POLITECNICO DI MILANO

Facoltà di Ingegneria

Scuola di Ingegneria Industriale e dell'Informazione

Dipartimento di Elettronica, Informazione e Bioingegneria

Master of Science in

Computer Science and Engineering

A comparative study on parallelization of streaming top-k algorithms in Kafka

Supervisor:

PROF. EMANUELE DELLA VALLE

Master Graduation Thesis by:

LUCA FERRERA

Student Id n. 10690341

Academic Year 2019-2020

ABSTRACT

Continuous top-k query monitoring, which reports the *k* top-scored objects from data streams, is a challenging problem in modern streaming applications due to the high data rates that exceed the data stream infrastructure's capacity. Streaming top-k algorithms try to solve it by optimizing resources. We focus on some of them to select the ones to use for our comparative study. We choose an algorithm that sorts every window's elements and keeps the first *k*s as the top-k result; also, we select the MinTop-K algorithm [16], an excellent resource optimization example.

We opt for Kafka as the distributed streaming platform where to implement them. Kafka is one of the most used distributed streaming platforms, so we want to understand if we can leverage it to implement streaming top-k algorithms. We develop both centralized and parallel algorithms using Kafka Streams and Processor APIs.

Moreover, we want to understand if it is worth parallelizing streaming top-k algorithms and the relationship between the top-k parameter, *k*, and the KPI we measured. As KPI, we use the time needed to compute top-k results for a fixed amount of streaming data.

To answer our questions, we perform a comparative study. We compare the centralized and parallel versions' KPI to understand if we can achieve better performance parallelizing the algorithms, playing with the parallelization degree. We compare the algorithm's execution with different top-k values to study the correlation between the chosen KPI and the *k* parameter.

The results show considerable improvements in the parallel versions over the centralized ones in both algorithms. Moreover, we find out that large enough top-k values significantly influence the experiments' total time, showing exponential growth, while small *k* values almost do not affect the observed KPI. Our implementations exploit Kafka Streams and Processor API, candidating Kafka to be an excellent solution for continuous top-k query monitoring.

ESTRATTO

Il monitoraggio continuo delle query top-k, che riporta gli oggetti top-scored provenienti dai data stream, è un problema impegnativo nelle moderne applicazioni di streaming a causa delle elevate velocità di trasmissione dei dati che superano la capacità dell'infrastruttura. Gli algoritmi di streaming top-k cercano di risolverlo ottimizzando le risorse. Abbiamo selezionato un paio di questi algoritmi per il nostro studio comparativo. Abbiamo scelto un algoritmo che ordina gli elementi di ogni finestra e mantiene i primi *k*; inoltre, abbiamo selezionato l'algoritmo MinTop-K [16], un eccellente esempio di ottimizzazione delle risorse.

Abbiamo optato per Kafka come piattaforma di streaming distribuito perché è una delle piattaforme di streaming distribuito più utilizzate, e abbiamo voluto capire se fosse possibile sfruttarla per implementare gli algoritmi di streaming top-k. Abbiamo sviluppato sia algoritmi centralizzati che paralleli utilizzando Kafka Streams e le Processor API.

Inoltre, abbiamo considerato se valesse la pena di parallelizzare gli algoritmi di streaming top-k e la relazione tra il parametro top-k, *k*, e il KPI che abbiamo misurato. Come KPI, abbiamo utilizzato il tempo necessario per calcolare i top-k per una quantità fissa di dati in streaming.

Per rispondere alle nostre domande, abbiamo eseguito uno studio comparativo. Abbiamo confrontato i KPI delle due versioni per capire se fosse possibile ottenere migliori prestazioni parallelizzando gli algoritmi, modificando il grado di parallelizzazione. Abbiamo confrontato l'esecuzione dell'algoritmo con diversi valori di *k* per studiare la correlazione tra il KPI scelto e il parametro *k*.

I risultati hanno mostrato notevoli miglioramenti nelle versioni parallele rispetto a quelle centralizzate. Inoltre, abbiamo scoperto che valori di top-k abbastanza elevati influenzano significativamente il KPI misurato, mostrando una crescita esponenziale, mentre valori di *k* piccoli quasi non influenzano il KPI osservato. Le nostre implementazioni sfruttano Kafka Streams e le Processor API, candidando Kafka ad essere una soluzione eccellente per il monitoraggio continuo delle query top-k.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

In this chapter, we show an overview of our work. In Section 1.1, we introduce the motivations that led to this thesis. Section 1.2 describes the contributions we offered. Finally, in Section 1.3, we show the thesis' outline.

1.1 MOTIVATION

We are in the Data era, where an enormous amount of data flows into systems and needs to be analyzed.

Real-time and near-real-time applications are crucial to analyze these data streams and extract useful insights. Examples of industries that use this kind of application include telecommunication, finance, smart energy, security, manufacturing, and many others. The streaming top-k monitoring applications give an emerging subset of these applications. For every window over a stream of data, they aim to compute the top-k elements based on a scoring function. Due to data streams' velocity and volume, streaming top-k applications need to optimize their resources, such as memory, CPU, storage, and communication, to meet their requirements, i.e., high throughput and low latency.

In the top-k monitoring of data streams, only a portion of data is valuable; this property can be exploited to sensibly manage storage and memory resources as Yang et al. made in the MinTop-K algorithm[16], that is deepened in Subsection 2.2.2. Moreover, streaming top-k applications could benefit from parallelization and distribution too. Moving the application from a centralized monolith to a parallelized application allows splitting the workload, optimizing storage, and computing resources.

The problem with state of the art streaming top-k algorithms is that they have been implemented as experimental proof of concept or to validate technologies in the lab. So they can not be used for real-world applications yet. Bringing these algorithms inside real-world streaming systems would require studying how to exploit systems' functionalities and properties to implement them. This process is crucial to benefit from these researches in modern industries' applications.

A distributed streaming platform as Kafka can be exploited to implement those algorithms. Kafka offers a stream processing system, Kafka Streams, that provides both existing operators for continuous and reactive computation on data streams and the flexibility to build new ones using the Processor API. On top of Kafka Streams, they build KSQL, a declarative streaming language, to write continuous queries over data stream in a SQL-like syntax. Unluckily, KSQL does not provide any **ORDER BY** clause and the **TOPK** operator computes the top-k using only one column, resulting in a useless operator for a top-k query where the scoring function takes as input more than one value associated with an object.

As we mentioned before, bringing streaming top-k algorithms into real-world systems is not trivial. Firstly, we need to study the system to learn how to exploit its properties and functionalities to implement these algorithms. In the case of Kafka Streams, are there any operators we can use? Or should we build a new one from scratch? Moreover, we said that parallelization and distribution could help optimize the applications' performance; distributed systems are by their nature prone to distribution, so how can we exploit this property to parallelize streaming top-k algorithms? Is it worth parallelizing algorithms in Kafka Streams application? Finally, different parameters influence streaming top-k applications' performance; examples of these parameters are the window *size* and *hopping_size*, the top-k value, *k*, or the number of parallel instances to use. Which are the correct values for these parameters? Which ones most influence the performance?

1.2 CONTRIBUTIONS

We decided to implement two different algorithms using Kafka Streams as a baseline for the implementation of KSQL operators for streaming top-k queries on top of them. We decided that a comparative study would have been the best way to explore the benefits of parallelizing streaming top-k algorithms and to analyze how algorithms and applications' parameters could influence performance. We opted to implement a naive algorithm that materializes the score of each object that comes during a data stream's window, sorts the objects based on their score, and keeps only the first *k* elements as the top-k result. Then we opted for the MinTop-K Yang et al. algorithm, as it is a more complex one that aims to optimize memory and CPU resources, leveraging on the prediction of future top-k results.

We exploited the Processor API offered by Kafka Streams to implement our algorithms. We used *State Stores*, which are key-value pairs associated with each instance of a Kafka Streams application, to keep track of the records arrived at each window and to manage the data structures.

We wanted to question the benefits of parallelizing streaming top-k algorithms, so we implemented a centralized and parallel version to compare their performance. Moreover, we ran experiments with different degrees of parallelization to find out how they influence performance.

Finally, we designed experiments to understand how the algorithms and applications' parameters could affect the overall application's performance. In each experiment, we use the total time as KPI, meaning the application's time to compute the top-k results for each window of the input data stream. As cited before, we ran experiments varying the degree of parallelization, so the number of parallel instances. Moreover, we played with the top-k parameter to understand if there is a correlation between its value and the KPI we measured.

1.3 DOCUMENT STRUCTURE

The rest of the document is organized as follows:

- **Chapter 2: State Of The Art -** which is divided into three parts. The first part describes the stream processing technique, the systems, and the requirements they should meet. In the second section, we explain the streaming top-k problem and the state of the art solutions. Then, we show the algorithms we choose to implement. The third part presents the Kafka ecosystem, describing Apache Kafka, Kafka Streams, the Processor API, and KSQL.

- **Chapter 3: Problem Setting -** which describes the problem that led to this thesis. Firstly, it shows the motivation that pushed us through this work; then, it focuses on the main problems behind implementing streaming top-k algorithms, such as how to exploit the existing streaming platform or how the different parameters influence the applications' performance. Finally, it states the research questions we aim to answer with our work.

- **Chapter 4: Parallel Streaming Top-k in Kafka -** which shows the implementation process, from the design to the code. Its first section describes the algorithms' design and explains how they work with the help of pseudo-code. The second section shows the application's infrastructure, while the last section presents the implementation experience, explaining the implementation choices with their rationale. Also, it presents some significant code fragments explained in detail.

- **Chapter 5: Experiments & Results -** which describes the experimental setup we used for the comparative research and a general study on the costs for both centralized and parallel versions. Then, it shows the design of experiments grouped by the different algorithms used. Finally, it describes the experiments' results group by the research questions they aim to answer and the finding we obtained from the results.

- **Chapter 6: Conclusions & Future Works -** which discusses the conclusion based on the experiments' results and proposes future research directions for this work.

2 STATE OF THE ART

In this chapter, we show the state of the art regarding stream processing in Section 2.1 and streaming top-k in Section 2.2. Section 2.3 describes the Kafka ecosystem for stream processing.

These are the bases where we build our work to implement parallel streaming top-k algorithms using Apache Kafka as a distributed streaming platform.

2.1 STREAM PROCESSING

Stream processing is the technique that analyzes data streams in real-time. A *data stream* is a continuous append-only sequence of immutable *records*[1] [5]. Stream processing applications usually require real-time processing of huge volumes of data and limited memory consumption, but these are not the unique requirements they need to meet. Stonebraker, Çetintemel, and Zdonik presented a list of requirements that stream process systems should meet [15]:

Keep the Data Moving - To achieve low latency processing, the system should process messages "in-stream", without the need to store them to perform any operations.

Query using SQL on Streams (StreamSQL) - Systems should implement a query mechanism to retrieve information or compute analytics result on streamings. It could be a high-level language like SQL with built-in extensible operators.

Handle Stream Imperfections (Delayed, Missing, and Out-of-Order Data) - Systems should provide mechanisms to tackle out-of-order and missing data, which are present in data streams.

Generate Predictable Outcomes - Systems should present deterministic outcomes, predicting the data stream in a repeatable way.

Integrate Stored and Streaming Data - Systems should support state information storage to integrate partial aggregator results with streaming data. The language used to deal with both data stream, and state information should be unique.

Guarantee Data Safety and Availability - Systems should provide high availability and high fault tolerance.

Partition and Scale Applications Automatically - To support the high volume of data and to meet the low latency requirement, systems should be able to distribute the workload reaching incremental scalability; preferably, this process should be transparent to the user.

Process and Respond Instantaneously - Systems should have an engine that optimizes the execution plan, with minimal overhead, resulting in real-time responses.

There are three main classes of systems that try to solve the problems related to low-latency and high-volume context. **Rule Engines** operate following a condition/action paradigm on the input stream. **Database Management Systems** (DBMS) can store and query huge volumes of data, solving problems related to high-volume; in-memory processing can tackle low-latency related problems. **Stream Processing Engines** (SPEs) operate on data as soon as they arrive without storing it for processing, and they typically work with **Data Stream Management Systems** (DSMS). DSMS usually uses SQL-like language to perform stream processing. Aurora [1] is probably the first SPEs, it is an example of how the *human-active, DBMS-passive* (HADP) model shifts to the *DBMS-active, human-passive* (DAHP) model to cope with the stream processing's problem we mentioned before. An example of DSMS is STREAM [3], it is a general-purpose DSMS for continuous query answering over data streams and stored relations. STREAM supports CQL [2] a SQL-like query language for registering continuous query against stored relations and data streams.

2.2 STREAMING TOP-K

This section presents the top-k query answering problem; even if researchers have studied it in different domains such as database, Semantic Web, and stream processing, we will focus only on the last one.

Let us begin with two crucial definitions that help to understand the problem better.

Definition 2.1 (Top-k Query)**.** A top-k query gets a user-defined scoring function and outputs only the top-k results with the highest score.

Definition 2.2 (Scoring Function)**.** A scoring function is a function $\mathcal{F}(p_1, p_2, ..., p_n)$ that provides a score for each result of the query aggregating multiple predicates, where p_i is a scoring predicate.

An assumption on monotonicity of the scoring function, i.e., $\mathcal{F}(x_1, ..., x_n) \geqslant \mathcal{F}(y_1, ..., y_n)$, is adopted in most top-k processing techniques. This assumption leads to efficient processing of the top-k query because an upper bound of the unseen object's score can be derived, guaranteeing early top-k processing termination.

Babcock and Olston [4] proposed a solution that avoids transmitting data from a physically distributed location to a centralized server to compute top-k query results on the data stream. Given the enormous amount of data, it can be impractical or expensive to send them continuously to a centralized location. They showed that it is unnecessary to transmit the whole data stream to process these queries. They applied arithmetic constraints at remote streams source on data to ensure that the top-k results remain valid within a user-defined error tolerance.

Distributed communications are necessary only when arithmetic constraints are violated; in this case, the centralized server updates the top-k result in the distributed location and set new constraints.

