Spark Steaming Framework for Large-Scale Multi-Stream Data Analytics

Mr. Tanwa Sirisakdiwan

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science in Computer Science
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Academic Year 2018

เฟรมเวอร์คสปาร์คสตรีมมิ่งสำหรับการวิเคราะห์มัลติสตรีมขนาดใหญ่

นายธันวา ศิริศักดิวรรณ

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาวิทยาศาสตร์คอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2561

Thesis Title	Spark Steaming Framework for Large-Scale Multi-Stream Data Analytics
By	Mr. Tanwa Sirisakdiwan
Field of Study	Computer Science
Thesis Advisor	Assistant Professor NATAWUT NUPAIROJ, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirement for the Master of Science

.... Dean of the Faculty of Engineering

(Professor SUPOT TEACHAVORASINSKUN, Ph.D.)

THESIS COMMITTEE

.... Chairman

(Assistant Professor KRERK PIROMSOPA, Ph.D.)

.... Thesis Advisor

(Assistant Professor NATAWUT NUPAIROJ, Ph.D.)

.... Examiner

(Duangdao Wichadakul, Ph.D.)

.... External Examiner

(Kanchana Silawarawet, Ph.D.)

ธันวา ศิริศักดิวรรณ : เฟรมเวอร์คสปาร์คสตรีมมิ่งสำหรับการวิเคราะห์มัลติสตรีมขนาดใหญ่. (Spark Steaming Framework for Large-Scale Multi-Stream Data Analytics) อ.ที่ปรึกษาหลัก : ผศ. ดร.ณัฐวุฒิ หนูไพโรจน์

โปรแกรมการประมวลผลข้อมูลแบบสตรีมในเวลาจริงด้วยข้อมูลที่ไม่เหมือนกันได้รับความสนใจเป็นอย่างมาก โดยเฉพาะในอินเทอร์เน็ตของสิ่งต่างๆซึ่งผลิตข้อมูลจากเซนเซอร์จำนวนมากในรูปแบบของข้อมูลสตรีมมิ่ง ทั้งนี้ยังคงมีปัญหามากมายโดยเฉพาะอย่างยิ่งปัญหาในการเปิดใช้และการบำรุงรักษาของ Spark Structured Streaming งานวิจัยนี้ขอเสนอ กรอบการทำงานของสปาร์คเพื่อการประมวลผลของข้อมูลที่ไม่เหมือนกันแบบหลายสตรีมโดยเน้นความง่ายในการเปิดใช้และการจัดการการกำหนดอันเหมาะสม โดยจะเป็นไลบรารี่ช่วยให้สามารถปรับใช้การประมวลผลของข้อมูลที่ไม่เหมือนกันแบบหลายสตรีมโดยใช้สปาร์คเพียงโปรแกรมเดียวซึ่งสามารถลดความยากในการปรับใช้ การตรวจสอบ ลดความฟุ่มเฟือยของโค้ดและแก้ปัญหาความไม่มีประสิทธิภาพในการเข้าคิวของงานในการประมวลผลของข้อมูลที่ไม่เหมือนกันแบบหลายสตรีม

สาขาวิชา	วิทยาศาสตร์คอมพิวเตอร์	ลายมือชื่อนิสิต ....
ปีการศึกษา	2561	ลายมือชื่อ อ.ที่ปรึกษาหลัก ....

5970199021 : MAJOR COMPUTER SCIENCE
KEYWORD: Real-Time, Apache Spark, Multiple Stream, Data Stream Processing, Heterogeneous Data

Tanwa Sirisakdiwan : Spark Steaming Framework for Large-Scale Multi-Stream Data Analytics. Advisor: Asst. Prof. NATAWUT NUPAIROJ, Ph.D.

Real-time streaming applications with multiple heterogeneous data streams have become increasingly popular especially in IoT applications where huge amount of sensors produce large amount of data in the form of data streams. However, many issues still exist, especially in deploying and maintaining these large amounts of data streams. Using Spark Structured Streaming, this research introduces a Spark Streaming framework for multiple heterogeneous data streams which focuses on the ease of deployment and proper scheduling. Our proposed framework is a library that allows the deployment of multiple heterogeneous data stream processing in a single Spark application. Our framework can reduce deployment difficulties, coding redundancy, monitoring difficulties, and solve the problem of inefficient job queueing in multi-stream applications.

Field of Study:	Computer Science	Student's Signature ....
Academic Year:	2018	Advisor's Signature ....

ACKNOWLEDGEMENTS

First I would like to thank the most important person my advisor Asst. Prof. Natawut Nupairoj for his expert advice and insight into overseeing this research. Even though he is a very busy person, he always contribute to me the times he has available. His dedication to advising me during this research gave me hope and wills to fight on during hard times.

Finally I would like to thank both of my parent for being there for me during difficult times, providing my with much needed emotional support.

Thank you all very much.

Tanwa Sirisakdiwan

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

1 Introduction

1.1 Motivations

Internet-of-Thing (IoT) has been widely used in many industries. Such as energy, manufacturing, and many others which involve the use of complex machinery. These sensors provide us with high volume of data and often at a very high velocity. Data generated from these devices are typically varied and heterogeneous. For example, power generators are being constantly monitored to prevent failures in many power plants. These generators usually have over 2000 sensors. Many of these sensors generate heterogeneous data which requires their own independent stream processing. Deploying real-time application such as anomaly detection in this case is very difficult as the processing need to be in real-time and the number of stream processing needed to be deployed is large. There are many definitions of heterogeneous data as mentioned in [1]. In this research, we shall define heterogeneous data as data that is generated from different sources with variety of sampling periods and independent machine learning models, such as anomaly detection, for each data stream.

Currently, there are many issues related to real-time analytics of multiple heterogeneous data, especially when deadlines are different. In Apache Spark [2], the conventional way to perform a streaming job is to run one Spark application per data stream. This can be a problem when we are dealing with multiple streams from different sources as the number of applications that needed to be launched grows. In addition, when there are multiple applications running simultaneously, the default resource scheduling between applications in Spark is FIFO (First-in-First-Out) policy. This, while give more priority to jobs which arrive first, causes problems when multiple streaming jobs are concurrently running. For example, a large size job may be the first in the queue and consume all available resources blocking smaller jobs with more immediate deadline. This caused the smaller job to be delayed and missed its deadline. At this moment, there is no proposed solution to these practical problems, especially for multiple heterogeneous data stream in Spark.

To solve these problems, this research introduces a framework for Spark which can launch multiple streaming jobs in a scalable manner, reduce coding redundancy involved, ease monitoring difficulty, and provide a proper solution to resource scheduling. This will provide a more suitable solution to use cases such as real-time anomaly detection of large number of sensors

1.2 Objectives

To develop a framework for spark streaming which allow efficient coding and deployment of multiple streaming process and provide proper scheduling methods for these jobs.

1.3 Scope of Work

- To develop an efficient and scalable coding framework to allow efficient coding and deployment of large amount of machine learning model on heterogenous data and handles configurating the appropriate job scheduling policy.

1.4 Research Procedure

1. Studies of knowledge involved and related works.
2. Analysis of research problems.
3. Experiment with options to find solution for research problem.
4. Development of proposed solution.
5. Experiment and evaluation.
6. Conclude research result.
7. Write research papers and thesis.

1.5 Expected Benefits

A Coding framework which allow efficient coding and deployment of large-scale multiple stream processing. As well as provide an appropriate job scheduling policy.

1.6 Content Structure

This thesis consists of 6 Chapters. The 6 chapters are the following:

1. Introduction
2. Technical background and related works
3. Spark Streaming Framework for Large-Scale Multi-Stream Data Analytics
4. Job Scheduling for Multi-Stream
5. Conclusion

1.7 Published Paper

- Spark Steaming Framework for Large-Scale Heterogeneous Data Analytics. Published at 2019 2nd International Conference on Communication Engineering and Technology (ICCET). Held on 12-15 April 2019 in Nagoya Institute of Technology, Nagoya, Japan.

2 Technical Backgrounds and Related Works

2.1 Technical Backgrounds

2.1.1 IoT Streaming Data

IoT (Internet of things) is the term commonly used for a network of interconnected devices, such as sensors as well as, devices, social media, health care applications, temperature sensors, various other software applications and digital devices. Data generated from IoT comes from various sensors. This creates heterogeneity, noise, variety, and rapid growth in size [3]. There are many applications for big IoT data, such as E-commerce, Smart cities, healthcare, and retail & logistic as shown in Figure 1.

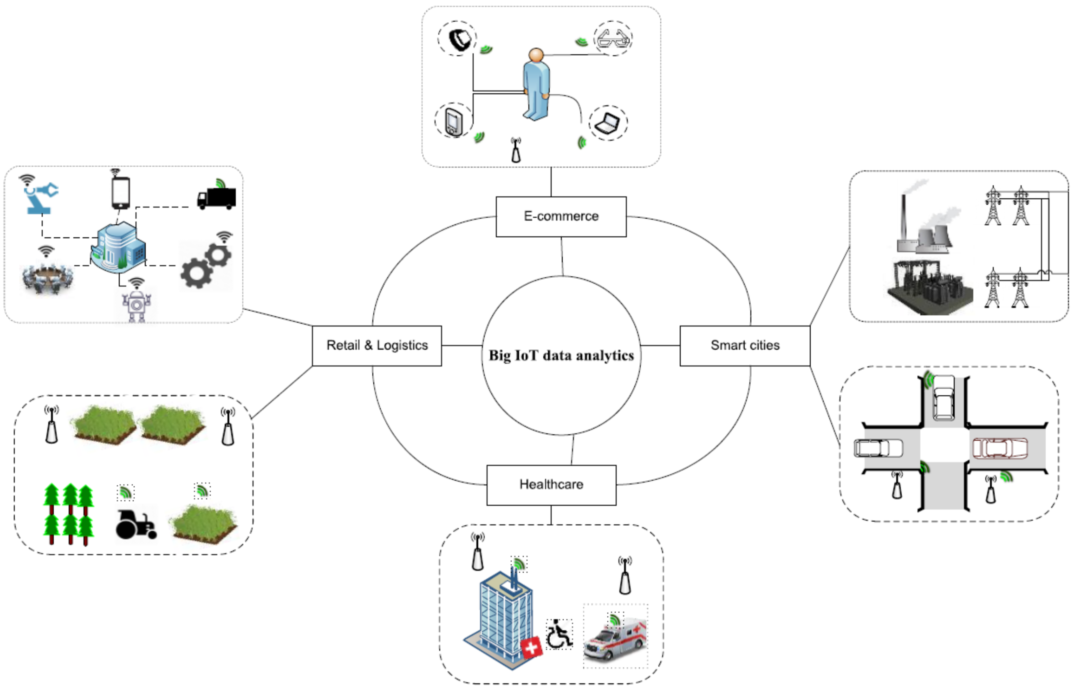


Figure 1 – Example of use cases and opportunities for big IoT data analytics architecture.

IoT data is usually generated in the form of streaming data from sensors of various type and from vast number of sources. There are several characteristics of IoT streaming data. First, streaming data have sampling periods, the time difference between two consecutive samples, i.e. new data is generated periodically creating a constant stream of data. Second, they are generated from many data sources, usually come in massive in numbers, making each stream heterogeneous. In addition, depending on the usage, some data may have deadlines. Deadline is the time that the data processing is expected to be completed. In general, this is usually the sampling period of the sensor. The time of each deadlines varies and is usually context dependent, therefore each sensor can have different deadline. Figure 2 shows data taken from a sensor in a combined cycle power generator. A typical generator of this type usually contains over thousands of sensors of various type.

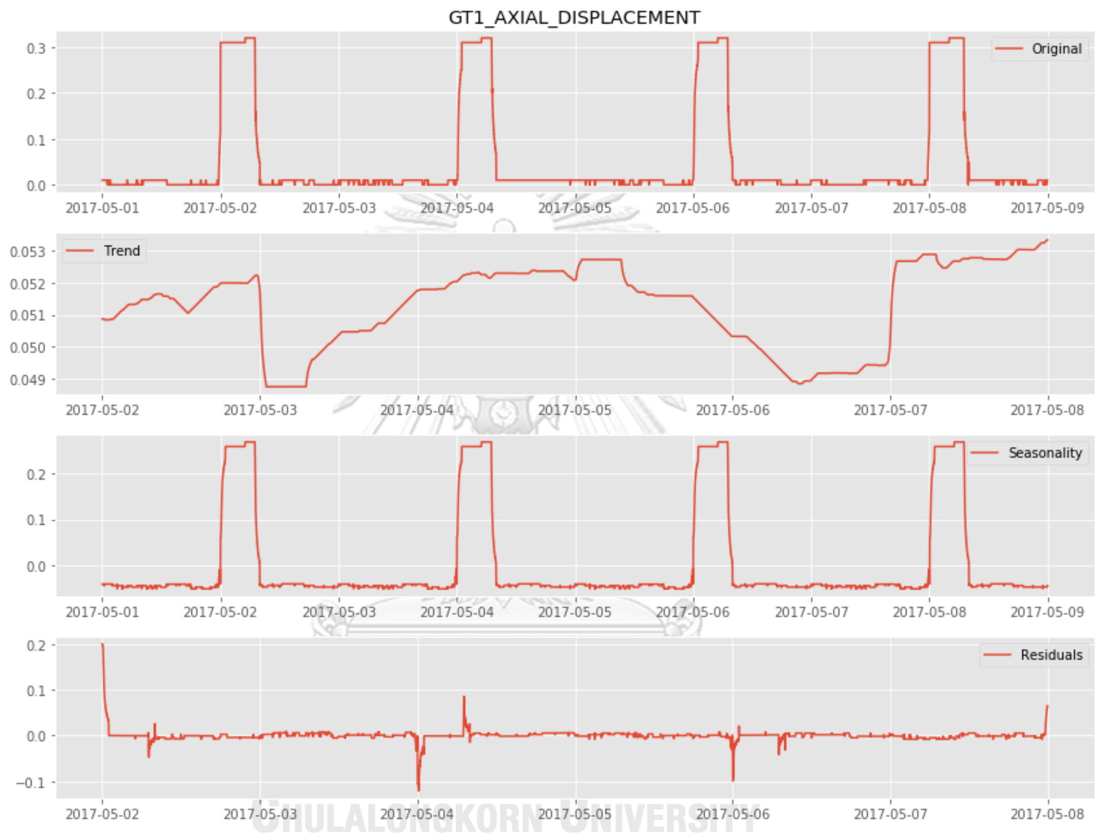


Figure 2 - Data from Combined Cycle Power Generator

2.1.2 Heterogeneous Data

There are many definitions of heterogeneous data as mentioned in [1]. These definitions are:

- **Syntactic heterogeneity** – when two data sources are not expressed in the same language.
- **Conceptual heterogeneity** – difference in modelling the same domain of interest. Also known as semantic heterogeneity.
- **Terminological heterogeneity** – variation in name when referring to the same object from difference sources.
- **Semiotic heterogeneity** – different interpretation of entities by people. Also known as pragmatic heterogeneity.

