NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

GRADUATE PROGRAM
COMPUTING SYSTEMS: SOFTWARE AND HARDWARE

MASTER THESIS

Distributed reservoir sampling algorithms for data pre-processing with use of Kafka Streams

Kostis I. Gerakos

Supervisor: **Hadjieftymiades Stathes,** Professor

ATHENS

NOVEMBER 2018

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ: ΛΟΓΙΣΜΙΚΟ ΚΑΙ ΥΛΙΚΟ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κατανεμημένοι αλγόριθμοι αποθέματος για προεπεξεργασία δεδομένων με Kafka Streams

Κωστής Η. Γεράκος

Επιβλέπων: **Ευστάθιος Χατζηευθυμιάδης,** Καθηγητής

ΑΘΗΝΑ

ΝΟΕΜΒΡΙΟΣ 2018

MASTER THESIS

Distributed reservoir sampling algorithms for data pre-processing with use of Kafka Streams

Kostis I. Gerakos
A.M.: M1428

SUPERVISOR : **Hadjieftymiades Stathes,** Professor

EXAMINATION COMMITTEE : **Nancy Alonistioti,** Associate Professor

November 2018

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κατανεμημένοι αλγόριθμοι αποθέματος για προ-επεξεργασία δεδομένων με Kafka Streams

Κωστής Η. Γεράκος
Α.Μ.: Μ1428

ΕΠΙΒΛΕΠΩΝ : **Ευστάθιος Χατζηευθυμιάδης,** Καθηγητής

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Αθανασία Αλωνιστιώτη,** Επίκουρος Καθηγητής

Νοέμβριος 2018

ABSTRACT

With the rapid growth of the Internet of Things (IoT) and with the number of devices expected to connect to it estimated to exceed 30 billion by 2020 and the consequent increase in data transmitted, it is necessary for big data processing systems to use efficient algorithms in combination with programming libraries that are widely used in the industry. This master thesis aims to analyze and present reservoir sampling algorithms as well as to develop them using the Kafka Streams API in order to solve the problem of their distribution. By taking advantage of the API and the algorithm specific characteristics, we aim to implement a tool that helps analysts and experimenters on the IoT field to preprocess data and quickly obtain results from a continuous data stream.

SUBJECT AREA: Stream Processing

KEYWORDS: Apache Kafka, Kafka Streams, reservoir sampling, IoT

ΠΕΡΙΛΗΨΗ

Με την ανάπτυξη του IoT και με τον αριθμό των συσκευών που αναμένεται να συνδεθούν σε αυτό να ξεπερνάει τα 30 δισεκατομμύρια μέχρι το 2020 καθώς και με την συνεπακόλουθη αύξηση στα δεδομένα που μεταδίδονται κρίνεται αναγκαίο από τα σύγχρονα συστήματα επεξεργασίας δεδομένων μεγάλης κλίμακας να χρησιμοποιούν αποδοτικούς αλγορίθμους σε συνδυασμό με προγραμματιστικές βιβλιοθήκες που χρησιμοποιούνται ευρέως στον τομέα της βιομηχανίας. Σκοπός της διπλωματικής εργασίας είναι η ανάλυση και παρουσίαση αλγορίθμων αποθέματος καθώς και η ανάπτυξη τους με την χρήση της βιβλιοθήκης Kafka Streams με σκοπό την επίλυση του προβλήματος της κατανομής τους. Αξιοποιώντας τις ιδιαιτερότητες της βιβλιοθήκης και των αλγορίθμων στοχεύουμε στην υλοποίηση ενός εργαλείου που βοηθάει αναλυτές και πειραματιστές στο τομέα του IoT στην προεπεξεργασία των δεδομένων και την ταχεία λήψη αποτελεσμάτων από μια συνεχόμενη ροή δεδομένων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Επεξεργασία Δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Apache Kafka, Kafka Streams, reservoir sampling, IoT

CONTENTS

1. INTRODUCTION 10

2. DATA STREAMS AND STREAM PROCESSING 11

2.1 Batch Processing 11

2.2 Stream Processing 13

2.2.1 Event Stream Processing 13

2.2.2 Complex Event Processing 14

3. KAFKA STREAMS 16

3.1 Apache Kafka High Overview 16

3.2 Topics 16

3.3 Partitions 17

3.4 Replication 17

3.5 Producers 18

3.6 Consumers 19

3.7 Kafka Streams 20

4. SENSOR FUSION ENGINE 22

4.1 Contextors and network module 22

4.2 Big data preprocessing 24

5. RESERVOIR SAMPLING ALGORITHMS 26

5.1 General Reservoir Sampling Algorithm 26

5.2 Efficient Reservoir Sampling Algorithm 27

5.3 Weighted Reservoir Sampling Algorithm 28

6. IMPLEMENTING DISTRIBUTED RESERVOIR SAMPLING WITH KAFKA STREAMS 29

7. EXPERIMENTAL RESULTS 31

8. CONCLUSIONS 33

ABBREVIATIONS – ACRONYMS.. 34

REFERENCES 35

LIST OF FIGURES

Figure 1: Batch Processing Systems 12

Figure 2: MapReduce jobs 12

Figure 3: Publish/Subscribe systems 16

Figure 4: Partitions in a topic 17

Figure 5: Kafka replication 18

Figure 6: Overview of producers 19

Figure 7: Consumer group 20

Figure 8: Kafka Streams API overview 21

Figure 9: Graphical depiction of a contextor 22

Figure 10: SFE architecture 23

Figure 11: Network Module of Fusion Module 24

Figure 12: Data acquisition 25

1. INTRODUCTION

This master thesis provides an explanation and description of the architectural specifications obtained during the development of a data pre-processing machine based on the reservoir sampling family of algorithms and using the Kafka Streams library. The structure of the work is as follows. Chapter 2 will present the general concept of data processing as well as its sub-concepts. In Chapter 3 we will look at the Kafka Streams library and we will talk about Kafka. Chapter 4 will present the general architecture on which the algorithms will be integrated. Chapter 5 will describe the reservoir sampling algorithms used. Chapter 6 will explain why the use of Kafka Stream is ideal for the implementation of algorithms, and in Chapter 7 some experimental results will be given of tests made on benchmarking different implementations of the algorithms.

2. DATA STREAMS AND STREAM PROCESSING

In recent years, Internet of Things (IoT) has been developed and has become a field of research, experimentation and study. Statistical studies have shown that 30 billion devices worldwide will be connected to IoT by 2020. Scholars expect that by that date autonomous vehicles will be on the streets with each one using dozens or maybe hundreds of devices and sensors to transmit information in order for these vehicles for safe transportation. [1] [2]

With this rapid expansion of IoT the amounts of information transmitted have also been increased, making IoT an attractive field of interest for big data researchers. But with this vast amount of, in many cases, identical data and values the question that remains is what amount of this information is crucial and needs to be stored, processed and further analyzed.

To answer this question, several subfields of big data technologies have been developed. The most famous are batch processing and stream processing. The sections below will describe what each one is and what their differences are.

2.1 Batch Processing

The term of batch processing was invented to describe jobs that can run without end user interaction. These were called batch jobs and could be scheduled to run as resources permit. Batch processing is for those frequently used programs that can be executed with minimal human interaction. An example of a batch processing job could be reading all the sale logs from an online shop for a single day and aggregating it into statistics for that day (number of users per country, the average spent amount, etc.). Doing this as a daily job could give insights into customer trends.

Historically the term batch job originated in the days when punched cards contained the directions for a computer to follow when running one or more programs. Multiple card decks representing multiple jobs would often be stacked on top of one another in the hopper of a card reader, and be run in batches. This practice goes back to 1890 when Herman Hollerith created punch cards to process census data. Working for the U.S. Census Bureau, he developed a system by which a card that he punched manually was read by an electromechanical device.