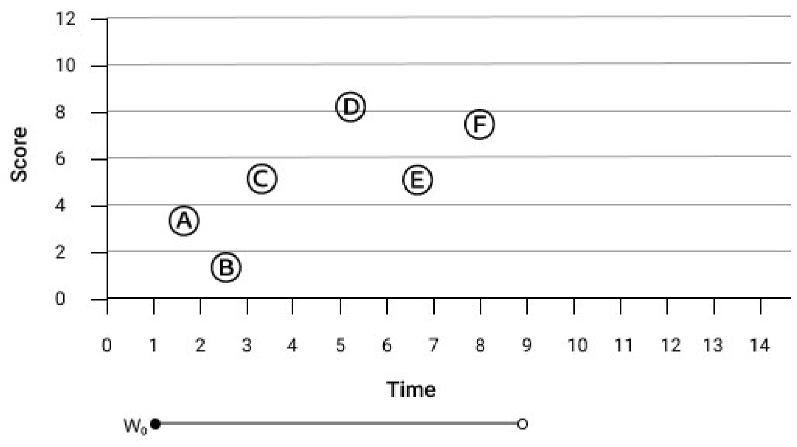
Mouratidis, Bakiras, and Papadias [10] have seen that the database domain's methodologies are inapplicable to stream processing due to the highly dynamic environment. They studied continuous monitoring of top-k query over fixed-size windows, which can be both physical or logical ones, i.e., their size can be expressed based on the number of active items or based on time units. They restricted processing to the workspace's sub-domains that influence some queries; these accurate records are indexed by a grid structure that maintains book-keeping information. They presented two different techniques to process top-k query: the first one compute the top-k result whenever some of the current top-k results expire, the second one achieve better running time precomputing the future changes in the result, as we will see the precomputation of the future result will be used in other techniques too.

The previous researches assume data completeness, meaning that no data is missing nor arrives out-of-order. [8], and [6] deal with incompleteness due to out-of-order arriving, so the missing is *temporary* because as soon as the late record arrives, there will be no missing anymore. [11] deal with *permanently* missing and despise the previous approaches they adopted *differential dependecy* rules [14] to imputed missing data.

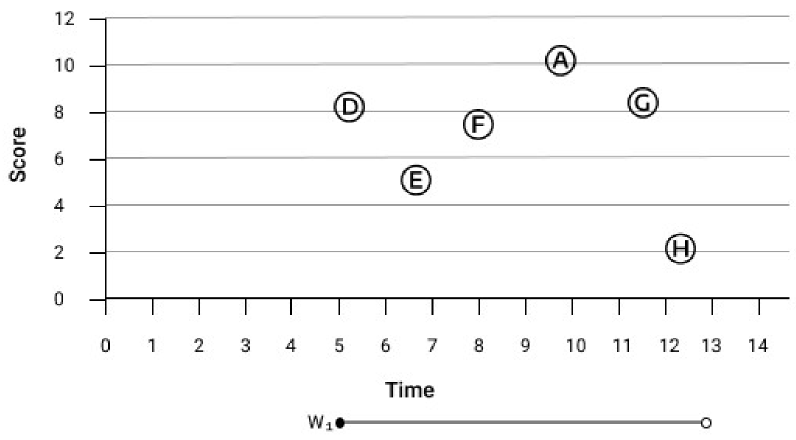
Now that we have presented the theory behind streaming top-k, we show a running example to understand the problem better. Figures 2.1a and 2.1b show a part of a stream between time 0 and 14. The X-axis shows the arrival time of a record in the stream, while the Y-axis shows the record's score computed by a user-defined scoring function. Alphabet letters represent every record in the plot to simplify the figure; we can see them as the record's ID. This data stream is observed using windows with a length of 8-time units and slides every 4-time units.

Figure 2.1a shows the content of window W_0, which starts at time 1 and ends at time 9 (excluded). Figure 2.1b represents the content of window W_1 which opens at time 5, after a slide of 4 time units, and closes at time 13 (excluded).

We can see that during window W_0 records A, B, C, D, E, F arrives, record D and F are the ones with the higher score, so if our goal is to find the top-2 records for a given window, they are the top-2 records for window W_0. When window W_1



(a) Evaluation of window W_0



(b) Evaluation of window W_1

Figure 2.1: Example of windows evaluation

starts records A, B and C are out of it, but the figure shows that A arrives again with a higher score. We can also see that new records G and H come during window W_1. The new score of record A put it in the top-2 for window W_1 together with the new record G.

In the following subsection, we present the algorithm we choose to implement for our study.

2.2.1 *Materialize Score & Sort*

Materialize score and sort is probably the most naive algorithm in the streaming top-k query answering world. First, it materializes the score for each record in the window over the data stream, then it sorts the result by the score, and finally, it keeps only the first *k* results. It is easy to see that this algorithm is costly in terms of time and memory. More specifically, in terms of time, we have three cost terms:

1. Materializing cost: $\mathcal{O}(n)$
2. Sorting cost: $\mathcal{O}(n \log n)$
3. Top-k cost: $\mathcal{O}(k)$

Where *n* is the number of records arrived within a single window, and *k* is the number of outputs for each window. In terms of memory, we have $\mathcal{O}(n)$ because we have to keep in memory all the data inside a window. Improvements in term of CPU and memory complexity has been made by Yang et al. with MinTopk algorithm[16] as shown in Subsection 2.2.2

2.2.2 *MinTopK*

MinTopk algorithm wants to tackle the problem of periodical recomputation of the top-k results, which is computationally expensive and requires keeping in memory all the records inside the query window, which translates to significant memory consumption.

"To solve this problem, we identify the 'Minimal Top-K candidate set' (MTK), namely the subset of data stream that is both necessary and sufficient for continuous top-k monitoring" Yang et al.[16]. They also present the *super-top-k list*, which is a compacted representation of predicted top-k result, and they show the MinTopk algorithm responsible for the maintenance of super-top-k-list.

Here we show in more detail the idea behind MTK; let us consider a window of size *w* that slice every *t*, a record that arrives in a given window will also be part of the next w/t windows. A subset of the top-k result for the current window will also be eligible for the w/t future windows. These predicted top-k results compose the *MTK set*.

The super-top-k list contains all the predicted top-k results; each entry has a *starting_window* and a *ending_window* that indicates in which windows the records can contribute to the top-k result.

A *lower_bound_pointer (lbp)* is associated with every window. It points to the record with the smallest score in the super-top-k list; the *lbp* helps efficiently handle new arrival objects.

Figure 2.2 shows an example of how the MinTopK algorithm evaluates a top-k query over a data stream. Figures 2.2a, 2.2b and 2.2c show a part of a data stream between time 0 and 14, the X-axis shows the arrival time of data items, while Y-axis shows the data item's score. We observe the stream using a window with a size of 5-time units that slide every 3-time units.

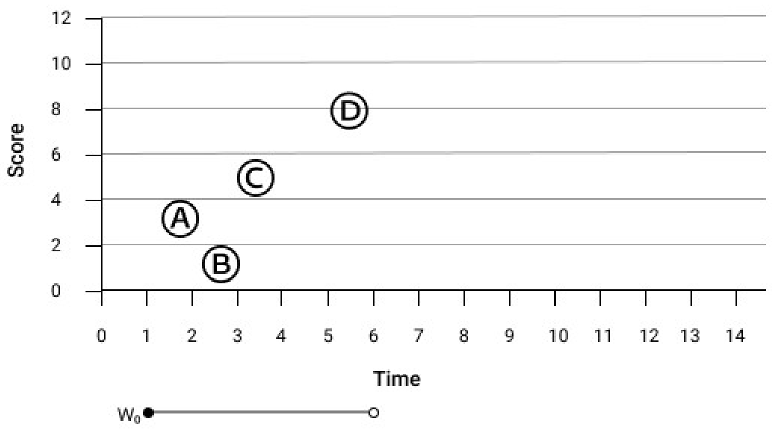
Going into the details of Figure 2.2a, it shows the content of window W_0, which starts at time 1 and ends at time 6 (excluded), items A, B, C, and D arrive during window W_0.

Figure 2.2b represents window W_1 that starts at time 4 and ends at time 9 (excluded). During this window items, E and F come, while items A, B, and C go out from the result when window W_0 expires.

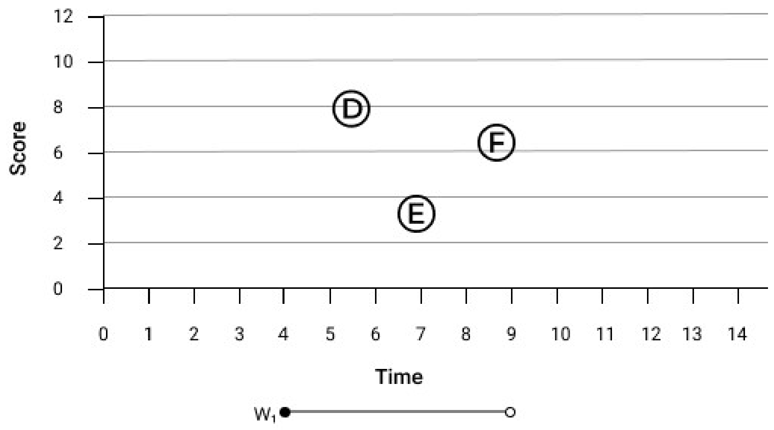
Figure 2.2c shows window W_2 content, it starts at time 7 and ends at time 12 (excluded), during window W_2 items G and H arrive while item D goes out from the result when window W_1 expires.

Let us assume that we want the top-3 item for each window; Figure 2.2d shows on the left the top-k result for window W_2 and the predicted top-k result for W_3, while on the right it shows the super-top-k list at the evaluation of window W_2.

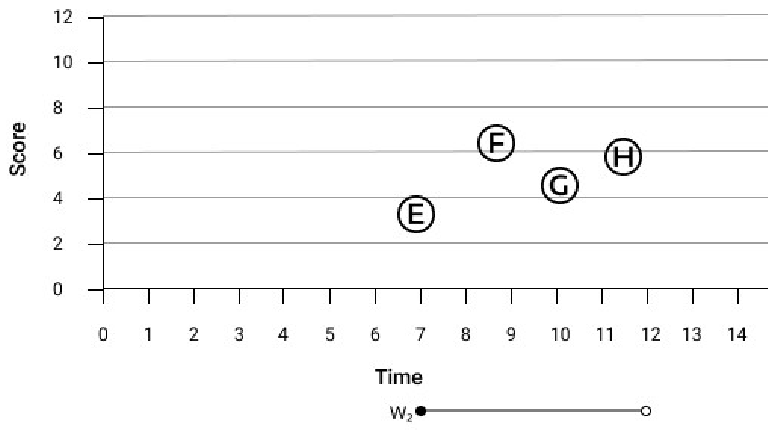
In the super-top-k list, items are sorted by their scores, W_s and W_e stands for starting and ending window respectively. Also, we can see the *lbp* of W_2, which indicates the item with the smallest score in the super-top-k list for W_2.



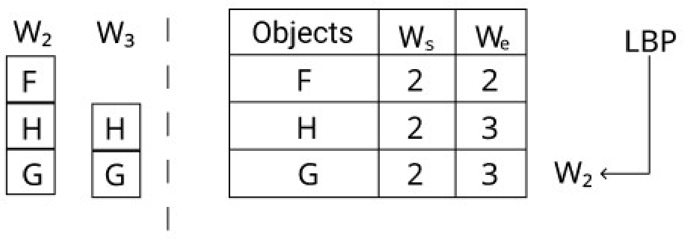
(a) Evaluation of window W_0



(b) Evaluation of window W_1



(c) Evaluation of window W_2



Objects	W_s	W_e
F	2	2
H	2	3
G	2	3

(d) Predicted top-k result & super-top-k

Figure 2.2: Example of MinTopK

Two maintenance steps compose the MinTopK algorithm:

> **Handling window expiration -** The expired window's top-k result must be removed from the super-top-k list. Since the first *k* items in the list are the top-k result of the expired window, it is sufficient to purge them. This process is implemented by increasing the *starting_window* mark of the first *k* objects in the list. If the *starting_window* becomes bigger than the *ending_window*, the item is removed from the super-top-k list, and the LBP set is updated if any *lbp* points to it.
>
> **Handling insertion of new arriving objects in the super-top-k list -** Firstly, it checks if the new object can enter the current or future top-k result. If all *predicted top-k result* lists have k elements or if the new object's score is smaller than any object in the super-top-k list, it discards the new object. Otherwise, if the new object's score is larger than any object in the super-top-k list or if some *predicted top-k result* lists are not full, the new object is inserted in the super-top-k list based on its score, and the *starting_window* and *ending_window* marks are computed. The next step removes the smallest score object in each window where it inserted the new item. The algorithm removes the item from *predicted top-k result* in the same way as when a window expires.

The MinTopK algorithm updates the *lbp* moving it one position up in the super-top-k list.

Yang et al. analyzed MinTopK complexity. The CPU complexity in the general case is $\mathcal{O}(N_{\text{new}} * (\log MTK.size))$, with N_{new} the number of object coming in that window slide. The memory complexity of MinTopK is $\mathcal{O}(MTK.size)$ since MinTopK requires a costant memory size to mantain each item in MTK set. Yang et al. showed that in the average case the size of MTK is *2k*, so the average-case CPU complexity becomes $\mathcal{O}(N_{\text{new}} * (\log k))$ while the average-case memory complexity becomes $\mathcal{O}(k)$.

Apache Kafka is a "distributed streaming platform that can be used to store and process data streams" [12]. A Kafka cluster is composed of Producer clients, Consumer clients, and Kafka servers called *brokers*; this whole architecture provides publish-subscribe service[9].

A **Kafka Producer** writes messages into Kafka, while a **Kafka Consumer** reads messages from it.

Kafka messages are stored into brokers and organized in *topics*, each message is a key-value pair, both key and value are variable-length byte arrays, optionally a message can have a time stamp.

Topics are divided into partitions. When a Producer writes a message into a topic, it must specify the partition; by default, if the message key is specified, the partition is computed hashing the key; thus, messages with the same key always go to the same partition. A partition is an immutable, ordered sequence of records that is continuously appended to a commit log. Each message is associated with an *offset*, which is the message position in the partition. Offsets are assigned implicitly by the order in which messages are appended to it. Kafka guarantees ordering within a single partition. Figure 2.3 shows the anatomy of a Kafka topic.

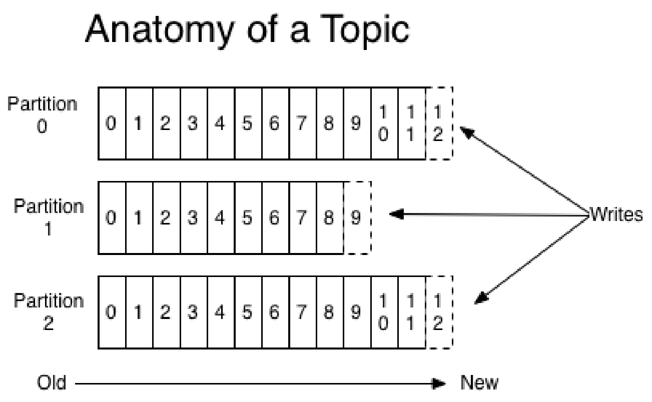


Figure 2.3: Anatomy of a Kafka topic [17].

Messages consumption in Kafka works in a pull flavor, meaning that the Kafka Consumer asks the broker a message. Also, the Consumer keeps track of the last record offset read; thus, the broker does not have to keep track of the offset of each Consumer, which can be a challenging job since a multitude of Consumers can read simultaneously from a broker.

