

Approximate Mining of Frequent Patterns on Streams

Claudio Silvestri and Salvatore Orlando

Dipartimento di Informatica, Università Ca' Foscari, Via Torino 155, Venezia, Italy
silvestri@dsi.unive.it, orlando@dsi.unive.it

Abstract. This paper introduces a new algorithm for approximate mining of frequent patterns from streams of transactions using a limited amount of memory.

The proposed algorithm consists in the computation of frequent itemsets in recent data and an effective method for inferring the global support of previously infrequent itemsets. Both upper and lower bounds on the support of each pattern found are returned along with the interpolated support. An extensive experimental evaluation shows that AP_{Stream} , the proposed algorithm, yields a good approximation of the exact global result considering both the set of patterns found and their support.

1 Introduction

Association Rule Mining (ARM), one of the most popular topic in the KDD field [13, 5], regards the extractions of association rules from a database of transactions \mathcal{D} . In this paper we are interested in the most computationally expensive phase of ARM, i.e the Frequent Pattern Mining (*FPM*) one, during which the set \mathcal{F} of all the *frequent* patterns (sorted itemsets) is built. In a stream setting new transactions are continuously added to the dataset. Hence, we need a notation for indicating that a particular dataset or result is referred to a particular time interval. To this end, we write the interval as a subscript after the entity. Thus $\mathcal{D}_{[1,t]}$ denote the part of the dataset preceding time t . If t is current time, and the notation is not ambiguous we will write just \mathcal{D} .

A pattern x is frequent at time t in dataset $\mathcal{D}_{[1,t]}$ with respect to a minimum support $minsup$, if its support is greater than $\sigma_{min[1,t]} = minsup \cdot |\mathcal{D}_{[1,t]}|$, i.e. the pattern occurs in at least $\sigma_{min[1,t]}$ transaction, where $|\mathcal{D}_{[1,t]}|$ is the number of transactions in the stream \mathcal{D} until time t . A k -pattern is a pattern composed of k items, $\mathcal{F}_{k[1,t]}$ is the set of all frequent k -patterns, and $\mathcal{F}_{[1,t]}$ is the set of all frequent patterns.

The infinite nature of stream data sources is a serious obstacle to the use of most of the traditional methods since available computing resources are limited. One of the first effect is the need to process data as they arrive. The amount of previously happened events is usually overwhelming, so they can be either dropped after processing or archived separately in secondary storage. Even the apparently simple discovery of frequent items in a stream is challenging [3], since

its exact solution requires to store a counter for each distinct item received. The frequent itemset mining problem on stream of transactions poses additional memory and computational issues due to the exponential growth of solution size with respect to the corresponding problem on streams of items. In [11] Manku and Motwani propose an extension of their Lossy Count approximate algorithm to the case of frequent itemset. In [8] R.Jin and G.Agrawal describe a new algorithm for frequent itemset mining based on a simple algorithm for iceberg query on a stream [10] inspired by [9].

In this paper we discuss a *streaming* algorithm for approximate mining of frequent patterns, AP_{Stream} (Approximate Partition for Stream), that exploits DCI [17], a state-of-the-art algorithm for FPM, as the miner engine for recent data. The proposed algorithm is based on AP [15], an algorithm for approximate distributed mining of frequent itemset which in turn uses a computation method inspired by the Partition algorithm [2].

This paper is organized as follow. Section 2 describes the AP_{Stream} algorithm and the Partition algorithm who inspired AP. In Section 3 we report and discuss our experimental results. Finally, in Section 4, we draw some conclusions.

2 The algorithm

Our AP_{Stream} (Approximate Partition for Stream) algorithm was inspired by Partition [2], a sequential algorithm which divides the dataset into several partitions processed independently and then merges local solutions.

