TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Incremental Windowed Aggregations at Apache Flink

Ntelmpizis Asterios

A thesis submitted in partial fulfillment of the requirements for the degree of Diploma in Electrical and Computer Engineering

Thesis Committee:

Prof. Deligiannakis Antonios (Supervisor)
Prof. Garofalakis Minos
Prof. Samoladas Vasilis

Abstract

Nowadays, stream data are produced at a constant and rapid pace, more and more applications attempt to use the streams in order to receive crucial decisions. This outcome can be achieved by using algorithms and data structures that effectively process large amounts of data. These data are called Big Data and can be generated from different sources (sensors, social media). Processing and analysis of Big Data has become essential. Synopses are used in queries in Big Data because of their quick response times. Synopses summarize data set and provide approximate answers to queries.

Apache Flink is one of the dominant systems for processing stream data. On data streams it is very important to calculate aggregated results and usually this is achievable using windows, since the number of streams is infinite. Results are, thus, produced after each window expires. However, Flink supports specific number of built-in implemented functions for windows.

The purpose of this work is to extend the number of built-in functions that can be supported by Flink, by allowing synopses to be computed and to then provide approximate results. In addition, to maximize performance, we must ensure that building the synopses is done during the time that data are inserted into their windows. This is very important to avoid the pitfall of processing the tuples of a window after it is closed, which would require a second pass over its elements.

Πρόλογος

Στις μέρες μας, δεδομένα παράγονται συνεχώς σε ασύλληπτους ρυθμούς και όλο και περισσότερες εφαρμογές προσπαθούν να χρησιμοποιήσουν όλα αυτά τα δεδομένα, για να πάρουν κρίσιμες αποφάσεις. Αυτό, μπορούν να το πετύχουν, χρησιμοποιώντας αλγορίθμους και δομές δεδομένων που επεξεργάζονται αποδοτικά μεγάλα σύνολα δεδομένων. Τα σύνολα αυτά, παράγονται από διάφορες πηγές (π.χ. αισθητήρες, μέσα κοινωνικής δικτύωσης) και ονομάζονται Μεγάλα Δεδομένα. Η επεξεργασία, καθώς και η ανάλυση των Μεγάλων Δεδομένων, έχει γίνει πλέον αναγκαία. Για τη γρήγορη απάντηση επερωτήσεων σε Μεγάλα Δεδομένα, χρησιμοποιούνται συνόψεις, οι οποίες συνοψίζουν το σύνολο δεδομένων και παρέχουν προσεγγιστικές απαντήσεις σε υποερωτήματα.

Το Apache Flink είναι ένα από τα κυρίαρχα συστήματα για επεξεργασία σε ροές δεδομένων. Πάνω σε ροές δεδομένων, είναι πολύ σημαντικό να υπολογίζουμε συναθροιστικά αποτελέσματα και συνήθως αυτό μπορεί να επιτευχθεί χρησιμοποιώντας παράθυρα, καθώς οι ροές είναι άπειρες. Έτσι, τα αποτελέσματα παράγονται μετά τη λήξη κάθε παραθύρου. Το Flink όμως υποστηρίζει συγκεκριμένο αριθμό ενσωματωμένων συναρτήσεων που έχουν υλοποιηθεί.

Σκοπός της διπλωματικής μας εργασίας είναι να επεκτείνουμε τον αριθμό των ενσωματωμένων συναρτήσεων που μπορεί να υποστηρίζει το Flink, επιτρέποντας τον υπολογισμό των συνόψεων και στη συνέχεια, την παροχή κατά προσέγγιση αποτελεσμάτων. Επιπλέον, για να μεγιστοποιήσουμε την απόδοση, πρέπει να διασφαλίσουμε ότι η κατασκευή των συνόψεων γίνεται κατά τη διάρκεια της εισαγωγής δεδομένων στα παράθυρα. Αυτό είναι πολύ σημαντικό για να αποφευχθεί η παγίδα της επεξεργασίας των πλειάδων ενός παραθύρου αφού κλείσει, κάτι που θα απαιτούσε ένα δεύτερο πέρασμα των στοιχείων του.

Acknowledgments

First of all, I would like to thank my supervisor, Prof. Antonios Deligiannakis, supervisor professor of this thesis for his essential and valuable guidance during my work. Furthermore, I am grateful to the rest members of examination committee, Prof. Vasilios Samoladas and Prof. Minos Garofalakis for their assistance too. I would like to thank my family members for their support and boost they gave me to my whole life. My last regards are referenced to my friends for their patience and the courage they gave me.

Contents

Chapter 1
Introduction

Massive data quantities are produced at a constant pace from many different types of sources (financing transactions, sensor networks) called Big Data. Data analysis grows rapidly in scientific (or non-scientific) areas. Most of them, use data to pick up valuable information in order to predict events for critical decisions in different aspects (medical issues etc). Data science has been transferred to every scientific field and has become a necessity. It is the development of scientific fields such as, statistics, predictive analytics, machine learning, data mining. These fields have become so evolutionary in recent years. As a consequence, they are responsible for the creation of data science.

One of the system for processing big data is Apache Flink. Apache Flink is an open source framework and distributed processing engine for large scale computations over unbounded and bounded data streams. Flink provides APIs for both Stream and Batch processing, and libraries for relational queries, complex event processing scenarios, graph processing and machine learning. DataStream programs in Flink are regular programs that implement transformations on data streams. One of the main transformations is window. Windows are at the heart of processing infinite streams. Flink supports the implementation of user defined functions, while natively supporting the sum, min and max aggregate functions.

In this work, research has been done on the functionalities of the windows. Some algorithms have been modified in order to perform different computations (Frequent Item, Median, TopK, Cardinality, Average, Membership, Sampling) on the windows. These algorithms are updated each time a new element is added to the window, rather than processing the elements of the windows after the window has been closed, which improves performance. The result is returned on window's shutdown.

1.1 Thesis Outline

In Chapter 2, we describe Apache Flink. In chapter 3, we state the study background and knowledge that required for usage of the stream algorithms. In Chapter 4, we analyze the characteristics of our WindowSynopsisLibrary library, and the way that it has been integrated to Flink. In Chapter 5, we present the results of this study. In Chapter 6, contains concluding remarks.

Chapter 2

Apache Flink

Apache Flink [1] is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. In 2010, a project started named "Stratosphere". This project is an object of study for three universities in Berlin. From 2014, it has been integrated to Apache and has been widely known as Flink. The core of Apache Flink is a distributed streaming dataflow engine written in Java and Scala. Flink's programs can be written in Scala, Java, Python, and SQL, and can be deployed in local, cluster or cloud mode. Flink does not provide a storage system but it uses "connectors" to thirdly-party systems like Apache Kafka and HDFS for data sources (data entry) and sinks (data export). Figure 2.1 depicts Flink's architecture.

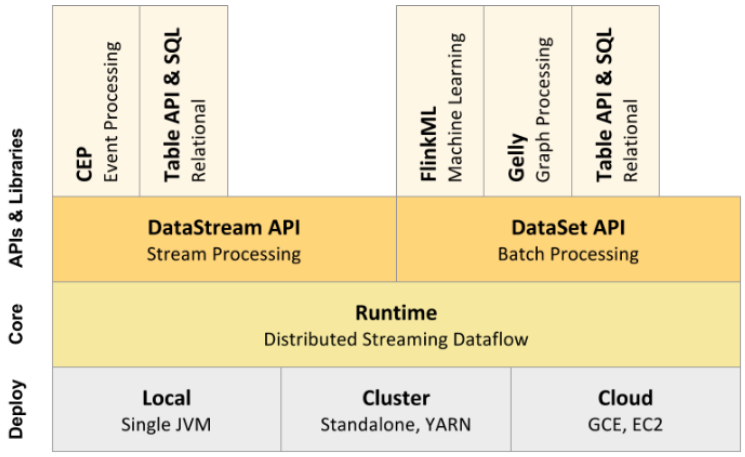


Figure 2.1: Architecture of Apache Flink (image from [1])

Figure 2.2 shows the three layered APIs that Flink provides. Each API offers a different trade-off between conciseness and expressiveness and targets different use cases.

ProcessFunctions is the most expressive function interface that Flink offers. ProcessFunctions processes individual events on streams (one or two) or events that were grouped in a window. ProcessFunctions enable control over time and state. Specifically, a ProcessFunction arbitrarily modify its state and register timers that will trigger a callback function in the future.

DataStream API provides many operators for processing stream. Some services are described in Section 2.3. The DataStream API is available for Java and Scala and is based on function that can be defined by extending interfaces such as Java or Scala lambda functions.

Table API and SQL are two relational APIs that Flink provides. These APIs are unified APIs that can process batch and stream. Apache Calcite is utilized by Table API and SQL for parsing, validation, and query optimization.

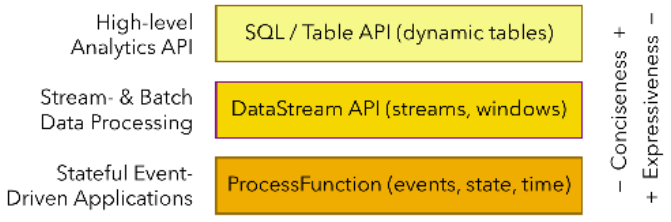


Figure 2.2: Layered APIs of Apache Flink (image from [1])

Flink has a variety of libraries for common data processing use cases. These libraries are included in an API and are not autonomous. So, they can benefit from all features of the API and be integrated with other libraries.

- **Complex Event Processing (CEP):** The CEP library is integrated with Flink's DataStream API and provides an API to specify patterns of events. Network intrusion detection, business process monitoring and fraud detection are some of the applications that can use the CEP library.

- **DataSet API:** The DataSet API is Flink's core API for batch processing applications. The DataSet API includes important transformations such as reduce, map, filter, co-group, and iterate. Algorithms and data structures that are used from all operations, manage serialized data in memory and save them to disk if the data size exceeds the size of memory.
- **Gelly:** Gelly is a library for scalable graph processing and analysis. Gelly is implemented on top of and integrated with the DataSet API and therefore, is favored for its scalable and robust operators.

2.1 Dataflow Programming Model

The basic building blocks of Flink programs are streams (is a flow of data records) and transformations (is an operation). A transformation takes one or more streams as input and produces one or more output streams as a result. During execution, Flink programs are mapped to streaming dataflows, consisting of streams and transformation operators. Each dataflow starts with at least one source and ends with at least one sink. The dataflows resemble arbitrary directed acyclic graphs (DAGs, Example in Figure 2.3)

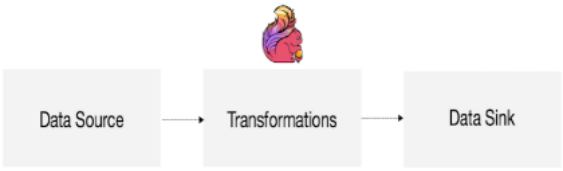


Figure 2.3: A directed acyclic graph of a dataflow (image from [1])

Figure 2.4 shows an example of a Flink program written in the DataStream API, along with the DAG of the streaming dataflow.

1. The program creates a data connector to consume data from the source (Apache Kafka) in the form of a stream of string records.
2. A Map operator uses the function "parse" for every string record so that it can transform the data stream of strings to events.

