Zero-Shot Cost Models for Distributed Stream Processing

Roman Heinrich
DHBW Mannheim

Manisha Luthra
Technical University of Darmstadt

Harald Kornmayer
DHBW Mannheim

Carsten Binnig
Technical University of Darmstadt & DFKI

ABSTRACT

This paper proposes a learned cost estimation model for Distributed Stream Processing Systems (DSPS) with an aim to provide accurate cost predictions of executing queries. A major premise of this work is that the proposed learned model can generalize to the dynamics of streaming workloads *out-of-the-box*. This means a model once trained can accurately predict performance metrics such as *latency* and *throughput* even if the characteristics of the data and workload or the deployment of operators to hardware changes at runtime. That way, the model can be used to solve tasks such as optimizing the placement of operators to minimize the end-to-end latency of a streaming query or maximize its throughput even under varying conditions. Our evaluation on a well-known DSPS, Apache Storm, shows that the model can predict accurately for unseen workloads and queries while generalizing across real-world benchmarks.

CCS CONCEPTS

• **Information systems** → **Stream management**.

KEYWORDS

Stream processing, Cost Models, Zero-shot learning

ACM Reference Format:
Roman Heinrich, Manisha Luthra, Harald Kornmayer, and Carsten Binnig. 2022. Zero-Shot Cost Models for Distributed Stream Processing. In *The 16th ACM International Conference on Distributed and Event-based Systems (DEBS '22), June 27–30, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3524860.3539639

1 INTRODUCTION

Motivation. Distributed Stream Processing System (DSPS) correlates and analyzes data streams from multiple data sources to derive higher-level information for a wide range of applications. At the core, a DSPS takes a continuous query that represents one or more tasks for the given application and processes the query in a distributed way using multiple hardware resources (cf. Figure 1). While doing so, for many applications, a DSPS has to provide guarantees in terms of one or more quality-of-service (QoS) cost metrics such as latency and throughput. For instance, in Facebook, queries for click stream analytics have to provide a very high throughput to process input event streams at around 9 GiB/s and at the same time also ensure low latency [1].

DEBS '22, June 27–30, 2022, Copenhagen, Denmark

ACM ISBN 978-1-4503-9308-9/22/06…$15.00
https://doi.org/10.1145/3524860.3539639

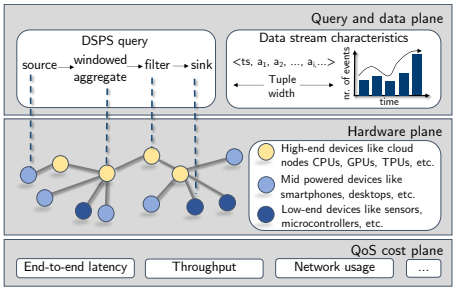

Figure 1: A DSPS has to provide guarantees in terms of one or more quality-of-service (QoS) cost metrics such as latency and throughput. The challenge is that DSPS serve a wide range of workloads on potentially diverse hardware, which makes the cost estimation harder.

Typically, a DSPS provides QoS guarantees using optimization mechanisms such as *operator placement* that usually monitors the costs to decide on the mapping of operators to hardware as shown in Figure 1 [2]. Moreover, frequent reconfigurations of the operator placement are required based on the observed changes of the workload (e.g., during peaks in the input stream). Further, to provide QoS guarantees, a DSPS uses multiple other techniques such as *elasticity* or adaption of *operator parallelism* [3]. [4] provides an extensive list of other DSPS optimization mechanisms.

However, many of these optimization mechanisms classically rely on heuristics [2, 5] or other analytical approaches like queueing theory [6] that approximates the effects of placement decisions, the degree of parallelism, etc., on the actual latency and throughput. As a result, these techniques often make simplifying assumptions on the estimation of the required QoS and thus take non-optimal decisions (e.g., they decide on a non-optimal placement).

More recently, machine learning (ML) has been used in solving these optimization tasks such as to place operators [7, 8], to autoscale DSPS [3] or even to estimate resource utilization [9], which has shown promising initial results [10]. However, the existing ML-based approaches cannot be used out-of-the-box as these often are specialized either for the given optimization task (e.g., elasticity) or support only a restricted set of workloads and hardware resources.

While, in this paper, we argue that learned cost models should be used for DSPS to allow better decisions for a broad set of optimization tasks. For example, the cost model can be used in advance before actually placing operators to different resources to know the expected throughput and latency. The same observations hold for other optimization tasks, such as determining the degree of parallelism for an operator. Recently, learned cost models have also been explored in database systems to estimate costs such as query runtime and thus solve problems related to query optimization or scheduling. Yet these learned models are highly *workload-driven* [11] – meaning that they make strong assumptions about what data and queries as well as hardware should be supported.

