Chimera: Large-scale Data Collection and Processing

JIAN GONG

A THESIS SUBMITTED FOR THE DEGREE OF MASTER OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
August 2011

Acknowledgments

I hereby give my heartiest thankness to my supervisor, Prof. Ben Leong, who has been offering me great guidance and help for this work. The research project would not have been possible without his support and encouragement. I have learned a lot from his advices not only in academic study but also in philosophy of life.

I also thank my friends for their help. My sincere gratitude goes to Ali Razeen, who offered me great help and invaluable suggestions for my thesis. I thank Daryl Seah, who helped me and inspired me in the thesis writing and project implementation. I also thank Wang Wei, Xu Yin, Leong Wai Kay, Yu Guoqing and Wang Youming, we have spent a great time together as lab mates and fellow apprentices.

I thank my parents who always offer me unwavering support. My gratitude goes to all the friends who accompany me during my study in National University of Singapore.

Table of Contents

List of Figures

Abstract

Companies depend on the analysis of data collected by their applications and services to improve their products. With the rise of large online services, massive amounts of data are being produced. Known as *Big Data*, these datasets are expected to reach 32.2ZB globally in 2011. As traditional tools are unable to process Big Data in a timely fashion, a new paradigm of handling Big Data has been proposed. Known as *Stream Processing*, there has been a lot of work on this paradigm from both the academic and commercial worlds, leading to a large number of stream processing systems with varying designs. They can be broadly classified into two categories: centralized or distributed. The former processes data atomically while the latter breaks up a processing operation, deploys the sub-operations across multiple nodes, and combines the output from those nodes to produce the final results.

In this thesis, we attempt to understand the limits of a centralized stream processing system when it is under real-world workloads. We do this by evaluating Esper, an open-source centralized stream processor, with data from a game deployed on Facebook. We also developed our own distributed stream processing system, called Chimera, and compared Esper with it. This is to understand how much more performance we can gain if we process the same data with a distributed system.

We found that Esper's performance varies widely depending on the kind of queries given to it. While, the performance is very good when the queries are simple, it quickly starts to deteriorate when the queries become complex. Therefore, although a centralized system might seem attractive due to lower costs in deployment, developers might be better off using a distributed system if they process data in a complex manner. We also found that a distributed system may perform better than Esper, even when both of them are deployed on a single machine. This is because the distributed system may be simpler in design compared to Esper. Therefore, if developers do not need the various features offered by Esper, using a simpler stream processing system would provide them with better performance.

Chapter 1

Introduction

Companies depend on the data produced by their applications and services to understand how their products can be improved. By analyzing this data, they can identify important trends and properties about their offerings and take any required action. With the increasing popularity of the Internet and the rise of large Internet services, such as Facebook and Twitter, massive amounts of data are being generated and tools traditionally used to analyze such data are becoming inadequate. Termed as *Big Data*, the total size of these datasets is expected to reach 34.2ZB (zetabytes) globally in 2011 [20].

In response to the problem of managing and analyzing Big Data, much work has been done in the area of stream processing. Instead of storing datasets in a database and running time-consuming queries on them, stream processing offers the ability to get answers to queries in real-time by processing data as they arrive. To use stream processing, developers are required to restructure their applications to generate data when important events occur and send them to a stream processing system. For example, when a user signs up for an account on Facebook, an event *AccountCreated* may be generated and properties such as the user's details could be associated with that event. A stream processing engine would then be used to help answer, in real-time, queries such as: "On average, in a 24-hour window, how many new account creations are there?".

A number of stream processing systems have been proposed in the literature [9, 12, 8, 15, 26, 19, 13, 10, 16]. Many still have been developed in the commercial world [7, 4, 1, 6, 2]. There are many variations to these systems and they offer different capabilities. However, they can be broadly classified as being either centralized or distributed systems. The difference between them is on how they execute a stream processing operation. Suppose there is an

operation that comprises of two steps: a filtering step to remove unwanted data, and an aggregation step to combine the remaining data. In a centralized system, both steps would be carried out in a single instance of the stream processor. On the other hand, in a distributed system, a set of nodes would execute the first step and pass the resulting data to another set of nodes, which would then execute the second step.

As those who need to handle Big Data have different requirements and as there is no "one-size-fits-all" stream processing system, a decision has to be made whether to use a centralized system or a distributed system. The cost of deploying the stream processing system must also be factored into the decision. For example, a company may have an application that generates events on a rate, while large enough to warrant the use of a centralized stream processing system, still small enough to not justify the cost of deploying a distributed system. *Hence, in this thesis, we evaluate the limits of centralized systems.* This helps identify the instances when it is better to use a distributed system.

In particular, we evaluated Esper [2], a widely known centralized stream processing system, and compared it against Chimera, a distributed stream processing system that we developed. We found that Esper's performance varies greatly depending on the kind of queries it is executing. If the queries are very complex, the rate at which Esper can process events would be low. This makes it easy for distributed systems to outperform Esper, even when sources generate events at low rates. Furthermore, we found that Chimera can perform better than Esper even when they are both deployed on a single machine. This is because Esper's design is more sophisticated than Chimera's. If developers do not require the various features offered by Esper, they would obtain better performance by switching to a simpler stream processing system.

1.1 Motivation

This study is motivated by the work on TankVille [22]. TankVille is a Facebook game that is used to evaluate another research project. When a user launches the game, data is collected to study both the attractiveness of the game and the underlying research system. After TankVille was launched, its developers found that running SQL queries on the database, where all the TankVille data was sent to, was too time-consuming and did not yield timely answers. There was a need for them to use a stream processing system. However, they were unable to decide on a system to use as there were many of them and their differences were not obvious.

There have been two previous studies that evaluated Esper [18, 23]. However, their evaluation was based on very generic queries. In our work, we use queries related to TankVille to understand how Esper would perform under real-world conditions. We then compare our findings with the previous studies to make inferences on Esper.

1.2 Our Approach

As Esper is open-source, well-known, and widely adopted, we take it to be representative of centralized stream processing systems in general and base our evaluation off it. We compared Esper against Chimera, a distributed stream processing system that we developed. We did not use an existing distributed system as previously proposed academic systems are no longer in active development. Even though their source code is publicly available, a significant amount of time and resources would be needed to understand their code, fix outstanding bugs, and adapt the systems for our use.

As we wanted to evaluate Esper with real-world queries, we attempt to answer questions that the TankVille developers had. In particular, we attempt to answer which game map in TankVille is most popular, so as to identify the attractive aspects of TankVille. However, instead of using live data from TankVille, we built a load generator to generate the same events that TankVille would. This was done for two reasons: (i) in a live deployment of the game, we cannot control the rate at which events are produced, making it difficult to run controlled experiments, and (ii) TankVille is currently inactive as it is undergoing upgrading due to changes in the Facebook API.

1.3 Organization

This thesis is organized as follows: in Chapter 2, we give an overview of stream processing, present related works, and describe Esper in detail. We present the design and implementation of Chimera, the distributed stream processing system that we developed, in Chapter 3. Our evaluation of Esper is discussed in Chapter 4 and finally, we conclude in Chapter 5.

Chapter 2

Related Work

In this chapter, we first give an overview of stream processing and introduce the various terminologies used. Next, we give an overview of several stream processing systems and describe Esper in some detail. Finally, we discuss the performance studies that have been conducted on these stream processing systems.

2.1 Overview of Stream Processing

In this section, we clarify some of the terminologies used in the area of stream processing.