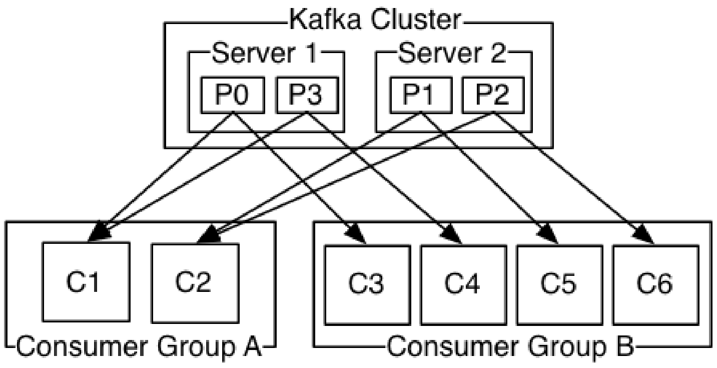


Figure 2.4: Two consumer groups reading from a topic with four partitions [17].

Kafka Consumer can be grouped in *Consumers groups*, in this case, a single partition is assigned to exactly one Consumer within a *Consumers group*, but a Consumer can read from more than one partition. In this way, it is possible to parallelize the reading from a specific topic. Figure 2.4 shows a Kafka Cluster with two brokers where two consumer groups consume messages from a topic with four partitions. Consumers that belong to different consumer groups or do not belong to any are independents, meaning that they can read messages from the same topic without interfering with each other, resulting in decoupling, a desirable property for a distributed system.

A **Kafka Broker** store messages on disk; differently from other publish-subscribe services, Kafka can be used for long term storage of messages since Kafka does not delete messages after delivery. A retention policy is applied to a topic, meaning that each message in a topic has a *retention time* that specifies how long a message should be stored. Kafka topics can also be configured for log compaction; we can use these topics to store updates. A compacted topic implies that a newer record replaces an older one with the same key; thus, a Consumer will always read the newer version of a message.

Kafka cluster scales linearly with the number of brokers. Since partitions are independent units within a topic, there is no need for broker synchronization even if partitions are stored in different brokers, so Kafka brokers can scale-out horizontally and balance load within the Kafka cluster.

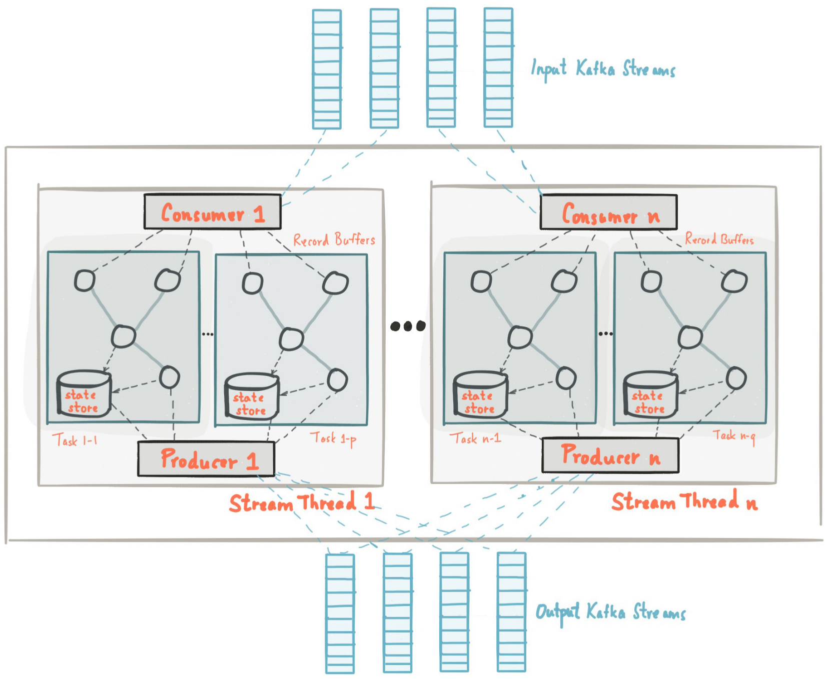


Figure 2.5: Kafka Streams application's architecture taken from Kafka Streams documentation[1]

2.3.1 *Kafka Streams*

Kafka Streams[2] is the Apache Kafka's stream processing library, built on top of Kafka's primitive for fault tolerance and scaling. Kafka Streams supports stateless and stateful stream processing operators such as windowing, joins, and aggregations. Also, it supports event-time semantics and exactly-once processing guarantees and can handle late-arriving and out-of-order data.

Given its properties, Kafka Streams can be used to build real-time stream processing applications, where data are read from a topic, processed using the operators provided by the Stream API, and written to another topic. Figure 2.5 shows the architecture of an application that uses the Kafka Streams library.

Applications that use Kafka Stream leverage on Kafka horizontal scaling to cope with huge data traffic. Each stream application forms a *Consumer group*, so topic partitions equally and transparently distribute across application instances. The programmer has not to cope with load balancing since Kafka cluster monitors all the instances' liveness, and if one or more

1 http://kafka.apache.org/11/documentation/streams/architecture
2 https://kafka.apache.org/10/documentation/streams/developer-guide/dsl-api.html

Operator	1st Input	2nd Input	Output
filter, mapValue	KStream		KStream
	KTable		KTable
map, flatMap	KStream		KStream
groupBy $\rightarrow$ agg	KStream		KTable
	KTable		KTable
groupBy + windowBy $\rightarrow$ agg	KStream		KTable
inner-/left-/outer-join	KStream	KStream	KStream
inner-/left-/outer-join	KTable	KTable	KTable
inner-/left-join	KStream	KTable	KStream

Table 2.1: Operators with their input and output types

instances go down, Kafka rebalances the load across the ones alive.

Table 2.1 taken from [13] shows the stream processing operators provided by the library with their input and output. These operators are quite exhaustive, but if some different operations need to be executed on the data stream, they should be implemented.

In the next subsection, we show Kafka Processor API, which we can use to develop user ad-hoc operators.

2.3.2 *Processor API*

Processor API[3] are suitable for operations that cannot be implemented with the Streams API.

The Processor API allows us to define custom stateless and stateful processors and interact with state stores easily. Developers can access and process every record of the data stream and connect to the state stores associated, defining ad-hoc processing topology. *State Stores* offer stream processing applications the possibility to store and query data; they can be either a **RocksDB** database, an in-memory hash-map, or other useful data structures. These stores are local to an instance, but it is possible to access remote state stores through *Interactive Queries*.

Another reason to use the Processor API is to access a record's metadata, such as its offset, topic, or partition. Also, it is possible to combine Streams and Processor API exploiting the ad-

3 https://kafka.apache.org/10/documentation/streams/developer-guide/processor-api.html

vantages of both libraries using the Streams API's methods KStream#process() and KStream#trasform()

2.3.3 *KSQL*

KSQL is a streaming SQL engine for Apache Kafka[7], where it is possible to write queries in a SQL-like syntax. KSQL is built on Kafka Streams and supports a wide range of stream processing operations, providing continuous queries over the data stream. A *schema* must be defined over a topic, and every record's value should conform to it. The schema defines a set of typed columns.

We can define both streams and tables over Kafka topics in KSQL, depending on how we interpreter the message inside a topic. When we define a stream over a Kafka topic, each record is independent; if we define a table, each record can either update a message with the same key or add a new message where there are no messages with the same key.

As we said, KSQL supports a wide range of stream processing operations; it is possible to perform aggregations, counts, joins, and windowing operations over a data stream. To stream top-k queries, the windowed and aggregated operators are the most interesting; windowing operators provide *windowed* stream processing, grouping records with the same key to perform stateful operations. KSQL supports three types of windows:

- **Tumbling window** which are time-based, fixed-sized, non-overlapping, and gap-less windows
- **Hopping window** which are time-based, fixed-sized, and overlapping windows
- **Session window** which are session-based, dynamically sized, non-overlapping, and data-driven windows

Aggregate operators always result in a table because KSQL computes the aggregate for each key and updates these results as it processes new input data for a key. Examples of aggregator functions are **COUNT**, **SUM**, **AVERAGE**, and **TOPK**; they all work on a single column, so we can not use them to perform aggregate operations on multiple columns such as sum the value of two columns or compute the top-k using a scoring function that takes in input multiple columns' values.

In Listing 2.1, we show an example of a KSQL query using windowed and aggregated operators. We create a new **TABLE** *sink* (Line 1), continuously monitoring the *metrics_stream* stream (Line 3). We use a hopping window of size 60 seconds that slides every 10 seconds (Line 4). Every time a window ends, the query returns the average of the *metric->value* using the **SUM** and **COUNT** operators (Line 2), grouping the results with the *instance* attribute (Line 5).

```
1 CREATE TABLE sink AS
2 SELECT instance, SUM(metric->value) / COUNT(*)
3 FROM metrics_stream
4 WINDOW HOPPING(SIZE 60 SECONDS, ADVANCE BY 10 SECONDS)
5 GROUP BY instance
```

Listing 2.1: Example of KSQL query using windowed and aggregated operators.

3 PROBLEM SETTING

This chapter presents the whole problem that we tackle with our work. In Section 3.1, we show the motivations that pushed us through this work. Section 3.2 describes the main problems behind the implementation of streaming top-k algorithms. Then, in Section 3.3, we state the research questions that we want to answer.

3.1 MOTIVATION

The problem of streaming top-k is integral to various sectors nowadays; examples of industries that use this kind of application include telecommunication, finance, smart energy, security, manufacturing, social media marketing, and many others. High throughput and low latency are desirable properties of real-time applications. Still, the data rates of online monitoring applications often exceed the data stream infrastructure's capacity in terms of processing resource, storage, and communication [5], leading to low throughput and high latency applications. That is why it is crucial to develop new algorithms that try to optimize the resources used. Another way to optimize online monitoring applications is parallelization and distribution. Moving the application from a centralized monolith to a parallel or distributed application allows splitting the workload.

As thoroughly explained in Section 2.2, a streaming top-k algorithm aims to evaluate, for each window over a data stream, the top-k objects based on a scoring function.

Consider the following example of the energy industry to understand the streaming top-k problem better. In the modern energy system, a countless number of IoT devices are connected to provide data in real-time to control the current load on the network or to monitoring the system to detect failures.

Assume that a water company has IoT sensors on the pipes running underground; these sensors can measure water pressure in the pipes and the humidity outside them. The water company knows that if the humidity outside the pipes is over 90% and the water pressure is under 3 bar, there may be a leak in the tube, so they have to send a team to check it. The sooner the possible leak is detected, the better it is for the company, so a reactive system is fundamental.

How does streaming top-k cope with this specific problem? The IoT sensors send the measured parameters to the water company's system, where a streaming application computes a score based on the humidity and the pressure and evaluates the top-k results, which correspond to the pipes where it is more probable a leak.

Another example of the usage of streaming top-k algorithms is Social Media marketing. Let us assume that a Social Media agency wants to find emerging influencers on Instagram to offer them a contract to advertise a new product. An Instagram user is an "emerging influencer" if it gets 1.000 new followers and 10.000 likes to its last photo in the previous hour. Hence, every hour the Social Media agency wants to know the five most "emerging influencers".

Assuming a data stream containing information for each Instagram user, such as the number of new followers and the number of likes to its photos, it would be great to use a query language such as KSQL to write a continuous top-k query to solve this problem. Unfortunately, KSQL does not provide any functionalities for streaming top-k answering since they have not implemented the **ORDER BY** clause yet, and their **TOPK** aggregate function only returns the top-k for a single column and window. The problem we are tackling is wider than returning the top-k for a single column and window; we need to return the top-k objects based on a scoring function that can work with more than one column. Listing 3.1 presents a possible workaround for answering the "emerging influencers" query we mentioned before. We need to write two different queries, the first computes the score, and the second computes the top-k result.

The first query takes *instagramUsersActivities* stream in input (Line 3), and filters the data using the **WHERE** clause to keep only those records that have more than 1000 *newFollowers* and more than 10000 *newLikes* (Line 4). Then, it outputs into the *tempScoring* stream (Line 1) a new record containing the *user* attribute and the score computed using the *Score* function (Line 2).

The second query reads each record from the *tempScoring* topic (Line 7) and outputs the result into the *top5InfluencerUsers* topic every hour using the **WINDOW** clause (Line 8). Using the **TOPK** operator in Line 6, we get the top-5 scoring records; then we group the result by the *user* with the **GROUP BY** clause.

```
1 CREATE STREAM tempScoring AS
2 SELECT user, Score(newFollowers, newLikes) AS score
3 FROM instagramUsersActivities
4 WHERE newFollowers >= 1000 AND newLikes >= 10000

5 CREATE TABLE top5InfluencerUsers AS
6 SELECT user, TOPK(score, 5)
7 FROM tempScoring
8 WINDOW TUMBLING (SIZE 1 HOUR)
9 GROUP BY user
```

Listing 3.1: Example of KSQL query to detect emerging influencer

Using two queries in sequence, we increase the number of reading and writing operations on topics, slowing down our application's performance. We want to write a single query in KSQL as the one shown in Listing 3.2 that solves the problem of finding "emerging influencers" in the Social Media marketing scenario we presented above.

```
1 CREATE TABLE top5InfluencerUsers AS
2 SELECT user, Score(newFollower, newLikes) AS score
3 FROM InstagramUsersActivities
4 WINDOW TUMBLING (SIZE 1 HOUR)
5 WHERE newFollower >= 1000 AND newLikes >= 10000
6 ORDER BY DESC (score)
7 LIMIT 5
```

Listing 3.2: Example of desirable KSQL query to detect emerging influencer

We define a new table called *top5InfluencerUsers* (Line 1), and we populate it with the query's results every hour using the **WINDOW** clause (Line 4). Every time the query evaluates, the **WHERE** clause (Line 5) is matched against the data in the window open on the data stream *InstagramUsersActivities*, the data is filtered, keeping only users with 1.000 or more new followers and with 10.000 or more new likes. Then, the results are ordered by *score* (Line 6), which is the result of the user-specified **Score** function that computes the score by the normalized sum of *newFollowers* and *newLikes*. Finally, only the firsts five results return using the **LIMIT** clause (Line 7).

As we already said, this query is not feasible in KSQL since the **ORDER BY** operator has not been implemented yet.

3.2 PROBLEMS

Researches in the streaming top-k field lead to different algorithms that differ from how they try to tackle the problem. However, they have something in common, all algorithms are the result of academic research, so usually, they are available as experimental proof of concept, and only in some rare cases as technology validated in lab. Both [4], [10] and [16] proposed technologies validated in lab, while [17] has lightly shown a technology validated in relevant environment.

We aim to bring these algorithms into Kafka to build a baseline for the **ORDER BY** clause we cited above. Precisely we focus on the MinTop-K [16] implementation. We opt to use Kafka Streams and Processor API because every KSQL clause and operator is built on top of their implementation on Kafka Streams.