Thus, a core problem of existing learned cost models is that they can hardly generalize to the changes in the workload (queries and

data) or even the placement on different hardware platforms at runtime. Consequently, either a separate model must be trained for a combination of workload and hardware or the existing one must be constantly retrained, which not only causes high overhead but also does not allow the DSPS to react instantly to changes.

Contributions. In this paper, therefore, we propose a new approach for *learned cost models* for DSPSs based on the concept of *zero-shot learning* [12, 13]. The key idea is to train a model on a broad spectrum of different streaming workloads and hardware resources to enable generalization. As such, a zero-shot cost model for DSPS can provide accurate estimates even under the changes in the streaming workload or for different hardware platforms that are used for placing operators. Moreover, the zero-shot cost model can even adapt to an entirely *new* workload *out-of-the-box*, with queries and data characteristics the model has not seen during the training. Thus, our approach comes as a huge benefit for DSPSs since they are known to be highly dynamic in nature (e.g., fluctuations in input stream). That way, using such a zero-shot cost model, several DSPS optimization tasks can be performed (e.g., operator placement) in a highly accurate manner such that the desired QoS guarantees are satisfied even under dynamics in streaming workload.

A key to enable a zero-shot cost model is a novel model architecture that represents data, queries and hardware as input for our cost model. At the core of our model architecture is a so-called *transferable feature representation* that allows the learned cost model to generalize to a broader set of workloads or even be used across hardware platforms. For instance, to make predictions under changing workloads, the transferable feature representation relies on general information such as *event rate at the source* and other information such as *tuple width* of data streams or *window size* of operators. Another important aspect of the transferable feature representation is that we include so-called *data characteristics* of the queries and data streams as features, such as selectivity of an operator, so that the zero-shot model can learn the runtime behavior of a query.

In summary, this work makes the following contributions:
(i) We discuss the training and inference procedure of the zero-shot model. For training, we provide a broad spectrum of workloads and hardware platforms to a zero-shot cost model for DSPS such that it can learn to generalize to unseen streaming workloads and provide estimates across different hardware platforms.
(ii) We present a new model architecture for learning such a zero-shot cost model for DSPSs that can generalize to unseen workloads by using a transferable feature representation.
(iii) We provide an evaluation of our model on Apache Storm, a well-known DSPS. The results show that our model accurately predicts the performance of DSPS queries on Storm, even for unseen operator and data stream parameters and generalizes across different real-world benchmarks without explicitly training for them.

Outline. In Section 2, we provide an overview of our approach before we explain the details of the model architecture and its feature representation in Section 3. Afterwards, we present the experiments in Section 4, related work in Section 5, and conclude in Section 6.

2 OVERVIEW OF APPROACH

In the following, we provide a high-level overview of our approach. We first start with the idea of how a zero-shot cost model is being trained for DSPS before we then explain its usage at runtime to estimate the cost for guaranteeing QoS.

Training a Zero-Shot Cost Model. The proposed zero-shot cost model learns from previous query executions and their observed costs in a supervised manner. To allow the model to generalize, we generate and train our model with a broad spectrum of queries and streaming data as well as different observed cost metrics (i.e., latency and throughput) that are induced by these queries. To be more precise, to enable a high variety in the training data, we represent standard query structures for DSPS and vary them in terms of complexity and operators properties such as *window size* (cf. Section 4 to see our training range). Similarly, we diversify the input data streams by training for several *event rates*, and we capture many different streaming workloads. Moreover, during training, operators of queries are also deployed on different hardware platforms. While this would seem like a huge effort, it is a one-time training effort in contrast to state-of-the-art learned approaches that need to train a model per streaming workload (data and query). The main idea to enable generalization across streaming workloads and hardware platforms is our model architecture that relies on the aforementioned transferable feature representation of workloads and hardware (cf. Section 3).

Using a Zero-Shot Cost Model. Once a zero-shot model is trained, it can be used at runtime to predict cost metrics for an unseen query across different hardware platforms. In particular, our model can infer costs accurately for an unseen query with an entirely different data distribution of input data stream and extrapolate for unseen operator properties, e.g., window size.

