POLITECNICO DI TORINO

Master Degree in Computer Engineering

Master Thesis

Geo-distributed multi-layer stream aggregation

Supervisors:
Prof. Paolo Garza
Prof. Vladimir Vlassov
PhD student Zainab Abbas

Pietro Cannalire

Academic year 2017-2018

Summary

The standard processing architectures are enough to satisfy a lot of applications by employing already existing stream processing frameworks which are able to manage distributed data processing. In some specific cases, having geographically distributed data sources requires to distribute even more the processing over a large area by employing a geographically distributed architecture.

The issue addressed in this work is the reduction of data movement across the network which is continuously flowing in a geo-distributed architecture from streaming sources to the processing location and among processing entities within the same distributed cluster. Reduction of data movement can be critical for decreasing bandwidth costs since accessing links placed in the middle of the network can be costly and can increase as the amount of data exchanges increase. In this work we want to create a different concept to deploy geographically distributed architectures by relying on Apache Spark Structured Streaming and Apache Kafka.

The features needed for an algorithm to run on a geo-distributed architecture are provided. The algorithms to be executed on this architecture apply the windowing and the data synopses techniques to produce a summaries of the input data and to address issues of the geographically distributed architecture.

The computation of the average and the Misra-Gries algorithm are then implemented to test the designed architecture.

This thesis work contributes in providing a new model of building a geographically distributed architecture. The experimental results show that the computation time is reduced by 88% for the algorithm to compute the average and by 25% for the Misra-Gries algorithm compared to the distributed setup. Similarly, the amount of data exchanged across the network is reduced on average by 99%, compared to the distributed setup.

Keywords: *stream processing, geo-distributed, architecture, algorithms, windowing, data synopses, Apache Spark Structured Streaming, Apache Kafka, Misra-Gries algorithm*

Contents

List of Figures

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 live. 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 [7] which finds expression in decision making and process automation as well as in building enhanced business insights [8]. 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 [9]:

- *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.

In order to analyze such a data, any stream processing framework should fetch it and later apply any kind of algorithm on it. This implies movement of data from one place to another, from the place where it is produced to the place where it is processed. This movement is considered as a cost which can appear in terms of latency to get a given result starting from the streaming sources and in terms of bandwidth required to exchange a huge amount of data. The bandwidth costs, moreover, are strictly related to money since ISPs will charge customers for the usage of their infrastructure [10, 11].

Hence, the problem that this work wants to face is the issue which arises when dealing with a huge amount of streaming data by aiming in reducing movement of data across the network.

1.2 Motivation

Many stream processing frameworks currently on the market offer the capability to interact with streaming data and analyze it for any purpose. The complexity of the data which they can work on forces these systems to rely on distributed architectures and parallel-based processing. Hadoop [12], Spark [13], Storm [14], Flink [15] 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 coordinate 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 geographically 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. In such critical conditions guaranteeing robustness may be a problem and the cost of sending data across several regions could be significant because of the bandwidth needed to exchange a lot of data at a potentially high rate. Recent researches have proposed new designs for data stream processing systems [16] and some systems have been developed to work with hundreds of CPU cores and terabytes of memory [17], 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: to move the processing closer to where the data is produced. The idea behind this approach is taken from edge computing which places "data acquisition and control functions, storage of high bandwidth content, and applications closer to the end user" [18]. Similarly we want to delegate the first elaboration of data to the edge of the network and to process it in independent entities with less strict concerns for resources utilization. In this way, the bandwidth is less utilized on large scale because the data is still received and elaborated, but the data exchange is reduced to the minimum one required to communicate to a central entity what is happening in every edge.

The designed architecture is made of two layers: the edge layer is responsible of the first elaboration of data, the reconciliation layer is responsible of gathering the elaborated summaries which are produced in the edge entities and of computing a global result.

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 in edge entities, 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 [19]. Two main milestones are achieved for which different methodologies/methods are employed:

1. Finding which frameworks better suit to design a multi-layer architecture
2. Implement a chosen algorithm on top of the designed multi-layer 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 hypotheses in a simple practical application by coding and to draw conclusions about which building blocks to use.

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

The chosen methods are followed with the intent to realize a scenario which involves independent clusters of machines running a stream processing framework that are able to communicate any kind of information about the input data to a different entity.

Coding in particular took the great part of the entire project because of the need, in several aspects of the development, to try how APIs really worked. In fact, the chosen tool, Structured Streaming, is relatively new as it was released in 2016. Therefore, most of the time was employed in exploring and applying its features to achieve the desired result. Moreover, two algorithms have been developed on top of two slightly different frameworks. As a result, Structured Streaming, a stream processing framework in the Apache Spark environment, turned to be more valuable in different aspects respect to Spark Streaming. Then, algorithms taken into consideration, which include the ones in the main algorithms but also the ones in the exploratory research method, have been developed with significant changes in input data structures and in their mere logic of computation of results because of different context environments in which they should run. More details about implementation will be provided in Chapter 5.

1.4 Contributions

This master thesis aims to achieve several contributions:

1. Providing a model for a concrete geographically distributed architecture, ready to execute algorithms, which finds its fundamental tools in Apache Kafka and Apache Spark Structured Streaming.

2. Providing an architecture which is able to elaborate streaming data and to prevent large amount of data to move throughout the network by reducing input data size and employing a summarization mechanism. The amount of data that is moving after being read from the sources is reduced from hundreds of megabytes every few minutes to few kilobytes.

3. Providing the features that an algorithm should have to be run on such an 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.

The third part discusses the tools which have been considered and the ones finally chosen. Then a section will explain different architecture configurations and where the tools are used. The fourth part will deal with methods and algorithms that can be applied on a geo-distributed infrastructure and that are chosen to run on the designed architectures. The fifth part will show the implementation of the system and the algorithms. Sixth part will compare and evaluate different architectural configurations, 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 a large volume of information. Big Data [9] 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. The *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 [20].

A cluster can efficiently support 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

The 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 [21]:

- Structured: data with known schema (relational databases, etc.)

- 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 different time references:

- *Event-time*, that corresponds to the time when the event generating the data has happened
- *Processing-time*, which is instead the time when the tuple is received by the system and processed
- *Ingestion-time*, that represents the time when the event enters the framework

They are differently defined because the application logic usually involves event-time to arrange correctly in time every tuple and processing-time to take corrective measures on data. Ingestion-time can be involved in specific logics as well.

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 [21].

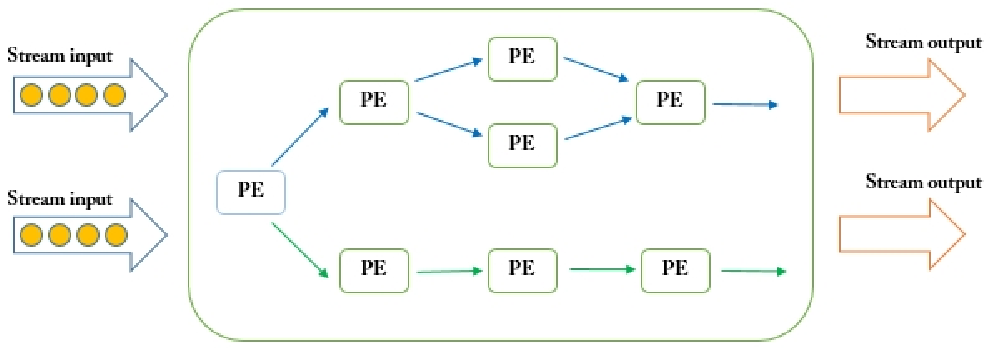


Figure 2.1. Processing model.

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. The tuples in a stream can be considered as rows in a relational database, but the difference with the standard rows is that, in a stream, each tuple passes once from the system and it should be elaborated in that moment. Hence, in order to run an algorithm and obtain a desired result, we cannot treat the tuples as they were standard data that we can fetch again later, but we need different kind of algorithms and techniques.

2.2.1 Type of queries

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. A *standing query* is continuously running and internally maintains a state that is updated while the stream is flowing. Instead, 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.

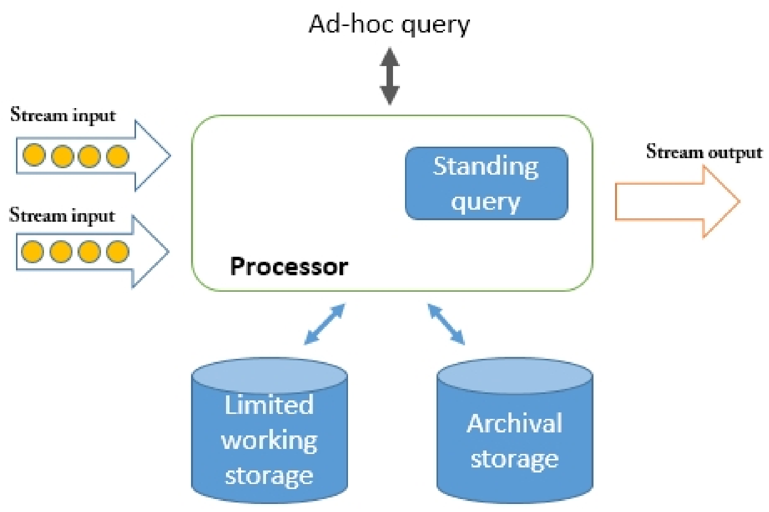


Figure 2.2. Type of queries.

A visual representation of queries can be found in figure 2.2. 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

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 [22]:

- 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 [23, 24]:

- *sampling*, which can obtain a "representative subset of data values of interests" [23]. Sampling can be achieved through different techniques and algorithms, some of them will be showed in the section 2.2.3.

- *histograms* which consists of a summarization of the dataset by grouping data in subsets called *buckets* for which statistics of interests are computed. The 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" [24].

- *sketches* are summaries constructed by applying a particular matrix to input data seen as a vector or a matrix of data points.

Windowing

Starting from the assumption that any system can't store the entire stream, the *windowing* [25] 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 [25]:

- *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 [25]:

- *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 each tuple only once, 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 the data stream.

Sampling and filtering algorithms

Generally, the 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 [26].

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 the 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* [27], 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 the 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 [28]. 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 one of the m keys, all k hash functions are applied to the corresponding bit to 1 in the bit-array. When a new tuple arrives, the hash functions are applied and, if the resulting value for every hash function is 1 into the bit-array, the tuple is accepted, otherwise it 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.

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" [29].

Flajolet-Martin approach In order to count distinct elements a basic approach is the Flajolet-Martin one [30] 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" [26].

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 $(m_i)^k$" [26]. 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 than two are calculated in a similar way as the second moment [31, 26].

