J Grid Computing
DOI 10.1007/s10723-013-9277-0

A Multi-Domain Architecture for Mining Frequent Items and Itemsets from Distributed Data Streams

Eugenio Cesario · Carlo Mastroianni · Domenico Talia

Received: 9 October 2012 / Accepted: 17 September 2013

Abstract Real-time analysis of distributed data streams is a challenging task since it requires scalable solutions to handle streams of data that are generated very rapidly by multiple sources. This paper presents the design and the implementation of an architecture for the analysis of data streams in distributed environments. In particular, data stream analysis has been carried out for the computation of items and itemsets that exceed a frequency threshold. The mining approach is hybrid, that is, frequent items are calculated with a single pass, using a sketch algorithm, while frequent itemsets are calculated by a further multi-pass analysis. The architecture combines parallel and distributed processing to keep the pace with the rate of distributed data streams. In order to keep computation close to data, miners are distributed among the domains where data streams are generated. The paper reports the experimental results obtained with a prototype of the architecture, tested on a Grid composed of three domains each one handling a data stream.

Keywords Distributed data mining · Frequent items · Frequent itemsets · Grid · Stream mining

1 Introduction

Mining data streams is a very important research topic and has recently attracted a lot of attention, because in many cases data is generated by external sources so rapidly that it may be unfeasible to store and analyze it offline. Moreover, in some cases streams of data must be analyzed in real time to provide information about trends, outlier values or regularities that must be signaled as soon as possible. The need for online computation is a notable challenge with respect to classical data mining algorithms [2, 19]. Important application fields for stream mining are as diverse as financial applications, network monitoring, security problems, telecommunication networks, Web applications, sensor networks, analysis of atmospheric data, etc.

A further difficulty occurs when streams are distributed, and mining models must be derived not only from a single stream, but from multiple and heterogeneous data streams [11]. This scenario can occur in all the application domains

E. Cesario (✉) · C. Mastroianni
ICAR-CNR, Via P. Bucci 41C,
Rende (CS) 87036, Italy
e-mail: cesario@icar.cnr.it

C. Mastroianni
e-mail: mastroianni@icar.cnr.it

D. Talia
ICAR-CNR and DIMES, University of Calabria,
Via P. Bucci 41C, Rende (CS) 87036, Italy
e-mail: talia@dimes.unical.it

Published online: 11 October 2013

mentioned before. For example, in a Content Distribution Network, user requests delivered to a Web system can be forwarded to any of several servers located in different and possibly distant places, in order to serve requests more efficiently and balance the load. In such a context, the analysis of user requests, for example to discover frequent patterns, must be performed with the inspection of data streams detected by different servers. Another notable application field is the analysis of packets processed by routers of an IP network. In any distributed scenario, it is essential that miners are located as close to data sources as possible, in order to limit the overhead of data communication. When there is the need for performing multiple passes on data, the presence of data cachers can help, provided that they are also appropriately distributed.

Sometimes the rate of a single data stream can be so fast that a single computing node can have difficulties to keep the pace with the generation of data. In these cases, it can be useful to sample the data stream instead of processing all the data [15], but of course this can lower the accuracy of derived models, depending on the sampling frequency and the adopted algorithm. A different, or complementary, solution is to partition a data stream among a set of miners, so that each miner processes only a fraction of data. This solution can be achieved by parallelizing the computation over the nodes of a cluster or a high-speed computer network, or can also be implemented by exploiting the multiple CPUs/GPUs offered by modern multicore and manycore machines.

Two important and recurrent problems regarding the analysis of data streams are the computation of *frequent items* and *frequent itemsets* from transactional datasets. The first problem is very popular both for its simplicity and because it is often used as a subroutine for more complex problems. The goal is to find, in a sequence of items, those whose frequency exceeds a specified threshold. When the items are generated in the form of *transactions*—sets of distinct items—it is also useful to discover frequent sets of items. A *k-itemset*, i.e., a set of k distinct items, is said to be frequent if those items concurrently appear in a given fraction of transactions. The discovery of frequent itemsets is essential to cope with many data mining problems, such as the computation of association rules, classification models, data clusters, etc. This task can be severely time consuming, since the number of candidates is combinatorial with their allowed size. The usually adopted technique is to first discover frequent items, and then build candidate itemsets incrementally, exploiting the Apriori property [4], which states that an i-itemset can be frequent only if all of its subsets are also frequent. While there are some proposals in the literature to mine frequent itemsets in a single pass, it is recognized that in the general case, in which the generation rate is fast, it is very difficult to solve the problem without allowing multiple passes on the data stream [22].

The architecture we designed and present in this paper addresses the issues mentioned above by exploiting the following main features:

- it combines the parallel and distributed paradigms, the first one to keep the pace with the rate of a single data stream, by using multiple miners (processors or cores), the second one to cope with the distributed nature of data streams. Miners are distributed among the domains where data streams are generated, in order to keep computation close to data.
- the computation of frequent items is performed through *sketch* algorithms. These algorithms maintain a matrix of counters, and each item of the input stream is associated with a set of counters, one for each row of the table, through hash functions. The statistical analysis of counter values allows item frequencies to be estimated with the desired accuracy. Sketch algorithms compute a linear projection of the input: thanks to this property, sketches of data can be computed separately for different stream sources, and can then be integrated to produce the overall sketch [13].
- the approach is *hybrid*, meaning that frequent items are calculated online, with a single pass, while frequent itemsets are calculated by a further multi-pass analysis. This approach allows important information to be derived on the fly without imposing too strict time constraints on more complex tasks, such as the extraction of frequent k-itemsets, as this could excessively lower the accuracy of models.

Moreover, the hybrid approach improves the flexibility of the architecture, which can be used both when the computation of frequent itemsets is required and when only frequent items are needed.
- to support the mentioned hybrid approach, the architecture exploits the presence of *data cachers* on which recent data can be stored. In particular, miners can turn to data cachers to retrieve the statistics about frequent items and use them to identify frequent *sets* of items. To avoid excessive communication overhead, data cachers are distributed and placed close to stream sources and miners.

To the best of our knowledge, this is one of the first attempts to combine the four mentioned characteristics. In particular, we are not aware of attempts to combine the parallel and distributed paradigms in stream mining, nor of implemented systems that adopt the hybrid single-pass/multi-pass approach, though this kind of strategy is suggested and fostered in the recent literature [31]. The major advantages of the proposed architecture are its scalability and flexibility. Indeed, the architecture can efficiently exploit the presence of multiple miners, when this is required by the amount of computation and the generation rate of stream data. And the model can be easily adapted to the requirements of specific scenarios: (i) the pull approach allows the algorithm to work with any set of miners, as it is the miners that declare their availability to perform part of the work; (ii) the use and degree of parallelism can be adjusted depending on the stream rate and miners' capabilities; (iii) the model can be quickly adapted to the cases in which only frequent items are to be computed, or both frequent items and itemsets are needed.

Beyond presenting the architecture, we describe an implemented prototype and discuss a set of experiments performed in a Grid environment composed of three domains each one handling a data stream. In this scenario, we computed frequent items and itemsets for two well known datasets, *Kosarak* and *WebDocs*, and analyzed the processing time changing the number of miners available in each domain and the rate of the data streams.

