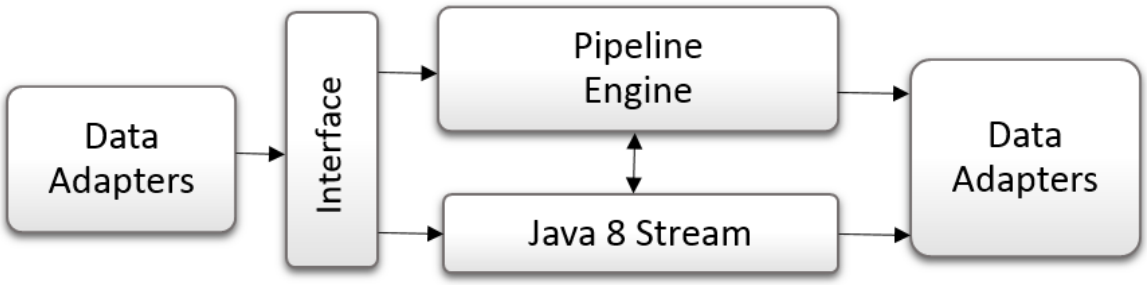
CHALMERS



Distributed Stream Analysis with Java 8 Stream API

Master's Thesis in Computer Systems and Networks

ANDY MOISE PHILOGENE
BRIAN MWAMBAZI

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, February 15, 2016

Distributed Stream Analysis with Java 8 Stream API

Andy Moise Philogene
Brian Mwambazi

February 15, 2016

Examiner: Marina Papatriantafilou

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone +46(0)31-772 1000

[The design overview of our API]

Department of Computer Science and Engineering
Göteborg, Sweden February 15, 2016

Abstract

The increasing demand for fast and scalable data analysis has impacted the technology industry in ways that are defining the future of computer systems. The booming of machine-to-machine communication, cloud based solutions etc. has led to a large amount of data (popularly termed as Big Data) to be generated and processed by supporting applications in a real-time fashion. These changes in the field have pushed the way for the introduction of stronger data processing paradigms and a proliferation of data stream processing systems. These systems usually come with an execution environment and programming models that developers have to adopt. Migrating applications from one system to another is in general tedious and comes with a learning curve.

The Stream package introduced in Java 8 brings a new abstraction on a sequence of data. Such a sequence can be used to represent any flow of data and thus is interesting in data streaming. Together with the support of new features like lambda expressions and functional interfaces, the stream package has been brought to increase the expressiveness of Java and ease the writing of complex operations in a declarative manner when dealing with a sequence of data. These language features are attractive in data stream processing. However the Java 8 Stream API is currently inclined towards batch processing bounded datasets like collections, than data streaming which involves unbounded datasets. For this reason it does not provide an easy way for programmers to write applications for typical data stream use-cases or stateful operations such as window-based aggregates. Furthermore, the current scalability mechanism provided in this API does not fit a typical data streaming environment.

In this thesis, we explore the expressiveness of the Java 8 Stream API together with the data streaming paradigm. We design and implement JxStream, an API around the existing Java 8 Stream that provides more stream analysis capabilities to the developer and eases scalable processing in a distributed environment. We evaluate JxStream with different test cases aimed at investigating specific performance attributes. The outcome shows that JxStream would allow programmers to solve a greater range of data streaming problems, without sacrificing the performance. The test results also show that our implementation performs fairly well when comparing similar features with a stream processing engine. We believe this tool can be of good use in writing stream analysis applications using Java.

Keywords: Data Streaming, Java 8 Stream, Stream Processing, Lambda Expression, Stream Pipeline

Acknowledgements

We would like to thank our supervisors Vincenzo Gulisano and Magnus Almgren and our examiner Marina Papatriantafilou for their guidance during this masters' thesis.

We also thank the Swedish Institute for their academic grant. This research has been produced during the period of our Swedish Institute Study Scholarship.

Andy & Brian, Gothenburg February 15, 2016

Contents

List of Figures

List of Tables

Acronyms

AMQP Advanced Message Queuing Protocol.

API Application Programming Interface.

CSV Comma-Separated Values.

DBMS Database Management System.

Forex Foreign Exchange.

JVM Java Virtual Machine.

RPC Remote Procedure Call.

SAM Single Abstract Method.

SPE Stream Processing Engines.

STOMP Simple Text Oriented Messaging Protocol.

UML Unified Modeling Language.

1

Introduction

The advent of computer applications and devices that generate a large volume of data has created a radical shift in modern computer systems. Due to the size of this data that is continuously created, it is often impractical to store it for later processing, as it would be done in relational Database Management System (DBMS) solutions. Similarly there has been a rising need for extracting and interpreting information from this data in near real-time in order to gain insights and make timely decisions. These demands have resulted in a new way of processing data, called Data Streaming. The main challenges in Data Streaming hinges on the fact that a continuous supply of data need to be processed in an efficient way. Stonebraker et al. [1] describe eight requirements a system should fulfill to meet the demands of a range of real-time stream processing applications. A lot of research work [2, 3, 4, 5] has been conducted to address these challenges. This has brought forth a new class of systems classified as Stream Processing Engines (SPE) [6, 7, 8, 9]. These applications provide a platform for near real-time data processing and analysis.

SPEs have been successful in easing the development and management of applications with near real-time requirements. Basically they provide a platform on which programmers can write and run applications for data streaming. These applications are built using the SPE's provided programming model. As a result programmers need to learn the model of each SPE they intend to use. This learning curve can hinder the progress of writing new stream applications. Additionally since these models are not standard, migrating applications across different SPEs can be problematic.

The latest version of Java (Java 8) has introduced the Stream package [10] which is a new abstraction that helps a programmer process a sequence of data. This together with new supporting features like lambda expressions [11] and functional interfaces [12] has brought expressiveness and ease of writing complex operations for a sequence of data.

Due to the fact that these constructs are part of the programming language itself as opposed to another processing engine, Java 8 Stream has a promising potential in the world of Data Stream Processing.

In the next section we define the problem this thesis is solving and follow-up with a discussion about the contribution in section 1.2. We then give the report outline in section 1.3.

1.1 Problem definition

The Stream interface introduced in Java 8 brings an abstraction on a sequence of data. However the Java 8 Stream Application Programming Interface (API) is currently inclined towards fixed size data and thus more suited for batch processing. Batch processing involves processing a fixed volume of data collected over a period of time. The computational result is only realized after the whole dataset is processed. In contrast, data stream processing involves a continuous input, process and output of data. To achieve this, stream processing employs a technique known as *windowing* on the continuous flow of data. A fixed size *window* which is either defined by number of tuples or time is applied on the data. This effectively breaks up the unbounded stream into groups of tuples whereby tuples of a particular group are processed together to produce the output. We refer the reader to section 2.1 for more details about window semantics.

The Java 8 Stream API does not provide an easy way for programmers to write applications for typical data stream scenarios where the data to be processed is not fixed but unbounded. In order to use this programming model in stream analysis applications the package needs to be extended into a solution that addresses the inadequacies. The solution should maintain the same level of abstraction and expressiveness of the Java 8 Stream package. The main short comings of the package when used in stream analysis are summarized below (For detailed discussion about these problems we refer the reader to chapter 3.

- The interface does not provide an easy way for programmers to write applications for typical stream analysis use-cases. Common stream analysis operators and the ability to define multiple continuous queries in one application is not directly supported by the Java 8 Stream API. As an example, stream operators with window semantics are often used in data streaming. These operators provide a window (time frame, tuple count) for which the action is to be performed.

- Although Java Streams come with a parallel function that help to ease the parallelization of the stream operations, the mechanism used to achieve this parallelization is not well suited for unbounded data. Java 8 Stream's parallelism is based on the *ForkJoin* framework which in turn is based on the approach of Doug Lea [13]. This approach to parallelization uses a recursive decomposition strategy of dividing a problem, in this case data elements are broken into sub-problems which

are then executed in parallel. This approach is designed for fixed datasets since the size is known before-hand. Applying this approach on unbounded data is not feasible without some modification to the implementation of the API.

- Java 8 Stream API does not come with tools to make it scale beyond one Java Virtual Machine (JVM). A programmer has to address this if working with a distributed environment/streams which is usually the typical case in data streaming.

The programming interface provided by the Java Stream API provides an attractive way to process in-memory collections. An approach based on the existing API to provide a stronger stream processing paradigm for unbounded streams would be highly useful and easier to use for developers. The objective of this thesis is to overcome the above mentioned pitfalls and thus build on top of the Java 8 Streams a framework that is suited for unbounded data stream processing while offering the same level of abstraction and expressiveness of the Java 8 Streams. Additionally this framework should have minimal overheads. Thus part of the aim is to maximize the throughput and minimize the latency of the framework.

1.2 Contribution

The overall contribution of this thesis is the design and implementation of JxStream: a data stream analysis tool built on top of the Java 8 Stream package. To build our solution we overcome the challenges that Java 8 Stream has when applied to unbounded streams. Our solution differs from current data stream systems in that its core is close to the Java language construct itself. Developers can directly integrate data streaming semantics in their applications without needing to deploy dedicated systems. To create a data streaming environment, we need to provide abstractions that can continuously read data from different sources. A data source can be a file, a database or a node from a cluster. The best practice is to use messaging queues to buffer incoming data and relieve the stress that can be created on the system if the data is produced at a faster rate than processed. Thus we provide abstractions to interface with different queuing systems and the mechanism to easily implement interfaces to new ones.

Furthermore we bring a new programming model that is based on the Java 8 Streams to data stream analysis. Adopting the Java 8 Stream programming model makes our system intuitive for users already familiar with the language. Following the same level of expressiveness, we implement data stream operators which support window semantics. This gives programmers the power to write complex windowing operations in an easy fashion.

We evaluate its performance by comparing its throughput and latency with a fully fledged stream processing engine called Apache Storm. The results indicate that JxStream can indeed provide high throughput and low latency applications hence can be an alternative to SPEs in certain scenarios. In addition we evaluate its scalability and compare

the performance of the different implementations of its operators. Apart from measuring the performance we also illustrate its expressiveness by implementing some real life use-case scenarios which can also be served as a tutorial to get acquainted with the system.

1.3 Outline

The report is structured in the following Chapters:

- **Chapter 2: Background** This chapter introduces the background knowledge needed to understand this thesis. We look at the data streaming paradigm, the Java 8 Stream package and queuing systems.
- **Chapter 3: Problem Description** This chapter presents a detailed description of the problem this thesis is tackling, together with the evaluation plan.
- **Chapter 4: System Design & Architecture** This chapter explains the design decisions made for JxStream and the overview of its architecture.
- **Chapter 5: Implementation** This chapter looks at how we turned the design into the actual system by discussing the system's implementation. Here we also give full details of how JxStream solves the problems of the thesis.
- **Chapter 6: Uses-cases** This chapter shows the usability of JxStream by solving different problems and using it on different use-case scenarios.
- **Chapter 7: Evaluation** This chapter presents the evaluation of the system's performance by using sets of experiments to measure different aspects of the system.
- **Chapter 8: Related Work** This chapter covers research and systems whose work are related to this thesis.
- **Chapter 9: Conclusion** This chapter gives a conclusion to the report and also discuss the further work that could be done to improve the implemented system.

2

Background

In this chapter, we introduce preliminary knowledge important for a better understanding of this thesis. In Section 2.1 we introduce the Stream processing paradigm and discuss the basics concepts about data streams. Then we present the Java 8 Stream package and underlying concepts in Section 2.2. Java 8 is a cornerstone in the reasons that motivated us to pursue this thesis. The evolution of the language and few of the features that were introduced make it attractive for data streaming systems. Finally in Section 2.3 we discuss Queuing systems that are particularly important in the implementation of our system. In a data streaming environment, data is being constantly generated. Queues overcome many issues for us. They reduce considerably the possibility to overflow the data consumer when the data producers are faster. They can also help us distributed data to consumers, without having to duplicate data by ourselves. Thus we provide abstractions off the shelf to different queuing systems.

2.1 Stream Processing Paradigm

The proliferation of real-time data producing devices and applications has brought in a demand of a new approach of dealing with a continuous flow of data. In view of this, a lot of research work has been done in this area to find an effective way of processing such data. The outcome of this work has led to a common consensus on the formal defining concepts that constitutes streams. In this section we discuss the fundamental concepts surrounding the stream processing paradigm.

2.1.1 Stream

A stream is a sequence of unbounded data continuously flowing from a source towards a sink. The source of a stream can be any application that produces data in an append only manner. Examples of such applications range from sensors, smart meter devices to telecoms, logging and financial applications. The single data element in a stream is known as a tuple. This is basically a fixed-schema defined list of fields or values. Thus formally with $Tuple\ t = f_1,f_2,f_3,....,f_i$ a stream S is be defined as:

$$S = t_1,t_2,t_3,t_4,t_5,t_6.....t_n$$

As the tuples flow through the stream a series of one or more operators(computations) can be performed to transform(create new), filter or extract information out of the tuples. All tuples coming from a particular operator will share the same schema. The process of performing these computations is what is known as Stream Processing. Figure 2.1 shows an illustration of streams in action.

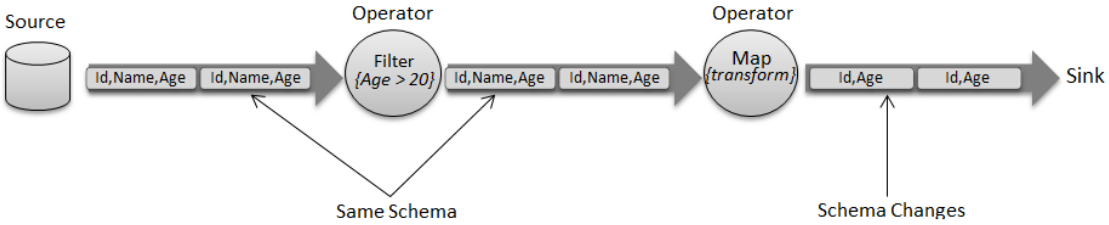


Figure 2.1: Stream processing in action. Two operators are applied on a stream of user data;the filter operator does not change the schema of the tuples but just allows tuples meeting certain conditions, in this example *Age* greater than 20.The map operator on the other hand produces new tuples with a different schema.

2.1.2 Stream Operators

A stream operator is a defined function that takes in a stream of tuples or streams of tuples and produces tuples(or have some other side effects e.g saving to disk). Stream operators are categorized into two groups namely *Stateless* and *Stateful* operators. *Stateless operators* perform a per tuple computation based on the current tuple only. Hence each tuple is processed without maintaining any dynamic state. *Stateful operators* on the other hand maintain dynamic state which evolves as tuples come in and use this state as the basis of the computations which affect the output tuples produced.

Stateless Operators

These operators do not maintain any internal state in processing the tuples. An operator of this kind performs computation on a tuple-by-tuple basis without storing or accessing

data structures created as a result of processing earlier data. The non-dependence property of these operators makes them easy to be parallelized. Below is a list and discussion about the most common stateless operators.

- **Filter** - Like the name suggests this operator filters out all tuples that do not meet the user set conditions. Thus the operator takes as input a tuple and if the condition passes outputs a tuple with same schema and values, otherwise it is discarded. Some variants of this operator can also route tuples according to the filter conditions hence producing multiple output streams.
- **Union** - The Union operator combines multiple streams into one. Thus it takes two or more different streams of tuples having the same schema and produces one stream as output. The basic implementation of this operator does not guarantee any ordering of the output tuples except that tuples from a particular stream will have their respective ordering maintained. However researchers have proposed [14, 15] and developed more advanced versions of this operator that tackle this issue and offer ordering semantics.
- **Map** - This operator takes in a tuple, perform computations on it and produces a new tuple. Since this operator adds, drops fields or does a total transformation of the tuple, the schema of the produced tuple is different from the input tuple.
- **FlatMap** - This is similar to the Map operator except that in this case from a single input tuple multiple output tuples are produced.

Stateful Operators

These type of operators create and maintains state as the incoming tuples are processed. In this regard the state, along with its internal algorithm, affects the results the operator produces. Thus the state maintained by the operator is usually a synopsis of the tuples received so far with respect to some set time or tuple count window. Operators falling in this category are generally of an aggregate nature in that they take in multiple tuples, performing computations on them and produce a single tuple as output. In general any operator that saves dynamic state and uses this state in the computations performed on the tuples fall in this category. Below are some examples of stateful operators

- **Join** - This operator combines two streams by correlating the tuples of these streams based on a specific match predicate. For instance the tuples can be matched on a particular field. The output tuple is composed of the two matching tuples. This operator is stateful in that it keeps track of the unmatched tuples from both streams. When a new tuple arrives, it is compared with the unmatched tuples from the other stream to see if there is a match. A match triggers the generation of an output tuple. Researchers have also proposed more advanced versions of this operator [16].

- **Sort** - This operator sorts tuples in the stream according to some defined ordering. This ordering may be based on a single field or a set of fields. Having as input tuples in any order the output tuples are ordered according to the defined ordering.
- **Average**, **Min**, **Max**, **Count**, **Sum** - These operators are implementations of the common math functions of the same names. In each of these operators a state is maintained in order to produce the same behavior of their math counterparts.

In all of these cases, multiple tuples are used as input before producing the output. Basically any operation that processes more than one input tuple before producing the output falls in this category.

In practice Stateful Operators have a defined *window size* W_S and *window advance* W_A which determine the tenure of a particular tuple's effect on the state. These parameters are either specified in time(seconds,minutes etc) or tuple count depending on whether the window is time based or tuple based respectively. However the former is more common in many existing solutions [17, 18, 19]. W_S determines the scope of the window; for instance in a time based window, this is the size of the time range t_{Start} to t_{End} within which all tuples with a time stamp falling within this range are aggregated. On the other hand W_A specifies the shift the window has to move to the next time range when the current range has expired. Thus given the current range as t_1 and t_2 the next range will be $t_1 + W_A$ and $t_2 + W_A$ respectively. A window expires when the time stamp of the current tuple is greater than t_{End}. Depending on the values of W_S *and* W_A a window can be classified into three categories:

- Sliding Window - when $W_A < W_S$. In this scenario consecutive windows overlap since at every window expiration the following window is defined by adding W_A to the range [t_{start},t_{end}) where $t_{end} - t_{start} = W_S > W_A$.
- Tumbling Window - when $W_A = W_S$. Unlike the sliding window, consecutive windows do not overlap. It should be noted that upon window expiration the new window range's start is equal to the old window end i.e $tnew_{start} = told_{end}$.
- Jumping Window - when $W_A > W_S$. Since the advance is greater than the window size two consecutive windows have a gap between them. Any tuples with time stamps falling in this gap range are discarded.

2.2 Java 8 Stream

The 8th release of Oracle's Java created a lot of momentum around the language [20, 21]. This release brings a set of features that are shifting the way developers use the language. Brian Goetz in [22] provides a good technical overview of the new features. With the move to a functional paradigm, this addition was even more welcomed as research suggests that Object-Oriented programming languages should support other

programming paradigms to make them more appealing to developers and easier in solving a wider range of problems [23].

The Stream API [10] introduced in Java 8 promises to ease the task of processing data in a more declarative manner than it was done before. Up to Java 7, developers had to rely mostly on collections and *"for"* loops to iterate over the elements of data. Such an implementation is rarely optimized for huge datasets and depends solely on the developer's skills to efficiently utilize the cores of today's typical multi-processors machines. In this section, we present few features of Java 8 that constitute the basis of our thesis.