The basic unit in stream processing is the *event*. An event refers to a system message representing some real world occurrence. Each event would have a set of attributes describing its properties. There are two types of events: simple and complex. A simple event corresponds directly to some basic fact that can be captured by an application easily while a complex event is one that is inferred from multiple simple events. For example, a game application may generate the simple event *(PlayerKill X Y)* to refer to the fact that player X has killed player Y. (Note that X and Y are attributes of the event). Suppose that the game keeps generating the events *(PlayerKill A B)* and *(PlayerKill B A)*. If these two events are generated very frequently, then we can infer that players A and B are rivals, and generate the complex event *(AreRivals A B)*.

An application that generates a continuous stream of events is said to be a *source* of an *event stream*. Event streams are processed by stream processing systems, which can refer to either *event stream processing* systems or *complex event processing* systems. The former is concerned mainly with processing streams of simple events and in doing simple mathematical computations such as `SUM`, `AVG`, or `MAX`. For example, given a stream of events

representing withdrawals in a bank account, the total sum of money withdrawn in a day can be calculated easily with a event stream processing system. Complex event processing systems have greater features and also provide developers the tools to correlate different kinds of events to generate complex events. In recent years, most stream processing systems have the ability to do complex event processing.

2.2 Existing Stream Processing Systems

2.2.1 Aurora

Aurora [9] is a stream processing system that receives streams of data from different sources, runs some operation on those streams and produces new streams of data as output. These new streams can then be processed further, be sent to some application, or be stored in a database. A developer would construct the stream processing operations (designated as *queries* in the Aurora terminology) by using seven built-in primitives (such as *filter* and *union*) and create a processing path that will transfer the input stream into a desired output stream. Aurora also has a quality-of-service (QoS) mechanism built in. When it detects that a system is overloaded, it starts dropping data from the streams so as to maintain its processing rate, while also trying to maintain accuracy of results.

2.2.2 Medusa and Borealis

Medusa [12] is a distributed stream processing system that has multiple nodes running Aurora. It manages loads using an economic principle. A node with heavy loads considers its jobs to cost high and unprofitable to complete. Therefore, it finds other nodes that are not as loaded and attempts to "sell" its jobs to them. These nodes will have a lower cost in processing the jobs and thus, will make a profit by "selling" the results to the consumer (the system egress point). All nodes in Medusa are profit-seeking and therefore, the system distributes load effectively.

Borealis [8] is another distributed stream processing system that builds upon Aurora. Each node in the Borealis system will run a Borealis server, which has improvements over Aurora. Namely, it supports dynamic query modifications, which allows one to redefine the operations in a processing path while the system is active. It also supports dynamic revision of query results, which can improve results previously produced when a new fact is available. For example, a source may send an event claiming that the data it produced hours ago were

inaccurate by some margin. In such a case, there is a need to revise the previous results.

2.2.3 TelegraphCQ

TelegraphCQ [15] combines stream processing capabilities with relational database management capabilities. By modifying the architecture of PostgreSQL, an open source database management system, TelegraphCQ allows SQL-like queries to be continuously executed over streaming data, providing results as data arrives. Based on the given query, the system builds up a set of operators that can pipeline incoming data to accelerate the processing. Their modifications to PostgreSQL allows the query processing engine to accept data in a streaming manner.

2.2.4 SASE

One of the earliest works on complex event processing is SASE [26]. It provides a query language with which a user can detect complex patterns in the incoming event streams by correlating the events. Users can also specify time windows in their queries so as to concentrate only on timely data. The authors compared their work to TelegraphCQ and demonstrated that the relational stream processing model in TelegraphCQ is not suited for complex event processing.

2.2.5 Cayuga

Cayuga [19] is another event processing system that supports its own query language. The novelty here is that a query in Cayuga can be expressed as a nondeterministic finite state automaton (NFA) with self-loops. Each state in the automaton is assigned a fixed relational schema. An edge $< S, \theta, f >$ between states P and Q identifies an input stream (S), a predicate (θ) over schema(P) $\times$ schema(S), and a function (f) mapping θ into schema(Q). If an event e arrives at the state P of the NFA and $\theta(schema(P), e)$ is satisfied, then the automaton transitions to state Q, with schema(Q) becoming $f(schema(P), e)$. Expressing queries in this way allows Cayuga to use NFA to process events in complex ways. For example, the use of self-loops in the NFA will allow a query to use its output as an input to itself, which allows the query to be recursive.

2.2.6 Microsoft CEP

Microsoft has also developed a complex event processing engine which they call CEP Server [10]. This is based on their earlier work, CEDR (Complex Event Detection and Response) [13] project. Amongst other things, CEDR can handle events that do not arrive in-order. For example, a query may depend on an event A and B, and either event may arrive first. CEDR handles such scenarios by requiring each event to have two timestamps, indicating the interval for which the event is said to be valid. When CEDR receives an event, it will buffer the event until the event is either processed or until the event's lifetime expires, whichever occurs first. Microsoft has deployed its CEP server for its own use. To achieve scalability, it supports stream partitioning and query partitioning. The CEP system runs multiple instances of the servers, partitions an incoming stream into sub-streams and sends each sub-stream to a different server. Queries are also partitioned in a similar manner.

2.2.7 MapReduce and MapReduce Online

MapReduce [17] is a distributed programming model proposed by Google. It runs batch processing on large amounts of data, e.g. crawled documents from the Internet. By defining the two functions, `map` and `reduce`, MapReduce is able to distribute a computation task across thousands of machines to process massive amounts of data in a reasonable time. This distribution is similar to parallel computing, where the same computations are performed on different datasets on each CPU. MapReduce provides an abstraction that allows distributed computing while hiding the details of parallelization, load balancing and data distribution.

To use MapReduce, a user has to write the functions `map` and `reduce`. `map` takes as input a function and a sequence of values from raw data, and produces a set of intermediate key-value pairs. The MapReduce library groups together all intermediate values associated with the same key and passes them to the `reduce` function. The `reduce` function accepts an intermediate key and a set of values for that key, then merges these values to form a smaller set of values. Data may go through multiple phases of `map` and `reduce` before reaching the final desired format.

The contribution of MapReduce is a simple and powerful interface enabling automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Recently, there has been some work in trying to use MapReduce for real-time data analysis. MapReduce Online [16] is one such work that attempts to process streaming data with

MapReduce. In this system, when the `map` function produces outputs, they are sent directly to the `reduce` function in addition to being saved to disk. The `reduce` function will work on the outputs from `map` immediately to produce early results of the desired computation. When the nodes in the system complete the map phase, the reduce phase will be executed again to get the final results. In this manner, the system provides approximate results when it is busy processing the input data, and provides the final results when all the data has been processed.

2.2.8 Dryad

Isard et al. proposed a distributed framework similar to MapReduce called Dryad [21]. Just like MapReduce, it allows parallel computation on massive amounts of data. However, the authors claim that Dryad is more flexible than MapReduce as it permits multiple phases, not just `map` and `reduce`. This allows developers to solve problems that cannot be converted into the map and reduce phases naturally. Dryad cannot be used to process data in real-time as it is still a batch processing system. However, we use their ideas of having multiple phases to design Chimera.

2.3 Description of Esper

Here, we give a detailed introduction to Esper [2]. Esper is a state-of-the-art complex event processing engine and is maintained by EsperTech. They provide an open-source version of Esper, written in Java, for academic use and also a commercial version of Esper with more features. To use Esper, developers will create their own application and link it with the Esper library. The library will handle the actual processing of the events and the production of outputs, but the developer has the responsibility of connecting the application to the appropriate event stream sources and in passing the events to Esper.

Figure 2.1 is a architectural diagram of Esper (taken from official Esper website). Incoming events are processed according to the queries registered in the system. The results are wrapped as POJOs (Plain Old Java Objects) and is sent to the result subscribers. Esper also provides a layer to store the results into a database. This allows the construction of queries that rely on historical data.

