KTH Royal Institute of Technology
Faculty of Software and Computer Systems
Department of Information and Communication Technology

KTH Information and Communication Technology

Decentralized stream processing: algorithms, methods and tools

Master thesis

Pietro Cannalire

Master programme: Software and Computer Systems
Branch of study: Information and Communication Technology
Supervisor: Zainab Abbas, PhD student
Examiner/supervisor: Vladimir Vlassov, Associate Professor

Stockholm, January 2018

Thesis Supervisor:
Zainab Abbas, PhD student
Department of Information and Communication Technology
Faculty of Software and Computer Systems
KTH Royal Institute of Technology
SE-100 44, Stockholm
Sweden

Declaration

I hereby declare I have written this master thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis. Moreover, I state that this thesis has neither been submitted nor accepted for any other degree.

In Stockholm, January 2018

.....
Pietro Cannalire

Abstract

Stream processing is continuously growing and it involves more and more application fields and some of them deal with high throughput and geographically distributed data. Centralized architectures are enough to satisfy a lot of applications by employing already existing stream processing frameworks which are able to manage distributed data processing, but it is not rare the necessity to distribute even more processing by employing decentralized infrastructures. This trend is justified by the necessity to lower the stress on high rate streaming applications geographically distributed, by the difficulty to access costly links placed in the middle of the network and the possibility, for example because of privacy concerns, to depend as less as possible from communication with environments geographically located far from each other. This thesis project aims to create a new concept to deploy decentralized architectures by relying on already existing stream processing frameworks analyzed in a brief survey.

Methods and algorithms are the basic elements for efficient processing in such architectures, therefore commonly used algorithms for centralized architectures are studied and their features are analyzed to check whether they can be adapted to work on decentralized architectures. Then features needed for an algorithm to run on a decentralized architecture are provided.

A use case is also presented to verify the decentralized architecture and an algorithm with proper features is implemented to show the effectiveness of such environment on data-intensive streaming application.

Keywords: *stream processing, decentralized, algorithms, Apache, Spark, Kafka, Misra-Gries algorithm*

Abstrakt

Nyckelord:

List of Figures

Contents

Chapter 1

Introduction

1.1 Problem definition

The world of data has been growing incessantly since the last decade. Every second an unbelievable amount of data is being generated from human activities, machines and sensors and from monitoring the environment in which people lives. The need to analyze this huge quantity of information goes along with the necessity to develop proper systems which are able to store it and elaborate it. The notion of big data is quite recent and is related to its characteristic of being high-volume and high-velocity information characterized by high variety with needs of veracity of data [1] which finds expression in decision making and process automation as well as in building enhanced business insights [2]. Big data can be generated from many different sources. Therefore, to have a concrete view of what Big Data is, we can classify it into three typologies [3]:

- *Social Networks (human-sourced information)*: data being produced as a consequence of human activity made by videos, pictures, Internet searches, text messages and generally by social network activity.
- *Traditional Business Systems (process-mediated data)*: business related data e.g. stock records, commercial transactions and e-commerce.
- *Internet of Things (machine-generated data)*: data mainly generated by sensors monitoring weather, home environment, security, phone locations and etc.

The described typologies depict a situation of continuously produced data: social networks pervade our days in every aspect, e-business and e-commerce are continuously growing and sensors are everywhere to offer sophisticated services.

The problem that this work wants to face is the issue which arises when dealing with a huge amount of geographically distributed streaming data. When there is the need to

analyze in a streaming fashion data produced by any kind of source to extract meaningful information out of it, the costs can be sensitive and important. Costs can appear in terms of latency to get a given result starting from streaming sources and in terms of bandwidth required to exchange huge amount of data. Moreover bandwidth costs are strictly related to money since ISPs will charge customers for the usage of their infrastructure [4, 5].

1.2 Motivation

Many stream processing frameworks currently on the market offer the capability to interact with streaming data and analyze them for any purpose. The complexity of the data on which they can work forces these systems to rely on distributed architectures and parallel-based processing. Hadoop [6], Spark [7], Storm [8], Flink [9] are only some of the most used frameworks to deal with large amount of batch or stream data and they are usually based on basic entities that coordinates the work of the whole system and other ones that process parallelized data to speed up the computation.

However, when the nature of data is geo-distributed, which means that data comes from regions physically far from each other and the amount of data is huge (it's Big Data), even previously mentioned systems are stressed by the high resource requirements to process such amount of data. In such critical conditions guaranteeing robustness may be a problem as well as the cost of sending data across several regions could be significant because of bandwidth needed to exchange a lot of data at potentially high rate. Recent researches have proposed new designs for data stream processing systems [10] and some systems have been developed to work with hundreds of CPU cores and terabytes of memory [11], but the upgrade of already existing systems is costly and may not be feasible in certain circumstances.

For this reasons the next step is to follow a different approach: decentralization. While systems named so far are distributed in processing and centralized in managing workload and resources, a decentralized system cedes power to local independent entities, as they were working alone [12], and then reconciles the information computed by using a network that links every system to each other and exchanges information to extract a global view. The idea behind decentralization is edge computing which places "data acquisition and control functions, storage of high bandwidth content, and applications closer to the end user" [13]. A decentralized architecture delegates first elaboration of data to the edge of the network which is responsible of a small part of the entire incoming data allowing to process it with less strict concerns to resources utilization and independently from other edge entities. In this way bandwidth is less utilized on large scale because data is still received and elaborated, but data exchange is reduced to the minimum one required to

communicate to a central entity what is happening to every edge.

This environment needs also proper algorithms to perform the same kind of analysis as a standard distributed structure: though the basic operations and transformations of streaming data can be exactly the same of a distributed framework, for example counting or aggregation, the following step,the reconciliation of information from edge nodes, introduces some constraints on algorithms and operations which can be executed on data which should be discussed and evaluated.

1.3 Approach

The work in this thesis follows methodologies and methods as defined in [14]. Two main milestones are achieved for which different methodologies/methods are employed:

1. finding which frameworks better suit to design decentralized architecture
2. implement a chosen algorithm on top of designed decentralized architecture

For the first milestone a *qualitative descriptive research method* has been used to examine stream processing frameworks currently on the market and studying their main features and characteristics together with their suitability in the project purpose. Then an *exploratory research method* has been used to investigate the possibility to interconnect possible designs hypothesis in a simple practical application by coding and to draw conclusions about building blocks to use.

Second milestone has been achieved first by employing *empirical research method* to evaluate which algorithm can be suitable to be developed in designed architecture, then by coding to implement chosen algorithm and be able to analyze data in the decentralized architecture.

Chosen methods are followed with the ultimate intent to realize a scenario which involves independent clusters of machines running a stream processing framework that are able to communicate to each other by means of a common shared infrastructure. Every cluster can be considered as a peer in a P2P network that exchanges information with other peers about data gathered and aggregates computed locally with a chosen and implemented algorithm.

Coding in particular took the great part of the entire project because of the need, in several aspects of development, to try how APIs really worked due to lack of clear and complete documentation and in which way they could be interconnected to each other to achieve the desired result. Moreover two chosen algorithms have been developed on top of two slightly different frameworks: Structured Streaming turned to be more valuable in different aspects respect to Spark Streaming. Than, algorithms taken into

consideration, which include the ones in the main algorithm development but also the ones in the exploratory research method, have been developed with significant changes in input data structures and in theirs mere logic of computation of result because of different context environment in which they should run: each application which computes a decentralized result is made by a first tier application which runs on each cluster and the second tier application which gathers results from them and computes a global result. More details will be provided in Chapter 5.

1.4 Contributions

This master thesis aims to achieve several contributions:

1. Providing a concrete decentralized architecture, ready to execute decentralized algorithms, which finds its fundamental tools in Apache Kafka and Apache Spark Structured Streaming. This architecture is able to elaborate more incoming data than a distributed configuration and to prevent large amount of data to move throughout the network by reducing output data size and employing a summarization mechanism.

2. Providing features that an algorithm should have to be decentralized in such architecture and showing two concrete examples of working algorithms and how they are implemented.

1.5 Outline

After this introductory part, the second chapter wants to give a general idea of the background context by explaining basic models for currently used stream processing frameworks, stream processing concepts which are useful throughout the whole work, most used algorithms working on data streams and a section to discuss related work.

In the third part it will be discussed the tools which have been considered and the ones finally chosen. Then a section will explain different architecture configurations and where the tools are placed. The fourth part will deal with methods and algorithms that can be applied on a decentralized infrastructure and that are chosen to run on designed architectures. The fifth part will show the implementation of the system and the algorithms. Sixth part will compare and evaluate the centralized and the distributed solutions with the decentralized one, and in the last part there will be a conclusive summary of the entire work.

Chapter 2

Background

2.1 Stream processing models

The strong presence of data in our lives has lead to a growing attention in developing new ways to process large volume of information. Big Data [3] needs proper technologies and lots of resources to be processed and even single high-performance machines are not enough for the goal: clusters of machines are the basic support to deal with high-volume and high-performance processing. Clusters usually are made of hundreds or thousands of processing units along with a great amount of storage capacity which are exploited to elaborate and analyze large datasets and to manage continuous queries from the network. While supercomputers provide really high computing power, clusters are based on cheap commodities hardware working together to provide reliability, efficiency and scalability when running data-intensive computing applications [15].

A cluster can support efficiently these features thanks to its distributed nature: most of current stream processing frameworks rely on different actors which play a partial but important role during the whole processing by communicating constantly with other actors in the cluster and exchanging information about the computation and the status of the work.

2.1.1 Data model

Stream processing frameworks work with streams of *tuples*: each tuple is an immutable and atomic item representing the information that the application should process, it can be a sequence of characters, bytes or anything else that can be further elaborated. Streaming data can be categorized into three classes [16]:

- Structured: data with known schema

- Semi-structured: data expressed by using markup languages (HTML, XML, JSON, etc...)
- Unstructured: custom or proprietary formats data (binary, video, audio)

In general the order in which the tuples are received by the system is not the same in which they are generated by the sources because the places where tuples are created can be different and far from each other and moreover communication delays can cause some tuples to be read much later. For this reason every tuple can be associated with two different time references:

- *event-time*, that corresponds to the time when the event generating the data has happened, and
- *processing-time*, which is instead the time when the tuple is received by the system and processed

They are differently defined because application logic usually involves event-time to arrange correctly in time every tuple and processing-time to take corrective measures on data.

2.1.2 Processing model

An incoming data stream enters into the system and passes through a series of Processing Elements (PEs) which can perform transformation on the data flow [17].

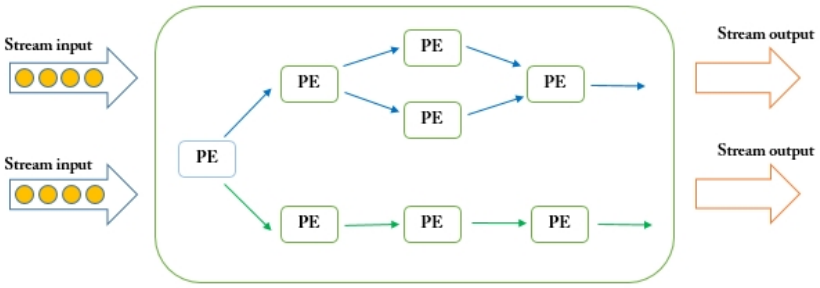

Figure 2.1: Processing model

As shown in figure 2.1, when an output tuple is generated by a processing element, if it is not permanently written on a data storage, it becomes the input of a new processing element: a set of processing elements connected together generates a *data flow graph* which has a key role in building a *logical plan* that shows how processing elements are linked together from the sources to the sink passing through transformations. Starting from the data flow graph and logical plan, every framework builds also a *physical plan*

which is the representation of how actually the computation is split among operative system processes.

In order to be easily executed on a distributed environment the application is usually split into jobs, which are a set of processing elements responsible to modify the data, and each job can be divided itself into tasks that can be executed in parallel over multiple machines of a cluster. In this way every framework can implement its own policies to exploit the cluster and run an application in a distributed manner.

2.2 Methods and algorithms

In general, a data stream can be unbounded, meaning that its size is unknown and that possibly contains infinite elements, therefore it is neither convenient nor feasible to store it on a permanent data storage because it could easily reach a lot of volume in terms of physical data.

2.2.1 Type of queries

Tuples in a stream can be considered as rows in a relational database and then we can wrongly think that many common database operations could be applied on them, but the main feature of a stream is that each tuple passes once from the system and in that moment should be elaborated. Hence, even if some algorithms aim to reach the desired purpose, they need to be adapted, modified or completely changed in the way they work.

Stream processing systems should answer to two different queries about data:

- *Ad-hoc query*: it is a question asked once to the system to discover the current state of the stream
- *Standing query*: it is stored and permanently executed to compute an answer considering new elements that have come

Both queries can ask the system to compute the same result but the way it is executed is different because a *standing query* is continuously running and internally maintains a state that is updated while the stream is flowing, whereas the system can answer to *ad-hoc queries* only relying on data and information that it can retrieve in that moment since the system can't store the entire stream and can't be prepared to satisfy every kind of queries on it, then it is dependent to the way the application is implemented. A visual representation of queries can be found in figure 2.2.