This work extends the contribution presented in [7] and offers deeper insights concerning the scalability properties, the amount of exchanged data and the computational efficiency of the developed architecture. The results of experiments on a real distributed platform are reported to analyze the mentioned aspects. The paper is structured as follows: Section 2 summarizes the main issues regarding the mining of data streams and illustrates the most common algorithms for the computation of frequent items and itemsets; Section 3 describes the proposed parallel/distributed architecture for mining data streams and discusses the adopted hybrid approach; Section 4 presents the prototype and the testbed scenario, and reports the related results; Section 5 discusses related work and Section 6 concludes the paper.

2 Mining Frequent Items and Frequent Itemsets in Distributed Data Streams

Data stream analysis is often performed with randomized and approximated algorithms, since exact and deterministic algorithms would require too much computing time and memory space. Accordingly, data mining algorithms for data stream analysis are generally evaluated with respect to three metrics [13]:

- **Processing time** needed to update the data structures and the mining models after the arrival of new stream items;
- **Storage space** used by the algorithm;
- **Accuracy** of the approximated algorithm, in general specified through two parameters set by the user: the accuracy parameter ϵ and the failure probability δ, which means that the estimation error is at most ϵ with probability $(1-\delta)$. Of course, processing time and storage size strongly depend on these parameters.

As mentioned, the discovery of *frequent items* [12] is a very important task, consisting of identifying the items whose frequency in a stream exceeds a specified fraction σ of the overall stream size. This problem has a huge number of applications in a variety of scenarios: frequent items can be the most popular destinations of IP packets, the most frequent queries submitted to a search engine,

the most common values observed by sensors in a wireless environment, etc. Moreover, frequent items are often used as the basis for more complex analysis processes.

The problem is formalized as follows [13]:

Problem Statement Given a stream S of n items $e_1, ..., e_n$, where the frequency of an item i is $f_i = |\{j|e_j = i\}|$, and a frequency threshold σ, the ϵ-approximate-frequent-items problem consists of finding the set F of items such that: $F = \{i|\, f_i \geq (\sigma - \epsilon)n\}$

Two basic categories of algorithms can be used to solve this problem: the *Counter-Based* and the *Sketch-Based* algorithms. The algorithms of the first type maintain counters for a subset of elements, and counters are updated every time one of these elements is observed in the stream. If the observed element has no associated counter, the algorithm must choose whether to ignore the element or replace an existing counter with a counter for the new item. At the end of the first pass, frequent items will surely be among those associated with counters, but the inverse is not true, which requires at least a second pass to verify which counters actually correspond to frequent items. Some of the most used *Counter-Based* algorithms are the *SpaceSaving* algorithm [28] and the *LossyCounting* algorithm [26].

Conversely, *Sketch-Based* algorithms [12] do not monitor a subset of elements but provide, with a given accuracy, an estimation of the frequency for all stream elements using a matrix of counters C with d rows and w columns. A set of d hash functions $h_1, ..., h_d$ are chosen among a family of *pairwise-independent* functions, and are associated to the different matrix rows. Each item i observed in the stream is mapped, for each row r, to the matrix element $C[r, h_r(i)]$. This counter is then modified depending on the specific sketch algorithm: in the sketch-based *CountMin* algorithm [14], at the arrival of a new item i, the counter is incremented as follows (see Fig. 1):

$$for\ r \in [1, d] \rightarrow C[r, h_r(i)] += 1$$

The number of counters in a row, w, is lower than the number of elements, so there are conflicts, because several distinct elements will be mapped by a hash function to the same counter.

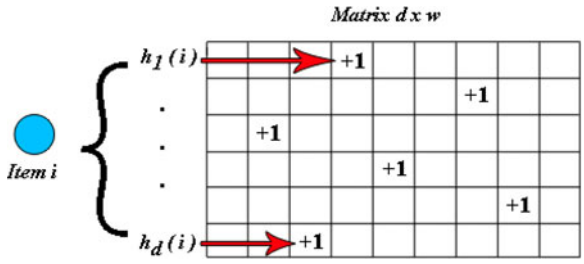

Fig. 1 `CountMin` algorithm. A new item is associated, for each row, to a different entry—computed with a hash function—which is then incremented

However, different elements are in conflict for different rows, which enables the adoption of statistical techniques to estimate the actual frequencies of elements. In *CountMin*, collisions always cause extra increments of counters, therefore the best estimation for the frequency f_i of element i is the minimum value of the counters associated to i:

$$f_i = \min_r\{C[r, h_r(i)]\}$$

Of course, the accuracy of sketch algorithms increases with the size of the matrix, since a larger matrix reduces the frequency of collisions of different elements on the same counter. In *CountMin*, setting $d = \lceil \ln \frac{1}{\delta} \rceil$ and $w = \lceil \frac{e}{\epsilon} \rceil$ ensures that f_i, in a stream with n elements, has error at most ϵn with probability of at least $1 - \delta$. The spatial complexity is $O\left(\frac{e}{\epsilon} \ln \frac{1}{\delta}\right)$ while the time for update is $O\left(\ln \frac{1}{\delta}\right)$.

More details about *CountMin* can be found in [14]. Here it is worth recalling that this algorithm, as all the sketch algorithms, has the important property that the sketch is a linear projection of the input. This means that the overall sketch of multiple streams can be computed by adding the sketches of single streams. This is the main reason why we decided to adopt *CountMin*: in a distributed architecture, it would be prohibitive to transmit source data to a central processing node, while the mere transmission of sketch summaries allows communication overhead to be drastically reduced.

The computation of frequent itemsets can either be performed directly, or by exploiting the statistics of frequent items. The direct computation in a single pass is feasible only if the stream

rate is moderate, due to the large number of candidate frequent itemsets. For example, in [22] a hybrid approach is used: first, a counter-based algorithm computes the candidate 2-itemsets, then a second pass is necessary to eliminate the *false* candidates, finally the Apriori property is exploited to find the frequent i-itemsets, for $i > 2$. The merit of hybrid approaches is that they try to combine the best of single-pass and multiple-pass algorithms [31], and can be particularly efficient in a distributed scenario. In our architecture, frequent items are computed on the fly with *CountMin*, and the results, stored in distributed data cachers, are used to compute frequent itemsets.

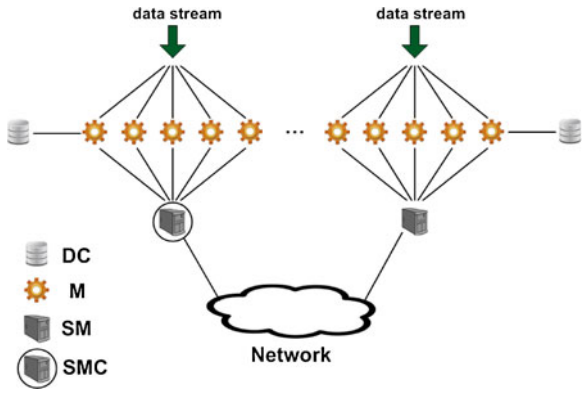

Fig. 2 Distributed architecture for data stream mining

3 A Hybrid Multi-Domain Architecture

This section presents the stream mining architecture that aims at solving the problem of computing frequent items and frequent itemsets from distributed data streams, exploiting a hybrid single-pass/multiple-pass strategy. We assumed that stream sources, though belonging to different domains, are homogenous, so that it is useful to extract knowledge from their union. Typical cases are the analysis of the traffic experienced by several routers of a wide area network, or the analysis of client requests forwarded to multiple web servers. Miner nodes are located close to the streams, so that data transmitted between different domains only consists of models (sketches), not raw data.