2.2.1 Functional Interface

In object-oriented languages, an *interface* refers to an abstract type that contains no actual data or instructions but defines the behavior of such a type through the method signatures. A class that provide logic for all the methods defined in the interface is said to *implement* such an interface. Java has defined a number of well knows interfaces that are really practical for a developer. We can cite interfaces like *java.lang.Runnable, java.util.Comparator*. Very often, these interfaces have only one abstract method, thus the name Single Abstract Method (SAM) interfaces. In Java 8, a *SAM* interface is replaced by a *functional interface*. These interfaces are not just a name substitute for *SAM*. They bring major capabilities in type declarations, object instantiation and support for lambda expressions. As an example, the *Consumer*$<T>$ is a functional interface that applies an action on objects of type T.

2.2.2 Lambda expressions

A *lambda expression* - often called anonymous function - is a function that is not tied to an identifier, to be executed at a later time. As an example, $(int\ a, int\ b)$ -> $a * b$ is a lambda expression that receives two integers as arguments and returns their product as a result. They can appear anywhere an expression can be evaluated and are typically used as a parameter or return type to other more conventional functions. This concept was already dominant in functional programming languages such as Haskell and Erlang [24, 25], where functions are seen as a *first-class entity* i.e. they could be stored and passed around just like any other primitive entity. In Java 8, this addition is closely related to the *functional interface* type. Instances of a functional interface can be created with lambda expression, method references, or constructor references as defined in the Java specifications.

The usefulness and capability of lambda expressions can be further appreciated in the publication of Franklin et al. [26]. It proposes *LambdaFicator* as a tool which automates the refactoring of anonymous inner classes to lambda expressions. It converts *"for"* loops that handle iterations over collections to functional operations using lambda

expressions. Other languages [27, 28] that are also JVM-based have integrated long before these concepts into their object-oriented paradigm.

2.2.3 Internal & External iterations

Collections are really important structures involved in virtually any software design. They often constitute the basis of algorithm and system design. While accessing elements efficiently is primary requirement, iteration over the elements is a fundamental design issue in many programming languages. One interesting problem is how to reconcile iteration with actions over the elements.

An iteration is said to be *external* when the developer controls the traversal of the elements. The developer explicitly defines how to access each element in the data structure and applies the desired action on it. On the other hand, an iteration is said to be *internal* when the developer only provides the operations to process the elements, without caring about how the elements are accessed and provided.

Internal iteration, widely accepted in functional programming languages, is gaining more ground in object oriented languages. It is attractive for the opportunities it offers to the compiler to optimize execution over the cores of the machine and provide any cleanup mechanism needed in the background. Processing of data is made easier, since they can be allocated in parallel among the cores. Thomas Kühne [29] shows the early trend to adopt internal iteration in languages that did not support *closure*, a data structure that encapsulate a function together with an environment [30]. It explores the problem of iteration in more depth and provides a framework that encapsulates a powerful internal iteration for the user to process data efficiently.

2.2.4 Stream Interface

The stream interface shipped in Java 8 provides a number of operators and utility methods to perform analysis on a stream of data. A *stream* is defined as a sequence of elements supporting sequential and parallel aggregate operations[10]. The elements can be generated from *Collections* already defined in Java or from any object implementing the *Supplier* interface, defining the mechanism to get the data from the source. The data goes through a series of *intermediate operations* that return another stream of data and finally a terminal operation which produces a final result or performs an operation on another system.

Operation on streams adopts a lazy evaluation mechanism in Java. Lazy evaluation is an optimization method of computing which delays a computation step until it is necessary [31]. A number of research prove the importance and efficiency gained with lazy evaluation [32, 33]. In the Java stream pipeline of operations, the stream starts the computation only when a terminal operation is hit. This allows the compiler to optimize how the data goes through the pipeline. In reality, a tuple doesn't pass through

one operator and go to the next one. All the operations are rearranged into one loop where each tuple go through all of them in one pass and data is pulled as needed. This approach minimizes, if not eliminates, buffering between the operators and ensures minimum latency. It also gives opportunities to parallelize the processing by dedicating different threads in a *fork-join* pool to process data separately. This mechanism is useful when the dataset is huge. However contention on the data source may arise, as a number of threads may be querying the source at the same time.

Streams can be operated on only once. It means that the stream object cannot be reused once a terminal operation is called. An *IllegalStateException* is thrown if the stream is reused. Listing 2.1 shows an example of a declarative processing on a collection of people objects. The collection is converted into a stream object in the first line and then processed to get the count of all women in the collection. In this example, the collection is processed in parallel.

Listing 2.1: Example of Java stream processing

```
int sum = employees.stream()
   .parallel()
   .filter(p -> p.getSex() == "Female")
   .count();
```

2.3 Messaging Systems

Stream processing in general relies on buffering mechanisms between producers and consumers. This is to ensure that no data is lost between processing applications and preventing a consuming application from being overflown with data when the production rate is faster. Typically queuing systems are used for buffering this data as they provide useful semantics. For instance some queues can provide message read guarantees in that a message is only deleted from the queue if the consumer acknowledges that it has fully processed it. Using such a feature an application can gain this semantic without implementing it directly into the application. In this thesis we employ the use of queuing systems and provide abstractions to some of them. Thus in this subsection we present queuing systems that are used or supported in JxStream.

2.3.1 RabbitMQ

RabbitMQ [34] is a messaging broker developed by Pivotal Software Inc, both open-sourced and commercially supported. It is supports many messaging protocols such as the Simple Text Oriented Messaging Protocol (STOMP)[35], the open standard Advanced Message Queuing Protocol (AMQP) [36]. RabbitMQ provides a number of features that make it widely accepted in the industry. Among other features, we can cite

clustering, complex message routing, support of a variety of messaging protocols and programming languages, reliability, availability. Rostanski et al. [37] discusses the performance of RabbitMQ as a middleware message broker system.

RabbitMQ can be used in a variety of scenarios. It can be used as a work queue to distribute messages among multiple workers; a publish/subscribe queue to send the same message to all consumers of the queue; a routing queue to selectively send message consumers; a topic queue to distribute messages based an attribute and finally a Remote Procedure Call (RPC) queue for clients to execute a command on a server and get the result on a queue. Figure 2.2 shows an illustration where RabbitMQ is used in a publish subscribe scenario. The producer (P) sends messages to an exchange (X), which decide how to send the message to the queues where the consumers (C) subscribe.

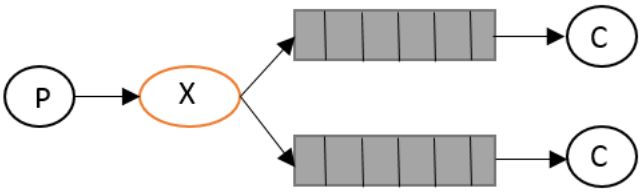


Figure 2.2: Publish/Subscribe scenario of RabbitMQ

2.3.2 Apache Kafka

Kafka [38] is a publish-subscribe messaging system initially developed by LinkedIn [39] and maintained by the Apache Software Foundation [40]. Originally designed as a distributed messaging system for log processing [41], it became very popular for the features it provides and its performance. Kafka is distributed in essence. It is built for durability, scalability and fault-tolerance. Kafka is maintained in clusters of computer nodes with one or several nodes serving as brokers. In case of failure, a new leader is quickly elected in the cluster to minimize down time and ensure consistency. Kafka uses Zookeeper [42] - a reliable distributed coordination service - for election and other meta-data bookkeeping.

Kafka organizes messages into topics, replicated and partitioned in the cluster. Each partition is strongly ordered and holds the messages for a configurable period of time whether they have been consumed or not. *Consumers* of data subscribes into topics and are organized into groups of consumers. Kafka delivers each message to all the groups but only one consumer in a group will consume the data. Kafka can control how messages are committed or the developer can choose to do so. Kafka programs can be written in many languages, which makes it even more attractive to developers. Figure 2.3 shows the overall architecture of Kafka where *Producers* push data to topics in the cluster for the consumers to use.

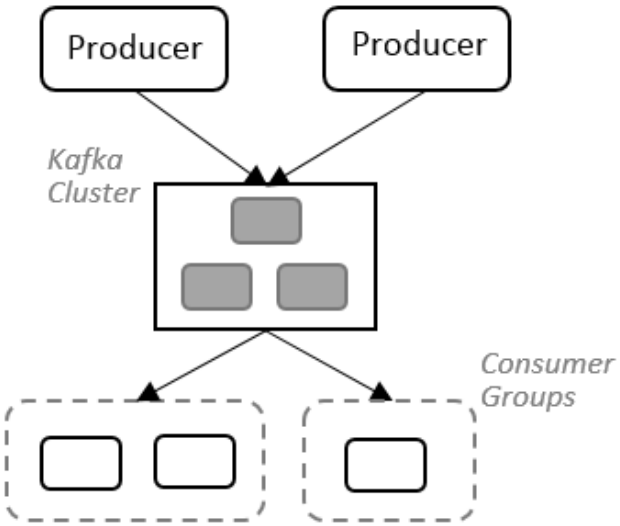


Figure 2.3: Kafka high-level architecture

2.3.3 Disruptor

Disruptor is a Java library that provides a concurrent ring buffer data structure for use in inter thread communication. The library was developed by Thompson et al. [43] at LMAX Exchange after research in high performance concurrent structures. The research focused on identifying the issues that slows down concurrent executions in modern systems. From this work the team designed a lockless architecture putting into account issues like cache friendliness, memory barriers and memory allocation. We refer the reader to their publication [43] for more details about the design.

Like messaging queues *Disruptor* is used in a publish-subscribe pattern where there can be multiple produces and consumers on one ring buffer. However in this case the producers and consumers are threads within the same process. Consumers of the same buffer view the same set of messages. Producers enter a blocking state when the buffer is full as the result of consumers being slow in reading the messages. *Disruptor* is useful when creating applications consisting of different threads exchanging data. Thus it can be used as a general inter-thread messaging transport in different application architectures.

3

Problem Description

In this chapter we discuss in detail the problem this thesis is addressing and the evaluation plan for the solution we provide. We first present the shortcomings of the Java 8 Stream package when applied to data stream processing in section 3.1. We then discuss how we plan to evaluate the solution we provide (*JxStream*) in section 3.2.

3.1 Java 8 Shortcomings toward Stream Processing

The Java 8 Stream package is well suited for batch processing, e.g operations on in-memory collections, disk files or any fixed-size dataset. The operators and the underlying stream construct provided by the API does not satisfy the typical needs of data streaming applications. These needs are centered on the fact that the data to process is unbounded. Below we discuss the inadequacies of the API when applied to such scenarios.

Lack of Data Stream Semantics

Data stream applications use certain semantics that enable the continuous input, process and output of data. While Java 8 Stream provides an abstraction that represents data as stream it still lacks some of the fundamental semantics of data streaming. In the following we discuss core concepts of data streaming that Java 8 Stream does not provide.

- **Aggregate Operators** Java 8 provides a number of operators that can be used to process a stream of data. The general approach is for each operator function to take in a user defined per-tuple action as a lambda expression. As the package was designed for processing over collections of data, most of the operators provided are of a stateless nature. There are a few stateful operators included in the package

(*sorted, limit, distinct, reduce*) which require a total view of the whole dataset. A typical requirement when implementing data streaming is the aggregation function. This can range from a simple counting or averaging functions to more complex user defined operations. The Java 8 Stream API currently does not provide a general aggregation function that can allow programmers to build upon custom operations (stateful) that suit their needs. Moreover it does not support definition of windows (Windowing) on operators since the data is expected to be bounded.

Windowing is an important concept when dealing with an unbounded data stream and stateful operations. It allows a programmer to break down the continuous flow of data into smaller groupings for processing together. Applications that produce an output over an interval of time or tuple counts typically require a windowing mechanism. Providing aggregate operators that are coupled with a windowing capability would be an important step towards supporting data streaming.

- **Stream Branching** Typical data streaming applications perform multiple queries on the same input stream. For example from a single stream of temperature and moist sensor readings, an application can be written to perform two distinct analyses on the same stream; track the average room conditions on a per 30 seconds basis and also perform a per minute correlation between the current readings and the number of people in the room. The results from the former can be targeted towards a dashboard while the latter could be used in a more complex operation downstream. To implement such a requirement a programmer would need to use the same upstream source between the two operations. The Java 8 Stream only supports a single chain construction. The API provides no way of branching out streams from one single stream. Thus this operation cannot be implemented right out-of-the-box using Java 8 Streams.

- **Multiple Queries** Similar to *Stream Branching* data streaming applications sometimes may be required to perform multiple analyses on multiple streams with distinct sources. This means in one application a programmer may define and run multiple stream pipelines that run independently of each other. Java 8 Streams are synchronous in that when they are called upon to start executing (by invoking a terminal operator) the calling thread is blocked until the current processing completes. This design works well when dealing with fixed datasets since the processing is bound. However when applied to unbounded datasets this creates challenges. It means that when a single stream pipeline is invoked to run, the calling thread is blocked until the application terminates. Thus no additional pipelines can be executed in this application. One solution to this would be to have the programmer launch a new threads per defined pipeline and subsequently call each pipeline in its own thread. Though this mechanism works, it complicates the use of the API as the programmer would also need to manually create and track these threads.

Parallelization Strategy

Java 8 Stream parallelization is based on the *ForkJoin* framework which expects a data source that can support recursive decomposition. The implementation of this framework is based on the work of Doug Lea [13]. In this strategy a problem (dataset) is recursively split into sub problems which are parallelized. The main algorithm of the framework is illustrated by the pseudocode in listing 3.1.

Listing 3.1: ForkJoin Pseudocode (Adapted from [13])

```
Result solve(Problem problem) {
 if (problem is small)
 directly solve problem
 else {
 split problem into independent parts
 fork new sub-tasks to solve each part
 join all sub-tasks
 compose result from sub-results
 }
}
```

Thus using this mechanism the Java 8 Stream parallelization expects the programmer to provide a data source that can support this operation. This requirement is natural to fixed size datasets since the size is finite and already known before hand. However when dealing with unbounded streams the dataset is not of a fixed size and thus cannot be split as required. Even though with effort a work-around solution can be devised to adapt unbound data sources to conform to this requirement, such a solution would be susceptible to inefficiencies. It would entail the use of locks on tuple generation to control access to the shared data source. This would lead to a high thread contention that would result in a performance hit. Additionally different data sources may require to be treated differently in order to optimize the processing. The Java 8 Stream mechanism does not support this flexibility. In chapter 5 section 5.3 we present our approach to parallelization.

Distributed Environment

Since the Java 8 Stream package is designed for fixed side datasets all the provided out-of-the-box data source readers and writers are targeted towards such sources. For instance Java collection classes and the *FileStreams* have a simple mechanism of converting into streams. Usually data stream applications are run in an environment where multiple instances of the application could be run in a distributed fashion. This require the applications to read data from a common repository; typically a Messaging System. Given such a scenario, a programmer using Java 8 Streams would need to implement readers and writers for the Messaging systems. Thus the API is no longer as simple when used

in data streaming. Having the same level of support for common data streaming data sources as is with fixed sized datasets would be very useful to data stream programmers.

3.2 Evaluation Plan

The solution this thesis provides covers the problems mentioned above. We break the evaluation of our solution into three parts;

- First we measure the absolute performance of the system. Here we are interested in establishing the overall performance benchmark of *JxStream* running as a single process. To have a frame of reference, we compare its throughput and latency with a fully fledged SPE given a simple use-case scenario. We also compare the overhead cost the two systems incur as more operators are added to the pipeline. The experiments in this category enable us to have a general perspective of *JxStream*'s performance in data stream processing. The importance of providing a solution but not compromise on the performance cannot be emphasized enough.

- Then we focus on the scalability of *JxStream*. The aim is to investigate how well *JxStream* utilizes the cores of a computing node. As modern computing devices are multicore, it is essential that applications and systems alike take advantage of these cores. In this category we use a CPU-intensive application to evaluate how *JxStream* performs when the parallelization factor increases. This helps us to predict how *JxStream* would perform given different execution environments.

- Finally we evaluate the two aggregate implementations we provide. One implementation is geared toward general use-cases where one single operation is applied when the window expires. The other implementation is applicable to use-cases where an intermediary action - a *combiner* - is used before applying the final operation at window expiry. We refer the reader to chapter 5 section 5.1 for details about these implementations.

4

System Design and Architecture

This chapter describes JxStream's design and its modules. Figure 4.1 shows an overview of the architecture of the system. JxStream is composed of three main components. Section 4.1 presents the interface module that the developer interacts with. Section 4.2 presents the pipeline engine where the programmer defined computational pipeline is created and executed. Section 4.3 presents the data adapter module that interact with the input and output data store.

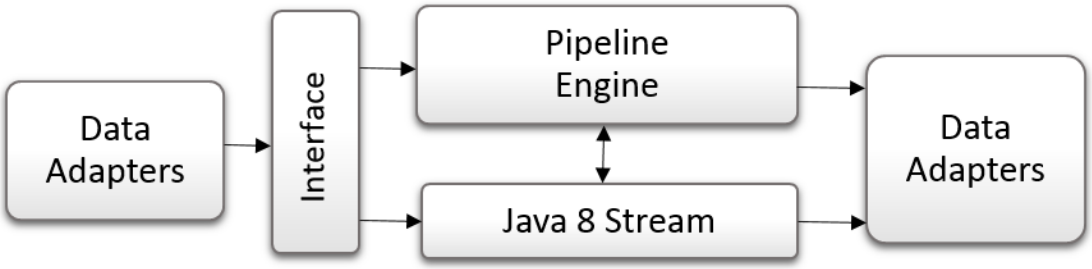


Figure 4.1: An overview of JxStream's Architecture

4.1 JxStream Interface

The JxStream interface is the primary point of contact between the developer and the system. We put a particular emphasis into its design to keep it simple while providing expressive operations. This section explains the programming model and the operations supported by the interface.

4.1.1 Programming Model

JxStream programming model is an abstraction that allows the user to manipulate streams of data in a simple and powerful way. The interface closely follows the Stream interface that was introduced in the Java 8 release [10]. From the very first concept, *JxStream* was thought of to reuse the existing expressiveness of the streaming package. We provide an interface that requires a flat or little learning curve to use the system. The programmer typically reuse operators, lambda expressions and functional interfaces concepts already familiar with from the package or from functional programming languages such as Scala [27] to express the computation over the data.

This programming model was also designed to make the system pluggable to the user's need. The programmer is expected to define certain configuration objects as a setup step for the system to work. As an example, there are consumer and supplier interfaces defined in the package that dictates how data is received and emitted. There are also certain queuing mechanisms that are provided out-of-the box. Nevertheless, the programmer can extend these interfaces to support the tools that fit his/her needs

4.1.2 Operators

Operators are the basis of our system. The Java 8 Stream package already offers some useful operations that can be applied on data. A handful of stateless operators are already defined in the package and are sufficient for data manipulation over collections. We have selected the operators that provide more useful insight in a data streaming perspective and complemented them with stateful operators. With these pieces glued together, *JxStream* provides a set of operators straight out of the box that increase the user's capability and productivity. Among the aggregate operations we have designed, we distinguish a custom *aggregate* that expects an operator to apply on each incoming tuple, a lambda to extract the field of interest and a window object to aggregate the tuples in. There are also options to pass in a finisher operation to apply when each window expires and a group-by capability. We have also designed aggregate operations such as *Min, Max, Median, Average*, and so on. The modularity and flexibility of the system is aimed at making it fairly easy to implement complex computations using the API.