Batch jobs are typically executed at a scheduled time or on an as-needed basis. Perhaps the closest comparison is with processes run by a CRON command in UNIX, although the differences are significant. Modern Batch Processing uses exception-based management alerts to create events. However the volume of data is often too big for a single server to process. Therefore, there was a need to develop code that runs on multiple nodes therefore batch processing was combined with MapReduce. [3]

MapReduce is a framework that allows a programmer to write code that is executed on multiple nodes without having to worry about fault tolerance, reliability, synchronization or availability. In order to decrease the duration of distributed computation, MapReduce tries to reduce shuffling (moving) the data from one node to another by distributing the computation so that it is done on the same node where the data is stored. This way, the data stays on the same node, but the code is moved via the network. This is ideal because the code is much smaller than the data. To run a MapReduce job, the user has to implement two functions, map and reduce, and those implemented functions are distributed to nodes that contain the data by the MapReduce framework. Each node

runs (executes) the given functions on the data it has in order the minimize network traffic. [4]

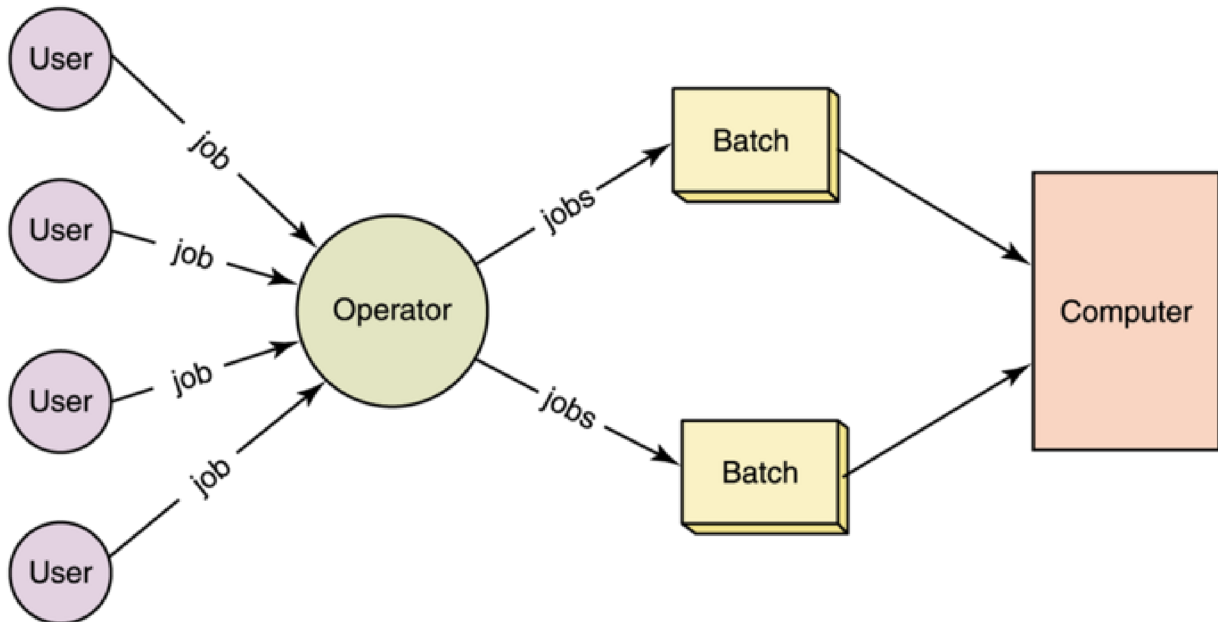

Figure 1: Batch Processing Systems

The most known batch processing system is Hadoop which is also the first open-source implementation of MapReduce. It also has its own distributed file storage called HDFS. In Hadoop, the typical input into a MapReduce job is a directory in HDFS. In order to increase parallelization, each directory is made up of smaller units called partitions and each partition can be processed separately by a map task. The map task is called once for every input partition and its job is to extract key-value pairs from the input partition and can generate any number of key-value pairs from a single input. Then the MapReduce framework collects all the key-value pairs produced by the map tasks and arranges them into groups with the same key and applies the reduce function. All the grouped values entering the reducers are sorted by the framework. [5]

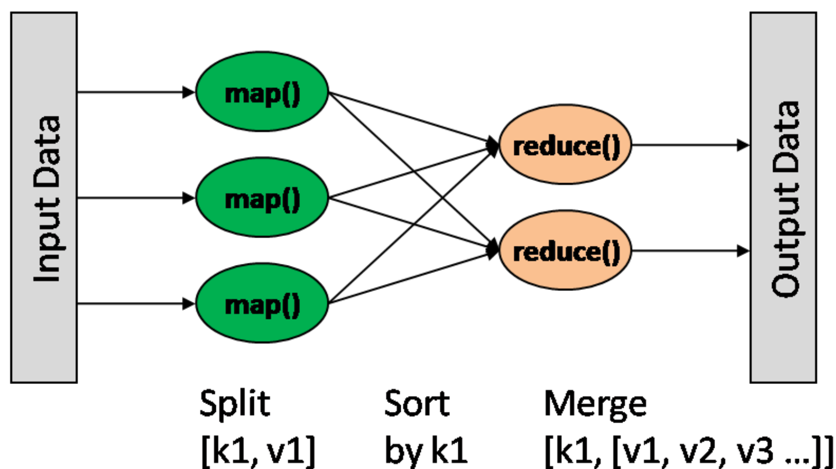

Figure 2: MapReduce jobs

2.2 Stream Processing

While batch processing works as generally a background job in already saved data sometimes coming from an input stream, stream processing came to provide real time analysis. Stream processing is the processing of data in motion, or in other words, computing on data directly as it is produced or received. The majority of data are born as continuous streams such as sensor events, user activity on a website, financial trades are created as a series of events over time.

Before stream processing, this data was often stored in a database, a file system, or other forms of mass storage. Applications would query the data or compute over the data as needed. Stream Processing turns this paradigm around. The application logic, analytics, and queries exist continuously, and data flows through them continuously. [6]

Upon receiving an event from the stream, a stream processing application reacts to that event: it may trigger an action, update an aggregate or other statistic, or “remember” that event for future reference. Streaming computations can also process multiple data streams jointly, and each computation over the event data stream may produce other event data streams. The systems that receive and send the data streams and execute the application or analytics logic are called stream processors. The basic responsibilities of a stream processor are to ensure that data flows efficiently and the computation scales and is fault tolerant. [7]

Some data naturally comes as a never-ending stream of events. To perform batch processing, we need to store, stop data collection at some time and process the data. Then we have to do the next batch and then worry about aggregating across multiple batches. In contrast, streaming handles never ending data streams gracefully and naturally. It can detect patterns, inspect results, look at multiple levels of focus, and easily look at data from multiple streams simultaneously.

Stream processing naturally fits with time series data and detecting patterns over time. For example, if you are trying to detect the length of a web session in a never-ending stream (this is an example of trying to detect a sequence), it is very hard to do it with batches as some session will fall into two batches. Stream processing can handle this easily. If you take a step back and consider, the most continuous data series are time series data. For example, almost all IoT data are time series data. Hence, it makes sense to use a programming model that fits naturally. [8]

Batch lets the data build up and try to process them at once while stream processing processes data as they come in, hence spread the processing over time. Because of this stream processing can work with a lot less hardware than batch processing. Furthermore, stream processing also enables approximate query processing via systematic load shedding. So stream processing fits naturally into use cases where approximate answers are sufficient. Sometimes the amount of data is huge and it is not even possible to store it. Stream processing lets you handle large firehose style data and retain only useful bits. [9]

Finally, there is a lot of streaming data available (e.g. customer transactions, activities, website visits) that will grow faster with IoT use cases (all kind of sensors). Streaming is a much more natural model to think about and program those use cases.