Events in Esper can be represented in three ways: (i) a POJO, (ii) a Java Map object with key-value pairs where the key is the name of the attribute and the value is the value of the

Figure 2.1: Esper architectural diagram (taken from the Esper website)

attribute, and (iii) an XML document object. An SQL-like query language is provided to detect different events (or patterns of events), and to take the appropriate processing action. The query results can either be automatically sent to a subscriber, or the developer can poll the Esper engine and see if new results are available.

2.4 Evaluation on Stream Processing Systems

Given that the different stream processing systems proposed previously work differently, several evaluations have been done to understand their performance.

One of the earliest studies on Esper was conducted by Dekker [18]. He compared Esper and StreamCruncher [5], another open-source centralized stream processing system. The focus of his work was on testing the complex event processing capabilities of both systems by running six different queries, each designed to produce a result by correlating different events. He shows that Esper performs consistently better than StreamCruncher, and gives good throughput. However, his study was done in 2007 and since then, Esper has gone through significant upgrades. Therefore, we do not compare our results with his results.

Mendes et al. did another evaluation and compared Esper with two other commercial products [23]. Due to licensing issues, they did not name any of the systems in their evaluation and simply referred to them as *X*, *Y*, and *Z*. However, one can infer that *Y* refers to Esper as the authors specifically mentioned that Esper is the only open-source product of the three stream processing systems and in their evaluation, they stated that they "examined Y's open-source code" to study its behaviour. Their results show that Esper's performance varies greatly depending on the kind of queries that are executed. For example, a simple *SELECT*

query can process events at a rate of 500K per second while a query that performs SQL-like joins may process only at a rate of 50K per second. Their evaluation is based on FINCoS framework [24], which is a set of tools designed to benchmark complex event processing engines. Instead of using their benchmarks, which are based on a set of generic queries, we evaluated Esper with our own set of queries based on TankVille. This allows us to understand how Esper performs when it is used to answer actual application queries.

Arasu et al. [11] compared Aurora against a relational database configured to process stream data inputs. They used the Linear Road project [3], another benchmark tool for stream data processing. By measuring the response time and the throughput of the system, the benchmark tool is able to identify the system more suitable for processing streaming data. According to the results, under the same response time requirement, Aurora achieves a throughput that is greater than 5 times of the database. The goal of their work is to confirm that stream systems perform better than databases in processing streaming data.

Tucker et al. built NEXMark [25], a benchmark for stream processing built as an online auction system. At any moment during the simulation, new users can create an account with the system, bid on any of the hundreds of open auctions, or auction new items. NEXMarks evaluates how a stream processing system can handle queries over all these events. This benchmark is still under construction and is not yet used to evaluate stream processing systems.

Chapter 3

Chimera Design and Implementation

To evaluate Esper, we developed our own distributed stream processing engine called *Chimera*. Chimera's design is inspired by both MapReduce and Dryad. It allows developers to define their own operations, organize a layered structure of nodes to process the data in a parallel manner according to the defined operations. Chimera requires the developer to only define the task to be processed. It transparently handles the details of distributed processing, such as monitoring the status of the machines in the system, the offloading of processing jobs to different machines depending on their availability, and the distribution of data between different nodes. This improves the usability of Chimera.

In Chimera, we use text string to represent an event. These strings are formatted as a comma-separated key value pairs. For example, the string $< key_1 = value_1, key_2 = value_2, \ldots, key_n = value_n >$ will represent an event. We use strings representation for events as it simplifies the implementation of Chimera.

The architecture of Chimera is illustrated in Figure 3.1. There are four kinds of nodes in Chimera: (i) Collectors, (ii) Workers, (iii) Sinks, and (iv) the Master. The role of the Collectors is to receive events from various sources and pass them to the Workers. The Workers would then process the events according to the user-defined operations and according to how the Workers are structured in the layer. The results are then sent to the Sink node, which can either provide the data to the developer in real-time or simply store it in a traditional database. The Master node is used to manage the previous three types of nodes, and ensures that they process the developer's tasks.

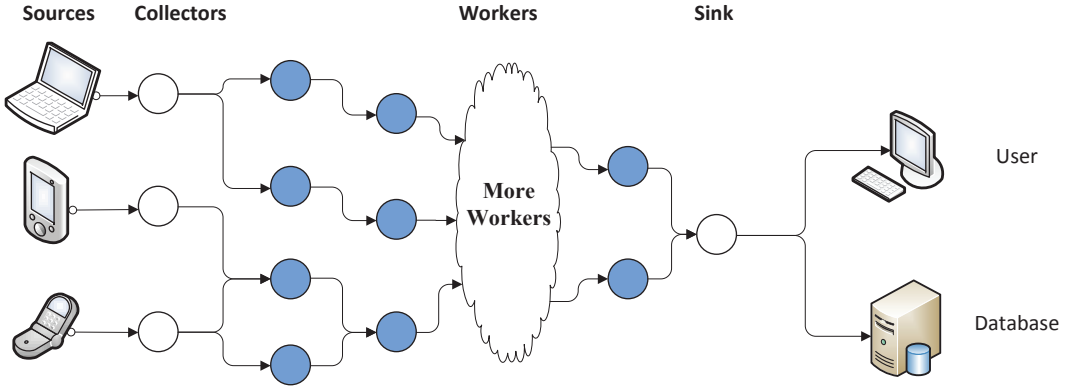

Figure 3.1: System Architecture of Chimera.

We validated our implementation by running processing tasks where the source events were saved in their raw form before they were passed to Chimera. Next, we manually processed the raw source events and compared the results obtained with that from Chimera. The two results turned out to be consistent. Further, we ran the same tasks with Esper and also found the results to be consistent. Therefore, we concluded that our Chimera implementation is correct.

We now proceed to describe the design of Chimera and the design of each node type in greater detail.

3.1 Collector Nodes

Different event sources (such as desktop PCs and mobile devices) will send events to Chimera by using an API exposed by the Collector nodes. In our current implementation, the API is provided as HTTP webservice call. Collectors would then stream these events to the Worker nodes so that they can be processed. As the Collector has few responsibilities, its design is simple.

3.2 Worker Nodes

Workers are nodes that performs actual processing of events in Chimera. They are structured in a topology to process events in layers. For instance, the first layer of Workers might transform the event stream from sources into some intermediate form. A second layer of Workers may process this intermediate form of data into yet another form. This can continue until

the events reach the final layer of Workers, where the expected results are produced. Each Worker is structured as three parts: (i) receiver, (ii) operator, and (iii) sender.

Receiver

The receiver manages all incoming connections from upstream Workers. It monitors the rate of incoming streams and the rate of processing. When the rate of incoming events overwhelms the processing capacity, the Worker will send the Master a warning message, to ask for it control the rate of upstream Workers.

Operator

The operator processes the events based on the user-defined operations. A Worker will configure its operator after receiving instructions from the Master on the operation it should execute.

Sender

The sender sends the output stream from the operator to the nodes in the next layer of the topology. It also monitors the rate at which it is sending events, R_s. If the Worker receives a rate-control message from the Master, it will adjust its sending rate so as to prevent the downstream nodes from being overwhelmed. The rate control message contains R_p, which is the processing rate of the downstream choked Worker. The Worker would then drop the events produced with a probability P_d based on the following equation:

$$P_d = \frac{R_s - R_p}{R_s} * 100\% \tag{3.1}$$

Although the dropping of events may affect accuracy, this feature is useful in situations where bursts at the sources occur, in terms of event generation rates. If these bursts is beyond the processing capacity of the Chimera system and if there is no rate control, the time taken to process events would increase. Consequently, the timeliness and "freshness" of the results would be affected. Developers can switch off this feature if they prefer to have accurate results at the cost of slower results.

