[Extended] Jarvis: Large-scale Server Monitoring with Adaptive Near-data Processing

Atul Sandur*, ChanHo Park†, Stavros Volos‡, Gul Agha*, Myeongjae Jeon†
* University of Illinois at Urbana-Champaign, Illinois, USA. Emails: {sandur2,agha}@illinois.edu
†UNIST, South Korea. Emails: {pch8286,mjjeon}@unist.ac.kr
‡Microsoft Research, UK. Email: svolos@microsoft.com

***Abstract*—Server monitoring pipelines are key to deploying highly available datacenter services. Monitoring data is collected from across the stack, transferred to a stream processor, and analyzed to quickly detect and understand failures, so they can be resolved to restore normal service operation. Unfortunately, scaling such an architecture to support large number of servers is challenging as it suffers from the network transfer of massive volumes of monitoring data while query analysis requires a large amount of compute resources.**

We propose Jarvis, a monitoring system that pushes queries near the data source by partitioning a query across data source and stream processor. Jarvis addresses two major issues of state-of-the-art monitoring systems. Unlike conventional coarse-grained operator-level partitioning schemes, Jarvis employs fine-grained data-level partitioning to process only a part of the input on each operator, thereby enabling resource-intensive operators to execute under resource constraints. Unlike fully centralized query planners, Jarvis makes decentralized near-data query refinement decisions, thereby enabling quick adaptation to dynamic resource conditions on each data source. Our evaluation of Jarvis on a diverse set of monitoring queries and data suggests that Jarvis converges to a stable query partition within a few seconds of a resource change occurring on data source, and handles up to 75% more data source nodes while improving throughput in resource-constrained scenarios by 1.2-4.4x when compared to state-of-the-art partitioning strategies.

***Index Terms*—analytics, stream processing, server monitoring, scale, adaptation, query partitioning**

I. Introduction

Today's large-scale datacenters use thousands of servers to host important services, such as web search, database systems, and machine learning (ML) pipelines, for millions of users. Operating these services with high availability requires that datacenter operators quickly detect performance and reliability issues, such as high network latency, disk failures, service outages from software bugs [1]–[7], and resolve these issues in a timely manner to restore normal service operation [8]–[10].

To deliver highly available services, datacenter operators deploy a dedicated monitoring system that analyzes real-time events collected across datacenter servers using a stream processor, as shown in Figure 1. By processing the collected data, the stream processor further enables operators to visualize the behavior of the monitored system via dashboards and generate alerts upon observing issues impacting service availability. *Monitoring data streams* are diverse, including service-level application logs and host-level metrics (e.g., network latency and CPU utilization) used to monitor the health of various system resources. Large-scale monitoring pipelines can generate up to 10s of PBs per day, processing data from hundreds of thousands of servers [10], [11]. Transferring such a high volume of data to remote stream processors can cause network bottlenecks. Moreover, the stream processor may require thousands of CPUs to keep up with such massive data streams while satisfying low time-to-detection requirements for reliability issues [9]. If more data is processed at the data source, amount of data transferred and processed by the stream processor can be significantly reduced.

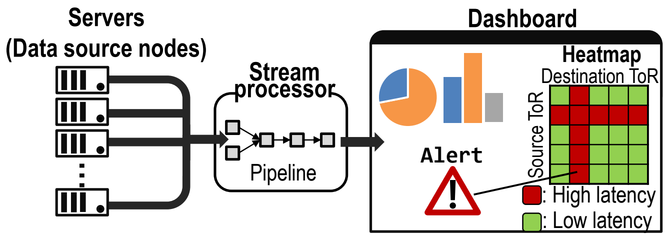


Fig. 1: Overview of a datacenter monitoring system.

Challenges. Not surprisingly, several monitoring pipelines have leveraged available compute resources on data sources (i.e. server nodes) to deploy a monitoring agent. Because available resources on the data source typically result from resource over-provisioning to handle peak resource demands [9], [12]–[15], often the monitoring agent is restricted to run a subset of query operations (e.g., aggregation, filtering) within a given compute budget to minimize interference with the hosted service. Prior work statically decides which operations to run on data source [16]–[18]. However, there is *variability* in monitoring data across time and between data sources. Thus static partitioning may be too conservative or, alternately, may affect query performance during periods of high peak resource demands by the agent's query operators [9], [11].

In the network telemetry domain, Sonata [8] has proposed a query cost model-based optimization approach to distribute query execution across programmable switches and stream processor. The query partitioning in Sonata, occurring at the operator level [16]–[18], is dynamically adjusted by a central query planner running on stream processor. A query operator is deployed to the switch only if the switch's available compute resources are sufficient to process all of the incoming data of the operator. Unfortunately, operator-level query partitioning is not effective in scenarios where available compute resource is highly constrained, such as data sources in monitoring systems. Furthermore, solving an expensive optimization using a centralized query planner is unsuitable for making frequent decisions in systems with data sources that exhibit fast changing resource conditions.

Our proposal. We propose Jarvis, a monitoring system that targets large-scale systems generating *high-volume data streams* with *high variability across time and data sources*. Jarvis identifies independently for the query workload on each data source, a fine-grained *data-level partitioning* strategy by controlling the amount of data processed per query operator, namely *load factor*. Jarvis can make frequent partitioning decisions by combining a model-based technique that quickly finds initial per-operator load factors with a model-agnostic technique that monitors query execution and fine-tunes the load factors if needed. Fine-grained data-level partitioning allows Jarvis to fully utilize the limited and dynamic compute resources over data sources while minimizing network data transfers. Jarvis is implemented in a fully decentralized manner, enabling it to scale to large number of data sources.

To achieve these goals, Jarvis introduces novel extensions to the pipeline of conventional query execution: the *Control proxy* and the *Jarvis runtime*. Control proxy is a light-weight routing logic—associated with a query operator—that given the current load factor decides "how many" incoming records should be forwarded to the associated query operator, realizing data-level partitioning per operator. In each data source, the local Jarvis runtime interacts with all the control proxies to identify their state (e.g., idle, congested, or stable) and fine-tune the load factors of the control proxies. On observing state changes for the control proxies, Jarvis runtime refines its plan for the data-level partitioning to keep the query execution in each data source stable.

We have built a fully functional proof-of-concept of Jarvis. Our evaluation with monitoring queries on host-level network latency metrics and application logs demonstrate that Jarvis enables a stream processor node to handle up to 75% more data sources while improving query throughput by up to 4.4x over state-of-the-art partitioning strategies. Jarvis converges to a stable query configuration within seven seconds of a resource change occurring on data source.

Contributions. In summary, we make the following contributions: (1) we propose a fine-grained data-level partitioning mechanism with the goal of minimizing network data transfers while maximizing resource utilization on each data source node, (2) we design a fully decentralized query workload partitioning engine which quickly adapts its data-level partitioning strategy to dynamic resource conditions on each data source node while scaling to a large number of data sources, and (3) we experimentally evaluate our system with several monitoring queries and show its effectiveness.

II. Background and Challenges for Large-scale Server Monitoring

The monitoring systems we target must be able to investigate timestamped measurement data from individual datacenter nodes corresponding to health of hardware, OS, and runtime systems [2], [4], [7], *and/or* textual logs generated by a specific service which spans a large number of nodes to be hosted [11], [19]. Although numerous useful scenarios correspond to large-scale server monitoring in real world [1], [4], [9], we introduce the following two monitoring scenarios in an enterprise company that guide the design of Jarvis:

- *Scenario 1:* Network team deploys PingMesh [2] agents on each datacenter node to collect probe latency data between server pairs. An internal Search team uses PingMesh to monitor network health of their latency sensitive service and generate an alert if median probe latency of nodes hosting their service exceeds a threshold. The threshold is determined by the search service SLA.

- *Scenario 2:* A log processing system (e.g. Helios [9]) enables live debugging of storage analytics services (e.g. Cosmos [20]). A bug in cluster resource manager leads to service resources being under-provisioned. To temporarily mitigate performance degradation, multiple TBs of log streams [9] are processed quickly to identify impacted tenants. Their latency and CPU/memory utilization data are queried to predict resource needs and make scaling decisions.

The above scenarios require processing different types of monitoring data. *Scenario 1* requires processing metrics which are structured numeric data and generated periodically. Queries on this data consist of operations such as filtering and aggregation. On the contrary, *Scenario 2* requires processing logs which are unstructured strings and are generated aperiodically. Queries consist of string processing operations such as parsing, splitting, search, etc. We observe the following common challenges facing modern monitoring pipelines in supporting the two scenarios:

Voluminous data transfer: A monitoring system must process up to tens of TB data per day or millions of events every second per monitoring task [2], [11], [21], straining compute, memory, and network capacities of the monitoring system. As an example, we illustrate how costly data transfer can be for network latency monitoring in *Scenario 1*, assuming a traditional approach where the entire data is aggregated into *stream processor* to be analyzed. Assume that a datacenter has 200k servers. Guided by [2], each server may probe 20k other servers with a probing interval of 5 seconds. Then, the data generation rate for this system alone is estimated at 512.6 Gbps as a result of large-scale probing performed by 200K servers. Similarly, logs collected from applications (as in *Scenario 2*) can generate up to 971 Gbps [11]. To keep up with these high data generation rates, stream processors need to employ a large number of compute cores, making it costly to build the monitoring system. This pressing issue suggests that achieving high efficiency and scalability is a key design challenge for

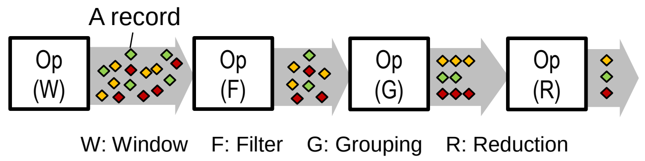


Fig. 2: Pipeline of stream operations for motivating example query (detailed query in Listing 1).

large-scale server monitoring.

Highly varying data: Monitoring input data and their generation rates vary significantly across different data source nodes. Section **??** describes in detail how metrics data differs across data source nodes. Recent works [9], [11] have also described high variation in ingestion rates for log streams generated in production monitoring systems. To keep up with high varying data, monitoring pipelines are equipped with large amount of compute and memory resources, which are highly distributed across dedicated nodes. To handle processing load from a large number of data source nodes, the stream processor may integrate several levels of computing nodes. Query optimization techniques need to support the requirements of highly dynamic workloads in a complex distributed environment.

III. Monitoring Workload Characteristics

Datacenter operators assign the monitoring system only a limited amount of HW resources (e.g., only few cores) from individual data source nodes to promise little interference with foreground services. The resources allowed per monitoring query could be even more scarce as multiple queries with different monitoring tasks could run on the system at the same time. Many prior works take an approach to pre-process the query on limited resources by pushing a subset of query computations close to data source, reducing data movements over the network [8], [16], [22]. In this section, we discuss one of our target scenarios in detail to reveal workload characteristics that limit effectiveness of the existing approaches.