Consequently, we envision zero-shot cost models as a foundation for complex optimization tasks like the operator placement problem. For instance, cost predictions of operator placement on different hardware platforms can be used to find a hardware resource with an objective to satisfy a certain latency constraint or demands on throughput. Moreover, such cost predictions can also serve as a basis for other optimization tasks, such as to determine the right parallelization degree or the number of resources for deployment. While the combination of zero-shot models with these optimization tasks is clearly an interesting direction, in this paper, we focus only on how to enable the cost estimation using zero-shot models.

3 ZERO-SHOT COST MODELS FOR DSPS

In Section 3.1, we first describe the cost metrics and the proposed model architecture. Afterwards, we discuss our transferable feature representation in Section 3.2 and conclude with the training and inference procedure in Section 3.3.

3.1 QoS Metrics and Model Architecture

Our approach draws inspiration from the zero-shot cost models for databases [12] that aims to predict query runtimes across unseen relational databases. However, [12, 13] is not easily extensible for DSPS because the characteristics of databases and their queries differ largely from that of a DSPS, as discussed in the following.

First, different from databases, a query is continuous in a DSPS and is composed of a logical set of *streaming operators* (Ω), i.e., operators that operate on unbounded data streams (D) instead of tables. Streaming queries are composed of multiple operators in a so-called *operator graph* (G). In this graph, each vertex represents

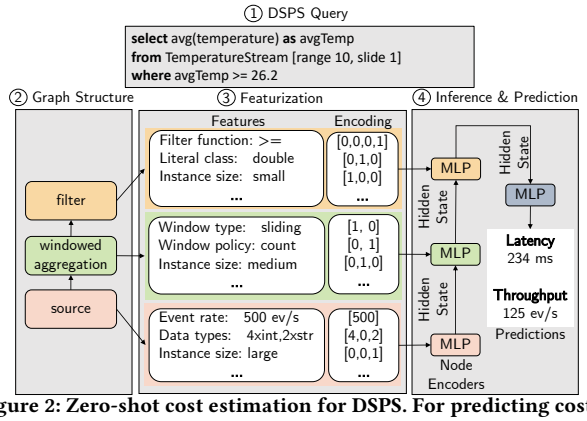

Figure 2: Zero-shot cost estimation for DSPS. For predicting costs a streaming query ① is transformed into a graph structure ② which uses a novel transferable representation ③. For cost predictions ④, we use the graph-based structure where features of every operator are encoded in a separate multi-layer perceptron (MLP) and propagated along the graph using the order of operators in the query. A final MLP predicts throughput and latency.

an operator ($\omega \in \Omega$), and the edge between them represents the data stream D. Hence, the operator graph describes the logical flow of data streams from one or multiple *sources* (data producer) to the *sink* (data consumer). The data stream D that flows through the operator graph represents an unbounded set of tuples.

Other differences from databases are: *(i)* The operator graph in a DSPS contains very different operators, e.g., window operator is typically used to bound the data stream. *(ii)* For DSPS, different QoS metrics need to be predicted by a cost model for continuous queries that varies over the time. For this paper, we consider *latency* and *throughput* for streaming queries in DSPSs that need to be predicted by our learned cost model. In the following, we now discuss the definitions for those metrics, which we use in this paper since several different definitions have been used in the literature [8, 14, 15]. Afterwards, we then explain the details of our model architecture.

QoS Metrics. While throughput is well-defined for DSPS, there is no unique definition of latency [2]. The reason is that there are several stages of an event tuple (production, ingestion and processing) at which it is timestamped and thus different classes of latency exist [15]. In this paper, we only consider the so-called *end-to-end latency* that includes all stages of an event.

Definition 3.1. End-to-end latency: For each output tuple d_O, the end-to-end latency is the interval between the time at which the oldest input event tuple d_I involved in producing the output tuple d_O is generated at the source and the time that d_O arrives at the sink. In the paper, we use the average end-to-end latency of all the output tuples that arrive at the sink for a given query.

Definition 3.2. Throughput: We define the second metric *throughput* for our cost models inline with the literature [14]. For the execution of a given query, throughput is the number of output tuples that arrive at the sink per time unit.

Model Architecture. For realizing zero-shot models for DSPS that can predict these QoS metrics, a core question we ask is, *"what can be learned from a given query and data stream that can be generalized for most of the streaming workloads?"*. We answer this by using a new model architecture that aims to represent streaming queries using a graph representation with so-called transferable features (Section 3.2). The overall approach of the model architecture is explained by using an example as shown in Figure 2. ① Given a DSPS query that computes whether an average temperature from a data stream `TemperatureStream` exceeds a threshold, ② we represent each operator of the query as a *node* in our graph structure. For instance, `source`, `windowed aggregation` and `filter` operators are represented as nodes in the model shown in the example. ③ For each graph node, we use transferable features that describe operator properties (e.g., `filter` predicate), data characteristics (e.g., `selectivity`) and hardware properties (e.g., `instance size`). ④ Once the graph representation is constructed, the model can predict the cost (e.g., throughput and end-to-end latency). The training and inference procedure is explained in detail in Section 3.3. Overall, the generic representation based on graphs allows us to make predictions for different query structures.