The AMSalgorithm [31] is used to compute the second moment. Supposing an infinite stream and the possibility to store k number of occurrences, each element's value and count are stored. 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, and, 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 [32] and later proved in his optimality by Fischer and Salzburg [33]. The Majority algorithm was then generalized by Misra and Gries [34] whose algorithm was not made for streaming problems but it works by 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, the Misra-Gries algorithm maintains a counter for up to k 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 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

Among similar works, first in [35] and later in [36], Teli et al. have worked on the necessity to reduce data movement between geographically distributed clusters by proposing a specific algorithm to exchange information by reducing costs. Yuan et al. in [37] have provided a heuristic algorithm to minimize energy and bandwith costs for cloud data centers (CDCs) providers which usually deal with several ISPs. In [38], Sajjad et al. propose SpanEdge, a novel approach in the stream processing over a geo-distributed infrastructure, which distributes stream processing applications across central and near-the-edge data centers by allowing the programmer to specify which part of the application should be closer to the data sources.

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" [39]. 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. [40] 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. [41] studied mergeability of summaries.

Chapter 3

Architecture and tools

3.1 General concept

To build up a geographically distributed multi-layer architecture with particular concerns in reducing data movement costs, 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 any cluster to efficiently exchange information about local processing

In all clusters, as in a standard distributed cluster which occupies a large geographical area, data exchange is needed in order to provide basic features like replication of data, fault tolerance or simply to exchange information. Providing the architecture with autonomous clusters firstly allows to move processing closer to data sources, secondly prevents continuous and unavoidable amount of data in a cluster to move around the network. In fact, when the data moves, it has to use the already existing WANs which have a cost in terms of bandwidth. Relying on autonomous clusters aims to reduce as much as possible the communication over large areas: there is no direct data exchange between clusters, but each cluster can autonomously produce a small information about input data stream which is the only data that will move across the network.

The second requirement represents the necessity of a way to communicate the information which is produced inside the clusters and that is related to the input data. Such an information should be delivered to a specific layer in the architecture which is responsible of collecting it to create a global view.

Deploying a geo-distributed 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. The only constraint is agreeing on a common language for representing data which every cluster produces. 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.

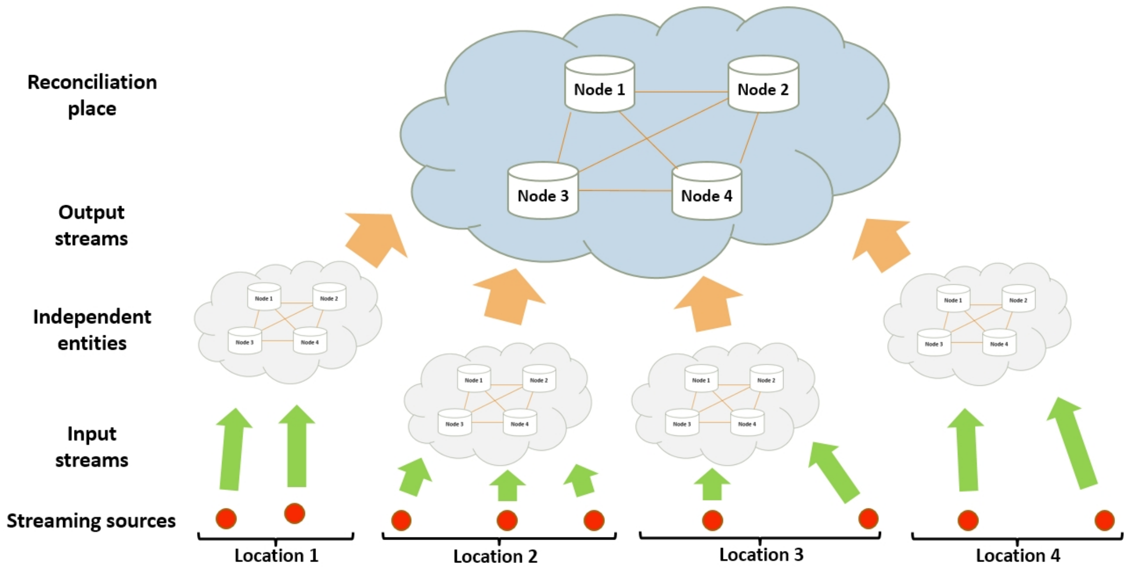


Figure 3.1. Geo-distributed multi-layer architecture.

Both requirements can be put together in a multi-layer architecture showed in figure 3.1. Several streaming sources are geographically distributed and are placed in different locations. The layers in the designed architecture are two:

- the first layer, called *edge layer*, includes different clusters each of which reads input data streams, processes them and produces an output stream.
- the second layer, called *reconciliation layer*, is a further cluster which reads output streams coming from the edge layer and reconciles all information into a unique one.

Each cluster can be itself distributed over a small area to take advantage of resources offered by different nodes.

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 the first requirement mentioned in the previous section, many stream processing frameworks have been considered: Storm [14], Flink [15], Spark Streaming [13], 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

The first category includes frameworks which consider data coming into the system as a pure stream of data: they work with a flow of atomic pieces of information that are processed as soon as possible by the framework. An hybrid framework is able to work as a stream-only framework does, but also with batch data, which are blocks of data that can be stored somewhere in the system and that can be retrieved and analyzed as they were streaming data.

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 HDFS 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 [13] 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" [2], 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 *Datasets* 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 the 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

The second thesis project requirement consists in researching the more suitable option to deliver information about local cluster processing 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 [48] 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" [49].

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 [50] 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 the supposed complexity in implementation gives priority to other solutions.

NiFi. "Put simply NiFi was built to automate the flow of data between systems" [51]. 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." [51].

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.

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

Livy. Livy [52] 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 [5] 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 [5]. 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 geo-distributed multi-layer 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 the thesis work, hence flexibility and integrability are preferred and Apache Spark was elected to be adopted in the 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.

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 and then switched 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* [13], which is Sparkâ€™s abstraction of an immutable, distributed dataset" [1].

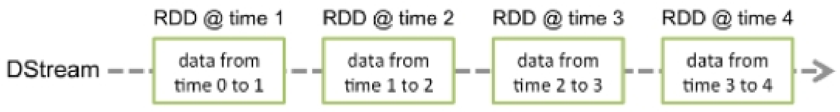


Figure 3.2. RDDs in DStream composition [1].

Figure 3.3 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 from a specific source, like Kafka, or maintain a state which

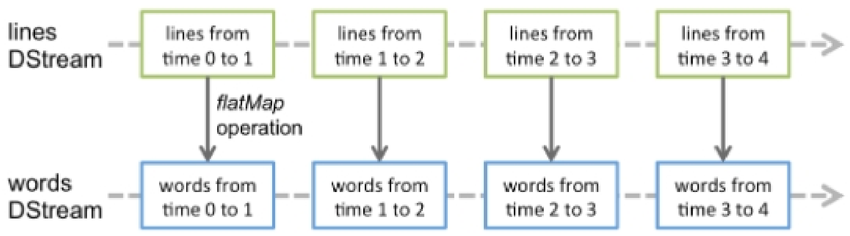


Figure 3.3. DStream transformation example: *flatMap* transformation on a DStream [1].

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 the chosen algorithm over the data stream because of its complexity. Therefore another possible solution was explored in the Apache Spark environment.

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 [2]."

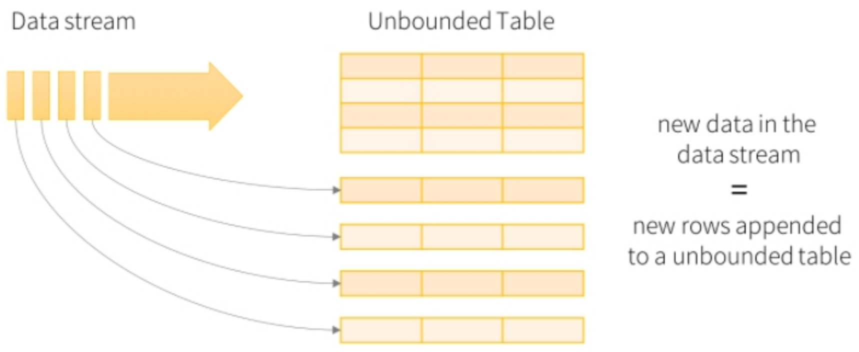

Data stream
Unbounded Table
new data in the data stream
=
new rows appended to a unbounded table

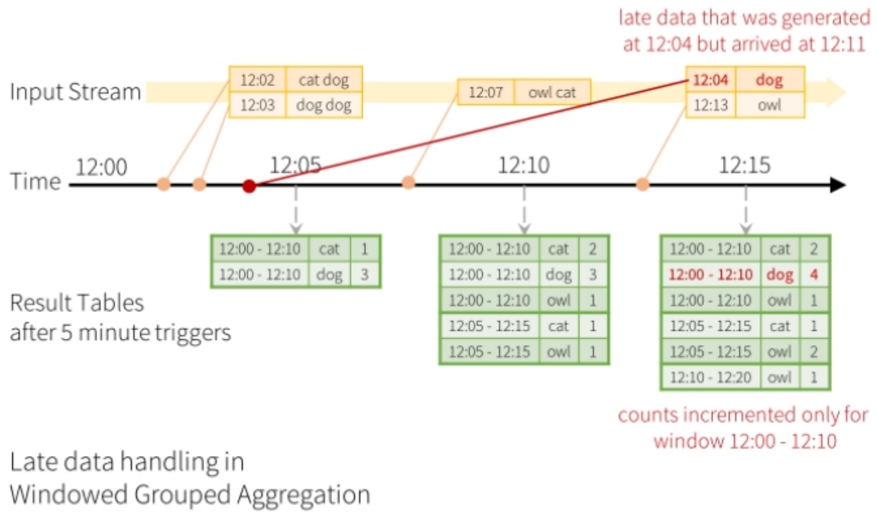
Figure 3.4. Data stream as an unbounded table [2].

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.


late data that was generated at 12:04 but arrived at 12:11
Input Stream
12:02 cat dog
12:03 dog dog
12:07 owl cat
12:04 dog
12:13 owl
Time
12:00
12:05
12:10
12:15
Result Tables after 5 minute triggers
counts incremented only for window 12:00 - 12:10
Late data handling in Windowed Grouped Aggregation

12:00 - 12:10	cat	1
12:00 - 12:10	dog	3

12:00 - 12:10	cat	2
12:00 - 12:10	dog	3
12:00 - 12:10	owl	1
12:05 - 12:15	cat	1
12:05 - 12:15	owl	1

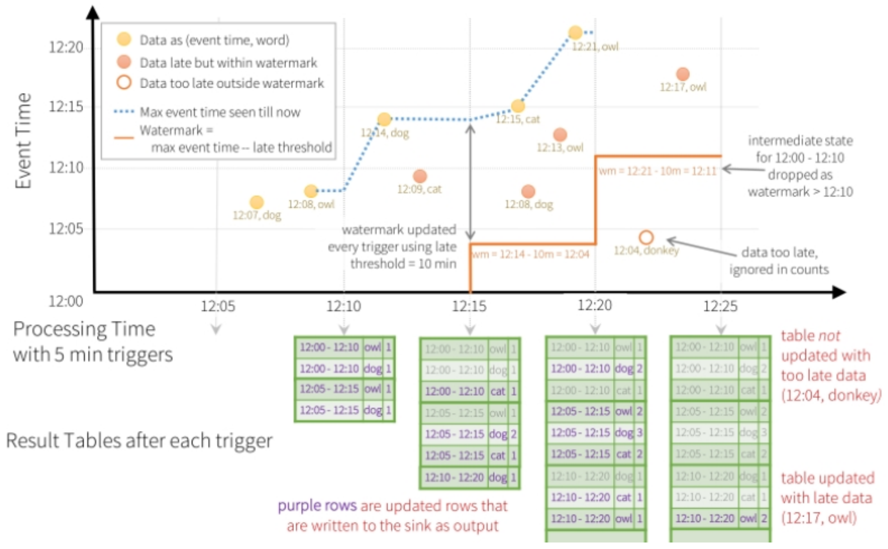
12:00 - 12:10	cat	2
12:00 - 12:10	**dog**	**4**
12:00 - 12:10	owl	1
12:05 - 12:15	cat	1
12:05 - 12:15	owl	2
12:10 - 12:20	owl	1

Figure 3.5. Late data handling [2].