3.3 Sink Nodes

The Sink is the egress point of the layered structure processing network. It collects results from the last layer of Workers, does some necessary operations and returns the final results to developers in real-time. Developers can also implement a Sink operation to store the results to a database for future query. If a Chimera system is configured with many Workers but just one Sink, the Sink might become the processing bottleneck as it may not be able to collect the results quickly enough.

To address this issue, an additional layer of nodes may be inserted between the last layer of Workers and the Sink. The job of these nodes would simply be to collect and do partial merging of the results from the Workers, and send them to the Sink. In this manner, the Sink handles inputs from a lesser number of nodes and it will not be overwhelmed. Note that this is similar to the reduce phase in MapReduce.

3.4 The Master Node

The Master node controls the Collectors, Workers, and Sinks, when executing a developer's task. It is responsible for arranging the topology of the nodes, including the organization of the Workers' layers, and manages the communication between the various nodes. It is also responsible for specifying the operations that the Workers and Sinks need to perform.

Machine Management

When a machine is added to the Chimera system, it will register with the Master and indicate the computing resources it has, such as the number of CPU cores available. This informs the Master that it has additional computing resources available and it may send the machine some processing task. The machines are also required to periodically send heartbeat messages to the Master. If the Master detects that a particular machine has not sent this heartbeat message for some time, it will mark the machine as unavailable and will not deploy any more tasks on them.

Worker Management

When a Master receives a task to be executed from the developer, it will determine the number of Workers that are needed, the operations needed for each work and the topology of the nodes. Next, it will create these nodes as logical nodes and deploys them on the available

set of machines. If there are insufficient computing resources, more than one logical node may share a single CPU core. The Master also informs the nodes the topology of the system, so that they know who the upstream and downstream nodes are.

3.5 Chimera Tasks

Users will send tasks to Chimera by completing a task interface. This interface has the following required fields:

- **srcNum**. This refers to the number of sources that will send event streams to Chimera.
- **eventID**. An array of event IDs. The IDs specified should be of events that are required in the processing.
- **var**. An array of key names to monitor when processing events.
- **operation**. The operation that would be executed on the values of the keys being monitored.
- **aggr**. The name of the key by which Chimera will perform aggregation.

Chimera provides a set of common operations by default, such as *SUM*, *MAX*, and *MIN*. However, developers can define their own custom operations. They can modify the Chimera operations library, add their own operations, and distribute the library to the machines used in Chimera.

The following is an example of what a task interface may look like:

```
srcNum = {3},
eventID = {1,2},
var = {ts},
operation = {max(sum(span(1->2)))},
aggr = {mapID(6)}.
```

When the Master receives this task, it will parse it and determine two things: (i) the topology of the Workers, and (ii) the operation on each Worker.

The field *aggr* on the above example indicates that events with the same *mapID* will be aggregated together. The value 6 indicates that the *mapID* may have six unique values. Using this information, the Master will construct 6 Workers and each Worker would handle each unique *mapID*. Similarly, the number of Collectors is decided by the field *srcNum*.

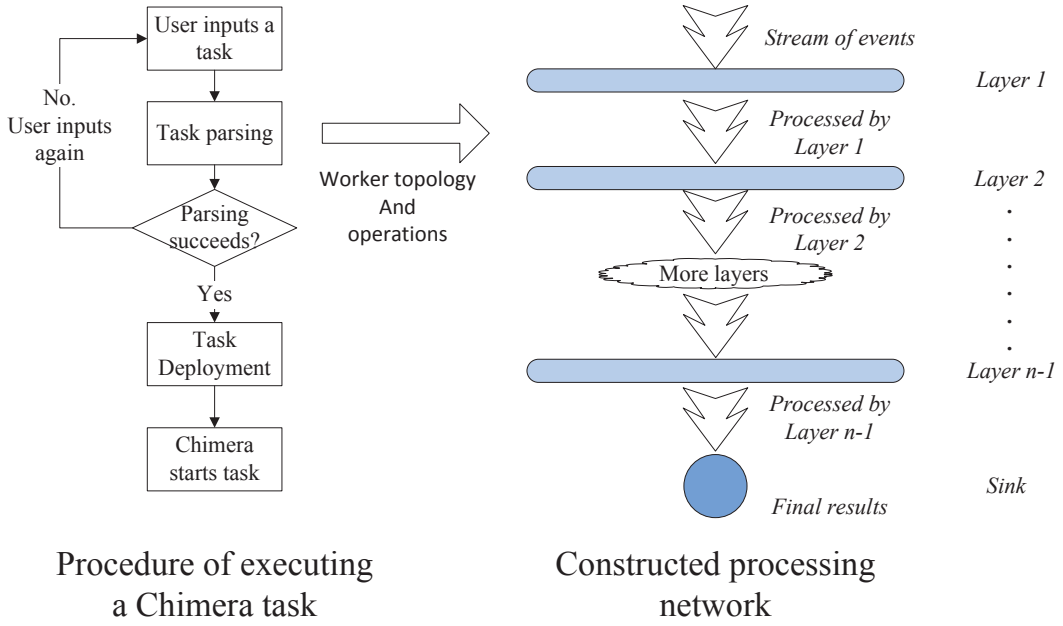

Figure 3.2: Overview of Chimera inputs and runs a task.

The fields *eventID*, *var*, and *operation* define the operations of Workers. In particular, the operation field indicates that the difference in the *ts* value between the events with IDs 1 and 2 should be summed. This summation happens individually for each unique *mapID* value. The final result returned is the map ID with the greatest summation.

3.6 Overview of Task Execution

In Figure 3.2, we show an overview of how Chimera executes a user's task.

1. The user inputs a task to Chimera, to inform the system of the task to execute.
2. The Master node parses the task content, determines the operations required, and the topology of Worker nodes on the set of available machines within the cluster.
3. The Master starts the Workers, deploys operations on them, and arranges them in the network according to the topology determined in the previous step.
4. The Master starts the system and the Collectors begin to provide the event streams (received from the sources) to the Workers.
5. Each Worker executes the operations deployed on it, and delivers the stream of results to the Workers at the next layer.

6. The event stream flows through each layer to produce the expected results, which are then given to the Sink node, where it is either displayed to the developer in real-time or is stored in a traditional database.

Chapter 4

Evaluation

In this chapter, we present our strategy of evaluating the limits of Esper. We begin by describing TankVille as the queries we use to compare Esper and Chimera are based on TankVille. Next, we describe the questions which we want answered, and the experiments we ran. We also provide details of our load generator, and the strategy of answering the questions with the generated events. Finally, we discuss the findings from our experiments.

4.1 TankVille

TankVille [22] is a real-time action game deployed on Facebook. It is used by the developers to evaluate Hydra [14], a peer-to-peer networking architecture. In TankVille, each player controls a tank and plays in a virtual battlefield, known as the game map, and competes with other players to collect resources and fight enemy AI-controlled tanks. To provide variety in the game, players can choose to play in any of the available maps. Players can either host a new game or join a game that is already in progress.

When users launch TankVille, measurement data from both the game and Hydra is collected. This allows the developers to understand how their game is performing and helps them reason about Hydra. In this chapter, we concern ourselves only with the game data.

4.2 Experiment Setup

Our goal is to evaluate Esper and Chimera with real-world queries. To this end, we attempt to answer a query that the TankVille developers had. They wanted to know which map in their game was most popular so as to identify which aspects of TankVille was most attractive

to players. This query can be answered via the following three questions:

1. How many players are there currently on each map?
2. Which map do players spend most of their time on?
3. What are the histograms of the time spent by every player on each map?