Node	Category	Feature	Description
all	hardware	`instance size`	Properties of the hardware
	data	`tuple width in`	Averaged incoming tuple width
	data	`tuple width out`	Outgoing tuple width
source	data	`input event rate`	Event rate emitted by the source
	data	`tuple data type`	Data type for each value in tuple
filter	operator	`filter function`	Comparison function
	operator	`literal data type`	Data type of comparison literal
	data	`selectivity`	see Definition 3.3
join	operator	`join-key data type`	Data type of the join key
	data	`selectivity`	see Definition 3.4
agg.	operator	`agg. function`	Aggregation function
	operator	`group-by data type`	Data type of group-by attribute
	operator	`agg. data type`	Data type of each value to aggregate
	data	`selectivity`	see Definition 3.5
window	operator	`window type`	Shifting strategy (sliding/tumbling)
	operator	`window policy`	Counting mode (count/time-based)
	operator	`window size`	Size of the window
	operator	`slide size`	Size of the sliding interval

Table 1: Transferable features categorized as operator-, data- and hardware-related. The zero-shot model can learn from them and can be applied to different streaming workloads (data and queries).

3.2 Transferable Feature Representation

Query execution costs like latency and throughput depend heavily on the operator parameters, the data characteristics of the streams and the operator placement. To describe these aspects and make them usable for learning, we propose transferable features. For this, we derive *nodes* from the operator graph G (excluding the sink) and collect transferable operator-related, data-related, and hardware-related input features for each of them. By *transferable*, we mean that these features can be applied for any arbitrary streaming workload (data and query). This representation targets commonly used operators for DSPS (e.g. filter, join, aggregation, window). A listing of all nodes with the features we use for our cost model is presented in Table 1 and explained as follows.

Operator-related features. The main idea behind operator-related features is to only include properties as features that describe the operator logic and are transferable for *any* query and DSPS. So, instead of using non-transferable features of operators, such as *literals* of a filter operator (e.g. `tuple temperature = 5`), which might have a very different meaning for different data streams, we use generic features such as `data type` of literal and the `function` of a filter predicate (e.g., an equality predicate or a range predicate).

Data-related features. A major aspect of zero-shot models is that it is agnostic of the underlying data distribution of the streaming workload. Thus, instead of using features like *attribute names* to

encode the semantics of the data, we learn from data characteristics (DCs) such as `tuple width` and `event rate` that can be transferred to any workload. While DCs like `tuple width` are easily derived, other DCs, such as `selectivity` of operators, are harder to derive or are even not well-defined for operators like windowed joins [16]. In the following, we thus first define the selectivity for the streaming operators shown in Table 1:

Definition 3.3. Filter selectivity: The selectivity $sel(\omega_\sigma)$ of a filter operator ω_σ is the ratio of the number of outgoing to incoming tuples in the input stream D, formally as:

$$sel(\omega_\sigma) = \frac{|f_{\omega_\sigma}(D)|}{|D|}, \quad \text{with } 0 \leq sel(\omega_\sigma) \leq 1. \tag{1}$$

Definition 3.4. Join selectivity: The selectivity $sel(\omega_{\bowtie})$ of a *windowed join* operator that considers tuples from windows W_{d_1} and W_{d_2} over two input streams d_1 and d_2 is the ratio of qualifying join partners to the cartesian product for all tuples in the input windows:

$$sel(\omega_{\bowtie}) = \frac{|W_{d_1} \bowtie W_{d_2}|}{|W_{d_1}| \times |W_{d_2}|}, \quad \text{with } 0 \leq sel(\omega_{\bowtie}) \leq 1. \tag{2}$$

Definition 3.5. Aggregation selectivity: The selectivity $sel(\omega_\xi)$ of a *windowed aggregation* operator that considers tuples in a window W from an input stream D, is the ratio of distinct group-by values in the window over the window length:

$$sel(\omega_\xi) = \frac{|group\text{-}by\ (W_D)|}{|W_D|}, \quad \text{with } 0 \leq sel(\omega_\xi) \leq 1. \tag{3}$$

In general, selectivities, as well as other DCs (e.g., tuple width) can be derived during the training phase since queries anyways are executed to collect observed metrics (throughput and latency) as training data. However, in the inference phase, this is obviously a challenge since for some optimizations tasks (e.g., the initial placement of operators), a query has not been executed and the DCs are unknown. Yet the cost model can clearly be used at runtime to re-optimize a query in case it does not meet the desired QoS metrics since then DCs are known from the execution before. We think that this approach is justified for DSPS, as queries are anyways long-running. In future, we aim to work on learned approaches such as [17] that can be used to predict DCs for DSPS before execution and thus allow our model also to be used for optimizations before a query starts (e.g., an initial placement decision).

