TECHNICAL UNIVERSITY OF CRETE, GREECE

ELECTRONIC & COMPUTER ENGINEERING

Data Mining Algorithms over Akka & Storm Frameworks

DIMITRA KARATZA

July 2016

ABSTRACT

Efficient processing over massive data sets has taken an increasing importance in the last few decades due to the growing availability of large volumes of data in a variety of applications in computer science. In particular, monitoring huge and rapidly changing streams of data that arrive online has emerged as an important data management problem. Relevant applications include analyzing network traffic, telephone call records, internet advertising and data bases. For these reasons, the streaming model has recently received a lot of attention. This model differs from computation over traditional stored data sets since algorithms must process their input by making only one pass over it, using only a limited amount of working memory. The streaming model applies to settings where the size of the input far exceeds the size of the main memory available and the only feasible access to the data is by making one pass over it.

Typical streaming algorithms use space at most polylogarithmic in the length of input stream. Using linear space motivates the design for summary data structures with small memory footprints, also known as synopses. Algorithms such as Misra Gries, Lossy Counting, Sticky Sampling and Space Saving use parameters support, error and probability of failure, which are specified by the user, in order to extract the items that exceed some threshold (support) from an unbounded data stream. Accuracy guarantees are typically made in terms of those parameters (support, error, probability of failure) meaning that the error in extracting those frequent items is within a factor of $1 + error$ of the true items' frequency with probability at least $1 - \delta$. The space will depend on these parameters.

Since we make only one pass over the unbounded data stream we have to use suitable computation systems. We introduce Storm and Akka frameworks which are both real-time, distributed, fault-tolerant models. Those two frameworks have a completely different architecture which are deeply explained in the current diploma thesis. The crucial difference is that in Storm framework data stream is processed synchronously while in Akka framework data stream is processed asynchronously.

We execute Misra Gries, Lossy Counting, Sticky Sampling and Space Saving algorithms in those two frameworks in a multi node cluster tuning the topologies in order to optimize performance. We observe throughput, the number of processed items in input data set per second. Our goal is to compare the algorithms' behavior in two frameworks.

The data set which is used in order to make our experiments contains two weeks HTTP requests to ClarkNet server. ClarkNet is a full Internet access provider for the Metro Baltimore –Washington DC area.

ΠΕΡΙΛΗΨΗ

Η επεξεργασία δεδομένων με αποτελεσματικό τρόπο έχει μονοπωλήσει, κατά τις τελευταίες δεκαετίες, το ενδιαφέρον των επιστημόνων λόγω του αυξανόμενου όγκου διαθέσιμων δεδομένων που αφορούν ποικίλες εφαρμογές της επιστήμης των υπολογιστών. Ειδικότερα, η εποπτεία ταχύτατα μεταβαλλόμενων ροών δεδομένων σε πραγματικό χρόνο έχει αναδειχθεί ως ένα σημαντικό ζήτημα στη διαχείριση δεδομένων. Σχετικές εφαρμογές αφορούν την ανάλυση της κίνησης στο διαδίκτυο, την καταγραφή τηλεφωνικών κλήσεων, τη διαφήμιση στο Internet και τις βάσεις δεδομένων. Για τους παραπάνω λόγους, υπάρχει μεγάλο ενδιαφέρον για το μοντέλο streaming. Πρόκειται για ένα διαφορετικό τρόπο διαχείρισης των αποθηκευμένων, με παραδοσιακό τρόπο δεδομένων. Το μοντέλο streaming χρησιμοποιεί αλγορίθμους οι οποίοι επεξεργάζονται, με ένα μόνο πέρασμα, τα δεδομένα στην πηγή τους καταναλώνοντας λίγη μνήμη, ώστε το μοντέλο αυτό να αναδεικνύεται ως μοναδική εφικτή λύση όταν ο όγκος των δεδομένων ξεπερνά κατά πολύ το μέγεθος της διαθέσιμης μνήμης .

Οι τυπικοί αλγόριθμοι που χρησιμοποιούνται στο εν λόγω μοντέλο streaming έχουν πολυλογαριθμική χωρική πολυπλοκότητα στην επεξεργασία της ροής δεδομένων. Η γραμμική χωρική πολυπλοκότητα αποτελεί κίνητρο για τον σχεδιασμό data synopsis. Ειδικότερα, αντί να αποθηκεύεται ο μεγάλος όγκος δεδομένων προς επεξεργασία, αποθηκεύονται μόνο τα γενικά χαρακτηριστικά της ροής δεδομένων σε μια δομή. Οι αλγόριθμοι του μοντέλου streaming είναι οι εξής: Misra Gries, Lossy Counting, Sticky Sampling, και Space Saving. Οι εν λόγω αλγόριθμοι χρησιμοποιούν κάποιες παραμέτρους όπως support, error και probability of failure οι οποίες καθορίζονται απ' τον χρήστη προκειμένου να παραχθεί το υποσύνολο των δεδομένων (από τη μη πεπερασμένη ροή δεδομένων) το οποίο υπερβαίνει κάποιο όριο (threshold). Ειδικότερα, το ζήτημα είναι να εξάγουμε τα δεδομένα (items) απ' την ροή τα οποία εμφανίζονται πιο συχνά σε σχέση με τα υπόλοιπα. Η ακρίβεια στην εξαγωγή των δεδομένων αυτών σχετίζεται άμεσα με τις παραπάνω παραμέτρους με πιθανότητα λάθους το πολύ $1 - \delta$ σε σχέση με τη πραγματική συχνότητα εμφάνισης των δεδομένων.

Δεδομένου ότι οι streaming αλγόριθμοι επεξεργάζονται τα δεδομένα της ροής μόνο μια φορά οφείλουμε να χρησιμοποιήσουμε ανάλογα υπολογιστικά συστήματα. Τέτοια συστήματα είναι το Storm, καθώς και το Akka τα οποία χρησιμοποιούνται για real-time ανάλυση δεδομένων. Ακόμα, είναι κατανεμημένα, και έχουν το χαρακτηριστικό ότι έχουν μεγάλη ανοχή λάθους στην ποιότητα και ακρίβεια των αποτελεσμάτων που εξάγουν. Τα δύο αυτά συστήματα, τα οποία ονομάζονται αλλιώς frameworks έχουν τελείως διαφορετική αρχιτεκτονική από τα συστήματα που χρησιμοποιούνται για την επεξεργασία δεδομένων τα οποία είναι ήδη αποθηκευμένα σε κάποια βάση δεδομένων (batch processing). Η αρχιτεκτονική τους αναλύεται σε βάθος στη παρούσα διπλωματική εργασία. Η βασική διαφορά των δύο αυτών συστημάτων έγκειται στο ότι το Storm επεξεργάζεται τα δεδομένα με σύγχρονο τρόπο σε αντίθεση με το Akka σύστημα το οποίο επεξεργάζεται τα δεδομένα με ασύγχρονο τρόπο.

Στην εργασία αυτή υλοποιούνται και εκτελούνται οι αλγόριθμοι Misra Gries, Lossy Counting, Sticky Sampling, και Space Saving και στα συστήματα Storm και Akka σε cluster με πολλούς κόμβους (nodes). Στόχος της εργασίας είναι η βελτιστοποίηση της απόδοσης των αλγορίθμων σε σχέση με τον τρόπο που εκτελούνται στον cluster. Η απόδοση καταγράφεται με βάση τον ρυθμό προσπέλασης των δεδομένων στη μορφή των tuples ανά δευτερόλεπτο (throughput). Ένας ακόμη στόχος της παρούσας εργασίας είναι η σύγκριση των δύο αυτών συστημάτων Storm και Akka σε σχέση με την αρχιτεκτονική αλλά και με τη συμπεριφορά τους καθώς εκτελούνται στον cluster.

Η ροή δεδομένων που χρησιμοποιείται στην εργασία αυτή, προκειμένου να εκτελεστούν τα πειράματα, είναι ένα data set το οποίο περιέχει HTTP requests διάρκειας δύο εβδομάδων στον Server ClarkNet. Ο Server ClarkNet είναι ένας provider που χρησιμοποιείται στο Metro της περιοχής Baltimore – Washington DC.

Contents

List of Figures

1

Problem to Solve

The frequent items problem is one of the most heavily studied questions in data stream research, dating back to the 1980s. Finding frequent items has played an essential role in many important data mining tasks, such as association rule mining, sequential patterns, classification, clustering, etc. The main goal of a frequent items application is the analysis of vast amounts of data for discovering useful information. The extracted information can generate association rules from large databases of transactions for retail industry. Such databases include a huge number of transactions, where each transaction contains a set of items together with other information such as transaction time, customer-id, etc. The market-basket problem is a typical association rule statement, assuming that we have some large number of items, e.g. "bread", "milk". Customers fill their market baskets with some subset of the items, and we get to know what items people buy together. Marketers use this information to position items, and control the way a typical customer traverses the store. Significantly, we care about association rules or causalities involving sets of items that appear frequently in baskets. Association rules can lead to customer profiling.

We could define frequent items phrase as follows.

A frequent element ε_i is an element whose frequency, or number of hits, F_i, in a stream S whose current size is N, exceeds a user specified support ϕN, where $0 < \varphi < 1$

The challenging part of those applications is the fact that in real life the useful information is extracted from data sources that are really huge. Web server logs, twitter logs etc. increase with a significantly high rate. This renders the long term storage of data impossible, as it would cost a lot of memory and time. We thus need to come up with appropriate mechanisms, so as to avoid storing this vast amount of data. Instead, we have to make up algorithms that quickly process each piece of information quickly. The data processing can, in general, be divided into batch processing and real-time processing. Batch processing is preferred in cases where data are preselected (through scripts or shell) and stored in memory. Real-time processing is preferred in cases where data are processed at the time they arrive on the service, making decisions in order to keep the preferred, throwing away the useless data. The selection between batch and real-time processing depends on the characteristics and requirements of the algorithm that we are going to apply. Some algorithms require multiple passes over dataset in order to extract the useful information. So, in this case, batch processing is obligatory. Otherwise, if an algorithm needs a single pass, then real-time processing is preferred.

Unfortunately, frequent items algorithms exhibit significant computational complexity, resulting in long processing times. The computational cost of the algorithms is usually influenced by a need to perform multiple passes over the source data and to perform a significant number of in memory operations. Moreover, the algorithms' performance is dependent on the *nature* of the source data. More specifically, we care about the characteristics (i.e., skew) of the data set

The issue we are dealing with is to find all frequent items in a data stream whose frequency exceeds a specified fraction of the total number of items processed at some point in time. The essential requirement for this problem is to identify frequent items in real time, since the data stream is unbounded, with a limited amount of memory. As we mentioned above, in real-time processing we have the restriction of making a single pass over the data stream, making a data synopsis for the data that we process each time making decisions for the next incoming elements and extracting finally the most frequent items.

We approach this problem with counter-based techniques. Counter-based algorithms track a subset of items from the input data stream and monitor their frequency. Every time a new item arrives those algorithms decide whether to store this item or not, and if so update its corresponding frequency. The prominent counter-based algorithms include Misra Gries, Sticky Sampling, Lossy Counting and Space Saving. We can divide those algorithms into probabilistic and deterministic. For example, Sticky Sampling is probabilistic though the rest of them are deterministic. More specifically, probabilistic algorithm Sticky Sampling decides whether to store an item or not based on sampling of the elements seen so far. It fails to provide frequent items that exceed a specified threshold with a not significant probability of failure. In other hand, deterministic algorithms such as Misra Gries, Lossy Counting and Space Saving are deterministic which means that every time the frequency of an element increases, items for which counts drop below some threshold are ignored or decremented depending on the exact algorithm. The algorithm analysis is explained in chapter 2.

Identifying frequent items is an important task in online monitoring of data streams over various fields of computing such as networking, databases, data mining algorithms and statistics in industry, business and science. This is why frequent patterns mining over data stream is a challenging and important problem to solve.

In chapter 1 we analyze some challenging examples where frequent items algorithms are applied in order to extract top K items that exceed some user specified threshold. Those examples are coming from networks, telecommunications and databases. Also we explain the main idea of frequent items algorithms.

In chapter 2 we analyze each proposed algorithm individually in more detail, while explaining the time complexity.

In chapter 3 we analyze architecture of Storm and Akka frameworks. We explain that those frameworks are used for real-time data processing adjusting Misra Gries, Lossy Counting, Sticky Sampling and

Space Saving algorithms which need a single pass over the data stream. Also we compare Storm and Akka frameworks.

In chapter 4 we make some experiments in Storm framework where we change some configuration on algorithms' topology. We tune algorithms' topology switching between the number of workers in cluster that are used in topology components. We make some diagrams considering number of used machines along with processed tuples per second in order to observe the performance of topologies.

We also make changes in Akka framework tuning configuration files in Actors' architecture in order each time to use a different thread pool for algorithms.

Finally we compare two algorithms Lossy Counting, and Sticky Sampling for both frameworks Storm and Akka explaining the main idea of synchronous and asynchronous threads.

1.1 Challenging examples

Frequent items algorithms are very useful in many applications where data synopsis is required. Here are four problems drawn from computer networks, Internet advertisement, telecommunication networks, and databases.

[1] **Network traffic monitoring**. It is of great importance to track IP addresses that generate considerable amount of traffic in network. The challenge of this problem is that the total number of items could be so large that is impossible to keep exact information for each item. Frequent items algorithms make a synopsis of the data stream, extracting items that exceed a certain fraction of total traffic.

[2] **Internet Advertisement monitoring**. The coordinators in Internet advertising are Internet advertising commissioners, Internet publishers and Internet advertisers. Internet advertising commissioners are positioned as the brokers between Internet publishers and Internet advertisers. An advertiser provides the advertising commissioner with its advertisements, and they agree on a commission for each action such as clicking an advertisement. The publishers contract with their commissioner to display advertisements on their Web sites. Every time a surfer visits a publisher's Web page, the surfer is referred to the server of commissioner who logs the click. To know when advertisements are more likely to be clicked, the commissioner has to know whether the surfer is a frequent "clicker" or not. Frequent items algorithms are useful in order to extract top K "clickers".

[3] **Telecommunication call records analysis**. A telecommunication network produces daily large amounts of call records. It is important for telecommunication operators to see

meaningful statistics about the operation of the network. Frequent items algorithms can give answers about those call records that generate the most traffic.

[4] **Aggregation queries**. In a data stream it is important to find groups whose frequency exceeds a certain threshold, usually expressed as percentage of the size of the relation seen so far. For example,

```
SELECT R.EventType, COUNT(*)

FROM R

GROUP BY R. EventType

HAVING COUNT(*) >  S |R|
```

Where S is support threshold and R is the length of data stream seen so far. All items whose frequency is greater than support threshold will be output of the query.

Those applications and others like them have led to the formulation of the so-called "streaming model". Algorithms for finding frequent items take only one pass over the data stream, computing approximately the list of items that exceed some threshold while using resources, space and time, that are strictly sub-linear in the size of the input. The output must be produced at the end of the stream, or when queried on the prefix of the stream seen so far.

1.2 Frequent items algorithms

One technique for finding the frequent items is counter-based algorithms. Counter-based algorithms maintain a synopsis of the data stream. If the data synopsis captures well the essential characteristics of the entire data set then algorithms may provide approximate results of frequent items. The proposed algorithms are Misra Gries, Lossy Counting, Sticky Sampling and Space Saving.

Counter-based algorithms keep an individual counter for each monitored element in data stream. In order to store monitored elements those algorithms use a data structure like hash map in format <Item, Frequency> where each item is unique. Every time a new element is processed in data stream we check whether exists or not in our data structure. If the element is monitored we increment its frequency counter by one. If the element is not monitored, i.e., there is no counter kept for this element, it is either disregarded or it isn't an active part in the algorithm.

Each counter based algorithm uses its own pattern in order to decide whether an item's frequency is incremented, or decremented. Some algorithms split the initial data set into buckets while some other algorithms apply sampling techniques before collecting the most frequent items.

Despite the different patterns of each counter based algorithm, all of them satisfy some theory properties. First of all, those algorithms accept three parameters which are specified by the user. Those are *s*, support threshold where s ∈ (0, 1), error parameter ε ∈ (0, 1) such that ε << s, and finally probability of failure, δ, where δ ∈ (0, 1). Let N denote the current length of data stream, i.e., the number of elements seen so far. At any point of time, those algorithms can be asked to produce a list of items along with their estimated frequencies. The answers produced by algorithms will have the following guarantees.

[1] All items whose true frequency exceeds sN are output.
[2] No items whose true frequency is less than $(s - \varepsilon)N$ is output.
[3] Estimated frequencies are less than the true frequencies by at most εN.

1.3 Top-K Items

The problem of tracking the top-K items in a data source has also been heavily studied. It is applied in different kinds of applications such as Internet advertising, web mining, twitter logs, stock tickers etc. This algorithmic problem does not always have always a solution under a memory constraint. The reason is that if the elements of a stream S are distinct, then there is a problem in holding all of this data in memory in order to find the most frequent items. This is why Charikar, Chen and Farach-Colton defined the top-K items problem as follows. Let k and ε be two parameters specified by the user. K specifies the number of elements that belong to the most frequent, while ε specifies the error.

An element e_i belongs to top K items if its guaranteed number of hits exceeds the number of hits for the element in position $k + 1$, $count_i - \varepsilon_i \geq count_{k+1}$ (1)

Charikar, Chen and Farach-Colton devised an interesting data structure that allows us to answer the query correctly with probability at least $1 - \delta$ and it uses space $O(k\ \left(\frac{logN}{\delta}\right))$ where N is the length of the data source.

It is mentioned that all of the algorithms in order to exctract top K items, in the worst case, require linear space while their performance depends on some assumptions on the input items distribution., i.e., zipf distribution, random distribution, skew data. According to some citations the top K items in the whole data stream must have linear space complexity.

In the current diploma thesis we follow definition (1). All of the data is processed applying each algorithm. When an item has frequency greater than the above threshold it is tracked as one of the most frequent item (top K).

2

Algorithm Analysis

As described in Chapter 1, the frequent items problem can be solved with many techniques which are classified into counter-based algorithms and sketch algorithms. In this diploma thesis frequent items problem is approached with counter-based algorithms such as Misra Gries, Space Saving, Lossy Counting and Sticky Sampling.

Each algorithm has main aim to extract top k frequent items that are observed in the data set which is processed. All algorithms make one pass over the data set, summarizing data and extracting items-elements with highest frequency depending on *support*, a parameter which is specified by user. So, any time user desires the list with items and corresponding frequencies, the output of each algorithm is a list of items that exceed that threshold.

Beside the fact that each algorithm extract top k values, each of them uses different rules on updating elements and their frequencies at the time they are about to be stored in a data structure.

In next chapters, each algorithm technique is analyzed, as well as there is a description of properties that satisfy each algorithm.

2.1 Misra Gries

The underlying idea of Misra Gries algorithm is that stores k-1 pairs in the exact format: (e, f), where f is the frequency of element e. Every time an element arrives from data stream we check whether it is monitored in some existing data structure S or not. If it is monitored, then we increment element's corresponding frequency. Otherwise, if there is some counter with count zero it is allocated to the new item, incrementing the counter by 1. If all k-1 counters are allocated to distinct items, then *all items* are decremented by 1. So the main parameter of this algorithm is the number of counters that is used and specified by the user, parameter *k*. Any item which occurs more than n/k times must be stored in data structure S.

Below there is the representation of algorithm,

```
S <- empty hash map;
n<- 0;
while not end of stream{
    n <- n + 1;
    if e exists in S{
        increment corresponding frequency of e by 1;
    }else if S< k-1{
        store e in S;
        put value 1 in counter of e;
    }else{
        for each element stored in S{
            decrement counters of element by 1;
            if counter of some element is 0{
                remove element from S;
            }
        }
    }
}
```

Figure 2.1 Misra Gries algorithm.

Misra Gries uses a balanced search tree for data structure in the format (e,f). Each element e requires $logn$ bits and each frequency f requires at most $logm$ bits. Since there are most k-1 key/value pairs in data structure at any time, the total space required is $O\big(k(logn + logm)\big)$.

This algorithm with $k = 1/\varepsilon$, where ε is *ERROR* parameter specified by user, guarantees that the count associated with each item on termination is at most εN below the true frequency.

Properties of Algorithm

Lemma 1: Algorithm with parameter k uses one pass and $O\big(k(logn + logm)\big)$ bits of space, and provides for any element e an estimate f'_e satisfying:

$$f_e - \frac{m}{k} \leq f'_e \leq f_e$$

If some element e has $f_e > \frac{m}{k}$, then its corresponding counter will be positive at the end of Misra Gries pass over the data set which is processed.

2.2 Space Saving

The underlying idea of Space Saving algorithm is to monitor only a pre-defined number of elements, from the data stream, with their corresponding frequencies as counters. The number of counters, denoted as *m*, is specified by user. Counters are related with *ERROR* parameter, denoted as ε, with the following relation: $m = \lceil \frac{1}{\varepsilon} \rceil$. Those counters are responsible for counting the accurate frequencies of the significant elements in the data stream. We use a data structure, denoted as S, in order to store monitored items, with key/value pairs where key represents distinct items of the data stream, and value represent item's corresponding frequency. Data structure is sorted in descending order.

