Ph.D. DISSERTATION

End-to-End Optimizations for Fast and Efficient IoT Stream Processing in the Cloud

클라우드 환경에서 빠르고 효율적인 IoT 스트림 처리를 위한 엔드-투-엔드 최적화

AUGUST 2021

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Taegeon Um

End-to-End Optimizations for Fast and Efficient IoT Stream Processing in the Cloud

클라우드 환경에서 빠르고 효율적인 IoT 스트림 처리를 위한 엔드-투-엔드 최적화

지도교수 전 병 곤

이 논문을 공학박사 학위논문으로 제출함

2021년 3월

서울대학교 대학원

컴퓨터 공학부

엄 태 건

엄태건의 공학박사 학위논문을 인준함

2021년 6월

Abstract

As a large amount of data streams are generated from Internet of Things (IoT) devices, two types of IoT stream queries are deployed in the cloud. One is a small IoT-stream query, which continuously processes a few IoT data streams of end-users's IoT devices that have low input rates (e.g., one event per second). The other one is a big IoT-stream query, which is deployed by data scientists to continuously process a large number and huge amount of aggregated data streams that can suddenly fluctuate in a short period of time (bursty loads). However, existing work and stream systems fall short of handling such workloads efficiently because their query submission, compilation, execution, and resource acquisition layer are not optimized for the workloads.

This dissertation proposes two end-to-end optimization techniques— not only optimizing stream query execution layer (runtime), but also optimizing query submission, compiler, or resource acquisition layer. First, to minimize the number of cloud machines and maintenance cost of servers in processing many small IoT queries, we build Pluto, a new stream processing system that optimizes both query submission and execution layer for efficiently handling many small IoT stream queries. By decoupling IoT query submission and its code registration and offering new APIs, Pluto mitigates the bottleneck in query submission and enables efficient resource sharing across small IoT stream queries in the execution. Second, to quickly handle sudden bursty loads and scale out big IoT stream queries, we build Sponge, which is a new stream system that optimizes query compilation, execution, and resource acquisition layer altogether. For fast acquisition of new resources, Sponge uses a new cloud computing service, called Lambda, because it offers fast-to-start lightweight containers. Sponge then converts the streaming dataflow of big stream queries to overcome Lambda's resource constraint and to minimize scaling overheads at runtime.

Our evaluations show that the end-to-end optimization techniques significantly improve system throughput and latency compared to existing stream systems in handling a large number of small IoT stream queries and in handling bursty loads of big IoT stream queries.

Keywords: Stream processing, distributed data processing, IoT, Cloud, Lambda, Serverless framework

Student Number: 2014-22686

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 IoT Stream Workloads

Stream queries (jobs) continuously process real-time data and extract insights with low latency [99, 88, 98, 75, 16]. Different from batch analytics jobs that process the finite amount of data [103], stream queries are long-running jobs that handle infinite data streams.

Recently, as the number of IoT devices and sensors rapidly increases, IoT devices generate diverse IoT data streams such as real-time temperature at home [69], heart rate of patients [36], and real-time location of children [2]. Each IoT device creates several events per second (e.g., current temperature per second), so the volume (size) of individual IoT data streams is small. However, due to the large number of devices and data streams, the aggregated volume is large.

Such characteristics of IoT data streams bring the following two stream workloads, which we call *small* and *big* IoT stream workloads. Section 1.1.1

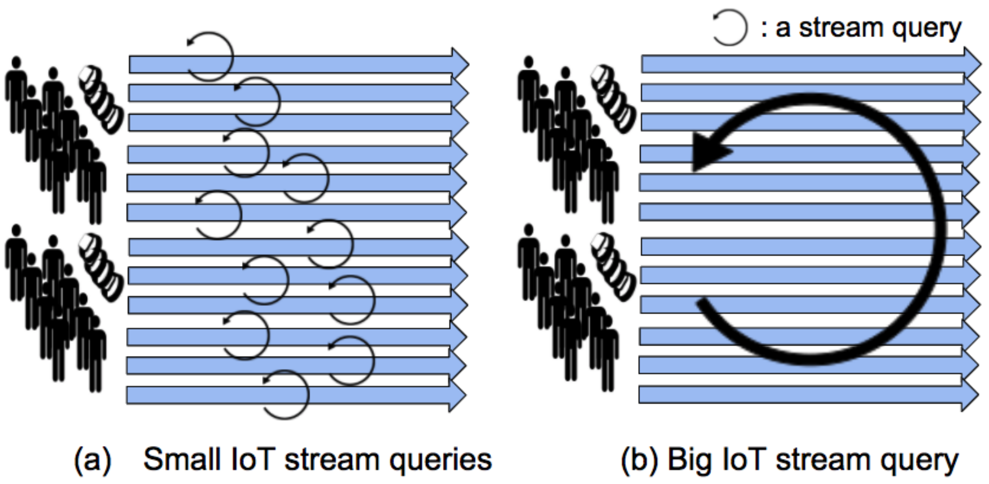

Figure 1.1: The illustration of (a) small and (b) big IoT stream workloads. In small IoT workloads, a large number of *small* IoT stream queries are created and executed on backend servers. In big IoT workloads, a small number of *big* IoT stream queries process a large volume of data streams that can suddenly fluctuate in a short period of time (bursty loads).

and Section 1.1.2 illustrate the IoT stream workloads with Figure 1.1.

1.1.1 Small IoT Stream Query

To provide useful services for end users who use IoT devices, diverse IoT applications create small IoT stream queries on behalf of end users. An IoT stream query processes a *small* amount of IoT data streams related to a certain user, where the small data stream has low event rate (e.g., less than 10 events per second) [69]. For example, when a user clicks a button, a smart home application creates an IoT stream query that continuously controls the user's air conditioner while processing the current home temperature event generated from an IoT sensor every second [82].

In the small IoT stream workload, there are two main participants: IoT application developers and end users. IoT application developers write the application logic of IoT stream queries on behalf of end users and deploy IoT applications to end users. These applications could be mobile or desktop programs, or web services that help users to easily create and configure their IoT stream queries with user-specific parameters [30, 4, 69].

Although such small IoT stream queries can be executed in edge devices, recently, IoT device vendors and application developers have shifted the burden of executing small IoT stream queries from the edge devices to cloud backend servers such as AWS IoT [7] and Azure IoT[9] due to the following benefits. First, IoT device companies can reduce the operational cost and investment of IoT device hardwares [50, 76], delegating the burden of query execution to the backend servers. For instance, according to RFC-7228 [80], constrained IoT devices have small RAM and code size (less than 100 KiB) with low computing power, which is limited to executing stream queries on the devices. Second, deploying the codes to the server enables the agile deployment of software codes. Third, application developers can easily connect multiple data streams together and control remote IoT devices in the cloud server [27]. Fourth, IoT applications can use and join data stored in the cloud.

As billions of IoT devices are being used [37], and as a large number of small IoT stream queries are generated from various applications, the number of small IoT stream queries submitted to the cloud backend servers is also expected to grow significantly. To minimize the maintenance cost and reduce the number of cloud machines for query processing, we focus on handling as many small IoT stream queries as possible on a single node.

Existing distributed stream processing systems [16, 88, 19, 63] fall short of handling a large number of small IoT stream queries in a machine because their system resources easily become a bottleneck due to their inefficient query submission and execution layer. First, their query submission layer is *tightly-coupled* with the code registration. Whenever a query is submitted, its code file is also submitted and uploaded to the systems. The more IoT queries are submitted, the more codes are uploaded, which leads to the bottleneck in uploading a large number of codes. Second, their query execution layer is inefficient in handling many small IoT stream queries because system resources are *separately allocated* for each query. For instance, modern distributed stream processing systems such as Flink [16] and Storm [88] are designed for optimizing individual queries that process data streams with high volumes. For each query, they create separate network connections and threads [88, 53, 16, 104], as well as separately allocating memory for each query code. However, this design causes high query maintenance overheads (e.g., threading overheads) and requires huge system resources when a large number of IoT stream queries are submitted.

1.1.2 Big IoT Stream Query

Different from the small IoT stream workload, in big IoT stream workload (Figure 1.1(b)), the number of queries is small, but a *big* stream query processes a huge volume of aggregated IoT data streams. Data scientists in companies and governments or system operators in datacenters deploy big queries on backend servers and analyze the aggregated IoT data streams to extract useful information or insights. For instance, a big IoT stream query analyzes crowds of people in various locations by processing a large number of IoT data streams generated from user's IoT devices (e.g., smart watches or wrist bands).

The traffic of aggreagted IoT data streams would be *bursty* and unpredictable [100, 62, 44], which means that the input rate and the load of big stream queries can significantly fluctuate in a short period of time. Due to the wide spread of social media and IoT devices, users can send data (e.g., text message) to the cloud at any time through their devices in reaction to unexpected real-world events, such as influencers's tweets, breaking news, or earthquake [81, 77]. As a result, the volume of data and events generated from IoT devices can be bursty and unpredictable [21].

Big IoT stream queries continuously process the real-time data streams to extract insights or make business-critical decisions with *low latency* [99, 88, 98, 75, 16]. In addition, as stream queries are long running, computing resources should be efficiently used for *low maintenance cost* ($). However, existing work does not handle unpredictable bursty loads with low latency and cost ($). First, over-provisioning computing machines causes under-utilization of resources in the load at the bottom and leads to high maintenance cost ($). Second, dynamically (de)allocating cloud virtual machines (VMs) according to the load [18, 39] can reduce the maintenance cost ($) of computing resources, but the *slow* start-up time of VMs that may take several minutes could delay the handling of unexpected bursty loads and increase processing latency. For instance, if the sudden bursty loads happen but creating VMs take one minute, handling the bursty load will be delayed until new VM resources are started. Preparing VMs in advance by predicting the future traffic patterns is not applicable for unpredictable bursty traffic. This paper focuses on designing a fast scaling mechanism of big IoT stream queries when scaling decision is made in reaction to unpredictable bursty loads.

1.2 Proposed Solution

In this dissertation, we propose end-to-end system optimization techniques, not only optimizing the query execution layer, but also optimizing the query submission layer for small IoT-query workloads (Section 1.2.1), and compilation and resource acquisition layer for bursty loads of big IoT stream queries (Section 1.2.2).

1.2.1 IoT-Aware Three-Phase Query Execution

To efficiently execute a large number of small IoT stream queries in a node, we propose IoT-aware three-phase query execution that optimizes both query submission and execution layer. We realize our proposed technique by designing Pluto, a new stream processing system for small IoT stream queries.

The end-to-end query execution of Pluto consists of a code-registration, query submission, and execution phase. Pluto allows application developers to register code *before* query submission by decoupling the tightly-coupled query and code submission. It enables Pluto to keep the registered code information and to be aware of the common application code shared across small IoT queries and to eliminate duplicate code registration overheads from query submission.

In the execution layer, we propose a new IoT-aware query execution model. Pluto shares system resources such as network connections, codes, and threads among small IoT queries to minimize resource bottleneck. Moreover, Pluto performs a locality-aware scheduling that processes the events of small IoT queries in a way that exploits the temporal locality of code references. Pluto enables locality-aware processing by using *Q-group* (IoT query group), a new IoT stream query scheduling unit for efficient load (re)balancing and locality-aware scheduling on multiple cores. A Q-group contains an event queue of multiple IoT stream queries that refer to the same application code, and Pluto dynamically (re)balances the load of Q-group assigning each of them to a thread for high throughput and low latency query execution.

Our evaluations on a 24-core machine with various IoT applications show that Pluto improves the number of small IoT stream queries that can be handled in a machine by an order of magnitude compared to existing distributed stream processing systems (Storm and Flink) and a stream database (PipelineDB), while keeping 99th-percentile latency below one second.

1.2.2 Streaming Dataflow Reshaping on Lambda

To quickly handle sudden bursty loads of big IoT stream queries, we propose Sponge, which is a new stream system that optimizes query compiler, execution, and resource acquisition layer for fast scaling of big IoT stream queries.

First, Sponge dynamically creates Lambda instances instead of VMs when bursty loads happen because Lambda offers lightweight containers faster to start than VMs [6, 8, 38, 42]. The start-up time of Lambda instances is less than a few seconds, which is at least 10× faster than that of VMs [34], as cloud vendors create Lambda containers on already running VMs that are pre-allocated for Lambda services.

However, realizing fast scaling of big stream queries on Lambda is challenging because of Lambda's data communication constraint (it does not allow direct data communication across Lambda instances) and the slow scaling mechanism of existing work [40, 94].

To address this problem, Sponge also optimizes the compiler layer by reshaping the logical dataflow of streaming queries to a fast-scaling-friendly dataflow on Lambda. Once a big stream query is submitted as a dataflow graph, Sponge inserts three utility stream operators in the logical graph for dataflow reshaping: router operators (ROs), transient operators (TOs), and state mergers operators (MOs), according to dataflow patterns and query semantics. Inserting ROs enable Sponge to easily distribute the load of operators with shuffle data communications to multiple Lambda instances. Inserting TOs and MOs reduces the delay of the slow scaling mechanism and allows Sponge to quickly redistribute the load of operators to Lambda.

We evaluate Sponge on EC2 instances (5× r5.xlarge) and AWS Lambda instances (up to 200 Lambda instances of $1,769$ MB memory) and show that reshaping the dataflow graph of stream queries can significantly reduce tail latencies when bursty loads happen compared to scaling queries without reshaping on VMs and Lambda instances.

1.3 Contribution

In this dissertation, we make the following contributions:

- We provide the insight and the workload characteristics of two different (small and big) IoT workloads and identify their problem.

- We propose end-to-end system optimization techniques to solve the problem of IoT stream workloads. Specifically, we propose IoT-characteristic-aware optimization that optimizes query submission and execution layer for handling a large number of small IoT stream queries. In addition, we propose optimizing query compilation, execution, and resource acquisition layer by reshaping streaming dataflow on Lambda to enable fast scaling mechanism and reduce latency spikes when bursty loads happen in big IoT stream queries.

- We design and implement complete and working systems that embody our idea and show that our proposed techniques significantly improve the performance in terms of throughput and latency.

1.4 Dissertation Structure

The rest of this thesis is organized as follows. Chapter 2 describes the characteristics of IoT stream queries and workloads. In Chapter 3, we present IoT-aware three-phase execution, which optimizes both query submission and execution layer, to efficiently handle small IoT stream queries in a node. Chapter 4 proposes a streaming dataflow reshaping approach on Lambda, which optimizes query compilation, execution, and resource acquisition layer, to handle sudden bursty loads of big IoT stream queries cost efficiently. In Chapter 5, we conclude the work.

Chapter 2

Background

In this chapter, we illustrate the stream query model and characteristics of small and big IoT stream queries through an example for better understanding of this dissertation.

2.1 Stream Query Model

A small and big IoT stream query (q) is represented as a directed acyclic graph (DAG), which describes stream queries that many stream processing systems adopt [88, 16, 104]. In a DAG, a vertex is either a source (s), which subscribes to IoT data streams, an operator (o), which processes the data with user-defined functions (UDFs), or a sink (k), which publishes the processed data. An edge represents the stream of data flowing from the upstream vertex to the downstream vertex (e.g., $s \rightarrow o$).

Small Stream Query. Figure 2.1 shows the DAG example of small and big IoT stream queries. In Figure 2.1(a), there are two small IoT stream queries

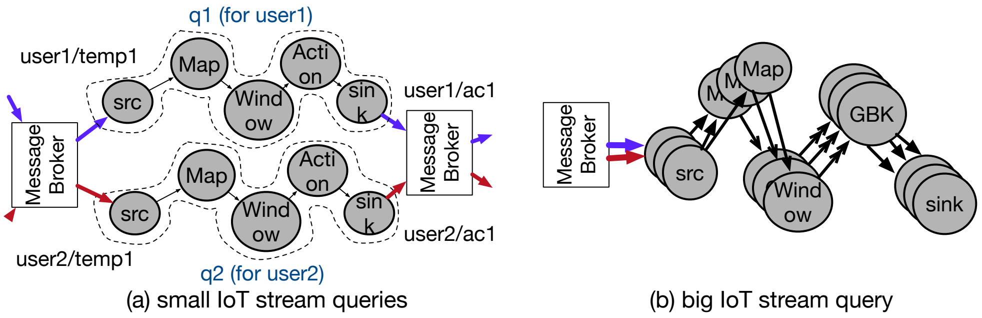

Figure 2.1: The dataflow (DAG) example of (a) small and (b) big IoT stream queries. A small IoT stream query has a single instance of an operator, but a big IoT stream query has multiple instances of an operator to parallelize the query processing due to the large volume of data.

that continuously control air conditioners for two different users generated from a smart-home IoT application. Each stream query (`q1` or `q2`) receives a temperature data stream related to a user (`user1/temp1` or `user2/temp1`) from different devices, The `Map` operator pre-processes the raw temperature data to change the data format, the `Window` operator gathers recent temperature data in a sliding window, and the `Action` operator decides the necessary action to control the target cooling temperature of the air conditioner according to the processing logic of the registered code. The sink then emits the decision to the message broker to change the cooling temperature of the air conditioner. Finally, the air conditioner (`user1/ac1` or `user2/ac1`) receives the decision event from the message broker and adjusts its cooling temperature. Each user can customize her query by setting the IoT temperature sensor, the air conditioner, and the target cooling temperature of the air conditioner via the web or smartphone IoT application [82].

Big Stream Query. Figure 2.1(b) shows a big IoT stream query that counts the number of people in locations. The query first receives the aggregated IoT data streams from the message broker, converts each event into a pair of key (location) and value (count) in the `Map` operator, assigns windows to the pairs and collects events within the windows (`Window` operator), and aggregates the windowed data to count the number of people in each location (`GBK`). As an example, the `GBK` operator receives events, consisting of the triple of key (location), value (1), and assigned window ([window-start-time, window-end-time)). The `GBK` operator then aggregates events with the same location and the same window to calculate the count of active users in each location.

2.2 Workload Characteristics

In this section, we highlight the workload characteristics of small and big IoT stream queries.

2.2.1 Small IoT Stream Query

Small Stream Size. A small IoT stream query processes a small amount of data streams because it receives data from a few IoT devices or sensors related to a certain user. The small data stream has a low event rate, as each IoT device usually samples metrics and generates an event periodically every second or minute. For instance, smart sensors generate data every one second [102, 69]. In this dissertation, we focus on the text-based or event-based IoT data streams (not images, video, or voice data) with low input rate. Due to the low event rate, there is little benefit of data parallelization for small IoT stream queries. Therefore, instead of distributing an IoT stream query across nodes, executing many IoT stream queries on a single node is important in reducing the required number of nodes and the server maintenance cost, which is the main topic of Chapter 3.