The architecture, depicted in Fig. 2, includes the following components:

- **Data Streams (DS)**, located in different domains.
- **Miners (M)**. They are placed close to the respective Data Streams, and perform two basic mining tasks: the computation of sketches for the discovery of frequent items, and the computation of the support count of candidate frequent itemsets. If a single Miner is unable to keep the pace of the local DS, the stream items can be partitioned and forwarded to a set of Miners, which operate in parallel. Each Miner computes the sketch only for the data it receives, and then forwards the results to the local Stream Manager. Parallel Miners can be associated to the nodes of a cluster or a high speed computer network, or to the cores of a manycore machine.
- **Stream Managers (SM):** in each domain, the Stream Manager collects the sketches computed by local miners, and derives the sketch for the local DS. Moreover, each SM cooperates with the Stream Manager Coordinator to compute global statistics, valid for the union of all the Data Streams.
- **Stream Manager Coordinator (SMC):** this node collects mining models from different domains and computes overall statistics regarding frequent items and frequent itemsets. The SMC can coincide with one of the Stream Managers, and can be chosen with an election algorithm. In the figure, the SM of the domain on the left also takes the role of SMC.
- **Data Cachers (DC)** are essential to enable the hybrid strategy and the computation of frequent itemsets, when this is needed. Each Data Cacher stores the statistics about frequent items discovered in the local domain. These results are then re-used by Miners to discover frequent itemsets composed of increasing numbers of items.

The algorithm for the computation of frequent items, outlined in Fig. 3, is performed continuously, for each new block of data generated by the data streams. A *block* is defined here as the set of transactions that are generated in a time

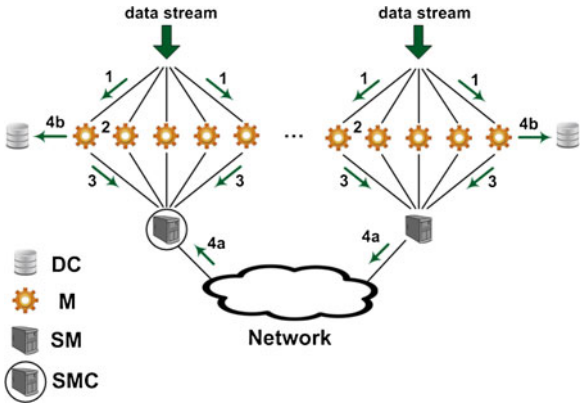

Fig. 3 Schema of the algorithm for mining frequent items

interval *P*. The algorithm includes the following steps, also shown in the figure:

1. a filter is used to partition the block into as many sub-blocks as the number of available Miners;
2. each Miner computes the sketch related to the received sub-block;
3. the Miner transmits the sketch to the SM, which overlaps the sketches, thanks to the linearity property of sketch algorithms, and extracts the frequent items for the local domain;
4. two concurrent operations are executed: every SM sends the local sketch to the SMC (step 4a), and the Miners send the most recent blocks of transactions to the local Data Cacher (step 4b). The last operation is only needed when frequent itemsets are to be computed, otherwise it can be skipped;
5. the SMC aggregates the sketches received by SMs and identifies the items that are frequent for the union of data streams.

Frequent items are computed for a *window* containing the most recent *W* blocks. This can be done easily thanks to the linearity of the sketch algorithm: at the arrival of a new block, the sketch of this block is added to the current sketch of the window, while the sketch of the least recent block is subtracted. The window-based approach is common because most interesting results are generally related to recent data [16].

Sketch-based algorithms are only capable of computing *frequent items*. To discover *frequent itemsets*, it is necessary to perform multiple passes on data. Candidate *k*-itemsets are constructed starting from frequent *(k–1)*-itemsets. More specifically, at the first step candidate 2-itemsets are all the possible pairs of frequent items: Miners must compute the support for these pairs to determine which of them are frequent. In the following steps, a candidate *k*-itemset is obtained by adding any frequent item to the frequent *(k–1)*-itemsets. Thanks to the Apriori property, candidates can be pruned by checking if all the *k–1* subsets are frequent: a *k*-itemset can be frequent only if all the subsets are frequent.

The approach allows us to compute both itemsets that are frequent for a single domain and those that are frequent for the union of distributed streams. Figure 4 shows an example of how frequent 3-itemsets are computed. The top part of the figure reports items and 2-itemsets that are frequent for the two considered domains and for the whole system. The candidate 3-itemsets, computed by the two SMs and by the SMC, are then reported, before and after the pruning based on the Apriori property. In the bottom part, the figure reports the support counts computed for the two domains and for the whole system. Finally, the SMs check which candidates exceed the specified threshold (in this case, set to 10 %): notice that the {abc} itemset is frequent globally though it is locally frequent in only one of the two domains. In general, it can happen that an itemset occurs frequently for a single domain and infrequently globally, or vice versa: therefore, it is necessary to separately perform the two kinds of computations.

The schema of the algorithm for mining frequent itemsets is illustrated in Fig. 5, which assumes that the steps indicated in Fig. 3 have already been performed.

The successive steps are:

6. each SM builds the candidate k-itemsets for the local domain (6a), and the SMC also builds the global candidate k-itemsets (6b);
7. the SMC sends the global candidates to the SMs for the computation of their support at the different domains;

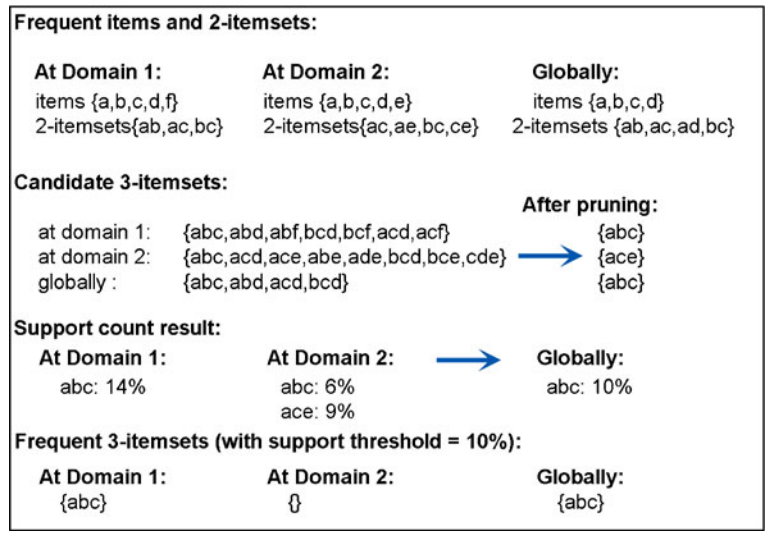

Fig. 4 Example of the computation of frequent 3-itemsets

8. SMs send both local and global candidates to the Miners;
9. Miners turn to the Data Cacher to retrieve the transactions included in the current window (this operation is performed only at the first iteration of the algorithm). Then, the Miners compute the support count for all the candidates, using the window-aware technique presented in [20];
10. Miners transmit the results to the local SM;
11. the SM aggregates the support counts received by Miners and selects the k-itemsets that are frequent in the local domain;
12. analogously, the SMs send the SMC the support counts of the global candidates;
13. the SMC computes the itemsets that are frequent over the whole system. At this point, the algorithm restarts from step 6 to find frequent itemsets with increasing numbers of items. The cycle stops either when the maximum allowed size of itemsets is reached or when no frequent itemset was found in the last iteration.