For any incoming element, denoted as e, we check whether it is stored in S or not. If so, we increment element's corresponding frequency $(count_e + 1)$, otherwise we choose from S the element, *min,* with the minimum frequency. Then, we replace *min* with e, incrementing the frequency of minimum value, $count_{\min} + 1$. Thus, the element e could have actually occurred between 1 and $count_{\min} + 1$ times.

Below is a representation of the algorithm,

```
begin
for each element e, in S {
        if e is monitored{
                let $count_i$ be the counter of e;
                increment $count_i$by 1
        }else{
                Let min be the element in data structure with minimum value of counter;
                Replace min with e;
                Increment $count_{min}$ by 1;
        }
}
end;
```

Figure 2.2 Space Saving algorithm.

If data follow skew distribution, the top elements is a minority of the most frequent elements which get the majority of the hits. We assign counters to distinct elements and keep monitoring the fast growing elements. The elements that are growing more popular will gradually be pushed to the list with the top frequencies as they receive more hits. If some of those top elements lose their popularity, they will receive less hits. Thus, its relative position will decline as other counters get incremented, and it might get dropped from the list.

If data don't follow skew distribution, the errors in the counters are inversely proportional to the number of counters. The more counters are kept, the less over-estimation errors in counter's values are occurred, since it eliminates the possibility of replacing elements.

Properties of Algorithm

Lemma 1: The minimum counter value, min, is no greater than $\lfloor \varepsilon N \rfloor$, where N is the length of the data stream.

Lemma 2: If an incoming element, e, with frequency value $count_{\mathrm{e}}$, is not monitored in data structure S, we replace the item, min, with minimum frequency, $count_{\min}$, with e, incrementing the frequency of minimum value by 1. So,

$$1 \leq count_{\mathrm{e}} \leq count_{\min} + 1.$$

Thus, $count_{\mathrm{e}}$ is overestimated by at most $count_{\min}$ times.

2.3 Lossy Counting

The underlying idea of Lossy Counting algorithm is the division of the incoming data stream into buckets. Every bucket has an Id and width $w = \lceil 1/\varepsilon \rceil$, where ε is *ERROR*, one of the parameters that is specified by user. The other parameter is *SUPPORT*, a threshold which is the minimum fraction limit of item's frequency in order to be specified as frequent item.

We denote the current bucket id by $b_{current}$ whose value is $\lceil N/w \rceil$. For an incoming element e, we denote its true frequency in the stream so far by f_e. We also denote parameter Δ which is the maximum possible error in f_e. So, we have tuples in the exact format: (e, f_e, Δ) stored in a data structure S. Whenever a new element arrives, we check whether is monitored in S or not. If it is monitored, we increase its' frequency by 1. Otherwise, a new entry is created in the same form $(\mathrm{e}, 1,\ b_{current} - 1)$. Also, we don't hold all of the elements, instead we delete some entries which satisfy the exact condition: $+\Delta \leq b_{current}$.

The representation of the algorithm is shown below,

```
S <- empty hash map
**while** not end of stream{
        e <- next element in data set
        **if** e is monitored{
                increment frequency of e by 1
        }**else**{
                insert (e,1, $b_{current} - 1$) into S
        }

        **if** $f_e + \mathit{\Delta} \leq b_{current}$ {
                delete $< e, f_e, \mathit{\Delta} >$
        }
}
```

Figure 2.3 Lossy Counting algorithm.

The output of the algorithm is the data structure S which contains the entries of elements with corresponding frequencies with the exact threshold: $f \geq (s - \varepsilon)N$.

Properties of Algorithm

Lemma 1: Whenever deletions occur, $b_{current} \leq \varepsilon N$

Lemma 2: Whenever an entry (e, f, Δ) is deleted, $f_e \leq b_{current}$

Proof: When such entry is deleted, satisfying the condition: $f + \mathit{\Delta} \leq b_{current}$, it happens for some $b_{current} > 1$. This entry was inserted when bucket $\mathit{\Delta} + 1$ was being processed. An entry for e could possibly have been deleted as late as the time when bucket Δ became full. The frequency of e when that deletion occurred was no more than Δ. Further, f is the true frequency of e ever since it was inserted. Thus the frequency of e in buckets 1 through $b_{current}$ is at most $+\mathit{\Delta}$. So, combining f $+ \mathit{\Delta} \leq b_{current}$, we get $f_e \leq b_{current}$.

Lemma 3: If e doesn't appear in S, then $f_e \leq \varepsilon N$

Proof: If the above equation is true for an element e whenever it gets deleted, it is true for all other N also. For Lemma 1 and Lemma 2 we infer that the initial equation is true whenever an element gets deleted

Lemma 4: Lossy Counting algorithm has $\frac{1}{\varepsilon}\log(\varepsilon N)$ space complexity.

2.4 Sticky Sampling

The underlying idea of Sticky Sampling algorithm is to create a synopses of items in a data stream and afterwards to *sample* the items in order to find the frequent ones. Sticky Sampling takes as input three parameters which are specified by user. Those are *SUPPORT* as s, *ERROR* as ε and *PROBABILITY OF FAILURE* as δ. SUPPORT is the minimum value of frequency of an item in order to be reported in the output of the algorithm. ERROR is another threshold which guarantees that some items don't have their true frequency. More specifically, some items may have as estimated frequency less than the true frequency. It follows the restrictions ε << s. PROBABILITY OF FAILURE indicates that the algorithm fails to provide as output the correct items with their estimated frequencies because Sticky Sampling is based in sampling. Thus the algorithm is probabilistic. All of the thresholds are in range (0, 1).

Every time an element is fetched from data stream we check whether we have monitored it before. In order to achieve this we need to use a data structure, a hash map S, with key-value pairs, in the form (e, f) where keys represent distinct elements of the stream and values represent element's corresponding frequencies. So, if an item is monitored in structure S we just increment its corresponding frequency. Otherwise, we sample the element with rate r. That means that this element is selected with probability $1/r$. If item is selected, there is a new entry in structure S as (e, 1). If item is not selected we just ignore it moving on next item.

Sampling rate changes over a lifetime of data stream according to the following formula: $t = \frac{1}{\varepsilon}\log(\frac{1}{s\delta})$. The algorithm starts with rate $r = 1$ for the first $2t$ items, continues with $r = 2$ for the next $2t$ items, and afterwards with $r = 4$ for the next $4t$ items and so on. Sampling rate increases geometrically. Every time sampling rate changes we toss an unbiased coin for each item contained in structure S, decreasing the item's frequency for each unsuccessful toss until the coin toss is successful. If the frequency of an item reach value 0, then we delete the specific entry from structure S.

The representation of algorithm is shown below,

```
S <- empty hash map
r <- 1
while not end of stream{
        e <- next element in data set
        if e exists in S{
                increment corresponding frequency of e by 1
        }**else**{
                sample e with rate r;
                **if** sampled{
                        add entry $(e, 1)$ to S
                }**else**{
                        ignore e
                }
        }
}

Whenever the sampling rate changes:
**for** each entry $(e, f)$ in S
**repeat**{
        toss an unbiased coin
        **if** toss is unsuccessful{
                diminish f by 1;
        }
        **If** f ==0 {
                delete entry from S;
                break
        }
} **until** toss is successful
```

Figure 2.4 Sticky Sampling algorithm.

When user request the list of items, we output the results according to s ($SUPPORT$) where

$f \geq (s - \varepsilon)N$.

Since the space requirements are $2t$ the space complexity is constant. We notice also that space complexity is independent of stream length.

The worst case scenario for Sticky Sampling algorithm is a sequence of items with no duplicates, arriving in any order.

Properties of Algorithm

Lemma 1: All item sets whose frequency exceeds sN are output.

Lemma 2: No item sets whose frequency is less than $(s - \varepsilon)N$ is output.

Lemma 3: Estimated frequencies are less than the frequencies by at most εN.

Theorem 1: Let $r \geq 2$ and let n be the number of items in data stream considered when the sampling rate is r. Then $\frac{1}{r} \geq \frac{t}{n}$, where $t = \frac{1}{\varepsilon}\log(\frac{1}{s\delta})$.

Proof: At the beginning of the phase where sampling rate is $r = 1$ the number of stream items is $n = rt$ (relation 1). The data structure S holds exactly $2t$ elements. As far the algorithm works, the sampling rate increases and rt new elements are considered. Thus, when the sampling rate doubles $r = 2$ at the end of the phase, we have $n = 2rt$ (relation 2) elements.

From relations 1, 2 we conclude that $n \geq rt$.

Theorem 2: Algorithm solves the frequent items problem with probability at least $1 - \delta$ using at most $\frac{2}{\varepsilon}\log(\frac{1}{s\delta})$ expected number of counters.

Proof: We first note that the estimated frequency of a sample element x is an underestimate of the tru frequency, that is, $f_e(x) \leq f(x)$. Thus, if the true frequency is smaller than $(s - \varepsilon)n$, the algorithm will not return x, since it must be $f_e(x) < (s - \varepsilon)n$.

We now prove that there are no false positives with probability greater than $1 - \delta$. Let k be the number of elements with frequency at least σ, and let $y_1, \dots, y_k$ be those elements. Clearly, is must be $k \leq \frac{1}{s}$. There are no false negatives if and only if all the elements $y_1, \dots, y_k$ are returned by the algorithm. We now study the probability of the complementary event, proving that it is upper bounded by δ.

$$\Pr[\exists \text{ false negatives}] \leq \sum_{i=1}^{k} \Pr[y_i \text{ is not returned}] = \sum_{i=1}^{k} \Pr[f_e(y_i) < (s - \varepsilon)n]$$

Since $f(y_i) > sn$ by definition of y_i, we have $f_e(y_i) < (s - \varepsilon)n$ if and only if the estimated frequency of y_i is underestimated by at least εn. Any error in the estimated frequency of an element corresponds to a sequence of unsuccessful coin tosses during the first occurrences of the element. The length of this sequence exceeds εn with probability,

$$(1 - \tfrac{1}{r})^{\varepsilon n} \leq \left(1 - \tfrac{t}{n}\right)^{\varepsilon n} \leq e^{-t\varepsilon} \ ,$$

Where the first inequality follows from Theorem 1. Hence,

$$\Pr[\exists \text{ false negatives}] \leq ke^{-t\varepsilon} \leq \frac{e^{-t\varepsilon}}{s} = \delta$$

This proves that the algorithm is correct with probability greater than $1 - \delta$.

In relation to space usage, the number of stream elements considered at the end of the phase in which the sampling rate r is used must be at most $2rt$ (fromTheorem 1). The algorithm behaves as if each element was sampled with probability $\frac{1}{r}$. Therefore, the expected number of sampled elements is $2t$.

2.5 Data Set

The above algorithms process a data set which contains real logs in ClarkNet server, which is a full internet access provider for the Metro Baltimore-Washington DC area. More specifically, data set contains two week's HTTP requests to ClarkNet server.

The format of logs data set is an ASCII file with one line per request with the following columns:

[1] **host** making the request. When host is available, there is the name of it, otherwise, there is the Internet address if the name could not be looked up.
[2] **timestamp** in the format "day-month-year:hour:minute:seconds". The time zone is -0400.
[3] **request** given in quotes.
[4] **Bytes** in the reply.

The distribution of log file is skewed data with size 39.1 MB.

3

Framework Utilization

Extracting and handling useful information from data sources measuring in gigabytes, or terabytes is a big challenge nowadays. When applying data mining algorithms we are interested in *parallel processing models* for handling data in order to achieve *scalability.*

Parallel processing is an important model for large-scale data-parallel applications. Usually, it is used for arbitrary data in applications such as web indexing and data mining. With this model it is easy to break computation into small tasks that run in parallel on multiple machines running across very large clusters. It originates the meaning of distributed systems in which many machines are grouped together in order to process an application simultaneously extracting the specified data. Some of those distributed systems provides scalability, which is the ability to reach out the maximum throughput for a given application. This is achieved by resizing the cluster, for instance increasing the number of nodes, and optimizing both operational and development perspectives.

Below, there is a schema showing the basic architecture of parallel processing in a cluster, with any data source as input, a load balancer which decides which node or nodes are going to execute the job for the algorithm implementation, a distributed system which implements the algorithm and configures the jobs to be executed, and a Sink where extracted data are going to be stored.

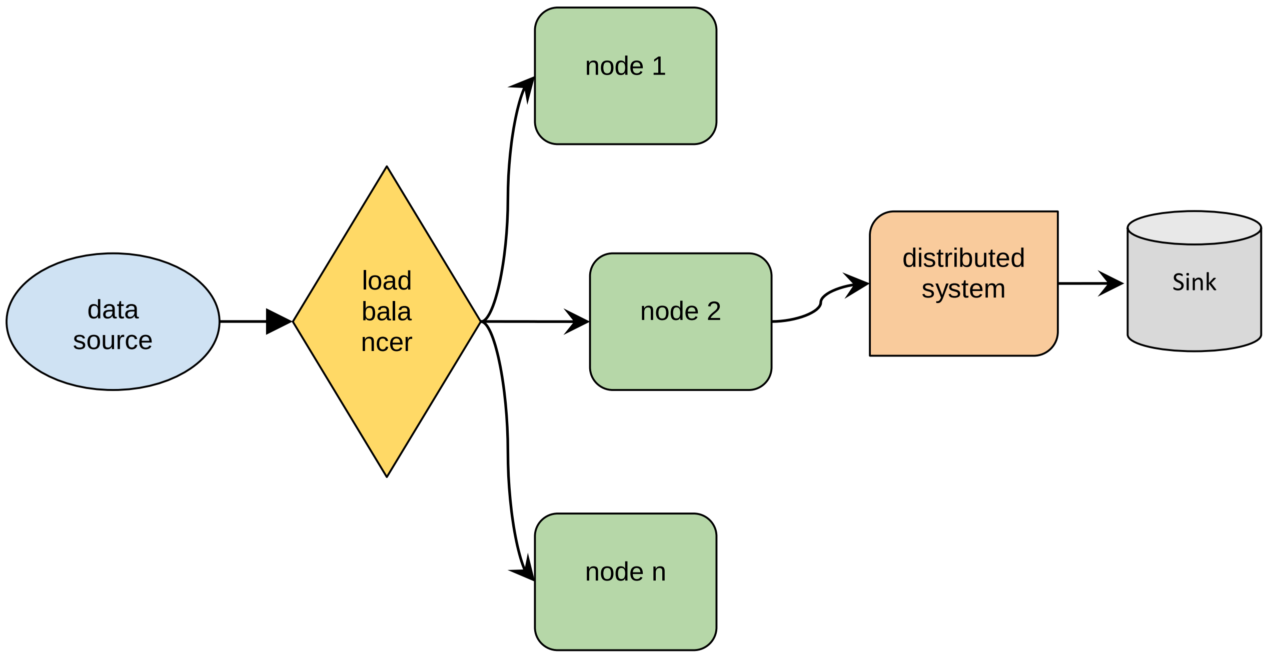


Figure 3.1. Data Source can be data logs, data streams from Web. Load balancer is just a configuration which is made in cluster in order to send data in node which is available that moment, and isn't too busy. Nodes are machines of a cluster which are responsible for the algorithm implementation which is specified in distributed system. Sink can be a Data Base, files system in which aggregated data results are stored.

There are many distributed processing models, which are named as frameworks for processing vast amounts of data. Frameworks are divided in categories with criteria such as architecture, the distribution of data input and of course the nature of the algorithm which is going to be applied in one of those frameworks. The most critical criteria, in my opinion, is whether we want to make batch processing or real time processing of the data. Whether we are obligated to make one or multiple passes over the data set, as it was described in section 1.1. The reason is because at the beginning of my diploma thesis, I was trying to implement clustering algorithms such as CLARA, CLARANS and BIRCH in Storm framework (it's described in next chapter) which demand multiple passes over the input data source. So, when an algorithm demands multiple passes the proposed framework is Hadoop, which processes data which are **already stored** in distributed file systems. On the other hand, when we want to process real time data, data which are not stored somewhere, we still can use batch processing frameworks, such as Hadoop, **but** we can use stronger frameworks such as Storm, S4, Spark, Flink, Samza, Samoa, Akka, which are processing data real time and more quickly.

Each framework has its own ecosystem. For example, *Hadoop*, the most popular distribution system is based on *Map Reduce* model, has *Pig*, *Hive*, *Flume* and *HDFS* (Hadoop Distributed File System). *Spark* is beyond Map Reduce model, which has *RDD* (Resilient Distributed Dataset). Spark differs from Hadoop because it works in memory, speeding up processing times. Also has more flexible pipeline construction. Storm is running on *YARN* which is a hadoop cluster expanding batch-oriented data into a multi-purpose platform supporting a wide range of real time processing of data. *Akka* has an architecture of *Actors* which communicate each other, following a protocol in order to handle data asynchronously.

It's obvious that there is a big variety of distributed systems each with specific architecture, and an entire ecosystem of tools. There is no wide analysis of those, on purpose, because the essence of the diploma thesis is not the comparison between all of them, but the comparison two of them Storm and Akka. In the next chapters, Storm and Akka are analyzed in detail.

3.1 Proposed Frameworks

The algorithms of diploma thesis are applied on frameworks Storm and Akka. Real time distributed systems. The main difference of those is that in Akka framework Actors communicate each other according to one or many protocols in order to process data, asynchronously. On the other hand, Storm has topologies which define the way that jobs are handled, synchronously.

In the next chapters, Storm and Akka architecture, and basic concepts of those are analyzed in detail. In Storm, there are Topologies, Spouts, Bolts, Streams, Streaming groupings, Tasks, Workers. In Akka there are Actors, Dispatchers, Mailboxes, Routing, Futures, Agents, Scheduling.

After this analysis, the algorithms that described in chapter 2 are applied in both frameworks. There is a completely different approach in those two frameworks.

3.1.1 Storm

Storm is a distributed real time computation system. It provides a set of general primitives for real time processing. The key properties for Storm are described as follows:

[1] **Extremely broad set of use cases**. Storm's small sets of primitives satisfy a stunning number of use cased. It can be used for processing and updating data bases, doing some queries, making decisions over data, on the fly.

[2] **Scalable**. Storm can scale to massive number of messages per second. Storm's topology can be easily extended, increasing number of workers that are executing the jobs. The scalability is obvious when we increase the number of nodes of a cluster. We can compare results, observing the computational time to 5, 10, or 20 nodes.

[3] **Guarantees no data loss**. A real time distribution framework is responsible for no data loss. When something goes really wrong in cluster. For instance, if we have internet connection problems, receiving time outs from a client application, or when a node goes down, Storm guarantees that any data is lost. When some data cannot be processed the time requested within an application, Storm saves some kind of pointer in order to "know" where to continue with processing again these data in order to process them.

[4] **Extremely robust**. Because of no data loss guarantee, Storm just works. When a task is not executed for the reasons described above, then it is executed again, until job is completed.

[5] **Fault tolerant**. Storm is responsible for running until all data is processed or until a task is killed manually by the user.

[6] **Programming language agnostic**. Storm topologies, and processing components can be defined in any language such as Scala, Java, python, etc.

Storm's main components are nodes, topologies, tasks or jobs. A Storm topology can run locally in a machine, or in a cluster, with many nodes. So, in a cluster Storm has two main nodes. The master node, which is called *Nimbus* and *workers*. Nimbus is responsible for distributing the jobs which are configured by user to the nodes of a cluster. Worker nodes are responsible to execute whatever Nimbus demand. Worker nodes also, are responsible for reporting the master node the stage of the tasks that they are in time. That's because Nimbus has control of everything and when something goes wrong in cluster, Nimbus has to make decisions. For instance, if Nimbus doesn't receive a report, a message from a worker, which means that the specific node may be down, or has big workload, it may decide to assign this job to another worker. So, this is why Storm guarantees no data loss. Because Nimbus is the controller, controlling everything in a cluster.

It is mentioned that Nimbus "make decisions" according to user's configuration files. A user is able to make some kind of protocol, in which can define the states of the Nimbus logic. For instance, when Nimbus is going to assign a task to a worker, up to a specific workload. What happens when a node fail. All this configuration is stored in a centralized service which is known as *Zookeeper*. Zookeeper provides group of services. All these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services, applications initially usually skimp on them, which make them brittle in the presence of change and difficult to manage. Even then done correctly, different implementations of these services lead to management complexity when the applications are deployed.

Zookeeper aims at distilling the essence of those different services into a very simple interface to a centralized coordination service. The service itself is distributed and highly reliable. Thus, group management, and presence protocols will be implemented by the service so that the applications do not need to implement them in their own. Application specific users of these will consist of a mixture of specific components of the Zookeeper and application specific conventions. Zookeeper recipes shows how this simple service can be used to build much more powerful abstractions.