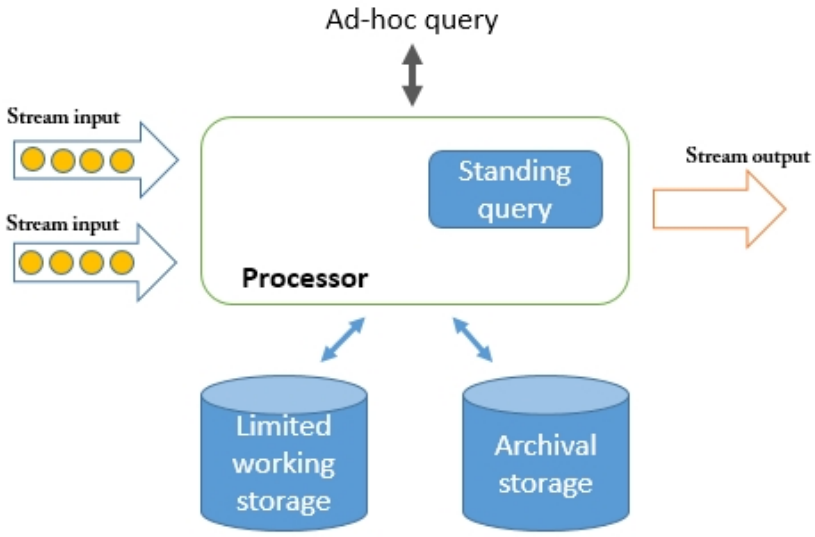

Figure 2.2: Type of queries. Data streams enter the system. Processing requires a limited working storage, typically main memory, and archival storage, like disks. While a standing query is stored inside the processing system and continuously executed over input data streams, ad-hoc query is asked externally to the system about incoming data.

2.2.2 Methods

2.2.2.1 Data synopsis

When dealing with large amount of data may be necessary to reduce input data set with a smaller version which represents the original one in some way. In several applications a data synopsis can lead to significant advantages [18]:

- it may be stored in main memory allowing faster access to data and avoid disk accesses
- it can be sent over the network at a smaller cost than sending original data

Synopses can be very different according to the method that has been used for the construction and according to the feature of original data that it should represent. Among main synopses categories we have [19, 20]:

- *sampling*, which can obtain a "representative subset of data values of interests" [19]. Sampling can be achieved through different techniques and algorithms, some of them will be showed in next section.
- *histograms* which consists of a summarization of the dataset by grouping data in subsets called *buckets* for which statistics of interests are computed. Statistics of a bucket can be utilized to approximately represent data which originally generated that bucket.

- *wavelets* which was initially applied for signal and image processing but now is also "used in databases for hierarchical data decomposition and summarization" [20].
- *sketches* are summaries constructed by applying a particular matrix to input data seen as a vector or a matrix of data points.

2.2.2.2 Windowing

Starting from the assumption that any system can't store the entire stream, the *windowing* [21] mechanism supports the computation of any algorithm on only a part of it. A *window* is a buffer that retains in memory only some elements received and it can be of two types, different from each other for the policy employed for triggering the computation over the elements it contains and evicting elements from the window when other tuples are received [21]:

- *Tumbling window*: it starts the computation when the window is full and evicts all tuples inside the window after the computation
- *Sliding window*: it is processed according to its trigger policy, which can be different, and stores only the most recent tuples evicting the oldest ones.

Windows can be defined in different ways according to window management policies which specify rules to build the window [21]:

- *Count-based policy*: the window is represented by a container that can contain a fixed size of tuples
- *Time-based policy*: the window is represented by a range of times and contains tuples with a time value included into it
- *Delta-based policy*: it is specified by defining a delta threshold value and a delta attribute which are used to accept or not new tuples into the window
- *Punctuation-based policy*: it is applied only to tumbling windows. The punctuation works as a boundary which delimits the window and triggers the computation

2.2.3 Algorithms

The impossibility to store all incoming data makes it harder to execute algorithms by reading only once each tuple, but a lot of literature has grown in this sense due to the high diffusion of such environments. In the following some algorithms will be presented as representative of basic operations and transformations over data streams for original size reduction and for the extraction of main sensitive information about data stream.

2.2.3.1 Sampling and filtering algorithms

First need for streaming data is to extract a reliable sample of the entire stream aiming to obtain a statistically representative answer by querying the sample as it was the whole stream [22].

Fixed proportion sampling This algorithm produces a sample proportional to stream size which grows as far as the new tuples are received. To get a sample which represents a fraction a/b of the entire stream, each tuple key is uniformly hashed into b buckets and only tuple whose hash value is less than a are kept into the sample.

Fixed-sized sampling: reservoir sampling If any particular memory constraints are required by the system, it is possible to generate a fixed-size sample by means of *reservoir sampling* [23], an algorithm that produces a fixed-size sample which is useful in very common situations in which the size of the entire stream is not known in advance, and it is not admitted to let the sample grow indefinitely.

To store a sample of size s, *reservoir sampling* stores first s elements into the sample S and, after seeing $n-1$ elements, n^{th} element is taken with probability s/n (with $n > s$): if the element is taken it replaces an element already in the sample S taken uniformly at random, otherwise it is discarded.

Bloom filter Another operation able to reduce volume of a data stream is filtering it to accept only some tuples which satisfy a criterion by using algorithms such as the one proposed by Bloom [24]. The *Bloom filter* consists of an array of n bits initialized to 0, k hash functions which map a tuple key to n buckets and m key values which represent the set of acceptable keys. For each of the m keys it is applied every of the k hash functions and set the corresponding bit to 1 in the bit-array. When a new tuple arrives, hash functions are applied and if the resulting value for every hash function is 1 into the bit-array then the tuple is accepted otherwise is discarded. This algorithm is not immune to false positive but the probability to find them is given by the value $(1 - e^{\frac{-km}{n}})^k$ and then it can be tuned to increase efficiency.

2.2.3.2 Count-based algorithms

Once a stream is sampled, filtered or is unmodified, several algorithms can be applied on it starting from count-based algorithms in which "the algorithm keeps a constant subset of the stream along with the counts for these items" [25].

Flajolet-Martin approach In order to count distinct elements a basic approach is the Flajolet-Martin one [26] which estimates the number of distinct elements in the stream according to the maximum number of trailing 0s obtained by hashing the key of every incoming tuple: supposing that R is the maximum number of trailing 0s seen so far, 2^R is the count of distinct elements. This approach is based on the idea that "the more different elements we see in the stream, the more different hash-values we shall see" [22].

Frequency moments A more general way to count distinct elements is to compute *moments*. "Suppose a stream consists of elements chosen from a universal set. Assume the universal set is ordered so we can speak of the i^{th} element for any i. Let m_i be the number of occurrences of the i^{th} element for any i. Then the k^{th}-order moment (or just k^{th} moment) of the stream is the sum over all i of $(m_i)^k$" [22]. Hence calculating 0^{th} moment means computing the sum of distinct elements, while first moment corresponds to the length of the stream. The second moment is called *surprise number* and represents the measure of stream unevenness. Moments higher that two are calculated in a similar way as the second moment [22, 27].

The Alon-Matias-Szegedy algorithm [27] is used to compute the second moment. Supposing an infinite stream and the possibility to store k number of occurrences, for each element it is stored the value and its count. Then, similarly to previously cited *reservoir sampling*, k values are kept and, starting from the following one, the element is chosen with probability k/n, where n is the number of elements seen so far, then if it is selected it will replace another element taken uniformly at random.

Heavy hitters Another common problem to be solved is finding the most frequent element in a data stream, finding heavy hitters. Among the algorithms that find heavy hitters there is the Majority algorithm, which was proposed by Moore [28] and later proved in his optimality by Fischer and Salzburg [29]. Majority algorithm was then generalized by Misra and Gries [30] whose algorithm was not made for streaming problems but it works doing only one pass over an array, a fundamental feature of streaming algorithms.

Supposing m elements in the stream and a set of maximum k elements in the buffer, Misra-Gries algorithm maintains a counter for up to $k-1$ distinct elements. When an element x arrives, if x is in the buffer its counter is incremented, if not and the set of distinct values is less then k the element is simply added with counter equal to 1, if x it is not in the set and the set is already of size k all counters are decremented by one and, if any counter reaches zero, it is eliminated from the set.

2.3 Related work

Work in this thesis shares inspiration and assumptions adopted by INRIA group in researching a decentralized machine learning system [31] and a decentralized adaptation in recommender systems [32]: main concerns are privacy of user's data and flexibility of data usage which are not spread out among different services but close to the user.

High diffusion of mobile devices with their sensors has lead to infrastructures able to gather data and provide special services in a decentralized way: 4Sensing [33] proposed a fully decentralized system which includes its own framework to develop applications by processing mobile sensing data.

Other systems process data generated by sensors in a decentralized way. Babazadeh et al. [34] process streaming data generated by web-enabled devices over a peer-to-peer network showing that by running their topology on top of it they can achieve a better performance than running it on a single powerful machine.

Recently the discussion is involving wide-area data analysis systems (WDAS) which "must incorporate structured storage that facilitates aggregation, combining related data together into succinct summaries" [35]. These systems provide data storage at the edge of the network where users can use it for their purposes and running ad-hoc as well as standing queries: main concerns is to reduce data movement in the system allowing data to be stored and aggregated close to where it is generated as much as possible.

For the algorithmic point of view, Dobra et al. [36] compute aggregate queries over data streams by means of sketches, summaries of streams that can be used to provide approximate answers to aggregate queries while Agarwal et al. [37] studied mergeability of summaries.

Chapter 3

Architecture and tools

3.1 General concept

To build up a decentralized architecture, basic requirements are:

- clusters with processing and storage capabilities which can ideally work as much autonomously as possible from each other
- a communication mechanism which allows cluster to efficiently exchange information about local processing with other clusters and construct a global view of the system

First requirement is explained by geographical distribution: different clusters may be in different places and far from each other, then data movement is needed in order to provide basic features like replication of data, fault tolerance or simply to exchange information. When data moves, it has to use already existing WANs which have a cost in terms of bandwith: according to [38], WAN bandwith cost per month has increased from 2003 and 2008 and ISPs charge customers for using their infrastructure since deploying such networks and guaranteeing efficiency is highly expensive. Different authors in [39] and [40] have studied the necessity to reduce data movement between clusters geographically distributed by proposing specific techniques and others [41] have provided a heuristic algorithm to minimize energy and bandwith costs for cloud data centers (CDCs) providers which usually deal with several ISPs.

Deploying a decentralized architecture by relying on as much autonomous clusters as possible provides a further advantage: assuming the purpose of streaming processing, each cluster can run a different framework to analyze incoming data streams with the only constraint of the communication layer which is common to all of them and to which they must agree. Theoretically if a cluster generates any kind of meaningful data which can be exchanged and understood by other clusters, the framework which creates it has no importance. Hence the second requirement is needed to make all clusters cooperate by

exchange proper information to recreate a global view and meanwhile keeping low data movements between them.

3.2 Brief survey on available tools

In order to achieve this thesis purpose, a brief survey on available tools is made to decide which one among the stream processing frameworks and related tools currently on the market are more suitable to be chosen.

3.2.1 Stream processing frameworks

For first requirement mentioned in the previous section, many stream processing frameworks have been considered: Storm [8], Flink [9], Spark Streaming [7], Samza [42], all Apache projects, similar in objectives but different in features.

These frameworks can be categorized as follows [43]:

- *Stream-only frameworks*: Storm and Samza
- *Hybrid frameworks*: Flink and Spark Streaming/Structured Streaming

Storm. Storm is suitable for near real-time processing because achieves very low latency respect to other solutions thanks to its topologies based on Directed Acyclic Graphs (DAGs). A topology is made by *spouts* and *bolts*: the former are sources of data streams, the latter are operations that process data and output results. Storm is a pure stream framework and by default it guarantees at-least-one processing of data: to achieve exactly-once processing guarantees Storm has to be integrated with Trident which gives Storm the ability to use micro-batches and that is the only possibility to maintain a state, a fundamental feature in some applications.

Samza. Samza is a stream processing framework "designed specifically to take advantage of Kafka's unique architecture and guarantees [...] to provide fault tolerance, buffering, and state storage" [43]. It allows to keep separate each processing step allowing subscription of different subscribers on the same stream and high flexibility for stream transformation and consumption. Samza works similarly as MapReduce in referencing HDFS but keeping advantages of Kafka architecture, then it might be not a good fit "if you need extremely low latency processing, or if you have strong needs for exactly-once semantics" [43].

Flink. First Flink feature that makes it an emerging but stable competitor is being a pure streaming framework, but with the capability of batch processing, which is considered by the framework itself as a special case of pure streaming processing [44]. Flink can guarantee ordering and grouping thanks to the capability of handling event time, which means the time that event actually has occurred, and moreover "in-built Hadoop cluster YARN support is there for Flink to processing the data in map-reduce style and also in iterative intensive stream processing" [45].

Spark Streaming. Apache Spark [7] is one of the most active open-source project in data processing [44] with many active partners actually using the framework for their business [46]. It has a wide language support by running on a Java Virtual Machine and providing APIs in Java, Scala, Python and R. Spark includes many libraries for SQL support on DataFrames, for machine learning, graph processing and stream processing, and they can be used seamlessly in the same application [44]. Spark provides fast in-memory processing of data thanks to its RDD abstraction which is the support to guarantee fault tolerance: RDDs are built and distributed across the cluster and they can be reconstructed and reassigned after a failure. Spark needs many resources, hence memory requirements may be an issue to run on specific cluster configurations and can interfere with resource usage of different applications running on the same cluster.