2.2.1 Event Stream Processing

In order to have solid understanding of event stream processing, we need to break it down into its simplest terms: event, stream and processing. An event is anything that happens at a clearly defined time and that can be specifically recorded. Not to be

confused, an event object is any type of object that represents or records an event, typically for the purpose of computer processing. Event objects usually include data about the type of activity, when the activity occurred, as well as its location and cause. An event stream is a constant and continuous flow of event objects that navigate into and around companies from thousands of connected devices, IoT, and any other sensors. An event stream is a sequence of events ordered by time. Enterprises typically have three different kinds of event streams: business transactions like customer orders, bank deposits, and invoices; information reports like social media updates, market data, and weather reports; and IoT data like GPS-based location information, signals from SCADA systems, and temperature from sensors. Processing is the final act of analyzing all of this data. Putting all three of these together event stream processing is the process of being able to quickly analyze data streaming from one device to another at an almost instantaneous rate after it's created. The ultimate goal of ESP deals with identifying meaningful patterns or relationships within all of these streams in order to detect things like event correlation, causality, or timing. [10] [11]

2.2.2 Complex Event Processing

Complex Event Processing (CEP) was developed to analyze event-driven simulations of distributed system architectures. CEP is a subset of Event Stream Processing. However, Stream processing engines and CEP engines are pretty different and they come from different background. Also the use cases they target and issues they choose to handle or not handle are different. Stream processing engines create a processing graph, and inject event into this processing graph. Each operator process and send events to next processors. In most stream processing engines an user writes code to create the operators, wire them up in a graph and run them. Then the engine runs the graph in parallel using many computers. In contrast, CEP engines let users write queries using an higher level language such as an SQL like query language and has build in operators such as time windows, temporal event sequences etc. CEP analysis is performed by finding patterns in events to determine whether a more complex event has occurred. A complex event is an event that summarizes or represents a set of other events. Events may be correlated over multiple dimensions, such as causal, temporal or spatial. CEP can take place over o longer period lime compared to the other types of event processing. [12]

Generally CEP has the following characteristics:

- Stream Processing Engines tend to be distributed and parallel natively as oppose to CEP engines tends to be more centralized.
- With most Stream Processing Engines it is necessary to write code. Also it forces the implementation of higher order operators like Windows, Temporal Patterns, and Joins while CEP engines support them natively. CEP engines often have a SQL like query language.
- CEP engines are tuned for low latency. Often they respond within few milliseconds and sometimes sub milliseconds. In contrast, most Stream processing engines takes close to a second to generate results.
- Stream processing engines focus on reliable message processing while CEP engines have often opt to throw away some events when failure happens and continue.

CEP is generally more suitable to handle IoT produced data due to three main reasons.

1. IoT data are time series data where data is auto correlated. CEP is much better placed to handle them.
2. Most IoT use cases deal with real life.
3. Most IoT use cases are complex, and they go beyond calculating aggregating data. Those use cases need support for complex operators like time windows and temporal query patterns.

3. KAFKA STREAMS

Apache Kafka was originally developed by LinkedIn, and was subsequently open sourced in early 2011. Its initial purpose was to be a fully scalable, distributed and fast log message bus. In recent years Apache Software Foundation and Confluent, a startup created by the original programmers of Apache Kafka in LinkedIn, transformed it to be a fully distributed streaming platform by developing Kafka Streams and Kafka Connector APIs. On the following section the architecture of Apache Kafka will be presented in terms of how it can achieve high throughput and scalability.

3.1 Apache Kafka High Overview

Apache Kafka is a publish/subscribe messaging system designed to be a distributed commit log. Publish/subscribe messaging is a pattern that is characterized by the sender (publisher) of a piece of data (message) not specifically directing it to a receiver. Instead, the publisher classifies the message somehow, and that receiver (subscriber) subscribes to receive certain classes of data. Publish/subscribe systems often have a broker, a central point where data are published.

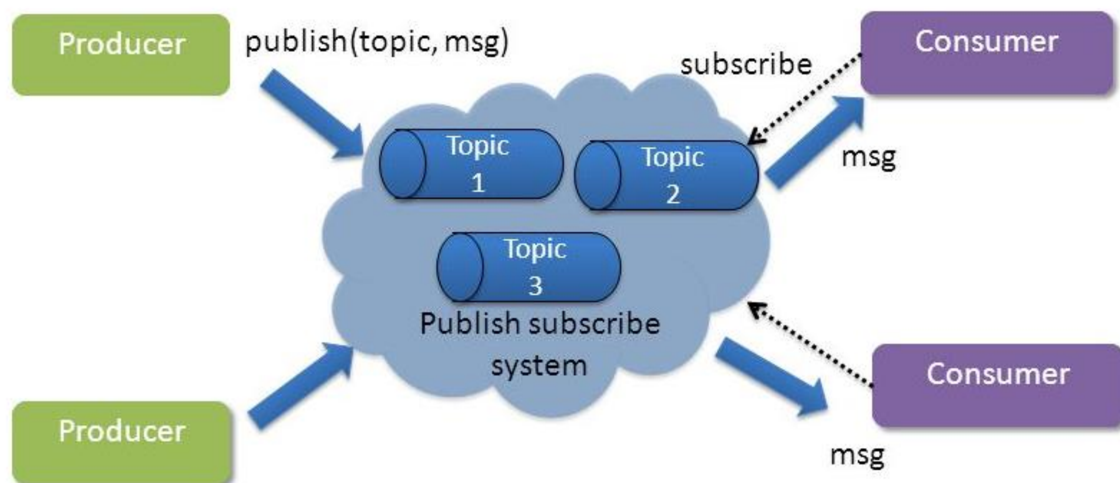

Figure 3: Publish/Subscribe systems

The unit of data within Kafka is called a message. A message is simply an array of bytes as far as Kafka is concerned, so the data contained within it does not have a specific format or meaning to Kafka. A message can have an optional bit of metadata, which is referred to as a key which is also a byte array and no specific meaning to Kafka. Messages within Kafka are stored durably, in order, and can be read deterministically. In addition, messages can be distributed within the system to provide additional protections against failures, as well as significant opportunities for scaling performance. For efficiency, messages are written into Kafka in batches. A batch is just a collection of messages, all of which are being produced to the same topic and partition. An individual roundtrip across the network for each message would result in excessive over- head, and collecting messages together into a batch reduces this. [13]

3.2 Topics

Topics are virtual groups of one or many partitions across Kafka brokers in a Kafka cluster. A Kafka topic is unique across a Kafka cluster and it is there that producers

publish messages and consumers pull messages. Kafka brokers stores messages in a partition in an ordered fashion by appending one message after another and creating a log file. Producers write messages to the tail of these logs that consumers read at their own pace. Kafka scales topic consumption by distributing partitions among a consumer group, which is a set of consumers sharing a common group identifier.

3.3 Partitions

Kafka topics are divided into a number of partitions. Partitions allow you to parallelize a topic by splitting the data in a particular topic across multiple brokers. Each partition can be placed on a separate machine to allow for multiple consumers to read from a topic in parallel. Consumers can also be parallelized so that multiple consumers can read from multiple partitions in a topic allowing for very high message processing throughput.

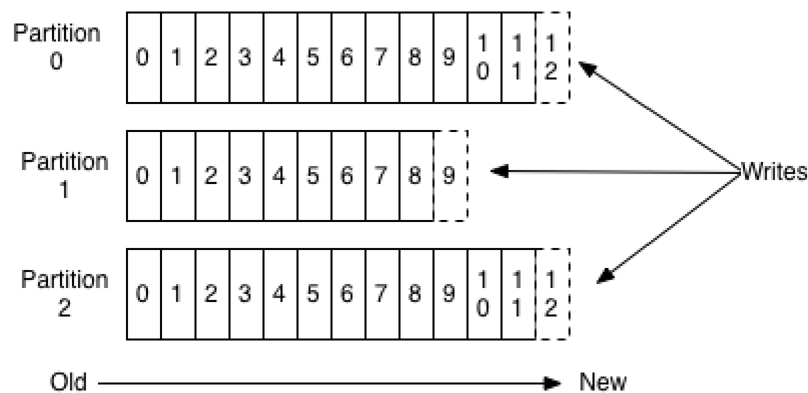

Figure 4: Partitions in a topic

Going back to the “commit log” description, a partition is a single log. Messages are written to it in an append-only fashion, and are read in order from beginning to end. Note that as a topic typically has multiple partitions, there is no guarantee of message time-ordering across the entire topic, just within a single partition. Partitions are also the way that Kafka provides redundancy and scalability. Each partition can be hosted on a different server, which means that a single topic can be scaled horizontally across multiple servers to provide performance far beyond the ability of a single server.

Each message within a partition has an identifier called its offset. The offset is the ordering of messages as an immutable sequence. Consumers can read messages starting from a specific offset and are allowed to read from any offset point they choose, allowing consumers to join the cluster at any point in time they see fit. Given these constraints, each specific message in a Kafka cluster can be uniquely identified by a tuple consisting of the message’s topic, partition, and offset within the partition.