As these definitions does not really fit with our experiment, for this research, we shall define heterogeneous data as *data that is generated from different sources, having variety of sampling periods, and independent machine learning models, such as anomaly detection, for each data stream.*

2.1.3 Apache Spark

Spark is a unified analytic engine for processing large-scale data, with built-in modules for streaming, SQL, machine learning, and graph processing as presented in Figure 3. Spark's core is RDD (resilient distributed dataset). An RDD [4] is a distributed memory abstraction of data partitioned across set of machines that can be rebuilt if a partition is lost. We will explain Spark Streaming in further detail in Section 3.

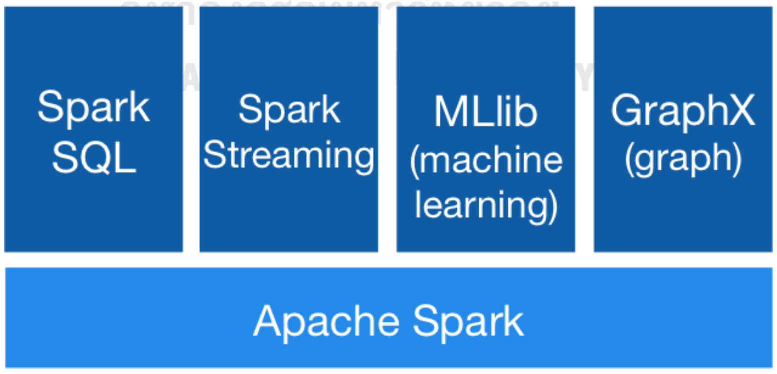


Figure 3 – Spark Framework

2.1.4 Spark Streaming

Spark's stream processing API, called Spark streaming, treats streaming data as a continuous series of unbounded micro-batches where available data is processed at every interval. This is called Dstreams or discretized stream [5]. Figure 4 shows the illustration of the high-level overview of spark streaming.

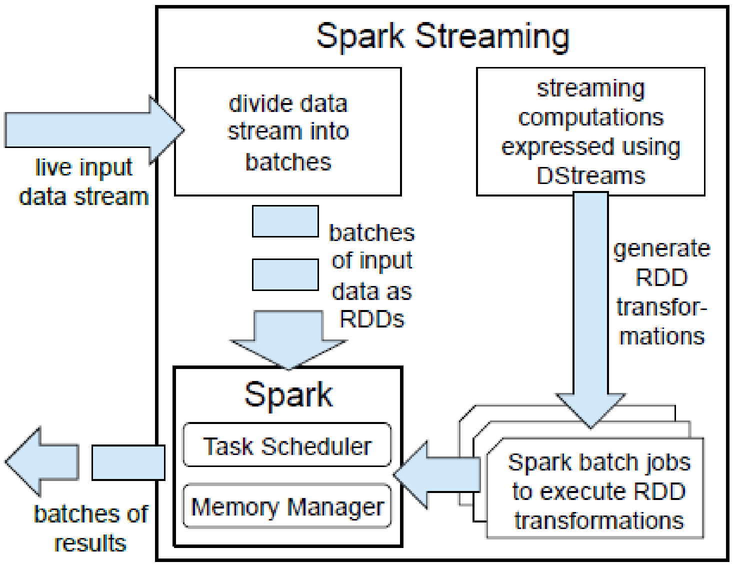

Spark Streaming
live input data stream
divide data stream into batches
streaming computations expressed using DStreams
batches of input data as RDDs
generate RDD transfor-mations
Spark
Task Scheduler
Memory Manager
Spark batch jobs to execute RDD transformations
batches of results

Figure 4 - High level overview of Spark Streaming API. Taken from [9]

2.1.5 Spark Structured Streaming

Spark Structured Streaming [6] is Spark's next API for stream processing. It aims to make stream processing easier and more accessible by allowing developer to create streaming application without having to reasons with streaming related configurations. Structured streaming offers much more processing methods than its predecessor Spark's Streaming. Most Spark's Data-frame Operations can be directly applied to Structured Streaming. As shown in Figure 5,Spark Structured Streaming operates on Spark's new Data Frame API and treats streaming data as unbounded tables of data instead, where every trigger (intervals) spark read all new data available from the data source. Figure 6 explains how Spark keeps track of old/new data by using its state management,

Our framework is based on Spark Structured Streaming, from this point on we will refer to Spark Structured Streaming as Spark Streaming as it is the new streaming API promoted by Spark and the old Spark Streaming is no longer in development.

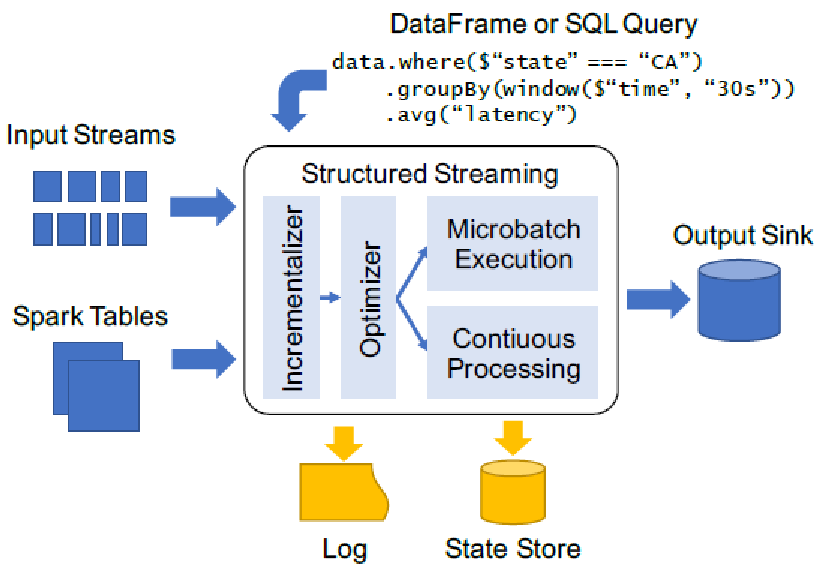


Figure 5 - Components of Structured Streaming. Taken from [10]

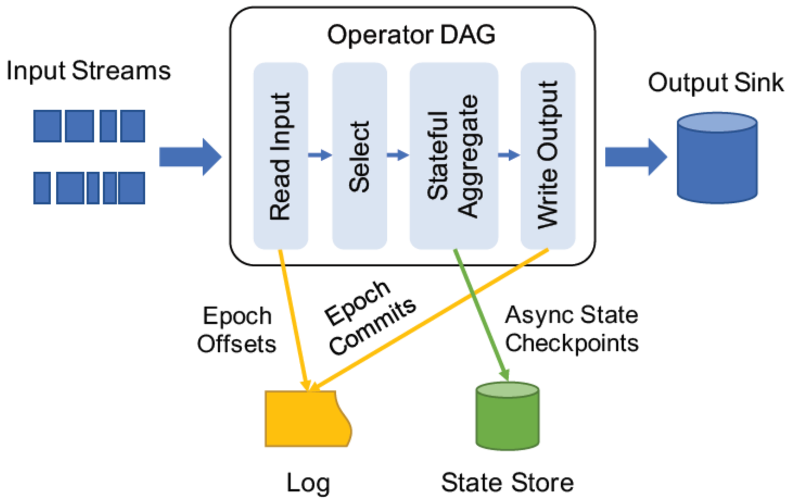


Figure 6 - State Management of Structured Streaming during execution, Taken from [10]

2.2 Related Works

2.2.1 Hybrid big data architecture for high-speed log anomaly detection

Real-time anomaly detection on streaming data has been previously achieved by [7] where network log data is analyzed in real-time using SARIMA anomaly detection. SARIMA (Seasonal Autoregressive Integrated Moving Average) is a statistical model commonly used for anomaly detection in time-series data with seasonality components. The reason SARIMA model could not be apply directly to streaming data is because SARIMA requires historical data to be preprocessed to create its model making it inapplicable.

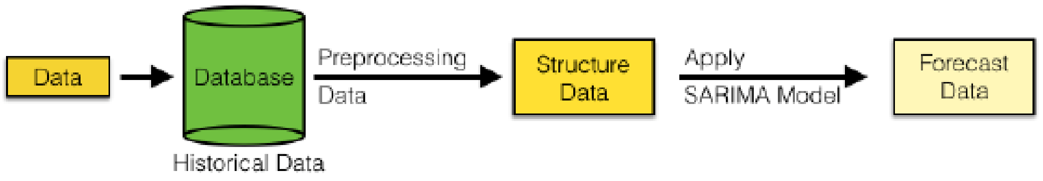


Figure 7 - SARIMA-based Anomaly Detection System

This research proposed a hybrid data processing architecture based on lambda architecture as presented in Figure 8, which allow SARIMA to be applied on to streaming data in real-time. Their approach to applying SARIMA in real-time is to use a bi-model approach instead, shown in Figure 9. This bi-model solution uses their proposed architecture to train SARIMA model with historical data in batch processing. The model is then applied onto streaming data in real-time.

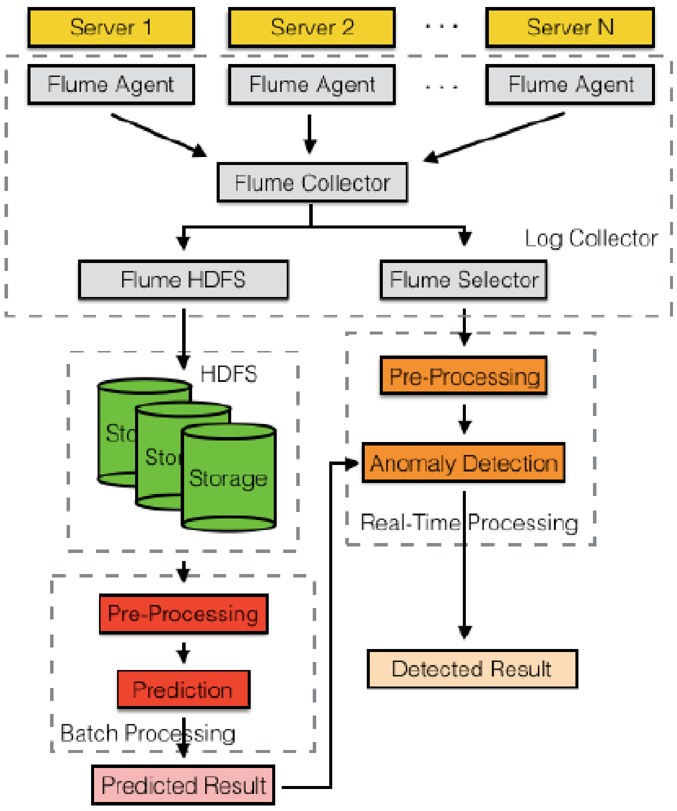


Figure 8 - Hybrid Big Data Architecture

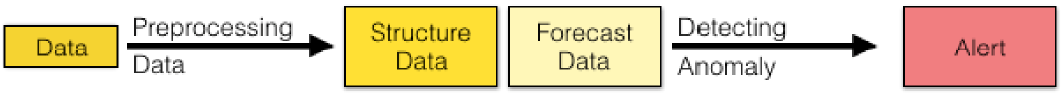


Figure 9 - Bi-Modal SARIMA-Based Anomaly Detection System

2.2.2 Real-time High-Performance Anomaly Detection over Data Streams: Grand Challenge

Real-time anomaly detection on N-number of data streams was proposed in [8]. It handles multiple data streams by dividing incoming data into data windows based on their time intervals. After each interval, all window from each stream is then processed in parallel using multi-threading as shown in Figure 10. However, this framework has limitation as it assumes identical period for all stream.

In Figure 11 the input component receives data from message broker in RDF format which is then parsed for the data window manager to build data window. The anomaly detector component receives the list of windows data and processes them in parallel. The processing includes clustering the data inside the window and calculating the Markov chain transition matrix and transition threshold which is then compared with data for detection result. The result of the anomaly detection component is forwarded to the output component for reporting.