Implementing an algorithm designed as a support for an experimental proof of concept into a mainstream technology as Kafka generates some problems:

1. How can we exploit Kafka's functionalities, libraries, and APIs to implement the algorithm?
2. Since Kafka is a distributed streaming platform, the next question that comes to mind is if it worth parallelizing the algorithm to achieve better performance and how Kafka can help us.
3. We have different parameters that can influence algorithms' performance for a given problem.

Examples of these parameters are the number of parallel instances to use, the window *size* and *hopping size*, or the top-k value, *k*. So we have to decide their values to optimize our algorithms, and it can be trivial. In our work, we realize experiments to understand how the top-k value, *k*, influences the algorithm's performance. Also, in Section 5.2, we make reasoning on algorithms' cost to understand how the values of the window size and the number of parallel instances influence the overall performance.

3.3 RESEARCH QUESTION

Given the considerations on continuous top-k query monitoring that we made before and the problems correlated to implementing a streaming top-k algorithm inside a mainstream distributed platform as Kafka, we decided to investigate the following **research questions**:

> **Q1 -** *Is it possible to leverage Kafka capabilities to implement streaming top-k algorithms?*
>
> **Q2 -** *Is it worth parallelizing streaming top-k algorithms to achieve better performance?*
>
> **Q3 -** *How does the top-k parameter,* k*, influence the algorithms' performance?*

To answer these research questions, we design ad-hoc experiments that Section 5.2 explains in detail.

4 PARALLEL STREAMING TOP-K IN KAFKA

In this chapter, we show the entire process behind the streaming top-k algorithms' implementation. In Section 4.1, we present the algorithms' design, explaining their functioning. Section 4.2 contains the infrastructure behind the applications, describing its part in detail. In Section 4.3, we describe the algorithms' implementation, showing the implementation choices and their rationale that lead us to the final applications.

4.1 ALGORITHM DESIGN

In this section, we explain in detail how the algorithms work. We describe the algorithms with the help of pseudo-codes.

4.1.1 *Baseline Algorithm*

As a baseline algorithm for our experiments, we use the most straightforward algorithm for streaming top-k; it goes under the family of "Materialize Score & Sort" algorithms. Algorithm 1 shows in more detail the pseudo-code for the parallel instance,

Symbol	Description
S	Data stream
k	Number of the outputs for each window
size	Global window size
hoppingSize	Global window hopping size
numInstances	Number of parallel instances
localSize	Local window size used by the instance
localHoppingSize	Local hopping size used by the instance
recordCount	Total number of records processed
O_i	An arriving object
windowsList	A list containing all the active window
windowToForward	Window to compute top-k results
windowToForward.objects	Records of the window to compute top-k
topkResult	Top-k results of a given window
currentWindowID	ID of the currently processed window

Table 4.1: List of symbols used in Algorithm 1

which sorts the record for each window and computes the top-k results. Table 4.1 contains the description of symbols used.

The algorithm gets in input the data stream, *S*, the number of outputs, *k*, the global window size, *size*, the global window hopping size, *hoppingSize*, and the number of instances, *numInstances*. *localSize* and *localHoppingSize* are computed using their respective global values and *numInstances*(Lines 1-2). *recordCount* is inizialized at 0 (Line 3) before starting ingesting the data stream. For every new arrival object O_i, the *currentWindowID* is computed (Line 5) and if it is not already in the *windowsList* it is added to it (Lines 6-7). Then, for every window W_i in the *windowsList* we add the O_i object to its list of objects W_i*.objects* (Lines 9-10). We check if the current window is ended, in this case, its objects are sorted by their score and we output the top-k results (Lines 12-15). Finally, we remove the ended window from *windowsList* (Line 17) and we increment the *recordCount* by one (Line 18).

4.1.2 *Parallel MinTop-K*

Algorithm 2 shows the pseudo-code for the parallel instance implementing a slightly modified version of Yang et al.'s MinTop-K[16]. Their version uses logical windows, while ours uses

Algorithm 1: Parallel Baseline Algorithm

```
input  : S, k, size, hoppingSize, numInstances
output: topkResult
1  localSize = size/numInstances;
2  localHoppingSize = hoppingSize/numInstances;
3  recordCount ← 0;
4  foreach new object $O_i$ in the stream S do
5      currentWindowID ← computeCurrentWindowID();
6      if currentWindowID NOT in windowsList then
7          add currentWindowID to windowsList;
8      end
9      foreach window $W_i$ in windowsList do
10         add $O_i$ to $W_i$.objects;
11     end
12     if recordCount ⩽ localSize OR recordCount % localHoppingSize =1 then
13         windowToForward ← windowsList.first;
14         topkResult ← SortAndTopK(windowToForward.objects, k);
15         return topkResult;
16     end
17     RemoveEndedWindow(windowsList);
18     recordCount ++;
19 end
```

Symbol	Description
S	Data stream
k	Number of the outputs for each window
size	Global window size
hoppingSize	Global window hopping size
numInstances	Number of parallel instances
localSize	Local window size used by the instance
localHoppingSize	Local hopping size used by the instance
windowsList	A list containing all the active window
currentWindow	The currently processed window
W_i	A window inside the *windowsList*
W_i.actualRecord	The number of records arrived during W_i
W_i.tkc	The size of the W_i's predicted top-k result set
W_{exp}	The expired window
O_i	An arriving object
O_i.score	The score of the new arriving object
O_{exp}	One of the first k object in the super-top-k list
O_{exp}.start_W	The starting window mark of O_{exp}
O_{exp}.end_W	The ending window mark of O_{exp}
$O_{min\_supertopk}$	The last object inside the super-top-k list
$O_{W_i.lbp}$	The object pointed by the W_i's *lbp*

Table 4.2: List of symbols used in Algorithms 2 and 3

physical ones. Table 4.2 contains the description of symbols used.

The algorithm gets in input the data stream, *S*, the number of outputs, *k*, the global window size, *size*, the global window hopping size, *hoppingSize*, and the number of instances, *numInstances*. *localSize* and *localHoppingSize* are computed using their respective global values and *numInstances*(Lines 1-2). The *MinTopK* function is the core of the algorithm; it includes the *MTK* maintenance, the window expiration handling, and the top-k result computation. Firstly, we check if the *WindowList* is empty; if so, we create a new window and add it to *windowList* (Lines 4-6). For every new arrival object O_i and every window W_i in *windowList*, we increase the window W_i's actual record counter (Line 10). Then, if the window W_i's actual record counter is equal to the local window size *localSize* + 1, it means that the window W_i has expired; in this case, we call *OutputTopKResults* function to output the top-k results for this window, we call *PurgeExpiredWindow* function passing W_i to handle W_i's expiration, and we update *currentWindow* (Lines 12-14). Then we check if it is time to create a new window; if so, we create and add it to *windowList* (Lines 16-18). Finally, we update the super-top-k list invoking *UpdateSuperTopK* function with O_i as a parameter.

Algorithm 3 shows in detail the functions used in Algorithm 2 and it uses the same symbols presented in Table 4.2:

OutputTopKResults - Which outputs the first k objects in the *super-top-k list*.

PurgeExpiredWindow - Which is the function that takes care of handling the input window's expiration. For the first *k* objects O_{exp} in the *super-top-k list*, we increase the starting window mark of O_{exp} (Line 5); then, if its starting window mark $O_{exp}.start\_W$ is larger than its ending window mark $O_{exp}.end\_W$, we remove O_{exp} from the *super-top-k list* (Lines 6-7). Finally, we remove the expired window W_{exp} from the windows list (Line 10).

UpdateSuperTopK - Which gets in input the new object O_i. Firstly, if its score is smaller than the last element of the *super-top-k list* and all windows have k elements associated within the *super-top-k list*, we discard the new object O_i (Lines 12-13). Otherwise, for every window W_i where its lower bound pointer's score ($O_{W_i.lbp}.score$) is less than the new object's score ($O_i.score$), we do the following:

if W_i's *top-k-counter* is less than k, we increase it by one (Lines 17-18); otherwise, we increase by one the starting window mark of the window's *lbp* ($O_{W_i.lbp}.start\_W$) (Lines 20-21). Then we check if the starting window mark of the window's *lbp* becomes greater than its ending window mark; in this case, we remove the window's *lbp* ($O_{W_i.lbp}$) from the *super-top-k list*, and we update the window's *lbp* by moving it one position up in the *super-top-k list* (Lines 23-25).

Algorithm 2: Parallel MinTop-K

```
input: S, k, size, hoppingSize, numInstances
1  localSize = size/numInstances;
2  localHoppingSize = hoppingSize/numInstances;
3  MinTopK()
4      if windowsList is empty then
5          currentWindow ← CreateNewWindow();
6          add currentWindow to windowsList;
7      end
8      foreach new object $O_i$ in the stream S do
9          foreach window $W_i$ in windowsList do
10             $W_i$.IncreaseActualRecord();
11             if $W_i.actualRecord$ = localSize + 1 then
12                 OutputTopKResults();
13                 PurgeExpiredWindow($W_i$);
14                 currentWindow ← windowsList.first;
15             end
16             if $W_i$ = currentWindow AND $W_i.actualRecord \neq 1$
                AND $W_i.actualRecord$ % localHoppingSize = 1
                then
17                 newWindow ← CreateNewWindow();
18                 add newWindow to windowsList;
19             end
20             UpdateSuperTopK($O_i$);
21         end
22     end
```

4.1.3 *Centralized Aggregator & Top-K*

Algorithm 4 shows the pseudo-code of the centralized aggregator used by both the Parallel baseline algorithm and the Parallel

Algorithm 3: Parallel MinTop-K auxiliary functions

```
OutputTopKResults()
    output first k objects on super-top-k list;
PurgeExpiredWindow ($W_{exp}$)
    for first k object $O_{exp}$ in super-top-k list do
        $O_{exp}$.start_W ++;
        if $O_{exp}$.start_W > $O_{exp}$.end_W then
            remove $O_{exp}$ from super-top-k list;
        end
    end
    remove $W_{exp}$ from windowsList;
UpdateSuperTopK ($O_i$)
    if $O_i$.score < $O_{min\_supertopk}$ AND All $W_i$.tkc = k then
        discard $O_i$;
    end
    else
        foreach $W_i$ that $O_{W_i.lbp}$.score < $O_i$.score do
            if $W_i$.tkc < k then
                $W_i$.tkc ++;
            end
            else
                $O_{W_i.lbp}$.start_W ++;
            end
            if $O_{W_i.lbp}$.start_W > $O_{W_i.lbp}$.end_W then
                remove $O_{W_i.lbp}$ from super-top-k list;
                move $W_i$.lbp by one position up in
                  super-top-k list;
            end
        end
    end
```

Symbol	Description
S	Data stream
k	Number of the outputs for each window
numInstances	Number of parallel instances
R_i	Record representing local top-k result
R_i.key	Window associated to the top-k result
R_i.value	Movie record
windowObjectMap	Hash map indexed by windowID with the list of partial top-k for that window
recordList	list of partial top-k
topkResult	Top-k results of a given window

Table 4.3: List of symbols used in Algorithm 4

MinTop-K algorithm. Table 4.3 contains the description of symbols used.

The algorithm gets in input the data stream, *S*, the number of outputs, *k*, and the number of instances, *numInstances*. For every new arrival record R_i in the data stream S, R_i*.value* is added to the list of top-k results associated to the window R_i*.key* (Line 2). Once we receive every partial top-k results from the parallel instances, we sort the *recordList* and compute the final top-k results (Lines 4-7), which return as the output of the algorithm.

```
Algorithm 4: Centralized Aggregator & Top-K
  input  : S, k, numInstances
  output : topkResult
1 foreach new record $R_i$ in the stream S do
2     windowObjectMap [$R_i$.key].add($R_i$.value);
3     recordList ← windowObjectMap [$R_i$.key];
4     if recordList.size = $k * numInstances$ then
5         topkResult ← SortAndTopK(recordList, k);
6         remove windowObjectMap [$R_i$.key];
7         return topkResult;
8     end
9 end
```

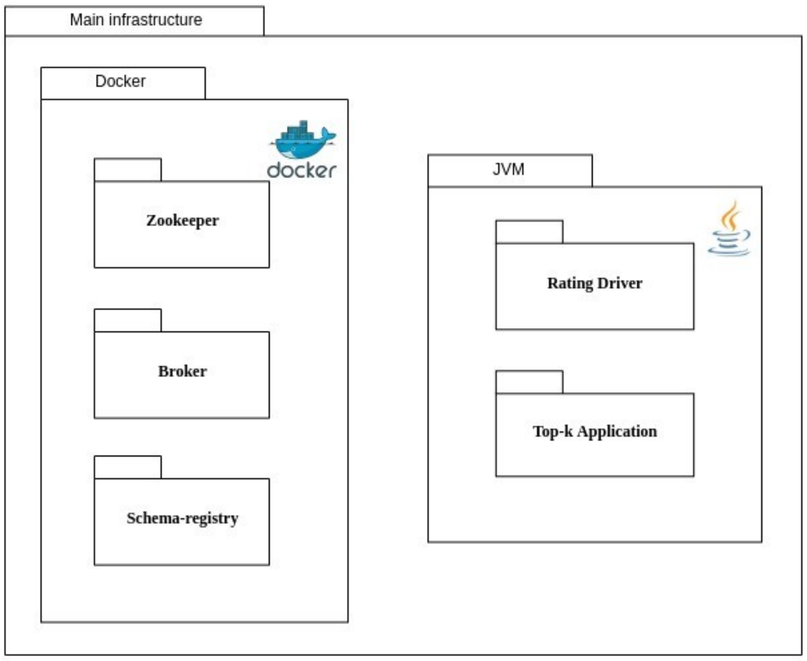


Figure 4.1: Main Infrastructure

4.2 INFRASTRUCTURE

The overall infrastructure is mainly based on Docker containers; except for the Java applications that run on the JVM, everything else runs inside containers. Figure 4.1 shows an overview of the whole infrastructure. It represents the two main parts, the one based on Docker containers on the left, while the JVM part is on the right.

In Subsection 4.2.1, we explain the Docker part more in-depth, while Subsection 4.2.2 describes the Java applications' architecture, which slightly differs from one algorithm to another.

