, bare-metal nodes) connected by the datacenter network. Each worker contains a fixed number of compute **slots**, such that one slot can accommodate at most one task. A slot corresponds to one processing thread in the implementation, however, tasks assigned to the same worker share other resources, such as memory, network, and I/O bandwidth. A slot is the smallest unit of compute that can be performed on a worker. The number of slots that are assigned to each logical operator is equal to its parallelism.

Task placement is the process of assigning each task in the physical execution graph to a compute slot in the worker cluster. An example is shown in Figure 1 ②. The **task placement plan** is a mapping from each worker to a subset of physical tasks. Given a physical execution graph and a worker cluster, there exists a large space of possible placement plans. In this work, our goal is to quickly and effectively select one plan that delivers good performance.

2.2 Limitations of existing task placement strategies

The placement plan is crucial to query performance, as co-located tasks share underlying resources. We find that placement strategies employed by industry-level and academic SPSs suffer from at least one of the following limitations:

Task homogeneity assumption. The default task placement strategies in systems like Apache Flink and Storm assume that tasks of different operators are homogeneous with respect to resource requirements. Specifically, Flink's default policy iterates over workers, filling up all of a worker's available slots before moving on to the next. However, the tasks to be scheduled are selected at random and placement plans, as well as their performance, can vary significantly across different runs of the same query on the same worker cluster. Flink and Storm also provide resource-aware strategies [2, 40], though, under the assumption of homogeneity, they evenly distribute the *number* of tasks to available workers rather than balance the actual load.

Necessity for manual tuning. A better approach has been recently proposed as an extension to the Storm scheduler [40], where users can specify the resource requirements of operators and the resource capacity of worker nodes. Flink's fine-grained resource management module [4] is a similar feature that allows users to match tasks to slots of configurable size. Though these efforts are a step in the right direction, they place the burden of resource profiling and task placement on the user. This is both error-prone, as users may not be experts in resource profiling, as well as time-consuming.

Long decision time and poor scalability. ODRP [13, 14] is a notable work that proposed solving parallelism assignment and task placement jointly, as an integer linear programming (ILP) problem. ODRP's objective function considers response time, network traffic usage, monetary cost, and availability of resources. Though this approach is able to find the optimal placement under the user-specified constraints, the ILP-based solver needs to exhaustively explore the search space. Furthermore, as the formulation does not specify an objective to sustain the input rate, it might return trivial under-provisioned plans that place all tasks in a single worker. We also found the model cumbersome to tune, as users are expected to set the weights of the multi-objective function. In Section 6.3, we experimentally show that ODRP is only practical for simple queries and does not scale well with the number of tasks.

3 Task placement matters

To address the aforementioned limitations and enable automatic, efficient, and resource-aware task placement, we perform an empirical evaluation study with two principal goals: (i) to quantify the effects of task placement on query performance and (ii) to understand resource contention caused by co-locating different types of tasks. The results of our study motivate the design of our proposed contention-aware placement search strategy at the core of *CAPSys*.

3.1 Experimental methodology

We use three queries with diverse characteristics and complexity. **Q1-sliding** is a simple stateful query that consists of a map operator followed by a sliding window. **Q2-join** is a more complex query with two sources, two map operators, and a tumbling window join that can accumulate large state. **Q3-inf** is a network-intensive query that performs image processing and model inference [27].

We deploy a Flink cluster on 5 AWS EC2 `r5d.xlarge` instances. Each worker has 2 cores, 4 vCPUs, 32GB of memory, and 150GB of SSD storage. The default network bandwidth within the cluster is 10Gbps. We use 1 worker to deploy the Flink Job Manager (JM) and 4 workers to deploy 4 Flink Task Managers (TM) with 4 slots per TM. We use RocksDB as the state backend and enable buffer debloating [19].

For each query, we configure the target input rate to match the capacity of the resource cluster by gradually increasing the input rate until it saturates all workers, and use DS2 [30] to assign parallelism to operators and generate the physical execution graph. We run each experiment for 15 minutes and start collecting metrics after a warm-up period of 10 minutes. We record metrics every 5s and plot the average values, unless otherwise specified. We report backpressure at the source instead of latency, as Flink's latency measurements do not reflect the queuing delay at the source [33].

3.2 Exhaustive placement plan search

We first perform an exhaustive evaluation using **Q1-sliding**. Deploying this query on our 4-worker cluster with 16 slots results in 80 possible placement plans. We execute the query using every plan and collect performance metrics. In Figure 2, we plot throughput and backpressure for the 3 best-performing plans, P1 - P3, and the 3 worst-performing plans, P4 - P6 (The performance of all possible plans can be found in the full technical report [47]). The results reveal a vast performance gap between the best and worst placement plans. With P1, the query achieves 14*k* records/s throughput and 6.8% backpressure, while with P6, the throughput drops to 9*k* records/s and backpressure climbs to 86.4%.

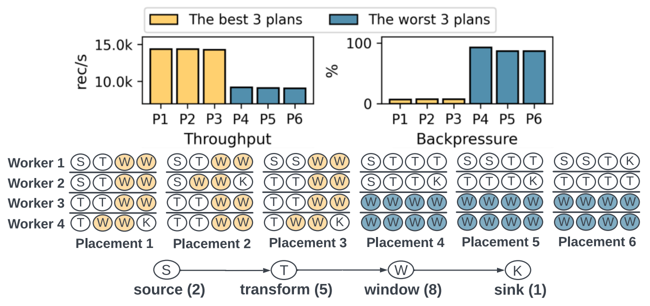

Figure 2. 6 out of 80 alternative placement plans that achieve the best and worst performance for **Q1-sliding** on a 4-worker cluster with 16 slots.

By analyzing the results for all 80 plans, we found that a random placement strategy leads to sub-optimal throughput and high backpressure with high probability. In fact, only 3 out of 80 plans meet the target performance. We compared the characteristics of plans that deliver high throughput and those that do not and we observed that they differ in the way they group sliding window tasks. Specifically, when window tasks are co-located on a few nodes, like in P4 - P6, performance suffers. On the contrary, when window tasks are balanced across multiple workers, as in P1 - P3, the query achieves high throughput and low backpressure. Our hypothesis is that *the performance penalty is caused by resource contention*, as co-located window tasks compete for shared compute and I/O. We investigate the effect of co-locating resource-intensive tasks in more detail next.

3.3 Co-locating resource-intensive tasks

To verify our hypothesis, we examine the contention caused by three resource types: compute, disk I/O, and network. We use **Q2-join** and **Q3-inf** to explore each resource type individually. An exhaustive evaluation is impractical for these complex queries, as the number of possible plans is 665 and 950, respectively, for our experimental setup. Therefore, for the experiments in this section, we manually select placement plans with varying degrees of resource contention.

Co-locating compute-intensive tasks. We use **Q3-inf**, a stateless workload, to explore the effect of co-locating compute-intensive tasks. In particular, we focus on the *inference* operator that performs model inference on images and has high compute requirements. This operator also triggers garbage collection that introduces periodic CPU utilization spikes. We select plans with low contention, P1 - P3, that balance *inference* tasks across all workers, plans with high contention, P7-P9, that co-locate all inference tasks on the same worker, and plans with medium contention, P4-P6. We deploy the query with all 9 plans and plot throughput and backpressure metrics in Figure 3a. The results verify our hypothesis that the co-location degree of *inference* tasks determines the performance of the query. Low-contention plans consistently achieve higer throughput and lowe backpressure than high-contention plans.

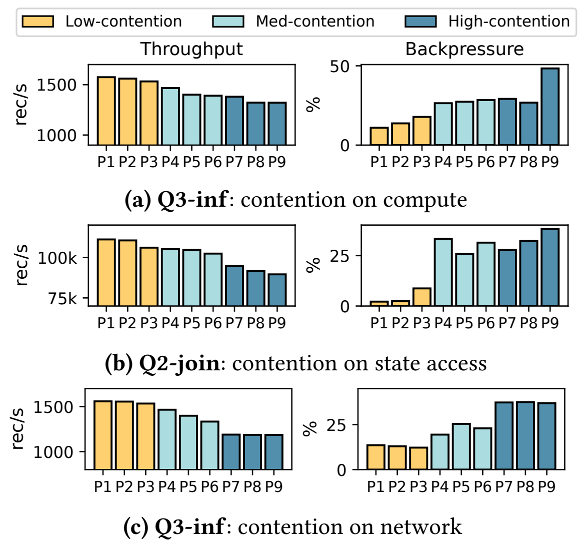

Figure 3. Effect of co-locating resource-intensive tasks of different types on query performance.

Co-locating I/O-intensive tasks. We now explore the effect of co-locating stateful tasks using **Q2-join**. The most I/O-intensive operator in this query is the *tumbling window join*. This operator frequently accesses the local state backend to buffer incoming events and retrieve join results when the window triggers. We run this query using 9 placement plans with varying degrees of contention indicated by the number of *tumbling window join* tasks that are co-located on a single worker. Figure 3b shows the performance results for the selected placement plans. Co-locating many stateful tasks causes high disk I/O utilization and contention in the RocksDB state backend. Specifically, P1-P3 achieve high throughput of 110k rec/s and low backpressure of up to 4%, while throughput for P7-P9 drops to 91k rec/s and backpressure increases to 32%.