Large Numbers of Codes and Queries. In small IoT stream query workloads, various IoT application codes can be registered to the cloud backend server due to the diverse needs for different end users. For instance, in the workloads of IFTTT [43] that is a platform for automating IoT devices, the number of IoT applications (called applets) is more than 75 millions, which represents that diverse IoT applications are registered for end users.

The more applications are developed and more IoT devices are installed, the more small IoT stream queries will be generated for many end users. As an example, $75K$ stream queries can be submitted per second in the IFTTT workload even if 0.1% of IoT applications are triggered by end users. In addition, location-based IoT applications offer new services for end users such as taxi hailing [90] with GPS-equipped IoT devices. According to the workload of taxi-hailing service in China, more than a thousand taxi-hailing requests per second are created by end users [58] during the peak load. These workloads indicate that a large number of small IoT stream queries can be submitted by end users, so it is important to process and handle as many stream queries as possible on a node to minimize the maintenance cost of backend servers.

2.2.2 Big IoT Stream Query

Large Stream Size and Bursty Loads. In contrast to small IoT queries, big IoT stream processing queries process a large volume of real-time data and extract insights with low latency [99, 88, 98, 75, 16]. The size of data streams that distributed stream processing [88, 16] targets is usually large, like $1M$ events per second as shown in the recent distributed stream benchmark [49].

Due to the aggregated data streams, the input rate and the load of the jobs fluctuate over time according to the real-time workload. Furthermore, at times, the input rate can increase sharply.

The number of big IoT stream queries is relatively smaller than the small IoT stream query workloads, so the important topic is to optimize individual big stream queries and to handle the bursty loads of a big stream query with low latency results, which is the main topic of Chapter 4.

Parallelism (Multiple Tasks). Each stream operator processes a large volume of data streams. Therefore, a big stream query parallelizes the processing of data streams by creating multiple instances of operators, which we call *tasks*. A task is a scheduling and execution unit that handles a part of big data streams.

N-to-N Communication (Shuffle). A task can send data to and communicate with multiple tasks of its downstream operator, which leads to N-to-N communication (shuffle). Data with specific keys can be sent to specific tasks in the shuffle communication. For instance, in Figure 2.1(b), tasks of the `Window` operator shuffle data based on the location and perform N-to-N communication with the tasks of the `GBK` operator.

Large State Size. Due to the large volume of data, the size of states of big IoT stream queries can be large. In the `GBK` operator in Figure 2.1(b), it aggregates data by keys. When the input rate increases, the number of keys (number of locations of users) and the size of aggregated values can also increase (e.g., when the operator appends values to calculate quantile), which results in the large state size of big IoT stream queries.

Chapter 3

IoT-Aware Three-Phase Query Execution

In this chapter, we explain how to efficiently process a large number of small IoT stream queries in a machine with IoT-characteristic-aware optimization.

Although a large number of small IoT stream queries are created from various applications, there are two commonalities among small IoT queries that we can exploit when they are executed on the cloud backend server. First, we have observed that many small IoT stream queries are created from the same IoT application and reference the common application code during the execution in the server, because multiple end-users use the same IoT application to create their IoT queries [82]. For instance, according to the IFTTT workload, one application is installed by $97K$ end users [10], which means that more than $90K$ IoT queries will reference the same code. Second, even if IoT stream queries process different IoT streams for different end-users' IoT devices, many IoT stream queries receive data streams from a common message broker, each of which handles hundreds of thousands of IoT connections [26] due to the lightweight IoT

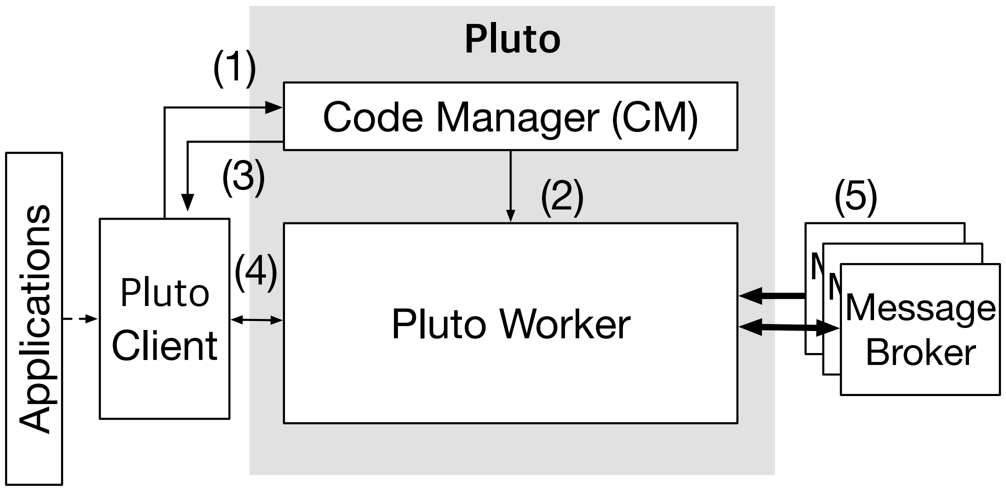

Figure 3.1: A system overview of Pluto. The gray-box is a single node.

messaging protocols (e.g., MQTT [67]). For example, EMQ [26], the message broker for MQTT connection, can support $1M$ MQTT connections and data streams.

3.1 Pluto Design Overview

Pluto is a new stream processing system that optimizes the end-to-end execution of large numbers of IoT stream queries by being aware of the IoT-query characteristics. Pluto addresses the limitations of existing stream systems by applying the following key design principles:

1. **Decoupling of Code and Query Submission**: Pluto decouples code registration from the query submission phase and performs a *three-phase* end-to-end query execution with new APIs. The decoupling of the code and query submission enables Pluto to optimize each phase individually with lightweight query submission, as well as code/network connection sharing across IoT queries.

2. **IoT-Aware Query Execution for Low-Latency and High-Throughput**: Pluto executes a large number of IoT stream queries (high throughput) by sharing system resources as much as possible on a node, while keeping sub-second latency results and evenly (re)balancing the load of IoT stream queries across multiple cores in a machine. In doing so, Pluto leverages commonalities among IoT stream queries and uses a new scheduling unit for groups of IoT stream queries, called *Q-group*. Pluto assigns the multiple IoT queries that refer to the same application code to a Q-group and schedules the group of queries for locality-aware scheduling that maximizes CPU code cache localities, while dynamically dividing the Q-group for load rebalancing across multiple cores.

For three-phase execution, Pluto is designed with three components: the client, code manager (CM), and worker, as shown in Figure 3.1.

To execute IoT stream queries, first, developers should implement applications and register their codes. Pluto client provides programming APIs to support various user-defined functions (UDFs), such as map, filter, window, flatMap, union, join, and complex event processing (CEP) [97]. Once the UDF is developed, the developer registers the application UDF code using the code registration API (Section 3.2.1). The client then sends the code file from the local code file path to the CM through the network.

The CM manages the registered code: it checks whether the code is already submitted or not. If it is a newly submitted code, the CM stores the code into the Pluto worker (Figure 3.1(2)) and returns a unique code identifier of the registered code to the developer (Figure 3.1(3)). Otherwise, it returns the existing code identifier for the duplicate code. To efficiently check for duplicate code during the registration, the CM compares the hash value of the bytes of codes. In the future, we plan to check malicious codes with static code analysis techniques [13, 20] to reject them for security.

Once the code is registered, the application developer should deploy the code identifier to end-user devices to submit IoT queries with the code identifier. The code identifier plays an important role for grouping and sharing codes across IoT queries in Pluto's worker, and the detail is described in Section 3.3.

For the query submission, the Pluto client provides a dataflow query submission API that converts a query as a directed acyclic graph. For each query, Pluto client explicitly receives the message broker address of the source of a query, the code identifier of the application code that the query references, and the values of user-specific parameters to customize the query logic for each end user (Section 3.2.2).

The query submission is lightweight, as Pluto decouples the code and query submission path. When submitting queries, Pluto client submits only the code identifier instead of the actual code bytes with the dataflow graph and user-specific parameters of the query. If there is a single path for the code and query submission without code registration, duplicate codes will be sent and uploaded to the server multiple times, which is the limitation of existing stream processing systems.

A worker, which is a long-running process that runs on a single node, automatically and quickly instantiates UDFs from the registered code for the submitted IoT queries (Figure 3.1(4)) and connects with the message broker (Figure 3.1(5)). To efficiently process many IoT stream queries, the worker shares system resources and performs locality-aware scheduling by being aware of the broker address and code identifier.

In the following sections, we explain the details of code registration and query submission API (Section 3.2), and the IoT-aware query execution model on a worker (Section 3.3).

Listing 3.1: Parameter Declaration API

```
1 @ParameterDeclare
2 public class Threshold implements ParameterDeclare<Double> {}
```

3.2 Decoupling of Code and Query Submission

3.2.1 Code Registration

At a high level, in the code registration phase, Pluto offers APIs for building and submitting UDFs and annotating UDF parameters, by which the actual user-specific values for customized queries are set in the query submission phase. Specifically, the Pluto client provides the following APIs: i) parameter declaration, ii) UDFs with parameter binding, and iii) code registration. Here, we explain the client APIs with a simple example written in Java.

Parameter Declaration (Listing 3.1). Application developers should first declare classes for parameters used in their UDFs, using `@ParameterDeclare` annotation (line 1 in Listing 3.1) and creating a class for the parameter (line 2). Here, as an example, a `Threshold` parameter with `Double` type is declared. The actual value of the parameter will be set in the query submission phase.

UDFs with Parameter Binding (Listing 3.2) Next, application developers should create their UDFs and bind the declared parameters to the UDFs. We show a simple `MyFilter` UDF example that filters a temperature value if the value is less than a certain threshold parameter, configured by an end-user in query submission.

To bind the threshold parameter to `MyFilter` UDF, `@ParameterBinding` annotation should be used, wrapping the declared parameter class name `Threshold.class` (line 4). In addition, `@Inject` annotation should be added above

Listing 3.2: Parameter Binding API

```
1  class MyFilter implements PlutoFilter<Double> {
2    private double threshold;
3    @Inject
4    MyFilter(@ParameterBinding(Threshold.class) double threshold) {
5        this.threshold = threshold;
6    }
7    @Override
8    public boolean filter (double temperature) {
9      return temperature < threshold;
10   }
11 }
```

the class constructor (line 3). With the `@Inject` and `@ParameterBinding` annotations, the Pluto worker automatically injects the actual parameter value of threshold and instantiates `MyFilter` whenever a query is submitted with the actual parameter values.

Code Registration. After writing the UDFs, application developers compile and submit the codes through Pluto's command-line API. The procedure returns a unique code identifier of the submitted jar file, as explained in Section 3.1.

3.2.2 Query Submission API

Once the code is registered in the code registration phase, application developers deploy the code identifier to end-user devices. IoT queries are submitted with the code identifier and query DAG from the device, setting the UDFs and the actual user-specific values of the annotated parameters. . As an example, Listing 3.3: Query Submission API

```
1 parameters = PlutoClient.parameterBuilder()
2             // value defined by an end−user
3             .setParameter(Threshold.class, value)
4             .build();
5 PlutoQueryBuilder.mqtt(brokerAddress)
6            . filter (MyFilter.class)
7            ...   // dataflow API
8            .submit(codeIdentifier, parameters)
```

for each IoT application, Listing 3.3 code is executed to submit a query for an end user.

In Listing 3.3, a query is written with Pluto's dataflow API. In lines 1—3, the value of the annotated `Threshold` parameter is set, which is defined and registered in the code registration phase. In lines 4—7, the query DAG is built, with the `brokerAddress` (line 4) and UDF class name used in the query (line 5). The query is then submitted with the `codeIdentifier` and parameters in line 7, and transformed into a DAG. As Pluto sets only the metadata of the registered code, as well as the actual value for binded parameters, the query submission is lightweight.

3.3 IoT-Aware Execution Model

In the worker, to minimize resource bottlenecks and improve the system performance while processing many IoT stream queries, Pluto exploits the commonalities among IoT stream queries. Figure 3.2 illustrates the key difference of execution models between the existing stream processing systems and Pluto.

First, compared to the existing stream processing systems that allocate sys-

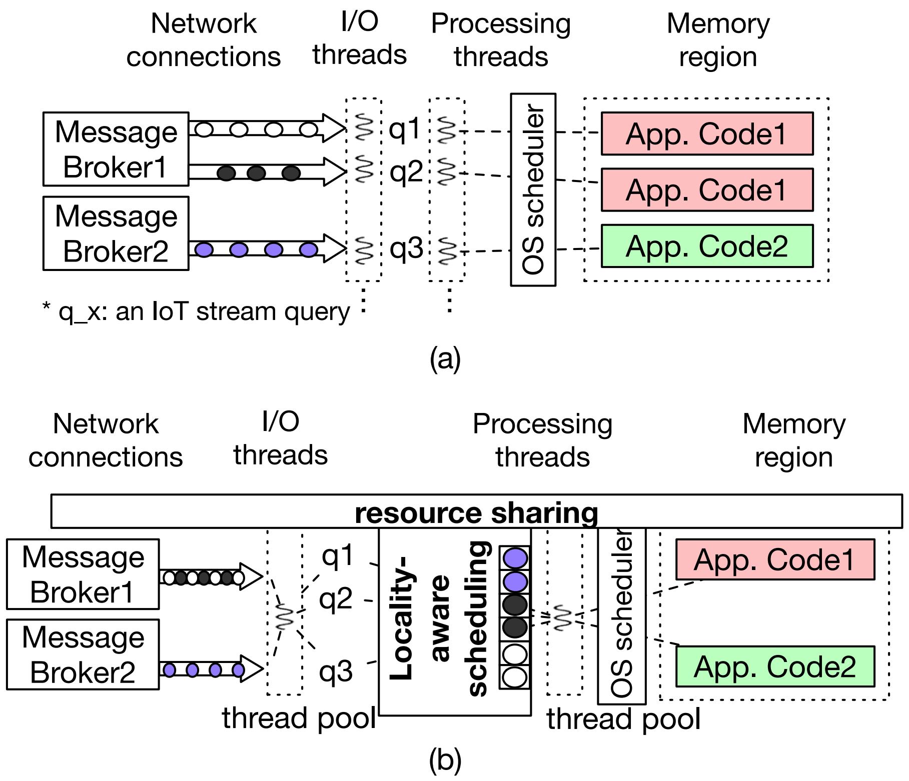

Figure 3.2: (a) The execution model of existing stream processing systems that creates separate system resources (network connection, thread, codes) for individual queries. (b) The overview of Pluto's IoT-aware execution model.

tem resources separately for each query (Figure 3.2(a)), Pluto shares network connections, I/O and processing threads using the thread pools [35, 19], and memory region for the same application code among IoT stream queries (Figure 3.2(b)). Each I/O and processing thread pool has a fixed number of threads (2× of cores by default, and it can be configured by system administrators), and threads are uniformly pinned to cores as illustrated in Figure 3.3. Separating the threads for I/O operations (sources and sinks) and CPU operations (operators) and pinning threads to cores prevent frequent context switching and CPU

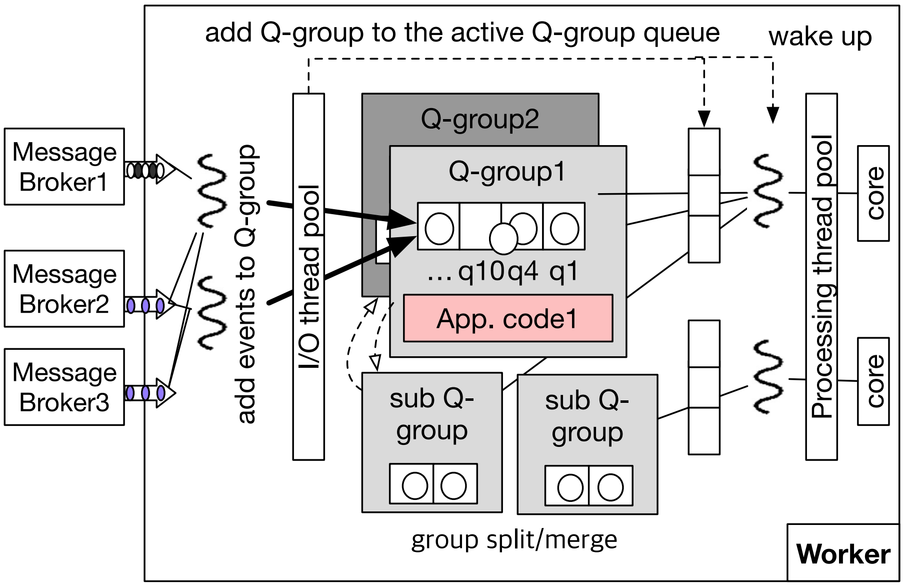

Figure 3.3: The detail of Pluto execution model in a worker.

cache misses [95]. To share network connections among queries, Pluto checks whether an open network connection exists for `brokerAddress` of the query. If the connection exists, the worker retrieves the source event from the open connection. Otherwise, the worker opens a new network connection and assigns it to an I/O thread.

Second, to further minimize the CPU bottleneck, Pluto adopts locality-aware scheduling that schedules the events of IoT queries in a way that exploits the temporal locality of code references. For instance, in Figure 3.2(b), as $q1$ and $q2$ queries look up the same code memory region (App. code1), Pluto consecutively processes the events of $q1$ and $q2$ on the same core to exploit the temporal locality of code references and minimize CPU cache misses.

Pluto uses a new scheduling unit, called Q-group, which groups stream queries for locality-aware processing. In the following section, we illustrate how Pluto creates Q-groups (Section 3.3.1), assigns Q-groups to threads (Section 3.3.2), processes the events within Q-groups (Section 3.3.3) for locality-aware processing, and dynamically rebalances the load of Q-groups across threads for low latency results (Section 3.3.4).

3.3.1 Q-Group Creation and Query Grouping

To share codes across IoT stream queries that refer to the same application codes, Pluto creates a *Q-group* for each unique `codeIdentifier`. A Q-group contains IoT stream queries referring to the code `codeIdentifier`. We define a Q-group $G = \{ref(C), \mathbb{Q}\}$ where $ref(C)$ is a reference of code C, and $\mathbb{Q}$ is a set of queries that access code C when they are executed. A Q-group is created whenever a new application code is registered (Figure 3.1(2)). When a query is submitted to the worker, the worker looks up the Q-group with the `codeIdentifier` of the query, and assigns the query to the Q-group for query grouping.