The answer to the first question provides a bird's eye view of the current state of the game. It would also help developers understand how activity on each map varies through time. The answer to the second question highlights clearly the map that is most popular. Answers to the third question provide a breakdown of how much time players spend on TankVille and on the individual maps.

The three questions above were answered with the aid of a load generator. We did not use live data from TankVille players due to two reasons. First, with live data, we would not be able to control the rate at which the events are produced. This makes it difficult to run controlled experiments. Second, TankVille is currently undergoing upgrades due to changes in the Facebook API. Therefore, we are unable to use it as a source of data. In our experiments, we take it that there are 6 maps in total. Events will be generated to simulate the activity of players joining and leaving these maps. The design of the load generator and the events produced to answer the above questions is detailed later in this chapter.

In our experiments, we concentrated on answering the above questions using both Esper and Chimera. Each experiment run uses either Esper or Chimera to produce results for one question. In each Esper experiment, the Esper server is deployed on one machine and the load generator is deployed on another machine. Both machines are on the same LAN. Before the experiment begins, a timing offset between both machines is calculated. This is to ensure that we can accurately measure the time taken by Esper to process an event, which is calculated by the elapsed time between the time the load generator creates the event and the time at which Esper finishes processing the event.

In our Chimera experiments, we configured Chimera to have 8 logical nodes: 1 data Collector, 6 Workers (one for each map), and 1 Sink. The Collector receives event streams from the load generator, splits them into sub-streams, and delivers them to a Worker depending on the event's map ID. After receiving the results from the Workers, the Sink may, depending on the question, do some final processing on the results before displaying them on the screen. Note that the Collector and Sink also require CPU resources. Like the Workers, they will suffer from bad performance if there are insufficient resources. We ran multiple experiments and

varied the number of CPU cores available to Chimera, from just 1 core to 8 cores. When only 1 core is available, each logical node shares the core to perform their respective operations. When 8 cores are available, each node will make use of one. In all other cases, the nodes will be evenly spread among the available cores.

We controlled the availability of CPU cores on multi-core machines by enabling only the required number of cores. In our experiments, we did this by adding the `maxcpus` option to *Grub*, the boot-loader used to load the GNU/Linux operating system. For example, by using the option `maxcpus=1`, the operating system would only be able to use one core.

Each experiment run lasts one minute. One minute is sufficient as the load generator stabilizes within 5 seconds and begins to generate events in the desired rates. We verified our approach by running experiments up to 30 minutes and found the results to be similar to the results obtained when we run the experiments for a minute.

As Esper is centralized, we evaluated it on a single machine that has a 2 GHz dual-core processor with 4 GB of RAM. However, only one core is enabled in the Esper experiments and in experiments where Chimera is deployed on only one core. On experiments where Chimera was deployed on multiple cores, the logical Worker nodes were deployed on a cluster of machines whose specifications ranged from a 2 GHz dual-core processor to a 2.67 GHz quad-core processor and with the RAM ranging from 2 to 4 GB. We were unable to run all the logical nodes of Chimera on machines with exactly the same specifications due to practical reasons. The Master node on all Chimera experiments was deployed on a separate machine that was not involved in any data processing.

We used only commodity hardware in our tests and the default Sun J2SE 1.6 JVM (without any custom tuning) so as to see what kind of standard performance developers can expect.

4.3 Load Generator

An actual instance of TankVille generates many different kinds of events, to evaluate both the game and the research platform it is running on. In our load generator, we generate just two events to answer the questions stated earlier. Namely, the events *PlayerJoin* and *PlayerLeave* are generated when players join and leave games respectively. These events are generated as a text tuple consisting of four fields: (i) a timestamp stating when the event was created, (ii) an event identifier that differentiates the various kinds of events from each other, (iii) an identifier of the map used in the game the player created, joined, or left, and (iv) an identifier of the player. As we demonstrate later, these two events with the four fields are sufficient to

answer the given questions.

Both events are produced continuously so as to match the desired event generation rates. However, to make the simulation realistic, the method by which we generate the *PlayerJoin* and *PlayerLeave* events for a given player is based on our observations of TankVille.

We noticed that the number of players arriving at the game within a time unit obeys a poisson distribution. Hence, according to probability theory, the inter-arrival time between every two players obey an exponential distribution. The cumulative distribution function (CDF) of an exponential distribution is given in Equation (4.1). The parameter λ is the inverse of the average inter-arrival time. This affects the time between the generation of two consecutive *PlayerJoin* events.

$$F(x) = 1 - e^{-\lambda x} \tag{4.1}$$

The amount of time a player spends in a map also obeys an exponential distribution. Therefore, the parameter λ here refers to the inverse of the average time a player spends in a map. This affects the time between the generation of a *PlayerJoin* and the *PlayerLeave* events for the same player. The λ parameters can be modified to adjust the event generation rate. For example, suppose the average inter-arrival rate is 100 ms and the average amount of time spent by players in a map is 10 seconds. This means that the load generator will produce (on average) 600 *PlayerJoin* and 600 *PlayerLeave* events in one minute. In total, 1200 events would be produced per minute.

The generator creates events with the help of a priority queue (i.e. a heap) that sorts events in ascending order of the their timestamps and with the help of a *PlayerArrival* trip event. When the generator is started, a *PlayerArrival* event with the current system time as its timestamp is inserted into the heap. The generator would then continuously extract events from the heap with the smallest timestamp and take the appropriate action. If this event happens to be the *PlayerArrival* trip event, the generator will perform the following actions:

1. A *PlayerJoin* event is created with the same timestamp as that of the trip event, and with a random map ID and player ID. The player ID would be one that was not created previously.

2. Next, a *PlayerLeave* event is created with the same map ID and player ID as the *PlayerJoin* event that was just created. The timestamp of this *PlayerLeave* event would be the timestamp of the newly created *PlayerJoin* event, but with a random amount of time

added. This random time is taken from the exponential distribution.

3. A new *PlayerArrival* trip event. Its timestamp is computed by adding another random amount of time to the timestamp of the newly created *PlayerJoin* event. This random time is taken from the Poisson distribution.

4. The events created above are added into the heap and the generator would continue its operation of extracting events from the heap.

If the event extracted from the heap is not a trip event, the generator would check if the current system time is either equal to or is past the event timestamp. If so, the event is sent to the stream processing system being tested. Otherwise, the generator would sleep until the system time has advanced to the extracted event's timestamp.

4.4 Answering the questions

In this section, we explain the strategy of using the *PlayerJoin* and *PlayerLeave* events to answer the three questions. We only present the high-level idea here; the actual queries written in Esper and Chimera are placed in Appendix A.

The first question is "How many players are there currently on each map?". This can be answered by keeping a counter for each map. When a *PlayerJoin* event is encountered, this counter can be incremented. Likewise, when a *PlayerLeave* event is received by the stream processor, the counter can be decremented.

The second question is "Which map do players spend most of their time on?". When a *PlayerJoin* event arrives, the player ID and the timestamp of the event has to be saved. Next, when a *PlayerLeave* event arrives, we have to use its player ID, retrieve the previously stored timestamp. Using this timestamp and the timestamp in *PlayerLeave*, the total amount of time spent in the game can be calculated. This duration can then be added to the total duration spent for the map specified by map ID in *PlayerLeave*. The map with the largest total duration is the most popular map.