In this paper we will use the terms local and global to refer to data or result concerning just a contiguous part of the stream, hereinafter called a block of transactions, and the whole stream. Furthermore, we suppose that each block corresponds to one time unit: hence, $\mathcal{D}_{[1,n]}$ will indicate the first $n-1$ data blocks, and \mathcal{D}_n the n^{th} block. This hypothesis allows us to adopt a lighter notation and cause no loss of generality.

The Streaming Partition algorithm. The basic idea exploited by Partition is the following: if the dataset is divided into several partitions then each *globally* frequent pattern must be *locally* frequent in at least one partition. This guarantees that the union of all local solutions is a superset of the global solution. However, one further pass over the database is necessary to remove all false positives, i.e. patterns that result locally frequent but globally infrequent.

In order to extend this approach to a stream setting, blocks of data received from the stream are used as an infinite set of partitions. Unfortunately, in the stream case, only recent raw data (transactions) can be maintained available for processing due to memory limits, thus the usual Partition second pass will be restricted to accessible data. We will name this algorithm **Streaming Partition**. The first time a pattern x is reported, its support corresponds to the support computed in the current block. In case it appeared previously, this mean introducing an error. If j is the first block where x is frequent, then this error can be at most $\sigma_{min[1,j]} - 1$.

The AP_{Stream} algorithm. The streaming algorithm we propose in this paper, AP_{Stream}, tries to overcome some of the problems encountered by Streaming Partition ([14]) and other similar algorithms for association mining on streams when the data skew between different incoming blocks is high. This skew might cause a globally frequent pattern x to result infrequent on a given data block \mathcal{D}_i . In other words, since $\sigma_i(x) < \text{minsup} \cdot |\mathcal{D}_i|$, x will not be found as a frequent pattern in the i^{th} block. As a consequence, we will not be able to count on the knowledge of $\sigma_i(x)$, and thus we can not exactly compute the support of x . Unfortunately, Streaming Partition might also deduce that x is not globally frequent, because $\sum_{j,j \neq i} \sigma_j(x) < \text{minsup} \cdot |\mathcal{D}|$.

AP_{Stream} addresses this issue in different ways, according to the specific situation. The following table summarize the possible cases and the action taken by AP_{Stream}:

$\sigma_{[1,i]}(x)$	$\sigma_i(x)$	Action
known	known	sum recent support to past support and bounds.
known	unknown	recount support on recent, still available, data.
unknown	known	interpolate past support

The first case is the simpler to handle: the new support $\sigma_{[1,i]}(x)$ will be the sum of $\sigma_{[1,i]}(x)$ and $\sigma_i(x)$. If $\sigma_{[1,i]}(x)$ was approximated, then the width of the error interval will remain the same. The second one is similar, except that we need to look at recent data for computing $\sigma_i(x)$. The key difference with Streaming Partition is the handling of the last case. AP_{Stream}, instead of supposing that x never appeared in the past, tries to interpolate $\sigma_{[1,i]}(x)$. The interpolation is based on the knowledge of:

- the exact support of each *item* (or optionally just the approximate support of a fixed number of most frequent items)
- the *reduction factors* of the support count of subpatterns of x in current block with respect to its interpolated support over the past part of the stream.

The algorithm will thus deduce the *unknown* support $\sigma_{[1,i]}(x)$ of itemset x on the part of the stream preceding the i^{th} block as follows:

$$\sigma_{[1,i]}(x)^{\text{interp}} = \sigma_i(x) * \min \left(\left\{ \min \left\{ \frac{\sigma_{[1,i]}(\text{item})}{\sigma_i(\text{item})}, \frac{\sigma_{[1,i]}(x \setminus \text{item})^{\text{interp}}}{\sigma_i(x \setminus \text{item})} \right\} \mid \text{item} \in x \right\} \right)$$

In the previous formula the result of the first min is the minimum among the ratios of supports of items contained in pattern x in past and recent data and the same values computed for itemsets obtained from x by removing one of its items. Note that during the processing of recent data, the search space is visited levelwise and the merge of the results is performed starting from shorter pattern too. Hence the interpolated supports $\sigma_{[1,i]}(x \setminus \text{item})^{\text{interp}}$ of all $k - 1$ -subpatterns of a k -pattern x are known. In fact each support can be either known from the processing of the past part of the stream or computed during the previous iteration on recent data.