A. Network Latency Monitoring Example

We use a real-world trace on datacenter network latency monitoring, *Scenario 1*, collected from Pingmesh in Microsoft. In Pingmesh, servers in the datacenter probe each other to measure network latency for server pairs across the datacenter. A server sending a probe message generates a 86B record, including timestamp (8B), source IP address (4B), source cluster ID (4B), destination IP address (4B), destination cluster ID (4B), round trip time (4B), and error code (4B). The round trip time is in microseconds.

```
/* 1. create a pipeline of operators */
query = Stream
.Window(10_SECONDS)
.Filter(e => e.errorCode == 0)
.GroupApply((e.srcIp, e.dstIp))
.Aggregate(c => c.avg(rtt) && c.max(rtt) && c
    .min(rtt))
/* 2. Execute the pipeline */
Runner r( /* config info */ );
r.run(query);
```

Listing 1: A temporal query for server-to-server latency probing in every 10-second fixed-size window.

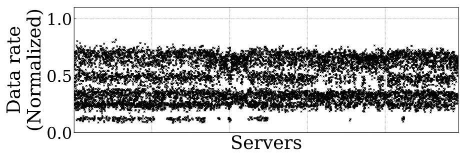


Fig. 3: Non-uniform data generation rates across data source nodes in real network latency monitoring trace.

This rich set of record fields allows exhaustive analyses: e.g., zoom-in/out latency probes by grouping them across datacenter network hierarchy to identify issues at the cluster, ToR or spine switch and server level [2], [23]. Here, we describe implementation of a real query based on Pingmesh dataset in Listing 1. As can be seen, we adopt a declarative programming model used by popular stream engines such as Flink [24] and Spark Streaming [25], which express a query using dataflow operations. Figure 2 shows a pipeline of how data flows between operators for the query in Listing 1.

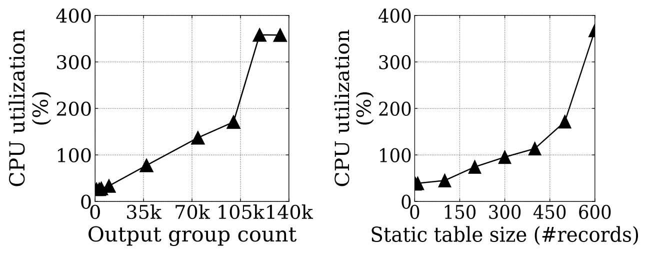
B. Variabilities in the Wild

Datacenter monitoring workloads exhibit variabilities in many aspects. We describe four aspects that make precisely estimating resource requirements of queries at scale highly complicated.

Data generation rates across data source nodes: Data source nodes often generate monitoring data tightly associated with their targets to monitor (e.g., # HW devices) and occurrences of anomalies (e.g., # high-latency events for network/storage), which happen to be non-uniform across the nodes. For example, in network latency monitoring, some servers probe a larger set of peers to cover a larger network range on behalf of other servers in the same ToR switch [2]. More recent works have reported variability in data generation rates in a variety of monitoring scenarios as well [9], [11]. As a result, the resource requirements of data source nodes would be *different* and *independent* of each other.

To show the significance of such variability in Pingmesh, we measure data rates of network probe messages generated by individual data sources, and show the results in Figure 3. The data rates are normalized to the highest one observed among data source nodes. Data generation rates exhibit high variability, with 58% of nodes generating 50% or lower of the highest data rate we observe.

Quantitative analysis of resource usage across operators: Popular declarative operators used in monitoring queries include Filter(F), Grouping(G), Reduction(R), Join(J), Project(P) and Map(M) [24], [26], [27]. These operators are computationally different and their costs are highly dynamic. For example, Filter (stateless operator) drops uninteresting records by applying the predicates on each record and typically operates at a low computational cost. Grouping (stateful operator)



(a) G+R operator (b) Join operator

Fig. 4: Profiling of query operators.

organizes records by key fields and hence requires key lookups and moving of records, making it an expensive operator. Join (stateful operator) joins between an input stream and a static table on data source[1].

Next, we observe dynamic costs of operators by profiling F, G+R[2] and J operator using our trace on a stream processor node with 4 2.4 GHz cores. We configure data input rate to be 26.2 Mbps (10x scaled up) to ease our measurements (see **Pingmesh** in Section VIII for details). We then adjust several parameters that affect operator resource usage: (i) filtering predicate for F operator to obtain different filter-out rates (between 20-80%), (ii) output group counts for G+R operator reflecting a range of grouping granularity and (ii) the size of static table for J operator which is joined with an input stream to add ToR switch IDes given server IP addresses to each record (for grouping by ToR IDes). We observed that for a range of filter-out rates, F operator requires 17-25% CPU, i.e., low compute cost and variance. However, Figure 4 shows that when the number of groups and the table size changes for G+R and J operators respectively, CPU usage varies substantially between 50%-350%.

Data reduction through a pipeline of operators: Declarative operators in a pipeline execute in sequence. Therefore, resource demand for an operator in the pipeline is affected by the data incoming rate to the operator out of executing all preceding operators. This further complicates resource usage characteristics of pipeline execution while processing the query on the input stream. For example, F operator leads to the first data volume reduction from the input stream (see filtering predicate in Listing 1). Since erroneous, insignificant records are not dominant, the operator happens to drop only a small fraction of records. A majority of records are then to be processed by G+R operator, increasing compute cost for the query. However, if we were to toggle the filtering predicates to perform a zoom-in analysis on a small subset of erroneous data, there will be a high degree of data reduction in F operator, so G+R operator considerably lowers its resource usage.

Data distribution: Large-scale server monitoring may see the data with distribution shifting time-to-time. For example, when servers are being over-loaded, latency profiles become more variable [28], [29]. If a monitoring system runs a query to diagnose such network condition, it should quickly adapt the query to process a larger amount of network probes of high latency.

[1]Due to limitations of data source-side execution discussed in Section VI, we do not consider other complex join operators in Jarvis.

[2]Reduction commonly follows grouping to apply aggregation functions incrementally per group, so both are often combined.

IV. Models and Problem Definition

This paper investigates techniques to partition/distribute query operators between data source and stream processor nodes, to address the challenges with building large-scale server monitoring pipelines. In this section, we discuss models/assumptions and then define the query partitioning problem. Table II provides a summary of the variables used in the rest of the paper.

Data and query models: Monitoring data is an unbounded stream of *records*. In the stream, a single record is a measurement or text log generated by the underlying host or application on the server node. Each record has an associated timestamp which defines an event time when the record was generated. The data stream also includes *watermark* records with strictly increasing timestamps. A watermark with timestamp w_{ts} guarantees no subsequent records will have a record time earlier than w_{ts}. As a result, on observing watermarks, a single stream could be delimited into ordered consecutive *epochs* of records while promising time progresses. An epoch may have records with timestamps greater than the epoch's end watermark due to out-of-order arrival.Each monitoring query is represented as a directed acyclic graph (DAG) $G = (V, E)$. The vertices V in the DAG represent stream operations such as filter, grouping, reduction, etc, which transform their input records and emit output records to downstream operations, as shown in Figure 2. The edges E denote dataflow dependencies between operations. An instance of the query G deployed on a data source i is denoted by $G_i = (V_i, E_i)$.

Resource model: Physical resources involved in the query execution can be seen as a tree structure as shown in Figure 6(b), where leaf nodes represent the data sources that primarily run customer applications. In each leaf node, fixed amount of compute resources are allocated to facilitate monitoring queries. Such allocation can be done by provisioning a container to isolate the resources (e.g. 1 core and 1 GB RAM) for monitoring from foreground services. Each data source can send data to its parent stream processor node (i.e., intermediate node) and leverage its compute resources for query processing. The intermediate nodes eventually send data to root stream processor node which completes the query before outputting final result.

Operator partitioning: Monitoring queries typically consist of operations which filter out uninteresting records, aggregate statistics, etc. [1]–[3], [30].iven a query instance with M operations, a partitioning plan identifies operations that can execute locally on each data source and those that require remote execution on it's parent node. To illustrate, consider a topologically sorted list of M operators from the query graph G following dataflow dependencies of the query. Due to underlying resource topology, we can $partition$ the operators into two groups and split the query execution across the data source and other remote nodes. To this end, we can define a boundary operator b_i for the i^{th} data source to indicate operators prior to b_i in the topological order are executed on data source and those after b_i are executed remotely. For instance, for a data source i with $M = 4$ operators, $b_i = 3$ means first 3 operators execute on data source and the last operator executes remotely.

Refinement model: When resource conditions change on a data source node, the query needs to be refined i.e. a new partitioning plan needs to be identified so that its compute needs meet available resource budget. To initiate refinement, data sources periodically inject *refinement watermarks* into the stream (similar to [27]) and they delimit ordered consecutive *refinement epochs* of records. Refinement watermarks are different from the traditional watermark described in data model. Epoch and watermark refer to refinement epoch and refinement watermark respectively, in the rest of the paper.

Problem definition: We define the operator partitioning problem as follows (Table II describes the variables used):

$$\min_{\mathbf{B}} \sum_{i=1}^{N_d} \sum_{j=1}^{M} dr_{ij} x_{ij}$$

$subject\ to,$

$$T_{local}(i, b_i) <= T_{remote}(i, b_i)\ \forall b_i > 0, i\epsilon[1, N_d]$$

Network transfer and compute costs are incurred on stream processor, when data is sent from data source for remote processing. Our optimization goal is to minimize the processing load sent to stream processor subject to the processing time not being delayed i.e. local computation not introducing additional delay compared to remote execution, in the processing of each data stream from being reflected in query output. We prove that the operator partitioning problem is NP-hard in Section A. This is because of finite resources available on stream processor which requires joint partitioning decision to be made across data sources.

V. Motivation for Jarvis

Data sources have limited compute, while query resource demands are highly dynamic as explained in Section III-B. Given this, we highlight the drawbacks of existing approaches in addressing the operator partitioning problem and insights that Jarvis provides.

(Drawback 1) Coarse-grained partitioning can significantly undersubscribe data source node resources: Existing works [8], [16], [22] have explored partitioning strategies which are coarse-grained, i.e., either all records are processed by an operator or none. To examine its limitation, we execute the query in Figure 2 using 80% of single core (2.4 GHz), to reflect compute constraints on data source. Figure 5 shows that F operator drops only a small portion of input records, so we are not able to execute the expensive G+R operator on data source. This leads to under-subscription of the compute and generates network traffic as much as 22.5 Mbps.

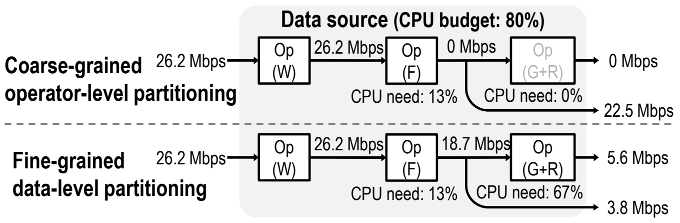


Fig. 5: Coarse-grained operator vs. fine-grained data partitioning on data source with CPU budget of 80%. G+R requires 80% CPU to process all the output data from F operator.