3.3.2 Q-Group Assignment

During the execution, many Q-groups can be created as various application codes are registered by diverse application developers. If the number of Q-groups is larger than the number of processing threads, multiple Q-groups are assigned to a processing thread. Pluto properly distributes the load of Q-groups among multiple processing threads by assigning the Q-groups with a variant of the least-load-based balancing mechanism [17]. To assign a Q-group, Pluto selects k smallest-load processing threads, and randomly selects one of them to prevent many Q-groups from being assigned to a thread in a short period of time. k is 2 by default, based on the power of two choices [64].

Pluto measures the load of a processing thread following the $M/M/1$ queueing model [24] as the event arrival of IoT stream is usually approximated with the Poisson distribution [61]. Pluto estimates the load of each thread by summing up the load of assigned queries in the Q-group. The load of each query is calculated by dividing the mean event arrival rate by the mean event processing rate.

3.3.3 Q-Group Scheduling and Processing

The processing thread should process events within Q-groups to exploit the temporal locality of code references, while minimizing processing latency for low latency results. To achieve both goals, Pluto adopts the event driven architecture [95], where processing threads are activated whenever an event is received.

To realize this architecture, each processing thread P maintains a two-level queue structure. The first-level queue, called active group queue, contains active Q-groups, $\mathbb{G}^a$=$\{G^a_1$,G^a_2,...,$G^a_k\}$ ($k \in \mathbb{Z}_{>0}$), where G^a_k is one of the assigned Q-groups to P that holds at least one event. Each G^a_k has a second-level queue, event queue, that contains the events of the assigned queries. If there are no active Q-groups in a processing thread, the processing thread sleeps, and waits until active Q-groups exist. This reactive approach enables processing new events with low latency without wasting CPU cycles for checking for Q-groups that have no events.

Algorithm 1 presents the algorithm of how Pluto schedules and processes events in detail. Lines 1–8 show the event scheduling. The I/O thread retrieves query events from network connections and adds the events to the Q-group event queue where queries are assigned. In addition, an I/O thread schedules a Q-group to the active Q-group queue of each processing thread and notify the processing thread when there are events in the Q-group. When a source I/O thread receives an event e of query q from the network connection, the `Event-`

Algorithm 1: Event Scheduling and Processing

```
   // Source I/O thread
1  Function EventScheduling(e, q)
2      // $e$: an event of query $q$;
3      $G \leftarrow$ queryGroupTable($q$) ;
4      add $e$ to the event queue of $G$;
5      if $G$ becomes active then
6          $P \leftarrow$ groupProcessingThreadTable($G$);
7          add $G$ to the active group queue of $P$;
8          awake($P$);

   // Processing thread $P$
9  Function EventProcessing(P)
10     $Q_P \leftarrow$ activeGroupQueue of $P$;
11     while $Q_P$ is not empty do
12         startTime $\leftarrow$ currentTime();
13         $G^a \leftarrow$ poll an active group from $Q_P$;
14         while until event queue of $G^a$ is empty do
15             if elapsedTime(startTime) $<$ timeout then
16                 $(e, q) \leftarrow$ poll an event from the event queue;
17                 processEvent($e$, $q$);
18             else
19                 // preemption and rescheduling ;
20                 add $G^a$ to $Q_P$;
21                 break;
22         if $Q_P$ is Empty then
23             wait();
```

`Scheduling` function is invoked. The thread first finds the Q-group G where q is assigned (line 3). If the G becomes active at this time, which means that G has one event to process at this time from its empty event queue, the I/O thread adds G to the active Q-group queue of P, where G is assigned (line 7). After adding G to the active Q-group queue of P, the I/O thread sends a signal to P where an active Q-group is added.

Lines 9–23 of Algorithm 1 illustrate how a processing thread processes the Q-group and events in the two-level queue structure. The `EventProcessing` function is invoked only once for each processing thread P when the Pluto worker starts. In line 23, P sleeps if an active Q-group queue is empty and wakes up when an I/O thread adds an active Q-group to the queue and sends a signal.

By processing all the events in the same Q-group consecutively, the Pluto worker utilizes the code locality. P processes the events in the event queue of each Q-group one after another consecutively (line 16) to exploit the temporal locality of code references. In doing so, P accesses the same code repeatedly, which reduces cache misses and CPU use.

To prevent a processing thread P from being occupied for a long time by an active Q-group (G^a) with a lot of events to be processed, Pluto preempts the active Q-group with a timeout value (line 15) to fairly process events in different Q-groups. When G^a is preempted, P adds the Q-group into the end of the active Q-group queue because G^a has remaining events (line 20). By rescheduling G^a to the end of the queue, Pluto provides other groups of stream queries with a fair chance to be processed by P. When the load of a certain Q-group is too large, Pluto eventually splits the large Q-group and distributes the load to other processing threads, which is illustrated in the following section.

3.3.4 Load Rebalancing: Q-Group Split and Merging

The load of Q-groups may change over time and threads could be overloaded while processing events. Pluto dynamically splits the Q-group into multiple sub Q-groups, and merges the sub Q-groups again for load rebalancing among multiple processing threads (and cores). Each sub Q-group inherits the parent Q-group's code reference, and has the subset of queries and events of the parent Q-group.

For load rebalancing, Pluto periodically checks the load of each processing thread, and detects overloaded and underloaded threads. Pluto then splits Q-groups assigned to the overloaded threads and moves the sub-groups to underloaded threads, or just reassigns the Q-groups from the overloaded to underloaded threads. In doing so, two threshold parameters are used to find the overloaded and underloaded threads, α and β ($\alpha < \beta$). We consider a thread T as overloaded or underloaded, if the load of T (ρ_T) is greater than β or lower than α. Threads with loads between α and β are considered stable, and they do not participate in the load rebalancing process to prevent the oscillation between the overloaded and the underloaded state. Through our evaluation, we empirically found that the rebalancing process works well with $\alpha = 0.8$ and $\beta = 0.95$. If $\beta > 0.95$ or $\alpha > 0.8$, the tail latency increases significantly because the threshold is too high to trigger rebalancing. In contrast, when we set $\beta < 0.95$ or $\alpha < 0.8$, the rebalancing occurs frequently, which increases the rebalancing overhead of Q-group split/merge or reassignment.

We iterate the following process for each overloaded thread T_o for load rebalancing: we select the Q-group G_l with the largest load among Q-groups assigned to T_o. We then try to move G_l to an underloaded thread T_u, if (1) $\rho_{T_u} + \rho_{G_l} < \beta$, and update the load of T_o and T_u. If the updated ρ_{T_u} is between

α and β (T_u becomes stable), we select another underloaded thread. We repeat this process until the load of T_o is less than β (not overloaded).

When the load of G_l is too large, the condition (1) may not be satisfied for all underloaded threads, while ρ_{T_o} is still larger than β (overloaded). In this situation, we split the Q-group G_l and create two sub Q-groups, G_{l1}, G_{l2}. The events in the event queue of G_l are evenly distributed to G_{l1} and G_{l2}, and the assigned queries to G_l are also evenly reassigned to G_{l1} and G_{l2}. Pluto moves one of the sub Q-groups to underloaded threads with condition (1) until $\rho_{T_o} < \beta$. If the sub Q-group is too large to be moved, Pluto splits the sub Q-group again recursively like a binary tree structure, where the root is G_l. Pluto also merges split sub Q-groups into one again to maximize the locality-aware processing, when threads are underloaded again.

3.4 Implementation

We have implemented Pluto with approximately $17K$ lines of code (excluding test code) in Java 1.8. For parameter declaration and binding in Pluto's API, we use Tang [85], a dependency injection framework. For communication with message brokers, we use EMQ [26] that supports thousands of connections in the broker.

In a worker, multiple I/O threads and a processing thread P can concurrently access the active Q-group queue of P. To minimize the thread contention, we use lock-free atomic counters. Using the counters minimizes the number of access to the active Q-group queue and the overhead of thread contention between I/O threads and processing threads. Each Q-group G maintains a counter that indicates the number of events in G. An I/O thread increments the counter and P decrements it concurrently. When the counter becomes one from zero in a Q-group, the I/O thread adds the Q-group to the active Q-group queue of P.

3.5 Evaluation

We answer the following questions in our evaluation:

i) How does Pluto improve the performance compared to existing distributed stream processing and stream database systems (Section 3.5.2?

ii) Which factors contribute to the performance of Pluto (Section 3.5.3)?

iii) Does Pluto well-balance the load among cores in a skewed workload while keeping low latency (Section 3.5.4)?

iv) Is Pluto still effective in large stream sizes (Section 3.5.5)?

Environment. For evaluation on a single machine, we run Pluto and other systems on a 24-core NUMA machine (2× Intel Xeon E5-2680 2.5GHz, 30M Cache, 8× 16GB RDIMM, Ubuntu 14.04.5, Kernel 4.4.0-124-generic). In this evaluation, the number of I/O and processing threads is set to 48, respectively (2× cores). All the events generated from the data streams and query results are delivered via MQTT [67], and EMQ [26] is used as the message broker server. We use two additional machines, one for the event stream generating and the other for the message brokering.

3.5.1 Methodology

For evaluation, we emulate many data streams, applications, and IoT stream queries with real-world datasets.

Workload

To emulate a large number of IoT data streams, we use 9 types of data ($\texttt{stype}_i$, $1 \leq i \leq 9$) by extracting the necessary data from real-world datasets [65, 73, 78, 66, 109, 79, 91]: temperature and humidity [78], heart rate and motion [79], user GPS [109], taxi GPS [66], livestock GPS [91], soil moisture [65], and household power consumption [73]. We then generate a new data stream by uniformly choosing the one of $\texttt{stype}_i$ for each query, and each data stream is generated following the Poisson process model. The input rate is one event per second on average.

To emulate the real-world scenario where a various number of codes is registered by application developers, we implemented 15 base application codes ($\texttt{C}_i$, $1 \leq i \leq 15$) referring to real-world IoT applications such as smart home and health care, and register thousands of application codes to Pluto from the 15 base codes (6000 codes in our evaluation). We copy each code $\texttt{C}_i$ n times, creating codes $C_{i,1}$, ..., $C_{i,n}$. Pluto then treats $C_{i,x}$ and $C_{i,y}$ ($x \neq y$) as different code for our evaluation to emulate real-world scenario: the code manager creates a unique code identifier for each $C_{i,x}$, and the worker loads each registered code in different memory locations by creating a separate Java Classloader. The full list of applications is described in Section 3.5.1.

To create many IoT stream queries, we iterate the following step. First, we select a code C among the registered codes. Second, we create a new data stream (`stream`) from one of the 9 types of data. Third, we create a stream query that subscribes to a `stream` that refers to code C. We uniformly choose a code C among registered codes for each stream query by default. We further show the performance when the created queries are skewed to certain application codes in Section 3.5.4.

Metrics

We measure the throughput (the number of processed queries) and the end-to-end latency to evaluate the performance of Pluto. The throughput is the number of processed events per second in Pluto, and the latency shows how responsive Pluto is in processing the incoming events. As each query processes one event per second on average of the IoT data stream in our evaluation, if the throughput is K, the number of processed queries is also K.

We find the maximum throughput following these steps: First, we submit X stream queries for each step. Second, we begin to generate the data streams for the submitted queries. Third, we measure the throughput and the latency. We repeat these steps until we find the point at which the median latency rises above 100 ms (sub-second latency) and the $P99$ latency rises above one second as many stream queries used in our evaluation require sub second latency for notification and anomaly detection.

Applications

This section describes 15 base IoT applications that we implement for our evaluation based on real-world scenarios. In the following list, (a) is the application name, (b) is the description, and (c) is the type of dataset used in the application. We create a large number of diverse small stream queries by changing the *parameters*.

1. (a) A/C control || (b) Controls air conditioners by analyzing the temperature data within a *window* generated from remote IoT sensors at home. Users can set the *target temperature*. || (c) Temperature [78]

2. (a) Abnormal heart rate (b) Detects abnormal heart rates from patient's heartbeat data streams generated from heart rate sensors within a *window*.

If more than 10% of measured heart rates inside a window exceed a *threshold*, it sends a notification to doctors. (c) Heart rate [79]

3. (a) Remote child care || (b) Notifies the parents when their children are far away from *a list of safe places* for a certain *distance*. || (c) User GPS [109]

4. (a) Elderly care || (b) Detects the inactive motions of elderly people by analyzing the movement data streams within *a certain window* of time and notifies the detection to their caregivers. || (c) Motion [79]

5. (a) Lightbulb control || (b) Turns off light bulbs at home when users are far away by a certain *distance* from their homes, or turns on the light bulbs when users are inside or near home. || (c) Taxi GPS [66]

6. (a) Child mood monitoring || (b) Continuously monitors the mood of a baby from the heart rate and notifies the parents if the child is in a bad mood. We determine the baby is in a bad mood if the average heart rate is constantly high (included in a high heart rate range) *in a window*. || (c) Heart rate [79]

7. (a) Medication management || (b) Detects whether the medication depository temperature is within the *acceptable range* or not in a *window*. || (c) Temperature [78]

8. (a) Fire alarm || (b) Continuously monitors the temperature and detect a fire at home if the temperature increases rapidly and constantly with a small deviation in a *window*. || (c) Temperature [78]

9. (a) Soil moisture monitoring || (b) Monitors the soil moisture in *a window* and informs the farmer when the moisture is not in *a normal range*. || (c) Soil moisture [65]

10. (a) Moisture removal || (b) Monitors humidity data stream in warehouses in *a window* and automatically turns on the dehydrator when the humidity is higher than the *humidity limit*. || (c) Humidity [78]

11. (a) Green house || (b) Automatically detects the temperature anomaly inside the green house if the current temperature is deviated from the previous temperatures in a *window*, and notifies the anomaly to the green house owner. || (c) Temperature [78]

12. (a) Find gas station || (b) Automatically informs the near of gas stations that offer the *price range* set by users. || (c) Taxi GPS [66]

13. (a) Find Parking lots || (b) Automatically informs the nearby parking lots when the car slows down or stops. || (c) Taxi GPS [66]

14. (a) Electricity warning || (b) Monitors home electric usage traffic in a *window* and if the total usage inside the window is bigger than the *designated limit*, warns it to users. || (c) Power consumption [73]

15. (a) Animal care || (b) Notifies the sheepdog owner when their sheepdogs are far away by a certain *distance* from their livestock. || (c) Livestock GPS [91]

3.5.2 Performance Comparison

Figure 3.4 shows the performance of Pluto and various stream processing systems. We evaluate Storm and Flink for distributed stream processing systems, and PipelineDB [71], which is a stream database based on PostgreSQL [72] that supports continuous SQL queries.

Overall, Pluto outperforms other stream processing systems because of its IoT-characteristic-aware optimization.

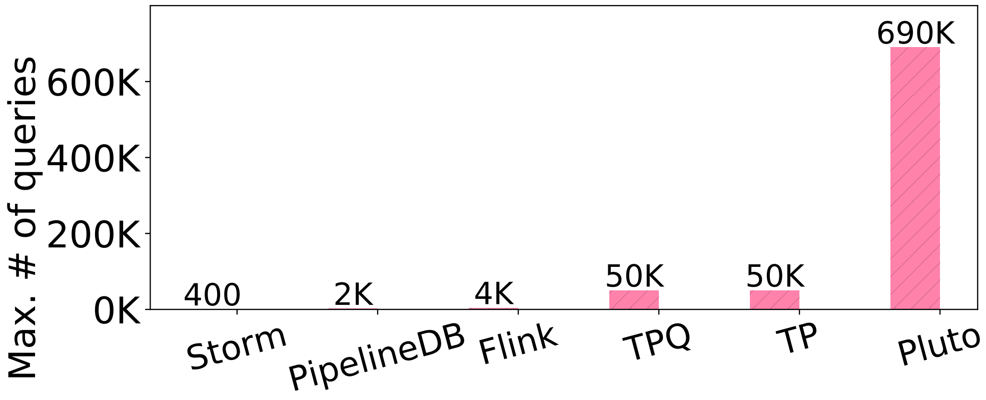

Figure 3.4: The performance of Pluto and other systems.

Storm. First, we evaluate Storm [88], a big stream query processing system. However, it executes up to a few hundred IoT stream queries (400) on the machine because it creates a JVM process for each stream query, which causes a memory bottleneck. Like Storm, we observed that Spark Streaming [104] also does not handle many stream queries because it creates multiple processes for each query.

Flink. We also evaluate Flink [16], which uses system resources more efficiently than Storm. Flink shares a process for multiple queries, while creating separate threads, network connections, and codes for each query in the shared process. Even though Flink creates a single process for multiple queries, we observed that Flink cannot handle stream queries over $4K$ on the machine due to the bottleneck in query submission. It has huge query submission overheads such as uploading and downloading code files (jar files) for each stream query because the query submission and code registration is tightly coupled.

PipelineDB. We evaluate PipelineDB [71] that supports continuous queries. However, we found that it cannot handle more than $2K$ stream queries in our evaluation environment because a large number of separate network connections and threads are created in the execution of queries to retrieve different data streams.

TPQ. As we cannot evaluate all of the modern stream processing systems, we implemented a **TPQ** (threads per query) model, which follows the Flink execution model that creates threads separately for each query. To compare the performance without the query submission overheads, we optimize **TPQ** by decoupling the code registration and query submission similar to Pluto. However, as TPQ is unaware of the commonalities of codes and message brokers, it has bottlenecks in creating thousands of network connections for the source and sink and query codes of each stream query, which results in a $50K$ maximum throughput.

TP. We also implemented a **TP** (thread pool) model that shares threads among stream queries to compare the thread pool model with Pluto's IoT-aware execution model. We also optimize the query submission in TP like TPQ. However, the performance of TP is still low compared to Pluto as it has the same bottlenecks with the TPQ due to the large number of generated connections and codes.

3.5.3 Performance Breakdown

In this section, we break down the performance of Pluto. We implemented the variant of TPQ and TP, as illustrated in Table 3.1 and Figure 3.5 to evaluate the performance gain of sharing resources.

Query Submission. First, decoupling the code registration and query submission reduces the bottleneck in query submission. We can see the benefit by comparing Flink with TPQ in Figure 3.4.

Network Connection Sharing. Comparing TP with TP+N, and TPQ with TPQ+N shows the effectiveness of sharing network connections. The maximum

Name	**Description**
TP+N	TP + exposing broker address for network sharing
TPQ+N	TPQ + exposing broker address for network sharing
TP+NC	TP + exposing broker address and application code for network and code sharing
TPQ+NC	TPQ + exposing broker address and application code for network and code sharing

Table 3.1: TP and TPQ with network connection and code sharing for performance breakdown of Pluto.

throughput of TP+N and TPQ+N is $120K$, which is $2.4\times$ larger than that of TP and TPQ ($50K$). TPQ and TP still have the bottleneck in creating many codes.