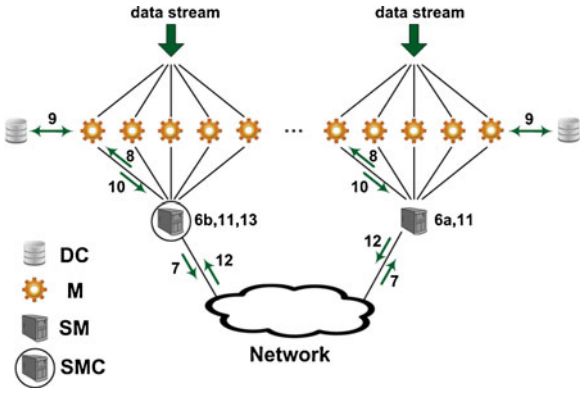

Fig. 5 Schema of the algorithm for mining frequent itemsets

4 Prototype and Performance Evaluation

The architecture described in the previous section was implemented starting from the *Mining@Home* system. *Mining@Home*, a Java-based framework partly inspired by the Public Computing paradigm, was adopted to perform several classes of data mining computations, among which the analysis of astronomical data to search for gravitational waves [27], and the discovery of closed frequent itemsets with parallel algorithms [24]. The main features of the stream mining prototype inherited from *Mining@Home*, are the *pull* approach (Miners are assigned jobs on the basis of their availability) and the adoption of Data Cachers to store reusable data. Moreover, some important modifications were necessary to adapt the framework to the stream mining scenario. For example, the selection of the Miners that are the most appropriate to perform the mining tasks is subject to vicinity constraints, because

in a streaming environment it is very important that the analysis of data be performed close to the data source. Another notable modification is the adoption of the hybrid approach for the single-pass computation of frequent items and the multi-pass computation of frequent itemsets.

Experiments were performed on a Grid composed of two remote ICAR networks (ICAR-CS located in Cosenza and ICAR-NA in Naples, 300 kilometers away) and one network owned by the University of Calabria (UNICAL), in Cosenza. The topology of the Grid, as well as the inter-domain and intra-domain transfer rates, are depicted in Fig. 6. The Miners and the Stream Managers were installed on the nodes of clusters while the Data Sources and the Data Cachers were put on different nodes, external to the clusters. The UNICAL cluster has twelve Cpu Intel Xeon E5520 nodes with four 2.27 GHz processors and 24 GB RAM; the ICAR-CS cluster has twelve Intel Itanium nodes with two 1.5 GHz CPU and 4 GB RAM; finally, the ICAR-NA cluster is an HP XC6000 with 64 nodes equipped with two 1.4 GHz CPU and 8 GB RAM. All the nodes run Linux, and the software components are written in Java.

To assess the prototype, we used the transactional datasets published by the *FIMI Repository* [18]. Some of these datasets are originated by data streams, so they are appropriate for our analysis. In particular:

- The *"kosarak"* dataset contains a list of *click-streams* generated by users of an online portal. The analysis of user visits can be useful to identify the most popular sections of the portal, the preferences and requirements of users, etc.;
- the *"webDocs"* dataset is generated from a set of Web pages. Each page, after the application of a filtering algorithm, is represented with a set of significant words included in it. The analysis of most frequent words, or sets of words, can be useful to devise caching policies, indexing techniques, etc.

Basic information about the two datasets is summarized below:

Dataset	MB	No. of tuples	No. of distinct items	Size of tuples (no. of items)		
				min	med	max
kosarak	30.5	990002	41270	1	8	2498
webDocs	1413	1692082	5267656	1	177	71472

The parameters used to assess the prototype are listed below:

- P: the time interval to receive a block of data. This interval determines the average number of transactions generated within a block, denoted as N_t, and the average size of a block in bytes, B;
- N_{MD}: the number of available miners per domain. In our experiments, this number is the same for the three domains;
- N_M: the total number of available miners in the Grid, equal to $3 \cdot N_{MD}$;
- F_{CPU}: the fraction of CPU time reserved on miners for the experiments, set to 30 %. This setting was used to make the results independent from the execution of other processes on the same nodes.[1]
- S: the support threshold used to determine frequent items and itemsets;
- W: the size of the sliding window, i.e., the number of consecutive blocks of data on which computation is performed;
- C_M: the capacity of the miner buffer. Unless otherwise stated, it is equal to the size of a data block B.
- ϵ and δ, the accuracy parameters of the sketch algorithm, which are both set to 0.01.

[1] the fraction of CPU is tuned using the program *"cpulimit"*, http://cpulimit.sourceforge.net.

Fig. 6 Topology of the Grid used for experimental evaluation

– the maximum size of candidate itemsets, set to 5 for Kosarak and to 8 for Webdocs.

Prior to the experiments, the datasets are retrieved and stored on the Data Source nodes of the three domains. During the experiments, data is sent to local Miners with a specified transfer rate, to reproduce the original data stream. The transmission rate is adjusted by setting the parameter N_t, the number of transactions generated during the time interval P.

The main performance index assessed during the experiments is the *execution time*, defined as the time interval between the transmission of a new block of stream data and the time at which the analysis of this block has been completed by the Stream Manager Coordinator (SMC). If this value is not longer than the time interval P, it means that the system is able to keep the pace with data production.

4.1 Experiments with the Dataset *Kosarak*

The experiments were executed assuming a time period P equal to 15 s. The generation rates were compatible with the Web sites of Wikipedia, Microsoft and Ebay, as estimated using the Web site http://www.webtraffic24.com. These rates correspond, respectively, to values of N_t equal to about 40,000, 30,000 and 20,000 transactions per block. The generation rates were equally partitioned between the three domains. These are very high generation rates, and allowed the prototype to be tested in challenging conditions.

Figures 7 and 8 report the execution time experienced for the computation of frequent items exclusively (I), and for the computation of both frequent items and itemsets (I+IS), vs. the total number of miners N_M. The execution time was averaged over 20 time periods in order to have a more robust statistical relevance. In these experiments, S was set to 0.02, the Miner cache size C_M was set to the average size of a block B, and the window size W was set to 5. Plots are reported for three different values of N_t. Both figures show that the processing time decreases as the number of miners increases, which is a sign of the good scalability of the architecture. Scalable behavior is ensured by two main factors: the linearity property of the sketch algorithm, and the placement of Data Cachers close to the miners. In this evaluation the value $N_M = 3$ (or $N_{MD} = 1$) corresponds to the case in which the mining computation is sequential on each domain, i.e., it is performed by a single node.