Co-locating network-intensive tasks. In this experiment, we use **Q3-inf** and we limit the outbound bandwidth of each worker to 1 Gbps to study the impact of network contention. **Q3-inf** presents an interesting case, as it contains multiple operators that consume and produce large records (images) and can potentially compete for network bandwidth. We consider these traffic-intensive tasks from multiple operators when selecting representative placement plans. The results in Figure 3c verify our hypothesis. The average throughput of low-contention plans is 1555 rec/s and it drops to 1185 rec/s when contention is high. Correspondingly, backpressure increases from 12% on average to 37%.

4 Contention-Aware Placement Search

Motivated by the results of our empirical study, in this section we propose the *Contention-Aware Placement Search (CAPS)*

Notation	Description
G_p	Physical execution graph
N_p	Number of logical operators in G_p
V_p	Set of streaming tasks in G_p
E_p	Set of physical data links in G_p
G_w	Worker cluster
V_w	Set of workers in G_w
E_w	Set of network links between workers in G_w
s	Number of compute slots per worker
$f(\cdot)$	A task placement plan that maps each task in V_p to a worker in V_w
F	The set of all possible placement plans
$U_{cpu}(t)$	CPU utilization of task $t \in V_p$
$U_{io}(t)$	State access rate of task $t \in V_p$
$U_{net}(t)$	Output data rate of task $t \in V_p$
$T_{cpu/io/net}$	Set of tasks with highest $U_{cpu/io/net}$ among all tasks in V_p where $\lvert T_{net}\rvert = s$
$D(t)$	Set of physical downstream data links that originate from task t (-1 for sink tasks)
$D_r(f,t)$	Set of cross-worker physical data links originating from task t under placement f

Table 1. Notation used in this paper

strategy. As shown before [13], computing an optimal task placement is NP-hard. Therefore, finding an exact solution is prohibitive for streaming deployments where dynamic workloads require frequent reconfiguration. To address this challenge, *CAPS* captures the cost of a placement plan by considering the *co-location degree* of resource-intensive tasks. After defining the task placement problem (§ 4.1), we introduce the *CAPS* cost model (§ 4.2) and describe a set of optimizations that we employ to aggressively prune the search space of valid placement plans (§ 4.4). We provide a summary of the notation used throughout this section in Table 1.

4.1 Problem definition

Given the physical execution graph G_p of a streaming dataflow, where operator parallelism is determined by the autoscaling controller (e.g. DS2 [30]), and a fixed worker cluster denoted by G_w, our objective is to find a task placement plan that minimizes resource contention in G_w. Specifically,

- $G_p = (V_p, E_p)$ defines the physical execution graph of the streaming job. V_p describes the task vertices, where each $t \in V_p$ belongs to a logical operator. E_p is the set of all edges, where $l \in E_p$ denotes a physical data link between two tasks.
- $G_w = (V_w, E_w)$ describes the cluster resources. V_w includes all workers, where each $w \in V_w$ is annotated with a fixed number of slots, compute resources, disk, and network bandwidth. E_w is the set of end-to-end network edges between workers, annotated with propagation delay and bandwidth.

We define a *task placement plan* as a task-to-worker mapping $f : V_p \to V_w$ such that each task is assigned to exactly one worker (cf. Eq. 1) and the number of assigned tasks per worker does not exceed its available slots s (c.f. Eq. 2):

$$\forall t \in V_p,\ \exists! w \in V_w\ such\ that\ f(t) = w \tag{1}$$

$$\forall w \in V_w,\ |\{t | t \in V_p \land f(t) = w\}| \le s \tag{2}$$

Given a pair of G_p and G_w, we denote the set of all plans that respect constraints (1) and (2) as F. We also define a cost function $c : f \to \vec{X}$ that assigns a cost vector to a placement plan. Our objective is to identify the plan f_i with the minimum cost, so that:

$$c(f_i) < c(f_j), \forall f_j \in F \setminus \{f_i\} \tag{3}$$

We describe the cost model in detail in Section 4.2.

Model assumptions. To keep our formulation concise, we consider the resource model described in Section 2, where the resources per worker and the number of slots are fixed. We also assume the total number of compute slots in G_w is sufficient to deploy all tasks in G_p. As a result, tasks of the same logical operator are considered identical and any effects of data skew must be addressed by a separate mechanism, such as custom partitioning [38, 42], before task placement. We discuss integration with skew mitigation techniques and other practical considerations in Section 5.2.

4.2 Cost model

The intuition of our cost model is to capture the resource imbalance in the worker cluster, as *the difference of the bottleneck worker's load from the ideal load*, wherein all workers are assigned an equally resource-intensive workload. We express this imbalance across three dimensions: (i) compute cost, (ii) state access cost, and (iii) network cost. We describe each of these cost functions next.

Compute cost. Let $L_{cpu}(f)$ (Equation 5) denote the highest CPU load among all workers under placement plan f and L_{cpu}^{min} (Equation 6) be the per-worker CPU load of a perfectly-balanced compute allocation, where the total CPU load is equally distributed to all workers. Let also L_{cpu}^{max}, given in Equation 7, denote the worst-case CPU load when the most compute-intensive tasks, T_{cpu}, are co-located on the same worker. Equation 4 gives the compute cost of a placement plan f, as the difference of $L_{cpu}(f)$ from L_{cpu}^{min}, normalized by the worst possible difference of CPU load. The case of $L_{cpu}^{max} = L_{cpu}^{min}$ corresponds to the case when all possible placement plans are equivalent. This can happen either when G_w has a single worker or when the number of tasks is equal to the number of slots and all tasks are identical.

$C_{cpu}(f)$ expresses the amount of additional compute requirements imposed on the bottleneck worker in the cluster, compared to the ideal allocation. The value of $C_{cpu}(f)$ is between 0 and 1, with 0 indicating a perfectly balanced assignment and 1 indicating the worst-case.

$$C_{cpu}(f) = \begin{cases} 0 & \text{if } L_{cpu}^{max} = L_{cpu}^{min} \\ \frac{L_{cpu}(f) - L_{cpu}^{min}}{L_{cpu}^{max} - L_{cpu}^{min}} & \text{otherwise.} \end{cases} \quad (4)$$

$$L_{cpu}(f) = \max_{w \in V_w} \sum_{\{t \in V_p | f(t) = w\}} U_{cpu}(t) \quad (5)$$

$$L_{cpu}^{min} = \frac{\sum_{t \in V_p} U_{cpu}(t)}{|V_w|} \quad (6)$$

$$L_{cpu}^{max} = \sum_{t \in T_{cpu}} U_{cpu}(t) \quad (7)$$

State access cost. We define the state access cost $C_{io}(f)$ analogously, by replacing the corresponding loads, $L_{io}(f)$, L_{io}^{min}, and L_{io}^{max}, in Equations 4-7. $L_{io}(f)$ represents the highest state access load among all workers. L_{io}^{min} and L_{io}^{max} denote the perfectly-balanced and the most imbalanced allocation for disk I/O resources. The state access load of a task $U_{io}(t)$ is defined as the sum of read and write disk I/O rate.

Network cost. Defining the network cost, $C_{net}(f)$, is not straightforward because a task's outbound network traffic is affected by its placement. As a result, calculating L_{net}^{min} and the L_{net}^{max} requires computing the placement plan itself. To address this challenge, we approximate L_{net}^{min} and L_{net}^{max} as follows. We set $L_{net}^{min} = 0$ to indicate the case when all tasks are placed on the same worker without incurring any network traffic. We further assume that L_{net}^{max} corresponds to the case of co-locating tasks with the highest output data rate, denoted by T_{net}, on the same worker. Given these approximations, we define the network load of plan f in Equation 8:

$$L_{net}(f) = \max_{w \in V_w} \sum_{\{t \in V_p | f(t) = w\}} (U_{net}(t) \cdot \frac{|D_r(f,t)|}{|D(t)|}) \quad (8)$$

We calculate the network load of a worker as the sum of outbound network traffic of all tasks placed on that worker. We assume that the output data rate of a task t, $U_{net}(t)$, is equally distributed to its downstream data links $D(t)$. We note that only cross-worker downstream data links $D_r(f,t) \subseteq D(t)$ contribute to the outbound network traffic of the worker.

Objective functions. The cost values for compute, state access, and network resources form the three dimensions in the cost vector $\vec{C} = [C_{cpu}, C_{io}, C_{net}]$ of a placement plan. We employ three independent objective functions in our formulation: (i) $\min_f C_{cpu}(f)$, (ii) $\min_f C_{io}(f)$, and (iii) $\min_f C_{net}(f)$. We define the *pareto-optimal* solution with respect to $\vec{C}$ as a placement plan whose cost is not dominated by any other feasible plan across all dimensions. Our goal is to find such a pareto-optimal solution and discard plans that are strictly more costly than others.