Code Sharing. We can see the benefit of code sharing by comparing TP+N with TP+NC and TPQ+N with TPQ+NC. When sharing the codes, the maximum number of stream queries of TPQ+NC and TP+NC reaches $220K$ and $500K$, respectively. TPQ+NC has the bottleneck in creating and maintaining thousands of threads, whereas TP+NC reduces the threading overheads with the thread pool model. This is why TP+NC shows better performance than TPQ+NC.

Locality-Aware Scheduling. Comparing TP+NC and Pluto shows the performance enhancement by the locality-aware scheduling of Pluto. TP+NC processes stream queries without exploiting the locality of code references. In contrast, the locality-aware scheduling of Pluto uses the Q-group for exploiting the temporal locality of code references among queries and minimizes CPU cache

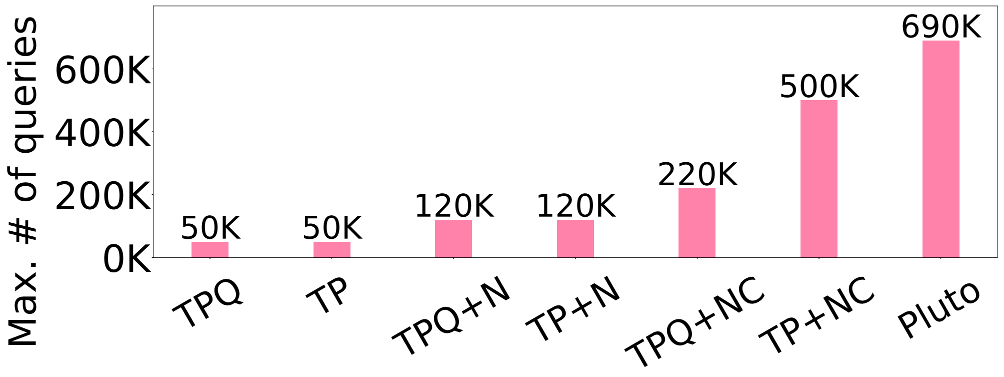

Figure 3.5: The results of the performance breakdown

misses.

Figure 3.6 shows the number of last-level CPU cache misses and CPU usage of TP+NC and Pluto, as well as showing the maximum throughput (number of queries). We measure the numbers up to the maximum number of queries that each system can handle. In this graph, we observe that TP+NC has more cache misses than Pluto (Figure 3.6(a)), especially when the number of queries is large. This shows that Pluto exploits the temporal locality of code references well by scheduling and processing queries with the Q-group. This difference leads to less CPU usage in Pluto, which enables Pluto to process more stream queries by up to $690K$ (Figure 3.6(b)) compared to TP+NC.

3.5.4 Load Rebalancing: Q-Group Split and Merging

To show that multiple IoT stream queries have little performance interference on multiple cores in skewed applications, we emulate an environment where many queries are submitted from a few applications. We create stream queries by selecting the registered codes in *Zipfian* distribution, setting the Zipf parameter to 1.0 for highly skewed distribution. This evaluation emulates the situation

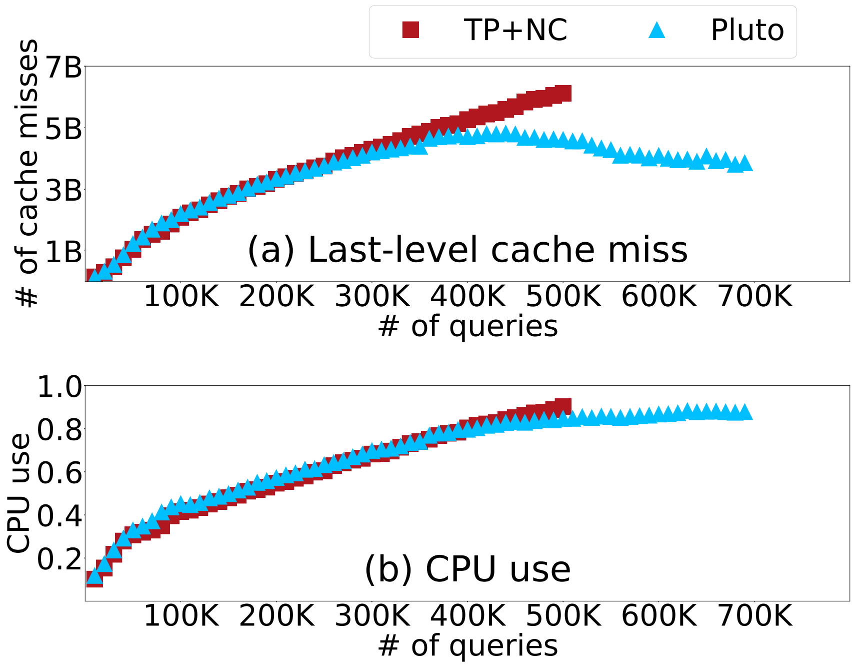

Figure 3.6: The number of (a) last-level cache misses, and (b) CPU usage of TP+NC and Pluto.

where many IoT stream queries share the popular IoT application codes.

Figure 3.7 shows the CDF of the latency when we turn on/off the load rebalancing (Q-group split/merging and reassignment) in Pluto. When the rebalancing mechanism is turned off, the $P99$ latency significantly increases up to 95 seconds as Pluto does not efficiently rebalance the load among multiple cores. However, when the rebalancing mechanism is turned on, the $P99$ latency becomes 961 ms, which indicates that Pluto well-balances the load among multiple threads and keeps latency.

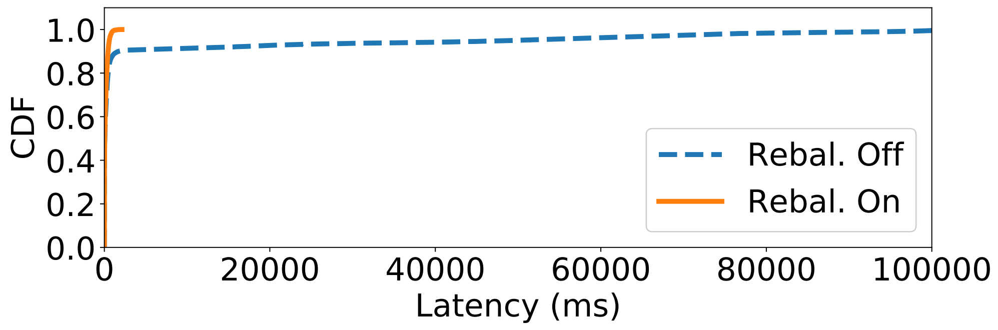

Figure 3.7: The cumulative distribution function of latency when we turn on/off the rebalancing mechanism in Pluto. When we turn off the rebalancing, the $P95$ and $P99$ (95th- and 99th-percentile) latencies are $49,453$ ms and $94,384$ ms, respectively. When we enable the rebalancing, the $P95$ and $P99$ latencies are 598ms and 961ms, respectively.

3.5.5 Tradeoff

We showed that Pluto outperforms existing stream processing systems when processing a large number of small IoT stream queries. In this section, we evaluate how the performance and the benefit of Pluto varies by increasing the input rates of data streams (stream sizes). We vary the input data rate of a stream query from $1/s$ to $100K/s$ and show the throughput of Pluto and Flink. The input data rate indicates the size of the data stream for each query. If it is large, it means that the query processes a big data stream.

Figure 3.8 shows the evaluation of Pluto and Flink. In this graph, the number of submitted queries is T/I, where T is the maximum throughput and I is the input rate of each query. Pluto outperforms Flink when the input rate of each query is small ($1/s$, $10/s$, $100/s$, and $1K/s$), as Pluto is optimized for processing small IoT stream queries. In contrast, Flink outperforms Pluto when the input rate of a query is large ($10K/s$, and $100K/s$), as they are targeted

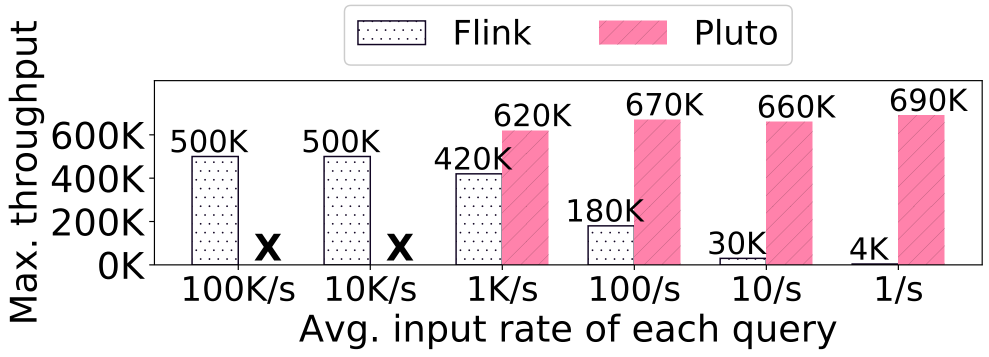

Figure 3.8: The throughput of Pluto and Flink in various input rates.

for processing individual big stream queries with massive parallelism.

Pluto cannot digest the large data stream ($100K/s$ and $10K/s$), which shows the limitation of Pluto in handling a small number of big stream queries because Pluto does not parallelize individual queries. As a single thread processes all of the events of a query in Pluto, the incoming events of a query are piled up and the processing latency significantly increases. This is why the throughput of Pluto is marked as **X** in $100K/s$ and $10K/s$. In contrast, Flink executes stream queries well when the input rate is large ($100K/s$ and $10K/s$) because it parallelizes individual big stream queries to multiple threads. However, when the input rate of each stream query becomes small ($< 1K/s$), Pluto outperforms Flink thanks to its IoT-aware optimization.

3.6 Discussion

API. As Pluto offers a new API for registering codes and building IoT queries, it requires additional steps for code and query submission. Different from existing stream processing systems, developers have to declare parameters and bind them to their UDF functions with Pluto's API to register codes. However, this step does not require the modification of the main application logic. If the main application code is written in the same language with Pluto's client, developers can reuse the code snippet. In addition, as Pluto's query building API follows the widely-used dataflow programming model, we believe that building a compiler that translates existing applications through Pluto's dataflow API and supporting various programming languages like Python can further reduce the developer's effort for porting and rewriting application codes, which is future work.

Edge-Cloud Query Processing. Although this paper focuses on query processing in the cloud, small IoT stream queries can be executed on the backend, on the edge IoT devices [96, 102], or on both [57] according to the IoT application workload and IoT devices. For instance, if an IoT device has enough computing power (e.g., RaspberryPi) for processing a query, and the query processes data streams generated from the local IoT device, we can execute the query on the edge device. In addition, if the network bandwidth of edge devices is limited, we can process data in the edge devices instead of sending data to the cloud. For optimization, the techniques used in Pluto's execution model (e.g., network/code sharing, thread-pool model) can also be applied on the edge devices. In contrast, if stream queries require joining remote data streams, or there are computations that can be shared across multiple stream queries, we can execute them in the cloud. Furthermore, when the query requires processing data streams generated from remote devices, or the edge device has limited computing power, delegating the query execution to the cloud is essential. Expanding the query processing on both edge and cloud is an interesting research topic, and building a compiler that automatically decides the placement of queries on edge and cloud based on the application workload and IoT device type is future work.

3.7 Related Work

Edgewise [35] optimizes stream processing in edge devices using the TP model. Fleet [87] is a framework for optimizing big stream queries on FPGAs, and FineStream optimizes them on a CPU-GPU integrated architecture [106]. Wukong+S [108] efficiently handles concurrent RDF stream queries dealing with both timeless and timing data. Different from RDF queries and the queries running on the edge devices, FPGA, or CPU-GPU integrated architecture, IoT stream queries submitted to the backend servers have commonalities (codes and network connections). Pluto exploits such characteristics, proposing the three-phase end-to-end query execution with new APIs and efficient execution model.

Stream databases [19, 71] are optimized for executing many stream database queries, which are different from IoT stream queries. The characteristic of stream database queries is that they usually process the same data streams of the same stream database fields and perform common SQL operations on the same data. Therefore, stream databases have optimized the execution of multiple stream database queries by extracting common computations in the SQL expression (e.g., multiple windows, join) and merging the same computations [25, 23, 107, 41]. Different from stream database queries, IoT stream queries perform user-defined operations on the different data streams for different IoT devices and users. Therefore, the query merging technique is limited because there are little duplicate computations among IoT stream queries.

Pluto's locality-aware processing and execution model is on the lines of Cohort [55] scheduling used in web servers in the sense that both consecutively process events with the same code. However, there are two fundamental differences. First, in Cohort scheduling, system developers should manually decide how to group events at compile time. Moreover, Cohort scheduling is applicable for events that invoke compiled server-side codes (e.g., HTTP web servers). In contrast, Pluto's locality-aware processing is designed for processing events that invoke UDF codes, registered at runtime by application developers (not by system developers). Therefore, Pluto automatically creates Q-groups to group events and queries that reference the same UDF codes. Pluto proposes new code registration and query submission APIs to achieve this. Second, Cohort scheduling is not optimized when the number of groups is large, as it is targeted for compiled server codes and does not provide load balancing mechanism for groups of events. In contrast, Pluto's locality-aware processing handles a large number of registered codes and Q-groups with a rebalancing mechanism by reassigning, splitting and merging Q-groups across threads.

Commercial IoT platforms such as AWS IoT [5], Scriptr [83], and Azure IoT [86] provide the backend servers for running IoT stream queries. Unfortunately, their proprietary backend systems are closed-source. Pluto is the first research work that investigates the characteristics of IoT stream queries and performs the IoT-aware optimization. We believe that Pluto can be adopted in the cloud platforms.

3.8 Summary

Pluto is a new IoT stream processing system with IoT-aware optimization. To efficiently share resources and map queries to the shared resources, Pluto decouples query submission and code registration with new APIs. Moreover, the IoT-aware execution model of Pluto exploits the temporal locality of code references and efficiently (re)balances the load among multiple cores with a new stream query scheduling unit, called Q-group. Our evaluation on various applications shows that Pluto improves the throughput by an order of magnitude compared to other stream processing and stream databases systems on a node with low latency.

Chapter 4

Streaming Dataflow Reshaping for Fast Scaling Mechanism on Lambda

4.1 Motivation

Big IoT stream processing queries process a large volume of real-time data and extract insights with low latency [99, 88, 98, 75, 16]. Due to the aggregated data streams, the input rate and the load of the jobs fluctuate over time according to the real-time workload. Furthermore, at times, the input rate can increase sharply.

Prior work has investigated fast, accurate, and automatic scaling decisions for streaming queries in the policy layer when bursty loads happen [48], such as quickly deciding when to trigger scaling and how much load to redistribute from existing resources to new resources. However, even if the policy is fast enough and makes quick decisions when bursty loads happen, if the underlying scaling mechanism (action) layer is slow, processing latency will increase until the scaling action is completed.

Unfortunately, when cloud virtual machines (VMs) are used to handle the load spikes cost efficiently without resource over-provisioning, the scaling of jobs is mainly delayed by the VM provisioning time. Dynamically (de)allocating VMs may take several minutes or tens of seconds to (re)start [34, 74, 51], but sudden bursty loads can happen in a few seconds [100, 62, 44].

Recently, cloud vendors provide new cloud computing resources called Lambda (a.k.a., serverless frameworks) that offer lightweight containers faster to start than VMs [6, 8, 38, 42]. The start-up time of Lambda instances is less than a few seconds (at least 10× faster than that of VMs [34]). Although the cost of using Lambda is more expensive than that of VMs, using Lambda instances only for infrequent bursty loads does not significantly increase the usage cost. In addition, even though there is a timeout for running Lambda instances [54], the maximum allowed timeout is usually longer than the duration of bursty loads that last few minutes [45], which is enough to handle the short-term bursty loads (e.g., the maximum timeout in AWS Lambda is 900 seconds). Recent research harnesses such properties of Lambda for low-latency data processing jobs that do not frequently happen, such as analyzing cold data or interactive video analytics [68, 34]. Our work is in the line of the existing work that harnesses Lambda for data processing. To the best of our knowledge, it is the first work to exploit Lambda for handling bursty loads of streaming queries.

4.2 Challenges

However, there are two key technical challenges in realizing fast scaling of stream queries on Lambda.

First, as data communication between Lambda instances is prohibited, it is

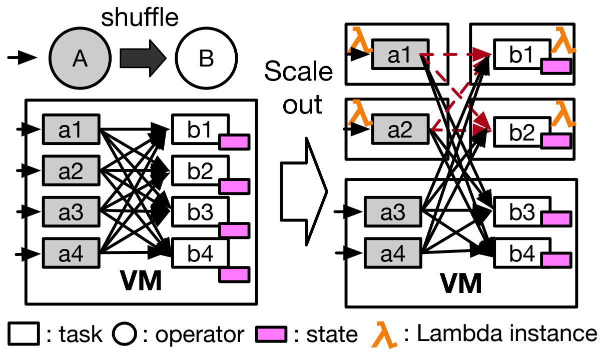

Figure 4.1: The illustration of Lambda that does not allow direct data communication when tasks perform N-to-N (shuffle) operations. For example, a_1 cannot send data to b_1 and b_2.

Symbol	Description
λ_{o_i}	The input rate of operator o_i
Λ_{o_i}	The maximum processing rate of o_i in a VM

Table 4.1: Notation used in this paper.

not possible to redistribute the load of tasks that are connected to each other to Lambda instances. To explain this situation, we use the notation defined in Table 4.1. For simplicity, we suppose that there is one VM, two operators $o_1 \Rightarrow o_2$ with shuffle edge are running on the VM, and the input rate of operators is proportional to the source rate. When $\lambda_{o_1} + \lambda_{o_2} < \frac{\Lambda_{o_1}+\Lambda_{o_2}}{2}$, the VM is not a bottleneck in processing the events of the operators because $\frac{\Lambda_{o_1}+\Lambda_{o_2}}{2}$, the maximum throughput of processing o_1 and o_2 events in the VM, is larger than the input rate of o_1 and o_2 $(\lambda_{o_1} + \lambda_{o_2})$. We divide $\Lambda_{o_1} + \Lambda_{o_2}$ by two to calculate the maximum throughput because the two operators share the CPU cores of the VM.