Figure 3.5 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 3.6, for example, while tuple $(12:09, cat)$ is accepted and will update the correspondent window aggregate because its window reference is still alive


Event Time
12:20
12:15
12:10
12:05
12:00
Data as (event time, word)
Data late but within watermark
Data too late outside watermark
Max event time seen till now
Watermark = max event time -- late threshold
12:07, dog
12:08, owl
12:14, dog
12:09, cat
12:15, cat
12:13, owl
12:08, dog
12:21, owl
12:17, owl
12:04, donkey
watermark updated every trigger using late threshold = 10 min
wm = 12:14 - 10m = 12:04
wm = 12:21 - 10m = 12:11
intermediate state for 12:00 - 12:10 dropped as watermark > 12:10
data too late, ignored in counts
12:05
12:10
12:15
12:20
12:25
Processing Time with 5 min triggers
Result Tables after each trigger
purple rows are updated rows that are written to the sink as output
table *not* updated with too late data (12:04, donkey)
table updated with late data (12:17, owl)

12:00 - 12:10	owl	1
12:00 - 12:10	dog	1
12:05 - 12:15	owl	1
12:05 - 12:15	dog	1

12:00 - 12:10	owl	1
12:00 - 12:10	dog	1
12:00 - 12:10	cat	1
12:05 - 12:15	owl	1
12:05 - 12:15	dog	2
12:05 - 12:15	cat	1
12:10 - 12:20	dog	1

12:00 - 12:10	owl	1
12:00 - 12:10	dog	2
12:00 - 12:10	cat	1
12:05 - 12:15	owl	2
12:05 - 12:15	dog	3
12:05 - 12:15	cat	2
12:10 - 12:20	dog	1
12:10 - 12:20	cat	1
12:10 - 12:20	owl	1

12:00 - 12:10	owl	1
12:00 - 12:10	dog	2
12:00 - 12:10	cat	1
12:05 - 12:15	owl	2
12:05 - 12:15	dog	3
12:05 - 12:15	cat	2
12:10 - 12:20	dog	1
12:10 - 12:20	cat	1
12:10 - 12:20	owl	2

Figure 3.6. Watermarking [2].

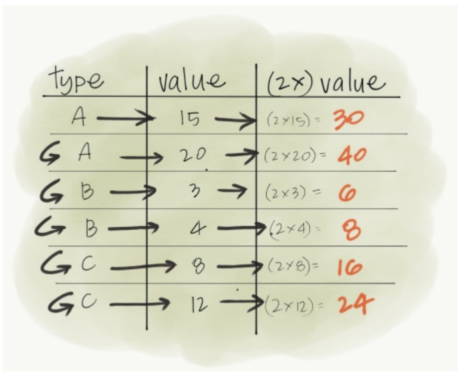
having maximum event-time behind the watermark, the tuple $(12:04, \textit{donkey})$ is not accepted because its reference window has been dropped since its maximum event time falls behind the watermark.

4. Structured Streaming is run over Spark SQL [53] 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 increases 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 (fig. 3.7) processes every incoming record and modifies one or more of its fields by applying a user defined function.

   Similarly, UDAF (fig. 3.8) 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 the standard aggregate functions in SQL language, which allows, for example, to compute sum, average or simply records count after grouping records.

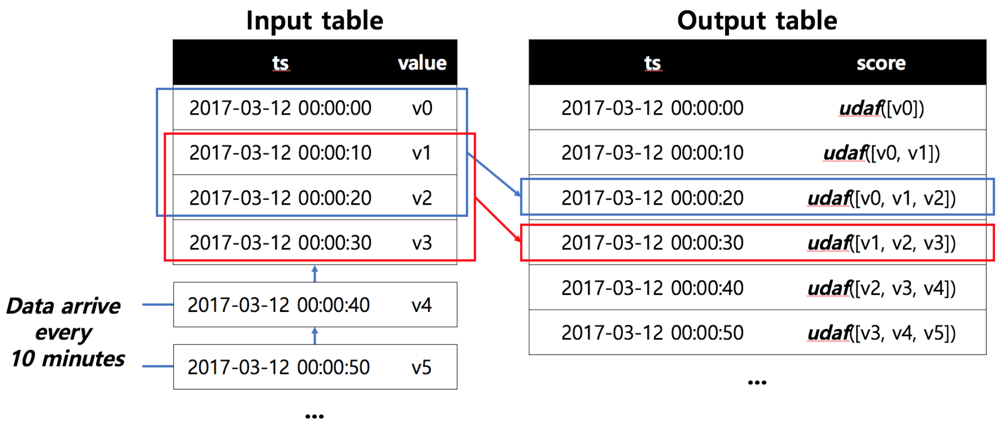
Apache Kafka

In order to deliver information from a layer to the following, Apache Kafka [5] was chosen since it offers all features to be part of the environment of this thesis work.



type	value	(2x) value
A	15	(2x15) = 30
A	20	(2x20) = 40
B	3	(2x3) = 6
B	4	(2x4) = 8
C	8	(2x8) = 16
C	12	(2x12) = 24

Figure 3.7. UDF example [3].



Input table

ts	value
2017-03-12 00:00:00	v0
2017-03-12 00:00:10	v1
2017-03-12 00:00:20	v2
2017-03-12 00:00:30	v3
2017-03-12 00:00:40	v4
2017-03-12 00:00:50	v5
...	

Output table

ts	score
2017-03-12 00:00:00	*udaf*([v0])
2017-03-12 00:00:10	*udaf*([v0, v1])
2017-03-12 00:00:20	*udaf*([v0, v1, v2])
2017-03-12 00:00:30	*udaf*([v1, v2, v3])
2017-03-12 00:00:40	*udaf*([v2, v3, v4])
2017-03-12 00:00:50	*udaf*([v3, v4, v5])
...	

Figure 3.8. UDAF example [4].

Kafka is basically a messaging system which is able to store a 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 [5]:

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" [5]. 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 3.9, 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.

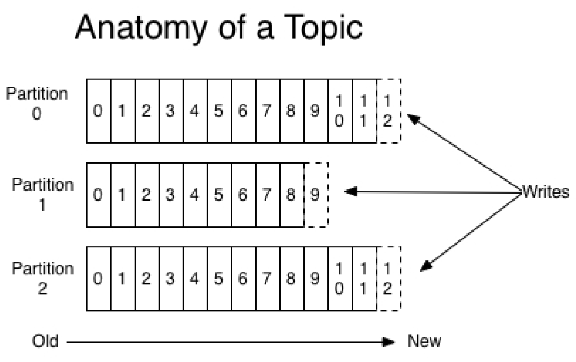


Figure 3.9. Anatomy of a topic: partitions of a topic [5].

A partition can be read from different consumers thanks to the *offset* which is an *ID* assigned to every record: an ID is maintained by each consumer to keep track of which records have been read before (fig. 3.10). At the end, Kafka cluster looks like figure 3.11: 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.

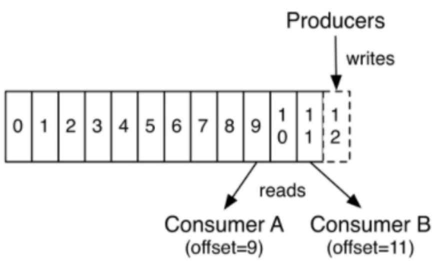


Figure 3.10. Where consumers read and where producers write [5].

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" [54]. It has information about the data distribution on the cluster nodes and the topic consumers and their offset. In practice it is responsible for managing topics and its related information and metadata, and when a consumer asks to read a topic, Zookeeper provides information about to

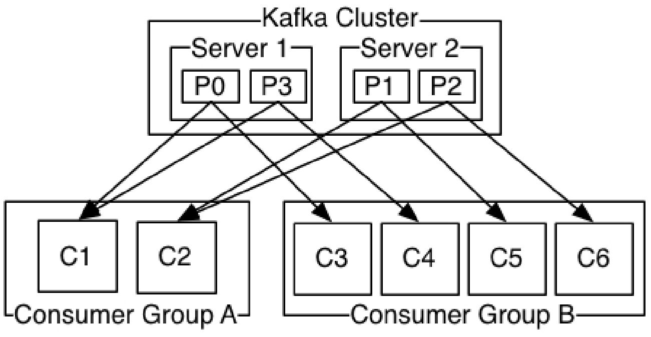


Figure 3.11. Example of how a Kafka cluster is made [5].

where to find data of requested topic. Hence, each Kafka cluster needs a Zookeper and its architecture looks like figure 3.12.

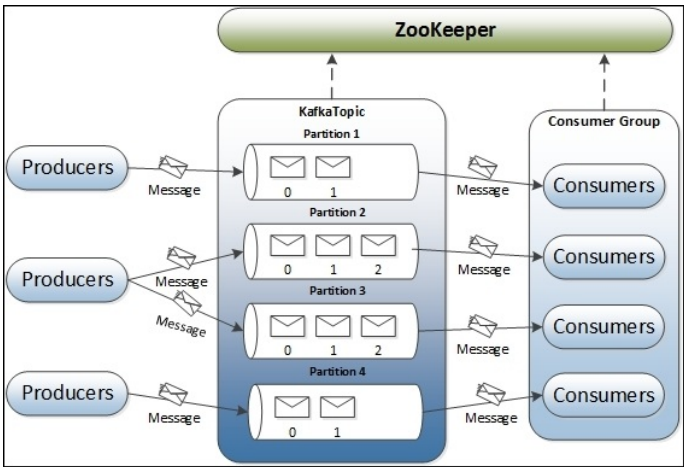


Figure 3.12. Kafka cluster including producers, consumers, topics and zookeeper [6].

3.3 Architectures

In this section different architecture configurations are described in their general settings and in the usage of tools selected in section 3.2.3. These configurations will be employed as an infrastructure where to execute the algorithms described in chapter 4 in order to perform an evaluation of those algorithms on different configurations.

The fundamental unit for the whole structure is the *Spark cluster* where a Structured Streaming application is running. A Spark cluster is in direct contact with real data and it is able to run a stream processing application to compute an aggregate on its own.