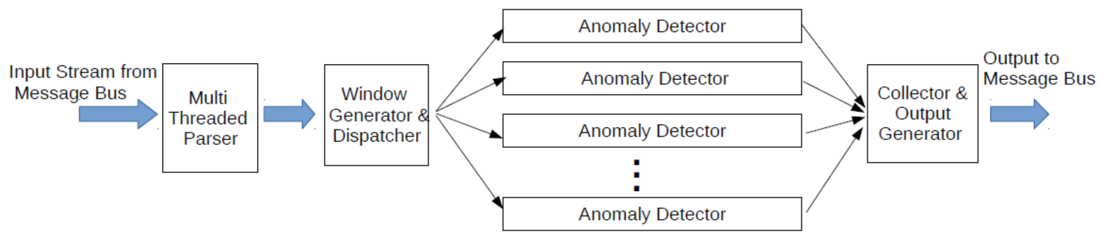


Figure 10- Architecture of Multi-Threaded/Multi-Processed Anomaly Detection System

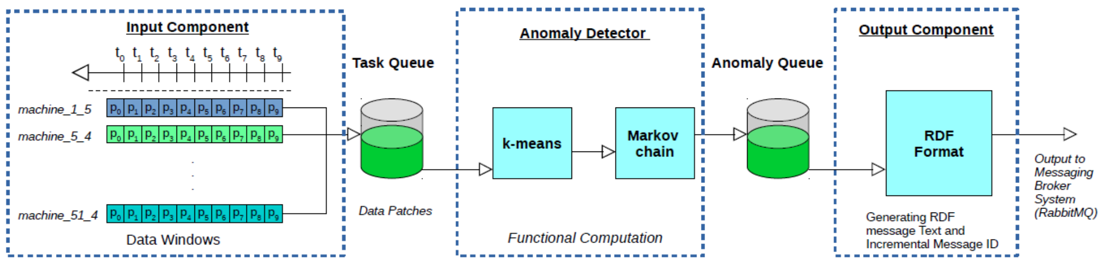

Input Component
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9
machine_1_5
machine_5_4
machine_51_4
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9
Data Windows
Task Queue
Data Patches
Anomaly Detector
k-means
Markov chain
Functional Computation
Anomaly Queue
Output Component
RDF Format
Generating RDF message Text and Incremental Message ID
Output to Messaging Broker System (RabbitMQ)

Figure 11 - Architecture of Anomaly Detection System.

2.2.3 Adaptive Scheduling of Parallel Jobs in Spark Streaming

An adaptive scheduler for Spark streaming was introduced by [9]. This adaptive scheduler dynamically adjusts resource scheduling for each stream. They have achieved this through extending Spark Streaming and uses "*spark.concurrentJob*", a config which control the amount of parallelism, to allow more than one micro-batch of each stream to be processed at a time.

This configuration is however not recommended by Spark and is not documented into Spark's official API. The problem of using this configuration is that it will cause some micro-batch to be processed incorrectly in the case that a batch-to-batch dependency exists. A batch-to-batch dependency is when the result of a micro-batch is not only determined by the data contained but also prior micro-batch e.g. window operation as shown in Figure 12.

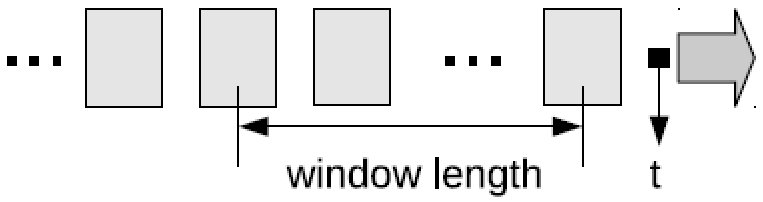


Figure 12 - Micro-batch Dependency

This research implemented a solution which classifies jobs by their job dependency then assigning them into appropriate job pools; as shown in Figure 13. Jobs without micro-bath dependency is put into the same pool to utilize the concurrent job setting and jobs with dependency is separated into another pool as they are not compatible with the setting. Classification is done by checking the job's directed acyclic graph (DAG). The Adaptive Optimizer takes performance statistic from Spark Engine and adaptively adjust the pool resource weight and the amount of parallelism through reinforcement learning after each micro-batch of execution. The workflow of Adaptive scheduler is shown in Figure 14.

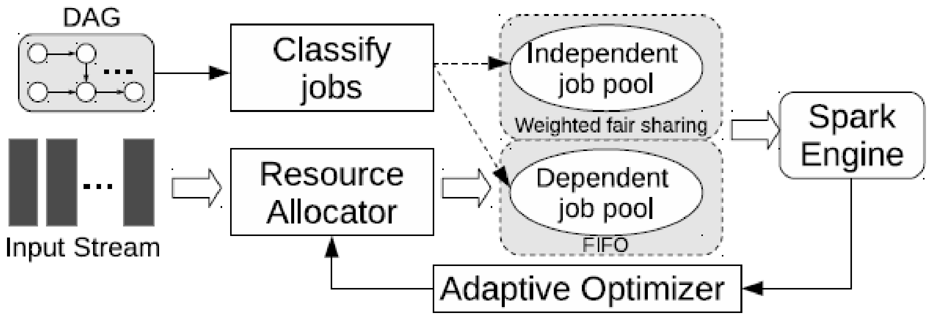


Figure 13 - Overview of Adaptive Scheduler

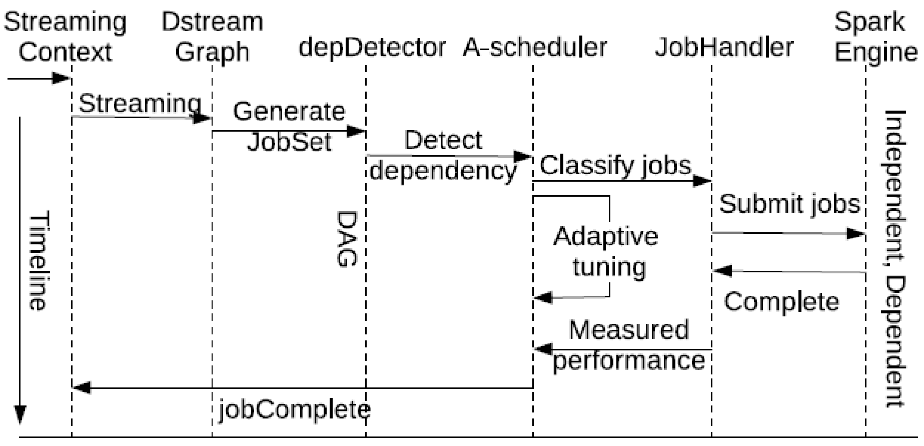
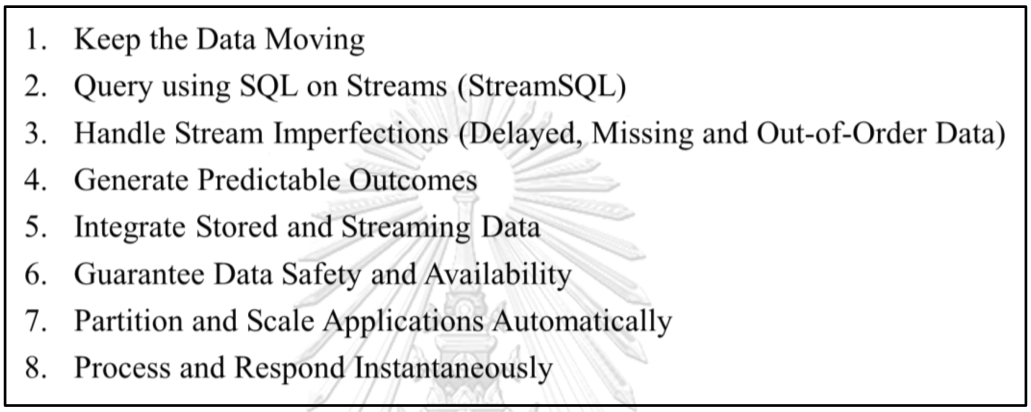


Figure 14 - Adaptive scheduler Workflow

2.2.4 The 8 Requirements of Real-time Stream Processing

In research [10], the 8 requirements of real-time stream processing is purposed. We want to emphasize on the 8th requirement which states that "a stream processing must be able to keep up a highly-optimized, minimal-overhead execution engine to deliver real-time response for high-volume applications". As the tools being used in this research, Spark Structured Streaming, can satisfy most requirements by default.



1. Keep the Data Moving
2. Query using SQL on Streams (StreamSQL)
3. Handle Stream Imperfections (Delayed, Missing and Out-of-Order Data)
4. Generate Predictable Outcomes
5. Integrate Stored and Streaming Data
6. Guarantee Data Safety and Availability
7. Partition and Scale Applications Automatically
8. Process and Respond Instantaneously

Figure 15 - The 8 Requirements of Real-Time Stream Processing

3 Spark Streaming Framework for Large-Scale Multi-Stream Data Analytics.

In this chapter, we provide more information on what Multi-Stream is, as well as, the problems it introduces, previously solutions to the problems, and finally we introduce our solution. Spark Streaming Framework for Large-Scale Multi-Stream Data Analytics. This framework aims to provide a more suitable solution to use cases such as real-time anomaly detection of large number of sensors.

3.1 Multi-Stream Definition

We defined a single-streams as a streaming process where all data passes through a single processing function before it is then send to the data sink. Figure 16 shows diagram of our definition of a single stream.

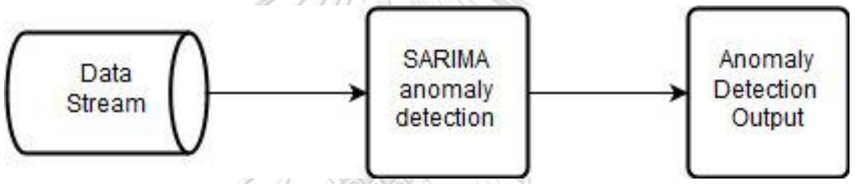


Figure 16 – Single Stream

Multi-stream is a streaming process where incoming data is divided into multiple streams and processed through its unique processing function before it is then sent to the data sink. Figure 17 shows diagram of our definition of a multi-stream.

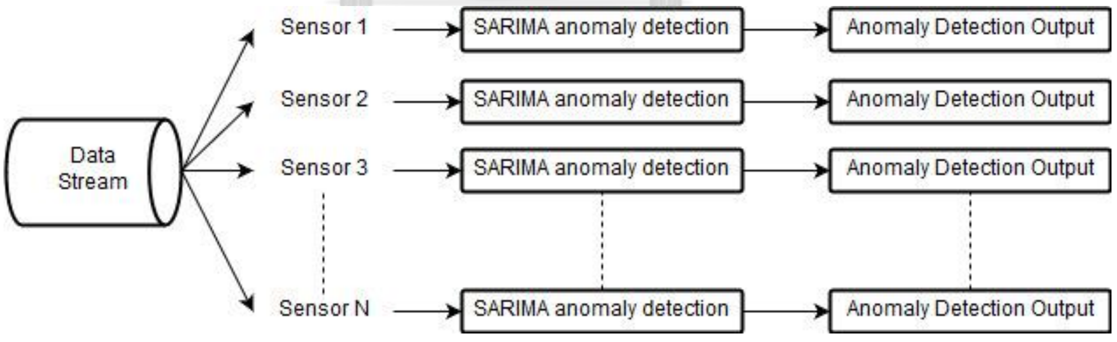


Figure 17- Multi-Stream

3.2 N-Application Multi-Stream Processing

Previously, the only way to operate multi-stream is to use N-application approach, where N is the number of streams. In N-application method, an application developed and deployed for each data stream. Each application is responsible for its own data stream. There are many issues related to managing large amount of these applications.

3.2.1 Application Submission

In Spark, it is impossible to submit multiple applications at once. Applications must be submitted one at a time via spark-submit. This can be very hard to manage when the number of applications grow sufficiently large. The only way to ease up this process is by preparing a cron script, a time-based job scheduler in Unix-like computer operating systems, to submit each application. This can cause many problems when running large amount of Spark applications.

3.2.2 Resource Overhead

A Spark application requires its own Spark executor. A Spark executor consumes, additional resource. By default, a Spark Driver program consume a minimum of 300mb of memory. Using an application per data stream in large scale multi streaming system can be very costly.

3.2.3 Monitoring Difficulty

Monitoring difficulty also increases as most spark monitoring tool, such as Spark web user-interface, only provide monitoring of individual application; leaving cluster-wide monitoring-tools like Ganglia [11] as the only few options.

3.2.4 Coding Redundancy

Deploying large amount of applications also causes a lot of redundant coding of configuring and instantiating Spark sessions.

3.2.5 Job Scheduling

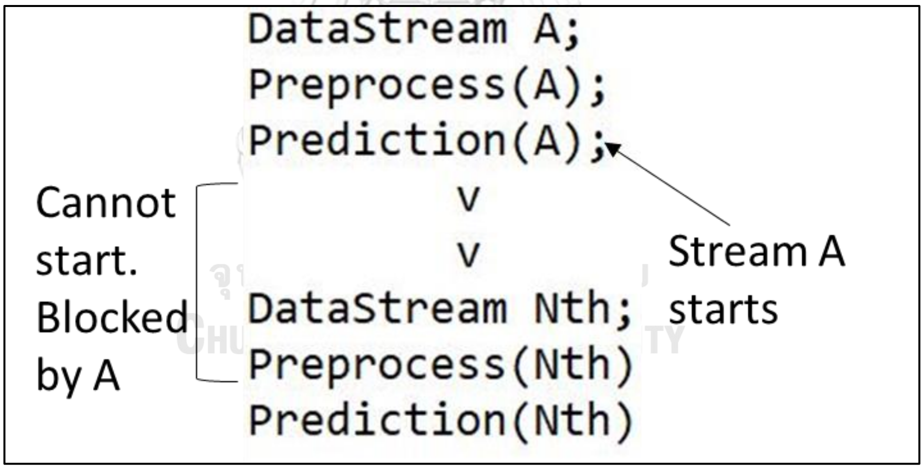
The default job scheduling is not suitable for handling multiple heterogeneous streams.

3.3 Difficulties in Multi-Stream in Single Application

Our final approach is to launch multiple stream simultaneously on a different thread. By running each stream on a different thread, the stream process continues to be a blocking action but does not block any execution as it is isolated within its own thread allowing the program to continue. We will go into more detail of our implementation later in this chapter. First, we will explain the difficulty of deploying multi-stream in single application.