Example of interpolation. Suppose that we have received 440 transactions so far, and that 40 of these are in the current block. The itemset $\{A, B, C\}$, briefly indicated as ABC , is frequent locally whereas it was unfrequent in previous data. Table 1 reports the support of every subpattern involved in the computation. The first column contains the patterns in compressed notation, the second and third columns contain the supports of the patterns in the last received block and in the past part of the stream. Finally the last column shows the reduction ratio for each pattern.

The algorithm looks among itemsets of size two and single items for the one having the minimum ratio. In this case the minimum is 2.5, corresponding to the subpattern $\{A, C\}$. Since in recent data the support of itemset $x = \{A, B, C\}$ is $\sigma_i(x) = 6$, the interpolated support will be $\sigma_{[1,i]}(x)^{interp} = 6 \cdot 2.5 = 15$

x	$\sigma_i(x)$	$\sigma_{[1,i]}(x)^{interp}$	$\frac{\sigma_{[1,i]}(x)^{interp}}{\sigma_i(x)}$
ABC	6	?	?
AB	8	50	6.2
AC	12	30	2.5
BC	10	100	10
A	17	160	9.4
B	14	140	10
C	18	160	8.9
{}	40	400	-

Table 1. Sample supports and reduction ratios.

It is worth remarking that this method works if the support of larger itemsets decreases similarly in most parts of the stream, so that a reduction factor (different for each pattern) can be used to interpolate unknown values. Finally note that, as regards the interpolated value above, we expect that the following inequality should hold: $\sigma_{[1,i]}(x)^{interp} < \text{minsup} \cdot |\mathcal{D}_{[1,i]}|$. So, if we obtain it is not satisfied, this interpolated result should not be accepted. If it was true, the exact value $\sigma_i(x)$ should have already been found. Hence, in those few cases where the above inequality does not hold, the interpolated value will be: $\sigma_{[1,i]}(x)^{interp} = (\text{minsup} \cdot |\mathcal{D}_{[1,i]}|) - 1$.

Obviously several other way of computing a support interpolation could be devised. Some are really simple as the average of the bounds while others are complex as counting inference, used in a different context in [16]. We chose this particular kind of interpolation because it is simple to calculate, since it is based on data that we already maintain for other purposes, and it is aware of the underlying data enough to allow for accurate handling of datasets characterized by data-skew on item distributions among different blocks.

We can finally introduce the pseudo-code of AP_{Stream} . As in **Streaming Partition** the transactions are received and buffered. **DCI**, the algorithm used for the local computations, is able to exactly know the amount of memory required for mining a dataset during the intersection phase. Since frequent patterns are processed sequentially and can be offloaded to disk, the memory needed for efficient computation of frequent patterns is just that used by the bitmap representing the vertical dataset and can be computed knowing the number of transactions and the number of frequent items.

Thus we can use this knowledge in order to maximize the size of the block of transactions processed at once. For the sake of simplicity we will neglect the quite obvious main loop with code related to buffering, concentrating on the processing of each data block. The interpolation formula has been omitted too for the same reason.

Each block is processed, visiting the search space level-wise, for discovering frequent patterns. In this way itemsets are sorted according to their length and the interpolated support for frequent subpattern is always available when required. The processing of patterns of length k is performed in two step. First frequent patterns are computed in the current block and then the actual insertion into the current set of frequent patterns is carried out. When a pattern is found to be frequent in the current block its support on past data is immediately checked: if it was already known then the local support is summed to previous support and previous bounds. Otherwise a support and a pair of bounds are inferred for past data and summed to the support in the current block. In both cases, if the resulting support pass the support test, the pattern is queued for delayed insertion. After every locally frequent pattern of the current length k has been processed, the support of every previously known pattern which is not locally frequent is computed on recent data. Patterns passing the support test are queued for delayed insertion too. Then the set of pre-inserted itemsets is sorted and the actual insertion take place.