Data partitioning is fine-grained i.e. it enables an operator to process a fraction of its input records on data source and drain the rest for remote processing. In this case, G+R can process 83% of its input 22.5 Mbps and greatly reduce it to result in a total of 9.4 Mbps (5.6+3.8 Mbps) leaving data source node. This is a significant decrease over 22.5 Mbps in operator partitioning.

Insight 1: *Extending the operator partitioning model to support data partitioning can significantly improve the utilization of limited compute available on data source nodes.*

(Drawback 2) Slow adaptation cannot keep up with highly dynamic resource conditions on data sources: The requirement for frequent and dynamic query refinement (as illustrated in Section III-B) is more prevalent in practice since data sources are shared by multiple in-flight queries which are initiated or terminated by engineers at runtime [7], changing the available compute per-query. Given these dynamics, query planners need to compute and realize (i.e. execute) a new refinement in a timely manner, even in seconds once requested.

In previous works, centralized planners leveraged global information about available resources to solve an expensive optimization function [8] and relied on accurate query performance models [31], [32] to make optimal partitioning decisions. Doing so is computationally expensive (see Section IV). Other centralized and decentralized query planners have been explored but they do not consider the dynamics resulting from co-located monitoring queries on data sources with high variability in available compute and query resource needs [33]–[38], [38]–[43] (details in Section IX). Jarvis investigates a greedy approach which is embarrassingly parallel and relies on a hybrid combination of query performance model-based and model-agnostic techniques to partition queries on each data source. It is fully transparent to the user with low cost of migrating to a new refinement.Insight 2: *A greedy and embarrassingly parallel approach can enable fast and fully decentralized query refinement on each data source, which is accurate, scalable and automatic.*

VI. Jarvis system

Jarvis design extends conventional query computation graphs, so it is compatible with most of the existing query engines that express a query execution as a computation graph of stream operators. Since the extension is done after a query

Common variables	
M	Number of operators in the query
N	Total number of records injected into the query in an epoch
Operator partitioning problem	
N_d	Number of data sources
b_i	$b_i \epsilon [0, M]$ to denote the index of boundary operator where the partitioning occurs for i^{th} data source
B	partition profile for the query with b_i denoting boundary operator for i^{th} data source
x_{ij}	indicates whether $b_i = j$
dr_{ij}	rate of data sent from data source to stream processor when $b_i = j$
$T_{local}(i, b_i)$	computes local computation time for executing operators until b_i on the i^{th} data source
$T_{remote}(i, b_i)$	computes network transmission time cost of intermediate output from b_i^{th} operator to its parent node along with computation time cost of executing operators after b_i on parent node
Data partitioning problem	
Op_i	i^{th} operator in the query
r_j	relay ratio of Op_j i.e. fraction of input records relayed to the next operator Op_{j+1} after they are consumed by Op_j
c_i	compute cost of Op_i for a single record
d_i	number of records drained in an epoch, at the i^{th} control proxy
C	compute budget available to the query
p_i	i^{th} control proxy's load factor i.e. fraction of input records added to downstream queue and consumed by downstream operator
e_i	effective load factor for i^{th} control proxy i.e. product of load factors of sequence of upstream query operators until Op_i, in the transformed optimization problem

TABLE I: Summary of variables used in the paper

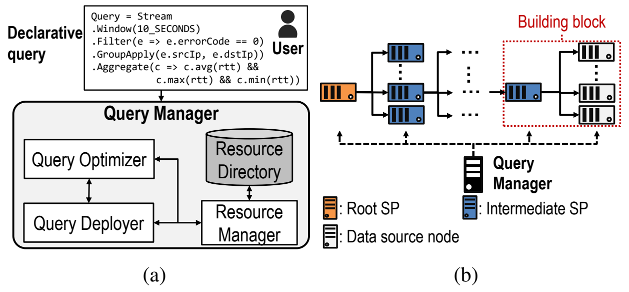
compiler constructs the query graph, Jarvis is fully transparent to users. In this section, we discuss Jarvis architecture, its overall design and an algorithm to achieve fast and accurate query refinement.

A. High-level Architecture

Figure 6(a) shows a user-defined a query that is submitted to Jarvis Query Manager. Query Manager consists of a Query Optimizer which performs query optimizations similar to those done by state-of-the-art streaming engines [24]. Query Manager consists of a Resource Manager, which maintains the current list of data source and stream processor nodes, along with their network topology in a Resource Directory. Topology information is updated whenever nodes are added, modified, or removed. Query Optimizer uses the node topology to generate an optimized physical plan for each node. Query Deployer generates the executable code corresponding to the physical plan and deploys it on the stream processing engine running on each node. Figure 6(b) shows how Query Manager interacts with data source and stream processor nodes. Our work in Jarvis focuses on changes to the query plan generation pipeline in Query Optimizer and runtime optimizations after queries are deployed on data source and stream processor nodes.

To realize data-level partitioning and fast query refinement, Jarvis adds two new primitives to the query processing pipeline: (i) *Control proxy*, a unified abstraction for stream operator and (ii) *Jarvis runtime*, a system runtime that coordinates executions of all control proxies. Control proxy is a light-weight operator bridging two adjacent stream operators and decides "how many" records shall be forwarded to the next downstream operator in the pipeline. Therefore, control proxy implements data-level partitioning at *operator level*. Jarvis runtime interacts with all control proxies within the data source node to coordinate their data-level partitioning actions, enabling a refinement plan at *query level*.

Jarvis modifies the query optimization pipeline to enable adding control proxy between adjacent stream operators. The



```
Query = Stream
.Window(10_SECONDS)
.Filter(e => e.errorCode == 0)
.GroupApply(e.srcIp, e.dstIp))
.Aggregate(c => c.avg(rtt) &&
           c.max(rtt) && c.min(rtt))
```

Fig. 6: (a) Architecture for Jarvis query manager, (b) Architecture for the full monitoring pipeline. SP refers to stream processor and L_0 to L_H refer to the hierarchy levels in a tree topology of height H.

optimized query is deployed onto data source and stream processor nodes. Figure 7 illustrates the query plan deployed on data source and stream processor for the running example query presented in Figure 2. Jarvis runtime deployed at each data source configures control proxies to execute a data-level partitioning plan which minimizes network transfer cost within compute budget on data source. Jarvis runtime continually probes the state of control proxies to observe their query state. If it detects changes to query resource conditions, a new data-level partitioning plan is computed for the query. Note that all partitioning decisions are made locally on each data source without relying on an external planner nor requiring coordination between data source and stream processor.

B. Query Plan Generation

Monitoring queries may consist of sequence of operators; complex queries may join results of two different subqueries [8]. Given the difference in compute capabilities and limitations in type of operations that can run on data source and stream processor, Query Optimizer generates custom query plans to deploy on the nodes.

We reuse the query plan generation mechanism for existing streaming engines [44]. The input query is parsed to do syntactic checks and then logical optimizations (e.g., constant folding, predicate pushdown) are introduced to produce a logical plan. The optimized logical plan is used to generate a physical plan for deployment and execution. Jarvis additionally inserts a control proxy between each of the adjacent stream operators in the logical plan, for deployment to data source and intermediate stream processor nodes. Jarvis runtime is also deployed on data sources. Certain operations such as non-incremental updates cannot be supported in the data source and intermediate stream processors. Resource constraints further limit the operations that can be supported on data sources. Jarvis performs rule-based physical optimizations to account for such constraints. We describe below the rules to identify operations which *cannot* execute on data sources:

- *[Rule 1]* Aggregation operations that are not incrementally updatable, such as exact quantiles. However, their approximate versions, such as approximate quantiles [30], [45], can benefit from Jarvis.
- *[Rule 2]* Downstream operators succeeding stateful operations that require aggregation across multiple data sources.
- *[Rule 3]* Stateful joins across streams. Prior work [8] has also excluded such operations because they are expensive and may not reduce data that needs to be sent out of data source.
- *[Rule 4]* Multiple physical operators for the same logical operator, which parallelize operator execution (e.g. [31]). Data sources have constrained compute budget; hence, benefits of exploiting operator parallelism are limited.

All rules except the fourth rule apply also to intermediate stream processors. As the latter are dedicated to run monitoring queries, hardware-level parallelism can be exploited to accelerate query operators. It may appear that the first rule limits queries that can leverage Jarvis. However, a significant number of real-world queries use operators that support incremental updates. For example, Drizzle [46] has reported 95% of aggregation queries on a popular cloud-based data analytics platform consist of aggregates supporting incremental updates, such as sum and count.

We note that after applying above rules, queries deployed on data sources typically consist of sequences of operators. Hereafter, while discussing query refinement on data sources, we restrict our discussion to operator sequence graphs. Nevertheless, our approach is extendable to handle graphs with split patterns that may execute on data sources—i.e., one operator output is an input to multiple downstream operators, and we leave this for future work.

C. Dynamic Query Refinement

After queries are deployed, a data-level partitioning plan is implemented by Jarvis runtime by guiding each control proxy to select a portion p (i.e., $0 \leq p \leq 1$) of incoming data to be inserted to the downstream queue and subsequently consumed by the downstream operator. Remaining data is drained over the network to be processed by the stream processor is routed

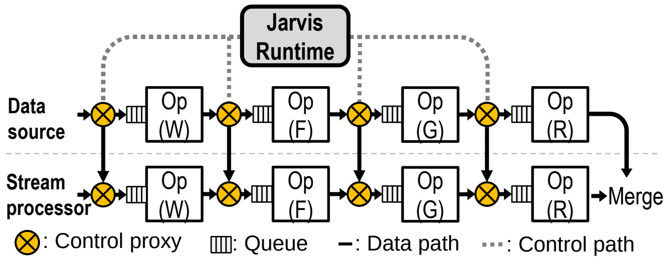


Fig. 7: Jarvis rewrites a query to incorporate control proxies and Jarvis runtime. Data path: routing path for incoming data; Control path: interaction between control proxies and Jarvis runtime.

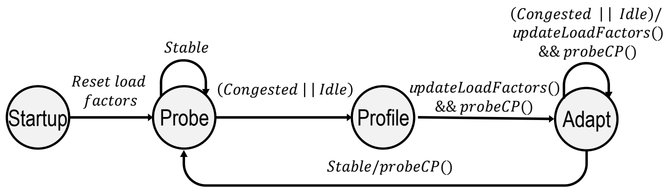

Startup
Reset load factors
Probe
Stable
(Congested || Idle)
Profile
updateLoadFactors() && probeCP()
Adapt
(Congested || Idle)/ updateLoadFactors() && probeCP()
Stable/probeCP()

Fig. 8: Jarvis runtime state machine.

to the control proxy associated with the same downstream operator. Hereafter, we refer to the p value configured for each control proxy as its *load factor*.

Unlike prior work that physically splits a query, our query partitioning scheme "replicates" query operators across the nodes. This approach simplifies scenarios that require switching to a new partitioning plan by locally updating load factors of each control proxy. Jarvis runtime orchestrates load factors through query refinement at an epoch granularity. During epoch processing, each control proxy monitors its downstream operator, and at the end of each epoch, it updates the state of the operator to one of three states: (i) *Congested*: operator contains more than a predefined number of pending records, experiencing backpressure; (ii) *Idle*: operator stays empty for longer than a predefined time duration; and (iii) *Stable*: operator is neither congested nor idle. Jarvis runtime collects state information from all control proxies per epoch and classifies current data-level partitioning plan as *non-stable* if all operators are idle or at least one operator is congested. Upon identifying the query as non-stable, Jarvis runtime triggers adaptation to bring the query back to the stable state.