4.3 Exploring the placement plan space

We represent the search space of feasible plans as a tree which we navigate in depth-first-search (DFS) order. We employ a 2-step process: (i) the *outer search* explores one computation stage (operator) as a layer of the DFS tree, and (ii) the *inner search* expands each node by considering all workers of the cluster. We illustrate the process with the example of Figure 4.

Outer search. The problem of Figure 4-a consists of a query with 4 operators and a cluster of 3 workers. The outer search, shown in Figure 4-b, explores operators in topological order S → T → I → K. To create a node, we place tasks of the current operator to the worker cluster. Non-leaf nodes correspond to a *partial placement plan*, while leaf nodes identify a complete task assignment. Each layer includes all feasible placement plans for the current operator as separate branches. Since slots, workers, and tasks of the same operator are homogeneous, there are two options in Layer 1: (i) to separate the source tasks on any 2 workers (node 1), or (ii) to co-locate the source tasks on any single worker (node 2).

Inner search. Each node created in the outer search step invokes an inner search process to consider alternative workers. In Figure 4-c, node 3 is expanded to discover task placements for the inference tasks, exploring one worker per layer. In this example, the inner search produces the children nodes 4 and 5, which are added to the tree.

Duplicate elimination. The inner search may produce leaf nodes that correspond to equivalent placement plans. We eagerly eliminate these duplicates, as shown in Figure 4-c: nodes 6 and 7 are identified as leading to duplicate branches and one of them is terminated. Two workers of the same node are handled as possible duplicates if they have been assigned the same set of tasks (e.g., workers 2 and 3 of node 3). In this case, the number of tasks allowed to be placed on a possible duplicate worker is restricted by the number of tasks placed on its parent in the tree. In Figure 4-c, the rightmost branch is terminated as Worker 2 has already been assigned two instances of task I.

4.4 Search space pruning

The cost of searching all feasible plans can become untenable, even for small deployments. As an example, we calculated 3.25 million alternative placement plans and 31 million nodes in the search tree for query **Q3-inf** (§ 3.1), given a cluster with 8 workers and 4 slots each. In this section, we propose two pruning techniques that can effectively reduce the search space and make *CAPSys* practical for the online setting.

4.4.1 Threshold-based pruning.

The first pruning technique relies on the critical observation that *the load L_i ($i \in$ [cpu, io, net]) of a worker increases monotonically* at each layer of the search tree. As more tasks are placed on a worker, its resource consumption grows, allowing us to safely prune

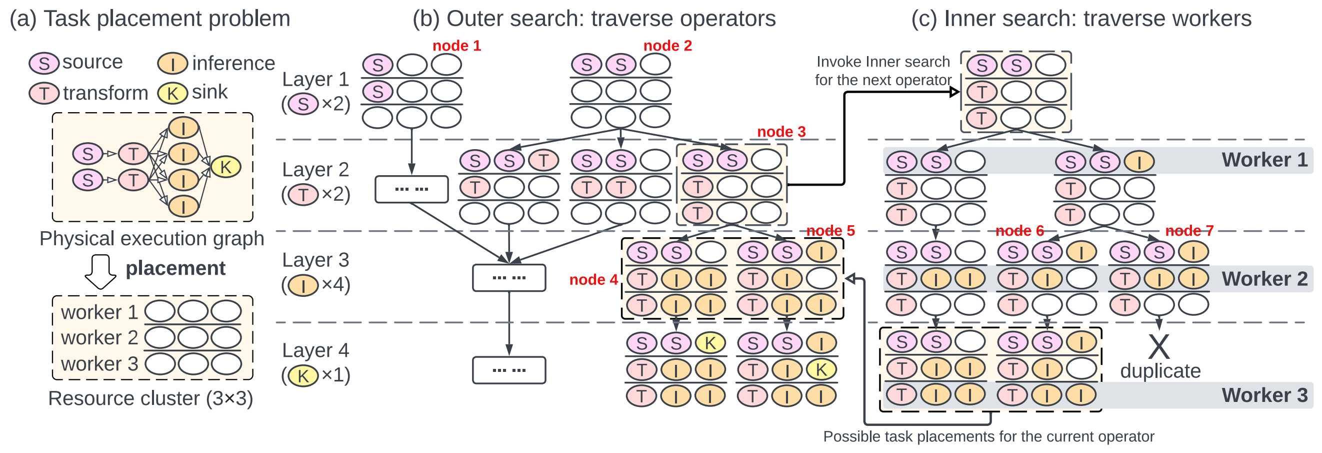

Figure 4. Outer and Inner DFS tree of an example execution. The streaming dataflow has 4 logical operators: S → T → I → K with parallelism 2, 2, 4, 1. The resource cluster has 3 homogeneous workers with 9 compute slots.

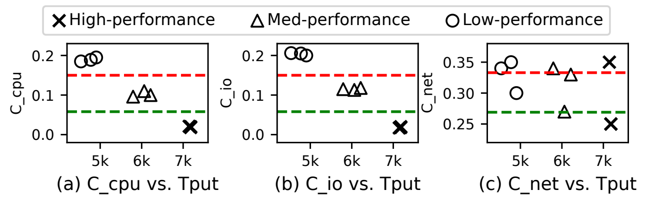

Figure 5. Compute, state access, and network cost of the sample placement plans for **Q1-sliding**

branches once the load of a partial placement plan exceeds some value. Specifically, we define a configurable threshold vector $\vec{\alpha} = [\alpha_{cpu}, \alpha_{io}, \alpha_{net}]$, such that the cost $C_i(f)$ $(i \in [cpu, io, net])$ of resulting placement plans must satisfy the following inequality:

$$C_i(f) \leq \alpha_i, \quad \text{s.t. } i \in [cpu, io, net], \alpha_i \in [0,1] \tag{9}$$

Given Equation 4, we can derive the following inequality that the cost $L_i(f)$ $(i \in [cpu, io, net])$ must satisfy:

$$L_i(f) \leq L_i^{min} + \alpha_i(L_i^{max} - L_i^{min}) \tag{10}$$

A partial placement plan can be eliminated if its accumulated load on any worker violates Equation 10 during the search.

Figure 5 visualizes the relationship between plan costs and their respective throughput for query **Q1-sliding** of Section 3.2. We see that high-performing plans can be effectively separated by setting cost threshold values (dashed lines) across the resource dimensions. We note that c_{net} is not a dominant performance factor, since **Q1-sliding** is not network-intensive. Although a query's sensitivity to each dimension varies, threshold-based pruning is applicable as long as performance is sensitive to at least one dimension.

We further consider task placement of **Q3-inf** on a cluster with 8 4-slot workers and we report the search space size for various compute thresholds α_{cpu} in Table 2. The lower the threshold value $\vec{\alpha}$, the more aggressively branches are pruned during the search. We describe how to automatically configure the pruning factor $\vec{\alpha}$ in Section 5.2.

α_{cpu}	∞	0.5	0.2	0.1	0.05	0.03	0.01
plans	3.25m	2.45m	796k	10k	4465	24	0
#nodes	31m	31m	11m	1.1m	1m	1m	798k
#nodes w/ reordering	31m	30m	11m	535k	519k	294k	28k

Table 2. Number of discovered plans and search tree size under various compute threshold factor α_{cpu}

4.4.2 Search tree exploration reordering.

Placement plans exceeding the threshold are pruned at different layers of the search tree. When an unsatisfactory plan reaches the threshold at a low-level layer rather than near the root, nodes are expanded unnecessarily. To avoid this overhead, we propose reordering the operator exploration sequence in the outer search procedure to achieve early pruning as follows. We prioritize operators with higher resource consumption and explore them at top layers of the tree. Our intuition is that tasks of resource-intensive operators tend to accumulate cost faster and approach the threshold earlier than others. To compute the exploration order, we rank operators based on their cost values (C_{cpu}, C_{io}, and C_{net}) before initiating the search. Reordering can be very effective in reducing the search space, as shown by the results in Table 2. We prove the correctness of the search tree exploration reordering in the full technical report [47].

5 Implementation and deployment

In this section, we describe the implementation of *CAPSys* on Apache Flink and discuss practical deployment issues.

5.1 *CAPSys* overview

Figure 6 shows the system architecture and workflow of query deployment. ① The user submits a query graph and the desired target throughput to the Job Manager. ② Then, *CAPSys* deploys a profiling job to estimate the resource costs for each operator. ③ The scaling controller, which is DS2 [30] in our implementation, decides the parallelism per operator and generates the physical execution graph. ④ The placement controller computes the task placement and ⑤ sends the scheduling information to the Job Manager. ⑥ Finally, the Job Manager deploys the query on the worker cluster.