3. Method keyby groups data, based on key “id”. Every 10 seconds, an aggregation function is called to do calculations in events with the same key.

4. To store the results (of the aggregation function) to rolling files in the system, a data sink is used.

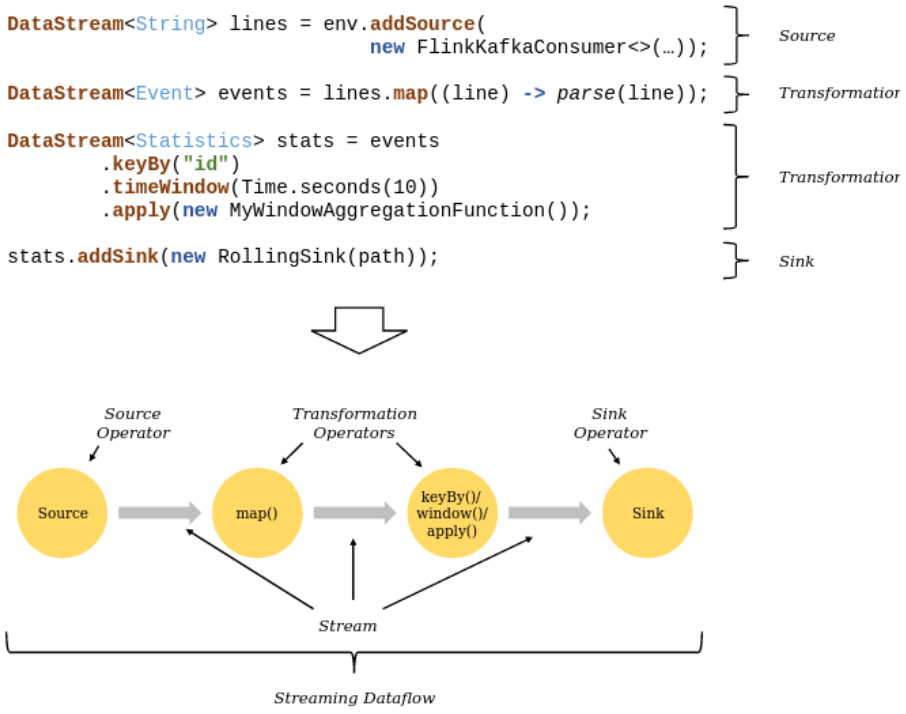


Figure 2.4: Streaming Dataflow (parallelism 1, image from [1])

About Flink’s programs that are executed in parallel, each stream has one or more stream partitions and each operator has one or more operator subtasks. The operator subtasks can execute in different threads of machines and are independent from each other. The program’s parallelism equals the number of operator subtasks. The parallelism level can vary to operators of the same program. Figure 2.5 shows the parallelism view of the previous example.

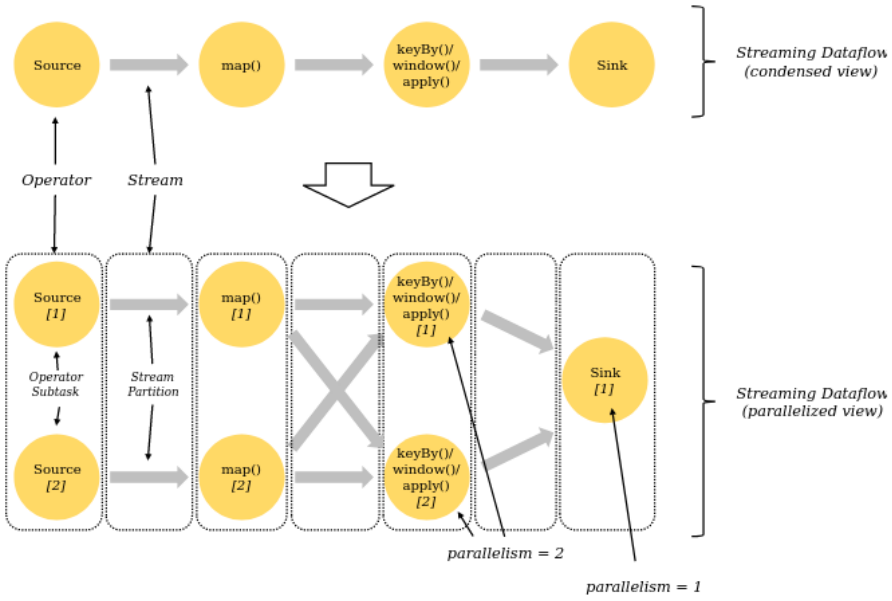


Figure 2.5: Example of a parallel streaming dataflow (image from [1])

Time
In a streaming program, there are different definitions of time:

- **Event Time:** The time that each individual event created on its producing device. This events usually described by a timestamp.
- **Ingestion time:** Is the time when an event enters the Flink dataflow at the source operator.
- **Processing Time:** Refers in time (system) that a machine requires to perform the corresponding operation.

2.2 Distributed Runtime Environment

In one distributed execution, Flink chains operator subtasks together into tasks, where each task is executed by one thread. Chaining operators together into tasks improves program's performance. As shown in Figure 2.5 above, we execute DAGs with parallelism five (five subtasks, and hence with five parallel threads).

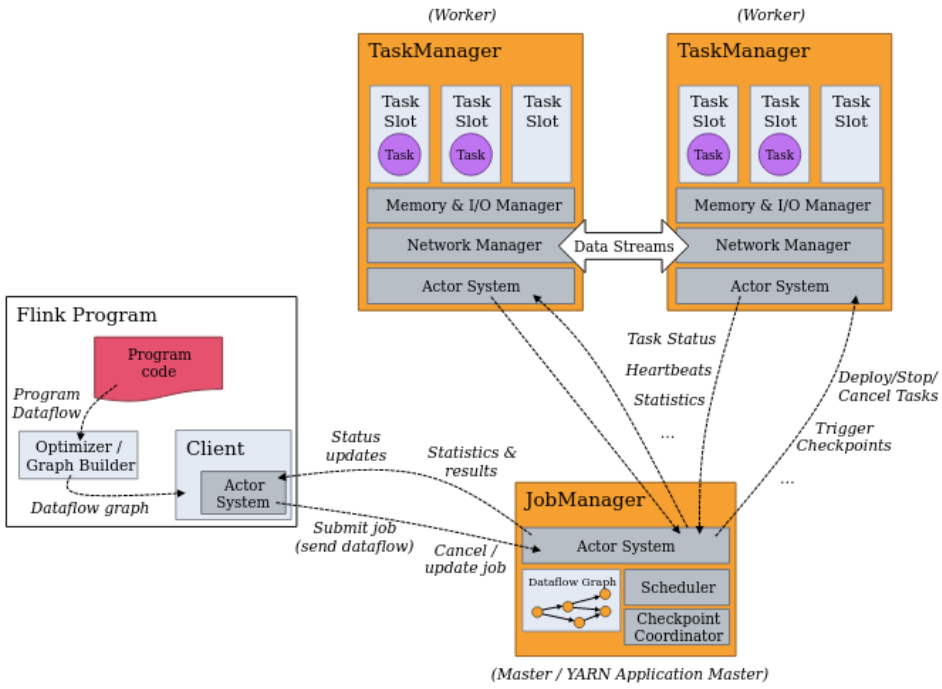


Figure 2.6: Distributed Architecture of the runtime environment (image from [1])

The Flink runtime consists of two types of processes (Figure 2.6) :

- The **JobManagers** (also called masters) coordinate the distributed execution. The JobManagers schedule tasks, react to finished tasks or execution failures, coordinate checkpoints, coordinate recovery on failures, etc. There is always one Job Manager. A high-availability setup will have multiple JobManagers, with a leader and the rest of them on hold.
- The **TaskManagers** (also called workers) execute the tasks of a dataflow, and buffer and exchange the data streams. There must exist at least one TaskManager. The smallest unit of resource scheduling is a task slot. The number of task slots indicates the number of concurrent processing tasks.

The JobManagers and TaskManagers can start in many ways such as: directly on the machines as a standalone cluster, in containers, or managed by resource frameworks

like YARN or Mesos. TaskManagers connect to JobManagers, in order to assign tasks.

2.3 DataStream API

DataStream is the core API for processing streams. DataStream programs in Flink can implement transformations such as defining windows, aggregating on data streams. Data streams are represented by special classes (e.g., DataStream<T>, KeyedStream<T, KEY>), which are immutable collections of data. Flink programs look like regular programs that transform DataStreams. Each program consists of the same basic parts:

1. Obtain a streaming execution environment. Function getExecutionEnvironment() defines automatic context that the program will execute (e.g local or remote environment).

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.
    getExecutionEnvironment();
```

2. Load or create input data from data sources. A connection is created with a source (Kafka topic, socket, txt, etc) that reads data.

```
DataStream<String> text = env.readTextFile("file:///path/to/outputFile");
DataStream<String> text = env.addSource(new FlinkKafkaConsumer<>(...));
```

3. Perform transformations on this data. DataStream API contains a set of stream operators. Some of them are described below.

   FlatMap: Takes one element and produces zero or more elements. A flatmap function splits sentences to words.

```
dataStream.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public void flatMap(String value, Collector<String> out)
```

```
        throws Exception {
        for(String word: value.split(" ")){
            out.collect(word);
        }
    }
});
```

KeyBy: This method splits a stream to streams with same key (described in Section 2.4).

```
dataStream.keyBy(value -> value.f0) // Key by the first element of a Tuple
```

Window: Windows group elements with the same key and other common characteristics like belonging to the same time interval (e.g., 12:00 - 12:05).

```
dataStream.keyBy(value -> value.f0).window(TumblingEventTimeWindows.of(Time
    .seconds(5))); // Last 5 seconds of data
```

WindowAll: In these windows belong all elements with some characteristic.

```
dataStream.windowAll(TumblingEventTimeWindows.of(Time.seconds(5))); // Last
     5 seconds of data
```

WindowFunction: Determines calculations that are going to take place on windows. The WindowFunction can be one of AggregateFunction, ReduceFunction, ProcessWindowFunction or FoldFunction. In the example below, ProcessAllWindowFunction is combined with an AggregateFunction to incrementally aggregate elements as they arrive in the window. When the window is closed, the ProcessWindowFunction will be provided with the aggregated result.

```
dataStream.windowAll(...).aggregate(new AggregateFunction(), new
    ProcessWindowFunction());
```

4. Store output results. Sink functions of DataStream API write down results to an external system. Below, it is shown an example of the creation of a sink.

```
dataStream.writeAsText("file:///path/to/outputFile");
```

5. Trigger the program execution: For the activation of the executions of transformations, the execution environment calls method execute().

```
env.execute("Job Name");
```

2.4 Windows

Windows are at the heart of processing infinite streams by dividing the stream into finite-size "buckets". On these buckets, different calculations can be made. A window is created as soon as the first element that should belong to this window arrives, and the window is completely removed when the time (event or processing time) passes its end timestamp plus the user-specified allowed lateness. The most important fact is the designation of the stream (Non-Keyed or Keyed). This happens before the definition of window.

Keyed Windows
Calling method "keyby", the stream is divided to x keyed streams where x is the number of keys. Any attribute can be set as key. A keyed stream can make parallel windows processing based on their keys. Specifically, all elements with the same key will be sent in the same parallel task.