4.1.3 Expressiveness

Another key focus of this thesis is to present a system that is intuitive and expressive enough for stream processing in a big data environment. We have followed the stream paradigm of Java 8 and maintained it across the operators and the classes we have designed. Listing 4.1 shows a snippet of code to give a first look at the expressiveness of JxStream. Further code examples will be provided later on this document. In this example, we create a window object that aggregate operation requires. We then generate

a stream of *Double* values from a supplier and perform an *average* operation after filtering the stream. We note that a long chain of operations could also be applied on the stream.

Listing 4.1: Expressiveness of JxStream

```
Window window = Window.createTupleBasedWindow(winSize, winAdv,
    defaultVal);
JxStream<Double> stream = JxStream.generate(SupplierConnectors sup);
JxQueue<Double> queue1 = stream.filter(x-> x>0 )
                               .average(x->x, window)
                               .subQuery();
```

4.2 Pipeline Engine

The *Pipeline Engine* is the core component of JxStream. When a programmer constructs the processing pipeline using the API, a conversion happens underneath to interpret the defined pipeline into the actual executing units that perform the computations. The constructed units will be launched as a Pipeline Engine in a Java Virtual Machine. The general approach we took in the design of the engine was to separate the conceptual pieces in a way that is less coupling and flexible enough. This enables an easy construction of the underlying complex data flows. Figure 4.2 shows a detailed view of the *Pipeline Engine* in action. At a high level the engine is made of two fundamental components; processing units called *JxPipes* and specialized internal queues which facilitates data flow between the different *JxPipes*. In the following subsections we expand more on the make up and approach of these components.

4.2.1 Stream Pipe

A *Pipe* is the single processing chain composed of one or more operators joined together. In essence this component performs computations and transformations on data coming through its source interface and outputs the results through its sink interface. Thus, with an operator represented as OP a *Pipe* P can be formally defined as:

$$P = \{OP_1 + OP_2 + OP_3 ... + OP_n\} \tag{4.1}$$

From the above construction it can be noted that P is a function of the form $P(t_n) = output_n$ over the *nth* tuple. Using a *Pipe* introduces a layer of abstraction over a complex series of operators and only exposing the source and sink interfaces as the means of interaction with the outside. Having such a modular design adds a lot of flexibility to the system while also simplifying the implementation. For instance in many situations parallelizing the computations might be desired so as to increase the throughput of the system. This can easily be achieved by having multiple identical *Pipes* having the source and sink interfaces connected to the same input and output respectively.

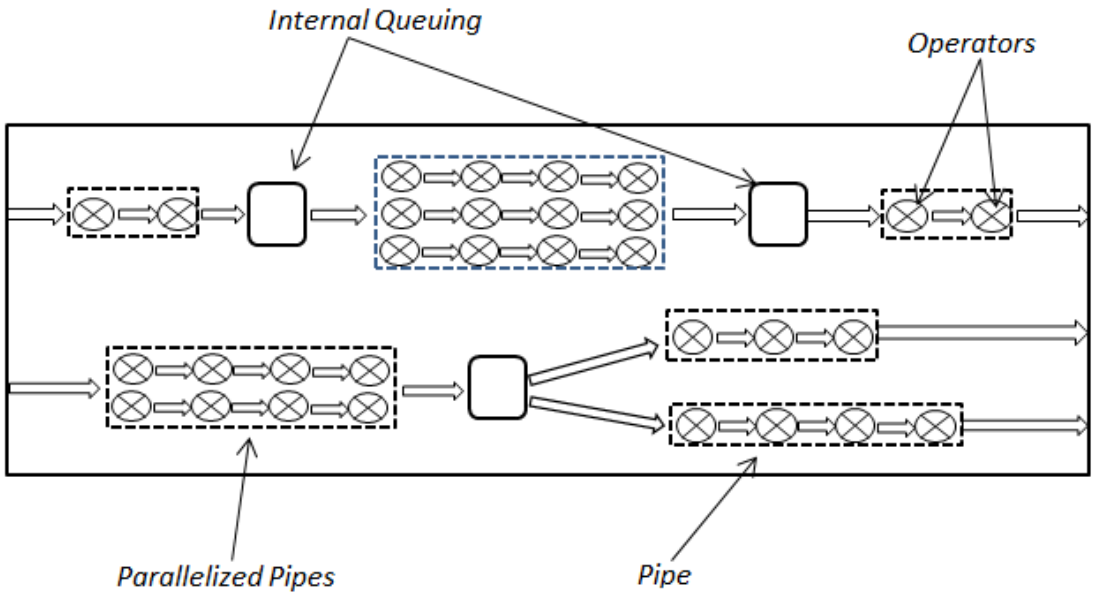


Figure 4.2: Pipeline Engine

4.2.2 Queuing

Joining independent executing units together requires the units to exchange data through some form of message passing. Similar systems with such a need in research [5, 44] have employed queuing as a mechanism for data exchange. We noticed how less complex our design would be with the use of queuing. This frees us from the complexities of defining protocols and interfaces to handle this. Thus the internal queuing does not need to have any logic to filter or route data but only serve one purpose which is to act as buffers between producer and consumer pipes.

4.2.3 Stream Graph

The joining of two or more *Pipes* together results in a graph-like data flow structure we call *Stream Graph* SG. This serves as the highest level of abstraction of the computing units, with defined input sources and outputs. From a high view this is simply a set of pipes whose stream of data is related and interdependent in some way. Hence similar to equation 4.1 an SG can be defined as:

$$SG = \{P_1, P_2, P_3 ... P_n\}$$

Depending on the nature of the construction, a single graph can have multiple inputs and/or outputs. Furthermore each of these external interfaces can have different tuple schema depending on the Pipes consuming from or producing to them. For this reason

SG is a function of the form

$$SG(t_a,t_b...) = output_w,output_x,output_y.....$$

$where schema(t_a) \neq schema(t_b), schema(output_w) \neq schema(output_x) \neq schema(output_y)$

This approach in the design of SG gives JxStream great ease and flexibility in tackling complex stream analysis scenarios.

4.3 Data Adapters

A data adapter is a module that specifies how data is transferred between data stores and JxStream. As a convention, a set of functional interfaces are provided to give the user the freedom to choose the data source JxStream is reading from and stores it is writing to. The user simply have to implement these interfaces by controlling the logic of how the data should be retrieved. We also noticed that most stream processing infrastructures make the use of messaging systems such as Kafka or RabbitMQ [34, 38]. In that regard, we provide adapters that are readily available in JxStream to interface with these queuing systems. The user then only have to provide the necessary configurations. Future work of this thesis can extend these adapters to support more data stores.

5

Implementation

In this chapter, we present the implementation work we carried out to build JxStream with reference to the design we presented in the previous chapter. Thus addressing the challenges we have outlined in the problem definition section of Chapter 1 and detailed in Chapter 3. We start off the discussion with the implementation details of Aggregate operators in section 5.1. Here we focus on the underlying infrastructure for aggregates as opposed to individual operators. Next we look at how the JxStream's stream pipeline abstraction is implemented in section 5.2. In section 5.3 we look at how the stream parallelization is achieved. We then discuss how the functionality of merging multiple streams into one is implemented in section 5.4. In section 5.5 we conclude the chapter with a look at stream branching.

5.1 Aggregate Operators

Java 8 Stream lacks the support of common data stream stateful operators usually referred to as aggregates. Since the stream is unbounded these operators are specified with a computational window specifying which tuples are aggregated together. We refer the reader to the Background chapter subsection 2.1.2 for more details.

One of the main objectives of *JxStream* is to ease the construction of stateful operations while maintaining the level of expressiveness of Java 8 Stream. Implementing this atop Java 8 Streams requires building an underlying infrastructure that handles the intricate details of windowing. Thus to the user these details are abstracted away in a simple yet powerful API. Additionally since performance is always of concern in data streaming, this infrastructure should be efficient. We use two different approaches based on array and ring structures to achieve this.

Array-Based Structure

The first approach taken toward stateful operations was to provide a flexible API that can be applied to any problem of a stateful nature with ease. As stream processing can be applied to a wide range of applications and problems, it is essential that the user can manipulate and represent the processing data in ways that suit his needs. This design gives the user the freedom to specify how each tuple should be treated and how to aggregate information on the window after its expiration. Since the windows are meant for a general purpose, we maintain an array that holds all the windows and decide on each incoming tuple which windows it applies to. As a window is defined by its *size* W_S and *advance* W_A, the array size is calculated as:

$$Array.size = W_S/W_A$$

Figure 5.1 shows a representation of the windows with an incoming tuple. The dark windows represents those that will aggregate the incoming data while the others are not activated. We can also see that there is a cycle in the maintenance of the windows in the array. At the expiration of the first window, the resources at the first index of the array are initialized to hold the fifth window.

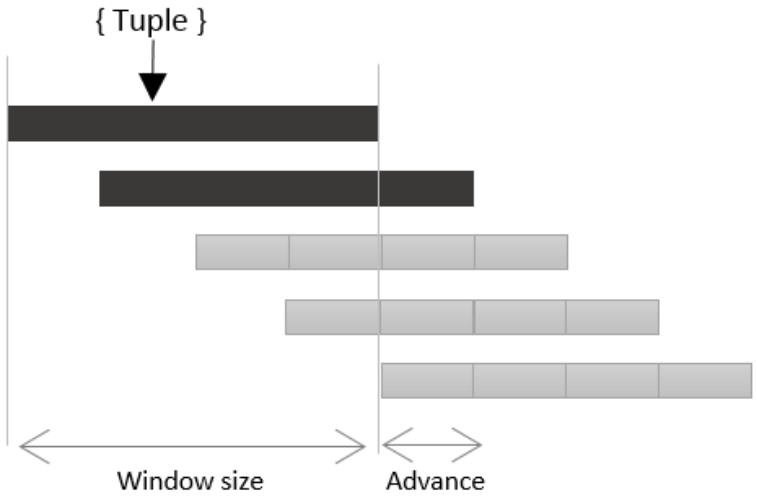


Figure 5.1: Operation on array-based windows

Ring-Based Structure

The approach described above is general and thus can be easily applied to all problems of a stateful problems. However the trade off for this generalization is the performance since on reception of a tuple, repeated per-tuple operations are performed in order to update all applicable windows. The *Ring-Based* approach optimizes the window operation by avoiding the repeated per tuple operations when a tuple is received. We adopted this

approach from the ideas of Michael Noll [45] in his proposal for performing sliding window aggregates. This step introduces a little complexity in dealing with certain classes of problems. In this design a window is represented as a ring divided into n slots. Each slot is defined by a range whose size is equal to the window advance W_a and thus with $window\ size = W_s$ the number of slots n(S) is defined as:

$$n(S) = W_s/W_a$$

Figure 5.2 gives an illustration of the ring used for computing time-based windows of size 60 seconds and advance of 10 seconds. At any given time one slot is designated as the current slot. This is the slot in which the last previous tuple was applied to. When a tuple arrives its timestamp (tuple count in tuple-based window) is checked to see which slot it applies to, if it falls in the current active slot, a user-defined per tuple function is performed on this tuple updating the state of the current slot. However, if the timestamp is greater than the current slot it means that the window has expired and needs to be outputted. The window value W_v is generated by performing a defined combiner function CF that takes all slots as input and outputs the window value. Thus

$$W_v = CF(S_1,S_2,S_3...S_n)$$

After generating W_v the current slot is reinitialized to the default state and the current slot pointer is moved to the next slot.

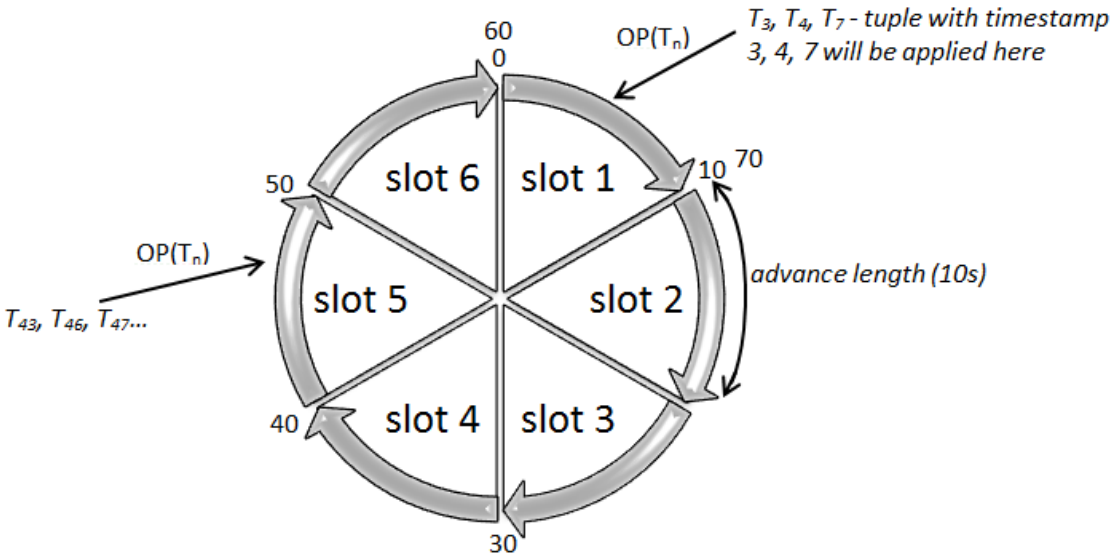


Figure 5.2: Ring used in computing windows. Per-tuple operator(OP) is applied to incoming tuples updating the respective slot.

5.2 Stream Pipeline

As we mentioned in Chapter 4, JxStream pipelines are composed of processing chains we call *JxPipes*. They basically represent a series of user defined computations that are to be applied on tuples. We build *JxPipes* directly on top of Java Streams, thereby using them as the provider of operator execution. Having Java 8 Streams as the building block enables *JxPipes* to benefit from the performance advantage Java 8 Streams have in the execution of operators. In this way Pipes are optimized in that a chain of operators making up a stream pipeline are all combined into one execution thread. Furthermore since part of the objective of our work was to maintain Java Streams' expressiveness as much as possible, taking this approach eases that task. It frees us from the complexities of building an efficient underlying infrastructure that supports an expressive API. Having Java Streams as the fundamental building block separates JxStream from other common stream processing solutions [6, 46] which usually isolate single (or a closely related group of) operators into different thread executions. Based on this fact we expect JxStream's performance to be superior than these solutions.

5.3 Parallelization

In order to achieve parallelism in JxStream we build a new mechanism not based on what is provided by the Java Stream API. We implement two types of stream parallelism that could be used depending on the scenario that needs to be parallelized. The first approach caters for situations where an already constructed single stream needs to be split up into multiple identical downstreams. On the other hand the second approach handles situations where stream parallelization starts from the data source. In figures 5.3, 5.4 we highlight the difference by showing an overview of the two approaches. Performance wise the latter strategy is superior since it is not constrained by any underlying stream splitting logic and internal queuing that is needed when splitting up a single. In the next subsections we expand more on these two strategies.

5.3.1 Stream Splitting Parallelization

There are scenarios where parallelization should be performed on a stream whose data source can only have one reader. An example of this is having a file on disk to which events are continuously written to as the data source. This would require a single reader as having multiple readers in such a case would create challenges in making sure reads are unique. Likewise in other situations a stream may need to be parallelized from a certain point in the processing chain. For instance it would be desirable to parallelize a CPU intensive downstream coming from an operator that cannot be parallelized. Both of these scenarios require a means of splitting a single stream into many parts and in turn running the parts in parallel to each other. The *Stream Splitting Parallelization* strategy can be applied in such situations. To achieve this we implement an internal

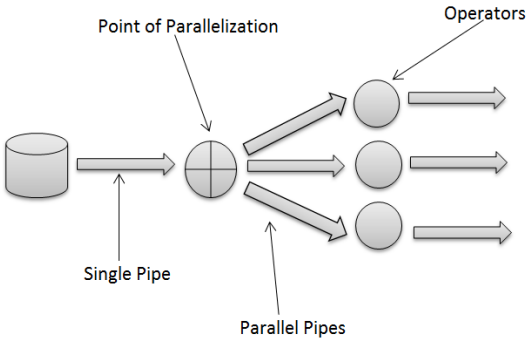


Figure 5.3: Parallelization by splitting a pipe. A single pipe is split up into multiple parts that are then parallelized

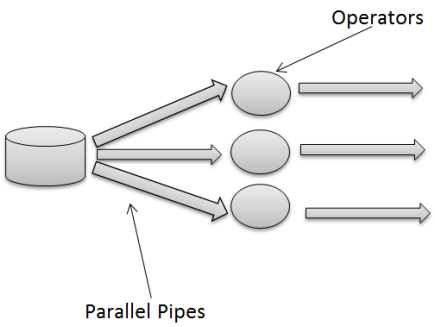


Figure 5.4: Parallelization starting from the data source. The source has multiple readers

Stream Splitting Function (*SSF*) to perform the actual splitting of the stream. With $JxPipes = P$ we define *SSF* as:

$$SSF(P,f,n) = \{P_1,P_2,P_3,....,P_n\}$$

The function takes in a *JxPipe* P, a *Flow Controller* f and the number of desired output streams n. The *Flow Controller* is a user-defined function that determines how the tuples are distributed among the split streams. As a default in JxStream parallelization, we use a round robin Flow Controller but this can be changed to a custom implementation defined by the API user. *SSF* outputs n *JxPipes* on which further stream operators are applied. It should be noted that only the operators which are applied on the output of the function will execute in parallel.

The *SSF* function builds a stream splitting infrastructure that is based on the flow controller. The actual flow of tuples is split by linking the upstream *JxPipe* to a tuple distribution object. We implement this object as a plain Java class implementing the Consumer Interface [47]. This object acts as the sink of the upstream *JxPipe* and interface to the output *JxPipes* at the same time. The object is in turn linked to the n output *JxPipes* via n ring buffers (LMAX Disruptor [43]) that acts as input queues to the individual *JxPipes*. The upstream *JxPipe* and the distribution object are executed in one thread while the individual output *JxPipes* together with the computations added to them downstream run in separate threads.

When tuples come through the upstream *JxPipe*, they are consumed by the distribution component. This in turn uses the *Flow Controller* to determine the next buffer in which to put the tuples. Decoupling the Flow Controller from the distribution component gives us the flexibility of specifying which tuple goes in which parallelized output stream. In order to make sure that these details are transparent to the API user, the whole chain consisting of *JxPipes*, distribution and buffers are abstracted in one JxStream pipeline. Applying *Map* or *Filter* operations on this pipeline will translate in applying the same operation on all the multiple *JxPipes* underneath.

5.3.2 Stream Source Parallelization

This approach optimizes parallelization by eliminating the tuple distribution and internal queuing mechanisms of the previous approach. It is well suited for scenarios where the stream data source can accommodate multiple readers. Basically n number of JxPipes with data adapters pointing to the data source execute in parallel. One assumption made here is that the read semantics of the tuples are handled by the data source itself or the provided data source adapters. This entails that the parallelization infrastructure does not handle deduplication or making sure that a tuple is read only once but rather will consume whatever the source produces.

Having multiple readers to one data source demands that the user supplies n distinct readers to the API. Instead of requiring the user to pass in a list of n data source providers, we took a different approach by using the Factory Pattern. We only require the user to supply the data adapter construction code as a lambda function to the parallel function. Since this lambda function creates a data adapter on demand we take a lazy approach by deferring the actual creation of the data providers to the time they are needed. The resulting pipeline from this parallel mechanism consist of n threads each executing a *JxPipe*.