***CAPSys* scheduler.** We have implemented a custom Flink scheduler that ensures tasks are assigned to workers according to a placement plan. The plan is encoded as a mapping of each task to a location, specified by a Task Manager IP address. We extended Flink's `ResourceProfile` class with a new attribute that specifies the scheduling location for each task, and implemented a custom `SlotMatchingStrategy` to enforce matching tasks to their specified Task Manager.

Metrics collector. The metrics collector component is responsible for recording Flink and system metrics required by the profiling phase and DS2. Metrics are either reported periodically from the Task Managers during runtime or are derived. Specifically, we collect useful time per task, observed and true input and output rates [30], selectivity statistics, and CPU utilization of each worker. The scaling and placement controllers pull their corresponding metrics on demand when they need to make a decision.

Cost profiling. To profile the resource requirements of a query, we deploy tasks of each operator on a separate Task Manager and monitor its behavior for a configurable profiling duration. For each operator, we record (i) the compute cost, as the CPU utilization of the Task Manager where it is deployed, (ii) the state access cost, as the sum of uncompressed bytes read from and written to the RocksDB state backend, and (iii) the network cost, as the number of bytes the operator emits per second. During the profiling phase, we calculate each operator's cost value per record for each dimension, by dividing its respective metric by its observed output rate. At the end of the profiling phase, we store these metrics on disk. As a result, profiling is only run once and does not need to be repeated on reconfiguration. On reconfiguration, we calculate the cost $C_{cpu}(t), C_{io}(t), C_{net}(t)$ of each task $t \in V_{dsp}$ by multiplying its target rate and its corresponding unit cost and provide these values to *CAPS*. If workload characteristics change over time, we could use our current infrastructure to have the Metrics Collector periodically feed metrics to DS2 and CAPS, to support online profiling. We leave this to future work.

Placement controller. The placement controller receives the physical graph from DS2 and uses the profiling metrics to calculate the compute, state, and network costs per operator (c.f.§ 4.2). It then invokes the *CAPS* algorithm to calculate a placement plan that satisfies pruning thresholds. Threshold values can either be provided by the user or automatically set by the auto-tuning process we describe in Section 5.2. *CAPS* parallelizes the search by leveraging a configurable thread pool. Each thread is initially assigned to a random partition of the search space and can subsequently dynamically offload work to other threads, if they become available. Threads cache any satisfactory plan they identify locally. When the search space has been fully explored, threads merge their results and return the pareto-optimal solution.

5.2 Practical considerations

Threshold auto-tuning. Threshold-based pruning (§ 4.4.1) requires users to set the factor $\vec{\alpha}$. However, identifying a good $\vec{\alpha}$ value can be challenging. Ideally, we desire the *minimum* feasible threshold that effectively reduces the search space and returns the most resource-balanced placement plan. To this end, we introduce an *auto-tuning* procedure that can automatically identify this threshold factor.

The auto-tuning process takes as input the set of available resources and the physical query graph and consists of two phases. In the first phase, we identify the minimum feasible threshold for each dimension (i.e., compute, state, network) when the other dimensions are disabled. For each dimension, we start from the tightest possible bound (corresponding to a perfectly-balanced placement) and gradually relax it until we find a valid plan for our deployment. However, the existence of feasible plans is not guaranteed when all lower bounds are jointly applied. In the second phase of auto-tuning, we find the minimum feasible threshold vector by collectively relaxing all threshold values. For both phases, we define configurable factors that decide the threshold relaxation step at each iteration. The higher the relaxation factor, the more aggressively thresholds are increased towards a valid plan. In our experiments, we set the relaxation factors for both phases to 1.1. Finally, users can set a timeout value that allows exiting the search early, to avoid long execution times for infeasible configurations.

Since the auto-tuning results depend only on the query graph and the available resources, we can pre-compute thresholds for various possible scaling scenarios (combinations of operator parallelism settings) *offline* and in parallel. The results can be used to select the pre-calculated thresholds when scaling is triggered at runtime.

Addressing data skew. *CAPS* considers all tasks of the same operator identical with regards to resource consumption. However, in the presence of data skew, one or more tasks of an operator may experience higher input rate than others, and hence, have higher resource demands. While skew mitigation is out of the scope of this paper, the *CAPSys* design is compatible with and could be easily integrated with existing solutions, such as custom partitioning [24, 38]. Such partitioning techniques could be used to organize tasks of an operator into *placement groups* with equal resource demand. Then, each task group can be explored as an individual outer layer in the *CAPS* algorithm. Interestingly, during our evaluation, we found that *CAPSys* already improves query

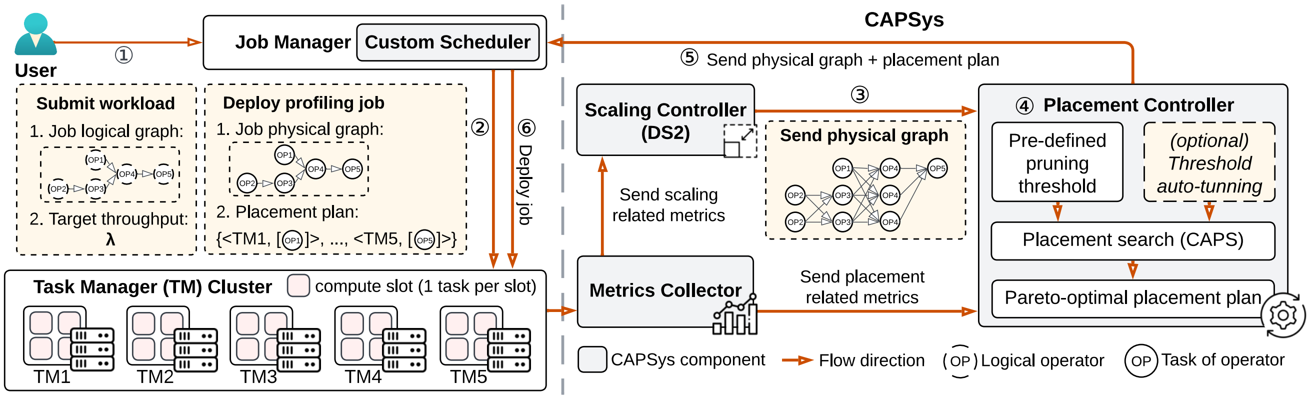

Figure 6. *CAPSys* system overview and workflow of query deployment.

performance in the presence of skew, compared to the baseline strategies. We report these results in the full technical report [47], but we leave a deeper investigation as future work.

6 Experimental evaluation

Our experimental evaluation is structured in four parts. In § 6.2, we compare *CAPSys* with the two baseline placement strategies of Apache Flink. We show results for two settings: (i) an experiment where each query is deployed in isolation and (ii) a large-scale multi-tenant experiment where all queries are deployed on the same cluster concurrently. Our results demonstrate that queries deployed with *CAPSys* consistently achieve the highest throughput and lowest backpressure, while also delivering stable performance across multiple runs. In § 6.3, we compare *CAPSys* with the state-of-art ODRP algorithm and report orders of magnitude improvement in computing the task placement and up to 6× higher throughput. In § 6.4, we evaluate *CAPSys* under variable workloads and show it enhances the accuracy and convergence of the DS2 auto-scaling controller. Finally, in § 6.5, we show *CAPS* is practical for online settings and can identify satisfactory placement plans within seconds, even for deployments with thousands of tasks.

6.1 Workload and experimental setting

Queries. We use six queries with diverse characteristics and complexity. **Q1-sliding**, **Q2-join**, and **Q3-inf** are from the motivation study in Section 3.1. **Q4-join** contains a stateful incremental join operator, **Q5-aggregate** includes a stateful join operator and a process function operator, and **Q6-session** has a session window operator that can potentially accumulate large state. Among the six queries, **Q1-sliding**, **Q2-join**, **Q4-join**, **Q5-aggregate**, and **Q6-session** are representative queries selected from the Nexmark benchmarking suite of Apache Beam [10, 45], corresponding to Nexmark queries Q5, Q8, Q3, Q6, and Q11, respectively. The remaining queries from the Nexmark benchmark are excluded in our evaluation since they either consist of a single stateless operator whose placement is trivial, or have equivalent logical graphs compared to the selected queries. Both *CAPSys* and baseline strategies use Flink `1.16.2`.

Operator chaining and slot sharing. In the following experiments, we disable Flink's operator chaining [6], according to DS2's default configuration [31]. Specifically, we only allow chaining between the source and timestamp operators, but we separate them from the rest of the query, as the source is responsible for generating events and has different resource requirements than downstream operators. Another reason for this setting is to stress the system, as Flink's default chaining would collapse most of the Nexmark queries to trivial dataflow graphs with a single operator. Nevertheless, CAPS works as-is with chaining enabled. It considers any chain as a single operator during profiling and when exploring the search space.