Spark Structured Streaming. Starting from its version 2.0, Apache Spark has introduced Structured Streaming [47], a new streaming model in Apache Spark environment. "Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine" [48], it reformulates the way Spark Streaming deals with stream data by allowing to interact with it as it was batch data: the engine will manage *Dataset*s through a set of APIs, being able to express streaming aggregations as well as computations on event-time windows. Data streams are considered as unbounded tables which grow by continuously appending new elements (new rows), hence it can be considered as a standard table on which performing usual operations leaving the engine to manage stream in its details such as state management, event-time based aggregations, triggers, which are, instead, left completely to the developer in Spark Streaming.

3.2.2 Inter-cluster communication

Second thesis project requirement to build up a decentralized architecture is a communication mechanism allowing clusters to exchange information about local processing and to permit a further aggregation or processing at system wide level.

Different solutions have been considered for the goal, they are different for being:

- complete stream processing frameworks employed to further process data on different clusters
- tools which expose cluster data to external network to be further gathered and then processed

Apache Spark. Spark Streaming has been introduced as alternative to be adopted as stream processing framework at cluster level. Spark and its built-in libraries can be used to read local clusters information from a distributed file system or a messaging system, process gathered data and compute a final result for the whole system.

Apache Ignite. Apache Ignite [49] is an in-memory computing platform providing consistency and availability for an in-memory distributed key-value store. Ignite uses memory as a fully functional storage, whereas common databases use memory only as a caching layer. Moreover Ignite employs a scalable approach called *collocated processing* which allows to "execute advanced logic or distributed SQL with JOINs exactly where the data is stored avoiding expensive serialization and network trips" [50].

For thesis goal, Ignite has data ingestion and streaming capabilities achieved first by equally distributing data across all Ignite nodes and then allowing processing in a collocated manner. It is also possible to run SQL queries and to subscribe to continuous queries with notifications when data changes.

Apache Toree. Toree [51] proposes itself as the solution to enable interactive applications against Apache Spark. This means that by using Apache Toree, underlying clusters should use Apache Spark as platform to process data streams. The solution is client-server based: Apache Toree Server is one endpoint of a communication channel remotely reachable by Apache Toree Client that talks to the server with RPC-like interaction. A use case of such communication is a client sending snippets of raw code to server, such as adding a JAR to Spark execution context or to execute shell commands.

For thesis purpose, Toree can be integrated in an application which interacts with a server for each cluster of which the system it is composed and to extract data from Spark contexts to remotely compute an overall result.

Apache Toree was not tested out, but supposed complexity in implementation gives priority to other solutions.

NiFi. "Put simply NiFi was built to automate the flow of data between systems" [52]. As described on the official NiFi documentation, NiFi can run within a cluster and each "node in a NiFi cluster performs the same tasks on the data, but each operates on a

different set of data. Apache ZooKeeper elects a single node as the Cluster Coordinator, and failover is handled automatically by ZooKeeper. All cluster nodes report heartbeat and status information to the Cluster Coordinator. The Cluster Coordinator is responsible for disconnecting and connecting nodes. Additionally, every cluster has one Primary Node, also elected by ZooKeeper. [...] Any change you make is replicated to all nodes in the cluster, allowing for multiple entry points." [52].

The description depicts well enough key features of NiFi as well as important aspects to take into account in realizing architecture within this thesis project: first, per cluster Zookeeper, necessary for clusters robustness, then the idea of Primary Node and its re-election, both of them will be taken as further requirements discussed in following sections to make architecture fault tolerant and safely decentralized.

Even if potentially remarkable in its features, NiFi risks to put much stress on communication between cluster nodes, that is one of main motivations and then requirements for thesis work.

Livy. Livy [53] is an Apache Incubator project and, as Apache Toree, it acts like an interface to interact with Spark cluster. It provides a mechanism to send snippets of Spark code, to retrieve results synchronously and asynchronously and to submit Spark jobs over a REST interface or through a RPC-like client library.

Livy can be integrated as well in an application which communicates with several clusters over proper interfaces, but other ready-to-deploy solutions take priority over it.

Apache Kafka. Kafka [54] is a distributed streaming platform which allows to publish or subscribe to streams of records, to store records providing fault-tolerance as well as process streams of records [54]. Kafka runs as a cluster composed by servers called *brokers*. Brokers retain records that are categorized in *topics*: each topic is partitioned among different brokers to provide fault tolerance and can have different consumers as well as different producers. A remarkable feature of Kafka is that it can act as a fault-tolerant storage because records published on a topic are written on a disk and then replicated. Kafka is a valuable and spread solution to provide fault tolerance for data streams and to be integrated in stream processing applications because of its features and flexibility.

3.2.3 Selected tools

Above brief survey has highlighted several tools that have been considered as potential building blocks for an effective decentralized architecture for stream processing.

Some of them, such as Toree and Livy, have been discarded because they need ad-hoc implementations and a further analysis must be done to design a functional and efficient

environment. Hence the choice drifts on already developed frameworks: between Apache Spark and Apache Ignite the choice was the former, first because of its flexibility, being able to integrate stream, batch, graph processing and even machine learning algorithms by using built-in libraries, and on the other hand because, although pure-stream processing frameworks perform better for low latency streaming applications, there is no strong constraint for the first phase of thesis work, hence flexibility and integrability are preferred and Apache Spark was elected to be adopted in following analysis and implementations.

Even if NiFi was discarded because of the same motivation of Toree and Livy, it is built on architectural concepts which are useful to be taken into consideration for some specific features of thesis architectural analysis and for further improvements.

Apache Ignite, although promising and interesting, is not covered in this thesis project and left for future works.

3.2.3.1 Apache Spark

As stream processing framework we opted for Apache Spark because of its possibility to seamlessly use different tools in order to potentially integrate different big data problems, like stream processing with batch processing, as well as stream processing with graph processing to achieve a higher level in developing streaming applications.

Spark Streaming. In first place the analysis was made over Spark Streaming to switch then to Structured Streaming. Spark Streaming is based on *DStreams* (Discretized Streams), a "continuous stream of data, either the input data stream received from source, or the processed data stream generated by transforming the input stream. Internally, a DStream is represented by a continuous series of *RDD* [7], which is Spark's abstraction of an immutable, distributed dataset" [55].

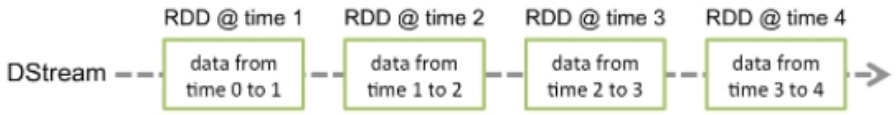

Figure 3.1: RDDs in DStream composition [55].

Figure 3.2 shows how a DStream is elaborated by applying the same kind of transformation on all RDDs of which DStream is made. This offers the possibility to achieve a very high control of each part of the stream of data, but, meanwhile, leaves the programmer with the task to maintain every aspect about data management at a relatively low level of programming. This is an efficient approach in terms of time and effort to develop a given application, but a little more tricky when there is the need to read data

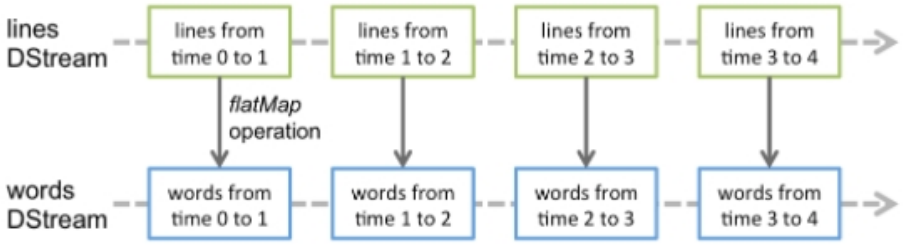

Figure 3.2: DStream transformation example: *flatMap* transformation on a DStream [55].

from a specific source, like Kafka, or maintain a state which lasts between different transformation of the stream. For the thesis purpose, in particular, it was found not trivial to maintain the state for computing result of chosen algorithm over the data stream because of it complexity. Therefore an other possible solution was explored among the ones taken into consideration.

Structured Streaming. "Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. You can express your streaming computation the same way you would express a batch computation on static data. The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive [48]."

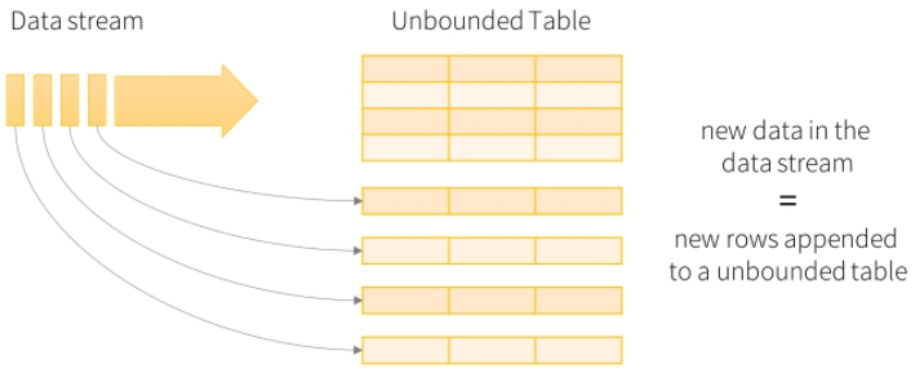

Figure 3.3: Data stream as an unbounded table [48].

The advantage of Structured Streaming approach is that programmer needs only to express how to modify incoming data with a SQL-like semantic and it offers many other useful features from a programming and designing point of view, that, though they can be reached in Spark Streaming as well, they take much less effort.

1. When dealing with streaming sources there is the related concept of offsets: the application must keep track of which records or tuples are read from which source,

mainly because it can employ recovery procedures in case of failures. Structured Streaming takes care of offsets of records being processed thanks to checkpointing and write ahead log without programmer intervention.

2. An important feature for streaming frameworks is handling late data, particularly when, and it is the case, there are windowed computation over incoming data streams. Late tuples are data whose event-time falls out of current reference window and, though it comes later, it should be counted when processing tuples which belong to that window. In the case of Spark Structured Streaming, the framework reads late data and simply applies computation of that data only on competence window.

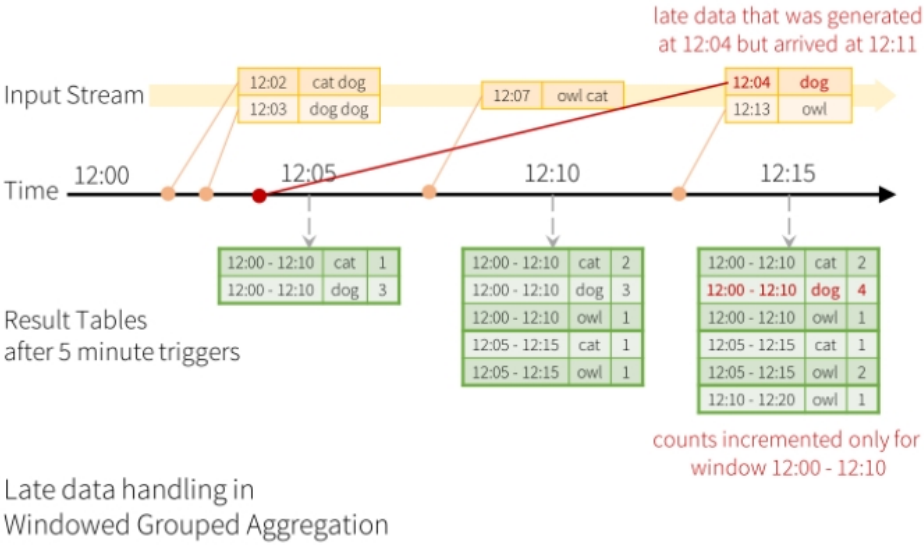

Figure 3.4: Late data handling [48].

Figure 3.4 shows an example of windowed computation where window is 10 minutes length with a sliding time of 5 minutes: each time a computation is triggered (every 5 minutes) incoming data is appended to an unbounded table and when late data comes into the system, thanks to its event time, it can be appended and update the right table.

3. Since the unbounded table can grow indefinitely, *watermarking* is provided as well: it allows late data to update in-memory state only if its event-time is above a specified threshold, in other words, when an aggregate for a window is too old, that aggregate is dropped and no more updated even if a late data for that window arrives.