Hardware-related features. Hardware characteristics have also a profound impact on the performance of the query in a DSPS, and thereby also on tasks such as placement decisions. In our featurization, we thus include common properties of hardware such as CPU cores, RAM and disk size. These hardware properties are clustered as `instance size` (small, medium and large), similar to the categorization of major cloud providers. These features are also encoded in the graph node to describe the placement characteristic of the operator. Clearly, more properties such as network bandwidth can be added in future, which is out of scope for this paper.

3.3 Training and Inference Procedure

We train the zero-shot models in a supervised way using Graph Neural Networks (GNNs) to learn from the transferable features. All categorical transferable features are encoded using a so-called one-hot encoding per operator, while numerical features are normalized.

During training, the encoded transferable features are used as input to the nodes of the GNN. Particularly, the features of each node are encoded by a corresponding node-specific multi-later perceptron (MLP) into a fixed vector (i.e., the hidden state). These hidden states of nodes are then combined along the graph, using a bottom-up message passing phase by adding up the hidden vectors from the children nodes and combining them with the parent node of the graph. Lastly, the hidden state of the parent node is fed into a final MLP that predicts cost metrics at inference time. In the training phase, instead, the predictions are compared to the real costs and the MLPs are updated accordingly.

Feature	Training data range
`instance size`	small, medium, large
`input event rate`	[0.25, 0.75, 0.5, 1, 1.5, 2.5] $\times 10^3$ e/s
`tuple data type`	[1...5] × [`int, string, double`]
`filter function`	<,>,<=, >=, !=, `startswith, endswith`
`literal data type`	`int, string, double`
`window type`	sliding, tumbling
`window policy`	count-based, time-based
`window size`	[0.25, 0.5, 1, 2, 3] sec; [3, 5, 10, 25, 50, 75, 100] tuples
`slide size`	[0.3 ... 0.7] × window length
`join-key data type`	`int, string, double`
`agg. function`	`min, max, mean, avg`
`group-by data type`	`int, string, double, none`

Table 2: Operator-related features for training data generation using synthetic data.

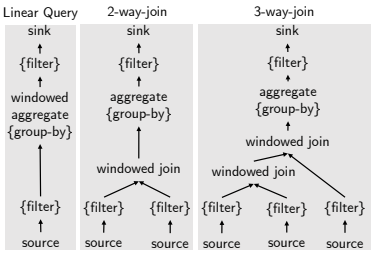

Figure 3: Basic query structures used for training data generation. *Filter* operators and *group-by* are optional.

VM size	CPU	RAM (GB)	Disk (GB)
small	1	1	10
medium	2	2	20
large	8	8	80

Table 3: Hardware-related parameters of virtual instances used for training.

4 EXPERIMENTAL EVALUATION

In this section, we discuss the results of our experimental evaluation. We first explain the generation of data that is used for training and the evaluation setup. Afterwards, we demonstrate the accuracy and robustness of our cost model by carrying out various experiments.

4.1 Data Generation and Setup

Training Data Generation: We created a synthetic training data set with randomly generated queries by enumerating over *(i)* operator parameters, *(ii)* different data streams and *(iii)* distinct query structures. Additionally, we evaluated our model against the existing streaming benchmarks [18]. The different operator parameters and streaming workload configurations (e.g. `event rate`, `data types`) are described in Table 2. A variety of queries are evaluated by the use of three query structures as depicted in Figure 3. They are composed of widely used streaming operators that we instantiated with different parameters. As a result, a high number of possible combinations of operator properties, streaming workloads and query structures are covered in the training set.