3.3.1 Distributed configuration

The distributed configuration is made of a single cluster which can be distributed over a large geographical area with different nodes. This cluster runs a Structured Streaming application which reads the input stream from a topic, executes window based aggregations and computations and produces an output stream with the results. A representation of this configuration is shown in figure 3.13

The computation is distributed because Spark cluster is spread over different machines and potentially different geographical areas and the application can exploit resources provided by Spark resource manager which are taken from those nodes.

The configuration also includes two Kafka clusters, one with the input topic and another one with the output topic. The input Kafka topic is distributed because the brokers run on different geographically distributed machines. The output Kafka topic instead is not distributed but it is placed close to where the Spark cluster is.

Since the Kafka brokers are geographically distributed, the producers can produce the data on the closest broker server to avoid significant data movement. This location not help indeed in reducing overall data movement because data is still read from input topics by Structured Streaming application generating in the same way a flow of data towards the distributed Spark cluster. Not distributing input topic would incur in the same issue but on a different side: the data would flow from sources to reach the input topic for being processed.

3.3.2 Multi-layer configuration

The designed multi-layer architecture consists of two levels. The 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. 3.13. The 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 a reduced input data set.

The multi-layer configuration is showed in fig 3.14. Each cluster has its own Kafka cluster with a zookeeper and one or more broker servers. The distributed topic where the first level aggregates are published is part of a different Kafka

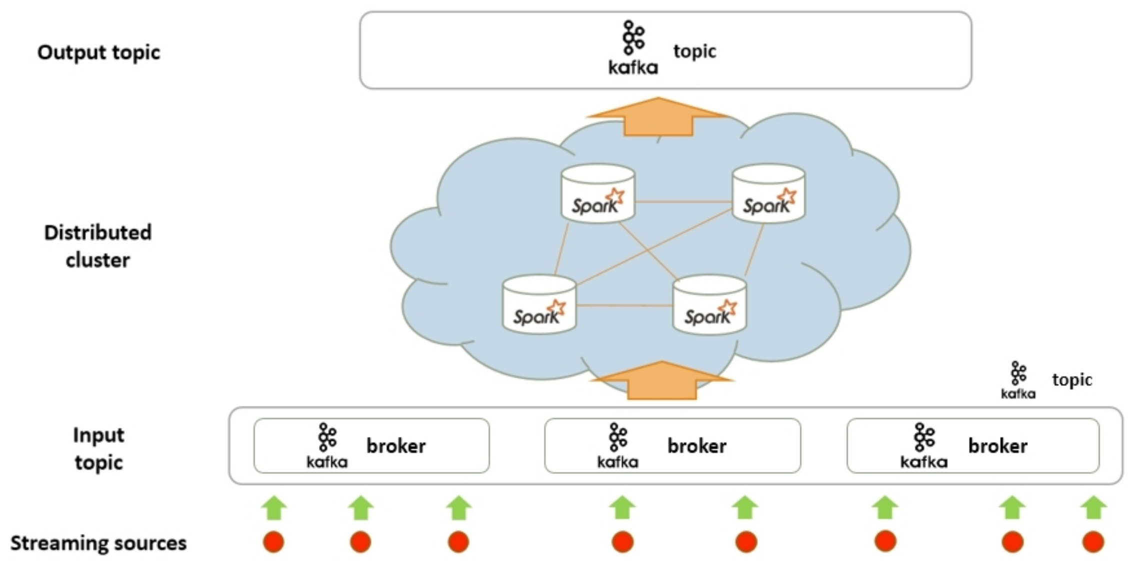


Figure 3.13. Distributed configuration.

cluster which is distributed to achieve fault tolerance, since Kafka is used also as a storage system. The topic where global aggregates are published is distributed as well for the same reason as the first level topic.

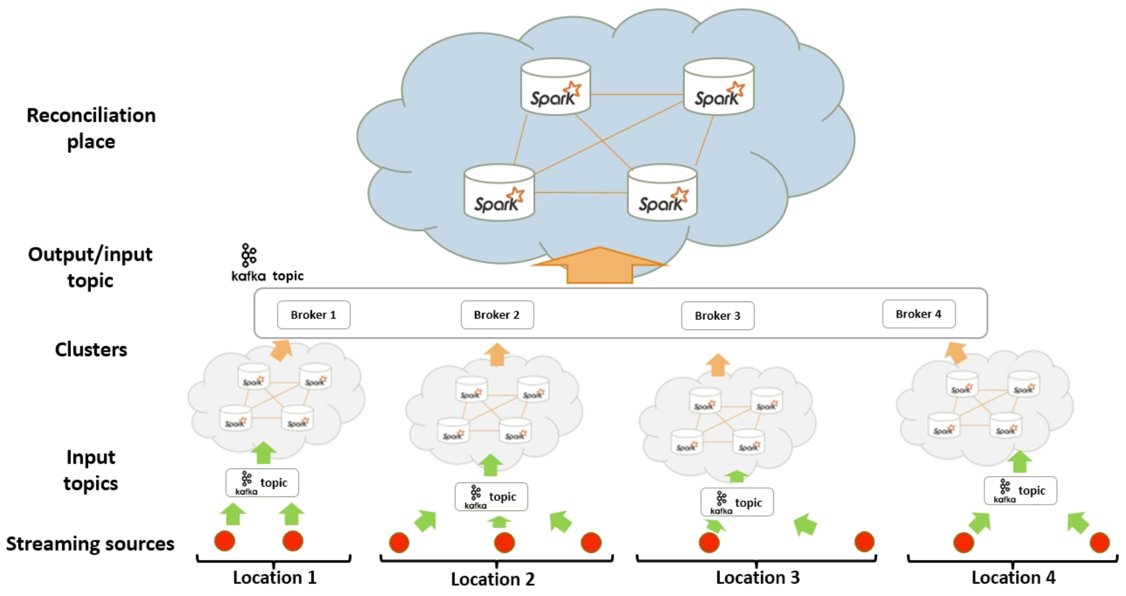


Figure 3.14. Geo-distributed multi-layer configuration.

Chapter 4

Methods and algorithms

4.1 Query

In this work, the 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 to monitor particular aggregates over it or compute 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, the methods employed are:

- *data synopsis* for each time window, describing data seen inside it

- *windowing*, in particular sliding windows

4.2.1 Data synopsis

The first method is critical in the whole thesis project because of the motivation which stands behind it: since we want to reduce the 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 an 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 important for the application in direct contact with high volume data which is running inside each cluster: without windowing, in order 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 limitations of actual computing systems which have limited resources for computing but, most of all, limited memory 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 the first level and by the second level of aggregation, should refer to a time window. The first level time windows are *tumbling windows* because of the need to aggregate incoming data into non-overlapping summaries, whereas the second level time windows are *sliding windows*. The choice of tumbling windows for the first level of aggregation allows the last level of aggregation to aggregate summaries coming from the tumbling windows in an easier way.

The sizes of the time windows are dependent to each other. Firstly, if all the time windows in the first level of aggregation have the same size, this time window must be a *sub-multiple* of the time window in the second level of aggregation. This means that the time window of the first level of aggregation should fit inside the time window of the second level of aggregation, it cannot be greater than it. Moreover, the *sliding time* of the second level time window is at least as the size of the first level time windows. An example is showed in 4.1. If the second level summaries refer to a time window of 30 minutes, the first level summaries may be 10 minutes time windows and the second level time window can slide ahead in time by 10 minutes.

Secondly, if in the edge layer there are time windows of different sizes, this time windows are related to each other and also to the second level time window.

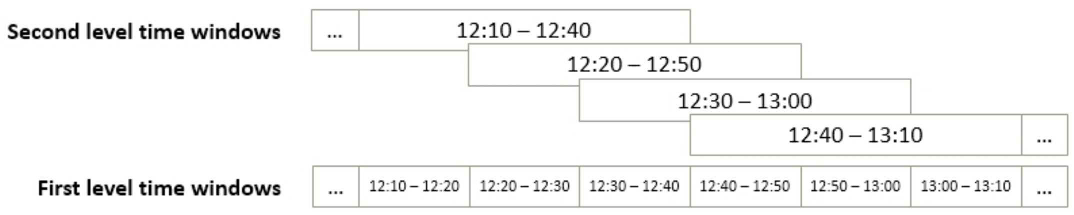


Figure 4.1. Windowing. First level time window of the same size.

- the second level time window is one of the *common multiple* among the time windows in the first level
- each first level time window is one of the *common divisors* of all the first level time windows that are greater than itself

For example (fig.4.2), if two first level time windows are 10 seconds and 20 seconds time windows, the second level summary can refer to a 20 seconds time window or a 40 second time window and so on. If we want to add a further time window in the first level, its time window can be a 10 seconds time window or a 20 seconds time window, as the already existing ones, but also a 5 seconds time window or a 1 second time window. The second level time window (i.e. 40 seconds) is still one of the common multiples among all the first level time windows, and the 5 (or 1) seconds time window is one of the common divisors among the time windows with a size of 10 and 20 seconds.

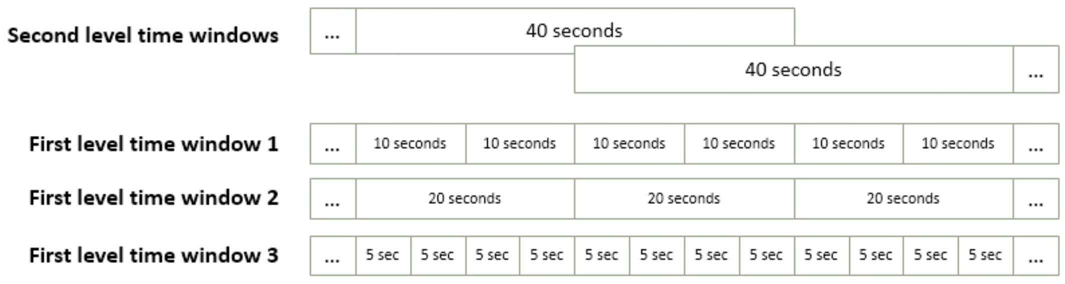

Second level time windows
...
40 seconds
40 seconds
...
First level time window 1
10 seconds
First level time window 2
20 seconds
First level time window 3
5 sec

Figure 4.2. Windowing. First level time window of the different size.

This constraints allow the first level summaries to fit the time window in the second level summary, without worrying about overlapping first level summaries which can fall out of the second level time range.

Finally, a further consideration should be done for the sliding time in the second level time window: it corresponds to the greater time window among the first level summaries (or one of its multiples) to prevent overlapping time windows during the computation. In the example in the figure 4.2, the sliding time in the second level time window is 20 seconds since the greater time window size is exactly 20 seconds.

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 run on the designed architecture. The analysis starts from an assumption which is derived from the architecture, as it was shown in figure 3.14.

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 a specific order (with respect to event time) and that the application logic should be aware of this behavior. Since the 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.

The 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. A summary produced by the first level of aggregation should have the same properties of the summaries which will be read by the second level of aggregation to create a final summary.

4.3.1 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 since 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 an input data sets D_i which produces a summaries S_i, where "+" is the merging operation between two different summaries:

- the *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 the 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.

- the *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}. 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.