The system is stable when the execution time is lower than the time period P (15 s): in such a case, the system is able to keep the pace with the generation of stream data. This condition is always verified when the system is only asked to compute frequent items, as is clear from Fig. 7. On the other hand, the computation of frequent itemsets is much more time consuming. The dashed line depicted in Fig. 8 corresponds to the time period P, and it is shown to easily check in which cases the system is stable. Results show that a single miner per domain is not sufficient: depending on

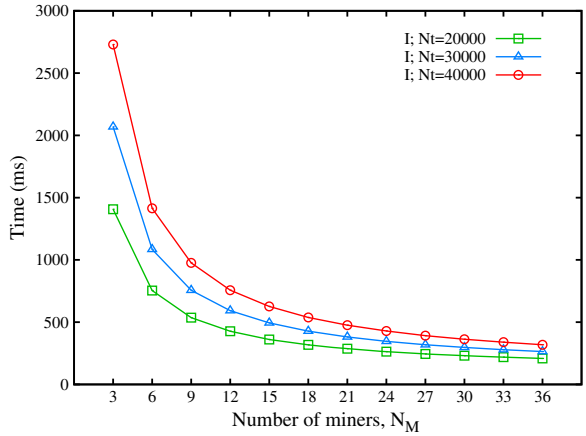

Fig. 7 Analysis of *Kosarak*: execution time for the computation of frequent items (I), vs. the number of miners, for different values of the number of transactions per block, N_t

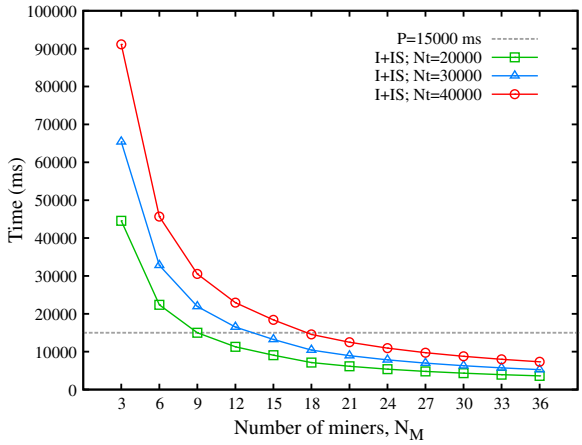

Fig. 8 Analysis of *Kosarak*: execution time for the computation of frequent items (I) and itemsets (IS), vs. the number of miners, for different values of the number of transactions per block, N_t

the generation rate, three, five, or six miners per domain are needed to keep the processing time below the period length P.

To better assess the system behavior it is useful to analyze the data traffic involved in the computation. Figure 9 shows the amount of data transmitted over the network at the generation of a new block of data, for different numbers of miners per domain. In these experiments, N_t was set to $30,000$ while the other parameters were set as detailed before. The first three groups of bars show the overall amount of data transferred between nodes of type A to nodes of type B in a single domain, denoted as $A \rightarrow B$. For example, $DS \rightarrow M$ is the amount of data transmitted by the Data Source to the Miners of a single domain at every time period. The values of $DS \rightarrow M$ and $M \rightarrow DC$ are equal since each Miner sends to the local DC the data received from the DS. The forth group reports DT_{Domain}, the overall amount of data transferred within a single domain, computed as the sum of the contributions of the first three groups (the contribution of the first group is considered twice). The contribution $SM \rightarrow SMC$ is the amount of data transferred between remote domains, i.e., between the Stream Managers and the Stream Manager Coordinator. Finally, the last group of bars reports the amount of data transferred over the whole network, DT_{Net}. This is computed as the term DT_{Domain} times the number of domains—in this case 3—plus the term $SM \rightarrow SMC$.

It is interesting to notice that the contribution $DC \rightarrow M$ decreases when the number of miners per domain increases, and becomes null when N_{MD} is equal or greater than 5. This can be explained by considering that each miner must compute the frequency of items and itemsets over a window of 5 time periods, and possesses a cache that can contain one complete block of data. If there is only one miner per domain, this miner can store one block and must retrieve the remaining four blocks from the local Data Cacher. As the number of miners increases, each Miner needs to

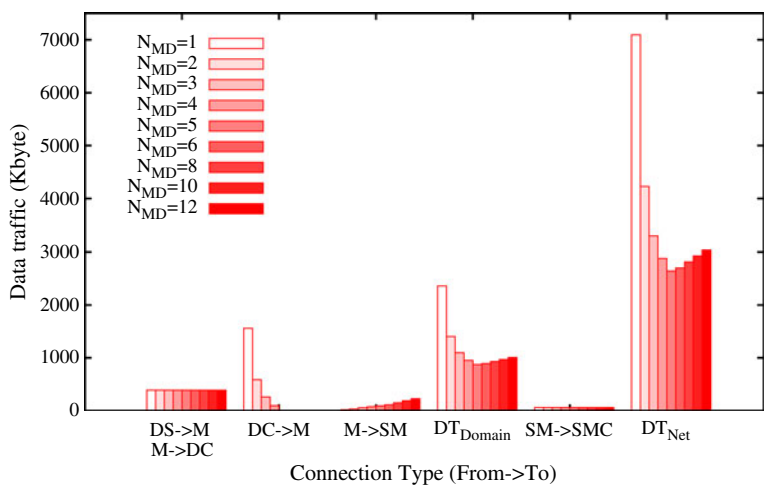

Fig. 9 Data traffic per each block of data generated by the Data Streams. In this experiments, $N_t = 30,000$, $W = 5$ and $C_M = B$

request less data from the Data Cacher, and needs no data when N_{MD} equals or exceeds the window size. Indeed, with N_{MD} miners per domain, each miner receives from the Data Source an amount of data approximately equal to B/N_{MD} (because the block is partitioned and distributed to the Miners), and then can store the data corresponding to N_{MD} consecutive blocks, since the size of such data amounts to B.

Conversely, the component $M \rightarrow SM$ slightly increases with the number of miners, because every miner sends the local SM a set of results (the sketch and the support counts of itemsets) having approximately the same size. However, as the contribution of $DC \rightarrow M$ has a larger weight than $M \rightarrow SM$, the overall amount of data exchanged within a domain decreases as N_{MD} increases from 1 to 5. For larger values of N_{MD}, DT_{Domain} starts to increase, because $M \rightarrow SM$ slightly increases and $DC \rightarrow M$ gives no contribution. A similar trend is observed for the values of DT_{Net}.

An efficiency analysis was performed in accordance with the study of parallel architectures presented in [21]. Specifically, we extracted the overall computation time T_C, i.e., the sum of the computation times measured on the different miners, and the overall overhead time T_O, defined as the sum of all the times spent in other activities, which practically coincide with the transfer times. These two indexes are shown in Fig. 10 using a logarithmic scale. Not surprisingly, the overhead time follows a similar trend as the overall amount of transferred data, DT_{Net}, reported in Fig. 9. The overall computation time T_C, on the other hand, has a nearly constant trend.

The efficiency of the computation can be defined as the fraction of time that the miners actually devote to computation with respect to the sum of computation and overhead time: $E = \frac{T_C}{T_C + T_O}$. Figure 11 reports efficiency values, which are very high, always larger than 0.9. It is appreciated that efficiency increases as the number of miners per domain increases up to 5, i.e., the window size. This effect is induced by the use of caching, as explained in [21]. Specifically, when N_{MD} increases up to the window size, each miner needs to request less data from the Data Cacher, because more data can be stored in the local cache: this leads to a higher efficiency. For larger values of N_{MD}, this effect does not hold anymore and the efficiency slightly decreases, being still very high. Moreover, it is noticed that the efficiency increases with the rate of data streams. This means that the distributed architecture is increasingly convenient when the problem size increases, which is a another sign of good scalability properties.