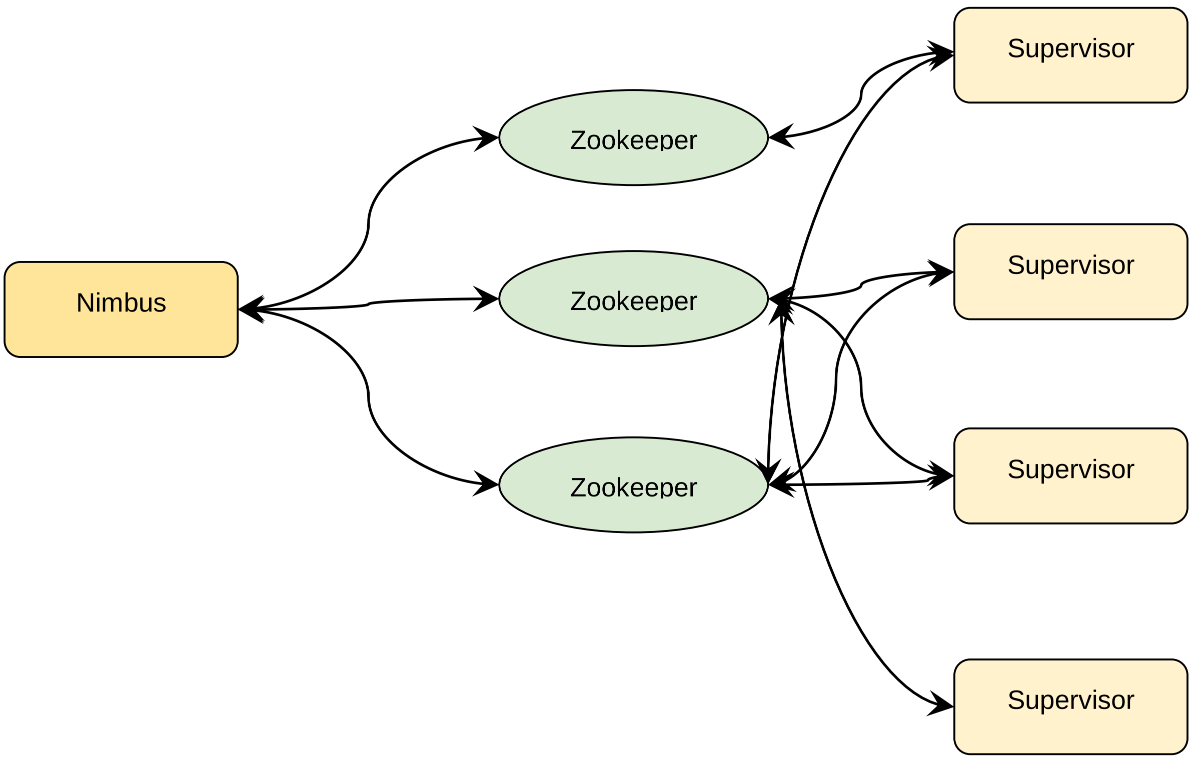


Figure 3.1.1.1. As the schema above shows, Nimbus can connect to more than one Zookeeper and each Zookeeper to whichever worker node it is specified from the configuration of the user. Nimbus and worker nodes store their stage in Zookeeper.

The logic of a real time application is specified in Storm *Topology.* A topology is a graph of *Spouts* and *Bolts* that are connected with *stream groupings*. We should explain Storm's topology in bottom-up way, explaining the main components which are Spouts and Bolts.

Bolts

Bolts, are responsible for the main processing of a real time application. In Bolts can be implemented the main algorithm of an application, aggregations, joins, some filtering. Bolts can also do simple stream transformations. When we have complex stream transformation we can use multiple Bolts. For example, when we want to extract elements with top N frequencies, we need one bolt to count those elements, and another bolt to aggregate those frequencies extracting the top N elements with higher frequencies. So, we assign to each bolt a specific functionality. It is mentioned that bolts can be connected with Data Bases such as *HBase*, *Mongo DB*, *postgres SQL* in order to store the extracted information.

Spouts

A Spout can be any source of streaming data in a topology. Spouts read data from any external Source such as transactions from Data Base, messages from one or more Servers, messages from Queues such as *RabbitMQ* or a Spout may connect to *twitter API* and reading a stream of tweets. While Spouts reading data emit them in Storm topology feeding the topology and specifically Bolts, in order Bolts fetch data and make the processing that an application requires. Spouts can be either reliable or unreliable. A reliable Spout guarantees to emit data to Bolts. When some tuples doesn't manage to go through Bolts, Spouts are responsible to re-send them until Bolts fetch them. An unreliable Spout might read data and forget about the data as soon as data is emitted.

In Spouts we can also make some transformation of the data input, such as in Bolts. For example, data is processed as sequential tuples, so we can change the order of some attributes, transforming the initial format of tuples. For example, in a project I had implemented before it was useful to extract *userId* from stackOverflow which initially was the last attribute, and transform it by putting as first attribute, reordering the tuple. This change was not demanding but it was helpful for the Bolts which were fetching the data.

Topology

A topology, as we said before, is a graph of Spouts and Bolts. Topology is the main architecture of an application. For example, if we want to read data from more than one external sources, we use many Spouts, as much as we want. Then, we emit tuples in Bolts, in every way we want, or it is suitable for our application. We can emit tuples from many sources to the same bolts.

We connect Spouts and Bolts with *Stream Groupings.* Stream Groupings indicate how tuples should be passed around Bolts. There are several types of Stream Groupings:

[1] **Shuffle Grouping**. Tuples are randomly distributed across the bolt's tasks in a way such that each bolt is guaranteed to get an equal number of tuples.

[2] **Fields Grouping**. The stream is partitioned by the fields specified in the grouping. For example, if the stream is grouped by some user Id, tuples with same user Id will always go to the same task, but tuples with different user Ids may go to different tasks. We say "may" because the destination of a tuple depends completely on Storm's hash functions which are responsible to decide where the tuples will be sent. It is mentioned that in Storm, a user is able to write his own hash functions in order to send tuples with his completely logic.

[3] **All Grouping**. The stream is replicated across all the bolt's tasks.

[4] **Global Grouping**. The entire stream goes to a single one of bolt's tasks. Specifically, it goes to the task with the lowest id.

[5] **None Grouping**. This grouping specified that we don't care how the stream is grouped. Currently, none groupings are equivalent to shuffle groupings. Eventually, though, Storm will

push down bolts with none groupings to execute in the same thread as the bolt or spout they subscribe from, when it's possible.

[6] **Direct Grouping**. This is a special kind of grouping. A stream grouped this way means that the producer of the tuple decides which task of the consumer will receive this tuple. Direct groupings can only be declared on streams that have been declared as direct streams. Tuples emitted to a direct stream must be emitted using one of the *emitDirect* methods. A bolt can get the task ids of its consumers by either using the provided Topology context or by keeping track of the output of the *emit* method in *Output collector*, which returns the tasks ids that the tuple was sent to.

Each node (Spouts and Bolts) in a Storm topology executes in parallel as many tasks across the cluster. In topology we can specify how much parallelism we want for each node, and then Storm spawns that number of threads across the cluster to do the execution job. The schema below shows a sample of topology,

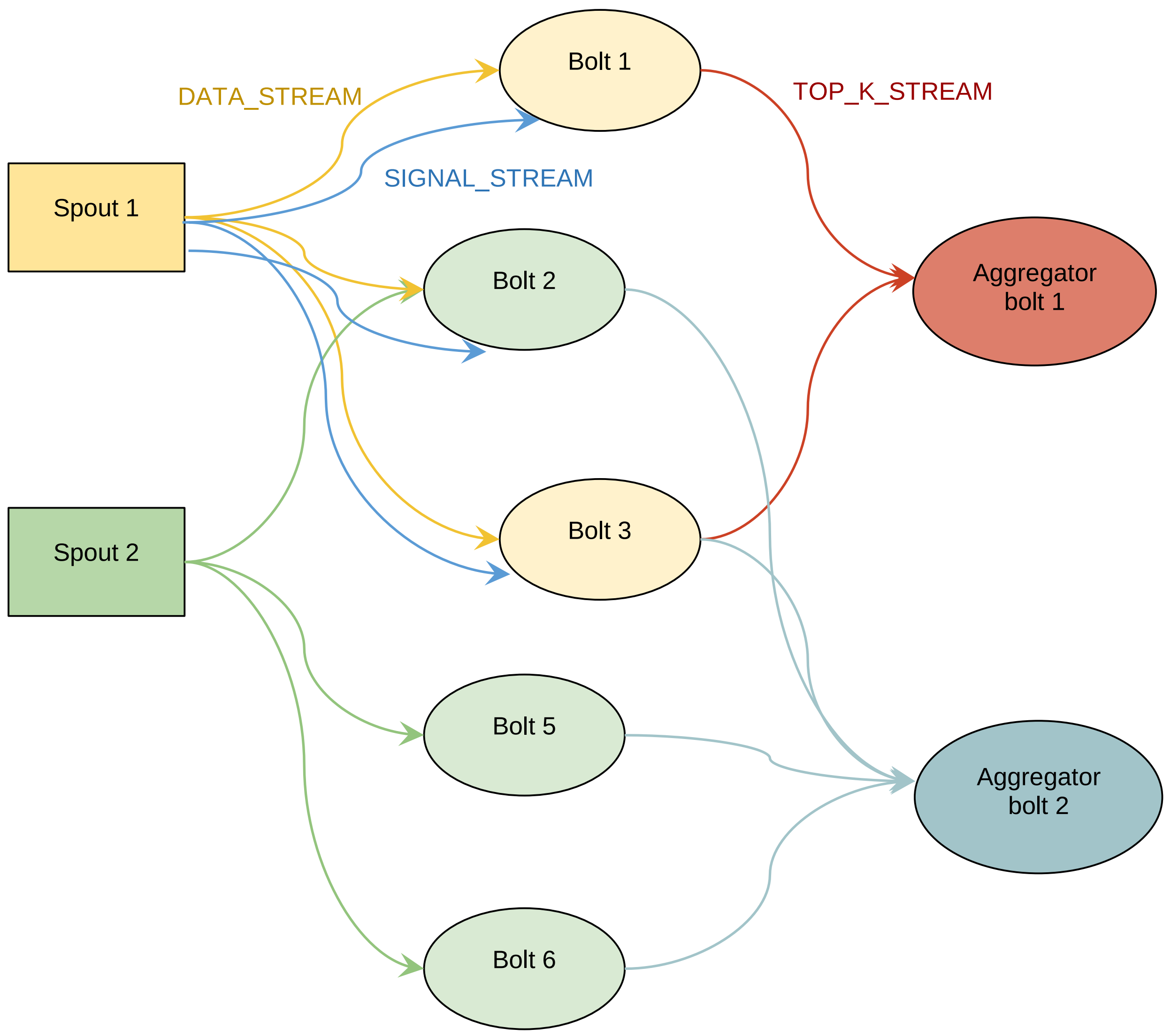


Figure 3.1.1.2. As we can see from the above topology, one spout can send data to any bolt. Aggregating data from any bolt.

Considering a subset of the above topology Spout 1, bolt 1, bolt 3 and aggregator bolt 1, we could write the following code in order to connect those components with some grouping.

```
Config. setNumWorkers(8);

TopologyBuilder.setSpout("Spout 1", new Spout(), parallelism_hint)

TopologyBuilder.setBolt("Bolt 1", new Bolt(), parallelism_hint)

                    .fieldsGrouping("Spout 1",DATA_STREAM, new Fields("id"))

                    .allGrouping("Spout 1",SIGNAL_STREAM);
```

```
TopologyBuilder.setBolt("Aggregator Bolt 1", new Bolt(), parallelism_hint)
                    .shuffleGrouping("Bolt 1", TOP_K_STREAM);
```

The above configuration is used for both local mode and cluster mode. Local mode simulates a Storm cluster in process by simulating worker nodes with threads. In cluster mode, Storm operates as a cluster of machines. When we submit a topology to the master node, master node takes the responsibility to assign the tasks in slave nodes. The *setNumWorkers* parameter specifies the number of allocated processes to execute the topology. Each component (Spout, Bolt) will execute as many threads. The number of threads allocates to a given component is configured through *parallelism_hint* parameter. It indicates how many threads should execute that component across the cluster. This parameter is optional and if is not specified the default value is *1* which means that Storm will allocate only one thread for that node.

Tasks perform the actual data processing. Each Spout or Bolt executes as many tasks across the cluster. In topology configuration we specify how tasks are connected with. We achieve this with *Stream Grouping* as we explained previously. In the above example we connect *Spout 1* with *Bolt 1, Bolt 2, Bolt 3* with fields grouping. Actually we send tuples to Bolts grouped by Id. This is indicated by DATA_STREAM configuration which reflects the actual data we processed from the source. Tuples with same user Id will always go to the same task, but tuples with different user Ids may go to different tasks. Storm framework implements fields grouping using mod hash functions in order to distribute tuples to tasks of the topology. A Storm user is also able to write his own hash functions passing out the tuples to tasks however he wants. Beside DATA_STREAM we have also *SIGNAL_STREAM* configuration with *All Grouping* connection. With All Grouping we send replicated data across the Bolt's tasks. We usually use this kind of Grouping when we want to send signals to Bolts, i.e., an indication that data reading from the source is completed. Another example is when we are doing some kind of filtering on data stream, we have to pass the filter parameters to all the Bolts. This can be achieved by sending those parameters over a stream that is subscribed by all Bolts' tasks with All Grouping.

Finally we connect *Bolt 1and Bolt 3* with *Aggregator Bolt 1* with *Shuffle Grouping.* With this grouping we distribute top K values of the processed data in a uniform random way across the tasks, with the guarantee that an equal number of tuples will be processed by each task. It is remarkable to mention that if we have only one task for Aggregator Bolt 1 it doesn't matter which grouping we use since tuples will always go to this Bolt.

3.1.2 Akka

Akka is another distributed real time computation system. Actor model has a more abstract API than Storm. It provides a better platform to build scalable, resilient and responsive applications. As it is mentioned in Akka's documentation it is a "let it crash" model because of the fact that with Akka Actors can build applications that are self-healed and systems that never stop. Actor model has some key properties,

[1] **Responsive.** The system responds in a timely manner if it all possible. Responsiveness is the cornerstone of usability and utility, meaning that problems may be detected quickly and dealt effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. The consistent behavior simplifies error handling, build end user confidence, and encourages further interaction.

[2] **Resilient.** The system stays responsive in the face of failure. This applies not only to highly-available, mission critical systems (any system that is nor resilient wil be unresponsive after a failure). Resilience is achieved by *isolation, delegation*. Isolation means in abstract that sender and receiver actors have independent life-cycles, which means that they don't need to be present at the same time for communication to be possible. Delegation means that the execution task – job which is going to be executed to a component will take place in the context of the component. Failures are contained within each component which is going to be executed, isolating components from each other and thereby ensuring that parts of the system can fail and recover without the whole application crashes.

[3] **Elastic.** The system stays responsive under varying workload. Actor model can react to changes in the input rate by increasing or decreasing resources, such as CPU, main memory bandwidth, or even external services like data bases. The elasticity and resilience of all these resources must be considered carefully, because a lack of those or a wrong configuration may lead to misbehavior of the system.

[4] **Message driven.** Actor model rely on asynchronous message-passing to establish a boundary between components. Message passing enables load management, elasticity and flow control by shaping and monitoring message queues in the system.

The essence of Akka actors, beyond asynchronous messages, is the fact that is Non-blocking. This means that if threads competing for a resource don't have their execution indefinitely postponed by mutual exclusion protecting that resource. If one actor needs resources it's going to make a request in order to use those resources. If resources are available that moment then the request is successful. Otherwise, actor is not going to wait until resources are free to use. Instead, when resources are free to use, then a notification is sent to Actor in order to use them. Until the specific actor receive the notification for using resources, the rest application is working.

Actors, are some entities that execute jobs. Actors communicate each other with messages. Each Actor has its *mailbox* which is some kind of queue with messages to process. Actors have a behavior which is specified by a *protocol* defined by the user. When modeling with Actors in order to make an application, Actors are assigned sub-tasks (by protocol) sending messages and waiting for some response consisting a whole application. Below there are described the basic components which consist an Actor model.

State

Actors contain some variables which reflect states the actor may be in. Each Actor as mentioned previously can send and receive messages. With Actor model, many real threads *may* execute one or more messages of the corresponding Actor. But, there is no guarantee that one thread will execute **all** of the messages of an Actor. This is why the possible states of an Actor must be consistent. Threads can be spawned and destroyed with minimal overhead. Thus, a large number of instances can be created a running in parallel. There are two kinds of Actors: *Thread-based* Actors and *Event-driven* Actors.

Thread-based Actors

There is a basic function in each Actor, the *receive* method which is responsible for handling a receiving message according to the behavior which is described in the possible cases of the specific method. When receive method is used, the Actor is backed by one *thread*. This fact limits scalability requiring the thread to suspend and block when waiting for new messages.

Event-driven Actors

Instead of one thread, there is a pool of threads which is used for a number of Actors. This means that when an Actor receives a message then a thread is backed up responsible to handle this messages, but it doesn't mean that the next messages which an Actor receives are going to be processed by the *same* thread. If another thread z is available that moment, some of the next messages are going to be processed by thread z.

Behavior

Every Actor is responsible for having a specific behavior to receiving messages. Behavior means a function which defines the actions to be taken in reaction to the message at that point in time. The whole behavior of an Actor is specified by its protocol. The user of the application is responsible to consider all of the possible states a message may be. The most important fact here is the error handling. For example, in current project when I firstly applied the algorithms to Akka Actors I didn't consider all of the possible states that a message may be. So, I was throwing errors or wrong results.

Mailbox

An Actor receives messages from another Actor, from an external service or even from itself. When this happens, Actor stores them in a queue ready to be processed. Enqueuing happens in the same time of sending operations, which means that messages sent from different actors may not have a defined order at runtime due to the apparent randomness of distributing Actors across threads. On the other hand, sending multiple messages to the same target from the same Actor is a guarantee that messages will be enqueued in the same order.

Supervisor Strategy

Supervision describes the dependency relationship between actors. One supervisor-Actor may have children-Actors delegating sub-tasks to them. Every time a supervisor Actor is resumed, is restarting, is terminated then children-Actors have the same state. When a child-Actor detects a failure (exception) it suspends itself and sends message to its supervisor. Then supervisor-Actor is responsible for error handling.

Below there is a topology of Actors, mailboxes, and messages which are executed asynchronously.

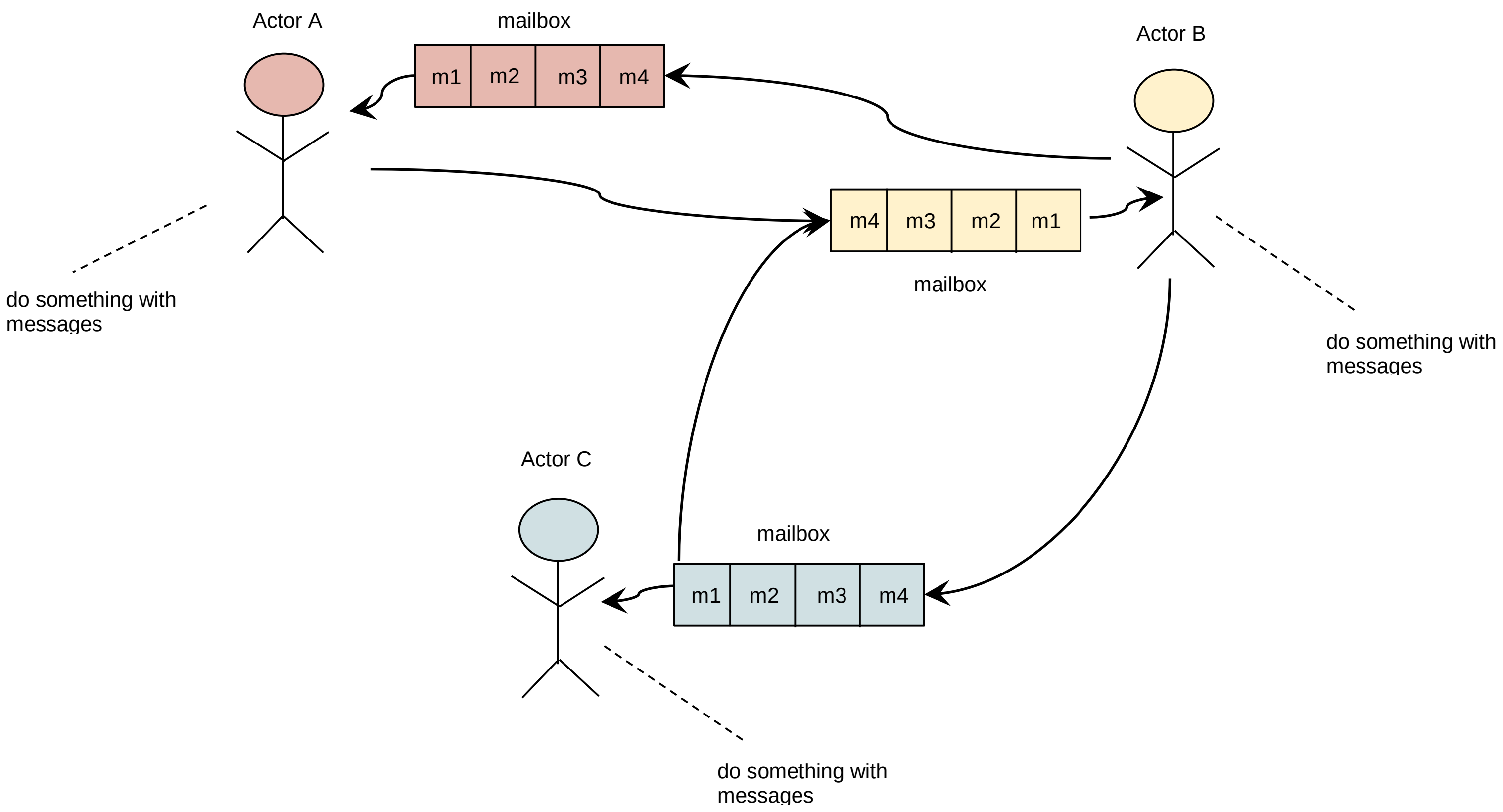

Actor A
mailbox
m1
m2
m3
m4
Actor B
m4
m3
m2
m1
mailbox
do something with messages
do something with messages
Actor C
mailbox
m1
m2
m3
m4
do something with messages

Figure 3.1.2. Every Actor is able to send messages to one or many Actors. Those messages are stored in each Actor's mailbox where they are temporarily stored, to be processed later one at a time.

