DISSERTATION

NEAR REAL-TIME PROCESSING OF VOLUMINOUS, HIGH-VELOCITY DATA STREAMS FOR CONTINUOUS SENSING ENVIRONMENTS

Submitted by

Thilina Hewa Raga Munige

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2020

Doctoral Committee:

Advisor: Shrideep Pallickara

V. Chandrasekar
Sudipto Ghosh
Sangmi Pallickara

ABSTRACT

NEAR REAL-TIME PROCESSING OF VOLUMINOUS, HIGH-VELOCITY DATA STREAMS FOR CONTINUOUS SENSING ENVIRONMENTS

Recent advancements in miniaturization, falling costs, networking enhancements, and battery technologies have contributed to a proliferation of networked sensing devices. Arrays of coordinated sensing devices are deployed in continuous sensing environments (CSEs) where the phenomena of interest are monitored. Observations sensed by devices in a CSE setting are encapsulated as multidimensional data streams that must subsequently be processed. The vast number of sensing devices, the high rates at which data are generated, and the high-resolutions at which these measurements are performed contribute to the voluminous, high-velocity data streams that are now increasingly pervasive. These data streams must be processed in near real-time to power user-facing applications such as visualization dashboards and monitoring systems, as well as various stages of data ingestion pipelines such as ETL pipelines.

This dissertation focuses on facilitating efficient ingestion and near real-time processing of voluminous, high-velocity data streams originating in CSEs. Challenges in ingesting and processing such streams include energy and bandwidth constraints at the data sources, data transfer and processing costs, underutilized resources, and preserving the performance of stream processing applications in the presence of variable workloads and system conditions. Toward this end, we explore design principles to build a high-performant and adaptive stream processing engine to address processing challenges that are unique to CSE data streams. Further, we demonstrate how our holistic methodology based on space-efficient representations of data streams through a controlled trade-off of accuracy, can substantially alleviate stream ingestion challenges while improving the stream processing performance. We evaluate the efficacy of our methodology using real-world

streaming datasets in a large-scale setup and contrast against the state-of-the-art developments in the field.

ACKNOWLEDGEMENTS

I am deeply indebted to Dr. Shrideep Pallickara, my Ph.D. advisor, for his unparalleled support and guidance throughout the last five years. He went above and beyond his responsibilities as an advisor to ensure that I reached my maximum potential as a researcher.

My sincere gratitude goes to the faculty who served in my Ph.D. committee: Dr. V. Chandrasekar, Dr. Bruce Draper, Dr. Sudipto Ghosh, and Dr. Sangmi Pallickara. Their valuable insights were instrumental in shaping up my research.

Over the years, I had the great pleasure of collaborating with many humble and talented researchers from the Distributed Systems Group and Big Data Lab: Dr. Shrideep Pallickara, Dr. Sangmi Pallickara, and fellow student researchers present and past. I will always cherish the memories of our team efforts tackling exciting research problems. Many thanks to the faculty, staff, and fellow graduate students at the Computer Science Department for all their help and support throughout this period.

I'd like to acknowledge the funding agencies who supported my research: the US National Science Foundation (CNS-1253908, OAC-1931363, ACI-1553685), the Dept. of Homeland Security (D15PC00279), a Monfort Professorship, and a Cochran Family Professorship.

My Ph.D. would not have been a reality if not for my family: my parents, my brother, and my wife. They stood by me through thick and thin and believed in me even when I did not believe in myself. I very much appreciate the help and encouragement of my extended family. I am incredibly grateful to my friends, especially those who are in Fort Collins, for being there for me and for making the last few years joyful.

DEDICATION

For my parents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

By the year 2025, 41.6 billion networked devices are expected to be in operation, generating around 79.4 zettabytes of data [1]. Most of these devices are equipped with one or more sensing capabilities. A confluence of factors has contributed to a dramatic increase in the number of connected sensing devices. These contributing factors include, among other things, advances in miniaturization, falling costs, networking enhancements, and sustained improvements in the quality and capacity of battery technologies. These sensing devices are now deployed in *continuous sensing environments (CSEs)* where phenomena of interest are monitored at ever increasing precisions and frequencies. CSEs arise in settings such as Internet-of-Things (IoT), cyber-physical systems, and mission critical monitoring systems. Domains that CSEs have been deployed include atmospheric and ecological monitoring [2], traffic [3], environmental monitoring [4], health care [5, 6], and industrial equipment monitoring [7] among others. The number of sensing devices combined with high data generation rates produce *voluminous data streams*.

Sensors and actuators form the *physical layer* in a typical CSE architecture [8] as depicted in Figure 1.1. Sensors produce data periodically or when a certain phenomenon is detected. Edge devices may have on-board sensors whose measurements they report, which is the case with mobile phones and smart wearables such as activity trackers. Alternatively, the edge devices may act as *data mules* [9] where the data is pulled from the sensors periodically using limited-range, custom transport protocols such as PSFQ [10] and ESRT [11] used in sensor networks. The edge device consolidates readings from multiple sensors into a single observation capturing multiple aspects of a monitored entity at a given point in time. The vast majority of the data streams originating in such settings contain a logical, relative, or absolute temporal component [12]. The data streams originating in continuous sensing environments are unbounded datasets [13].

Next, the data is transferred to a central location, also known as the *data sink* or the *middleware layer*, for processing [9]. This process is called as *data ingestion*. Data ingestion usually

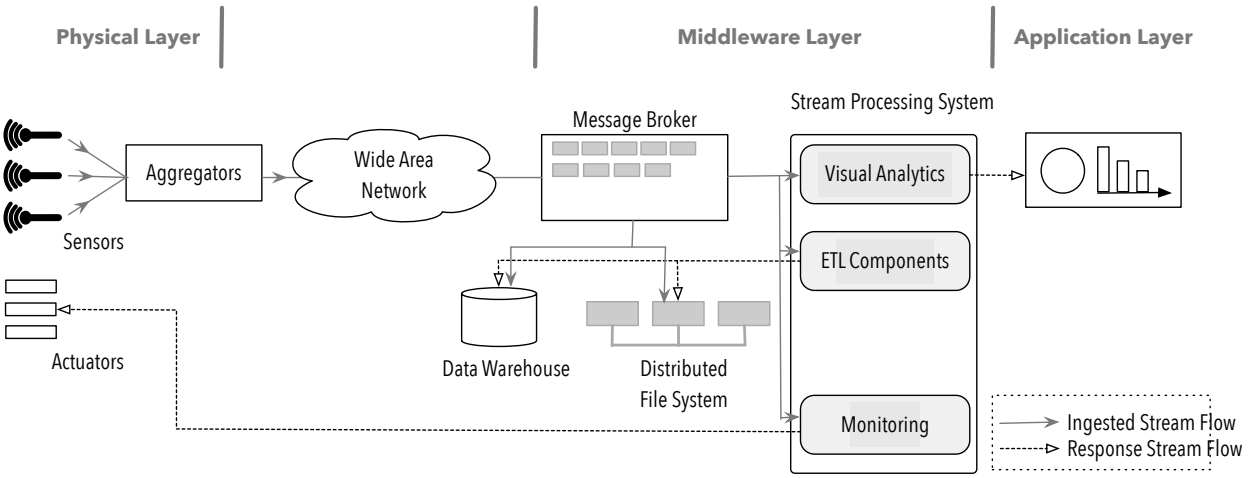


Figure 1.1: A typical data stream ingestion and processing architecture. Streams are originating at the edges of the network and transferred to the cloud via wide area networks. Stream processing systems executes multiple stream processing applications simultaneously powering various user-facing applications, monitoring applications, segments of ETL pipelines, etc.

takes place via a wide area network (WAN) [14] using TCP or machine-to-machine(M2M) protocols such as MQTT [15]. Transferred data is made available for processing using message queues. Message queues implement the publisher/subscriber architecture providing time, space, and synchronization decoupling between the stream publishers and consumers. These message queues are termed *cloud gateways* in some IoT reference architectures [8].

The transferred data can be either processed in near real-time as streams, or stored and subsequently processed as a batch. In this dissertation, we focus on near real-time processing of data streams using distributed stream processing systems. Example applications of stream processing systems include monitoring systems, visualization dashboards, extract-transform-load (ETL) pipelines, and anomaly detection systems. Stream processing systems are designed to run on commodity hardware — a stream processing engine orchestrates multiple processes (stream processing nodes) executing on a set of machines to provide a distributed runtime for multiple stream processing applications running in parallel. Stream processing applications are modeled as directed acyclic graphs (DAGs) of stream computations, each responsible for executing a portion of the application logic. Adjacent stream computations communicate with each other with uni-directional data flows from upstream nodes and bi-directional control traffic. Stream processing applications are

expected to be continuously executed. Unlike batch processing jobs where termination points are well-defined, stream processing applications are far more likely to be subjected to variability in their operating conditions.

1.1 Research Challenges

The challenges in near-real time processing of CSE data streams can be broadly categorized into two classes: *ingestion related* and *processing related*.

Stream ingestion challenges pertain to making data streams available to the middleware layer for processing — this may involve data transfer between different networks. Stream ingestion challenges include:

- *Power constraints:* Most edge devices are battery powered or have limited power profiles. Communication is the dominant source of energy consumption on these devices [16, 9, 17]; transferring voluminous data can become challenging.

- *Limited bandwidth:* Edge devices are connected to the remainder of the data ingestion pipeline via wide area networks with limited bandwidth [14]. Limited bandwidths increase data transfer times, which adversely impact latency-sensitive applications. As an alternative, bandwidths can be upgraded, but this can be both financially expensive and unsustainable in the long term due to increasing data volumes.

- *Data transfer costs:* When the processing middleware is operating in public clouds, users are billed for the volume of data transferred across data center network boundaries. This is in addition to the bandwidth costs for the data transfer as mentioned before.

- *Transient storage costs:* Once a data stream is transferred from the data sources, it gets stored in a message queue such as Apache Kafka [18] or Amazon Kinesis [19] before processing. Storage in queues provides temporal and spatial decoupling between the stream sources and the processing middleware [20]. Cloud users are billed separately for their message queue usage in addition to data transfer fees. For instance, in AWS, Amazon's Public Cloud, users are charged

for the number of shards used to store data in Amazon Kinesis and also for the number of PUT requests used to transfer data into the cloud.

Stream processing engines execute applications, encapsulating various use cases, to process the ingested streams. Stream processing applications are inherently distributed to scale horizontally with the volume of data. Challenges in distributed stream processing include:

- *Underutilized resources:* The amount of resources required for executing stream processing middleware is proportional to the data volumes. Data streams generated in CSEs comprise small packets resulting in underutilization of the network links connecting the processing nodes. Combined with large volumes of messages, the increase in the execution overheads at stream processing computations lead to underutilization of CPU and memory.

- *Non-uniform resource utilization:* Unexpected changes in stream workloads as well as in system conditions cause non-uniform resource utilizations across a stream processing cluster — these often materialize as performance hotspots. The ambient system conditions can vary due to activities of the other applications sharing the resources (e.g., CPU, memory, network bandwidth, etc.), contending for global resources (e.g., network switches), power limits (e.g., CPUs mitigating thermal effects by throttling down), energy management (e.g., power saving modes), and periodic maintenance activities (e.g., log compaction, reindexing of distributed file systems). Variability in stream processing workloads also occur due to the deployment and termination of stream processing jobs and also because of operators switching between active and dormant phases either due to data availability or the condition satisfiability. Data induced load imbalances also create fluctuations in the computation workloads [21]. For instance, a particular stream partition may be accounting for a majority of the stream over time even though the partitions were initially assumed to be balanced. For example, a monitoring device attached to a patient may be configured to take measurements more frequently when physiological conditions breach thresholds or flash-crowds responding to certain events via social networks.

- *Reprocessing past data:* Changes in business requirements and discovery of bugs often entail reprocessing *past* data using the updated versions of the applications [13, 22]. Storing data streams, which are unbounded datasets that grow over time, for extended periods, increases operational expenses in both private and public clouds.

1.2 Research Questions

Based on the challenges discussed previously, we formulate the following research questions:

RQ-1: How can we improve the performance of multi-stage stream processing through effective use of available memory, CPU, and network bandwidth?

RQ-2: How can we preserve the performance characteristics of stream processing applications in the presence of variability in the workload and the system conditions?

RQ-3: How can we design a holistic methodology to address both data ingestion and data processing challenges pertaining to voluminous multi-feature data streams originating in CSEs?

1.3 Approach Summary

Our methodology is organized as three interconnected components to address the ingestion and processing challenges: (1) high-throughput distributed stream processing, (2) adaptive stream processing, and (3) space-efficient representations of streams to reduce the network and energy footprints. We developed a new stream processing engine called *Neptune* [23], built on top of the Granules data flow engine [24], to prototype and evaluate our methodology.

Neptune enables high-throughput stream processing while maintaining near real-time latencies through the efficient use of CPU, memory, and network bandwidth. It incorporates application level buffering, batched scheduling, object reuse, dynamic compression, and backpressure to improve resource utilization and to reduce the execution overhead. These design principles collectively address the unique workload challenges observed in data streams originating in CSEs as validated by our benchmarks. Neptune provides an expressive API to compose stream processing appli-

cations, and transparently handles the deployment, distributed coordination and execution of the applications.

We have designed an online scheduling algorithm [25] to enable adaptive stream processing in the presence of dynamic workload and system conditions. This algorithm uses a metric called *interference score* to quantify the interference experienced by a stream processing computation from the other collocated stream processing computations (internal interference) as well as from collocated external processes (external interference). We use a novel data structure called *prediction rings* to estimate interferences in the future to enable proactive scheduling decisions. Computations experiencing higher interferences are migrated to nodes with low interference within the cluster. Continuous adjustments to the placement of computations alleviate performance hotspots to reduce the non-uniform resource utilization within the cluster. Our scheme is able to preserve the performance of stream processing applications measured in terms of throughput, latency, and variability in latency without incurring a significant overhead.

To address data ingestion challenges, we explored the applicability of data sketching algorithms for space-efficient representation of multi-feature streams. As part of this work, we developed a new sketching algorithm called *Pebbles* [26] especially designed for multi-feature, high velocity data streams originating in CSEs. Pebbles addresses limitations of existing frequency-based sketching algorithms when used in the context of stream processing use cases. More specifically, Pebbles is able to preserve ordering between observations and provides high-level operations to access and manipulate data while guaranteeing bounds on accuracy. Pebbles leverages the gradually evolving nature of data streams through controlled reduction of resolutions of individual feature values to reduce its space footprint. Pebbles sketch instances can be inflated to generate constituent observations through an operation called materialization. We have developed a lightweight computation module to be deployed at the edge devices in proximity to data sources, such as cloudlets [27] and distributed telco clouds [28], to preprocess data streams and convert them into a stream of Pebbles sketches. These sketched streams are compact in data volumes and reduces the energy consumption at the edges of the network.

As a natural extension to the sketched streams, we developed a *sketch-aware stream processing API* to provide native processing support for Pebbles sketch instances. This API relies on sketches as the unit of data transfer and scheduling in addition to providing a set of operations tailored specific for sketches. Using sketches as the unit of data transfer and scheduling improves the resource utilization while reducing the execution overheads. Our sketch-aware stream processing API supports in-sketch operations and materialization operations. The in-sketch operations allow users to directly manipulate the sketched streams without prior materialization to provide high-throughputs and to reduce memory management overheads. Sketches can be materialized to produce a traditional data stream enabling using our API alongside of the existing stream processing applications. We developed this API as a drop-in library to facilitate seamless integration into exiting stream processing infrastructures and applications.

1.4 Contributions

This dissertation addresses ingestion and processing challenges pertaining to data streams originating in continuous sensing environments. In particular, our contributions include:

1. A high-throughput stream processing engine, *Neptune*, specialized for data streams originating in CSEs. The design principles we propose and validate are transferable to new and existing near real-time data processing systems. ***(RQ-1)***

2. An online scheduling scheme based on interference to proactively alleviate performance hotspots in a stream processing cluster in the presence of dynamic workloads and system conditions. ***(RQ-2)***

3. A new sketching algorithm, named *Pebbles*, for space-efficient representations of multi-feature data streams for approximate stream processing. Pebbles trades off the resolution of the individual feature values with guaranteed accuracy bounds while maintaining the inter-feature relationships in an observation. Sketched streams significantly reduce the energy and data transfer costs at the edges of the network. ***(RQ-3)***

4. A sketch-aware stream processing API based on Pebbles to achieve highly performant stream processing. We validate the applicability of sketch-aware stream processing using two different near real-time data processing systems. *(**RQ-1**, **RQ-2**, **RQ-3**)*

1.5 Dissertation Organization

The remainder of this dissertation is organized as follows. Background and related work are presented in chapters 2 and 3, respectively. We present our high-throughput stream processing system, Neptune, in chapter 4. Chapter 5 focuses on adaptive stream processing. Our sketch-based holistic approach to addressing stream ingestion and processing related challenges is discussed in Chapter 6. Chapter 7 outlines the conclusions. We conclude the dissertation with future work in Chapter 8.

Chapter 2

Background

In this section, we introduce the key concepts related to near-real time processing of data streams from CSEs. We first discuss how streams are transferred from sources to the data centers where they are processed followed by a discussion on how stream processing applications are modeled, deployed, and orchestrated by stream processing engines. We conclude this chapter with a discussion on the need for reprocessing past data, and existing alternatives.

2.1 Stream Ingestion

Observations generated by sensors are aggregated into multi-feature observations using aggregator nodes or edge processing modules deployed in proximity to the sensors as depicted in Figure 1.1. Sensors and aggregator modules may share the same physical device (e.g., smart phones, smart wearables) or use limited-range protocols such as PSFQ [10] and ESRT [11] otherwise. Amazon's IoT SDK [29], Apache Edgent [30], and Cisco Kinetic edge and fog processing module (Kinetic EFM) [31] are examples of the commercial and open source edge processing modules widely used today. The resulting multi-feature data streams are then transferred to centralized locations for processing. The streams are transferred over wide area networks using protocols such as TCP or machine-to-machine(M2M) protocols such as MQTT [15] and CoAP [32].

In modern ingestion pipelines, transferred data are stored in intermediary message queues, also known as IoT gateways [33, 8], to provide space, time, and synchronization decoupling between the edge modules (data producers) and the data processing middleware (data consumers). Edge modules can join and leave the data ingestion pipeline without any changes in the user applications running at the center. Similarly, the stream processing cluster can be subjected to horizontal scaling, node failures, etc. without requiring any changes to data sink configurations used by the edge modules. This design also facilitates aggregating sub-streams from multiple, spatially distributed data sources into a single data stream. Using a message broker supporting multiple consumers

like Apache Kafka [18] enables other data consuming applications within the organization (e.g.: batch jobs) in addition to the stream processing jobs share the same data stream without having to maintain multiple copies.

2.2 Near Real-time Stream Processing

Data streams generated at CSEs are unbounded, necessitating continuous processing. Batch processing systems such as MapReduce [34] deal with the unboundedness of datasets through temporal partitioning — batch processing jobs are launched periodically or when sufficient amount of data is accumulated. For instance, the backend log of a web service is analyzed at the end of the day to calculate metrics such as failed request statistics and service time latencies. Neglecting the execution time of the batch processing job, the minimum latency for results to be available is equal to the time scope of the batch. With respect to the previous example, the log record appears at the beginning of the day will not be processed until the end of the day. This type of delayed response may not be ideal for certain use cases. For instance, an unexpected increase in the service time latencies of the web service is an indicator of an underprovisioned system caused by machine failures or a surge in the service requests. Such situations should be immediately detected and remedied for a smoother operation of the web service. Near real-time processing systems are designed for implementing such latency sensitive data processing use cases where events are processed shortly after they happen.

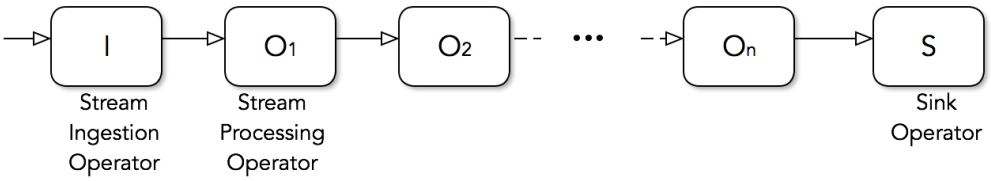
Near Real-time Stream Processing systems closely follow the *dataflow engine* design principles. In dataflow engines, jobs are comprised of operators interconnected through network connections. Operators are a loosely defined computation unit compared to *map* and *reduce* operators with strictly-defined contracts in a traditional MapReduce setting [13]. Downstream operators in a workflow can often start processing as soon as data is available without having to wait for upstream operators to complete their execution.

Stream processing systems [35, 36, 23, 37] and *micro-batching systems* [38, 39] are two main types of near real-time stream processing systems. The key difference between this categorization

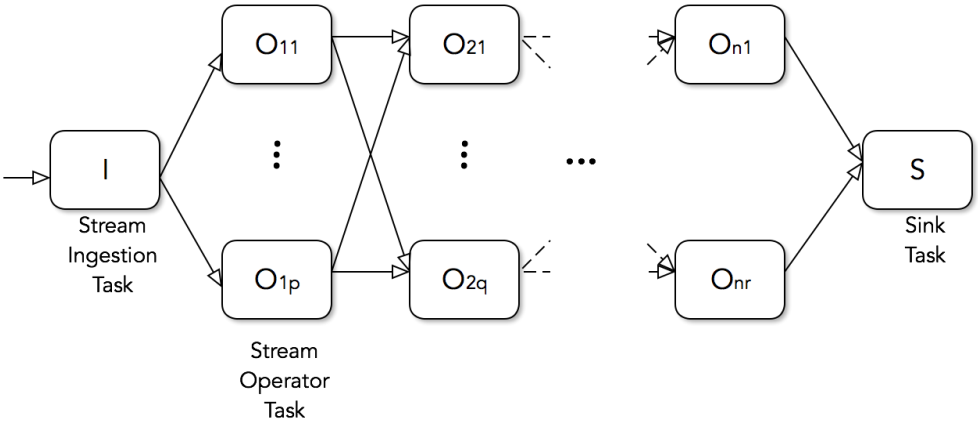
is in the smallest data unit considered for processing. Stream processing systems process individual messages in the input streams, one at a time. Micro-batching systems partition the incoming data streams into smaller batches along the temporal axis, called *micro-batches*, and consider them as the unit of processing. In this dissertation, we focus on stream processing systems.

2.2.1 Structure of Stream Processing Jobs

A stream processing job is usually modeled as a *directed acyclic graph (DAG)* of stream operators. A stream operator continuously transforms the data items in a data stream [40]. A stream operator can either be a *stream ingestion operator* or a *stream computation*. There may be one or more stream ingestion operators that ingest data streams into the system from external sources. A stream computation implements a portion of the stream processing logic, usually following the



(a) Logical plan: a directed acyclic graph of stream operators used for defining a stream processing application.



(b) Physical plan: each stream operator is mapped to a set of tasks. Tasks are distributed across multiple nodes of a stream processing cluster.

Figure 2.1: Logical plan Vs. physical plan of a stream processing application. Users model stream processing applications as directed acyclic graphs of stream operators. Stream processing runtime deploys one or more instances of each operator, called tasks, forming the physical plan of a stream processing application.

separation of concerns principle. Stream operators are connected through *streams* that form the edges of the stream processing graph. *Sink operators* are a special type of stream computation that does not have any outgoing streams; streams flowing into sink operators are called *terminal streams* [41]. Sink operators make the results of a stream processing job available to external systems such as a visualization system, a persistent storage system, or even another stream processing job. Stream operators arranged as a directed acyclic graph is known as the *logical plan* of a stream processing application.

2.2.2 Execution of Stream Processing Jobs

During the deployment of a stream processing job, multiple instances of a stream operator may be deployed to exploit the parallelism provided by multi-core processor architectures as well as distributed computing clusters. This enables data-parallel processing of streams [40]. Streams between operators need to be partitioned to ensure that each instance receives a proportional share of the input streams. The choice of the stream partitioning function depends on the nature of the processing performed at each operator and how the use case is mapped into a stream processing graph in general. Each of these operator instances is executed as a *stream processing task*. Once a logical plan is converted to a set of tasks during deployment, it is known as the *physical plan*. Logical plan and physical plan of a stream processing application are depicted in Figure 2.1.

These tasks need to be deployed within a set of distributed machines for execution where the number of tasks are two orders of magnitude greater in general. In addition to satisfying the resource matching requirement, the placement plan is further governed by constrains such as collocation and quality of service (QoS) requirements such as guaranteed upper bounds on response times. This problem is considered an NP-Hard/NP-Complete problem [42, 43, 44, 45]. Generating the initial placement of stream processing tasks in a distributed setup is a well-studied problem [46, 47, 41, 48, 49]. A naive approach would be to distribute the tasks among the available processes in a round-robin manner. For instance, the default scheduler of Apache Storm [35] follows this approach. A heuristic based scheme would be to estimate the resource requirements of

each of the tasks and use a variation of the bin-packing problem to generate the initial placement of the tasks [46]. For instance, Apache Storm's ResourceAwareScheduler [50] follows this approach by allowing users to augment the stream processing job specification with resource requirements for stream operators. Existing work has also relied on the properties of the stream processing graph itself, such as the communication patterns between tasks, in order to derive a more efficient placement plan. One example is to collocate tasks with a higher amount of pairwise communication within a single process [47].

2.2.3 Reprocessing Past Data

Reprocessing past data is necessary to support human fault-tolerance [51] and software evolution. Human fault-tolerance is the flexibility offered by a system to rectify any errors introduced by the humans into the system in the forms of bugs in the processing logic, configuration errors, etc. Like any other software, the code for data processing jobs should evolve alongside the business use cases. Once the issues in the data processing jobs are rectified, or the code is updated to meet the new requirements, the past data should be reprocessed with the latest version of the job. But near real-time processing systems are designed assuming the input streams are *transient* — input streams are discarded once processed. To counter this impedance mismatch, different architectural patterns such as *Lambda Architecture* [51] and *Kappa Architecture* [22] are proposed. Despite differences in the implementation specific details, these architectures in common require storing streams for extended periods of time.

Chapter 3

Related Work

In this section, we compare Neptune's online scheduling algorithm and the sketched based data volume reduction technique with the alternative schemes. At the end of each subsection, we perform a gap analysis comparing different schemes.

3.1 Online Scheduling for Stream Processing Applications

Current state of the art stream processing systems support different scheduler implementations that are used for the initial placement of tasks in the cluster. Besides allowing users to implement custom schedulers, Apache Storm [35] includes a set of built-in schedulers such as the Even-Scheduler, IsolationScheduler and ResourceAwareScheduler [50]. EvenScheduler, which is the default scheduler of Apache Storm, distributes stream computation tasks across cluster nodes in a round-robin manner. With the ResourceAwareScheduler which was intially implemented on top of R-Storm [46], the initial scheduling plan is derived based on the CPU, memory and network requirements of each Storm Spout and Bolt and resource availability of nodes as manually set by the user. In R-Storm, memory is considered a hard constraint that is always fully satisfied, whereas CPU and network are considered soft constraints which may not always be fully satisfied. These requirements are matched with the resource availability of each node using a Euclidean distance function; also tasks that communicate with each other are attempted to be scheduled with minimum network distance. Apache Flink [37] attempts to collocate tasks on a execution pipeline (a sequence of tasks through which the data flows) to a single slot in a task manager (equivalent to a Neptune node). Spark Streaming [52], being a micro-batch based stream processing system follows a different task scheduling scheme from the continuous operator systems mentioned earlier. Instead of allocating a task to a node for the lifetime of the stream processing job (assuming no dynamic scheduling), in Spark Streaming tasks are short-lived and are allocated at the beginning of each interval to calculate output RDDs for that interval. During this task allocation, it attempts to

preserve the data locality, assign adjacent operators to a single task, and avoid shuffling data across the network. Xing et al. [49] proposes an initial placement algorithm for operators that is resistant to short-term load fluctuations. The expected load at each operator is modeled as a linear function of stream input rates and selectivity which is then used for operator distribution in the cluster based on two heuristics: equal load distribution and avoiding creation of bottlenecks when multiple operators are collocated. In fact, this algorithm is complementary with dynamic scheduling algorithms like ours as the authors suggest because together they can withstand short, medium and long-term load fluctuations. In such a setup, the initial placement is derived using a static algorithm, and the stream processing runtime will automatically switch to an online scheduling algorithm (such as ours) during runtime.