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

4.3.2 Limitations

The implementation of an algorithm in a multi-layer architecture should be done after an accurate analysis of the algorithm itself and of the expected input and output data size. In fact, provided that the infrastructure already exists, the realization and the adaptation of an algorithm to be run on the designed architecture should be avoided if:

- the second step in the aggregation process which happens in the merging application needs input data. In fact, the second level aggregation relies only on data which is reported inside summaries. If a second level summary needs input data to better characterized each summary, in order to reach the reconciliation place, the input data should move to the merging application by undermining the effort to keep the first processing in the edge layer and closer to the data sources.

- the output size of a first level result is comparable with the input data size: the output data size of the first level of aggregation should be at least one order of magnitude smaller than the input data size. In other words, the summary should really summarize input data or make it smaller in some way and not only transform it or providing a simpler representation of it without reducing its size.

- applying the merging procedure increases overall error in computing the summary. This means that the second level summary should still maintain an error comparable with the ones in the first level summary and represent as accurately as possible the input data.

- there is the need of achieve a strict real-time analysis. The system will be aware of any item coming into the system only after it has been aggregated in a summary and the last level of aggregation has aggregated the lower level summaries. For a two layer architecture, the delay depends on the sliding time of the second level time window, the time that should pass before recomputing the aggregated summary.

Therefore, in order to run on the designed architecture, an algorithm should produce a summary that is *self-explanatory*, it should contain all information needed to be characterized and to be aggregated with other summaries. Moreover, a summary should be *smaller* then the input data size in order to fulfill the main motivation of the multi-layer architecture related to the placement of the first processing of data closer to the data sources. Finally, any summary should be merged with other summaries without increasing the *error* in representing the entire input data, the final summary should still represent as better as possible the input data from all sources.

4.3.3 Summary

The properties defined in 4.3.1 are not really strict but a guideline since a summary can be easily built as it is required by the application logic according to the desired values and the aggregates to compute and merge. The limitations provided in 4.3.2 instead should be considered with more attention because are the basic requirements that lead the deployment of a geographically distributed multi-layer architecture.

After this considerations, the concrete features for an algorithm that should run on the designed architecture are:

- producing a summary which actually aggregates the input data in an efficient way (following the guidelines in the section 4.3.2)

- producing a summary that can be easily merged with other summaries by satisfying firstly the guidelines and properties defined in the section 4.3.1

4.4 Chosen algorithms

Considering the algorithms features and the requirements defined in sections 4.3.1 and 4.3.2, first a simple algorithm has been chosen to test the feasibility of implementing an algorithm on the designed architecture, then a more complex algorithm has been implemented.

The first algorithm is the computation of the average and it was chosen because it is simple and based only on mathematical operations, then it is suitable to test on first hand framework flexibility and possibilities. The second one is the Misra-Gries algorithm, which was already mentioned in section 2.2.3. 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 the size of the output data and, hence, to control data movement.

In both cases, the 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 the average in a multi-layer fashion two steps are necessary. In the first step the average for the incoming elements is computed by using the built-in aggregate function for computing the average of a set of elements. At this moment, the total sum and the count is also computed to be sent to the next level of aggregation. In the second step, the sums and the counts are summed up and the average is computed by dividing their values once computed.

Algorithm 1 First level of aggregation. Computation of the average.

```
1: procedure COMPUTATION OF THE AVERAGE 1
2:     group elements in time windows according to event-time
3:     for each time window do
4:         compute sum
5:         compute count
6:         compute average
7:         pack all in a JSON record
8:     send computed records to output topic
```

Pseudo-code for computing average in the first step is described in algorithm 1. Grouping is first required in order to obtain different groups of records according to their event-times. A record can be assigned to one or more time windows. Each group is independent and, for each of them, the sum, the count and the average are computed.

A final summary will contain the *time window*, the *sum*, the *count*, optionally the *average*, which is useful to easily grasp the average of numbers in that time range, and the *cluster id*, which is a unique identifier among the clusters which is set to identify who have produced that summary.

The second step is executed in the merging application which receives the summaries generated as output from the first step. The pseudo-code for the merging-application is presented in the algorithm 2.

Algorithm 2 Second level of aggregation. Computation of the average.

```
1: procedure COMPUTATION OF THE AVERAGE 2
2:     extract the start time of the time window
3:     group elements in time windows according to the start time of the time window
4:     for each time window do
5:         compute the sum of sum values
6:         compute the sum of count values
7:         divide the sum of the sums by the count of the counts to compute the average
8:         pack all in a JSON record
9:     send computed records to output topic
```

The incoming summaries are read and the start time of the time window which they contain is extracted. This time is used to group all incoming summaries into time windows. The summaries inside each time window are aggregated first by summing up all sums and the counts, then by dividing the sum of the sums and the count of the count to obtain the value of the average.

A second level summary created with the algorithm 2 will contain the *time window* and the *average* of all input numbers which entered the system in that time window. The summary is also sent out to a Kafka topic to be stored and potentially further analyzed.

4.4.2 Misra-Gries algorithm

The Misra-Gries algorithm [34] generates a set of k pairs made by the element i and its frequency c_i. This set is the representation of the k most frequent items in the summary which is built according to the algorithm 3 which takes as input a stream of elements (is) and generates as output a set of k pairs in the form of (i, c_i).

For each incoming record i, the algorithm checks if the element is already contained in the summary. If the element is already contained, its counter is incremented. If the element is not contained in the summary and the size of summary is

Algorithm 3 Misra-Gries algorithm. Computation of the summary.

```
1: procedure MISRA-GRIES ALGORITHM
2:     $n \leftarrow 0$                                   ▷ count of incoming element
3:     $T \leftarrow \emptyset$                                   ▷ set of pairs
4:     for each incoming element $i$ do
5:         $n \leftarrow n + 1$
6:         if $i \in T$ then
7:             $c_i \leftarrow c_i + 1$                   ▷ $c_i$ is the frequency of the item $i$
8:         else if $|T| < k$ then
9:             $T \leftarrow T \cup \{i\}$
10:            $c_i \leftarrow 1$
11:        else
12:            for each $j \in T$ do
13:                $c_j \leftarrow c_j - 1$
14:                if $c_j < 0$ then
15:                    $T \leftarrow T \setminus \{j\}$
```

less then k, then x is added to the summary with counter equal to 1. In any other case i is discarded and all counters in the summary are decreased by 1.

The resulting summary is made of elements which occur at least $n/(k+1)$ times. In fact, when an element is not contained in the summary, all k elements are decremented by 1. This operation is equivalent to deleting $k+1$ elements from the stream (k elements of the summary plus the element itself). Hence, if an element occurs more than $n/(k+1)$ times cannot be deleted because there are not enough elements to decrease its counter until zero.

The Misra-Gries algorithm finds all true most frequent items which occur more than $n/(k+1)$ times, but not all elements in the summary are necessarily the most frequent items: the summary can contain some *false positives*.

As it is proved in [55, 41], it is possible to achieve an approximation guarantee which is called ϵ-approximation. If we run the Misra-Gries algorithm with the size of the summary equal to $k = (1/\epsilon)$, we approximate the value of any frequency by at most ϵn.

The first step to obtain a first level summary makes use of the algorithm 3 and it is described in the algorithm 4. It reads incoming elements and groups them according to their event-time. Within each group, the standard Misra-Gries algorithm is applied to build the summary which is first packed into a JSON record and later sent to the distributed topic between the first and the second layer.

Algorithm 4 First level of aggregation. Application of the Misra-Gries summary.

```
1: procedure MERGING PROCEDURE
2:     group elements in time windows according to event-time
3:     for each time window do
4:         apply Misra-Gries algorithm as in algorithm 3
5:         pack resulting summary in a JSON record
6:     send computed records to output topic
```

The merging procedure is applied in the second level of aggregation. It is taken from [41] and follows algorithm 5. The algorithm merges a summary S with input summaries S_i. The input summary and the summary in the buffer are combined by summing up counters with the same key. Then, the $k+1$-th largest element is subtracted by all elements in the summary. Finally elements with a non positive counter are deleted from the summary.

The procedure, as it is defined in [41], will produce a summary with the same property of the input summaries, including a similar error bound. The error in the merged summary is the sum of the underestimations for the input summaries.

The second step to obtain a global summary makes use of the algorithm 5 and it is described in the algorithm 6. It is executed in the *merging application* and it is similar to what happens in the first level of aggregation. The difference is that the input records are summaries and the algorithm applied after grouping summaries is the algorithm 5.

Algorithm 5 Second level of aggregation. Merging procedure.

```
1: procedure MERGING MISRA-GRIES SUMMARIES
2:     $S \leftarrow \emptyset$                              ▷ set of pairs, global summary
3:     for each incoming Misra-Gries summary $S_i$ do
4:         $S \leftarrow S + S_i$                        ▷ $S_i$ is combined with summary $S$
5:         $r \leftarrow (k+1)$-th largest counter in $S$
6:         for each $j \in S$ do
7:             $c_j \leftarrow c_j - r$
8:             if $c_j < 0$ then
9:                 $S \leftarrow S \setminus \{j\}$
```

Algorithm 6 Second level of aggregation. Application which merges Misra-Gries summaries.

```
1: procedure MERGING APPLICATION
2:     extract the start time of the time window
3:     group elements in time windows according to the start time of the time window
4:     for each time window do
5:         apply merging procedure as in algorithm 5
6:         pack resulting summary in a JSON record
7:     send computed records to output topic
```

Chapter 5

Implementation

5.1 Kafka as publish-subscribe and storage system

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

Kafka acts also as storage system by 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 the JSON format which "defines a small set of formatting rules for the portable representation of structured data" [56]. The key idea in this aspect is the *portability* because in this way any kind of device or application can publish data on a given topic provided that it contains coherent data to be published on 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 the producer of this record.

A first level summary and a global summary instead contain more fields according to the computed aggregates which will be discussed in the sections 5.3 and 5.4.

5.2 Query and methods

The programming language employed for developing is Scala in its version 2.11. The *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 Spark API called *window* which, together with the *groupBy* function, automatically groups all incoming records in windows. The *groupBy* function groups records according to the time windows obtained by the window function. These APIs group 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 section 5.1). The time is passed as parameter to the *window* function and is used to decide which window that record should be appended to.

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 20, fig. 3.4) and on which transformations can be applied. 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 [57] 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.

Once a *streaming query* is defined by defining operations and transformations on a data stream, the *start* function is called to actually start the computation. This is the Spark behavior in streaming processing called *lazy evaluation* which allows, before the execution of the code, to internally optimize it and to check any error in the definition, while postponing the execution to the time it is required.

The query applied on any input will generate a *result table* every trigger interval, which is the time that the application will wait before processing the most recent incoming data which arrived before the last trigger. If the trigger interval is not defined, the query will execute the update of the result table every time a new data is available. In our case, trigger interval is set to one minute.

A further variable to chose before starting the streaming computation is the *output mode*, which defines what to write to the external output. In our case we chose the *append output mode* to write to the external storage only new rows that are appended to the *result table*. In particular, the external output is Apache Kafka.

5.3 Multi-layer 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 and the id of the producer of that record.

All of them are values of particular fields in the JSON data structure.

