JASMINE - Data Stream Processing

Simone Mancini
University of Rome - "Tor Vergata"
0259568
2simonemancini5@gmail.com

Francesco Ottaviano
University of Rome - "Tor Vergata"
0259193
fr.ottaviano@gmail.com

Andrea Silvi
University of Rome - "Tor Vergata"
0258596
andrea.silvi94@gmail.com

***Abstract*—Nowadays, processing data as they arrive is increasingly widespread. Huge amount of data coming from IoT sensors, websites and social networks requires efficient real-time analysis and processing. At the same time, the spread of information channels such as Internet has led to the possibility to easily access information.**

Some of the most prestigious newspaper's editorial staff has decided to allow users registered to their platforms to comment and discuss on articles they share.

This paper describes architecture and DSP frameworks of the system developed to analyze streaming data flow concerning comments on *New York Times* articles.

Keywords: Apache Flink, Data Stream Processing, Big Data, Apache Kafka, Cloud Computing, Distributed Systems.

I. Introduction

The purpose of this project is to develop an efficient real-time analytic system. Processing is performed on a dataset[1] with comments[2] related to some articles shared on *New York Times* website.

It is required to obtain some relevant information on the most popular articles and users and other info on when comments are created.

In the following paragraphs design choices and processing details are reported. The complete architecture and relevant metrics are also described.

II. Design Choices

The principal framework is *Apache Flink*, mainly chosen for its time management. It is required to perform the same queries with another data stream processing framework so that can be possible to compare their metrics. For this reason, it has been established to implement queries with *KafkaStreams* library.

Apache Kafka, supported by *Zookeeper*, has been chosen as publish-subscribe distributed platform to manage communication between system components, in particular simulator and core.

Redis stores some useful information to perform queries, as described later.

Programming languages used are *Java*, in particular for processing core and *Javascript* for streaming simulator implementation.

III. Data Replay

In order to guarantee data stream processing, it is necessary to simulate data flow according to the effective time the tuples have been generated. Skipping

[1] Reduced dataset.
[2] Created from Jan 2018 to Apr 2018.

this phase means to perform pure batch processing. So the team has developed a simulator that, scaling time properly, reproduces comments flow.

The first decision concerns which time field in the csv file[3] must be considered as the tuple generation time. In particular two fields are examined: *approveDate* and *createDate*. Even if both are not good candidates to be generation time[4], the team decided to look at *createDate* field as the timestamp the tuple is generated.

Simulator is therefore configured to produce stream according to the scaled[5] difference between each tuple's *createDate* value and the previous one's.

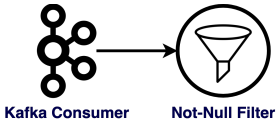
NodeJS has been used to implement data replay.

IV. Apache Flink

Apache Flink has been chosen as mainly data stream processing framework. In the following paragraphs, queries' processing is described in detail.

Parallelism is set to 4[6] and each query is required to be computed for 3 different time windows which the team has considered to be *tumbling*. Windows are considered by Flink starting from Thursday because they are aligned with epoch[7].

Before executing each query, stream coming from *Kafka* (*InputStream*) is filtered in order to discard tuples with null values.



A. Query 1

It is firstly asked to compute real time top 3 articles ranking. In particular, outputs must show those articles with the largest number of comments, both direct and indirect.

Time windows are different from other queries and they are required to be:

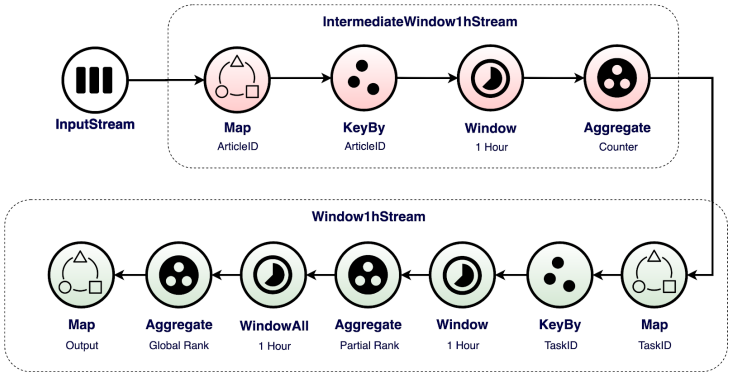
- 1 hour
- 24 hours
- 7 days

Results are finally displayed with the following schema:

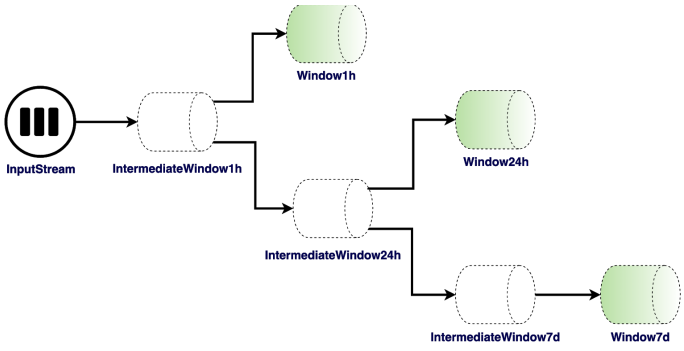
ts	artID_1	nCmnt_1	artID_2	nCmnt_2	artID_3	nCmnt_3

where *ts* field represents initial timestamp.

The following DAG represents the actions performed to get results on the first window.



Windows' chaining is finally implemented to improve efficiency.



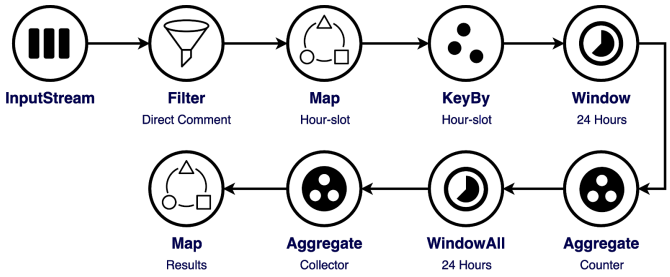
[3] Dataset is sorted by *createTime*.
[4] For example, a comment with recommendations updates would have the same tuple generation time.
[5] Configurable.
[6] Except for *windowAll* operator (=1).
[7] For example, month starts from 28 December.

B. Query 2

It is now required to analyze 2-hours time slots to show the number of direct comments[8] on three different time windows:

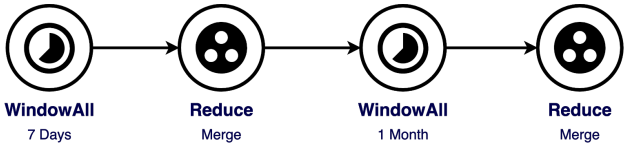
- 24 hours
- 7 days
- 1 month

The query's DAG for 24 hours (*window24hoursStream*) follows.

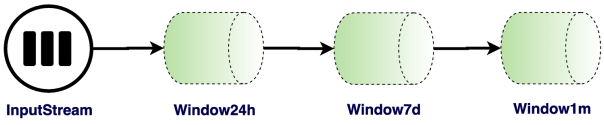


As the DAG shows, initial filter guarantees that only direct comments[9] are considered.

It is now easy to compute 7-days and 1-month window summing intermediate results[10].



Windows' chaining for the second query follows.



[8] Daily average values are considered because they are more significant to perform analysis.
[9] CommentType = COMMENT.
[10] It can be easily possible to compute mean by dividing for window.

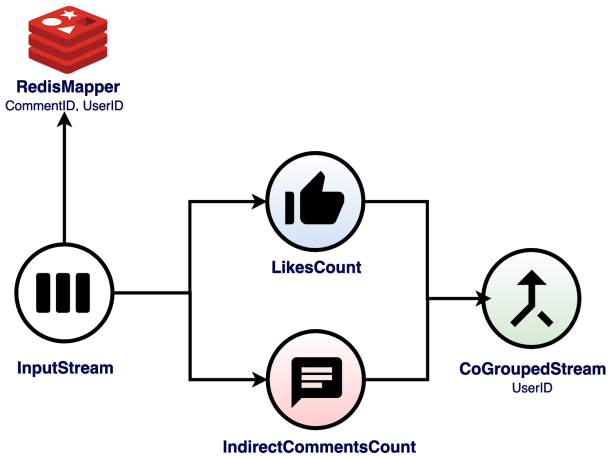
C. Query 3

The third query is about the most popular users. Popularity rating is proportional to the number of likes his direct comments get (a) and the discussions[11] he produces (b) according to the following relationship:

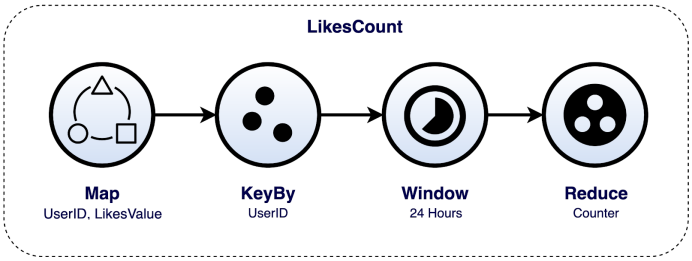
$$rating = w_a a + w_b b$$

where $w_a = 0.3$ and $w_b = 0.7$. Time windows are the same as the second query.

The following schema explains query processing at high level.



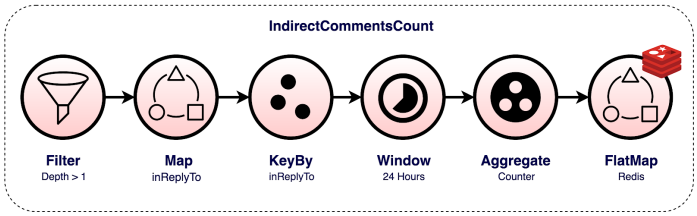
A first *DataStream*, *likesCount*, is created. Its operations are described in detail by the following DAG.



Another *DataStream* (*indirectCommentsCount*) is created from *inputStream* in order to be joined with the first one. It is obtained by filtering indirect comments[12]

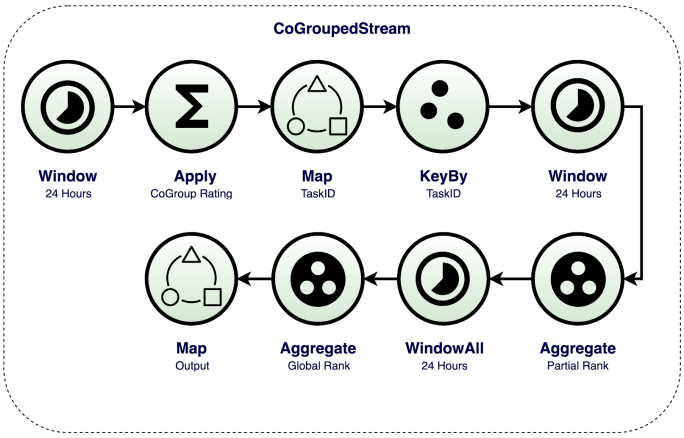
[11] Number of indirect comments to his comments.
[12] Depth > 1.

and by getting *UserID* related to the comment from status stored in *Redis*.

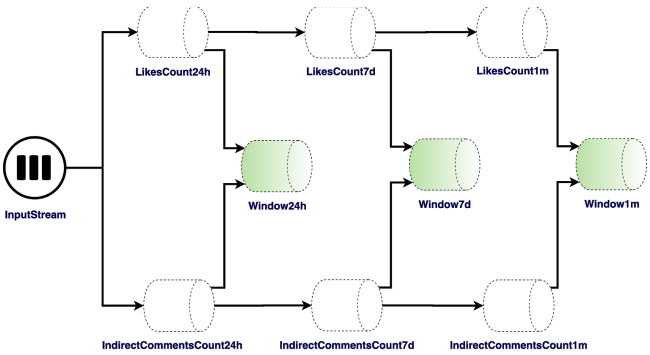


Storing state is necessary since *UserID* field in *indirectCommentsCount* tuples represents the ID of the user who has replied to the comment instead of the identification of the user the comment is directed to.

Finally, streams described before are joined on *UserID* through *coGroup* operation. Final stream for 24 hours is the following:



It is now reported windows' chaining for this query.



V. KafkaStreams

KafkaStreams library has been chosen for its lightness and for its compatibility with *Apache Kafka*. Unfortunately, poor documentation is provided so computing queries in an efficient way has required relevant efforts.

It has been used in continuous update mode because of *KafkaStreams*' purpose. Initially it has been tried to use *suppress* operator but it seems to introduce side effects in windows flushing and results retaining.