FUGU [53] employs a rebalancing scheme to maximize resource utilization by migrating operators between hosts upon addition or removal of queries in a complex event processing setting. FUGU migrates operators from underutilized hosts and terminates those hosts in order to improve overall resource utilization whereas in our system, computations are migrated in order to alleviate any resource imbalances in the system. In FUGU, rebalancing is triggered when queries are removed or a new host is spawned as a result of adding a new query. In our system, we continually attempt to alleviate resource imbalances caused not only due to a change in the number of concurrent stream processing jobs (equivalent to number of queries in a complex event processing system), but also due to fluctuations in both the workload and in system conditions. There has been recent work on developing network-traffic aware continuous scheduling schemes [47, 54, 55, 56] for Apache Storm [35] in order to reduce the latency by reducing the amount of network communication in a given stream processing application. Computations that communicate the most (hot edges) in a stream processing graph are identified by monitoring the communication between each computation pair, and there are continuous attempts to migrate such pairs into the same process or two processes running within the same physical node. They also ensure that the worker processes are not overloaded by taking into account the performance requirements of the computations identified through continuous profiling. T-Storm [54] goes one step further by trying to consoli-

Table 3.1: Comparison of stream processing schedulers.

Scheduler	Static Vs. Online	Extensibility	Proactive	Target Metric	Elasticity
Apache Flink	Static	No	No	Latency	No
Spark-Streaming	Online	Yes	No	Throughput	Scale in/out
R-Storm	Static	Yes	No	Throughput	No
T-Storm	Online	No	No	Latency	Scale in
Neptune	Online	Yes	Yes	Throughput/Latency	No

date worker processes in order to reduce the number of worker processes required to run a given workload. Chatzistergiou et. al [55] improves the above task allocating approach by attempting to collocate the majority of the tasks of two communicating computations (called groups). Fischer et al. [56] uses a graph partitioning algorithm, METIS [57], to partition the query processing graph to reduce the number of messages communicated through the network. These approaches are reactive and focus on improving the latency, whereas our approach is proactive, relying on time-series analysis, and improves both throughput and latency.

Table 3.1 is summarizes the characteristics of schedulers used by various stream processing engines. Following is a summary of the evaluation metrics used.

- *Static Vs. Dynamic* - If the scheduling plan produced by the stream processing engine is fixed (static) or dynamically changes over the duration of the job (online)
- *Extensibility* - Ability to extend the scheduling algorithm to incorporate new resource types
- *Proactive* - If the scheduling decisions are made ahead of time based on predicted metrics
- *Target Metric* - The target performance metric the scheduler is designed to improve
- *Elasticity* - If the scheduler autonomously expands the resource pool to preserve the performance of stream applications

3.2 Data Volume Reduction at the Edges

3.2.1 Federated stream processing

Federated stream processing [14, 58, 16, 59] has been proposed as an alternative approach to the centralized stream processing in the presence of geo-distributed data sources. A subset of the operators of the stream processing graph are deployed at the edge devices close to the stream sources in order to locally process as many data. This approach is useful for providing low latency responses, location-aware services, and to reduce data transmissions to the center. In SpanEdge [14], a set of operators, called local operators, are deployed at the data centers closer to the edges and the remainder of the operators (called global operators) deployed at the center based on a classification provided by the user. Global operators running on data centers process derived partial data streams originating at the local operators such as aggregations. Renart et al. [58] proposes an edge based stream processing model to provide location-aware services where stream producers and consumers are connected via a content based subscription model. While federated stream processing fits well for certain use cases, it presents a few challenges including: (1) limited processing capabilities at the edge devices that are inadequate to execute certain operators, (2) dependencies on non-local data sources, and (3) an inability to reprocess past data. To circumvent these challenges, raw data streams (straight from the sources) or sometimes the derived data streams must be transferred to the cloud where our proposed approach based on sketched streams can be useful. It is also possible to incorporate the sketched streams within the federated stream processing solutions to efficiently transfer data from edge operators to operators in data centers.

3.2.2 Data processing at the edge

Apache Edgent [30], AWS IoT SDK [29], and Cisco Kinetic Edge and Fog Processing Module [31] are few examples of commercial and open source alternatives available for processing data at the edges of the network. These modules provide a programming model and an execution runtime for lightweight stream computations along with support for two-way communications with their counterparts running in the cloud. The connectivity with the cloud enables offloading process-

ing, transferring derived data, and device management. These edge modules are often a component in a larger IoT echo system offered by the vendor and expected to reduce the data transfer to cloud and enable low latency processing. Apache Edgent and Amazon's IoT SDK provide a functional programming style API to connect with data sources and manipulate the data streams. These edge modules provide built-in connector modules to transfer data to external systems. Despite several functional similarities with edge operators in federated stream processing systems, these edge modules are general purpose components that can be used to implement diverse use cases such as ETL pipelines. We posit that the our proposed methodology around sketched streams complements these edge modules - sketched streams generated with Pebbles sketchign algorithm can be used to further improve the data transfer costs between these edge modules and the center when applicable.

3.2.3 Edge mining

These techniques leverage seasonal patterns, predictable trends in data, and minor variations between adjacent observations. Edge mining [60, 61, 62, 63] is a class of algorithms designed to locally estimate a contextual state out of the streams and transmit only the changes to the aforementioned state to the receiver. For instance, the time-discounted histogram encoding algorithm [62] maintains a histogram representing the proportion of time spent on a set of states (e.g. walking and sitting in case of a stream produced by a gyroscope attached to a human) by locally processing a time series data stream. CARROM [63] uses a clustering based approach to detect and transfer only the significant changes to a stream.

3.2.4 Selective forwarding

These techniques [64, 65] attempt to reduce a stream into a fewer messages while ensuring the reconstructability of the stream at the receiving end with a certain accuracy. AdaM [64] is an adaptive sampling technique that adjusts the sampling frequency based on the variability of data — data streams experiencing a low variability will be sampled at a lower rate reducing the amount of data transferred. Traub et al.[65] proposes a read-fusion scheme where read requests from multiple

applications are fused together into a single sensor read (through user-defined sampling functions) which is then shared between the applications at the center. These approaches are designed for single-feature streams whereas Pebbles can natively handle multi-feature streams. Deligiannakis et al. [66] propose data reduction technique that works with multi-feature streams. At each sensor, consecutive readings are batched for each feature. Before the transfer, each batch is represented with respect to a base signal using the correlation of the feature values with the base signal. Batches are partitioned until a matching section of the base signal is found with a high correlation. At the receiving end, the streams are reconstructed based on the regression parameters and the base signal. These approaches focus only on the reducing the data transfer overhead and whereas sketched streams provide a holistic approach that encompasses data transfer and processing.

3.2.5 Compression

Compression attempts to reduce the network footprint of the data streams by exploiting the low entropy of the data streams. Most lossless compression algorithms developed to run on energy constrained devices are dictionary based due to their lower power profiles [67, 68, 9]. LTC [69] is a lossy temporal compression algorithm which exploits the linear trends in data. These compression algorithms are designed for single-feature streams. To reduce multi-feature streams, it is required to use a binary compression algorithm like GZip, Snappy, LZ4. etc. But these algorithms do not improve processing at the center — Pebbles provides a high performant sketch-aware stream processing API.

In Table 3.2, we evaluate each class of data reduction techniques based on four metrics in the context of stream processing use cases: (1) support for multi-feature streams, (2) if the data reduction scheme improves performance at the center (during stream processing), (3) if the future application requirements are constrained, and (4) if it supports latency sensitive applications.

Table 3.2: Comparison of data volume reduction schemes.

Data Reduction Scheme	Multi-feature Streams	Improved Processing at Center	Future Application Requirements	Latency Sensitive Applications
Federated Stream Proc.	Yes	Yes	No	Yes
Edge Mining	Yes	Yes	No	No
Selective Forwarding	No	No	No	No
Compression	Yes	No	Yes	No
Sketched Streams (Pebbles)	Yes	Yes	Yes	No

Chapter 4

Neptune: High-throughput Stream Processing for CSEs

In this chapter, we describe our stream processing system, Neptune [23], that is designed for near real-time, high throughput processing of data streams originating in CSEs. Achieving high throughput stream processing in IoT and sensing settings involves challenges that impact the efficiency of network, CPU, and memory utilization.

- **Small packets:** The packet sizes in IoT settings tend to be very small ($\sim$100 bytes). Since these packets are processed in Ethernet-based clusters, the small payload sizes results in a significant portion of each Ethernet packet frame (with an MTU of 1500 bytes) being unused. This contributes to lower throughputs due to network bandwidth underutilization.

- **Context switches:** Packets are typically processed in thread pools with each thread processing a packet at a time from a shared queue. The processing performed per-packet is not CPU intensive. However, since packets arrive at a high rate, context-switching costs start to dominate the overall processing costs. This is true even when packets are being processed in thread pools where the context switching costs are significantly lower than those for processes.

- **Buffer overflows:** Stream processing is performed in stages, some of which may execute on different machines. End-to-end processing in these settings is determined by the slowest stage. When packets arrive at a stage faster than the rate at which that stage can process, queues build up at these stages leading to buffer overflows, and in some cases, subsequent process crashes.

- **Object creation:** Prior to packets being processed, the raw bytes need to be transformed from their serialized representations into objects through which data fields (of different

types) can be accessed. Object creation costs in these settings can add up because of the rates at which packets arrive; this is applicable regardless of whether the memory reclamation scheme is implicit (e.g.,: Java/C#) or explicit (e.g.,: C). In extreme cases, as memory utilization increases, page faults and thrashing may occur as well.

Challenges are also exacerbated by the interactions between these issues. For example, as object creations increase there is a processing cost involved in identifying the objects (that have gone out of scope) to garbage collect. Similarly, lack of flow control may trigger unimpeded object creations and the associated memory and processing overheads.

We posit that inefficiencies in stream processing – spanning memory, scheduling, networking, and kernel issues – preclude high throughput stream processing. Each of these aforementioned issues and challenges introduces delays that adversely impact the rate at which stream processing is performed. These challenges necessitate a holistic solution that addresses the CPU, memory, network, and kernel issues (context switches and page faults) involved in stream processing.

Neptune is implemented on top of our Granules computing framework [70, 24]. The Granules runtime builds on the NaradaBrokering publish/subscribe system [71, 72] that has been deployed in domains such as multimedia and peer-to-peer grids [73, 74, 75]. Neptune leverages the general computing abstractions provided by Granules to provide a specialized and intuitive programming model for stream processing. More specifically, Neptune provides an intuitive stream processing API alongside a processing graph description model that describes processing as a collated set of modular stages. The framework initializes individual stages, establishes communication between stages and manages the life-cycle of a stream processing job. Besides these primitive constructs, users can augment a stream processing graph with a *degree of parallelism* for each stage and *stream partitioning schemes* in order to scale the stream processing job at runtime to better utilize the available cluster resources and to achieve desired levels of performance.

We profile Neptune using a comprehensive set of experiments that include evaluating the validity of individual design principles used in Neptune, and as a complete solution for stream processing. Neptune is contrasted with Apache Storm [35], a widely used stream processing system.

Experimental Setup: An in-house cluster comprising 50 physical machines connected over a 1 Gbps LAN was used for experiments. There were 46 HP DL160 servers (Xeon E5620, 12 GB RAM) and 4 HP DL320e servers (Xeon E3-1220 V2, 8 GB RAM).

We have used version 0.9.5 of Apache Storm [35] with reliable message processing feature disabled to ensure that the throughput of Storm is not adversely affected by the additional overhead introduced by acknowledgments. In our experiments, we optimized Storm for high throughput using settings recommended by developers and research literature [76, 77].

For Neptune, we have used the default configurations where the buffer size is set to 1 MB. Thread pool sizes are determined automatically depending on the number of cores in the machine it is running on. Heap sizes of both Storm workers and Granules resources were set to 1 GB.

We have used a few different stream processing jobs for the evaluation depending on the objective of the experiment. If the experiment focused on the underlying communication framework, we used stream processing jobs that were not CPU intensive to minimize interference on the communication layer from the stream processing logic. We have used complex multi-stage stream processing jobs and the associated datasets otherwise.

4.1 Design Principles

In this section, we discuss individual design choices we have made when designing Neptune. The rationale behind many of the design principles in Neptune is to achieve efficient consumption of resources: network IO, CPU, and memory. These design decisions are individually validated by micro-benchmarks.

4.1.1 Application Level Buffering

Stream processing jobs, especially in IoT settings, require processing streams comprising stream packets with small payload sizes. These streams could be either input streams that originate CSEs as well as intermediate streams that originate within stream processing jobs. Neptune is designed for deployment in commodity clusters inter-connected using Ethernet links. Because of

the significant mismatch in sizes of the MTU of Ethernet frames and serialized versions of these small stream packets, the available network bandwidth is underutilized. Also sending individual small stream packets introduces computational overheads due to the increased number of network stack traversals [78]. In the absence of low level buffering, this could also lead to a large number of system calls at the kernel network IO layer.

Instead of sending individual stream packets, Neptune implements application level buffering at the stream dataset layer to increase throughput. The size of these buffers are defined in terms of their capacity as opposed the number of messages being buffered. The rationale behind this design decision is to flush the buffer as soon as the required threshold is reached irrespective of the number of the messages in the buffer and their sizes. We have found this to be quite useful when a stream operator is producing stream packets of different sizes. The buffer size is configurable for each stream processing graph. Additionally, buffering has helped us reduce the amount of queue contention between worker threads and IO threads.

One of the challenges in buffering is to handle data streams with low data rates. This could be due low data rates in the input streams, an underperforming stream operator, or the nature of the processing logic. For instance, if a stream operator calculates a descriptive statistic for a sliding window over incoming stream packets and emits a new stream packet only if it detects a significant change in the value that is of interest, the outgoing stream will have a low and a variable data rate. This will increase the time it takes to trigger a buffer flush causing an increased queuing delay consequently increasing the end-to-end latency. This can result in failing to satisfy

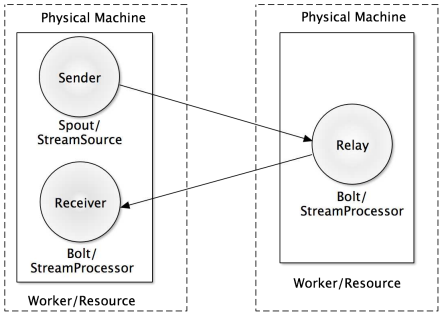


Figure 4.1: Three-stage stream processing job acting as a message relay.

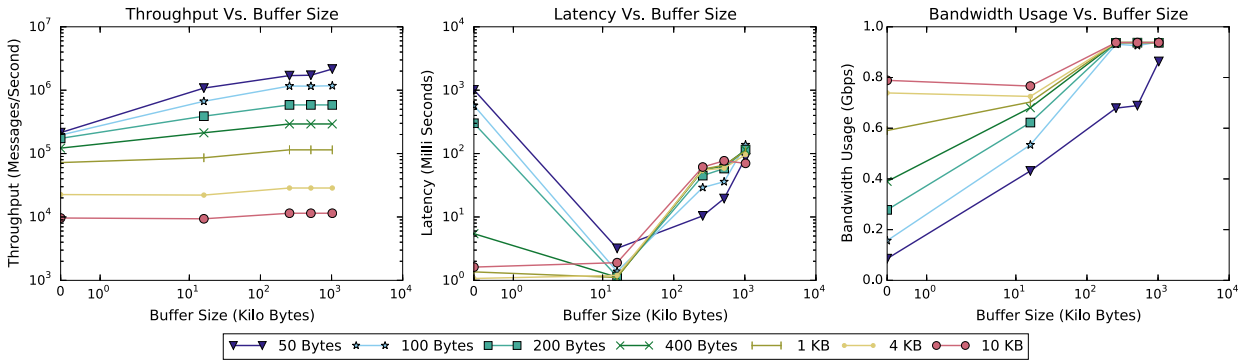


Figure 4.2: Evaluating the effect of application level buffering. Throughput, end-to-end latency and bandwidth usage Vs. application level buffer size for different message sizes.

strict real time processing constraints and violations of latency related quality-of-service (QoS) requirements [40]. To circumvent this problem, each buffer in Neptune is equipped with a timer that guarantees flushing of the buffer after a certain time period since arrival of the first message. This allows Neptune to set a soft upper bound on expected end-to-end latency even in the presence of buffering.

We observed how throughput, latency and bandwidth usage varied with the buffer size for different message sizes; this is depicted in Figure 4.2. Buffer size was varied from 1 KB to 1 MB at different step sizes. Message sizes were chosen to cover a wide spectrum from 50 Bytes to 10 KB. We have focused more on relatively small sized messages, which are in the range of 50 to 400 bytes, since majority of the message sizes found in IoT and sensing environment datasets are within that range [79]. A three-stage stream processing job, as depicted in Figure 4.1, was used for this experiment. This simulates a message relay where a stream processor in the second stage relays messages that it receives from the stream source at stage 1 to a stream processor at stage 3. The sender and receiver are deployed in the same Granules resource whereas the message relay was deployed in a different resource running on a separate physical machine. This deployment plan helps us measure the end-to-end message latency with high accuracy without having to account for clock synchronization issues such as skews and drifts.

Figure 4.2 shows the results of this experiment. As expected, the system throughput increases until it reaches a steady state with the buffer size. The bandwidth usage reaches 0.937 Gbps (out of

1 Gbps) for message sizes greater than 200 KB when the buffer size increases. Stabilization of the bandwidth consumption causes the throughput to reach and stay at a steady state for larger message sizes. The latency, on the other hand, increases slightly with the buffer size due to increased queuing delay at the application layer. For smaller message sizes, we see a very high latency when buffering is disabled due to high number of context switches as explained in section 4.1.2. With a lower, middle-range buffer sizes like 16 KB, the observed latency is less than 10 ms for all message sizes.

4.1.2 Batched Scheduling

Processing multiple stream packets in a single scheduled execution of the stream processor can improve system throughput. This is mostly by amortizing warm-up costs in instruction cache and reduced context switches between worker threads [40]. In Neptune, batch processing is tightly integrated with application level buffering. Neptune schedules processing a set of messages buffered together as a batch in a single scheduled execution. Users need to provide processing logic for a single packet while Neptune transparently manages batched execution.

We evaluated the impact of batching by measuring the number of non-voluntary context switches when batching is enabled and disabled. The Neptune process that was executing the relay processor of the message relay stream processing graph was used for this experiment. We used a modified version of Neptune where batched scheduling and application level message buffering are decoupled from each other, to ensure that the effectiveness of batched processing can be measured without any interference from buffering. Messages of size 50 Bytes were used in the experiment with a buffer size of 1 MB. The results of this experiment are tabulated in Table 4.1. The number of context switches (per 5 seconds) when batched scheduling is disabled is 22 times higher than the number of context switches when batched scheduling is enabled.

4.1.3 Object Reuse

Rather than separately and repeatedly create data structures used in serialization and deserialization for individual messages, Neptune creates them once and reuses them for the entire set

Table 4.1: Contrasting context switches.

Mode	Context Switches Per 5 Seconds	
	Mean	Std. Dev.
Batched Processing	4085.2	91.8
Individual Message Processing	89952.4	1086.5

of buffered messages. This reduces the number short-lived runtime objects at a Neptune process, which in turn reduces the strain on the garbage collector. We measured the time spent on garbage collection with and without object reuse using the same experiment setup as before. Object reuse helped reduce the percentage of time spent by the JVM on garbage collection over the time spent on actual processing from 8.63% to 0.79%.

4.1.4 Compression

Neptune incorporates support for entropy based dynamic compression. Compression can effectively reduce the volume of data transmitted between two stream computations, especially when dealing with data streams with low entropy. This could increase the effective amount of data (when uncompressed) transmitted within a given time. However, this introduces extra processing overhead at both sending and receiving ends. We have explored if this extra computational overhead can outweigh the gains due to compacted data size. Neptune employs a selective compression scheme that compresses a payload only if its entropy is less than a configurable threshold. To reduce the latency that can be introduced by compression, we used the LZ4 compression algorithm (http://lz4.org) which provides faster compression and decompression with a reasonable compression ratio.

The impact of compression on the performance of a stream processing job was evaluated using two data sets. One dataset was from the manufacturing equipment monitoring use case presented in DEBS Grand Challenge [80]. Here, sensor readings do not change frequently over time which results in a low entropy when consecutive stream packets are buffered together. To simulate a data stream with higher entropy, we created a synthetic data stream with random binary data with stream

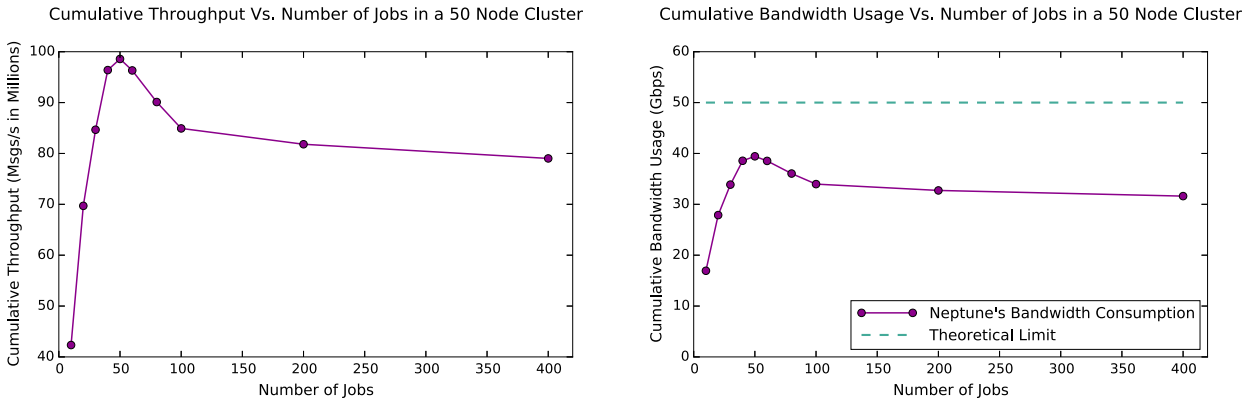


Figure 4.3: Cumulative throughput and cumulative bandwidth usage with the number of concurrent jobs.

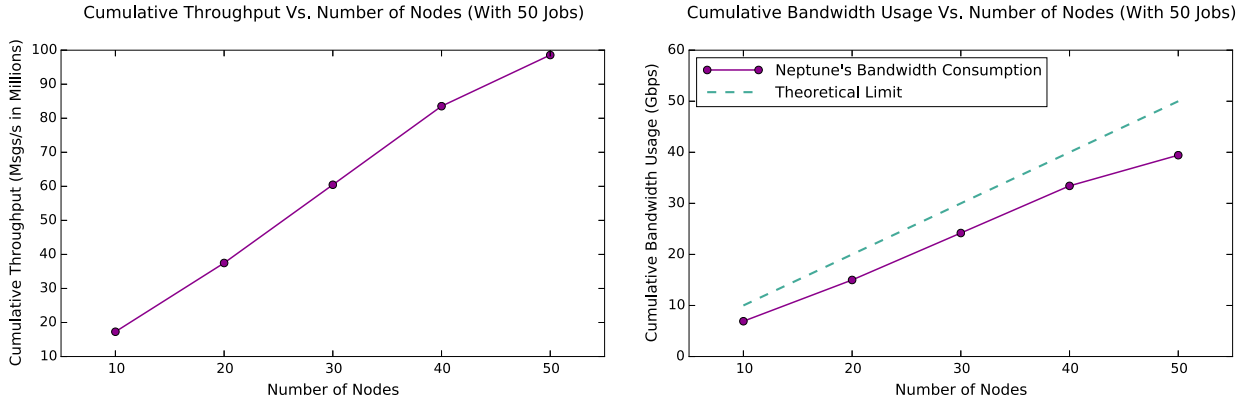


Figure 4.4: Cumulative throughput and cumulative bandwidth usage with the number of nodes in the cluster.

packets of the same size as the first dataset. We evaluated how compression impacts throughput, latency and bandwidth consumption of a stream processing job. The results were statistically validated using a Tukey's HSD multiple comparison procedure. There is a clear improvement in performance when the compression is completely disabled for random data (*p-values for individual comparisons* < 0.0001) whereas there is no strong evidence to support any negative or positive impact of the compression for the sensor readings dataset (*p-values for individual comparisons* ≥ 0.1561). This is due to the significant difference in effective compactions for each dataset. To this end, effectiveness of compression depends on the nature of the stream data, hence should be enabled and configured for each stream individually even within the same stream processing job.

4.2 System Benchmarks

A series of evaluations were performed on Neptune to profile its performance, scalability, and other features. Three metrics were used for evaluation: throughput, latency, and bandwidth consumption. We also compared Neptune with a leading open source stream processing system: Apache Storm [35].

4.2.1 Scalability of Neptune

A stream processing system should be scalable with respect to the number of concurrent stream processing jobs as well as the complexity of stream processing jobs. Complexity of a stream processing job can have many aspects: number of processing stages, complexity of the processing logic embedded in stream processors, level of parallelism and complexity of partitioning schemes.

We have used a two stage stream processing graph to measure the scalability of NEPTUNE as it helped us to create a setup where there is data flow between every pair of nodes in the cluster. Figure 4.3 depicts the cumulative throughput and cumulative bandwidth usage of a NEPTUNE cluster with 50 nodes when the number of concurrent jobs is increased. Both cumulative metrics increase until the number of jobs is equal to 50. This phase of the plot corresponds to an adequate provisioning of resources. Beyond this point, when the number of jobs increased further, the cluster reaches an overprovisioned stage and there is a drop in both cumulative throughput and cumulative bandwidth usage. We carried out another experiment by fixing the number of jobs to 50 and varied the number of machines in the cluster. Figure 4.4 shows the cumulative throughput and cumulative bandwidth usage with the cluster size. Both these metrics linearly scale with the cluster size and it is expected to reach a maximum and stabilize when the cluster size is further increased.

4.2.2 Contrasting Neptune and Apache Storm

We compared the performance of communication modules of Neptune with Apache Storm using the message relay setup depicted in Figure 4.1. The evaluation metrics: throughput, latency and bandwidth usage were recorded by varying the message size from 50 bytes to 10 KB.

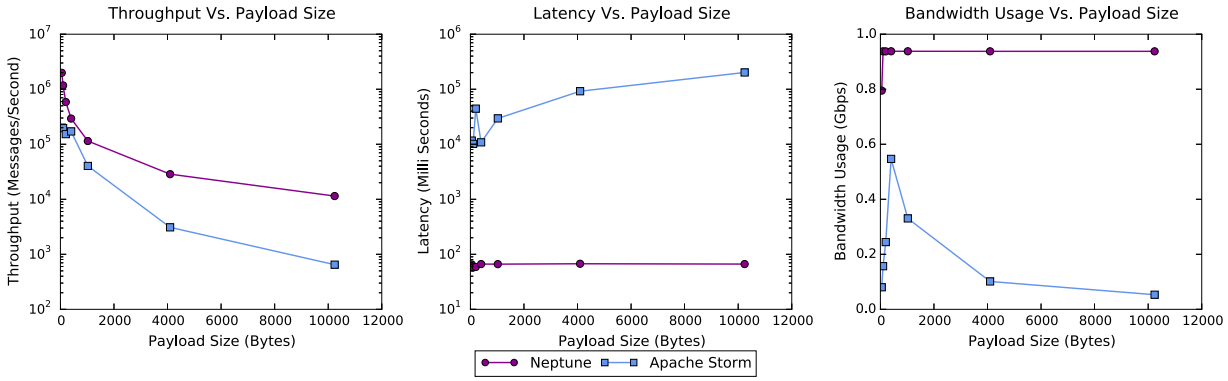


Figure 4.5: Throughput, end-to-end latency and bandwidth usage Vs. message size in Neptune and Storm.

As illustrated in Figure 4.5, the results of this experiment show that Neptune outperforms Storm in all three metrics. The latency observed with Storm was drastically increasing with the message size. This was mainly due to the absence of backpressure in Storm. The storm spout was emitting a single tuple every invocation. The relay processor (which is a Storm bolt) is relatively slower than the sender (which is a Storm spout) which creates a bottleneck in the entire Storm topology. If a small wait period is introduced between emitting tuples, then Storm was able to perform well with a very low latency. But this was not a viable option given its adverse effect on throughput and bandwidth usage. This issue has been discussed in other Storm performance benchmarks [36, 77].