3.4 Replication

Every partition in a Kafka topic has a write-ahead log where the messages are stored and every message has a unique offset that identifies its position in the partition’s log. A partition may be assigned to multiple brokers, which will result in the partition being replicated. This provides redundancy of messages in the partition, such that another broker can take over leadership if there is a broker failure. Topic partitions in Kafka are replicated n times, where n is the replication factor of the topic. This allows Kafka to automatically failover to these replicas when a server in the cluster fails so that messages remain available in the presence of failures. Replication in Kafka happens at the partition granularity where the partition’s write-ahead log is replicated in order to n

servers. Out of the n replicas, one replica is designated as the leader while others are followers. As the name suggests, the leader takes the writes from the producer and the followers merely copy the leader's log in order.

The fundamental guarantee a log replication algorithm must provide is that if it tells the client a message is committed, and the leader fails, the newly elected leader must also have that message. Kafka gives this guarantee by requiring the leader to be elected from a subset of replicas that are "in sync" with the previous leader or, in other words, caught up to the leader's log. The leader for every partition tracks this in-sync replica list by computing the lag of every replica from itself. When a producer sends a message to the broker, it is written by the leader and replicated to all the partition's replicas. A message is committed only after it has been successfully copied to all the in-sync replicas. Since the message replication latency is capped by the slowest in-sync replica, it is important to quickly detect slow replicas and remove them from the in-sync replica list.

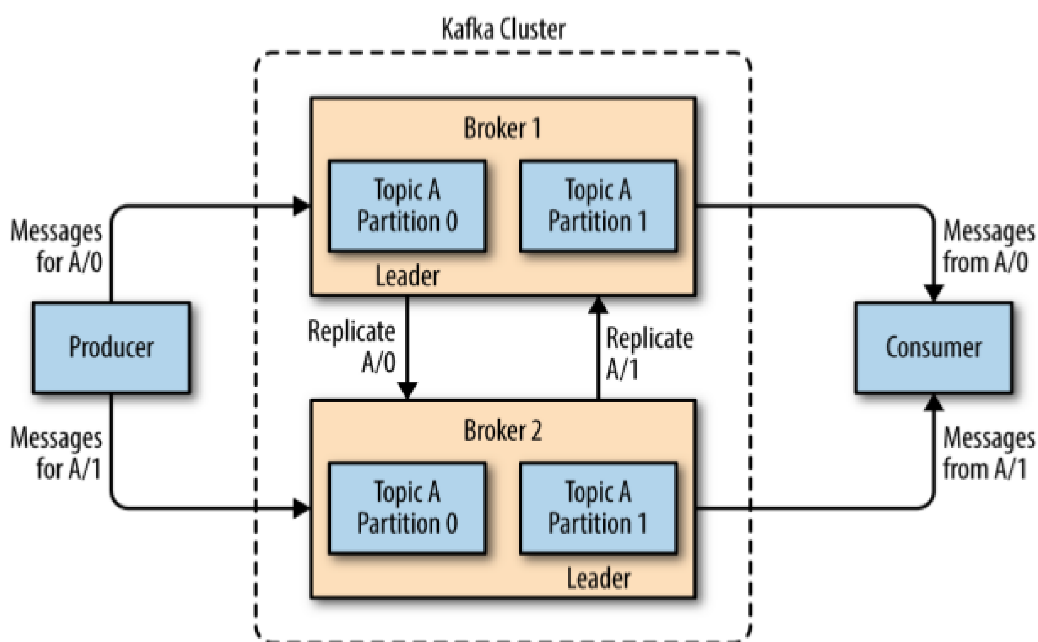

Figure 5: Kafka replication

3.5 Producers

Producers create new messages. In other publish/subscribe systems, these may be called publishers or writers. In general, a message will be produced to a specific topic. By default, the producer does not care what partition a specific message is written to and will balance messages over all partitions of a topic evenly. In some cases, the producer will direct messages to specific partitions. This is typically done using the message key and a partitioner that will generate a hash of the key and map it to a specific partition. This assures that all messages produced with a given key will get written to the same partition. The producer could also use a custom partitioner that follows other business rules for mapping messages to partitions.

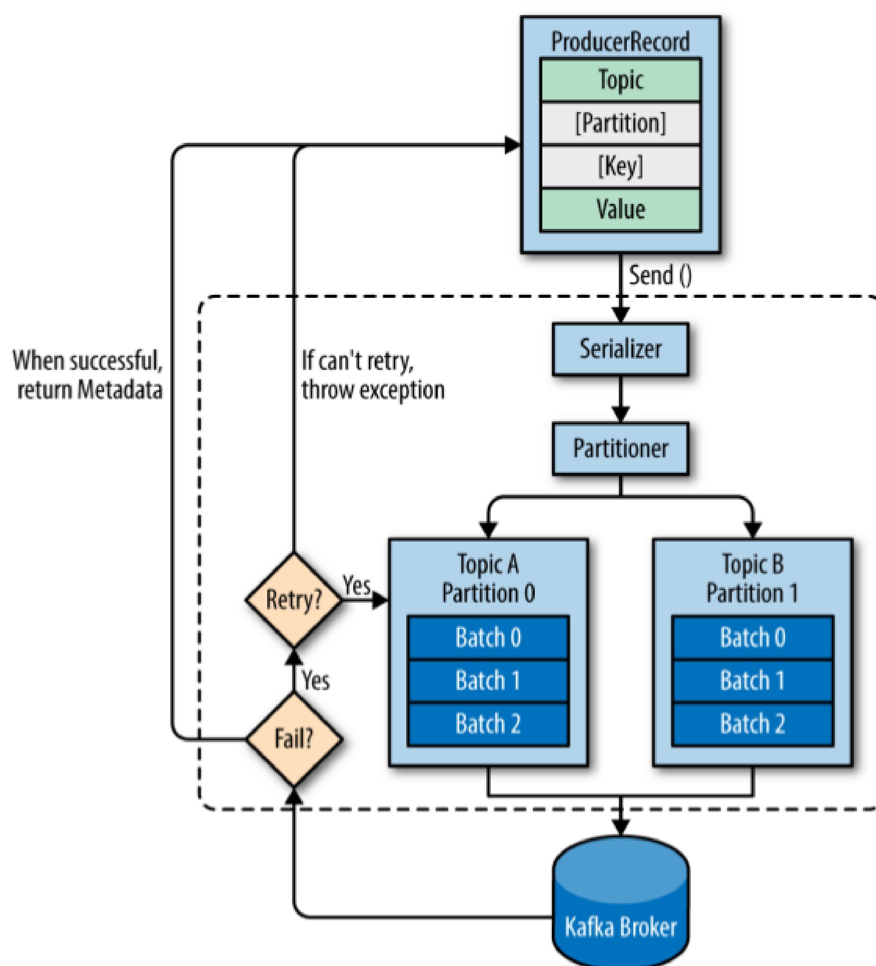

Figure 6: Overview of producers

3.6 Consumers

Consumers read messages. In other publish/subscribe systems, these clients may be called subscribers or readers. The consumer subscribes to one or more topics and reads the messages in the order in which they were produced. The consumer keeps track of which messages it has already consumed by keeping track of the offset of messages. The offset is another bit of metadata—an integer value that continually increases—that Kafka adds to each message as it is produced. Each message in a given partition has a unique offset. By storing the offset of the last consumed message for each partition, either in Zookeeper or in Kafka itself, a consumer can stop and restart without losing its place. Consumers work as part of a consumer group, which is one or more consumers that work together to consume a topic. The group assures that each partition is only consumed by one member. In Figure 7, there are three consumers in a single group consuming a topic. Two of the consumers are working from one partition each, while the third consumer is working from two partitions. The mapping of a consumer to a partition is often called ownership of the partition by the consumer. In this way, consumers can horizontally scale to consume topics with a large number of messages. Additionally, if a single consumer fails, the remaining members of the group will rebalance the partitions being consumed to take over for the missing member.