3.3.1 Blocking Action in Multi-Stream in Single Application

Previously, when we try to fit multiple streaming process into a single application, some processes will not launch. In Spark, a stream processing is a blocking action. This meant that applications will wait until the streaming process finishes before moving on to the next line of code. Therefore, in a normal manner of coding the first streaming process that starts will block the rest of the code in the program from running, hence other stream process cannot start as shown in Figure 18.



```
DataStream A;
Preprocess(A);
Prediction(A);
      v
      v
DataStream Nth;
Preprocess(Nth)
Prediction(Nth)
```

Figure 18 - Blocking action of multi-stream in single application

3.3.2 Redundant Data Reading in Spark Streaming

In Spark Streaming, when multiple stream processing shares the same data, it is commonly understood that data from the input stream is read once and distributed amongst each stream process as shown Figure 19.

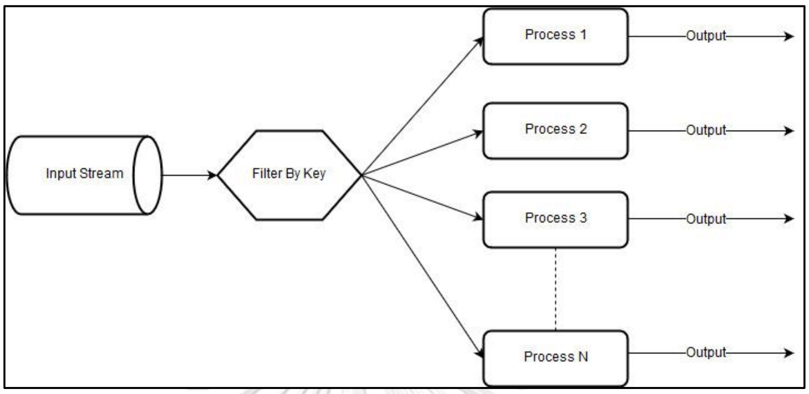


Figure 19 – Expected Workflow of Multi-Stream Behavior

However, after testing and tracing the execution DAG, we have found that multiple readings occurred at the input stage. Figure 20 illustrated what happened during execution. Input is read once per output/streaming process even though they share the same data.

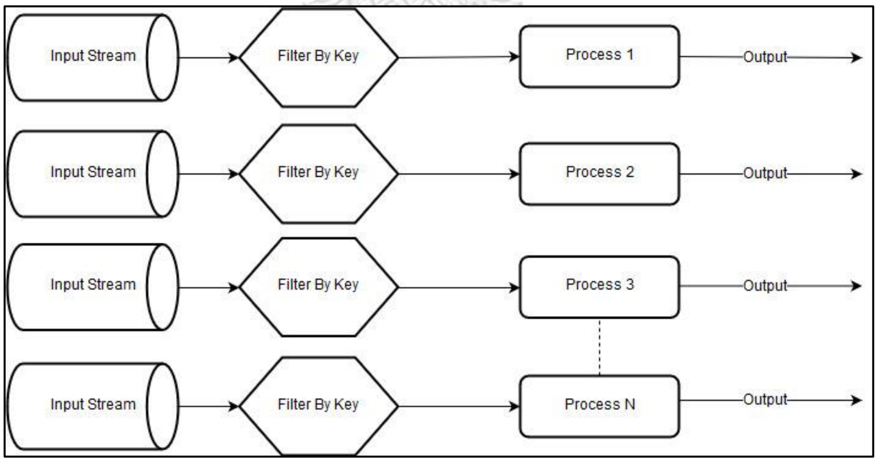


Figure 20 – Actual Workflow of Spark Streaming with Multi-Stream

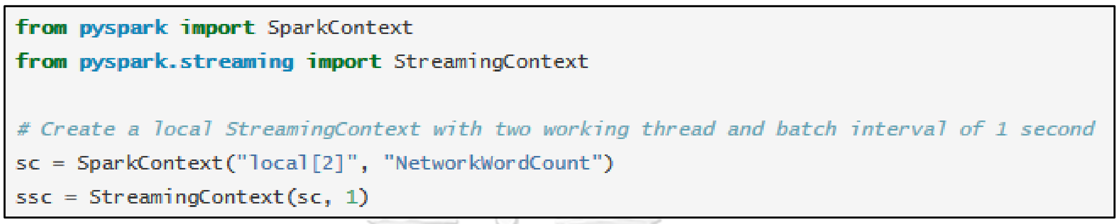
From this finding, we tried to solve the issue by having spark cache the input data immediately after it is read to avoid repeated reading. This does not solve the problem. Instead causes Spark to cache the same data multiple time.

3.3.3 Redundant Data Reading in Spark Structured Streaming

In Structured Streaming, we applied the same approach onto Spark Structured Streaming to test whether the problem persists but yielded the same results. This is because Spark Structured Streaming treats an output operation as one query, where a query consists of inputs, processing action, and an output. There is currently no way to mitigate this as it is a part of Spark's design.

3.4 Multi-Stream Applications In Spark Streaming vs Spark Structured Streaming

For Spark Streaming (D-Stream), even though we can use threading to allow multiple stream processing, only one shared micro-batch interval is possible as it is controlled by the spark context; as presented in Figure 21. In multi-stream with heterogeneous data, this is not practical as data does not always have the same period.



```python
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

# Create a local StreamingContext with two working thread and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)
```

Figure 21 - Spark Streaming context and batch interval setting
Source: https://spark.apache.org/

A workaround for this problem is to group each stream process into a windowed computation instead which can help mimic multiple batch intervals, Figure 22. This provides very limited options for micro-batch interval since only intervals that is a multiple of the micro-batch interval defined in spark-context is allowed.

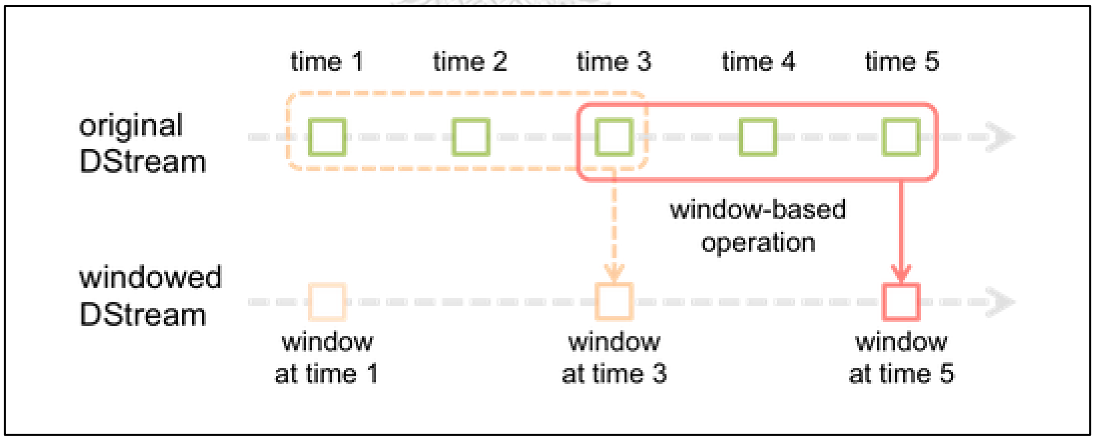
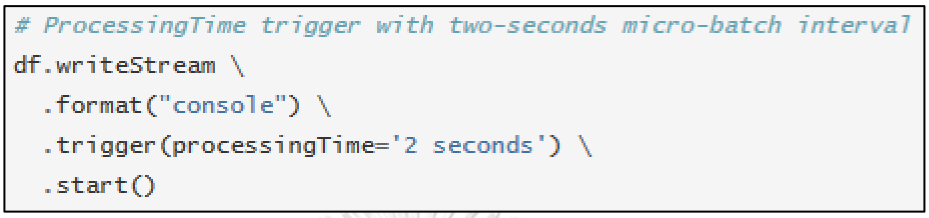


Figure 22 – Grouped Window Operation on Dstreams
Source: https://spark.apache.org/

This problem is, however, not an issue in Spark Structured Streaming as micro-batch interval is no longer defined inside spark-context but within the output of the streaming query, as illustrated in Figure 23. This allow each streaming query to define its own micro-batch interval even when sharing spark-context. We implemented our framework using Spark Structured Streaming.



```
# ProcessingTime trigger with two-seconds micro-batch interval
df.writeStream \
  .format("console") \
  .trigger(processingTime='2 seconds') \
  .start()
```

Figure 23 – Trigger Setting in Structured Streaming showing micro-batch interval setting
Source: https://spark.apache.org/

3.5 Spark Streaming Framework for Real-Time Analytics of Multiple Heterogeneous Data Streams

We proposed a Spark Streaming framework for multiple heterogeneous data streams. This framework integrates multiple data streams into one single Spark application using python’s threading package to allow simultaneous launching of multiple stream. This framework is an API operates on top of existing Spark Streaming.

Our framework is designed to address the problem that is generated from the need to deploy a large amount of smaller sized stream processing within an environment with limited resources such that streams are forced to compete or queue for resource. Stream processing that requires multiple stream processing such as real-time anomaly detection of large amount of sensor data is a great example.

3.5.1 Satisfy Streaming Function Requirements.

The requirement which streaming function need to be satisfied to work with our framework; Example shown in Figure 24. Function to be register will need to contain the following:

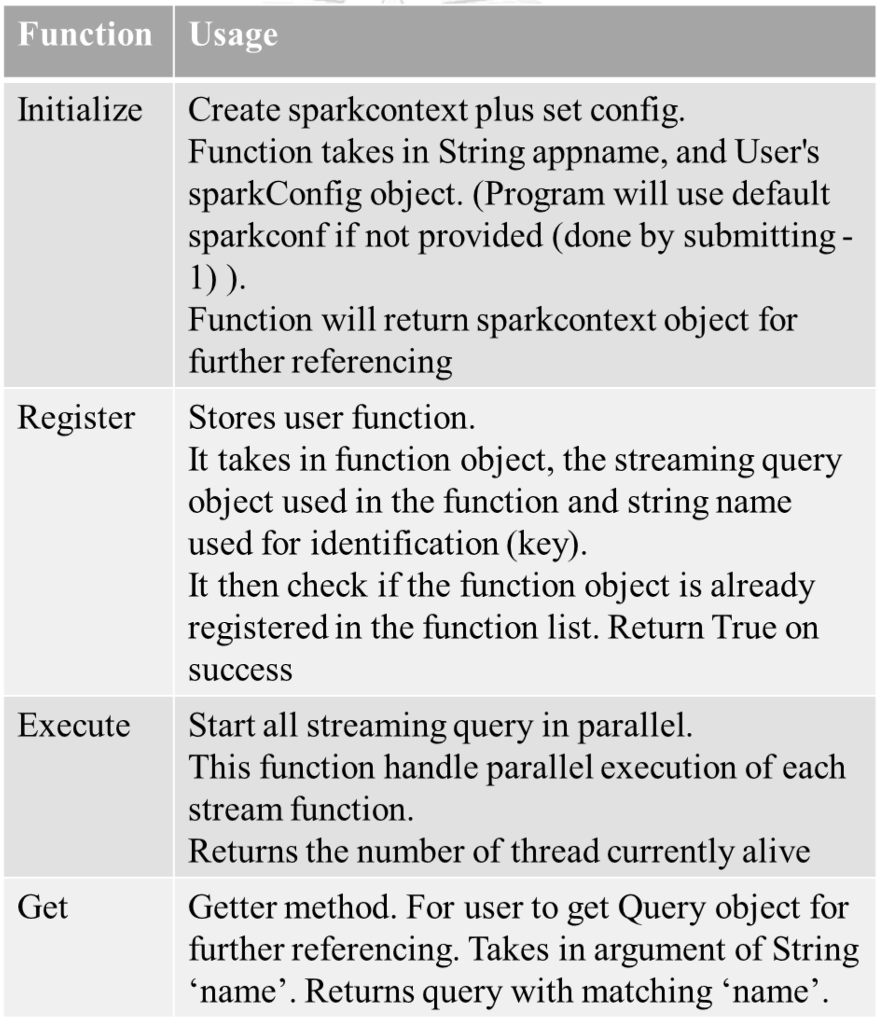
1. Data Source – clear and defined streaming input source in which further transformation/action will be applied to. The data source maybe defined either inside the function itself, or anywhere within the application where the transformation/action can access the data-frame that read the streaming data.
2. Processing command (transformation/action) – a transformation or action to transform the input stream.
3. Output – an output with defined sink, output mode, and start call.

```
// Define a DataFrame to read streaming data
data = spark.readStream.format("json").load("/in")

// Transform it to compute a result
counts = data.groupBy($"country").count()

// Write to a streaming data sink
counts.writeStream.format("parquet")
      .outputMode("complete").start("/counts")
```

Figure 24 – Example of Streaming Function
Source: https://spark.apache.org/

3.5.2 API and Functions

To combine each streaming process, our framework provides three functions; Initialize, Register, and Execute. The Initialize function setups Spark with the required configurations. These include importing framework dependency, initializing spark sessions, and configuring scheduling mode. The Register function takes in a python function object and append it to a function list which will be called and executed by the Execute function. The Execute function launches all functions registered with the Register function in parallel by calling these functions using python's Threading package. Table 1 summarize the functions of our framework. In addition, we provide a get function to allow the user to access the streaming query object for further management.



Function	Usage
Initialize	Create sparkcontext plus set config. Function takes in String appname, and User's sparkConfig object. (Program will use default sparkconf if not provided (done by submitting -1)). Function will return sparkcontext object for further referencing
Register	Stores user function. It takes in function object, the streaming query object used in the function and string name used for identification (key). It then check if the function object is already registered in the function list. Return True on success
Execute	Start all streaming query in parallel. This function handle parallel execution of each stream function. Returns the number of thread currently alive
Get	Getter method. For user to get Query object for further referencing. Takes in argument of String 'name'. Returns query with matching 'name'.

Table 1 - Framework Function Description

To use this framework, user must call Initialize, registers required function with the Register function, then start all stream processing with Execute. Please refer to the appendix for the complete code and example.