```
stream.keyBy(...).window(...)
```

Non-Keyed Windows
In case method keyby is not used, the stream is not keyed, consequently, all elements are assigned to the same task. Therefore, the parallelism will be 1.

```
stream.windowAll(...)
```

2.4.1 Window Assigners

WindowAssigners are responsible for assigning each incoming element to one or more windows. Flink provides four predefined WindowAssigner types, whose operations

are described below. Additionally, WindowAssigners' extensions for differentiation or construction new types of windows, are enabled.

- **Tumbling Windows:** Tumbling Windows: A "tumbling windows assigner" assigns elements arriving at windows with size "window size". The size is constant and windows do not overlap.
- **Sliding Windows:** The "sliding windows assigner" assigns elements to windows of fixed length. Apart from window size, it receives the parameter "window slide" that defines the frequency of the window starting. In case "windows size" is greater than "window slide", windows are overlapped. Therefore, an element can belong to multiple windows.
- **Session Windows:** The "session windows assigner" groups elements by sessions of activity. These windows do not have constant size and be terminated when new elements do not arrive for some limit (session gap) e.g. a time period.
- **Global Windows:** A "global windows assigner" assigns all elements with the same key to the same single global window. Unless a trigger is not specified, global windows will constantly receive elements without executing calculations.

Chapter 3
Streaming Algorithms

In computer science, streaming algorithms [2] are used for processing data streams in which the input is presented as a sequence of items and can be examined in only a few passes (typically just one). In this kind of model, these algorithms have access to limited memory (generally logarithmic in the size of and/or the maximum value in the stream). They may also have limited processing time per item. Restraints can be satisfied through algorithms approximate answers to data stream queries based on a summary or "sketch".

3.1 Frequent Itemsets

In the data stream model, the frequent elements problem is to output a set of elements that constitute more than some fixed fraction of the stream. More formally, fix some positive constant c>1, let the length of the stream be m, and let f_i denote the frequency of value i in the stream. The frequent elements problem is to output the set $\{i|f_i > m/c\}$.

CCFCSketch [3]
A different version of Count Sketches from Charikar, Chen, Farach-Colton. The data structure that CCFCSketch uses is named account sketch. Achieves better space-bound than Count sketches.

WindowElement
This algorithm stores the number of elements of a set.

Majority
In this implementation, the algorithm receives boolean input values and counts their frequency.

SimpleTopKCounting
A simple implementation of a stream-counting model. In this model, it is given as an argument a threshold value k. This represents the maximum number of items that may be tracked/monitored by the model.

Misra−Gries [4]
The Misra-Gries summary is an additional algorithm for the solution of the objects' frequency problem. In this algorithm, parameter k determines the size of the array. In case k has value 2, this problem is encountered as a majority problem. This algorithm uses $O(k(log(m) + log(n)))$ space, where m are distinct values and n the length of the stream.

LossyCounting [5]
This algorithm maintains a data structure, to observe objects, frequency and ε. The algorithm gets as input the allowable error. Imported data are split into windows with dimension $w = \lceil \frac{1}{\varepsilon} \rceil$. For every element, there is a counter which is increased when the element appears again. At the end of each window, there is a compression for the internal data structure so that the less appearing elements are deleted.

StickySampling [6]
StickySampling shows similarities with LossyCounting. Unlike Lossy Counting, the size of the window is designated as follows: the first window has magnitude w=t (where $t = \frac{1}{\varepsilon} \cdot \log(\frac{1}{s\delta})$ with ε, δ, s parameter) and every next one will have a magnitude two times larger than the last one. Also, the number of elements is dependent

on sampling probability.

AMSSketch [7]
Suppose there is a zero vector v. While new elements arrive to the data stream, vector v is modified by a weight w (positive or negative) in index i as $v_i \leftarrow v_i + w$. AMSSketch can be considered as a two-dimensional array n (Buckets) $\times$ m (Depth) with $n = O(1/\varepsilon^2)$ where ε is the allowable error and $m = O(log\frac{1}{\delta})$ where $1 - \delta$ is probabilistic confidence. In each row n, a hash function h_n is set. This hash function maps the input domain U uniformly to $\{1, 2, \ldots, m\}$. A second hash function called g_n, maps elements from U uniformly onto $\{-1, +1\}$. For the analysis to hold, we require that g_n is fourwise independent. All positions of the sketch are initialized to zero. For the updating functionalities in index i with weight w, the element goes through the hash function in line j and updates the corresponding counter with the value $w \cdot gj$. For computation estimateF2 taking the sum of the squares of row of the sketch in turn, and finds the median of these sums.

CountMinSketch [8]
CountMinSketch's structure looks like a two-dimension array with w columns and d rows. Width (w) and depth (d) are formed during the initialization of the sketch depending on restraints (time, space, accuracy). For every line d, there is a hash function that differs from the other series. Every time an element enters in the data structure, these hash functions update the counter in line. This counter is initialized to zero. The frequency of an object can be obtained by keeping the lowest counter returning from the hash functions.

3.1.1 Top-K

TopK algorithms save high-frequency elements and find application to problems such as Internet advertising, twitter logs, web mining, stock tickers, etc. In this class of problems, a solution cannot always be found with memory restriction. The reason behind this, is the large number of different items that can be found in a data stream.

This memory insufficiency have impact in finding elements with maximum frequency.

StreamSummary [9]
The Stream-Summary data structure is utilized by the Space-Saving algorithm to guarantee strict error bounds for approximate counts of elements, using very limited space.

Frequent
The Frequent algorithm utilizes a data structure whose dimension is determined by the error received as an argument.

StochasticTopper
Estimates most frequently occurring items in a data stream using a bounded amount of memory.

3.2 Quantiles

Quantiles are cut points dividing the range of a probability distribution into continuous intervals with equal probabilities or dividing the observations in a sample in the same way. Median (or $2-$quantile) is a number located right in the middle, so that 50% of ordered numbers are over the median and the other 50% is below.

Frugal2U [10]
Algorithm Frugal-2U is an extension of algorithm Frugal-1U. Frugal-1U computes ϕ-quantiles without considering previous elements. Frugal-2U receives two more arguments *step* and *sing* for estimation improvement. Step is a variable, which is increased when the current stream element is on the same side of the current estimate. On a contrary, *step* is reduced. Sing represents a bit, which indicates the increment (or decrement) of the estimation.

Quantiles Sketch (from Apache DataSketch [11])
The quantiles algorithm is an implementation of the Low Discrepancy Mergeable Quantiles Sketch. Receives parameter k, which affects the accuracy of the results, as well as the size of sketch. Accuracy of this quantile sketch is always with respect to the normalized rank (e.g. for $k = 256$ to normalized rank error is <1%).

MPQuantiles [12]
Algorithm MPQuantiles (Munro-Paterson) can handle calculations about ϕ-quantiles of a sequence of N data elements in p passes, will need $O(N^{1/p})$ space.

3.3 Cardinality

The count−distinct problem (also known in applied mathematics as the cardinality estimation problem) is the problem of finding the number of distinct elements in a data stream with repeated elements. The elements of a data collection or a stream can contain IP addresses of packets passing through a router, unique visitors to a web site, motifs in a DNA sequence, elements in a large database, or elements of RFID/sensor networks. For instance, assume there is a stream: a, b, a, c, d, b, d. Cardinality estimation requires the number of different elements in the stream. As a result, there are the unique elements $n = |\{a, b, c, d\}| = 4$ in the stream.

LogLog [17]
The algorithm uses a hashing function to bring data in a binary and uniformly randomized form. Then, algorithm calculates the maximum number of consecutive zeros e.g. 4 consecutive zeros (probability of zero 0.5) then, the cardinality estimate is $2^k = 2^4 = 16$. The n bits from the hash function for the representation of the buckets are bound, to improve the k as well as the estimations. These buckets retaine the number of maximum zeros. Final result k is computed by the average number of buckets. The returned estimate is $m \cdot a_m \cdot 2^k$ where a is a_m correction factor.

AdaptiveCounting [16]
Uses the same data structure as LogLog counting, i.e. an array of m counters. Let b, the ratio of empty buckets and b_s the ratio of empty buckets when $t = t_s$ ($b_s = b_e/m = e^{-t_s} = 0.0051$). The cardinality estimate is given by the formula:

$$\hat{n} = \begin{cases} m \cdot a_m \cdot 2^{\frac{1}{m}\sum_{j=1}^{m} M^{(j)}} & \text{if } 0 \leq \beta < 0.0051 \\ -m \cdot \ln(\beta) & \text{if } 0.0051 \leq \beta \leq 1 \end{cases}$$

HyperLogLog [15]
An improvement of the LogLog algorithm. The general model resembles with LogLog, differentiated with two modifications and offers much better estimations. First, harmonic mean E calculation is made with values of the buckets. Second, there are changes in conditions: i) Empty posts in buckets, then the estimation $Es = -m \cdot log(V/m)$ where V is the number of buckets with zero numbers ii) Multiple conflicts in the buckets, for 32-bit registers then $Es = -2^{32} \cdot log(1 - E/2^{32})$. Under normal circumstances, the estimate is $Es = m \cdot \alpha_m \cdot E$.

KMinValue [14]
Method k minimum values (KMV) solves the cardinality problem, using hash function h which converts the elements in a sequence with range [0,1]. Limit k (given from user) specifies how many values will be saved in hash space. In this particular hash space, are retained the k lowest hash values. Eventually, the estimate of the population is given by the formula $\widehat{|X|} = \frac{k-1}{U_{(k)}}$ where $U_{(k)}$ is k−th smallest hash value.

LinearCounting [13]
This method faces the cardinality problem in two simple steps. It creates a bitmap with size m (given by the user) and initializes all positions to zero. Every element that is arrived, is converted to a position in bitmap through the hash function. In this position, the value of the bit map is changed from 0 to 1. After that, zeros in the bitmap are counted. The final estimation is given by the formula $n = -m \cdot log(Vn)$ where $Vn =$ (sum of zeros)$/m$.