Once bursty loads happen, the input rate of o_1 and o_2 can be larger than the maximum throughput, $\lambda_{o_1} + \lambda_{o_2} > \frac{\Lambda_{o_1}+\Lambda_{o_2}}{2}$, which means that the events of o_1 and o_2 are piled up, and the processing latency increases. To mitigate the CPU bottleneck in the VM, we should migrate the tasks of o_1 and o_2 to Lambda instances, but it is not possible because direct communication is not allowed between different Lambda instances, as shown in Figure 4.1.

A simple solution that overcomes this limitation is to avoid migrating the tasks with shuffle communication to Lambda at the same time. For instance, we can move all of the tasks of o_2 (if $\Lambda_{o_2} < \Lambda_{o_1}$) to Lambda instances and reduce the load of o_2 in the VM. However, even though we reduce the overhead of o_2 from the VM, if $\lambda_{o_1} > \Lambda_{o_1}$, the VM can be the bottleneck, and latency increases. This situation happens when λ_{o_1} is high, or Λ_{o_1} is low (e.g., CPU-intensive aggregation). We investigate various stream query workloads in our evaluation to show the limitation.

Second, the overhead of migrating tasks and their states increases as the bursty load increases, which leads to the bottleneck in scaling stream queries on Lambda. When the peak load is high, it requires a huge amount of load redistribution and task migration to Lambda. In addition, the number of tasks (parallelism) should be large to reduce the portion of data that a task processes when bursty loads happen. As the number of tasks and amount of states distributed to Lambda instances increase, the time to task stopping, rescheduling, (de)serializing their states, and restarting tasks also increase. This situation indicates that we cannot fully take advantage of the fast-to-start Lambda instances because of the slow task migration mechanism.

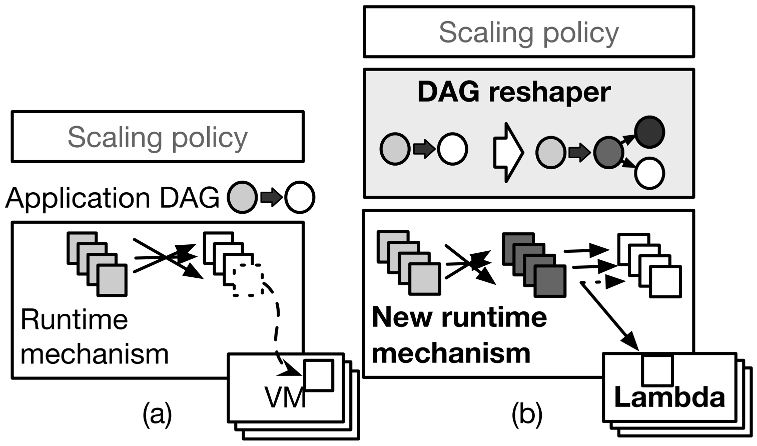

Figure 4.2: The overview of (a) existing stream scaling mechanism and (b) Sponge' scaling mechanism.

4.3 Design Overview

To address the challenges, our key idea is to optimize both query compiler layer and execution layer for fast scaling on Lambda. We found that there are opportunities to optimize stream queries by converting the logical dataflow graph to a fast-scaling-friendly DAG on Lambda, by being aware of *query semantics*: data communication patterns and stateful operators, and by being aware of *Lambda's container properties*: indirect data communication and warm instances. In doing so, we propose a novel streaming dataflow reshaping algorithm by inserting new utility stream operators into the logical DAG of streaming queries: router operators (ROs), transient operators (TOs), and state mergers operators (MOs), and the detail is explained in the next section.

To realize the DAG reshaping approach, we design a new stream system, called Sponge. Figure 4.2 illustrates the key difference between existing stream systems and Sponge. Sponge receives a query as a DAG, and the DAG reshaper converts the DAG (Section 4.4). Once the DAG is converted, Sponge creates a physical plan, consisting of tasks of operators. The number of tasks per operator is configurable and decided by users. Sponge schedules and deploys the tasks to the runtime (VM workers), which supports new utility operators with a new scaling protocol for correctness guarantee (Section 4.5). Once bursty loads happen and the scaling policy decides to scale out, Sponge redistributes the load of tasks from VM workers to Lambda workers.

In this work, we do not claim the novelty of the policy layer, such as deciding when to scale in/out, and which operators and tasks to scale in/out, because the policy is sensitive to traffic workloads. Rather than building a universal and automatic policy, it is more desirable for system developers, who have domain-specific knowledge on their workloads, to build an optimized policy based on their workload characteristics. Therefore, we expose scaling primitives in the policy layer, in order for the system developers to easily plug in/out their policies and scale in/out queries to Lambda, to VMs, or to both (Lambda and VMs) (Section 4.6).

4.4 Reshaping Rules

Algorithm 2: The algorithm of reshaping rules.

```
1 Function Reshaping(DAG)
2     return R3(R2(R1(DAG)))
```

In this section, we illustrate the algorithm of three reshaping rules and how Sponge realizes the fast scaling on Lambda while preserving application semantics. Basically, Sponge receives as input a logical streaming DAG at compile time and applies three reshaping rules: R1, R2, and R3, consecutively (Algorithm 2) before creating a physical plan. The output of the reshaping algorithm

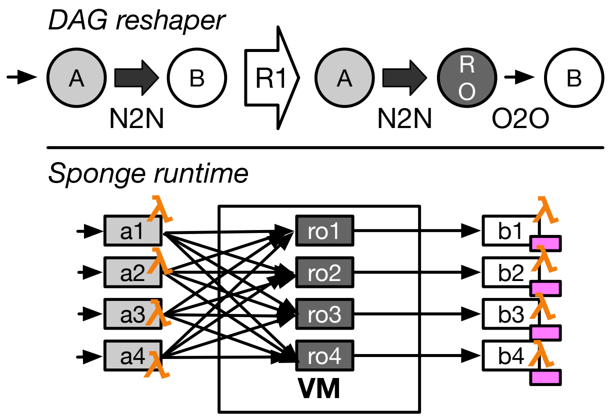

Figure 4.3: The illustration of applying the R1 reshaping rule to a stream query and how the execution layer supports the converted DAG for scaling.

is a modified logical DAG, which is converted to a physical plan and executed on the runtime.

4.4.1 R1: Inserting Router Operators

First, to enable data communication across tasks on Lambda instances, we insert a router operator (RO) between two operators with N-to-N data communication. The router operator routes data coming from upstream operators to the downstream operators and decouples N-to-N data communication between them. By keeping router operators on VMs, Sponge can move and scale out upstream and downstream operators together to Lambda instances.

Algorithm 2 and Figure 4.3 describes how Sponge converts a DAG by inserting router operators. Sponge traverses the DAG in a topological order to insert ROs. As a rule of thumbs, Sponge skips adding a router operator when $o_1 \rightarrow o_2$ has one-to-one connection (Line 4) because tasks with one-to-one edges can be executed in the same Lambda instance without data communication across Lambda. Increasing the parallelism of o_1 and o_2 prevents the Lambda instance

Algorithm 3: The algorithm of inserting router operators.

```
1  Function R1(DAG)
2      for vertex, inedges in DAG.topological_sort() do
3          for inedge in inedges do
4              if inedge.comm != OneToOne then
                   // insert a router operator
5                  RO = new RO(); DAG.add_vertex(RO); DAG.remove(inedge)
6                  $o1$ = inedge.src(); $o2$ = inedge.dst()
7                  edge_to_router = Edge({$o1$ →RO, inedge.comm})
8                  edge_from_router = Edge({RO→$o2$, OneToOne comm})
9                  DAG.add_edges([edge_to_router, edge_from_router])
10     return DAG
```

from being a bottleneck although multiple tasks with one-to-one edges are co-located in the same Lambda instance, and preserving one-to-one connection without inserting a router operator can also reduce additional data transfer overheads. Therefore, Sponge migrates them together in the same Lambda instance when operators are connected with one-to-one connection.

In contrast, whenever the N-to-N communication (shuffle) edge is encountered, Sponge inserts a router operator between o_1 and o_2. Adding an RO between o_1 and o_2 enables scaling o_1 and o_2 on Lambda at the same time and mitigates VM bottleneck, as shown in Figure 4.3. As RO only routes data without additional computations, RO has no input/output (de)serialization overhead, and the computational overhead of RO is smaller than other streaming operators. Therefore, Λ_{RO} is always higher than Λ_{o_i}, and Sponge can relax the bottlenecks in VMs by keeping ROs instead of other streaming operators on VMs. Sponge sets the parallelism of an RO equal to that of the downstream

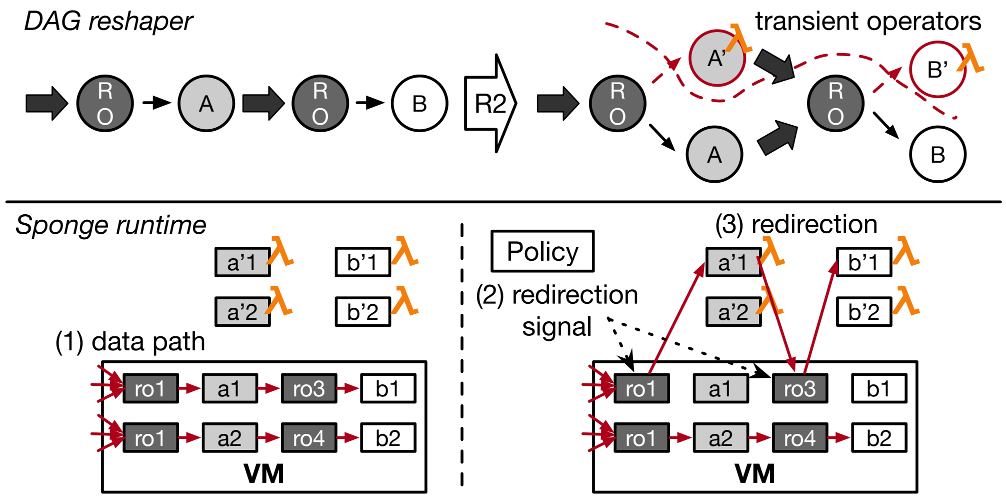

Figure 4.4: The example of applying the R2 reshaping rule and how the execution layer supports the redirection to transient operators for scaling.

operator when generating a physical plan after the rule is applied, because an RO has one-to-one connection with its downstream operator (e.g., $ro1 \rightarrow b1$ in Figure 4.3).

The limitation of inserting router operators is that VMs could be a bottleneck if the number of events to route and the routing overhead is large: $\lambda_{RO} > \Lambda_{RO}$. In this situation, it is inevitable to over-provision resources or dynamically create new VMs to mitigate the VM bottleneck. For this situation, Sponge provides scaling primitives that enable scaling on VMs (Section 4.6), in order for policy developers to design a policy that dynamically allocates both VMs and Lambda for various traffic workloads.

4.4.2 R2: Inserting Transient Operators

Applying R1 enables scaling of tasks with N-to-N data communications, so Sponge can migrate a large number of tasks from VMs to Lambda to mitigate

Algorithm 4: The algorithm of inserting transient operators.

```
Function R2(DAG)
    for origin, inedges in DAG.topological_sort() do
        if origin is not RO vertex then
            trans = new TransOperator(origin); DAG.add_vertex(trans)
            origin.set_trans(trans); trans.set_origin(origin)
        for inedge in inedges do
            src = if inedge.src() is RO: inedge.src() else inedge.src().trans()
            dst = if origin is RO: origin else trans
            trans_edge = TransEdge({src→dst, inedge.comm})
            DAG.add_edge(trans_edge)
    return DAG
```

bursty loads. Sponge minimizes the overheads of migrating a large number of tasks with the R2 reshaping rule, which harnesses Lambda's warm container property and inserts transient operators (TOs).

A transient operator (o_{trans}) is a copy of an existing streaming operator (o_{origin}), and it is deployed and scheduled on Lambda. By default, Sponge runs one transient task on a Lambda instance to prevent the CPU bottleneck in Lambda, but the scheduling of transient tasks can be configured by users. Sponge then redirects data from o_{origin} running on VMs to o_{trans} running on Lambda to reduce the task stop, restart, and rescheduling overheads when bursty loads happen. For redirection, an RO conditionally routes data from o_{origin} to o_{trans} based on the control signal provided by Sponge.

Sponge harnesses Lambda's *warm* container property to quickly redistribute the load from VMs to Lambda and reduce the task migration overheads. When Lambda instances are invoked and used once, they are not reclaimed by cloud vendors for a time even if we do not use them, which is known as warm containers (instances) [93]. Recent study [93] shows that periodically invoking warm Lambda instances can increase the retention time of Lambda. Sponge therefore pre-schedules transient operators to the warmed Lambda instances and periodically invokes the Lambda instances (every one minute by default). The duration of periodic invocation is small (less than 100 ms), so the warm-up cost is negligible.

Algorithm 4 illustrates the R2 reshaping rule, and Figure 4.4 shows an example of applying the R2 reshaping rule and how the runtime supports redirection. For each stream operator o_{origin} except for ROs in the input DAG of `R2` function, Sponge creates a new corresponding TO (Lines 3—5). After creating and adding a TO, Sponge creates transient edges for redirection (Lines 6—10). In Figure 4.4, A' and B' represent transient operators of A and B, respectively. When the load is stable, Sponge processes data through A and B (Figure 4.4(1)). Once bursty loads happen, Sponge sends the redirection signal to router tasks ($ro1$ and $ro3$ in Figure 4.4(2)). The router tasks then redirect data from the original tasks ($a1$ and $b1$) to the tasks of transient operators ($a'1$ and $b'1$) (Figure 4.4(3)). Such redirection is a lightweight mechanism, as it does not require the stopping, restarting, and rescheduling of tasks.

The downside of transient operators is that when a Lambda instance is not used during the stable load, it can be reclaimed by cloud vendors even though Sponge periodically invokes them. As a result, the transient tasks scheduled in the reclaimed Lambda instance can be lost. However, as transient operators do not process any data and have no states during the stable load, and they are dummy operators during stable load, Sponge can quickly re-schedule them and warm-up Lambda instances again immediately.

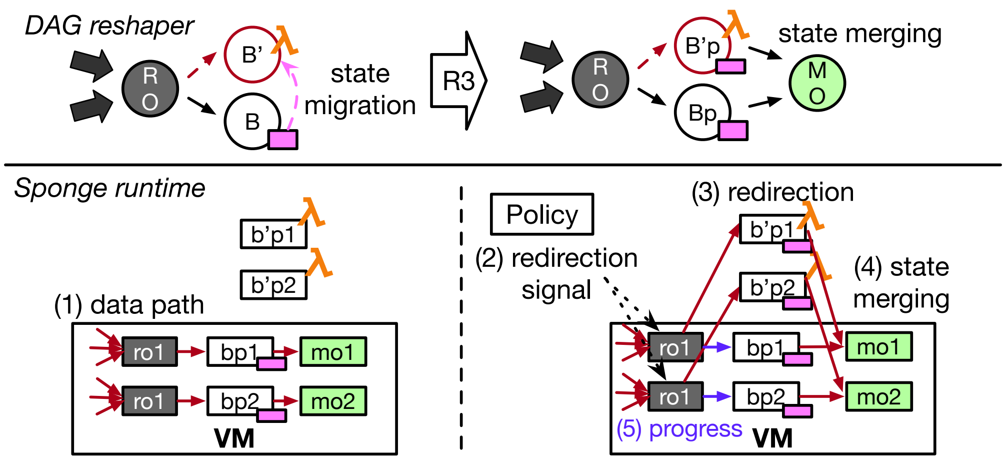

Figure 4.5: The example of applying the R3 reshaping rule and how the execution layer supports the merging operators to minimize state migration overheads.

4.4.3 R3: Inserting State Merger Operators

Although scheduling transient operators and redirecting data reduces the task migration overheads, Sponge should synchronize and migrate the states of operators to Lambda for correctness before redirecting data.

To reduce the overheads of state migration, the final reshaping rule is to insert state merger operators (MOs), by being aware of stateful operator's semantics. When a stream operator (os_{origin}) is a stateful operator with commutative and associative operation properties, Sponge prevents the migration of a task state from os_{origin} to os_{trans} by merging their states in the MO operator.

Figure 4.5 shows the example of applying the R3 and how the execution layer supports the MO. To add an MO in the DAG reshaper, first, Sponge finds stateful operators with commutative and associative properties from the input DAG (B and B' in Figure 4.5). Second, Sponge converts the stateful operators to partial aggregate operators (Bp and $B'p$ in Figure 4.5), and inserts an MO that performs final aggregation. Third, Sponge connects the partial operators with the MO, and the MO with the downstream operators of the original stateful operators.

When the load is stable, the router tasks do not redirect data to $B'p$, which means that the MO does not perform merging as it receives data only from Bp (Figure 4.5(1)) with complete states. Therefore, the overhead of MO during the stable load is negligible. Once bursty loads happen, Sponge sends the redirection signal to the router tasks (Figure 4.5(2)). Without synchronizing the state between Bp and $B'p$ tasks, the router redirects the data from Bp to $B'p$ tasks to quickly redistribute the load (Figure 4.5(3)). The tasks of $B'p$ then receive data without states, build up partial states from scratch by processing input data, and emit output data computed from the partial states to the MO (Figure 4.5(4)). Meanwhile, the tasks of Bp also emit corresponding output results to the MO at the same time. The MO then receives the outputs from $B'p$ and Bp tasks and combines the outputs to finalize the result with complete states.

For Bp to emit the corresponding output with $B'p$, Sponge injects the progress information (watermarks) of $B'p$ to Bp (Figure 4.5(5)). For instance, if $B'p$ receives data until time $t1$ and emits outputs, the router sends this information to Bp, in order for Bp to emit outputs until time $t1$. By doing so, the MO can merge outputs at $t1$. We will illustrate the details in the next section.

The limitation of the state merger is that it cannot be applied to non-associative operations. In this case, Sponge also performs state migration. However, according to the recent research [92], most aggregation queries use the commutative and associative aggregate operations (e.g., sum, min, max). Therefore, we believe that the R3 reshaping rule is applicable for a large number of stateful stream operations.

4.5 Scaling Protocol

A stream query should generate the same result after scaling, and Sponge should guarantee the correctness result. In stream analytics, out-of-order processing is allowed for data parallel operation: e_1 event can arrive before e_2 event where e_1 event timestamp is after e_2 event timestamp: e_1.timestamp $>$ e_2.timestamp. To indicate the progress of out-of-ordered events, stream queries use *watermarks* [1]. When a query receives watermark $w1$, the query assumes that all of the events generated before $w1$ are already received and processed.