2.1 Tighter bounds

As a consequence of using an interpolation method to guess an approximate support value in the past part of the stream, it is very important to establish some bounds on the support found for each pattern. In the previous subsection we have already indicated a pair of really loose bounds: each support can not be negative, and if a pattern was not frequent in a time interval then its interpolated support should be less than the minimum support threshold for the same interval.

```

processBlock(buffer, globFreq)
  locFreq[1] = <frequent items>;
  k = 2;
  while size(locFreq[k - 1]) >= k do
    locFreq[k] = computeFrequent(k, locFreq, globFreq);
    commitInsert(k, locFreq, globFreq);
  end while
end;

commitInsert(k, locFreq, globFreq)
  for all pat in globFreq[k] and not in locFreq[k] do
    <count support of pat in recent data>
    if <pat is frequent> then
      <pre-insert pat in globFreq[k]>
    end if
  end for
  <update globFreq>
end;

computeFrequent(k, locFreq, globFreq)
  < compute local frequent pattern >
  for all pat locally frequent do
    <compute global interpolated support and bounds>
    if <pat is frequent> then
      <insert pat in locFreq[k]>
      <pre-insert pat in globFreq[k]>
    end if
  end for
  return Fk;
end;

```

Fig. 1. AP_{Stream} pseudo-code.

This criteria is completely true for non-evolving distributed dataset (*distributed frequent pattern mining*) or for the first two data block of the stream. In the stream case the upper bound is based on previous approximate results and could be inexact if the pattern corresponds to a false negative. Nevertheless it does represent a useful indication.

Bounds based on pattern subset The first bounds that interpolated supports should obey derive from the *Apriori property*: no set can have a support greater than those of any of its subset. Since recent results are merged level-wise with previously known ones, the interpolation can exploit already interpolated subset support. When a subpattern is missing during interpolation this mean that it has been examined during a previous level and discarded. In that case all of its superset may be discarded as well. The computed bound is thus affected by the approximation of past results: a pattern with an erroneous support will affect the bounds for each of its superset. To avoid this issue it is possible to compute the upper bound for a pattern x using the upper bounds of its sub-patterns instead of their support. In this way the upper bounds will be weaker but there will be less false negatives due to erroneous bounds enforcement.

Bounds based on transaction hash In order to address the issue of error propagation in support bounds we need to devise some other kind of bounds which are computed exclusively from received data and thus are independent of any previous results. Such bounds can be obtained using inverted transaction hashes. This technique was first introduced in the algorithm IHP [7], an association mining algorithm where it was used for finding an upper bound for the support of candidates in order to prune infrequent ones. As we will show this method can be used also for lower bounds. The key idea is that each item has an associated hashed set of counters which are accessed by using *transaction id* as a key. More in detail, each array $hcnt[item]$ associated with an *item* is an array of $hsize$ counters initialized to zero. When the t_{id}^{th} transaction $t = \{t_i\}$ is processed a hash function transforms the t_{id} value into an index to be used for the array of counters. Since t_{ids} are consecutive integer numbers, a trivial hash function as $h(tid) = tid \bmod hsize$ will guarantee a equal repartition of transactions among all hash bins. For each item $t_i \in t$ the counter at position $h(tid)$ in the array $hcnt[t_i]$ is incremented.