   In figure 3.5, for example, while tuple $(12:09, cat)$ is accepted and will update the correspondent window aggregate because its window reference is still alive having maximum event-time behind the watermark, the tuple $(12:04, donkey)$ is not

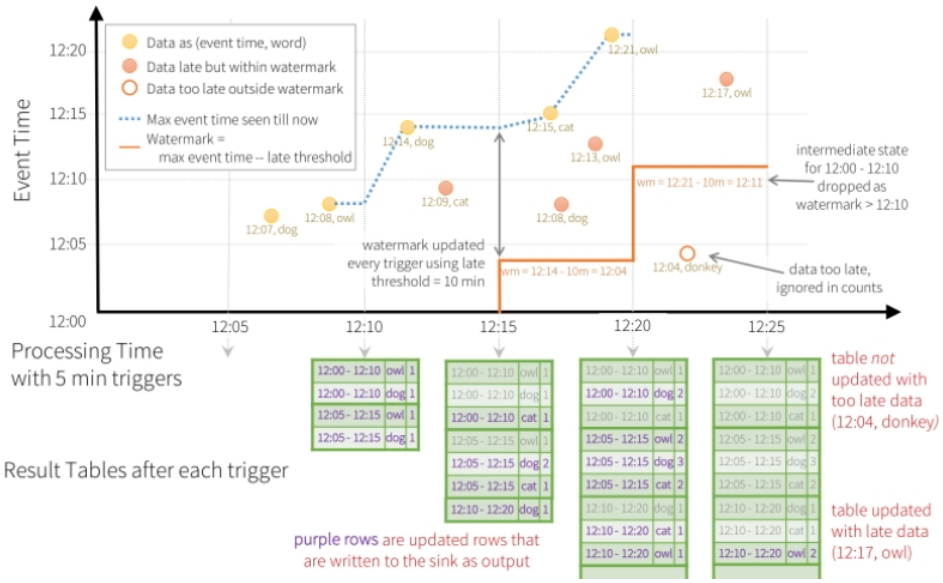

Figure 3.5: Watermarking [48].

accepted because its reference window has been dropped since its maximum event time falls behind the watermark.

4. Structured Streaming is run over Spark SQL [56] engine which works well with structured data and provides the capability of registering User Defined Functions (UDFs) and User Defined Aggregate Functions (UDAFs) which greatly increased flexibility of data transformation and computation. These functions can be used as they were running on a relational table because in Structured Streaming every record is considered as a *Row* composed by fields of different data type (String, Integer, Timestamp, etc...). A UDF processes every incoming record and modifies one or more of its fields by applying a user defined function.

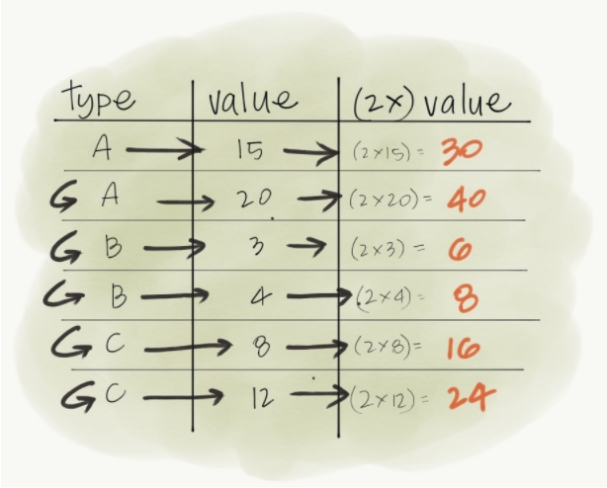

Figure 3.6: UDF example [57].

Similarly, UDAF processes every incoming records, but it maintains an aggregate

which is updated as records pass by. Concept behind UDAFs is the same as that of standard aggregate functions in SQL language, which allows, for example, to compute sum, average or simply records count after grouping records.

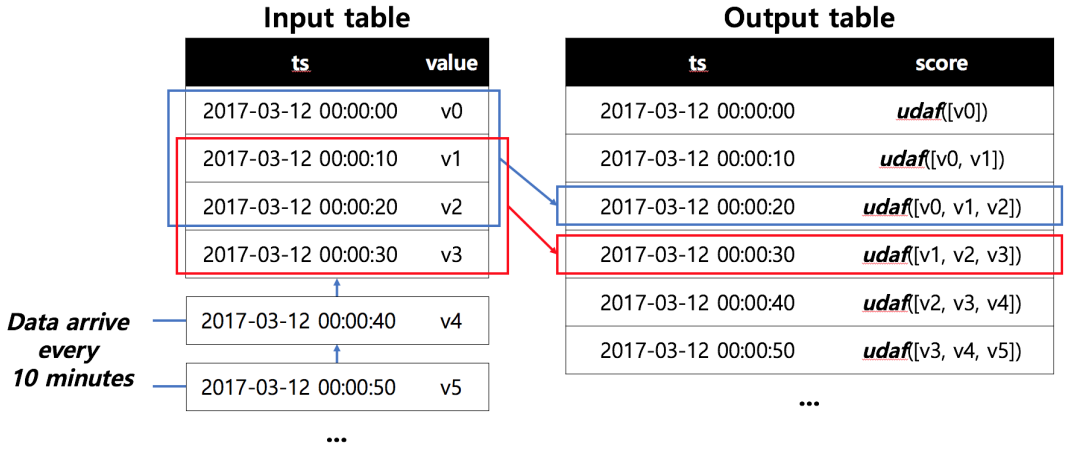

Figure 3.7: UDAF example [58].

3.2.3.2 Apache Kafka

In order to create an inter-cluster communication mechanism, Apache Kafka [54] was chosen because it offers all features to be part of the environment of this thesis work. Kafka is basically a messaging system which is able to store stream of records in a fault tolerant way. Moreover, Kafka is very supported from the community and offers easy integration with almost all stream processing frameworks, including Spark and Structured Streaming.

Kafka basic concepts are [54]:

1. It runs as a cluster on one or more servers.
2. The Kafka cluster stores streams of records in categories called topics.
3. Each record consists of a key, a value, and a timestamp.

"A topic is a category or feed name to which records are published" [54]. A topic can have from zero to multiple subscribers: when no one is subscribed to a topic, Kafka acts in practice as a distributed data storage. Each topic is partitioned, as showed in figure 3.8, and "each partition is an ordered, immutable sequence of records that is continually appended to a structured commit log". In this way records can be easily distributed because partitions are spread on different servers in the same Kafka cluster in order to achieve fault tolerance.

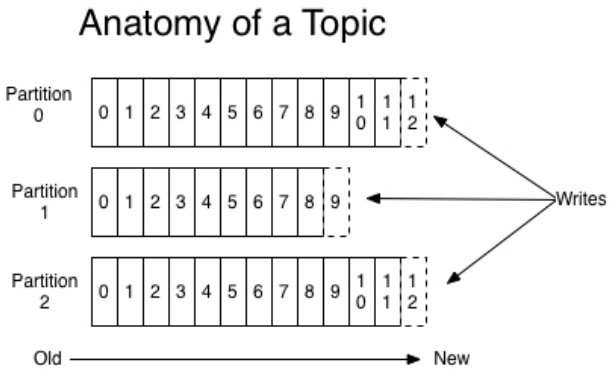

Figure 3.8: Anatomy of a topic: partitions of a topic [54].

A partition can be read from different consumers thanks to the *offset* which is an *ID* assigned to every records: an ID is maintained by each consumer to keep track of which records have been read before (fig. 3.9). At the end, Kafka cluster looks like figure 3.10: different servers retain different partition of the same topic and different consumers, that can be part of a consumer group, which identifies them as part of the same category of consumers, can read from different partitions on different servers.

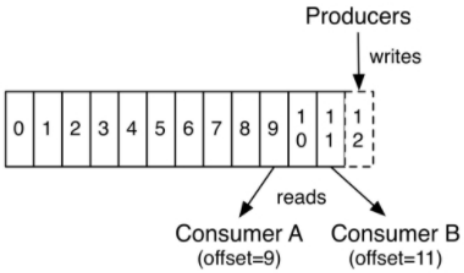

Figure 3.9: Where consumers read and where producers write [54].

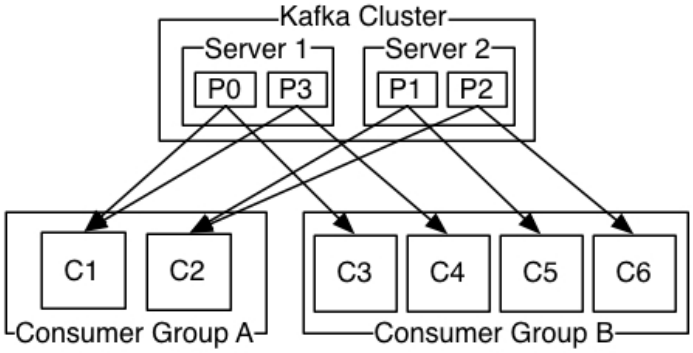

Figure 3.10: Example of how a Kafka cluster is made [54].

A fundamental actor in a Kafka cluster is the *Zookeeper*: "is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services" [59]. It has information about data distribution on cluster

nodes and topic consumers and their offset. In practice it is responsible for managing topics and related information and metadata, and when a consumer asks to read a topic, Zookeeper provides information about to where to find data of requested topic. Hence each Kafka cluster needs a Zookeper and its architecture looks like figure 3.11.

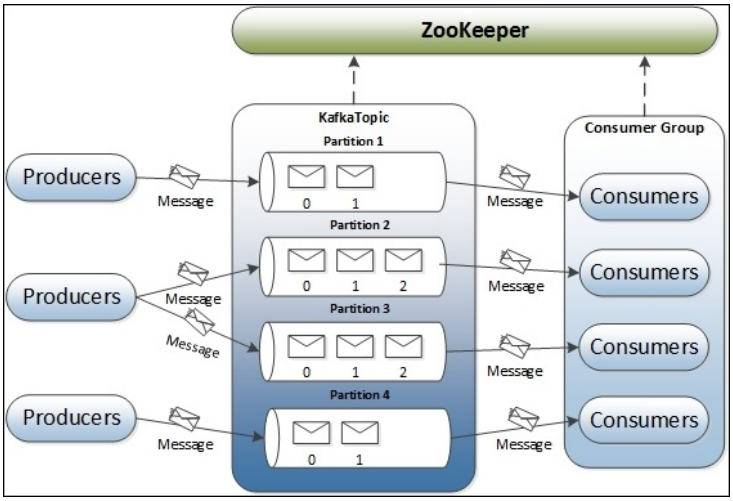

Figure 3.11: Kafka cluster including producers, consumers, topics and zookeeper [60].

3.3 Architectures

In this section different architecture configurations are described in their general settings and in the placement of tools selected in previous section. These configurations will be employed as infrastructure where to execute algorithms described in section 4 in order to perform an evaluation of those algorithms on different configurations.

Fundamental unit for the whole structure is the *Spark cluster* where a Structured Streaming application is running. A Spark cluster is an autonomous entity which is able to run a stream processing application and compute an aggregate on its own. Such a cluster is in direct contact with real data which is generated and published on a *cluster level topic* and send output records to an output topic.

3.3.1 Centralized configuration

Centralized configuration consists of a Spark cluster running on a single machine. Input and output data are placed in two topics of two different Kafka clusters whose brokers are also in the same machine of Spark cluster.

A given algorithm runs in a Structured Streaming application and reads input data from a topic, executes a window-based aggregation and computation and then sends output records to the output topic. Everything happens in the same machine from ingestion

of input data to storage of final output by employing available resources in the machine. The framework still runs different executors and workers, starts jobs and executes tasks, but it is limited by available resources which can prevent actual processing parallelization of input data.

A centralized configuration looks like figure 3.12. Data is produced on a *cluster level topic*, Structured Streaming application running in a Spark cluster reads a stream of record published on the cluster level topic and sends output records to a different topic. All topics are inside the same machine.

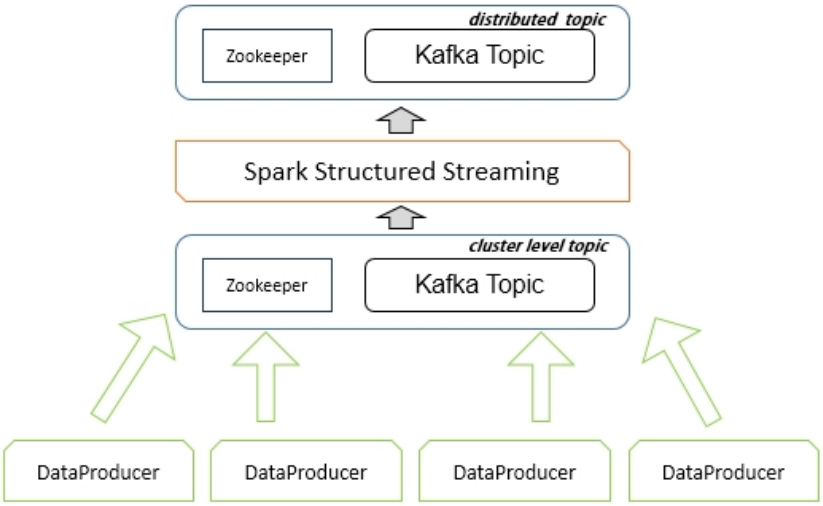

Figure 3.12: Spark Cluster and its composition

3.3.2 Distributed configuration

Distributed configuration is similar to the centralized configuration with some differences:

- Spark Cluster is running on at least two different machines
- There are two Kafka clusters, one with input topic and another one with output topic. These clusters cover all machines with at least a Kafka broker each.

Computation is distributed because workers are placed on different machines and Spark can take advantage of resources of all of them by dividing the workload.

Kafka cluster is actually distributed since brokers are also on different machines. That would be important when brokers are geographically distributed because producers can produce data on the closest broker server to avoid significant data movement. Indeed this does not help in reducing data movement because data is still read from input topic by Structured Streaming application generating in the same way a flow of data towards the distributed Spark cluster.

3.3.3 Decentralized configuration

Designed decentralized architecture consists of two levels. First level is made of a set of autonomous clusters in a distributed configuration: each cluster reads input data and publishes *first level aggregates* to a *distributed topic*, similarly as shown in fig. 3.12. Second level is made of a different Spark cluster where a Structured Streaming application subscribes to the same topic on which all clusters publish data: first level aggregates are reconciled into one or more *global aggregates* and sent out to a different Kafka topic to be stored or further analyzed.