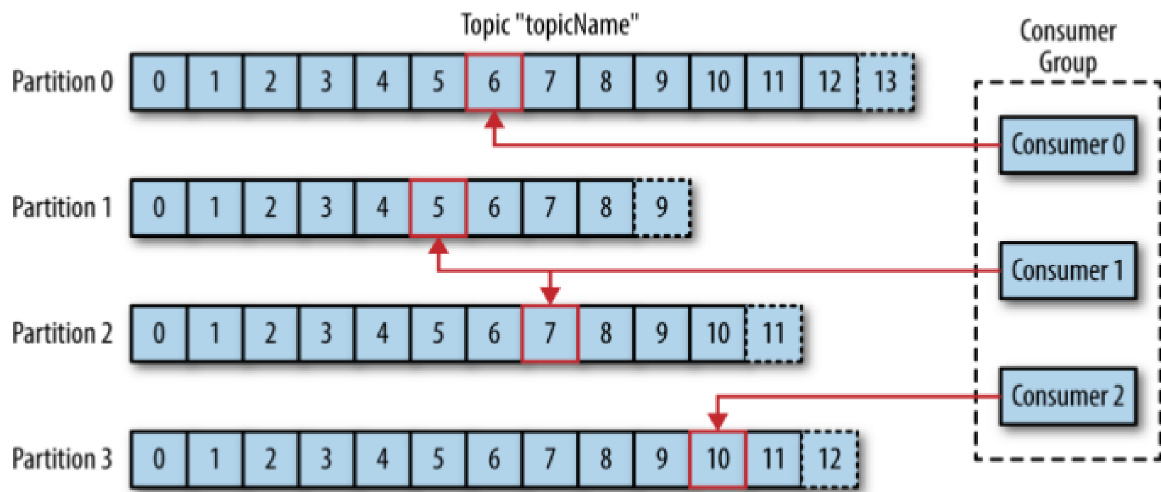

Figure 7: Consumer group

3.7 Kafka Streams

Kafka Streams is an API for building stream processing applications on top of Apache Kafka. This is achieved by applying stream processing techniques while the data is inside the Kafka log files and output transformed data in different Kafka topics, specifically applications that transform input Kafka topics into output Kafka topics (or calls to external services, or updates to databases, or whatever). It lets you do this with concise code in a way that is distributed and fault-tolerant. Stream processing is a computer programming paradigm, equivalent to data-flow programming, event stream processing, and reactive programming, that allows some applications to more easily exploit a limited form of parallel processing.

Kafka Streams has support for joining, data transform, windowing and aggregation of streams into other streams or Kafka topics. This allows you to quickly build applications to handle use cases such as joining two incoming data streams (e.g. data ETL), denormalizing incoming data (e.g. CompanyID to Company Name) or creating aggregates (e.g. Rolling average). Also Kafka Streams has the concept of viewing your stream as a changelog (KStream) or as a snapshot (KTable). This is something that is refered as the stream-table duality. This duality allows to easily process data in different ways based upon its nature (static vs. dynamic) or how you need to interact with it. Generally we can give the following analogy of stream and table in Kafka Streams:

- A stream in Kafka is the full history of events from the beginning of time to today. It represents the past and the present. A stream is a topic with a schema. Keys and values are no longer byte arrays but have specific types.
- A table in Kafka is the state of today. It represents the present. It is an aggregation of the history of world events, and this aggregation is changing constantly.

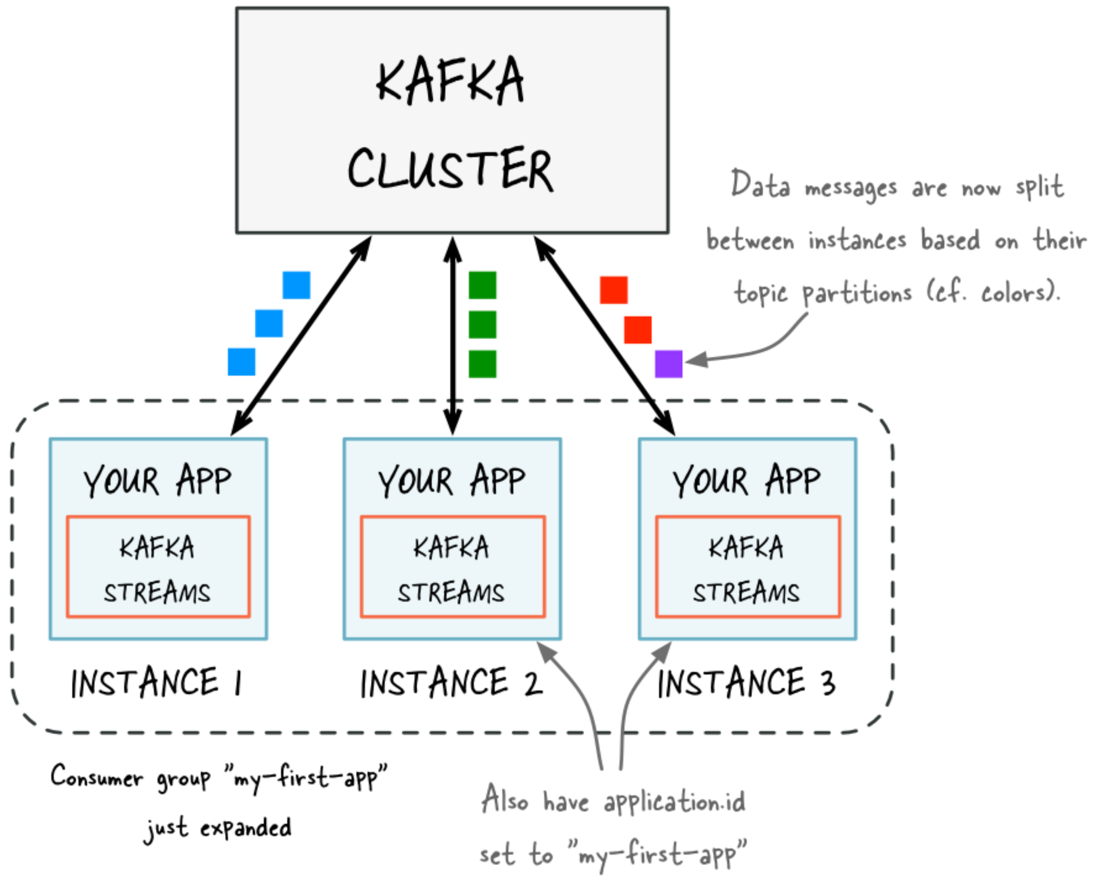

Figure 8: Kafka Streams API overview

4. SENSOR FUSION ENGINE

In this section a brief description and architecture of a sensor fusion engine (SFE) will be presented. This sensor fusion engine is implemented to enable processing and consolidating data from heterogeneous sources enabling the integration and interpretation of different types of data, with the use of multiple algorithmic flows. In addition to the statistical advantage gained by combining same-source data the use of multiple types of sensors increases the accuracy with which a quantity can be observed, interpreted and used for an event recognition. The most fundamental mechanism of the SFE involves:

1. the detection of pre-defined events,
2. the decision or inference regarding the characteristics of an observed entity
3. the interpretation of the observed entity in the context of a surrounding environment and relationships to other entities.

4.1 Contextors and network module

The architecture of SFE is partially based on the contextor's theory. A typical contextor is a software abstraction that models a relation between variables of an Observed System Context which is the composition of situations as observed by the system. A contextor is comprised of a functional core and of typed input and output communication channels as depicted in the figure bellow.