Let $hsize = 1$, A and B two items and $h_A = hcnt[A][0]$ and $h_B = hcnt[B][0]$ the only counters contained in their respective hashes, i.e. h_A and h_B are the number of occurrences of items A and B in the whole dataset. According to the Apriori principle the support $\sigma(\{A, B\})$ for the pattern $\{A, B\}$ can be at most equal to $\min(h_A, h_B)$. Furthermore we are able to indicate a lower bound for the same support. Let $n[i]$ be the number of transactions associated with the i^{th} hash position, which, in this case, corresponds to the total number of transactions n . We know from the inclusion/exclusion principle that $\sigma(\{A, B\})$ should be greater than or at least equal to $\max(0, h_A + h_B - n)$. In fact if $n - h_A$ transactions does not contains the item A then at least $h_B - (n - h_A)$ of the h_B transactions

containing B will also contain A . Suppose that $n = 10, h_A = 8, h_B = 7$. If we represent with an X each transaction supporting a pattern and with a dot any other transaction we obtain the following diagrams:

	Best case (ub(AB)= 7)	Worst case (lb(AB)=5)
A:	XXXXXXXX..	XXXXXXXX..
B:	XXXXXXXX..	...XXXXXX
AB:	XXXXXXXX..	...XXXXX..

Then no more than 7 transactions will contain both A and B . At the same time at least $8+7-10 = 5$ transactions will satisfy that constraint. Since each counter represents a set of transaction, this operations is equivalent to the computation of the minimal and maximal intersections of the tid-lists associated with the single items.

Usually $hsize$ will be larger than one. In that case the previously explained computations will be applied to each hash position, yielding an array of lower bounds and an array of upper bounds. The sums of their elements will give the pair of bounds for pattern $\{A, B\}$ as we will show in the following example. Let $hsize = 3, h(tid) = tid \bmod hsize$ the hash function, A and B two items and $n[i] = 10$ be the number of transactions associated with the i^{th} hash position. Suppose that $hcnt[A] = \{8, 4, 6\}$ and $hcnt[B] = \{7, 5, 6\}$. Using the same notation previously introduced we obtain:

h(tid)=0		h(tid)=1		h(tid)=2	
Best case	Worst case	Best case	Worst case	Best case	Worst case
A: XXXXXXXX..	XXXXXXXXXX..	A: XXXX.....	XXXX.....	A: XXXXXX....	XXXXXX....
B: XXXXXXXX..	...XXXXXXX	B: XXXX.....XXXXX	B: XXXXXX....	...XXXXXX
AB: XXXXXXXX..	...XXXXXX..	AB: XXXX.....	AB: XXXXXX....XX....
supp	7	supp	4	supp	6
	5		0		2

Each pair of columns represents the transactions having a tid mapped into the corresponding location by the hash function. The lower and upper bounds for the support of pattern AB will be respectively $5 + 0 + 2 = 7$ and $7 + 4 + 6 = 17$.

Both lower bounds and upper bounds computations can be extended to larger itemsets by associativity: the bounds for to the first two items are composed with the third element counters and so on. The sums of the elements of the last pair of resulting arrays will be the upper and the lower bounds for the given pattern. This is possible since the reasoning previously explained still holds if we considers the occurrences of itemsets instead of those of single items. The lower bound computed in this way will be often equal to zero in sparse dataset. Conversely on dense datasets this method did proved to be effective in narrowing the two bounds.

3 Experimental evaluation

In this section we study the behavior of the proposed method. We have run the AP_{Stream} algorithm on several datasets using different parameters. The goal of these tests is to understand how similarities of the results vary as the stream length increase, the effectiveness of the hash based pruning, and, in general, how dataset peculiarities and invocation parameters affect the accuracy of the results. Furthermore, we studied how execution time evolves in time when the stream length increases.

Similarity and Average Support Range. The method we are proposing yields approximate results. In particular AP_{Stream} computes pattern supports which may be slightly different from the exact ones, thus the result set may miss some frequent pattern (false negatives) or include some infrequent pattern (false positives). In order to evaluate the accuracy of the results we use a widely used measure of similarity between two pattern sets introduced in [12], and based on support difference. To the same end, we use the Average support Range (ASR), an intrinsic measure of the correctness of the approximation introduced in [15].