Each cluster does not read all input data. A cluster reads a small part of the entire input data, possibly that part(s) whose producers are close to the cluster itself. This allows to reduce the cluster workload because it should process reduced input data set.

Decentralized configuration is showed in fig 3.13. Each cluster has its own Kafka cluster with a zookeeper and one or more broker servers. Distributed topic where first level aggregates are published is part of a different Kafka cluster which is distributed to achieve fault tolerance, since Kafka is used also as a storage system. Topic where global aggregates are published is distributed as well for the same reason as first level topic.

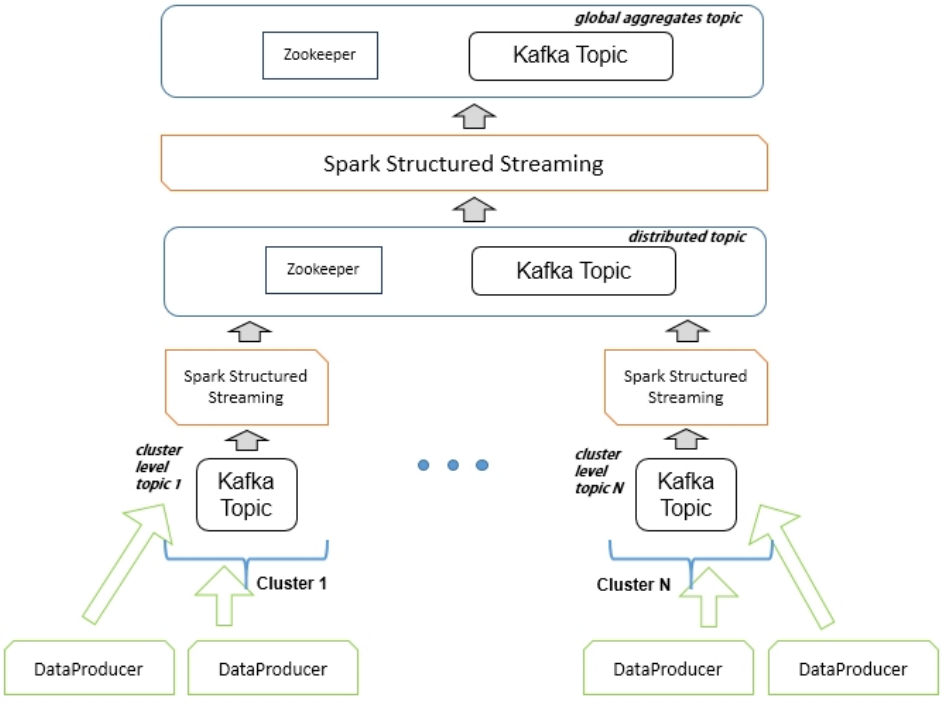

Figure 3.13: Global configuration

Chapter 4

Methods and algorithms

4.1 Query

In this work, queries taken into account are *standing queries* because:

- they are well suited for streaming data which is continuously flowing into the system and it can be mission critical monitoring particular aggregates over it or computing sensitive statistics for describing its composition, features and other metrics
- there is great support in current streaming processing systems in long running standing queries, which can run for days or week continuously executing a given query over incoming data
- standing queries support is very flexible, specifically within chosen Structured Streaming environment, because it allows to expose aggregates and/or metrics to run ad-hoc queries over them. It adds one more feature which can be useful in particular situations when it is needed to query data for better insights.

4.2 Methods

Since we are talking of data whose main feature is to be streaming and high volume, methods employed are:

- *data synopsis* for each time window, describing data seen inside it
- *windowing*, in particular sliding windows

4.2.1 Data synopsis

First method is critical in the whole thesis project because of the motivation which stands behind it: since we want to reduce bandwidth costs, sending all incoming data to a

single place it is not an optimal solution. To avoid this, a synopsis can be computed as intermediate result to be sent to the reconciliation place. Hence, in order to prevent all incoming data to move from their sources to the place where the global aggregate is computed, each cluster acts as a first level of aggregation by computing a summary of incoming data. This summary is then sent to a further application which is responsible to read summaries coming from all clusters and reconcile them to build up a global summary.

4.2.2 Windowing

Windowing allows to execute a standing query only on a part of the entire stream, reducing in this way the amount of data needed to compute an aggregate and/or a particular value. This is particularly important for the application in direct contact with high volume data which is running inside each cluster: without windowing, to execute any algorithm on incoming data it is required to store all data received so far and then run it by parsing all records which entered the system. This is not feasible because of resource limitation of actual computing systems which have limited resources for computing but most of all limited storage capabilities which are not able to store an unlimited stream of data running for weeks or months.

In order to functionally apply windowing method, both summaries, the ones produced by first level and by second level of aggregation, should refer to a time window. These summaries are dependent:

- summaries in second level of aggregation should be computed with respect to a time window which is at least as large as the largest time window computed in first level of aggregation
- summaries in first level of aggregation should be multiple between themselves and be submultiple of time window in the second level of aggregation

For example, if second level summaries refers to a time window of 60 seconds, first level summaries should refer to a time window of a maximum of 60 seconds. Meanwhile, other first level summaries could refer to a time window of 30 seconds, 20 seconds and so on, being this time range submultiple of time range in second level summaries. This constraint allows first level summaries to fit the time window in second level summary, without worrying about first level summaries falling out of the second level time range.

[Image of summaries and/or windows]

4.3 Algorithms

Before choosing which algorithms to implement and run, an analysis has been done to understand what features and properties an algorithm should have to be decentralized.

4.3.1 Decentralizable algorithm

The analysis starts from an assumption which is derived from the designed architecture, as it was explained in Chapter 3 (see 3.13).

Streaming data coming from different sources is a continuous flow of records entering the system for which *ingestion time* can be different and later than *event time*. This means that data enters the system without specific order (with respect to event time) and that application logic should be aware of this. Since incoming data stream in the first level of aggregation is made of simple data, this issue is easily managed by Structured Streaming because of its capability to handle late data, as most of currently used streaming processing frameworks are able to do. Nothing more is then required here.

Second level of aggregation is really similar in its composition to the first one, but incoming data is not simple data as before, but a summary which represents all data which entered a specific cluster. Even though here late data is still handled by the framework (a summary is still normal data), in order to merge different summaries, they should have specific properties. Summary produced by first level aggregation should have same properties of summaries which will be read by second level aggregation to create a final summary.

4.3.1.1 Summary features and properties

To employ *windowing* method in every part of the architecture, each summary should refer to a time range, which actually corresponds to a *time window*. This is necessary because the merging application needs a time reference for incoming data in order to select only those summaries which fall inside the time range for its own summary computation.

Another important characteristic that a summary should have to be mergeable is satisfying *commutative* and *associative* properties. Considering input data sets D_i which produces a summaries S_i, being "+" the merging operation between two different summaries:

- *commutative* property is satisfied if $S_1 + S_2$ and $S_2 + S_1$ will produce the same merged summary S_{12}. This means that does not matter the order in merging operation because the resulting summary will be the same. Such property is necessary since, as said above in other words, order is not preserved: summaries are generated far

from each other and can reach merging application with different delays due to distance from the source, latency of transmission, etc...

- *associative* property is satisfied if $(S_1 + S_2) + S_3$ and $S_1 + (S_2 + S_3)$ will produce the same merged summary S_{123}. This means that, dealing with N different clusters, each of which produces a summary, the merging application will receive N different summaries S_i $(i = 1...N)$. In this case the application should be able to merge two summaries at a time, produce a summary, take the next one and proceed merging until all summaries are finished: final summary should be the same regardless arriving order of summaries and merging order.

This properties easily apply when the summary is a simple aggregate like sum, average, count and many others because they trivially has both commutative and associative properties. But summaries can also be more complex, being composed by different simple aggregate values or even arrays as long as they can be decomposed, elaborated in every parts by applying a specific merging procedure and then composed back as a completely built summary as the input ones.

4.3.2 Non decentralizable algorithms

Decentralization should be done only after an accurate analysis of algorithm itself and of expected input and output data size. In fact, decentralization as depicted in this thesis project comes to high costs and should be avoided if:

- an algorithm needs to pass multiple times input data. Second level aggregate relies only on which is reported inside summaries, if a second level summary needs input data the algorithm cannot be optimally decentralized, unless other methods of summary representation are provided.
- output size of first level aggregates is comparable with input data size: output data size should be at least one order of magnitude smaller. In other words, the summary should really summarize input data or make it smaller in some way and not only transform it without reducing its size.
- applying merging procedure increases overall summary error. This means that second level summary should still maintain an error comparable with the ones in first level summary
- the actual costs and effort to decentralize an algorithm fall behind the real advantages which can come after decentralization. This means that fundamental motivation for decentralization should be the first priority: costs to decentralize, in all aspects, should be less than bandwidth costs saved after decentralization

4.4 Chosen algorithms

Considering algorithms features and requirements defined in previous section, first a simple algorithm has been chosen to test feasibility of implementation of such algorithm on the designed architecture, then a more complex algorithm has been implemented in a decentralized fashion.

First algorithm is the computation of average and it was chosen because it is based only on mathematical operations, then it is suitable to test on first hand framework flexibility and possibilities without make use of extended framework features. The second one is Misra-Gries algorithm, which was mentioned in a previous section. It was chosen because it deals with the known problem of computing objects frequency, a hot topic in many applications, and it is based on a fixed-size summary, which is a way to define indirectly size of output data and, hence, of data movement.

In both cases, input data is similar except that for type of values: numbers for first algorithm and words of text for the second one. Each input record has a timestamp in it, which represent its event-time.

4.4.1 Computation of average

To compute a decentralized average two steps are necessary.

Inside each cluster:

1. input data is grouped according to its event-time to assign each record to one or more time windows
2. records inside each time window are aggregated: their *average* is calculated
3. for each time window a summary is created. A summary is made by *time window*, *average* and *cluster id*, which is a unique identifier inside the cluster to identify who produced that summary
4. every summary is sent as input to the *merging application*

Inside merging application:

1. input summaries are grouped according to start time of time window which they refer to
2. records inside each time window are aggregated: *aggregated average* is computed as average of the averages of all summaries in that time window

3. for each time window a summary is created. A summary is made by a *time window*, which is a multiple in duration of all summaries which contributed to that time window, and the *aggregated average*

4. every summary is sent out of the system to be stored or analyzed

4.4.2 Misra-Gries algorithm

Misra-Gries algorithm [30] generates a set of k $(key, counter)$ pairs. This set is a summary which is built according to the following steps. For each incoming record x:

1. if x is already contained in the summary, its counter is incremented

2. if x is not contained in the summary and the size of summary is less then k, then x is added to the summary with counter equal to 1

3. if x is not contained in the summary and the size of summary is greater or equal than k, then x is discarded and all counters in the summary are decreased by 1

Algorithm provided in [30] is built to create a single summary which contains at most k items starting from a stream of incoming records. Such a summary is the output of the computation which happens inside each cluster and the input for the merging application.

Merging application reads a stream of incoming records which are made of a time window and its related summary. Merging application receives at least one record from each cluster. Each record is the information about frequency of keys within that time window in a certain cluster. The computation is slightly different because input records are summaries themselves and it follows algorithm provided in [61]. For each incoming summary S_i, a global summary S is maintained:

1. input summary S_i is combined with summary S: counters of same keys are summed up to generate a single summary

2. $(k-1)$-th largest counter in the summary is subtracted from all counters

3. non-positive counters are removed

To compute the global summary, the algorithms above are used in different places over the overall architecture. First step which is executed *inside each cluster* makes use of standard Misra-Gries algorithm:

1. input data is grouped according to its event-time to assign each record to one or more time windows

2. records inside each time window are aggregated into a single summary built by applying Misra-Gries algorithm

3. every summary is sent as input to the *merging application*

Second step is executed in the *merging application* and it makes use of algorithm provided in [61]:

1. input records are grouped according to start time of time window to assign each summary to one or more time windows

2. summaries inside each time window are aggregated into a single summary built by applying the merging procedure provided in [61]

3. every global summary is sent out as output

Chapter 5

Implementation

5.1 Kafka as publish-subscribe and storage system

Kafka plays an important role in the implementation aspect. First, being a publish-subscribe system, it enables streaming capabilities of applications located at all levels in the architecture. Publication of data on a topic is seen as a new data coming into the system at any level, subscription to a topic corresponds to reading all data that are published on that topic and react consequently.

Kafka acts also as storage system, providing robustness and availability of data. Every topic where data is published stores input data or summaries in a specific format. Records inside topics are written following JSON format which "defines a small set of formatting rules for the portable representation of structured data" [62]. Key idea in this aspect is *portability* because in this way any kind of device or application can publish data on a given topic provided that it contains coherent data fields for that topic.

Input data from sources contains three fields: $created_at$, the event-time, *value*, the objective value for aggregation, which can be a number or a word of text, and $producer_id$, an id which identifies producer of this record.

A first level summary and a global summary contain instead more fields according to computed aggregates which will be discussed in next sections.

[Data workflow]

5.2 Query and methods

Standing query is obtained by defining in Scala language a sequence of operations to be performed on input data which may enter the system. Scala is used to create a *streaming query* on which apply operations to transform the stream or to save the stream to a specific output. All operations in their entirety represent a standing query which take

care of running the same on all input data.