To guarantee this semantic, Sponge builds new scaling protocols by injecting control messages into the data plane like Chi [60]. In this section, we illustrate how Sponge routes data from VMs to Lambda (Section 4.5.1) and merges states with merger operators (Section 4.5.2) while guaranteeing correct results.

In addition, Sponge also supports the existing task migration mechanism, in order for distributing the load of tasks to VMs as well as to Lambda (Section 4.5.3). With the task migration, the policy developers of Sponge can design a scaling policy optimized for their workloads that scale out queries on both VMs and Lambda with policy primitives (Section 4.6), and Sponge offers the playground for building various policies that harness the benefit of both VMs and Lambda.

4.5.1 Redirection Protocol

Figure 4.6(a) illustrates the redirection protocol. Once an RO task receives a signal for redirecting data from VM to Lambda (1), it injects a control message, cm, to the data channel of its downstream VM task T (2). The cm represents a checkpoint before which T processes all of the events $E_{<cm}$ emitted from the RO task: $E_{<cm}$=$\{e|e$ is emitted before cm in the RO$\}$). The RO task then

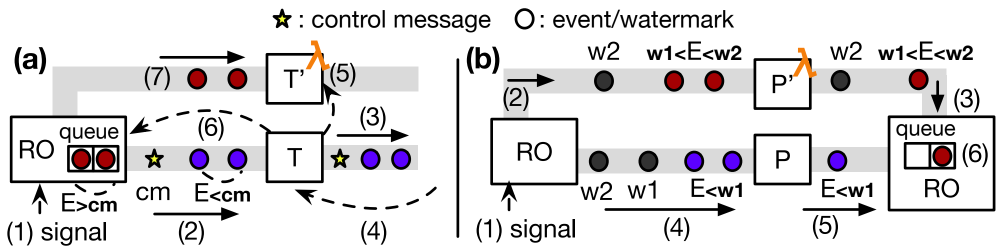

Figure 4.6: The illustration of redirection (a) and merging (b) protocols of Sponge.

buffers events arriving after cm, $E_{>cm}$, in its buffer queue.

Once T receives cm, it sends cm to its downstream tasks DTs (3) and waits for the acks from DTs (4). When T receives all of the acks, it indicates that there are no pending and inflight $E_{<cm}$ between $R \rightarrow T$ and $T \rightarrow DTs$. As all of the events before cm are processed by T, when T has non-mergeable state, Sponge can safely migrate the state of T to its transient task T' (5). T sends the ack to the RO task after the state migration is finished (6). The RO task then emits the buffered events and redirects data to T' (7).

This protocol guarantees that T consumes all of the $E_{<cm}$ with its state, and T' restores the state and processes event $E_{>cm}$ after state migration. Therefore, the event processing order is preserved between T and T', and Sponge guarantees correctness results.

4.5.2 Merging Protocol

Figure 4.6(b) illustrates the merging protocol. Sponge does not synchronize states between tasks when merging is possible because the partial states are merged in the MO. Therefore, once an RO receives a signal for redirection from VM to Lambda (1), it immediately redirects data to the Lambda task P' (2). As a result, the progress of event processing can be out-of-ordered between P and

P'. For instance, P' can consume events generated after $w1$ and emits partial states with $w2$ ($w2 > w1$) (3), whereas P consumes events generated before $w1$ (4) and emits partial states $w1$ (5). To prevent out-of-order processing, the MO reorders the partial states of P and P' based on the watermark (6). For example, the MO waits for P's output to merge the partial states of P and P' at $w2$. The RO sends watermarks both P and P', in order for P to emit its partial state according to the watermark and for the MO to enable merging in the MO based on the watermark.

4.5.3 Migration Protocol

For migration, Sponge sends a stop signal to T. T then sends control messages to its upstream and downstream tasks and waits for acks from them to guarantee that there are no pending and inflight events between upstream and downstream channels (like (3) and (4) in Figure 4.6(a)). Once all of the acks is received in T, Sponge checkpoints the state of T, removes T from the VM, and reschedules T to a Lambda instance. Within the Lambda instance, T is restarted by instantiating its operator function, restoring its state, and reconnecting and initializing data channels between its upstream and downstream tasks. These procedures are skipped in redirection as explained in Section 4.5.1, because Sponge redirects data to the pre-scheduled transient task where the data channel is already established, so the redirection protocol is more lightweight than the migration protocol.

4.6 Implementation

We have implemented Sponge with Java 1.8, AWS Lambda, and various open-source projects, such as Apache Beam [11] for frontend and Nemo [101] for

DAG reshaping, and boto3 [14] for managing and deploying Lambda workers.

Frontend: To receive a stream query as a DAG, we use Apache Beam [11], which is widely used frontend for various systems (e.g., Spark, Flink, Cloud-Dataflow). In addition, as Beam provides APIs for developers to build associative and commutative operations (e.g., combiner), Sponge extracts this information to identify which stateful operations are associative and commutative and to apply the R3 rule.

Compiler: To reshape the DAG, we use Apache Nemo [101] because Nemo provides IR (intermediate representation) optimization passes, by which the DAG can be converted to an optimized one. We implement three IR optimization passes for R1, R2, and R3 rules on top of Nemo and reshape the DAG received from Beam.

Runtime: We modify Nemo runtime (Java) for fast scaling on Lambda to support the migration of tasks and redirection of data from VMs to Lambda. Sponge executes a worker process on a VM and Lambda instance. Each worker manages a thread pool that contains a fixed number of threads and assigns the tasks to the threads. VM workers communicate with other VM and Lambda workers, but Lambda workers communicate only with VM workers due to the Lambda's data communication constraint. For deploying the worker code to AWS Lambda, we use AWS SDK boto3 [14].

Policy Primitives. Sponge offers the following policy primitives for policy developers who have their workload-specific knowledges to write customized policies:

- redirection(op, ratio): redirect *ratio* percent of *op* tasks (load) to Lambda or to VM. If *op* is running on a VM, Sponge redirects its data to the corresponding transient operator. We adjust the amount of load to be distributed from VM to Lambda by changing the *ratio* in our evaluation.

- migration(op, ratio, VM/Lambda worker): migrate *ratio* percent of *op* tasks (load) to a VM/Lambda worker. This primitive enables policy developers to migrate tasks across VMs or Lambda instances. We use this for evaluating the existing task migration approach in our evaluation.

Source Throttling: Traditional backpressure mechanism [31] is slow in preventing systems from being overloaded when sudden bursty loads happen, as it sends backpressure signal from the sink of queries to the sources, but the bursty events flow from the source to the sink. To prevent Sponge from being overloaded, Sponge preemptively throttles the source of queries by measuring the input rate. When the input rate exceeds the processing rate of the cluster, Sponge throttles the source until scaling decisions are triggered and the scaling mechanism is finished. Once the scaling is done and the load of tasks is distributed, Sponge smoothly increases the emission rate of sources to warm up newly allocated resources for event processing.

4.7 Evaluation

We answer the following questions in our evaluation:

i) How does Sponge quickly mitigate bursty loads compared to other systems (Section 4.7.2)?

ii) Which factors contribute to the performance of Sponge (Section 4.7.3)?

iii) What is the latency-cost($) trade-off of using Sponge (Section 4.7.4)?

4.7.1 Methodology

Environment. We use AWS EC2 r5.xlarge instances (32GB of memory and 4 vcores) for VM workers and AWS Lambda instances for Lambda workers

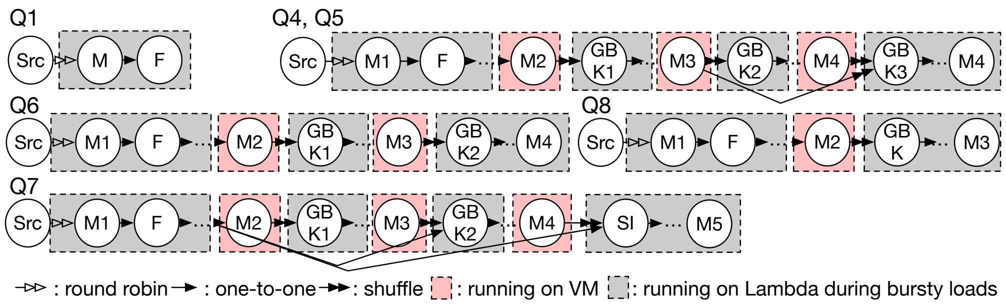

Figure 4.7: The simplified original DAG of queries used in our evaluation (Origin-L). M is a map operator, F is a filter operator, GBK is a group-by-key window operator, SI is a operator that handles side inputs. We migrate operators in gray box to Lambda when bursty loads happen to prevent the bottleneck in VMs. All of the GBKs are commutative and associative stateful operators, and SI is a non-mergeable stateful operator.

($1,769MB$ of Lambda as AWS Lambda offers one vCPU at $1,769MB$). We set up VPC for data communication between VMs and Lambda instances. To prevent bottlenecks in data stream generation and data forwarding, we use two additional instances, one (c5.16xlarge) for generating data and the other one (c5.12xlarge) for running Kafka [52] that forwards data streams to the stream processing systems.

Workloads. NEXMark benchmark [70] is a widely used streaming benckmark [48, 56] that contains diverse stream queries with complex dataflow graphs and stateful operations. Among 8 (Q1-8) NEXMark queries, we choose 6 queries because they represent distinctive data communication patterns, stateless and stateful operations: Q1, Q4, Q5, Q6, Q7, and Q8.

NEXMark emulates data streams of auctions and bids, and the queries analyze the auction and bid data in real time. We omitted Q2-3 because Q2 is a stateless query similar to Q1, and Q3 is a non-associative stateful query like

Query	Stateful	State Size	# of Tasks (per Op.)	Stable input rate
Q1	X	-	120	550 K/s
Q4	O	~90 MB	60	190 K/s
Q5	O	~2.4 GB	70	19 K/s
Q6	O	~73 MB	70	230 K/s
Q7	O (not mergeable)	~1.5 GB	90	15 K/s
Q8	O	~7 GB	60	60 K/s

Table 4.2: The characteristics of the NEXMark queries used in our evaluation.

Q7.

Table 4.2 summaries the characteristics of the original DAG of each query, and Figure 4.7 illustrates the simplified DAG of queries. **Q1** simply converts concurrency of bids, and it represents a simple query containing map operations without N-to-N data communication and stateful operations. **Q4** analyzes the average price for a category of bid items on a window, and **Q5** selects hot auction items in a window. **Q6** computes average selling price by seller on a a global window, and emits output whenever a winning bid event is created. **Q7** selects the bid with the highest bid price in a window. Q7 in Apache Beam uses a side input operation [12], and Apache Nemo processes the side input with non associative and commutative operations. We use this query as a representative one of non-associative stateful query to illustrate the limitation of R3 reshaping rule. **Q8** monitors new users who created auctions in a window and performs join operations. As the join operation appends data, the size of states of the join is large.

To measure the latency of query outputs every one second for windowed operations, we set the window interval size to 1 second, and the window size of queries to 60 seconds for Q4-5 and Q7 in our evaluation. For Q8 that uses a tumbling window, we modify Q8 to aggregate data in sliding windows with window size 60 seconds and interval 1 second.

Baseline. We compare Sponge with the following baselines:

- **NoScaling** does not scaling out stream queries when bursty loads happen.
- **Origin-VM** dynamically creates VMs and migrates tasks for scaling of queries without dataflow reshaping.
- **Origin-L** dynamically creates Lambda instances and migrates tasks for scaling of queries without dataflow reshaping.

For Origin-VM, we created VMs and stopped them before the evaluation. When bursty loads happen, we start the stopped VMs, launch workers on the VMs, and move tasks to the newly launched VMs from the existing VMs.

For scaling on Lambda (both Origin-L and Sponge), we launch the workers on Lambda instances and warm-up the runtime by processing several events at the start time of the evaluation. Due to the JIT compilation, JVM is cold if it does not process any events, and the event processing throughput is too low at the start time. To minimize such JIT compilation effects and focus on the effectiveness of the reshaping rules, we warm up JVM to trigger JIT compilation before bursty loads happen on Lambda instances in each baseline.

Bursty Traffic and On-Demand Allocation. We emulate bursty traffic where the input rate significantly increases in a short period of time to analyze how the lightweight and fast scaling mechanism of Sponge can reduce latency during the bursty loads. Figure 4.8 shows an example of the pattern of bursty traffic used in our evaluation. In this traffic pattern, we first generate stable

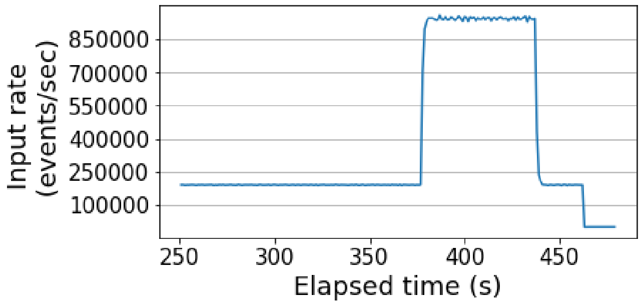

Figure 4.8: An example of the bursty input pattern used in our evaluation. At time $t = 380$, the input rate suddenly increases during 60 seconds.

input streams where the input rate is stable and does not fluctuate. At some point ($t = 380$ in this evaluation), we increase the input rate for a configured time (60 seconds) to emulate a sudden bursty load, and then decrease the rate back to the stable input rate. In our evaluation, we increase the input rate from 3 to 6 times of the stable input rate. By default, we set the burstiness ($\frac{\text{bursty input rate}}{\text{stable input rate}}$) to 5.

Basically, we run 5 VM workers when the load is stable and does not fluctuate. For the stable input, we generate events (per second) until all of the VM cluster CPU utilization (5 VM workers) is between 0.6 and 0.8 to prevent clusters from being under-loaded and over-loaded resources. As queries have different computational complexity, the stable input rate is configured differently for each query (see Table 4.2). The required computation of $Q5$, $Q7$, and $Q8$ is larger than that of $Q1$, $Q4$, and $Q6$, which is why the stable input rate of $Q5$, $Q7$, and $Q8$ is lower than that of $Q1$, $Q4$, and $Q6$. Once bursty loads occur, we dynamically invoke from 100 to 200 Lambda instances for Sponge and Origin-L, and dynamically allocate from 25 and 35 VM instances for Origin-VM depending on the query load to prevent resource bottlenecks in both VMs and

Lambdas when bursty loads happen.

Parallelism (number of tasks). We empirically set the number of tasks for each operator. Setting the number of tasks too small leads to the bottleneck in processing a task when bursty loads happen because a task will handle a large portion of bursty data. In contrast, setting the number of tasks too large increases task migration and runtime overheads of Origin-L. We therefore set the number of tasks as small as possible, while processing a task does not cause a CPU bottleneck in both VM and Lambda of all baselines when bursty loads happen. Table 4.2 includes the number of tasks of each operator in the evaluated NEXMark queries. We will show how to set the number of tasks in Section 4.7.3.

Policy Configuration. As the goal of our evaluation is to show the effectiveness of Sponge's reshaping rules for fast scaling mechanism, we manually configure the best scaling policy for each baseline to eliminate the effect of scaling policy. For instance, all systems make scaling decisions at the same time. In addition, we vary the amount of load to be redistributed and the amount of tasks to be migrated (or redirected) when bursty loads happen for each baseline, in order not to make resource bottlenecks in VMs and Lambda as much as possible. We then select the best configuration that quickly mitigates latency increase for each baseline. By default, we create one Lambda instance to run a task and assign one Lambda vCPU to a task to prevent the CPU bottleneck in Lambda.

4.7.2 Performance Analysis

Figure 4.9 illustrates the latency plot of all systems when bursty loads happen at $t = 380s$ and scaling is triggered at $t = 381s$. Overall, Sponge significantly reduces latencies compared to Origin-VM and compared to Origin-L (except for Q1). The latency of NoScaling continuously increases as the existing VM

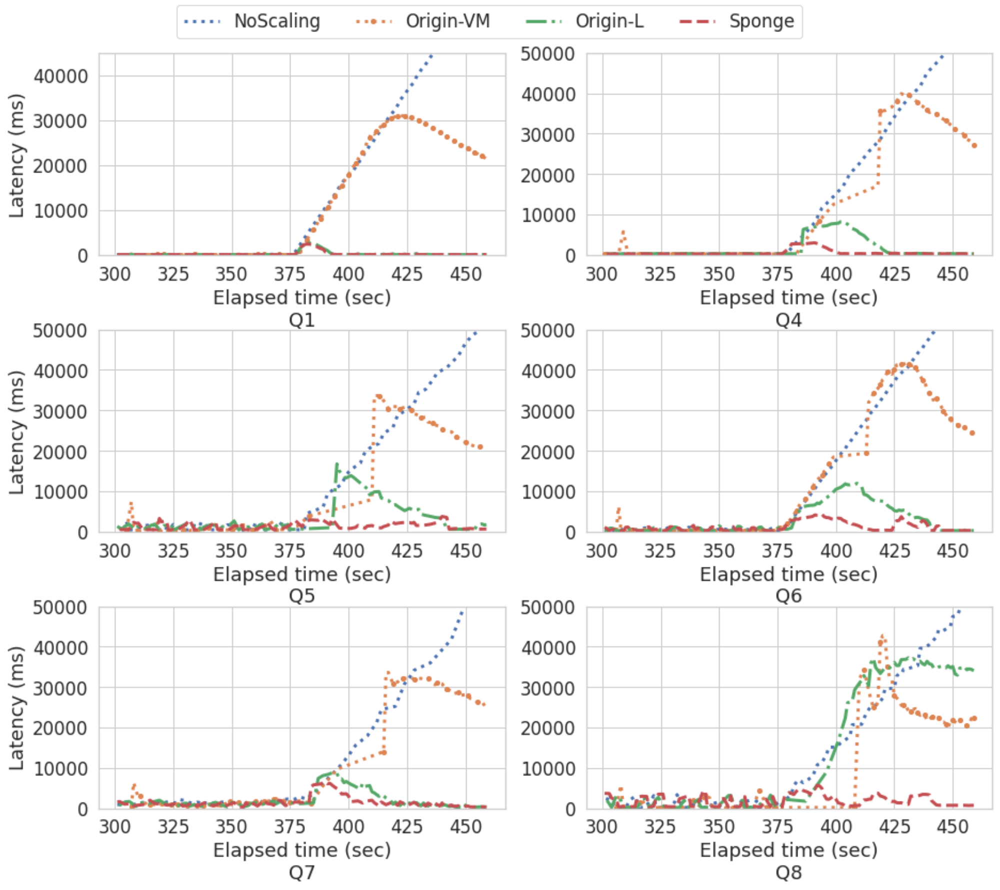

Figure 4.9: The latency graph in various baselines. The bursty load happens at $t = 380s$. We configure all of the systems to detect the bursty load and make a scaling decision at $t = 381s$.

resources become the bottleneck in handling the bursty loads.