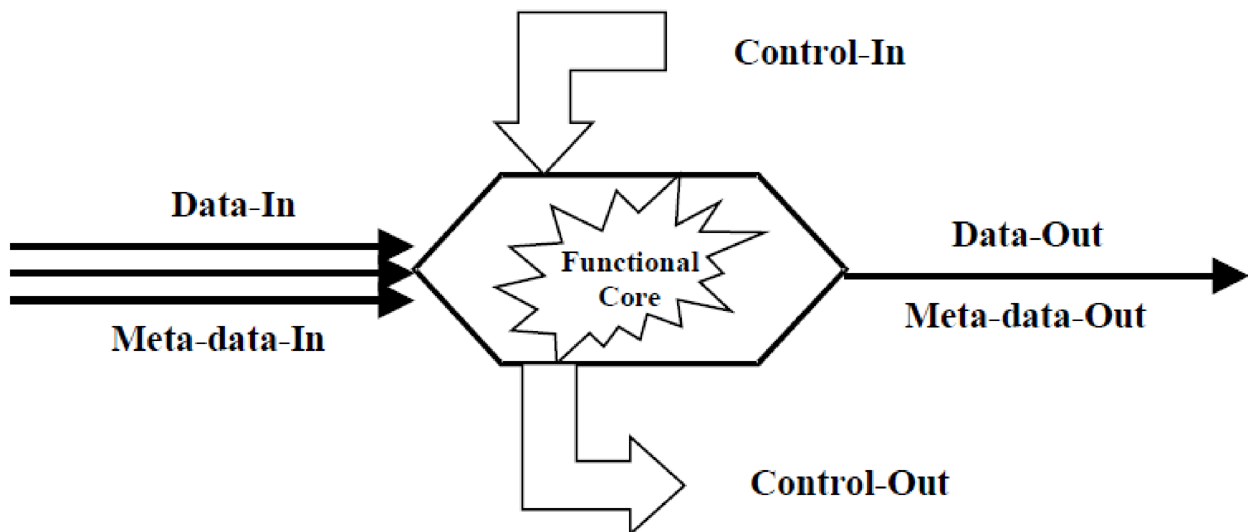

Figure 9: Graphical depiction of a contextor

The *functional core* of a contextor implements a relation between variables of the Observed System Context. The input channels of a typical contextor are of two types:

- *Data-In* corresponds to the variables of the Observed System Context that are used as inputs by the functional core of the contextor.
- *Control-In* corresponds to commands received from other contextors to set the internal parameters of the contextor. These parameters may concern the functional behavior of the contextor as well as non-functional aspects such as the QoS (Quality of Service) expected by other contextors.

Symmetrically, a typical contextor provides two types of output (output channels):

- Data-Out corresponds to the values of some variables of the Observed System Context returned by the contextor.
- Control-Out is used by the contextor to send control commands to other contextors. For example deactivate another contextor, if detected that it does not provide desired QoS. [14]

The Sensor Inbound Service(SIS) is responsible of the external data sources that have been selected as inputs to a specific data processing workflow of. Generally speaking SIS is a network module responsible for providing the data sources needed to the workflows and to the contextors of the SFE.

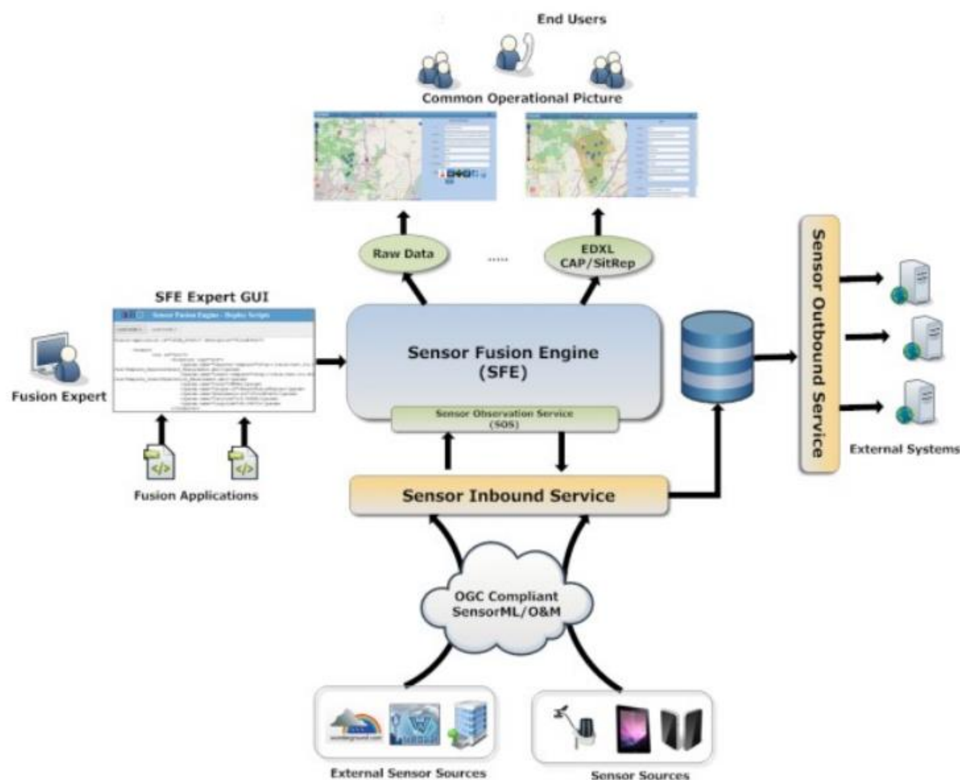

Figure 10: SFE architecture

The SIS module is a critical component of the SFE since it is responsible for the management of the input data streams that deployed FM's applications depend on, and the accurate dispatching of new values coming from the underlying network to them. Based on the information provided by the deployed application script in the SFE, the SIS module is responsible to recognize the input streams (i.e., streamers) on the fusion process and to collect their values.

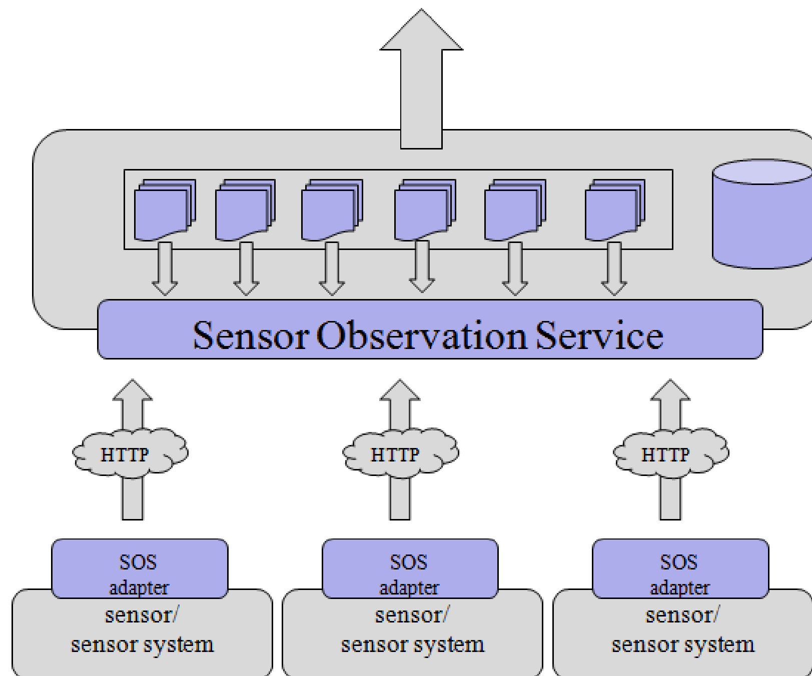

Figure 11: Network Module of Fusion Module

4.2 Big data preprocessing

Contextors of the SFE and the SIS module have a tight relationship and the interconnection between these two software layers should be functioning properly in order the stream of data to be processed normally. In the scope of this thesis was decided to implement the SIS with the Apache Kafka as an inbound service in order to support the incoming data stream. With this technical decision ensured high throughput of the incoming data streams. However more design decisions should be made in order to accumulate modules capable of withstanding large capacities of data streams.

Big Data is often described as the "4Vs" which are Volume, Variety, Velocity and Veracity referring of course to data. In its lifecycle, data travels through four different phases as shows in the next figure. These are:

1. Data generation
2. Data acquisition
3. Data storage
4. Data analytics

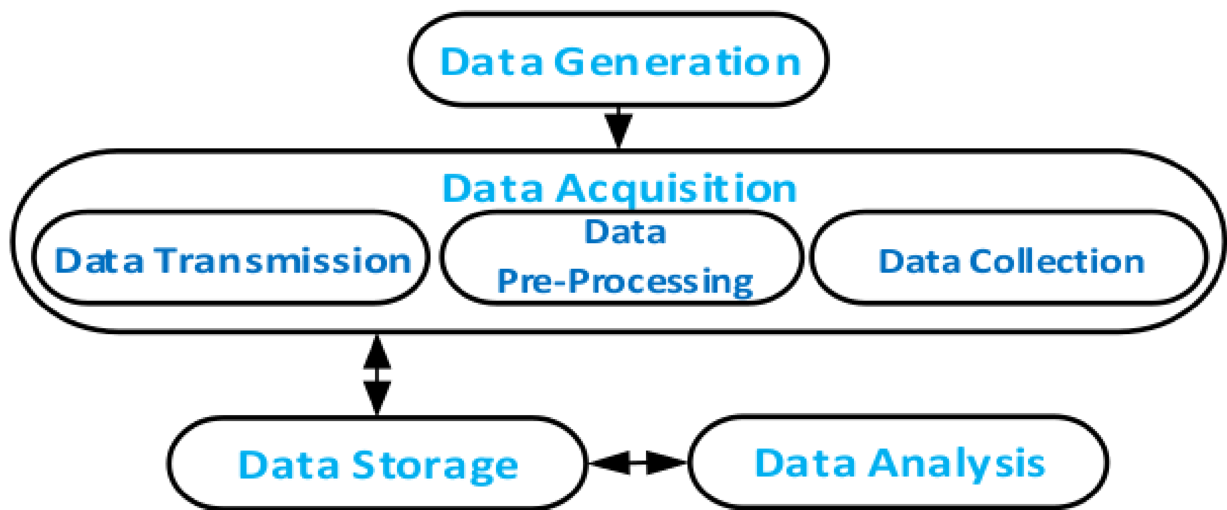

Figure 12: Data acquisition

Data generation is the phase where data creation is taking place from several different sources. These could be IoT sensors, mobile devices or users. Data acquisition consists of data collection, data transmission, and data pre-processing. While large amounts of data can lead to better analysis and therefore better results a lot of the raw data that is been collected can be useless and leads to unnecessary waste of processing power. Therefore this raw data must be channeled through a pre-processing in which activities such as data cleansing, de-duplication, compression, filtering, and format conversion take place. The pre-processing step is crucial since the computations and manipulation of data could either improve the general performance of the next two phases or could sabotage correct analysis and decision making in the top software levels. [15]

5. RESERVOIR SAMPLING ALGORITHMS

Reservoir Sampling Algorithms are a family of algorithms widely used in data science and statistical analysis. They are a subset of randomized algorithms because they use randomness in selecting samples and values for replacement inside the samples. It is precisely because of this randomness that offers equal opportunities through probabilities throughout the sample of values to be represented in the final pool and that it has an agnostic behavior towards the source size, that the reservoir algorithms are suitable for big data and handling data streams.

A basic approach of taking a sample of a set of data stream elements could be that of assigning a random number between 0 and 1 to each element that we process and then revisit the stream in order to determine from the random that we assigned if that number could be saved into the reservoir. Of course an algorithm like this can be translated in the very disappointing O(n^2) or at best O($2n$). However the biggest problem of taking samples in data streams lies on the unknown number of elements in the source. Practically we assume that their number approaches the infinity. By requiring a sample of finite number we requiring a single pass algorithm of O(n) complexity.

Following is a description of various implementations of reservoir algorithms and the challenges of implementing them in a distributed environment for the new Fusion box architecture.

5.1 General Reservoir Sampling Algorithm

One of the most known Reservoir algorithms is the algorithm R described by Jeffrey Vitter. The algorithm is a simple but elegant solution of randomly choosing values from an infinite stream of numbers to collect inside a predefined pool of values(reservoir) while maintaining fair selection and removal from the reservoir possibilities with the increased amount of incoming data. This is achieved by making use of the mathematical induction. [16]

For our origin problem we have $X = [x_1, \dots, x_n]$ a stream of unknown size or unknown n. We want to obtain a sample $T_i$of $k < n$ items where:

$$\forall\, 1 \leq j \leq i \leq n : Pr(x_j \in T_i) = \Pr(x_j \in T_i)$$

after the i-th step. The two steps of the algorithm are the following:

1) if $i \leq k$: $T_i = T_{i-1} \cup \{x_i\}$
2) else: With probability $\frac{k}{i}$ replace one with equal probability chosen element in T_{i-1} with x_i.

The first step of the algorithm solving the above problem is as the rest of the Reservoir algorithms to add the first n elements of the data stream into a reservoir. This can be a data structure, a file or a set of pointers to the memory. Then the rest of the data stream elements are sequentially processed. For every element with position inside the data stream k with $k > R$ where R equals the reservoir size a random number j between 0 and 1 is being calculated. Then if j is smaller than R divided by k then the element is stored inside the reservoir replacing a randomly chosen stored element. This algorithm can be described as follows:

```
(* S has items to sample, R will contain the result *)
ReservoirSample (S[1..n], R[1..k])
  // fill the reservoir array
  for i = 1 to k
     R[i] := S[i]
  // replace elements with gradually decreasing probability
  for i = k+1 to n
    j := random(1, i)   // important: inclusive range
    if j <= k
       R[j] := S[i]
```

To prove that all the elements have equal probability to enter the reservoir we use induction. After the $(i-1)$-th round, let us assume, the probability of a number being in the reservoir array is $k\ /\ (\ i\ -\ 1\ )$. Since the probability of the number being replaced in the i-th round is $1/i$, the probability that it survives the i-th round is $(\ i\ -\ 1\ )\ /\ i$. Thus, the probability that a given number is in the reservoir after the i ith round is the product of these two probabilities, i.e. the probability of being in the reservoir after (i-1)th round, and surviving replacement in the i-th round. This is $(\ k\ /\ (\ i\ -\ 1\ )\ )\ *\ (\ (\ i\ -\ 1\ )\ /\ i\ )\ =\ k\ /\ i$. The result holds for i, and is therefore true by induction.

A simpler example of the equality is given bellow. Supposedly the probability of the (i)th element to be selected will be equal of $\frac{1}{i}$ multiplied of all the next elements to not be selected until n. That is possible to be calculated with induction with the following mathematical operations:

$$\frac{1}{i} * \left(1 - \frac{1}{i+1}\right) * \left(1 - \frac{1}{i+2}\right) * \dots * \left(1 - \frac{1}{n}\right) =$$

$$\frac{1}{i} * \left(\frac{i}{i+1}\right) * \left(\frac{i+1}{i+2}\right) * \dots * \left(\frac{n-1}{n}\right) =$$

$$\frac{1}{n}$$

The average number of records in the reservoir at the end of the algorithm is:

$$n\ +\ \sum_{n \le t < N} \frac{n}{t+1}\ = n(1 + H_N - H_n)\ \approx n(1 + \ \ ln\frac{N}{n}\ )$$

Where the average number of records chosen for the reservoir after t records have been processed so far is:

$$n(H_N -\ H_t) \approx n\ ln\frac{N}{t}$$

5.2 Efficient Reservoir Sampling Algorithm

As it was mentioned the traditional approach of the reservoir sampling algorithm it has a complexixity of O(n). Each element must be traversed directly and the random number generation must be invoked on each element. O(n) invocation of the random number generation is a substantial cost since it can be more expensive compared to the cost of iterating to the next element in a sequence. On this scope a different pattern was perceived. Instead of calculating a random number for every element skipping reservoir sampling algorithms decide whether to calculate the random number or skip to the next one. Other implementations of this family of algorithms decide the number of elements to skip achieving faster results. [17]

The algorithm of this can be described as follows:

```
top := N - n; Nreal := N;
while n>=2 do begin
        V := UNIFORMRV( ); S := 0; quot := top/Nreal;
        while quot > V do begin
                S := S + 1;
                top := -1.0 + top;
                Nreal := -1.0 + Nreal;
                quot := (quot x top)/Nreal
                end;
        Skip over the next S records and select the following one for the sample;
        Nreal := -1.0 + Nreal; n := -1 + n
        end;
{ Special case n = 1)
S := TRUNC(ROUND(Nrea1) x UNIFORMRV( ));
Skip over the next S records and select the following one for the sample;
```

5.3 Weighted Reservoir Sampling Algorithm

In the algorithms bellow it is assumed that the importance of data was equal and with a weight value of 1. However that is not always the case and a weighted version of the reservoir sampling algorithm was needed. Pavlos S. Efraimidis figured a solution in in a paper titled Weighted Random Sampling with a Reservoir. [18]

As you process the stream, assign each item a key. For each element i in the stream , we have k_i the item's key, w_i the weight of that item and u_i a random number between 0 and 1. The key is a random number to the n'th root where n is the weight of that item in the stream:

$$k_i = u_i^{\frac{1}{w_i}}$$

We keep the top elements ordered by their keys k_i, where n is the size of the sample. With non-weighted elements (i.e. weight = 1) w_i is always 1, so the key is simply a random number and this algorithm degrades into the general reservoir sampling algorithm mentioned above.