Slot sharing allows multiple tasks to be deployed in a single compute slot [5]. To keep the formulation clean and intuitive, the *CAPS* model in Section 4 does not consider slot sharing and we disable slot sharing in our experiments. We emphasize that the placement problem with slot sharing enabled is equivalent to one with more slots per worker but slot sharing disabled. This is because slots do not enforce resource isolation. Colocating two tasks in the same slot is equivalent to separating them into two slots on the same worker from a resource contention perspective, as each task is allocated a dedicated thread regardless of slot sharing.

6.2 Comparison with Flink strategies

We compare the performance that queries can achieve when executed on *CAPSys* in contrast to Flink's `default` and `evenly` policies (c.f. § 2.2). For all queries, we deploy a Flink cluster on AWS ec2 `m5d.2xlarge` instances. Each worker has 4 cores, 8vCPUs, 32GB memory, and a 300GB SSD disk. The default network bandwidth within the cluster is 10 Gbps.

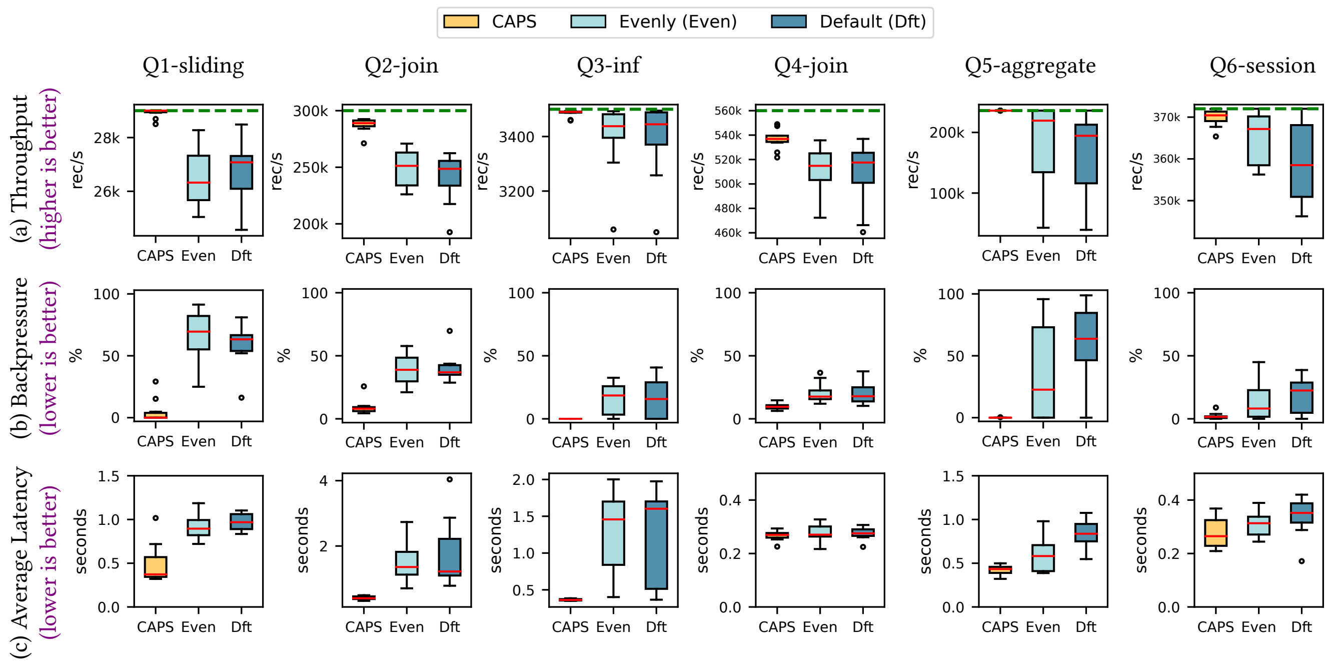

Figure 7. Individual query performance with different placement strategies when each query is deployed in isolation. Queries on *CAPSys* consistently achieve higher throughput and lower backpressure than queries deployed with Flink's strategies. Further, *CAPSys* provides stable performance, while Flink's randomized strategies exhibit significant variance across runs.

We use 4 8-slot workers for single-query experiments (§ 6.2.1) and 18 workers in the multi-tenant setting (§ 6.2.2). *CAPSys* cost thresholds are set with auto-tuning (c.f. § 5.2). To ensure realistic experimental conditions and allow stateful operators to accumulate state, we set the profiling duration to 20*min*. We collect metrics for 10*min* after a warm-up period of 6*min*. We repeat each experiment 10 times and summarize the results in a box plot. We report average throughput, backpressure at the source operator, and average latency.

6.2.1 Impact on individual query performance.

In this experiment, we evaluate the effect of task placement on query performance when each query is deployed in isolation. Figure 7 shows box plot results of 10 runs for each placement strategy. The green dashed lines indicate the target input rate. The results show that *CAPS* outperforms resource-unaware strategies, achieving higher throughput, lower backpressure, and lower latency for all queries. Further, we observe that *CAPS* provides more *stable* performance, as opposed to the baselines whose random placement decisions result in high variance across runs. For the simplest query, **Q1-sliding**, *CAPSys* provides 1.18× higher throughput and 11.8× lower backpressure, on average, compared to the `default` strategy. The improvements are more significant for the complex **Q5-aggregate** query, which consists of two sources, a join, and an aggregation operator. In this case, *CAPSys* achieves up to 6× and 5.5× higher throughput compared to the `default` and `evenly` strategies, respectively. *CAPSys* reduces backpressure by 84% and latency by 48%, on average, across queries.

6.2.2 Multi-tenant experiment.

In this experiment, we evaluate *CAPSys*' scalability to deployments with hundreds of tasks and its ability to perform placement on a multi-query workload. To this end, we deploy all six queries on the same 18-worker cluster concurrently. *CAPSys* views the entire query workload as a single dataflow graph and optimizes task placement globally, taking resource contention across queries into account. On the other hand, `default` and `evenly` can only deploy a single query at a time, hence, they are sensitive to the query submission order. To account for this behavior, we repeat the experiment 10 times and randomize the query submission order for every run. Figure 8 shows the results. *CAPSys* is the only policy that achieves the target throughput across all six queries, while maintaining low backpressure and end-to-end latency. On the other hand, `evenly` reaches the target throughput only for **Q2-join** and exhibits high backpressure for all other queries. The `default` strategy performs slightly better, reaching the target for three out of six queries.

6.3 Comparison with ODRP

Here, we compare *CAPSys* with the state-of-the-art ODRP algorithm proposed by Cardellini et al. [14], which can jointly decide the task parallelism and placement of a query. As ODRP can only handle queries with a single source, we use

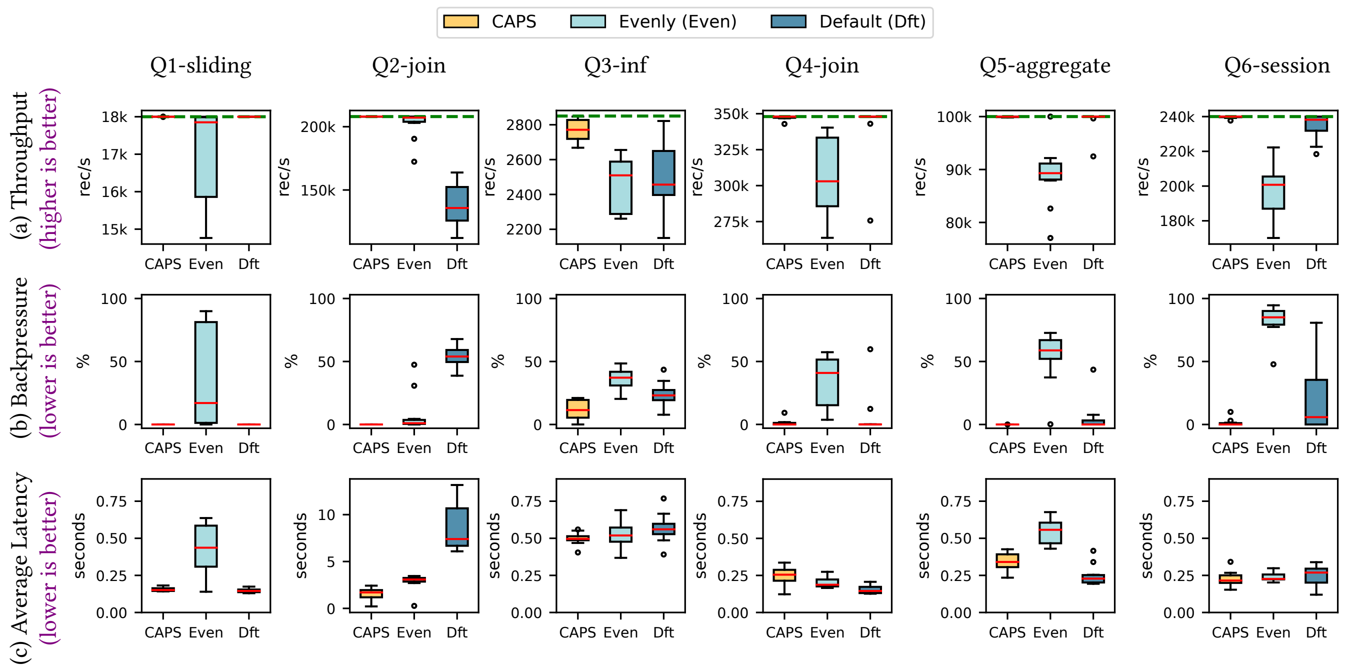

Figure 8. Experimental results for the multi-tenant setting, where all queries are deployed on the same 144-slot cluster concurrently.