Setup and Metrics. To obtain sufficient training data, 15.000 training queries (5000 per query structure) were generated and executed for 1.5 min over Storm clusters with ten virtual machines of different sizes (small, medium, and large — S, M, L). We used the *Stream-API* of Apache Storm and the standard round-robin scheduler for placement. The worker nodes have properties as specified in Table 3,

Query structure	Latency		Throughput	
	median	95th	median	95th
Linear query	1.09	2.14	1.15	2.68
2-way-join	1.13	2.69	1.14	2.91
3-way-join	1.21	5.25	1.22	5.59
Overall	1.13	3.19	1.16	3.50

Table 4: Q-errors (median and 95th percentile) for unseen combinations of operator and workload properties inside the training range.

which is used as `instance size` in the zero-shot model training. For queries that did not receive an output tuple at the sink during their runtime due to unfavorable random operator properties, no costs could be determined. Consequently, 11.175 queries remained that were divided into training (80%), validation (10%) and test (10%) sets. The model was trained with the training data as explained in Section 3.3. We report the *Q-error* metric $q(c, \hat{c})$ to describe the relative deviation of the true cost metric value c (latency or throughput) and its prediction $\hat{c}$, which is a standard metric for the quality of cost models, defined as: $q(c, \hat{c}) = max\left(\frac{c}{\hat{c}}, \frac{\hat{c}}{c}\right)$, with $q \geq 1$ [12].

4.2 Experimental Results

In the following evaluation, we investigate the accuracy of the zero-shot cost models on different workloads. In the experiments, we mainly focus on the generalization ability to predict *(i)* for an unseen test set but using parameters inside the defined training range, *(ii)* for workloads with parameters outside the defined training range, *(iii)* for unseen workloads from existing streaming benchmarks and *(iv)* for workloads under varying hardware heterogeneity.

Q1: How accurate is the model for an unseen combination of operator and workload properties (inside the training range)? At first, we provide the observed Q-errors in predictions for the test set. These are queries and data streams not considered during the training, but use values from the same ranges as shown in Table 2. It can be seen in in Table 4 that the median and the 95th percentile of the Q-error of our model is highly accurate for the given query structures. Typically, cost models based on heuristics have much higher Q-errors of up to 300 for the 95th percentile or more [17].

Q2: How accurate is the model for completely unseen workloads (outside the training range)? To answer this, we generated queries that are within the known parameter space except for one property that exceeds or falls below the training value range. For each property, we generated 50 queries (for all three query structures). The median Q-error from all estimations is reported in Figure 4. To be more precise, we tested tuple widths larger than the training set (A) as well as lower and higher event rates (B). To investigate extrapolation for unseen operator properties, we applied bigger time-based (C) and count-based (D) window sizes as well. In almost all cases, a fairly low median Q-error could be achieved. Intuitively, the Q-error increases with the distance to the training value range as more extreme properties will lead to more extreme costs, which are hard to predict precisely; still way better than existing heuristics [17]. In a second set of experiments, we observed the accuracy for unseen query structures; i.e., we use 4-way and 5-way-joins instead of only up to 3-way joins that were used for training. Moreover, for linear queries, we added 2 to 4 additional filter operators after the `windowed aggregate` operator (cf. Table 5). While the median Q-errors for those are also promising, the tail accuracy of the unseen query structures increases with the increasing query complexity. We aim to tackle this in a follow-up work by incorporating extreme unseen properties in an additional fine-tuning phase (called *few-shot learning*), where only a few unseen queries are used for retraining.

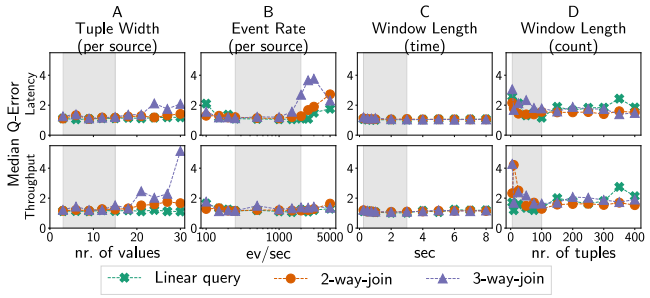

Figure 4: Median Q-errors for queries with unseen tuple widths (A), event rates (B), time-based and count-based window lengths (C, D). The gray shaded area marks the training range. The model extrapolates for unseen queries and workloads (white shaded areas), often with very high accuracy.

Query structure	Latency		Throughput	
	median	95th	median	95th
2-filter-chain	1.14	2.41	1.59	3.65
3-filter-chain	2.67	46.34	2.82	27.78
4-filter-chain	7.33	54.68	3.94	59.73
4-way-join	1.95	24.30	1.33	20.79
5-way-join	1.91	26.76	1.35	21.87

Table 5: Q-errors for unseen query structures. Although the model has never seen these query structures during training, it is able to make accurate cost predictions for them.