3.1.3 Storm Vs Akka

As mentioned previously Storm is a distributed real time computation system. On a Storm cluster there are executed topologies which process streams of tuples. Each topology is a graph consisting of Spouts which produce tuples and bolts which transform tuples. Storm takes care of cluster communication, fail-over and distributing topologies across cluster nodes.

Akka is also a real-time distributed system for building concurrent, fault tolerant applications. In Akka application the basic construct is an Actor. Actors process messages asynchronously and each Actor can also be deployed remotely.

The concept of both frameworks is the same. The basic unit of data in Storm is a tuple. A tuple can have any number of elements, each tuple element can be any object. In Akka, the basic unit is a message, which can be any object.

The basic unit of computation in Storm is bolts and spouts. A bolt can be any piece of code, which does arbitrary processing on the incoming tuples. It can also store some mutable data. For example, to accumulate results. Moreover, bolts run in a single thread, so unless there are started additional threads in bolts, there is no worry about concurrent access to bolt's data. This is very similar to Actor. Actors can receive arbitrary messages, bolts can receive arbitrary tuples. Both are expected to do some processing basing on the data received. Both, also, have internal state which is private and protected from concurrent thread access.

One crucial difference is how Actors and bolts communicate. An Actor can send a message to any other Actor as long as it has the Actor reference. It can also send back a reply to the sender of the message that is being handled. Storm, on the other hand, is one-way. Bolts can't send back messages to spouts. Bolts can send ack (acknowledgement) messages which is also a form of communication to other bolts.

In Storm, multiple copies of bolt's or spout's code can be run in parallel, depending on the user's parallelism setting. So this corresponds to a set of potentially remote Actors, with load-balancer Actor in front of them. There are couple of choices on how tuples are routed to bolt instances in Storm, random, grouping on a field, etc (as they were described in chapter 3.1.1). This roughly corresponds to the various options of Akka like round robin, consistent hashing on the message.

There is also a difference in the "weight" of a bolt and an Actor. In Akka, it is normal to have lots of Actors – up to millions. In Storm, the expected number of bolts is significantly smaller. This isn't any case downside of Storm, but rather a design decision. Also, Akka Actors typically share threads, while each bolt instance tends to have a dedicated thread for executing each task.

Storm also has one crucial feature which isn't implemented in Akka. Storm tracks the whole tree of tuples that originate from any tuple produces by a spout. If all tuples aren't acknowledged, the tuple can be replayed, depending on the user's setting. As mentioned in chapter 3.1.1, spouts are reliable or unreliable.

The layout of the communication in Storm – the topology- is static and defined upfront. In Akka, on the other hand, the communication patterns can change over the time and can be totally dynamic. Actors can send messages to any other Actors, or can even send addresses.

3.2 Applying algorithms in Storm

In this chapter algorithms methodology is described as well as the functions which are implemented in Storm framework. As mentioned in chapter 2.5, data set contains two weeks information of all HTTP requests to ClarkNet www server. Data set has skewed distribution consist of five columns: *host making the request, timestamp, request, HTTP reply code* and *bytes in the reply*. The main functionality of all algorithms *Misra-Gries, Lossy-Counting, Space-Saving, and Stick-Sampling* is the extraction of top K host names. More specifically K host names with highest frequencies.

Below, is the topology of the above algorithms with the extracted information of each component of topology.

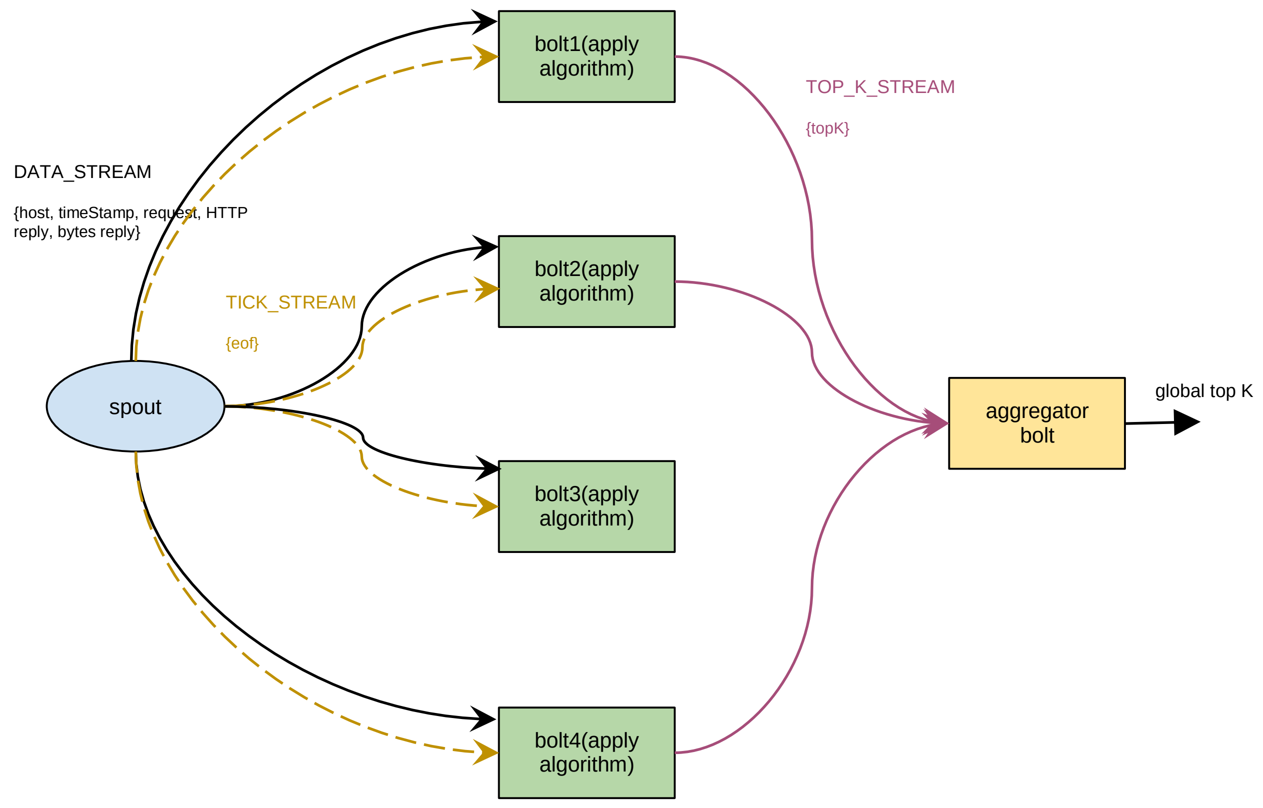


Figure 3.2. The topology structure representing all algorithms in Storm framework. Spout component emits host name and eof (endo of file) to Storm's collector. Host name is emited via DATA_STREAM and eof message via TICK_STREAM. Host name is emited with fields grouping while eof is emited via TICK_STREAM. Each bolt extracts each top K items, via TOP_K_STREAM, while aggregator bolt extracts global top K results. It doesn't matter which stream grouping is used between bolts and aggregator bolts since there is only one aggregator bolt.

In Spout is processed every single tuple of the data set emiting it to bolts. In bolts it is implemented the main algorithm extracting the top k host names. The stream grouping which is used is fields grouping in which is guaranteed that tuples with same host names will always go to the same bolt, but tuples with different host names go to different bolts. So, each bolt extracts top K host names of the tuples that are emited. Eof message is emited and replicated to all of bolts in order each bolt is informed when reader reach the end of file. So, all grouping is used in stream grouping configuration. In aggregator bolt are emited groups of top K host names extracting the global top K items. It is mentioned that it doesn't matter which stream grouping is used between bolts and aggregator bolt since we have only one aggregator bolt.

Functions

Since Storm is a framework it contains specific functions which have to be overridden in order to be specified the behavior of the topology. Of course, except the following functions there are implemented helper functions which are needed in order to handle the tuples such as transformations, serialization of objects, etc.

In Spout there are implemented the functions below:

- ***declareOutputFields***. The fields which are declared are the *host name* and the indication that the entire data set is processed named as *eof*.
- ***open***. It is specified the way that the file of the data set is opened.
- ***nextTuple.*** This is the main function in which is configured the possible states when reading the tuples from the file. Tuples with host names are emited to output collector of Storm and when the reader of file reach the end of it, the *eof* message is emited also to output collector.
- ***createTuple.*** In this function there are made transformations in order to extract the desired information which is host name of each tuple.

In Bolt there are implemented the functions below:

- ***declareOutputFields***. The fields which are declared are top K items.
- ***prepare***. In this function it is initialized each algorithm with parameters such as support, max error, number of counters (in space saving algorithm).
- ***execute***. This is the main function in which there are three states. If is tick tuple, there are loged to a file just the statistics of each bolt. For instance the rate in which tuples are processed. If a host name is received the algorithm handles the specific item, updating its counter. If is a message from TICK_STREAM, top K results are emited to collector of Storm after they have been serialized to a single object.

In Aggregator Bolt there are implemented the functions below:

- ***prepare***. This function is responsible for receiving each bolt's top K items.
- ***execute***. This function is responsible for logging top K items when all of them have been received. When this is achieved the global top K results are extracted, after they have been deserialized.

3.3 Applying algorithms in Akka

In this chapter algorithms methodology is described as well as the functions which are implemented in Akka framework. It is mentioned that it is used the same data set with host names in order to extract the top K items. The algorithms that are implemented in Akka framework are *Lossy-Counting* and *Sticky-Sampling*.

Below is the architecture of Actors as well as the communication of each other with messages.

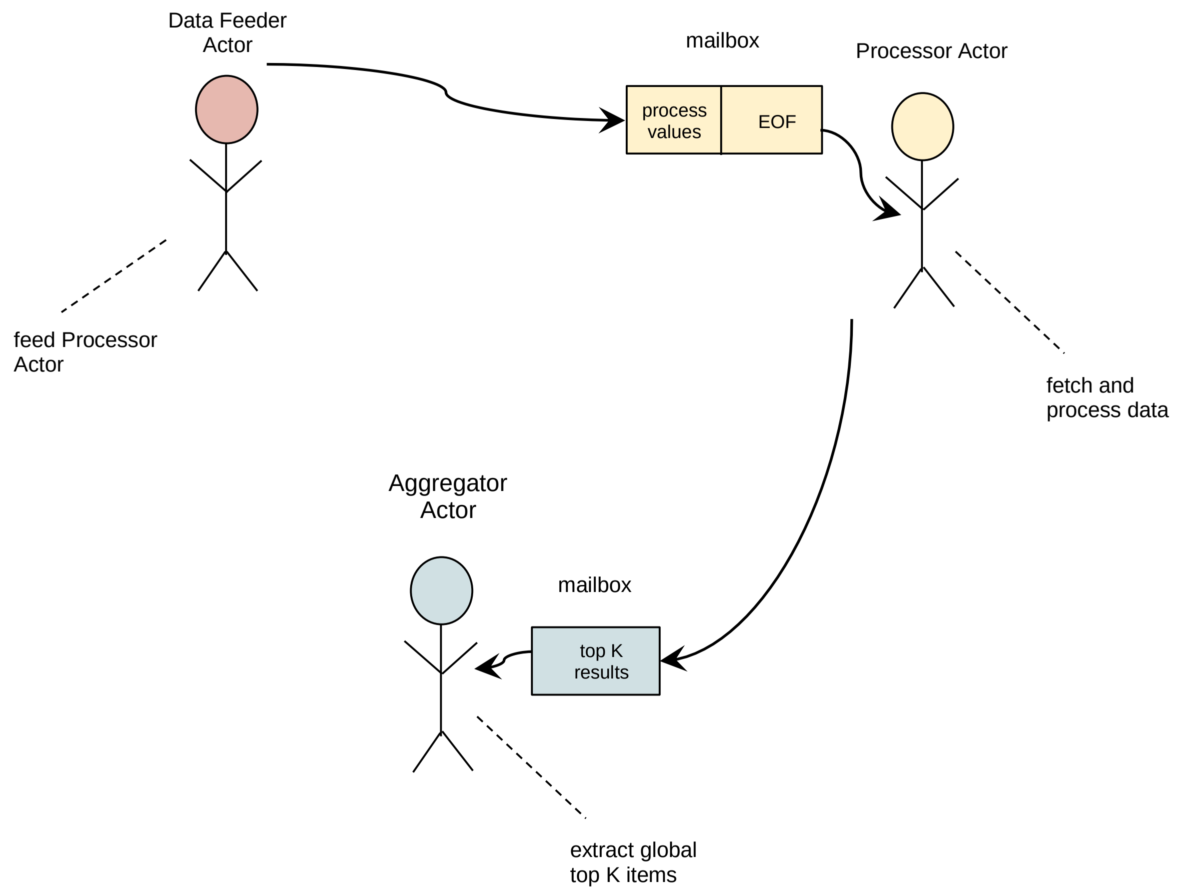


Figure 3.3. The topology structure representing Lossy-Counting and Sticky-Sampling algorithms in Akka framework. Processor Actor has two possible messages to receive. Either to process values or indicate that the entire data set is processed by Data Feeder Actor. When Processor Actor process all of the data applying the algorithm, it sends one message to Aggregator bolt in order to extract global top K items.

Every Actor communicate with each other via messages. Messages are declared and described in a protocol which is defined by me. Each message is a *state* in which Actor obtains a behavior. Data *Feeder Actor* processes every single tuple of the data set and indicates whether the reading of file is completed. Those two messages are send to *Processor Actor* asynchronously. That means that Processor Actor doesn't know apriori which message is going to receive and when. For that reason, Processor Actor is responsible to handle appropriately each message. This Actor also handles the values applying the main algorithm. Finally, Aggregator Actor receives a message when all of the items have been processed from all processors in order to extract the top K results.

Functions

Below there are described functions that were implemented in order to Actors have desirable behavior and implement properly each algorithm. Of course, except the following functions there are implemented helper functions which are needed in order to handle the tuples such as transformations, serialization of objects, etc.

In Data Feeder Actor there are implemented the functions below:

- ***prestart***. This function is called after the Actor is started. The resources are initialized connecting all the Actors: *DataFeeder* Actor, *Processor* Actor and *Aggregator* Actor.
- ***receive***. This method contains two states, reading the data set, and indicating the eof(end of file).

In Processor Actor there are implemented the functions below:

- ***prestart***. In this function it is described the way in which periodic messages are scheduled. Scheduled periodic messages send are restarted when the Actor is restarted. This means that the time period that elapses between two tick messages during a restart may drift off based on when the scheduled messages are restarted. It is also specified the initial delay is going to be.
- ***receive***. Three states are proposed here. If there is a *processValue* message, then the logic of the algorithm is applied in every tuple of the data set. If it is a request for eof, a message is send to DataFeeder Actor. The third state is sending the top K items to Aggregator Actor, when all of the data is processed.

In this Actor the logic of each algorithm is implemented.

In Aggregator Actor there are implemented the functions below:

- ***prestart***. This function just logs statistics for Aggregator Actor.
- ***receive***. There is only one possible state in this function. If there is a message from some Processor Actor that holds top K items, then top K items are collected in a Map[hostname, frequency]. When all the Processor Actors send their top K items, then the global top K items are calculated from Map.

There is also the Protocol class in which are declared and defined all the possible messages of the Actor's topology. This messages are shown below:

- ***startFeeder***. Message that indicated the start of reading from source and sending data for further processing.
- ***processValue***. Message that indicates the item should be processed by the Processor Actor.
- ***isEOFRequest***. Message to request if end of input or not.
- ***topKData***. Message that holds the top K results from some Processor Actor.

4

Performance Tuning

It is mentioned in chapter 3 that Misra-Gries, Space-Saving, Lossy-Counting and Sticky-Sampling are implemented based on Storm architecture and furthermore that Lossy-Counting and Sticky-Sampling are implemented in Akka Actors logic. After running and testing all of the algorithms in local mode, the next step was to run those algorithms in multi-node cluster. It is mentioned that a Storm cluster architecture is some kind of different of Akka cluster architecture. Thus, we explain individually those two frameworks in cluster mode.

Storm Cluster

As it is described in chapter 3.1.1, Storm architecture consists the master node (*Nimbus*) followed by worker nodes (*Supervisors*). Master node is responsible for assigning tasks to worker nodes monitoring the cluster for failures. Worker nodes are executing the tasks that are assigned to them, starting and stopping whenever Nimbus tells them to. Each worker process runs a subset of Storm topology and may run one or more executors for one or more components such as Lossy-Counting-Spout, Lossy-Counting-Bolt, Lossy-Counting-Aggregator-Bolt. An executor is a thread that is spawned by a worker and it may run one or more tasks for the same components described previously. Tasks execute the actual data processing.

So, for our topology architecture for all algorithms we have two Spouts and two Bolts bind them as follows,

```
Config. setNumWorkers(numOfWorkers);

TopologyBuilder.setSpout("lcFileSpout1", new LossyCountingSpout(), parallelism_hint_lcSpout1);

                    .setNumTasks(numTasks_lcSpout1)

TopologyBuilder.setSpout("lcFileSpout2", new LossyCountingSpout(),parallelism_hint_ lcSpout2);

                    .setNumTasks(numTasks_ lcSpout2)


TopologyBuilder.setBolt("lcBolt", lcBolt, parallelism_hint)

                    .setNumTasks(numTasks_lcBolt)

                    .fieldsGrouping("lcFileSpout1", StreamConfig.DATA_STREAM, new Fields("hostName"))

                    .fieldsGrouping("lcFileSpout2", StreamConfig.DATA_STREAM, new Fields("hostName "))

                    .allGrouping("lcFileSpout1", StreamConfig.TICK_STREAM);

                    .allGrouping("lcFileSpout2", StreamConfig.TICK_STREAM);


TopologyBuilder.setBolt("lcAggregator", new LossyCountingAggregatorBolt(), parallelism_hint)

                    .shuffleGrouping("lcBolt", StreamConfig.TOP_K_STREAM);
```

Figure 4.1. The topology structure for Lossy-Counting algorithm with parallelism level.

For our experiments in Storm cluster we configured the parameters above considering the total number of nodes that exist and they are active in the cluster. The total number of nodes is 12. So, we configured **numOfWorkers** with the following restriction $1 \leq \mathbf{numOfWorkers} \leq 12$. The main goal of experiments below is to optimize the performance of each topology in cluster.

Performance of Storm topology can be measured in many ways such as monitoring the *throughput* of each algorithm and *CPU time* that it takes in order to process and execute the tuples that are emited and transferred. In this diploma thesis we are going to consider only the throughput of each algorithm, which is the rate of total processed tuples per second. According to this, we are concerned only for number of workers, number of executors and number of tasks that run for each topology. More specifically, the first category of our experiments is increasing the number of workers that run each time, and the second category is increasing and/or decreasing the parallelism level for one or more Spouts or one or more Bolts, defining the number of executors and tasks that execute each component of our topology but having a fixed number of workers running in topology. At the end of experiment results we compare all algorithms for a specific behavior. For instance, which is the bottleneck for each algorithm topology? Which is the best performance (best throughput) for each algorithm?

Akka Cluster

Comparing Storm to Akka Cluster, there is a leader node which is similar to Nimbus and nodes which are similar to Supervisors. A leader node is responsible for *load balancing*, that is an Actor that apply default hash functions to messages in order to "decide" which tuples are processed by specific simple Actors. As described in chapter 3.1.2 Actors receive arbitrary messages asynchronously. This is the main difference between those two frameworks. Spouts and bolts have dedicated threads that run. In contrast, Akka Actors share threads. Also, the layout of Storm topology is static compared with Akka architecture which is dynamic because communication patterns between Actors may change over time. In Akka Actors parallelism level is defined in configuration file within project by specifying the minimum and the maximum threads that run in cluster. This configuration is declared as follows,

```
parallelism-min = numOfThreads

parallelism-max = numOfThreads
```

In Akka cluster we run Lossy-Counting and Sticky-Sampling algorithms considering that there were 5 nodes active. The only experiment that we made was configuring different values for minimum and maximum number of threads that run tasks.