Figure 12 shows the execution time measured when setting the value of N_t to 30,000 and varying the support threshold S. This parameter does

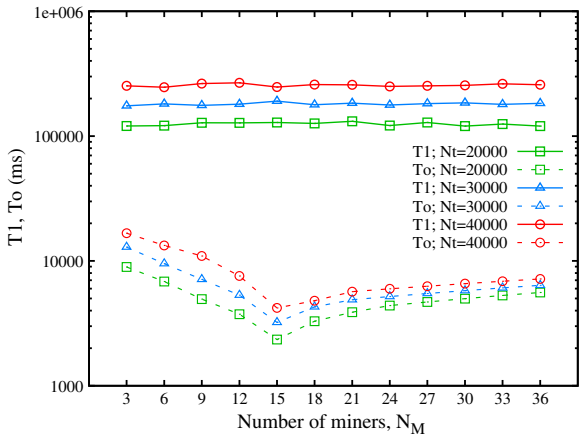

Fig. 10 Analysis of *Kosarak*: overall computing time (T_C) and overhead time (T_O), vs. the number of miners, for different values of the number of transactions per block, N_t

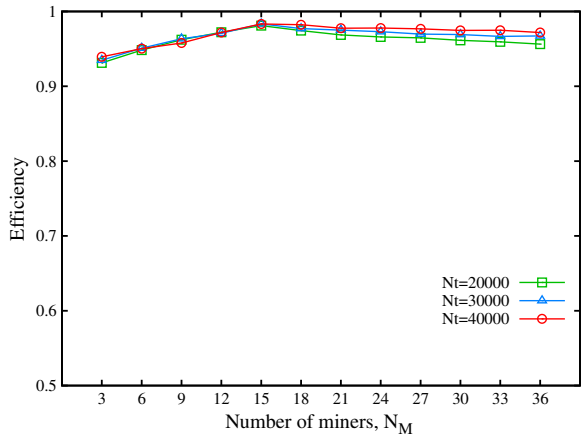

Fig. 11 Analysis of *Kosarak*: efficiency vs. the number of miners, for different values of the number of transactions per block, N_t

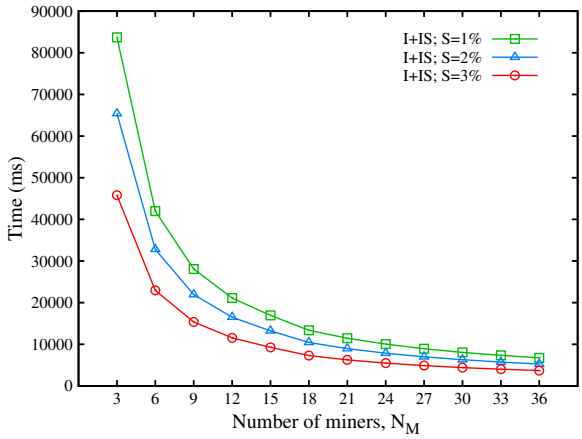

Fig. 12 Analysis of *Kosarak*: execution time for the computation of frequent items and itemsets vs. the number of miners per domain, for different values of the threshold *S*. The value of N_t is set to 30,000

not influence the time to execute the frequent items algorithm, which is executed in a single pass: therefore the figure reports the total execution times regarding the combined computation of frequent items and frequent itemsets (I+IS). As expected, a lower value of the threshold leads to an increase of the processing time, since the number of frequent itemsets, computed at each step of the algorithm (see Fig. 5), is larger. Again, when computation is more demanding the absolute processing time increases, but the system is more efficient. This is visible in Fig. 13: the trend of curves is similar, but efficiency is higher with lower values of the support threshold.

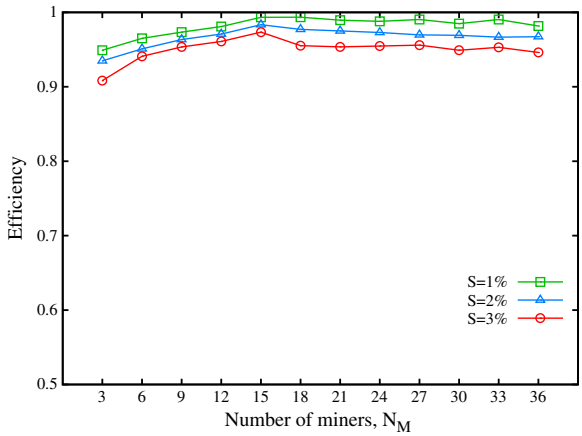

Fig. 13 Analysis of *Kosarak*: efficiency vs. the number of miners per domain, for different values of the threshold *S*. The value of N_t is set to 30,000

4.2 Experiments with the Dataset *Webdocs*

A second set of experiments was performed taking the dataset *Webdocs* as input of the data stream. As the dataset contains representative words of Web pages filtered by a search engine, the data rate was set to typical values registered by servers of Google and Altavista, again using the site http://www.webtraffic24.com to do the estimation. The considered values for N_t were 1000, 3000 and 6000 transactions, with the time period P set to 15 s. A single transaction contains on average many more items than the typical Kosarak transaction, so the block size is larger: for example, with $N_t = 3000$, the value of B is about 750 KBytes, while it was about 150 Kbytes in Kosarak experiments. As for the previous set of tests, the frequency threshold S was set to 0.02 and the window size to 5 blocks.

Figure 14 shows the average time needed to compute frequent items and itemsets after the arrival of a new block. Owing to the higher complexity of the dataset—in particular, the larger number of items per transaction—the computation of frequent itemsets is more challenging. To perform this task, 15 miners are sufficient if the data rate is equal to 1000 transactions per time period, but

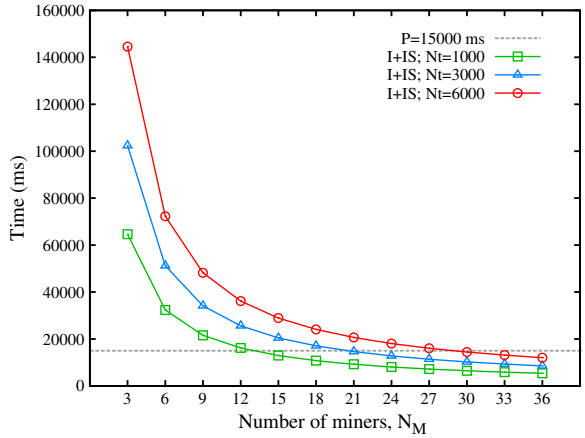

Fig. 14 Analysis of *Webdocs*: execution time for the computation of frequent items and itemsets (I+IS), vs. the number of miners, for different values of the number of transactions per block, N_t

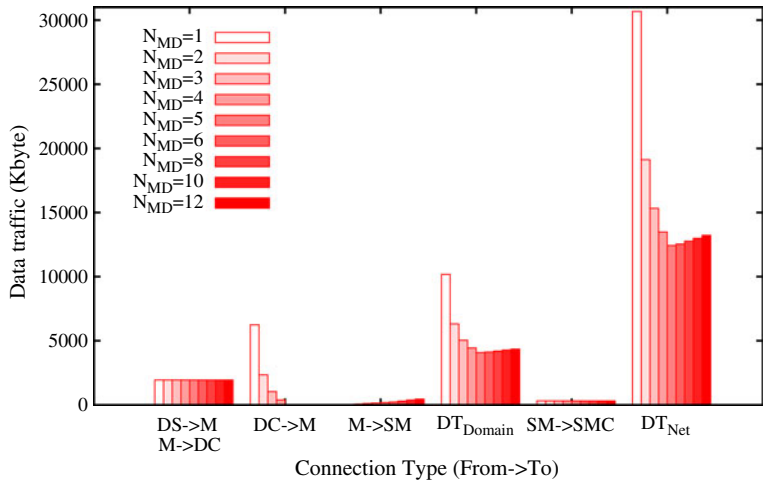

Fig. 15 Data traffic per each block of data generated by the Data Streams. In this experiments, $N_t = 3000$, $W = 5$ and $C_M = B$

as many as 30 miners are needed when N_t is equal to 6000. It is worth noting that in this scenario any centralized architecture would have few chances to keep pace with data, which means that the computation of frequent itemsets would have to be done offline, while the architecture proposed here can achieve the goal of online processing by using an appropriate degree of parallelism.