To compute the first level summary, the input records are grouped according to their *event-time*. Then, the average of the input *value*s is computed by exploiting a built-in aggregate function which computes the average of the values in each group. The resulting summary is composed by a time window, the count and the sum of all elements and the average of all numbers whose event-time is included in that time window. The id of the cluster is also added to identify who has produced this summary. Snippet 5.1 shows the piece of code to get summaries.

Listing 5.1. Grouping input records by *event-time*. Computation of sum, count and average values 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(sum("value"), count("value"), avg("value") as
     "average")
  .withColumn("clusterId", lit(clusterId))
  .start()
```

5.3.2 Second level of aggregation

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

This application reads input data and groups records by the starting time of the time window in the record itself. After grouping, summaries in each group are aggregated by computing the average by summing up values for the *sum* column, for the *count* column and then by dividing those values between them to obtain the actual average. The division is performed by defining a UDF (User Defined Function) which takes two values and divides them. Snippet of code is in 5.2).

Listing 5.2. Grouping summaries coming from clusters in time windows. The 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 getAvg = udf( (sum:Long,count:Int) =>
   {sum.toDouble/count.toDouble} )
```

```
val globalSummaries = summariesFromClusters
  .groupBy(window($"windowStart", windowLength,
     slideInterval))
  .agg(sum("sum"), sum("count"))
  .withColumn("aggregatedAverage", getAvg($"sum", $"count"))
  .start()
```

5.4 Multi-layer Misra-Gries algorithm

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

Implementing a UDAF requires the creation of a class (*UserDefinedAggregateFunction*) and definition of its members. The most important members to define are *input*, *output* and *buffer schema*s, 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, the update function is called for each record to update the buffer, and the evaluate function is called at the end of the aggregation to prepare output data which should have the schema defined in the output schema.

5.4.1 First level of aggregation

The Misra-Gries algorithm is applied in its standard version in the first level of aggregation similarly as it was running in a non-multilayer configuration. Input records are grouped according to their event-time. The 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 the *tumbling* window method is chosen: in this way summaries do not overlap in time, allowing second level of aggregation to easily merge summaries.