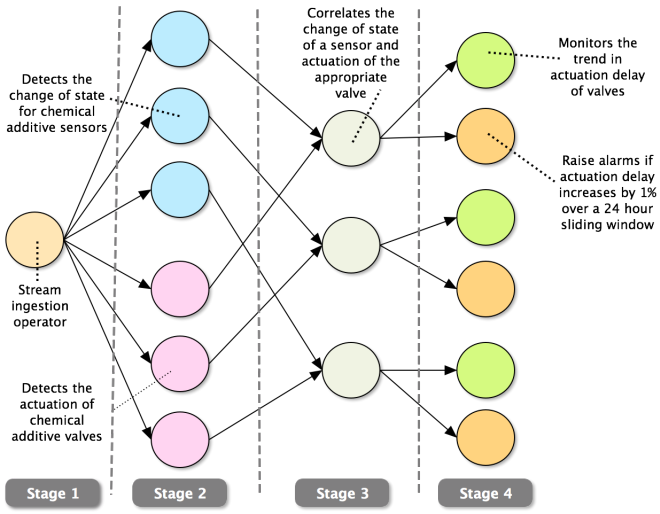


Figure 4.6: Multi-stage stream processing graph for monitoring a manufacturing equipment.

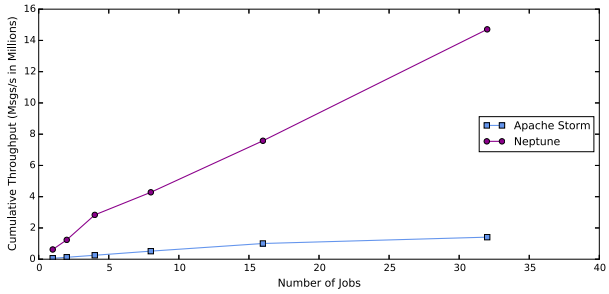


Figure 4.7: Cumulative throughput Vs. number of concurrent jobs for the manufacturing equipment monitoring use case.

Neptune was evaluated with a real-world stream processing application involving the monitoring of manufacturing equipment by processing data streams generated by sensors attached to the equipment in real time [80] as illustrated in Figure 4.6. The system ingests a continuous stream of readings captured by sensors. For this particular use case, we used 6 different data fields and the timestamp out of 66 different data fields available in a single reading. Three of these sensor readings correspond to the states of three chemical additive sensors whereas the other three readings capture the states of the corresponding valves. When the state of a sensor changes, the valves actuate resulting in a change of its state. The objective of the job is to monitor the delay between the sensor state change and actuation of the corresponding valve over a 24-hour time window.

Multiple instances of the stream processing job were executed simultaneously and the cumulative throughput at the stream source was measured across a 50 node cluster for different number of concurrent jobs. As depicted by Figure 4.7 both systems scale linearly with the number of concur-

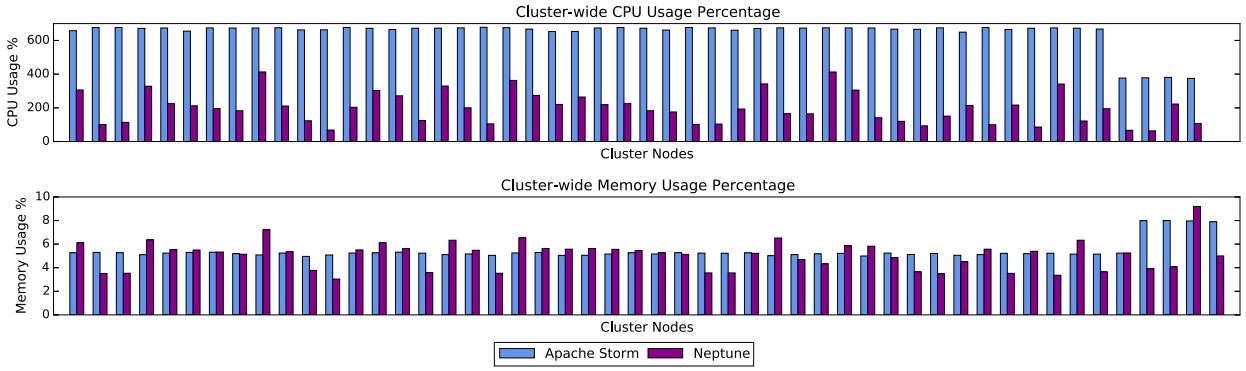


Figure 4.8: Average cluster-wide resource consumption by Storm and NEPTUNE.

rent jobs. But the throughput is higher in Neptune. With 32 jobs, Neptune's throughput is 8 times higher than Storm.

We also measured cluster wide resource consumption by both systems. Due to a scheduling constraint in Storm that dedicates a Storm worker process to a single job, the maximum number of jobs we could run at a given time was 50 using a cluster of 50 workers. Figure 4.8 depicts the cluster-wide CPU consumption and memory consumption by both systems. Please note that the CPU usage shows the cumulative value over 8 virtual cores. Memory usage is the amount of memory consumed by the system as a percentage of total available memory. Neptune's CPU consumption is consistently lower compared to the CPU consumption of Storm across all 50 nodes (*p-value for the one tailed t-test* < 0.0001) The high CPU consumption in Storm is due to its threading model which requires every message to go through four different threads from the point of entry to exit from a stream processor [36]. Neptune uses a simplified 2-tier thread model, which results in less overhead and reduced queue contention. With respect to memory consumption, there is no noticeable difference between the systems (*p-value for the two-tailed t-test* $= 0.0863$).

Chapter 5

Adaptive Stream Processing

A distributed deployment of stream processing engine experiences fluctuations in workload and system conditions. Such fluctuations can be transient (short-term) or medium and long-term. Short-term fluctuations may occur due to the event-based aperiodic nature of stream sources, the transient inconsistencies of the performance of the networking infrastructure [49], and short-term system runtime activities such as garbage collection. Contributing factors for medium and long-term fluctuations include resource contention, power limits, periodic maintenance activities, and data induced load imbalances. In this chapter, we discuss two schemes employed in Neptune to counter these two classes of dynamic workload and system conditions: *backpressure management* [23] and *online scheduling* [26].

5.1 Handling Short Term (Transient) Workload and System Condition Changes

Transient performance degradation at a stream computation could lead to situations where the processing rate is lower than the data arrival rate causing the queues to build up. Also if the queues are unbounded, it may cause long and inefficient garbage collection cycles and eventual out of memory errors at the stream processor. Some frameworks employ a fail-fast technique where the senders drop messages during such conditions, which causes loss of messages as well as wasted computation cycles if the dropped messages were already processed upstream [36].

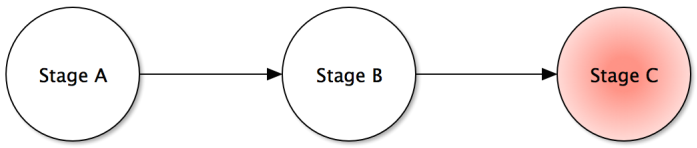


Figure 5.1: Three-stage stream processing graph used to trigger backpressure. Stream processor at stage C has a variable stream processing rate.

Neptune uses a backpressure model that leverages the TCP flow control to control the movement of data from upstream operators. For each inbound buffer of a stream processor, we maintain high and low watermarks. Once the buffer is filled up to the high watermark, the IO worker threads are not allowed to write to the buffer unless the buffer contents are consumed by the worker threads and the buffer usage reaches the low watermark level. Consequently, receive buffers associated with the corresponding TCP connections reach their maximum capacity, narrowing the TCP sliding window. This causes sending buffers at the senders to remain filled. Since Neptune uses shared bounded buffers at IO threads that are handling outbound traffic, this prevents worker threads from writing to these shared buffers. The stream processors are not scheduled again until these write operations are successful. The high and low watermarks of the inbound buffers are set sufficiently apart from each other to avoid the system oscillating between the two states rapidly. We used the setup shown in Figure 5.1 to simulate a stream processing job with a stream processor with varying performance. The thread of execution for the stream processor at stage C sleeps for some time after processing a stream packet. The sleep interval varies between 0 ms and 3 ms in a cycle that proceeds in steps of 1 ms as illustrated in Figure 5.2. The backpressure should be propagated to stream source at stage A through the stream processor at stage B. The throughput at the stream

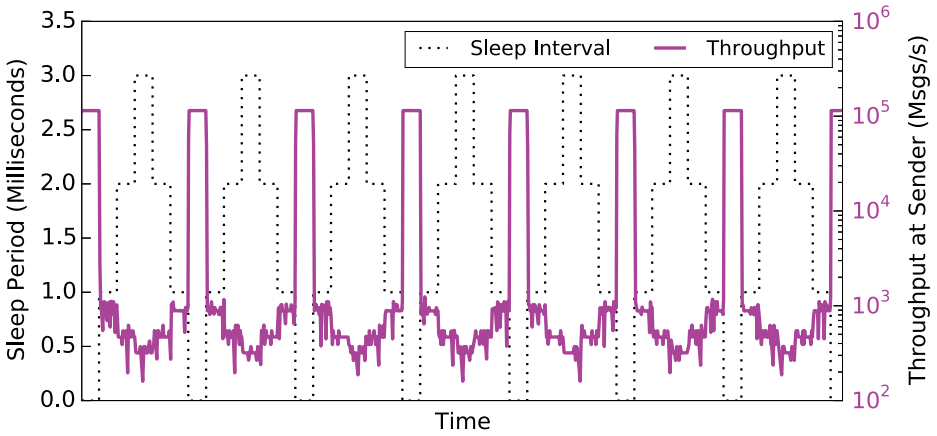


Figure 5.2: Demonstrating backpressure in Neptune. The throughput at stage A is adjusted based on the data processing rate at stage C. Data processing rate at stage C is varied using a sleep after processing each message.

source is inversely proportional to the sleep interval at stage C. As can be seen in Figure 5.2, Stage A controls the emission rate of new packets to be aligned with the processing rate at stage C.

5.2 Handling Medium/Long Term Workload and System Condition Changes

As explained in Section 2.2.2, each stream operator in the logical plan is translated into one or more tasks during the deployment to enable data parallel processing. Mapping of these tasks is a NP-Hard/NP-Complete problem — which encouraged the development of various heuristics based approaches [46, 50, 47].

Regardless of the scheme used to generate the initial placement plan, placements are bound to become inefficient over time. This is due to the inherent variability in system conditions and stream processing workloads. The system conditions can vary due to activities of the other applications sharing the resources (e.g., CPU, memory, network bandwidth, etc.), contending for global resources (e.g., network switches), power limits (e.g., CPUs mitigating thermal effects by throttling down), energy management (e.g., power saving modes), and periodic maintenance activities (e.g., periodic log compaction, reindexing of distributed file systems) [21]. Changes in the stream processing workloads occur due to the deployment and termination of stream processing jobs and operators switching between active and dormant phases due to data availability or the satisfiability of other conditions. Data induced load imbalances can also create fluctuations in the computation workloads [21]. For instance, a particular stream partition may be accounting for a majority of the stream over time even though the partitions were initially assumed to be balanced (e.g., a monitoring device attached to a patient in a critical condition is configured to take measurements more frequently or flash-crowds responding to certain events via social networks). Additional processing may be triggered at an operator based on characteristics of the data items within a stream (e.g., triggering an alarm upon detecting an anomaly). These are categorized as medium and long-term load fluctuations as opposed to short-term load fluctuations that happen due to the event-based

aperiodic nature of stream sources and the transient inconsistencies of the performance of the networking infrastructure [49].

The changes in the workload and system conditions influence a stream computation's level of contention for shared resources with other collocated computations and external processes. Contention for shared resources causes an *interference* in the execution of a stream computation. When caused by collocated computations within the stream processing engine, it is called *internal interference*. When caused by collocated external processes on the same machine, it is called *external interference*. A stream processing computation will experience both internal and external interference in varying degrees throughout its lifetime. Though the motivation behind most heuristic initial placement schemes is to ensure minimal interference across the cluster, it is hard to maintain this property over time due to changes in workload and system conditions as discussed above. Increased contention for shared resources beyond their available capacity can degrade the performance of a stream processing computation, both throughput and latency, because these computations have to wait longer for their share of the shared resource and/or receive a reduced share of resources. Variability in the workload and system conditions can affect individual machines in varying degrees causing resource utilization imbalances that result in different levels of interferences experienced by computations. Computations placed on underutilized machines may experience lower interference levels whereas computations on overutilized machines may experience higher interference levels.

To understand the impact of interference on the performance of a stream processing system, we measured the cumulative performance of a set of stream processing computations when subjected to varying degrees of internal interference. Thorax extension processing computations, explained in Section 5.2.4, were used for this experiment. The machines running the stream ingestion operators were adequately provisioned to ensure that they did not become a bottleneck during this scalability test. Stream ingestion and acknowledgment operators were dispersed over a group of 10 machines, whereas the thorax extension processing computations were all collocated on a single machine. The number of concurrent stream processing jobs was increased, which in turn increased

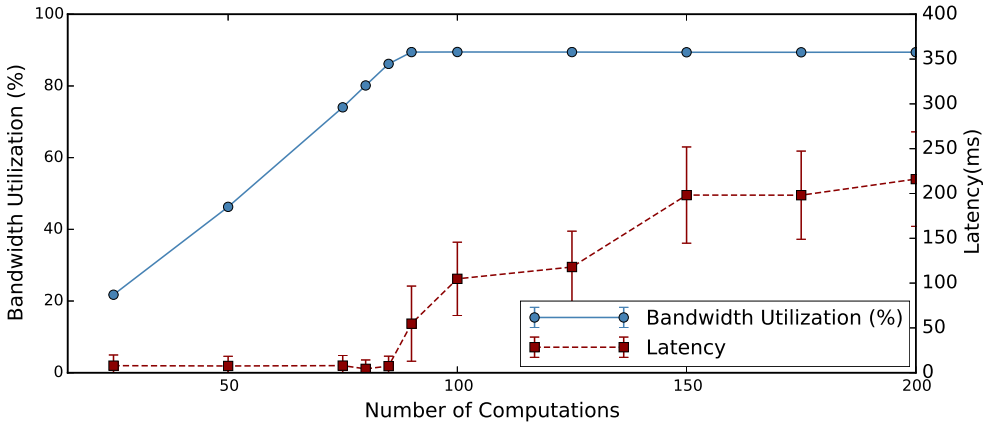


Figure 5.3: Demonstrating internal interference. Bandwidth consumption and processing latency observed at a single machine against the number of collocated computations.

the number of collocated thorax extension processing computations; that produced increased internal interference. Figure 5.3 depicts the bandwidth utilization and the processing latency observed at the machine which hosted the thorax extension processing computations. Incoming traffic that encapsulates thorax readings dominates the bandwidth consumption, which is also a measure of cumulative throughput. Even though the cumulative throughput is expected to increase with the number of stream processing computations, it does not continue to do so beyond a certain point. This is because the node has reached the maximum possible bandwidth utilization at this point; any additional computation placed on the node will interfere with collocated computations with respect to network bandwidth thereby degrading throughput at individual computations. Average end-to-end latency of the cluster also increased mainly due to increased data transfer times between data ingestion operators and data processing computations. Even though the network bandwidth was the first resource to exhaust its capacity for this particular use case, it is possible that any other resource or a combination of resources can become the limiting factor for other use cases. Horizontal scaling [40, 81] and load shedding [40] are two well-studied solutions that are often used in such scenarios. But we argue we must attempt to effectively utilize the available resources *before* provisioning more resources (horizontal scaling) or enabling graceful degradation (load shedding).

This necessitates an online scheduling scheme that is able to continuously adjust the placement of tasks based on the changing workload and system conditions. Such an online scheduling scheme

can alleviate hotspots in the cluster and reduce the imbalances in resource utilization. Accomplishing hotspot alleviation and imbalance reduction will improve the performance (throughput and latency) of the stream processing system and reduce the performance variability, especially for latency related metrics. We have implemented such an online scheduling scheme [26] to reduce the interference experienced by computations. Our online scheduling algorithm continuously and incrementally migrates computations to reduce the interference experienced by computations. It ends up moving computations away from overutilized nodes towards the under-utilized nodes that have spare capacity to host additional computations while also alleviating hotspots within the cluster. Our algorithm complements any existing initial placement algorithms and focuses on the dynamic online scheduling of stream processing jobs under varying loads and system conditions. Even though, our dynamic online scheduling scheme preserves the necessary QoS guarantees through improved resource utilization, the system may need to scale out if the workload demands more system capacity. Also for certain applications, load shedding is a viable alternative where the input streams are sampled. Even though this impacts the accuracy of the results (hence considered unsafe), it is acceptable for certain types of applications [40]. If the online scheduling algorithm repeatedly fails to reduce the interference experienced by a computation, either of those schemes can be triggered.

5.2.1 System Overview

We have validated our methodology (Section 5.2.2) for online scheduling in the context of our stream processing system, Neptune [23]. Here, we discuss the key components of the system, their responsibilities, and interactions with each other.

A Neptune cluster comprises a set of *worker nodes* running on a cluster of interconnected physical or virtual machines. Each worker node is an independent JVM process that concurrently executes a set of stream processing computations. Each worker has a *node manager* that supervises execution of computations assigned to the node. Initial placement and online scheduling of tasks in the cluster is managed by a separate process called the *orchestration manager*. These

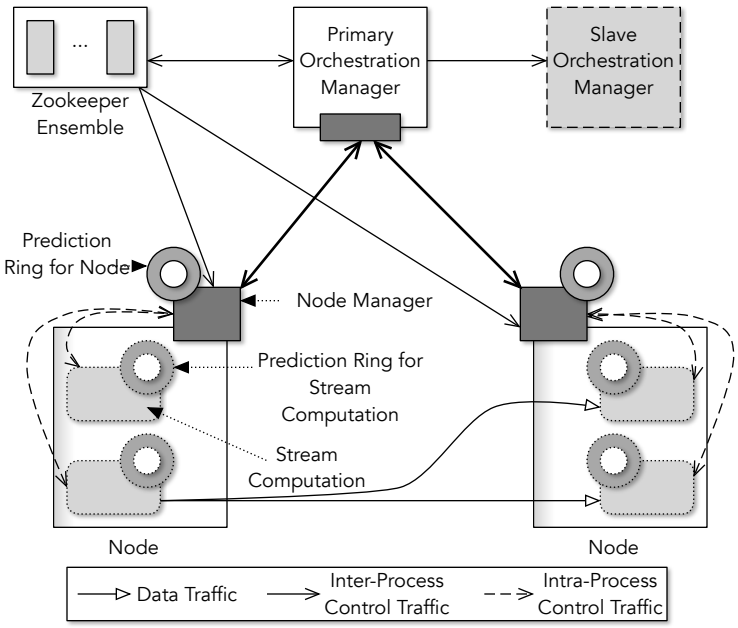


Figure 5.4: System architecture depicting key components and interactions.

entities are depicted in Figure 5.4. Node managers send periodic status updates to the orchestration manager incorporating both node and task status using a data structure called a *Prediction Ring*. At the orchestration manager, these periodic updates are used to infer the global state that informs scheduling decisions to migrate tasks and mitigate resource imbalances. The orchestration manager coordinates a migration (as explained in Section 5.2.3) by communicating with node managers at the new node, current node, and upstream nodes to redirect data streams through the exchange of control messages. The *control plane* is logically independent from the *data plane* that carries data streams that are input to stream processing computations. This helps to reduce the end-to-end latency involved in processing control messages without being affected by queuing delays and backpressure if the same channel is used for both types of traffic.

To ensure failure resiliency, a secondary instance of the orchestration manager is run in parallel in the active replication mode. The secondary replica is actively synchronized with computation placement information after deploying new jobs or at the end of a migration in addition to a job's physical deployment plan: tasks and the data flow between tasks. The remainder of the state can be reconstructed through periodic updates from node managers within a time interval less than or

equal to the periodicity of state update messages after the secondary orchestration manager has taken over the role of the primary. Neptune uses Zookeeper [82] for metadata management. We leveraged the same Zookeeper ensemble for leader election of orchestration manager nodes to appoint and discover the primary orchestration manager.

Using its global knowledge of the entire system, the centralized orchestration manager can make efficient scheduling decisions. Alternatively, it is possible to use a more decentralized approach such as, peer-to-peer or cluster-to-cluster, where nodes will arrange themselves as a virtual network [83]. By making scheduling decisions based on local knowledge encompassing a subset of the nodes, this provides a more scalable and failure resilient model at the expense of efficient resource utilization. Such an approach facilitates work-stealing [84] as opposed to the work-pushing approach employed by our methodology, which is more stable and introduces lower communication overhead.

5.2.2 Online Scheduling Algorithm

Our algorithm closely resembles the MAPE loop used in autonomous systems, which is widely adapted for implementing elastic and dynamic systems [85]. It comprises four phases: monitoring (M), analysis (A), planning (P) and execution (E). We use *prediction rings* during analysis and planning phases.

Prediction Rings - Data Structure

Prediction rings track data stream arrivals for a given stream processing computation. The data structure is then used to track and predict a computation's expected resource utilization. We use prediction rings to compute an *interference score* that quantifies the impact of placing an additional stream computation alongside other collocated computations on a machine.

The prediction rings data structure consists of multiple footprint vector "rings", each implemented as a circular buffer. Each element of a ring represents the expected resource utilization during a given discrete time window. The value stored in each time window can vary by the goals of the application, provided it is some metric indicating the amount of resources required by the

computation. For instance, the rings may be biased towards memory utilization, penalizing higher memory consumption. In this study the prediction rings are biased towards tracking CPU and network bandwidth consumption.

The number of rings and the resolution of the time windows within each ring are configurable. Having a multi-ring data structure allows us to capture arrival patterns (and expected resource utilization) at both fine and coarse-grained levels. Each ring radiating outwards represents progressively increasing time frames; the ring is bigger, with larger sectors each of which is at a coarser grained resolution. The innermost rings capture expected packet arrival rates at fine-grained resolutions. During an update to the prediction ring, the current window pointer is set to the current time, and each following clockwise window indicates the expected utilization at an increasingly distant future time. Furthermore, moving from the inner rings to the outer rings represents moving further out into the future. As such, each ring has a static offset equal to the total time capacity of the preceding inner rings. The window offset for any additional outer rings begins immediately after the last offset of the preceding inner ring. As wall time progresses, the current window pointer advances to a future clockwise window. Previous windows become invalid and are filled with a new prediction for the far future.

A conceptual view of a prediction ring with 3 rings is depicted in Figure 5.5. The innermost ring contains 16, $1250ms$ windows accounting for the next $20s$ from the current time t'. Similarly the middle ring accounts for the time period of [$t' + 20s$, $t' + 41s$) using 12 windows of $1750ms$ resolution. In our reference implementation we have used a prediction ring with 3 rings. Each ring from innermost ring to the outermost ring contained 30 windows with window lengths of $1s$, $2s$, and $3s$ respectively. Together, these three rings account for the next 3 minutes from the current time.

Prediction rings are designed to be compared against each other. Furthermore, the data structure is amenable to aggregation; i.e., we can take a collection of prediction rings and aggregate them into a single combined footprint vector. This is done by summing the window contents of each individual ring. The combined vector can then be part of a pairwise interference calculation

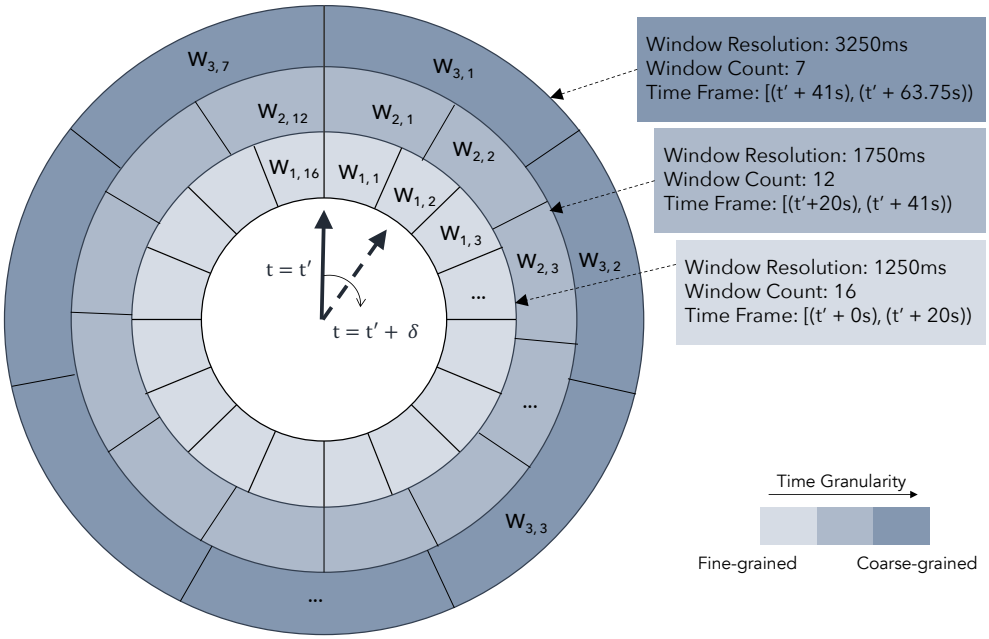


Figure 5.5: A conceptual view of a prediction ring. The ring tracks how many packets are expected in a given time window. The arrow points at the current window. Clockwise windows represent the future. The number of rings as well as the resolution is configurable.

with another computation, eliminating the need to individually check for interference with each existing computation. During aggregation operations, prediction rings are aligned with each other by shifting their current window pointers which indicates the time each prediction ring was last updated to the current timestamp.

Populating Prediction Rings

A prediction ring is updated periodically to reflect operating conditions. These updates invalidate past windows. It is also possible that current values of future windows have become obsolete due to changes in the workload and system conditions. Updates to a prediction ring are performed in two steps:

1. Initial value assignment with predicted message arrivals

2. Projecting the normalized resource consumption using the predicted input rates

For the first step, we use exponential smoothing, a time-series prediction model, to predict the message arrival rate for a computation. Exponential smoothing relies on the entire set of past

observations but assigns exponentially decaying weights for older values. This is different from other moving average models where an equal weight is assigned to every past observation [85]. We use the triple exponential smoothing method that has a seasonal component (β) in addition to a smoothing constant (α) and a trend component (γ) [86]. More specifically, we use the Holt-Winters method of exponential smoothing for predicting the message arrival rates for a computation based on prior arrival rates. The average message arrival rate calculated over a sliding window is used as the input for building the time series model because it eliminates short-term variations in arrival rates arising due to shared, overloaded network resources and other optimizations such as application-level buffering.

There are two challenges when using the time series prediction model mentioned above. The triple exponential smoothing model requires data gathered from at least two seasons in order to produce predictions with higher accuracy. This creates a cold start problem due to unavailability of observations at the beginning of the execution of a stream computation. Possible solutions to this problem would be to feed the model with observations gathered offline or to collect data during the execution of a computation and start predictions only when a sufficient number of observations are collected. Using data gathered offline can be error prone because the input rates observed by a computation (which may be different from the rate at which its upstream computation emits data) is heavily dependent on its operating environment. So we opted for the latter approach to address the cold start problem. Until enough observations are recorded for the time series prediction model, we use a simple moving average model over a sliding window of past observations to approximate the next value. The second challenge when using the Holt-Winters model is the necessity to adjust the smoothing parameters (α, γ and β) over time if there is a significant change in the environment that invalidates the current model [87]. Due to changes in the system conditions (e.g., changes in the number of collocated computations), the time-series model may become inefficient. We address this issue by recalculating the smoothing parameters using only the latest observations if the prediction error exceeds a certain threshold for a consecutive number of prediction cycles. Figure 5.6 shows the predicted value vs. actual value of message arrival rates for a single computation. The

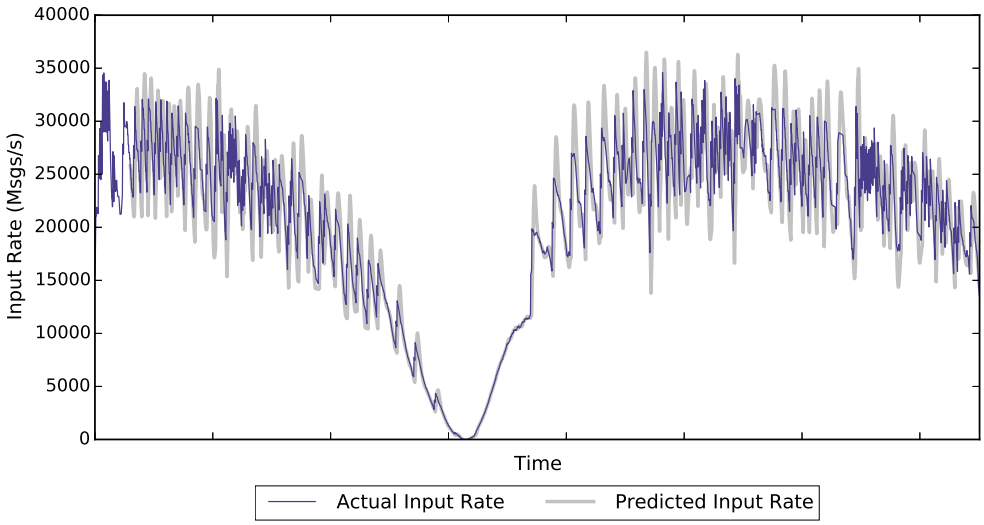


Figure 5.6: Predicted message arrival rate using Holt-Winters model vs. actual message arrival rate for a single computation.

message rate of the simulated input stream follows a recurring pattern that is effectively captured by our time-series model.

