# SaM: A Split and Merge Algorithm for Fuzzy Frequent Item Set Mining

Christian Borgelt[1] and Xiaomeng Wang[2]

1. European Center for Soft Computing, c/ Gonzalo Gutiérrez Quirós s/n, 33600 Mieres, Spain
2. Otto-von-Guericke-University of Magdeburg, Universitätsplatz 2, 39106 Magdeburg, Germany
Email: christian.borgelt@softcomputing.es, xwang@iws.cs.uni-magdeburg.de

***Abstract*—*This paper presents SaM, a split and merge algorithm for frequent item set mining. Its distinguishing qualities are an exceptionally simple algorithm and data structure, which not only render it easy to implement, but also convenient to execute on external storage. Furthermore, it can easily be extended to allow for "fuzzy" frequent item set mining in the sense that missing items can be inserted into transactions with a user-specified penalty. In order to demonstrate its performance, we report experiments comparing it with the "fuzzy" frequent item set mining version of RElim (an algorithm we suggested in an earlier paper [15] and improved in the meantime).*

***Keywords***— data mining, frequent item set mining, fuzzy frequent item set, fault tolerant data mining

## 1 Introduction

Although frequent item set mining and association rule induction has been a focus of research in data mining for a long time now, leading to well-known algorithms like Apriori [1], Eclat [11] and FP-growth [7], there is still room for improvement. Recent research lines include filtering the found frequent item sets and association rules [16, 17], identifying temporal changes in discovered patterns [3, 4], and mining fault-tolerant or "fuzzy" frequent item sets [6, 10, 15].

In this paper we follow the last of these lines by presenting SaM, a split and merge algorithm for frequent item set mining, which can easily be extended to allow for "fuzzy" mining in the sense that missing items can be inserted into transactions with a user-specified penalty. Other distinguishing qualities of our method are its exceptionally simple processing scheme and data structure, which not only render it very easy to implement, but also convenient to execute on external storage.

## 2 Frequent Item Set Mining

Frequent item set mining is the following task: we are given a set $B$ of *items*, called the *item base*, and a database $T$ of *transactions*. An item may, for example, represent a product, and the item base may then represent the set of all products offered by a supermarket. The term *item set* refers to any subset of the item base $B$. Each transaction is an item set and may represent, in the supermarket setting, a set of products that has been bought by a customer. Since several customers may have bought the exact same set of products, the total of all transactions must be represented as a vector or a multiset (or, alternatively, each transaction must be enhanced by a *transaction identifier (tid)*). Note that the item base $B$ is usually not given explicitly, but only implicitly as the union of all transactions. The *support* $s_T(I)$ of an item set $I \subseteq B$ is the number of transactions in the database $T$ it is contained in. Given a user-specified *minimum support* $s_{\min} \in \mathbb{N}$, an item set $I$ is called *frequent* (in $T$) iff $s_T(I) \geq s_{\min}$. The goal of frequent item set mining is to find all item sets $I \subseteq B$ that are frequent in the database $T$ and thus, in the supermarket setting, to identify all sets of products that are frequently bought together.

A standard approach to find all frequent item sets w.r.t. a given database $T$ and a support threshold $s_{\min}$, is a *depth-first search* in the subset lattice of the item base $B$. This approach can be seen as a simple *divide-and-conquer* scheme. For a chosen item $i$, the problem to find all frequent item sets is split into two subproblems: (1) find all frequent item sets containing $i$ and (2) find all frequent item sets *not* containing $i$. Each subproblem is then further divided based on another item $j$: find all frequent item sets containing (1.1) both $i$ and $j$, (1.2) $i$, but not $j$, (2.1) $j$, but not $i$, (2.2) neither $i$ nor $j$ etc.

All subproblems occurring in this recursion can be defined by a *conditional transaction database* and a *prefix*. The prefix is a set of items that has to be added to all frequent item sets that are discovered in the conditional database. Formally, all subproblems are tuples $S = (C, P)$, where $C$ is a conditional database and $P \subseteq B$ is a prefix. The initial problem, with which the recursion is started, is $S = (T, \emptyset)$, where $T$ is the given transaction database and the prefix is empty.

A subproblem $S_0 = (C_0, P_0)$ is processed as follows: choose an item $i \in B_0$, where $B_0$ is the set of items occurring in $C_0$. This choice is arbitrary, but usually follows some predefined order of the items. If $s_{C_0}(i) \geq s_{\min}$, then report the item set $P_0 \cup \{i\}$ as frequent with the support $s_{C_0}(i)$, and form the subproblem $S_1 = (C_1, P_1)$ with $P_1 = P_0 \cup \{i\}$. The conditional database $C_1$ comprises all transactions in $C_0$ that contain $i$, but with this item removed. This implies that all transactions are removed that do not contain any no other item than $i$. If $C_1$ is not empty, process $S_1$ recursively. In any case (that is, regardless of whether $s_{C_0}(i) \geq s_{\min}$ or not), form the subproblem $S_2 = (C_2, P_2)$ with $P_2 = P_0$. The conditional database $C_2$ comprises all transactions in $C_0$ (including those that do not contain $i$), but again with the item $i$ removed. If $C_2$ is not empty, process $S_2$ recursively.