5.4 Stream Merging

Parallelizing stream computations helps to increase the performance of the pipeline by making use of the cores on the machine. However there is a limitation on which stream operators can be applied on parallel streams. As a general rule, only stateless operators can be safely parallelized. Since most stream pipelines consist of a combination of stateless and stateful operators, this entails that parallelization cannot be performed on many common scenarios. A solution to this is to parallelize the stateless operators and use a *Union* operator to merge the parallel streams into one before applying a stateful operator. Thus the importance of the union operator in stream applications cannot be be emphasized enough.

We take a simple *First-in First-out* (FIFO) approach in implementing this functionality. Merging multiple streams using a FIFO approach results in a downstream with no strict ordering on the tuples. This is because tuples are pushed to downstream in the order they arrive. Since there are multiple independent streams involved this order cannot be predetermined. Therefore our solution does not offer ordering guarantees as part of the merging function. However if such an ordering is desired, the API user can define and add to the downstream an *Aggregate* function that performs this ordering. Figure 5.5 depicts the implementation of the merging function. To merge the n *JxPipes* into one we create n consumer objects that act as data sinks of the *JxPipes*. All the consumer objects are linked to the same *Disruptor* object consequently adding to it the tuples they consume from the *JxPipes*. Having this creates a multi-producer scenario

where access to the *Disruptor* object's buffer is contended. However owing to lockless architecture of *Disruptor* the impact of this contention on performance is not huge. We then create a new downstream *JxPipe* that reads tuples from this buffer and output this *JxPipe* as the merged stream.

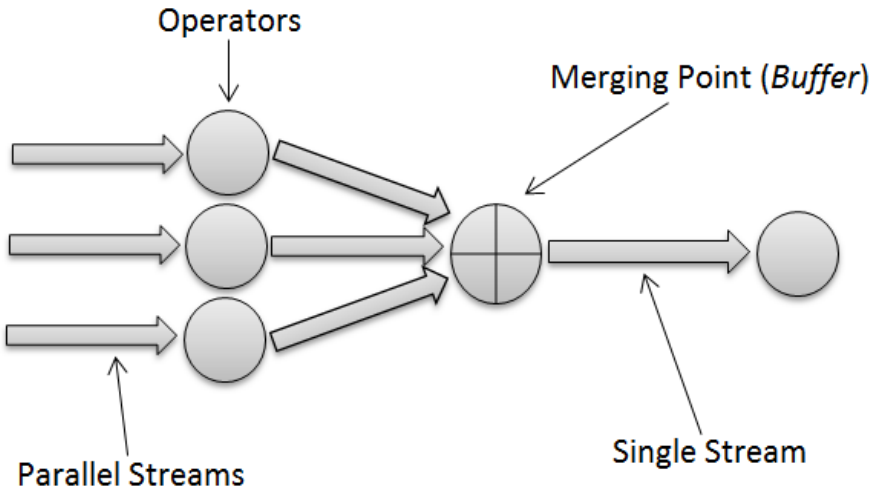


Figure 5.5: Stream merging. Parallel streams are merged into one stream using buffer.

Since this solution creates a fan-in flow of data, it is possible for the downstream to process tuples at a rate slower than the upstream in certain situations. In such a scenario it would be necessary to increase the size of the buffer to alleviate the pressure applied by the upstream *JxPipes*. We give the user an option to give this size when calling the union operator. In all our test scenarios we found it adequate to use the default size of 1024 elements.

5.5 Stream Branching

Support for sub-queries is an important attribute of our API as this enables the creation of graph-like stream pipelines. Usually in stream applications there is need for running more than one computation pipeline on intermediary results. In Java 8 Stream only a single pipeline can be constructed thus branching is not supported. Owing to this JxStream's *JxPipes* suffer the same limitation since they are built on top of Java 8 Streams. To overcome this, we implement a *Subquery* function which when applied on a stream creates a temporal repository of the intermediary data and outputs an access provider that can be used as a data source to new stream pipelines. We give the user the power to specify the type of repository to use when invoking the function. At the writing of this document JxStream supports three types of repositories: Disruptor, RabbitMQ and Kafka.

From an implementation stand point, the strategy used here is similar to the approach

used in stream merging for the union operator. When *Subquery* is called on a stream (single or parallel) n consumer objects are created just as in stream merging. However in this case the consumer objects are linked to the respective repository the user specifies. After creating and linking to this repository, an access provider is constructed and given as the output of *Subquery*. It should be noted that the access provider is not a data reader itself but rather provides distinct data readers when creating new streams. In this way the API user can use this object to create new streams that branch from this intermediary repository. All branching streams will have the same view and order of the tuples. Thus a tuple is fully read and discarded from the repository when it has been read by all branching streams.

6

Use-Cases

In this chapter, we proceed to show a variety of use-cases JxStream can be applied to and the expressiveness of its operators. In a stream processing environment, data is typically transformed as it progresses along a pipeline of computation. Data can also be correlated with other datasets in the pipeline. Sliding window analysis is typically used when analyzing unbounded streams. It is impractical to gain insight from the stream in its entirety as it is unbounded. Information are instead retrieved within a timeframe or a fixed amount of data. While JxStream is well-equipped for stateless operations, it was designed primarily to provide powerful stateful operations.

In the following sections, we present two applications of JxStream. These use-cases were chosen not only to show off some features of JxStream, but also for their relevance in real world scenarios. Section 6.1 presents the use-case of trending topics in Twitter. Section 6.2 presents an application that monitors a grid of sensors in real-time.

6.1 Twitter's trending topics

Analyzing trends in social media is an important task for a social media marketer. As a matter of fact, a number of tools are now available to perform data analytics on social media. Devindra Hardawar in [48] relays some of the best social media analytics tools as of 2013. The increasingly important presence of users on social media has pushed corporations to be more aware of their social media footprints and to measure how their products are perceived. Getting constant feedback about what is trending on the web is sometimes key to growth and strategic decision.

This use-case relates to the popular website twitter [49], an online social network service that allows users to interact with each other by sending/reading 140-character-

long messages called *tweets*. A *topic* in a tweet is highlighted by the use of the hashtag(#) character in front of a word and a tweet may relate to many topics. We present an application that reads tweets from a queuing system and produces a list of the n most popular topics in a time frame.

6.1.1 Data Processing

Figure 6.1 shows a sample of data processed in this use-case, with an emphasis on the data schema. The first table represents the input data to JxStream at a random rate. The timestamps appear here as a human readable date but it is represented in milliseconds in the system. The second table represents the output of the system, as a list of top five trending topics during time intervals of five minutes. For the scope of this

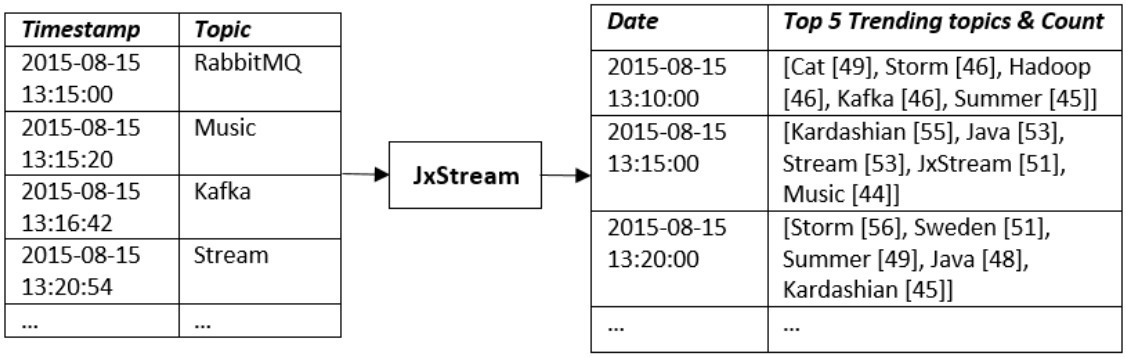


Figure 6.1: Input and Output data of the Twitter Trending Topics use-case

example, we created a tweet generator to feed Kafka with data. We consider an array of random topics and the generator will randomly pick one to feed it to Kafka with the timestamp of the machine at a speed of a tweet every 100 milliseconds. An example of data fed to Kafka is shown below, with topic name first and the timestamp between square brackets:

```
RabbitMQ [1438252882270]
Kafka [1438252882471]
Java [1438252882672]
Hadoop [1438252882873]
Stream [1438252883074]
Music [1438252883275]
```

The data structures needed for this scenario in *JxStream* are relatively simple. As *JxStream* handles much of the rolling window mechanism, we implement very few external classes. Much of the logic required is handled by lambda expressions and functional interfaces.

We encapsulate incoming tweets in a class called *MyTweets* with the necessary fields. We make this class *serializable* for object transmission over the queuing system by providing functional interfaces that control the de/serialization processes to byte object, as part of the requirements to instantiate supplier/consumer objects to connect to the queue. Figure 6.2 shows the representation of the *MyTweets* class in the Unified Modeling Language (UML).

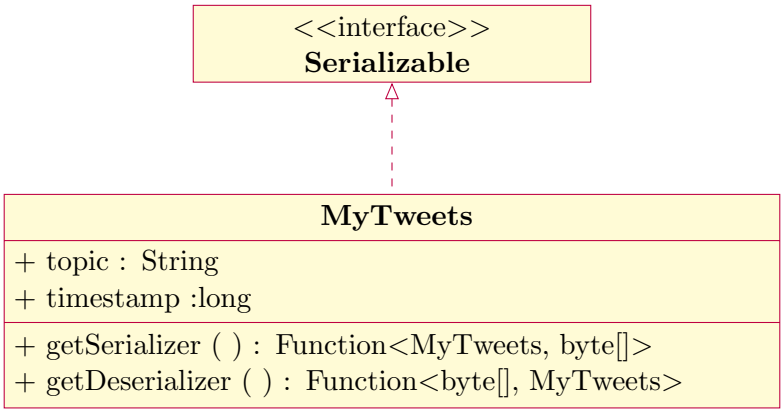


Figure 6.2: UML representation of *MyTweet* class.

The second class implemented, *RankObject*, is to represent the ranking of the topics. This class is implemented mostly for the easy sorting of the topics in a list based on their popularity. Figure 6.3 shows the UML representation of such class. The fields are simply the topic name and the count that tracks how many times the topic has been mentioned in the desired time-span. We implement the *compareTo()* methods from the *Comparable* interface to dictate the sorting mechanism in the list. Topics are sorted in descending order based on their count, showing the most trending topics first.

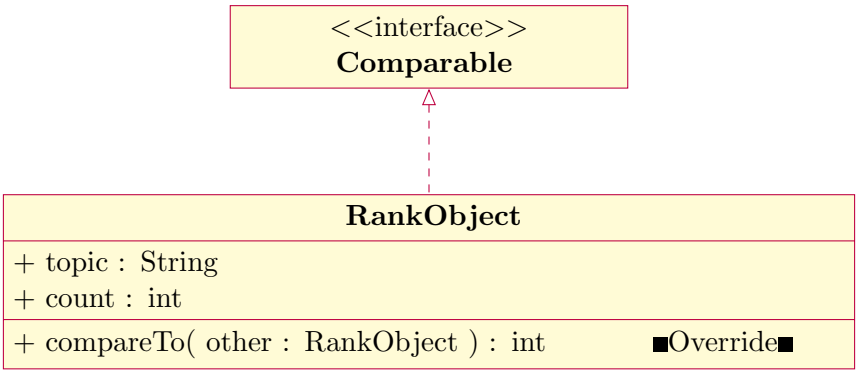


Figure 6.3: UML representation of *RankObject* class.

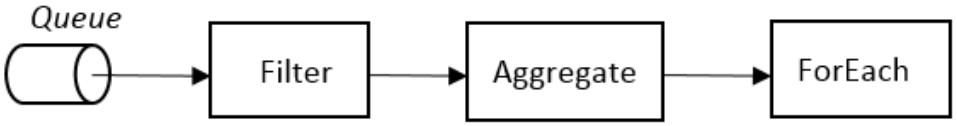


Figure 6.4: Data processing Pipeline of the Twitter Trending Topic application.

6.1.2 Implementation in JxStream

The following paragraphs provide the details toward the implementation of the trending topics solution. Figure 6.4 shows the processing pipeline we use for this implementation. We use the Apache Kafka queuing system to store the tweet objects. Kafka connectors are already supported in *JxStream*. This line of code shows how trivial it is to instantiate a supplier to read from Kafka:

```
KafkaSupplier<MyTweets> sup = new KafkaSupplier
             .Builder(topic, groupID, MyTweets.getDeserializer())
             .build();
```

The second variable needed is a *Window* object. In JxStream, we provide a tuple based and a time based window. For this example we will use the time based window. An instance of such window is shown below:

```
Window win = Window.createTimeBasedWindow(5 * JxTime.Minute,
                      5 * JxTime.Minute,
                      new HashMap<String,Integer>(),
                      (MyTweets x)-> x.timestamp);
```

The first and second arguments denote the size and the advance of the window. The equal values of these parameters indicates that we are performing a tumbling window operation. Timing metrics are already supported by the API, so the developer can simply specify the amount of seconds, minutes until weeks through the *JxTime* interface. The two remaining arguments denote the default object or value of a window and the way to extract the timestamp from each incoming tuple. In a *TupleBased* window, the last parameter is obviously omitted.

Next, we create a functional interface that dictate how to process each incoming tuple in a window. Listing 6.1 shows the bifunction that accomplishes such task. This function expects two objects and returns a result. For every tweet received, we increment its count in a map object that is kept in an intermediary state during the lifetime of the window.

```
1 BiFunction< Map<String,Integer>, MyTweets, Map<String,Integer> >
    operator =
```

```
   (map, newObj) -> {
3     if(map == null){
         map = new HashMap<>();
5     }else{
         if(map.containsKey(newObj.topic)){
7           map.put(newObj.topic, map.get(newObj.topic)+1);
         }else{
9           map.put(newObj.topic, 1);
         }
11    }
      return map;
13 };
```

Listing 6.1: Functional interface that is applied to each incoming tuple

Then we need to specify the action to apply when a window expires. Listing 6.2 shows the functional interface to do so. This function receives a map of topics to their popularity and returns an arraylist containing the topN trending topic. Each entry from the map is converted to a *RankObject* and is fed to an arraylist. As we already specified in the *RankObject* class how to sort its objects, we get an arraylist that is sorted in a decreasing order. To return the top N topics, we simply have to keep the first N elements in the arraylist and erase the rest.

```
1  Function< Map<String,Integer>, ArrayList<RankObject> > finisher = map
       ->{
   Integer i = 0;
3  ArrayList<RankObject> rankings = new ArrayList<>();
   map.entrySet().forEach(e ->{
5     rankings.add( new RankObject(e.getKey(), e.getValue()));
   });
7  Collections.sort(rankings);
   Iterator<RankObject> it = rankings.iterator();
9  while(it.hasNext()){
      RankObject o = it.next();
11    if(++i > TopN){
         it.remove();
13    }
   }
15 map.clear();
   return rankings;
17 };
```

Listing 6.2: Lambda that is applied when a window expires

Finally, we implement the processing pipeline for the computation, as shown in Listing 6.3 provides the implementation code. First we generate a *JxStream* object from the Kafka supplier previously created. Then we filter out any empty input tuple from the queue as a safety measure and use an aggregate operator with the arguments explained above. Finally the *forEach* operator prints the list of top N trending topics every 5 minutes.

```
1  JxStream<MyTweets> jstream = JxStream.generate(sup);
   jstream
3     .filter(x->x != null)
      .Aggregate( operator, finisher, win )
5     .forEach( v->{
         AggregateTuple<ArrayList<RankObject> > agg =
            (AggregateTuple<ArrayList<RankObject> >) v;
7        ArrayList<RankObject> res = agg.getAggregate();
         System.out.println("Trending topics: " + res.toString() );
9     })
      .start();
```

Listing 6.3: Main computation logic

6.1.3 Results

As we can see, implementing a trending topic algorithm with *JxStream* is straightforward and intuitive. The full implementation is available in Appendix A.1 and is relatively small compared to an implementation in Storm realized by Michael Noll in [45]. A consumer object could have been used at the *forEach* operator to receive the result and display it on a specific interface or store it. In this example, we simply print the trending topics in the console. Below, we show few examples of the top 5 trending topics every 5 minutes:

```
[Cat [49], Storm [46], Hadoop [46], Kafka [46], Summer [45]]
[Kardashian [55], Java [53], Stream [53], JxStream [51], Music [44]]
[Storm [56], Sweden [51], Summer [49], Java [48], Kardashian [45]]
[ZeroMQ [57], Kafka [56], Kardashian [53], Summer [45], Cat [45]]
[Storm [63], Apache [50], Cat [47], Summer [46], ZeroMQ [45]]
```

Note that all tweets in this example are considered as belonging to the same entity. In this application we don't distinguish tweets from different location or other attributes. JxStream provides a way to get the result based on a specific attribute by grouping the results based on a certain attribute. We present such example in the next use-case.

6.2 Sensor Monitoring

Sensor monitoring is an important function in many applications and systems. In this era of Internet of Things [50], active monitoring of devices has become even more relevant and essential. Typically, smart devices - embedded systems connected to the Internet - would send information to each other or a server with more computing resources. At the server side, the data can be logged and further processed for more complex analysis, such as fraud detection, verification, validation, etc ...

In this use-case, we present an application that receives data from a queue connected to a data source and monitors the state of a group of sensors. This use-case shows the convenience of performing multiple computations on the same stream of data.

6.2.1 Data Processing

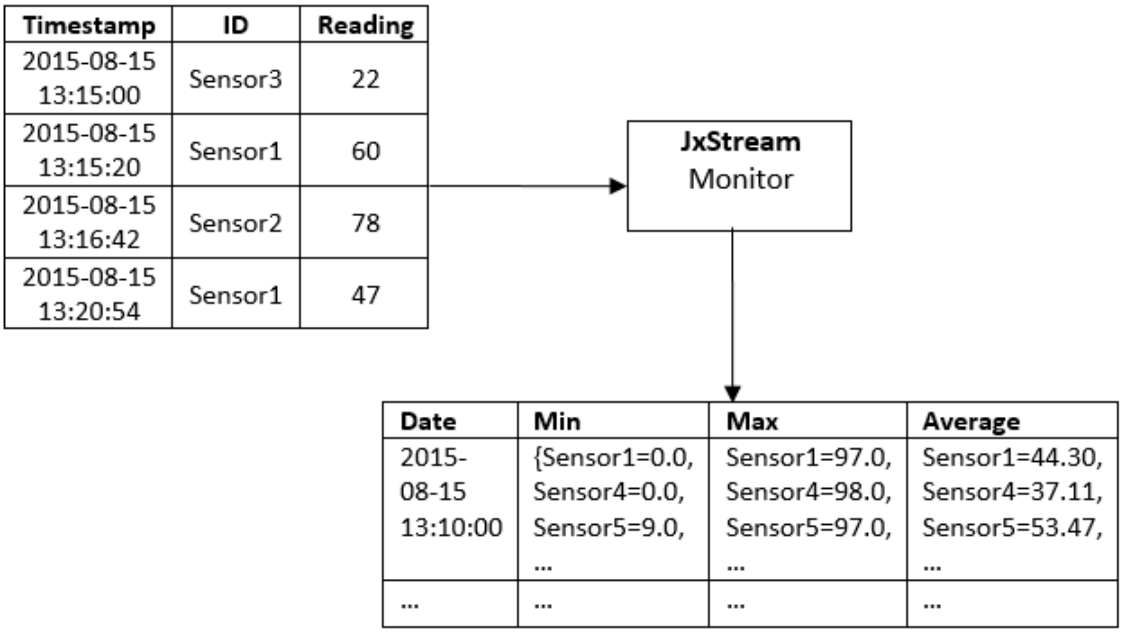


Figure 6.5: Input and Output data of the Sensor Monitoring use-case

Figure 6.5 shows the data being processed in this use-case. The tuples received by the application represent readings from a network of sensors about a specific task. The meaning of this reading is left open to interpretation, as it does not impact the goal of this scenario. For example, the sensors could be monitoring the temperature of the ambient air at their location. Table 6.1 shows the schema of the input data. For the sake of simplicity, we use only three fields to represent the data. The *ID* is a string that gives the id of the device that collected the data; the *Reading* field is the actual data read by the sensor and with the *Timestamp* of the event. The data are in Comma-Separated

Values (CSV) format and fired once every 100ms. We assume that the tuples are sorted in time in the queuing system.