CountThenEstimate
This algorithm avoids allocating a large block of memory for cardinality estimation until a specified "tipping point" cardinality is reached.

3.4 Average

For a data set, the arithmetic mean, also called the expected value or average, is the central value of a discrete set of number.

MovingAverage
A simple method for computing the expected value of a stream of numbers by only averaging the last P numbers from the stream, where P is known as the period.

ExponentialMovingAverage (EMA)
Also known as an exponentially weighted moving average (EWMA), is an algorithm that can calculate average, giving greater importance in the recent elements of a stream, without ignoring completely the older ones.

SimpleEWMA
Represents the exponentially weighted moving average of a series of numbers. It has no warm-up period and it uses a constant decay. These properties let it use less memory.

VariableEWMA
Represents the exponentially weighted moving average of a series of numbers. Unlike SimpleEWMA, it supports a custom age, and thus uses more memory.

AverageElement
A simple algorithm that keeps sum and count of set's elements.

3.5 Membership

A Bloom filter is a space-efficient probabilistic data structure, conceived by Burton

Howard Bloom in 1970, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not – in other words, a query returns either "possibly in set" or "definitely not in set". Elements can be added to the set, but not removed the more items added, the larger the probability of false positives.

BloomFilter [18]
The base data structure of a Bloom Filter is a bit vector. The values of bit vector are initialized to zero. Furthermore, there are different hash functions which correspond a stream element in a bit vector's position. The number of hash functions affects throughput (the more the slower) and accuracy (the lower the more false positives) of the result. To inspect if an element does not contain in a set, one bit in the position that hash function exported must be zero. Otherwise, the element maybe exist.

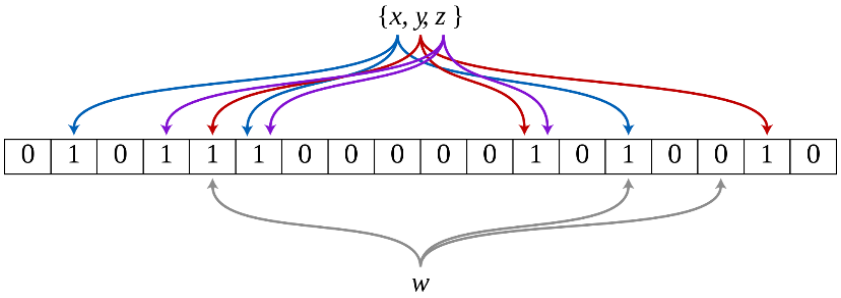


Figure 3.1: Bloom Filter testing if the element 'w' is member of the set (image from [19])

In Figure 3.1, a Bloom Filter is illustrated with three elements 'x', 'y', and 'z'. Bit vector has 18 positions that can be filled from 3 hash functions. The arrows of each element, show the position over bit vector. Element w is not contained to set since it hashes to a bit position containing 0.

CountingBloomFilter [20]
A Counting Bloom filter is a generalized data structure of the Bloom filter. It contains a "bit vector" similar to the Bloom filter. Every position of bit vector includes

counters (as shown to Figure 3.2.b) instead of 0 or 1. These counters are increasing or decreasing when updates or deletions are made in the algorithm respectively.

DynamicBloomFilter
The basic idea is to express a dynamic set with a dynamic s × m bit matrix that consists of s standard bloom filters.

VarCountingBloomFilter [21]
A variation of the algorithm CountingBloomFilter, where at each element insertion, the hashed counters are incremented by a hashed variable increment instead of a unit increment.

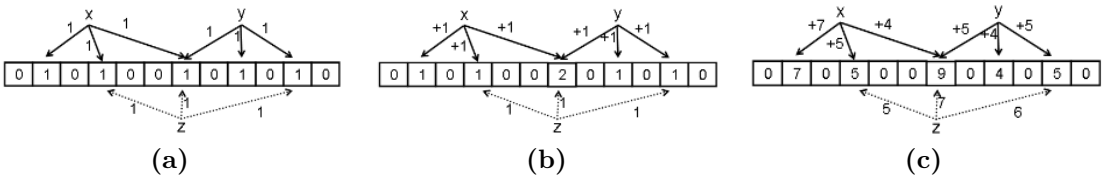


Figure 3.2: (a) BF (b) CBF (c) V-CBF (image from [21])

In Figure 3.3, filter comparisons are depicted: BF, CBF, and V-CBF to query of element $z \in S$ where $S = \{x, y\}$.

3.6 Sampling

Stream sampling is the process of collecting a representative sample of the elements of a data stream. Usually, the elements of the sample are less than the elements of the entire stream, but can have important data stream features. Unlike sampling from a stored data set, stream sampling collects elements on arrival. Every element that is not included in the sample, it vanishes without the ability of recovery.

ReservoirSampler
Reservoir sampling is a family of randomized algorithms for choosing a simple random sample without replacement. Besides sample's size, it is set a skip factor for new elements when the list is full.

BernoulliSampler
Bernoulli sampling is an equal probability, without replacement sampling design. In this method, for each sample, roll the dice and pass/fail.

WeightedRandomSampler
In weighted random sampling (WRS) the items are weighted and the probability of each item to be selected is determined by its relative weight.

SystematicSampler
Systematic sampling chooses population unit with a fixed period. The longer the algorithm period, the more elements remain to sample.

WRSampler (With Replacement)
In this sampling algorithm, elements that belong to a sample in a list, are stored. When the list is full, a decision is taken for every new element about replacement with an old element.

SpaceSavingSampler
SpaceSavingSampler takes a sample and retains every element arrived, to a list (increase their frequency if it is already found). When the list is full, new elements will take place of the old ones with the least frequency to the list.

Chapter 4
Implementation

In this chapter, we are gonna discuss the changes that we had to make on Apache Flink, to support the new-functionalities on windows. Initially, we had to study in-depth Flink's functionalities for streaming data as well as the management of its very own windows. We start to analyze the flow of the program, after the insertion of our elements in the windows. Our first job after that, was to understand the functionality of functions sum, min, max that Flink provides on windows. We must also comprehend the functionality of some Flink's functions, which are necessary for the processing of elements on windows. Those functions are ProcessWindowFunction, ReduceFunction, AggregateFunction, FoldFunction, which are provided by Flink and their settings are determined by the user. Moreover, we must recognize how different types of assignment on windows can reflect on those functions mentioned above. Upon those protocols, we structured our very own classes on Flink to support the extra-functionalities on windows.

4.1 flink-streaming-java

In regards to the Flink sector, we have used folder flink-streaming-java to access the streaming data. Our purpose is the library WindowSynopsisLibrary to be embodied on the Flink-Windows. To achieve this, two classes have been created (WindowSynopsisAggregator and WindowSynopsisFunction will be referred on Chapter 4.3) on

folder aggregation. Also, some adjustments had to be made on classes WindowStream, AllWindowedStream in order to have them available for the call through the main function, so that the user can have access in those classes.

Searching through Flink's source code, we had to find the point in which the insertion of elements on our windows was made, because the algorithms in library WindowSynopsisLibrary are updated upon a new insertion of an element in our windows. This way we ensure the user has direct access to the result upon window's shutdown.

4.2 WindowSynopsisLibrary

The body of implementation begins with the creation of interface WindowSynopsis, which includes methods such as:

- *initialize*: This function initializes the algorithm given by the user. Without initialization, none of the following actions could be made.
- *add*: This function is called by the class created from the user and updates the algorithm with the value given as a parameter.
- *merge*: This function inherits and unite (wherever that is supported) two objects of the same class.
- *result*: This function returns an object of the class that is selected for the computation the algorithm supports.

Moreover, the interface is implemented from one of the following abstract classes (Frequent, TopK, Quantiles, Cardinalty, Average, Membership, Sampling) as shown in Figure 4.1. Each abstract class implements methods *add*, *merge*, *result* and contains an abstract *initialize* method. This method is defined on sub-classes that extends their abstract classes (such as LossyCounting extend Frequent). In Figures 4.2-4.7 below, we display the hierarchies of our abstract classes.

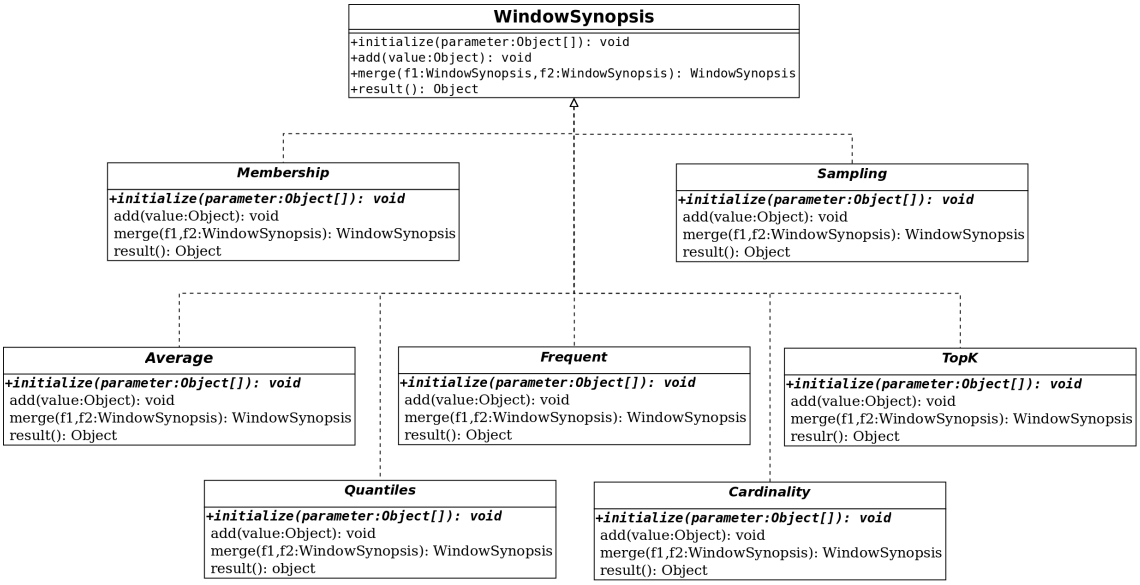


Figure 4.1: Class diagram for WindowSynopsis

4.3 Enter data in windows

When an element arrives, it gets assigned a key using a KeySelector and it gets assigned to zero or more windows using a WindowAssigner. The set of elements with the same key and window is called a pane. Based on this, the element is put into panes. An element can be in multiple panes if it was assigned to multiple windows by the WindowAssigner.

Each pane gets its own instance of the provided Trigger. This trigger determines when the contents of the pane should be processed to emit results. When a trigger fires, the given InternalWindowFunction is invoked to produce the results that are emitted for the pane to which the Trigger belongs.

4.4 Flink and WindowSynopsisLibrary

For the integration of WindowSynopsisLibrary to Flink, two classes have been created:

WindowSynopsisAggregator: Initially, we define the constructor of our classes for the necessary initializations and actions to be made, such as controlling the position and type of data the algorithm gets. In this class, we also define the function *reduce*. This function with arguments the values (value1 and value2) is called each time a new element is inserted on our window. It must be mentioned, that in the body of *reduce* function we call the function that updates the class of WindowSynopsisFunction.

WindowSynopsisFunction: In this abstract class, we ensure (through method *getForClass*), that when called (*getForClass*) the type of data we have on the field of computation are an object, this is required for us to be able to return our results because in any other case they are rejected. Moreover, in this class we define the abstract method *update* of class ObjectWindowSynopsis. This class (ObjectWindowSynopsis) is initialized from method *getForClass* and its object is returned to WindowSynopsisAggregator. This transpires as we want to call it from function *reduce*. Function *reduce* calls method *update* for each new element. As for method *update*, it is an essential part of this procedure; it's the connection line with library WindowSynopsisLibrary. In this method, we also have the following functionalities:

- Initialization: When the first two values arrive on method *update*, initialization follows the algorithm that was selected by user according to table parameter. After that, we update the algorithm (through *add*) with values: value1, value2. The following steps are to return the result (the object of our algorithm) and save value1 until the window's shutdown.
- Add: Every next time *update* method is being called, an object of the algorithm class, that user selected, is being saved to argument value1. As a result, we update the algorithm with value2 and return back value1.
- Merge: In case the value of argument value2 is an object of a subclass of WindowSynopsis, then it's required to unite the two objects. Hence, the function *merge* is called.