Experimental data. We performed several tests using both real world datasets, mainly from the FIMI'03 contest [1], and synthetic dataset generated using the IBM generator. We randomly shuffled each dataset and used the resulting datasets as input streams.

Table 2 shows a list of these datasets along with their cardinality. The datasets having the name starting with T are synthetic datasets, which mimic the behavior of market basket transactions. The sparse dataset family T20I8N5k has transactions composed, on average, of 20 items, chosen from 5000 distinct items, and include maximal patterns whose average length is 8. The dataset family T30I30N1k was generated with the parameters synthetically indicated in its name and is a moderately dense, since more than 10,000 frequent patterns can be extracted even with a minimum support of 30%. A description of all other datasets can be found in [1]. Kosarak and Retail are really sparse datasets, whereas all other real world dataset used in experimental evaluation are dense. Table 2 also indicates for each dataset a short identifying code which will be used in our charts.

Experimental Results. For each dataset and several minimum support thresholds we computed the exact reference solutions by using DCI [17], an efficient sequential algorithm for frequent pattern mining (FPM). Then we ran AP_{Stream} for different values of available memory and number of hash entries.

The first test is focused on catching the effect of used memory on the behavior of the algorithm when the block of transactions processed at once is sized dynamically according to the available resources. In this case data are buffered as long as all the item counters, and the representation

of the transactions included in the current block fit into the available memory. Note that the size of all frequent itemsets, mined either locally or globally, is not considered in our resource evaluation, since they can be offloaded to disk if needed. The second test is somehow related to the previous one. In this case the amount of required memory is varied, since we determine a-priori the number of

Dataset	Reference	#Trans.
accidents	A	340183
kosarak	K	990002
retail	R	88162
pumbs	P	49046
pumbs-star	PS	49046
connect	C	67557
T20I8N5k	S2..6	77302..3189338
T25I20N5k	S7..11	89611..1433580
T30I30N1k	D1..D9	50000..3189338

Table 2. Datasets used in experimental evaluation.

transactions to include in a single block, independently of the stream content. Since the datasets used in the tests are quite different, in both cases we used really different ranges of parameters. Therefore, in order to fit all the datasets in the same plot, the number reported in the horizontal axis are relative quantities, corresponding to the block sizes actually used in each test. These relative quantities are obtained by dividing the memory/block size used in the specific test by the smallest one for that dataset. For example, the series 50KB, 100KB, 400KB thus becomes 1,2,8.

The first plot in figure 2 shows the results obtained in the fixed memory case, while the second one when the number of transactions per block is fixed. The relative quantities reported in the plots refer to different base values of either memory or transactions per blocks. These values are reported in the legend of each plot. In general when we increase the number of transaction processed at once, either statically or dynamically on the basis the memory available, we also improve the results similarity. Nevertheless the variation is in most cases small and sometimes there is also a slightly negative trend caused by the nonlinear relation between used memory and transactions per block. In our test we noted that choosing an excessively low amount of available memory for some datasets lead to performance degradation and sometimes also to similarity degradation. The last plot shows the effectiveness of the hash-based bounds on reducing the Average Support Range (zero corresponds to an exact result). As expected the improvement is evident only on more dense datasets.

The last batch of tests makes use of a family of synthetic datasets with homogeneous distribution parameters and varying lengths. These datasets are obtained from the larger dataset of the series by truncating it to simulate streams with different lengths. For each truncated dataset we computed the exact result set, used as reference value in computing the similarity of the corresponding approximate result obtained by AP_{Stream} . The first chart in figure 2 plots both similarity and ASR as the stream length increases. We can see that similarity remains almost the same, whereas the ASR decreases when an increasing amount of stream is processed. Finally, the last plot shows the evolution of execution time as the stream length increases. The execution time increases linearly with the length of the stream, hence the average time per transaction is constant if we fix the dataset and the execution parameters.