The third question is "What are the histograms of the time spent by every player on each map?". In this question, for each map, we plot a histogram of the time spent by players on that map. Each bin in this histogram specifies some range of time, and the frequency specified by the bin refers to the number of players who spent the amount of time specified by the bin interval in TankVille. We also plot another histogram that illustrates the total amount of time spent by players in all maps. The amount of time a player spent would be

the time elapsed between the *PlayerJoin* and *PlayerLeave* event pair while the grouping of the player time by map can be done by taking into account the *MapID* attribute in the events.

4.5 Scalability

We tested the scalability of Esper and compared it with Chimera. To this end, we ran the aforementioned experiments to answer the three questions and varied the event generation rate. The aim is to see how the time taken by the systems to process each event varies with the amount of load they are under. These experiments will also show the maximum number of events Esper and Chimera can process before becoming completely overwhelmed. We plot the results of answering the three questions in Figure 4.1, Figure 4.2 and Figure 4.3 respectively. Note that for purposes of clarity, we did not plot results where Chimera was running on 2, 4, and 6 cores.

As shown, the maximum throughput of Esper for the three questions is 467K, 260K, and 125K per minute. In other words, at its fastest, Esper performs at a rate of 7783 events per second. This is different from official benchmarks released by EsperTech, the company developing Esper, which claims that Esper can handle 500K events per second. There are two reasons for this. Firstly, the official benchmarks are executed on a two dual-core processors and on a JVM with tuned with custom parameters. In contrast, our experiments use only one core of a dual-core processor and uses a JVM instance with default parameters. The aim of our tests is to see how Esper will perform without any specific, performance-related tweaks.

Secondly, and more importantly, the queries executed in the benchmark are simple *SELECT* statements and Esper may be well-optimized answer those questions. However, Esper's performance can vary widely. Mendes et al. [23] showed that on a machine with two 2.5GHz quad-core processors (8 cores in total), a simple *SELECT* statement processes events at a rate of 500K per second but a *SELECT* statement with joins may perform only at a rate of 50K events per second. This suggests that there may be some bias to the benchmarks.

Furthermore, as demonstrated earlier, we have to maintain state for our queries. Esper may not be as optimized in keeping track of the various counters. The key finding here is that *Esper's performance varies widely between different query types.* A developer who wishes to use Esper needs to spend the effort to optimize his queries. He could even tune the JVM to squeeze out more performance.

The next observation to be made is that Chimera outperforms Esper in terms of throughput even when both are deployed on only one core. This is most likely due to Chimera being

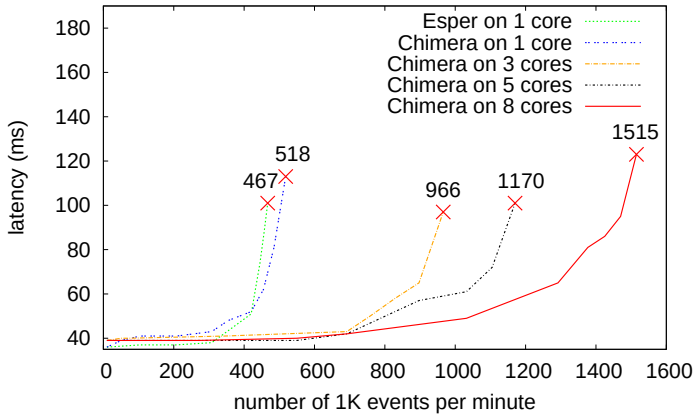

Figure 4.1: Processing capacity of Chimera and Esper for question 1: number of players on each map

simpler in design compared to Esper, which is complex and supports features such as a general, SQL-like query language. Note however that when both are deployed on one core, Esper uses one thread to answer the defined questions while Chimera uses multiple threads, one for each logical node.

Therefore, to have a fairer comparison and to better understand the complexity tradeoff between Chimera and Esper, we ran another experiment where all logical nodes in Chimera use a single thread. Figure 4.4 shows the results of this experiment contrasted with Esper's performance for the three questions. Chimera performs even better than Esper compared to when it was using multiple threads. In questions 2 and 3, Chimera offers over twice the performance of Esper. This shows that *if the features offered in a complex system are not required, it is better for the developer to use a simpler system to obtain better performance.*

When Chimera is deployed on multiple cores (with the Master node located on a separate machine that is not involved in the actual computation), the maximum throughput increases and the latency of processing events decreases. For example, in question 1, Chimera on one core has a latency of 100 ms when processing 500K events per minute. When Chimera is deployed on eight cores, where each logical node occupies a separate core, Chimera can handle 1200K events per minute with 60 ms latency. However, it costs more resources to deploy Chimera on eight cores than on just one core. The take-away here is that *when deploying a distributed stream processing system, the benefit of having more throughput with lower latencies has to be balanced with the cost of needing more cores.*

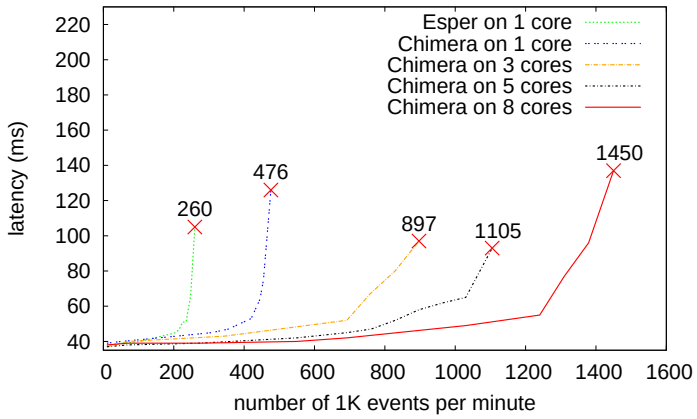

Figure 4.2: Processing capacity of Chimera and Esper for question 2: time spent by players on each map

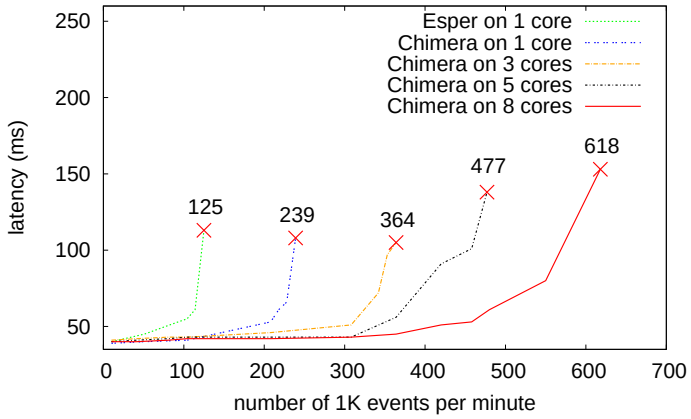

Figure 4.3: Processing capacity of Chimera and Esper for question 3: histogram of players gaming time

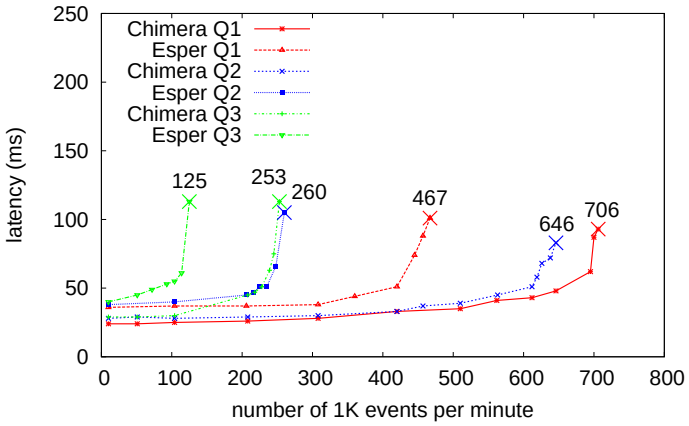

Figure 4.4: Chimera on one thread compared with Esper for all three questions (on one core)

Chapter 5

Conclusion

In this thesis, we conducted a study to understand the limits of centralized stream processing systems so as to better understand when it is good to use centralized systems and when it is better to use distributed systems. To this end, we evaluated Esper, a state-of-the-art centralized stream processing system. We used Esper to answer some queries that the developers had of TankVille, a game on Facebook. We also processed these same queries on Chimera, a distributed stream processing system we built.