However with weights the probability of choosing i over j is the probability that $k_i > k_j . k_i$ can have any value from 0 - 1. However, it's more likely to be closer to 1 the higher w is. The distribution of this looks like when comparing to a weight 1 element by integrating k over all values of random numbers from 0 – 1 and it has the following form:

$$\int_0^1 u^{\frac{1}{w}} du = \frac{w}{w+1}$$

6. IMPLEMENTING DISTRIBUTED RESERVOIR SAMPLING WITH KAFKA STREAMS

In a distributed environment we deal with an input stream consisting of several sub-streams and each sub-stream is feed to a single process. The problem lies with the extension of the simple algorithm to efficiently sample all sub-streams in parallel and still generate k uniform samples from the entire input stream in the end.

We assume that there are two sub-streams of size m and n, respectively. Both m and n are far greater than k. In the first step of the algorithm, workers work on their own sub-streams in parallel, using the basic algorithm. When both workers finish their sub-stream traversal, two reservoir lists R and S are generated. In addition, both workers also count the number of items in their own sub-streams during the traversal, and thus m and n are known when R and S are available.

The critical step is to combine the two reservoir lists to get k items out of them. To do this, we assign weights to items according to the sizes of the sub-stream where they were sampled in the first step, and then do a second sampling phase. We run k iterations for this secondary sampling. In each iteration, we flip a random coin such that, with probability $p = m/(m+n)$, we pick one random sample from reservoir list R, and with probability 1-p, we pick one random sample from reservoir list S. At the end of the k-th iteration, we will get the final reservoir list for the entire stream. This algorithm is described as follows:

```
for(sub-stream s: sub-streams) do in parallel {
  simple sequential reservoir sampling and count length of s;
}
double p = (double) m / (m+n);
for(int i = 0; i < k; ++i){
  j = rand.nextDouble();
  if(j <= p)
    move a random item from R to T;
  else
    move a random item from S to T;
}
return T;
```

It can be shown by induction that in each iteration of the second sampling phase, any item in the entire stream has probability of 1/(m+n) being chosen. Again, by total probability, any item has probability of k/(m+n) being chosen during the execution of the algorithm, and thus the algorithm generates k random samples from the entire stream. This algorithm runs in O(max(m,n)) time and uses O(k) in space. To generalize the algorithm to cases with more than two sub-streams, one only needs to combine reservoirs lists in pairs, which can also be done in parallel. The proof techniques remain the same and thus are omitted.

So in general the problems that we encounter by implementing the distributed version of a reservoir sampling algorithm are:

1. The division of the initial stream to equal size of streams
2. The simultaneously processing of the sub-streams
3. The final collection of the samples and the final merging of them

By using Kafka Streams we solved these problems easily by utilizing the characteristics of the Kafka platform. The division of the sub-streams and the allocation of processors can easily be determined by the number of Kafka Stream clients. The unit of parallelization in Apache Kafka is the number of partitions. By using an equal number of clients as the number of partitions we divide the initial stream (in Kafka is translated as a single topic) into equal sub-streams as KStreams and the final collection is achieved by aggregating the samples into a Ktable.

7. EXPERIMENTAL RESULTS

In order to evaluate the implementation of the algorithms in Kafka Streams API 4 virtual machines running the same instance of the algorithm were created. All the virtual machines were in the same network consuming a stream of infinite events with the following message avro format:


```
{
  "namespace": "eu.rawfie.uxv",
  "name": "Location",
  "type": "record",
  "doc": "Geographic location",
  "fields": [
   {
    "name": "header",
    "type": "Header"
   },
   {
    "name": "latitude",
    "type": "double",
    "unit": "rad",
    "doc": "Latitude in the WGS 84 reference coordinate system",
    "min": -1.570796326794897,
    "max": 1.570796326794897
   },
   {
    "name": "longitude",
    "type": "double",
    "doc": "Longitude in the WGS 84 reference coordinate system",
    "unit": "rad",
    "min": -3.141592653589793,
    "max": 3.141592653589793
   },
```


In order to collect the results of the benchmarking JMH library was utilized with throughput mode (number of operations in a time unit) as a measurement. The results were the following:

Algorithms	Simple	Skipping	Weight
Average Time in ms	0.512	0.381	0.598

Figure 13: Benchmarking results

Not surprisingly the skipping algorithm was faster than the other two implementations. However the implementation with the reservoir sampling with weights has only a small performance penalty over the basic reservoir sampling algorithm.

8. CONCLUSIONS

Data pre-processing is a stage that can be critical to the overall process of data processing. By using the appropriate theoretical background we can accurately predict an event before even the initial stream of data passes through the processing step. Reservoir sampling algorithms can provide speed and precision in the decision making process and provide us with additional help in a more sophisticated computing resource allocation.

ABBREVIATIONS – ACRONYMS

IoT	Internet of Things
HDFS	Hadoop Distributed File System
GPS	Global Positioning System
SCADA	Supervisory Control And Data Acquisition
CEP	Complex Event Processing
SFE	Sensor Fusion Engine
SIS	Sensor Inbound Service

REFERENCES

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, “Internet of Things (IoT): A vision, architectural elements, and future directions,” Future Generation Computer Systems, vol. 29, no. 7, pp. 1645–1660, 2013.

[2] L. Kong, M. K. Khan, F. Wu, G. Chen, and P. Zeng, “Millimeter-Wave Wireless Communications for IoT-Cloud Supported Autonomous Vehicles: Overview, Design, and Challenges,” IEEE Communications Magazine, vol. 55, no. 1, pp. 62–68, 2017.

[3] A. H. Kashan and B. Karimi, “Scheduling a single batch-processing machine with arbitrary job sizes and incompatible job families: An ant colony framework,” Journal of the Operational Research Society, vol. 59, no. 9, pp. 1269–1280, 2008.

[4] J. Dean and S. Ghemawat, “MapReduce,” Communications of the ACM, vol. 51, no. 1, p. 107, Jan. 2008.

[5] A. Jlassi, P. Martineau, and V. Tkindt, “Offline Scheduling of Map and Reduce Tasks on Hadoop Systems,” Proceedings of the 5th International Conference on Cloud Computing and Services Science, 2015.

[6] Y. Zhou, “Scalable and Adaptable Distributed Stream Processing,” 22nd International Conference on Data Engineering Workshops (ICDEW06), 2006.

[7] S. Takano, “Adaptive processor: a model of stream processing,” 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.

[8] K. Patroumpas and T. Sellis, “Event Processing and Real-Time Monitoring over Streaming Traffic Data,” Web and Wireless Geographical Information Systems Lecture Notes in Computer Science, pp. 116–133, 2012.

[9] Y. Liu, W. Chen, and Y. Guan, “Approximate membership query over time-decaying windows for event stream processing,” Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems - DEBS 12, 2012.

[10] “The Future of Event Processing,” Event Processing for Business, pp. 195–235, Sep. 2015.

[11] B. Minaei-Bidgoli and S. B. Lajevardi, “Correlation Mining between Time Series Stream and Event Stream,” 2008 Fourth International Conference on Networked Computing and Advanced Information Management, 2008.

[12] M. P. R. Junior, “Dg2Cep: An On-Line Algorithm For Real-Time Detection Of Spatial Clusters From Large Data Streams Through Complex Event Processing.”

[13] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein, “Building a replicated logging system with Apache Kafka,” Proceedings of the VLDB Endowment, vol. 8, no. 12, pp. 1654–1655, Jan. 2015.

[14] J. Coutaz and G. Rey, “Foundations for a Theory of Contextors,” Computer-Aided Design of User Interfaces III, pp. 13–33, 2002.

[15] I. Taleb, R. Dssouli, and M. A. Serhani, “Big Data Pre-processing: A Quality Framework,” 2015 IEEE International Congress on Big Data, 2015.

[16] Vitter, Jeffrey S. (1 March 1985). "Random sampling with a reservoir"

[17] J. S. Vitter, “An efficient algorithm for sequential random sampling,” ACM Transactions on Mathematical Software, vol. 13, no. 1, pp. 58–67, Jan. 1987.

[18] “Weighted Random Sampling, 2005; Efraimidis, Spirakis,” SpringerReference.