Benchmark	Latency		Throughput	
	median	95th	median	95th
Advertisement (clicks)	1.51	1.53	1.38	1.39
Advertisement (impressions)	1.51	1.52	1.38	1.39
Advertisement (join)	1.99	2.06	1.55	2.16
Spike Detection	1.01	1.04	1.73	1.94
Smart Grid (local)	1.21	1.23	1.92	1.92
Smart Grid (global)	1.20	1.66	1.91	1.91

Table 6: Q-errors for benchmarks from [18]. Each query has been executed 5 times with different event rates. This shows our model is able to accurately predict out-of-the-box for these queries.

Q3: How accurate is the model for existing benchmark workloads? To evaluate the generalization capability of the model, we applied it on a selection of three different existing benchmark workloads [18]. *(i)* The *Advertisement Analysis* is a benchmark that computes the ratio of aggregates of two incoming streams (i.e. clicks, impressions), grouped by two attributes. In our evaluation, we used two sub-queries with a windowed aggregation for both of these streams and a third sub-query that joins both streams. *(ii)* The *Spike Detection* benchmark is derived from an IoT use case that computes and compares the moving average of sensor values. *(iii)* The *Smart Grid* benchmark targets the computation of energy consumption in households from the DEBS Grand Challenge 2014. Its *outlier detection* task is decomposed into three sub-queries as presented in [19]. We consider two sub-queries that contain a large sliding window of 60 min (and 60s sliding length) and build average over the local and the global energy consumption. As our model was trained with window lengths up to 3 secs, we reduce the benchmark window length to 5 min (and 5s sliding length), which is still is 100× higher than our training range. As no event rates were specified in the benchmark, we execute the queries five times with different rates each and report the results in Table 6. This shows that the zero-shot model can adapt to streaming workloads of existing benchmark out-of-the-box and estimates the costs very accurately.

Q4: Does the model predict appropriately for different hardware types? As streaming operators are placed on distinct instances (cf. Table 3), we expect the model to take the underlying node performance into account during the prediction. To investigate that, we take all queries from the test set and obtain the predictions under the assumption that all operators are either located on a small, medium or large instance and report the mean of the predictions. Given the linear query, for instance, the model predicts for larger instances higher throughput (31.17/36.14/39.98 ev/sec for S/M/L, resp.). Hence, the zero-shot model correctly learns the effect of instance size on the query performance. In further studies, we aim to model and evaluate hardware characteristics more fine-grained.

5 RELATED WORK

We classify the existing work on cost estimation into two main categories: *(i)* analytical approaches, heuristics and *(ii)* learning-based approaches as stated below.

Analytical Approaches and Heuristics. Several analytical approaches have been applied to predict resource costs in DSPS, such as using queuing theory [6, 20] for different DSPS optimization tasks. However, analytical approaches often make assumptions about the data distribution of incoming streaming workload that renders the estimates either inaccurate or ill-suited for DSPS due to the inherently dynamic nature of the streaming workload. Many works propose heuristics to optimize tasks like placement, namely, greedy approaches [21], meta heuristics [22] and custom heuristics [5]. Storm [23] also adopts heuristics for placement [2]. However, heuristics often also make simplifying assumptions leading to sub-optimal decisions for the optimization tasks. Flink DSPS [15] uses a query optimizer from Stratosphere [24], however, it explicitly targets only user-defined operators in the cost modelling.

Learned Stream Processing. Approaches related to resource cost estimation are the closest to this work. Regression models have been used extensively for estimating performance metrics such as latency [8] and throughput [9, 25] in DSPS. However, these works only consider a limited set of features and largely ignores featurization related to queries that have a major impact on the performance. Different learning methods have been applied to optimize DSPS tasks [10], namely for elasticity [3] or operator placement [7, 8]. Similar to this work, [26] employs neural networks to model the performance of DSPS; however, they stumble around the challenge of re-training for a large number of heterogeneous queries and workloads. In contrast, we introduce a data and query agnostic approach that can support a variety of stream processing optimization tasks that are dependent on the cost estimation of resources *without* any cost of additional training.

6 CONCLUSION

In this paper, we show that zero-shot cost models are highly effective in predicting costs even in the presence of unseen workload conditions. The model can robustly predict QoS metrics for a wide range of queries out of a large set of possible configurations. As such, we believe that the cost estimations are readily usable for optimization tasks of DSPS, such as the operator placement. In the future, an interesting direction is to solve optimization tasks like placement *out-of-the-box* and more explicitly modelling hardware and network properties in the proposed graph structure.