Our methodology does not preclude the use of other time-series prediction schemes such as ARIMA [88], artificial neural networks or genetic algorithms [87]. Artificial neural networks and genetic algorithms provide very accurate predictions in highly dynamic systems but require prolonged training times. We chose exponential smoothing because it satisfies several of our requirements: fast training, accuracy, compactness and quick evaluations.

The predicted input rates are then transformed to reflect the predicted resource utilization of the computation. A prediction ring will undergo a series of manipulations during this process to ensure that the calculated resource utilization values are normalized. Normalizing computations and host machines is necessary for fair comparisons to find a better location for a computation. We now discuss the rationale behind the multipliers used for normalizing prediction rings of computations.

- *Processing time per message:* This is a measure of a computations CPU requirements. This also accounts for the heterogeneity of computations and data induced additional load.

- *Message size:* This is a measure of the computation's bandwidth requirements when used together with its input rate.

Once the prediction rings of individual computations are transformed, they are summed to generate the prediction ring for the resource. Then a series of multiplications are performed on the resulting aggregated ring in order to reflect the resource utilization of the node.

- *Normalized load average of the host machine:* This is calculated by dividing the load average of the last minute by the number of CPU cores as a measure of how saturated the host is.
- *Excess bandwidth utilization:* This reflects the bandwidth consumed in excess of the preferred upper limit.
- *1 - fraction of load average caused by the node:* This is a measure of the CPU-wise external interference on the process.
- *1 - fraction of bandwidth utilization incurred by the node:* This is a measure of the network bandwidth-wise external interference on the process.

Normalizing the prediction ring of the machine is more involved than normalizing prediction rings of individual computations. The original prediction ring of the machine, calculated by summing up the prediction rings of the individual components, is preserved for aggregations with prediction rings of computations in order to calculate interference scores (as explained in the next section). The resulting prediction ring from the aggregation operation is then normalized using multipliers discussed above. These multipliers, captured using various monitoring tools, are valid only for a short duration of time because they are dependent on the stream rates and the load profile of external processes. So instead of multiplying the values in windows of every ring, the multiplication operations are applied only to the windows of the innermost ring.

Using Prediction Rings to Quantify Interference

We use the notion of *interference scores* to inform migration decisions. The interference score is a floating point value that indicates the degree of interference between computations. The larger the score, the greater the degree of interference. There are two main properties that we wanted in our interference score.

Table 5.1: Notation used in interference score calculation algorithm.

$score$	Interference score
$ring$	Current ring number
$ringCount$	Number of rings in the prediction ring
$dist$	Distance to window pointed by window pointer
$ringOffset$	Offset to first window in a ring
p	Window pointer
$ringSz$	Number of windows in a ring
ru	Resource usage score of a window in the nodes's prediction ring without computation
rw	The value in the window in a prediction ring of the node
rs	Weighted resource usage Score of a window in the nodes's prediction ring without computation
n	Interference score difference amplifier
$cpuFrac$	Fraction of the available processor cores
$bwFrac$	Fraction of the available bandwidth
$ringRes$	Resolution of the window
cu	Resource usage score of a window in the node's prediction ring with computation
cw	The value in a window in the computation's prediction ring
cs	Weighted resource usage score of a window in the node's prediction ring with computation
$totalDist$	Total length of time represented by the entire prediction ring

- *Property-1: Identifying how soon an interference is likely to occur* - We accomplish this by assigning less weight to more distant interferences. This counteracts prediction errors too far out into the future.

- *Property-2: Ability to reflect the load on a given node* - This is to ensure that computations contribute only slightly to the score if the node is lightly loaded for a given window and contribute much more significantly if the load exceeds available resource capacity.

Fig. 5.7 depicts pseudocode for our interference score algorithm. The notation used in the algorithm is defined in Table 5.1. The usage score for each window is computed with and without the existence of the computation being considered for migration. In order to calculate the prediction ring of node without a computation, we subtract the computation's prediction ring from the node's

prediction ring. We use the non-normalized version of node's prediction ring in this subtraction. The difference is then normalized as explained in Section 5.2.2. The difference between these weighted usages scores is added to the final interference score so that the score effectively reports only the impact of placing the new computation on the resource *and* is not unduly weighted by previously placed computations. This also means that the interference score operation is asymmetric. Checking how much a single computation interferes with all computations on a resource will produce a different score than checking how much the many computations interfere with the single computation, which is logical behavior for an interference scoring algorithm. The parameters $cpuFrac$ and $bwFrac$ in the interference score calculation take into account the slack we set aside to accommodate bursty traffic. By setting these upper resource consumption thresholds, we are trying to achieve both an efficient and a safe resource consumption across the cluster similar to most dynamic systems that support workload migrations and horizontal scaling.

The $dist$ indicates how many milliseconds into the future the window at position p is, growing as more windows are processed. Dividing by the total millisecond capacity of all rings, $totalDist$, allows the result to be scaled down based on the distance into the future. For each window in the prediction rings, the utilization reported by that window is scaled and contributes to the final score, thus satisfying *Property-1* that is expected in the calculated interference score.

$score \leftarrow 0$
for $ring = 0$ to $ringCount$ **do**
  $dist \leftarrow$***ringOffset***
  for $p = 0$ to $ringSz$ **do**
    $ru \leftarrow .5 \times rw[p-1] + rw[p] + .5 \times rw[p+1]$
    $rs \leftarrow ru^n/(cpuFrac * bwFrac)^n/ringRes$
    $cu \leftarrow ru + .5 \times cw[p-1] + cw[p] + .5 \times cw[p+1]$
    $cs \leftarrow cu^n/(cpuFrac * bwFrac)^n/ringRes$
    $score \leftarrow score + (cs - rs) \times (1 - dist/totalDist)$
    $dist \leftarrow dist + ringRes$
  end for
end for
return $score$

Figure 5.7: Pseudocode for computing interference scores.

Property-2 is achieved by exponentiating the score components with an integer greater than 1 – this exponent is called the interference score difference amplifier (n). This allows the score contribution to grow quickly as more computations are assigned to a time window. This, in turn, differentiates between placements resulting in collisions involving two computations versus collisions involving three computations, with fewer collisions being more desirable. Another reason arises from the tendency for computations with a high arrival rates to unnecessarily produce high interference scores, even if few collisions occur.Exponentiating allows the many windows without collisions to contribute only slightly, while allowing the occasional collision to contribute appropriately based on the severity of the collision.

5.2.3 Migrating Computations Using Interference Scores

Prediction rings and interference scores are eventually used for online scheduling where computations are migrated to nodes where they are subject to less interference and improved performance. The steps involved in the migration of a computation are depicted in Figure 5.8 in chronological order. There are three periodic tasks every node manager executes:

1. Update prediction rings of individual computations
2. Calculate the prediction ring for the node and send it to the orchestration manager
3. Calculate interference scores for individual computations

During the third task, the computation that records the highest interference against the rest of the computations — exceeding a predefined threshold in consecutive evaluations — is chosen for the next migration attempt. If there is such a computation, the node manager sends a migration request to the orchestration manager. Migration requests are piggybacked with the periodic prediction ring update messages sent by the node manager. A migration request contains the prediction ring of the computation chosen for migration and the interference score it recorded against the rest of the computations[I_c].

Upon receiving a migration request, the orchestration manager identifies the best possible node for the computation. The orchestration manager calculates interference scores individually for each node (except for its current host node) using their prediction rings and the prediction ring of the computation. If the minimum resulting interference score (I_n) is significantly lower than the interference score reported at its current location (I_c), then a migration is initiated. Otherwise a rejection response to the migration attempt is sent back to the current node; this can also be used to inform provisioning of new nodes or horizontal scaling in a cloud setting. To decide whether to initiate the migration, the percentage reduction in interference for the impacted computation (ΔI) is calculated as follows.

$$\Delta I = \frac{(I_c - I_n)}{I_c} \times 100\%$$

If the percentage reduction is greater than a configurable threshold, a migration is initiated. For example, the default threshold in our implementation was set to 5%.

The first step of the migration is to deploy an empty instance of the computation, i.e., without any state, in the new location. Once the deployment of the empty instance is complete, the upstream computation needs to be paused until the current state of the computation is successfully migrated to the newly deployed instance. Instead of completely pausing the entire upstream computation, it temporarily stops emitting messages to the stream connected to the computation being migrated while continuing to emit messages to other streams. The messages destined to the paused stream are buffered in memory. If the memory consumed by the buffered messages exceeds a certain threshold, the upstream computation completely pauses to ensure that the performance of the other collocated computations are not affected due to increased garbage collection activities. Pausing the stream from the upstream computation ensures *safety*, in other words preserving the correctness of the stream processing job during the migration [40]. Pausing is required because we have used point-to-point communications in Neptune to optimize for high throughput settings [23]. Switching to Neptune's publisher-subscriber communication mode can completely eliminate the need for pausing of the stream albeit at reduced throughput. The pub-sub mode will make the stream packet flow asynchronous and improve the mobility of stream computations as advocated

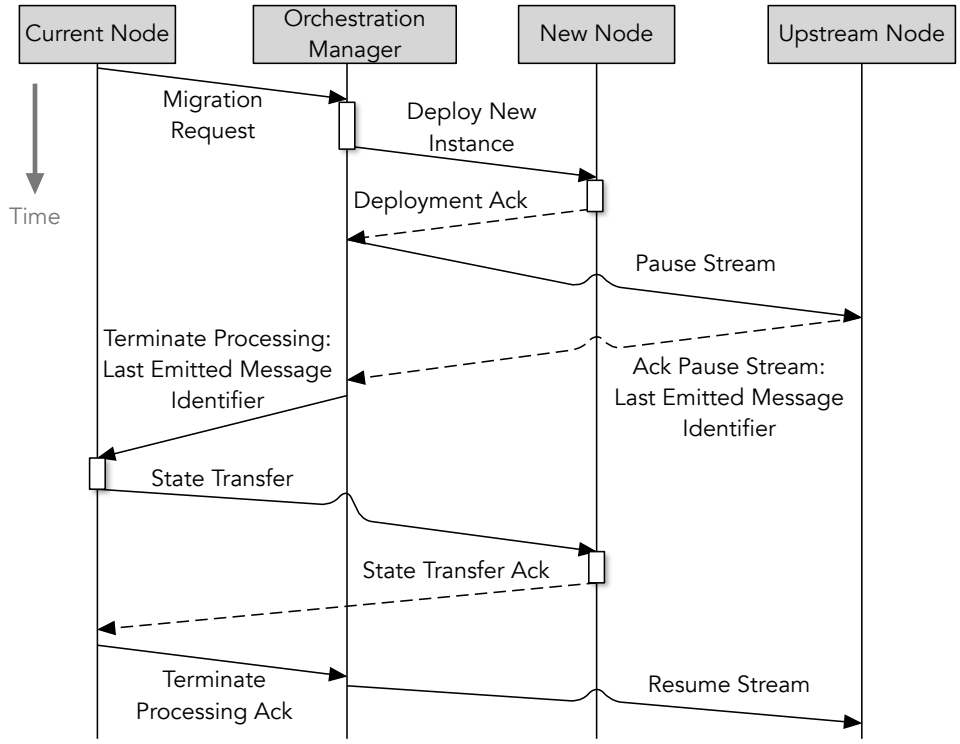


Figure 5.8: Sequence diagram depicting a computation migration.

in [89, 90]. After pausing, in its acknowledgment to the orchestration manager, the upstream computation includes the sequence number of the last message emitted to the paused stream. This information is relayed back to the stream computation by the orchestration manager. The stream computation will wait until it has completed processing this particular message before moving to the next phase ensuring that no messages are left unprocessed during the migration. Next, the computation being migrated will serialize its state and send it over to the fresh instance of the computation placed at the new location. Upon processing this message, the new instance of the computation restores its state. Once the new instance is ready to process messages, the upstream computation will first play the buffered messages and resume its regular operations. The computation at the old location is then terminated permanently.

Ensuring System Stability

Migrations incur a significant overhead mainly because they interrupt the regular operation of the upstream computations and pause the processing of a subgraph for the duration of the migra-

tion. The throughput of the stream processing job drops for a while, and the latency will show a sudden spike when processing the buffered messages. Hence triggering a migration should be done only if the expected performance-gains outweigh this temporary degradation in performance. There are some built-in measures in our implementation to reduce unnecessary migrations.

Using a *dynamic threshold function* at the orchestration manager is one such measure. As discussed before, a migration is triggered only if the expected percentage reduction of interference (ΔI) is greater than a certain threshold. This threshold value is dynamically adjusted based on the state of the system: if there is a significant variation in resource utilization within the cluster, it is set to a lower value and vice versa. This dynamic threshold function encourages migrations when there is a significant resource imbalance in the cluster, even if there is a small improvement in interference. In our implementation, we have used a simple step function that sets different threshold values based on the variance in the interference scores reported for a computation (targeted for migration) against every node.

Another stability measure that we leverage is to force a *cooling down period* [85] on nodes after they have participated in a migration, either as the source or destination. During this period, such nodes are not allowed to either trigger any migrations nor are they considered a candidate to receive computations from other nodes. The cooling down period also provides time for the monitoring system to capture changes that occurred during the previous migration allowing time series models to stabilize and recalibrate if necessary. This also reduces the number of migrations triggered due to unreliable prediction rings.

Since the computation with the highest interference score at a node is picked for the next migration attempt, the orchestration manager may not be able to find a better node. The migration request will be rejected and possibly will continue to get rejected in successive attempts. Such successive rejections will prevent the node from making any progress towards alleviating the hotspot. As a countermeasure, if a migration request for a particular computation is rejected then it will not be scheduled for migration for some time. By moving computations with less interference to better alternative locations, it may reduce the interference at the current node. If it does not reduce the

interference of the original computation as expected, then it is an indication that the system either needs to be scaled out horizontally or start load shedding.

5.2.4 Empirical Evaluation

Experimental Setup

The benchmarks reported here were performed in a cluster comprising 54 physical machines connected over a 1 Gbps LAN. Each machine is a HP-DL60 server (Xeon E5-2620 CPU and 16 GB RAM) running Fedora 23 and Oracle Java 1.8.0_65. Primary and secondary instances of the orchestration manager were running on dedicated machines. A three node Zookeeper ensemble with each Zookeeper server running on a dedicated machine was used. Stream ingestion operators were scheduled to run in an adequately provisioned setting with 33 dedicated machines ensuring that ingestion operators do not become a bottleneck during the experiments. Stream computations were scheduled to run on group of 15 dedicated physical machines that did not overlap with the machines allocated for stream ingestion operators. Each physical machine was running a single Neptune process. A central statistics server was used to gather various cluster-wide benchmark

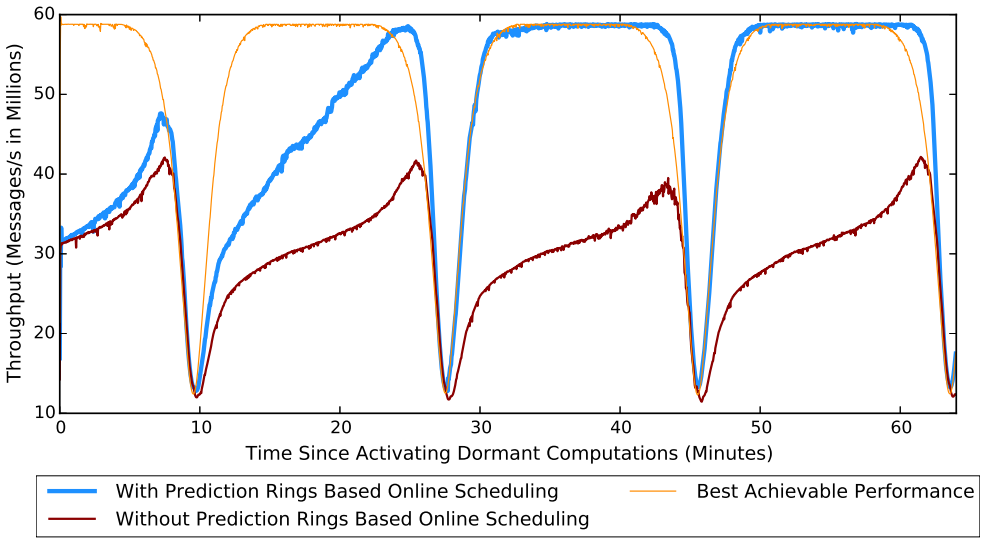


Figure 5.9: Cumulative throughput of the cluster over time with **variable input rates** under **internal interference**. When migrations are enabled, the throughput gradually increases as hotspots are alleviated. Its peak performance is approximately equal to the near-ideal performance.

metrics: stream processing performance, resource utilization at individual nodes, and migration activity data. Having a centralized statistics server helped us accurately analyze how different metrics varied over time across the cluster.

Stream Processing Use Cases

Our empirical evaluations are performed with two use cases from the health stream processing domain: thorax extension and ECG processing using the Pan-Tompkins algorithm. In order to simulate high throughput streams, the records from these datasets were ingested at much faster rates than the actual message rates. This did not affect the correctness of the stream computations, because the timestamp value encoded in the record itself was used for processing instead of the ingestion timestamp or the wall clock time.

Thorax extension processing was more CPU intensive than ECG processing. Due to its high throughput, ECG processing computations were creating more strain on the network bandwidth. Due to their smaller in-memory state, none of them caused significant memory pressure. Stream processing graphs of both these use cases have the same structure. A single stream processing job contains a stream ingestion operator, a stream computation operator which implements the thorax extension or ECG processing logic and a sink operator that is used primarily for measuring end-to-end latencies. The stream ingestion operator and the sink operator were collocated on the same Neptune node allowing us to measure the end-to-end latency without being affected by clock synchronization and skew issues. The workloads for these benchmarks comprised a mixture of these stream processing jobs with a ratio of 1:1. Though real-world health stream processing use cases will only be executed on dedicated clusters due to their critical nature, we have used them to evaluate the efficacy of our online scheduling scheme in both shared and dedicated cluster setups.

Thorax Extension Processing: The thorax monitoring computation we use here is designed to act as a backend for a visual monitoring application. It retains the last 10 seconds of chest expansion and contraction data in memory of 6 patients, while also maintaining a running average, minimum, and maximum values seen. The thorax extension dataset we use was gathered by Dr. J. Rittweger

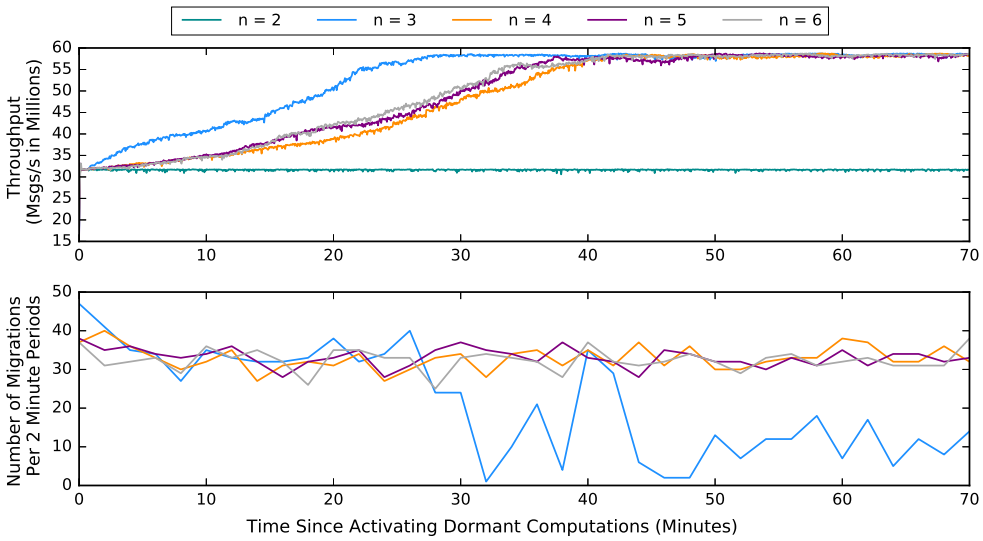


Figure 5.10: Determining the interference score amplifier (n).

at the Institute for Physiology, Free University of Berlin [91].

ECG Processing with Pan-Tompkins Algorithm Our ECG computations process ECG waveform data from 10 ICU Patients that is available as part of the MIMIC dataset from physionet.org [5, 92]. An ECG monitors the heart's electrical activity, which drives the expansion and contraction of heart muscles based on the generated polarity.

We preprocessed ECG waveforms using the well-known Pan-Tompkins algorithm to detect the QRS complex [93]: this includes bandpass filtering, differentiation, and integration. Since ECG waveforms need to have all its frequency components within the 5-15 Hz range, the waveform is bandpassed to filter out undesired frequency components and then differentiated to attenuate the higher variations and squared to remove negative components. We then used integration to identify the peaks of the squared wave. Integrated signal peak points are used to find the QRS locations, and hence the distance between two QRS complexes, and the amplitude of the QRS is the same as that of the bandpassed wave.

Determining Parameter Values for the Interference Score Calculation Algorithm

For our benchmarks, the values for $bwFrac$ and $cpuFrac$ were assigned based on widely used upper thresholds reported in research literature allowing a $\sim 30\%$ slack to handle possible load spikes. For network bandwidth utilization, 70% was used as the upper threshold [94]. Similarly, 66.6% was used for $cpuFrac$ considering the most commonly used range of 50% - 80% for CPU consumption [95].

The interference score difference amplifier (n) was determined empirically. We simulated an uneven workload across the cluster by activating a set of dormant computations on a select subset of nodes. We deployed 2250 stream processing computations across 15 machines (150 computations per node). Only 20% of those computations (450) were active at the beginning. After a while, we activated the remaining 80% dormant computations on 5 of those machines (600) which created a total of 1050 active computations across the cluster. After dormant computations become active, (as intended) there was a significant imbalance in the cluster where 33% of nodes became performance hotspots. We evaluated the choice for different values of n based on two metrics: time to reach the best achievable performance and the stability of the system. As shown in Figure 5.10, when n was set to 3, we were able to achieve the best achievable throughput within the shortest amount of time while keeping the number of migrations low over time — achieving a relatively higher stability compared to other practical values. Furthermore, exponents less than 3 did not provide enough amplification to trigger any migrations whereas exponents greater than 3 were triggering large number of (less) effective migrations causing a longer time to reach the best possible performance and a lower system stability over time.

Internal Interference: Alleviating Resource Imbalances

The objective of this set of experiments is to profile how effectively our prediction ring based online scheduling algorithm alleviates resource imbalances caused by internal interference. Such situations can arise when the workloads are unevenly distributed among nodes.

We used the same setup discussed in Section 5.2.4 to simulate an uneven workload. To contrast with the achievable near-ideal performance, we evenly distributed the stream processing work-

load within the cluster. Specifically, it placed 1050 computations evenly across the cluster with each node executing 70 active stream computations from the very beginning. This represented the best possible placement where the workload is evenly distributed and there are no imbalances in resource utilization. Over time, our online scheduling algorithm should be able to achieve a placement closer to this near-ideal distribution through migrations even when the placements are initially highly imbalanced.

Our evaluation metrics include: cumulative throughput of the cluster, 99^{th} percentile and standard deviation of the end-to-end latency, and resource utilization of the cluster.

The experiment was conducted for both fixed rate input streams as well as variable rate input streams. In order to generate a stream with a variable message rate, the stream ingestion operator employs a load profile that defines the message emission rate over an outgoing stream at any given time. The load profile is a function that takes the time elapsed since it was activated as the input variable to calculate the stream emission rate. Fixed rate input streams were used primarily for assessing latency related metrics. This is because the end-to-end latency of a stream packet is also governed by the message rate on that stream. If the message rate is high, the latencies tend to be higher due to prolonged queuing delays awaiting access to resources such as CPU time and network buffers.

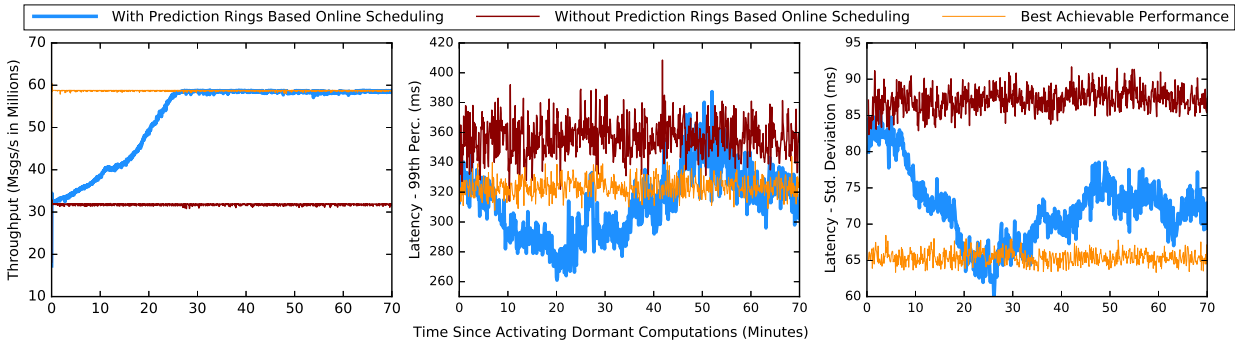


Figure 5.11: Variation of cumulative throughput, 99^{th} percentile of latency and standard deviation of latency of the cluster over time with **fixed rate** streams under **internal interference**. The performance of the online scheduling algorithm is compared with the near-ideal performance (computations are evenly distributed across the cluster) and the initial performance (the setup without the scheduling algorithm).

Figure 5.9 depicts the cumulative throughput of the cluster observed over time with variable rate input streams. This experiment relies on the time-series models to predict the stream rates and migrate computations accordingly. Due to the bandwidth-bound nature of the stream processing use cases, computations at crowded nodes were underperforming mainly due to heavy bandwidth interference from collocated computations. As our online scheduling moves computations over to nodes with less interference, individually they start to perform better, resulting in increased cumulative throughput. The cumulative throughput improved by 48.89% compared to the original peak throughput values, and near-ideal performance is achieved after alleviating the hotspots. Next, we repeated the same benchmark with fixed rate streams in order to understand the impact of our scheduling scheme on end-to-end latency. Figure 5.11 depicts the performance of the system over time with and without our prediction ring based online scheduling alongside a comparison with the near-ideal performance achievable. Using the 99^{th} percentile, we have evaluated how the long-tail latency improved as the computations are moved away from overutilized nodes. The predictability of measured latencies is evaluated using standard deviation, which is a measure of the variability in the recorded latency values. Latency related metrics for even scheduling (near-ideal performance) and for the setup without the online scheduling algorithm demonstrate a relatively steady

Table 5.2: Summary of performance improvements provided by our online scheduling under **internal interference**. For throughput, positive is better whereas for latency metrics, negative is desirable.

Use case: *Stream Computations with Variable Rate Streams*		
Metric	Deviation from Initial Perf.	Deviation from Near-Ideal Perf.
Throughput	+48.891%	-0.001%
Use case: *Stream Computations with Fixed Rate Streams*		
Metric	Deviation from Initial Perf.	Deviation from Near-Ideal Perf.
Throughput	+83.932%	-0.413%
Latency - 99^{th} perc.	-5.241%	+4.768%
Latency - Std. Dev.	-15.845%	+12.672%

series of observations. The latencies exhibit a high initial variation when the online scheduling algorithm is running, especially when there is a large number of migrations taking place due to message buffering at upstream computations. However, once the system reaches a steady state (with a smaller number of migrations), we observe improved latency with respect to both the 99^{th} percentile and standard deviation. We observed a 5.24% improvement in the 99^{th} percentile and 15.85% improvement in the standard deviation of the end-to-end latency mainly due to reduced communication delays experienced by packets when processing is moved to nodes with less saturated links. Similar to the previous benchmark with variable rate streams, **there is a significant improvement in throughput of 83.93% compared to the setting without online scheduling.** Table 5.2 summarizes the performance improvements for different metrics with respect to near-ideal and initial performance for the aforementioned benchmarks.