Field Name	Field Type
SensorID	*String*
Reading	*int*
Timestamp	*long*

Table 6.1: Sensor data schema definition

6.2.2 Implementation in JxStream

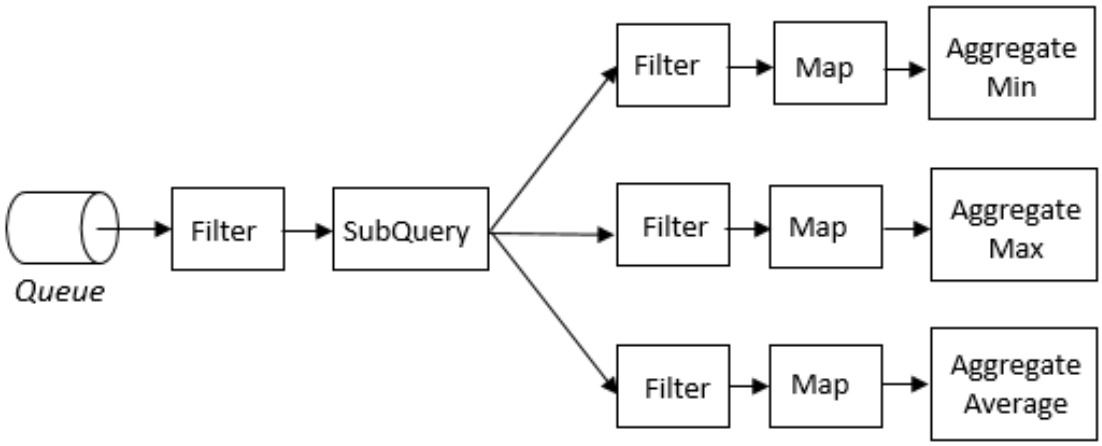


Figure 6.6: Graph Representation of the sensor monitoring application

In this subsection we proceed to explain the code needed to implement a solution to the monitoring task. Figure 6.6 shows the queries involved in this implementation. The main input pipeline simply filters out empty tuples from the queue and send data to three different branches. Each branch performs different processing on the stream, namely min, max, average.

As mandatory for all windowing operations, we need to specify the window parameters, as seen in Listing 6.4. We also define lambda expressions *fieldExtract* and *groupBy* to extract the readings from the tuple and the field to group by incoming data, respectively. In this example we group the data by their sensor id.

```
Window<String,Double> window = Window.createTimeBasedWindow(
                        JxTime.Minute,
                        JxTime.Minute,
                        0.0,
```

```
                        x-> Long.valueOf(x.split(";")[2])
6                     );
Function<String, Double> fieldExtract =
    x->Double.valueOf(x.split(";")[1]);
8 Function<String, String> groupBy= x -> x.split(";")[0];
```

Listing 6.4: Parameters needed later in processing

Listing 6.5 shows the implementation of the main stream that is directly connected to the queue. In line 1, the stream is generated from the queue supplier. On the stream object returned, we perform a simple filtering to discard null and empty tuples, as the supplier can return null if there are no data in the queue. To indicate that we want to branch out the stream, we call the method *subQuery* that will return a data source object. This object can be used as a supplier to generate as many branches as needed. Each branch will receive exactly the same set of data. The data source object encapsulates different types of queuing systems. In local mode, it is a disruptor object that buffers the result of the computation. In distributed mode, the data are dumped into either Kafka or RabbitMQ.

```
JxStream<String> stream = JxStream.generate(supplier);
2 JxDataSource<String> q = stream
    .filter(x->x!=null && !x.isEmpty())
4   .subQuery();
```

Listing 6.5: Main stream connected to the Queue

Listing 6.6 shows a stream being generated and processed from the previous main stream. As seen in line 1, this stream is generated from the data source object returned from the previous pipeline. As in the previous step, we filtered out unwanted tuples first. We append a string tag to the incoming tuple and then perform the aggregate operation. In this branch we want to compute the minimum readings in the window for each sensor, by passing the previously created lambdas in the *min* operator. The returned tuple when a window expires is a map with sensor ids as key and their minimum readings respectively. All the mapping occurs under the hood in JxStream. At the end of this branch we simply print the result of the window.

```
JxStream<String> stream1 = JxStream.generate(q);
2 stream1
    .filter(x->x!=null && !x.isEmpty())
4   .map(x-> x + ";Stream1")
    .min(fieldExtract, groupBy, window)
6   .forEach(x->{
      System.out.println("Min: "+x.getTupleCount() +" "+
          x.getAggregate());
8   });
```

Listing 6.6: The first branched-out stream

Listing 6.7 shows the second branch that is derived from the main stream. As the previous branch, it is generated from the data source object returned at the main stream. In this branch we compute the maximum of the readings in the window, grouped by the sensor id.

```
JxStream<String> stream2 = JxStream.generate(q);
stream2
   .filter(x->x!=null && !x.isEmpty())
   .map(x-> x + ";Stream2")
   .max(fieldExtract, groupBy, window)
   .forEach(x->{
      System.out.println("Max: "+x.getTupleCount() +" "+
          x.getAggregate());
   });
```

Listing 6.7: The second branched-out stream

Listing 6.8 shows the third branch derived from the main stream. In this branch, we compute the average of the readings, grouped by sensor id. The *forEach* method returns a *PipelineEngine* object to control when the processing needs to start. We could have get the engine object in the previous branch, but it is important that the *start* method is called at the end of all the branches.

```
JxStream<String> stream3 = JxStream.generate(q);
PipelineEngine engine = stream3
   .filter(x->x!=null && !x.isEmpty())
   .map(x-> x + ";Stream3")
   .average(fieldExtract, groupBy, window)
   .forEach(x->{
      System.out.println("Average: "+x.getTupleCount() +" "+
          x.getAggregate());
   });
engine.start();
```

Listing 6.8: The third branched-out stream

6.2.3 Results

We show the output of our computation in the example below:

```
Min: 1440113369780 {Sensor1=0.0, Sensor4=0.0, Sensor5=9.0, Sensor2=2.0,
```

```
                    Sensor3=3.0, Sensor6=2.0}
Max: 1440113369780 {Sensor1=97.0, Sensor4=98.0, Sensor5=97.0, Sensor2=92.0,
                    Sensor3=99.0, Sensor6=97.0}
Average: 1440113369780 {Sensor1=44.30, Sensor4=37.11, Sensor5=53.47,
                    Sensor2=44.01, Sensor3=47.70, Sensor6=49.40}
-----------------
Min: 1440113429780 {Sensor1=0.0, Sensor4=0.0, Sensor5=9.0, Sensor2=2.0,
                    Sensor3=3.0, Sensor6=2.0}
Average: 1440113429780 {Sensor1=49.28, Sensor4=39.71, Sensor5=52.46,
                    Sensor2=40.08, Sensor3=51.04, Sensor6=45.02}
Max: 1440113429780 {Sensor1=97.0, Sensor4=98.0, Sensor5=97.0,
                    Sensor2=94.0, Sensor3=99.0, Sensor6=97.0}
```

A line starts with the name of the computation, followed by the timestamp at the beginning of the window and the list of results per sensor. Note that the timestamps are different exactly by 60000ms which is one minute as defined in our window. The order in which the *min, max, average* results are printed are not constant because each branch is running in its own thread. Thus we can see that *JxStream* helped us perform parallel computations on the readings of the sensors. This capability is highly desirable in stream computation. A full implementation is available in Appendix A.2.

7

Evaluation

In this chapter we present the performance of JxStream against different test cases and scenarios. Part of the goal of this thesis is to build a high performing system with minimum overheads. Therefore the main focus of the performance tests is to establish the throughput and latency of the system given different situations. This help us to validate the assumptions and decisions made during the design and implementation of the system. In section 7.1 we look at how the performance of JxStream in a single process (JVM) compares with Apache Storm [6] a well known open source stream processing engine running under the same conditions (Single node). We then look at the scalability of JxStream in section 7.2. Here we investigate how JxStream running on a single machine performs as the processing pipelines' parallelism is increased thereby taking advantage of the available cores. Doing so gives us an idea about the predictability of the system's performance in different execution environments and configurations. Finally in section 7.3 we compare the performance of the two underlying implementations (Window mechanisms) for aggregate operators.

To run these tests, we used a laptop with an Intel core-i7 CPU (4 physical/8 logical cores), 16 gigabytes of RAM and a hard drive speed of 5400 RPM running Ubuntu 14.04 on a 256 gigabyte partition.

7.1 Local Processing

Every distributed system is composed of independently executing processes working in a coordinated manner. Two factors will determine the performance of such a system: the processing capability of the individual processes and the infrastructure that is used for inter process/node communication. In this section we focused on investigating how a JxStream process fairs in terms of throughput and latency with other stream processing

engines in industry. For this task we chose Apache Storm as a system to compare against as it is a well known and received open-source system for real-time stream processing. The main objective of comparing against a fully fledged industry accepted system is to bring the performance of JxStream into perspective with existing solutions.

To achieve maximum throughput for each test case, we found it necessary to adjust configurations for both *JxStream* and Apache Storm. We took an iterative approach of trying out different configurations (parallelism) and picking the one yielding the highest results for each system. Thus each system was tuned for performance independently. For *JxStream* there were not many different settings to vary apart from the parallelism of the Streams to gain the highest performance. On the other hand we had to experiment with a lot of different configurations to reach the best setup for Apache Storm with the same topology. Overall this was expected from the onset owing to the fact that Apache Storm is a fully fledged sophisticated distributed system.

7.1.1 Apache Storm Setup

For these experiments we used Apache Storm v0.9.4. At a high level Apache Storm's programming model is composed of *Topology*, *Spout* and *Bolt*. An Apache Storm *topology* is a programmer-defined graph representing the stream pipeline of the system. Basically it is a network of *Spouts* and *Bolts*. A Spout acts as a data source of the Apache Storm topology. Using this a programmer is free to read data from any source. On the other hand a *Bolt* acts as an operator in that it consumes tuples from *Spouts* or other Bolts and optionally emits new tuples. Using *Bolts* a programmer can define complex logic to be applied onto tuples.

Tuning Apache Storm so as to achieve a maximum performance in a single node environment was an important step in the preparation of the tests. Below are the settings we went for and the rationale behind them.

- To maintain reliability Apache Storm provides *at least once semantics* by employing tuple tracking and replaying. This feature comes with its own overheads that results from tuple acknowledgment and message replay mechanisms. In these tests we had to turn off this feature as it would deter the performance of Apache Storm. Moreover JxStream currently has no support for such semantics hence we found it necessary to level the playing field in this regard.

- By default each Apache Storm worker process is configured to use up to 768 megabytes of heap size, exceeding this threshold will cause a run time error in the JVM. To this effect the JVM's garbage collector will try its best to reclaim memory in order to avoid a crash. If an application is generating a lot of objects, the frequency of garbage collection will rise which will result in stalls in the JVM's execution. In this test we had to make sure that garbage collection pauses do not

affect the results thus we increased this threshold to 3 gigabytes which was at par with the run time configuration of JxStream.

7.1.2 Data

To gain insight about the maximum throughput of JxStream, we used a test scenario that is not computationally expensive. Trying as much as possible to stay close to a real world use-case we chose Foreign Exchange (Forex) trade data as a basis for these experiments. This data is usually of a simple defined structure of 7 fields or less representing a currency pair information arising from a bid or sell. The granularity of this data ranges from one to a few seconds per currency pair. Traders use this data together with real-time software solutions to calculate many different indicators that are important in financial decision making. We downloaded a publicly available dataset[51] comprising of the currency pair Euro to Swedish Krona(Eur-SEK) for the year 2014 as a text file. In order to avoid the data source from being a possible bottleneck in this experiment, we opted for an execution where we simulate the messaging queue with an in-memory data-generator that would produce data using a sample from the original dataset. Table 7.1 shows the schema for the currency data.

Field Name	Field Type
lTid	*long*
cDealable	*char*
CurrencyPair	*long*
RateDateTime	*datetime*
RateBid	*double*
RateAsk	*double*

Table 7.1: Currency data schema definition

7.1.3 Experiments

Currency Simple Moving Average

Forex traders use many different indicators in making decisions about what they buy, hold or sell. One important indicator used is the Simple Moving Average (SMA) [52]. This is basically a time based sliding window operation averaging the *RateBid* and *RateAsk* for a particular currency pair. We designed a 4-operator pipeline to perform this computation. Figure 7.1 depicts this data flow pipeline.

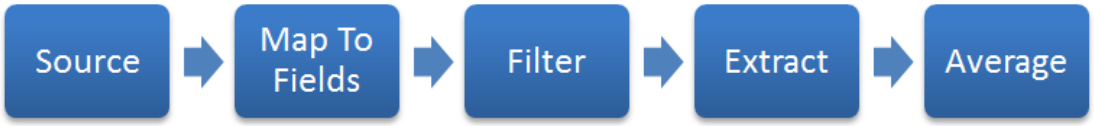


Figure 7.1: Pipeline for calculating the Simple Moving Average for Currency data.

Implementing this pipeline in JxStream didn't require much effort owing to the expressiveness and readily supported time based window aggregates.We give the full implementation in Appendix A.3.

To bring this to Apache Storm a little more effort had to be put in. Apache Storm does not support sliding window operations out of the box. The closest candidate we found that provided something close was Trident [53]. This is a high level abstraction built on top of Apache Storm and allows stateful stream processing with low latency distributed querying. In addition it provides *exactly-once semantics* and support stream grouping,aggregation, joins e.t.c. However it still does not support sliding window operations and furthermore we found it difficult to extend it to do so. For this reason we instead built on top of the raw Apache Storm's Spout/Bolt to achieve this. We had to make sure that the implementation of the Average function was computationally equivalent to that of JxStream and thus we adopted the same logic used in JxStream Aggregate Implementation. Table 7.2 below shows a summary of the pipeline implementation for both systems.

Operator	Description	JxStream	Storm Component
Source	The data source of the pipeline.Uses an in memory queue simulator	SupplierConnector	Spout
Map To Fields	Converts to the string tuple to the defined schema	map	Bolt
Filter	Filters out any tuples with missing values	filter	Bolt
Extract	Extracts out the needed values for computation	map	Bolt
Average	Performs the window operation	average	Bolt

Table 7.2: SMA implementation

In Figure 7.2 and 7.3 we present the throughput and latency of both systems. The

throughput of JxStream was 5 times that of Apache Storm in this experiment. Also the processing latency for Apache Storm was much higher than JxStream. With these results it was evident that JxStream benefits from the Java Stream's optimized way of dealing with operators. In Apache Storm each stage of the pipeline run in a separate thread with inter stage communication necessitated by the queues. This means that a pipeline having 4 operators will at the very least spawn 4 threads to host the operators. This is even before we factor in operator parallelism. On the other hand in JxStream the whole processing pipeline is run in one thread. This means that if there are four operators in a pipeline, they will all be executed sequentially thus eliminating any intermediary queues. Also if the stream is parallelized into say five streams, only five threads will be created to host each stream. It should however be noted that Apache Storm is a fully fledged SPE and because of this it comes with a lot of other capabilities and benefits. Thus JxStream can only be used in a subset of scenarios where Apache Storm and other SPEs can be applied.

Pipeline Cost

Given the results from the previous experiment we investigate further how much the fundamental difference in architecture between Apache Storm and JxStream impacts the performance gap. We need to establish the relative overhead cost that comes with every addition of an operator. To do this, we test both systems with varying number of operators. In these experiments it is important that the computation of the operators remain simple so as not to have the logic itself as the bottleneck. For this reason we create a simple operator (*Dummy Mapper*) that upon receiving a tuple only added the *Ratebid* and *Rateask* values after which sending the original tuple down stream. We use the *Dummy Mapper* to vary the length of the pipeline as shown below in figure 7.4.

Figure 7.5 presents the throughput results from the experiment. As can be seen from the graph, Apache Storm's throughput degrades sharply with each addition of an operator while for JxStream the performance does not vary much. The results from Apache Storm show that the system's infrastructure itself has a relatively high overhead which limits and affects the maximum throughput the system can attain. Each operator added has a sizable overhead attached to it. On the other hand JxStream does not suffer from this since Java 8 Stream performs optimization by combining all the operators into one execution unit per thread. It should also be noted that since each operator runs in a different thread in Apache Storm, the number of threads can quickly grow as operators are added and parallelized. This results in issues related to CPU contention and context switching.