3.5.2.1 Initialize Function

The Initialize function will setup Spark with the configuration required for our framework. Inside the function, we import framework dependency, initialize spark sessions, and configure it to use FAIR scheduling mode. The user can provide their own spark config by passing the spark config object to the function. In addition, this function also fetches important information such as the amount of worker available.

3.5.2.2 Register Function

The Register function takes in a python function object, a query object, and a String name for identification. The registered function will be called and executed by the Execute function.

3.5.2.3 Execute Function

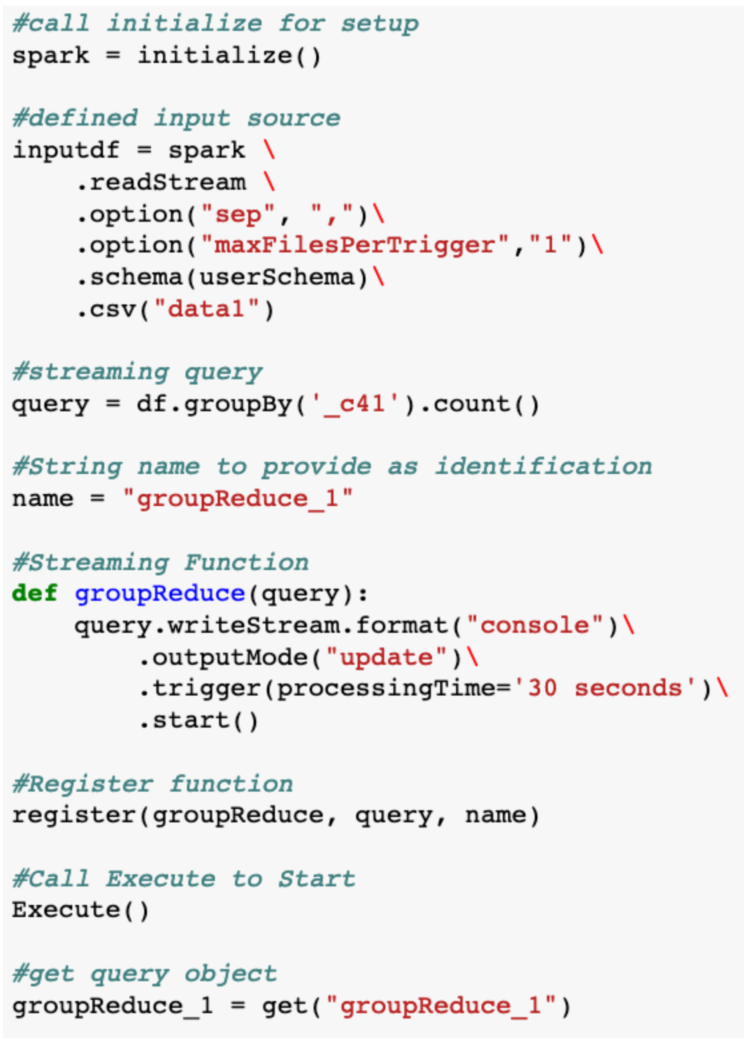
The Execute function launches all stream processing functions registered with the Register function in parallel. It will start all streaming query in the function list by calling these functions using python's Threading so that stream processes may run concurrently without blocking one another.

3.5.2.4 Get Function

The get() function can be used by the user to access the query object of a streaming queries. This can be done by providing the get() function with a name string as an argument. The function will return the query object with matching name.

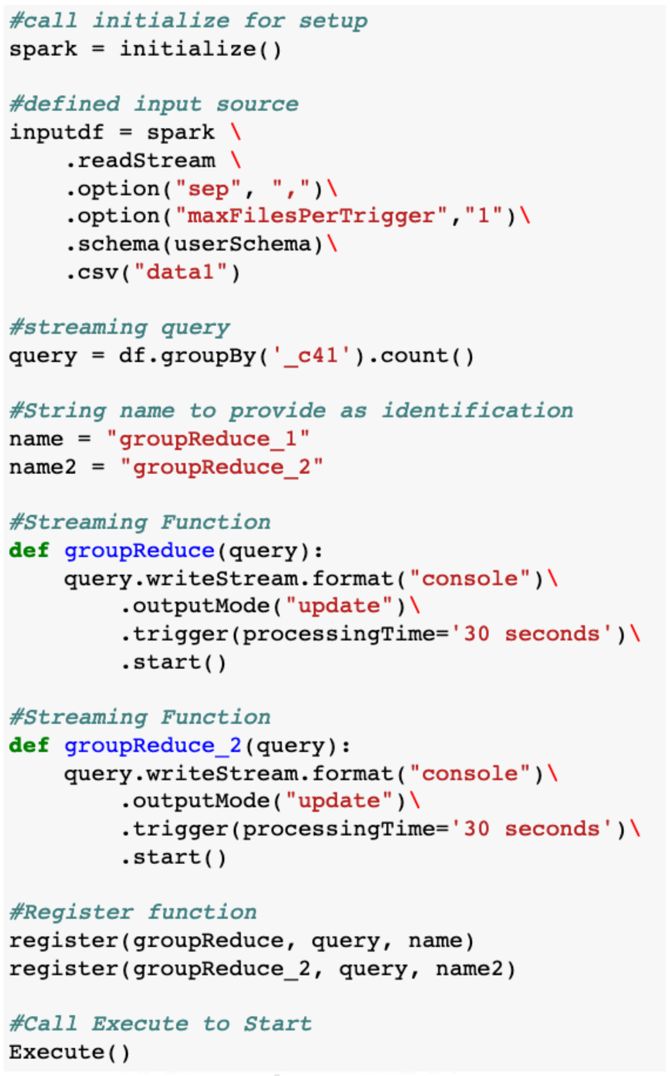
3.5.3 Framework Example

To use this framework, user must call Initialize to prepare the framework, register required function with the Register function, then start all stream processing with Execute; Figure 25 and 26 illustrate examples of how to use our framework.



```
#call initialize for setup
spark = initialize()

#defined input source
inputdf = spark \
    .readStream \
    .option("sep", ",")\
    .option("maxFilesPerTrigger","1")\
    .schema(userSchema)\
    .csv("data1")

#streaming query
query = df.groupBy('_c41').count()

#String name to provide as identification
name = "groupReduce_1"

#Streaming Function
def groupReduce(query):
    query.writeStream.format("console")\
        .outputMode("update")\
        .trigger(processingTime='30 seconds')\
        .start()

#Register function
register(groupReduce, query, name)

#Call Execute to Start
Execute()

#get query object
groupReduce_1 = get("groupReduce_1")
```

Figure 25 – Example of Framework: Single Function



```
#call initialize for setup
spark = initialize()

#defined input source
inputdf = spark \
    .readStream \
    .option("sep", ",")\
    .option("maxFilesPerTrigger","1")\
    .schema(userSchema)\
    .csv("data1")

#streaming query
query = df.groupBy('_c41').count()

#String name to provide as identification
name = "groupReduce_1"
name2 = "groupReduce_2"

#Streaming Function
def groupReduce(query):
    query.writeStream.format("console")\
        .outputMode("update")\
        .trigger(processingTime='30 seconds')\
        .start()

#Streaming Function
def groupReduce_2(query):
    query.writeStream.format("console")\
        .outputMode("update")\
        .trigger(processingTime='30 seconds')\
        .start()

#Register function
register(groupReduce, query, name)
register(groupReduce_2, query, name2)

#Call Execute to Start
Execute()
```

Figure 26 – Example of Framework: Multiple Functions

3.5.4 Advantages of Single Application for Multi-Stream

By integrating multiple streams processing into one Spark application, we can solve many issues. First, we reduce the complexity of running a Spark application, since we only need to submit only one application instead of N-application. Using more applications to achieve the same amount of works will consume more resources. Second, we simplify application monitoring process as we can now use Spark monitoring tools such as Spark web user interface. Spark’s built-in monitoring tools, Spark Web UI, only provide monitoring of single application. In case of N-application, multiple monitoring windows will be required. Third, we reduced lots of redundant codes such as initializing spark context and configuration, which is needed to be written only once. The only necessary code is one responsible for processing each data stream. Finally, integrating all stream processing processes into one application allows us to utilize Spark’s FAIR scheduling mode to solve the problem of job queuing (will be discussed later in the next chapter). Table 2 compares the difference between our framework and traditional N-application method.

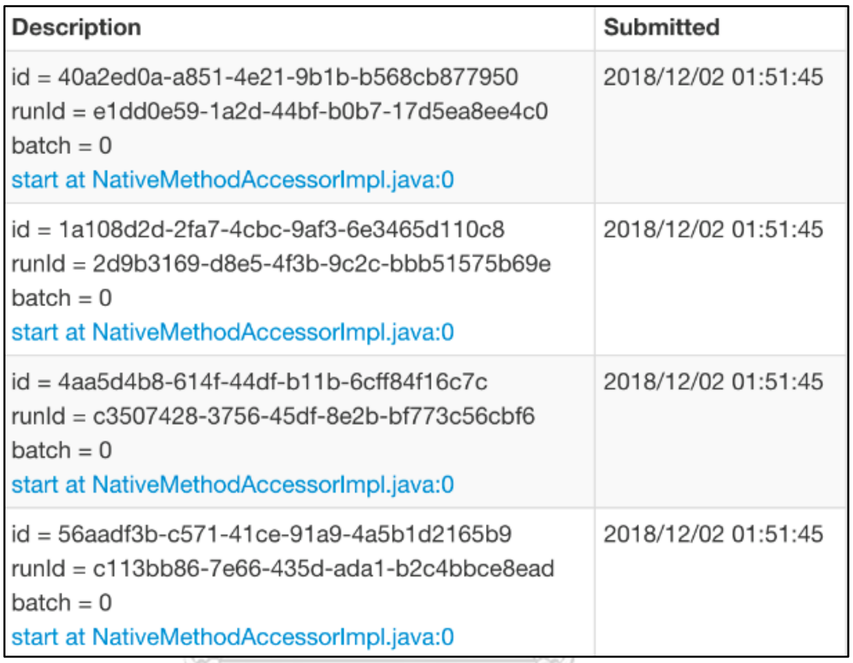
Methods	N-Application	Our Framework
Number of Spark-Submit	N-times	Only Once
Monitoring Tools	Only cluster-wide monitoring tools	Any Spark monitoring tools
Coding Redundancy	Redundant	Minimal Redundancy
Scheduling Mode	FIFO	FAIR

Table 2 - Comparison of the Difference between Our Framework and N-Application Method

Despite all advantages our single application method offer, there are also some limitations. First, our framework starts/stops all streaming queries together and provides very little accessibility in term of individual query management. Lastly, this framework is developed for a single owner operation and does not support multiple owner operations.

3.6 Framework Experimental Results

By applying our framework which make use of threading to concurrent launch spark streaming, we have enabled multi-stream processing from a single application in Spark. Figure 27 shows the result of our framework. From these results, we can clearly see that multiple job is able to start at the same time (as seen in the "submitted" column).



Description	Submitted

id = 40a2ed0a-a851-4e21-9b1b-b568cb877950
runId = e1dd0e59-1a2d-44bf-b0b7-17d5ea8ee4c0
batch = 0
start at NativeMethodAccessorImpl.java:0
	2018/12/02 01:51:45

id = 1a108d2d-2fa7-4cbc-9af3-6e3465d110c8
runId = 2d9b3169-d8e5-4f3b-9c2c-bbb51575b69e
batch = 0
start at NativeMethodAccessorImpl.java:0
	2018/12/02 01:51:45

id = 4aa5d4b8-614f-44df-b11b-6cff84f16c7c
runId = c3507428-3756-45df-8e2b-bf773c56cbf6
batch = 0
start at NativeMethodAccessorImpl.java:0
	2018/12/02 01:51:45

id = 56aadf3b-c571-41ce-91a9-4a5b1d2165b9
runId = c113bb86-7e66-435d-ada1-b2c4bbce8ead
batch = 0
start at NativeMethodAccessorImpl.java:0
	2018/12/02 01:51:45

Figure 27 – Spark Web UI showing how multi-stream can now be run in a single application

Even though we have solved the problem of deploying multi-stream in a single application. We have found that the default job scheduler Spark provide is not designed for multi-stream and scheduled job inefficiently causing multiple problems. We will address this in the next chapter.

4 Scheduling for Multi-Stream

In this chapter, we will provide explanation of how job scheduling is done in Spark, our findings of how Spark job scheduling handle multi-stream and introduce how to optimize Spark job scheduling to achieve higher performance for multi-stream.

To solve the problems of Head-of-Line blocking in multi-stream caused by Spark's default job scheduler FIFO. Our proposed framework utilizes Spark's FAIR Scheduling scheme to resolve the scheduling problem.

4.1 Job Scheduling in Spark

A Spark cluster component, as presented in Figure 28, involved in Spark's data processing includes:

- The driver program is the main program (application) and is coordinated by the SparkContext.
- The cluster manager handles job scheduling, there are many cluster managers but for this research we use Spark Standalone.
- The worker node which handles computation and stores data.