We found that Esper's performance varies widely depending on the complexity of the query. While it can process events very quickly when the queries are simple, the processing rate drops significantly when the queries are complex. In such cases, distributed systems outperform Esper easily. However, if a developer only uses simple queries, it may be more cost-effective to just use Esper instead. There may also be instances where developers do not require the various features offered by Esper. In such cases, it is better for them to use a simpler stream processing system as that would offer better performance than Esper.

Through our study, we have showed that centralized stream processing systems are not necessarily better than distributed systems, even when the rate at which events arrive from sources is low. Developers have to decide whether to use a centralized system or a distributed system based on the complexity of their queries, based on the features they need from a stream processing system, and based on the rate of incoming events.

Bibliography

[1] Complex event processing (cep) technology & real-time business process management - sybase inc. http://www.sybase.com/products/financialservicessolutions/complex-event-processing.

[2] Esper. http://esper.codehaus.org/.

[3] Linear road. http://pages.cs.brandeis.edu/ linearroad/.

[4] Streambase: Complex event processing, event stream processing. http://www.streambase.com/.

[5] Streamcruncher. http://www.streamcruncher.com/.

[6] Tibco businessevents. http://www.tibco.com/products/business-optimization/complex-event-processing/businessevents/.

[7] Truviso web analytics software. http://www.truviso.com/.

[8] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, , and S. Zdonik. The design of the borealis stream processing engine. In *CIDR '05: Proceedings of the Conference on Innovative Data Systems Research*, 2005.

[9] D. J. Abadi, D. Carney, U. ÃĞetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB Journal: International Journal on Very Large Data Bases*, 12:120–139, 2003.

[10] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Y. Li, V. Di Nicola,

X. Wang, D. Maier, S. Grell, O. Nano, and I. Santos. Microsoft CEP Server and Online Behavioral Targeting. *VLDB '09: International Journal on Very Large Data Bases*, 2, August 2009.

[11] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, and M. Stonebraker. Linear road: A stream data management benchmark. In *VLDB '04: Proceedings of the 30nd International Conference on Very Large Data Bases*, 2004.

[12] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *NSDI '04: Symposium on Networked Systems Design and Implementation*, 2004.

[13] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR '07: Proceedings of the Conference on Innovative Data Systems Research*, 2007.

[14] L. Chan, J. Yong, J. Bai, B. Leong, and R. Tan. Hydra - a massively-multiplayer peer-to-peer architecture for the game developer. In *Proceedings of NetGames '07*, September 2007.

[15] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. Telegraphcq: Continuous dataflow processing for an uncertan world. In *CIDR '03: Proceedings of the Conference on Innovative Data Systems Research*, 2003.

[16] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI '10: USENIX Symposium on Networked Systems Design and Implementation*, 2010.

[17] J. Dean, S. Ghemawat, and G. Inc. Mapreduce: simplified data processing on large clusters. In *In OSDI '04: Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*. USENIX Association, 2004.

[18] P. Dekkers. Complex event processing. In *Master Thesis, Radboud University Nijmegen*, October 2007.

[19] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *CIDR '07: Proceedings of the Conference on Innovative Data Systems Research*, 2007.

[20] InfoWorld. Big data to get even bigger in 2011, January 2011. http://www.infoworld.com/d/data-explosion/big-data-get-even-bigger-in-2011-064.

[21] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[22] W. K. Leong, D. Seah, A. Razeen, and B. Leong. Tankville. http://apps.facebook.com/tankville.

[23] M. R. Mendes, P. Bizarro, and P. Marques. A performance study of event processing systems. In *TPCTC: The TPC Technology Conference on Performance Evaluation and Benchmarking*, 2009.

[24] M. R. N. Mendes, P. Bizarro, and P. Marques. A framework for performance evaluation of complex event processing systems. In *DEBS '08: International Conference on Distributed Event-Based Systems*, 2008.

[25] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. Nexmark - a benchmark for queries over data streams. 2002.

[26] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD '06: Proceedings of the ACM International Conference on Management of Data*, 2006.

Appendix A

Solving Questions with Esper and Chimera

A.1 Solving Question 1

Question: *How many players are there currently on each map?*

A.1.1 Using Esper

We first define the two events that Esper is interested in, and the variables used in the queries.

```
Configuration engineConfig = new Configuration();
engineConfig.addEventType("PlayerJoin", PlayerJoinEvent.class.getName());
engineConfig.addEventType("PlayerLeave", PlayerLeaveEvent.class.getName());

EPServiceProvider engine =
EPServiceProviderManager.getProvider("myEsperEngine", engineConfig);
```

We define auxiliary variables in Esper to be used in the queries.

```
/*
 * create a Integer variable named "playerCount" with
 * initial value as 0
 */
engine.getEPAdministrator().getConfiguration().addVariable("playerCount",
Integer.class, 0);

/*
 * create counters for players on each map, maps are indexed from 0 to 5
```

```
 */
engine.getEPAdministrator().getConfiguration().addVariable("pC0", Integer.class, 0);
engine.getEPAdministrator().getConfiguration().addVariable("pC1", Integer.class, 0);
engine.getEPAdministrator().getConfiguration().addVariable("pC2", Integer.class, 0);
engine.getEPAdministrator().getConfiguration().addVariable("pC3", Integer.class, 0);
engine.getEPAdministrator().getConfiguration().addVariable("pC4", Integer.class, 0);
engine.getEPAdministrator().getConfiguration().addVariable("pC5", Integer.class, 0);
```

Then we use the queries below to count the number of players.

```
"on PlayerJoin set playerCount = playerCount + 1, curEvtTs = timeStamp";
"on PlayerLeave set playerCount = playerCount - 1, curEvtTs = timeStamp";
"on PlayerJoin(mapID=0) set pC0 = pC0 + 1";
"on PlayerLeave(mapID=0) set pC0 = pC0 - 1";
"on PlayerJoin(mapID=1) set pC1 = pC1 + 1";
"on PlayerLeave(mapID=1) set pC1 = pC1 - 1";
"on PlayerJoin(mapID=2) set pC2 = pC2 + 1";
"on PlayerLeave(mapID=2) set pC2 = pC2 - 1";
"on PlayerJoin(mapID=3) set pC3 = pC3 + 1";
"on PlayerLeave(mapID=3) set pC3 = pC3 - 1";
"on PlayerJoin(mapID=4) set pC4 = pC4 + 1";
"on PlayerLeave(mapID=4) set pC4 = pC4 - 1";
"on PlayerJoin(mapID=5) set pC5 = pC5 + 1";
"on PlayerLeave(mapID=5) set pC5 = pC5 - 1;";
```

Finally we input the queries into the Esper engine, and start Esper. We observe the auxiliary variables to obtain the results required for this question.

A.1.2 Using Chimera

We submit the task defined as follows.

```
srcNum = {1},
eventID = {1,2},
var = {playerID},
operation = {count,+1,-2},
aggr = {mapID(6)}.
```

There is 1 data source. Generated events with ID 1 (*PlayerJoin*) and 2 (*PlayerLeave*) are required to answer the question. Chimera will count the number of *playerIDs* in each of the 6 maps. *PlayerJoin* event will increment the counter and *PlayerLeave* event will decrement the counter.