4 Conclusions

In this paper we have discussed AP_{Stream} , a new algorithm for approximate frequent pattern mining on streams. AP_{Stream} exploits a novel interpolation method to infer the unknown past counts of some patterns, which are frequent only on recent data. Since the support values computed by the algorithm are approximate, we have also proposed a method for establishing a pair of upper and lower bounds for each interpolated value. These bounds are computed using the knowledge of subpattern frequency in past data and the intersection of an hash based compressed representation of past data.

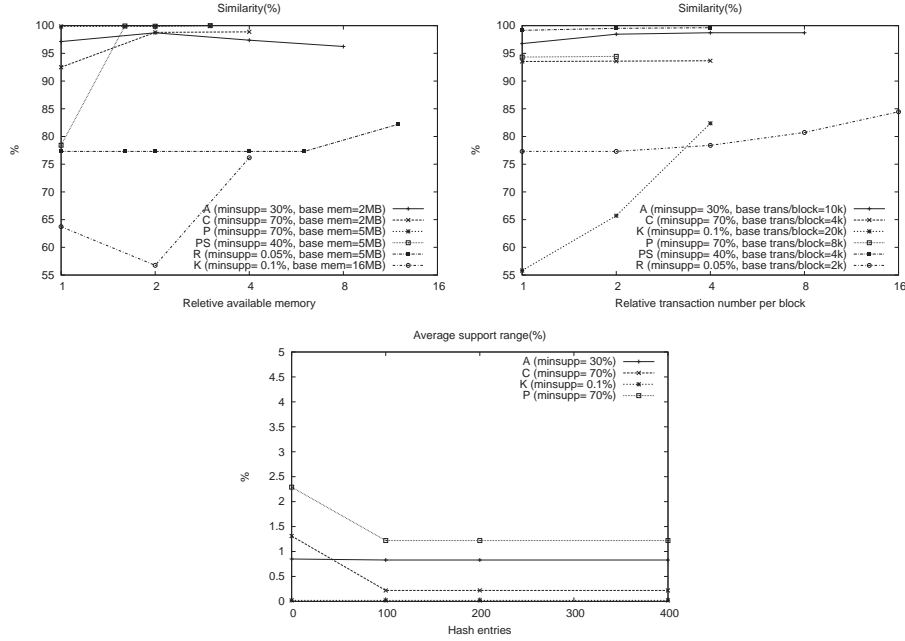


Fig. 2. Similarity as a function of available memory, number of transactions per block, number of hash entries.

The experimental tests show that the solution produced by AP_{Stream} is a good approximation of the exact global result. The comparisons with exact results consider both the set of patterns found and their support. The metric used in order to assess the quality of the algorithm output is the similarity measure introduced in [12], used along with the novel false positive aware similarity proposed in [15]. The interpolation works particularly well for dense datasets, achieving a similarity close to 100% in best cases. The adaptive behavior of AP_{Stream} allows us to limit the amount of used memory. As expected, we have found that a larger amount of available memory corresponds to a more accurate result. Furthermore, as the length of the processed stream increases, the similarity to the exact result remains almost the same. At the same time we observed a decrease in the average difference between upper and lower bounds, which is an intrinsic measure of result accuracy. Finally, the time needed to process a block of transactions does not depend on the stream length, hence the total execution time is linear with respect to the stream length. In the future we plan to improve the proposed method by adding other stricter bounds on the approximate support and to extend it to closed patterns.

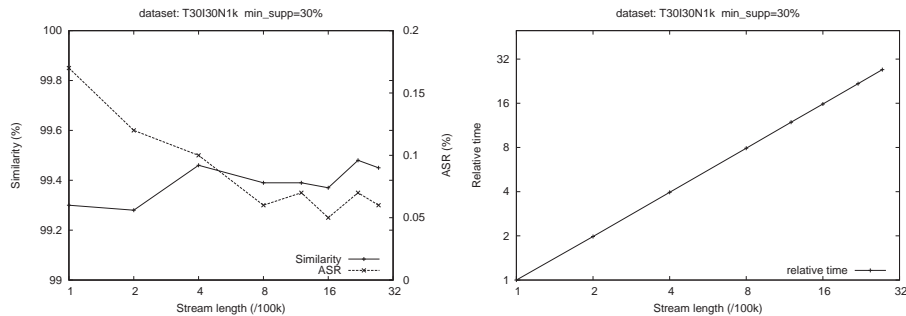


Fig. 3. Similarity and Average Support Range as a function of different stream lengths.