4.2.1 *Docker infrastructure*

Three Docker containers compose the Docker infrastructure:

- Zookeeper built using Docker image *confluentinc/cp-zookeeper:5.4.0*
- Broker built using Docker image *confluentinc/cp-enterprise-kafka:5.4.0*
- Schema-Registry built using Docker image *confluentinc/cp-schema-registry:5.4.0*

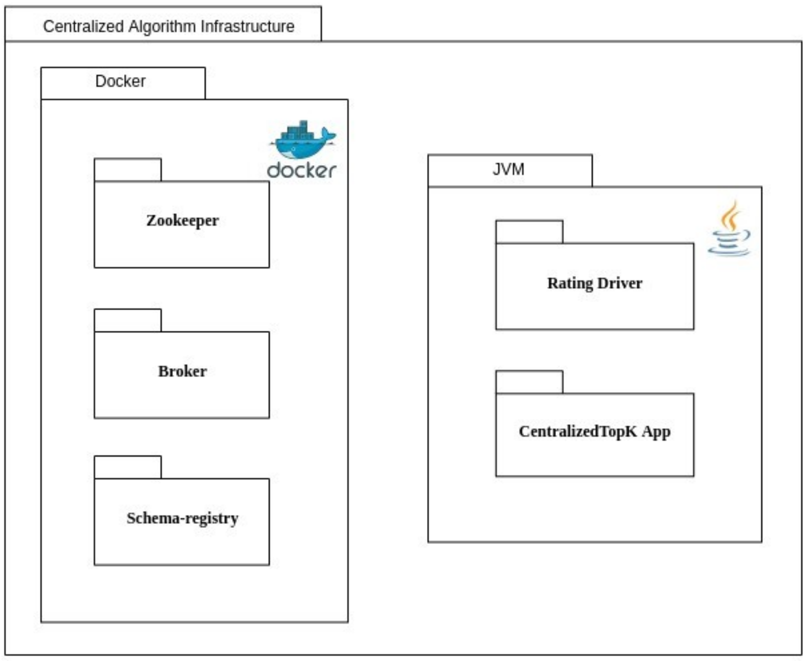


Figure 4.2: Centralized Algorithm Infrastructure

The fundamentals of the Kafka infrastructure run inside the containers. Zookeeper, Schema-Registry, and a Kafka broker are the sufficient parts of the Kafka infrastructure.

We use a *docker-compose.yml* file to build the containers all together from the images[1].

4.2.2 *JVM infrastructure*

This subsection shows the experiment infrastructure, focusing on the part based on the Java applications that run inside the JVM.

Figure 4.2 shows the infrastructure in the case of the centralized algorithms' version; it is the simpler one since it is composed of only two components: *Rating Driver* and *Centralized Top-K App*. The former is the application that sends the input data to a specific topic into the Kafka Broker; the latter computes the top-k results; hence, it is the most crucial infrastructure component.

In Figure 4.3, we find a slightly more complex infrastructure; it represents the infrastructure of the parallel algorithms. N parallel instances compute their local top-k result and send it to

1 https://github.com/Luca-Ferrera/kstream/blob/master/docker-compose.yml

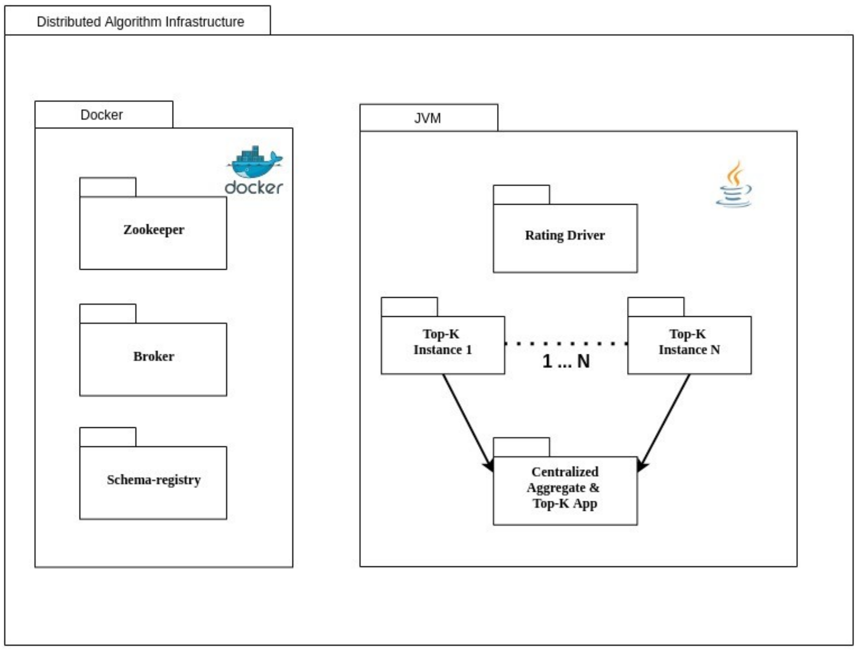


Figure 4.3: Parallel Algorithm Infrastructure

a single topic with one partition. Then we have the *Centralized Aggregate & TopK* application that aggregates all the local top-k results from the N parallel instances, sorts them, and computes the final top-k result.

4.3 IMPLEMENTATION EXPERIENCE

The process behind algorithms' implementation is simple. Firstly we implement the centralized version, we test its correctness, and then we run the experiment. Secondly, we use this centralized version as one element of our parallel application; each parallel instance runs the streaming top-k algorithm as it is independent of the other instances, then it sends the top-k results to a topic with one partition. A centralized application aggregates the results by consuming the parallel part's output topic, sorts them by their score, and outputs the final top-k results.

We decide to base the algorithm's implementation on Kafka because we want to experiment inside a real word environment. Thanks to its Streams and Processor API, Kafka provides excellent tools for data streaming computation and analysis.

Firstly, we reasoned on the windowing operation, meaning dividing the data stream into portions to compute top-k results. Kafka Streams provides windowing operators such as *groupBy* or *Count*, but none of these operators offer functionalities useful to solve our problem. To implement our algorithms, we needed a level of freedom that only the Processor API provides, but Kafka Processor API does not give windowing operations. Since we could not rely on Kafka Streams windowing operator, we had to implement it by ourselves.

Implementing the centralized baseline algorithm, we came out with a solution that implies the use of *ProcessorContext#schedule()*[2]. This method is used to schedule a punctuation function, which is a function that periodically triggers the punctuate method of the *Punctuator* interface.

We used *ProcessorContext#schedule()* to trigger the sorting of all the elements that arrived inside a window. These elements are saved in the processor's *State Store*; every store record is a key-value pair, with the *windowID* as key and an *ArrayList* of records as value.

Unfortunately, when it was time to implement the parallel algorithm's version, we realized that this solution was not feasible because, in a distributed environment, we have a state store associated with each instance that only keeps track of the elements that arrived there. Kafka Streams provides *Interactive Queries*, a way to access state stores of the parallel instances. We needed to access the state stores in write mode to maintain a distributed data structure updated to keep track of all the records arrived at every instance. Unfortunately, *Interactive Queries* provides only read access to remote *State Stores*, so we could not exploit them for our purpose.

We started the implementation from scratch, basing it on *State Stores*. We decided to use physical windows instead of logical ones since we found it easier to keep track of them. For each instance, our idea was to use a local state store to count the number of elements that arrived inside a window.

Each parallel instance works on a subset of the window of size $window\_size/num\_instances$. In this way, we know that when $window\_size/num\_instances$ records arrive at an instance, the current global window has expired, meaning that the instance will not receive another element for that window, and it can start processing the new one as soon as a new record arrives.

To make this work, we made two assumptions: *window_size* and *hopping_size* should be multiple of *num_instances*, and input data should be sent to the input topic in a Round-Robin fashion.

2 https://kafka.apache.org/10/documentation/streams/developer-guide/processor-api.html#defining-a-stream-processor

To meet the assumption of sending input data in a Round-Robin fashion, we came across a problem using the Kafka *DefaultPartioner*, since it does not send data strictly in Round-Robin when the partitioning key is null, to improve performance[3]. So we implemented our custom *RoundRobinPartitioner* class.

4.3.1 *Parallel MinTop-K Implementation*

Let us discuss the most interesting points of our MinTop-K implementation.

We use *State Store* to save the *super-top-k list* and keep track of the active windows. Listing 4.1 shows how we create, register, and pass the stores to the *Transformer*.

```
StoreBuilder superTopkStoreBuilder = Stores.
    keyValueStoreBuilder(
  Stores.persistentKeyValueStore("super-topk-list-store"),
  Serdes.Integer(),
  minTopKEntryAvroSerde(envProps));
builder.addStateStore(superTopkStoreBuilder);

StoreBuilder windowsStoreBuilder = Stores.
    keyValueStoreBuilder(
  Stores.persistentKeyValueStore("windows-store"),
  Serdes.Long(),
  physicalWindowAvroSerde(envProps));
builder.addStateStore(windowsStoreBuilder);

builder.<~>stream(scoredMovieTopic).transform(
  new TransformerSupplier<String,ScoredMovie,KeyValue<~>>(){
    public Transformer get(){
      return new DistributedMinTopKTransformer(k,
          cleanDataStructure);
    }
  },
  "windows-store",
  "super-topk-list-store");
```

Listing 4.1: Parallel MinTop-K: how to create, register and pass *State Stores* to a Kafka *Transformer*.

3 https://cwiki.apache.org/confluence/display/KAFKA/FAQ#FAQ-Whyisdatanotevenlydistributedamongpartitionswhenapartitioningkeyisnotspecified?

```java
for(PhysicalWindow window: lowerBoundPointer){
  MinTopKEntry lowerBoundPointed = window.
      getLowerBoundPointer();
  if(movie.getScore() < lowerBoundPointed.getScore() &&
      window.getTopKCounter() < k){
    window.setLowerBoundPointer(newEntry);
    window.increaseTopKCounter(1);
    physicalWindowsStore.put(window.getId(), window);
  }
  int index = superTopKList.indexOf(lowerBoundPointed);
  if(index == -1){
    MinTopKEntry newLowerBound = superTopKList
      .get(superTopKList.size() - 1);
    window.setLowerBoundPointer(newLowerBound);
    physicalWindowsStore.put(window.getId(), window);
  } else if(lowerBoundPointed.getScore() < movie.getScore())
      {
      if(window.getTopKCounter() < k){
        window.increaseTopKCounter(1);
        physicalWindowsStore.put(window.getId(),window);
      } else {
          //increase starting window
          lowerBoundPointed.increaseStartingWindow(1L);
          window.setLowerBoundPointer(lowerBoundPointed);
          if(lowerBoundPointed.getStartingWindow() >
              lowerBoundPointed.getEndingWindow()) {
            //remove from superTopKList
            superTopKList.remove(index);
            superTopKListStore.delete(index);
          } else{
            //update lowerBoundPointed into superTopKList
            superTopKList.set(index,lowerBoundPointed);
            superTopKListStore.put(index,lowerBoundPointed);
          }
          //move lbp one position up in the superTopKList
          MinTopKEntry newLowerBound = superTopKList.get(
              index - 1);
          window.setLowerBoundPointer(newLowerBound);
          physicalWindowsStore.put(window.getId(), window);
        }
    }
}
```

Listing 4.2: Parallel MinTop-K: LBP update after new object insertion into *super-top-k list*.

To ease our process, we use auxiliary data structures to save *State Stores*'s content at the beginning of each record processing. When iterating on *KeyValueStore*, no ordering guarantees are provided[4], but we need ordering guarantees for the *super-top-k list*, so we save the *"super-topk-list-store"* content into an *ArrayList*. Also, we store the *"windows-store"* content into a *LinkedList*, which we use as the *Lower Bound Pointer list (LBP)*. Then, at the end of each record's processing, we save them back to the *State Stores*. Also, we store the changes into *State Stores* as soon as we modify the data structures not to lose any update.

Listing 4.2 shows the code for updating the *LBP*; it is a portion of the *updateSuperTopK* function, which is the most critical part of the algorithm. We are right after the new item insertion in the *super-top-k list*.

For every active window, we get the object pointed by its *lbp* (Lines 1-2), then if the window has not *k* items yet, and the new entry's score is less than the window *lbp*'s score, we increase the TopKCounter and set the new entry as window's *lbp* (Lines 3-6). Now, we check if the element pointed by the *lbp* has already been removed from the *super-top-k list* (because it was the *lbp* of another window already processed); in this case, we set as new *lbp* the last *super-top-k list's* object (Lines 8-13). Otherwise, we update the *super-top-k list* according to the algorithm (Lines 14-34), we leave the code explanation to the reader.

4 https://kafka.apache.org/21/javadoc/org/apache/kafka/streams/state/ReadOnlyKeyValueStore.html#all--

5 EXPERIMENTS & RESULTS

In this section, we show the experimental setup for comparative research in 5.1, the design of experiments in 5.2, the final results in 5.3, and the findings in 5.4.

As an experimental environment, we use an m5d.4xlarge instance on AWS.[1] The operating system is Ubuntu 16.04; OpenJDK 11.0.8, Docker 19.03.13 and docker-compose 1.27.4 are installed on the machine.

5.1 EXPERIMENTAL SETUP FOR COMPARATIVE RESEARCH

In order to perform comparative research, we used the following experimental setups:

- Centralized Baseline Algorithm
- Parallel Baseline Algorithm
- Centralized MinTop-K
- Parallel MinTop-K

1 https://aws.amazon.com/it/ec2/instance-types/m5/

Running experiments with both centralized and parallel versions of the two algorithms make it possible to compare them and tell if it is worth parallelizing streaming top-k algorithms.

Setting up the experiments, we noticed that some parameters influence the results more than others. More specifically, we found that it is fundamental to choose the *window_size* and the *num_instances* in the right way.

Below, we show the reasoning that leads us to the parameters' values for the experiments. The following are the cost functions for both the centralized and the parallel versions of the Baseline algorithm. In both functions, there are three constant values:

- C_m: materialize cost
- C_s: sorting cost
- C_k: top-k cost

Equation 5.1 shows the centralized algorithm's cost, while Equation 5.2 contains the parallel one. More in detail lines 1,2,3 of 5.2 represent materializing cost, sorting cost and top-k cost respectively for the parallel part, while lines 4 and 5 represent sorting cost and top-k cost respectively for the centralized aggregation part.

$$\begin{aligned} &C_m * window\_size + \\ &C_s * window\_size * \log_2(window\_size) + C_k * k \end{aligned} \tag{5.1}$$

$$\begin{aligned} &C_m * window\_size + \\ &C_s * window\_size * \log_2(\frac{window\_size}{num\_instances}) + \\ &C_k * k * num\_instances + \\ &C_s * k * num\_instances * \log_2(k * num\_instances) + \\ &C_k * k \end{aligned} \tag{5.2}$$

Looking closer to the above cost functions, we can see that the first and last terms are the same so that they can be simplified.

$$\begin{aligned} &C_s * window\_size * \log_2(window\_size) = \\ &C_s * window\_size * \log_2(\frac{window\_size}{num\_instances}) + \\ &C_k * k * num\_instances + \\ &C_s * k * num\_instances * \log_2(k * num\_instances) \end{aligned} \tag{5.3}$$

Here, we can see that by choosing the *window_size* and the *num_instances* in the right way, we can have the parallel cost lower than the centralized one.