4.1 Tuning the Performance of Algorithms in Storm

As we described in intro of this chapter, we are considering two types of experiments in Storm cluster. Firstly, we increase number of workers in order to observe throughput (tuples/sec) of each topology algorithm. Secondly we have a fixed number of workers switching over the parallelism level of each topology's component such as Spout, Bolt and Aggregation Bolt. The aim of the second experiment which is the most important is to figure out which components are the "heavy" workers. In other words we have to observe the workload progress of each topology.

4.1.1 Tuning the Performance of Misra Gries Topology

Below we show the results of the first category of experiments,

workers	Spout Executors	Spout Tasks	Bolt Executors	Bolt Tasks	Aggr. Executors	Aggr. Tasks	Rate (tuples/sec)
4	2	4	4	8	2	4	38
6	2	4	4	8	2	4	41
8	2	4	4	8	2	4	44
10	2	4	4	8	2	4	47

The number of executors and tasks for all components is the same as the number of worker is increased. As the number of workers increases the rate also increases between 38-47 tuples/sec. We can consider that when we increase the number of workers we actually increase the number of worker processes across machines in the cluster. So, when we use more machines the workload is distributed. The most important hint here is the fact that for each component we set number of tasks multiple to number of executors. That means that each machine has the same number of threads and roughly the same amount of work.

For example, if we consider the third case, we are using 8 machines, for Spout we are using 2 executors and 4 tasks, for Bolt we are using 4 executors and 8 tasks and for Aggregator Bolt we are using 2 executors and 4 tasks. That means that we available: 8 executors/8 nodes => *1 thread per machine*. So, if we consider that they are running 2 tasks/executor for Spout, 2 tasks/executor for Bolt and 2 tasks/executor for Aggregator Bolt, the workload is distributed as follows,

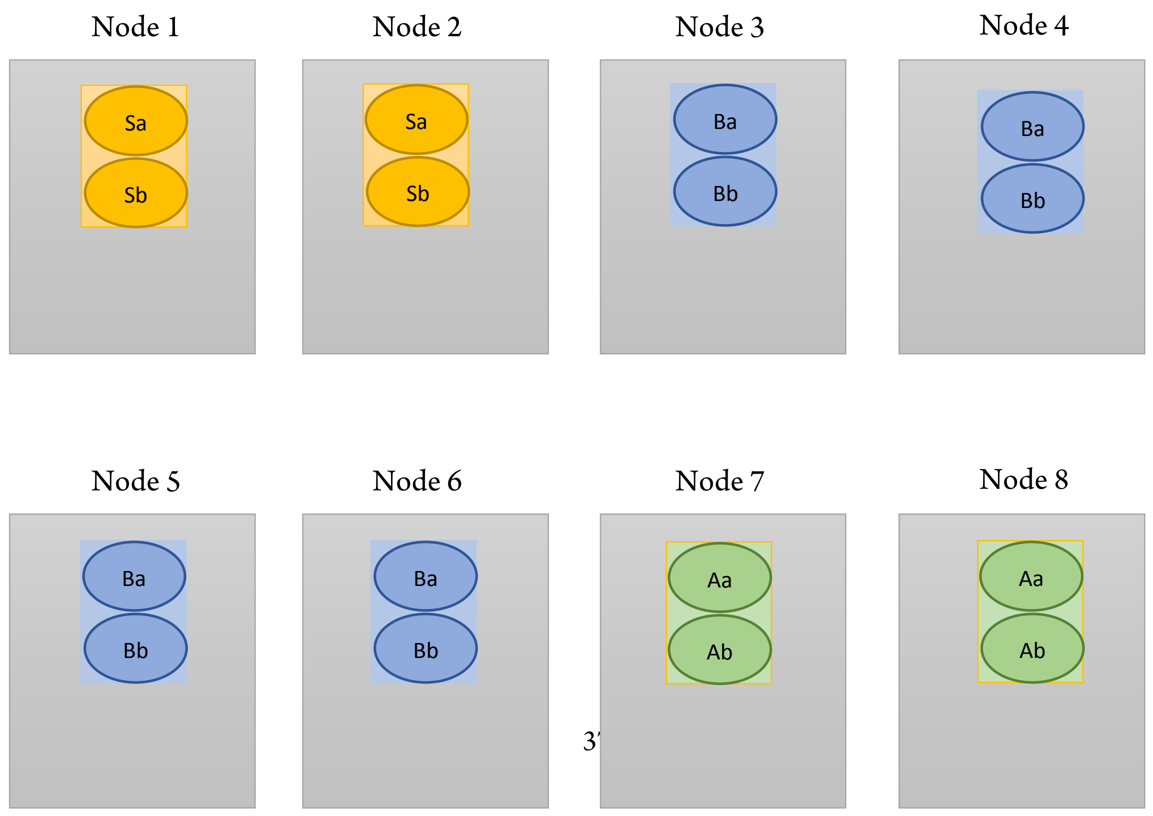


Figure 4.1.1.1 Misra Gries workload distribution in cluster's machines. Each machine uses 1 thread. All components need 2 tasks per executor. Spout in yellow has 2 executors running in 2 machines, Bolt in blue has 4 executors running in 4 machines and Aggregator Bolt has 2 executors running in 2 machines. Each machine has the same number of threads as well as the same amount of work to execute.

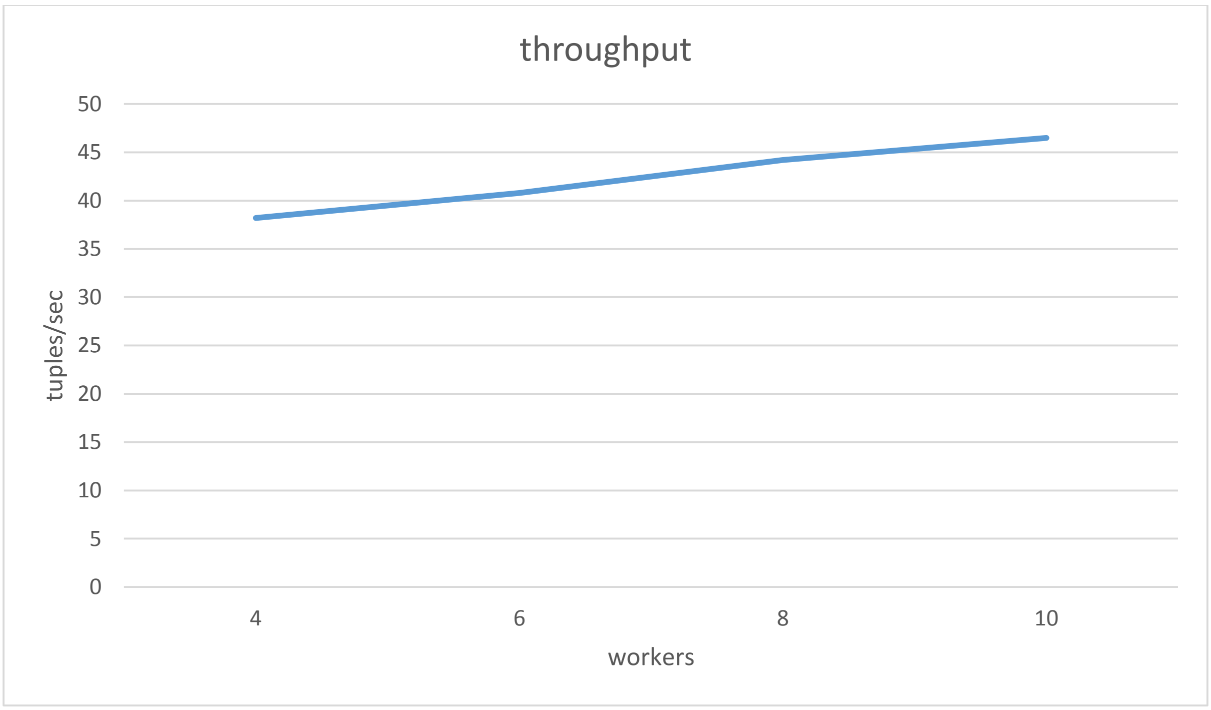


Figure 4.1.1.2 Misra Gries throughput increasing number of workers. Rate of executed tuples/sec is increased as number of workers is increased.

Executing the second category experiments we use 6 workers modifying the parallelism level each time we run Misra Gries topology. The results are shown below,

	workers	Spout Executors	Spout Tasks	Bolt Executors	Bolt Tasks	Aggr. Executors	Aggr. Tasks	Rate (tuples/sec)
1	6	2	4	4	8	2	4	30
2	6	1	2	8	16	2	4	37
3	6	2	4	8	16	2	4	47
4	6	2	4	8	16	4	8	49
5	6	4	8	8	16	4	8	67
6	6	2	4	16	32	2	4	74

- Observing lines (3, 4) we see that although we increase number of Aggregator Bolt's executors the rate has no significant difference. That means that the "heavy work" isn't in the specific Bolt. But, if we increase Spout's executors (line 5) rate is increased by 35 %.
- Observing lines (2, 3) we see that increasing Spout's executors from 1 to 2, rate is increased approximately by 24%.
- Observing lines (1, 3, 6) we see that decreasing Bolt's executors from 8 to 4 rate is decreased by 44% and increasing Bolt's executors from 8 to 16 rate is increased by 44%.
- Observing lines (1, 6) we see that increasing Bolt's executors from 4 to 16 rate is increased by 85%.

Thus, we come to a conclusion that the most "heavy work" is on Spout's and Bolt's executors. Bolt's tasks execute the actual Job of Misra Gries algorithm. Every time a new tuple is emited to Bolts we apply algorithm in order to handle each element individually. Aggregator Bolt, on the other hand, receives top K batched elements, this is why we don't have a large workload.

4.1.2 Tuning the Performance of Space Saving Topology

Below we show the results of the first category of experiments,

workers	Spout Executors	Spout Tasks	Bolt Executors	Bolt Tasks	Aggr. Executors	Aggr. Tasks	Rate (tuples/sec)
4	2	4	4	8	2	4	35
6	2	4	4	8	2	4	38
8	2	4	4	8	2	4	41
10	2	4	4	8	2	4	44

The number of executors and tasks for all components is the same as the number of worker is increased. As the number of workers increases the rate also increases between 35-44 tuples/sec.

For example, if we consider the fourth case, we are using 8 machines, for Spout we are using 2 executors and 4 tasks, for Bolt we are using 4 executors and 8 tasks and for Aggregator Bolt we are using 2 executors and 4 tasks. That means that we available: 8 executors/10 nodes => *1 thread per machine*. So, if we consider that they are running 2 tasks/executor for Spout, 2 tasks/executor for Bolt and 2 tasks/executor for Aggregator Bolt, the workload is distributed as follows,

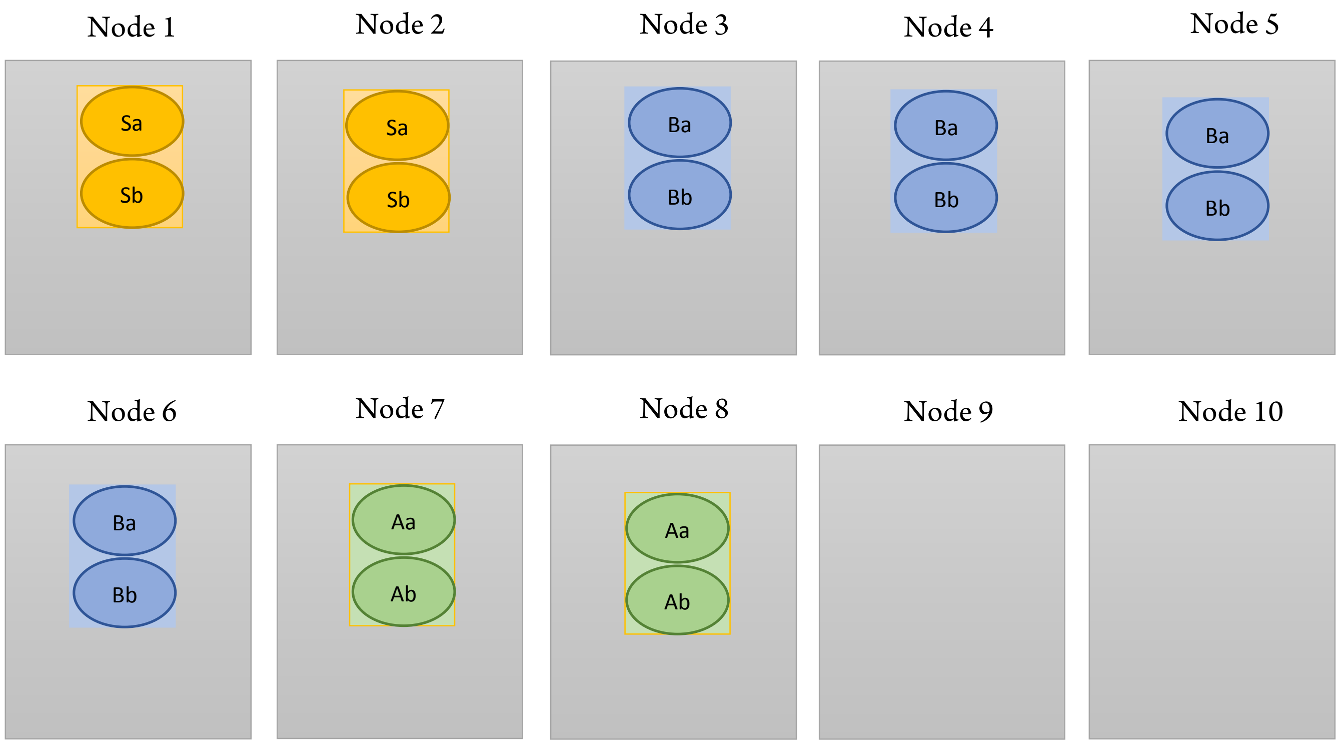


Figure 4.1.2.1 Space Saving workload distribution in cluster's machines. Each machine uses 1 thread. All components need 2 tasks per executor. Spout in yellow has 2 executors running in 2 machines, Bolt in blue has 4 executors running in 4 machines and Aggregator Bolt has 2 executors running in 2 machines. Each machine except the last two has the same number of threads as well as the same amount of work to execute. We notice that two machines have no jobs to execute.

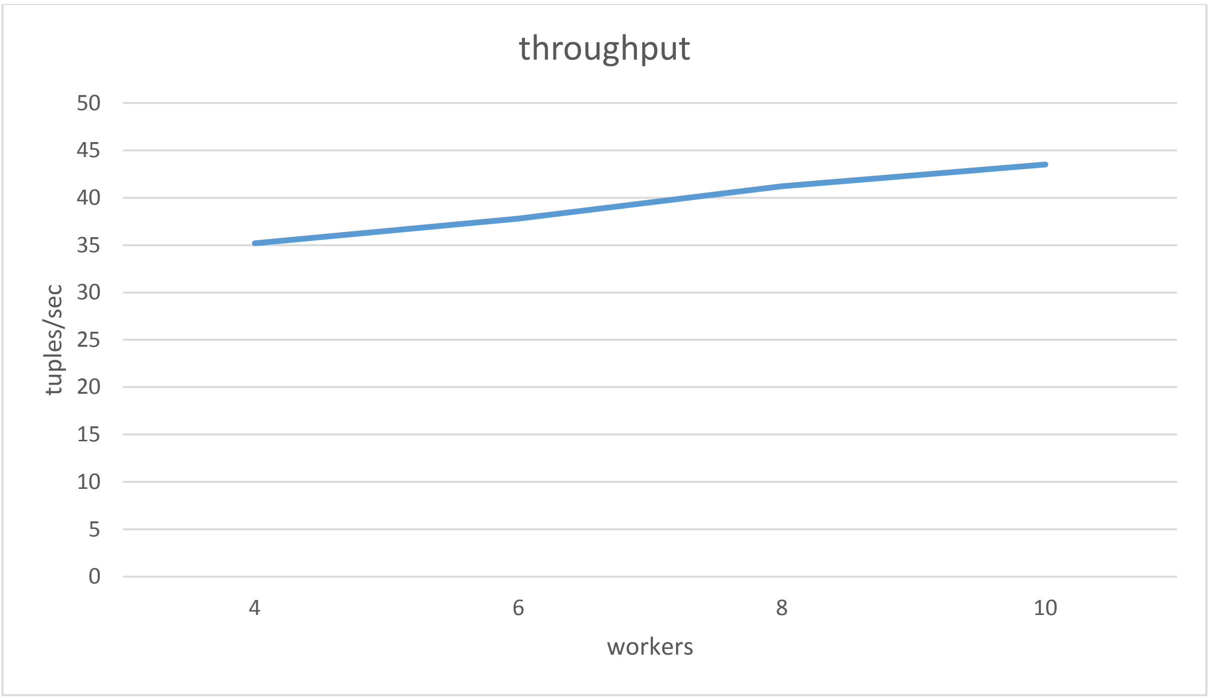


Figure 4.1.2.2 Space Saving throughput increasing number of workers. Rate of executed tuples/sec is increased as number of workers is increased.

Executing the second category experiments we use 6 workers modifying the parallelism level each time we run Space Saving topology. The results are shown below,

	workers	Spout Executors	Spout Tasks	Bolt Executors	Bolt Tasks	Aggr. Executors	Aggr. Tasks	Rate (tuples/sec)
1	6	2	4	4	8	2	4	31
2	6	1	2	8	16	2	4	37
3	6	2	4	8	16	2	4	49
4	6	2	4	8	16	4	8	50
5	6	4	8	8	16	4	8	67
6	6	2	4	16	32	2	4	73

- Observing lines (3, 4) we see that although we increase number of Aggregator Bolt's executors the rate has no significant difference. That means that the "heavy work" isn't in the specific Bolt. But, if we increase Spout's executors (line 5) rate is increased by 31 %.
- Observing lines (2, 3) we see that increasing Spout's executors from 1 to 2, rate is increased by 28 %.
- Observing lines (1, 3, 6) we see that decreasing Bolt's executors from 8 to 4 rate is decreased by 45 % and increasing Bolt's executors from 8 to 16 rate is increased by 39 %.

- Observing lines (1, 6) we see that increasing Bolt's executors from 4 to 16 rate is increased by 81%.

Thus, we come to a conclusion that the most "heavy work" is on Spout's and Bolt's executors. Bolt's tasks execute the actual Job of Space Saving algorithm. Every time a new tuple is emited to Bolts we apply algorithm in order to handle each element individually. Aggregator Bolt, on the other hand, receives top K batched elements, this is why we don't have a large workload.

4.1.3 Tuning the Performance of Lossy Counting Topology

Below we show the results of the first category of experiments,

workers	Spout Executors	Spout Tasks	Bolt Executors	Bolt Tasks	Aggr. Executors	Aggr. Tasks	Rate (tuples/sec)
4	2	4	4	8	2	4	32
6	2	4	4	8	2	4	35
8	2	4	4	8	2	4	38
10	2	4	4	8	2	4	40

As we can see, the number of executors and tasks for all components is the same as the number of worker is increased. As the number of workers increases the rate also increases between 32-40 tuples/sec. We can easily explain this considering that when we increase the number of workers we actually increase the number of worker processes across machines in the cluster. So, when we use more machines the workload is distributed. The most important hint here is the fact that for each component we set number of tasks multiple to number of executors. That means that each machine has the same number of threads and roughly the same amount of work.

For example, if we consider the first case, we are using 4 machines, for Spout we are using 2 executors and 4 tasks, for Bolt we are using 4 executors and 8 tasks and for Aggregator Bolt we are using 2 executors and 4 tasks. That means that we available: 8 executors/4 nodes => *2 threads per machine*. So, if we consider that they are running 2 tasks/executor for Spout, 2 tasks/executor for Bolt and 2 tasks/executor for Aggregator Bolt, the workload is distributed as follows,

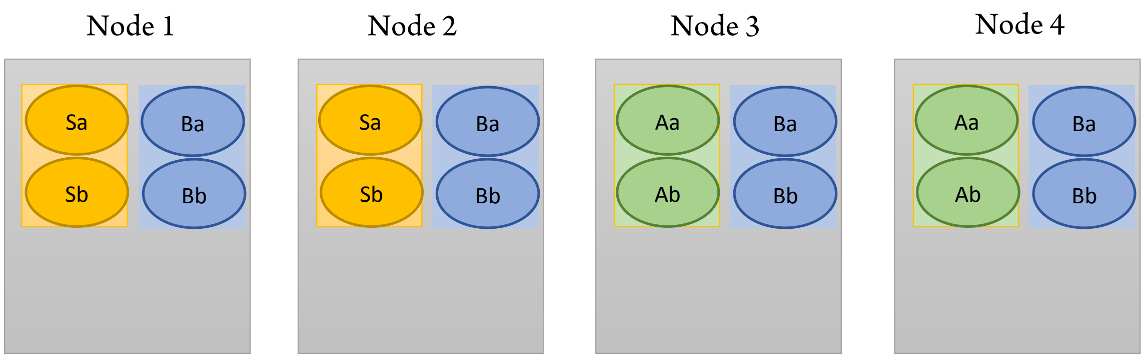


Figure 4.1.3.1 Lossy-Counting workload distribution in cluster's machines. Each machine uses 2 threads. All components need 2 tasks per executor. Spout in yellow has 2 executors running in 2 machines, Bolt in blue has 4 executors running in 4 machines and Aggregator Bolt has 2 executors running in 2 machines. Each machine has the same number of threads as well as the same amount of work to execute.