Q3-inf in this experiment. We deploy a Flink cluster on four AWS ec2 `c5d.4xlarge` instances and configure each worker with 8 slots. To compare with the ODRP model, we followed the instructions in the corresponding paper and open source repository [7]. To apply ODRP to Flink, we compute an operator's execution time as the inverse of its true processing rate. We set the *lambda* value (data transfer rate) according to the target input rate and operator selectivity. We use the same speed-up rate for all nodes in the resource graph, and the same latency and bandwidth values for all links. We set the number of available resources per node to the number of slots and the required resources per operator to one slot per task. Finally, we assume perfect availability and no failure for all nodes and links.

We report results with three alternative configurations:

- *Default*: This configuration assigns equal weight to all model objectives and includes the improvements described above.
- *Weighted*: This is a hand-tuned configuration that emphasizes throughput and resource efficiency.
- *Latency*: This configuration considers only the latency objective and disables the others.

Table 3 shows results for backpressure, throughput, and latency, using the three ODRP configurations and *CAPSys*. The plans generated with the ODRP-*Default* and ODRP-*Weighted* policies cannot reach the target throughput and exhibit high backpressure. This is because these configurations aim to reduce the resource usage and result in under-provisioned queries. For example, the *Weighted* policy co-located multiple inference tasks on the same worker, causing high contention. On the contrary, the *Latency* policy achieves low latency and throughput close to the target, however, this comes at the cost of high resource demand and backpressure at the source. Finally, we observe that ODRP takes a very long time to generate the physical plans, in the worst case taking 67*min* to converge. Such high decision times make ODRP impractical for online scenarios. Instead, *CAPSys* runs in 0.2*s*, including the threshold auto-tuning phase.

6.4 *CAPSys* under variable workloads

We now evaluate *CAPSys* under variable workloads and demonstrate it can improve the accuracy and convergence of the DS2 auto-scaling controller. We use the **Q3-inf** query and compare the performance of *CAPSys* with that of DS2 when coupled with Flink's `default` and `evenly` policies. Workers are deployed on `r5d.xlarge` AWS instances and are configured with 8 slots. The DS2 activation time is set to 90*s* and the policy interval to 5*s*. We run two experiments to evaluate accuracy and convergence, respectively.

6.4.1 Effect on auto-scaling accuracy. To evaluate the effects of task placement on auto-scaling accuracy, we run a controlled experiment where we vary the input rate every 10*min* and trigger four scaling decisions. To ensure DS2 initially receives good-quality metrics, we manually tune the starting configuration with the optimal task placement and parallelism, so that the query can meet the target rate without being over-provisioned. We set the initial target rate to 720 rec/s and subsequently increase it by 2× in the

	Backpressure	Throughput (rec/s)	Average Latency (s)	Resources (#slots)	Decision Time (s)
CAPSys	0.5%	**4236**	0.292	27	0.2
ODRP-Default	90%	680	0.255	14	1636
ODRP-Weighted	48%	3396	0.268	26	4037
ODRP-Latency	15%	4043	0.157	32	1607

Table 3. Comparison with ODRP policies using **Q3-inf**. *CAPSys* is the only placement controller that can achieve the target throughput. The ODRP configurations either over-provision the query or exhibit poor performance.

	Step #1		**Step #2**		**Step #3**		**Step #4**	
	Throughput	Resources	Throughput	Resources	Throughput	Resources	Throughput	Resources
CAPSys	✓	✓	✓	✓	✓	✓	✓	✓
Default	✗	✓	✗	✗	✗	✗	✓	✓
Evenly	✓	✓	✓	✗	✗	✗	✗	✗

Table 4. Effect of task placement on auto-scaling accuracy. A ✗ in the column *Throughput* indicates the strategy could not reach the target rate, while a ✗ in the *Resources* column indicates the strategy over-provisioned the query.

first two steps and then decrease it by 2× in the next two. We trigger a DS2 scaling action after each rate change and record the average throughput and the number of resources provisioned by DS2. Table 4 shows the results. We put a ✓ in the *Throughput* column if the policy met the target rate and in the *Resources* column if the policy accurately calculated the required resources. Correspondingly, a ✗ in the *Throughput* column indicates throughput below target, and one in the *Resources* column indicates over-provisioning, where the auto-scaler indicated operator parallelism higher than the minimum required to sustain the target rate.

We observe that *CAPSys* can always reach the target throughput and avoids over-shooting in all cases. On the contrary, when task placement is performed by Flink's `default` and `evenly` policies, the behavior of DS2 degrades. Starting from a good initial placement gives both strategies an advantage in the first step, as the elasticity model is informed by accurate metrics. However, subsequent sub-optimal placements impact the controller's accuracy and cause both strategies to over-provision. We further see that the inherent randomness in these strategies causes DS2 to become unstable.

6.4.2 Effect on auto-scaling convergence.

To evaluate the effects of task placement on auto-scaling convergence, we run an experiment under variable workload and let DS2 decide when a scaling decision is necessary. We initially set the parallelism of all operators to 1 and let each placement strategy select its own initial plan. We periodically vary the input rate between a high and a low value every 20*min* and record the number of scaling decisions and the amount of resources that DS2 predicts.

Figure 9 shows the results over the experiment timeline. We plot the observed throughput and the number of resources (tasks) that the query occupies over time. The auto-scaling decisions are marked with gray vertical dashed lines and are annotated with numbers, while the target input rate is shown with a green solid line.

Overall, we see that *CAPSys* improves DS2's convergence behavior and effectively avoids oscillations during both scale-up and scale-down actions. Most of the time, *CAPSys* converges to a stable configuration within a single step after a rate change and is always successful in achieving the target throughput without over-provisioning. In contrast, when DS2 is coupled with the `default` and `evenly` policies, it takes longer to converge, incurring up to eight additional scaling decisions in this experiment. Specifically, the first time the rate increases, the `default` policy makes two scaling decisions before reaching the target rate, while `evenly` causes DS2 to oscillate. It triggers three scaling decisions and fails to meet the target rate in the next 20*min*. When the rate drops, we notice oscillation caused by the inherent randomness in the `default` policy. The poor task placement informs DS2 with inaccurate metrics and prevents it from reaching the target throughput. This unstable behavior continues for `default` and `evenly` throughout the duration of the experiment. Finally, we note that even when `default` and `evenly` manage to meet the target rate, they often induce over-provisioning, occupying up to four underutilized additional slots.

6.5 *CAPS* performance and scalability

We now investigate how various parameters affect the performance and scalability of the *CAPSys*. We demonstrate that *CAPSys* can quickly find good placement plans for queries with over a thousand tasks and is suitable for dynamic settings where frequent reconfigurations may occur. We run experiments with **Q2-join**, a workload with both compute-intensive and state-intensive tasks. We use a Cloudlab [17] `c220g2` instance, with 20 cores and 160GB of memory. *CAPSys* uses 20 threads for parallel execution.

6.5.1 Scalability of placement search.

In the first experiment, we evaluate how the *CAPSys* performance scales with the problem size for various manually-configured threshold values. We increase the number of tasks (slots) from

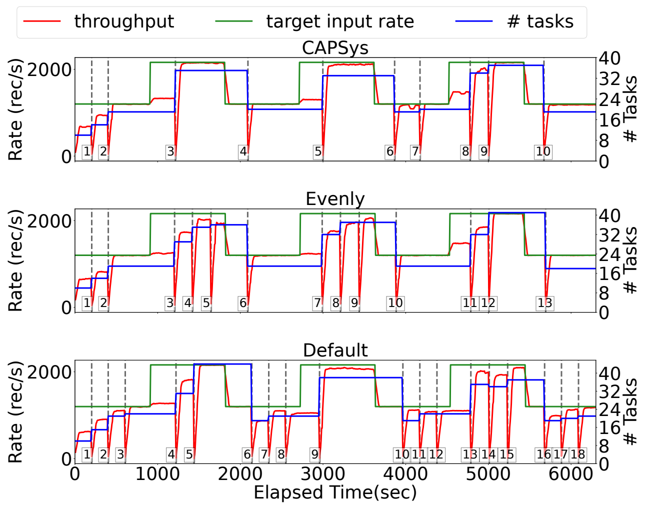

Figure 9. Effect of task placement strategies on auto-scaling convergence.