Figure 8 shows the interplay between Jarvis runtime and control proxies and the overall workflow with different operational phases:

- **Startup: initialization.** All load factor values are initialized to 0 (the smallest value), so all stream records are processed by stream processor.
- **Probe: normal operation.** At the end of every epoch, Jarvis runtime executes `ProbeCP()` function to query all control proxies and determine their congestion states. It continues to do so until it identifies the computation pipeline as congested or idle. After that, the runtime enters `Profile` phase.
- **Profile: query plan diagnosis.** When the query is congested/idle, Jarvis obtains new estimates for the following during

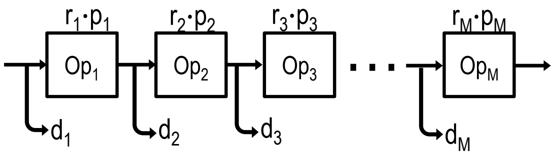


Fig. 9: Computation graph of M operators and parameters used by Jarvis for updating load factors in control proxies.

profiling phase: (1) compute cost of each operator by executing an operator at a time on the incoming epoch, (2) reduction in size of input stream by executing each operator, and (3) available compute budget for the query. These estimates are used to adapt load factors in the next `Adapt` phase.[3].

• **Adapt: load factor adaptation.** After profiling, Jarvis runtime executes an adaptation algorithm for computing a new data-level partitioning plan. Initial load factors are first calculated using the profiling estimates and set for each control proxy. Jarvis runtime executes `ProbeCP()` to probe the query state and perform iterative fine-tuning if necessary, until the computation pipeline is back to stable state. At this point, it returns to `Probe` phase.

Small workload variation when in congested state or idle state can trigger a series of profile-adapt phases that put the control system in an oscillating behavior with small implications on the optimal load factors. To avoid this undesired behavior, each control proxy is configured with a threshold fraction of epoch records (`DrainedThres`) that can be drained by control proxies and tolerated by `ProbeCP()` without signaling congested state. Similarly, each control proxy is configured with a threshold fraction of epoch duration (`IdleThres`) that allows control proxies to stay idle and can be tolerated by `ProbeCP()` without signaling idle state.

Supporting multiple queries in data sources reqiores that a single Jarvis runtime instance is dedicated to a single individual query; each instance refines its query independently. When allocating the compute budget of data source node to the competing monitoring queries, Jarvis uses a popular fair resource allocation policy, such as non-weighted max-min fairness [47] without any loss in generality. Available budget per query is obtained by considering the number of queries and total compute available for the queries on data source node.

D. Computing Data-level Partitioning Plan

Jarvis runtime in its `Adapt` phase needs to compute a new data-level partitioning plan to bring the query back to stable state. This is done by solving an optimization problem to compute initial load factors and then iteratively fine-tuning load factors if needed, to stabilize the query.

1) Problem Definition: Consider a computation pipeline as shown in Figure 9 which contains M operators. Table II summarizes the variables used. For each control proxy, guided by its load factor, a record will be either pushed to the downstream operator or drained. We want to minimize the total number of drained records (i.e. $\sum_{i=1}^{M} d_i$) from data source given the compute budget C available to the query. This yields the following optimization formulation:

Summary of variables used in the paper	
M	Number of operators in the query
N	Total number of records injected into the query in an epoch
Op_i	i^{th} operator in the query
r_j	Relay ratio of Op_j i.e. fraction of input records relayed to the next operator Op_{j+1} after they are consumed by Op_j
c_i	Compute cost of Op_i for a single record
d_i	Number of records drained in an epoch, at the i^{th} control proxy
C	Compute budget available to the query
p_i	i^{th} control proxy's load factor i.e. fraction of input records added to downstream queue and consumed by downstream operator
e_i	Effective load factor for i^{th} control proxy i.e. product of load factors of sequence of upstream query operators until Op_i, in the transformed optimization problem

TABLE II

$$\min_{p_1,p_2,\dots p_M} \sum_{i=1}^{M} [\prod_{j=0}^{i-1} p_j r_j](1-p_i) \tag{1}$$

subject to

$$\sum_{i=1}^{M} [\prod_{j=0}^{i-1} p_j r_j] p_i c_i \le C/N, \tag{2}$$

$$0 \le p_i \le 1,\ 0 \le r_i \le 1,\ c_i \ge 0\ \forall\ i\ \epsilon\ [1, M], p_0 = 1, r_0 = 1$$

N, M, and C are fixed for an instance of the problem.

Unfortunately, there are two challenges in solving this optimization problem. First, it can be proved that Hessian matrix is not positive semi-definite even for the two-variable cases, resulting in a computationally hard non-convex formulation [48]. While it is possible to enumerate all possible combinations of load factor values, doing so is expensive for online optimization. Second, the formulation assumes certain conditions, which may not always be satisfied in practice. For instance, to estimate c_i accurately, each operator needs to be evaluated on a sufficient number of input records. As seen in Section **??**, for grouping operation, c_i is affected by the number of groups in input data which can change at runtime. Furthermore, relay ratio r_i can vary non-linearly for grouping, as it is affected by grouping key distribution in the input.

2) StepWise-Adapt Algorithm: We now describe how Jarvis deals with the data-level partitioning problem. Jarvis employs our *StepWise-Adapt*, a novel hybrid algorithm that combines two techniques: (i) a model-based technique which searches for near-optimal load factors based on the modeling assumptions of data-level partitioning problem defined in Equation 1. and (ii) a model-agnostic technique that monitors query execution using load factors obtained from step (i) and fine-tunes them if the query behavior deviates considerably from stable state—i.e., the available resources are either over/under-subscribed by the query. For fast fine-tuning, step (ii) uses a heuristic inspired by first fit decreasing heuristic for bin packing problem [49], to prioritize load factor updates of operators that contribute significantly to network data reduction.

We observe that the cost of solving data-level partitioning is very sensitive to the nature of transformation applied to

[3]Profiling estimates can still improve convergence time even if they do not exactly match the compute cost during normal operation (Section VI-D.)

the problem. For example, we transformed the problem to a geometric program (GP) [50] for a small scale instance of the problem with just 2 optimization variables. When varying different parameters in the optimization, GP solver takes up to 7 seconds for solving the problem using CVXOPT solver [51] on a machine with four 2-GHz i7 CPUs. This is expensive for the fast convergence requirement of query refinement. To enable online optimization, we thus identified another simple transformation of the original problem that converts it into a linear program (LP). The transformation is done by introducing a new optimization variable e_i for the i^{th} control proxy where $e_i = \prod_{j=0}^{i} p_j$. Then the optimization problem in Equation 1 can be rewritten as:

$$\min_{e_1,e_2,\ldots e_M} \sum_{i=1}^{M}[(\prod_{j=0}^{i-1} r_j).(e_{i-1} - e_i)] \tag{3}$$

subject to

$$\sum_{i=1}^{M}[(\prod_{j=0}^{i-1} r_j).e_i.c_i] \leq C/N,$$
$$0 \leq e_i \leq e_{i-1}\ \forall\ i\ \epsilon\ [1, M],\ e_0 = 1$$

Rest of the conditions on N, M, C, c_i, r_i remain the same as in the original formulation. When resource change occurs on data source, load factors need to change to stabilize the query.

A feasible solution provided by LP solver assumes that operator relay ratios/costs of operators are fixed and independent of load factors. However, if these parameters are unsteady or vary non-linearly, the solver provides load factors which would either over-subscribe or under-subscribe the compute budget, making the query execution unstable. To address the issue, StepWise-Adapt takes a post-LP solver step to further tune the load factors. In this step, StepWise-Adapt observes the congestion state of the query after it executes an epoch with current load factors of control proxies and updates them based on priorities of their downstream operators. An operator that has the *lowest* data relay ratio is assigned the *highest* priority, and StepWise-Adapt aims to increase its load factor first when a query is in idle state. For a query in congested state, load factor for the operator in the lowest priority will be decreased first. In this way, we can give more resources to operators that potentially bring about larger data reduction. All the information necessary for StepWise-Adapt is unknown beforehand, but can be readily obtained with profiling. Note that a high priority operator may have a high compute cost, but since our objective is to minimize the network transfer cost within the compute budget, we choose to find the smallest number of load factor updates to fully utilize available compute.

Fine-tuning strategy. We update load factors obtained in the previous step (per LP solver) based on operator priorities, which are determined based on relay ratio of each operator. If a query is idle, we increase the load factor for the highest priority operator that is not fully executing yet—i.e., $p < 1$. In contrast, if a query is congested, we reduce the load factor for the lowest priority operator that is still executing—i.e., $p > 0$. When updating load factors for an operator, we execute a binary search over discretized load factor values of the operator to improve convergence time—i.e., from the current load factor to the maximum value in idle state or to the minimum value in congested state.

Jarvis limitations. Our greedy approach may incur processing latency overhead. This overhead can be limited by configuring the epoch size for making partitioning decisions.

VII. Implementation

Jarvis is implemented using lightweight query execution runtime Apache MiNiFi [52] for data source side and Apache NiFi [53] for stream processor side runtime. RxJava is used to implement query computation pipelines within NiFi/MiNiFi custom processors. Kryo serialization framework [54] is used for transferring data over the network between data source and stream processor nodes. We highlight implementation issues that Jarvis addresses.

Maintaining streaming semantics. Input records may be processed by all query operators on data source or drained via one of the intermediate control proxies based on their load factors. For accuracy, a watermark must be present in the regular execution path after processing all operators, as well as the drained path at each control proxy. So if records are drained by control proxy, it replicates the watermark in both regular and drained paths. On stream processor, each control proxy receives records from upstream operator on the node and corresponding control proxy on data sources. Records in both the paths are demarcated by watermarks. To consume records from multiple streams, we use the methodology used by Flink [55]. Each operator advances its time based on the minimum of all it's incoming input streams' event times.

Merging results at stream processor. When control proxy on data source drains records in an epoch, it attaches an ID with records for receiver control proxy on stream processor to continue execution. If preceding operator is stateless, control proxy sends drained records to corresponding control proxy on stream processor. However, if preceding operator is stateful and requires aggregation across multiple data sources, control proxy sends records to a downstream merge operator on stream processor to perform global aggregation.

Externalizing operator results on time: Congestion control method in Jarvis is designed for continuous stream computation model [56]. Therefore, to estimate congestion state for each operator precisely, all query operators must externalize computed results continuously to the next operator. With the implementation of G+R operator in RxJava, this was not the case as it collects all epoch records during grouping into a hash table and externalizes it with grouped records when an epoch ends. This does not reflect processing and idle time of next operator R as records never arrive during the current epoch, reporting an inaccurate congestion state and breaking our query refinement algorithm. We address this issue by dividing an epoch into sub-epochs and externalizing grouping results on a sub-epoch basis. This does not require changes on stream operators; instead, it requires each control proxy to initiate query refinement only when the control proxy encounters end of the epoch, not sub-epoch. Jarvis currently divides an epoch into 100 sub-epochs.