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"))
```

The final records have a time window, the size of the summary and the summary itself. The id of the 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

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 a 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 the Misra-Gries algorithm is applied as it is described in the algorithm 3 in the section 4.4.2. Values in the map is updated by adding elements, increasing, decreasing and pruning when needed.
- *Evaluate function.* The 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, the input records are the output records of the first level of aggregation. It contains a time window, the size of the summary, the summary itself and the id of the cluster which generates the summary.

Input records are grouped according to the start time of the time window in each record. Records in each group are aggregated according to a UDAF which generates the summary out of the input records. The UDAF in this application follows the algorithm in [41].

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

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

UDAF - Merge Summaries

The input records are summaries which need to be merged. The summaries coming from the same cluster will refer to non overlapping time windows in order to simply merge two summaries in the UDAF following the 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 a parameter together with a 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 clusters 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, the time window in buffer is set to the time window in the merging application. This time window is used to select which summary to aggregate. The algorithm 5 in the section 4.4.2 is applied to merge two Misra-Gries summaries. Then the resulting summary is stored back into the buffer.
- *Evaluate function.* The global summary is prepared to be outputted. The output is a JSON object made of a time window, a 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 two architectural configurations to obtain evaluation results. The considered architecture configurations are: distributed and geographically multi-layer distributed configurations.

6.1 Metrics

For each configuration we computed two metrics, namely:

1. *Job time*: the time that it is spent to execute the query on the newly received data since the last trigger. A single value for this metric is computed when the processing is triggered, which happens every minute.

2. *Input data size*: the size of input data, which represents the amount of bytes of data received by the application.

6.2 Input data

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

Input data for the Misra-Gries algorithm is taken from real Amazon products reviews in JSON format, from which review texts are extracted and split into words. Each word is 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 [58], and it was used also for the work in [59, 60].

Input data in all configurations enters the system with an average input rate of 870 records per second, which means about 50 thousands records per minute.

6.3 Output visualization

All metrics are collected by using Graphite [61], 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 the start or end of Spark stages or the end of the execution of a streaming query, one or more well-formatted records are sent to Graphite server.

The data stored into Graphite database is then used as source by Grafana [62] which allows a better data visualization than Graphite dashboard by enabling simple visualization and useful operations on time-series data.

6.4 Setup

The experiments are not really executed over a geographically distributed infrastructure, they are executed on two different servers placed in two machines which are close to each other. These servers, that we call *sky1* and *sky2*, come with a similar hardware specification. Each server contains 2 six-core processors, the Intel® Xeon® CPU, model X5660, which is running at a maximum frequency of 2.80 GHz. The total amount of available CPUs are 24, among physical and logical ones provided by Hyper-Threading. The total amount of main memory is 72 GB, which is built starting from different DIMM DDR3 memories with a memory speed of 1333 MHz each. The network adapter supports transfer rates until 1000 Mbps.

The distributed configuration is set up by running a single Spark cluster over *sky1* and *sky2*. This cluster is made of a total of 4 workers, 2 on *sky1* machine and 2 on *sky2* machine. Each worker is assigned 1 GB of memory and 16 CPU cores. The Kafka cluster is placed in the same machines, one Kafka cluster per machine.

The multi-layer configuration is built first by running two different clusters, one on *sky1* and the other one on *sky2*. Then, because of the impossibility to use a third machine where to run a third cluster and the merging application, the Spark cluster running on *sky2* machine is used to execute also the merging application. In order to reduce as much as possible the interference between the applications which logically belong to different layers (and clusters), more resources have been assigned to the workers of this cluster. The Spark cluster running on *sky1* is assigned 2 workers whereas the Spark cluster running on *sky2* is made of 4 workers. Every worker is assigned 1 GB of memory and 16 cores. Moreover, in both clusters the maximum amount of CPU cores to request for an application from across the cluster is 16. In this way all applications will obtain the same amount of CPUs and, in the case of *sky2*, they can run together without stealing resources to each other.

6.5 Results

When presenting the results for the multi-layer configuration in both algorithms, metrics and graphs for the applications running on the edge layer are not shown because they are comparable with the ones gathered for the distributed configuration since they run on comparable clusters. In this section the graphs are presented and showed. Later in the section 6.5.3 an organic evaluation is lead by referring to the following graphs.

6.5.1 Job time

Computation of average

The figure 6.1 represents the job time for the computation of the average. The first graph is referring to the distributed configuration, the second one to the multi-layer configuration. Every data point represents the job time in seconds (milliseconds). The behavior of the job time over the time is reported.

Misra-Gries algorithm

The figure 6.2 represents the job time for the Misra-Gries algorithm. The first graph is referring to the distributed configuration, the second one to the multi-layer configuration. As in the computation of the average, every data point represents the job time in seconds. The behavior of the job time over the time is reported.

6.5.2 Communication amount

Internal communication

Considering the amount of data exchanged, the communication amount inside a cluster when running an application is also shown in figure 6.3. This amount of data gives an idea about the data that is moving between nodes of a Spark cluster in the shuffle phase. During this phase data moves among worker to be partitioned for computing aggregated results and, therefore, it is transmitted. Internal communication data refers to the amount of data when running Misra-Gries algorithm, during which more data is being moved.

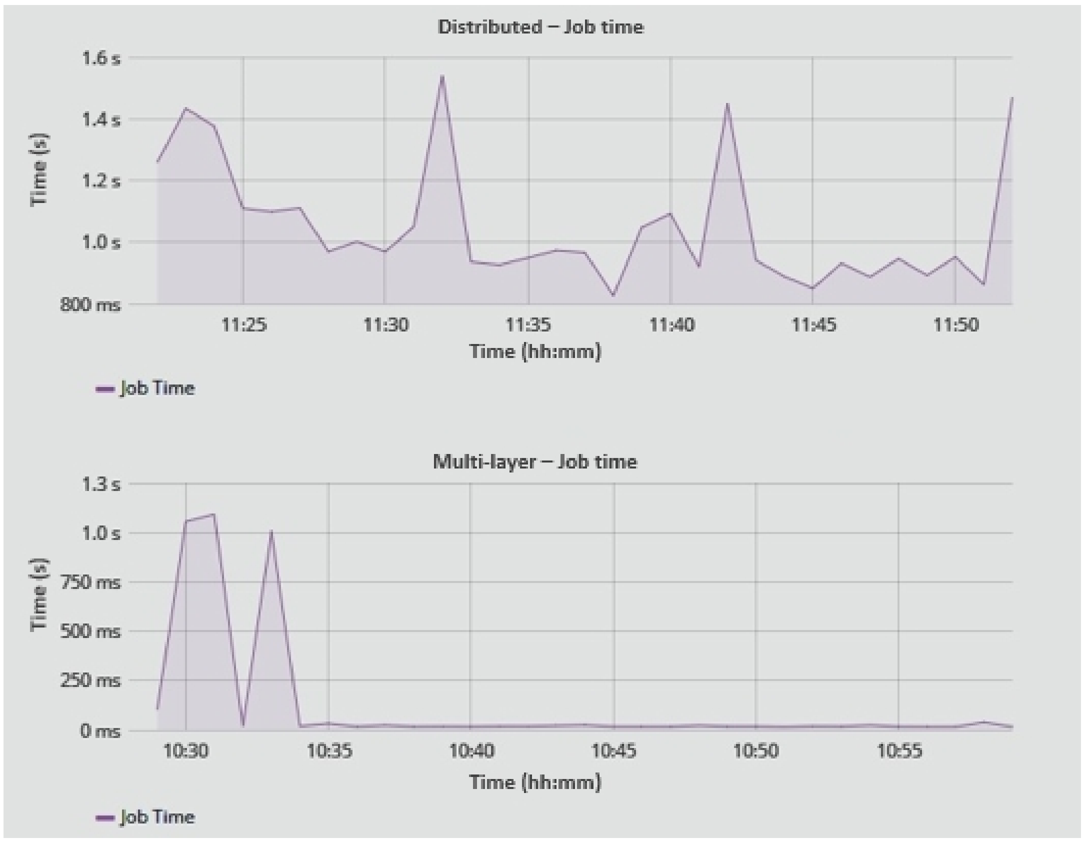


Figure 6.1. Job time. Graphs for the computation of the average.

Computation of average

The figure 6.4 shows the amount of data that an application running the algorithm to compute the average is reading from the Kafka topic as input. In the first row there is the amount of data read in the distributed configuration, in the second row there is the amount of data read in the multi-layer configuration considering only the reconciliation place, where the merging application is running. This is the total size of the summaries that are moving from the edge layer to the reconciliation layer.

Misra-Gries algorithm

The figure 6.5 shows the amount of data that an application running the Misra-Gries algorithm is reading from Kafka topic as input. This amount of data is represented by the summaries produced by the edge layer. As for the graphs for the computation of the average, the first row is the amount of data read in the distributed configuration and the second row represents the amount of data read in the multi-layer configuration. The amount o data in the second row is referring only

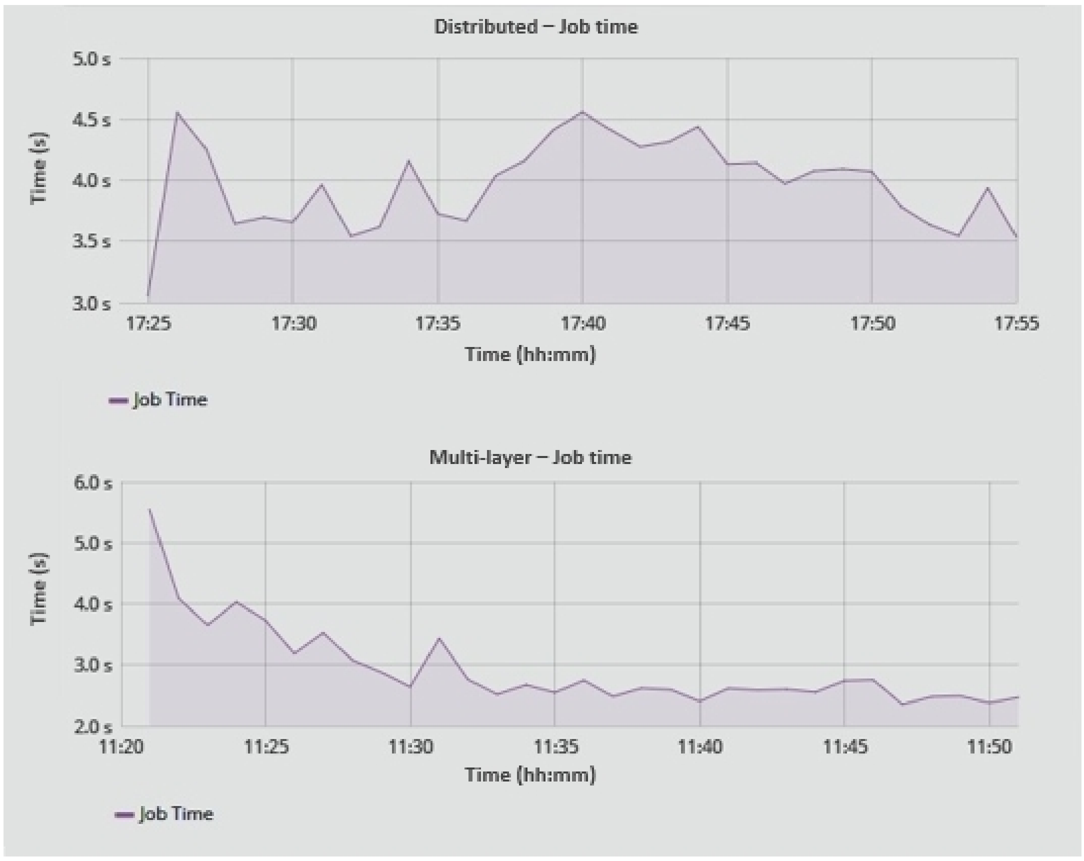


Figure 6.2. Job time. Graphs for the Misra-Gries algorithm.

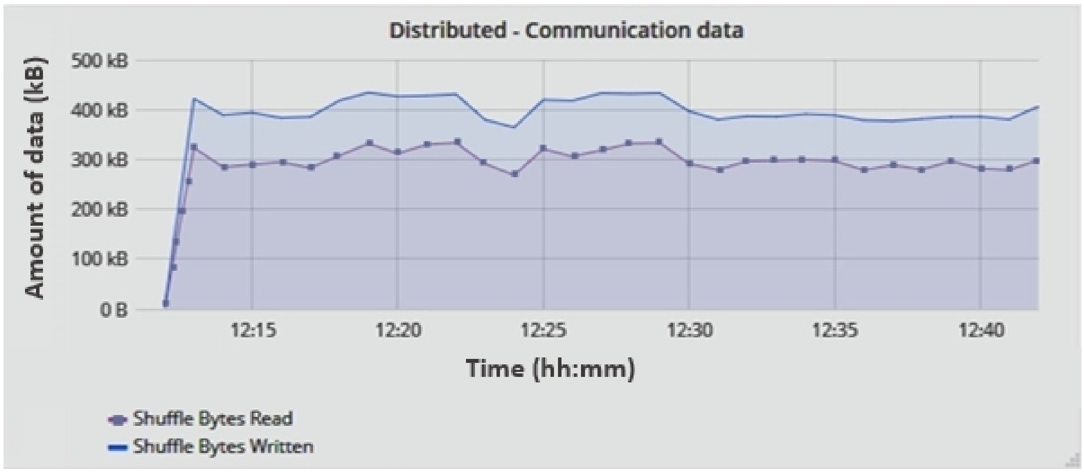


Figure 6.3. Graphs for communication data inside a cluster.

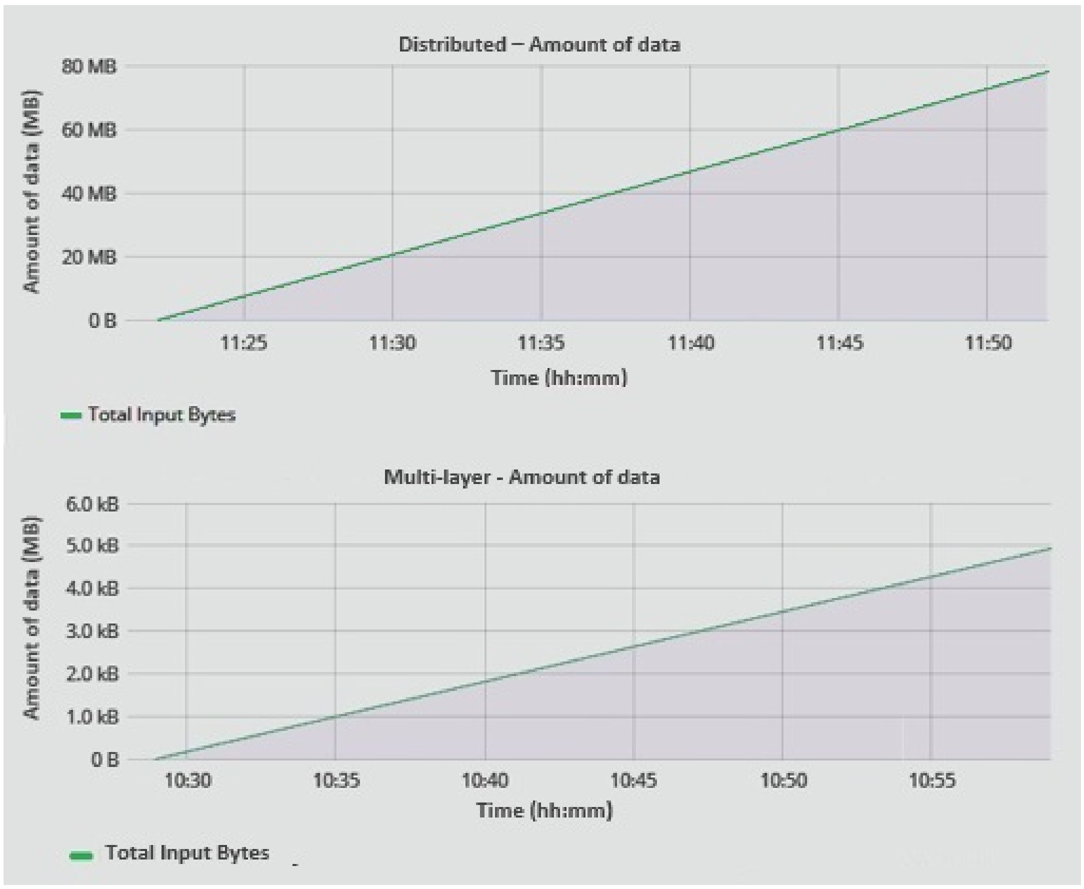


Figure 6.4. Communication amount. Graphs for the computation of the average.

to the merging application because the amount of data which is read corresponds to the amount of data the is moving from the edge layer to the reconciliation place passing through the network.

6.5.3 Evaluation

Job time

The *job time* for the computation of the average in the two configuration is significantly different as shown in figure 6.1. The distributed configuration reports an average job time of 1.052 second, whereas in the multi-layer configuration the average job time is reduced by approximately 10 times with a value of 122 milliseconds. The average job time in the multi-layer configuration is affected by the first transitory behavior with higher values. In fact, the job time seems to settle later to values between 10 and 20 milliseconds. If we consider only the average job

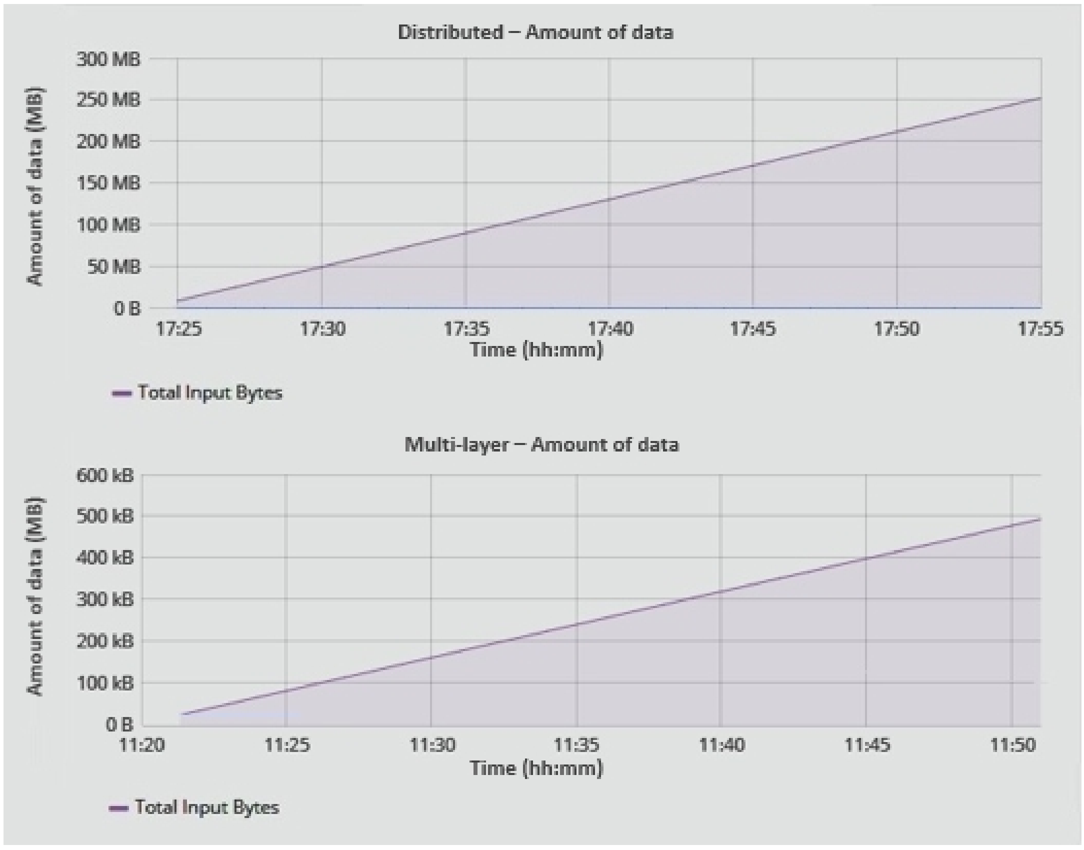


Figure 6.5. Communication amount. Graphs for the Misra-Gries algorithm.

time, the job time is reduce by 88% for the multi-layer configuration with respect to the distributed configuration. If we consider the job time without considering the transitory part, the reduction is by a factor of 50.

The *job time* for the Misra-Gries algorithm is also decreased in the experiment with the multi-layer configuration with respect to the distributed configuration. The behavior is showed in figure 6.2. In the distributed configuration the average job time is 3.970 seconds, whereas in the multi-layer configuration the average job time is 2.939 seconds.

The reduction in the average job time for the Misra-Gries algorithm is by 25% and it is not as much significant as in the computation of the average because the processing in the merging application involves a User Defined Aggregate Function which is computationally intensive. The computation of the average instead exploits mainly built-in aggregate functions like the *sum* and the *count* which are simpler and are optimized as part of the Spark SQL libraries.

Communication amount

Regarding the computation of the average, figure 6.4 shows significant differences for the *input data size*. The distributed configuration reads, at the end of the experiment, almost 80 MB and the multi-layer configuration reads almost 5 KB in total. In this experiment, the amount of data which is read by the merging application and that is moving from edge layers across the network is reduced by more than 99% with respect to the total amount of data read by the distributed application, more concretely, from dozens of megabytes in few minutes to few kilobytes.

The *input data size* is reduced also in the experiment where the Misra-Gries algorithm is running, as showed in the figure 6.2. In the distributed configuration the amount of data that is read by the application is around 250 MB, whereas in the multi-layer configuration the merging application reads around 500 KB. This experiment shows a significant reduction in the amount of data that would cross the network. The amount of data which would move in the multi-layer configuration is reduced by more than 99%, more concretely from hundreds of megabytes to few kilobytes.

The differences in the measurements between the two configurations are due to the first aggregation performed in the edge layer. This step reduces the data size by preserving only a small statistic or a summary regarding the input data. Only the computed summary is sent over the network in the multi-layer configuration and this explains the reduction of the amount of data that is read by the applications and, therefore, that are moving across the network. In the case of the multi-layer configuration, the reported measurement of the amount of data is the only data that is moving from the edge layer to the reconciliation place.