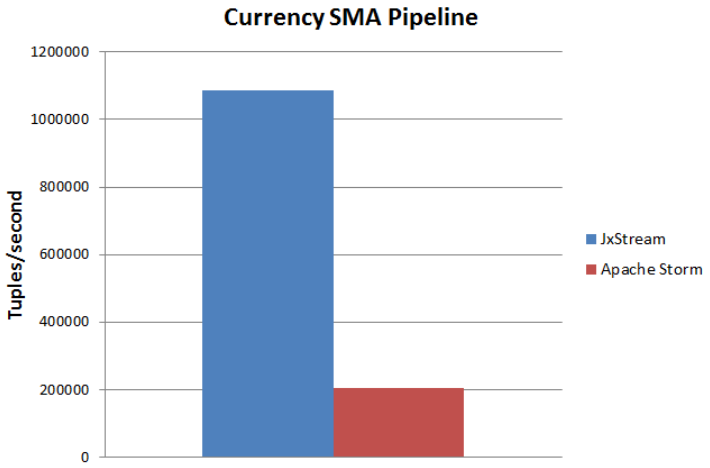


Figure 7.2: Throughput for Currency SMA

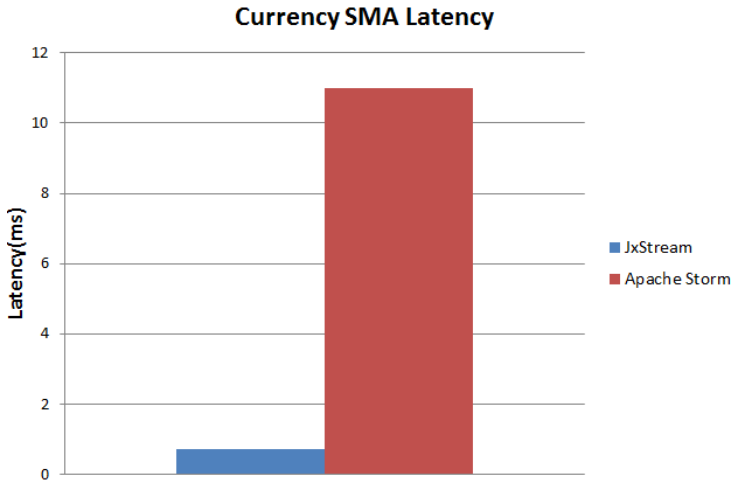


Figure 7.3: SMA processing latency

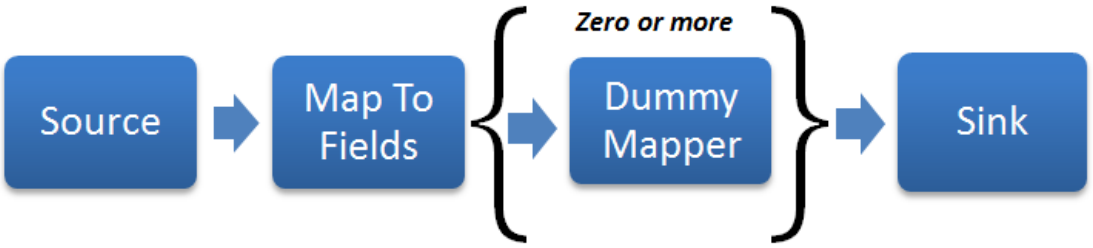


Figure 7.4: General construction of the pipelines using a *Dummy Mapper*

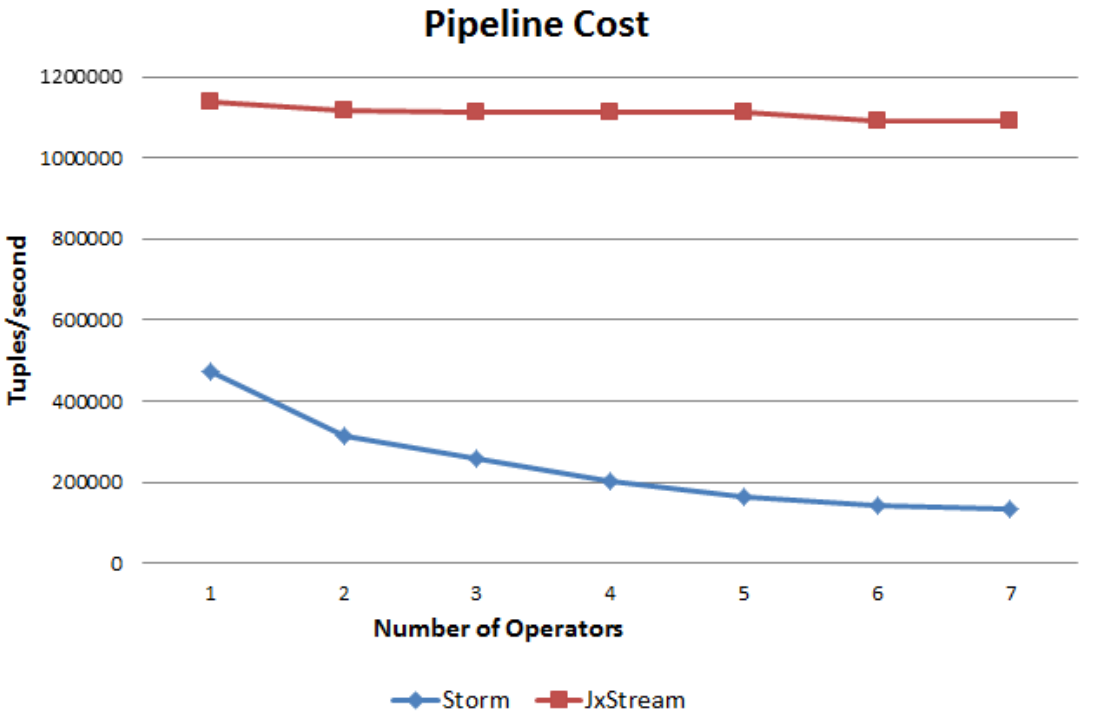


Figure 7.5: Pipeline construction

7.2 Scalability

The ability of a software solution to scale with an increase in the number of processing cores is an important attribute in modern computer applications. This is due to the trend in modern computer hardware where higher processing power is being achieved through parallelization of cores rather than increase in CPU clock speed. What this entails is application software should be designed to take advantage of the concurrent execution environment offered by the hardware. In this section we focus on investigating the performance behavior of JxStream as the number of processing cores is increased. Our aim in this experiment is to deduce the pattern and establish whether JxStream could effectively utilize the hardware available to it. Having this information allows us to predict the system's performance given different execution environments.

Software applications can be classified as either being CPU bound or Input/Output (IO) bound depending on whether the bottleneck is the CPU or IO. In order to scale IO bound applications, not only the number of cores need to be increased but also the IO itself which could be disk or network. In the case of the disk being the bottleneck, the speed of the disk or the number of disks need to be increased in order to improve the performance. Given the environment of our evaluation, we based our performance test on a CPU bound test scenario since we were using only one machine.

To construct a fairly easy pipeline but yet highly CPU intensive we chose cryptographic operations as a basis for our experiment. Depending on the primitives used cryptographic functions can be highly CPU intensive. With that in mind we settled to using a 128 bit key-sized Advanced Encryption Standard (AES) [54], a popular symmetric block cipher used in many software solutions. Though in typical use-cases encryption/decryption would not be used in data streaming, there are still a few use-cases where encrypting/decrypting text could be part of a processing pipeline. Our choice of using it here was purely motivated by the CPU intensity of these operations.

Figure 7.6 shows the pipeline construction used in this experiment. The data source was an in-memory data generator. This produced bytes of cipher texts drawn from a pool of pre-encrypted text. Since the objective of this experiment was to achieve a highly CPU intensive pipeline, we had to make sure that the data source was not a bottleneck of the pipeline hence the need for pre-encrypting the text. Following the source was a series of JxStream *map* operations each performing configured rounds of decryption on the bytes as illustrated in the figure. In this pipeline a tuple composed of one field holding a *CryptoObject* which is basically a plain Java object containing the cipher bytes and the 128 bit encryption key used. Each *map* operation would take the bytes and perform a configured number of decryption rounds using the key. The *map* operation will then produce the original *CryptoObject* as output to the downstream. In doing so we avoided the need of running encryption and creating new CryptoObject for the next operator. As shown in the figure the first *map* performs one decryption round followed by the second and third *map* operators which perform two and one rounds respectively. Using this pipeline we run a series of experiments adjusting the number of parallel streams to use in each and calculating the average throughput.

Figure 7.6: CPU Intensive Decryption Pipeline

Figure 7.7 presents the results of these experiments in form of a graph. We noticed a linear performance increase in the throughput as the parallelization factor was increased from one to four streams. With five parallel streams the throughput increment factor decreased a bit and remained constant up until seven streams. Beyond seven streams we saw a deteriorating performance as the number of parallel streams were equal to or more than the number of logical cores of the test machine. This behavior was expected since CPU resource was now scarce resulting in thread contention issues in the system.

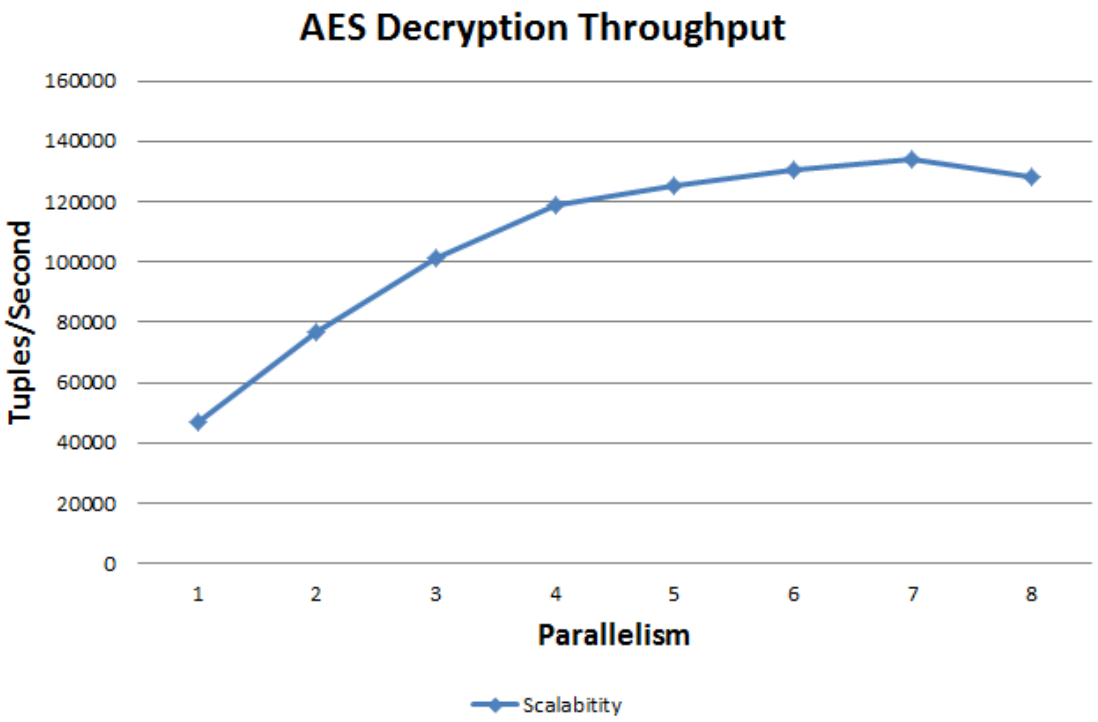


Figure 7.7: Throughput for AES 128 decryption

7.3 Aggregate Implementation Comparison

As we discussed in chapter 5 aggregate operators in JxStream are based on two underlying implementations. These being either Ring-based or Array-based implementations. In terms of usability for a programmer trying to build upon these aggregates, the Array-based implementation has a simpler programming interface. This is because the Ring-based aggregate requires the programmer to specify a *Combiner* function in addition to the *per tuple* and *finisher* operators. The *Combiner* function is used in combining all the slots of the ring into one upon window expiry. Nevertheless the Ring-based approach has a performance advantage in most scenarios. The objective of the experiment in this section is to compare the performance of the two implementations and thus validate the assumptions we made during the implementation of the aggregate infrastructure.

For this experiment we chose to use the *Twitter Trending Topics* use-case we presented in chapter 6 as the basis for our tests. We refer the reader to that chapter for more details about this scenario. The source data for this experiment was simulated by generating tweet texts on demand using predefined texts. Doing this avoided any bottlenecks that could have resulted from IO if we were reading the text from file or message queue. We downsized the schema of the tweet text to contain only the basic information needed in the computation and a few dummy fields. We found it necessary

to use a trimmed down version of a tweet to avoid the unnecessary garbage collection pauses that would results from the rapid creation and disposing of large string objects. It should be noted that using this version would not affect the overall meaning of the results since the comparison was between two implementations in the same system. Aggregate operations can either be *time based* or *tuple based*. In this experiment we used the more popular time based window aggregate. Nevertheless the performance results here still apply to the *tuple based*.

Table 7.3 gives a summary of the setup used for the aggregate operator in the tests. In all the tests for both of the implementations we fixed the window size to one hour (3600 seconds). It is important to realize that the absolute values of the parameters would not affect the comparison but rather the ratio of *Window Size:Window Advance* is what is cardinal. This is because this ratio represents the number of overlapping windows and these are handled differently in the two implementations. For this reason, in this experiment we kept the Window Size fixed and varied the Window Advance in order to get different ratios. In the Tweet generator object we used a fixed stride of 2 seconds to separate consecutive tweets.

Window Size	3600 seconds	This was fixed for all tests
Window Advance	Variable(4sec to 1200 seconds)	This was what changed between experiments
Time Stride	2 seconds	This was fixed for all tests

Table 7.3: Setup summary of the Trending Topics performance test

Results

In figure 7.8 we show the results from all the tests we run in this experiment. Figure 7.9 is a zoomed-in view of the results from tests with window advance equal to or less than one minute. As can be seen from the graph the overall performance of the ring based approach is greater than the array based. However an interesting trend can be seen on the results with the ratio of 60 and above. As discussed in the implementation chapter section 5.1 the ring based approach creates slots based on the formula; $number\ of\ slots = W_s/W_a$. Having a high W_s/W_a ratio results in a huge number of slots on the ring. Depending on the computation in question this may end up making the combiner function very expensive. From this observation we can deduce that the ring based approach is superior in general but not in cases where the combiner function is expensive.

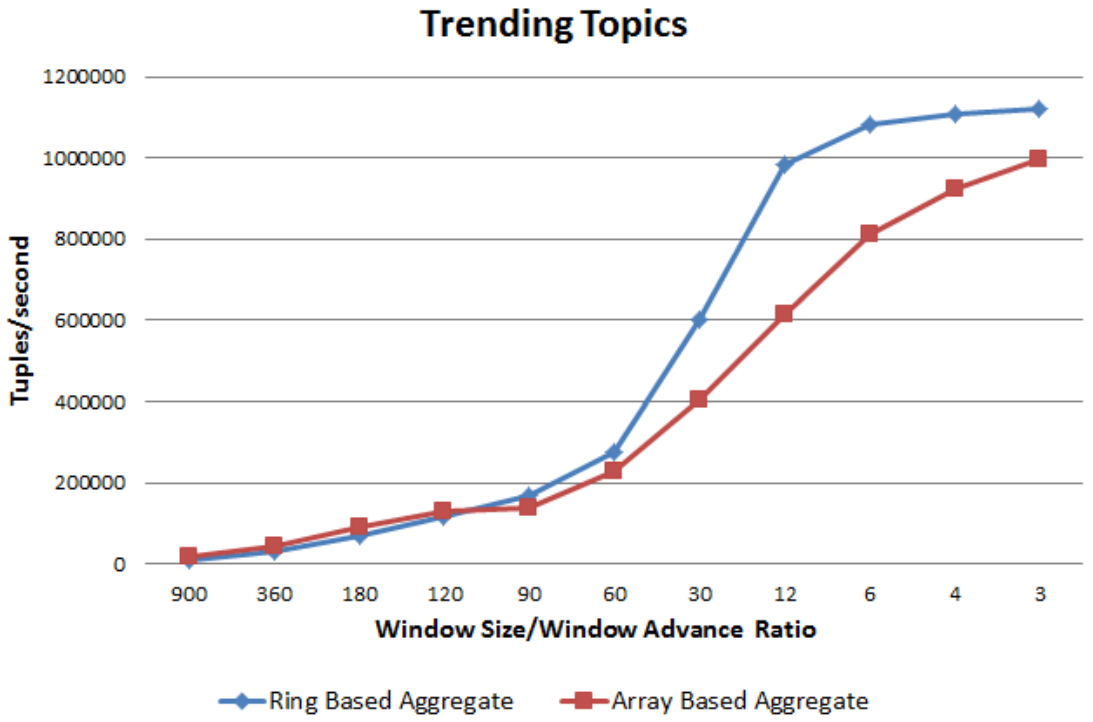


Figure 7.8: Throughput for Trending Topics

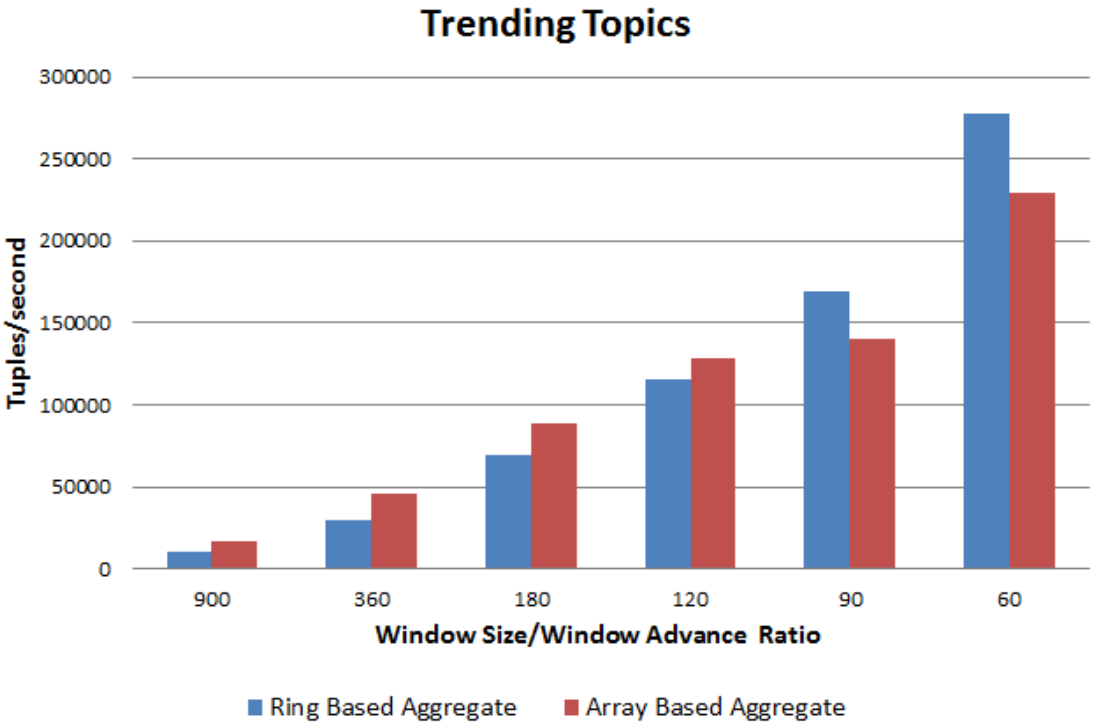


Figure 7.9: A zoomed view of the throughput for advance less than 60 seconds

8

Related Work

Data Stream Processing has attracted a lot of attention in academia and industry alike. This is due to the increase of data being generated by different applications and devices. This data often need to be processed in a real-time fashion. Thus it has influenced the technology community's drive towards finding better ways of handling the challenges behind this type of analysis.

In this chapter we discuss past research and systems whose area of work is related to data stream processing and this thesis. We break the discussion into three sections. In Section 8.1 we look at work that has focused on creating programming abstractions in order to make stream processing easier. The area of research relates to this thesis as we are adapting the Java programming model to the Data Streaming Paradigm. In Section 8.2 we then discuss work whose focus has been to create software solutions that handle stream processing on a single node. These researches put in evidence how processing data in a centralized fashion is also desirable and useful. Lastly in Section 8.3 we end the chapter with a discussion on popular distributed stream processing engines that are in use today. These researches relate to JxStream in their objective of adapting a batch system to a distributed streaming tool.

8.1 Stream Programming Abstraction

In this section we present research motivated by the need to ease the processing of streams of data with more powerful abstractions. A number of papers [55, 56] proposed new languages or extended existing ones solely to support data streaming paradigm. In this section we will discuss the StreamIt [57] and StreamFlex [58] languages.

8.1.1 StreamIt

In 2002, at the MIT Laboratory for computer science, Thies et al. introduced *StreamIt* [57] as a new language for streaming applications. Thies et al. strongly argue that general-purpose languages are not in sync with the stream programming construct. These languages leave out a lot of assumptions and patterns that could be used to provide more flexibility and functionality to the developers. Compelling looping optimization are not implemented at the compiler to improve the performance of operations on streams. Moreover, the common machine language abstractions for Von-Neumann architectures no longer hold for the emerging grid-based architectures that is driving the Big Data trend. All these factors pushed for the specification of the StreamIt language.

StreamIt is a language and compiler designed to provide better stream programming abstractions to developers, to support more efficiently grid-based processors and ultimately to achieve expert-level performance optimization under the hood. Filters are considered the basic unit of computation in the language. They typically receive an input from a FIFO channel, process the data and push the result to an output channel. The filters communicate with each other by following three connecting mechanism: a *Pipeline* follows a sequential flow; a *SplitJoin* splits the flow at a filter into multiple streams and later joins them into one stream; a *FeedbackLoop* follows the same construct as the previous mechanism except that the streams are joined at an anterior filter, thus creating a loop. Another interesting feature in StreamIt is the ability for filters and streams to send and receive unicast or broadcast control messages. This messaging system can be used to change anything that is related to the data flow in the network.