Probing downstream queues. Some streaming engines may hide the operator's internal queue state by abstracting data transfer between two operators with well-defined APIs [57]. Our RxJava based implementation also shared this concern originally. We address it by forcing synchronous queueing on the incoming data stream. This allows us to manage our own custom queues external to query operators, and the queueing time of records in the queue can correctly reflect the processing time for the operator.

VIII. Evaluation

We evaluate the effectiveness of Jarvis in improving the system throughput and in quickly adapting query partitioning plans to dynamic resource conditions at the data source.

A. Methodology

Testbed setup. For experiments in Section VIII-B, VIII-C and VIII-D, we deploy our data source on Amazon EC2 t2.micro nodes, each with one Intel Xeon E5-2676 core operating at 2.4GHz and 1 GB RAM running Ubuntu 16.04. For the experiment of multiple queries in Section VIII-E, we exercise 2 data source cores by deploying Amazon EC2 t2.medium node with two Intel Xeon E5-2686 cores operating at 2.4 GHz and 4 GB RAM, running Ubuntu 16.04. A stream processor instance is deployed on an Amazon EC2 m5a.16xlarge node with 64 AMD EPYC 7000 cores operating at 2.5GHz, with hyper-threading disabled, and 256 GB of memory running Ubuntu 16.04. Data source nodes are connected to stream processor with 12 Gbps Ethernet link. We use *cpulimit* [58] to control the available compute budget for a query running on a data source node and *wondershaper* [59] to shape the effective uplink bandwidth of a single data source node.

We conduct our experiments on two types of setups: (i) a single data source node connected to a single stream processor node to evaluate performance of partitioning and adaptation strategies in Jarvis, and (ii) multiple data sources (up to 250 nodes) connected to a single stream processor node to evaluate Jarvis as we scale number of data sources. For each experiment, the first three minutes are used to warm up the system and the next five minutes are used to obtain performance results.

Performance metrics. We measure *query processing throughput* in Mbps (megabit per second) with a latency bound of 5 seconds, *epoch processing latency* in seconds, and *convergence duration* in number of epochs after resource conditions change.

Workloads. We use two datasets: (i) **Pingmesh** dataset as described in Section **??** and (ii) text-based logs **LogAnalytics** which includes tenant name, job running time in milliseconds along with CPU and memory utilization for handling tenant-wise performance issues for jobs running in an analytics cluster. We run the following queries:

- **S2SProbe** (Listing 1) that runs on Pingmesh dataset, with the filter predicate delivering 14% filter-out rate.
- **T2TProbe** (Listing 2) that runs on Pingmesh dataset. It uses join operator to measure network latency aggregates for ToR-to-ToR pairs by joining the input stream with a table that maps server IP address to its ToR switch ID.
- **LogAnalytics** (Listing 3) that runs on LogAnalytics dataset. It parses unstructured logs to extract per-tenant latency and resource utilization. It then bucketizes the data to output histograms for tenant-wise job latencies and resource utilization.

```
Stream
.Window(10_SECONDS).Filter(e => e.errorCode
    == 0)
.Join(m, e => e.srcIp, m => m.ipAddr, (e,m)
    => (e, srcTor=m.torId))
.Join(m, e => e.dstIp, m => m.ipAddr, (e,m)
    => (e, dstTor=m.torId))
.GroupApply((e.srcToR, e.dstToR))
.Aggregate(c => c.avg(rtt) && c.max(rtt) && c
    .min(rtt))
```

Listing 2: A temporal query for ToR-to-ToR latency probing. m is a table to map server IP address to its ToR switch ID.

```
patterns={"*tenant name*", "*job running time
    *","*cpu util*","*memory util*"}
Stream
.Window(10_SECONDS)
.map(l -> l.trim().toLowerCase())
.filter(l -> patterns.stream().anyMatch(s->l.
    contains(s)))
.map(j -> new JobStats(j.split('=')))
.map(j -> j.stat = width_bucket(j.stat, 0,
    100, 10))
.groupapply(j.tenant_name,j.stat_name, j.stat
    )
.aggregate(c -> c.count())
```

Listing 3: A text query for computing histogram data for per-tenant analytics job latency and resource utilization. JobStats is an object to store job related information.

- S2SProbe, described in Listing 1, running on Pingmesh dataset, with the filter predicate delivering 14% filter-out rate.
- T2TProbe, described in Listing 2, running on Pingmesh dataset. It uses join operator to measure network latency aggregates for ToR-to-ToR pairs by joining the input stream with a table that maps server IP address to its ToR switch ID.
- LogAnalytics, described in Listing 3, running on LogAnalytics dataset. It parses unstructured logs to extract per-tenant latency and resource utilization information. It then bucketizes the data to generate histograms for tenant-wise job latencies and resource utilization.

For Pingmesh, we assume latency probing in a large-scale datacenter where more frequent probing occurs to include enough data for statistical significance. Specifically, guided by [2], we set each server to probe 20K other servers at a time interval of 5 seconds. As a probe record is 86B, each server generates data approximately at 2.62 Mbps. For LogAnalytics, guided by [11] which reports text log data generated at 10s

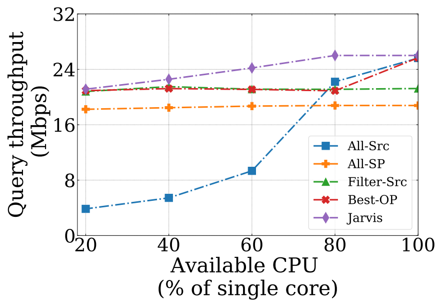


Fig. 10: S2SProbe throughput plot.

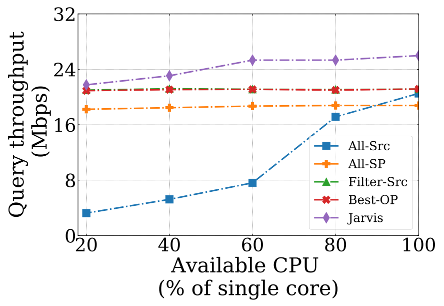


Fig. 11: T2TProbe throughput plot. Stream joined with table of size 500.

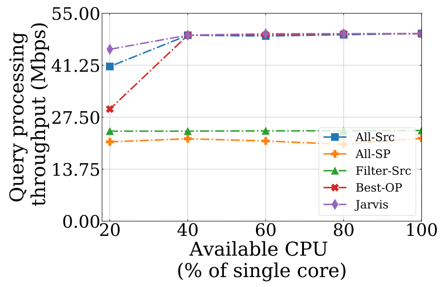


Fig. 12: LogAnalytics throughput plot.

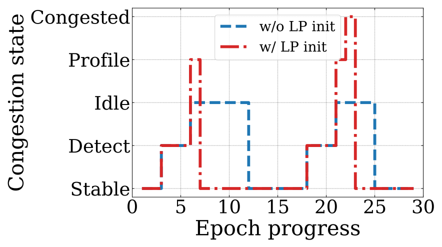


Fig. 13: S2SProbe convergence.

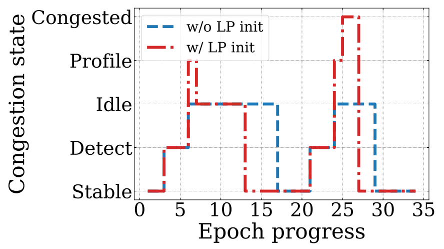


Fig. 14: T2TProbe convergence.

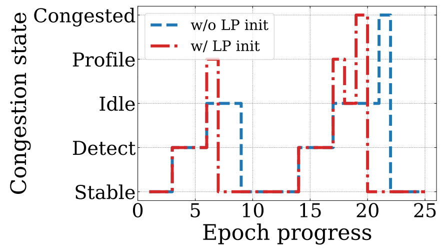


Fig. 15: LogAnalytics convergence.

PB per day across 200K data source nodes in a production system, we set each server to generate 0.62 MBps (or 4.96 Mbps) of log data. For experimentation purpose, we scale up the data generation rate by 10x, i.e., 26.2 Mbps for Pingmesh and 49.6 Mbps for LogAnalytics per data source node.

Network configuration. The number of data sources supported by a single stream processor node depends on the compute and network resources available on stream processor. Based on conversations with engineers at a large scale datacenter operator, we find that monitoring pipelines can typically have up to 250 data source nodes sending log data to a single stream processor node. And backed by estimates in [8], we assume we have around 20 monitoring queries that concurrently run on each data source. We also assume that a stream processor node would have a network link of 10 Gbps [16]. For ease of experiment, we assume this bandwidth is fairly utilized across 250 nodes and 20 queries per node, allowing 2.048 Mbps effective bandwidth per query per data source node. We again scale up the obtained bandwidth by 10x to match with data rate scaling.

Baseline systems. We compare Jarvis with various baseline systems, including (i) **All-SP** that runs a query entirely on stream processor node, (ii) **All-Src** that runs a query entirely on data source (i.e., Gigascope [17]), (iii) **Filter-Src** that applies static operator-level partitioning and runs only filter operations on the data source node (i.e., Everflow [16]), and (iv) **Best-OP** that applies dynamic operator-level partitioning and runs the best part of query obtained from a query planner on the data source node (i.e., Sonata [8]).

B. Query Throughput Analysis

We use the single data source setup to evaluate query throughput for different partitioning strategies. As compute may be shared with other queries, we perform a sensitivity analysis by varying the available compute between 0-100% of a single core. We note that Jarvis incurs little overhead, consuming less than 1% of a core while profiling operator states and making adaptation decisions.

S2SProbe query. Figure 10 shows the results of query throughput on S2SProbe. Jarvis outperforms other techniques in the 40-80% CPU budget range, with throughput gains of 2.6x over All-Src at the 60% budget and 1.25x over Best-OP at the 80% budget. The query requires nearly 85% CPU to execute entirely on data source. Thus, All-Src cannot meet the processing rate requirements at budgets lower than 80% CPU. At 80% CPU, Best-OP hits compute bottleneck and runs G+R operator on stream processor, thus making it bounded by the network. Best-OP executes F and G+R operators on data source only at 100%.

Jarvis partially processes the input of the G+R operator within available compute to reduce network traffic. In comparison, All-SP is restricted by available network bandwidth, and thus its throughput does not change with available CPU. Filter-Src always processes F at the source as its compute cost is only around 13%. However, it is still limited by network bandwidth as F is not effective in filtering out data.

T2TProbe query. Figure 11 shows the results of query throughput on T2TProbe. This query requires more than a core to execute due to expensive J operator. Thus, All-Src cannot handle the input rate even at 100% CPU, and its throughput declines drastically as CPU budget lowers further.

Jarvis does data-level partitioning on the query to process the input partially on J operator, thereby reducing network traffic, outperforming other techniques in the 40-100% CPU range. Jarvis performs 4.4x better than All-Src at 40% CPU and 1.2x better than Best-OP between 60-100% CPU range. Note that J operator is followed by a project on the fields srcToR, destToR and rtt, so output size of project is less than input size of J operator, leading to data reduction. Both Filter-Src and Best-OP execute only F at the source while Best-OP cannot accommodate J operator even at 100% CPU.