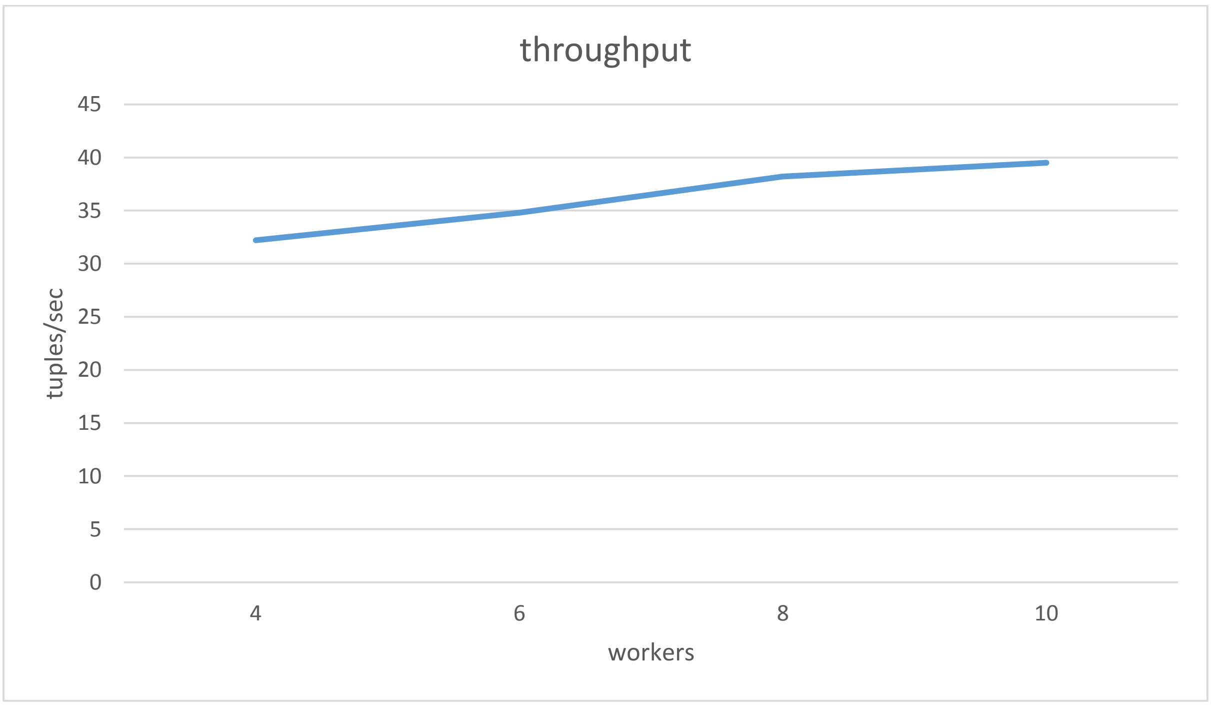


Figure 4.1.3.2 Lossy-Counting throughput increasing number of workers. Rate of executed tuples/sec is increased as number of workers is increased.

Executing the second category experiments we use 6 workers modifying the parallelism level each time we run Lossy Counting topology. The results are shown below,

	workers	Spout Executors	Spout Tasks	Bolt Executors	Bolt Tasks	Aggr. Executors	Aggr. Tasks	Rate (tuples/sec)
1	6	2	4	4	8	2	4	29
2	6	1	2	8	16	2	4	36
3	6	2	4	8	16	2	4	49
4	6	2	4	8	16	4	8	49
5	6	4	8	8	16	4	8	68
6	6	2	4	16	32	2	4	73

- Observing lines (3, 4) we see that although we increase number of Aggregator Bolt's executors the rate is the same. That means that the "heavy work" isn't in the specific Bolt. But, if we increase Spout's executors (line 5) rate is increased by 32.5 %.
- Observing lines (2, 3) we see that increasing Spout's executors from 1 to 2, rate is increased by 30.5%.
- Observing lines (1, 3, 6) we see that decreasing Bolt's executors from 8 to 4 rate is decreased by 51% and increasing Bolt's executors from 8 to 16 rate is increased by 40%.
- Observing lines (1, 6) we see that increasing Bolt's executors from 4 to 16 rate is increased by 86%.

Thus, we come to a conclusion that the most "heavy work" is on Spout's and Bolt's executors. Bolt's tasks execute the actual Job of Lossy Counting algorithm. Every time a new tuple is emited to Bolts we apply algorithm in order to handle each element individually. Aggregator Bolt, on the other hand, receives top K batched elements, this is why we don't have a large workload.

4.1.4 Tuning the Performance of Sticky Sampling Topology

Below we show the results of the first category of experiments,

workers	Spout Executors	Spout Tasks	Bolt Executors	Bolt Tasks	Aggr. Executors	Aggr. Tasks	Rate (tuples/sec)
4	2	4	4	8	2	4	33
6	2	4	4	8	2	4	37
8	2	4	4	8	2	4	40
10	2	4	4	8	2	4	43

As we can see, the number of executors and tasks for all components is the same as the number of worker is increased. As the number of workers increases the rate also increases between 33-43 tuples/sec.

For example, if we consider the second case, we are using 4 machines, for Spout we are using 2 executors and 4 tasks, for Bolt we are using 4 executors and 8 tasks and for Aggregator Bolt we are using 2 executors and 4 tasks. That means that we available: 8 executors/6 nodes => *2 threads per machine*. So, if we consider that they are running 2 tasks/executor for Spout, 2 tasks/executor for Bolt and 2 tasks/executor for Aggregator Bolt, the workload is distributed as follows,

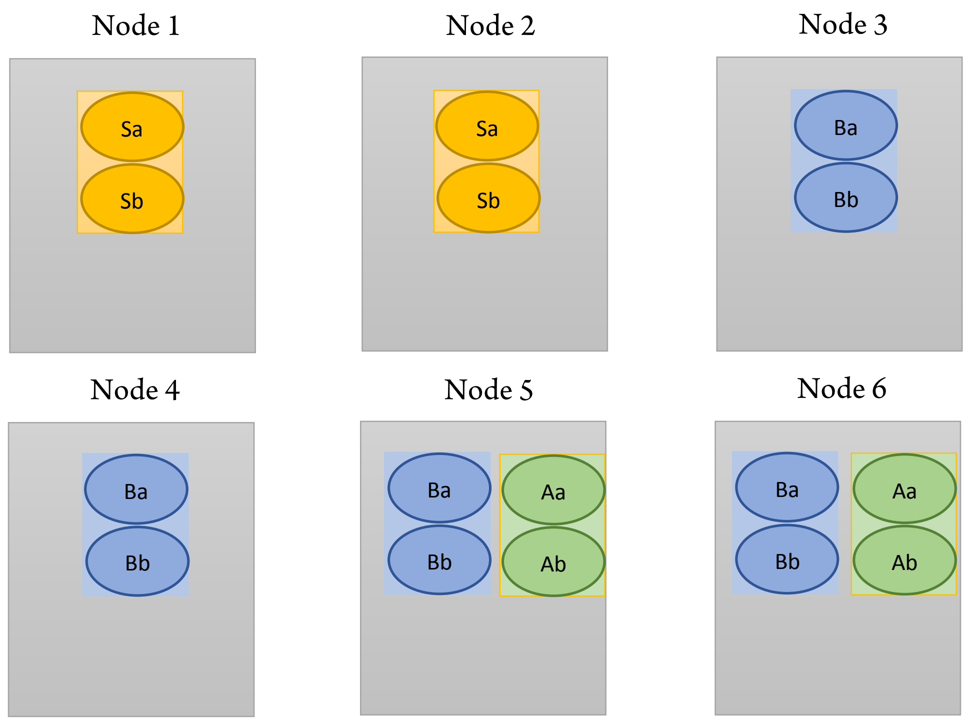


Figure 4.1.4.1 Sticky Sampling workload distribution in cluster's machines. Each machine uses 2 threads. All components need 2 tasks per executor. Spout in yellow has 2 executors running in 2 machines, Bolt in blue has 4 executors running in 4 machines and Aggregator Bolt has 2 executors

running in 2 machines. Each machine, except the last two, has the same number of threads as well as the same amount of work to execute.

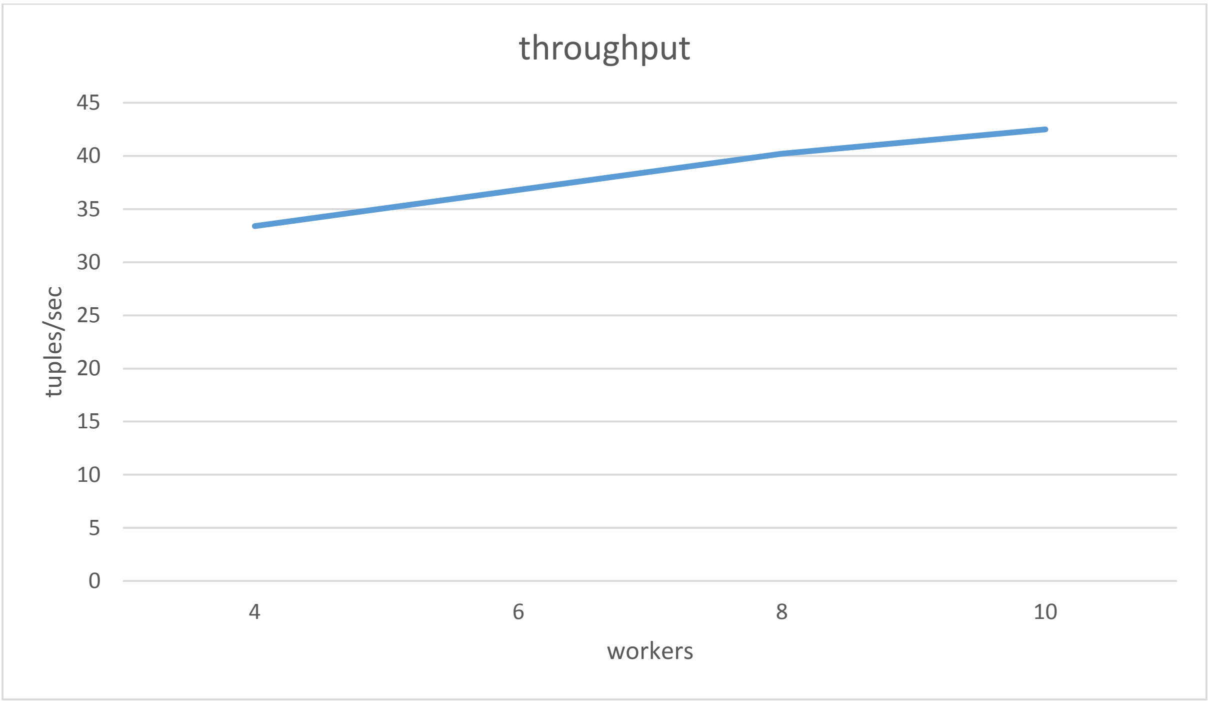


Figure 4.1.4.2 Sticky Sampling throughput increasing number of workers. Rate of executed tuples/sec is increased as number of workers is increased.

Executing the second category experiments we use 6 workers modifying the parallelism level each time we run Sticky Sampling topology. The results are shown below,

	workers	Spout Executors	Spout Tasks	Bolt Executors	Bolt Tasks	Aggr. Executors	Aggr. Tasks	Rate (tuples/sec)
1	6	2	4	4	8	2	4	28
2	6	1	2	8	16	2	4	36
3	6	2	4	8	16	2	4	48
4	6	2	4	8	16	4	8	50
5	6	4	8	8	16	4	8	55
6	6	2	4	16	32	2	4	67

- Observing lines (3, 4) we see that although we increase number of Aggregator Bolt's executors the rate is almost the same. That means that the "heavy work" isn't in the specific Bolt. But, if we increase Spout's executors (line 5) rate is increased by 14 %.
- Observing lines (2, 3) we see that increasing Spout's executors from 1 to 2, rate is increased by 28 %.
- Observing lines (1, 3, 6) we see that decreasing Bolt's executors from 8 to 4 rate is decreased by 53% and increasing Bolt's executors from 8 to 16 rate is increased by 33 %.
- Observing lines (1, 6) we see that increasing Bolt's executors from 4 to 16 rate is increased by 82%.

Thus, we come to a conclusion that the most "heavy work" is on Spout's and Bolt's executors. Bolt's tasks execute the actual Job of Sticky Sampling algorithm. Every time a new tuple is emited to Bolts we apply algorithm in order to handle each element individually. Aggregator Bolt, on the other hand, receives top K batched elements, this is why we don't have a large workload.

4.2 Tuning the Performance of Algorithms in Akka

In order to compare Storm and Akka frameworks we have to follow the same categories of experiments in Akka framework. In the first category of experiments we increase the number of available machines on cluster for Lossy Counting and Sticky Sampling algorithms. In the second category of experiments we use a fixed number of machines switching the number of threads that handle the workload of algorithm architecture. In both categories we observe the throughput which mean that we focus on the rate: messages/sec that arrive and processed by Actors.

There are two types of performance problems: *throughput is too slow* which means that the arrival rate of messages is too slow, and the other is that *latency is too slow* which means that each message takes too long to be processed by the processing unit (Actor). We can deal with the first problem increasing the number of available active machines (scaling technique) on cluster, while we deal with the second problem modifying the design patterns of our implementation architecture. In the current diploma thesis we are interested in switching the number of machines in cluster, increasing and decreasing the number of threads that we use to process our messages, in other words the tuples of our data set.

Dealing with the throughput of architecture we have to detect which Actor has a significant workload, which means that "does all the work", we have also to consider how we use threads in order to check how problems arise running an algorithm in multi node cluster.

Firstly we find the critical point when running both Lossy Counting and Sticky Sampling Akka Actors technique in which the mailbox queue of each Actor can handle up to a number of messages, otherwise messages can't be processed by the next Actor. Thus, the last Actor is waiting to process messages and does no jobs at all. This is what we call bottleneck of a system.

In order to specify our metrics we implemented a module in both algorithms, a trait in Scala language that handles the statistics of each Actor's mailbox. Thus, we hold three parameters:

- t1: time parameter in which a message is received in mailbox
- t2: time parameter in which a message was removed from mailbox and handled by another Actor
- t3: time parameter in which a message was processed.

4.2.1 Tuning the Performance of Lossy Counting Architecture

We made the first category of experiments specifying the minimum and maximum threads to use in *application.conf* file while we increase number of processors.

```
worker-dispatcher{

        parallelism-min = 8

        parallelism-factor = 3.0

        parallelism-max = 64 }
```

The above configuration means that we have at least 8 threads in our system, and at most 64 threads. The parallelism factor is used in order to calculate the number of threads from the available processors (for 4, 6, 8, 10 processors). For example for 4 processors we have 4*3.0 = *12 threads*, for 6 processors we have 6*3.0 = *18 threads*, etc.

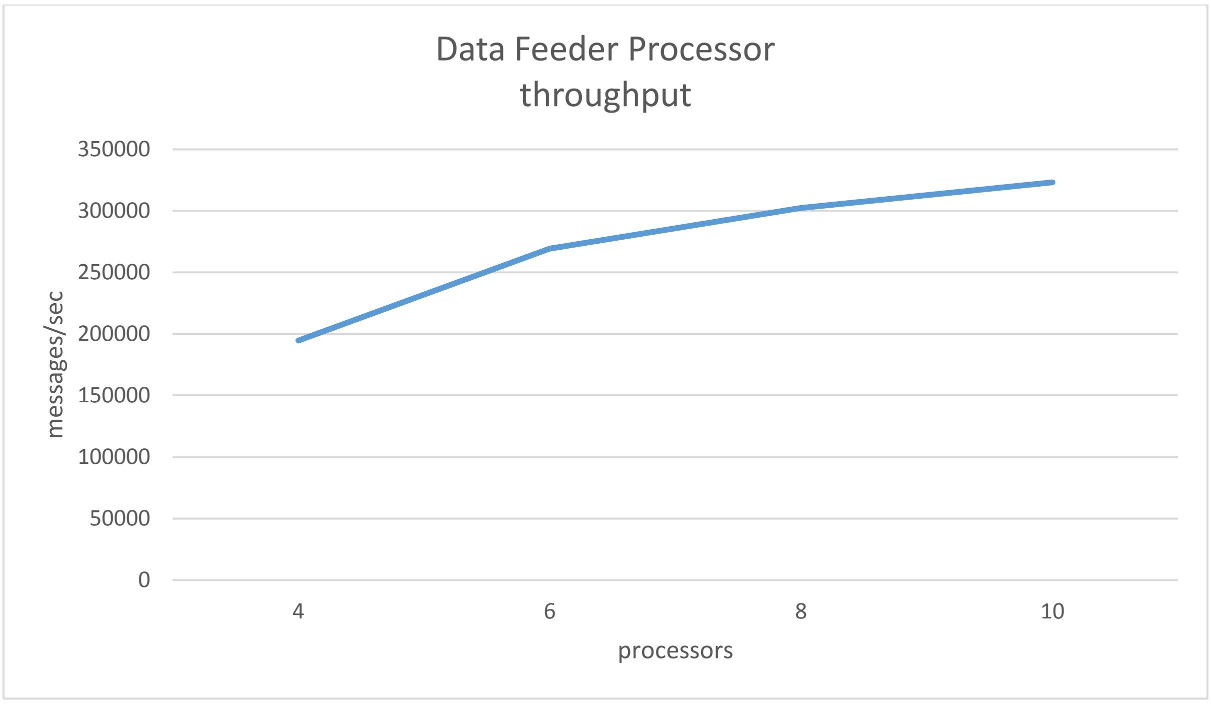
Below we use the following metrics: *rate* of each Actor which represents the tuples/sec that each Actor processes and *utilization* which is the percentage of time in which an Actor is busy processing messages.

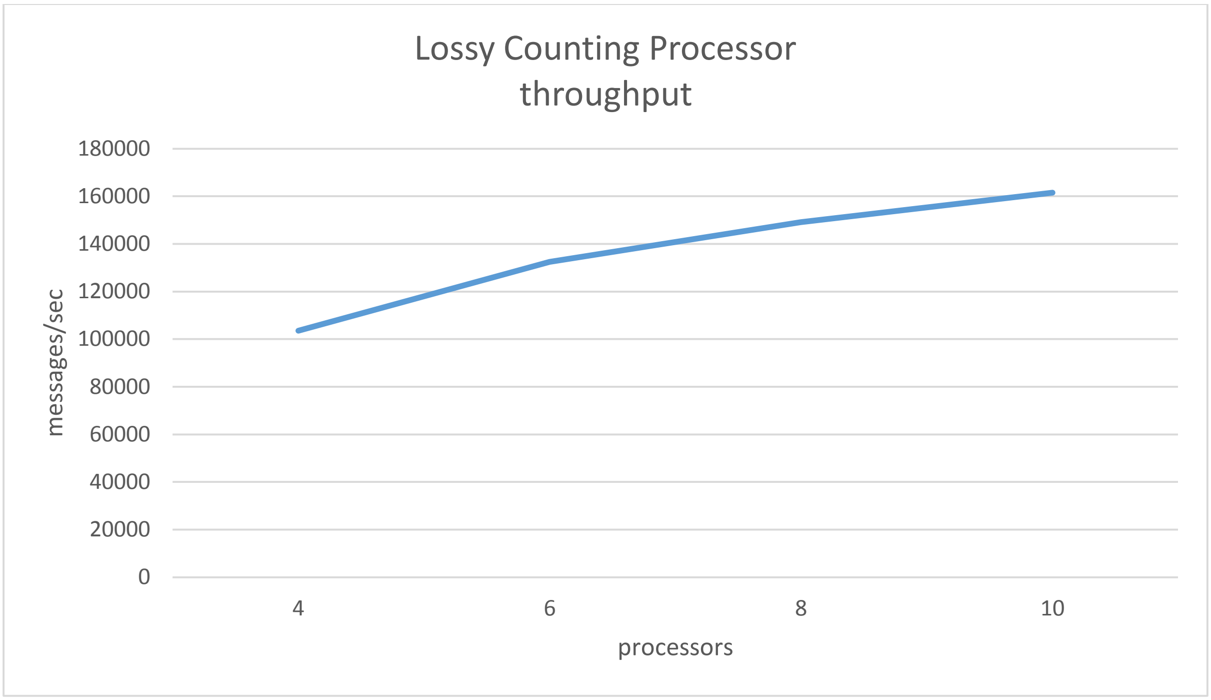
	processors	DataFeeder rate (messages/sec)	DataFeeder utilization (%)	lcProcessor rate (messages /sec)	lcProcessor utilization (%)	lcAggregator rate (messages /sec)	lcAggregator utilization (%)
1	4	194.627	47	103.525	25	49.692	12
2	6	269.165	65	132.512	32	66.256	16
3	8	302.293	73	149.076	36	69.254	18
4	10	322.998	78	161.499	39	78.538	18

Firstly we observe that increasing the number of processors, the number of messages per second also increases. While increasing the number of machines, that is the number of threads, the workload is distributed to more threads.