Using the data gathered by monitoring individual nodes during the benchmark with variable rate streams, we evaluated how the resource utilization of individual nodes changed over time. The objective of this evaluation is to observe how well our scheduling algorithm can alleviate resource imbalances present in the cluster. Resource utilization was measured based on the normalized CPU load average of the process (provided by `OperatingSystemMXBean` class of Java 8) and bandwidth utilization as a percentage of the available bandwidth. Memory utilization was not considered as part of the resource utilization because our stream processing use cases did not introduce a significant memory pressure; however, our methodology facilitates incorporation of memory utilization when appropriate. Figure 5.12 shows how the resource utilization at individual nodes was changing over time after the dormant computations were activated. Table 5.3 lists the mean and standard deviation for bandwidth utilization percentage and CPU load average at different points in time. Initially, the nodes where the dormant computations were activated show significantly higher resource utilization compared to the rest of the nodes, which results in a resource imbalance within the cluster. As the online scheduling algorithm moved computations away from these hotspots, gradually the resource utilization across the cluster became more consistent and even. This can be clearly observed by how the standard deviation reduced over time. Also the average resource

Table 5.3: Resource utilization of individual machines over time since the activation of dormant computations.

Time Elapsed	CPU Load Average		Bandwidth Utilization (%)	
	Mean	Std. Dev.	Mean	Std. Dev.
1 min	0.1727	0.1023	47.3592	25.5610
10 mins	0.2012	0.0805	53.3550	16.9064
25 mins	0.2919	0.0082	77.4808	0.6902
70 mins	0.2836	0.0098	77.6729	0.0642

utilization increased as our online scheduling algorithm attempted to spread the workload evenly. This also improved the system throughput significantly as evident from our previous benchmarks (Figure 5.9, Figure 5.11, Table 5.2).

External Interference: Alleviating Hotspots

The objective of this benchmark is to evaluate the effectiveness of our online scheduling algorithm when a subset of nodes are affected by external interference. Each node is allocated the same number of stream processing computations, and 33.3% of the nodes were subjected to external in-

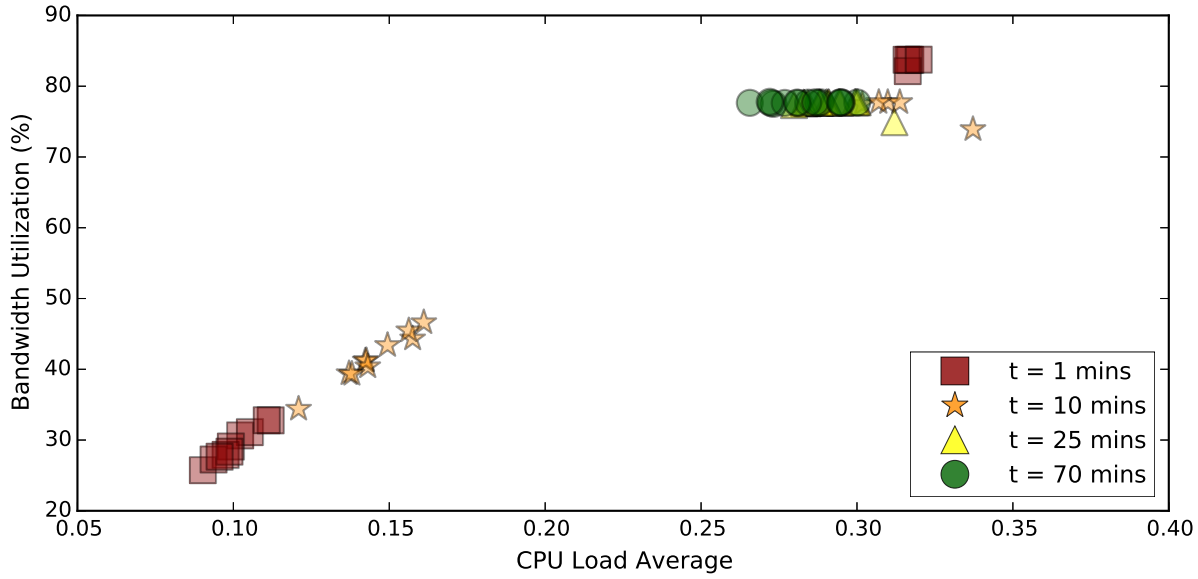


Figure 5.12: Resource utilization of machines at different points in time after the dormant computations are activated.

terference. External interference was simulated using a separate process that generated significant CPU and network bandwidth pressure. Similar to the previous benchmarks cumulative throughput, 99^{th} percentile and standard deviation of the observed latencies were used as the evaluation metrics. As the near-ideal performance, the same number of computations were executed on a cluster of equal size without any external interference. Online scheduling algorithm was disabled when measuring the near-ideal performance. Thorax and ECG processing computations with fixed rate streams were used; but a fixed variability was introduced to the message rate as it closely simulates most real world streams.

Figure 5.13 plots the variation of the three metrics over time. By migrating computations away from the nodes with external interference, **our algorithm is able to recover 77.25% of the lost throughput due to external interference**. Computations migrated away from nodes with external

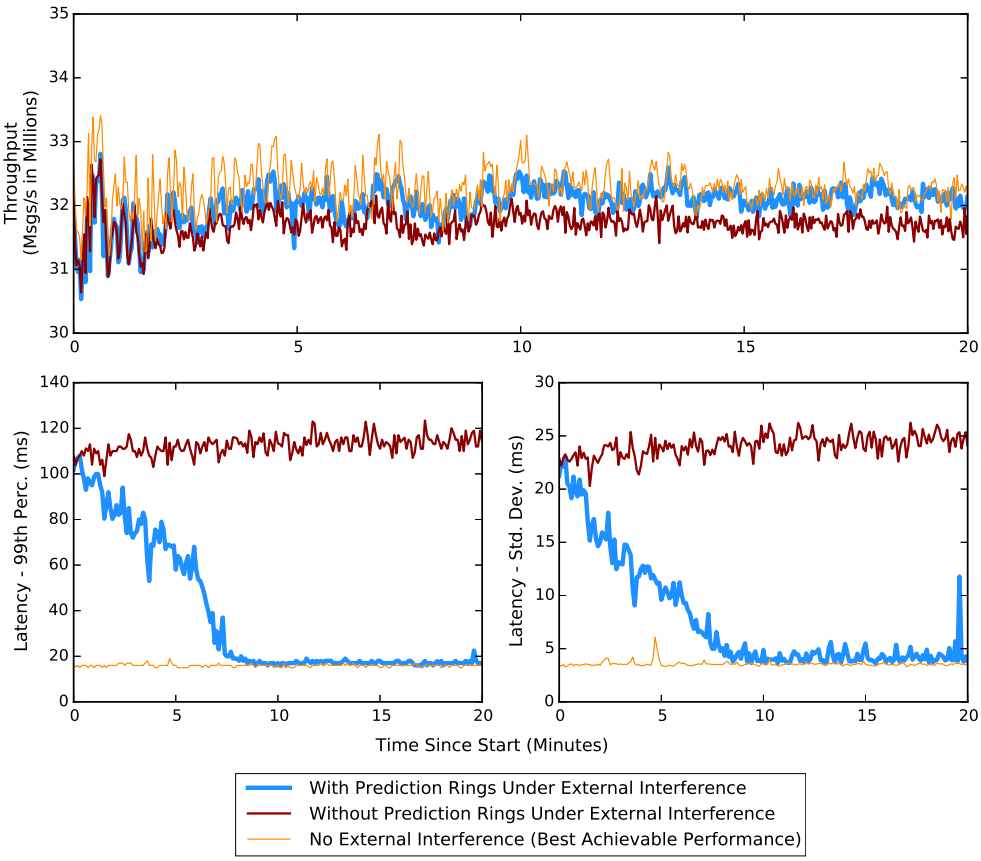


Figure 5.13: Variation of cumulative Throughput, 99^{th} percentile of latency and standard deviation of latency of the cluster over time when 33.3% of nodes in cluster is subjected to **external interference**.

Table 5.4: Summary of performance improvements provided by our online scheduling under **extenal interference**. For throughput, positive is better whereas for latency metrics, negative is desirable.

Metric	Deviation from Initial Perf.	Deviation from Near-Ideal Perf.
Throughput	+1.300%	-0.376%
Latency - 99^{th} perc.	-85.0041%	+8.2079%
Latency - Std. Dev.	-82.8989%	+19.2485%

interference benefit from less contention for network bandwidth, which is the main reason behind the throughput improvement. We could observe significant improvements in latency related metrics with our scheduling algorithm. **Both the 99^{th} percentile and the standard deviation showed an improvement of over 82%.** This is mainly due to reduced waiting times experienced by computations for their CPU and network bandwidth shares at nodes with less interference after the migration. The improved performance is still slightly less than the maximum achievable performance (setup without any external interference) because regardless of migrating computations away from the nodes affected by external interference, system resources are still shared between two groups of processes— Neptune nodes and interfering processes. Hence it is not possible to completely recover from performance degradation. A summary of performance improvement compared to the initial setting and the near-ideal setting is available in Table 5.4.

Evaluating the Stability of the System

Measures taken to maintain system stability by ensuring that only the migrations yielding significant improvements are allowed are discussed in Section 5.2.3. We evaluated the effectiveness of these measures using the variable stream rate benchmark.

Dynamically adjusting the threshold for the expected reduction of interference (ΔI) is one measure to ensure system stability. Figure 5.14 shows how the threshold is dynamically adjusted based on resource utilization imbalances within the cluster as indirectly captured by the variance in the interference scores for the impacted computation at nodes. Figure 5.15 shows how the num-

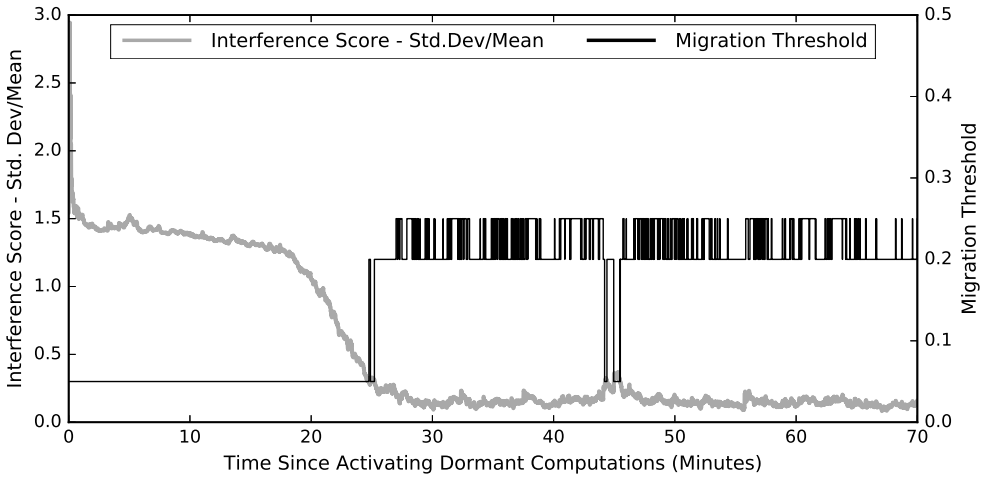


Figure 5.14: Dynamically adjusted threshold of expected reduction of interference (ΔI) for triggering migrations over time.

ber of migrations completed in successive, non-overlapping two-minute intervals is changing over time. As seen in Figure 5.14, the variance in interference scores recorded against individual nodes decreases gradually over time. This is indicative of the alleviation of resource utilization imbalances that were present at the beginning. There is also a decrease in the number of migrations over time (as shown in Figure 5.15) mainly due to the adjustment of the threshold, the deciding factor for initiating a migration, to higher values at later stages. Our benchmarks demonstrate that our algorithm encourages aggressive migrations when there is a significant imbalance in the resource utilization among nodes until the resource utilization in the cluster reaches a reasonably consistent state. Also, in the case of computations with variable rate streams, there will be continuous attempts for migrations. This is because of the different degrees of interference expected by computations due to the variable input rates, even though the amount of work performed per message is similar. This can be seen in Figure 5.15 from the relatively low number of migrations taking place after the initial aggressive scheduling period. In this benchmark, the average completion time for a migration is 54.68 ms (std. dev. = 65.70 ms). The time required to complete a migration is dominated by the state transference phase and the backlog clearance phase of the current computation tasks.

5.2.5 Profiling the Runtime Overhead for Online Scheduling

Running our prediction rings based online scheduling algorithm incurs additional processing and memory overheads. Periodic execution of prediction ring updates, interference score calculation, and maintaining the prediction ring data structures in memory are the primary contributing factors to this additional overhead. We monitored the memory consumption and CPU utilization of nodes when our scheduling algorithm was running and contrasted it with regular Neptune operations with the online scheduling algorithm disabled. An equal number of computations were placed with fixed input rates on each node in both cases, and their CPU and memory consumption was measured periodically. We have calculated the mean CPU and memory utilization per node using these metrics. In order to maintain a fixed number of computations at a node when the online scheduling algorithm is running, we disabled migration triggers at the orchestration manager. Node managers were still executing their periodic tasks of updating prediction rings, calculating interference scores, and sending periodic status updates to the orchestration manager. So this benchmark does not capture the additional resource utilization caused when a migration is triggered that includes: mainly processing of a few additional control messages and serialization/deserialization overhead when transferring, and restoring the state of the migrated computation. We posit that this

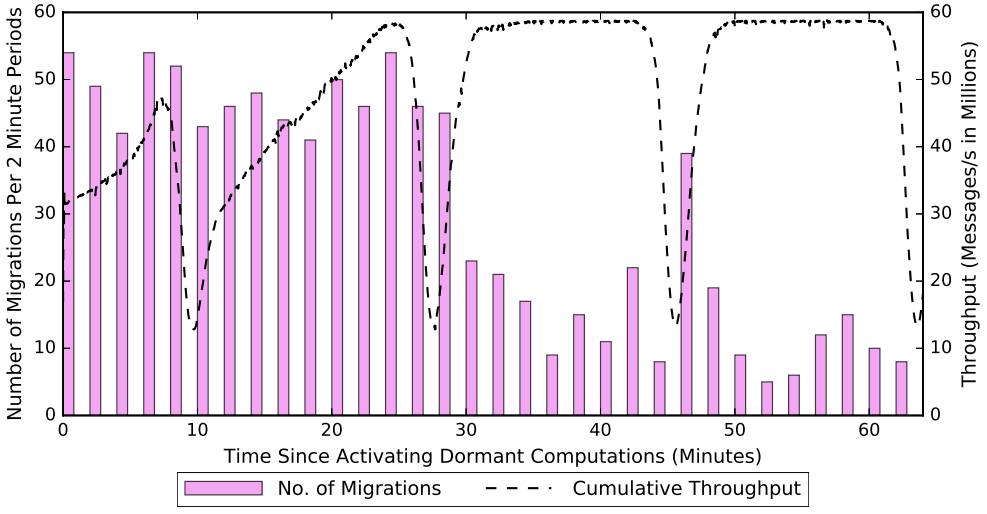


Figure 5.15: Number of migrations (over a window of 2 mins) over time.

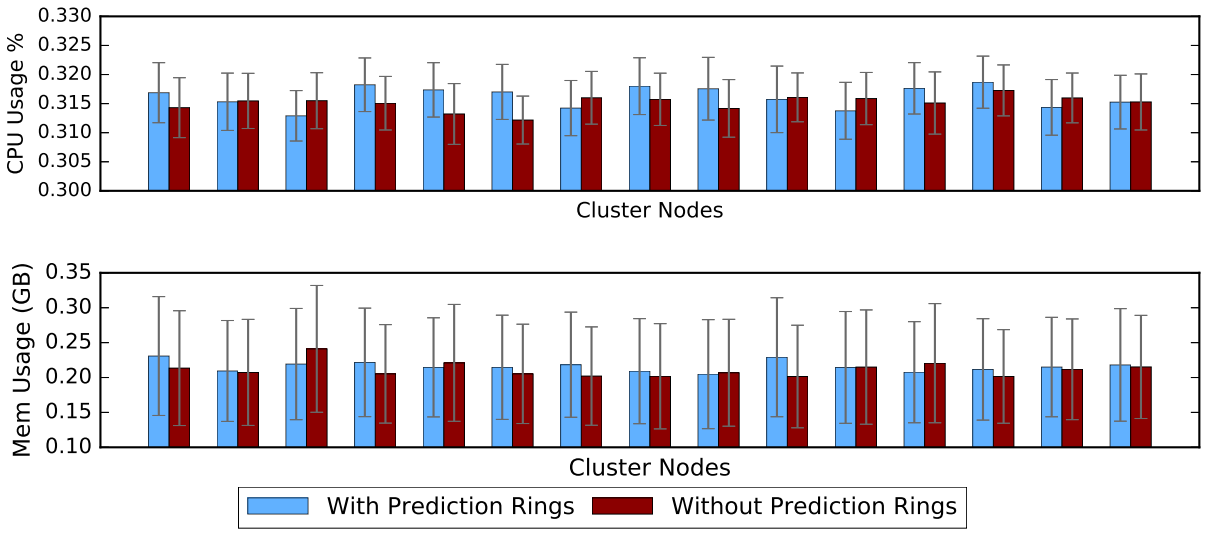


Figure 5.16: CPU and memory overhead of running prediction ring algorithm and maintaining relevant data structures.

is still a valid comparison because it captures the processing and memory overheads caused by all periodic monitoring and reporting operations.

Figure 5.16 shows the average CPU and memory utilization at each node with and without the online scheduling algorithm. We observed a high standard error in the memory utilization readings due to periodic garbage collection cycles. Single tail two sample t-tests were performed to check if our online scheduling algorithm caused a significant overhead. CPU utilization has increased slightly due to the online scheduling algorithm (p-value $= 0.03896$, $\alpha = 0.05$) and there was no significant evidence to suggest that the memory utilization has increased (p-value $= 0.08924$, $\alpha = 0.05$). There was approximately a 0.33% increase in CPU utilization, which we believe is acceptable given that it did not disrupt the regular execution of computations. Further, this benchmark substantiates our claim: prediction rings are lightweight and do not introduce significant memory pressure.

Chapter 6

Sketched Streams

Previous chapters focused on improving the processing of voluminous data streams once ingested into the center (cloud) by ensuring efficient resource utilization. However, stream ingestion remains to be challenging due to:

- *Power constraints* - Most edge devices are battery powered or have limited power profiles. Communication is the dominant energy consuming task on these devices [16, 9, 17]; transferring voluminous data can become infeasible.

- *Limited bandwidth* - Edge devices are connected to the remainder of the data ingestion pipeline via wide area networks with limited bandwidth [14].

- *Data transfer costs* - When the processing middleware is operating in public clouds, users are billed for the volume of data transferred across data center network boundaries. This is in addition to the bandwidth costs for the data transfer.

Some of the stream processing related challenges that we have not considered in previous chapters include:

- *Reduction of processing workload* - The amount of processing required at a stream processing cluster is proportional to the volume of the input streams in-general. The processing requirements directly translate into the size of the resource pool allocated for the stream processing job.

- *Reprocessing past data* - Changes in business requirements, discovery of bugs entail reprocessing *past* data using the updated versions of the applications [13, 22]. This entails storing data streams for extended periods of time.

With the ability to add limited processing and storage capabilities to the edge devices, and decentralized data processing initiatives such as cloudlets [27], distributed telco clouds [28], and

fog computing [96], several attempts have been made to preprocess/process data at the edges and reduce data transfer to the center. The current work, centered around edge processing to address data ingestion challenges, can be broadly categorized as: (1) deploying part of the stream processing topology at the edges (federated stream processing) [14, 58, 16, 59], (2) reducing the number of messages sent to the center (static and dynamic sampling, sending a derived stream instead of the raw stream) [60, 61, 62, 64, 63], and (3) reducing the size of the payload (lossy and lossless compression, encoding techniques) [67, 68, 9, 69]. There is a rich body of work on improving stream processing at the center through efficient resource management and scheduling of operators. These solutions only partially address the data ingestion and processing challenges and have the following drawbacks.

- *Limited applicability* - The applicability of federated stream processing is constrained due to the limited processing power and storage capabilities of edge devices, and non-local data dependencies.

- *Poor support for multi-feature streams* - Majority of the data reduction techniques are designed for single-feature streams. Applicability to multi-feature streams entails expensive stream joins at the center.

- *Focuses only on reducing the data transfer* - Most data reduction techniques focus only reducing the data transfer and reconstruct the entire stream at the center [97], and do not address the data processing challenges.

- *Considering input streams as transient* - The aforementioned schemes employ edge processing directives that are applicable to the current user requirements and consider data streams as transient (process and discard). This precludes reprocessing past data to support newer versions of the stream processing applications.

Our methodology is centered around the idea of *sketched streams*. Our reference implementation is code named *Pebbles*. An observational stream is partitioned into contiguous, non-overlapping temporal windows called *segments* and observations within each of the segments are

represented using a sketch — a space efficient representation of the multi-feature observations. Towards this end, as part of this study we present a new sketching algorithm, referred to as *Pebbles sketching algorithm*, designed especially for time-series data streams. Sketches are already leveraged in implementing both in-memory and on-disk storage for time-series data [98, 99, 100, 101]. Sketches are also used for implementing metadata management schemes. Effective metadata management is key to supporting queries over large scientific data collections [102, 103]. For example, the Galileo subsystem [104] leverages sketching based metadata management scheme to incorporate support for queries that are approximate [104], facilitate analytics [105], identify anomalies [106], and are geometry constrained [107].

Pebbles sketching algorithm is different from other frequency-based sketch algorithms such as Count-Min [108] and Misra-Gries [109], because it can preserve the ordering between observations within a segment, which is critical for most stream processing use cases. Existing frequency-based sketch algorithms offer point queries limited to cardinality or occurrence frequencies [110]. We leverage processing capabilities at the edges of the network to preprocess streams to produce *sketched streams*. In addition to the ability to reconstruct the ordered stream segments from the sketch and supporting frequency queries, Pebbles sketches also support certain stream processing related operations on stream segments such as partitioning and transformations. We validate the proposed methodology using three different real-world streaming datasets from diverse domains such as industrial monitoring, smart homes, and atmospheric monitoring.

Specific contributions we make in chapter include:

- A new sketching algorithm designed from the ground up for space-efficient, order-preserving representations of multi-feature, time-series data streams with guaranteed accuracy
- A novel methodology based on data sketching to address both ingestion and processing challenges pertaining to data streams originating in CSE settings
- High-performance, sketch-aware stream processing API that can be seamlessly integrated with existing data infrastructures used by organizations

6.1 System Architecture

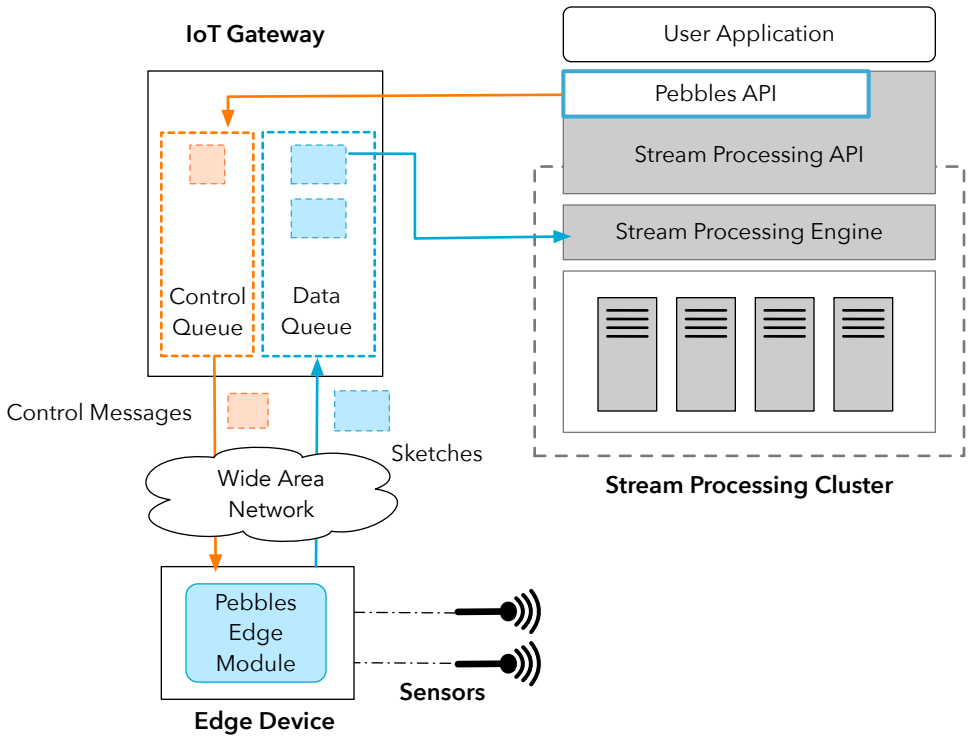


Figure 6.1: System Architecture of Pebbles. Pebbles edge module is responsible for generating sketches and send them over to the center via the IoT gateway. Pebbles center module extends the API of the stream processing engines allowing users to write applications on sketched data streams.

In this section, we introduce the key components and their interactions in our systems architecture, code named *Pebbles*. This is depicted in Figure 6.1.

6.1.1 Pebbles Edge Module

The Pebbles edge module is deployed in proximity to the source of the data stream. Edge modules are responsible for converting the observational streams into sketched streams. Data from sensors can be fetched into the edge module using either push or pull ingestion modes. For instance, a data collector node of a sensor network can be used to deploy the edge module which contacts the individual sensors at regular intervals and construct a multi-feature observation [9]. Cloudlets [27], fog computing devices [96], mobile phones, and telco clouds at the edges [28] are

other possible target devices to run Pebbles edge modules. Alternatively, the methodology of the edge module we explain in the following sections can be integrated into various edge computing modules such as Amazon's Greengrass [111] and Apache Edgent [30]. The sketches generated at the end of each segment is transferred to the IoT gateway via MQTT [15] or TCP. MQTT is a lightweight machine-to-machine (M2M) protocol built on top of TCP/IP to be used devices with low power profiles and unreliable networks.

6.1.2 IoT Gateway

The IoT gateway acts as the intermediary to provide space and time decoupling between the edge modules and the data processing middleware (in center). Edge modules can join and leave the data ingestion pipeline without any changes in the user applications running at the center. The stream processing cluster can be subjected to horizontal scaling, node failures, etc. without requiring any changes to data sink configurations used by the edge modules. This design also facilitates aggregating sub-streams from multiple, spatially distributed data sources into a single data stream. Further, the IoT gateway is used to establish a control channel from the center to the edge modules. IoT gateways are a common component in IoT deployments [33, 8] and usually implemented using a message broker. In our implementation, we used an Apache Kafka [18] to implement the IoT gateway. Using a message broker supporting multiple consumers like Kafka enables other data consuming applications within the organization (e.g.: batch jobs) in addition to the stream processing jobs to consume the data streams without having to maintain multiple copies of data. In a real world deployment, it is possible to use the existing message brokers in the data ingestion infrastructure to implement the IoT gateway without any additional operational cost.

6.1.3 Pebbles Stream Processing Module

Pebbles stream processing module extends the API of the stream processing engine to handle sketched data streams. It acts as a facade for the sketched data streams by internally handling the materialization of sketches and in-sketch operations transparent to the user. Currently, Pebbles support Apache Storm [35] and Neptune [23] stream processing engines. Data ingestion operators

in stream processing applications (e.g.,: Spouts in Apache Storm) subscribe to the topics in the IoT gateway corresponding to the input data streams. Optionally, user applications can send control messages using the Pebbles API to the edge modules contributing to a particular data stream enabling server-initiated steering.

6.2 Methodology

We present our methodology based on sketched streams for processing voluminous, high-velocity data streams generated at the edges of the network. We leverage the power of edge devices to preprocess data to effectively reduce the volume and the cloud to provide scalable streaming analytics over the data streams. More specifically:

- At the edge devices, multi-feature observational streams are temporally partitioned and each partition is sketched using a new sketching algorithm producing compact representations of the data with the desired accuracy. Transferring of the resulting stream of sketches is efficient in terms of both energy and network footprint.

- Pebbles stream processing API enables writing stream processing applications on sketched data streams. It provides a range of processing strategies such as topK, bottomK, and sampling operations in addition to the traditional process all option provided by stream processing engines. Further, it provides efficient in-sketch operations for operations like filtering and partitioning to achieve high processing throughputs. By providing an extended API to the existing stream processing engines, Pebbles can be seamlessly integrated with existing data processing infrastructures and coexist with traditional stream processing applications.

- Preprocessing at the edge module has a minimal impact on the fidelity of the data stream and independent of the requirements of the current workload compared to traditional approaches such as sampling. These factors combined with the lower storage footprints of the sketched data streams enables long term storage within the IoT gateway to cater changes in the applications, hence providing better human fault tolerance.