LogAnalytics query. LogAnalytics is relatively cheaper and uses 31% CPU to process the input at 49.6 Mbps. Nonetheless, Jarvis does data-level partitioning to reduce compute cost at 20% CPU budget, outperforming Best-OP by 1.5x. All-Src shows lower throughput than Jarvis at 20% CPU as it is resource constrained. Filter-Src executes filtering on data source, but is bound by network cost due to low filter-out rate. On the contrary, Best-OP can perform the filter and map operators at the source, thus outperforming Filter-Src. All-SP is always bound by the network, and hence Jarvis outperforms it by 2.3x in the 40-100% CPU range.

Summary. Our results show Jarvis provides higher throughput (1.2-4.4x) in scenarios where compute resources are constrained on data source. The throughput gains are higher when query is compute- or network-bound.

C. Convergence Analysis

We evaluate how fast Jarvis adapts to changes in resource conditions on data source by measuring the convergence time in number of epochs. We compare convergence time of StepWise-Adapt in two different configurations: (i) *with LP init* which initializes load factors using the LP solver and (ii) *without LP init* which initializes load factors to zero. We show that Jarvis converges in a few one-second epochs after resource change occurs. On the contrary, Sonata [8] may take several minutes to compute a new query plan.

S2SProbe query. As shown in Figure 13, We vary the available compute on the data source by starting with 10% CPU, then switching to 90% CPU in the 3^{rd} epoch, and finally reducing CPU down to 60% of the CPU to cause congestion in the 18^{th} epoch. Note that three epochs are required to detect that compute budget has changed to avoid triggering adaptation due to scheduling noise in the system.

When the budget is increased from 10% to 90% CPU, Jarvis reduces the convergence time from six epochs down to one epoch when employing initialization using LP (With LP-init). When CPU drops from 90% to 60%, the query reaches a stable state within two and four epochs with and without LP initialization, respectively. The additional epoch for the LP-Init is required because profiling within a one-second epoch is not sufficient for G+R to process all records, resulting in less accurate estimates for the cost of G+R.

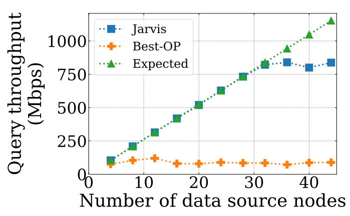
T2TProbe query. Performance of a join-bound query is affected by the size of the static table. As shown in Figure 14, we vary the available compute and the size of the static table by starting with 10% CPU and a static table of size 50, then switching to 100% CPU in the 3^{rd} epoch, and finally increasing the static table size by 10x to cause congestion.

Jarvis reduces convergence duration from 11 epochs (without LP init) to 7 epochs (with LP init) when the budget is increased to 100%. The high convergence of the LP initialization is attributed to the fact that the expensive J operator cannot be executed on all records to get accurate profiling estimates. As a result, the downstream G+R operator is not profiled accurately. Thereafter, fine-tuning plays a critical role in stabilizing their load factors. When join table size increases, the compute cost of J operator increases leaving no resources for G+R to execute. It takes five and three epochs to converge without LP init and with LP init.

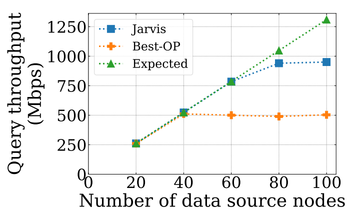
LogAnalytics query. We evaluate Jarvis while varying input rates and fixing compute budget to 25% CPU. We start with an input rate of 49.6 Mbps causing congestion as the query requires 31% CPU. We then reduce the rate by almost half, to 25 Mbps causing the query to idle. We finally increase the rate up to 80 Mbps. Figure 15 shows that when input rate drops to 25% in the 3^{rd} epoch, StepWise-Adapt converges immediately with LP init while it requires three epochs to converge without LP init. When the input rate is increased to 80 Mbps in the 14^{th} epoch, the query takes five epochs to converge without LP init, and with LP init, it takes only three epochs to converge.

Simulator-based analysis. We further analyze the potential benefit of LP solver in terms of convergence cost as we increase the number of query operators. We simulate the query execution guided by the execution behavior captured by query refinement problem in Section VI-D1. By varying different parameters in the query refinement problem formulation, we obtain a range of possible convergence costs for each value of M i.e. number of query operators. Figure 17 shows that the convergence time can go up to 21 epochs in the worst case for 4 or more operators. The curve levels off at about 21 epochs because we fixed the input rate. Taking 21 epochs to converge is not fully satisfactory if the goal is to adapt to resource changes in the order of a few seconds. This analysis shows that using only fine-tuning in StepWise-Adapt with fixed load factor initialization can incur a high convergence cost with larger M, indicating that LP solver is indispensable in StepWise-Adapt especially when a given query includes a number of operators.

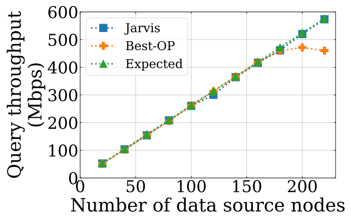
Summary. On one hand, a model-based approach relying on LP solver to determine load factor is quite effective in reducing convergence time if an accurate profiling of the operators can be obtained. On the other hand, fine-tuning is critical to stabilize the query if the profiling is not accurate. Previous model-based approaches [8] cannot fully rely on online profiling as they do not have a fine-tuning mechanism to stabilize the query. Our hybrid approach achieves convergence within a few epochs for various resource conditions.



(a) Input rate scaled up by 10x.



(b) Input rate scaled up by 5x.



(c) Input rate with no scaling.

Fig. 16: Throughput for multiple data source nodes at different input rates

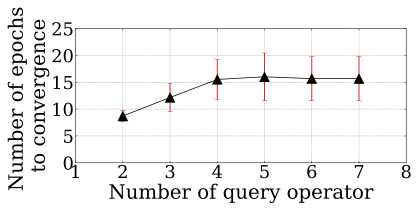


Fig. 17: Convergence time in number of epochs as we increase number of query operators. 150K records per epoch.

D. Multiple Data Source Nodes

We discuss the efficacy of Jarvis when multiple data sources are sending data to a single stream processor node. We compare Jarvis and Best-OP, i.e., the state-of-the-art in dynamic operator-level partitioning, on S2SProbe query while varying the following dimensions: (i) number of data source nodes and (ii) input data rate per data source node.

We start with the input rate of 26.2 Mbps for Pingmesh, which is scaled by 10x over the calculated rate for this dataset. On each data source, we set CPU to 55% to ensure that Best-OP executes only the F operator while not fully utilizing the given CPU budget. Figure 16(a) shows the results as we increase number of data sources. In Best-OP, F operator does not reduce data significantly so the policy suffers from network bottleneck as soon as we increase the number of data sources. However, Jarvis scales up to 32 nodes without throughput degradation. Further scaling stops due to network bottleneck while peak compute usage on stream processor was at 24 cores. Beyond 32 nodes, we observe backpressure which causes epoch processing latency to grow, and number of data sources for which no data is being received on stream processor increases.

We now decrease input rate to 13.1 Mbps (5x scaling) on each data source and set available CPU to 30%, to reflect the change in query compute demand from decreasing input rate. Figure 16(b) shows that Best-OP scales to 40 nodes after which it becomes network bottlenecked. Jarvis scales up to nearly 70 nodes, 75% improvement in number of data sources supported over Best-OP. When we further reduce input rate to 2.62 Mbps in Figure 16(c) and allocate 5% CPU to the query, Best-OP starts to degrade in throughput at 180 nodes while Jarvis is seen to scale even for 250 data sources, an 39% improvement.

Epoch processing latency. Jarvis also improves epoch processing latency as a result of network traffic reduction. Based on the observed epoch latency distributions, we note for example, that Jarvis improves median latency of Best-OP by around 3.4x in the configuration with 5x scaling and 40 data sources, when both policies are able to handle the input data rate. Moreover, Best-OP exhibits 1.8 and 5.2 seconds for the median and max latency, respectively, which are much larger than 500 ms and 2 seconds for the two measures in Jarvis. For configurations where Best-OP could not keep up with input data after it hits network bottleneck, such as 5x scaling with 60 nodes, we see max latency of Jarvis to be within 5 seconds while Best-OP latency grows beyond 60 seconds.

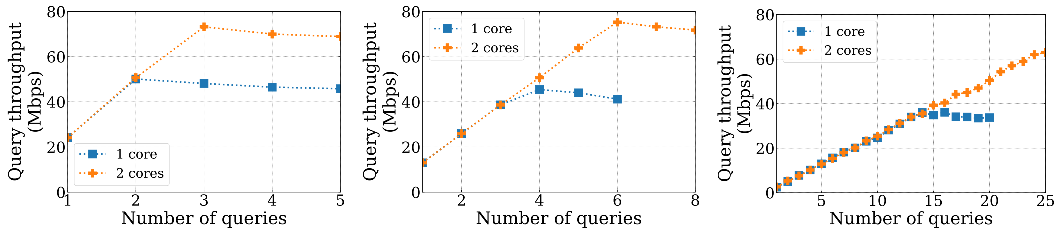
E. Multiple Queries on Data Source Node

The goal of this experiment is to investigate potential interference when we execute multiple queries on a data source node with Jarvis. For this experiment, we measure *aggregate* query throughput as we increase the number of S2SProbe queries on a single data source. Similar to Section VIII-D, we restrict the total memory capacity to 1 GB, while fixing load factors so the query utilizes 55% CPU for input rate of 26.2 Mbps (i.e., 10x scaled). We vary the following dimensions: (1) input data rate per query and (2) number of cores on data source. Figure 18 shows the results.

Under system stress at 10x input scaling, single-core throughput saturates at 2 queries given per-query CPU cost. Two-core throughput does not increase beyond 3 queries, indicating saturation at 70 Mbps. At 5x scaling, per-query CPU cost drops to 30% and Jarvis supports up to four and six queries on a single- and two-core setup, respectively. At no input scaling, Jarvis supports 15 queries and 25 queries with one and two cores, respectively. We observe that there is no significant interference from resources such as memory bandwidth until the system is bottlenecked by either compute or network. Furthermore, with query partitioning, as we add more queries, Jarvis can adapt to support more queries for a given compute budget.

IX. Related Work

Streaming query optimizations. Several runtime query optimizations, such as operator scaling [31], [60], [61], operator reordering [62]–[65] and resource allocation [66], [67] have been investigated for streaming systems which are complementary to our work. For operator placement, Turbo [33] and Sparrow [34] have looked at decentralized scheduling



(a) Input rate scaled up by 10x. (b) Input rate scaled up by 5x. (c) Input rate with no scaling.

Fig. 18: Throughput for multiple queries on data source, at different input rates

for batch processing. Several decentralized placement techniques for streaming [35]–[37], [41], [43], [68] focus on reducing network usage. Similarly, cost model-based query planners have been explored for placement [33]–[38], [38]–[43]. Cardellini, et.al. [42] proposed a decentralized scheme that combines query cost model and reinforcement learning to meet application throughput requirements for operator scaling. These works do not consider partitioning under highly variable data source resources.