```
static class ObjectWindowSynopsis extends WindowSynopsisFunction {
    private static final long serialVersionUID = 1L;

    @Override
    public Object update(Object value1, Object value2,WindowSynopsis wsn,Object[]
     parameter)  {
      if (value2 instanceof WindowSynopsis) {       //merge valu1,value2
        value1 = ((WindowSynopsis) value1).merge(((WindowSynopsis) value1),((
    WindowSynopsis) value2));
        return value1;
      }else if (value1 instanceof WindowSynopsis){  //add value2
        ((WindowSynopsis) value1).add(value2);
        return value1;
      }else {                                        //init algorithm
        Class<? extends WindowSynopsis> clazz = wsn.getClass();
        try {
          wsn = clazz.getDeclaredConstructor().newInstance();
          wsn.initialize(parameter);
          wsn.add(value1);
          wsn.add(value2); }
        catch (InstantiationException | IllegalAccessException
          | NoSuchMethodException | InvocationTargetException e) { e.
    printStackTrace(); }
        return wsn;
      }
    }
  }
```

These classes (WindowSynopsisFunction, WindowSynopsisAggregator) must appear on class DataStream through classes AllWindowedStream, WindowedStream. So, they can be called and after initialized. One of this two classes according to the type of window, will call WindowSynopsisAggregator.

AllWindowedStream class is about non-keyed streams and required addition of a method *sa* to have access in class WindowSynopsisAggregator. Method *sa* has as arguments: the position, the name of field for computation, the object of subclass WindowSynopsis, the table with parameters based on which we will have our initial-

izations, the function ProcessAllWindowFunction (which is defined by the user for all initializations to be possible by the algorithm).

WindowedStream class is about keyed streams. Our changes in this class are similar to the ones in class AllWindowedStream with the difference that method *sa* has as its last parameter function ProcessWindowFunction (not ProcessAllWindow-Function), which concerns keyed streams.

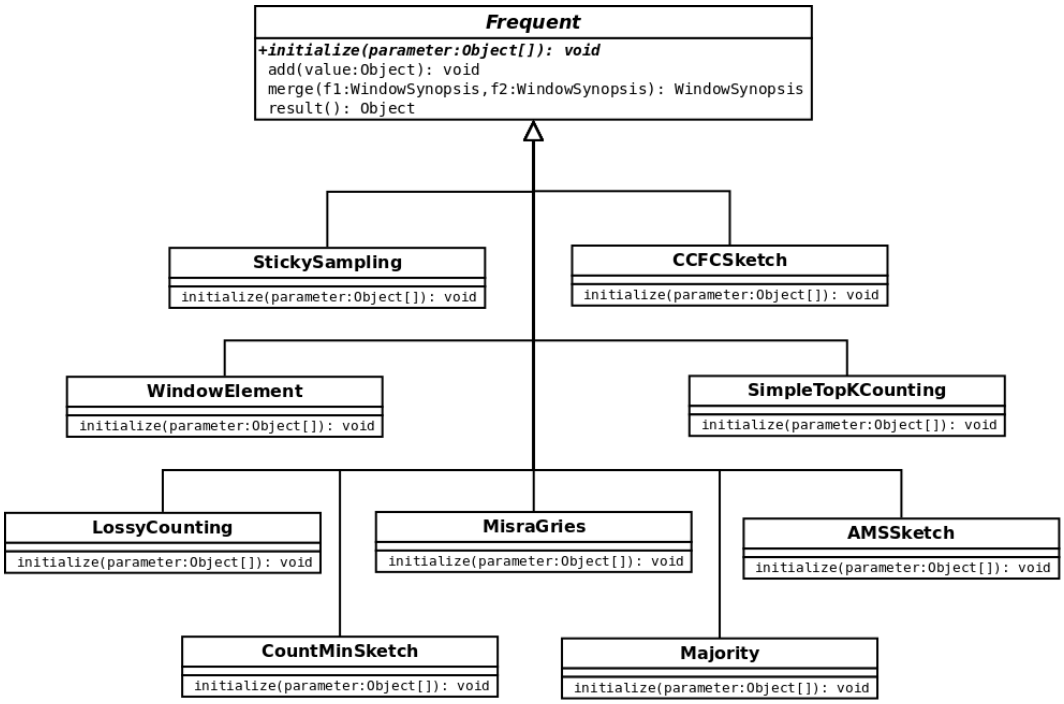


Figure 4.2: Frequent class diagram

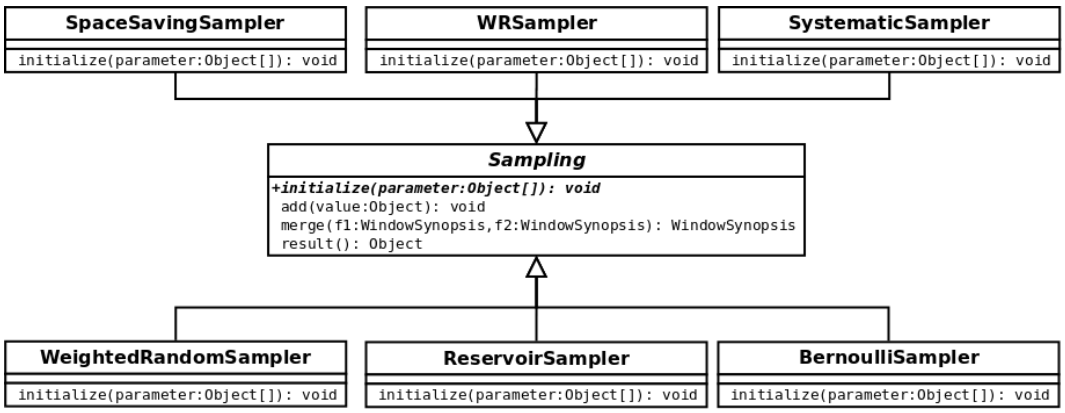


Figure 4.3: Sampling class diagram

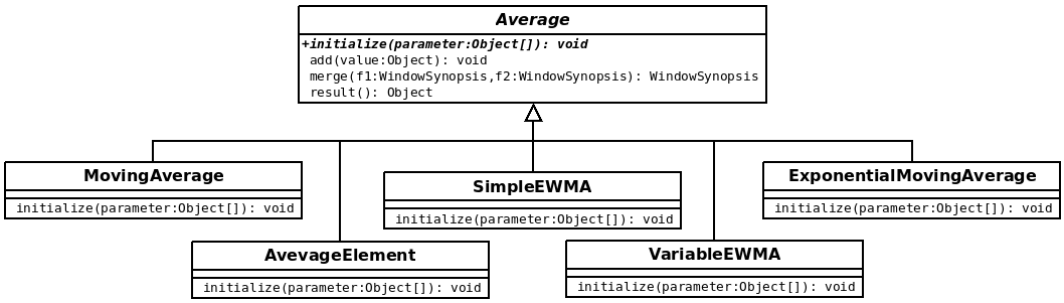


Figure 4.4: Average class diagram

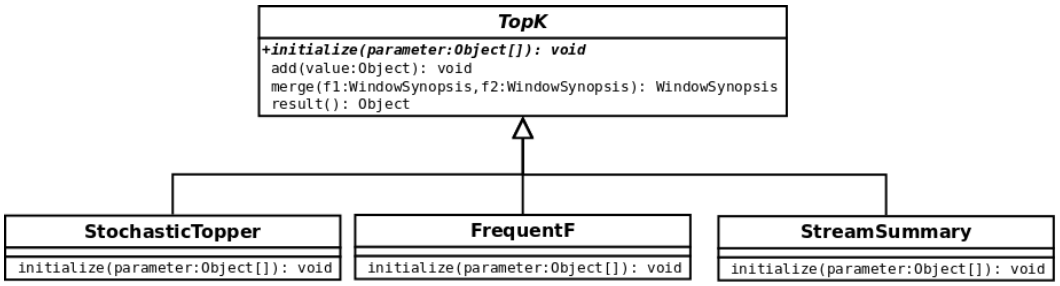


Figure 4.5: TopK class diagram

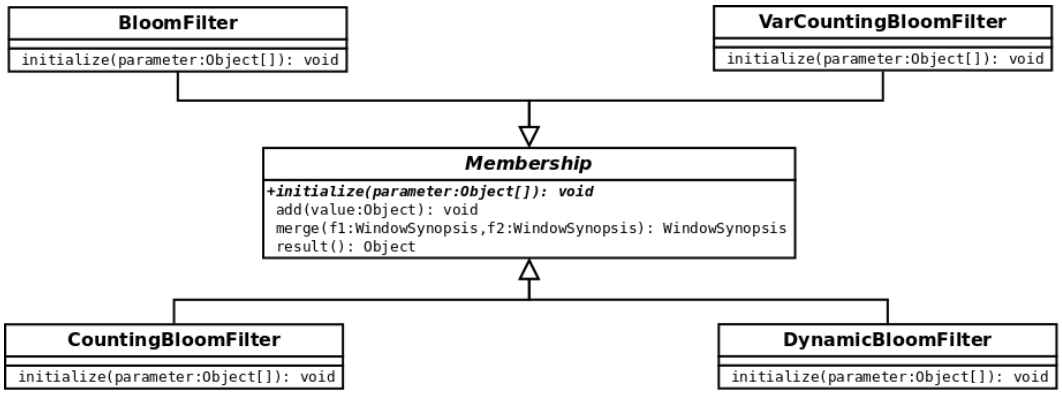


Figure 4.6: Membership class diagram

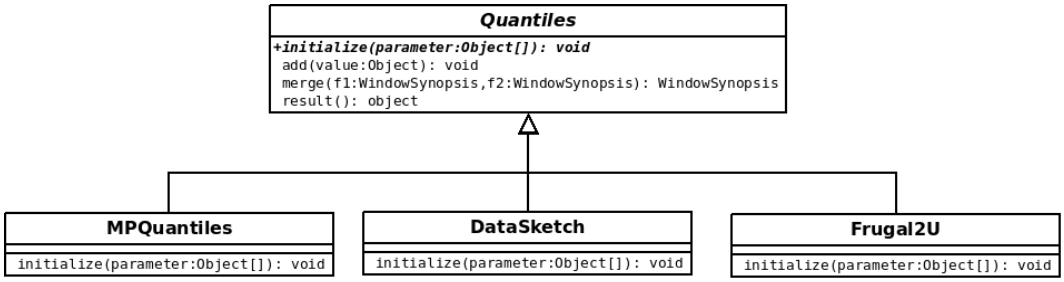


Figure 4.7: Quantiles class diagram

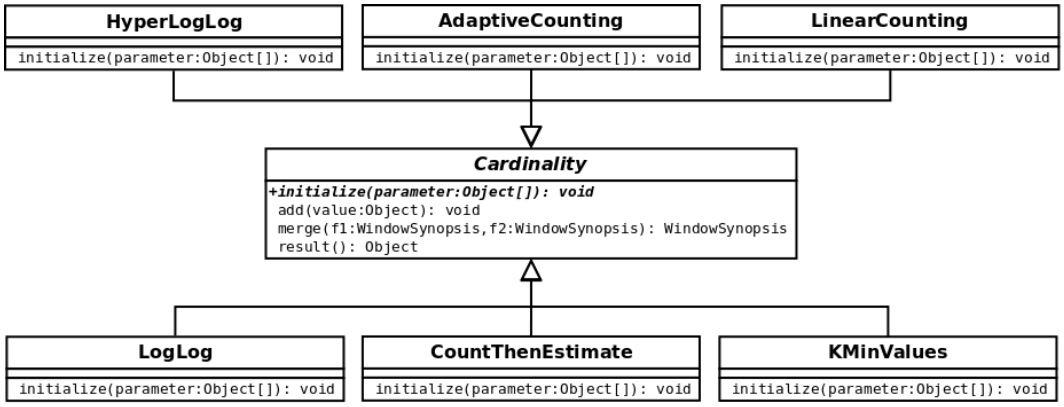


Figure 4.8: Cardinality class diagram

Chapter 5

Experimental Evaluation

We conducted several experiments locally and remotely using the multi-node cluster of our university.

In order to run our experiments to the multi-node cluster of our university, we deployed Flink by using the Standalone Cluster setup. This setup includes a single Job Manager (master node) and at least one Task Manager (worker nodes). In our setup, we used 12 Task Managers with maximum number of parallel task slots 36 (i.e. 36 physical cores). During our experiments, the maximum Job parallelism that we used was 16, so we let Flink's runtime to make the choice of the specific task slots. The table below, presents the system specifications of the Job and Task managers.