16 to 256 and measure the time *CAPSys* requires to search the space of placement plans, until it identifies the first one that satisfies the threshold constraints. We repeat the experiment for the following three threshold configurations: $\vec{\alpha}_1$ (α_{cpu}: 0.08, α_{io}: 0.15, α_{net}: 0.6), $\vec{\alpha}_2$ (α_{cpu}: 0.15, α_{io}: 0.25, α_{net}: 0.8), and $\vec{\alpha}_3$ (α_{cpu}: 0.25, α_{io}: 0.3, α_{net}: 0.9). $\vec{\alpha}_1$, $\vec{\alpha}_2$, and $\vec{\alpha}_3$ are empirically-obtained thresholds that prune the search space with different granularity.

The results in Figure 10-a show that *CAPSys* can efficiently find satisfactory placement plans within tens of milliseconds, even for queries with hundreds of parallel tasks. For very tight threshold values, the runtime increases slightly as the number of tasks grows. This is expected, as discovering a placement plan that satisfies the constraint gets harder. In all cases, *CAPSys* returns a task assignment in up to 100*ms*.

6.5.2 Performance of threshold auto-tuning.

We also evaluate the performance of the threshold auto-tuning component (c.f.§ 5.2). We vary the number of workers from 8 to 16 and the number of slots per worker from 4 to 64. The timeout period is set to 5*s*. We execute the threshold auto-tuning algorithm for all combinations of the above values, where the total number of tasks grows from 32 to 1024. Figure 10-b shows the total execution time for each configuration.

We observe that auto-tuning is very efficient for small and medium-size deployments. For example, finding threshold bounds for 64 tasks with a configuration of 4 workers and 16 slots per worker takes only 1.16*s*. As the number of tasks grows, the runtime increases. For 1024 tasks deployed on 16 workers with 64 slots each, the process takes 125.08*s*. We believe this is acceptable, as threshold auto-tuning can be performed offline, as we describe in Section 5.2. This result could be further improved by exploiting the accuracy-performance tradeoff or by running auto-tuning on a machine with higher

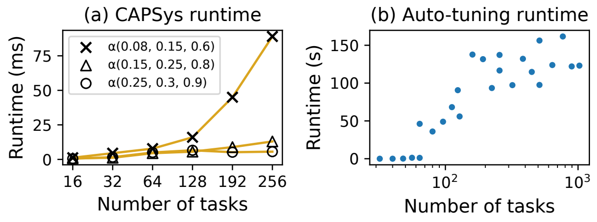

Figure 10. Scalability results for the *CAPSys* placement search and threshold auto-tuning.

parallelism. Another approach would be to first partition the dataflow graph and apply *CAPS* per partition. We plan to explore this direction as future work.

7 Related Work

We have already discussed the closely related work in Section 2.2. Here, we highlight important works that consider task scheduling in slightly different settings than ours.

Stream processing in WAN and edge environments. Task placement has been an important issue in the design of wide-area analytics systems and edge stream processing frameworks. In such environments, schedulers need to address various challenges, such as heterogeneous and dynamic network conditions, transmission delays, and congestion. Examples include WASP [29], which uses an ILP-based solution to minimize network delays between computation stages, DROPLET [18], which follows a dynamic programming approach coupled with heuristics to minimize job completion time, and SWAN [44], which proposes a heuristic model to balance the number of tasks across nodes, while also considering each node's available bandwidth. SBON [41] and DART [35] are other noteworthy systems that organize workers into overlay networks and propose decentralized solutions to the placement problem. Though in this paper we focus on datacenter environments, where propagation delays are negligible, extending the *CAPS* cost model with additional objectives is an exciting direction for future work.

Scheduling long-running workloads. Cluster scheduling of machine learning (ML) workloads and other long-running jobs [11, 23, 48, 52] also bears similarity with task placement in SPSs. In contrast to ML training and batch processing jobs, which consume finite data and whose execution can be optimized offline, streaming workloads are dynamic and often unpredictable. As a result, streaming task schedulers need to be capable of operating online with low latency. In the future, we plan to investigate integrating *CAPSys* with cluster schedulers like Medea [23], Cilantro [11], and control planes for SPSs, like Chi [37], as their approaches are complementary to ours.

8 Conclusion

We presented *CAPSys*, an adaptive resource controller for stream processing systems that considers auto-scaling and task placement in concert. The core of *CAPSys* is the contention-aware placement search strategy that leverages empirically-verified heuristics and applies various optimizations to prune the vast search space of alternative task assignments. Our results show that *CAPSys* computes task placement in orders of magnitude lower time compared to the ODRP algorithm [14] and achieves up to 6× higher throughput compared to Flink task placement strategies, while it also enhances the effectiveness of the DS2 auto-scaling controller.

9 Acknowledgements

We thank the anonymous EuroSys reviewers for their insightful comments and our shepherd, Etienne Riviere, for his guidance in improving the paper. This work was partially supported by a Google DAPA award and a Red Hat Collaboratory Research Incubation Award (ID:2024-01-RH10). This work has been partly supported by National Science Foundation Award CNS-1908677.

References

[1] Apache Flink. https://flink.apache.org/. Last access: January 2024.

[2] Apache Flink configuration: Evenly spread out slots. https://nightlies.apache.org/flink/flink-docs-release-1.16/docs/deployment/config/#cluster-evenly-spread-out-slots. Last access: January 2024.

[3] Apache Storm. https://storm.apache.org/. Last access: January 2024.

[4] Fine-Grained Resource Management in Apache Flink. https://nightlies.apache.org/flink/flink-docs-stable/docs/deployment/finegrained_resource/. Last access: January 2024.

[5] Flink Architecture: Task Slots and Resources. https://nightlies.apache.org/flink/flink-docs-release-1.16/docs/concepts/flink-architecture/#task-slots-and-resources. Last access: September 2024.

[6] Flink Architecture: Tasks and Operator Chains. https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/flink-architecture/#tasks-and-operator-chains. Last access: September 2024.

[7] Optimal dsp placement and replication, 2023. Last access: Oct 2023.

[8] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 2015.

[9] Esmail Asyabi, Yuanli Wang, John Liagouris, Vasiliki Kalavri, and Azer Bestavros. A new benchmark harness for systematic and robust evaluation of streaming state stores. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 559–574, New York, NY, USA, 2022. Association for Computing Machinery.

[10] Apache Beam. Nexmark Benchmark Suite, 2022. Last access: March 2022.

[11] Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Cilantro: Performance-Aware resource allocation for general objectives via online feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 623–643, Boston, MA, July 2023. USENIX Association.

[12] Muhammad Bilal and Marco Canini. Towards automatic parameter tuning of stream processing systems. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24 - 27, 2017*, pages 189–200, 2017.

[13] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*, DEBS '16, page 69–80, New York, NY, USA, 2016. Association for Computing Machinery.

[14] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator replication and placement for distributed stream processing systems. *SIGMETRICS Perform. Eval. Rev.*, 44(4):11–22, may 2017.

[15] Xenofon Chatziliadis, Eleni Tzirita Zacharatou, Alphan Eracar, Steffen Zeuch, and Volker Markl. Efficient placement of decomposable aggregation functions for stream processing over large geo-distributed topologies. *Proc. VLDB Endow.*, 17(6):1501–1514, may 2024.

[16] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 16:1–16:13, New York, NY, USA, 2014. ACM.

[17] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[18] Tarek Elgamal, Atul Sandur, Phuong Nguyen, Klara Nahrstedt, and Gul Agha. Droplet: Distributed operator placement for iot applications spanning edge and cloud resources. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 1–8, 2018.

[19] Apache Flink. Network memory tuning guide. https://nightlies.apache.org/flink/flink-docs-release-1.16/docs/deployment/memory/network_mem_tuning/, 2023.

[20] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: Self-regulating stream processing in heron. *PVLDB*, 10(12):1825–1836, August 2017.

[21] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. A survey on the evolution of stream processing systems. *The VLDB Journal*, pages 1–35, 2023.

[22] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. DRS: auto-scaling for real-time stream analytics. *IEEE/ACM Trans. Netw.*, 25(6):3338–3352, 2017.

[23] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: Scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[24] Bugra Gedik. Partitioning functions for stateful data parallelism in stream processing. *VLDB J.*, 23(4):517–539, 2014.

[25] Herodotos Herodotou, Lambros Odysseos, Yuxing Chen, and Jiaheng Lu. Automatic performance tuning for distributed data stream processing systems. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 3194–3197. IEEE, 2022.

[26] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for {Fine-Grained} resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[27] Sonia Horchidan, Po Hao Chen, Emmanouil Kritharakis, Paris Carbone, and Vasiliki Kalavri. Crayfish: Navigating the labyrinth of machine learning inference in stream processing systems. In *27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, Mar 25 2024-Mar 28 2024*, pages 676–689, 2024.

[28] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential

building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.

[29] Albert Jonathan, Abhishek Chandra, and Jon Weissman. Wasp: Wide-area adaptive stream processing. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 221–235, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, 2018.

[31] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. https://github.com/strymon-system/ds2/blob/master/flink-examples/src/main/java/ch/ethz/systems/strymon/ds2/flink/nexmark/queries/Query1.java#L49, October 2018. Last access: May 2024.