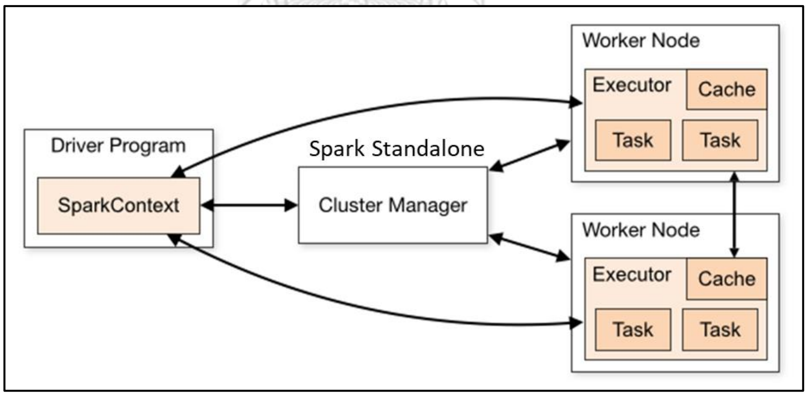


Figure 28 – Spark Cluster Component
Source: http://spark.apache.org/

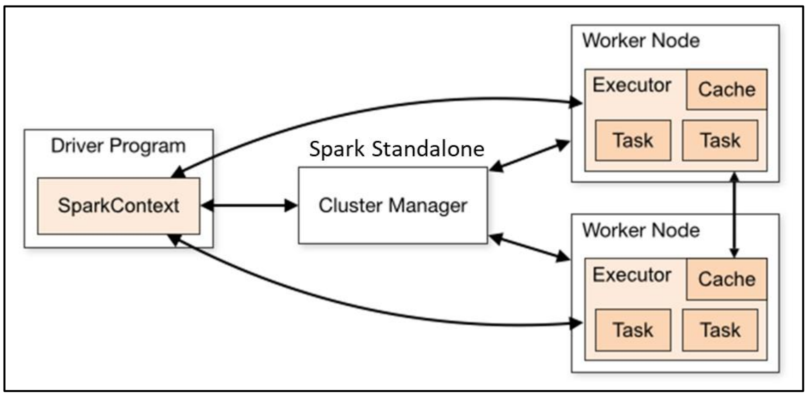


Figure 29 – Job Scheduling in Spark (Single Application, Single Job)

We illustrate how a single data processing job flows through each component in Figure 29. The driver program communicates with the cluster manager to request for worker. After the request is granted, Spark then connects to the executors on the allocated worker nodes. SparkContext then send task to each worker node to run. The worker node then returns the computation results to the driver program.

4.2 Micro-batch Behavior of Spark Streaming and Structured Streaming

In Spark Streaming, there is no long running process aside from a listener which is used to detect new data every micro-batch interval. Once new data is detected, Spark then submit a job to process the new data. Meaning that there is no long running job, but a small job every small interval instead.

Here are 3 main micro-batch behaviors:

1. If the previous micro-batch completes within the interval, then the engine will wait until the interval is over before kicking off the next micro-batch.
2. If the previous micro-batch takes longer than the interval to complete (i.e. if an interval boundary is missed), then the next micro-batch will start as soon as the previous one completes (i.e., it will not wait for the next interval boundary).
3. If no new data is available, then no micro-batch will be kicked off.

4.3 Scheduling Modes in Spark

Spark employs two job scheduling schemes. FIFO and FAIR. FIFO is the default job scheduling.

4.3.1 FIFO Scheduling

By default, Spark schedules jobs from each application in FIFO (First-in-First-Out) mode. In FIFO, when there are not enough resources, job will start to queue up one after the other waiting for resources. This can happen from two scenarios, when several jobs are submitted from the same application concurrently (single application, multi-stream, Figure 30) and when several applications are submitting jobs concurrently (multi-application, single-stream, Figure 31) Both cases are not suitable for real-time applications as it can cause head-of-line blocking problems or HoL, of which jobs with immediate deadline are queued behind a larger job with more relaxed deadline.

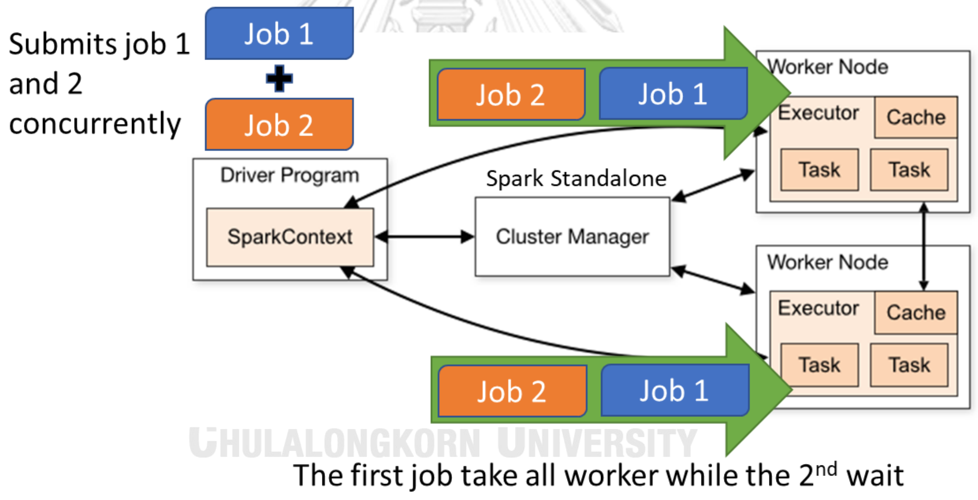


Figure 30 – FIFO Job Scheduling (Single Application, Multi-Stream)

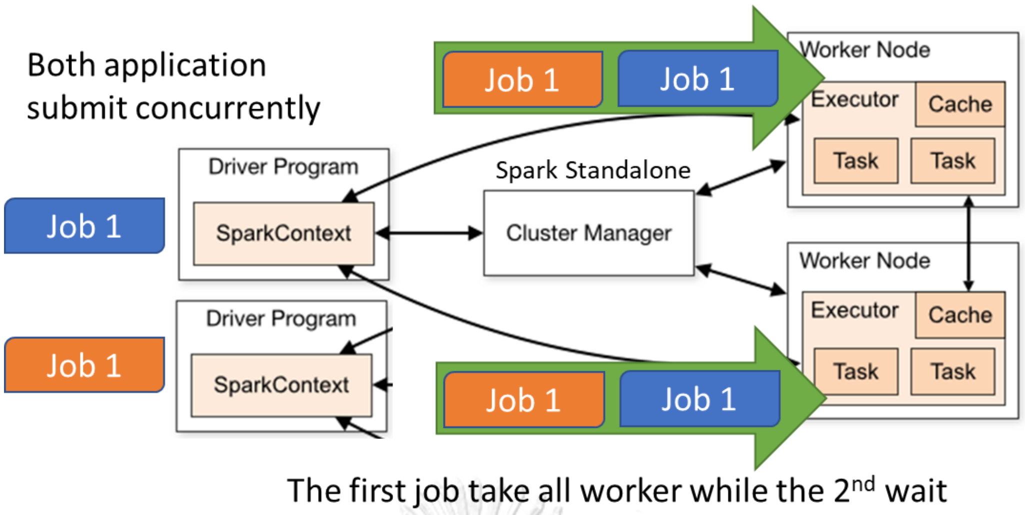


Figure 31 – FIFO Job Scheduling (Multi-Application, Single Stream)

4.3.1.1 Job Queuing issues in FIFO

In FIFO when several applications are running and there are not enough resources, job will start to queue up one after the other waiting for resources, as shown in Figure 32. This is very bad for real-time applications as it can cause serious delay, especially when a job with immediate deadline is queued behind a larger job with more relaxed deadline. Figure 33 shows blocking effect that can be identified in Spark Web UI when running multi-stream in FIFO. It shows multi-streams processing with identical micro-batch size, but different processing time caused by job queuing.

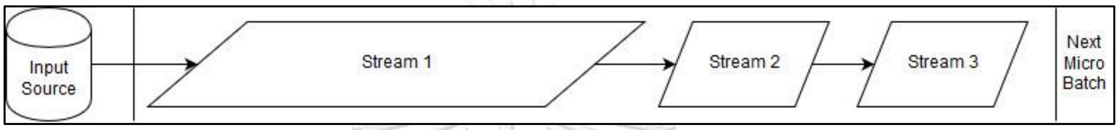
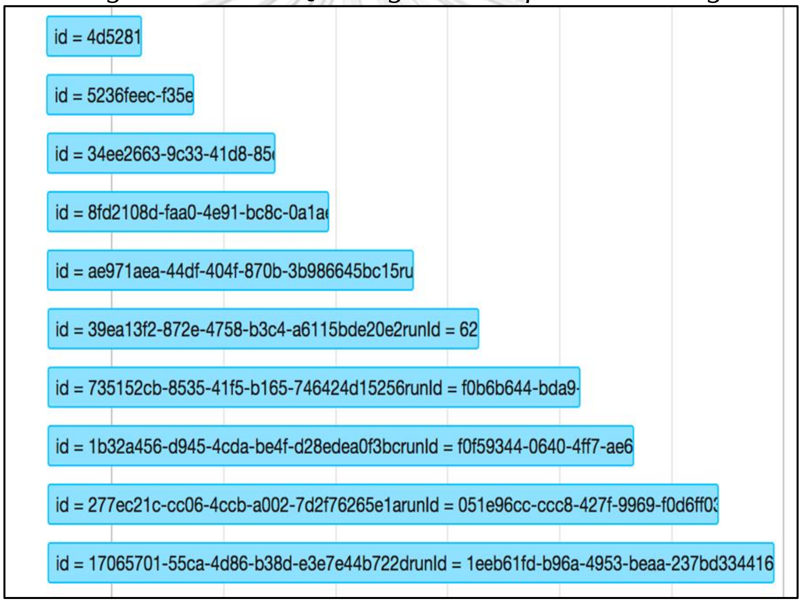


Figure 32 - Job Queuing / Head-of-Line Blocking



```
id = 4d5281
id = 5236feec-f35e
id = 34ee2663-9c33-41d8-85
id = 8fd2108d-faa0-4e91-bc8c-0a1a
id = ae971aea-44df-404f-870b-3b986645bc15ru
id = 39ea13f2-872e-4758-b3c4-a6115bde20e2runId = 62
id = 735152cb-8535-41f5-b165-746424d15256runId = f0b6b644-bda9-
id = 1b32a456-d945-4cda-be4f-d28edea0f3bcrunId = f0f59344-0640-4ff7-ae6
id = 277ec21c-cc06-4ccb-a002-7d2f76265e1arunId = 051e96cc-ccc8-427f-9969-f0d6ff0
id = 17065701-55ca-4d86-b38d-e3e7e44b722drunId = 1eeb61fd-b96a-4953-beaa-237bd334416
```

Figure 33 – Blocking Effect in Multi-Stream from Spark Web UI

4.3.2 FAIR Scheduling

In FAIR scheduling, Spark assigns resources to all jobs at the same time. This make all jobs in the queue start processing right away including job at the end of the queue. Figure 34 shows how both concurrent jobs get a worker node each and can start processing immediately as oppose to Figure 30 – FIFO Job Scheduling (Single Application, Multi-Stream) or Figure 31 – FIFO Job Scheduling (Multi-Application, Single Stream).

This, however, does not eliminate job queue, as when the number of workers is not enough for sharing across jobs, some job will be queued. As our finding suggest in the next section. Thus, multiple jobs can be processed concurrently, eliminating the waiting time in the job queue. However, FAIR is only available when running as a single application as Spark does not support a FAIR scheduler for scheduling multiple applications. Therefore, it was necessary for our framework to operate in a single application.

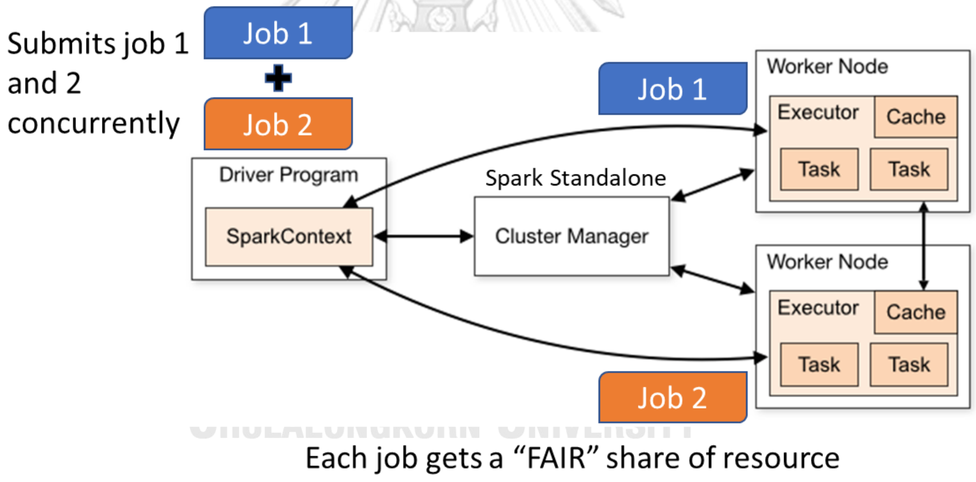


Figure 34 – FAIR Scheduling

4.3.2.1 FAIR Scheduling limitation

After experimenting with different numbers of concurrent stream. We have found that even though we are using FAIR scheduling mode, Spark will only run concurrently processes at most N number of streams, where N equals the number of core available, Figure 35 presents the case of launching 8 concurrent streams and 4 streams finished first. This imply that Spark will only process as many jobs concurrently as the number of cores available to the system. For example, if there are 8 jobs submitted concurrently but only 4 cores are available, Spark will process 4 jobs at a time.