Near-data stream computations. State-of-the-art approaches on alleviating network and stream processor resources target analyzing network packets and are mostly based on operator-level partitioning. EverFlow [16] and dShark [22] apply lightweight operations on commodity network switches and computing nodes near data source, respectively. Gigascope [17] classifies operators based on domain-specific knowledge and pushes low-cost operators close to data source. Sonata [8] utilizes an ILP solver to optimize operator-level partitioning, but at the cost of latency as high as 20 minutes. These approaches neglect data-level partitioning and/or are based on expensive centralized query refinement.

Stream computations on the edge. EdgeWise [69] is a streaming engine that incorporates a scheduler for congestion control to improve throughput and latency by prioritizing operators whose queues experience backpressure.

Wide-area streaming analytics. Novel streaming systems have been designed to operate efficiently on low-bandwidth networks, such as wide-area network (WAN). Sol [70] introduces an early-binding of analytics tasks to workers and dedicated communication tasks to improve resource utilization over WAN. AWStream [71] trades off accuracy for WAN bandwidth by controlling the sampling of data. MQO [72] provides a technique to share input/output data and operators across stream queries, saving WAN bandwidth.

X. Discussion

Scaling to more data sources. Jarvis optimizes a core building block, composed of a single stream processor node and a number of data sources as high as 250. As the number of data sources increases, Jarvis replicates the core building block, resulting in a set of intermediate stream processor nodes; each node aggregates results obtained from its data source nodes. A root node computes the final query result by aggregating results obtained from intermediate stream processor nodes. As there is no communication between building blocks, the system can scale better with improved scaling of the core building block and availability of upstream network bandwidth between intermediate nodes and the root node.

Fault tolerance. We are currently extending Jarvis to support recovery in case of node failures. When data source fails, no further data is generated from that node. Intermediate state accumulated by the data source for the current processing window can be periodically checkpointed by re-using the path used for draining data from control proxies. The stream processor node identifies data source failures (using heartbeats) and reads the latest checkpoint for the node and processes the remaining data for the current window. When the stream processor node fails, its processing state can be checkpointed using existing fault recovery techniques [73] so that a new node can restart processing from the last checkpoint. However, an external data store may need to be updated by stream processor so that data sources can read its last successful checkpoint. Data sources can then replay records as they contain the upstream query operators.

Internet-of-things (IoT) analytics. The techniques investigated in Jarvis can be applied to IoT scenarios where billions of devices can sense, communicate, compute and potentially actuate. IoT scenarios exhibit similar constraints as the ones assumed in our work, mainly network bandwidth (e..g, WAN) and limited compute and power resources.

XI. Conclusion

We presented Jarvis, a fully decentralized data-level query partitioning engine for server monitoring systems. Our analysis using real-world monitoring query workloads showcased that Jarvis substantially improves the system's throughput while adapting to changes in resource conditions within seconds.

Appendix A
Compute cost of optimal query partitioning

Consider the hierarchical architecture of resource nodes in Figure 6 where each node in the tree topology can communicate with its parent node. A query with M operators are to be executed on monitoring stream being generated on data sources. Each data source has a query instance which can be represented as a computational pipeline of operators as vertices and edges denoting dataflow dependencies. On the stream processor side, a fixed number of compute nodes are provisioned for processing the query on data from all data sources. Each data source needs to identify operators which need to be executed locally vs. those that require remote execution on parent node. Due to dataflow dependencies, we can $partition$ the query graph at $M + 1$ possible locations in the graph i.e. a partition splits the operators into two sets. Operators before the partition point are executed on data source and those after the partition are executed remotely. As an example, for a data source i with $M = 4$ operators, $p_i \epsilon [0, M]$ represents the operator where partition occurs. So $p_i = 3$ means first 3 operators execute on data source and the last operator executes remotely.

Compute resources on stream processor are used to process operators sent for remote execution after partitioning. Data transport also incurs a network overhead cost. Our goal is to parallelize query processing between compute resources on data source and stream processor nodes, while minimizing the utilization of finite compute and network resources on stream processor. As we increase the number of data sources, processing load increases on the system. And the processing load from different data sources can result in network/compute bottleneck on stream processor, hurting query processing time. Thus, we cannot independently partition operators for each data source, instead joint partitioning decision needs to be made across data sources. When the remote resources are saturated, it is more beneficial for the data source to execute operators locally to avoid long processing time.

Let **p** denote the partitioning profile for a query, where p_i denotes query partition point for i^{th} data source. We now define the following query partitioning problem as follows:

$$\min_{\mathbf{p}} \sum_{i=1}^{N_d} \sum_{j=1}^{M} c_j x_{ij}$$

$$subject\ to,$$

$$T_{local}(i, p_i) <= T_{remote}(i, p_i)\ \forall p_i > 0, i \epsilon [1, N_d]$$

Here x_{ij} indicates if partition point of i^{th} data source is j, c_j is the processing cost on stream processor due to partitioning the query on data source at j^{th} operator, $T_{local}(i, p_i)$ computes the local computation time cost for operators 1 to p_i on the i^{th} data source node and $T_{remote}(i, p_i)$ denotes the network transmission time cost of intermediate data from p_i^{th} operator to its parent node along with computation time cost of executing operators $p_i + 1$ till M on the parent node. We want to incentivize executing operators on data source so the partitioning costs are ordered as $c_1 > c_2 >, ... > c_M$.

Unfortunately, solving the partitioning problem is extremely challenging.

Theorem 1. Query partitioning problem to distribute query operators across data source and stream processor nodes, with the objective of maximizing the resource utilization of data source nodes without degrading query processing time, is NP-hard.

Proof. We introduce the generalized assignment problem (GAP) [74] which finds a minimum cost assignment of n items to m bins such that each item is assigned to precisely one bin subject to capacity restrictions on the bin. Following is the definition:

$$min\ \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}$$

$$subject\ to,$$

$$\sum_{i=1}^{m} x_{ij} = 1, j \epsilon \{1, ..., n\},$$

$$\sum_{j=1}^{n} w_{ij} x_{ij} \le b_i, i \epsilon \{1, ..., m\},$$

$$x_{ij} \epsilon 0, 1, i \epsilon \{1, ..., m\}, j \epsilon \{1, ..., n\}$$

where c_{ij} is the cost associated with assigning item j to bin i, w_{ij} the claim on the capacity of bin i by item j if it is assigned to bin i, b_i the capacity of bin i and x_{ij} a 0-1 variable indicating whether item j is assigned to bin i ($x_{ij} = 1$) or not ($x_{ij} = 0$). This problem is known to be NP-hard [74].

A data source n can execute operators remotely with partitioning p_n while improving query processing time, if and only if resource usage on stream processor due to remaining data sources is below a threshold data rate D_n i.e. $\sum_{i \epsilon [1,N] \setminus \{n\} : p_i = p_n} d_i(p_i) < D_n$. Here, $d_i(p_i)$ is a function that provides rate of data leaving data source i given the partitioning p_i and D_n is the threshold data rate for all the remaining nodes with partitioning p_n so that data source n with p_n has enough resources to execute operators remotely. We use the data rate as a measure for the network and compute overhead on the stream processor. Based on this, we transform generalized assignment problem to a special case of our problem of configuring minimum number of data sources to execute operators remotely. We can regard the items and bins in GAP as the data source nodes and possible partitioning values i.e. possible values in **p** in our problem, respectively. Then the weight of an item n assigned to bin m is $w_{mn} = d_n(p_n)$ where $m = p_n$, capacity constraint of each bin $b_m = D_n + d_n(p_n)$ and cost of each assignment $c_{mn} = c_{p_n}$. Here, $c_{n1} > ... > c_{nM}$. By this, we can ensure that as long as data source n on its assigned partition p_n is able to improve processing time compared to executing the query locally, total size of items assigned to bin p_n will not violate capacity constraint b_{p_n}. This is because $\sum_{i \epsilon [1,N] \setminus \{n\} : p_i = p_n} d_i(p_i) < D_n$ which implies $\sum_{j=1}^{n} w_{nj} x_{nj} = \sum_{i \epsilon [1,N] \setminus \{n\} : p_i = p_n} d_i(p_i) + d_n(p_n) < b_{p_n}$.

Therefore, if we have an algorithm that can minimize the number of operators sent to the stream processor without sacrificing processing time, then we can also obtain optimal solution to GAP. Since GAP is NP-hard, our problem is also NP-hard. □

The key idea of the proof is that GAP can be reduced to a special case of our query partitioning problem. We note that previous works have also established the hardness of optimal query partitioning in other related domains [75], [76]. In this paper, we investigate a greedy heuristic approach which is embarrassingly parallel, for making partitioning decisions. We enable the potential latency overhead resulting from our greedy solution to be bound by an upper limit, by configuring the processing duration or epoch for making partitioning decisions. Our approach is implemented in a fully decentralized manner to ensure high scalability of our partitioning system.

References

[1] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the Datacenter: Automated Classification of Performance Crises," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 111–124. [Online]. Available: https://doi.org/10.1145/1755913.1755926

[2] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 139–152. [Online]. Available: https://doi.org/10.1145/2785956.2787496

[3] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang, "NetBouncer: Active Device and Link Failure Localization in Data Center Networks," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 599–614. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/tan

[4] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, M. Chintalapati, A. Krishnamurthy, and T. Anderson, "Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure," in *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'18. USA: USENIX Association, 2018, p. 519–532.

[5] S. Karumuri, F. Solleza, S. Zdonik, and N. Tatbul, "Towards observability data management at scale," *SIGMOD Rec.*, vol. 49, no. 4, p. 18–23, Mar. 2021. [Online]. Available: https://doi.org/10.1145/3456859.3456863

[6] S. Das, F. Li, V. R. Narasayya, and A. C. König, "Automated Demand-Driven Resource Scaling in Relational Database-as-a-Service," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1923–1934. [Online]. Available: https://doi.org/10.1145/2882903.2903733

[7] Uber, "The Billion Data Point Challenge: Building a Query Engine for High Cardinality Time Series Data," 2018, https://eng.uber.com/billion-data-point-challenge/.

[8] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-Driven Streaming Network Telemetry," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 357–371. [Online]. Available: https://doi.org/10.1145/3230543.3230555

[9] R. Potharaju, T. Kim, W. Wu, V. Acharya, S. Suh, A. Fogarty, A. Dave, S. Ramanujam, T. Talius, L. Novik, and R. Ramakrishnan, "Helios: Hyperscale indexing for the cloud & edge," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 3231–3244, Aug. 2020. [Online]. Available: https://doi.org/10.14778/3415478.3415547

[10] Z. Chothia, J. Liagouris, D. Dimitrova, and T. Roscoe, "Online Reconstruction of Structural Information from Datacenter Logs," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 344–358. [Online]. Available: https://doi.org/10.1145/3064176.3064195

[11] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa, S. Dhulipalla, and S. Rao, "Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems," *Proc. VLDB Endow.*, vol. 11, no. 10, p. 1303–1316, Jun. 2018. [Online]. Available: https://doi.org/10.14778/3231751.3231765

[12] S. Srivatsan, "Observability at Scale: Building Uber's Alerting Ecosystem," 2018, https://eng.uber.com/observability-at-scale/.