Setting up the experiments with different values of *window_size* and *num_instances*, we concluded that we could not use small values of *window_size*. The centralized algorithm's workload will be too low, and the gain of distributing this load between the instances will be almost null or even negative due to the cost of coordination between the parallel instances, the cost of the centralized aggregation of partial top-k results, and the final top-k computation.

Also, the *num_instances* should not be too small to gain in the parallel version, so we decided to use 6 instances.

To sum up, we have seen that a *window_size* of 1200 brings no gain in using the parallel version, while a *window_size* of 3600 brings gain. Said so, 3600 is the value used for *window_size* in the experiments.

Moreover, the design we have chosen for the parallel applications leads to the following constrain that we have to meet to gain from the parallelization:

$$k * num\_instance < window\_size$$

In the Equation 5.4, we place $k * num\_instance$ equal to $window\_size$ to see how the cost function become if we do not meet the constrain; we can see that the second term is equal to the first one plus two positive addends, so the parallel cost is greater than the centralized one. Given this constrain, the number of instances, and the value of *window_size* we choose for the experiments, we know upfront the experiments that for $k \geqslant 600$ we will have a worse performance of the parallel algorithm over the centralized one.

$$\begin{aligned} & C_s * window\_size * \log_2(window\_size) = \\ & C_s * window\_size * \log_2(k) + \\ & C_k * window\_size + \\ & C_s * window\_size * \log_2(window\_size) \end{aligned} \tag{5.4}$$

5.2 DESIGN OF EXPERIMENTS

We run each experiment five times using five different datasets as input data. In this way, we have more statistically consistent results.

Each dataset is randomly generated using a python script and contains 100.000 records in Avro format. The idea is to simulate a movie-rating platform where users can rate movies; each record in the dataset represents a movie with its unique ID, title, year of release, user rating, and score (Listing 5.1 shows an example). The score is computed as a weighted sum of the year of release and user rating (5.5).

$$score = \frac{rating}{10} * 0.8 + \frac{release\_year}{2020} * 0.2 \tag{5.5}$$

```
{"id": 140, "title": "La Grande Bellezza", "release_year":
    2013, "rating": 8.77084821316125, "score":
    0.9009747877459693}
```

Listing 5.1: Record example

Before each experiment, a Kafka Producer sends the input data to a different topic for each experiment. We decide to ingest the data before starting the measurements because the ingestion time is irrelevant compared to the experiment's total duration.

We measure the total time that passes between the first record consumed and the last top-k result of each experiment's last window.

This is the list of the parameters used and their explanation:

- *k*: number of the outputs for each window
- *size* or *window_size*: by how many records a window is made
- *hopping_size*: by how many records each window moves forward relative to the previous one
- *#dataset*: a number indicating which dataset is used as input
- *num_instances*: how many application instances run in parallel

We divide the experiment into four parts corresponding to each algorithm.

5.2.1 *Centralized Baseline Algorithm*

This subsection shows the different settings we used for testing the Centralized Baseline Algorithm.

For each dataset, we run the experiments using different values of the *k* parameter, without varying the others.

We have three different configurations for our experiment:

1. $k = 5$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 1$
2. $k = 10$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 1$
3. $k = 50$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 1$

5.2.2 *Centralized MinTop-K*

In this subsection, we present the settings we used for testing the Centralized MinTop-K algorithm.

For each dataset and experiments, we use different values of the *k* parameter, while keeping constant the others.

These are the three different configurations for our experiment:

1. $k = 5$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 1$
2. $k = 10$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 1$
3. $k = 50$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 1$

5.2.3 *Parallel Baseline Algorithm*

This subsection shows the settings we used for testing the Parallel Baseline Algorithm.

For the parallel version, we run two kinds of experiments. In the first one, we run the experiments using different values of the *k* parameter, without varying the others. We want to see any correlation between the value of the top-k and the experiment's total time.

We have seven configurations for this experiment:

1. $k = 5$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 6$
2. $k = 10$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 6$
3. $k = 50$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 6$
4. $k = 100$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 6$
5. $k = 150$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 6$
6. $k = 200$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 6$

7. $k = 300$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 6$

In the second experiment, we change the number of parallel instances *num_instances*, while keeping the other parameters constant. We expect that by increasing the number of parallel instances, the experiment's total time will decrease.

We have three configurations:

1. $num\_instances = 3$, $k = 10$, $size = 3600$, $hopping\_size = 300$
2. $num\_instances = 6$, $k = 10$, $size = 3600$, $hopping\_size = 300$
3. $num\_instances = 10$, $k = 10$, $size = 3600$, $hopping\_size = 300$

5.2.4 *Parallel MinTop-K*

In this subsection, we present the different settings we used for testing the Parallel MinTop-K algorithm.

For the parallel version, we run two kinds of experiments. In the first one, we run the experiments using different values of the k parameter, without varying the others. We want to see any correlation between the value of the top-k and the experiment's total time.

These are the seven configurations we use:

1. $k = 5$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 6$
2. $k = 10$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 6$
3. $k = 50$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 6$
4. $k = 100$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 6$
5. $k = 150$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 6$
6. $k = 200$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 6$
7. $k = 300$, $size = 3600$, $hopping\_size = 300$, $num\_instances = 6$

In the second experiment, we change the number of parallel instances *num_instances* while keeping the other parameters constant. We expect that by increasing the number of parallel instances, the experiment's total time will decrease.

We have three different configurations:

1. $num\_instances = 3$, $k = 10$, $size = 3600$, $hopping\_size = 300$
2. $num\_instances = 6$, $k = 10$, $size = 3600$, $hopping\_size = 300$
3. $num\_instances = 10$, $k = 10$, $size = 3600$, $hopping\_size = 300$

5.3 RESULTS

In this section, we expose the experiments' results. In detail, in Subsection 5.3.1, we show the comparative research results that compare the centralized and the parallel versions of the two implemented algorithms (baseline and MinTop-K) by varying the number of parallel instances.

Subsection 5.3.2 describes the comparative research results based on the top-k parameter; the experiment aims to define if there is a correlation between the value of the top-k parameter, *k*, and the experiment's total time. We measure the total time that passes between the first record consumed and the last top-k result of each experiment's last window.

As already described in Section 5.2, we run each experiment using five different datasets to have more statistically consistent results.

5.3.1 *Centralized and Parallel comparison*

This experiment aims to prove that by parallelizing the streaming top-k algorithm, we can achieve better performance in terms of total time to elaborate a given amount of streaming data. We run the experiments using a fixed value of the top-k parameter equal to 10, varying the number of instances between values 1, 3, 6, and 10.

5.3.1.1 *Baseline Algorithm*

Figure 5.1 shows the box-plots of the *total_time* using one instance for the centralized version, and six instances for the parallel one. We can see that the parallel version's *total_time* is significantly less than the centralized's one; as we expected, parallelizing the algorithm improves its performance.

Table 5.1 shows the *total_time* average and standard deviation for the experiment. Figure 5.2 compares the total time using a different number of instances; on X-axis, we have the number of parallel instances (1 means a single centralized instance), while on Y-axis, we have the experiments' *total_time* in second. We can see that between 1 and 3 instances, there is already a considerable gain in performance. We can see that by increasing the number of parallel instances, we still get better performance.

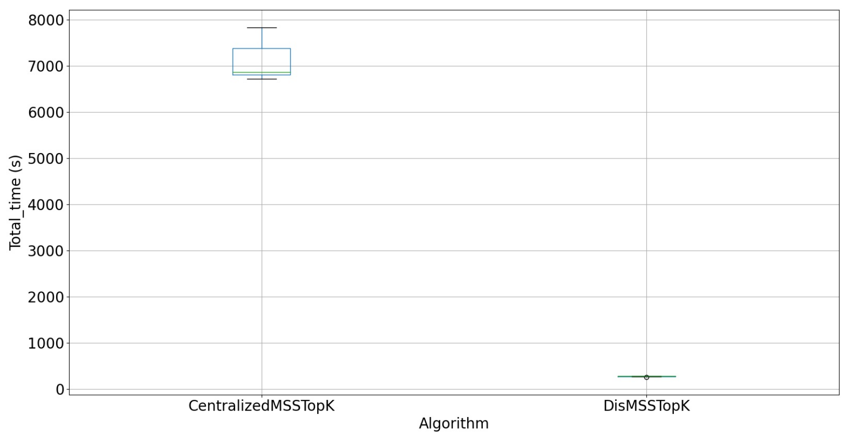


Figure 5.1: Baseline Algorithm: comparison between Centralized and Parallel versions.

	1 Instance	**3 Instances**	**6 Instances**	**10 Instances**
Average (s)	7117.44	868.55	270.13	152.69
Standard Deviation (s)	426.03	23.70	8.67	9.09

Table 5.1: Baseline Algorithm: *total_time* average and standard deviation for *num_instances* comparison.

5.3.1.2 *MinTop-K*

This experiment is equivalent to the one in Section 5.3.1.1, but we use a different algorithm; we want to verify that by parallelizing the MinTop-K algorithm, we can achieve better performance.

Figure 5.3 compares the centralized and parallel version of the MinTop-K algorithm. The experiment confirms our hypothesis; we see that, as in the Baseline Algorithm, we have improvements in terms of *total_time*.

Figure 5.4 shows the experiment's result using different numbers of parallel instances. On Y-axis, we have the *total_time* in second, while X-axis represents the number of parallel instances. The MinTop-K algorithm shows the same trend as the baseline algorithm, but we can see from the values measured for both experiments that it is more performant in terms of *total_time* than the baseline algorithm. Tabel 5.2 contains the *total_time* average and standard deviation for the experiment.

Finally, to end the comparison between the centralized and parallel versions of the proposed algorithms, we group the

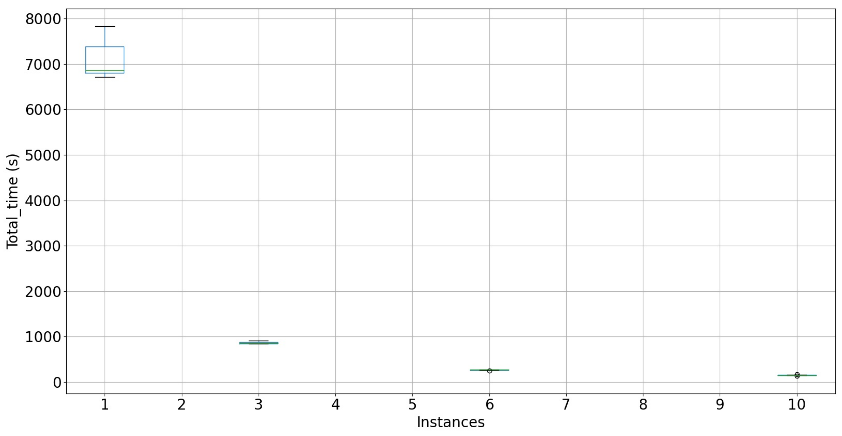


Figure 5.2: Baseline Algorithm: comparison on *num_instances*.

	1 Instance	**3 Instances**	**6 Instances**	**10 Instances**
Average (s)	5632.19	677.58	199.46	119.01
Standard Deviation (s)	58.67	15.22	11.67	6.57

Table 5.2: MinTop-K Algorithm: *total_time* average and standard deviation for *num_instances* comparison.

experiments' results into a single plot. For each experiment, we use *k=10*, and for the parallel ones, we use 6 instances. As shown in Figure 5.5, both versions of MinTop-K are faster than the respective versions of the baseline algorithm.

Moreover, Table 5.3 reports the parallel version's speed-up over the centralized one. The columns represent the number of parallel instances, while each row represents the algorithm. Here, we can see better how significant the improvement is in the parallel versions.

The speed-up is computed as the ratio between the average *total_time* of the centralized and the parallel versions.

$$avg\_total\_time = \sum_{d=1}^{D} \frac{total\_time_d}{D} \tag{5.6}$$

$$speedup = \frac{avg\_total\_time_{Centralized}}{avg\_total\_time_{Parallel}} \tag{5.7}$$

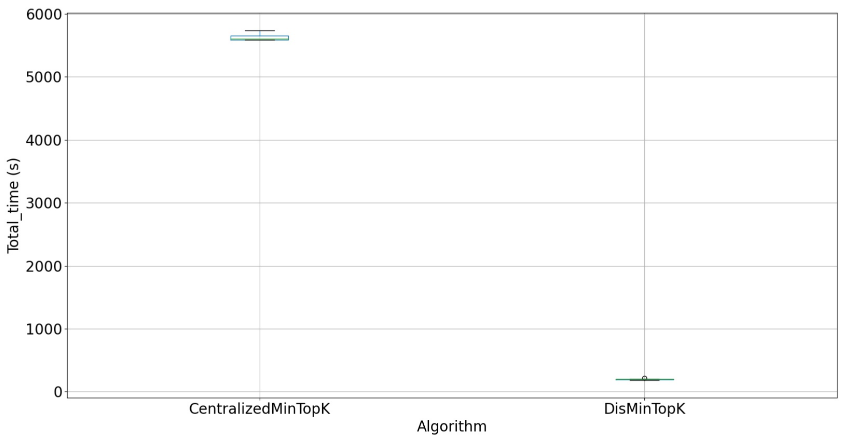


Figure 5.3: MinTop-K algorithm: comparison between Centralized and Parallel versions.

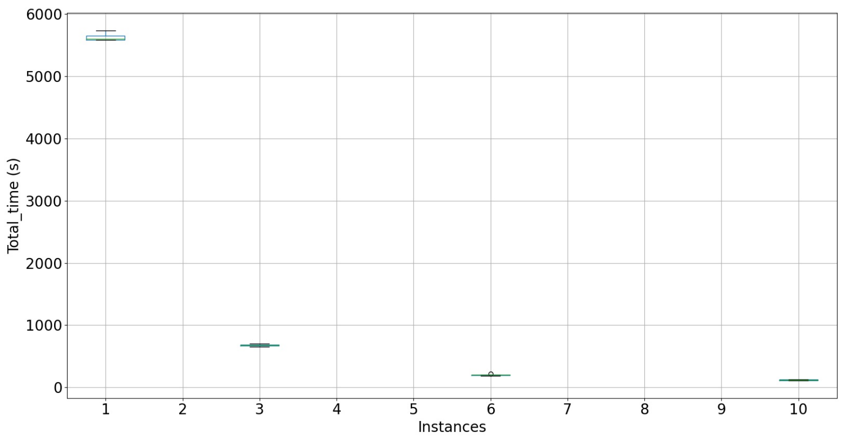


Figure 5.4: MinTop-K algorithm: comparison on *num_instances*.