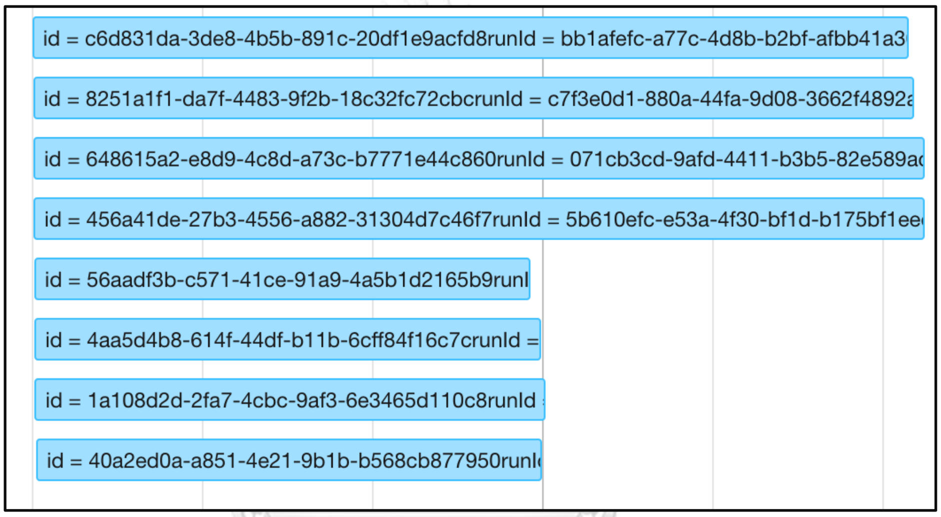

id = c6d831da-3de8-4b5b-891c-20df1e9acfd8runId = bb1afefc-a77c-4d8b-b2bf-afbb41a3
id = 8251a1f1-da7f-4483-9f2b-18c32fc72cbcrunId = c7f3e0d1-880a-44fa-9d08-3662f4892
id = 648615a2-e8d9-4c8d-a73c-b7771e44c860runId = 071cb3cd-9afd-4411-b3b5-82e589a
id = 456a41de-27b3-4556-a882-31304d7c46f7runId = 5b610efc-e53a-4f30-bf1d-b175bf1ee
id = 56aadf3b-c571-41ce-91a9-4a5b1d2165b9runI
id = 4aa5d4b8-614f-44df-b11b-6cff84f16c7crunId =
id = 1a108d2d-2fa7-4cbc-9af3-6e3465d110c8runId
id = 40a2ed0a-a851-4e21-9b1b-b568cb877950runI

Figure 35 - Behavior of 8 Concurrent Streams with FAIR

4.4 Alternatives for Solving Job Queuing in Multi-Stream

There are several methods to avoid job queueing in multi-stream environment. The following are possible solutions

- ***Increase computing power***

Increasing number of workers can reduce job queuing. This is because if the first job does not take up all available resources then the next job may start right away with any resources that are left even if Spark job scheduling is in FIFO mode. Since this requires investment in computing power, it may not be the preferred option.

- ***Resource Partitioning***

We can limit the maximum amount of resources a Spark application can use. By limiting the number of workers an application can use, allowing resources to be spared for other applications. Unfortunately, applications will hold onto all resources allocated to them for the entire duration instead of when needed.

- ***Dynamic Resource Allocation***

Dynamic resource allocation allows resource that is no longer use by its application to be freed and returned to the cluster after the application is idle. The application may request more resources later when required. This method is not fully supported on spark streaming. In stream processing, applications are long continuous processes where small jobs are submitted every small interval. This makes the idle time of each application very small causing the application to never return resources to the cluster. Therefore, dynamic resource allocation does not work for Spark Streaming.

Similarly, 3rd party cluster management like Mesos will also not work. When Spark application are communicating with Mesos on resource usages, it will always ask for resources it requires and will hold on to it. The only exception is the unlikely case where the data source is interrupted, and the data arrival interval is extended. Spark may free up resources until new data arrives, not in regular case where spark streaming idle time is usually only a few seconds.

- ***FAIR scheduling***

FAIR scheduling allows jobs to start processing immediately by sharing resources. It can effectively eliminate job queuing. This is the more ideal solution to multi-stream queuing problem.

4.5 Solving Job Queuing and Resource Scheduling in Multi-Stream

When running multi-streams, we run into scheduling problem as described in chapter 3. To solve the job queuing and resource scheduling problem of Multi-Stream, we configure Spark to use a FAIR scheduling mode. In FAIR scheduling, Spark assigns resource to jobs in a round-robin fashion. This allows multi-stream to equally share the cluster's resources. Thus, multiple jobs can be processed concurrently, eliminating the waiting time in the job queue. The performance result for using FAIR will be presented in Section 4.7.

4.6 Advantages of Using FAIR scheduler in Multi-Stream

- When a stream finishes processing early, its resources will be relocated to other streams that are currently processing immediately. In a multi-application method resource will not be relocated. This provides better resource utilization.
- No job queueing as all streaming job can start immediately.
- Head-of-Line blocking is not an issue as many streams can start processing at once.

4.7 Experiments

In this Section, we evaluate the performance of our framework while using FAIR scheduling.

4.7.1 Experimental Environment

We used a MacBook pro (2016) for this experiment. Our configurations and specification are shown in Table 3.

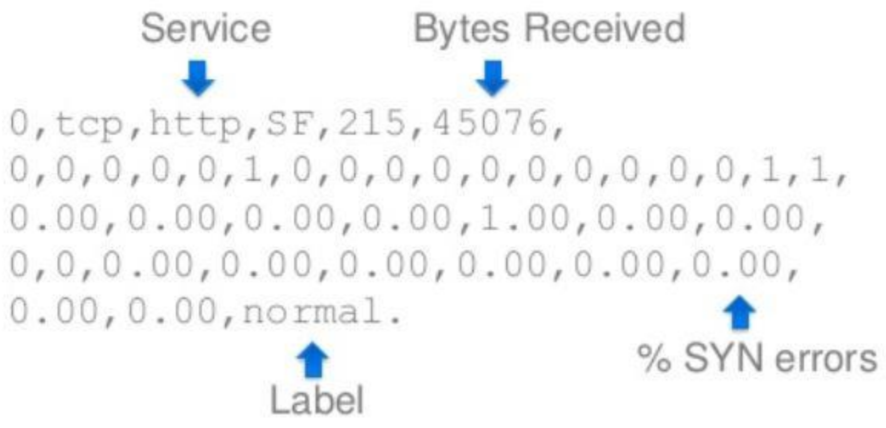
Configuration Item	Item Value
OS	MacOS High Sierra version 10.13.6
CPU	2.9 GHz Intel Core i5
Memory	8 GB 2133 MHz LPDDR3
Spark	2.3.1 (Standalone mode)

Table 3 - Test Environment and Configuration

4.7.2 Experiment Data

We use KDD1999 [12] competition data, a widely used dataset for anomaly detection of network intrusion in all experiment. The entire dataset is 753.1MB in size and contains 4.3 million records. Figure 36 shows an example of a record in KDD1999.

During the experiment, we will be using only one percent of the dataset (6.4MB of data) per micro-batch, as it is closer to the size of a streaming data. A special case where we will use the full dataset (753.1 mb) is when we are replicating a Head of Line blocking issue where we will have the first in-queue have a larger than average size (753.1mb vs 6.4mb).



```
0,tcp,http,SF,215,45076,
0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,
0.00,0.00,0.00,0.00,1.00,0.00,0.00,
0,0,0.00,0.00,0.00,0.00,0.00,0.00,
0.00,0.00,normal.
```

Figure 36 - a record in KDD1999
Source: Anomaly Detection with Apache Spark, Cloudera, Inc

4.7.3 Testing Scenario

During each micro-batch, data is queried from a local file source using Spark file-stream. then a group-by operation is done, followed by a count aggregation which then output the results to a console sink.

For this experiment, we test with 4 concurrent streams. This is because we have found that even though we have FAIR scheduling mode, Spark will only start up and concurrently process at most N number of streams; Where N is equal to the number of worker available, we feel that we should test in the scenarios where resource is a constraint as it is much harder to evaluate the performance of each scheduler mode if we have excess computing power.

Data is then recorded for both FAIR and FIFO to compare its performance.

4.7.4 Test Cases

We perform our experiment by replicating 2 cases of scheduling. Worst-case and Best-Case, Table 4. Worst-case is when a large job (753.1mb) is in front of the job queue and is causing a HoL blocking for small job (6.4mb). Best-case is when there is no significantly large job in the queue to cause HoL blocking.

Table 4 - Test Cases

Worst-Case	Best-Case
• 3x Regular Stream of normal size (6.4mb) • 1x Large Stream/Job (753.1mb) (always queued first)	• 4x Regular Stream of normal size (6.4mb) • No blocking by large head of queue.

4.7.5 Result Collection Method

All data is collected by recording from Spark Web UI. The processing time recorded is taken from the “Duration” column. This “Duration” column is the time of job submission to the time it finished processing. For each test cases we record the processing time of the regular stream that took the longest to process for that run.

We also observed Spark Web UI’s job timeline to verified if resource is being scheduled as it should according to our test cases.

4.7.6 Performance Results

To compare the results, we perform an approximate visual test [13]. The comparison of the processing time between FIFO and FAIR in the worst-case scenario is shown in Figure 37. From these results, we can clearly see that FAIR out-performed FIFO with a significant difference.

From these results we can calculate the speed up of Processing Time (FIFO) / Processing Time (FAIR) = 16.14/14.59 = 1.106 or 10.6% speed up.

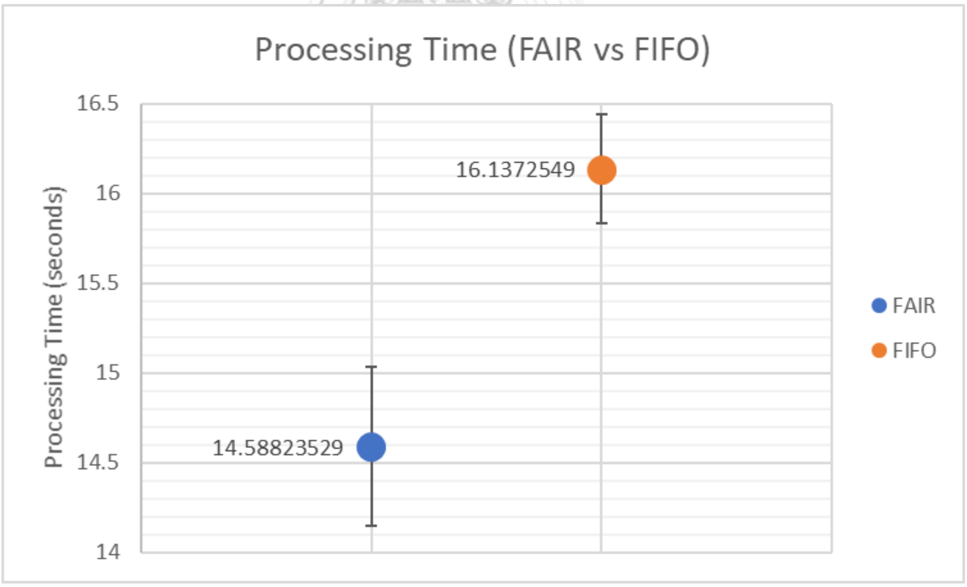


Figure 37 – Processing Time (FAIR vs FIFO) (Worst-Case)

Next, we compare the processing time of FAIR vs FIFO in the best-case scenario where each data stream is of identical size. In this scenario there is no blocking caused by a large job. Only regular queuing for resource. The result shows that the average processing time of the two have no significant difference; as shown in Figure 38. With the average time of FAIR being 53 seconds and FIFO being 52 seconds.

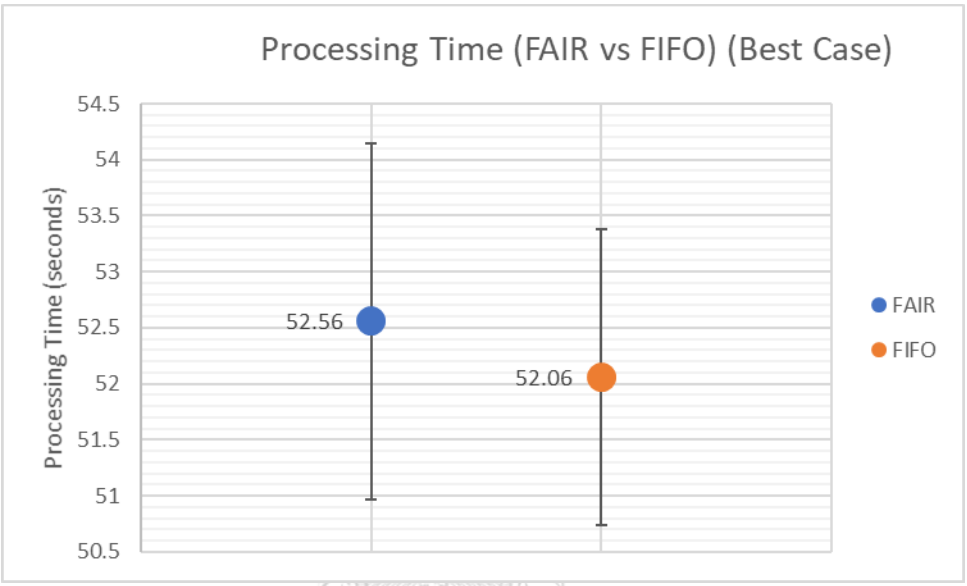


Figure 38 – Processing Time FAIR vs FIFO (best-case)

4.7.7 Result Discussion

From our experimental results, we have made several findings. First, FAIR will outperform FIFO in scenario where job queueing is presence and maintain equivalent performance otherwise. Second, there is an increase in execution time for each individual stream in FAIR. This is caused by resource sharing between each stream, resulting in the increase in individual processing time. In four multi-streams processing, the resources available is shared amongst four streams. Processing time versus the number of multi-streams is shown in Figure 39.