Windowing is performed by applying a specific Scala API called *window* which automatically groups all incoming records in windows. Window API groups input data according to a given timestamp. In our case, all input records always have a JSON data field which contains the *event-time* (for example, see data field *created_at* in previous section). Time passed as parameter to *window* function is used to decide to which window append that record.

Structured Streaming builds a streaming DataFrame starting from input data. A streaming DataFrame is seen as a continuously growing relational database to which new data is appended (see Chapter 3, page 19, fig. 3.3) and on which apply transformations. Semi-structured data in JSON format is well suited to be transformed in a streaming DataFrame because it is defined by data fields from which infer a database schema. This analogy, as with CSV format [63] for example, allows to treat JSON (or CSV) data fields as columns in a relational database and to provide a uniform manipulation of data coming from different sources.

5.3 Decentralized average computation

5.3.1 First level of aggregation

This application reads as input stream a sequence of records. Each record contains, besides the event-time, an integer number. Both of them are values of particular fields in the JSON data structure.

To compute first level summary, input records are grouped according to their *event-time*. Then, the average of input *value*s is computed by exploiting a built-in aggregate function which computes the average of values in each group. Resulting summary is composed by a time window and the average of all numbers whose event-time is included in that time window. Id of cluster is also added to identify who has produced this summary. Snippet 5.4 shows piece of code to get summaries.

Listing 5.1: Grouping input records by *event-time*. Computation of average value and first level aggregate by using built-in aggregate function. *lit* function is needed to pass a literal value to *withColumn*

```
val summaries = records
  .groupBy(window($"created_at", windowLength, slideInterval))
  .agg(avg("value") as "average")
  .withColumn("clusterId", lit(clusterId))
```

5.3.2 Second level of aggregation

Second level aggregation reads one or more summaries for each cluster. Input records are summaries, still in JSON format, which are exactly summaries generated in the first level of aggregation. They are made of a *time window*, the *average* for that time window and a *cluster id*.

This application reads input data and groups records by the starting time of time window in the record itself. After grouping, summaries in each group are aggregated by computing the average of the averages in each record. The same aggregate function as in the first level of aggregation is used. Snippet of code is in **??**).

Listing 5.2: Grouping summaries coming from clusters in time windows. Reference time to build windows is the start time of the time window inside each record. *Avg* aggregate function computes average of values in the field passed as parameter. This means that it computes the average of the averages previously computed in each cluster.

```
val globalSummaries = summariesFromClusters
  .groupBy(window($"windowStart", windowLength, slideInterval))
  .agg(avg("average"))
```

5.4 Decentralized Misra-Gries algorithm

Key role in building a summary by applying Misra-Gries algorithm is *User Defined Aggregate Function* (UDAF). Structured Streaming provides built-in aggregate functions like sum, count, min, max, etc. (which are used in decentralized computation of average) but also the possibility to implement custom functions to aggregate records after being grouped.

Implementing a UDAF requires creation of a class (*UserDefinedAggregateFunction*) and definition of its members. Most important members to define are *input*, *output* and *buffer schemas*, and *update* with *evaluate* functions. Input schema is the schema that is passed as input to UDAF, buffer schema is the schema of buffer which is maintained during aggregation, update function is called for each record to update the buffer, and evaluate function is called at the end of aggregation to prepare output data which should have the schema defined in the output schema.

5.4.1 First level of aggregation

Misra-Gries algorithm is applied in its standard version in the first level of aggregation. Input records are grouped according to their event-time. Records grouped in each time window are then grouped according to a UDAF which takes care of extracting a summary by applying Misra-Gries algorithm. Sliding window is not specified here because *tumbling* window method is chosen: in this way summaries do not overlap in time, allowing second level of aggregation to easily merge summaries. This is better explained in next section.

Listing 5.3: Grouping input records by *event-time*. Computation of summary by applying Misra-Gries algorithm in the UDAF.

```
val summaries = records
  .groupBy(window($"created_at", windowLength))
  .agg(summary($"value").as("summary"))
```

Final records have a time window, the size of the summary and the summary itself. The id of cluster is added as further column to specify which cluster generates the record. Everything is then packed up in a single JSON object before being sent out to the output topic

5.4.1.1 UDAF - Create Summary

- *Input schema.* It is a string corresponding to the word inside each record.
- *Buffer schema.* It is a Map made of $(word, frequency)$ pairs
- *Output schema.* It is JSON representation of the summary made of the number of elements inside the summary, and the summary itself, made of the JSON representation of pairs.
- *Update function.* Here Misra-Gries algorithm is applied. Values in the map is updated by adding elements, increasing, decreasing and pruning when needed.
- *Evaluate function.* Summary is prepared to be outputted. Output is a JSON string created starting from a JSON object containing summary size and the summary itself.

5.4.2 Second level of aggregation

In the merging application input records are the output records of the first level of aggregation. It contains a time window, the size of summary, the summary itself and the id of cluster which generates the summary.

Input records are grouped according to start time of time window in each record. Records in each group are aggregated according to a UDAF which generates the summary out of the input records. UDAF in this application follows algorithm in [61].

Listing 5.4: Grouping input records by *event-time*. Computation of summary by applying Misra-Gries algorithm in the UDAF.

```
val globalSummaries = summariesFromClusters
  .groupBy(window($"windowStart", windowLength, slidingInterval))
   .agg(decentralizedSummary($"summary", $"window",
      $"summaryWindow") as "mergedSummary")
```

5.4.2.1 UDAF - Merge Summaries

Input records are summaries which need to be merged. Summaries coming from the same cluster will refer to non overlapping time windows in order to simply merge two summaries in the UDAF following given algorithm.

If summaries were overlapping, it would be necessary to take care of identifying which data belongs to more than one summary and count it only once. This would have added complexity to the application because of the necessity of raw input data to distinguish which record count only once in every window. This would turn away from the primary concern of the whole architecture because data should flow from input topics directly to the merging application causing big data to move around the network.

Since windowing is applied in the merging application, a time window is defined. This window is taken as parameter together with time window in each record in order to select only time windows which fall within the time window defined in the merging application.

- *Input schema.* Made of time window defined in the merging application, time window contained in the record and the summary.
- *Buffer schema.* Made of an array of $(word, frequency)$ pairs, the time window defined in the merging application and an array containing clusters involved in the aggregation to keep track of which cluster contributed to the summary.
- *Output schema.* It is a JSON object containing JSON representation of the time window, the summary and the size of the summary.
- *Update function.* First time this function is called, time window in buffer is set to the time window in the merging application. This time window is used to select which summary aggregate. Algorithm to merge two Misra-Gries summaries is applied. Then the resulting summary is stored back into the buffer.

- *Evaluate function.* Global summary is prepared to be outputted. Output is a JSON object made of time window, summary size and the summary itself.

Chapter 6

Experimental evaluation

Methods and algorithms as they were presented and described in Chapter 4 and 5 are executed on different architectural configurations to obtain evaluation results. Considered architecture configurations are: centralized, distributed and decentralized.

6.1 Metrics

For each configuration we computed three metrics, namely:

1. *Job time*: the time that a job takes to execute the streaming query, from its start to the end.

2. *Input rate* and *processing rate*: the number of input records per second and the number of processed records per second. Against these metrics, number of input rows per trigger is also computed, it is a metrics which refers to the effective number of records which are taken as input in the application.

3. *Input* and *output data size*: the size in terms of bytes of data ingested into the cluster coming from a topic and the size of data produced as result of computation and sent to following topic in the workflow.

6.2 Input data

Input data for average computation is randomly generated by using built-in Scala library to generate random numbers in a range of values and packed into a JSON string with a timestamp attached to it and the id of its producer.

Input data for Misra-Gries algorithm is taken from real Amazon products reviews in JSON format, from which review texts are extracted, split into words each of which packed into a JSON string with a timestamp attached to it and the id of its producer.

Two files have been used, one for machine (*sky1* and *sky2*), whose sizes are, respectively, about 6 GB and about 17 GB. This dataset can be retrieved in [64], and it was used also for the work in [65, 66].

6.3 Output visualization

All metrics are collected by using Graphite [67], which is able to store numeric and time-series data. Data is sent to Graphite by ad-hoc code running on listeners. When a particular event happens, such as start or end of Spark stages or the end of execution of a streaming query, one or more well-formatted records are sent to Graphite server.

Data stored into Graphite database is then used as source by Grafana [68] which allows a better data visualization than Graphite dashboard by enabling simple visualization and useful operations on time-series data. All graphs in this section are taken from Grafana dashboard.

6.4 Setup

When talking of workers, be noticed that, independently from their total number in each Spark cluster, the amount of memory assigned to each of them is 1 GB and the maximum number of assigned cores is 10.

Environment for the centralized configuration consists of a Spark cluster for which 4 workers are all inside the same machine.

Distributed configuration is set up by running a single Spark cluster with a total of 4 workers, 2 on *sky1* machine and 2 on *sky2* machine.

Decentralized configuration is built first by running two different clusters, one on *sky1* and the other one on *sky2* with 4 workers each. Then, because of impossibility to use a third machine where to run a third cluster, Spark cluster running on *sky2* machine is used to execute merging application as well.

6.5 Results

When presenting results for decentralized configuration in both algorithms, metrics and graphs for applications running on single clusters are not shown because they are comparable with the ones gathered for the centralized configuration since they run exactly on centralized clusters.

6.5.1 Average computation

6.5.1.1 Centralized configuration

In figure 6.1, first row represents job time (left) and input and processing rate (right) for centralized configuration. Second row shows input and output data size.

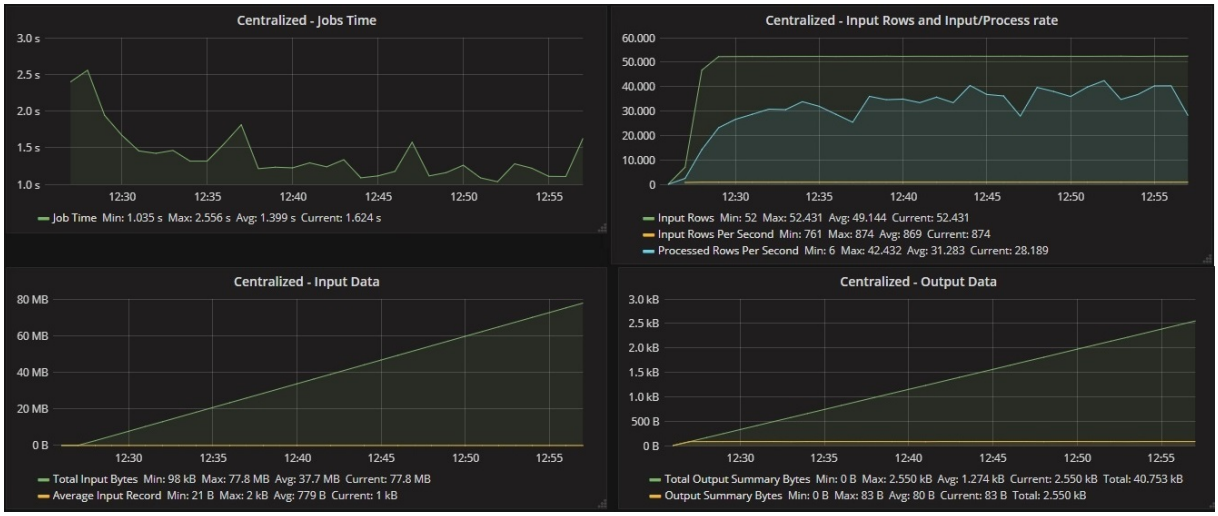

Figure 6.1: Graphs for average computation: centralized configuration.

6.5.1.2 Distributed configuration

In figure 6.2, first row represents job time (left) and input and processing rate (right) for distributed configuration. Second row shows input and output data size.

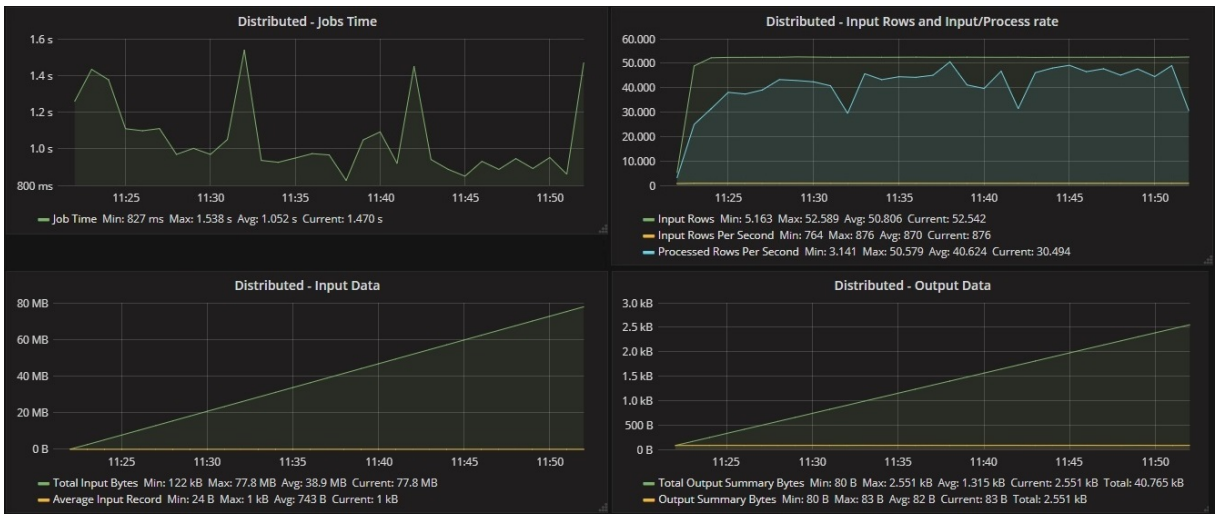

Figure 6.2: Graphs for average computation: centralized configuration.