The *Pipeline* concept of StreamIt is very similar to the mechanism we have chosen in this thesis. Although the name "filters" in StreamIt may be misleading in their intended function, they are very much the equivalent of the thesis' operators. We provide a split functionality to the stream while the join feature is left for future work. While many of StreamIt's features are attractive for stream processing, we find that the introduction of a special purpose language limits its potential and incurs substantial work to integrate to other systems. JxStream is built on the Java language that hosts a rich library of systems, making it easier to work with other systems.

8.1.2 StreamFlex

Inspired by languages such as StreamIt [57], Gryphon [59, 60], Banavar et al. introduce *StreamFlex* [58]. *StreamFlex* is an abstraction for stream processing designed in Java. Built on top of the real-time Ovm virtual machine [61], it provides a programming model that allows developers to write stream applications in a Java syntax. Like StreamIt, *filters* are the computing components that communicate with non-blocking and bounded input/output channels. Messages mutate and flow between filters without the need to be copied. By connecting filters together, a developer can build a *graph* that suits the problem to be solved. With its extended version of the *javac* compiler, StreamFlex

achieved interesting results in terms of scheduling and memory management. Results show that StreamFlex performs about 4 times better than implementation on other Java variants.

This thesis is built on similar motivations as StreamFlex. Java was selected as the host language to support the system in both cases. Using a language that is familiar and widely used makes the new product more intuitive. The StreamFlex team made this decision after the implementation of the Real-Time Specification for Java (RTSJ) [62]. As the Java language got more mature, the Stream package was the major enabler of this thesis.

Nevertheless, a number of features sets our work apart. The programming models are inherently different. Our model follows the Java streaming operator concept, with no input/output channels between operators in a stream pipe (refer to Section 5.2). This thesis' solution is solely implemented on top of Java. Although StreamFlex achieve good results with a modified version of the compiler, we preclude ourselves from this task. This decision removes the necessity of maintaining a separate tool and provide a simpler API for the scope of this thesis. We rely on the optimization techniques provided by the language to handle the processing.

8.2 Centralized Stream Processing

In this section we discuss stream processing solutions designed to run on a single machine. Centralized stream processing frameworks are often used as building blocks in distributed stream processing systems. In fact JxStream falls under this category of systems though it comes with support that makes it easily distributable.

8.2.1 Esper

Esper [46] is a centralized Complex Event Processing (CEP) engine built for real-time applications in an event driven architecture. The system provides an Event Processing Language (EPL) similar to SQL that allows expressive queries over the data spanning time windows. Users of the engine write event conditions or patterns that are evaluated possibly over multiple streams. The system triggers the appropriate actions when such conditions are met by sending an event to the listener. Esper provides a client interface for Java and one on the .NET framework under the name NEsper.

The fundamental difference that separates this system from JxStream is that it is a CEP engine. Thus Esper is built towards inferring events and patterns in streams of data. On the other hand JxStream is a general purpose tool with which a range of streaming applications can be built. As a result JxStream can be used to build applications with CEP features and functionality. Additionally the two systems use different programming models. Esper's EPL is tailored towards writing queries similar to SQL while JxStream adopts the Java 8 Stream model. Also Esper can be integrated in other SPEs while

JxStream is not intended nor designed for such integrations. JxStream is to be used as an alternative to an SPE in certain scenarios.

8.2.2 Siddhi

In their publication [63] Suhothayah et al. presented Siddhi a lightweight complex event processing (CEP) engine on which the massively scalable WSO2 Complex Event Processor [64] is built. Their motivation for Siddhi was to rethink the design of complex event processing systems by adopting a high level architecture that resembles stream processing systems in order to achieve high performance.

The core architecture of Siddhi consists of processors connected through input and output event queues. A Processor is composed of executors that expresses the queries. Each executor processes the incoming input events and produces a boolean value to indicate if an event matches the query conditions it represents. The matching events are processed by other executors downstream while the non matching ones are discarded. Using this approach insures that events that fail certain conditions are eliminated early in the pipeline thus enhancing performance. The use of processors follows the pipelining model where execution is broken down into different stages and the data moves through the stages using the publish-subscribe pattern. To get faster execution and higher throughput, every Stage is assigned to a different thread thus having them run in parallel.

In terms of the programming abstraction, Siddhi query object model resembles an SQL like structure and hence the queries falls in line with the relational algebraic expressions. The researchers chose this approach so that users, especially those with an SQL background, can easily understand and write queries. Query creation is done using the Java API provided where users can use methods mimicking SQL statements like SELECT, FROM, WHERE etc. It also supports complex nested query definition where one query is composed of other defined sub queries. This enhances the expressiveness of the interface but still maintaining its ease of use. The interface also supports both time based and tuple based window queries thus a user need only specify a few arguments to define window operations.

Though Siddhi's general approach of pipelining is similar to JxStream's the two systems differ in many respects. Siddhi's core design breaks down the execution into different stages with each in a separate thread while in JxStream a chain of computations are all run within one thread. Also like Esper, Siddhi uses an SQL like programming model. In addition the backbone of JxStream is built using the Java 8 Stream, this means that it can benefit from compiler optimizations that can be implemented as part of the language.

8.3 Distributed Stream Processing

The volume and rate of data produced in modern typical data streaming scenarios is often huge. This has lead to solutions that distribute the processing of this data over to multiple machines. This section looks at solutions that have come out of academic research. Apache Storm a system we used as part of the evaluation also falls in this category. We however do not discuss it here as it was adequately covered.

8.3.1 Spark Streaming

Spark Streaming [7] is an extension of Apache Spark (Spark) [65] a batch processing engine. It enables scalable, high-throughput, fault-tolerant stream processing to be performed on top of the batch system. Spark Streaming originated from the research done by Zaharia et al. [66]. As data flows into the Spark Streaming system it is broken into small fixed sized batches which are then processed by the Spark engine. Spark Streaming is based on the programming abstraction called discretized stream (*DStream*) which represent a continuous stream of data. This abstraction provides two classes of operators; Transformation operators and Output operators. Transformation operators always produce DStream from one or more parent streams. These operators can either be stateful or stateless. On the other hand Output operators facilitate writing to external systems.

At a high level the goal of Spark Streaming is similar with the goal of JxStream in that it brings data streaming capabilities to a batch system. However the respective targeted systems differ as well as the approach taken. Spark Streaming is build on top of Apache Spark a fully fledged batch processing system while JxStream is built on top of the Java 8 Stream. Additionally the programming model in Spark Streaming is based on *DStream* while our model is based on the Java language itself.

8.3.2 DistributableStream with Java 8

Su et al. [67] of Oracle presented DistributableStream, a Java 8 based API that enables programmers to write distributed and federated queries on top of pluggable compute engines namely, Hadoop Map Reduce and Coherence initially but extensible to other engines.

The framework addresses the challenges faced in Java 8 Streams when trying to process huge datasets that cannot fit in a single machine. These datasets are thus stored in a distributed fashion and require an efficient way of processing. The researchers' solution provide a Java 8 Stream like interface providing similar operations (i.e Map,filter,collect) that can be applied on distributed datasets. At the lower level these operations are translated, serialized and shipped to the respective distributed third party compute engines for processing. The focus of this solution is to provide a computational model and API

to be used across different external compute engines. This means that a programmer can write an application to run on one compute engine and able to easily run it on another without making any changes. DistributableStream also includes a local compute engine built on top of a Java thread pool that is used in cases when the dataset is not distributed but reside on the same machine as the JVM.

This solution does not cover the focus area of our thesis which is building a framework on top of Java 8 that can be used in analyzing unbounded streams. Although it is similar in that it includes a local compute engine built atop Java, this solution's focus is towards bounded data and hence the processing is batch oriented.

9

Conclusion

We present JxStream, a tool that can be used in building scalable data stream applications targeted for Big data scenarios. In these scenarios continuous streams of data need to be processed in a real-time fashion and thus having tools to ease the writing of such applications is important. JxStream provides a flexible and expressive programming abstraction adopted from the Java language. This abstraction is aimed at simplifying how a sequence of data is viewed and processed. We employ this model to data streaming by building on top of Java 8 Streams a framework that supports typical data streaming semantics. We do this while maintaining the same programming model and expressiveness of the Java 8 Streams.

The evaluation results show that JxStream can indeed provide high throughput and low latency applications hence can be an alternative to SPEs in certain scenarios. The high performance of JxStream can be alluded to the fact that it directly benefits from the optimized design of the Java 8 Stream abstraction. In this design a chain of operators are combined to execute in one thread thus avoiding the need for inter thread messaging. This gives JxStream an advantage over systems whose design break up a chain of operators into separate execution threads per operator. Furthermore because the in-process stream parallelization of JxStream has a low overhead cost, JxStream shows a good and predictable scalability curve.

From the different data stream use-case tests performed JxStream demonstrated how easy and flexible its API was in solving typical data stream problems. The adopted Java 8 stream abstraction model which uses many of the functional programming concepts like lambdas,lazy evaluation etc. makes the API highly expressive. Similarly the inclusion of aggregate operators which supports typical data stream semantics like windowing and group by functionality makes JxStream a powerful tool. In addition any programmer familiar with Java 8 would use JxStream without the need of learning a new programming

model.

9.1 Future Work

JxStream in its current state is a first step towards a fully fledged stream processing system. This thesis has shown that the Java 8 Stream package can be used to create high performing data streaming systems. The tool poses as a building block in highly scalable systems. For solutions that require a single or few nodes, the tool can be used as it is. However to compete with current stream processing engines which can scale to huge cluster of nodes, a lot more has to be done to turn JxStream into a fully fledged stream processing engine of this sort. Below are recommendations of issues that need to be worked on.

9.1.1 Inbuilt Inter Process Messaging

Communication between JxStream application pipelines running in different processes is currently achieved by using external messaging queues. JxStream's comes with adapters to these systems thus easing their use. However using these systems as default transports has a performance hit on JxStream. These external systems have been built in a generic fashion so as to cater for a wide range of use-cases. Owing to the fact that these are fully fledged systems on their own, they come with a lot of overheads which result in them under performing. JxStream should be extended to have its own inter process messaging. This messaging transport can be built on top of existing brokerless messaging solutions like ZeroMQ [68], Aeron [69], Netty [70] etc. An in-built messaging solution would increase the throughput and reduce the latency of the communication between processes/nodes thereby resulting in an overall improvement of performance when running on multiple nodes.

9.1.2 Distributed Infrastructure

The current version of JxStream consist of just a library that can be used in building stream applications. In cases where the application needs to be deployed on multiple nodes, the programmer needs to achieve this manually. Though this can be manageable with a few nodes it becomes more complex with an increase in the number of nodes as the challenges related to distributed systems need to be addressed. For JxStream to be useful in current big data establishments, a distributed middleware need to be implemented as part of JxStream. The middleware should handle issues related to distributed systems like node coordination, failure recovery,task allocation etc. Having this in place would free application programmers from having to solve these issues but rather concentrate on the application itself.

Bibliography

[1] M. Stonebraker, U. Çetintemel, S. Zdonik, The 8 Requirements of Real-time Stream Processing, SIGMOD Rec. 34 (4) (2005) 42–47.
URL http://doi.acm.org/10.1145/1107499.1107504

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik, Aurora: A New Model and Architecture for Data Stream Management, The VLDB Journal 12 (2) (2003) 120–139.
URL http://dx.doi.org/10.1007/s00778-003-0095-z

[3] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, P. Valduriez, StreamCloud: An Elastic and Scalable Data Streaming System, Parallel and Distributed Systems, IEEE Transactions on 23 (12) (2012) 2351–2365.

[4] V. Gulisano, StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine, Ph.D. thesis, Universidad Politécnica de Madrid (2012).

[5] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. Zdonik, The Design of the Borealis Stream Processing Engine., in: CIDR, Vol. 5, 2005, pp. 277–289.

[6] Apache Storm.
URL https://storm.apache.org/

[7] Apache Spark Community, Spark Streaming.
URL http://spark.apache.org/streaming/

[8] Apache Samza Community, Apache Samza.
URL http://samza.apache.org/

[9] IBM, InfoShere Streams.
URL http://www-03.ibm.com/software/products/en/infosphere-streams

[10] Oracle, The Java 8 Stream Interface.
URL https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

[11] Oracle, Java 8 Lambda Quick Start.
URL http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html

[12] Oracle, Package java.util.function.
URL https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

[13] D. Lea, A Java Fork/Join Framework, in: Proceedings of the ACM 2000 Conference on Java Grande, JAVA '00, ACM, New York, NY, USA, 2000, pp. 36–43.
URL http://doi.acm.org/10.1145/337449.337465

[14] V. Gulisano, Y. Nikolakopoulos, I. Walulya, M. Papatriantafilou, P. Tsigas, Deterministic Real-time Analytics of Geospatial Data Streams Through ScaleGate Objects, in: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, ACM, New York, NY, USA, 2015, pp. 316–317.
URL http://doi.acm.org/10.1145/2675743.2776758

[15] D. Cederman, V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, P. Tsigas, Brief Announcement: Concurrent Data Structures for Efficient Streaming Aggregation, in: Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, ACM, New York, NY, USA, 2014, pp. 76–78.
URL http://doi.acm.org/10.1145/2612669.2612701

[16] V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, P. Tsigas, ScaleJoin: a deterministic, disjoint-parallel and skew-resilient stream join enabled by concurrent data structures, Tech. rep., Chalmers University of Technology (2014).

[17] M. Callau-Zori, R. Jiménez-Peris, V. Gulisano, M. Papatriantafilou, Z. Fu, M. Patiño Martínez, STONE: A Stream-based DDoS Defense Framework, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, ACM, 2013, pp. 807–812.

[18] V. Gulisano, M. Almgren, M. Papatriantafilou, Online and scalable data validation in advanced metering infrastructures, in: Innovative Smart Grid Technologies Conference Europe (ISGT-Europe), 2014 IEEE PES, 2014, pp. 1–6.

[19] V. Gulisano, M. Almgren, M. Papatriantafilou, METIS: a two-tier intrusion detection system for advanced metering infrastructures, in: Proceedings of the 5th international conference on Future energy systems, ACM, 2014, pp. 211–212.

[20] P. Krill, Java 8 officially arrives at last.
URL http://www.infoworld.com/article/2610817/java/java-8-officially-arrives-at-last.html

[21] P. Krill, Survey: Developers eager for Java 8.
URL http://www.infoworld.com/article/2610155/java/survey--developers-eager-for-java-8.html

[22] B. Goetz, State of the Lambda.
URL http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html

[23] K. Davis, Y. Smaragdakis, J. Striegnitz, Multiparadigm Programming with Object-Oriented Languages, in: J. Hernández, A. Moreira (Eds.), Object-Oriented Technology ECOOP 2002 Workshop Reader, Vol. 2548 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2002, pp. 154–159.
URL http://dx.doi.org/10.1007/3-540-36208-8_13

[24] Haskell Programming Language.
URL https://www.haskell.org/

[25] Erlang Programming Language.
URL http://www.erlang.org/

[26] L. Franklin, A. Gyori, J. Lahoda, D. Dig, LAMBDAFICATOR: From Imperative to Functional Programming Through Automated Refactoring, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 1287–1290.
URL http://dl.acm.org/citation.cfm?id=2486788.2486986

[27] The Scala Programming Language.
URL http://www.scala-lang.org/

[28] Clojure Programming Language.
URL http://clojure.org/

[29] T. Kühne, Internal Iteration Externalized, in: R. Guerraoui (Ed.), ECOOP' 99 — Object-Oriented Programming, Vol. 1628 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1999, pp. 329–350.
URL http://dx.doi.org/10.1007/3-540-48743-3_15

[30] G. Sussman, J. Steele, GuyL., Scheme: A Interpreter for Extended Lambda Calculus, Higher-Order and Symbolic Computation 11 (4) (1998) 405–439.
URL http://dx.doi.org/10.1023/A%3A1010035624696

[31] S. Narain, Lazy evaluation in logic programming, in: Computer Languages, 1990., International Conference on, 1990, pp. 218–227.

[32] T. Johnsson, Efficient Compilation of Lazy Evaluation, in: Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, SIGPLAN '84, ACM, New York, NY, USA, 1984, pp. 58–69.
URL http://doi.acm.org/10.1145/502874.502880

[33] J. Launchbury, A Natural Semantics for Lazy Evaluation, in: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '93, ACM, New York, NY, USA, 1993, pp. 144–154.
URL `http://doi.acm.org/10.1145/158511.158618`

[34] RabbitMQ.
URL `https://www.rabbitmq.com/`

[35] Simple Text oriented Messaging Protocol (STOMP).
URL `http://stomp.github.io/`

[36] J. Kramer, Advanced Message Queuing Protocol (AMQP), Linux J. 2009 (187).
URL `http://dl.acm.org/citation.cfm?id=1653247.1653250`

[37] M. Rostanski, K. Grochla, A. Seman, Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ, in: Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on, IEEE, 2014, pp. 879–884.

[38] Apache Kafka.
URL `http://kafka.apache.org/`

[39] Linkedin.
URL `https://www.linkedin.com/`

[40] The apache software foundation.
URL `http://www.apache.org/`

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

[42] Apache, Apache Zookeeper.
URL `https://zookeeper.apache.org/`

[43] LMAX Exchange, Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads.
URL `http://lmax-exchange.github.com/disruptor/files/Disruptor-1.0.pdf`

[44] S. Madden, M. Franklin, Fjording the stream: an architecture for queries over streaming sensor data, in: Data Engineering, 2002. Proceedings. 18th International Conference on, 2002, pp. 555–566.

[45] M. Noll, Implementing real-time trending topics with a distributed rolling count algorithm in Storm (January 2013).
URL `http://www.michael-noll.com/blog/2013/01/18/implementing-real-time-trending-topics-in-storm/`

[46] EsperTech, Esper.
URL http://www.espertech.com/esper/

[47] Oracle, Java Consumer Interface.
URL https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.htm

[48] Devindra Hardawar, Top 10 social media analytics tools (December 2013).
URL http://venturebeat.com/2013/12/20/top-10-social-media-analytics-tools-the-venturebeat-index/

[49] Twitter Inc, Twitter.
URL https://twitter.com/

[50] H. Kopetz, Internet of Things, in: Real-Time Systems, Real-Time Systems Series, Springer US, 2011, pp. 307–323.
URL http://dx.doi.org/10.1007/978-1-4419-8237-7_13

[51] Euro-SEK data for 2014.
URL http://ratedata.gaincapital.com/2014/

[52] Simple Moving Average - SMA.
URL http://www.investopedia.com/terms/s/sma.asp

[53] Apache Storm Trident.
URL https://storm.apache.org/documentation/Trident-API-Overview.html

[54] J. Daemen, V. Rijmen, Rijndael, the advanced encryption standard, Dr. Dobb's Journal 26 (3) (2001) 137–139.

[55] R. Koster, A. P. Black, J. Huang, J. Walpole, C. Pu, Infopipes for Composing Distributed Information Flows, in: Proceedings of the 2001 International Workshop on Multimedia Middleware, M3W, ACM, New York, NY, USA, 2001, pp. 44–47.
URL http://doi.acm.org/10.1145/985135.985150

[56] C. Herath, B. Plale, Streamflow Programming Model for Data Streaming in Scientific Workflows, in: Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on, 2010, pp. 302–311.