This recursive scheme is adopted by Eclat, FP-growth, RElim and several other frequent item set mining algorithms. They differ in how conditional transaction databases are represented: in a *horizontal representation*, a database is stored as a list (or array) of transactions, each of which lists the items contained in it. In a *vertical representation*, the items are first referred to with a list (or array) and for each item the transactions containing it are listed. However, this distinction is not pure, since there are combinations of the two forms of representing a database. Our SaM algorithm is, as far as we know, the first algorithm that is based on the general scheme outlined above and uses a purely horizontal representation.

ISBN: 978-989-95079-6-8

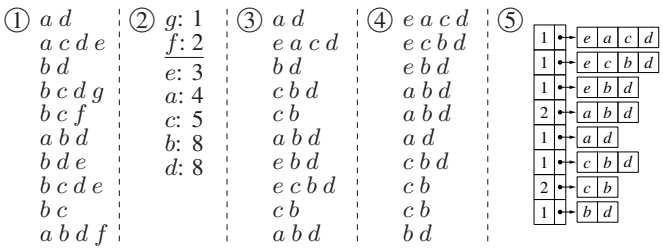


Figure 1: An example database: original form (1), item frequencies (2), transactions with sorted items (3), lexicographically sorted transactions (4), and the used data structure (5).

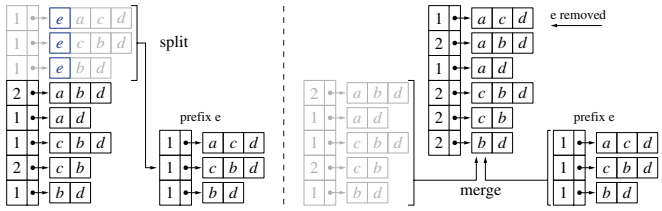


Figure 2: The basic operations: split (left) and merge (right).

The basic processing scheme can be improved with so-called *perfect extension pruning*: an item $i \notin I$ is called a *perfect extension* of an item set $I$, iff $I$ and $I \cup \{i\}$ have the same support. Perfect extensions have the following properties: (1) if an item $i$ is a perfect extension of an item set $I$, then it is also a perfect extension of any item set $J \supseteq I$ as long as $i \notin J$ and (2) if $K$ is the set of all perfect extensions of an item set $I$, then all sets $I \cup J$ with $J \in 2^K$ (where $2^K$ denotes the power set of $K$) have the same support as $I$. These properties can be exploited by collecting in the recursion not only prefix items, but also, in a third element of a subproblem description, perfect extension items. They are also removed from the conditional databases and are only used to generate all supersets of the prefix that have the same support.

## 3 A Simple Split and Merge Algorithm

In this section we describe the basic form of our SaM (split and merge) algorithm. Preprocessing is very similar to many other frequent item set mining algorithms. The steps are illustrated in Figure 1 for a simple example transaction database: step 1 shows the transaction database in its original form. In step 2 the item frequencies are determined in order to discard infrequent items. With a minimum support of 3, items $f$ and $g$ are infrequent and thus eliminated. In step 3 the (frequent) items in each transaction are sorted according to their frequency, because processing the items in the order of increasing frequency usually leads to the shortest execution times. In step 4 the transactions are sorted lexicographically into descending order, with an item with higher frequency preceding an item with lower frequency. In step 5 the basic data structure is built by combining equal transactions and setting up an array, in which each element consists of two fields: an occurrence counter and a pointer to the sorted transaction.

The basic operations of the recursive processing, which follows the general divide-and-conquer scheme reviewed in Section 2, are illustrated in Figure 2: in the *split step* (left) the given array is split w.r.t. the leading item of the first transaction (item $e$ in our example): all elements referring to transactions starting with this item are transferred to a new array. In this process the pointer (in)to the transaction is advanced by one item, so that the common leading item is "removed" from all transactions. Obviously, this new array represents the conditional database of the first subproblem (see Section 2), which is then processed recursively to find all frequent item sets containing the split item (provided this item is frequent).