6.5.1.3 Decentralized configuration

In figure 6.3, first row represents job time (left) and input and processing rate (right) while second row shows input and output data size.

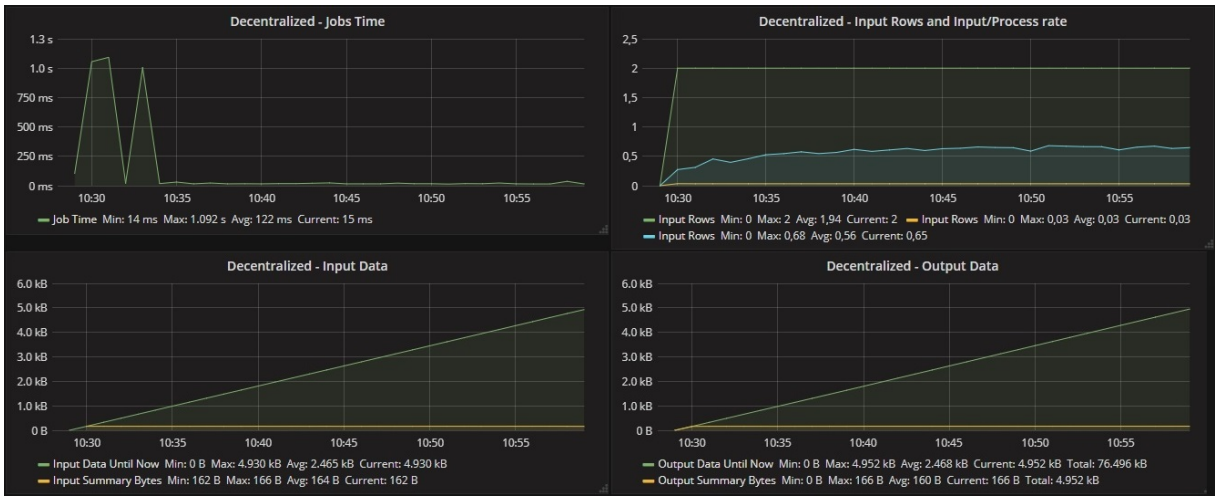

Figure 6.3: Graphs for average computation: decentralized configuration.

6.5.2 Misra-Gries algorithm

6.5.2.1 Centralized configuration

Figure 6.4 shows in the first row the job time (left) and input and processing rate (right) for centralized configuration. In the second row there are input and output data size.

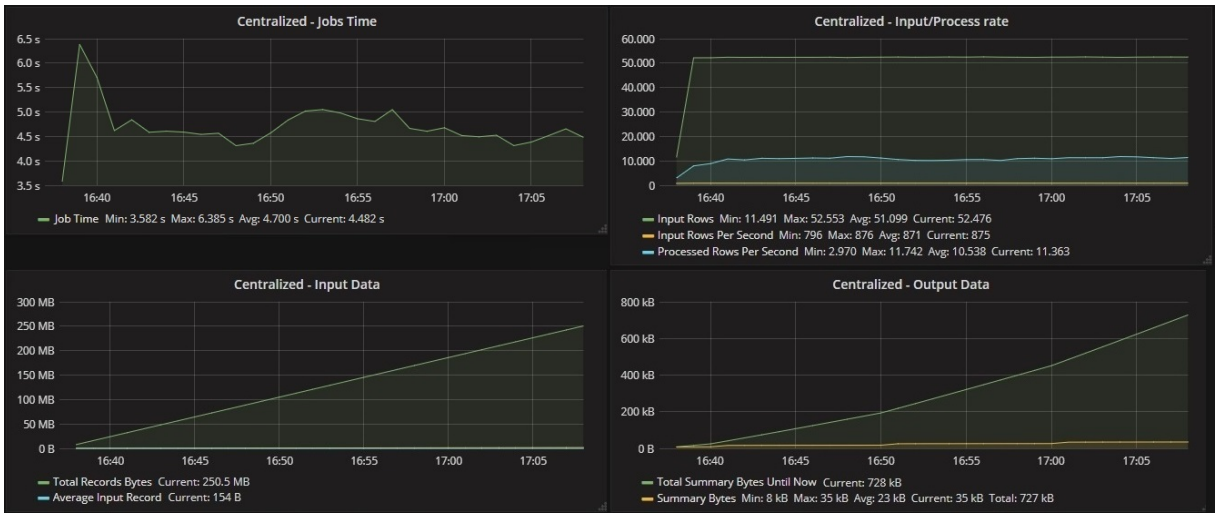

Figure 6.4: Graphs for Misra-Gries algorithm: centralized configuration.

6.5.2.2 Distributed configuration

Figure 6.5 shows in the first row the job time (left) and input and processing rate (right) for distributed configuration. In the second row there are input and output data size.

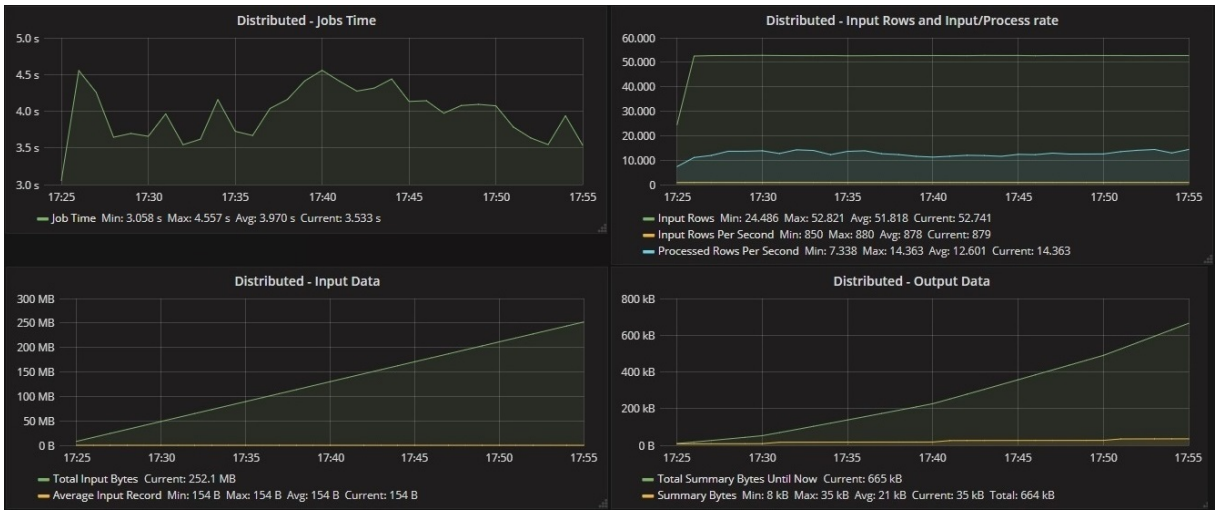

Figure 6.5: Graphs for Misra-Gries algorithm: distributed configuration.

6.5.2.3 Decentralized configuration

In figure 6.6, first row shows job time (left) and processing rate (right) for decentralized configuration. Second row is input and output data size.

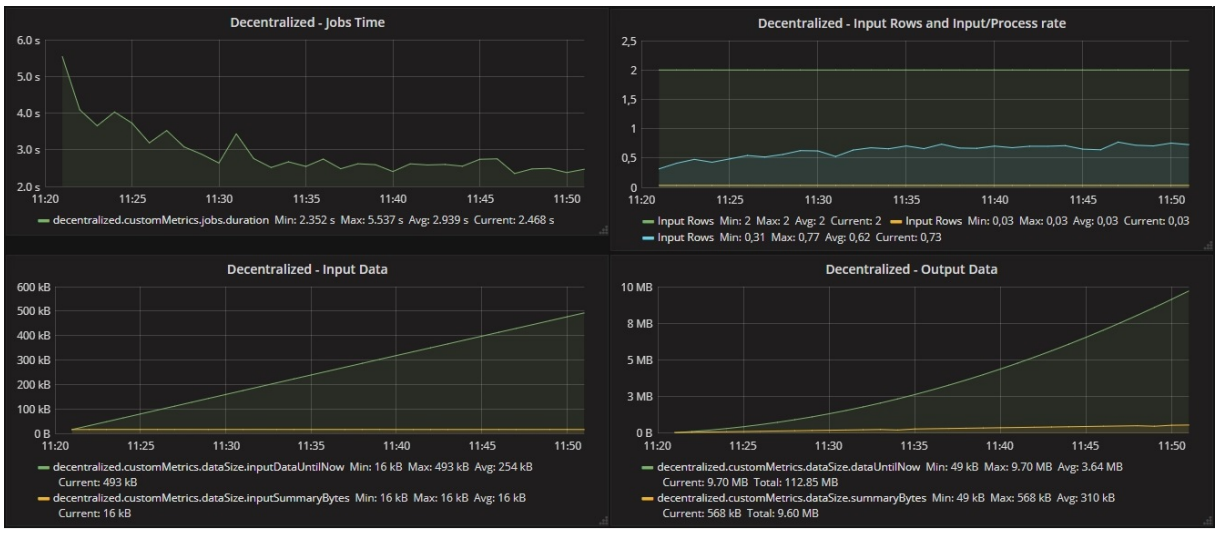

Figure 6.6: Graphs for Misra-Gries algorithm: decentralized configuration.

6.5.3 Summary

6.5.3.1 Average computation

Centralized and distributed statistics are really similar but with subtle differences. For job time there are no sensitive differences. Input rows entering the system are the same in both cases, and, therefore, the same happens for the number of input rows per second. Number of processed rows per second is instead different: it is about 31 thousands for centralized configuration and 40 thousands for distributed configuration. Also output data size is several order of magnitude smaller respect to input data size: from an input size of 60 MB, total output generated is 2,5 KB for both centralized and distributed configuration.

Differences in processing rate is given by advantages obtained by distributing workload among several machines. Reduction in output data size is the expected behavior because of the application of a summarization algorithm: from all input records, only the average is computed as representative value.

Metrics for decentralized configuration show that job time is reduced of an order of magnitude, from an average of about 1 second in the distributed configuration to 100 milliseconds in decentralized setting. Instead of about 50 thousands input per trigger, decentralized configuration reads only 2 records per trigger because there are 2 clusters producing summaries which are taken as input. It reads close to 0 input records per second because, being a trigger every 60 seconds, the input rate per second is close to zero but not exactly zero. Also processed rows per second are close to zero because there are essentially few records to read and process.

6.5.3.2 Misra-Gries algorithm

For Misra-Gries algorithm, centralized and distributed configurations perform in a similar way. There are no sensitive differences in job time and in the number of input rows and input rows per second. We can notice that number of processed rows per second is slightly higher in distributed configuration. Moreover, processing rate settles to a lower value respect to average computation because here UDAF is executed to compute summaries and this may put an upper bound to processing capabilities. Input and output data size are consistent with the concept of data synopses as from an input size of about 250 MB, generated output is about 700 KB.

Decentralized configuration shows a high job time respect to the average computation. This is explained because Misra-Gries algorithm is a more intensive computing algorithm that requires to execute a UDAF to merge different input summaries.

In graph showing number of input rows and input rows per second, we can read that

the application reads only 2 rows per trigger with values of input rate and processing rate close to zero because few records have being read as input.

Input data is, as expected, small in its size because merging application reads only summaries and not input data. Actual input data would be, as we can see in fig. 6.4, about 250 MB in size. Output data size is larger than the size of computed summary in centralized and distributed configuration because at this point big values of frequency are counted and, therefore, a summary occupies more space than the others in terms of bytes.

Chapter 7

Conclusions

7.1 Discussion

This thesis project has achieved two important results.

Firstly, a decentralized architecture has been designed with regard to possible algorithms to be implemented on it. Final design is the result of research conducted by exploring several tools with different features and capabilities. A consistent and coherent solution has been then characterized by choosing most suitable tools among them. Provided architecture is a flexible solution where Apache Spark and Apache Kafka have been used. The former is utilized as engine for data processing by exploiting its streaming capabilities, the latter is used both as publish-subscribe system and as storage system.

Secondly, some streaming algorithms have been analyzed. Features and properties that algorithms should and should not have to be decentralized have been described. Algorithms which have been dealt are the first to apply to manage a huge amount of streaming data: algorithms to reduce initial size and to extract statistics, both with respect to maintaining original data characteristics.

Then two algorithms have been chosen and implemented in a decentralized fashion on the designed architecture. First average computation have been implemented, then Misra-Gries algorithm have been chosen by applying original paper [30] and then merging algorithm of Misra-Gries summaries provided in [61].

The system is able to compute an aggregate by performing subsequent aggregations on two levels. First level of aggregation generates summaries, while the second one reconciles summaries into a global aggregate. This mechanism effectively reduces the amount of data which moves from the sources to the place where processing is performed. Aggregation and summarization prevents huge amount of data to reach reconciling application by crossing the network by limiting this flow of data to small summaries. These behavior can lead to save bandwidth costs and then money, which was the main motivation behind

this thesis project.