Figure 15 reports the amount of data exchanged within single domains and in the whole network. It can be noticed that the absolute amount of transmitted data is higher than that experienced with Kosarak, and reported in Fig. 9.

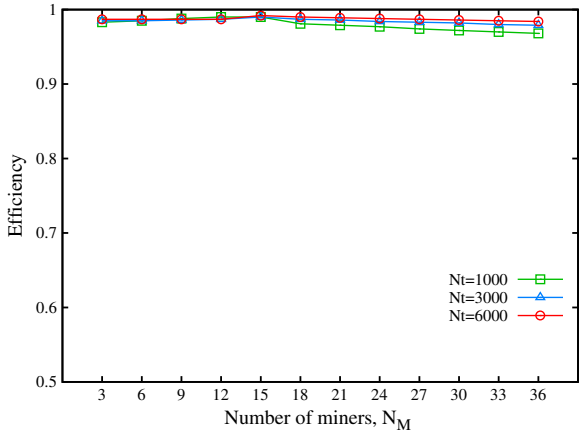

Fig. 16 Analysis of *Webdocs*: efficiency vs. the number of miners per domain, for different values of the number of transactions per block, N_t

Figure 16 displays the values of efficiency for the same experiments. Trends are very similar to those obtained with the Kosarak data, but here absolute values of efficiency are higher. The reason is that, although more data must be transmitted in the Webdocs scenario, the increase in computation complexity is even more remarkable. Overall, the fraction of computation time is higher with respect to the total time than in the Kosarak scenario, therefore efficiency is higher. As done for Kosarak, also for Webdocs we analyzed the architecture behavior with varying support threshold and window size. These experiments took us to very similar conclusions, so related results are not reported here.

5 Related Work

The increase in the data produced by large-scale scientific and commercial applications necessitates innovative solutions for efficient transfer and analysis of data [32]. The analysis of data streams has recently attracted a lot of attention owing to the wide range of applications for which it can be extremely useful. Important challenges arise from the necessity of performing most computation with a single pass on stream data, because of limitations in time and memory space. Stream mining algorithms deal with problems as diverse as clustering and classification of data streams,

change detection, stream cube analysis, indexing, forecasting, etc [1].

For many important application domains previously mentioned in this paper, a major need is to identify frequent patterns in data streams, either single frequent elements or frequent sets of items in transactional databases. A rich survey of algorithms for discovering frequent items is provided by Cormode and Hadjieleftheriou [13]. In their paper, discussion focuses on the two main classes of algorithms for finding frequent items. Counter-based algorithms have their foundation on some techniques proposed in the early 80s to solve the *Majority* problem [17], i.e., the problem of finding a majority element in a stream, using a single counter. Variants of this algorithm were devised, sometimes decades later, to discover items whose frequencies exceed any given threshold. Lossy-Counting is perhaps the most popular algorithm of this type [26].

The second class of algorithms compute a *sketch*, i.e., a linear projection of the input, and provide an approximated estimation of item frequencies using limited computing and memory resources. Popular algorithms of this kind are CountSketch [8] and CountMin [14], and the latter is adopted in this paper. Advantages and limitations of sketch algorithms are discussed in [3]. Important advantages are the notable space efficiency (required space is logarithmic in the number of distinct items), the possibility of naturally dealing with negative updates and item deletions, and the linear property, which allows sketches of multiple streams to be computed by overlapping the sketches of single streams. The main limitation is the underlying assumption that the domain size of the data stream is large, however this assumption holds in many significant domains.

Even if modern single-pass algorithms are extremely sophisticated and powerful, multi-pass algorithms are still necessary either when the stream rate is too rapid, or when the problem is inherently related to the execution of multiple passes, which is the case, for example, of the frequent itemsets problem [9]. Single-pass algorithms can be forced to check the frequency of 2- or 3-itemsets, but this approach cannot be generalized easily, as the number of candidate k-itemsets is combinatorial, and it can become very large when increasing the value of k [22]. Therefore, a very promising avenue could be to devise hybrid approaches, which try to combine the best of single- and multiple-pass algorithms. A strategy of this kind, discussed in [31], is adopted in the mining architecture presented in this paper.

The analysis of streams is even more challenging when data is produced by different sources spread in a distributed environment, an ever more frequent case. For example, NASA [5] produces many terabytes of data per month through thousands of observation instruments located at multiple and distant data centers, and needs to extract information from this massive amount of stream data using efficient data mining applications. Another example comes from the multiplicity of data streams occurring in complex scientific workflows or in large-scale distributed collaborations exacerbate this problem, particularly when different streams have different performance requirements [6].

A thorough discussion of the approaches currently used to mine multiple data streams can be found in [30]. The paper distinguishes between the *centralized* model, under which streams are directed to a central location before they are mined, and the *distributed* model, in which distributed computing nodes perform part of the computation close to the data, and send to a central site only the models, not the data. Of course, the distributed approach has notable advantages in terms of degree of parallelism and scalability.

An interesting approach for the continuous tracking of complex queries over collections of distributed streams is presented in [11]. To reduce the communication overhead, the adopted strategy combines two technical solutions: (i) remote sites only communicate to the coordinator concise summary information on local streams (in the form of sketches); (ii) even such communications are avoided when the behavior of local streams remains reasonably stable, or predictable: updates of sketches are only transmitted when a certain amount of change is observed locally. The success of this strategy depends on the level of approximation on the results that is tolerated. A similar approach is adopted in [29]: here stream data is sent to the central processor after being filtered at remote data sources. The filters adapt

to changing conditions to minimize stream rates while guaranteeing that the central processor still receives the updates necessary to provide answers of adequate precision.

In [25], the problem of finding frequent items in the union of multiple distributed streams is tackled by setting a hierarchical communication topology in which streams are the leaves, a central processor is the root, and intermediate nodes can compress data flowing from the leaves to the root. The amount of compression is dynamically adapted to make the tolerated error (difference from estimation and actual frequency of items) follow a precision gradient: the error must be very low at nodes close to the sources, but it can gradually increase as the communication hierarchy is climbed. The objective of this strategy is to minimize load on the central node while providing acceptable error guarantees on answers.