Origin-VM. When we scale out queries by dynamically allocating VMs (Origin-VM), latency increases up to 44 due to the slow start-up time of VMs. Specifically, we have observed that the time to start VMs takes around $25 \sim 30$ seconds, the time to start JVM worker processes on the newly started VMs takes around 5 seconds. Moreover, as the JVM processes are cold at the start

time and JIT compilation is not triggered, the processing throughput is low at the start time of VMs, which leads to latency increase up to 44 seconds. After new VMs are instantiated, tasks are migrated to new VMs, and the JVM processes are warmed up, the latency of Origin-VM decreases as shown in Figure 4.9 because the processing throughput becomes larger than the input rate.
Origin-L. The slow start-up time of Origin-VM can be mitigated by using Lambda, and Origin-L illustrates how the latency is improved. In scaling out Q1 (a simple stateless query), Origin-L significantly reduces the latency compared to Origin-VM owing to the fast-startup time of Lambda, where the start-up time of Lambda takes less than one second in our evaluation. This result represents that only using Lambda instead of VMs can significantly improve the latency for scaling out a simple stateless query, as shown in MArk [105].

However, for scaling out other complex queries with N-to-N data communication and stateful operations, the performance gain of Origin-L compared to Origin-VM lessens, which indicates that naively scaling queries on Lambda without DAG reshaping has limitations. In Q4, Q6, latency increases up to 12 seconds due to the overheads of operators running on VMs. In Q5, Q7, and Q8, there are latency spikes due to task and state migration overheads. Compared to Origin-L, the latency of Sponge does not significantly increase thanks to the Sponge's DAG reshaping. In the following sections, we provide the detailed analysis and the effectiveness of the Sponge's reshaping rules, and why the Origin-L has limitations in scaling various queries on Lambda.

4.7.3 Performance Breakdown

To analyze the performance gain of Sponge, we also evaluate the following two baselines:

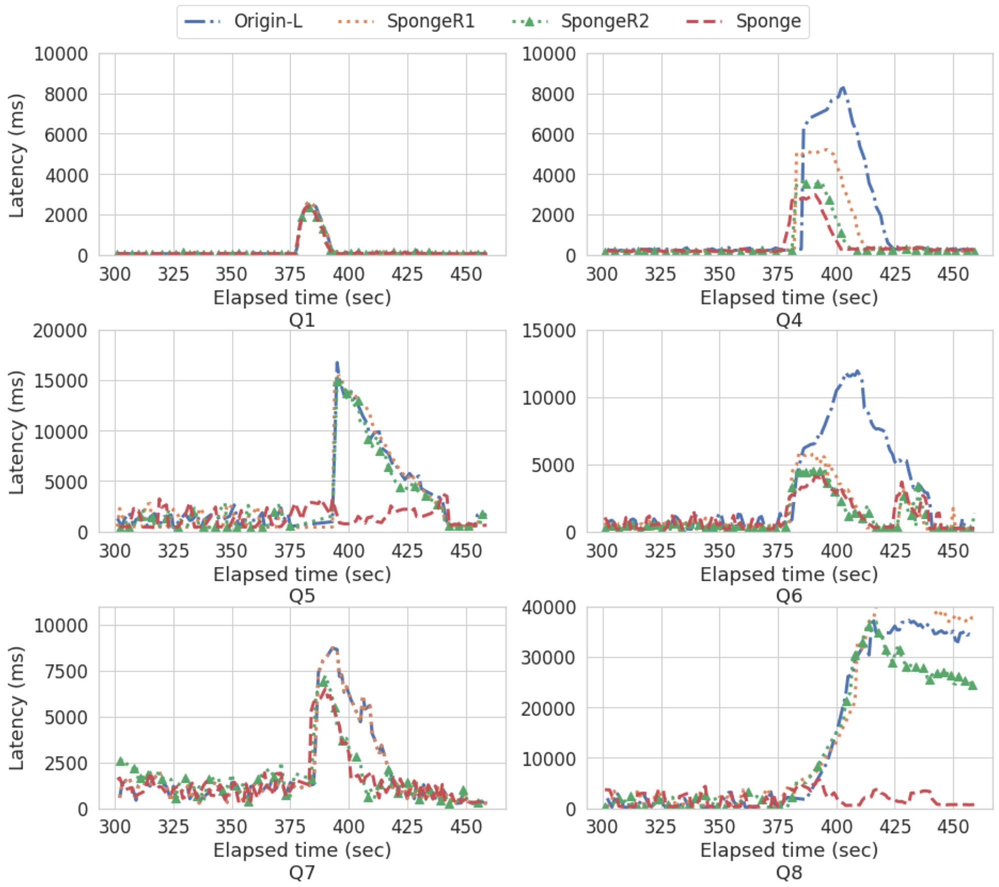

Figure 4.10: The latency graphs of Origin-L, SpongeR1, SpongeR2, and Sponge to analyze and breakdown the performance of Sponge.

- **SpongeR1** dynamically creates Lambda instances for scaling of queries with R1 reshaping.
- **SpongeR2** dynamically creates Lambda instances for scaling of queries with R1 and R2 reshaping.

Figure 4.10 illustrates the latencies of Origin-L, SpongeR1, SpongeR2, and this section breaks down the performance of Sponge with Figure 4.10.

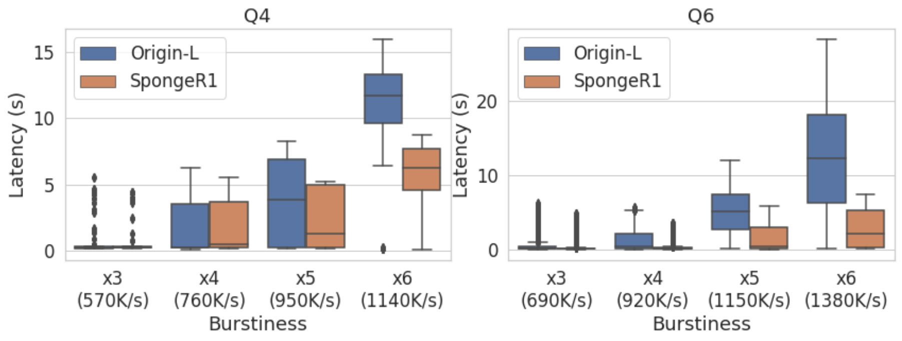

Figure 4.11: The latency during bursty period with various burstiness in scaling Q4 and Q6.

Router Operator Effect

Comparing SpongeR1 with Origin-L shows the router effect. As Q1 has no N-to-N data communication, there is no difference between the DAG of SpongeR1 and Origin-L, which is why their latencies are almost the same. In contrast, the latency of Origin-L is higher than that of SpongeR1 in Q4 and Q6 because VMs become the bottleneck in processing the events of M operators running on VMs (only 3% of input events are filtered out before $M2$). Usually, the amount of computation of M is smaller than that of GBK, as GBK requires additional aggregation. This is why we keep $M2$, $M3$, and $M4$ in Q4 and $M2$ and $M3$ in Q5 on VMs when bursty loads happen to minimize the CPU bottleneck on VMs as shown in Figure 4.7. However, although we migrate other operators to Lambda, the input rate of M operator is higher than the maximum throughput on VMs during bursty period in Q4 and Q6, so the events of M operators are piled up, and latency still increases.

To show the overhead of M operators, we vary the burstiness ($\frac{\text{bursty input rate}}{\text{stable input rate}}$ from 3 to 6 and show the latency during bursty period (Figure 4.11) in Q4 and Q6. When the burstiness is 3 and 4, VMs can process all of the input events of M operators, so the latency of Origin-L does not increase and is similar to SpongeR1. However, when the burstness increases to 6, VMs become the bottleneck in processing the input events of M in Origin-L, which is why the latency increases. Different from Origin-L, SpongeR1 adds an RO between M and GBK, and migrates both M and GBK to Lambda while keeping the RO on VMs. As RO does not (de)serialize events and does not perform computation, the amount of computation of RO is always smaller than that of M. As a result, SpongeR1 mitigates the bottleneck on VMs and reduces latencies compared to Origin-L down to 70%.

In Q5, Q7, and Q8, the latency of Origin-L is similar to SpongeR1 as the overhead of running M operators on VMs is low. The main bottlenecks in Q5, Q7, and Q8 are GBK operators, which require huge aggregate computations. The input rate of M operators in Q5, Q7, and Q8 is smaller than that of Q4 and Q6 (less than 10% of the Q4 and Q6 input rate), so keeping them on VMs does not cause the VM bottleneck. This result indicates that R1 is effective when the input rate and the overhead of operators running on VMs is high.

Transient Operator Effect

Sponge further reduces latencies when scaling out queries by inserting transient operators and redirecting data, instead of stopping, rescheduling, and restarting tasks. Comparing SpongeR1 and SpongeR2 shows the effectiveness of transient operators.

The effectiveness of R2 increases as the number of tasks to be migrated (or redirected) increases. In Q1, the number of tasks to be migrated or redirected is 100 in both SpongeR1 and SpongeR2, and the time to migration and redirection is 836 ms and 517 ms, respectively, which has little difference between the latency of SpongeR1 and SpongeR2.

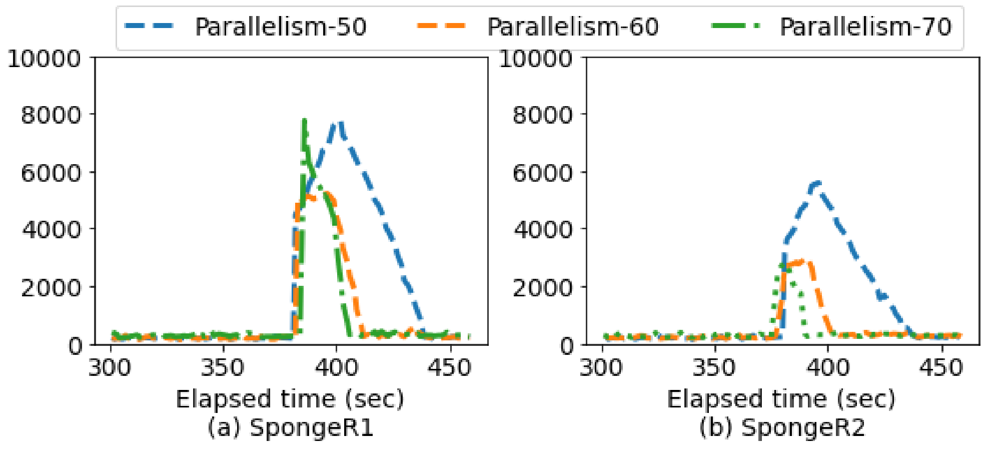

Figure 4.12: The latency graph with various parallelism (number of tasks per operator) in SpongeR1 (a) and SpongeR2 (b) for scaling Q4. As the parallelism increases, the number of tasks to be migrated or redirected also increases.

Among the evaluated queries, Q4 requires a large number of tasks to be migrated (or redirected). For Q4, we migrate and redirect 85% of total tasks to Lambda to mitigate the bottleneck in the VMs in SpongeR1 and SpongeR2. When migrating the tasks, SpongeR1 takes around $2,828$ ms for migration. In contrast, for redirection, SpongeR2 takes around $1,358$ ms. Due to the fast redirection mechanism, SpongeR2 reduces latency down to 28% compared to SpongeR1. Similarly, in Q7, SpongeR2 also reduces the latency compared to SpongeR1.

When the number of parallelism increases, the amount of tasks to be migrated or redirected also increases. To show the benefit of transient operators in this situation, Figure 4.12 illustrates the latency plot in various parallelisms of Q4 operators. When we set the parallelism to 50 for each operator, the task migration/redirection overhead is small, but the latency increases after the migration and redirection in both SpongeR1 (a) and SpongeR2 (b), because one

Lambda vCPU is overloaded in processing the events of a single task. With 70 parallelism, latency decreases after the migration and redirection, but the task migration overhead increases in SpongeR1 (a). As a result, the peak latency increases up to 8 seconds. In contrast, due to the lightweight redirection, the peak latency of SpongeR2 is around 3.5 seconds with 70 parallelism, which is 56% smaller than SpongeR1.

Creating a large number of tasks is inevitable when handling high bursty loads, in order to prevent processing a task from being overloaded in a Lambda vCPU. Even though the number of tasks increases for bursty loads, Sponge can handle high bursty loads with small redirection overheads.

Merger Operator Effect

Although R1 and R2 rules are applied, SpongeR2 still suffers from high latencies in Q5, Q7, and Q8 due to the state encoding/decoding overheads. Q4 and Q6 have small states (less than 100 MB), so the state migration overhead is negligible.

The state migration overhead increases to the state size. The time to encode/decode the state of Q5, Q7 and Q8 takes around 13s (for $\sim$ 2.4 GB state), 6s (for $\sim$ 1.5 GB state), and 35s (for $\sim$ 7 GB state), respectively. As a result, the latency of SpongeR2 increases up to 15, 7, and 38 seconds in Q5, Q7, and Q8. In contrast, Sponge significantly reduces the latencies in Q5 down to 4 seconds and in Q8 down to 6 seconds, preventing the state migration with merger operators. In Q7, the latency of SpongeR2 and Sponge is similar, which means that Sponge also suffers from the state encoding/decoding overheads because Q7 is non-mergeable stateful query. When queries have non-mergeable stateful operations, it is inevitable to over-provision resources in order to minimize the latency spikes in Q7 or to replicate states across VMs like Rhino [22].

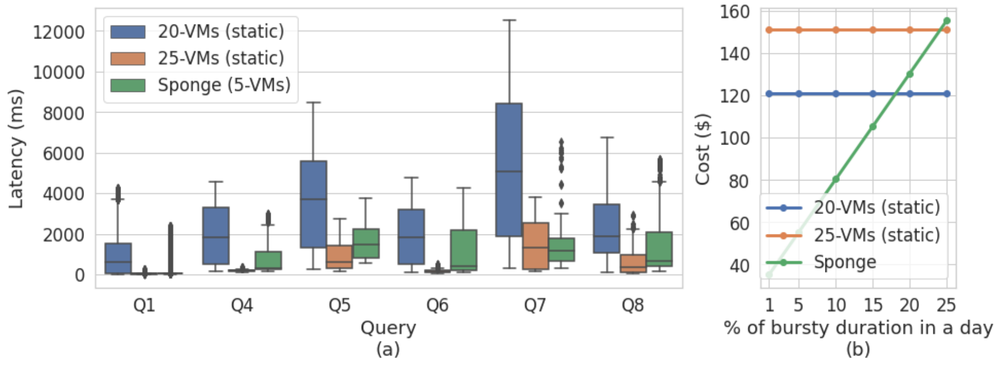

Figure 4.13: (a) The latency during bursty period. (b) The back-of-the-envelope calculation of cost according to the bursty duration in a day.

4.7.4 Latency-Cost($) Trade-Off

The cost ($) of Sponge increases to the duration of using Lambda, and the cost may be higher than over-provisioning VMs when the burst duration is too long. To investigate the latency-cost trade-off, we compare the following two VM-overprovisioning approaches with Sponge in terms of latency and cost. One is **20-VMs (static)**, where 20 VMs are statically allocated without dynamic scaling, and the other one is **25-VMs (static)**, where 25 VMs are statically allocated without dynamic scaling. As the default number of VMs used in Sponge is 5, the 20-VMs and 25-VMs allocate 4× and 5× of VMs compared to Sponge, respectively.

Figure 4.13(a) illustrates the latency of 20-VMs, 25-VMs, and Sponge during bursty period. The latency of Sponge is in the middle of 20-VMs and 25-VMs, which means that Sponge outperforms over-provisioning 20 VMs. Compared to 25-VMs, which has enough resources to handle bursty loads (5× burstiness), Sponge has inherent scaling overheads due to the redirection and migration protocols. This is why the latency of Sponge is slightly higher than that of

25-VMs.

In terms of cost, Figure 4.13(b) shows the back-of-the-envelope calculation of cost according to the bursty duration in a day. For instance, 1% of bursty duration represents that bursty loads happen during $24hr * 0.01$ in a day. Basically, the cost of Sponge is smaller than others when bursty loads happen infrequently (when the duration of bursty load is less than 15%). When the bursty load frequently happens (more than 25% in Figure 4.13(b)), the cost of Sponge is higher than that of 25-VMs due to the high cost of Lambda instances. In this case, it is more beneficial to statically over-provision VM resources in terms of latency and cost.

4.8 Discussion

Fault Tolerance. The traditional approach for fault tolerance is to checkpoint and replay events using the DAG information [15, 18]. They periodically checkpoint operator state and buffer the input events in their upstream operators (or in reliable sources like Kafka [52]). Once downstream operators fail, they restore the checkpoint states and replay the buffered events from the upstream to downstream operators for state recovery. The benefit of Sponge's DAG reshaping is that the converted DAG still keeps the data dependency between operators. Therefore, even though Sponge's utility operators (RO, TO, and MO) are failed in VM or Lambda, Sponge can also use the traditional operator checkpoint and replay approach by periodically checkpointing their states and buffering input events in their upstream operators using the DAG.

Automatic Policy. Sponge provides the playground for building scaling policies that harness both VMs and Lambda instances. As we configured policies in the evaluation based on the query workload and characteristics, policy developers who have workload-specific knowledge can build their optimized policies. For example, developers dynamically pre-allocate VMs for predictable workloads, while using Lambda for unpredictable workloads. Designing an automatic policy for various workloads is an interesting future work, and we believe that Sponge will bring interesting research for automatic and optimized scaling decisions on VMs and Lambda for various queries and diverse traffic workloads.

Applicability. The R1 and R2 reshaping rules of Sponge are specific to Lambda instances, as R1 is designed for enabling data communication across Lambda instances, and R2 harnesses the warm-container property of Lambda instances. However, the R3 reshaping rule, which minimizes state migration overheads, is not specific to Lambda instances. Therefore, we can apply the rule on not only Lambda instances, but also VM instances, and we can take advantage of R3 as shown in Section 4.7.3 in different resource environments.

4.9 Related Work

Sponge integrates various techniques (routing, redirection, and merging) with a principled approach: reshaping streaming dataflows. In this section, we compare the techniques used in Sponge's reshaping rules with existing work.

Enabling Data Communication across Lambdas. Researchers have exploited the fast-to-start Lambda for various workloads such as interactive data analytics [74, 46], video analytics [3, 34], and daily applications [33]. These applications are also represented as DAGs, and shuffle operations are required between Lambda instances. Their solutions to enable data communication across Lambdas are to use additional VM relay servers [34], use HDFS in VMs [46], build an ephemeral storage service [51], and use a NAT-traversal technique [33].