[32] Faria Kalim, Le Xu, Sharanya Bathey, Richa Meherwal, and Indranil Gupta. Henge: Intent-driven multi-tenant stream processing. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 249–262, New York, NY, USA, 2018. Association for Computing Machinery.

[33] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th international conference on data engineering (ICDE)*, pages 1507–1518. IEEE, 2018.

[34] Nikos R. Katsipoulakis, Alexandros Labrinidis, and Panos K. Chrysanthis. A holistic view of stream partitioning costs. *Proc. VLDB Endow.*, 10(11):1286–1297, August 2017.

[35] Pinchao Liu, Dilma Da Silva, and Liting Hu. Dart: A scalable and adaptive edge stream processing engine. In *USENIX Annual Technical Conference*, 2021.

[36] Yuan Liu, Xuanhua Shi, and Hai Jin. Runtime-aware adaptive scheduling in stream processing. *Concurrency Computat.: Pract. Exper*, 28:3830–3843, 2016.

[37] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. Chi: a scalable and programmable control plane for distributed stream processing systems. *Proc. VLDB Endow.*, 11(10):1303–1316, jun 2018.

[38] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, Nicolas Kourtellis, and Marco Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 589–600, 2015.

[39] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 69–84, New York, NY, USA, 2013. Association for Computing Machinery.

[40] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, page 149–161, New York, NY, USA, 2015. Association for Computing Machinery.

[41] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49, 2006.

[42] Nicoló Rivetti, Leonardo Querzoni, Emmanuelle Anceaume, Yann Busnel, and Bruno Sericola. Efficient key grouping for near-optimal load balancing in stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 80–91, 2015.

[43] Guillaume Rosinosky, Donatien Schmitz, and Etienne Rivière. Streambed: capacity planning for stream processing. *arXiv preprint arXiv:2309.03377*, 2023.

[44] Won Wook Song, Myeongjae Jeon, and Byung-Gon Chun. Swan: Wan-aware stream processing on geographically-distributed clusters. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '22, page 78–84, New York, NY, USA, 2022. Association for Computing Machinery.

[45] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. Nexmark—a benchmark for queries over data streams. Technical report, OGI School of Science & Engineering at OHSU, 2002.

[46] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–16, 2013.

[47] Yuanli Wang, Lei Huang, Zikun Wang, Vasiliki Kalavri, and Ibrahim Matta. Capsys: Contention-aware task placement for data stream processing. Technical report, BUCS-2024-001, Computer Science Department, Boston University, https://hdl.handle.net/2144/49285, 2024.

[48] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 595–610, USA, 2018. USENIX Association.

[49] Le Xu, Boyang Peng, and Indranil Gupta. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *2016 IEEE International Conference on Cloud Engineering, IC2E 2016, Berlin, Germany, April 4-8, 2016*, pages 22–31, 2016.

[50] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. Move fast and meet deadlines: Fine-grained real-time stream processing with cameo. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 389–405, 2021.

[51] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. Awstream: Adaptive wide-area streaming analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 236–252, New York, NY, USA, 2018. Association for Computing Machinery.

[52] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

A Artifact Appendix

A.1 Abstract

CAPSys is an adaptive resource controller for dataflow stream processors, that considers auto-scaling and task placement in concert. *CAPSys* relies on *Contention-Aware Placement Search (CAPS)*, a new placement strategy that ensures compute-intensive, I/O-intensive, and network-intensive tasks are balanced across available resources. We integrate *CAPSys* with Apache Flink and show that it consistently achieves higher throughput and lower backpressure than Flink's strategies, while it also improves the convergence of the DS2 auto-scaling controller under variable workloads. When compared with the state-of-the-art ODRP placement strategy, *CAPSys* computes the task placement in orders of magnitude lower time and achieves up to 6× higher throughput. The experiments require both AWS and Cloudlab resources to support the above claims.

A.2 Description & Requirements

A.2.1 How to access. *CAPSys* is accessible at the following GitHub link:
https://github.com/CASP-Systems-BU/CAPSys
or using the following DOI: `10.5281/zenodo.13717642` (https://zenodo.org/doi/10.5281/zenodo.13717642)

A.2.2 Hardware dependencies. We run experiments with Flink deployments on three types of AWS EC2 instances: `m5d.2xlarge` (8 vCPUs, 32 GB memory, 300GB SSD disk), `c5d.4xlarge` (16 vCPUs, 32 GB memory, 400GB SSD disk), and `r5d.xlarge` (4 vCPUs, 32 GB memory, 150GB SSD disk). We run *CAPSys* performance and scalability experiments on the Cloudlab [17] `c220g2` instance (20 cores, 160GB memory).

A.2.3 Software dependencies. We run the experiments on Ubuntu `22.04` and Apache Flink `1.16.2`. We provide scripts for installing Flink and all dependency libraries.

A.2.4 Benchmarks. We use six queries with diverse characteristics and complexity: **Q1-sliding**, **Q2-join**, and **Q3-inf** from the motivation study in Section 3.1, and queries Q3, Q6, Q11 from the Nexmark benchmark [45], referred to in the following as **Q4-join**, **Q5-aggregate**, and **Q6-session**.

All queries are defined in https://github.com/CASP-Systems-BU/CAPSys/tree/main/queries.

A.3 Set-up

We provide instructions for setting up the evaluation environment on AWS under https://github.com/CASP-Systems-BU/CAPSys/tree/main/scripts/aws/README.md. We also provide SSH access to our pre-configured AWS clusters for simplicity.

For Cloudlab experiments, we provide instructions and the configuration profile under https://github.com/CASP-Systems-BU/CAPSys/blob/main/scripts/README.md#preparation-3.

A.4 Evaluation workflow

A.4.1 Major Claims.

- (C1): *CAPSys* outperforms Flink's `default` and `evenly` strategies. This is proven by the experiments (E1) described in Section 6.2 where we compare *CAPSys* with Flink's `default` and `evenly` policies on all six queries described in Section A.2.4. We provide instructions on how to reproduce Figure 7, as described in Section 6.2.1, which is sufficient to support the claim. Figure 8, described in Section 6.2.2, evaluates the same workload on a larger problem size.
- (C2): We compare *CAPSys* with the state-of-the-art ODRP algorithm proposed by Cardellini et al. [14], which can jointly decide the task parallelism and placement of a query. Comparing with *CAPSys*, the plans generated with different ODRP policies either cannot reach the target throughput and exhibit high backpressure, or show higher resource demand. This is proven by the experiment (E2), described in Section 6.3, whose results are reported in Table 3.
- (C3): Under variable workloads, *CAPSys* can improve the accuracy and convergence of the DS2 auto-scaling controller. This is proven by the experiment (E3) described in Section 6.4. Reproducing results in Table 4 is sufficient to support the claim.
- (C4): *CAPS* and *auto-tuning* can quickly and effectively find satisfying placement plans to support dynamic settings where frequent reconfigurations may occur. This is proven by the experiments (E4), described in Section 6.5, where we measure the runtime of *CAPS* and *auto-tuning* on varying problem sizes. For this part, we reproduce results reported in Figure 10.

A.4.2 Experiments.

Experiment (E1): [Comparison with Flink strategies] [30 human-minutes + 1 compute-hour]: For all queries described in A.2.4, we compare the *CAPSys* performance with Flink's `default` and `evenly` policies. Each experiment is repeated 10 times for each policy to capture the randomness inherent in the baseline approaches. Please see https://github.com/CASP-Systems-BU/CAPSys/blob/main/scripts/README.md#experiment-e1 for a detailed description of performing experiment E1.

Experiment (E2): [Comparison with ODRP] [30 human-minutes + 1 compute-hour]: We use query **Q3-inf** to compare *CAPSys* with the state-of-the-art ODRP algorithm proposed by Cardellini et al. [14]. Please see https://github.com/CASP-Systems-BU/CAPSys/blob/main/scripts/README.m d#experiment-e2 for a detailed description of performing experiment E2.

Experiment (E3): [*CAPSys* under variable workloads] [30 human-minutes + 1 compute-hour]: We evaluate *CAPSys* under variable workloads and demonstrate it can improve the accuracy and convergence of the DS2 auto-scaling controller. We use query **Q3-inf** to compare the performance of *CAPSys* with that of DS2 when coupled with Flink's `default` and `evenly` policies. Please see https://github.com/CASP-Systems-BU/CAPSys/blob/main/scripts/README.md#experiment-e3 for a detailed description of performing experiment E3.

Experiment (E4): [*CAPSys* performance and scalability] [30 human-minutes + 10 compute-minutes]: We explore the runtime of *CAPS* and *auto-tuning* on varying problem sizes. We run experiments with **Q2-join**, a workload with both compute-intensive and state-intensive tasks. Please see https://github.com/CASP-Systems-BU/CAPSys/blob/main/scripts/README.md#experiment-e4 for a detailed description of performing experiment E4.