There are 6 Workers and each Worker handles each unique *mapID* (there are 6 unique *mapIDs* in total). Collectors will receive events from the source and deliver them to each Worker depending on the event's *mapID*. The Sink receives the player counts and displays it to the developer.

A.2 Solving Question 2

Question: *Which map do players spend most of their time on?*

A.2.1 Using Esper

Similar to question 1, we define the two events that Esper is interested in. The auxiliary variables to be used in the queries are defined as follows.

```
/*
 * create counters for player number on each map, maps are indexed from 0 to 5
 */
engine.getEPAdministrator().getConfiguration().addVariable("pC0", Integer.class, 0);
engine.getEPAdministrator().getConfiguration().addVariable("pC1", Integer.class, 0);
engine.getEPAdministrator().getConfiguration().addVariable("pC2", Integer.class, 0);
engine.getEPAdministrator().getConfiguration().addVariable("pC3", Integer.class, 0);
engine.getEPAdministrator().getConfiguration().addVariable("pC4", Integer.class, 0);
engine.getEPAdministrator().getConfiguration().addVariable("pC5", Integer.class, 0);

/*
 * create time counter for each map, maps are indexed from 0 to 5
 */
engine.getEPAdministrator().getConfiguration().addVariable("mT0", Double.class, 0);
engine.getEPAdministrator().getConfiguration().addVariable("mT1", Double.class, 0);
engine.getEPAdministrator().getConfiguration().addVariable("mT2", Double.class, 0);
engine.getEPAdministrator().getConfiguration().addVariable("mT3", Double.class, 0);
engine.getEPAdministrator().getConfiguration().addVariable("mT4", Double.class, 0);
engine.getEPAdministrator().getConfiguration().addVariable("mT5", Double.class, 0);

// timer recording the last event
engine.getEPAdministrator().getConfiguration().addVariable("lastTime", Long.class, 0);
```

We use the following queries to get the map with the greatest play time.

```
/*
 * Auxiliary string used in queries
 */
String updateMapTime = "mT0 = mT0 + pC0 * (timeStamp - lastTime)" +
", mT1 = mT1 + pC1 * (timeStamp - lastTime)" +
```

```
", mT2 = mT2 + pC2 * (timeStamp - lastTime)" +
", mT3 = mT3 + pC3 * (timeStamp - lastTime)" +
", mT4 = mT4 + pC4 * (timeStamp - lastTime)" +
", mT5 = mT5 + pC5 * (timeStamp - lastTime)";

/*
 * queries for PlayerJoin events
 */
"on PlayerJoin(mapID=0) set " + updateMapTime + ", lastTime = timeStamp, pC0 =
pC0 + 1";
"on PlayerJoin(mapID=1) set " + updateMapTime + ", lastTime = timeStamp, pC1 =
pC1 + 1";
"on PlayerJoin(mapID=2) set " + updateMapTime + ", lastTime = timeStamp, pC2 =
pC2 + 1";
"on PlayerJoin(mapID=3) set " + updateMapTime + ", lastTime = timeStamp, pC3 =
pC3 + 1";
"on PlayerJoin(mapID=4) set " + updateMapTime + ", lastTime = timeStamp, pC4 =
pC4 + 1";
"on PlayerJoin(mapID=5) set " + updateMapTime + ", lastTime = timeStamp, pC5 =
pC5 + 1";

/*
 * queries for PlayerLeave events
 */
"on PlayerLeave(mapID=0) set " + updateMapTime + ", lastTime = timeStamp, pC0 =
pC0 - 1";
"on PlayerLeave(mapID=1) set " + updateMapTime + ", lastTime = timeStamp, pC1 =
pC1 - 1";
"on PlayerLeave(mapID=2) set " + updateMapTime + ", lastTime = timeStamp, pC2 =
pC2 - 1";
"on PlayerLeave(mapID=3) set " + updateMapTime + ", lastTime = timeStamp, pC3 =
pC3 - 1";
"on PlayerLeave(mapID=4) set " + updateMapTime + ", lastTime = timeStamp, pC4 =
pC4 - 1";
"on PlayerLeave(mapID=5) set " + updateMapTime + ", lastTime = timeStamp, pC5 =
pC5 - 1";
```

Then we input the query into Esper engine, and get the required answer by observing the variables of the play time counter for each map.

A.2.2 Using Chimera

We submit the task defined as follows.

```
srcNum = {1}
eventID = {1,2},
var = {ts},
```

```
operation = {max(sum(span(1->2)))},
aggr = {mapID(6)}.
```

There is 1 data source. Generated events with ID 1 (*PlayerJoin*) and 2 (*PlayerLeave*) are required to answer the question. The system will calculate the time spent by each player and sum them to obtain the total amount of time spent in each map. Of the total times, the map with the greatest time spent is selected and returned.

As per question 1, there are 6 Workers and once again, the Collector will deliver events to the Workers based on the *mapID*. The Workers will calculate the total time spent in each map and give the timings to the Sink. The Sink will perform an additional operation, to choose the map with the greatest time, and will return the result to the developer.

A.3 Solving Question 3

Question: *What are the histograms of the time spent by every player on each map?*

A.3.1 Using Esper

In this question, we use a combination of both Esper and our own separate program to generate the histograms. We define the two events, *PlayerJoin* and *PlayerLeave* for Esper to capture as per the previous two questions. Then we use two queries in Esper engine to process the events. We take the results from Esper and pass it to our program to compute the histograms.

```
HistogramSubscriber sub = new HistogramSubscriber();
EPStatement joinStatement = engine.getEPAdministrator().createEPL("Select␣*
from␣PlayerJoin");
joinStatement.setSubscriber(sub);

EPStatement leftStatement = engine.getEPAdministrator().createEPL("Select␣*␣from
PlayerLeave");
leftStatement.setSubscriber(sub);
```

We provide here a code snippet of the key functions of *HistogramSubscriber*, the Java class used in our separate program.

```
class HistogramSubscriber{
        /*
         * create a set storing all players in the game
```

```
         */
        private HashSet<Player> playerSet = new HashSet<Player>();

        /*
         * receive a PlayerJoin event
         */
        public void update(PlayerJoin event){
                Player p = new Player(event.getPlayerID());
                updateHistogram(event.getTimestamp());
                playerSet.add(p);
        }

        /*
         * receive a PlayerLeave event
         */
        public void update(PlayerLeave event){
                Player p = new Player(event.getPlayerID());
                updateHistogram(event.getTimestamp());
                playerSet.remove(p);
        }

        /*
         * update the histogram
         */
        private void updateHistogram(long currentTime){
                for(Player p : playerSet){
                        double time = currentTime - p.joinTime;
                        p.gameTime = time;
                        histogram.adjustBucket(p.playerID, p.gameTime);
                }
        }
}
```

A.3.2 Using Chimera

We submit the task defined as follows.

```
srcNum = {1}
eventID = {1,2},
var = {ts},
operation = {histogram(span(1->2), 5 sec)},
aggr = {mapID(6)}.
```

There is 1 data source. Generated events with ID 1 (*PlayerJoin*) and 2 (*PlayerLeave*) are required to answer the question. The system will calculate the gaming time of each player (span on values of key *ts* between event 1 and event 2), and put them into a histogram, with

bucket width as 5 seconds. This will be done for each of all the 6 maps.

As per the previous two questions, there are 6 Workers and once again, the Collector will deliver events to the Workers based on the *mapID*. The Workers will compute the histogram for each map and send it to the Sink. The Sink does not do any processing and simply returns the histograms to the developer.