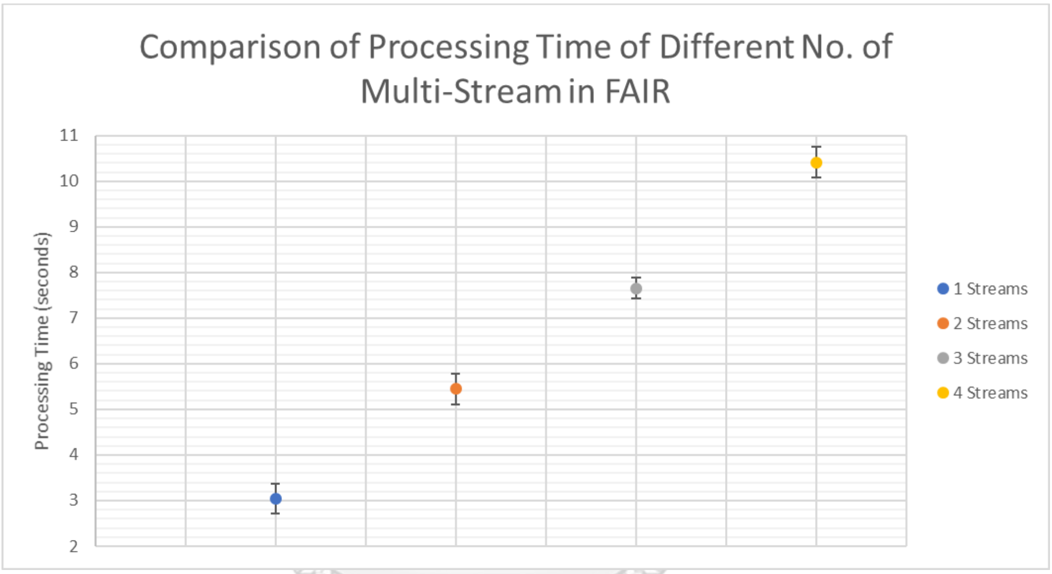


Figure 39 - Processing Time vs Number of Concurrent Streams

While the increase in individual stream execution time make FAIR scheduling seems worst, our test results shows that FIFO scheduling with job queuing have worst performance than FAIR scheduling as shown in Figure 37. Figure 40 shows the comparison of FAIR and FIFO job scheduling timeline in Spark Web UI. From the Spark Web UI, we can clearly see that streams 2-4, performed much better in FAIR where job queuing is not an issue whereas in FIFO stream 4 have missed the real-time deadline of 15s. In a real-time environment where deadline is important the solution with the better worst-case scenario is preferred. Please note that the start of block indicates that the job is submitted not when it starts processing.

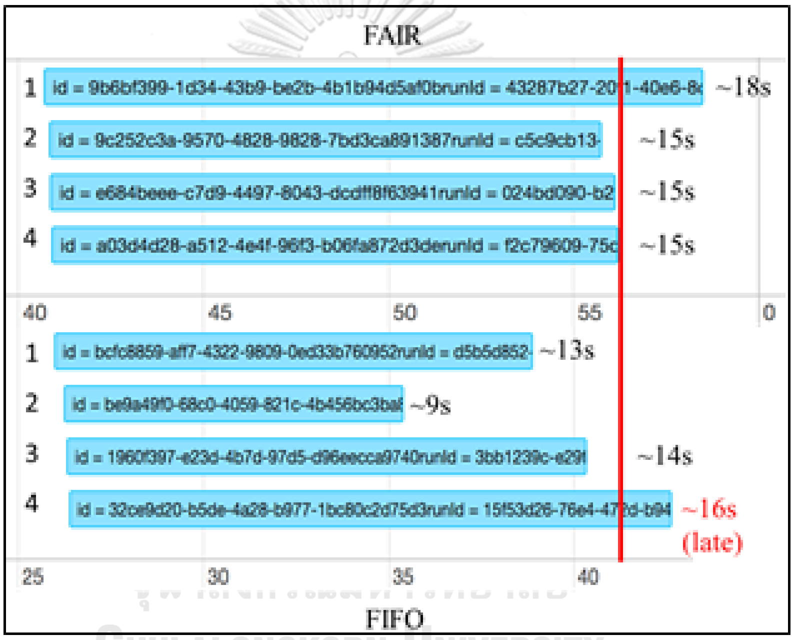

FAIR
1 id = 9b6bf399-1d34-43b9-be2b-4b1b94d5af0brunId = 43287b27-201-40e6-8
~18s
2 id = 9c252c3a-9570-4828-9828-7bd3ca891387runId = c5c9cb13-
~15s
3 id = e684beee-c7d9-4497-8043-dcdff8f63941runId = 024bd090-b2
~15s
4 id = a03d4d28-a512-4e4f-96f3-b06fa872d3derunId = f2c79609-75c
~15s
40
45
50
55
0
1 id = bcfc8859-aff7-4322-9809-0ed33b760952runId = d5b5d852-
~13s
2 id = be9a49f0-68c0-4059-821c-4b456bc3ba
~9s
3 id = 1960f397-e23d-4b7d-97d5-d96eecca9740runId = 3bb1239c-e29
~14s
4 id = 32ce9d20-b5de-4a28-b977-1bc80c2d75d3runId = 15f53d26-76e4-47d-b94
~16s
(late)
25
30
35
40
FIFO

Figure 40 - Scheduling difference of FAIR vs FIFO in Spark Web UI

This behavior can be explained in further detail with Figure 41, a graphical representation of how job is scheduled including when they start processing. Dotted red vertical line represent micro-batch interval. We will assume that our large stream takes twice as long as small stream to process. Arrows represents job that is submit and is in queue but have not started processing. Red dotted arrow represents job that are submitted later than its interval.

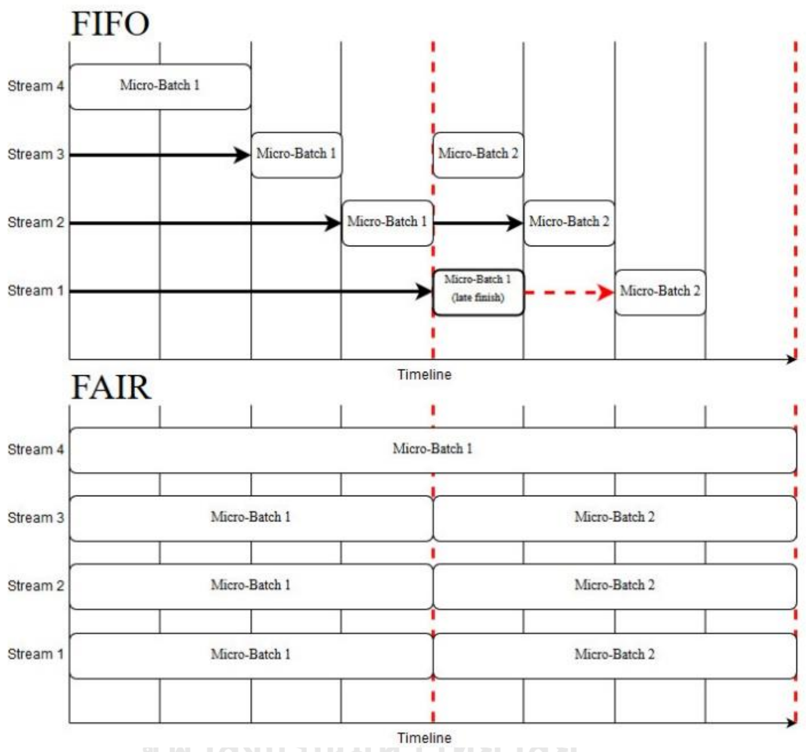


Figure 41 - Diagram of Job scheduling of FIFO vs FAIR

There are a few important behaviors of micro-batch. First, even though a micro-batch finishes processing, the next micro-batch will not start until its interval arrived. Second, if a micro-batch have not finished processing then the next micro-batch will not start until it is finished. This is very important as it means that if a delay happens it will cause delay buildup which can get out of control unless given time to recover.

5 Conclusion

In this research, we have introduced a framework for Spark Structured Streaming based on Pyspark, our framework provided solution to problems related to multiple heterogeneous data streams processing. These problems include deployment of application, monitoring difficulty, coding redundancy, and Job scheduling between data streams.

We have solved the problem of deploying large amount of streaming application by providing a framework which integrate multiple stream processing into one Spark application. With this framework, we reduced the amount of Spark-Submit required, minimize the coding redundancy in Spark application, reduce the monitoring difficulty caused by running a large amount of Spark application, and finally solves the scheduling problem of job queueing by using Spark's FAIR scheduling mode to assign each stream an equal share of resources.

Difficulties found during this research are as followed. As we implemented our framework using python programming language and coded on top of pyspark, our access to Spark's core function is very limited. We cannot extend or reimplement some functionality and most importantly Spark's pyspark API does not provide access to Spark listener, a class use to access spark's monitoring statistic.

With limited accessibility provided by pyspark API, we cannot change job scheduling pool during live operation, this stops us from being able to create a framework that can dynamically assigned resource to jobs. As we have found out later that it is possible with some method overwrite with the Scala API.

For future research direction, re-implementing our framework in Scala would provide more research opportunity by creating a dynamic job prioritization, which can be done by reassigning job to job pools of different weight to match their performance. In addition, we could also implement a function to allow new stream process to be added on the fly

6 Appendix

6.1 Framework Code

```python
# written by Tanwa Sirisakdiwan
# last updated 23/4/19

# import dependancy

import threading

from pyspark import SparkConf
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split
from pyspark.sql.functions import trim
from pyspark.sql.functions import window
from pyspark.sql.types import StructType
from pyspark.sql.streaming import DataStreamReader

########################## VARIABLE ##################################

# the number of available cores/worker, initially set to -1, value will change once application start
available_cores = -1
# spark configuration, user can provide some spark config at initialization
conf = null
# the list which will store the function object and streaming query object
# that is registered by user from register() function in dictionary form
funcList = []

########################## INITIALIZE FUNCTION ##################################

# create sparkcontext plus set config
# function takes in app name, User's sparkConfig object
# program will use default sparkconf if not provided (done by submitting -1)
# function will return sparkcontext object for further referencing
def initialize(str, SparkConf):
    # check if sparkConf is provided
    if SparkConf==-1:
        # set conf to spark default
        conf = SparkConf()
    else:
        # set conf to user's config
        conf = SparkConf
    # set FAIR scheduling mode
    conf = conf.set("spark.scheduler.mode", "FAIR")
    # start spark session
    spark = SparkSession\
        .builder\
        .appName(str)\
        .config(conf=conf)\
        .getOrCreate()
    # get number of available core
    available_cores = spark.defaultParallelism
    return spark
```

```
########################## REGISTER FUNCTION ##################################

# this function will store user function
# takes in function object, the streaming query object used in the function
# and string name used for identification
# it then check if the function object is already registered in the funclist
def register(func, query, name):
    # check if number of registered function exceed the amount of core/worker available
    if len(funcList)==available_cores:
        # warn if the number of registered function exceed the amount of core/worker available
        print("warning, no. of stream exceed available worker, some stream will be process in a queue")
    # check if function to add is a duplicate
    if name in funcList:
        print("this name already exist in the function list")
        return False
    for x in funcList.values():
        func_stored = x["function"]
        query_stored = x["query"]
        if func_sored is func or query_stored is query :
            # warn that function already exist/ is already registered
            print("this function already exist in the function list")
            return False
    # store func, query, and name as a dictionary into a list. user may access query to manage streaming queries
    dict = {'function': func , 'query': query}
    funcList[name] = dict
    # return true when sucessful
    return True

########################## EXECUTE FUNCTION #################################
# this function handle parallel execution of each stream function
# returns the number of thread currently alive
def execute():
    if len(funcList)==available_cores:
        # warn if number of registered stream exceed available worker/core, stream will be process in queue
        print("warning, no. of stream exceed available worker, some stream will be process in a queue")
    for i in range(0,len(funcList)):
        function = funcList[i]["function"]
        query = funcList[i]["query"]
        process = threading.Thread(target=function, args=query)
        process.start()
        process.join()
    return threading.enumerate()

########################## GET FUNCTION ################################

# this function return the query object with matching key 'name' to the user
# takes in a string 'name' as argument
# returns the query object
def get(name):
    if name in funcList:
        return funcList[name]["query"]
    print("query not found")
    return null
```

REFERENCES

[1] L. Wang, "Heterogeneous Data and Big Data Analytics," *Automatic Control and Information Sciences*, vol. 3, no. 1, pp. 8–15, Oct. 2017.

[2] "Apache Spark^{TM} - Unified Analytics Engine for Big Data." [Online]. Available: https://spark.apache.org/. [Accessed: 13-Nov-2018].

[3] M. Marjani *et al.*, "Big IoT Data Analytics: Architecture, Opportunities, and Open Research Challenges," *IEEE Access*, vol. 5, pp. 5247–5261, 2017.

[4] M. Zaharia *et al.*, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," p. 14.

[5] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, Farminton, Pennsylvania, 2013, pp. 423–438.

[6] M. Armbrust *et al.*, "Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark," in *Proceedings of the 2018 International Conference on Management of Data*, New York, NY, USA, 2018, pp. 601–613.

[7] P. Tangsatjatham and N. Nupairoj, "Hybrid big data architecture for high-speed log anomaly detection," in *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, 2016, pp. 1–6.

[8] D. Jankov, S. Sikdar, R. Mukherjee, K. Teymourian, and C. Jermaine, "Real-time High Performance Anomaly Detection over Data Streams: Grand Challenge," 2017, pp. 292–297.

[9] D. Cheng, Y. Chen, X. Zhou, D. Gmach, and D. Milojicic, "Adaptive scheduling of parallel jobs in spark streaming," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, 2017, pp. 1–9.

[10] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 Requirements of Real-time Stream Processing," *SIGMOD Rec.*, vol. 34, no. 4, pp. 42–47, Dec. 2005.

[11] "Ganglia Monitoring System." [Online]. Available: http://ganglia.info/. [Accessed: 13-Nov-2018].

[12] "KDD Cup 1999 Data." [Online]. Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html. [Accessed: 13-Nov-2018].

[13] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques For Experimental Design, Measurement, Simulation, and Modeling, NY: Wiley*. 1991.

VITA

NAME	Tanwa Sirisakdiwan
DATE OF BIRTH	8 December 1995
PLACE OF BIRTH	Bangkok, Thailand
INSTITUTIONS ATTENDED	Chulalongkorn University
PUBLICATION	2019 2nd International Conference on Communication Engineering and Technology (ICCET 2019)