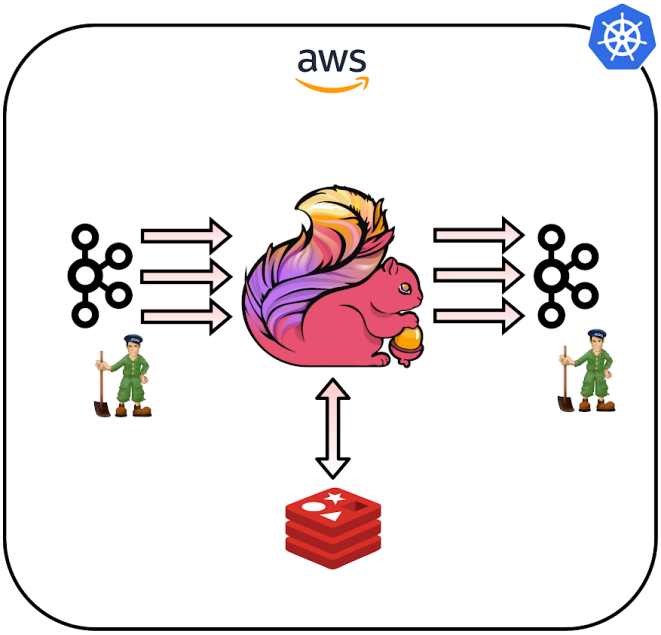
In order to compare KafkaStreams with Flink, it has been decided to keep the same logic in implementing queries but it has been hard to reach this goal.

For example, additional maps are required to shuffle data in distributed environment.

Moreover, aggregations have required keys manipulation and additional components to execute aggregations similarly to Flink.

VI. Architecture and Deployment

A system high-level overview is depicted by the following picture.



AWS available grants have been used to deploy system's architecture which consists of three principal

components: *Apache Flink*[13] for queries processing, *Apache Kafka*[14] used to get data from simulator and to save results and *Redis* to keep state whenever it is necessary. Moreover, the team decided to use *Kubernetes* as cluster manager.

In particular, m4-large EC2 instances[15] are deployed with the following configurations:

- *Kafka*: 1 node per machine, 3 nodes with replication factor 3 on topics.
- *Zookeeper*: 1 node per machine, 3 nodes.
- *Flink*: 1 job manager, 8 task manager.

VII. Evaluation

Evaluations concern the most significant metrics for data stream processing: *latency* and *throughput*.

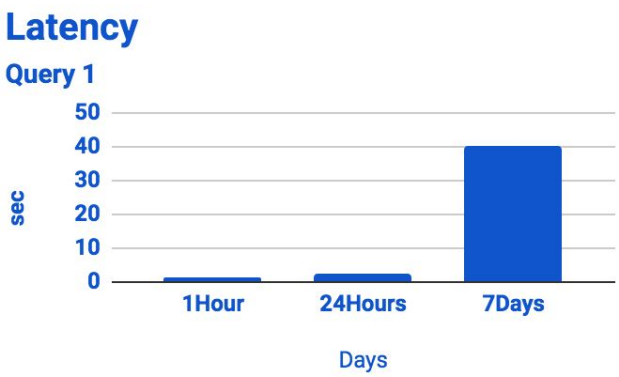
Evaluations are performed with time compression factor = 10.000.

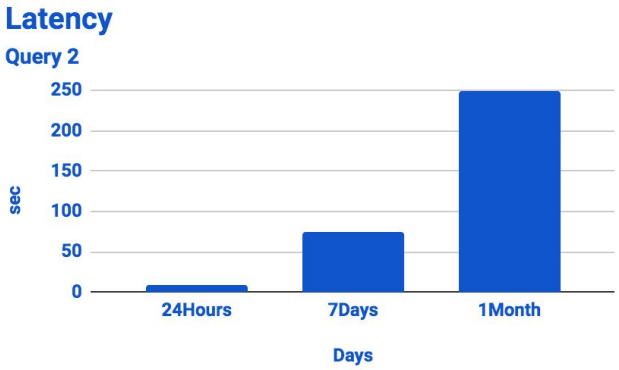
A. Apache Flink

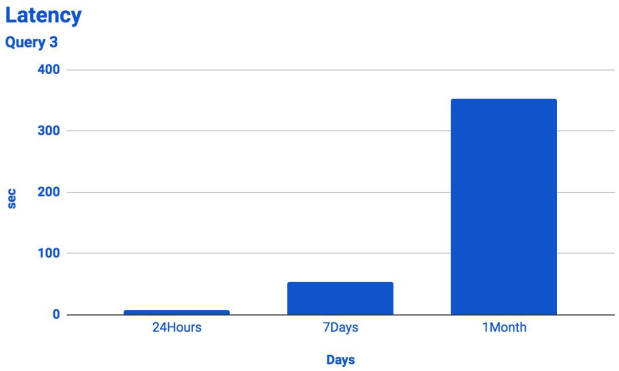
Apache Flink allows gathering metrics through its metric system but it has been established to compute them in two different ways.