5 Acknowledgements

This work was partially supported by the PRIN'04 Research Project entitled "GeoPKDD - Geographic Privacy-aware Knowledge Discovery and Delivery". The datasets used during the experimental evaluation are some of those used for the FIMI'03 (Frequent Itemset Mining Implementations) contest [1]. We thank the owners of this data and people who made them available in current format. In particular Karolien Geurts [6] for *Accidents*, Ferenc Bodon for *Kosark*, Tom Brijs [4] for *Retail* and Roberto Bayardo for the conversion of *UCI* datasets. Synthetic datasets were generated using the publicly available synthetic data generator code from the IBM Almaden Quest data mining project.

References

1. Workshop on frequent itemset mining implementations in conjunction with ICDM'03. In *fimi.cs.helsinki.fi*, 2003.
2. A.Savasere, E.Omicinski, and S.B.Navathe. An efficient algorithm for mining association rules in large databases. In *VLDB'95, Proc. of 21th Int. Conf. on Very Large Data Bases*, pages 432–444. Morgan Kaufmann, September 1995.
3. B.Babcock, S.Babu, M.Datar, R.Motwani, and J.Widom. Models and issues in data stream systems. In *Proc. of the 21st ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 1–16. ACM Press, 2002.
4. Tom Brijs, Gilbert Swinnen, Koen Vanhoof, and Geert Wets. Using association rules for product assortment decisions: A case study. In *Knowledge Discovery and Data Mining*, pages 254–260, 1999.
5. U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI Press, 1998.
6. Karolien Geurts, Geert Wets, Tom Brijs, and Koen Vanhoof. Profiling high frequency accident locations using association rules. In *Proceedings of the 82nd Annual Transportation Research Board, Washington DC. (USA), January 12-16*, page 18pp, 2003.
7. John D. Holt and Soon M. Chung. Mining association rules using inverted hashing and pruning. *Inf. Process. Lett.*, 83(4):211–220, 2002.

8. R. Jin and G. Agrawal. An algorithm for in-core frequent itemset mining on streaming data. Submitted for publication, July 2003, 2003.
9. J.Misra and D.Gries. Finding repeated elements. Technical report, Ithaca, NY, USA,, 1982.
10. Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28(1):51–55, 2003.
11. G. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases*, August 2002.
12. Srinivasan Parthasarathy. Efficient progressive sampling for association rules. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, page 354. IEEE Computer Society, 2002.
13. R.Agrawal, T.Imielinski, and A.N.Swami. Mining association rules between sets of items in large databases. In *Proc. of the 1993 ACM SIGMOD Int. Conf. on Management of Data*, pages 207–216, Washington, D.C., 1993.
14. C. Silvestri and S. Orlando. Approximate mining of frequent patterns on streams. Technical report, Venice, Italy,, 2005.
15. C. Silvestri and S. Orlando. Distributed approximate mining of frequent patterns. In *Proceedings of ACM Symposim on Applied Computing SAC*, March 2005,.
16. S.Orlando, P.Palmerini, R.Perego, C.Lucchese, and F.Silvestri. kdci: a multi-strategy algorithm for mining frequent sets. In *Proc. of the Int. Workshop on Frequent Itemset Mining Implementations in conjunction with ICDM'03*, 2003.
17. S.Orlando, P.Palmerini, R.Perego, and F.Silvestri. Adaptive and resource-aware mining of frequent sets. In *Proc. of the 2002 IEEE Int. Conf. on Data Mining, ICDM*, 2002.