[57] W. Thies, M. Karczmarek, S. Amarasinghe, StreamIt: A Language for Streaming Applications, in: R. Horspool (Ed.), Compiler Construction, Vol. 2304 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2002, pp. 179–196.
URL http://dx.doi.org/10.1007/3-540-45937-5_14

[58] J. H. Spring, J. Privat, R. Guerraoui, J. Vitek, Streamflex: High-throughput Stream Programming in Java, SIGPLAN Not. 42 (10) (2007) 211–228.
URL http://doi.acm.org/10.1145/1297105.1297043

[59] G. Banavar, M. Kaplan, K. Shaw, R. Strom, D. Sturman, W. Tao, Information flow based event distribution middleware, in: Electronic Commerce and Web-based Applications/Middleware, 1999. Proceedings. 19th IEEE International Conference on Distributed Computing Systems Workshops on, 1999, pp. 114–121.

[60] R. E. Strom, G. Banavar, T. D. Chandra, M. A. Kaplan, K. Miller, B. Mukherjee, D. C. Sturman, M. Ward, Gryphon: An Information Flow Based Approach to Message Brokering, CoRR cs.DC/9810019.
URL `http://arxiv.org/abs/cs.DC/9810019`

[61] A. Armbruster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, J. Vitek, A Real-time Java Virtual Machine with Applications in Avionics, ACM Trans. Embed. Comput. Syst. 7 (1) (2007) 5:1–5:49.
URL `http://doi.acm.org/10.1145/1324969.1324974`

[62] G. Bollella, J. Gosling, The real-time specification for Java, Computer 33 (6) (2000) 47–54.

[63] S. Suhothayan, K. Gajasinghe, I. Loku Narangoda, S. Chaturanga, S. Perera, V. Nanayakkara, Siddhi: A Second Look at Complex Event Processing Architectures, in: Proceedings of the 2011 ACM Workshop on Gateway Computing Environments, GCE '11, ACM, New York, NY, USA, 2011, pp. 43–50.
URL `http://doi.acm.org/10.1145/2110486.2110493`

[64] WSO2, WSO2 Complex Event Processor.
URL `http://wso2.com/products/complex-event-processor/`

[65] Apache Spark Community, Apache Spark.
URL `http://spark.apache.org`

[66] M. Zaharia, T. Das, H. Li, S. Shenker, I. Stoica, Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters, in: Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing, HotCloud'12, USENIX Association, Berkeley, CA, USA, 2012, pp. 10–10.
URL `http://dl.acm.org/citation.cfm?id=2342763.2342773`

[67] X. Su, G. Swart, B. Goetz, B. Oliver, P. Sandoz, Changing Engines in Midstream: A Java Stream Computational Model for Big Data Processing, Proc. VLDB Endow. 7 (13) (2014) 1343–1354.
URL `http://dl.acm.org/citation.cfm?id=2733004.2733007`

[68] ZeroMQ Community, Zeromq.
URL `http://zeromq.org`

[69] Real Logic, Aeron.
URL `https://github.com/real-logic/Aeron`

[70] Netty Project Community, Aeron.
URL `http://netty.io/`

A

Code Listings

A.1 Trending Topic use-case implementation

```
1
    Integer TopN = 5;
3   Window win = Window.createTimeBasedWindow( JxTime.Minute, // Size
                        JxTime.Minute, // Advance
5                       null, // Default value
                        (MyTweets x)-> x.timestamp); // timestamp
                            retrieval
7
    /** Operator Lambda to apply to each incoming tuple ****/
9   BiFunction< Map<String,Integer>, MyTweets, Map<String,Integer> >
        operator = (map, newobj) -> {
        if(map == null){
11          map = new HashMap<>();
        }else{
13          if(map.containsKey(newobj.topic)){
                map.put(newobj.topic, map.get(newobj.topic)+1);
15          }else{
                map.put(newobj.topic, 1);
17          }
        }
19      return map;
    };
```

```
21      /** Finisher Lambda to aggregate the result when a window expires
            ***/
        Function< Map<String,Integer>, ArrayList<RankObject> > finisher =
            map ->{
23          Integer i = 0;
            ArrayList<RankObject> rankings = new ArrayList<>();
25          map.entrySet().forEach(e ->{
                rankings.add( new RankObject(e.getKey(), e.getValue()));
27          });
            Collections.sort(rankings);
29          Iterator<RankObject> it = rankings.iterator();
            while(it.hasNext()){
31              RankObject o = it.next();
                if(++i > TopN){
33                  it.remove();
                }
35          }
            map.clear();
37          return rankings;
        };
39
        /**  Stream computation logic ***/
41      JxStream<MyTweets> jstream = JxStream.generate(sup);
        jstream
43          .filter(x-> x != null)
            .Aggregate(operator, finisher, win)
45          .forEach( v->{
                AggregateTuple<ArrayList<RankObject> > agg =
                    (AggregateTuple<ArrayList<RankObject> >) v;
47              ArrayList<RankObject> res = agg.getAggregate();
                System.out.println("Trending topics: " + res.toString() );
49          })
            .start();
```

Listing A.1: Twitter Trending Topic use-case Implementation

A.2 Sensor Monitoring use-case implementation

```
1   SupplierConnectors<String> sup = factory.createNewSupplier();
    Window<String,Double> window =
        Window.createTimeBasedWindow(JxTime.Minute, JxTime.Minute, 0.0,
```

```
3                                                   x->
                                                        Long.valueOf(x.split(";")[2]));
Function<String, Double> fieldExtract =
    x->Double.valueOf(x.split(";")[1]);
5 Function<String, String> groupBy= x -> x.split(";")[0];

7 JxStream<String> stream = JxStream.generate(sup);
JxDataSource<String> q = stream
9         .filter(x->x!=null && !x.isEmpty())
          .subQuery();
11
JxStream<String> stream1 = JxStream.generate(q);
13 PipelineEngine engine =
    stream1
15          .filter(x->x!=null && !x.isEmpty())
          .map(x-> x + ";Stream1")
17          .min(fieldExtract, groupBy, window)
          .forEach(x->{System.out.println("Min: "+x.getTupleCount() +" "+
              x.getAggregate());
19          ;});

21 JxStream<String> stream2 = JxStream.generate(q);
    stream2
23          .filter(x->x!=null && !x.isEmpty())
          .map(x-> x + ";Stream2")
25          .max(fieldExtract, groupBy, window)
          .forEach(x->{System.out.println("Max: "+x.getTupleCount() +" "+
              x.getAggregate());
27          ;});

29 JxStream<String> stream3 = JxStream.generate(q);
    stream3
31          .filter(x->x!=null && !x.isEmpty())
          .map(x-> x + ";Stream3")
33          .average(fieldExtract, groupBy, window)
          .forEach(x->{System.out.println("Average: "+x.getTupleCount() +"
              "+ x.getAggregate());
35          ;});

37 engine.start();
```

Listing A.2: Sensor monitoring Implementation

A.3 JxStream Simple Moving Average Implematation

```
JxStream<String> stream = JxStream.generateParallel(factory,
    numThreads);

//map the text to a list to hold the fields
JxStream<List<Object>> fieldStream = stream.map(x -> {
    List<Object> rtVal = new ArrayList<>();
    int i = 0;
    for (String field : x.split(",")) {
        if (i == 0) {
            rtVal.add(field);
        }
        if (i == 1) {
            rtVal.add(field);
        }
        if (i == 2) {
            rtVal.add(field);
        }
        if (i == 3) {
            rtVal.add(new Long(field));
        }
        if (i == 4) {
            rtVal.add(Double.parseDouble(field));
        }
        if (i == 5) {
            rtVal.add(Double.parseDouble(field));
        }
        i++;
    }
    return rtVal;
});

//filter any tuple with field length less than 6
fieldStream = fieldStream.filter(x -> x.size() >= 6);

//extract the needed fields for computing the SMA
fieldStream = fieldStream.map(fields -> {
    ArrayList<Object> rtVal = new ArrayList<>();
    final long time = (Long) fields.get(3) / 1000;
    final double ratebid = (Double) fields.get(4);
    final double rateask = (Double) fields.get(5);
    String pair = (String) fields.get(2);
```

```
41      rtVal.add(time);
        rtVal.add(pair);
43      rtVal.add(ratebid);
        rtVal.add(rateask);
45      return rtVal;
});
47
//Create a window to use in for the aggregate
49 Window<List<Object>, Double> win = Window.<List<Object>,
        Double>createTimeBasedWindow(120, 30, new Double(0.0), x -> (Long)
        x.get(0));

51 //Insert a throughput monitor to tack performance of pipeline
fieldStream= fieldStream.insertTupleCount("AverageCount");
53
//add the average aggregate
55 JxStream<AggregateTuple<Double>> avgStream
        =fieldStream.average(x->(Double)x.get(2), win);
//Create the sink for the pipeline
57 PipelineEngine engine=avgStream.forEach(x->{});

59 //start the engine
engine.start();
61
//Monitor the engine
63 PipelineMonitor.register("Average Stream", engine);
PipelineMonitor.monitor(2);
```

Listing A.3: SMA implementation in JxStream

B

JxStream API

```
1  package jxstream.api;

3  /**
    *
5   * @author Andy Philogene & Brian Mwambazi
    * @param <T>
7   */
   public interface JxStream<T> extends BaseStream<T, JxStream<T>> {
9
       /**
11      *
        * @param <T>
13      * @param supplier
        * @return a JxStream of type T
15      * @throws java.io.IOException
        */
17     public static <T> JxStream<T> generate(SupplierConnectors<T>
           supplier) throws IOException {
           return new JxStreamImpl(supplier,null,null);
19     }

21     /**
        * This create parallel streams under the hood.The number of streams
            will be equal to the number of cores
23      * available to the JVM
        * @param <T>
```

```
25      * @param supplier
        * @return a JxStream of type T
27      * @throws java.io.IOException
        */
29      public static <T> JxStream<T> generateParallel(SupplierConnectors<T>
            supplier) throws IOException {
            int cores=Runtime.getRuntime().availableProcessors();
31          return generateParallel(supplier,cores);
        }
33
        /**
35      * This create parallel streams under the hood. Since the data
            source is a supplier object a Single Stream is first used
        * to as upstream then split into many streams.
37      * @param <T> The type of tuples
        * @param supplier the source of the stream
39      * @param num number of parallel streams
        * @return a JxStream of type T
41      * @throws IOException
        */
43      public static <T> JxStream<T> generateParallel(SupplierConnectors<T>
            supplier,int num) throws IOException {
                return new JxStreamImpl(supplier,new RoundFlow(num) ,null);
45      }

47      /**
        *
49      * @param <T>
        * @param factory Factory that holds the implementation of a supplier
51      * @return
        * @throws IOException
53      */
        public static <T> JxStream<T> generate(SupplierConnectorsFactory<T>
            factory) throws IOException {
55          return generateParallel(factory, 1);
        }
57
        /**
59      * This create parallel streams under the hood. The number of
            streams will be equal to the number of cores
        * available to the JVM. Using the passed in factory each thread has
            an independent supplier
61      * thereby making this a more optimized parallel stream.
```

```
     * @param <T>
63   * @param factory Factory that holds the implementation of a supplier
     * @return
65   * @throws IOException
     */
67  public static <T> JxStream<T>
        generateParallel(SupplierConnectorsFactory<T> factory) throws
        IOException {
        int cores=Runtime.getRuntime().availableProcessors();
69      return generateParallel(factory, cores);
    }
71
    /**
73   * This create parallel streams under the hood.Using the passed in
         factory each thread has an independent supplier
     * thereby making this a more optimized parallel stream.
75   * @param <T>
     * @param factory Factory that holds the implementation of a supplier
77   * @param num
     * @return
79   * @throws IOException
     */
81  public static <T> JxStream<T>
        generateParallel(SupplierConnectorsFactory<T> factory,int num)
        throws IOException {

83          List<SupplierConnectors<T>> connectors=new ArrayList<>();

85          for (int i = 0; i < num-1; i++) {
                connectors.add(factory.createNewSupplier());
87          }

89          return new JxStreamImpl(connectors,null);
    }
91
    /**
93   *
     * @param <T>
95   * @param queue
     * @return a JxStream of type T
97   * @throws java.io.IOException
     */
```

```
99      public static <T> JxStream<T> generate(JxDataSource<T> queue) throws
            IOException {
            return new JxStreamImpl(queue,null);
101     }

103     /**
         *
105      * @param <T>
         * @param queue
107      * @return
         */
109     public static <T> JxStream<T> generateParallel(JxDataSource<T>
            queue) {
            throw new UnsupportedOperationException("Not supported yet.");
111     }

113     /**
         *
115      * @param <T>
         * @param queue
117      * @param num
         * @return
119      */
        public static <T> JxStream<T> generateParallel(JxDataSource<T>
            queue, int num) {
121         throw new UnsupportedOperationException("Not supported yet.");
        }
123
        /**
125      *Computes the whole topology
         */
127     void compute();

129     /**
         *
131      * @return
         */
133     JxDataSource<T> subQuery();

135     /**
         * This method aggregate the tuples according to the parameters.
137      * If the stream was parallelized, it will be merge into one stream,
         * before computing the aggregate, for consistency purposes.
```

```
139      * @param <R>
         * @param operator
141      * @param window
         * @return
143      */
        <R>  JxStream<AggregateTuple<R>>
145         Aggregate(BiFunction<R, T, R> operator, Window<T,R> window);

147     /**
         * This method aggregate the tuples according to the parameters.
149      * If the stream was parallelized, it will be merge into one stream,
         * before computing the aggregate, for consistency purposes.
151      * @param <I>
         * @param <R>
153      * @param operator
         * @param finisher
155      * @param window
         * @return
157      */
        <I, R> JxStream<AggregateTuple<R>>
159         Aggregate(BiFunction<I, T, I> operator, Function<I, R> finisher,
               Window<T,I> window);

161     /**
         *
163      * @param <I>
         * @param operator
165      * @param combiner
         * @param finisher
167      * @param window
         * @return
169      */
        <I> JxStream<AggregateTuple<Double>>
171         aggregate(BiFunction<I, T, I> operator, BiFunction<I, I, I>
               combiner, Function<I, Double> finisher, Window<T, I> window);

173     /**
         *
175      * @param <K> Type of the Key to group by
         * @param <I>
177      * @param <R>
         * @param operator
179      * @param finisher
```

```
     * @param groupBy
181  * @param window
     * @return
183  */
    <K, I, R> JxStream< AggregateTuple<Map<K,R>> >
185     aggregate( BiFunction<I, T, I> operator, Function<I, R>
            finisher, Function<T,K> groupBy, Window<T, I> window);

187 /**
     *
189  * @param <K> Type of the Key to group by
     * @param <I>
191  * @param <R>
     * @param operator
193  * @param combiner
     * @param finisher
195  * @param groupBy
     * @param window
197  * @return
     */
199 <K, I, R> JxStream< AggregateTuple<Map<K,R>> >
        aggregate( BiFunction<I, T, I> operator, BiFunction<I, I,
            I>combiner, Function<I, R> finisher, Function<T, K> groupBy,
            Window<T, I> window);
201
    /**
203  *
     * @param numberExtractor
205  * @param window
     * @return
207  */
     JxStream<AggregateTuple<Double>>
209     Min(Function<T, Double> numberExtractor, Window<T,Double>
            window);

211 /**
     *
213  * @param <K> Type of the Key to group by
     * @param numberExtractor
215  * @param groupBy
     * @param window
217  * @return
     */
```

```
219     <K> JxStream<AggregateTuple<Map<K,Double>>>
            min( Function<T, Double> numberExtractor, Function<T, K>
                groupBy, Window<T, Double> window);
221     /**
         *
223      * @param numberExtractor
         * @param window
225      * @return
         */
227      JxStream<AggregateTuple<Double>>
            Max(Function<T, Double> numberExtractor, Window<T,Double>
                window);
229
        /**
231      *
         * @param <K> Type of the Key to group by
233      * @param numberExtractor
         * @param groupBy
235      * @param window
         * @return
237      */
        <K> JxStream< AggregateTuple<Map<K, Double>> >
239         max(Function<T, Double> numberExtractor, Function<T, K> groupBy,
                Window<T, Double> window);

241     /**
         *
243      * @param numberExtractor
         * @param window
245      * @return
         */
247      JxStream<AggregateTuple<Double>>
            sumAggregate(Function<T, Double> numberExtractor,
                Window<T,Double> window);
249
        /**
251      *
         * @param <K> Type of the Key to group by
253      * @param numberExtractor
         * @param groupBy
255      * @param window
         * @return
257      */
```

```
    <K> JxStream<AggregateTuple<Map<K, Double>>>
259         sumAggregate(Function<T, Double> numberExtractor, Function<T, K>
                groupBy, Window<T,Double> window);

261     /**
         *
263      * @param fieldExtractor
         * @param window
265      * @return
         */
267      JxStream<AggregateTuple<Double>>
            average(Function<T, Double> fieldExtractor, Window<T,Double>
                window);
269
        /**
271      *
         * @param <K> Type of the Key to group by
273      * @param fieldExtractor
         * @param groupBy
275      * @param window
         * @return
277      */
        <K> JxStream<AggregateTuple<Map<K, Double>>>
279         average(Function<T, Double> fieldExtractor, Function<T, K>
                groupBy, Window<T,Double> window);

281     /**
         *
283      * @param fieldExtractor a function to extract the number to be used
         * @param window definition of the window
285      * @return a JxStream of type AggregateTuple< Double>
         */
287      JxStream<AggregateTuple<Double>>
            median(Function<T, Double> fieldExtractor, Window<T,Double>
                window);
289
        /**
291      *
         * @param <K> Type of the Key to group by
293      * @param fieldExtractor
         * @param groupBy
295      * @param window
         * @return
```

```
297      */
         <K> JxStream<AggregateTuple<Map<K,Double>>>
299          median(Function<T, Double> fieldExtractor, Function<T, K>
                 groupBy, Window<T, Double> window);

301      /**
          *
303       * @param <R>
          * @param <A>
305       * @param collector
          * @return
307       */
         <R, A> R collect(Collector<? super T, A, R> collector);
309
         /**
311       *
          * @return
313       */
         Stream<T> toStream();
315
         /**
317       *
          * @param action
319       * @return
          */
321       JxStream<T> peek(Consumer<? super T> action);

323      /**
          * Limit a stream to a maximum tuple count to processs.
325       * If the stream is parallelized, the same limit will be applied to
              each thread
          * @param maxSize
327       * @return a JxStream of type T
          */
329      JxStream<T> limit(long maxSize);

331      /**
          *
333       * @param predicate
          * @return
335       */
         JxStream<T> filter(Predicate<? super T> predicate);
337
```

```
    /**
339  *
     * @param <R>
341  * @param mapper
     * @return
343  */
    <R>  JxStream<R> map(Function<? super T, ? extends R> mapper);
345
    /**
347  *
     * @param <R>
349  * @param mapper
     * @return
351  */
    <R>  JxStream<R> flatMap(Function<? super T, ? extends Stream<?
       extends R>> mapper);
353
    /**
355  *
     * @param action
357  * @return
     */
359 PipelineEngine forEach(Consumer<? super T> action);

361
    /**
363  *
     * @return
365  */
    PipelineEngine getPipelineEngine();
367
    /**
369  * This will create a tuple count meter that can be later monitored
        using the {@link PipelineMonitor}
     * @param name of this
371  * @return
     */
373 JxStreamImpl<T> insertTupleCount(String name);

375 /**
     *
377  * @return
     */
```

```
379     JxStream<T> convergeParallelStreams();

381 }
```

Listing B.1: JxStream Application Interface