In [10] authors deal with the problem of mining closed frequent itemsets from a data stream over a sliding window using limited memory space. They propose the *Moment* algorithm and an efficient compact data structure, i.e., CET, the Closed Enumeration Tree. This structure is aimed at dynamically maintaining a selected set of itemsets over a sliding window. The number of nodes needed by CET is shown to be proportional to that of discovered closed frequent itemsets, which guarantees the compactness of the structure. Another approach for mining frequent itemsets on streaming data, claimed as the first one requiring no out-of-core memory structure, is presented in [23]. The adopted method, named *StreamMining*, is based on two main steps. The first is implemented by a one-pass algorithm executed on the streaming data, and has deterministic bounds on the accuracy. The second step consists in the execution of a two-pass algorithm, which improves the accuracy by pruning possible false negatives detected at the first step. The algorithm has shown to achieve good performances, but it tends to degrade when the average size of frequent itemsets increases.

6 Conclusions

In recent years, the progress in digital data production and pervasive computing technology have made it possible to produce and store large streams of data. Data mining techniques became vital to analyze such large and continuous streams of data for detecting regularities or outlier values in them. In particular, when data production is massive and/or distributed, decentralized architectures and algorithms are needed for its analysis.

The distributed stream mining system presented in this paper is a contribution in the field and it aims at solving the problem of computing frequent items and frequent itemsets from distributed data streams by exploiting a hybrid single-pass/multiple-pass strategy. Beyond presenting the system architecture, we described a prototype that implements it and discussed a set of experiments performed in a real Grid environment. The experimental results confirm that the approach is scalable and can manage large data production by using an appropriate number of miners in the distributed architecture.

References

1. Aggarwal, C.: An introduction to data streams. In: Aggarwal C. (ed.) Data Streams: Models and Algorithms, pp. 1–8. Springer, New York (2007)
2. Aggarwal, C.C.: Data Streams: Models and Algorithms. Springer, New York (2007)
3. Aggarwal, C.C., Yu, P.S.: A survey of synopsis construction in data streams. In: Aggarwal C. (ed.) Data Streams: Models and Algorithms, pp. 169–207. Springer, New York (2007)
4. Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A.I.: Fast Discovery of Association Rules. American Association for Artificial Intelligence, Menlo Park (1996)
5. Barkstrom, B., Hinke, T., Gavali, S., Smith, W., Seufzer, W., Hu, C., Cordner, D.: Distributed generation of nasa earth science data products. J. Grid Computing **1**(2), 101–116 (2003)
6. Cai, Z., Kumar, V., Schwan, K.: Iq-paths: Predictably high performance data streams across dynamic network overlays. J. Grid Computing **5**(2), 129–150 (2007)
7. Cesario, E., Grillo, A., Mastroianni, C., Talia, D.: A sketch-based architecture for mining frequent items and itemsets from distributed data streams. In: Proc. of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2011), pp. 245–253. Newport Beach, CA (2011)
8. Charikar, M., Chen, K., Farach-Colton, M.: Finding frequent items in data streams. In: Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP) (2002)

9. Cheng, J., Ke, Y., Ng, W.: A survey on algorithms for mining frequent itemsets over data streams. Knowl. Inf. Syst. **16**, 1–27 (2008)
10. Chi, Y., Wang, H., Yu, P.S., Muntz, R.R., Fischer, M.: Catch the moment: maintaining closed frequent itemsets over a data stream sliding window. Knowl. Inf. Syst. **10**(3), 265–294 (2006)
11. Cormode, G., Garofalakis, M.: Approximate continuous querying over distributed streams. ACM Trans. Database Syst. **33**(2), 9:1–9:39 (2008)
12. Cormode, G., Hadjieleftheriou, M.: Finding frequent items in data streams. In: International Conference on Very Large Data Bases (2008)
13. Cormode, G., Hadjieleftheriou, M.: Finding the frequent items in streams of data. Commun. ACM **52**(10), 97–105 (2009)
14. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. J. Algorithms **55**(1), 58–75 (2005)
15. Cormode, G., Muthukrishnan, S., Yi, K., Zhang, Q.: Optimal Sampling from Distributed Streams. ACM Principles of Database Systems (PODS) (2010)
16. Datar, M., Gionis, A., Indyk, P., Motwani, R.: Maintaining stream statistics over sliding windows. SIAM J. Comput. (SIAMCOMP) **31**(6), 635–644 (2002)
17. Fischer, M., Salzburg, S.: Finding a majority among n votes: solution to problem 81-5. J. Algorithms **3**(4), 376–379 (1982)
18. Frequent itemset mining dataset repository. Available at http://fimi.ua.ac.be/data/. Accessed June 2013
19. Gaber, M.M., Zaslavsky, A., Krishnaswamy, S.: Mining data streams: A review. ACM SIGMOD Rec. **34**(1), 18–26 (2005)
20. Giannella, C., Han, J., Pei, J., Yan, X., Yu, P.S.: Mining frequent patterns in data streams at multiple time granularities. In: Kargupta, H., Joshi, A., Sivakumar, K., Yesha Y. (eds.) Data Mining: Next Generation Challenges and Future Directions, chap. 3, pp. 191–210. MIT Press, Menlo Park (2004)
21. Grama, A.Y., Gupta, A., Kumar, V.: Isoefficiency: measuring the scalability of parallel algorithms and architectures. IEEE Parallel Distrib. Technol. **1**(3), 12–21 (1993)
22. Jin, R., Agrawal, G.: An algorithm for in-core frequent itemset mining on streaming data. In: 5th IEEE International Conference on Data Mining ICDM, pp. 210–217. Houston, TX (2005)
23. Jin, R., Agrawal, G.: An algorithm for in-core frequent itemset mining on streaming data. In: Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005), pp. 210–217. Houston, TX (2005)
24. Lucchese, C., Mastroianni, C., Orlando, S., Talia, D.: Mining@home: toward a public resource computing framework for distributed data mining. Concurr. Comput.: Pract. Exper. **22**(5), 658–682 (2009)
25. Manjhi, A., Shkapenyuk, V., Dhamdhere, K., Olston, C.: Finding (recently) frequent items in distributed data streams. In: ICDE '05: Proceedings of the 21st International Conference on Data Engineering, pp. 767–778. Tokyo, Japan (2005)
26. Manku, G., Motwani, R.: Approximate frequency counts over data streams. In: International Conference on Very Large Data Bases (2002)
27. Mastroianni, C., Cozza, P., Talia, D., Kelley, I., Taylor, I.: A scalable super-peer approach for public scientific computation. Futur. Gener. Comput. Syst. **25**(3), 213–223 (2009)
28. Metwally, A., Agrawal, D., Abbadi, A.: Efficient computation of frequent and top-k elements in data streams. In: International Conference on Database Theory (2005)
29. Olston, C., Jiang, J., Widom, J.: Adaptive filters for continuous queries over distributed data streams. In: SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. San Diego, CA (2003)
30. Srinivasan Parthasarathy, A.G., Otey, M.E.: A survey of distributed mining of data streams. In: Aggarwal C. (ed.) Data Streams: Models and Algorithms, pp. 289–307. Springer, New York (2007)
31. Wright, A.: Data streaming 2.0. Commun. ACM (CACM) **53**(4), 13–14 (2010)
32. Yildirim, E., Kosar, T.: End-to-end data-flow parallelism for throughput optimization in high-speed networks. J. Grid Computing **10**(3), 395–418 (2012)