Error in the aggregated summary

In the case of the computation of the average, the error in the output summary computed by the merging application is zero because it relies on the exact values for the sum and the count of the elements for each cluster.

For the Misra-Gries algorithm, the summary generated by the merging application does not report the exact frequencies, but the error is bounded as it was explained in the section 4.4.2. The elements with highest values of frequencies for each cluster, and not all k elements inside each summary, are shown after the merging procedure. Though we are not able to exactly define the frequencies for the most frequent elements, we can obtain a view of the most frequent elements in all the clusters.

Limitations and drawbacks

We should consider that the experiments are executed with an edge layer made by only two clusters. This could be a limitation since real use cases can have several geographically distributed clusters.

By having more clusters in the edge layer, more than two summaries are produced and sent to the reconciliation place to be merged. If the merging application receives more summaries, the job time and the amount of data received increase. Indeed, it is not expected a significant increase for the amount of data coming from summaries: a summary can be at most few dozens of kilobytes and we would need hundreds of summaries to reach the same amount of input data that we have sent across the network in the experiments (i.e. 250 MB or 80 MB). The merging application for the Misra-Gries algorithm can decrease its performance for the time that it spend to apply the merging procedure to all summaries, but this drawback is more preferable than allowing a lot of data to move from the sources to a single processing place.

The designed architecture is called *multi-layer* because more than two layers can also be deployed. Adding a further layer is possible but comes with advantages as well as drawbacks. A further layer can be needed, for example, when there are a lot of data sources over a small area a first aggregation can be performed and then sent to the reconciliation place. First, this layer represents a cost to be deployed, then a further aggregation can produce a summary with a bigger error. Hence, the decision of introducing more than two layers should be the result of a proper analysis of costs and error of the summary representation of the input data.

The main disadvantage in deploying such an architecture and running an algorithm to aggregate input data in different steps is the accuracy in the representation of the input data. If we do not consider simple algorithms like the computation of a simple aggregated value, such as the average, the computation of a data synopsis by exploiting summarization algorithms leads to a representation of the whole input data which is only an approximation. In this case, we are not able to obtain an exact global aggregated value because the last step of aggregation is not in touch with the real data and relies only on the summaries computed in the edge layer. The summarization is needed because the input data comes with high input rates and with high volume. Hence, in order to give a representation of such a data, in some cases an approximation is needed.

Moreover, we are not able to include the data as soon as it enters the system into the global aggregate. The time that it is needed to be aware that a record entered the system and that contributes to the evaluation of the global aggregate is related to the size of the time range in the last level of aggregation. The smaller the time window in the merging application is, the smaller the time that a data actually is included into the final aggregate. This behavior is not suitable for applications for which a quick reaction is required after the input data has been read.

Chapter 7

Conclusions

7.1 Discussion

This thesis project has achieved two important results. Firstly, a geo-distributed multi-layer architecture has been designed with regard to possible algorithms to be implemented on it. The final design is the result of the research conducted by exploring several tools with different features and capabilities. A consistent and coherent solution has been then characterized by choosing the most suitable tools among them. The provided architecture is a flexible solution where Apache Spark with Structured Streaming 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. The features, the properties that the algorithms should have and the guidelines that they should follow to run on the designed architecture have been described. The algorithms which have been dealt are the first to apply when there is the need 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 on the designed architecture. First, the computation of the average has been implemented, then the Misra-Gries algorithm has been chosen.

The system is able to compute an aggregate by performing subsequent aggregations on two levels. The first level of aggregation generates summaries, while the second one merges 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. The aggregation and the summarization prevent a huge amount of data to reach the reconciling application by crossing the network and allow to limit this flow of data to small summaries. This behavior can save bandwidth costs and then money, which is the main motivation behind this thesis project.

7.2 Future work

It is left for the future work to test intensively the implemented applications by horizontally scaling the architecture with multiple geographically distributed autonomous clusters. The idea is to allow the merging application to read more than two summaries and reconcile them. In such a situation, the actual input data size would be really high as well as the 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 the architecture has been already mentioned in the section 6.5.3 but it has not been dealt in this work. The architecture can be vertically extended by having more levels of aggregation. The first level application can produce an aggregate which is read by a second level application. The second level application is reading, as the actual second level of aggregation does, only summaries and then generates merged summaries for the last level of aggregation. The second and the last level of aggregation can follow the same algorithmic logic since both of them can rely only on summaries and not on input data and since the summary could have the same format. A multi-layer system, with more than two layers, can be useful when the second level of aggregation is still close to the sources but sources produce a huge amount of data. The 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. Architectures with different numbers of layers can be compared.

A further experiment left for future work is the usage of a different tools 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.

This work aims also to project the streaming processing towards the decentralization. The architecture cannot be considered to be decentralized because all clusters do not communicate directly with each other, but by using a layered structure: a proper layer is responsible to gather messages coming from each cluster and obtain a global view of the input data. A possible work in this direction can be to exploit some of the tools considered in the section 3.2.1, as well as others, to make the clusters able to exchange information with other clusters, still with the ultimate intent to keep low the amount of data exchanged among them. A first idea is to try to use Apache Ignite or NiFi.

Bibliography

[1] "Spark Streaming - Spark 2.2.0 Documentation." [Online]. Available: https://spark.apache.org/docs/latest/streaming-programming-guide.html

[2] "Structured Streaming Programming Guide - Spark 2.2.0 Documentation." [Online]. Available: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[3] "A gentle intro to udafs in apache spark." [Online]. Available: https://www.jowanza.com/blog/a-gentle-intro-to-udafs-in-apache-spark

[4] "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

[5] "Apache Kafka." [Online]. Available: https://kafka.apache.org/intro

[6] "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

[7] D. Kossmann and N. Tatbul, "Introduction to big data—the four v's." [Online]. Available: http://www.dbs.ifi.lmu.de/Lehre/BigData-Management&Analytics/WS15-16/Chapter-1_Introduction_to_Big_Data.pdf

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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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%2EC42B82B87617960C&CFID=840547315&CFTOKEN=12027865&__acm__=1513438595_45fd2528fd7bca1d6301feee8220cf0e

[14] 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

[15] 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

[16] 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.

[17] “SGI UV™ 300h system specifications.” [Online]. Available: https://www.sgi.com/pdfs/4559.pdf

[18] 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

[19] 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

[20] . 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

[21] H. C. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 2014. [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

[22] 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

[23] 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

[24] 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

[25] 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

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

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

[28] 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.

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

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

[31] 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.

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

[33] 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.

[34] 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

[35] 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.

[36] P. Teli, M. V. Thomas, and K. Chandrasekaran, "Big data migration between data centers in online cloud environment," *Procedia Technology*, vol. 24, pp. 1558–1565, 2016.

[37] 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.

[38] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, "Spanedge: Towards unifying stream processing over central and near-the-edge data centers," in *Edge Computing (SEC), IEEE/ACM Symposium on*. IEEE, 2016, pp. 168–178.

[39] R. Dobrescu and F. Ionescu, *Large Scale Networks: Modeling and Simulation*. Crc Press, 2016.

[40] 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.

[41] 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. [Online]. Available: http://www.cs.ust.hk/~yike/pods12-mergeable.pdf

[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

[47] "Structured Streaming In Apache Spark," Jul. 2016. [Online]. Available: https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html

[48] "Open Source In-Memory Computing Platform - Apache Ignite." [Online]. Available: https://ignite.apache.org/

[49] "Collocated Processing - Apache Ignite." [Online]. Available: https://ignite.apache.org/collocatedprocessing.html

[50] "Apache Toree." [Online]. Available: https://toree.apache.org/
[51] "Apache NiFi Overview." [Online]. Available: https://nifi.apache.org/docs.html
[52] "Apache Livy." [Online]. Available: https://livy.incubator.apache.org
[53] 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
[54] "Apache ZooKeeper." [Online]. Available: https://zookeeper.apache.org/
[55] K. Tangwongsan, S. Tirthapura, and K.-L. Wu, "Parallel streaming frequency-based aggregates," in *Proceedings of the 26th acm symposium on parallelism in algorithms and architectures.* ACM, 2014, pp. 236–245.
[56] T. Bray, "The javascript object notation (json) data interchange format," 2017.
[57] Y. Shafranovich, "Common format and mime type for comma-separated values (csv) files," 2005.
[58] "Amazon review data." [Online]. Available: http://jmcauley.ucsd.edu/data/amazon/links.html
[59] 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.
[60] 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.
[61] "Overview — Graphite 1.1.0 documentation." [Online]. Available: http://graphite.readthedocs.io/en/latest/overview.html
[62] "Docs Home| Grafana Documentation." [Online]. Available: http://docs.grafana.org/