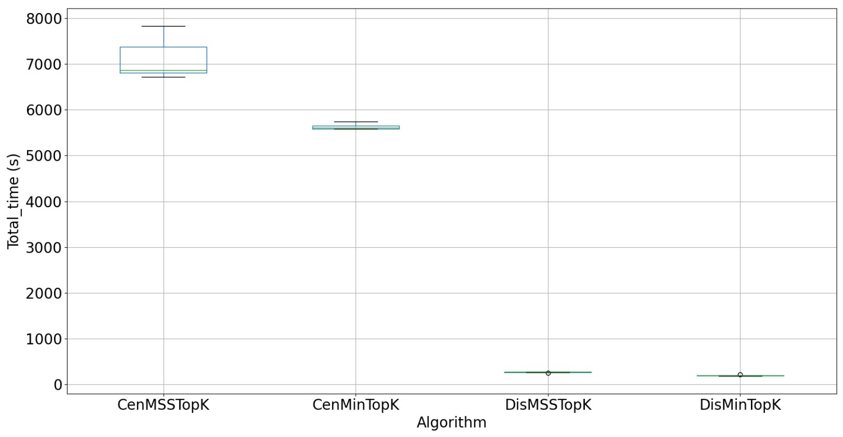


Figure 5.5: Algorithms comparison.

	3 Instances	**6 Instances**	**10 Instances**
Baseline	8.19	26.34	46.61
MinTop-K	8.31	28.71	47.32

Table 5.3: Parallelization speed-up

5.3.2 *Top-K Comparison*

Once we confirm our hypothesis that by parallelizing the streaming top-k algorithms, we can achieve better performance; we wanted to seek the correlation between the value of top-k and the *total_time* measured.

Before we started with the experiments, we reasoned on the possible outcomes. In order to explain our reasoning, it is better to recap the structure of our parallel solutions: we have *num_instances* parallel instances of the same application that send their partial top-k result to the centralized application, which aggregates the partial results and compute the final top-k sorting them and keeping the first *k* elements.

Starting from this, we expected that for large enough values of top-k, the experiment's total time would grow up significantly. Increasing the top-k's value means having bigger partial top-k results from the parallel instances, leading to an even larger amount of data that the centralized application has to sort to compute the final top-k result. For the same reason, we think that small *k*'s values would not significantly influence the measurements.

5.3.2.1 *Parallel Baseline Algorithm*

In this subsection, we show the results of the experiments described in Section 5.2.3.

Figure 5.6 shows the comparison between the total time for computing top-k results, using different k values. On X-axis, we have the top-k values, while on Y-axis, we have the *total_time* measured in seconds. The upper cap in the error bar represents the max value of *total_time*, the downer cap represents the min value of *total_time*, and the central marker represents the average value of *total_time* for a given experiment.

We can see from this plot that starting from $k = 100$, the *total_time* value grows exponentially. We can confirm our first hypothesis that after a large enough k value, the time needed to compute the top-k results grows exponentially.

Our second hypothesis is confirmed too. Figure 5.7 focuses on the top-k values that are smaller than the threshold we defined before ($k = 100$). Thanks to the box-plots, we can see that the *total_time* values do not vary substantially between the experiments, considering that the *avg_total_time*'s variation between $k = 5$ and $k = 100$ is 30 s, while its variation between $k = 100$ and $k = 150$ is 210 s.

Table 5.4 contains the average, the min and the max values of *total_time* measured during the experiments.

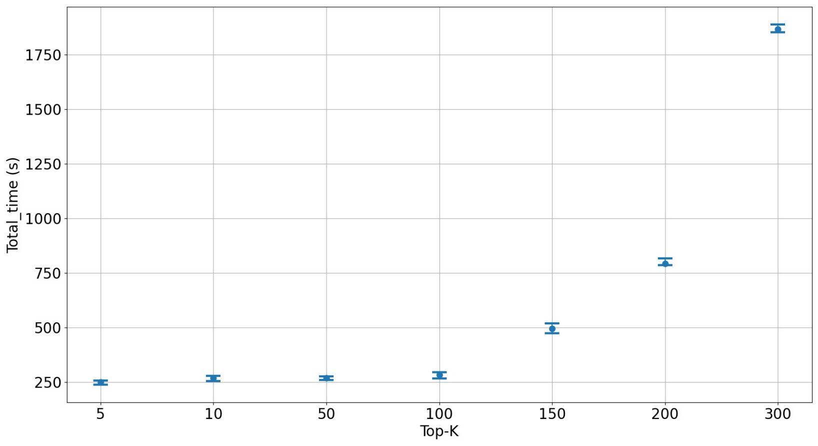


Figure 5.6: Baseline Algorithm: *total_time* for different k parameter values.

	Top-5	Top-10	Top-50	Top-100	Top-150	Top-200	Top-300
Average (s)	250.63	270.13	269.97	282.38	496.12	793.03	1865.77
Min (s)	239.15	254.54	260.27	267.64	474.05	786.86	1851.96
Max (s)	256.20	279.57	275.54	295.13	518.45	816.67	1887.47

Table 5.4: Baseline Algorithm: *total_time* average, min and max for *k* values comparison.

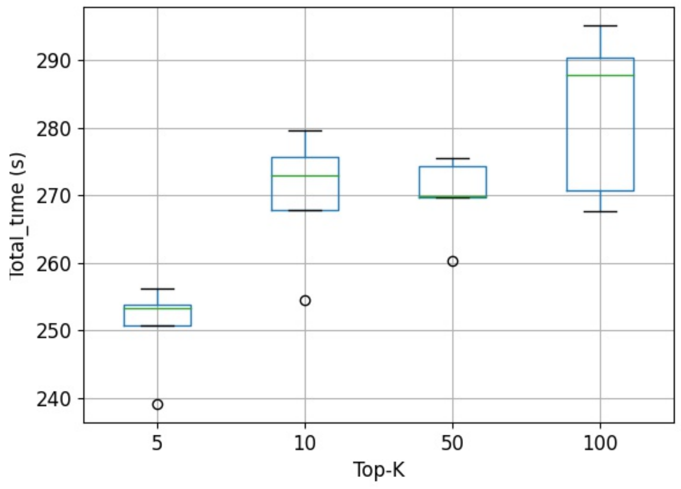


Figure 5.7: Baseline Algorithm: focus on small values of *k*.

5.3.2.2 *Parallel MinTop-K*

Here we want to confirm our hypothesis for the MinTop-K algorithm case. Section 5.2.4 describes in detail the design of these experiments; we used different values of top-k parameter, *k*, to seek for a correspondence between its value and the *total_time* of the experiment.

In Figure 5.8, we have on X-axis the top-k values and on Y-axis the *total_time* in seconds. The plot shows how the different values of *k* used influence the experiment's duration. We can see how the Y-axis values grow exponentially from top-K=100. It is true that starting from a certain value of *k*, $k = 100$ in our case, the experiment's *total_time* grows exponentially because of the increasing work that the centralized aggregator has to perform to sort all the partial top-k results received.

Figure 5.9 shows the results that confirm our second hypothesis. We can see that for top-k values inside the range [5;100], we have a small variation of *total_time*, meaning that top-k values inside that range do not substantially influence *total_time*. We have seen from the experiments that the *avg_total_time* variation between $k = 100$ and $k = 150$ is 220 s, while its variation between $k = 5$ and $k = 100$ is only 28 s.

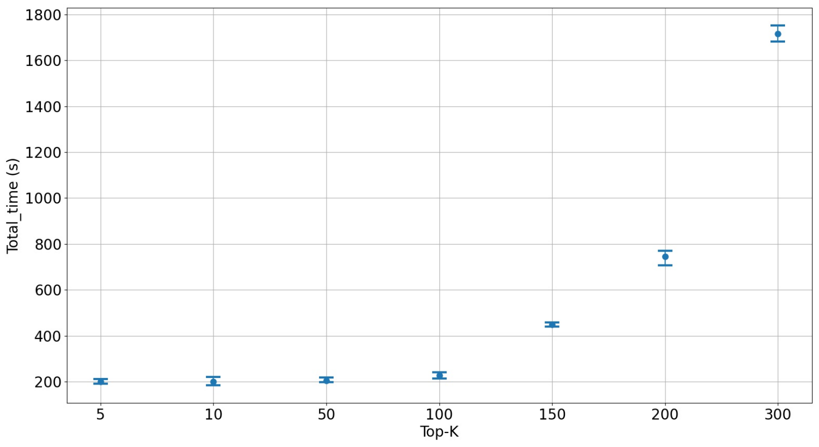


Figure 5.8: MinTop-K: *total_time* for different k parameter values.

Table 5.5 sums up the experiments' results showing the average, the minimum, and the maximum values of *total_time* measured varying the top-k.

	Top-5	**Top-10**	**Top-50**	**Top-100**	**Top-150**	**Top-200**	**Top-300**
Average (s)	200.32	199.46	203.78	228.20	448.64	745.23	1715.86
Min (s)	190.41	185.09	197.02	213.64	438.84	707.17	1681.89
Max (s)	210.71	219.96	217.46	240.53	458.48	769.92	1751.23

Table 5.5: MinTop-K Algorithm: *total_time* average, min and max for k values comparison.

5.3.3 *Algorithms Comparison*

In the last part of the experiments' results, we compare the baseline and MinTop-K algorithms. We know that the centralized version of MinTop-K outperforms the baseline algorithm; is it valid for the parallel version too? To answer this question, we run both algorithms with the same configuration; we perform seven different experiments, varying only the k parameter.

We expect that the results show that the parallel MinTop-K outperforms the parallel baseline version. Both have the centralized aggregator as the last processing unit, so this part should weigh in the same way for both. The differences are inside the parallel instances, but we know that the centralized MinTop-K

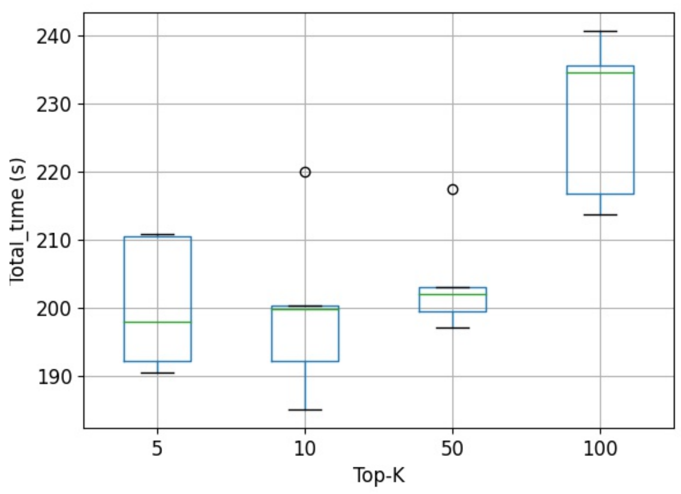


Figure 5.9: MinTop-K: focus on small values of k.

outperforms the baseline algorithm and since each parallel instance runs exactly the centralized algorithm's version but on a subset of the window, why should the performance change?

Figure 5.10 shows in blue the baseline algorithm's error bars for the different values of k, while the red ones refer to the MinTop-K algorithm. We can see that every downer blue cap, so the min value for the baseline algorithm, is above every upper red cap, so the max value for the MinTop-K algorithm; this means that even the MinTop-K's worst case is better than the baseline's best case.

Table 5.6 compares the *total_time* average between the baseline and the MinTop-K algorithms for different top-k values. We can see that baseline data are always larger than the MinTop-Ks.

	Top-5	**Top-10**	**Top-50**	**Top-100**	**Top-150**	**Top-200**	**Top-300**
Baseline average (s)	250.63	270.13	269.97	282.38	496.12	793.03	1865.77
MinTop-K average (s)	200.32	199.46	203.78	228.20	448.64	745.23	1715.86

Table 5.6: Baseline and MinTop-K algorithms' comparison: *total_time* average varying k parameter.

Moreover, we want to calculate the speed-up we achieve using the parallel MinTop-K with respect to the baseline algorithm. We compute the speed-up as the ratio between the average

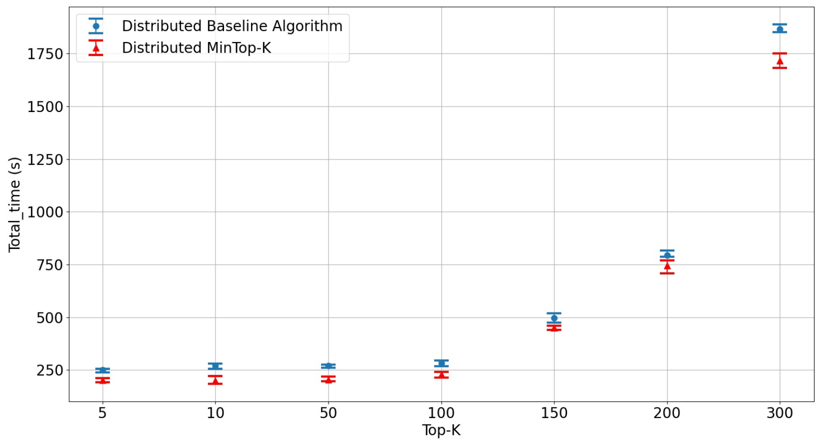


Figure 5.10: Baseline and MinTop-K algorithms' comparison: *total_time* for different k parameter values.

total_time of the baseline algorithm and the average *total_time* of the MinTop-K algorithm.

$$\text{avg\_total\_time} = \sum_{d=1}^{D} \frac{\text{total\_time}_d}{D} \tag{5.8}$$

$$\text{speedup} = \frac{\text{avg\_total\_time}_{\text{Baseline}}}{\text{avg\_total\_time}_{\text{MinTop-K}}} \tag{5.9}$$

$$\text{worst\_case\_speedup} = \frac{\text{min\_total\_time}_{\text{Baseline}}}{\text{max\_total\_time}_{\text{MinTop-K}}} \tag{5.10}$$

$$\text{best\_case\_speedup} = \frac{\text{max\_total\_time}_{\text{Baseline}}}{\text{min\_total\_time}_{\text{MinTop-K}}} \tag{5.11}$$

	Top-5	**Top-10**	**Top-50**	**Top-100**	**Top-150**	**Top-200**	**Top-300**
speed-up	1.25	1.35	1.32	1.23	1.10	1.06	1.08

Table 5.7: Speed-up MinTop-K over baseline algorithm

Table 5.7 reports the speed-up achieved for each k value, which is bigger for small values of k and decreases for larger values. Figure 5.11 shows the *speed-up*'s trend. On X-axis, we have the top-k values used in the experiments, while on Y-axis, we have

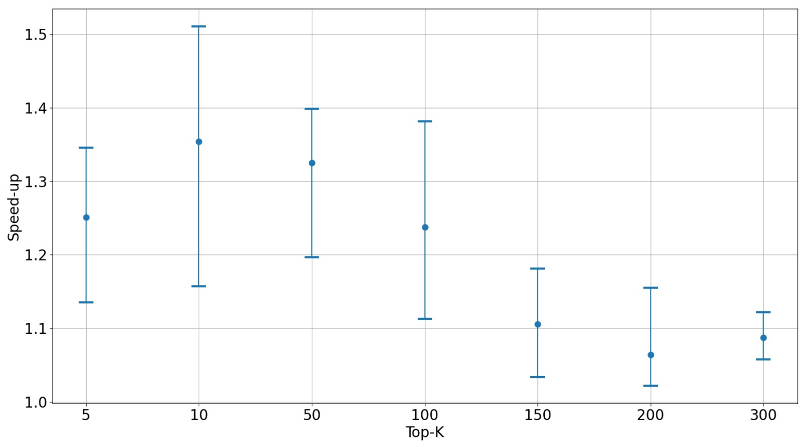


Figure 5.11: Speed-up MinTop-K over baseline algorithm

the speed-up. The blue circles represent the speed-up computed using the average *total_time* as explained in the Equation 5.9. The downer caps indicate the worst-case speed-up, computed as the ratio between the minimum *total_time* value for the baseline algorithm and the maximum *total_time* value for the MinTop-K algorithm (Equation 5.10). The upper caps represent the best-case speed-up, computed as the ratio between the maximum *total_time* value for the baseline algorithm and the minimum *total_time* value for the MinTop-K algorithm (Equation 5.11). We can see that values in the left part of the plot are larger than those on the right side.