7.2 Future work

It is left for future work testing intensively the implemented applications by running multiple autonomous clusters geographically distributed. The idea is to allow merging application to read more summaries and reconcile them. In such situation, actual input data size would be really high as well as input rate: if each cluster in this project receives 50 thousands records per minute, multiple clusters would receive 50 thousands records multiplied by the number of clusters. Since input data size would be much larger, the effect of summarization, hypothetically, will be more significant than the one observed with current setting.

An other design for decentralized application is possible but it has not been considered in this work. The architecture can have more levels of aggregation instead of only two. First level application can produce aggregate which is read by a second level application. Second level application is reading, as actual last level of aggregation which merges summaries does, only summaries and then generates merged summaries for the last level of aggregation. Second and last level of aggregation can follow same algorithmic logic since both of them can rely only on input summaries and not on input data. This system can be useful when the second level of aggregation is still close to the sources but sources produce a huge amount of data considered as a group. Second level of aggregation acts as a further actor which gathers data from entities which are lower into aggregation process, and reports summaries to the last level of aggregation to extract the global summary.

Last experiment left for future work is the usage of a different tool inside each cluster. Instead of using Apache Spark with Structured Streaming, it is possible to use any kind of stream processing framework according to our requirements, such as Apache Flink or Storm. The only unchanged requirement is the usage of Kafka as publish-subscribe system and storage system. In fact, whatever stream processing framework should produce summaries transformed in JSON format and sent to the Kafka topic to be read as input from merging application. Architecture will remain the same, but the impact of different frameworks may be different.

Bibliography

[1] D. Kossmann and N. Tatbul, "Introduction to big data—the four v's", [Online]. Available: https://pdfs.semanticscholar.org/ae1f/1ed3d9b44bedd291cb991f pdf.

[2] D. Laney, "3d data management: Controlling data volume, velocity, and variety", 2001. [Online]. Available: https://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf.

[3] P. Struijs, "Big data for official statistics: Game-changer for national statistical institutes", [Online]. Available: http://www.eustat.eus/productosServicios/datos/58_Big_Data_for_Official_Statistics_Peter_Struijs.pdf.

[4] N. Cory, "Cross-border data flows: Where are the barriers, and what do they cost?", *Information Technology and Innovation Foundation (ITIF)*, 2017. [Online]. Available: http://www2.itif.org/2017-cross-border-data-flows.pdf.

[5] A. Odlyzko, "Internet pricing and the history of communications", *Computer networks*, vol. 36, no. 5, pp. 493–517, 2001. [Online]. Available: http://www.dtc.umn.edu/~odlyzko/doc/history.communications1b.pdf.

[6] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system", in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, IEEE, 2010, pp. 1–10. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.178.989&rep=rep1&type=pdf.

[7] 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*, ACM, 2013, pp. 423–438. [Online]. Available: http://delivery.acm.org/10.1145/2530000/2522737/p423-zaharia.pdf?ip=130.192.108.45&id=2522737&acc=OA&key=4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35%2EC42B82B876CFID=840547315&CFTOKEN=12027865&__acm__=1513438595_45fd2528fd7bca1d

[8] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, *et al.*, "Storm@ twitter", in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, ACM, 2014, pp. 147–156. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.699.1496&rep=rep1&type=pdf.

[9] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine", *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015. [Online]. Available: http://asterios.katsifodimos.com/assets/publications/flink-deb.pdf.

[10] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinze, "Revisiting the design of data stream processing systems on multi-core processors", in *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, IEEE, 2017, pp. 659–670.

[11] [Online]. Available: https://www.sgi.com/pdfs/4559.pdf.

[12] E. L. Yuliani, "Decentralization, deconcentration and devolution: What do they mean?", 2004. [Online]. Available: http://www.cifor.org/publications/pdf_files/interlaken/Compilation.pdf.

[13] S. Carlini, "The drivers and benefits of edge computing", *Schneider Electric–Data Center Science Center*, p. 8, 2016. [Online]. Available: http://www.apc.com/salestools/VAVR-A4M867/VAVR-A4M867_R0_EN.pdf?sdirect=true.

[14] A. Håkansson, "Portal of research methods and methodologies for research projects and degree projects", in *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)*, The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2013, p. 1. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-136960.

[15] . L.E. D. Bryant R. E. Katz R. H., "Big-data computing: Creating revolutionary breakthroughs in commerce, science, and society", 2008. [Online]. Available: http://cra.org/ccc/wp-content/uploads/sites/2/2015/05/Big_Data.pdf.

[16] D. S. T. Di Henrique C. M. Andrade Buğra Gedik, *Fundamentals of stream processing: Application design, systems, and analytics*. [Online]. Available: https://books.google.se/books?id=aRqTAgAAQBAJ&lpg=PA46&ots=-5Cixcke8R&dq=stream%20tuple%20classes%20structured%20unstructured&hl=it&pg=PA47#v=onepage&q&f=false.

[17] H. C. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of stream processing: Application design, systems, and analytics*. Cambridge University Press, 2014.

[18] P. B. Gibbons and Y. Matias, "Synopsis data structures for massive data sets", *External memory algorithms*, vol. 50, pp. 39–70, 1999. [Online]. Available: http://theory.stanford.edu/~matias/papers/synopsis-soda99.pdf.

[19] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine, "Synopses for massive data: Samples, histograms, wavelets, sketches", *Foundations and Trends in Databases*, vol. 4, no. 1–3, pp. 1–294, 2012. [Online]. Available: http://db.cs.berkeley.edu/cs286/papers/synopses-fntdb2012.pdf.

[20] C. C. Aggarwal and S. Y. Philip, "A survey of synopsis construction in data streams", in *Data Streams*, Springer, 2007, pp. 169–207. [Online]. Available: http://charuaggarwal.net/synopsis.pdf.

[21] B. Gedik, "Generic windowing support for extensible stream processing systems", *Software: Practice and Experience*, vol. 44, no. 9, pp. 1105–1128, 2014. [Online]. Available: http://repository.bilkent.edu.tr/bitstream/handle/11693/12959/10.1002-spe.2194.pdf?sequence=1&isAllowed=y.

[22] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of massive datasets*. Cambridge university press, 2014.

[23] J. S. Vitter, "Random sampling with a reservoir", *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.

[24] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors", *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[25] S. Kamburugamuve, G. Fox, D. Leake, and J. Qiu, "Survey of distributed stream processing for large stream sources", *Technical report*, 2013.

[26] P. Flajolet and G. N. Martin, "Probabilistic counting", in *Foundations of Computer Science, 1983., 24th Annual Symposium on*, IEEE, 1983, pp. 76–82.

[27] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments", in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, ACM, 1996, pp. 20–29.

[28] J. S. Moore, "A fast majority vote algorithm", *Automated Reasoning: Essays in Honor of Woody Bledsoe*, 1981.

[29] M. J. Fischer and S. L. Salzberg, "Finding a majority among n votes.", YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE, Tech. Rep., 1982.

[30] J. Misra and D. Gries, "Finding repeated elements", *Science of computer programming*, vol. 2, no. 2, pp. 143–152, 1982. [Online]. Available: https://ac.els-cdn.com/0167642382900120/1-s2.0-0167642382900120-main.pdf?_tid=2bc091c4-ebf7-11e7-a31e-00000aab0f6b&acdnat=1514483428_6d3a4efd8b6b7f2c235dbcca032ff54f.

[31] A. Bellet and M. Tommasi, "Decentralized machine learning under constraints",

[32] D. Frey, A.-M. Kermarrec, C. Maddock, A. Mauthe, and F. Taïani, "Adaptation for the masses: Towards decentralized adaptation in large-scale p2p recommenders", in *Proceedings of the 13th Workshop on Adaptive and Reflective Middleware*, ACM, 2014, p. 4.

[33] H. Ferreira, S. Duarte, and N. Preguiça, "4sensing–decentralized processing for participatory sensing data", in *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, IEEE, 2010, pp. 306–313.

[34] M. Babazadeh, A. Gallidabino, and C. Pautasso, "Decentralized stream processing over web-enabled devices", in *European Conference on Service-Oriented and Cloud Computing*, Springer, 2015, pp. 3–18.

[35] R. Dobrescu and F. Ionescu, *Large scale networks: Modeling and simulation*. Crc Press, 2016, pp. 226–230.

[36] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi, "Processing complex aggregate queries over data streams", in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, ACM, 2002, pp. 61–72.

[37] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi, "Mergeable summaries", *ACM Transactions on Database Systems (TODS)*, vol. 38, no. 4, p. 26, 2013.

[38] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, *et al.*, "Above the clouds: A berkeley view of cloud computing", Technical Report UCB/EECS-2009-28, EECS Department, University of California Berkeley, Tech. Rep., 2009.

[39] P. Teli, M. V. Thomas, and K Chandrasekaran, "An efficient approach for cost optimization of the movement of big data", *Open Journal of Big Data (OJBD)*, vol. 1, no. 1, pp. 4–15, 2015.

[40] ——, "Big data migration between data centers in online cloud environment", *Procedia Technology*, vol. 24, pp. 1558–1565, 2016.

[41] H. Yuan, J. Bi, B. H. Li, and W. Tan, "Cost-aware request routing in multi-geography cloud data centres using software-defined networking", *Enterprise Information Systems*, vol. 11, no. 3, pp. 359–388, 2017.

[42] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful scalable stream processing at linkedin", *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017. [Online]. Available: `http://www.vldb.org/pvldb/vol10/p1634-noghabi.pdf`.

[43] J. Ellingwood, *Hadoop, storm, samza, spark and flink: Big data frameworks compared*, 2016.

[44] Y. Wang *et al.*, "Stream processing systems benchmark: Streambench", 2016.

[45] B. Srikanth and V. K. Reddy, "Efficiency of stream processing engines for processing bigdata streams", *Indian Journal of Science and Technology*, vol. 9, no. 14, 2016.

[46] *Apache Spark Foundation - Powered By*. [Online]. Available: `https://spark.apache.org/powered-by.html` (visited on 09/12/2017).

[47] *Structured Streaming In Apache Spark*, Jul. 2016. [Online]. Available: `https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html` (visited on 09/16/2017).

[48] *Structured Streaming Programming Guide - Spark 2.2.0 Documentation*. [Online]. Available: `https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html` (visited on 12/16/2017).

[49] *Open Source In-Memory Computing Platform - Apache Ignite*. [Online]. Available: `https://ignite.apache.org/` (visited on 09/13/2017).

[50] *Collocated Processing - Apache Ignite*. [Online]. Available: `https://ignite.apache.org/collocatedprocessing.html` (visited on 09/13/2017).

[51] *Apache Toree*. [Online]. Available: `https://toree.apache.org/` (visited on 02/16/2017).

[52] *Apache NiFi Overview*. [Online]. Available: `https://nifi.apache.org/docs.html` (visited on 12/13/2017).

[53] *Apache Livy*. [Online]. Available: `https://livy.incubator.apache.org` (visited on 12/16/2017).

[54] *Apache Kafka*. [Online]. Available: https://kafka.apache.org/intro (visited on 12/17/2017).

[55] *Spark Streaming - Spark 2.2.0 Documentation*. [Online]. Available: https://spark.apache.org/docs/latest/streaming-programming-guide.html (visited on 12/16/2017).

[56] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, *et al.*, "Spark sql: Relational data processing in spark", in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ACM, 2015, pp. 1383–1394. [Online]. Available: https://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf.

[57] *A gentle intro to udafs in apache spark*. [Online]. Available: https://www.jowanza.com/blog/a-gentle-intro-to-udafs-in-apache-spark (visited on 12/17/2017).

[58] *Stack overflow - how to define udaf over event-time windows in pyspark 2.1.0*. [Online]. Available: https://stackoverflow.com/questions/42747236/how-to-define-udaf-over-event-time-windows-in-pyspark-2-1-0 (visited on 12/17/2017).

[59] *Apache ZooKeeper*. [Online]. Available: https://zookeeper.apache.org/ (visited on 12/17/2017).

[60] *Kafka: A detail introduction | Mukesh Kumar, Big Data and Hadoop Expert | Pulse | LinkedIn*. [Online]. Available: https://www.linkedin.com/pulse/kafka-detail-introduction-mukesh-kumar (visited on 12/17/2017).

[61] G. Cormode, *Misra-gries summaries*. 2016.

[62] T. Bray, "The javascript object notation (json) data interchange format", 2017.

[63] Y. Shafranovich, "Common format and mime type for comma-separated values (csv) files", 2005.

[64] *Amazon review data*. [Online]. Available: http://jmcauley.ucsd.edu/data/amazon/links.html (visited on 01/10/2018).

[65] J. McAuley, C. Targett, Q. Shi, and A. Van Den Hengel, "Image-based recommendations on styles and substitutes", in *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ACM, 2015, pp. 43–52.

[66] R. He and J. McAuley, "Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering", in *Proceedings of the 25th International Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, 2016, pp. 507–517.

[67] *Overview — Graphite 1.1.0 documentation*. [Online]. Available: http://graphite.readthedocs.io/en/latest/overview.html (visited on 12/20/2017).

[68] *Docs Home| Grafana Documentation*. [Online]. Available: http://docs.grafana.org/ (visited on 12/20/2017).