I. Latency

Latency tracking mechanism is allowed when using *Flink* but it has some important limitations (e.g. processing time is not considered). So latency is evaluated by tracking manually a single tuple from when it enters the system (source) and when the first output arrives (sink).



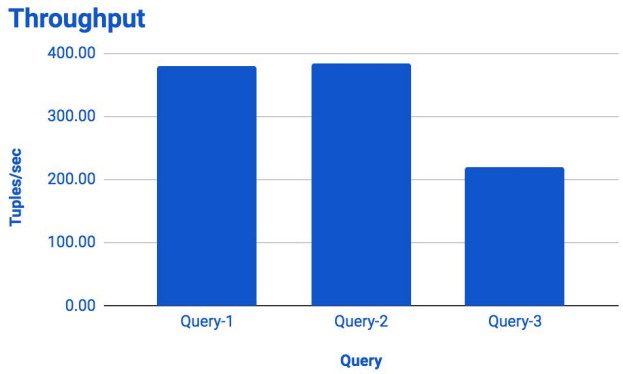




II. Throughput

Throughput can be computed as number of records emitted per second by an operator. In particular, *NumRecordsOutPerSecond* for source operators. Differently from latency metric, throughput values are gathered directly from *Flink* UI setting the simulator as fast as possible so to guarantee a relevant input load in order to show framework's behaviour when it is subjected to significant efforts.

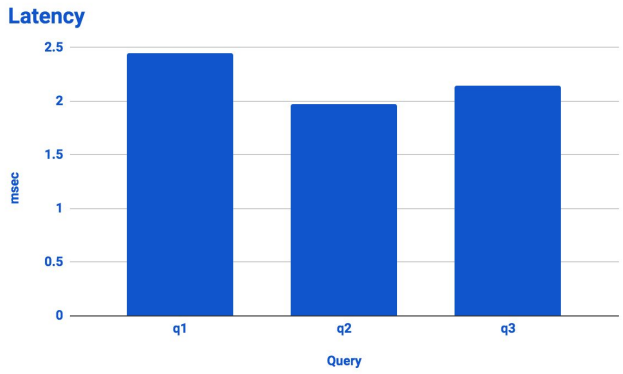
[13] Alternatively *KafkaStreams*.
[14] 3 brokers.
[15] 1 t2.micro as master; 4 m4.large as nodes.



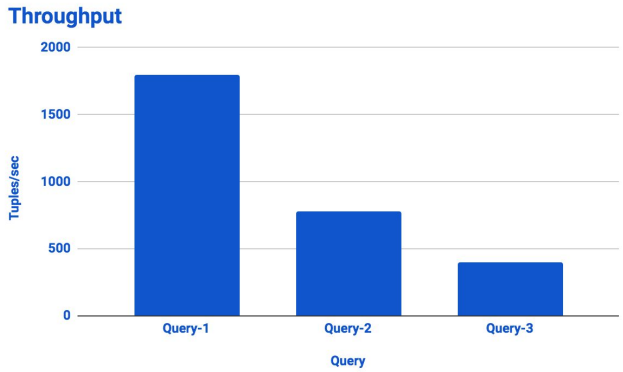
B. KafkaStreams

KafkaStreams reports metrics through JMX.

III. Latency



IV. Throughput



VIII. Configuration

To configure, deploy and run the application, follow the instructions reported on *README.txt* file at https://github.com/sashacodes/JASMINE-DSP.

Complete source code is available here: https://github.com/sashacodes/JASMINE-DSP.

Results of the performed queries are also reported into a separated file available into the Github repo.

Reference

[1] *Apache Flink website:* https://flink.apache.org
[2] *Apache Kafka website:* https://kafka.apache.org
[3] *KafkaStreams documentation:* https://kafka.apache.org/documentation/streams/
[4] *Kubernetes website*: https://kubernetes.io/it/