5.4 FINDINGS

In this section, we summarize the results we obtained from the experiments, but first, we want to recap our research question and the hypotheses we made.

Through this work, we wanted to understand if it is worth parallelizing streaming top-k algorithms in Kafka. Our first hypothesis is that we can achieve better performance by splitting the work into parallel instances than using a single instance.

Secondary to this, we seek a correlation between the performance of the application and the top-k parameter's size, *k*. This seeking led us to our second hypothesis: for large enough top-k values, the application's performance will significantly decrease, while small top-k will not substantially influence the performance.

Lastly, we wanted to understand if the MinTop-K algorithm implemented using Kafka Stream would be as convenient as it shows in [16], meaning that the MinTop-K algorithm will outperform the naive Materialize Score & Sort approach.

Thanks to the results shown in Subsection 5.3.1, we can confirm our first hypothesis; both the baseline and the MinTop-K algorithms' parallel versions outperform their centralized versions. Moreover, results show that we take less time to process all the input data by increasing parallel instances.

Experiments' results in Subsection 5.3.2 show that the *total_time* of our experiments grows exponentially when the top-k value is larger than 100, while under this threshold, we do not see any considerable changes in the measurements. We can confirm our second hypothesis under our experimental conditions.

As we can see from Subsection 5.3.3, our implementation of MinTop-K using Kafka Stream outperforms the baseline implementation. We have better performance in every experiment; however, the speed-up is higher in experiments with small *k* values. The gain of MinTop-K over the baseline algorithm is probably mitigated by the centralized aggregator's work, which we know influence more the cost when the top-k value increases, so that is why we reasonably have lower speed-up in experiments with high *k*. To sum up, we can affirm that it is convenient to use our implementation of the MinTop-K algorithm using Kafka Stream over the baseline algorithm.

6 CONCLUSIONS & FUTURE WORKS

In this final chapter, we sum up the work we have done, we present the conclusion we have drawn from the experiments in Section 6.1, and we outline some directions for future works in Section 6.2.

6.1 CONCLUSIONS

This comparative research aimed to determine if it is worth parallelizing streaming top-k algorithms to achieve better performance for real-time and near-real-time applications. Secondary, we sought a correlation between the top-k parameter, meaning the number of outputs for each window, and *the total_time* that we used as a KPI. Moreover, we wanted to understand if Apache Kafka could be the right distributed streaming platform to implement streaming top-k algorithms.

In the beginning, we did some researches on state of the art for streaming top-k query computation, finding out that Yang et al.'s MinTop-K algorithm could have been the right candidate for our research. Then, we worked incrementally on the design and implementation of the algorithms. We started with the centralized version of the algorithm that we would use as a baseline; then, we moved to its parallel version. Finally, we implemented the MinTop-K algorithm, starting with its centralized version and then realizing the parallel one. Once all the algorithms were implemented, we designed our ad-hoc experiments to answer our research questions. We compared a centralized and parallel version of each algorithm using different degrees of parallelization; we set up experiments using different top-k values to understand how this parameter could influence the application's performance.

Based on these experiments, we can conclude that there are significant advantages in parallelizing streaming top-k algorithms; that the top-k parameter considerably influences the performance negatively when it assumes large values, while the application's performance is higher and nearly stable when *k* assumes small values. Lastly, we can state that implementing streaming top-k algorithms using Kafka Streams could be both an excellent solution for real-time and near-real-time applications and a baseline to build KSQL operator upon it. Also, it is possible to leverage Kafka's benefits, such as the scalability and load balancing, when building a streaming top-k application; focusing only on the application itself would ease the development of stream monitoring systems like the ones presented in Section 3.1.

6.2 FUTURE WORKS

Future development of our work can move in different research directions.

It could be possible to implement other streaming top-k algorithms to perform more comparisons between them or study how to exploit existing Kafka's APIs or plug-ins to implement new algorithms. An example could be the TopK+N algorithm, presented by Zahmatkesh and Valle in their *Relevant Query Answering over Streaming and Distributed Data - A Study for RDF Streams and Evolving Web Data*[17], which could exploit Kafka Connect API [1], to collect the updates of a distributed data source, used to perform joins with the data stream.

Another possible direction for our work could be to re-run the experiments inside a real distributed system. Thus, it could be possible to understand if the results we achieved with our

1 https://docs.confluent.io/current/connect/index.html

algorithms' parallelization could be achieved by distributing the parallel instances into different physical machines.

For our parallel implementations, we assumed that the window's *size* and *hopping_size* must be a multiple of the number of parallel instances, so it would be interesting to relax this assumption and manage the distribution of the data inside a window differently.

Moreover, it would be worth focusing on implementations based on logical windows instead of physical ones, maybe studying deeper how Kafka Streams' windowing operators work with logical windows.

Last but not least, it would be possible to implement KSQL operators for streaming top-k based on the Kafka Streams' implementation that we proposed in this thesis. An example could be a different **TOPK** operator that works with more than one column.

BIBLIOGRAPHY

[1] Daniel J. Abadi et al. "Aurora: a new model and architecture for data stream management." In: *VLDB J.* 12.2 (2003), pp. 120–139. DOI: 10.1007/s00778-003-0095-z. URL: https://doi.org/10.1007/s00778-003-0095-z (cit. on pp. 5, 7).

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. "The CQL continuous query language: semantic foundations and query execution." In: *VLDB J.* 15.2 (2006), pp. 121–142. DOI: 10.1007/s00778-004-0147-z. URL: https://doi.org/10.1007/s00778-004-0147-z (cit. on p. 7).

[3] Arvind Arasu et al. "STREAM: The Stanford Stream Data Manager." In: *IEEE Data Eng. Bull.* 26.1 (2003), pp. 19–26. URL: http://sites.computer.org/debull/A03mar/paper.ps (cit. on p. 7).

[4] Brian Babcock and Chris Olston. "Distributed Top-K Monitoring." In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. Ed. by Alon Y. Halevy, Zachary G. Ives, and AnHai Doan. ACM, 2003, pp. 28–39. ISBN: 1-58113-634-X. DOI: 10.1145/872757.872764. URL: https://doi.org/10.1145/872757.872764 (cit. on pp. 7, 24).

[5] Brian Babcock et al. "Models and Issues in Data Stream Systems." In: *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*. Ed. by Lucian Popa, Serge Abiteboul, and Phokion G. Kolaitis. ACM, 2002, pp. 1–16. ISBN: 1-58113-507-6. DOI: 10.1145/543613.543615. URL: https://doi.org/10.1145/543613.543615 (cit. on pp. 5, 21).

[6] Parisa Haghani, Sebastian Michel, and Karl Aberer. "Evaluating top-k queries over incomplete data streams." In: *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*. Ed. by David Wai-Lok Cheung et al. ACM, 2009, pp. 877–886. ISBN: 978-1-60558-512-3. DOI: 10.1145/1645953.1646064. URL: https://doi.org/10.1145/1645953.1646064 (cit. on p. 8).

[7] Hojjat Jafarpour and Rohan Desai. “KSQL: Streaming SQL Engine for Apache Kafka.” In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. Ed. by Melanie Herschel et al. OpenProceedings.org, 2019, pp. 524–533. ISBN: 978-3-89318-081-3. DOI: 10.5441/002/edbt.2019.48. URL: https://doi.org/10.5441/002/edbt.2019.48 (cit. on p. 18).

[8] Kostas Kolomvatsos, Christos Anagnostopoulos, and Stathes Hadjiefthymiades. “A time optimized scheme for top-k list maintenance over incomplete data streams.” In: *Inf. Sci.* 311 (2015), pp. 59–73. DOI: 10.1016/j.ins.2015.03.035. URL: https://doi.org/10.1016/j.ins.2015.03.035 (cit. on p. 8).

[9] Jay Kreps. “Kafka : a Distributed Messaging System for Log Processing.” In: 2011 (cit. on p. 14).

[10] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. “Continuous monitoring of top-k queries over sliding windows.” In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. Ed. by Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis. ACM, 2006, pp. 635–646. ISBN: 1-59593-256-9. DOI: 10.1145/1142473.1142544. URL: https://doi.org/10.1145/1142473.1142544 (cit. on pp. 8, 24).

[11] Weilong Ren, Xiang Lian, and Kambiz Ghazinour. “Effective and efficient top-*k* query processing over incomplete data streams.” In: *Inf. Sci.* 544 (2021), pp. 343–371. DOI: 10.1016/j.ins.2020.08.011. URL: https://doi.org/10.1016/j.ins.2020.08.011 (cit. on p. 8).

[12] Matthias J. Sax. “Apache Kafka.” In: *Encyclopedia of Big Data Technologies*. Ed. by Sherif Sakr and Albert Y. Zomaya. Springer, 2019. ISBN: 978-3-319-63962-8. DOI: 10.1007/978-3-319-63962-8_196-1. URL: https://doi.org/10.1007/978-3-319-63962-8%5C_196-1 (cit. on p. 14).

[13] Matthias J. Sax et al. “Streams and Tables: Two Sides of the Same Coin.” In: *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics, BIRTE 2018, Rio de Janeiro, Brazil, August 27, 2018*. Ed. by Malú Castellanos et al. ACM, 2018, 1:1–1:10. DOI: 10.1145/3242153.3242155. URL: https://doi.org/10.1145/3242153.3242155 (cit. on p. 17).

[14] Shaoxu Song and Lei Chen. “Differential dependencies: Reasoning and discovery.” In: *ACM Trans. Database Syst.* 36.3 (2011), 16:1–16:41. DOI: 10.1145/2000824.2000826. URL: https://doi.org/10.1145/2000824.2000826 (cit. on p. 8).

[15] Michael Stonebraker, Ugur Çetintemel, and Stanley B. Zdonik. “The 8 requirements of real-time stream processing.” In: *SIGMOD Rec.* 34.4 (2005), pp. 42–47. DOI: 10.1145/1107499.1107504. URL: https://doi.org/10.1145/1107499.1107504 (cit. on p. 5).

[16] Di Yang et al. “An optimal strategy for monitoring top-k queries in streaming windows.” In: *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*. Ed. by Anastasia Ailamaki et al. ACM, 2011, pp. 57–68. ISBN: 978-1-4503-0528-0. DOI: 10.1145/1951365.1951375. URL: https://doi.org/10.1145/1951365.1951375 (cit. on pp. III, V, 2, 3, 10, 13, 24, 28, 60, 61).

[17] Shima Zahmatkesh and Emanuele Della Valle. *Relevant Query Answering over Streaming and Distributed Data - A Study for RDF Streams and Evolving Web Data*. Springer, 2020. ISBN: 978-3-030-38338-1. DOI: 10.1007/978-3-030-38339-8. URL: https://doi.org/10.1007/978-3-030-38339-8 (cit. on pp. 14, 15, 24, 62).

RINGRAZIAMENTI

Per prima cosa vorrei ringraziare il Prof. Emanuele Della Valle per avermi affiancato in questo lavoro e per avermi aiutato nella stesura di questo documento. Sono grato dei consigli che mi ha dispensato in questi mesi di lavoro, ne farò tesoro per il mio futuro, sia in ambito professionale che personale. Ancor di più lo ringrazio per avermi appassionato e avvicinato al mondo dello streaming di dati, il quale vorrò approfondire nel prossimo futuro. Inoltre, vorrei ringraziare il gruppo di ricerca del Prof. Della Valle per l'aiuto mostratomi durante le fasi preliminari di questa ricerca, ed in particolare la Dott.ssa. Shima Zahmatkesh, PhD, per il suo supporto durante le fasi di progettazione degli algoritmi.

Ringrazio la mia famiglia per avermi sempre sostenuto durante questi anni di studio, e per aver gioito con me per i traguardi raggiunti. Grazie mamma e papà per avermi sempre spronato a dare il meglio di me, per avermi cresciuto con l'idea che avrei potuto puntare al massimo, e per avermi sempre sostenuto in questo percorso ed in tutte le altre esperienze che mi hanno reso ciò che sono ora. E' soprattutto grazie a voi se ho raggiunto questo traguardo! Ringrazio i miei nonni, che mi hanno cresciuto come un figlio, e che mi hanno insegnato la disciplina che mi ha portato fin qui. Ringrazio i miei zii, che mi hanno allietato nei giorni di festa e mi hanno sostenuto quando ne ho avuto bisogno. Infine ringrazio i miei due cugini, che sono per me come fratelli.

Ringrazio i miei compagni ed amici universitari, con cui ho trascorso questo magnifico periodo, vi ringrazio per le intense sessioni di studio affrontate insieme, per le chiacchierate nelle pause studio e per tutti i momenti di svago che mi hanno aiutato a vivere al meglio questa avventura.

Ringrazio gli amici di una vita, con cui ho condiviso questo ed altri percorsi di vita, e con cui mi auguro di condividerne ancora. Vi ringrazio per essermi stati vicino sempre, anche quando ci separavano 500km. Grazie per le giornate chiusi in biblioteca a preparare esami, e per le giornate trascorse fuori, a ridere di cuore.

Per ultimo, ma non per importanza, ringrazio Giulia, la mia compagna di vita in questi ultimi tre anni. Ti ringrazio per

avermi sostenuto in questo percorso, e per essermi stata accanto nonostante la distanza. Grazie per tutti i momenti di gioia e di amore che abbiamo vissuto insieme, mi hanno sempre accompagnato in questi anni e mi hanno aiutato a diventare l'uomo che sono. Grazie per non essere gelosa del mio computer, con il quale a volte trascorro più tempo che con te. Questo non è il primo e non sarà l'ultimo traguardo che festeggeremo insieme.

Luca