Acknowledgments. This work has been supported by the IPF program and the Cloud Computing Competence Centre of DHBW Mannheim, the Collaborative Research Center (CRC) 1053 MAKI funded by the German Research Foundation (DFG), the NHR4CES Program, hessian.AI at TU Darmstadt as well as DFKI Darmstadt.

REFERENCES

[1] Z. Shao, "Real-time analytics at facebook," *XLDB*, 2011.
[2] M. Nardelli, V. Cardellini, V. Grassi, and F. L. Presti, "Efficient operator placement for distributed data stream processing applications," *IEEE TPDS*, vol. 30, no. 8, pp. 1753–1767, 2019.
[3] G. R. Russo, V. Cardellini, and F. L. Presti, "Reinforcement learning based policies for elastic stream processing on heterogeneous resources," in *ACM DEBS*, 2019, p. 31–42.
[4] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys*, vol. 46, no. 4, 2014.
[5] L. Eskandari, J. Mair, Z. Huang, and D. Eyers, "I-scheduler: Iterative scheduling for distributed stream processing systems," *Future Generation Computing Systems*, vol. 117, pp. 219–233, 2021.
[6] G. Mencagli, P. Dazzi, and N. Tonci, "Spinstreams: A static optimization tool for data stream processing applications," in *ACM Middleware*, 2018, p. 66–79.
[7] M. Luthra, B. Koldehofe, N. Danger, P. Weisenberger, G. Salvaneschi, and I. Stavrakakis, "Tcep: Transitions in operator placement to adapt to dynamic network environments," *Journal of Computer and System Sciences*, vol. 122, pp. 94–125, 2021.
[8] T. Li, Z. Xu, J. Tang, and Y. Wang, "Model-free control for distributed stream data processing using deep reinforcement learning," *PVLDB*, vol. 11, no. 6, p. 705–718, 2018.
[9] C. Wang, X. Meng, Q. Guo, Z. Weng, and C. Yang, "Automating characterization deployment in distributed data stream management systems," *IEEE TKDE*, vol. 29, no. 12, pp. 2669–2681, 2017.
[10] A. Alnafessah, G. Russo Russo, V. Cardellini, G. Casale, and F. Lo Presti, *AI-Driven Performance Management in Data-Intensive Applications*, 2021, pp. 199–222.
[11] G. Li, X. Zhou, and L. Cao, "Machine learning for databases," ser. AIMLSystems, 2021.
[12] B. Hilprecht and C. Binnig, "Zero-shot cost models for out-of-the-box learned cost prediction," 2022, arXiv. [Online]. Available: https://arxiv.org/abs/2201.00561
[13] B. Hilprecht and C. Binnig, "One Model to Rule them All: Towards Zero-Shot Learning for Databases," in *CIDR*, 2022.
[14] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *ICDE*, 2018, pp. 1507–1518.
[15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink™: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015.
[16] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Computing Surveys*, vol. 44, no. 3, 2012.
[17] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig, "Deepdb: Learn from data, not from queries!" *PVLDB*, vol. 13, no. 7, pp. 992–1005, 2020.
[18] M. V. Bordin, D. Griebler, G. Mencagli, C. F. R. Geyer, and L. G. L. Fernandes, "Dspbench: A suite of benchmark applications for distributed data stream processing systems," *IEEE Access*, vol. 8, pp. 222 900–222 917, 2020.
[19] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *ACM SIGMOD*, 2016, p. 555–569.
[20] T. De Matteis and G. Mencagli, "Elastic scaling for distributed latency-sensitive data stream operators," in *PDP*, 2017, pp. 61–68.
[21] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *ACM DEBS*, 2013, p. 207–218.
[22] B. Chandramouli, J. Goldstein, R. Barga, M. Riedewald, and I. Santos, "Accurate latency estimation in a distributed event processing system," in *ICDE*, 2011, p. 255–266.
[23] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *ACM SIGMOD*, 2014, p. 147–156.
[24] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The stratosphere platform for big data analytics," *VLDBJ*, vol. 23, no. 6, p. 939–964, 2014.
[25] S. Imai, S. Patterson, and C. A. Varela, "Maximum sustainable throughput prediction for data stream processing over public clouds," in *IEEE/ACM CCGRID*, 2017, pp. 504–513.
[26] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni, "Elastic symbiotic scaling of operators and resources in stream processing systems," *IEEE TPDS*, vol. 29, no. 3, pp. 572–585, 2018.