[13] Nutanix, "Nutanix Agent - collector agent," 2021, https://docs.epoch.nutanix.com/setup-guide/collectors/configuration/.

[14] W. Song, Z. Xiao, Q. Chen, and H. Luo, "Adaptive Resource Provisioning for the Cloud Using Online Bin Packing," *IEEE Trans. Comput.*, vol. 63, no. 11, p. 2647–2660, Nov. 2014. [Online]. Available: https://doi.org/10.1109/TC.2013.148

[15] X. Sun, N. Ansari, and R. Wang, "Optimizing Resource Utilization of a Data Center," *Commun. Surveys Tuts.*, vol. 18, no. 4, p. 2822–2846, Oct. 2016. [Online]. Available: https://doi.org/10.1109/COMST.2016.2558203

[16] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, "Packet-Level Telemetry in Large Datacenter Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, p. 479–491, Aug. 2015. [Online]. Available: https://doi.org/10.1145/2829988.2787483

[17] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: A Stream Database for Network Applications," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 647–651. [Online]. Available: https://doi.org/10.1145/872757.872838

[18] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, "Aggregation and degradation in jetstream: Streaming analytics in the wide area," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 275–288. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/rabkin

[19] M. Jeon, S. Venkataraman, A. Phanishayee, u. Qian, W. Xiao, and F. Yang, "Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 947–960.

[20] R. Chaiken, B. Jenkins, P.-r. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: Easy and efficient parallel processing of massive data sets," *Proc. VLDB Endow.*, vol. 1, no. 2, p. 1265–1276, Aug. 2008. [Online]. Available: https://doi.org/10.14778/1454159.1454166

[21] Z. Li, Q. Cheng, K. Hsieh, Y. Dang, P. Huang, P. Singh, X. Yang, Q. Lin, Y. Wu, S. Levy, and M. Chintalapati, "Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Large-Scale Cloud Infrastructure," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 389–402. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/li

[22] D. Yu, Y. Zhu, B. Arzani, R. Fonseca, T. Zhang, K. Deng, and L. Yuan, "dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 207–220. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/yu

[23] G. Pekhimenko, C. Guo, M. Jeon, P. Huang, and L. Zhou, "TerseCades: Efficient Data Compression in Stream Processing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 307–320. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/pekhimenko

[24] 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. [Online]. Available: http://sites.computer.org/debull/A15dec/p28.pdf

[25] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 423–438. [Online]. Available: https://doi.org/10.1145/2517349.2522737

[26] H. Miao, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, "StreamBox-HBM: Stream Analytics on High Bandwidth Hybrid Memory," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association

for Computing Machinery, 2019, p. 167–181. [Online]. Available: https://doi.org/10.1145/3297858.3304031

[27] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing, "Trill: A High-Performance Incremental Query Processor for Diverse Analytics," *Proc. VLDB Endow.*, vol. 8, no. 4, p. 401–412, Dec. 2014. [Online]. Available: https://doi.org/10.14778/2735496.2735503

[28] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast Analytical Power Management for Latency-Critical Systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 598–610. [Online]. Available: https://doi.org/10.1145/2830772.2830797

[29] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards Energy Proportionality for Large-Scale Latency-Critical Workloads," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. USA: IEEE Press, 2014, p. 301–312.

[30] G. Lim, M. S. Hassan, Z. Jin, S. Volos, and M. Jeon, "Approximate quantiles for datacenter telemetry monitoring," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. USA: IEEE, 2020, pp. 1914–1917.

[31] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three Steps is All You Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 783–798.

[32] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 69–80. [Online]. Available: https://doi.org/10.1145/2933267.2933312

[33] H. Wang, D. Niu, and B. Li, "Turbo: Dynamic and decentralized global analytics via machine learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1372–1386, 2020.

[34] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 69–84. [Online]. Available: https://doi.org/10.1145/2517349.2522716

[35] 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)*, 2006, pp. 49–49.

[36] S. Rizou, F. Dürr, and K. Rothermel, "Solving the multi-operator placement problem in large-scale operator networks," in *2010 Proceedings of 19th International Conference on Computer Communications and Networks*, 2010, pp. 1–6.

[37] Q. Zhu and G. Agrawal, "Resource allocation for distributed streaming applications," in *2008 37th International Conference on Parallel Processing*, 2008, pp. 414–421.

[38] M. Luthra, B. Koldehofe, P. Weisenburger, G. Salvaneschi, and R. Arif, "Tcep: Adapting to dynamic user environments by enabling transitions between operator placement mechanisms," 06 2018.

[39] F. Starks and T. P. Plagemann, "Operator placement for efficient distributed complex event processing in manets," in *2015 IEEE 11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, 2015, pp. 83–90.

[40] S. Bokhari, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system," *IEEE Transactions on Software Engineering*, vol. SE-7, no. 6, pp. 583–589, 1981.

[41] Y. Ahmad and U. Çetintemel, "Network-aware query processing for stream-based applications," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB '04. VLDB Endowment, 2004, p. 456–467.

[42] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, "Decentralized self-adaptation for elastic data stream processing," *Future Generation Computer Systems*, vol. 87, pp. 171–185, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17326821

[43] B. J. Bonfils and P. Bonnet, "Adaptive and decentralized operator placement for in-network query processing," in *Information Processing in Sensor Networks*, F. Zhao and L. Guibas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 47–62.

[44] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire, "Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 221–230. [Online]. Available: https://doi.org/10.1145/3183713.3190662

[45] Prometheus, "Prometheus-estimate percentiles from histograms." 2021, https://prometheus.io/docs/practices/histograms/.

[46] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica, "Drizzle: Fast and adaptable stream processing at scale," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 374–389. [Online]. Available: https://doi.org/10.1145/3132747.3132750

[47] B. Radunović and J.-Y. L. Boudec, "A Unified Framework for Max-Min and Min-Max Fairness with Applications," *IEEE/ACM Trans. Netw.*, vol. 15, no. 5, p. 1073–1083, Oct. 2007. [Online]. Available: https://doi.org/10.1109/TNET.2007.896231

[48] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[49] G. Dósa, R. Li, X. Han, and Z. Tuza, "Tight absolute bound for first fit decreasing bin-packing: Ffd(l)¡=11/9 opt(l)+6/9," *Theoretical Computer Science*, vol. 510, pp. 13–61, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397513006774

[50] A. M. Peterson E. L., *Geometric Programming*. Boston, MA: Springer US, 1980, pp. 31–94. [Online]. Available: https://doi.org/10.1007/978-1-4615-8285-4_3

[51] L. V. Martin Andersen, Joachim Dahl, "Cvxopt: Python software for convex optimization," May 2015, http://cvxopt.org/.

[52] A. S. Foundation, "A subproject of Apache NiFi to collect data where it originates," 2018, https://nifi.apache.org/minifi/.

[53] ——, "An easy to use, powerful, and reliable system to process and distribute data," 2018, https://nifi.apache.org/.

[54] Esoteric, "Java binary serialization and cloning: fast, efficient, automatic," 2020, https://github.com/EsotericSoftware/kryo/.

[55] A. Flink, "Apache Flink - Timely Stream Processing," 2020, https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/concepts/time/.

[56] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "TimeStream: Reliable Stream Computation in the Cloud," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 1–14. [Online]. Available: https://doi.org/10.1145/2465351.2465353

[57] W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou, "STREAMSCOPE: Continuous Reliable Distributed Processing of Big Data Streams," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16. USA: USENIX Association, 2016, p. 439–453.

[58] G. Herrmann, "CPU Usage Limiter for Linux," 2020, http://cpulimit.sourceforge.net/.

[59] S. S. Bert Hubert, Jacco Geul, "Command-line utility for limiting an adapter's bandwidth," 2020, https://github.com/magnific0/wondershaper/.

[60] L. Xu, B. Peng, and I. Gupta, "Stela: Enabling stream processing systems to scale-in and scale-out on-demand," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, 2016, pp. 22–31.

[61] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, "Dhalion: Self-regulating stream processing in heron," *Proc. VLDB Endow.*, vol. 10, no. 12, p. 1825–1836, Aug. 2017. [Online]. Available: https://doi.org/10.14778/3137765.3137786

[62] R. Avnur and J. Hellerstein, "Eddies: Continuously adaptive query processing," vol. 29, 06 2000.

[63] K. T. Claypool and M. Claypool, "Teddies: Trained eddies for reactive stream processing," in *Database Systems for Advanced Applications, 13th International Conference, DASFAA 2008, New Delhi, India, March 19-21, 2008. Proceedings*, ser. Lecture Notes in Computer Science, J. R. Haritsa, K. Ramamohanarao, and V. Pudi, Eds., vol. 4947. Springer, 2008, pp. 220–234. [Online]. Available: https://doi.org/10.1007/978-3-540-78568-2_18

[64] V. Raman, B. Raman, and J. Hellerstein, "Online dynamic reordering for interactive data processing," in *VLDB*, 1999.

[65] S. Ch, O. Cooper, A. Deshp, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah, "Telegraphcq: Continuous dataflow processing for an uncertain world," in *In First Biennial Conference on Innovative Data Systems Research (CIDR*, 2003.

[66] L. Xu, S. Venkataraman, I. Gupta, L. Mai, and R. 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)*. USENIX Association, Apr. 2021, pp. 389–405. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/xu

[67] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure, "Adaptive control of extreme-scale stream processing systems," in *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, 2006, pp. 71–71.

[68] V. Kumar, B. Cooper, and K. Schwan, "Distributed stream management using utility-driven self-adaptive middleware," in *Second International Conference on Autonomic Computing (ICAC'05)*, 2005, pp. 3–14.

[69] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, "Edgewise: A Better Stream Processing Engine for the Edge," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 929–945.

[70] F. Lai, J. You, X. Zhu, H. V. Madhyastha, and M. Chowdhury, "Sol: Fast Distributed Computation Over Slow Networks," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 273–288. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/lai

[71] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee, "AWStream: Adaptive Wide-Area Streaming Analytics," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 236–252. [Online]. Available: https://doi.org/10.1145/3230543.3230554

[72] A. Jonathan, A. Chandra, and J. Weissman, "Multi-Query Optimization in Wide-Area Streaming Analytics," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 412–425. [Online]. Available: https://doi.org/10.1145/3267809.3267842

[73] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 725–736. [Online]. Available: https://doi.org/10.1145/2463676.2465282

[74] M. Savelsbergh, "A branch-and-price algorithm for the generalized assignment problem," *Operations Research*, vol. 45, pp. 831–841, 12 1997.

[75] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, 2016.

[76] B. Liu, X. Xu, L. Qi, Q. Ni, and W. Dou, "Task scheduling with precedence and placement constraints for resource utilization improvement in multi-user mec environment," *Journal of Systems Architecture*, vol. 114, p. 101970, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762120302174