6.2.1 Preprocessing at the Edges

The Pebbles edge module is responsible for converting a multi-feature observational stream into a stream of sketches. An observational stream is partitioned into non-overlapping windows, called *segments*, based on a preconfigured window length. Segments are similar to tumbling windows as defined in stream processing literature [13]. Observations within a segment is represented using a sketch instance. Figure 6.2 depicts the key components of the Pebbles edge module.

Once a multi-feature observation is made available to the Pebbles edge module, it is transformed into a *feature-bin combination (FBC)*. The resulting feature-bin combination is then included in the sketch corresponding to the current time segment by the *Sketch Manager*. The sketch Manager tracks the progress of the stream and splits it at the correct boundaries generating new segments. Once a new segment is instantiated, the sketch and the metadata corresponding to the previous segment is serialized and passed to the *Transport Handler*. The transport Handler is responsible for handling communications with the center via the IoT gateway via the configured transport protocol. Next, we will discuss the discretization and sketching phases in detail.

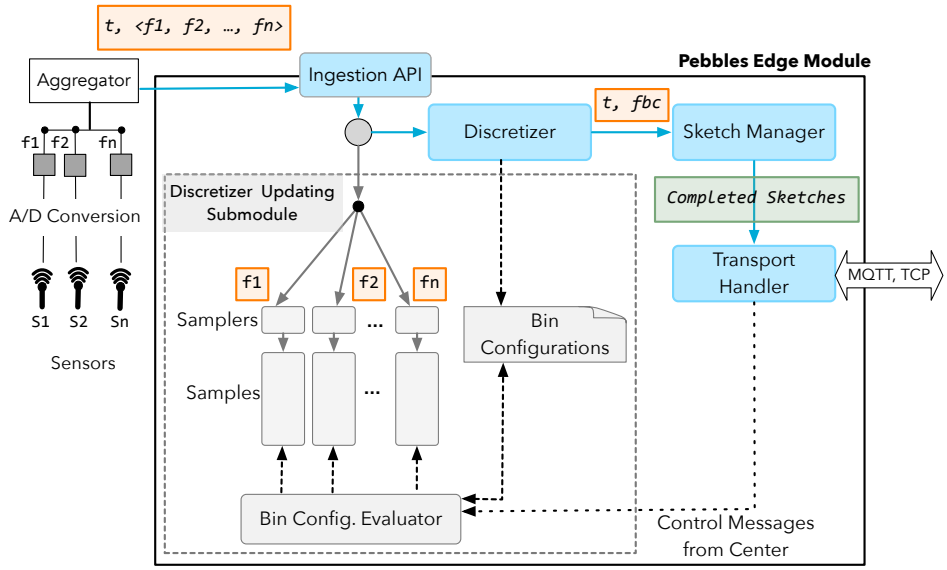


Figure 6.2: Components of the Pebbles edge module. Multi-feature observations are discretized and sketched prior to sending to the center. Bin configurations are dynamically updated as the stream evolves or upon the request from the user applications.

Discretization and Generating FBCs

Discretization is the process of discretizing continuous, individual feature values by mapping them to corresponding bins. To aid this process, the Pebbles edge module maintains a bin configuration for each feature in an observational stream — a set of points which partitions the range of the possible values of the feature into a given number of bins. During discretization, each feature value in an observation is mapped to the appropriate bin in its bin configuration. The identifiers of the corresponding bins are concatenated together to construct the feature-bin combination corresponding to the observation. For instance, let's consider a simple stream with two features with the bin configurations: {[100.0, 120.9), [120.9, 150.1), [150.1, 200)} and {[-0.1, -0.02), [-0.02, 0.05), [0.05, 1.1), [1.1, 1.9)}. Suppose the observation at time t' is $\langle t',\ \langle 129.1,\ 0.09 \rangle \rangle$. The first feature value 129.1 is mapped to the second bin, hence replaced by identifier 2. Similarly the second feature value is mapped to the third bin (with the identifier 3) resulting in a feature-bin combination of $\langle 2,\ 3 \rangle$. It should be noted that our methodology is equally applicable for single-feature streams as well in which case a FBC will contain a single bin identifier.

Discretization exploits the gradually evolving nature of feature streams to achieve compression similar to existing edge data reduction schemes [9, 64, 62]. Multiple subsequent feature values are likely to be resolved into the same bin, which eventually results in a few feature-bin combinations representing all the observations within a segment. We leverage this behavior within our sketching algorithm to effectively reduce the volumes of the data streams.

Estimating bin configurations: We leverage Online Kernel Density Estimation (oKDE) [112, 113] to autonomously generate bin configurations — both the number of bins and the range of values encapsulated by individual bins. For each feature, the probability density function is estimated using online kernel density estimation with respect to a sample of observed values. The numerical range of feature values is then partitioned into a given number of bins such that each bin having an equal area under the curve. This results in a bin configuration with each bin having an equal probability of landing the next observation. An example probability density function and the associated bin configuration is depicted in the first sub-figure of Figure 6.3. Discretization is

a non-reversible operation, therefore reconstructing the feature-value vectors from the feature-bin combinations incurs an estimation error, called the *discretization error*. In our reference implementation the middle value of a bin is considered the estimated value for the bin introducing a maximum error of half the bin width. Using an oKDE based binning configuration ensures that the high-density bins (bins with high probability) cause a lower discretization error. The number of bins in a bin configuration is the minimum number of bins that satisfy the desired discretization error threshold with respect to an online updated sample of observed values. For instance, in our benchmarks we used normalized root means square error (NRMSE) to measure the discretization error and set the threshold to 2.5% for each feature.

Enforcing an upper-bound on discretization error: We extend the basic discretization algorithm outlined in Section 6.2.1 to ensure that the discretization error for each feature does not exceed the pre-configured threshold. The basic discretization algorithm is susceptible to higher discretization errors in following scenarios.

- Anomalies - Anomalies usually fall into low-density bins (with low probability) therefore increasing the discretization error.
- Concept drift - As a stream evolves, the probability density function estimated by the oKDE may not adequately represent the current state of the stream. This may result in more observations being discretized into low-density bins.

Our improved discretization algorithm, *adaptive discretization*, provides a guaranteed upper bound on the discretization error by only discretizing an observed value if the discretization error is less than the threshold. Otherwise, the algorithm sends the observed value *as is* resulting in a discretization error of 0.0%. In the latter case, as an optimization we also inject a new bin into the bin configuration with the observed value as the middle value of the bin — we use the bin identifier of the newly introduced bin to construct the feature-bin combination. This optimization increases the likelihood of subsequent observations getting discretized using the newly introduced bin in the case of a concept drift. Discretized regions corresponding to a probability density function and

the associated bin configuration is illustrated in Figure 6.3. Another advantage of our adaptive discretization is the ability to preserve anomalies. Adaptive discretization prevents high-error discretization of anomalous feature values which is useful for anomaly detection use-cases; this is illustrated in Figure 6.4.

The adaptive discretization algorithm is outlined in Algorithm 1. The time complexity of discretizing an observation is $O(mlogk)$ where m is the number of features and k is the average number of bins in a bin configuration. The average number of bins is usually in the order of hundreds based on our empirical observations.

Algorithm 1 Adaptive discretization algorithm.

v_i : Value of feature i at a given time
bc_i : Bin configuration for feature i
f_{min}, f_{max} : Min and Max values for feature i
th_i : Discretization error threshold for feature i

function ADAPTIVE_DISCRETIZE($v_i, bc_i, f_{min}, f_{max}, th_i$)
    $b_k \leftarrow highest\_bin\_lower\_than\_value(bc_i, v_i)$
    $b_{k+1} \leftarrow next\_bin(bc_i, b_k)$
    $midpoint \leftarrow b_k + (b_{k+1} - b_k)/2$
    $nrmse \leftarrow RMSE(midpoint, v_i)/(f_{max} - f_{min})$
    if $nrmse < th_i$ **then return** b_k
    else
        if $v_i < midpoint$ **then**
            $b' = b_k + 2 * (v_i - b_k)$
        else
            $b' = b_{k+1} - 2 * (b_{k+1} - v_i)$
        end if
        $add\_new\_bin(bc_i, b')$ **return** b'
    end if
end function

Pebbles dynamically updates the bin configurations over time to mitigate concept drifts. Otherwise, the size of a bin configuration can increase due to injected bins causing issues such as insufficient memory to hold bin configurations at edge devices, higher discretization costs, and higher network footprints after serializing sketches (higher bin counts require more bits to repre-

sent a bin identifier when constructing feature-bin combinations). These updates are performed independently per feature in a particular stream. We evaluate the existing bin configuration without the injected bins against an online updated sample for each feature. If the overall discretization error for the sample is higher than the threshold, the system triggers a bin configuration update in the background. In order to capture concept drift, the sample should be representative of the recently observed values of a particular feature. Reservoir sampling [114] is a single-pass unbiased sampling algorithm designed for inputs with unknown sizes (such as streams) and ensures that every element has the same probability of getting selected — sampled elements are uniformly distributed throughput the stream. Weighted reservoir sampling extends this behavior through a weight function to assign different selection probabilities for the sample.

We support two weighted reservoir sampling algorithms with temporal bias functions that assign more weight to recent items in the stream: Aggarwal's reservoir sampling algorithm [115] and Uniform Variable Input Rate Biased Sampling [116]. While both algorithms are capable of main-

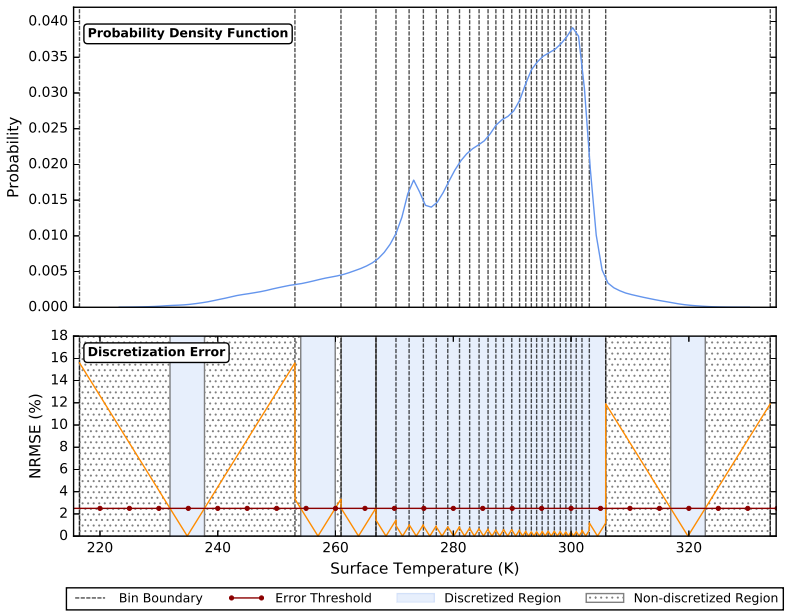


Figure 6.3: Probability density function (PDF) and the associated bin configuration for surface temperature feature in the NOAA North American Mesoscale forecast dataset at the beginning of the stream. Our discretization scheme enforces an upper-bound on the discretization error by selectively discretizing feature values that result in a lower discretization error. The PDF is updated in the background and the bins are recalculated as the stream evolves.

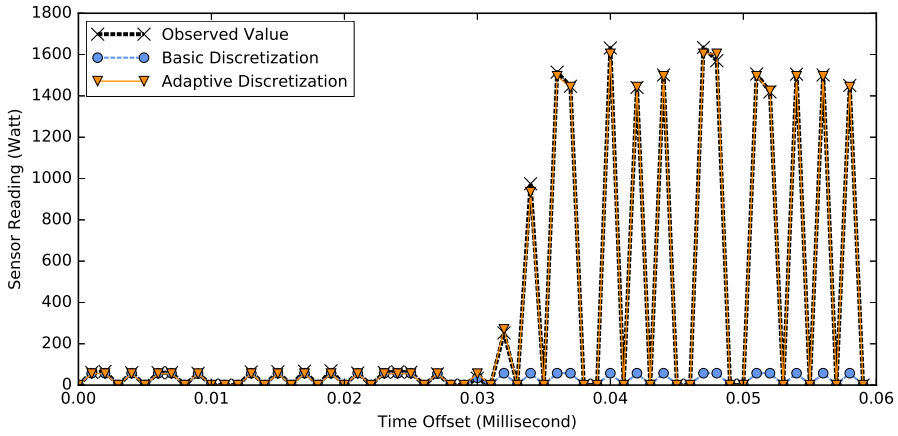


Figure 6.4: Adaptive discretization provides a guaranteed upper-bound for the discretization error while representing anomalies with a lower error compared to the basic discretization algorithm. This figure demonstrates how the adaptive discretization algorithm is able to capture a short-term peak in the load of one of the smart plugs in the smart home dataset.

taining samples that are well representative of the recent items in a stream, Uniform Variable Input Rate Biased Sampling is able to deal with variable observation rates of streams. Updating samples, evaluating current bin configuration, and updating bin configurations are performed in the background in parallel to the data ingestion for each feature separately. This allows the discretization process to autonomously be in sync with the evolution of individual features.

Figure 6.5 depicts a microbenchmark that demonstrates the effectiveness of different sampling algorithms on evaluating the effectiveness of a bin configuration. We recorded two metrics: (1) actual error - discretization error of the stream data over a sliding window of length 2 seconds (with 1 second sliding period) was used for the evaluation (2) observed error - discretization error of the sample at a given time. If the observed discretization error exceeds the preconfigured threshold (2.5% in this benchmark), an update to the bin configuration is triggered. As can be observed in Figure 6.5, observed errors for both weighted reservoir sampling techniques following the actual error closely; therefore, triggering bin configuration updates at appropriate intervals compared to the traditional reservoir sampling.

We also support serverside-steering for the discretization process. Stream processing applications can trigger a bin configuration update through control messages as shown in Figure 6.2. Either they can adjust the discretization error threshold or directly override the bin configuration of a par-

ticular feature with a custom bin configuration. Additional fine tuning of the discretization process such as adjusting the frequency of periodic bin configuration evaluation and enabling/disabling online updates to the bin configuration are supported. The resulting end-to-end dynamism of the data ingestion pipeline facilitates accommodating changes at the stream processing system such as addition and removal of new data processing jobs.

The bin configuration used for discretization should be available at the stream processing layer in order to estimate the individual feature values from the bin identifiers in the FBCs. We piggy-back on the actual messages (with serialized sketches) to the send the bin configuration updates instead of sending them over a separate stream. Otherwise a stream join is required at the center between the data stream and the bin configuration update stream to determine the corresponding bin configuration update required to resolve the FBCs. A higher QoS is used for guaranteed de-

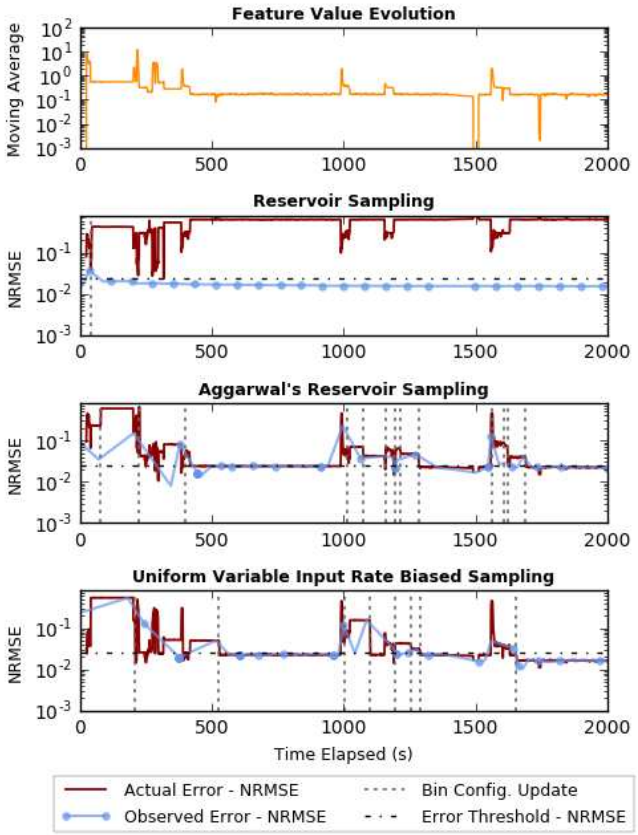


Figure 6.5: Effectiveness of weighted reservoir sampling when assessing bin configurations. Weighted reservoir algorithms can effectively capture the concept drift when compared to the reservoir sampling algorithm.

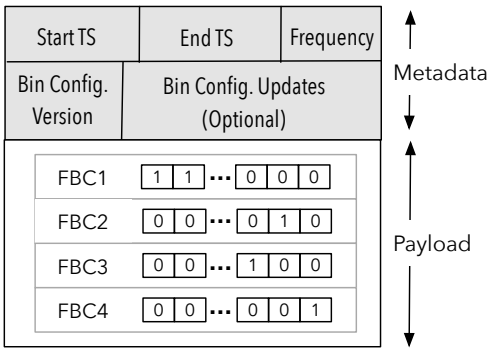


Figure 6.6: The structure of a Pebbles sketch instance. Metadata encapsulates information needed for materialization. Bitmaps in the payload are dynamically compressed.

livery of the messages containing the bin configuration updates. For each stream, a monotically increasing version number is assigned to a set of bin configuration updates performed within a time segment. Every sketch carries the version number of the bin configuration used for discretization when it was updated. This version number helps with dealing with out-of-order arrivals of messages, which are common especially when messages are transferred over public networks [117]. In case of an out-of-order arrivals, messages will temporarily reside in a memory buffer until the message with appropriate bin configuration update arrives. Most modern stream processing engines are equipped with built-in buffers to deal with such short-term, out-of-order arrivals.

6.2.2 Pebbles Sketching Algorithm

The Pebbles sketching algorithm is an *order preserving* sketching algorithm for *multi-feature streams* designed from the ground up. It addresses a common limitation of current frequency based sketching algorithms: the inability to preserve the ordering between observations. Maintaining the ordering between observations is critical for most stream processing use cases. For instance, maintaining temporal windows and tracking state changes over time require the ordering between observations.

Let's assume a data stream (already discretized into feature bin combinations) with observations produced at every p time units, i.e., with a frequency of $1/p$. The m observations produced within the time segment $[t_n, t_{n+mp})$ in stream S is denoted as;

$$S_{[t_n,t_{n+mp})} = \{(t_n, fbc_n), (t_{n+p}, fbc_{n+p}), ..., (t_{n+(m-1)p}, fbc_{n+(m-1)p})\}$$

where $fbc_i : n \leq i < n + mp$ is a feature bin combination occurring at time i. By encoding the timestamp t_i of an observation as a temporal offset o_i from the starting timestamp of the time segment t_n as a multiple of p, the same set of observations can be represented as;

$$S_{[t_n,t_{n+mp})} = t_n, \{o_i, f_i \mid o_i = (t_i - t_n)/p;\ 0 \leq i < m)\}$$

A Pebbles sketch maintains an inverted index of unique feature-bin combinations observed within a time segment as a set of ⟨key, value⟩ pairs where the set of keys corresponds to the set of unique feature-bin combinations. The value is the list of temporal offsets relative to the segment start timestamp (as calculated above) in ascending order where the given feature-bin combination occurred. For instance, let's consider a time segment [10,20) with an observation period of 2, therefore generating 5 observations — `(10, fbc1), (12, fbc1), (14, fbc2), (16, fbc1), (18, fbc1)`. The corresponding Pebbles sketch instance will store ⟨`fbc1, {0,1,3,4}`⟩ and ⟨`fbc2, {2}`⟩.

Pebbles sketching algorithm is a linear sketching algorithm where a linear transform over the input stream segments are applied to generate a sketch instance. In a linear sketch, an update on the sketch has the same impact irrespective of the previous updates [108]. The sketch of two adjacent segments can be calculated by merging sketches for the individual segments.

Maintaining an inverted index of temporal offsets can still incur a significant space overhead. We exploit the gradually evolving nature of most observational streams to gain further space reductions. Due to this gradually evolving nature, we posit that the consecutive observations are likely to transformed into a single feature bin combination once discretized. This results in stretches of consecutive temporal offsets within the inverted indexes making them amenable to compression using various techniques such as compressed bitmaps and inverted lists.

When using compressed bitmaps, each list of offsets within the inverted index is represented using a bitmap. The number of bits of the bitmap is equal to the number of observations within a segment — the position of a bit within a bitmap represents the temporal offset as a multiple of inter-observation interval. If a certain feature bin combination occurs, the corresponding bitmap is located within the inverted index and the bit in the position equal to the temporal offset is set. Following the example used before, the bitmap representation of the Pebbles sktech instance will be ⟨`fbc1, {11011}`⟩ and ⟨`fbc2, {00100}`⟩. Bitmap compression is a well studied research area with several contributions such as WAH [118], EWAH [119], CONCISE [120], and Roaring [121]. WAH, EWAH, and CONCISE are variants of run-length encoding where series of identical bits are compressed to the bit value and the count. Roaring adapts a hybrid compression technique incorporating an uncompressed integer list and uncompressed bitmap. Inverted lists are an alternative to compressed bitmaps which usually compresses the differences (a.k.a deltas) between the successive integers [122] — e.g., Variable Byte [123], PforDelta [124], and Simple9 [125]. In addition to being highly compressible, bitmaps support efficient bit-wise operations that are useful for high-performance data manipulations as discussed in Section 6.2.4.

Figure 6.6 illustrates the key building blocks of a Pebbles sketch. The metadata includes the temporal bounds of the segments and the frequency which is required for reconstructing the actual timestamps from the temporal offsets during the materialization. Additionally, the start timestamp provides ordering between messages within a stream. These metadata fields are included in every sketch instance to accommodate dynamic changes in sensing configurations such as frequency and segment duration. Version number of the bin configuration used for discretization, and the updates to the bin configurations (if there is any and applicable from next sketch onwards) are required for approximating the observations back from the FBCs.

Reference Implementation

For our reference implementation of Pebbles sketching algorithm, we used a combination of uncompressed bitmaps and Roaring bitmaps (paired with run-length encoding) to implement the inverted indexes. We chose Roaring bitmaps over other bitmap compression techniques for:

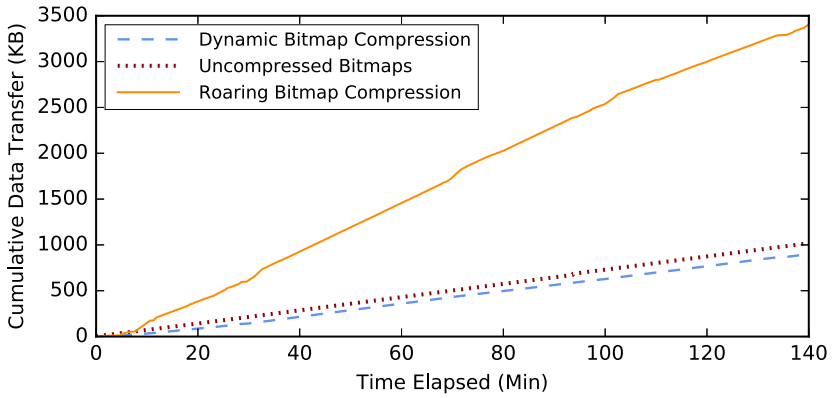


Figure 6.7: Effectiveness of dynamic compression in the reference implementation of the Pebbles sketch algorithm.

(1) space efficiency - desirable at resource constrained edge devices, and (2) faster decompression - for faster processing during materialization [122].

Our compression scheme is dynamic. None of the inverted index compression techniques are effective with highly randomized sequences of integers — in fact, they incur more overhead than maintaining an uncompressed bitmap. We initialize a Pebbles sketch instance with uncompressed bitmaps and maintain an online updated metric between the ratio of the number of bits set and the highest offset for each bitmap. At the end of the segment, if the ratio of these metrics is too high or too low, it is an indication that the bitmap is mostly empty or full. In such cases, we convert the uncompressed bitmap into a roaring bitmap and further optimize it with run-length encoding; otherwise we continue to use the uncompressed bitmap as is. In Figure 6.7, we demonstrate the effectiveness of the dynamic compression algorithm. We used the feature with the highest variability in Smart Home dataset [126] in this microbenchmark. Enabling roaring bitmaps by default is not effective due to the randomness of the data. Dynamically enabling the compression based on the occupancy heuristic can reduce the data transfer by $\sim$74% compared to using roaring-bitmaps and by $\sim$12% compared to using regular bitmaps alone.

The definition of Pebbles sketch algorithm does not preclude different implementations optimized for the use case in hand depending on the characteristics of the datasets and types of operations performed at the center.

6.2.3 Transferring Data to the Center

Once a segment expires, a sketch is transferred to the IoT gateway. Sketched streams reduces both the number of communications initiated as well as the overall volume of the data transferred. As we show in our benchmarks, the overhead of the sketch generation is outweighed by the energy saving due to reduced communication with the center. Also, this approach ensures that the limited bandwidth available at the edge devices are utilized more efficiently.

Sketched data streams are published over dedicated topics assigned for each stream within the IoT gateway similar to most regular data stream ingestion pipelines. Stream processing applications subscribe to the topics corresponding to their input streams. Given that the bin configuration updates are available as part of the metadata, if a new stream processing application is deployed it needs to consume the entire set of previous messages containing bin configuration updates in order to be able to construct an up to date bin configuration. This requires storage as well as processing of the entire stream which is expensive. There are two solutions to address this issue:

- Maintaining the history of bin configuration updates as shared state across all applications — e.g.: using an in-memory key-value store.

- Storing another copy of messages with bin configuration updates on a separate topic. New applications will consume the messages in this topic up to the offset where they want to start consuming the data stream.

In our reference implementation, we opted for the second option because it does not incur the operational overhead of another system. Bin configuration updates are typically manageable when compared to the total number of messages and can be subjected to merging if required — an older update can be replaced by a newer update to the bin configuration of a particular feature and bin updates to multiple features can be merged into a single message.

Due to the smaller storage footprints for sketched streams, they can be stored for extended periods of time in the IoT gateway. This enables reprocessing of the streams in case of any updates to the stream processing applications due to software bugs uncovered later or changed business

requirements. Kappa architecture [22] is one architectural pattern to support processing of past streams by storing them in message broker infrastructures (similar to the IoT gateway), and processing them with higher parallelism.

6.2.4 Stream Processing API

A typical stream processing API allows stream processing applications to be designed as directed acyclic graphs (DAG) comprising of stream operators (vertices) connected via streams (edges). An operator can be either an ingestion operator which ingests the input streams from the external sources into the DAG or stream processors which carries out part of the application logic and may produce a stream of messages to the downstream processors (derived streams). The Pebbles stream processing API follows the same model, but considers a different unit of workload — sketch instances for processing and communication between stream operators in the DAG.

Given that a Pebbles sketch instance is a compact representation of a set of observations from input streams or messages produced as part of the stream processor logic; the Pebbles stream processing API achieves the same benefits as *batching enabled stream processing*. In batching enabled stream processing, a group of messages is considered as the unit of data for transferring and processing in order to improve the throughput by amortizing the communication and stream processor execution costs [40]. Processor execution costs are improved through amortizing the costs associated with loading the instruction cache, scheduling, and context switches over a group of messages. Transferring larger payloads compared to smaller payloads improves the efficiency of the network utilization eventually leading to improved throughputs. Sketched streams are more compact than batched streams packing more observations in a single sketch instance through controlled trade-off of accuracy. This further improves processing throughput, memory overhead, as well as per-message execution overhead as we demonstrate in our systems benchmarks. Further, Pebbles sketches can support in-sketch operations (as explained below) where certain operations can be performed without expanding a sketch into the set of constituent observations delivering high-throughputs and reduced memory overhead.

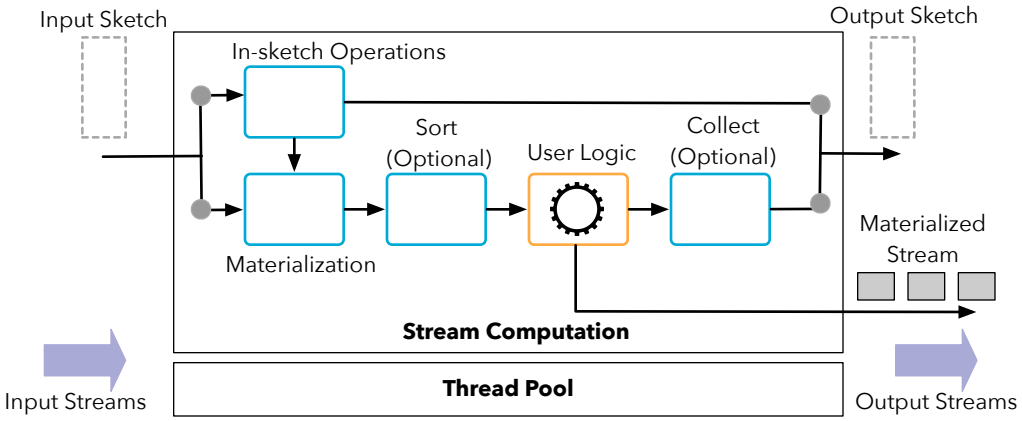


Figure 6.8: Implementation of the Pebbles sketch processing API at the center.