Beside, observing line 4 the utilization metric for Data Feeder Processor we notice that it reaches 78 % which is very close to 100 %, the bottleneck of topology. The same metric for Lossy Counting Processor as well as for Lossy Counting Aggregator Processor is 39 % and 18 % which is significant lower. That happens because Data Feeder Actor cannot handle the receiving messages correctly, because this Actor doesn't have enough threads in order to process and send them to Lossy Counting Processor which is going to process and send some those messages to Lossy Counting Aggregator Processor. Aggregator Processor has to wait until Lossy Counting Processor has finished before it can process the waiting messages. We come to a conclusion that certainly the workload is in Data Feeder Processor.

Below we observe throughput for each processor while the number of processors increases.





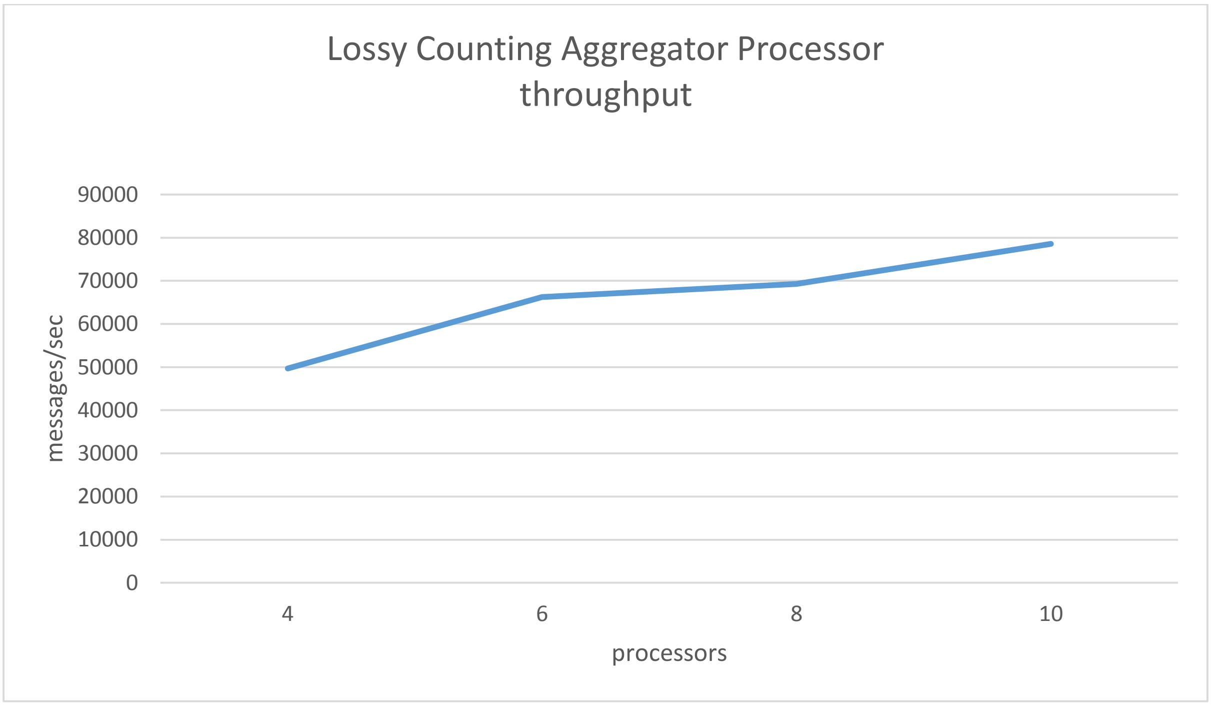


Figure 4.2.1.1 Lossy Counting throughput for each processor increasing number of processors. Rate of executed messages/sec is increased as number of workers is increased.

In order to fix the problem of insufficient number of threads, we execute the second category of experiments, modifying the previous configuration file (application.conf) as follows:

```
worker-dispatcher{
            core-pool-size-min = 8
            core-pool-size-factor = 3.0
            core-pool-size-max = 64

            max-pool-size-min = 8
            max-pool-size-factor = 3.0
            max-pool-size-max = 64

            keep-alive-time = 60 sec }
```

At first block we set the minimum and the maximum thread pool size same with the first category of experiments. At second block we set the maximum thread pool size. Core pool size specifies the number of available threads for work, while max pool size defines an outer bound for the available threads. For example, if a new task is submitted and the available threads are less than the core pool size (3.0*4 = 12 threads) a new thread will be created as long as the maximum thread pool size is not exceeded. The minimum, maximum and factor settings for core and max pool size provide a way to dynamically size those pools based on the number of processors available.

With the above configuration Actors share a pool of threads which are dynamically assigned to Actors when the Actors have a big workload, in other words when they have to send many messages (e.g. send the tuples with host in order to calculate their frequency). When a message has been processed threads are returned back to the pool and the Actor is idle until new messages arrive. Thus, the dynamic thread pool increases the size to the number of active workers when the number of workers increases, but decreases when the threads are idle too long. In this way, threads that are unused are cleaned-up in order not to waste resources. For this action, the responsible parameter is keep-alive-time which means that 60 seconds is idle time before a thread will be cleaned up.

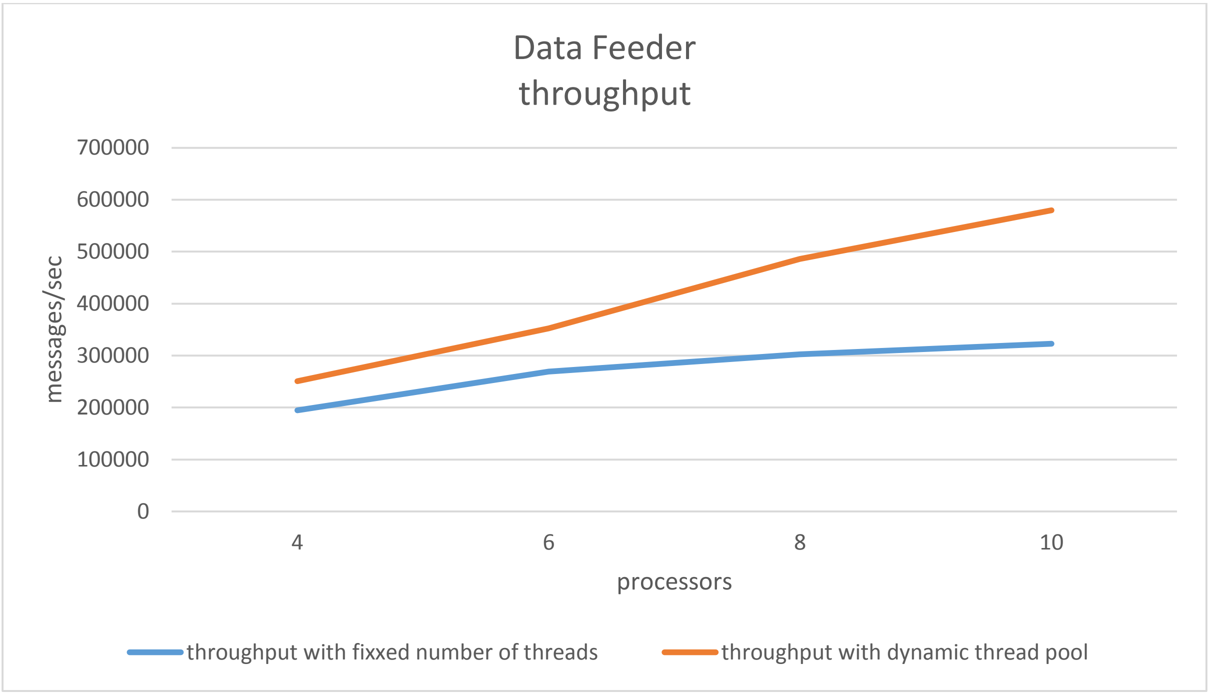
The essential difference between those two experiments is the fact that with dynamic thread pool the Actor that is busy than others "picks" a thread from thread pool when it needs it without waiting for an upcoming thread.

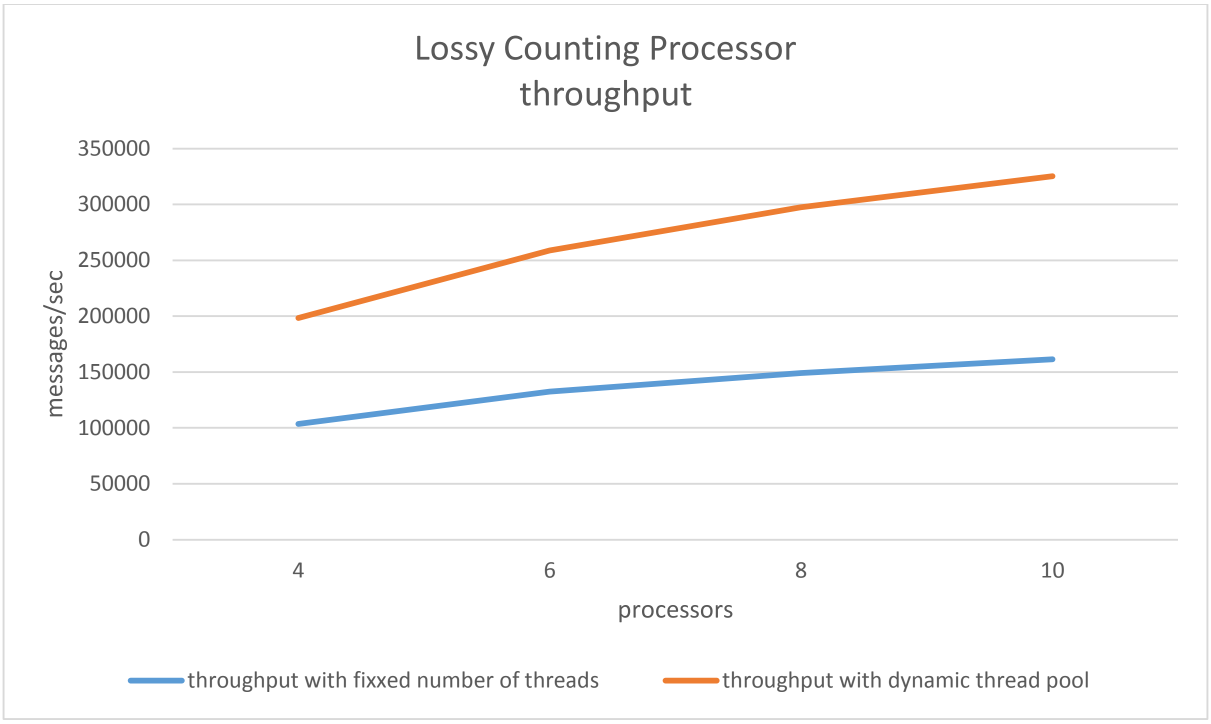
	processors	DataFeeder rate (messages/sec)	DataFeeder utilization (%)	lcProcessor rate (messages /sec)	lcProcessor utilization (%)	lcAggregator rate (messages /sec)	lcAggregator utilization (%)
1	4	250.659	22	198.254	15	82.458	6
2	6	352.485	32	258.781	26	131.351	12.5
3	8	486.325	36.5	297.473	13	148.845	5
4	10	579.548	40	325.365	19	154.746	7

Firstly we observe that increasing the number of processors, the number of messages per second is increased.

Beyond this, we observe that the utilization of Data Feeder, lossy counting and lossy counting aggregator processors has decreased, while the rate has been increased. That means that each processor processes much more messages than the previous experiment while it is less busy. The load of the processors changes during the operation time. Data feeder processes the tuples from data set, which means that firstly has many messages to process, but after a while the workload of lossy counting processor increases drastically since the specific processor is responsible for applying the logic of the algorithm. In this point of time, the dynamic thread pool increases creating more and more threads in order to handle the heavy workload. While lossy counting processor has processed the top K items calculating their corresponding frequency the workload is now on lossy counting aggregator processor which means that threads created for the last processor, while the threads for lossy counting processor are idle until they cleaned-up (60 seconds). Thus, we save more resources than the previous experiment, while we have better throughput, better performance of the whole system.

Below we can observe throughput for each processor, comparing throughput of previous experiment,





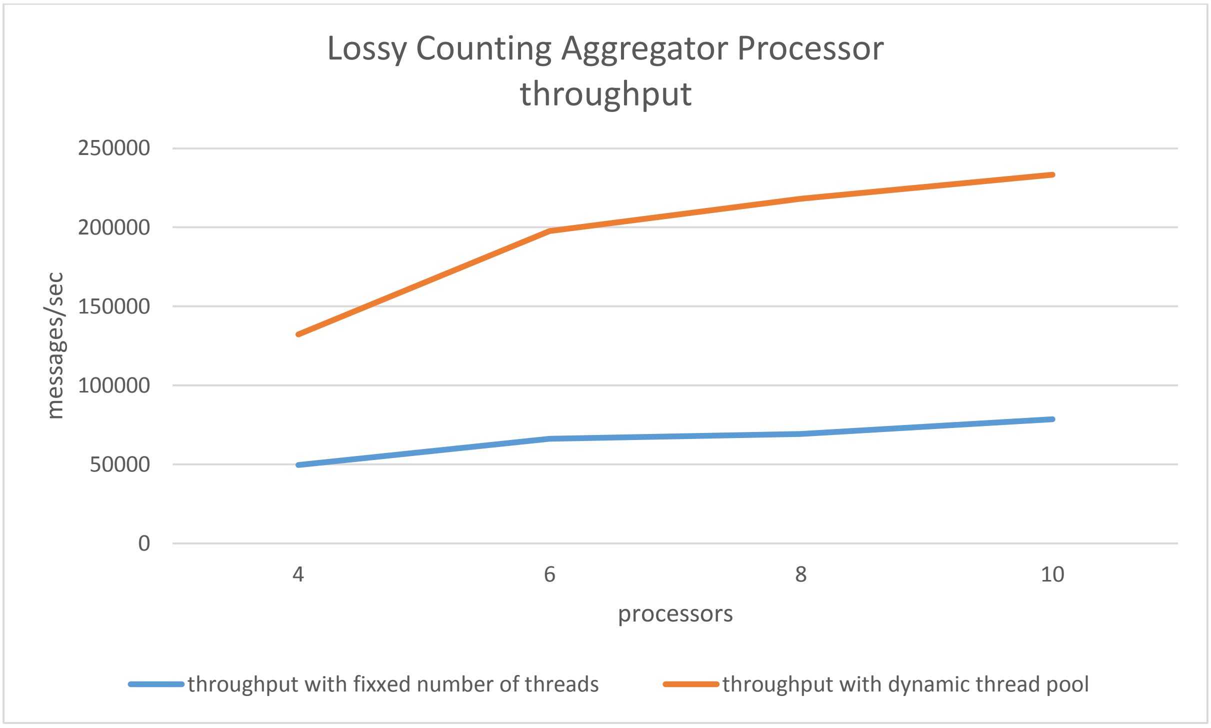


Figure 4.2.1.2 Lossy Counting throughput for each processor using a fixed number of threads and a dynamic thread pool size. Rate of executed messages/sec when we use a dynamic number of threads is greater than a fixed number of threads.

4.2.2 Tuning the Performance of Sticky Sampling Architecture

Following the same steps with Lossy Counting algorithm in Akka framework, we made the first category of experiments specifying the minimum and maximum threads to use in *application.conf* file while we increase number of processors.

```
worker-dispatcher{
        parallelism-min = 8
        parallelism-factor = 3.0
        parallelism-max = 64 }
```

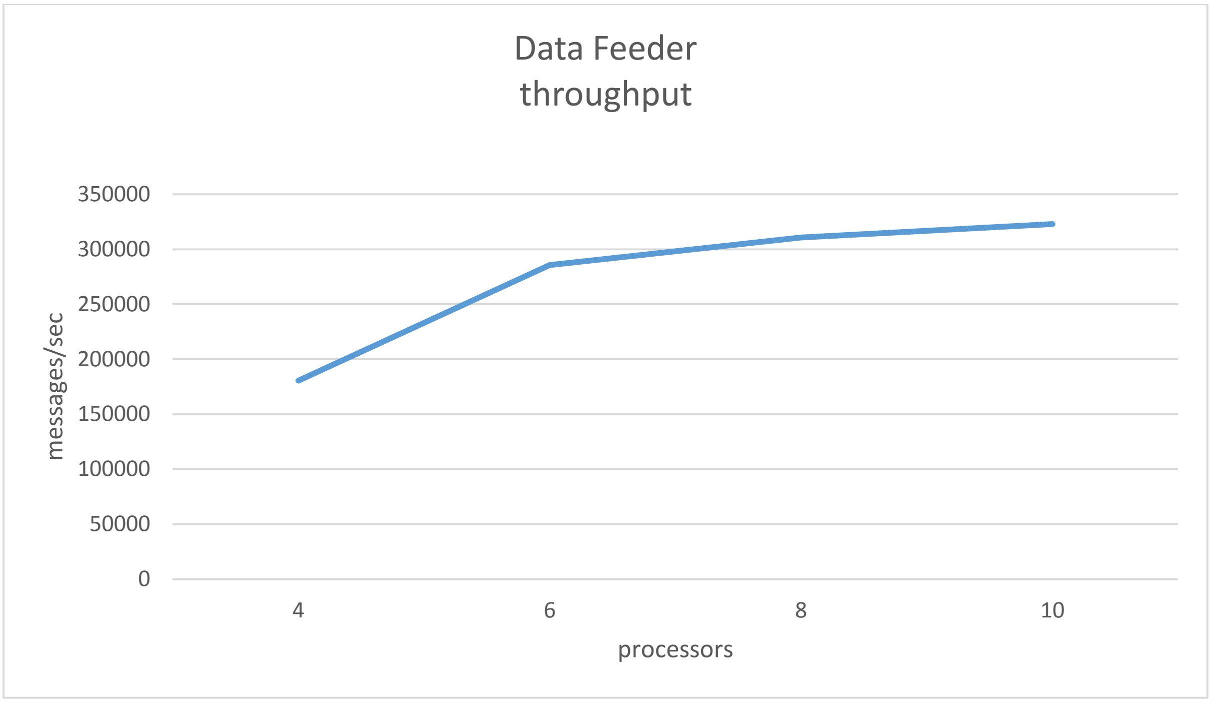
Below we use the following metrics: *rate* of each Actor which represents the tuples/sec that each Actor processes and *utilization* which is the percentage of time in which an Actor is busy processing messages.

	processors	DataFeeder rate (messages/sec)	DataFeeder utilization (%)	lcProcessor rate (messages /sec)	lcProcessor utilization (%)	lcAggregator rate (messages /sec)	lcAggregator utilization (%)
1	4	180.582	66	132.512	32	62.115	15
2	6	285.729	69	140.794	34	74.538	18
3	8	310.575	75	144.935	35	95.243	23
4	10	322.998	78	157.358	38	103.525	25

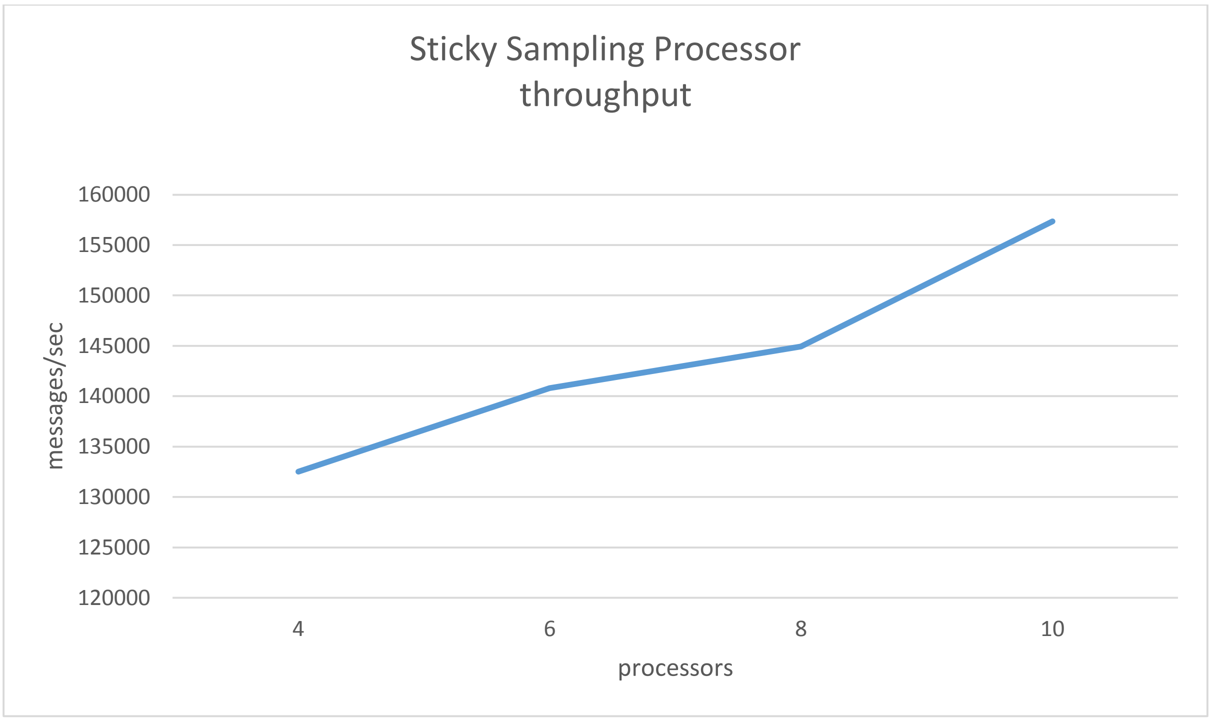
Also in Sticky Sampling algorithm we observe that increasing the number of processors, the number of messages per second also increases. While increasing the number of machines, that is the number of threads, the workload is distributed to more threads.