Node	CPU	Cores	Ram
1 Job Manager	Intel Xeon E5-2430 v2	6	32
12 Task Managers	Intel Xeon X3323	4	8

In the following diagrams, we display the results of each algorithm category. The arguments for each display are upon each diagram. The main source for data was random numbers (that we were sending on Kafka's topic), the range of those numbers was decided through the following logic:

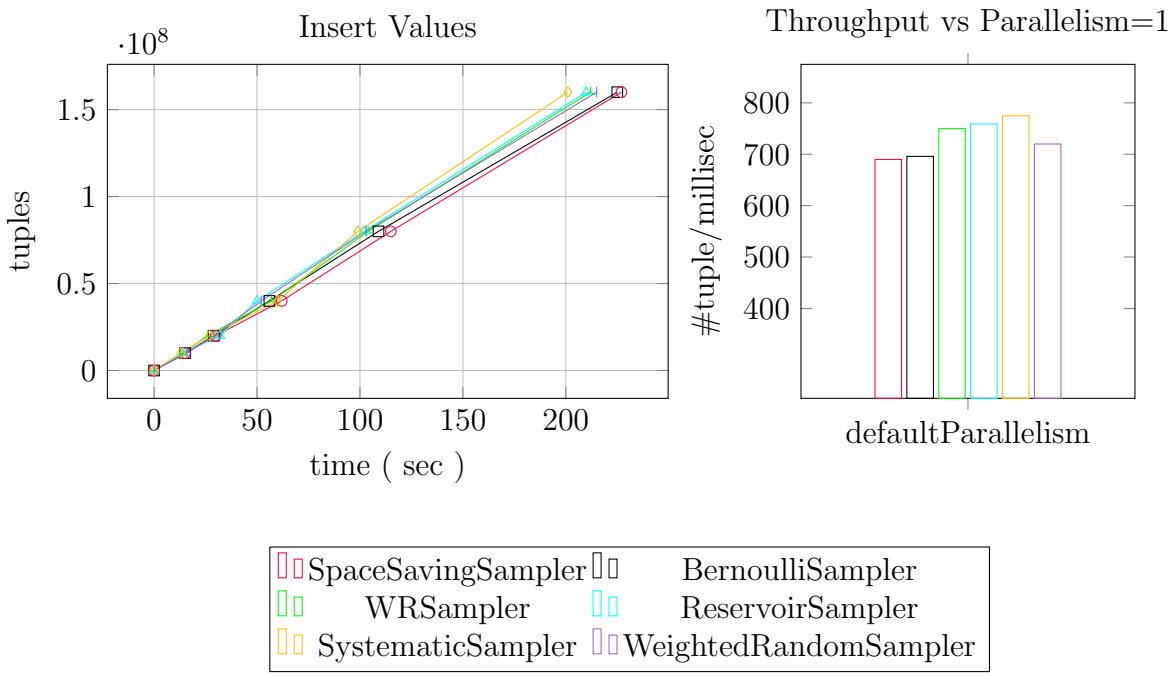
- Average , Quantiles , Cardinality , Sampling : range $0 - 10.000.000$
- Membership range : $0-100.000$
- TopK , Frequent range : $0-1000$ (key)

Regarding the diagram for "Insert Value", we recognize a number of tuples and their time frame (in seconds) that are ready for insertion, as for the diagram for "Throughput vs Parallelism=1" it displays the average number of tuples that is processed per millisecond by the algorithm. In "Throughput vs Parallelism" diagrams, scalling per level parallelism is presented, only for algorithms that can be executed in parallel. What is to be expected considering these two, is that doubling the parallelism (wherever that is supported) will actually double our throughput as well.

5.1 Sampling

The experiments of sampling collect elements from a large dataset. The criteria of collection of those elements are in user's ease for every algorithm. Once sampling is done, a list of elements, that were collected, will be returned.

- ReservoirSampler : size = 1000 , skipFunction = R(10)
- SystematicSampler : period = 750L
- WRSampler : sampleSize = 1000
- WeightedRandomSampler :weight=400, generator=JDKRandomGenerator()
- SpaceSavingSampler : k = 100
- BernoulliSampler : percent = 0.005



The throughput of our results from the sampling category is good enough as expected. As a new element arrives, our algorithm must decide if that element will be included in our sample. As a consequence, insertions don't require huge cost of processing.

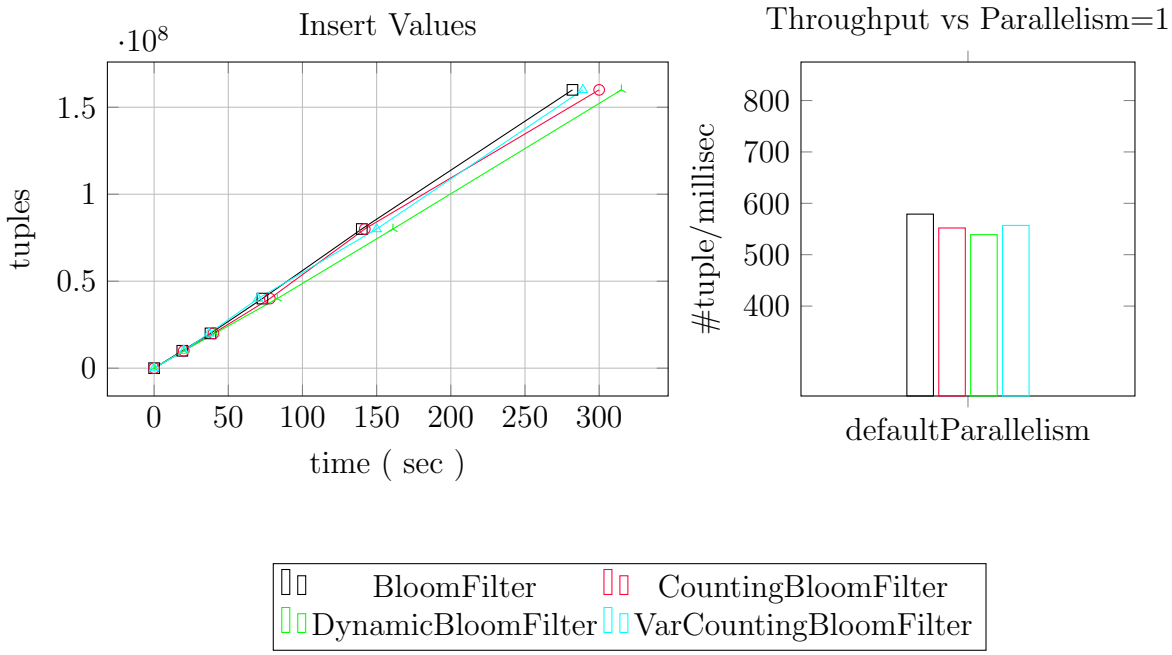
Algorithm SpaceSavingSampler processes fewer tuples per millisecond. That's the result of its process: each time an element arrives, which is not included in our list, the element with the smallest frequency will be replaced. So as we increase the size, efficiency decreases. Regarding other algorithms by increasing frequency of sampling, we increase the size of list with elements.

5.2 Membership

The experiments for Membership can answer rapidly if an element exists in a large dataset. If the answer is false, the element will surely not exist in the total. Whereas

if the answer is true, it may exist. The larger the size of our structure the better our answers will be.

- BloomFilter : vectorSize = 1000000 , nbHash = 10 ,
  hashType = Hash.MURMUR_ HASH
- CountingBloomFilter : vectorSize=1000000 ,nbHash= 10,
  hashType = Hash.MURMUR_ HASH
- DynamicBloomFilter : vectorSize=1000000,nbHash = 10,
  hashType = Hash.MURMUR_HASH, nr = 1000
- VarCountingBloomFilter : numElements = 1000000 ,bucketsPerElement = 10,
  exp = 4

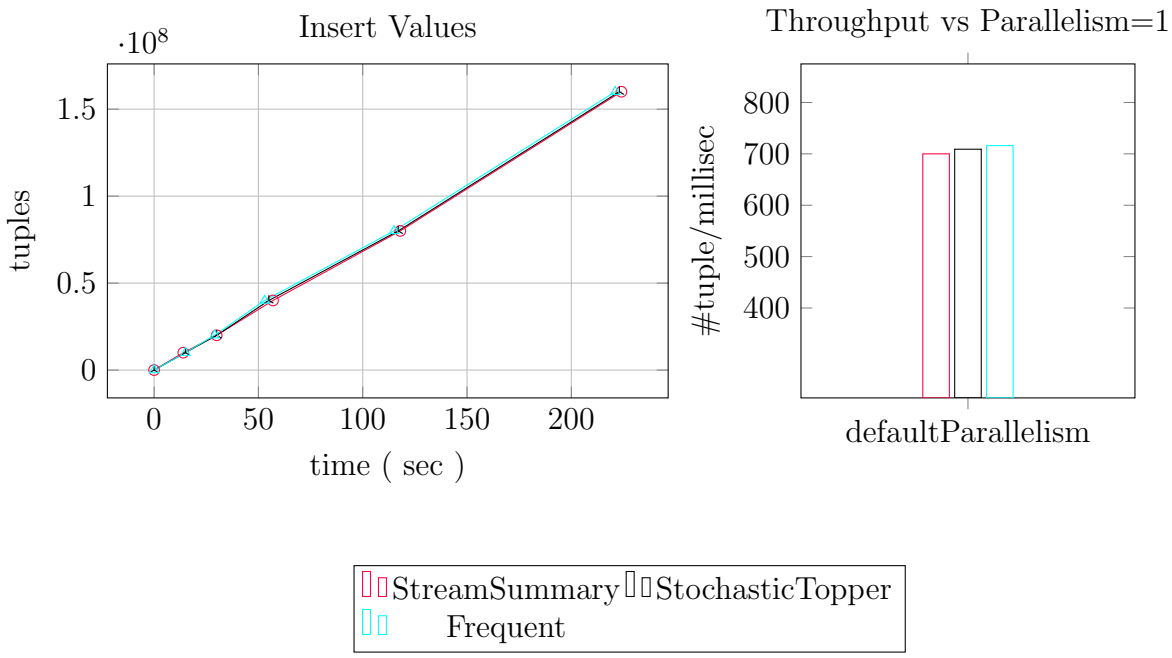


The Membership had a lower throughput in comparison to the rest of our categories. That's a result of a decrement in the speed of update of our algorithms comparing with the other categories. Whereas, their deviation is quite small.

5.3 TopK

The TopK experiments find the objects, which appear to have the highest frequency inside a total.