The Pebbles stream processing implementation is designed as an extension to the existing stream processing APIs. Currently we support Apache Storm [35] and Neptune [23]. It is deployed as a drop-in library similar to any runtime dependency without requiring any modifications to the stream processing engine itself. This is advantageous because:

- The Pebbles stream processing applications can execute alongside stream processing applications developed with regular stream processing APIs
- This supports hybrid stream processing applications - Applications where a subset of the stream processors are implemented using Pebbles API while the remainder are implemented using the regular stream processing API
- It builds on the large body of existing work in the development of feature-rich, robust stream processing engines

In this dissertation, we have used stream processing engines to prototype the Pebbles stream processing API. But our proposed methodology is applicable for micro-batching systems like Spark Streaming [52] as well. We will discuss the key concepts of the Pebbles stream processing API next.

Sketch Materialization

Materialization is the process of converting a sketch back to the set of individual discretized observations. Pebbles provides different materialization modes:

1. **all:** Generate *all* discretized observations represented by a sketch instance.

2. **sample:** Generate a *sample* of the discretized observations represented by a sketch based on a user-defined sampling function.

3. **topK/bottomK:** Generate the observations corresponding to *most/least frequent* observations.

Materialization internally expands bitmaps of the underlying Pebbles sketch instance and creates an observation for each set bit in the bitmaps. For materialization modes such as *topK/bottomK*, only a subset of the bitmaps are expanded based on their cardinality. The timestamps of the observations are generated based on the starting timestamp of the sketch and frequency (available as part of the metadata) combined with the offset (derived from the bit position within the bitmaps). Having support for multiple materialization modes at the API level not only enables stream processors to use the appropriate materialization mode for their processing logic, but also allows dynamically adjusting their processing semantics during runtime based on metrics related to operating conditions and workload. More specifically, a stream processor can dynamically switch between different materialization modes in runtime or adjust the parameters of its current processing mode. For instance, if a processor cannot keep up with the incoming stream, it can temporarily switch from *all* mode to *sample* mode essentially creating a load shedding setup or further reduce the sampling rate if it was already using *sample* mode.

The output of the materialization operation is made available either as an Iterator or a Java Stream [127]. Support for Java streams presents a functional style API to process the sketched streams with the added advantages of lazy evaluation and parallel execution.

Ordering of observations: The resulting set of discretized observations from a materialization operation is not ordered based on the timestamps by default. If required, users may opt for sorting the observations chronologically based on the observation timestamps.

Parallel execution: A Pebbles sketch instance can be materialized using multiple threads in parallel where each bitmap or contiguous block of a bitmap is expanded in parallel. Similar to sorting, parallel materialization should be explicitly enabled. Users can request a `parallelStream` if they use Java Streams or enable parallelization and provide a parallelization factor if they access observations through an Iterator.

In-Sketch Operations

In-sketch operations are a class of operations that can be directly performed on a Pebbles sketch instance without requiring a priori materialization. These operations directly work on the feature-bin combinations (keys) or bitmaps (values) of a Pebbles sketch instance. Following is a list of in-sketch operations supported by Pebbles sketches.

1. Key transformations - Apply transformations on FBCs such as projections, enrichments, and cleaning.

2. Filtering - Remove a subset of the FBCs based on a user-defined filtering criteria.

3. Key based partitioning - Horizontally partition a sketch based on a user-defined grouping criteria on FBCs.

4. Temporal partitioning - Vertically partition a sketch along the temporal axis such that a partition holds observations corresponding to a contiguous period of time.

5. Round-robin partitioning - Assign observations into partitions in a round-robin fashion.

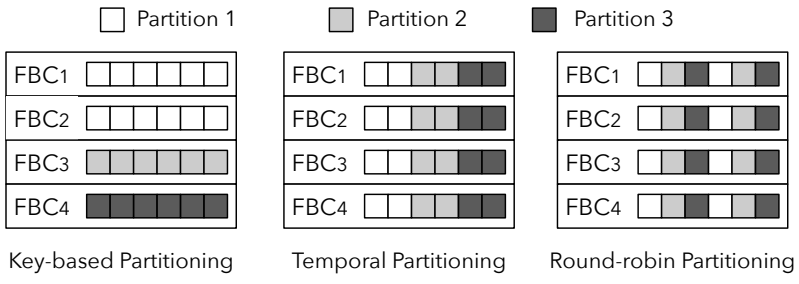


Figure 6.9: Supported partitioning schemes using in-sketch operations.

Figure 6.9 depicts various sketch partitioning schemes implemented as in-sketch operations. Given that these operations do not need to materialize the sketches in advance, they enable high-throughput processing as demonstrated in Section 6.3.4. Furthermore, they incur a less memory and garbage collection overhead by eliminating the instantiation of large number of short-lived objects. Some in-sketch operations such as temporal partitioning benefit from the fast bit-wise operations supported by modern bitmap implementations. For instance, temporal partitioning is implemented by performing a bit-wise AND operation on the bit masks generated for each partition.

Collect Operation

Collect operation, part of the Pebbles' Java Stream integration, constructs a sketch back from a stream of events. A stream processor can materialize a sketch as a Java stream, run the user-defined processing logic, and pack the derived event as a sketch using the *collect* operator ready to be transmitted to the remainder of the processing DAG. The applicability of this operator depends on the ability to represent an event as a $\langle$*key, temporal offset from the base timestamp*$\rangle$ tuple. Using sketches to transfer events within the DAG improves the overall performance of the stream processing application, especially with respect to throughput and bandwidth utilization as discussed earlier.

Pebbles Stream Processing Topology

Pebbles stream processing applications are directed, acyclic graphs of stream ingesters and stream processors connected via streams similar to any stream processing application. One key differences is in the Pebbles ingester, which consumes data from the IoT gateway and injects them into the remainder of the processing DAG. This ingester is responsible for resolving the feature-bin identifiers — each bin identifier is replaced by the bin boundaries based on the appropriate bin configuration. Because the bin identifier resolution is implemented as an in-sketch key-transformation operation it does not add a significance latency or processing overhead to the application. Once the bin identifiers are resolved, each Pebbles sketch instance becomes self-contained. If the past data

need to be consumed due to some reason, the ingester can be configured to consume data from the topic storing the bin configuration updates up to the required bin configuration version, and switch back to the sketched stream.

The remainder of the processing DAG can be implemented using combinations of either the Pebbles stream API and the regular stream processing API of the underlying stream processing engine. By maintaining the sketched streams as far as possible in the processing DAG using *in-sketch* and *collect* operations, the overall throughput of the stream processing graph can be improved substantially. The ability to use both APIs simultaneously within a single application facilitates a seamless interoperation of Pebbles with existing or legacy stream processing code. For instance, once a sketch is materialized, it can be processed as the equivalent regular data stream using the regular stream processing API.

6.3 Evaluation

We profiled the efficacy of our approach with respect to both data volume reduction at the edge devices and improved processing at the center using real world datasets.

6.3.1 Experimental Setup and Datasets

Edge devices: We used the Raspberry Pi 3 model B (1.2 Gz Quad Core Processor, 1 GB RAM) running Arch Linux and Oracle JDK 1.8.0_121 as the edge devices. Power measurements at the edge were carried out using Ubiquiti mFi mPower Mini smart plugs. MQTT was used as the messaging protocol between the edge devices and the center.

Stream Processing Cluster: We used HP DL60 servers (Intel Xeon E5-2620 2.40GHz processors, 16 GB RAM) running Fedora 28 and Oracle JDK 1.8.0_121. We used Apache Storm v1.2.2 for benchmarks involving Storm.

Datasets: We used three real world datasets in our benchmarks encompassing multiple domains: smart homes, industrial monitoring, and meteorological forecasting.

1. Smart homes dataset [126] includes readings from smart plugs deployed in set of households in Germany. Sensors attached to each smart plug periodically report current load and cumulative work since the last reset of the sensor. We preprocessed the dataset to organize data based on households (entities) such that each observation contains the current load on 12 different plugs (12 features).

2. Gas sensor array under dynamic gas mixtures dataset [7], created by the BioCircuits Institute at UCSD, includes time series data from 16 chemical sensors exposed to gas mixtures at varying concentration levels. For our benchmark, we used a subset of the dataset corresponding to Ethylene and CO gas mixture containing 18 features.

3. NOAA North American Mesoscale forecast dataset [2] contains periodic recordings of meteorological features by weather stations deployed across north America. We considered 10 features including temperature, humidity, and precipitation for year 2014.

Given that streaming workloads are unbounded in real world settings, we performed our benchmarks based on the duration of our datasets. Each stream was ingested using the original frequency associated with each dataset; smart homes, gas sensor array, and NOAA datasets required $\sim$43 mins, 6 hours, and 24 hours respectively for complete ingestion.

6.3.2 Efficacy of Pebbles at the Edge

Applicability to Single-feature Streams

Even though, Pebbles is designed for multi-feature streams, it is still applicable in single-feature stream settings. In this benchmark, we evaluate the efficacy of Pebbles at the edge when used with single-feature streams based on two metrics: amount of data transferred and energy consumption at the edge. We compare Pebbles with two other schemes specifically designed for single-feature streams in CSE settings.

AdaM: AdaM [64] is an adaptive monitoring framework designed for IoT devices with the goal of dynamically adjusting the sampling intensity based on the evolution of the data stream. During

the stable phases of the stream, the sampling rate is lowered to reduce the data transfer and energy consumption at the edge devices. The sampling period is adjusted based on a probabilistic metric calculated based on the deviation between the actual and estimated standard deviation (future observations are estimated using a probabilistic exponential moving average model).

LEC: Lossless Entropy Encoding [9] (LEC) is a lookup table based lightweight compression algorithm proposed for wireless sensor networks. A set of observations are modeled as a series of differences by taking the difference between each observation and its predecessor. LEC operates on the assumption that consecutive readings do not deviate much from each other, hence their differences are smaller. Each difference is then encoded using a bit string which is a concatenation between the number of bits required to represent the difference (looked up from a dictionary of Huffman codes) and the actual difference. Huffman codes ensure that frequent values are represented using shorter bit sequences.

For this benchmark, data from an individual entity was extracted from each of the three datasets to simulate a stream originating at an edge device. We have chosen the feature with highest variability in each of the datasets. AdaM was configured to provide the approximately the same accuracy as Pebbles (NRMSE = 2.5%). For LEC, consecutive observations were batched and compressed; the batching interval was set equal to the segment length used by Pebbles. The energy consumption encompasses the preprocessing and data transfer over MQTT.

Figure 6.10 summarizes the results of this benchmark. **Pebbles demonstrated significant improvements in data transfer (by $\sim$5 - 91.3$\times$) and energy consumption (by $\sim$5.4 - 20.6$\times$) across all three datasets.** In general, Pebbles' performance was comparable to the other two schemes. It should be noted that Pebbles is applicable for both multi-feature and single-feature data streams whereas the other schemes are limited to single-feature streams. We expect to see similar improvements as the cumulative ingested data volumes grow with both multiple edge devices and time.

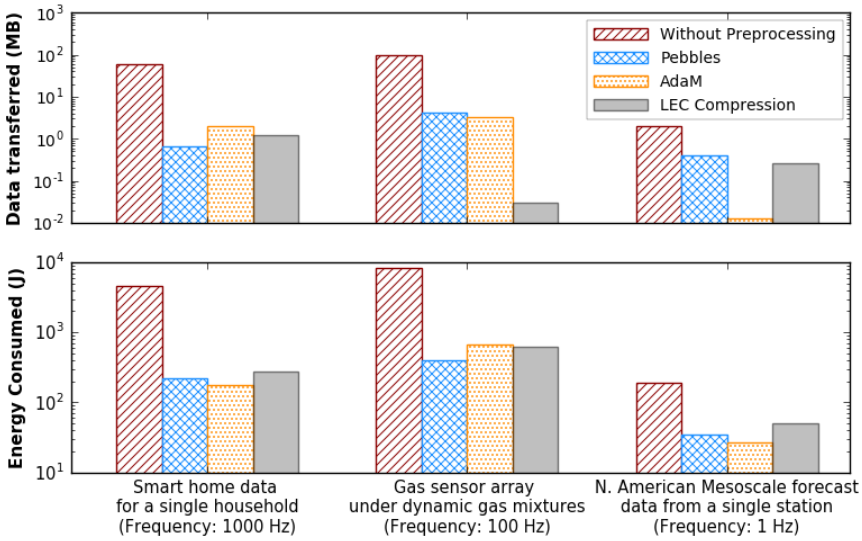


Figure 6.10: Effectiveness of Pebbles sketch algorithm in comparison to Lossless Entropy Compression (LEC) and Adaptive Monitoring Framework (AdaM) in **single-feature stream** settings.

Applicability to Multi-feature Streams

In this benchmark, we compared the effectiveness of Pebbles at the edge with multi-feature streams. Similar to Section 6.3.2, we have used data from a single entity but considered multiple features instead of a single feature. We compared Pebbles with binary compression using the fol-

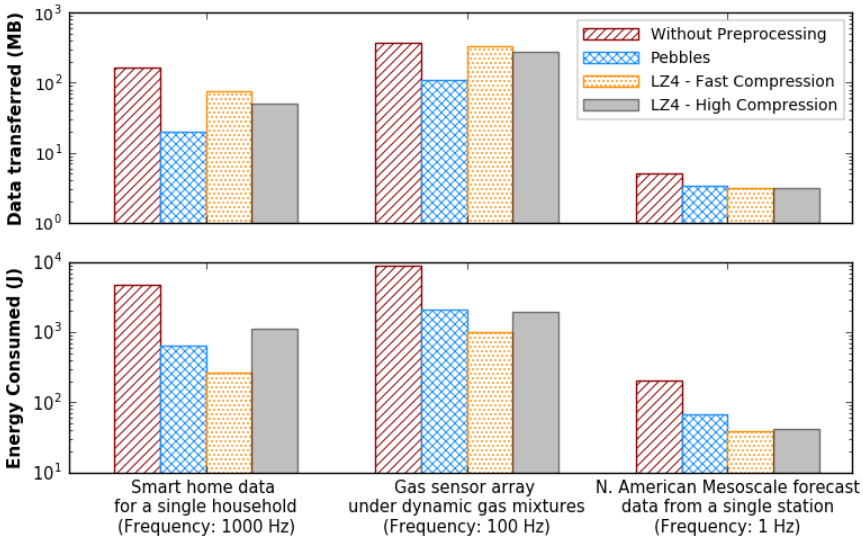


Figure 6.11: Effectiveness of Pebbles at the edge w.r.t amount of data transferred and energy consumption in **multi-feature stream** settings. We contrast Pebbles sketch algorithm's performance with binary compression using LZ4 algorithm.

lowing metrics: amount of data transfer and energy consumption. LZ4 [128] was chosen as the binary compression algorithm because of its ability to offer a good balance between the compression ratio and speed and configurable compression levels. LZ4 was evaluated under two settings: high compression (provides highest possible compression in the expense of compression speed and energy) and fast compression (provides faster and energy efficient compression by compromising the compression ratio). Observations occurring during a time interval equal to the length of a time segment were batched together for compression.

As listed in Figure 6.11, **Pebbles was able achieve significant improvements in data transfer (by $\sim$1.5 - 8.1$\times$) and energy consumption (by $\sim$3 - 7.3$\times$).** Pebbles was able demonstrated a comparable performance with respect to data transfer and energy consumption compared to both configurations of LZ4. LZ4 fast compression demonstrated the lowest energy consumption compared to other schemes. Pebbles not only improves the bandwidth utilization and energy consumption at the edges but also enables efficient processing at the center using its sketch-aware Stream Processing API. A disadvantage of binary compression is that the data needs to be decompressed at the stream operators — this precludes reducing the amount of processing that must be performed.

Evaluating the Discretization Error

We quantified the discretization error observed by individual features within each dataset as summarized using a histogram in Figure 6.12. The upper-bound on the discretization error is set to 2.5%. Pebbles discretization algorithm was able to maintain the discretization error well below the given threshold.

6.3.3 Trade-off Analysis: Discretization Error, Data Transfer, and Energy Consumption

We analyzed how the amount of data transfer and energy consumed at the edges change with different discretization error thresholds. We used the smart home dataset (with the entire set of features), and each experiment was launched with the same base bin configuration. The results are depicted in Figure 6.13. As the discretization error threshold increases (after an error threshold of

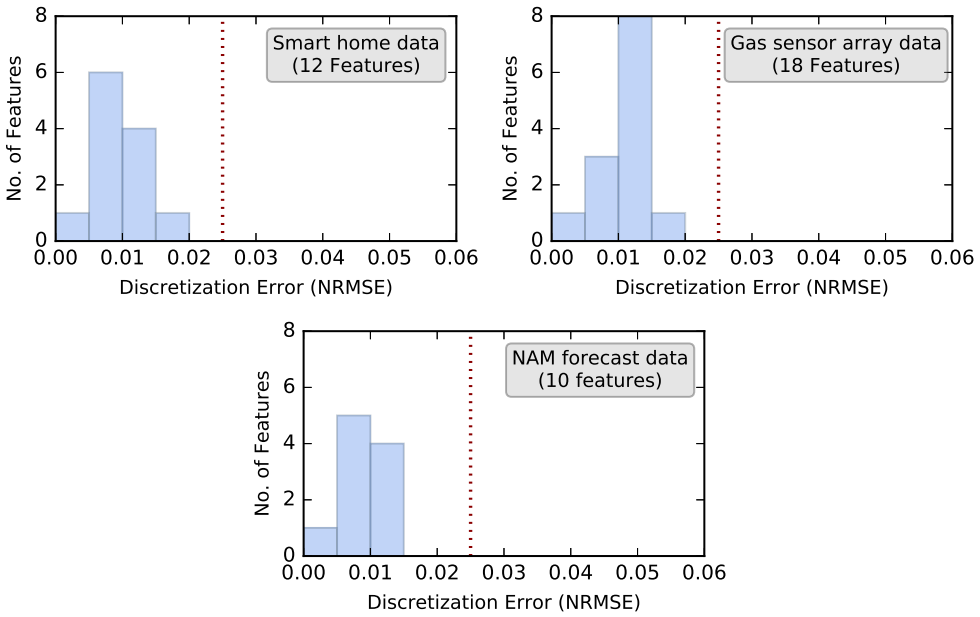


Figure 6.12: Histogram depicting the discretization error observed for each feature within different datasets measured as NRMSE. Pebbles is able to maintain the discretization error below a threshold of 2.5%.

5% NRMSE), fewer new bins are introduced by the adaptive discretization algorithm. This causes the gains from sketched streams generated at higher error thresholds to plateau resulting in slower growth in energy reduction and data transfer volumes. Furthermore, the high correlation between

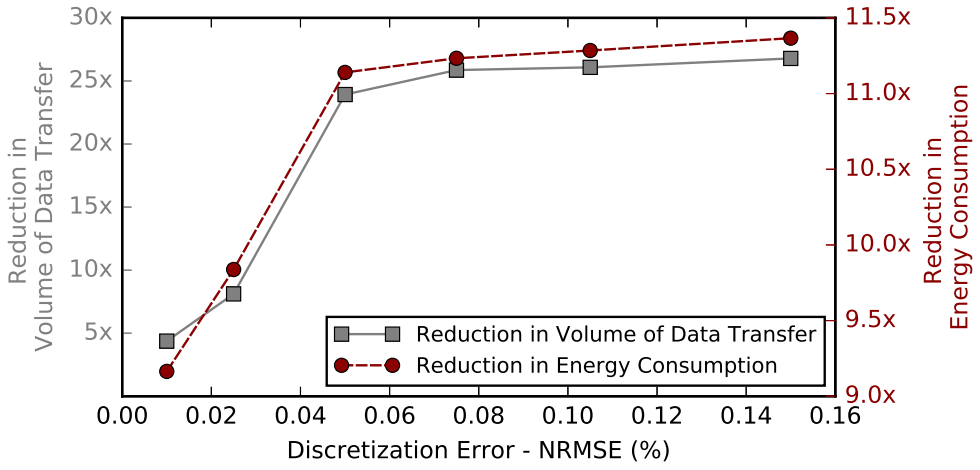
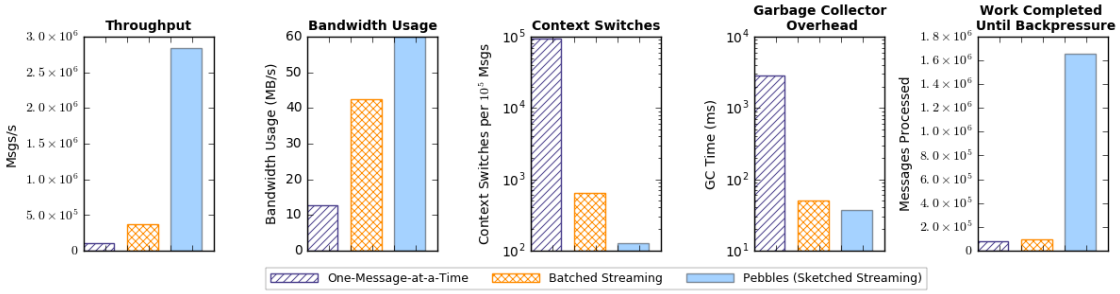
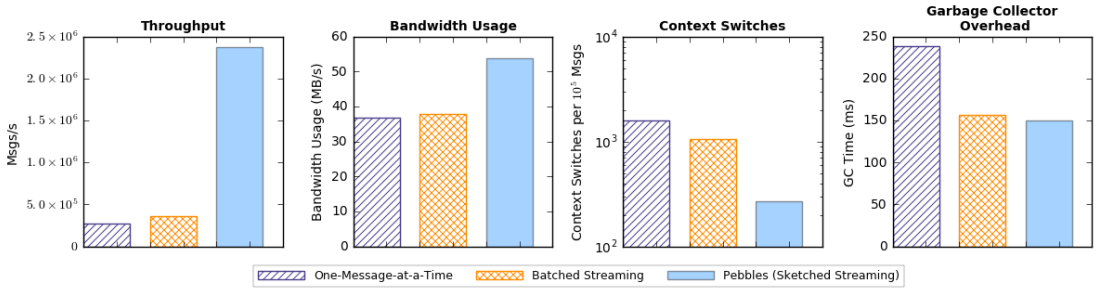


Figure 6.13: Trade-off analysis of discretization error, volume of data transfer, and energy consumption at the edges. The discretization error threshold of the Pebbles sketch algorithm is configurable and can achieve varying levels of reduction in both data transfer volumes and energy consumptions.



(a) With Neptune stream processing engine.



(b) With Apache Storm stream processing engine.

Figure 6.14: Sketched streams improves the performance at the center compared to traditional *processing one-message-at-a-time* and *batched streaming* as shown with two stream processing engines: Neptune and Apache Storm.

the amount of data transferred and the energy consumption also validates that communication is the dominant energy consumption factor at edge devices as verified by previous research [16, 9, 17].

6.3.4 Pebbles Stream Processing API

Improving Performance at the Center

Figure 6.14 depicts the performance improvements delivered by Pebbles stream processing APIs of Neptune [23] and Apache Storm [35] compared to the regular stream processing APIs. We evaluated regular stream processing APIs under two configurations: one-message-at-a-time and batched streaming. We used a data processing DAG with two vertices: an ingester and a processor. The processor is responsible for calculating the moving average over a sliding window for each feature in the smart plug dataset. Sketches are materialized (using the default serial mode) with sorting enabled before running the processing logic.

Pebbles outperforms the native one-message-at-a-time processing mode w.r.t. all metrics. For throughput, bandwidth utilization, and voluntary context-switches, Pebbles outperforms batched streams due to its compactness and the high transfer rate. Ingester nodes are able to transfer more sketches per unit time compared to the number of batches due to less serialization overhead. **Pebbles' Neptune API can achieve $\sim$27$\times$ and $\sim$7$\times$ improvement in throughput compared to one-message-at-a-time and batched streaming modes respectively.** Both Pebbles and batched streaming incur similar garbage collection overhead which is significantly lower than the one-message-at-a-time configuration. This is due to the significantly lower short-lived objects being created at the transport layer when multiple messages are transferred as a single unit. Finally, we measured how well each scheme can tolerate transient performance degradations that can occur due to varying workload and system conditions [21]. If a stream processor cannot keep up with the workload, the engine's backpressure scheme throttles upstream computations in the DAG to avoid possible buffer overflows at the stream processor. However, backpressure schemes reduce the overall throughput of a stream processing application and should therefore be deferred or avoided if possible. Lower memory footprint of sketched streams used in Pebbles helps the buffers to accommodate more unprocessed messages; this postpones backpressure maneuver as shown in the results. This may provide ample time for the processor to resume its normal operations without triggering backpressure. In our benchmark, **Pebbles can hold $\sim$18$\times$ more messages compared to the other two processing modes** (we do not see a significant difference between one-message-at-a-time mode and the batched stream because batched streaming does not compact the unprocessed messages). Since Storm does not expose any APIs or metrics to capture backpressure triggering, we performed this benchmark only with Neptune.

Performance of In-sketch Operations

We profiled the performance of in-sketch operations as shown in Figure 6.15. The throughput of different in-sketch operations were compared against the performance of the same operations performed on regular streams. For comparison purposes, we include the results corresponding to applying the regular operations on the materialized sketches as well. To isolate performance of

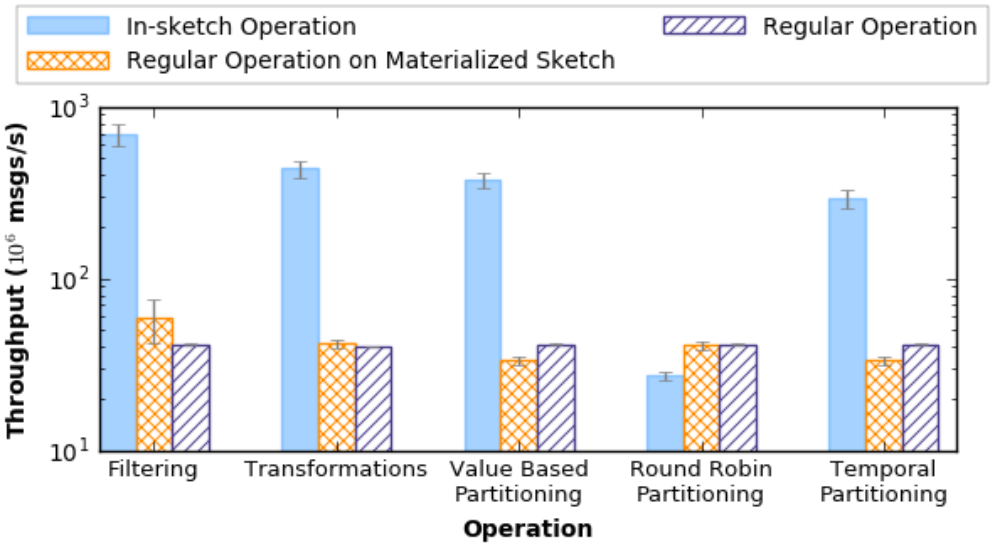


Figure 6.15: Performance comparison of multiple implementations of the common stream processing operations. In-sketch operations directly manipulate the sketch.

the operations, the data was generated locally within the stream processor. We did not enable the parallel processing for sketch materialization and in-sketch operations. Except for round-robin partitioning, the in-sketch operations outperform regular streams significantly. Cloning of bit masks for individual bitmaps was inefficient, therefore the in-sketch implementation of round-robin partitioning did not perform well. We plan to improve the performance on this operation in future by exploring libraries with better support for bit-shift operations. In addition to the gains in throughput, in-sketch operations incur reduced garbage collection overhead by not creating a large number of short lived objects each corresponding to a single message.