Beside, observing line 4 the utilization metric for Data Feeder Processor we notice that it reaches 78 % which is very close to 100 %, the bottleneck of topology. The same metric for Sticky Sampling Processor as well as for Lossy Counting Aggregator Processor is 38 % and 25 % which is significant lower.

Below we observe throughput for each processor while the number of processors increases.



Data Feeder throughput
messages/sec
350000
300000
250000
200000
150000
100000
50000
0
4
6
8
10
processors



Sticky Sampling Processor throughput
messages/sec
160000
155000
150000
145000
140000
135000
130000
125000
120000
4
6
8
10
processors

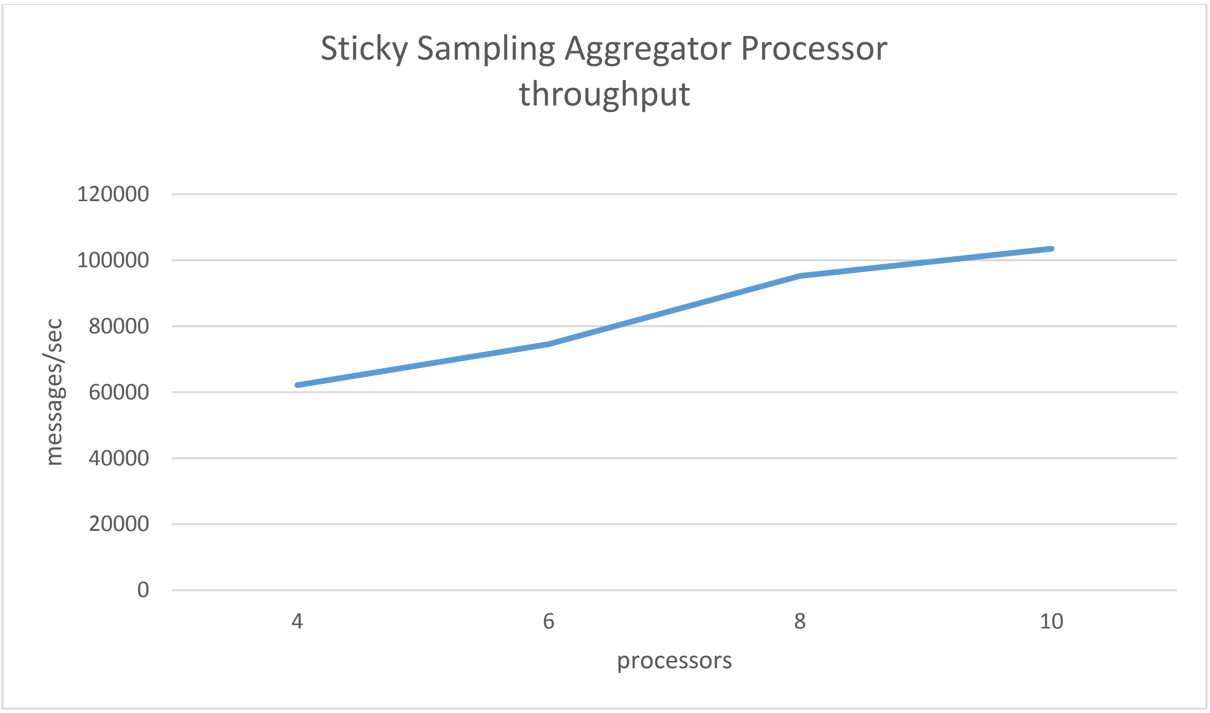


Figure 4.2.2.1 Sticky Sampling throughput for each processor increasing number of processors. Rate of executed messages/sec is increased as number of workers is increased.

In order to fix the problem of insufficient number of threads, we execute the second category of experiments, modifying the previous configuration file (application.conf) as follows:

```
worker-dispatcher{
        core-pool-size-min = 8
        core-pool-size-factor = 3.0
        core-pool-size-max = 64

        max-pool-size-min = 8
        max-pool-size-factor = 3.0
        max-pool-size-max = 64

        keep-alive-time = 60 sec }
```

Dynamic thread pool increases the size to the number of active workers when the number of workers increases, but decreases when the threads are idle too long. In this way, threads that are unused are cleaned-up in order not to waste resources. For this action, the responsible parameter is keep-alive-time which means that 60 seconds is idle time before a thread will be cleaned up.

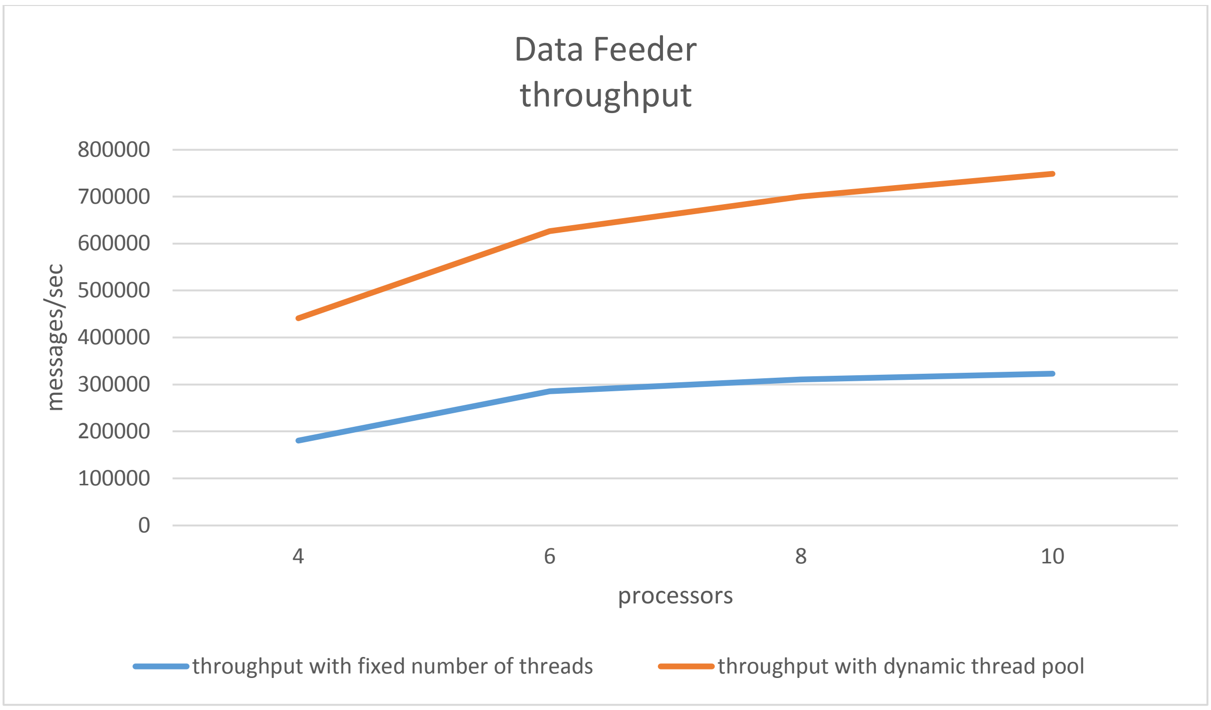
The essential difference between those two experiments is the fact that with dynamic thread pool the Actor that is busy than others "picks" a thread from thread pool when it needs it without waiting for an upcoming thread.

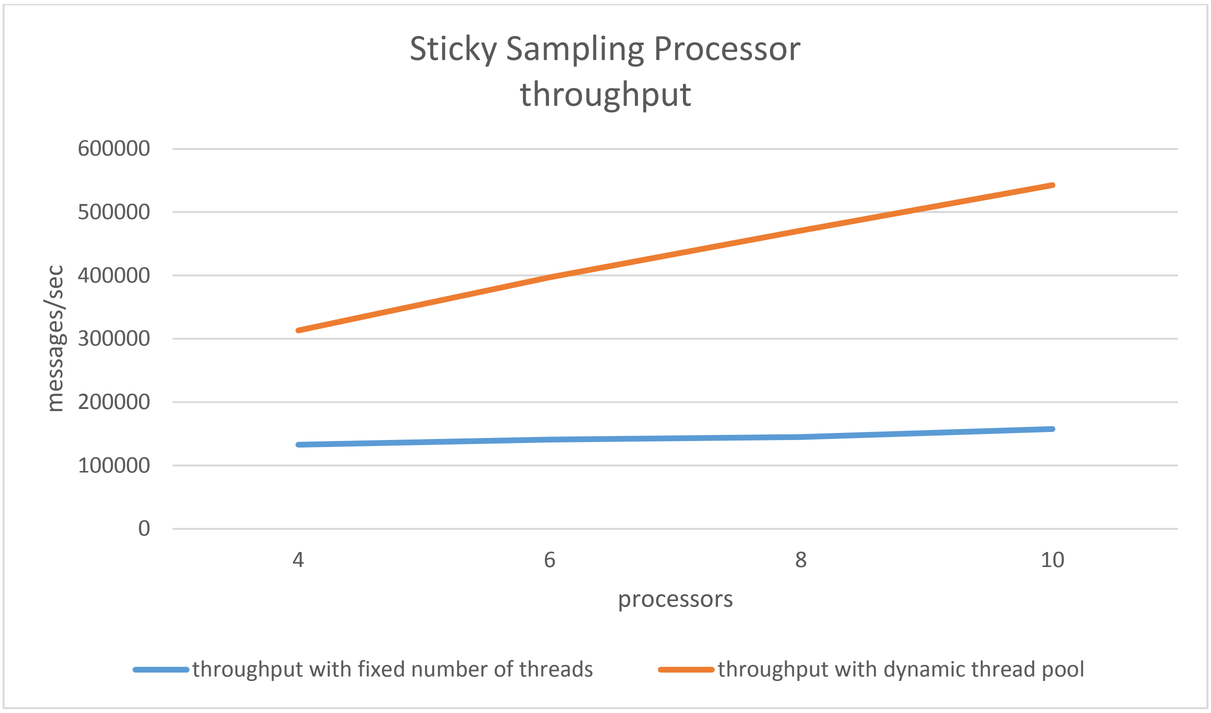
	processors	DataFeeder rate (messages/sec)	DataFeeder utilization (%)	lcProcessor rate (messages /sec)	lcProcessor utilization (%)	lcAggregator rate (messages /sec)	lcAggregator utilization (%)
1	4	260.587	33	180.568	16	100.547	7
2	6	340.612	35	256.345	18	134.587	10
3	8	389.452	38	325.785	17	159.785	12
4	10	425.815	40	385.421	22	184568	14

Firstly we observe that increasing the number of processors, the number of messages per second is increased.

Beyond this, we observe that the utilization of Data Feeder, Sticky Sampling and Sticky Sampling aggregator processors has decreased, while the rate has been increased. That means that each processor processes much more messages than the previous experiment while it is less busy. The load of the processors changes during the operation time. Data feeder processes the tuples from data set, which means that firstly has many messages to process, but after a while the workload of Sticky Sampling processor increases drastically since the specific processor is responsible for applying the logic of the algorithm. In this point of time, the dynamic thread pool increases creating more and more threads in order to handle the heavy workload. While Sticky Sampling processor has processed the top K items calculating their corresponding frequency the workload is now on Sticky Sampling aggregator processor which means that threads created for the last processor, while the threads for Sticky Sampling processor are idle until they cleaned-up (60 seconds). Thus, we save more resources than the previous experiment, while we have better throughput, better performance of the whole system.

Below we can observe throughput for each processor, comparing throughput of previous experiment,


Data Feeder
throughput
messages/sec
800000
700000
600000
500000
400000
300000
200000
100000
0
4
6
8
10
processors
throughput with fixed number of threads
throughput with dynamic thread pool


Sticky Sampling Processor
throughput
messages/sec
600000
500000
400000
300000
200000
100000
0
4
6
8
10
processors
throughput with fixed number of threads
throughput with dynamic thread pool

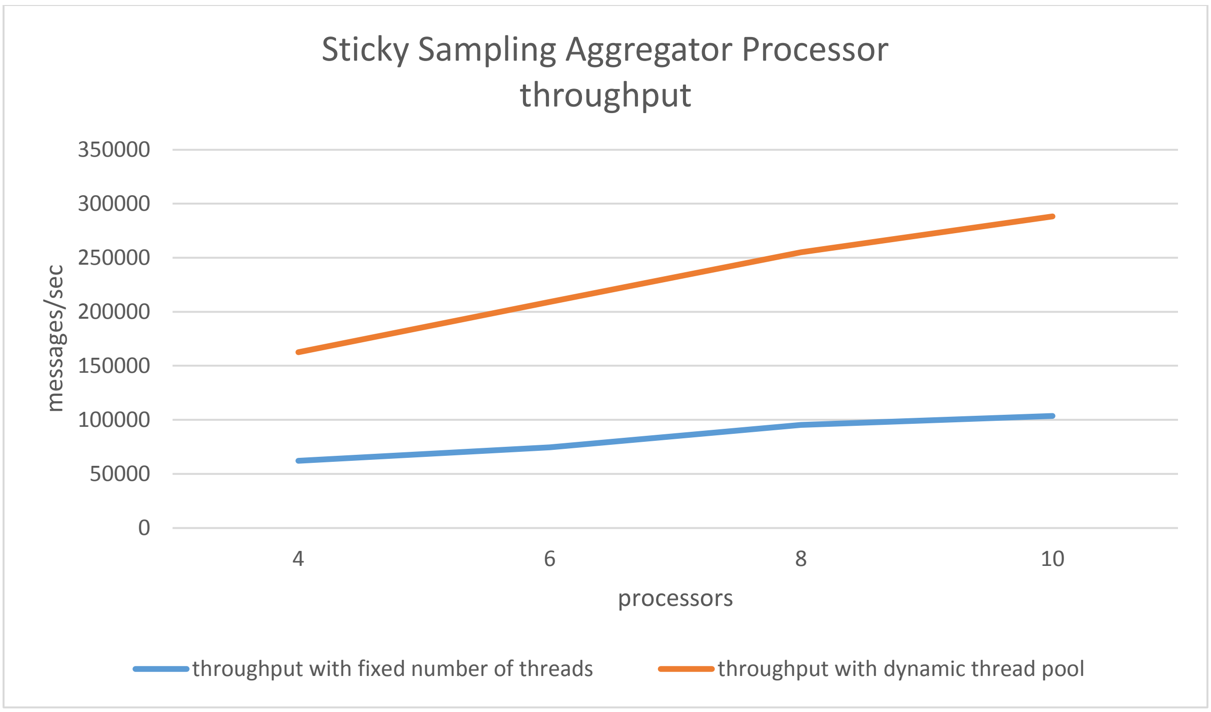


Figure 4.2.2.2 Sticky Sampling throughput for each processor using a fixed number of threads and a dynamic thread pool size. Rate of executed messages/sec when we use a dynamic number of threads is greater than a fixed number of threads.

Conclusion

We come to the conclusion that the tuning of the performance of an algorithm in both frameworks is a trade-off. Giving one task more resources means that other tasks get less. We can optimize the performance of an algorithm in two major steps. Firstly, we can detect the components, such as Spouts, Bolts, Actors that handle the heavy workload. Secondly, we can direct the resources that are available in the cluster to those components. That means that we direct resources to the most critical tasks for our system from tasks that are less critical and can wait. We can achieve this, assigning threads to those tasks in order to execute them. There are two considerable approaches in order to accomplish the assignment of threads in tasks. The first one is changing the *thread pool statically* and the other one is using a *dynamic thread pool* size.

When we change the thread pool of a system statically, that actually means that we should increase the number of threads up to the maximum number of threads that are available in each node of a cluster. If we increase the number of threads exceeding that threshold, the performance will decrease. So, there is always an optimum number of threads for each node in cluster. We can figure out when to increase the number of threads observing the parameter *utilization*, explained in sections 4.2.1, 4.2.2 in Akka implementation. This parameter indicates the percentage of the time a node is busy processing messages. For example, when utilization of a process is 30%, the process is processing 30% of the time and is idle 70% of the time. When utilization parameter amount approximately to 80% then we assume that our system has reached bottleneck. So, even if we increase the number of threads wouldn't help increasing the performance of the system. This is exactly what we did in Storm and Akka experiments when we held a fixed number of workers, while rebalancing the whole topology (for Storm) or Architecture (for Akka) directing the available resources from one component to another.

In static thread pool we are able to direct resources from one tasks to another because we know how many available workers we have. But when the number of workers depends on the system's workload, we don't know how many threads we need. Therefore, we used a dynamic thread pool size in Akka framework. We changed the *application.conf* file many times in order to observe the minimum and the maximum number of threads that are assigned to tasks. When we create a small thread pool our performance is slow, while when we create and use a big thread pool we are wasting resources. This is again a trade-off between resources and performance. But when the size of thread pool is low or stable and sometimes increases drastically, a dynamic thread pool can improve the performance without wasting resources. The dynamic thread pool increases the size to the number of active workers when the number of workers increases, but decreases when the threads are idle for too long. In this way, unused threads are cleaned up. In Storm cluster, we changed dynamically the size of thread pool rebalancing the algorithms' topology in Storm UI when we noticed the system's bottleneck. In contrast, in Akka cluster we just configured the size of threads in the configuration file in order to let Akka framework to handle this for us. Thus, in Akka cluster was much easier for us to increase the performance of the system.

Monitoring the system's performance was much easier in Akka cluster than the Storm cluster. Another crucial difference between those two frameworks is the fact that in Storm framework topology structure is predefined by the user. In our case, we have one Spout to read the data from the data stream with HTTP requests emit them in Bolt which is responsible for the update of items' counter. Finally we have another Bolt which takes as input batch items with their counters extracting the top K items. This topology structure cannot be changed when we run the topology in Storm cluster. In contrast, in Akka framework we have a data feeder Actor which does the exact same job as Spout, a data processor which is corresponding to Bolt responsible for the update of items' frequency. Finally we have an aggregator processor Actor in order to extract the top K items. In Akka Actors model, where Actors communicate with messages, this structure changes over time since the message sending is asynchronous. Actors may maintain state and may change their behavior as needed.

Storm and Akka are both real-time, distributed and fault tolerant frameworks. Both guarantee that jobs will be executed but in different way. If something goes wrong, i.e., some tuples (or messages) cannot be emitted to other components (or Actors), then tasks are executed again. In this occasion, in Storm framework the master node is responsible for re-assigning the tasks to worker nodes. In contrast, Akka framework, message sending is asynchronous. So, in this occasion an Actor waits for the tuple in order to process it. In general, Akka Actors model is more lightweight than Storm, since it can send even a serialized Java Object as a message. Akka Actor framework is built with Scala libraries while Storm is built with Java libraries.

References

[1] Approximate Frequency Counts over Data Streams. Gurmeet Singh Manku, Rajeev Motwani, Standford University

[2] Efficient Computation of Frequent and Top-k Elements in Data Streams. Ahmed Metwally, Divyakant Agrawal and Amr El Abbadi, Department of Computer Science (University of California, Santa Barbara).

[3] Finding Frequent Items in Data Streams. Graham Cormode, Marios Hadjieleftheriou, AT & T Labs-Research, Florham Park, NJ.

[4] Frequent Itemset Discovery. Springer, Marko Salmenkivi. The Market-Basket Model.

[5] Apache Storm Reference Guide.

[6] Akka in Action, Raymond Roestenburg, Rob Bakker, Rob Williams, MEAP (Manning Early Access Program) Edition.

[7] Akka Manifesto, the Reactive Manifesto, Reactive Streams.

[8] HDFS Architecture Guide.

[9] Akka vs Storm, JBossDeveloper blog, article by Adam Warski.

[10] Mixing C and Java for High Performance Computing, Nazario Irizarry.

[11] Data Set, http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html