The key difference between the existing work and Sponge is that Sponge focuses on streaming workloads, whereas the existing systems target batch workloads. Different from batch intermediate data that are created and consumed as chunks or objects [51], stream systems continuously process events with low latency. Therefore, using HDFS [46] or object-store [51] is not appropriate for low-latency stream event processing. Using additional VM relay servers requires additional VM resource provisioning, and NAT-traversal workaround can be prohibited [32] by cloud vendors at any time. Our DAG reshaping technique enables data communication across Lambdas preserving event-based stream processing with low latency, without requiring additional VM resources or NAT-traversal technique.

Data Redirection. Eddy [59] has proposed data routing to dynamically support multiple stream queries or adaptive queries where streaming dataflows change at runtime. Multiple stream queries and their DAGs can be merged at runtime, which requires adaptive event routing for changing data communications across operators. Elasticutor [94] has proposed rerouting data from a local to remote executor when migrating tasks between them for fast load rebalancing. Different from Eddy and Elasticutor's data rerouting, the data redirection of Sponge is not for adaptivity and task migration, but for fast scaling mechanism on Lambda. Sponge adds router operators to decouple shuffle dependencies and quickly redirects data from VM to transient operators running on Lambdas without task migration.

Reducing State Migration Overhead. Rhino [22] and ChronoStream [98] replicate states across machines to minimize state migration overheads. However, replicating and holding states requires long-running resources (like VMs).

In addition, holding states on Lambda will cause additional state recovery and cost when Lambda is reclaimed by cloud vendors. Megaphone [40] proposes fluid migration that smoothly migrates states from source to destination resources to mitigate latency spikes. However, when the bursty load is high, Megaphone still causes a large amount of state migration, which can delay the load redistribution and increase processing latencies. In contrast, Sponge prevents state migration by adding merger operators and merging states by being aware of query semantics (commutative and associative operations).

4.10 Summary

Sponge is the first work that harnesses Lambda for handling streaming bursty loads. Sponge addresses the constraint of Lambda's data communication and minimizes task migration overheads by reshaping streaming dataflows and inserting new stream operators: router operators, transient operators, and merger operators. Our evaluations on AWS EC2 and Lambda show that the reshaping of streaming dataflow is effective in significantly reducing tail latencies on Lambda compared to scaling out stream queries on VM and Lambda without reshaping when bursty loads happen.

Chapter 5

Conclusion

In this dissertation, we have presented two end-to-end optimization solutions that improve the performance of stream systems for small IoT stream and big IoT stream workloads. Specifically, Pluto optimizes both the query submission and execution layer, to minimize query submission overhead and efficiently share system resources across small IoT queries. Sponge optimizes both the query compilation, execution layer, and resource acquisition layer, to quickly handle sudden bursty loads of big stream queries and enable fast scaling on Lambda. Evaluations show that these end-to-end optimizations are effective in handling small and big IoT stream workloads and significantly improves system throughput and latencies. We believe that these optimizations techniques can be applied not only on the cloud backend servers and Lambda, but also on edge devices and VMs for optimizing various stream workloads.

Bibliography

[1] 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. *VLDB Journal*, 6(11):1033–1044, 2013.

[2] Angelsense gps wearables. `https://www.angelsense.com/gps-wearables/`.

[3] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter. Sprocket: A serverless video processing framework. In *SocC*, pages 263–274, 2018.

[4] Apple health. https://www.apple.com//ios/health/.

[5] Iot cloud platform — samsung artik cloud services. `https://artik.cloud/`.

[6] Aws lambda. `https://aws.amazon.com/lambda`.

[7] Aws internet of things. `https://aws.amazon.com/iot/?nc1=h_ls`.

[8] Azure function. `https://docs.microsoft.com/en-us/azure/azure-functions/`.

[9] Azure iot suite. `https://www.microsoft.com/en-us/cloud-platform/internet-of-things-azure-iot-suite`.

[10] I. Bastys, M. Balliu, and A. Sabelfeld. If this then what? controlling flows in iot apps. In *ACM SIGSAC*, pages 1102–1119, 2018.

[11] Apache beam. `https://beam.apache.org/`.

[12] Beam side input pattern. `https://beam.apache.org/documentation/patterns/side-inputs/`.

[13] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, N. Tawbi, et al. Static detection of malicious code in executable programs. *Int. J. of Req. Eng*, 2001(184-189):79, 2001.

[14] Aws sdk for python (boto3). `https://boto3.amazonaws.com/v1/documentation/api/latest/index.html`.

[15] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in apache flink®: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, 2017.

[16] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[17] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, 1999.

[18] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *ACM SIGMOD*, 2013.

[19] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *ACM SIGMOD*, pages 668–668, 2003.

[20] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. Technical report, WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES, 2006.

[21] Stream processing with iot data: Challenges, best practices, and techniques. `https://www.confluent.io/blog/stream-processing-iot-data-best-practices-and-techniques/`.

[22] B. Del Monte, S. Zeuch, T. Rabl, and V. Markl. Rhino: Efficient management of very large distributed state for stream processing engines. In *ACM SIGMOD*, page 2471–2486, 2020.

[23] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.

[24] R. G. Dimitri Bertsekas. *Data Networks*. Prentice Hall, 2nd edition, 1992.

[25] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Multi-query optimization for sketch-based estimation. *Information Systems*, 34(2):209–230, 2009.

[26] Emq: The massively scalable mqtt broker for iot and mobile applications. `http://www.emqtt.io/`.

[27] Aws case study: Ems. `https://https://aws.amazon.com/solutions/case-studies/ems/`.

[28] H. Fernandez, G. Pierre, and T. Kielmann. Autoscaling web applications in heterogeneous cloud infrastructures. In *IEEE ICCE*, pages 195–204, 2014.

[29] H. Fernandez, G. Pierre, and T. Kielmann. Autoscaling web applications in heterogeneous cloud infrastructures. In *2014 IEEE International Conference on Cloud Engineering*, pages 195–204, 2014.

[30] Fitbit. `https://www.fitbit.com/`.

[31] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment*, 10(12):1825–1836, 2017.

[32] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-peer communication across network address translators. In *ATC*, pages 179–192, 2005.

[33] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *ATC*, pages 475–488, 2019.

[34] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *NSDI*, pages 363–376, 2017.

[35] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee. Edgewise: A better stream processing engine for the edge. In *ATC*, pages 929–946, 2019.

[36] Garmin - heart rate monitor. `https://buy.garmin.com/en-US/US/p/10996`.

[37] Gartner says 5.8 billion enterprise and automotive iot endpoints will be in use in 2020. `https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io`.

[38] Google cloud function. `https://cloud.google.com/functions/docs/`.

[39] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12), 2012.

[40] M. Hoffmann, A. Lattuada, and F. McSherry. Megaphone: latency-conscious state migration for distributed streaming dataflows. *Proceedings of the VLDB Endowment*, 12(9):1002–1015, 2019.

[41] M. Hong, A. J. Demers, J. E. Gehrke, C. Koch, M. Riedewald, and W. M. White. Massively multi-query join processing in publish/subscribe systems. In *ACM SIGMOD*, pages 761–772, 2007.

[42] Ibm cloud functions. `https://console.bluemix.net/openwhisk/`.

[43] Ifttt. `https://ifttt.com/`.

[44] S. Islam, S. Venugopal, and A. Liu. Evaluating the impact of fine-scale burstiness on cloud elasticity. In *SoCC*, pages 250–261, 2015.

[45] S. Islam, S. Venugopal, and A. Liu. Evaluating the impact of fine-scale burstiness on cloud elasticity. In *ACM SoCC*, page 250–261, 2015.

[46] A. Jain, A. F. Baarzi, G. Kesidis, B. Urgaonkar, N. Alfares, and M. Kandemir. Splitserve: Efficiently splitting apache spark jobs across faas and iaas. In *Middleware*, page 236–250, 2020.

[47] H. S. Jang, H. Jin, B. C. Jung, and T. Q. S. Quek. Resource-optimized recursive access class barring for bursty traffic in cellular iot networks. *IEEE Internet of Things Journal*, pages 1–1, 2021.

[48] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *OSDI*, pages 783–798, 2018.

[49] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream data processing systems. In *ICDE*, pages 1507–1518, 2018.

[50] Aws case study: Kemppi. `https://aws.amazon.com/solutions/case-studies/kemppi/`.

[51] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*, pages 427–444, 2018.

[52] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *NetDB*, 2011.

[53] 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 *ACM SIGMOD*, pages 239–250, 2015.

[54] Aws lambda limits. `https://docs.aws.amazon.com/lambda/latest/dg/limits.html`.

[55] J. R. Larus and M. Parkes. Using cohort scheduling to enhance server performance. In *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 182–187, 2001.

[56] S. Li, P. Gerver, J. MacMillan, D. Debrunner, W. Marshall, and K.-L. Wu. Challenges and experiences in building an efficient apache beam runner for ibm streams. *Proceedings of the VLDB Endowment*, 11(12):1742–1754, 2018.

[57] S. Luo, X. Chen, and Z. Zhou. F3c: Fog-enabled joint computation, communication and caching resource sharing for energy-efficient iot data stream processing. In *ICDCS*, pages 1019–1028, 2019.

[58] S. Luo, B. Kao, G. Li, J. Hu, R. Cheng, and Y. Zheng. Toain: a throughput optimizing adaptive index for answering dynamic k nn queries on road networks. *Proceedings of the VLDB Endowment*, 11(5):594–606, 2018.

[59] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *ACM SIGMOD*, pages 49–60, 2002.

[60] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa, S. Dhulipalla, and S. Rao. Chi: A scalable and programmable control plane for distributed stream processing systems. *Proceedings of the VLDB Endowment*, pages 1303–1316, 2018.

[61] F. Metzger, T. Hossfeld, A. Bauer, S. Kounev, and P. Heegaard. Modeling of aggregated iot traffic and its application to an iot cloud. *Proceedings of the IEEE*, 107:679 – 694, 2019.

[62] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Injecting realistic burstiness to a traditional client-server benchmark. In *ICAC*, pages 149–158, 2009.

[63] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin. Streambox: Modern stream processing on a multicore machine. In *USENIX ATC*, pages 617–629, 2017.

[64] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[65] M. MOGHADDAM, A. SILVA, D. CLEWLEY, R. AKBAR, S. HUSSAINI, J. Whitcomb, R. DEVARAKONDA, R. Shrestha, R. COOK, G. PRAKASH, S. SANTHANA VANNAN, and A. BOYER. Soil moisture profiles and temperature data from soilscape sites, usa. `https://daac.ornl.gov/cgi-bin/dsviewer.pl?ds_id=1339`, 2016.

[66] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas. Predicting taxi–passenger demand using streaming data. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1393–1402, 2013.

[67] Mqtt.org. `http://mqtt.org/`.

[68] I. Müller, R. Marroquín, and G. Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *ACM SIGMOD*, pages 115–130, 2020.

[69] Nest app. `https://nest.com/app/`.

[70] Nexmark benchmark suite. https://beam.apache.org/documentation/sdks/java/testing/nexmark/.

[71] Pipelinedb–the streaming sql database. https://www.pipelinedb.com/.

[72] Postgresql. https://www.postgresql.org/.

[73] Household power consumption dataset. https://data.world/databeats/household-power-consumption.

[74] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *NSDI*, pages 193–206, 2019.

[75] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: Reliable stream computation in the cloud. In *EuroSys*, pages 1–14, 2013.

[76] Aws case study: Rachio. https://aws.amazon.com/solutions/case-studies/rachio/.

[77] S. H. Rastegar, A. Abbasfar, and V. Shah-Mansouri. Rule caching in sdn-enabled base stations supporting massive iot devices with bursty traffic. *IEEE Internet of Things Journal*, 7(9):8917–8931, 2020.

[78] REFIT Smart Home dataset. https://figshare.com/articles/REFIT_Smart_Home_dataset/2070091.

[79] A. Reiss and D. Stricker. Creating and benchmarking a new dataset for physical activity monitoring. In *ACM PETRA*, pages 40:1–40:8, 2012.

[80] Rfc-7228: Terminology for constrainted-node networks. https://datatracker.ietf.org/doc/html/rfc7228.

[81] B. Robinson, R. Power, and M. Cameron. A sensitive twitter earthquake detector. In *WWW Companion*, pages 999–1002, 2013.

[82] Samsung smartthings: Smart home app for home automation. `https://www.samsung.com/us/explore/smartthings/`.

[83] Scriptr: Agile iot application platform. `https://www.scriptr.io/`.

[84] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *SoCC*, page 5, 2011.

[85] Tang. `https://reef.apache.org/tang.html`.

[86] Temboo: Tools for digital transformation. `https://temboo.com/`.

[87] J. Thomas, P. Hanrahan, and M. Zaharia. Fleet: A framework for massively parallel streaming on fpgas. In *ASPLOS*, page 639–651, 2020.

[88] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *ACM SIGMOD*, pages 147–156, 2014.

[89] B. Trushkowsky, P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *FAST*, pages 163–176, 2011.

[90] Uber. `https://www.uber.com`.

[91] L. van Bommel and C. N. Johnson. Where do livestock guardian dogs go? movement patterns of free-ranging maremma sheepdogs. *PLOS ONE*, 9(10):1–12, 2014.

[92] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, pages 374–389, 2017.

[93] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng. Infinicache: Exploiting ephemeral serverless functions to build a cost-effective memory cache. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 267–281, 2020.

[94] L. Wang, T. Z. J. Fu, R. T. B. Ma, M. Winslett, and Z. Zhang. Elasticutor: Rapid elasticity for realtime stateful stream processing. In *ACM SIGMOD*, page 573–588, 2019.

[95] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP*, pages 230–243, 2001.

[96] Z. Wen, D. L. Quoc, P. Bhatotia, R. Chen, and M. Lee. Approxiot: Approximate analytics for edge computing. In *ICDCS*, pages 411–421, 2018.

[97] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *ACM SIGMOD*, pages 407–418, 2006.

[98] Y. Wu and K. Tan. Chronostream: Elastic stateful stream computation in the cloud. In *ICDE*, pages 723–734, 2015.

[99] W. Xie, F. Zhu, J. Jiang, E. Lim, and K. Wang. Topicsketch: Real-time bursty topic detection from twitter. *IEEE Transactions on Knowledge and Data Engineering*, 28(8):2216–2229, 2016.

[100] D. Xu, X. Liu, and A. V. Vasilakos. Traffic-aware resource provisioning for distributed clouds. *IEEE Cloud Computing*, 2(1):30–39, 2015.

[101] Y. Yang, J. Eo, G.-W. Kim, J. Y. Kim, S. Lee, J. Seo, W. W. Song, and B.-G. Chun. Apache nemo: A framework for building distributed dataflow optimization policies. In *ATC*, pages 177–190, 2019.

[102] N. Yoshimura, T. Maekawa, D. Amagata, and T. Hara. Upsampling inertial sensor data from wearable smart devices using neural networks. In *ICDCS*, pages 1983–1993, 2019.

[103] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[104] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438, 2013.

[105] C. Zhang, M. Yu, W. Wang, and F. Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *ATC*, pages 1049–1062, 2019.

[106] F. Zhang, L. Yang, S. Zhang, B. He, W. Lu, and X. Du. Finestream: Fine-grained window-based stream processing on cpu-gpu integrated architectures. In *ATC*, pages 633–647, 2020.

[107] S. Zhang, H. T. Vo, D. Dahlmeier, and B. He. Multi-query optimization for complex event processing in sap esp. In *ICDE*, pages 1213–1224, 2017.

[108] Y. Zhang, R. Chen, and H. Chen. Sub-millisecond stateful stream querying over fast-evolving linked data. In *SOSP*, pages 614–630, 2017.

[109] Y. Zheng, X. Xie, and W.-Y. Ma. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data Eng. Bull.*, 33(2):32–39, 2010.

초록

다양한 IoT 디바이스로부터 많은 양의 데이터 스트림들이 생성되면서, 크게 두 가지 타입의 스트림 쿼리가 클라우드에서 수행된다. 첫째로는 작은-IoT 스트림 쿼리이며, 하나의 스트림 쿼리가 적은 양의 IoT 데이터 스트림을 처리하고 많은 수의 작은 스트림 쿼리들이 존재한다. 두번째로는 큰-IoT 스트림 쿼리이며, 하나의 스트림 쿼리가 많은 양의, 급격히 증가하는 IoT 데이터 스트림들을 처리한다. 하지만, 기존 연구와 스트림 시스템에서는 쿼리 수행, 제출, 컴파일러, 및 리소스 확보 레이어가 이러한 워크로드에 최적화되어 있지 않아서 작은-IoT 및 큰-IoT 스트림 쿼리를 효율적으로 처리하지 못한다.

이 논문에서는 작은-IoT 및 큰-IoT 스트림 쿼리 워크로드를 최적화하기 위한 엔드-투-엔드 최적화 기법을 소개한다. 첫번째로, 많은 수의 작은-IoT 스트림 쿼리를 처리하기 위해, 쿼리 제출과 수행 레이어를 최적화 하는 기법인 IoT 특성 기반 최적화를 수행한다. 쿼리 제출과 코드 등록을 분리하고, 이를 위한 새로운 API를 제공함으로써, 쿼리 제출에서의 오버헤드를 줄이고 쿼리 수행에서 IoT 특성 기반으로 리소스를 공유함으로써 오버헤드를 줄인다. 두번째로, 큰-IoT 스트림 쿼리에서 급격히 증가하는 로드를 빠르게 처리하기 위해, 쿼리 컴파일러, 수행, 및 리소스 확보 레이어 최적화를 수행한다. 새로운 클라우드 컴퓨팅 리소스인 람다를 활용하여 빠르게 리소스를 확보하고, 람다의 제한된 리소스에서 스케일-아웃 오버헤드를 줄이기 위해 스트림 데이터플로우를 바꿈으로써 큰-IoT 스트림 쿼리의 작업량을 빠르게 람다로 옮긴다.

최적화 기법의 효과를 보여주기 위해, 이 논문에서는 두가지 시스템-Pluto 와 Sponge-을 개발하였다. 실험을 통해서, 각 최적화 기법을 적용한 결과 기존 시스템 대비 처리량을 크게 향상시켰으며, 지연시간을 최소화하는 것을 확인하였다.

주요어: 스트림 처리, 분산 데이터 처리, IoT, 클라우드, 람다, 서버리스

학번: 2014-22686