Chapter 7

Conclusions

Processing voluminous data streams in near-realtime presents unique challenges concerning stream ingestion and processing. These challenges are related to energy consumption at the edges, data transfer and storage costs, and inefficient resource utilization during processing. In this dissertation, we designed and validated our methodology to counteract these challenges. Our efforts are centered around:

- Aiming for efficient resource utilization in stream processing engines (***RQ-1***)
- Proactively alleviating performance hotspots in stream processing clusters (***RQ-2***)
- Compact stream representation through a controlled trading-off with accuracy (***RQ-1***, ***RQ-2***, ***RQ-3***)

Neptune, the stream processing engine we developed as part of this work, takes a holistic approach to efficient resource utilization that accounts for the CPU, memory, network, and kernel issues that arise. To this end, Neptune employs multiple design principles: efficient scheduling of workloads through the use of thread pools, minimizing context-switches by processing streams in batches, reusing objects to reduce memory utilization, application-level buffering, dynamic compression, and backpressure management. Neptune outperformed Apache Storm with respect to throughput ($\sim 8\times$ improvement) in our benchmarks based on real-world streaming datasets while consistently maintaining lower end-to-end latencies.

We propose an online scheduling algorithm, which continuously revises the placement of the stream processing computations through a series of micro-adjustments to avoid performance degradations due to variations in workloads and operating conditions. We rely on interference as the driving metric that informs these scheduling decisions. We introduce a light-weight new data structure called prediction rings to capture the interference experienced by a stream processing

computation. Prediction rings can proactively identify internal interference by tracking packet arrivals at both fine and coarse-grained scales. Tracking changes in resource utilization at a machine allows us to account for external interference from collocated processes. Together, prediction rings and interference scores, allow identifying computations that are most impacted by interference and also the machines that are best suited to host them. Our benchmarks demonstrate how our online scheduling algorithm can detect and alleviate performance hotspots resulting in improvements in throughout and latency.

We developed a holistic methodology for addressing both voluminous streams related to ingestion and processing challenges by leveraging on data sketching algorithms. Pebbles, our new sketching algorithm designed from the ground up, produces space-efficient representations of both single-feature and multi-feature streams through a controlled trading off of the resolution of individual feature values. Uniquely, our algorithm can preserve ordering between observations, handle anomalies effectively, and provide accuracy guarantees. Sketched streams effectively reduce data volumes at the edges of the network, resulting in lower data transfer costs and energy consumption footprints.

Our sketch-aware stream processing API can significantly improve the performance of the stream processing applications reducing resource footprints at the center. Space-efficient representation of streams allows storage of past data for extended periods, enabling reprocessing of past data to support new and updated applications. Pebbles is a drop-in extension to existing stream processing APIs enabling; (1) Pebbles applications to coexist with regular stream processing applications, and (2) development of hybrid stream processing applications that leverage both APIs within a single application.

In summary, the contributions of this work include: (1) a high throughput stream processing engine, (2) an online, proactive scheduling algorithm that can alleviate performance hotspots in the presence of dynamic workloads and operating conditions, (3) an order-preserving sketching algorithm for data streams with guaranteed accuracy bounds, and (4) a sketch-aware stream processing API for high performant and resource-efficient stream processing at the center.

Chapter 8

Future Work

Our online scheduling algorithm alleviates performance hotspots through a series of stream computation migrations within the existing set of stream processing nodes at a given time. Current work can be further extended to support a scale-out/in architecture where new stream processing nodes are provisioned or a subset of existing nodes are decommissioned. Scaling out is useful where a cluster's spare capacity at the moment is unable to keep pace with the workload. There is also the possibility to consolidate the placement of the stream computations to fewer nodes in an over-provisioned setting to release excess resources.

Stream computation migrations are expensive since they entail disruptions to the processing and also involve increased bandwidth consumption. Sketched streams can be integrated with the online-scheduling algorithm to reduce the number of migrations in approximate stream processing use cases. Stream computations experiencing a higher interference may benefit from the curtailed materializations where only a sample of the sketched data is materialized.

We believe sketched streams can be a useful construct in general edge computing and Internet of Things applications beyond stream processing. With the recent popularity of edge computing runtimes, both commercial [29, 111, 31] and open-source [30], energy- and network-efficient data transfers within an edge device cluster as well between edge devices and cloud services have become critical. For instance, edge computing applications implemented using AWS IoT SDK [29] require continuous data transfer with cloud services such as AWS Simple Storage Service (S3) and AWS Simple Queue Service (SQS). Amazon's Greengrass [111] enables building applications involving multiple edge devices; however, inter-device data transfers are still required in these settings. The low energy and network bandwidth profiles of sketched streams make it a potential building block implementing certain edge use cases. Depending on the use case, sketched streams based on other sketching algorithms such as Count-Min [108] or Frequent Items sketches [109] are also a possibility.

Table 8.1: Evaluating sketched stream based data transfer from edge to AWS cloud. We use count-min sketching algorithm to generate sketched streams. Performance of sketched streams are contrasted with binary compression (two compression levels of LZ4).

Approach	Data Transferred (MB/Hour)	Energy Consumption (J/Hour)	Estimated Cost (USD/Year)
Sketched Streams (1-min Segments & Count-Min Sketch)	0.21	230.70	12
LZ4 High Compression	3.41	250.34	12
LZ4 Fast Compression	3.71	217.57	12
Without Sketching (Baseline)	5.54	1586.83	540

As a proof of concept, we ran a benchmark with NOAA North American Mesoscale Forecast System (NAM) [2] data. Weather data with 10 features from 17 weather stations in northern Colorado spread across an area of $408km^2$ are transferred using an array of edge devices running Amazon IoT SDK into AWS cloud. Data from each weather station was handled by a separate Raspberry Pi. As summarized in Table 8.1, we were able to observe significant reductions in data transfer ($\sim$26$\times$) and energy consumption ($\sim$6.9$\times$). Further, we estimated the annual data ingestion costs for each approach (assuming a monthly billing cycle). This calculation only considers the cost of data transfer and does not include other costs such as device connectivity costs. Because message sizes in all our approaches were within the limit enforced by Amazon IoT, the cost was primarily determined by the number of messages transferred in units of 1 million messages. We were able to reduce the costs by 97.8% compared to regular data ingestion. Even though the reduction in data volume does not affect the ingestion cost in this scenario, it directly affects the storage costs. This may also contribute to increased data ingestion costs with other cloud providers such as Google Cloud where ingestions costs are calculated based on the volume of data transfer [129].

Bibliography

[1] I. D. C. (IDC). (2019) The growth in connected iot devices is expected to generate 79.4zb of data in 2025, according to a new idc forecast. [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=prUS45213219#:~:text=A%20new%20forecast%20from%20International,these%20devices%20will%20also%20grow.

[2] National Oceanic and Atmospheric Administration. (2018) The North American Mesoscale Forecast System. [Online]. Available: http://www.emc.ncep.noaa.gov/index.php?branch=NAM

[3] Massachusetts Department of Transportation. (2017) MassDOT developers' data sources. [Online]. Available: https://www.mass.gov/massdot-developers-data-sources

[4] United States Environmental Protection Agency. (2017) Air Data: Air Quality Data Collected at Outdoor Monitors Across the US. [Online]. Available: https://www.epa.gov/outdoor-air-quality-data

[5] M. Saeed, M. Villarroel, A. T. Reisner, G. Clifford, L.-W. Lehman, G. Moody, T. Heldt, T. H. Kyaw, B. Moody, and R. G. Mark, "Multiparameter intelligent monitoring in intensive care ii (mimic-ii): a public-access intensive care unit database," Critical care medicine, vol. 39, no. 5, p. 952, 2011.

[6] S. R. Islam, D. Kwak, M. H. Kabir, M. Hossain, and K.-S. Kwak, "The internet of things for health care: a comprehensive survey," IEEE Access, vol. 3, pp. 678–708, 2015.

[7] J. Fonollosa, S. Sheik, R. Huerta, and S. Marco, "Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring," Sensors and Actuators B: Chemical, vol. 215, pp. 618–629, 2015.

[8] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," Future generation computer systems, vol. 29, no. 7, pp. 1645–1660, 2013.

[9] F. Marcelloni and M. Vecchio, "An efficient lossless compression algorithm for tiny nodes of monitoring wireless sensor networks," The Computer Journal, vol. 52, no. 8, pp. 969–987, 2009.

[10] C.-Y. Wan, A. T. Campbell, and L. Krishnamurthy, "Pump-slowly, fetch-quickly (psfq): a reliable transport protocol for sensor networks," IEEE Journal on selected areas in Communications, vol. 23, no. 4, pp. 862–872, 2005.

[11] Y. Sankarasubramaniam, Ö. B. Akan, and I. F. Akyildiz, "Esrt: event-to-sink reliable transport in wireless sensor networks," in Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing. ACM, 2003, pp. 177–188.

[12] C.-W. Tsai, C.-F. Lai, M.-C. Chiang, and L. T. Yang, "Data mining for internet of things: A survey," IEEE Communications Surveys & Tutorials, vol. 16, no. 1, pp. 77–97, 2014.

[13] M. Kleppmann, Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. " O'Reilly Media, Inc.", 2017.

[14] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, "Spanedge: Towards unifying stream processing over central and near-the-edge data centers," in Edge Computing (SEC), IEEE/ACM Symposium on. IEEE, 2016, pp. 168–178.

[15] D. Locke, "Mq telemetry transport (mqtt) v3.1 protocol specification," IBM developerWorks Technical Library, 2010.

[16] P. Michalák and P. Watson, "Path2iot: A holistic, distributed stream processing system," in 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, 2017, pp. 25–32.

[17] P. Edara, A. Limaye, and K. Ramamritham, "Asynchronous in-network prediction: Efficient aggregation in sensor networks," ACM Transactions on Sensor Networks (TOSN), vol. 4, no. 4, p. 25, 2008.

[18] J. Kreps, N. Narkhede, J. Rao et al., "Kafka: A distributed messaging system for log processing," in Proceedings of the NetDB, 2011, pp. 1–7.

[19] A. W. Services. (2020) Amazon kinesis: Easily collect, process, and analyze video and data streams in real time. [Online]. Available: https://aws.amazon.com/kinesis/

[20] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," ACM computing surveys (CSUR), vol. 35, no. 2, pp. 114–131, 2003.

[21] J. Dean and L. A. Barroso, "The tail at scale," Communications of the ACM, vol. 56, no. 2, pp. 74–80, 2013.

[22] J. Kreps. (2014-07-02) Questioning the lambda architecture. [Online]. Available: https://www.oreilly.com/ideas/questioning-the-lambda-architecture

[23] T. Buddhika and S. Pallickara, "Neptune: Real time stream processing for internet of things and sensing environments," in IEEE Intl. Parallel & Distributed Processing Symposium (IPDPS), 2016, pp. 1143–1152.

[24] S. Pallickara, J. Ekanayake, and G. Fox, "Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce," in 2009 IEEE International Conference on Cluster Computing and Workshops. IEEE, 2009, pp. 1–10.

[25] T. Buddhika, R. Stern, K. Lindburg, K. Ericson, and S. Pallickara, "Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams," IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 12, pp. 3553–3569, Dec 2017.

[26] T. Buddhika, S. Pallickara, , and S. Pallickara, "Pebbles: Leveraging sketches for processing voluminous, high velocity data streams," IEEE Transactions on Parallel and Distributed Systems, vol. Under Review, 2019.

[27] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," IEEE pervasive Computing, no. 4, pp. 14–23, 2009.

[28] B. Theeten and N. Janssens, "Chive: Bandwidth optimized continuous querying in distributed clouds," IEEE Transactions on cloud computing, vol. 3, no. 2, pp. 219–232, 2015.

[29] (2020) Aws iot core. [Online]. Available: https://aws.amazon.com/iot-core/

[30] T. A. S. Foundation. (2016) Apache edgent: A community for accelerating analytics at the edge. [Online]. Available: http://edgent.apache.org/

[31] C. Inc. (2019) Cisco kinetic edge & fog processing module. [Online]. Available: https://www.cisco.com/c/dam/en/us/solutions/collateral/internet-of-things/kinetic-datasheet-efm.pdf

[32] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap)," Tech. Rep., 2014.

[33] G. C. Fox, S. Kamburugamuve, and R. D. Hartman, "Architecture and measured characteristics of a cloud based internet of things," in 2012 international conference on Collaboration Technologies and Systems (CTS). IEEE, 2012, pp. 6–12.

[34] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," Communications of the ACM, vol. 51, no. 1, pp. 107–113, 2008.

[35] "Apache Storm," https://storm.apache.org, 2015.

[36] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in Proceedings of the SIGMOD. ACM, 2015, pp. 239–250.

[37] "Apache Flink," https://flink.apache.org/index.html, 2015.

[38] M. Z. Tathagata Das and P. Wendell, "Diving into Apache Spark Streaming's Execution Model," https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html, 2015.

[39] A. S. Foundation, "Apache Storm: Trident API Overview," http://storm.apache.org/releases/current/Trident-API-Overview.html, 2017.

[40] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," ACM Computing Surveys, vol. 46, no. 4, p. 46, 2014.

[41] J. Wolf et al., "Soda: an optimizing scheduler for large-scale stream-based distributed computer systems," in ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing. Springer, 2008, pp. 306–325.

[42] M. R. Garey, R. L. Graham, D. S. Johnson, and A. C.-C. Yao, "Resource constrained scheduling as generalized bin packing," Series A Journal of Combinatorial Theory, vol. 21, no. 3, pp. 257–298, 1976.

[43] D. Fernández-Baca, "Allocating modules to processors in a distributed system," IEEE Transactions on Software Engineering, vol. 15, no. 11, pp. 1427–1436, 1989.

[44] M. Wall, "A genetic algorithm for resource-constrained scheduling," Thesis, 1996.

[45] J.-K. Kim, H. J. Siegel, A. A. Maciejewski, and R. Eigenmann, "Dynamic resource management in energy constrained heterogeneous computing systems using voltage scaling," IEEE Transactions Parallel and Distributed Systems, vol. 19, no. 11, pp. 1445–1457, 2008.

[46] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in Proc. of the ACM Middleware Conference, 2015, pp. 149–161.

[47] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in Proc. of the ACM DEBS, 2013, pp. 207–218.

[48] J. Ghaderi, S. Shakkottai, and R. Srikant, "Scheduling storms and streams in the cloud," in ACM SIGMETRICS Performance Evaluation Review, vol. 43, no. 1, 2015, pp. 439–440.

[49] Y. Xing et al., "Providing resiliency to load variations in distributed stream processing," in Proc. of the 32nd international conference on Very large data bases. VLDB Endowment, 2006, pp. 775–786.

[50] "Apache Storm - Scheduler," http://storm.apache.org/releases/1.0.1/Storm-Scheduler.html, 2015.

[51] N. Marz, "How to beat the CAP theorem," http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html, 2011.

[52] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in Proc. of the 4th USENIX HotCloud, 2012.

[53] T. Heinze, Y. Ji, Y. Pan, F. J. Grueneberger, Z. Jerzak, and C. Fetzer, "Elastic complex event processing under varying query load." in BD3@ VLDB. Citeseer, 2013, pp. 25–30.

[54] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in ICDCS. IEEE, 2014, pp. 535–544.

[55] A. Chatzistergiou and S. D. Viglas, "Fast heuristics for near-optimal task allocation in data stream processing over clusters," in CIKM. ACM, 2014, pp. 1579–1588.

[56] L. Fischer, T. Scharrenbach, and A. Bernstein, "Network-aware workload scheduling for scalable linked data stream processing," in Proc. of the 2013 Intl. Conference on Posters & Demos Track-Vol:1035. ceur-ws.org, pp. 281–284.

[57] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," SIAM Journal on scientific Computing, vol. 20, no. 1, pp. 359–392, 1998.

[58] E. G. Renart, J. Diaz-Montes, and M. Parashar, "Data-driven stream processing at the edge," in Fog and Edge Computing (ICFEC), 2017 IEEE 1st International Conference on. IEEE, 2017, pp. 31–40.

[59] S. Esteves, N. Janssens, B. Theeten, and L. Veiga, "Empowering stream processing through edge clouds," SIGMOD Rec., vol. 46, no. 3, pp. 23–28, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3156655.3156661

[60] D. Goldsmith and J. Brusey, "The spanish inquisition protocol—model based transmission reduction for wireless sensor networks," in SENSORS, 2010 IEEE. IEEE, 2010, pp. 2043–2048.

[61] J. Brusey, R. Rednic, E. I. Gaura, J. Kemp, and N. Poole, "Postural activity monitoring for increasing safety in bomb disposal missions," Measurement Science and Technology, vol. 20, no. 7, p. 075204, 2009.

[62] E. I. Gaura, J. Brusey, and R. Wilkins, "Bare necessities—knowledge-driven wsn design," in SENSORS, 2011 IEEE. IEEE, 2011, pp. 66–70.

[63] W. Sherchan, P. P. Jayaraman, S. Krishnaswamy, A. Zaslavsky, S. Loke, and A. Sinha, "Using on-the-move mining for mobile crowdsensing," in Mobile Data Management (MDM), 2012 IEEE 13th International Conference on. IEEE, 2012, pp. 115–124.

[64] D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Adam: An adaptive monitoring framework for sampling and filtering on iot devices," in Big Data (Big Data), 2015 IEEE International Conference on. IEEE, 2015, pp. 717–726.

[65] J. Traub, S. Breß, T. Rabl, A. Katsifodimos, and V. Markl, "Optimized on-demand data streaming from sensor nodes," in Proceedings of the 2017 Symposium on Cloud Computing. ACM, 2017, pp. 586–597.

[66] A. Deligiannakis, Y. Kotidis, and N. Roussopoulos, "Compressing historical information in sensor networks," in Proceedings of the 2004 ACM SIGMOD International Conference on

Management of Data, ser. SIGMOD '04. New York, NY, USA: ACM, 2004, pp. 527–538. [Online]. Available: http://doi.acm.org/10.1145/1007568.1007628

[67] C. M. Sadler and M. Martonosi, "Data compression algorithms for energy-constrained devices in delay tolerant networks," in Proceedings of the 4th international conference on Embedded networked sensor systems. ACM, 2006, pp. 265–278.

[68] M. Oberhumer. minilzo: mini version of the lzo real-time data compression library. [Online]. Available: http://www.oberhumer.com/opensource/lzo/

[69] T. Schoellhammer, B. Greenstein, E. Osterweil, M. Wimbrow, and D. Estrin, "Lightweight temporal compression of microclimate datasets," 2004.

[70] S. Pallickara, J. Ekanayake, and G. Fox, "An overview of the granules runtime for cloud computing," in 2008 IEEE Fourth International Conference on eScience. IEEE, 2008, pp. 412–413.

[71] G. Fox and S. Pallickara, "Deploying the naradabrokering substrate in aiding efficient web and grid service interactions," Proceedings of the IEEE, vol. 93, no. 3, pp. 564–577, 2005.

[72] G. Fox, S. Pallickara, M. Pierce, and H. Gadgil, "Building messaging substrates for web and grid applications," Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, vol. 363, no. 1833, pp. 1757–1773, 2005.

[73] G. Fox, S. Lim, S. Pallickara, and M. Pierce, "Message-based cellular peer-to-peer grids: foundations for secure federation and autonomic services," Future Generation Computer Systems, vol. 21, no. 3, pp. 401–415, 2005.

[74] G. Fox, S. Pallickara, and X. Rao, "Towards enabling peer-to-peer grids," Concurrency and Computation: Practice and Experience, vol. 17, no. 7-8, pp. 1109–1131, 2005.

[75] A. Uyar, S. Pallickara, and G. C. Fox, "Towards an architecture for audio/video conferencing in distributed brokering systems." in Communications in Computing, 2003, pp. 17–23.

[76] Michael G. Noll, "Understanding the Internal Message Buffers of Storm," http://www.michael-noll.com/blog/2013/06/21/understanding-storm-internal-message-buffers/, 2013.

[77] Z. Nabi, E. Bouillet, A. Bainbridge, and C. Thomas, "Of streams and storms," IBM White Paper, 2014.

[78] M. Sústrik, "ZeroMQ," http://aosabook.org/en/zeromq.html.

[79] S. Fu, Y. Zhang, Y. Jiang, C.-Y. Shih, and P. J. Marron, "An experimental study towards understanding data delivery performance over a wsn link," arXiv preprint arXiv:1411.5210, 2014.

[80] "6^{th} acm debs - grand challenge," http://www.csw.inf.fu-berlin.de/debs2012/grandchallenge.html.

[81] S. Imai, T. Chestna, and C. A. Varela, "Elastic scalable cloud computing using application-level migration," in Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on. IEEE, 2012, pp. 91–98.

[82] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in USENIX Annual Technical Conference, vol. 8, 2010, p. 9.

[83] K. E. Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela, "The internet operating system: Middleware for adaptive distributed computing," Journal of High Performance Computing Applications, vol. 20 (4), pp. 467–480, 2006.

[84] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," Journal of the ACM, vol. 46 (5), pp. 720–748, 1999.

[85] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano, "Auto-scaling techniques for elastic applications in cloud environments," University of Basque Country, Tech. Rep. EHU-KAT-IK-09, vol. 12, p. 2012, 2012.

[86] "Engineering Statistics Handbook - Triple Exponential Smoothing," http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc435.htm, National Institute of Standards and Technology, 2013.

[87] N. Wagner et al., "Time series forecasting for dynamic environments: the dyfor genetic program model," IEEE transactions on evolutionary computation, vol. 11, no. 4, pp. 433–452, 2007.

[88] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, Time series analysis: forecasting and control. John Wiley & Sons, 2015.

[89] G. A. Agha, "Actors: A model of concurrent computation in distributed systems." DTIC Document, Tech. Rep., 1985.

[90] C. A. Varela and G. Agha, Programming Distributed Computing Systems: A Foundational Approach, 2013.

[91] J. Kohlmorgen, K.-R. Müller, J. Rittweger, and K. Pawelzik, "Identification of nonstationary dynamics in physiological recordings," Biological Cybernetics, vol. 83, no. 1, pp. 73–84, 2000.

[92] A. L. Goldberger et al., "Physiobank, physiotoolkit, and physionet components of a new research resource for complex physiologic signals," Circulation, vol. 101, no. 23, pp. e215–e220, 2000.

[93] J. Pan and W. J. Tompkins, "A real-time qrs detection algorithm," Biomedical Engineering, IEEE Transactions on, no. 3, pp. 230–236, 1985.

[94] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in Proc. of the ACM SIGMOD, 2013, pp. 725–736.

[95] F. Al-Haidari, M. Sqalli, and K. Salah, "Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources," in Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on, vol. 2. IEEE, 2013, pp. 256–261.

[96] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in Proceedings of the first edition of the MCC workshop on Mobile cloud computing. ACM, 2012, pp. 13–16.

[97] A. Papageorgiou, B. Cheng, and E. Kovacs, "Reconstructability-aware filtering and forwarding of time series data in internet-of-things architectures," in 2015 IEEE International Congress on Big Data. IEEE, 2015, pp. 576–583.

[98] T. Buddhika, M. Malensek, S. Pallickara, , and S. Pallickara, "Living on the edge: Data transmission, storage, and analytics in continuous sensing environments," ACM Transactions on Internet of Things, vol. In Revision, 2019.

[99] T. Buddhika, M. Malensek, S. L. Pallickara, and S. Pallickara, "Synopsis: A distributed sketch over voluminous spatiotemporal observational streams," IEEE Transactions on Knowledge & Data Engineering, no. 11, pp. 2552–2566, 2017.

[100] D. Rammer, T. Buddhika, M. Malensek, S. Pallickara, and S. Pallickara, "Enabling fast exploratory analyses over voluminous spatiotemporal data using analytical engines," IEEE Transactions on Big Data, 2019.

[101] D. Rammer, W. Budgaga, T. Buddhika, S. Pallickara, and S. L. Pallickara, "Alleviating i/o inefficiencies to enable effective model training over voluminous, high-dimensional datasets," in 2018 IEEE International Conference on Big Data (Big Data). IEEE, 2018, pp. 468–477.

[102] S. L. Pallickara, S. Pallickara, M. Zupanski, and S. Sullivan, "Efficient metadata generation to enable interactive data discovery over large-scale scientific data collections," in

2010 IEEE Second International Conference on Cloud Computing Technology and Science. IEEE, 2010, pp. 573–580.

[103] S. L. Pallickara, S. Pallickara, and M. Zupanski, "Towards efficient data search and subsetting of large-scale atmospheric datasets," Future Generation Computer Systems, vol. 28, no. 1, pp. 112–118, 2012.

[104] M. Malensek, S. L. Pallickara, and S. Pallickara, "Galileo: A framework for distributed storage of high-throughput data streams," in 2011 Fourth IEEE International Conference on Utility and Cloud Computing. IEEE, 2011, pp. 17–24.

[105] M. Malensek, S. Pallickara, and S. Pallickara, "Analytic queries over geospatial time-series data using distributed hash tables," IEEE Transactions on Knowledge and Data Engineering, vol. 28, no. 6, pp. 1408–1422, 2016.

[106] W. Budgaga, M. Malensek, S. Lee Pallickara, and S. Pallickara, "A framework for scalable real-time anomaly detection over voluminous, geospatial data streams," Concurrency and Computation: Practice and Experience, vol. 29, no. 12, p. e4106, 2017.

[107] M. Malensek, S. Pallickara, and S. Pallickara, "Evaluating geospatial geometry and proximity queries using distributed hash tables," Computing in Science & Engineering, vol. 16, no. 4, pp. 53–61, 2014.

[108] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," Journal of Algorithms, vol. 55, no. 1, pp. 58–75, 2005.

[109] J. Misra and D. Gries, "Finding repeated elements," Science of computer programming, vol. 2, no. 2, pp. 143–152, 1982.

[110] G. Cormode, "Sketch techniques for approximate query processing," Foundations and Trends in Databases. NOW publishers, 2011.

[111] A. W. Services. (2019) Aws iot greengrass: Bring local compute, messaging, data caching, sync, and ml inference capabilities to edge devices. [Online]. Available: https://aws.amazon.com/greengrass/

[112] E. Parzen, "On estimation of a probability density function and mode," The annals of mathematical statistics, vol. 33, no. 3, pp. 1065–1076, 1962.

[113] M. Rosenblatt et al., "Remarks on some nonparametric estimates of a density function," The Annals of Mathematical Statistics, vol. 27, no. 3, pp. 832–837, 1956.

[114] J. S. Vitter, "Random sampling with a reservoir," ACM Transactions on Mathematical Software (TOMS), vol. 11, no. 1, pp. 37–57, 1985.

[115] C. C. Aggarwal, "On biased reservoir sampling in the presence of stream evolution," in Proceedings of the 32nd international conference on Very large data bases. VLDB Endowment, 2006, pp. 607–618.

[116] J. Arfa. (2016) Virbs and sampling events from streams. [Online]. Available: http://tech.magnetic.com/2016/04/virbs-sampling-events-from-streams.html

[117] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: fault-tolerant stream processing at internet scale," Proceedings of the VLDB Endowment, vol. 6, no. 11, pp. 1033–1044, 2013.

[118] K. Wu, "Notes on design and implementation of compressed bit vectors," 2001.

[119] D. Lemire, O. Kaser, and K. Aouiche, "Sorting improves word-aligned bitmap indexes," Data & Knowledge Engineering, vol. 69, no. 1, pp. 3–28, 2010.

[120] A. Colantonio and R. Di Pietro, "Concise: Compressed'n'composable integer set," arXiv preprint arXiv:1004.0403, 2010.

[121] S. Chambi, D. Lemire, O. Kaser, and R. Godin, "Better bitmap performance with roaring bitmaps," Software: practice and experience, vol. 46, no. 5, pp. 709–719, 2016.

[122] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson, "An experimental study of bitmap compression vs. inverted list compression," in Proceedings of the 2017 ACM International Conference on Management of Data. ACM, 2017, pp. 993–1008.

[123] D. Cutting and J. Pedersen, "Optimization for dynamic inverted index maintenance," in Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval. ACM, 1989, pp. 405–411.

[124] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar ram-cpu cache compression," in Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on. IEEE, 2006, pp. 59–59.

[125] V. N. Anh and A. Moffat, "Inverted index compression using word-aligned binary codes," Information Retrieval, vol. 8, no. 1, pp. 151–166, 2005.

[126] A. I. C. on Distributed and E. based Systems. (2014) DEBS 2014 Grand Challenge: Smart homes. [Online]. Available: http://debs.org/debs-2014-smart-homes/

[127] J. C. Process. (2012) Jsr 335: Lambda expressions for the javatm programming language. [Online]. Available: https://jcp.org/en/jsr/detail?id=335

[128] Y. Collet et al. (2013) Lz4: Extremely fast compression algorithm. [Online]. Available: https://lz4.github.io/lz4/

[129] (2020) Cloud iot core. [Online]. Available: https://cloud.google.com/iot-core/