- StreamSummary : capacity = 10000
- Frequent : error = 0.00001
- StochasticTopper : sampleSize = 10000

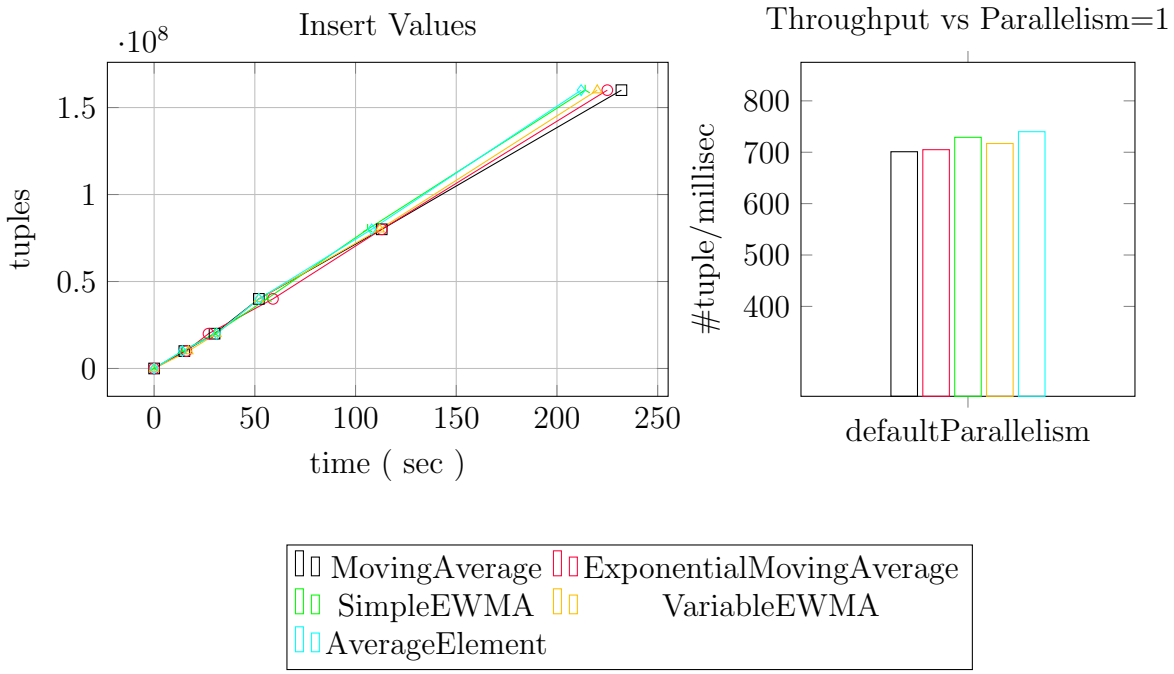


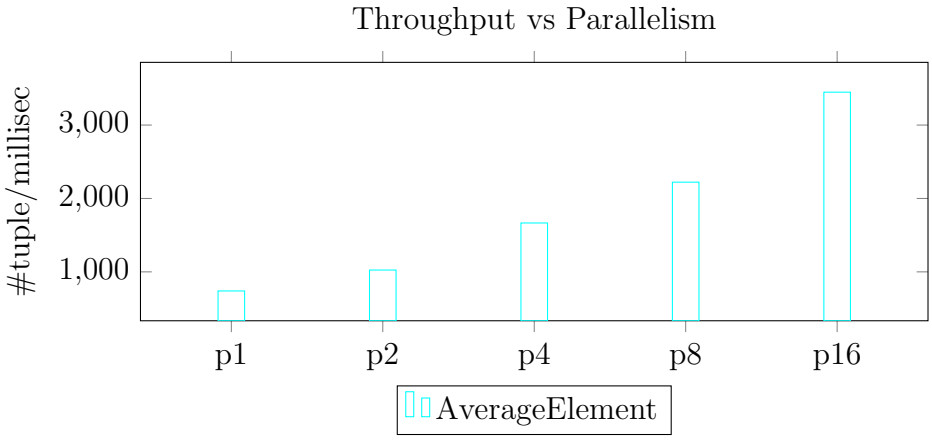
In category TopK diagrams, we recognize that the throughput between algorithms have a small deviation. Moreover, the increment in size of our structure decreases our throughput.

5.4 Average

In the experiments about Average, we sum up the elements of our dataset and hold the total of our elements. These two numbers differ in every algorithm. The return value is the Average.

- MovingAverage : period = 15000
- ExponentialMovingAverage : window = 15000
- SimpleEWMA : –
- VariableEWMA : age = 0.0001
- AverageElement : –





In category Average, the throughput of results was good enough and had small deviations from algorithm to algorithm. That's a consequence of no delays in the update procedure (complexity O(1)). The scaling of throughput, based on the parallelism, is not ideal but yet good enough. This is due to the fact of the merge of parallel windows' termination so that the result can be exported. This has some cost.

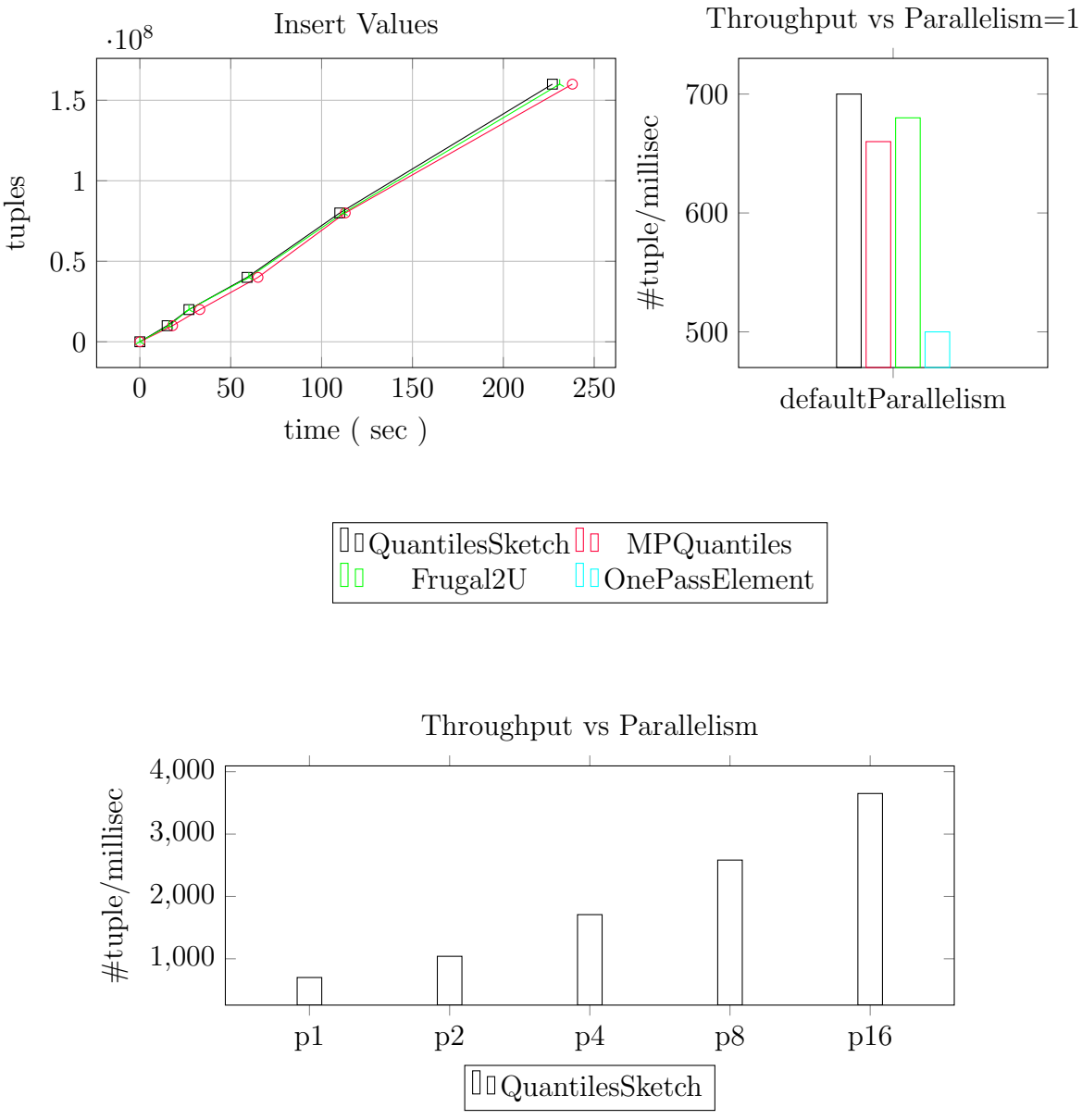
5.5 Quantiles

From our experiments in Quantiles, we used the ability of median. We have the following problem: from a large dataset find the median, this problem in a simple solution (sort and take back length/2) could have huge complexity as of O(nlogn). We compared our experiments with an algorithm (Median of medians), which needs only one passage of our elements (complexity O(n)).

The algorithm OnePassElement (Median of medians) requires one passage of the elements of the window upon its shutdown to be able to compute the median value. It's clear that, this way of commutation of the median value is significantly slower to the computation by our Synopsis.

- Frugal2U : quantiles = $double[]\{0.01, 0.25, 0.5, 0.75, 0.95\}$
  , initialEstimate = 0

- Apache QuantilesSketch : k = 128 (default)
- MPQuantiles : numQuantiles = 10



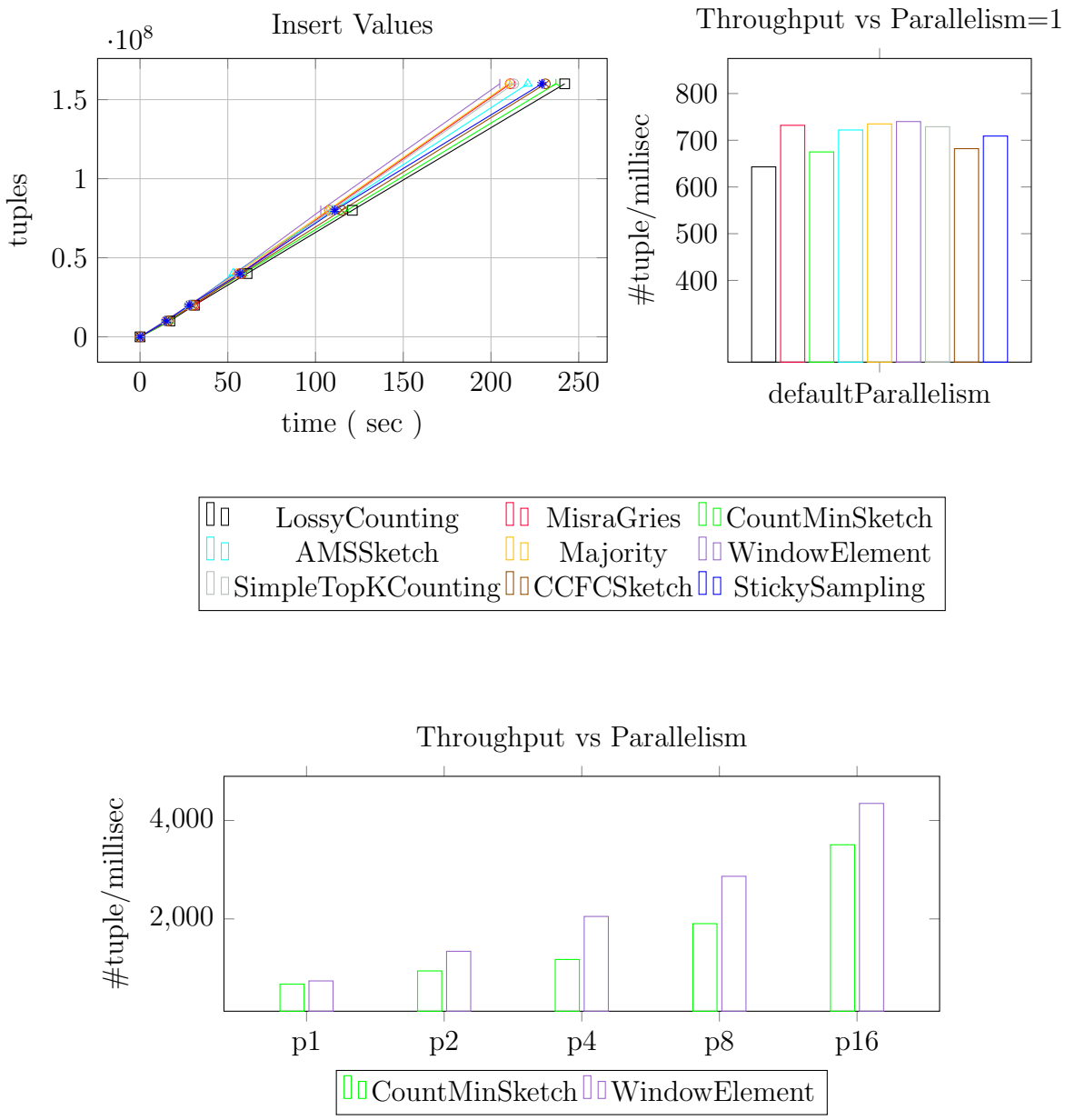
In Synopsis Quantiles, the throughput of results is better than the algorithm OnePas-

sElement as we expected. That's a result of the fact that, Synopsis only need one passage of our elements (plus the cost of update). On the otherhand, algorithm OnePassElement requires two passages (save the windows and process on shutdown). The scaling in the diagram above throughput vs parallelism has good enough results.

5.6 Frequent

In this category of experiments, we had to find the frequency of elements of a data collection. A special category is WindowElement (count), which counts the elements of our window. Majority is another one, which counts a value (true or false in our experiments) as a majority.

- CountMinSketch : epsOfTotalCount = 0.0001 , confidence = 0.99, seed = 7364181
- LossyCounting : maxError = 0.00001
- MisraGries : k = 10.000
- StickySampling : support = 0.001 ,error = 0.09 , probabilityOfFailure = 0.0001
- SimpleTopKCounting : k = 10.000
- AMSSketch : depth = 5, buckets = 512
- CCFCSketch : buckets = 512, tests =5 , lgn = 4 ,gran = 1
- Majority : −
- WindowElement (count) : −



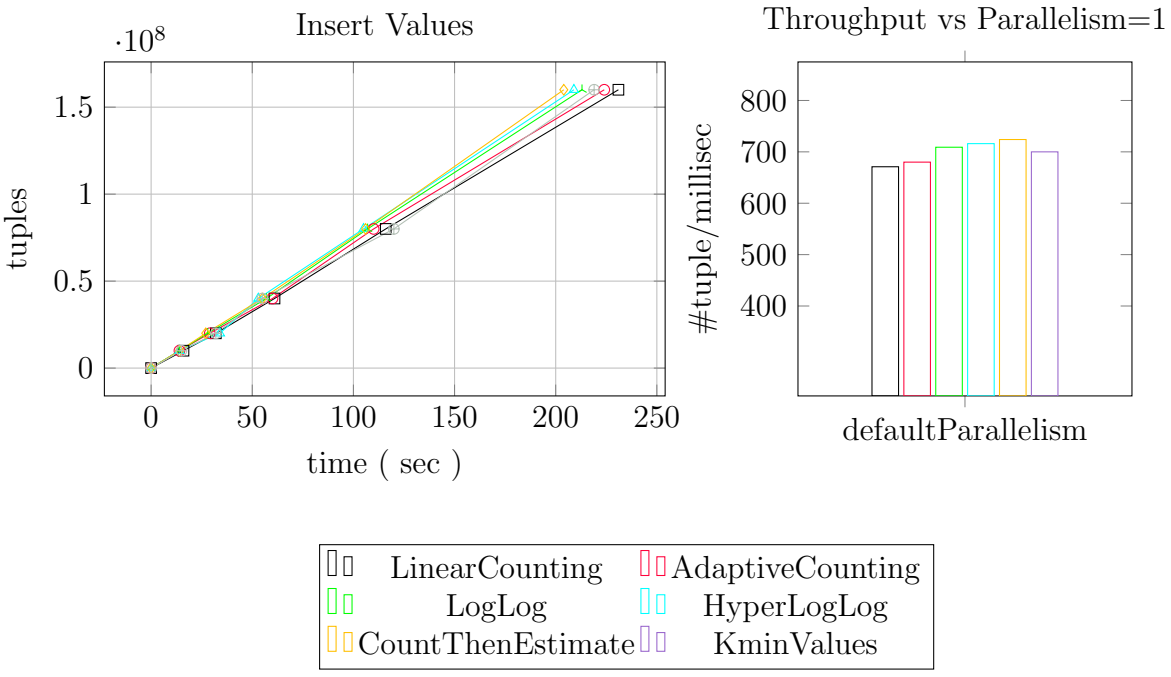
Regarding the algorithms of frequency of objects, the throughput of most algorithms is quite close to the WindowElement. That shows us how algorithms operate at rapid pace. In the diagram throughput vs parallelism, it's a clear difference in scal-

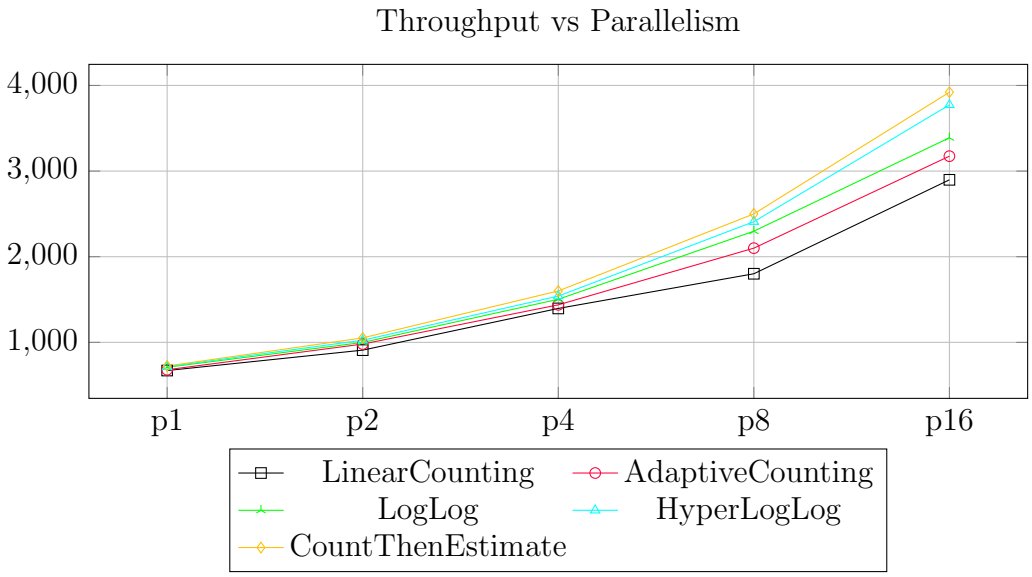
ing between CountMinSketch compared to WindowElement. The difference which is presented in throughput, is caused from the highest cost (renewal and merge of windows) of CountMinSketch.

5.7 Cardinality

In these types of experiments, we must recognize through a large dataset the number of different elements existing.

- AdaptiveCounting : k = 10
- LogLog : k = 10
- HyperLogLog : log2m = 10
- LinearCounting : size = 10000000
- CountThenEstimate : tippingPoint = 1000, builder = 1000000000
- KMinValues : k = 100000





In the category of cardinality, the throughput has very little deviations between the algorithms. The insertions are quite fast, as the speed of update in this case, is huge. In the diagram throughput vs parallelism, the transition to the next level of parallelism is good enough and differs on each algorithm. The differences between algorithms' throughput are rising as parallelism level is increasing. This is due to the merge cost of the windows.

Chapter 6
Conclusions & Future Work

The purpose of this research is to provide the opportunity to the user, to select over a wide range of functionalities of Flink′s-Windows, those that are required for their needs through algorithm selection. In this work, we embody library WindowSynopsisLibrary to Apache Flink. There is another option for this library to be called through an aggregate function (so that a newer version of Flink can be supported). Furthermore, the scalability of operations of this library is easily achieved. That's because creating a new category of algorithms (or reshape existing ones) is possible by only having to construct methods: Initialize, add, merge and result.

References

[1] https://flink.apache.org/

[2] https://en.wikipedia.org/wiki/Streaming_algorithm

[3] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.695&rep=rep1&type=pdf

[4] https://people.csail.mit.edu/rrw/6.045-2019/encalgs-mg.pdf

[5] https://www.vldb.org/conf/2002/S10P03.pdf

[6] https://www.vldb.org/conf/2002/S10P03.pdf

[7] http://dimacs.rutgers.edu/~graham/pubs/papers/encalgs-ams.pdf

[8] http://dimacs.rutgers.edu/~graham/pubs/papers/cm-full.pdf

[9] https://www.cse.ust.hk/~raywong/comp5331/References/EfficientComputationOfFrequentAndTop-kElementsInDataStreams.pdf

[10] https://arxiv.org/pdf/1407.1121.pdf

[11] https://datasketches.apache.org/

[12] https://polylogblog.files.wordpress.com/2009/08/80munro-median.pdf

[13] http://dblab.kaist.ac.kr/Prof/pdf/Whang1990(linear).pdf

[14] http://cs.haifa.ac.il/~ilan/randomized_algorithms/bar-yosef_jayram.pdf

[15] http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf

[16] http://conferences.sigcomm.org/sigcomm/2005/paper-CaiPan.pdf

[17] http://algo.inria.fr/flajolet/Publications/DuFl03-LNCS.pdf

[18] http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=B0C816A54D472E69C65A8C47EC3F5ACF?doi=10.1.1.20.2080&rep=rep1&type=pdf

[19] https://en.wikipedia.org/wiki/Bloom_filter

[20] http://pages.cs.wisc.edu/~jussara/papers/00ton.pdf

[21] https://webee.technion.ac.il/~isaac/p/infocom12_variable.pdf

[22] https://www.ververica.com/blog/kafka-flink-a-practical-how-to

[23] https://github.com/mayconbordin/streaminer