The conditional database for frequent item sets *not* containing this item (second subproblem, see Section 2) is obtained with a simple *merge step* (right part of Figure 2). The new array and the rest of the original array are combined with a procedure that is almost identical to one phase of the well-known *mergesort* algorithm. Since both arrays are lexicographically sorted, one merging traversal suffices to create a lexicographically sorted merged array. The only difference to a *mergesort* phase is that equal transactions (or transaction suffixes) are combined: There is always only one instance of each transaction (suffix), while its number of occurrences is kept in a counter. In our example this results in the merged array having two elements less than the input arrays together: the transaction (suffixes) $cbd$ and $bd$, which occur in both arrays, are combined and their occurrence counters are increased to 2.

Pseudo-code of SaM is shown in Figure 3: a single page of code suffices to describe the whole recursion in detail. The actual C code is even shorter, despite the fact that it contains additional functionality (like perfect extension pruning, Section 2), because certain operations can be written very concisely in C (especially when using pointer arithmetic).

## 4 Fuzzy Frequent Item Set Mining

There are many applications of frequent item set mining, in which the transactions do not contain all items that are actually present. However, standard algorithms are based on exact matching and therefore are not equipped to meet the needs arising in such applications. An example is the analysis of alarm sequences in telecommunication networks, where a core task is to find collections of alarms occurring frequently together, so-called *episodes*. One approach to accomplish this task is to slide a time window over the alarm sequence. Each window position then captures a specific slice of the alarm sequence [12]. The underlying idea is that in this way the problem of finding frequent episodes is reduced to that of finding frequent item sets in a database of transactions: each alarm can be seen as an item and the alarms in a time window as a transaction. The support of an episode is the number of window positions, so that the episode occurred in the window.

Unfortunately, alarms often get delayed, lost, or repeated due to noise, transmission errors, failing links etc. If alarms do not get through or are delayed, they are missing from the transaction (time window) its associated items (alarms) occur in. If we required exact containment of an item set in this case, the support of some item sets, which would be frequent if the items did not get lost, may be lower than the user-specified minimum. This leads to a possible loss of potentially interesting frequent item sets and to distorted support values.

To cope with such missing information, we rely on the notion of a "fuzzy" or approximate frequent item set. In contrast to research on fuzzy association rules (see, for example, [13]),

ISBN: 978-989-95079-6-8

```
function SaM (a: array of transactions,    (* conditional database *)
              p: set of items,             (* prefix of the cond. database a *)
              s_min: int) : int            (* min. support of an item set *)
var i: item;                               (* buffer for split item *)
    s: int;                                (* support of current split item *)
    n: int;                                (* number of frequent item sets *)
    b, c, d: array of transactions;        (* cond. and merged database *)
begin                                      (* – split and merge recursion – *)
  n := 0;                                  (* init. number of freq. item sets *)
  while a is not empty do                  (* while database is not empty *)
    b := empty; s := 0;                    (* init. split result, item support *)
    i := a[0].items[0];                    (* get the leading item *)
    while a is not empty                   (* of the first transaction and *)
    and   a[0].items[0] = i do             (* split database w.r.t. this item *)
      s := s + a[0].wgt;                   (* sum occurrences (support) *)
      remove i from a[0].items;            (* remove the split item *)
      if    a[0].items is not empty        (* if trans. is not empty *)
      then remove a[0] from a and append it to b;
      else remove a[0] from a; end;        (* move it to cond. db., *)
    end;                                   (* otherwise simply remove it *)
    c := b; d := empty;                    (* initialize the output array *)
    while a and b are both not empty do    (* merge step *)
      if    a[0].items > b[0].items        (* copy trans. from a *)
      then remove a[0] from a and append it to d;
      else if a[0].items < b[0].items      (* copy trans. from b *)
      then remove b[0] from b and append it to d;
      else b[0].wgt := b[0].wgt +a[0].wgt;
           remove b[0] from b and append it to d;
           remove a[0] from a;             (* combine weights and *)
      end;                                 (* move and remove trans.: *)
    end;                                   (* only one instance per trans. *)
    while a is not empty do                (* copy the rest of a *)
      remove a[0] from a and append it to d; end;
    while b is not empty do                (* copy the rest of b *)
      remove b[0] from b and append it to d; end;
    a := d;                                (* loop for second recursion *)
    if s ≥ s_min then                      (* if the split item is frequent: *)
      p := p ∪ {i};                        (* extend the prefix item set and *)
      report p with support s;             (* report the frequent item set *)
      n := n + 1 + SaM(c, p, s_min);
      p := p − {i};                        (* process cond. db. recursively, *)
    end;                                   (* sum the frequent item sets, *)
  end;                                     (* then restore the orig. prefix *)
  return n;                                (* return num. of freq. item sets *)
end;    (* function SaM() *)
```

Figure 3: Pseudo-code of the SaM algorithm.

where a fuzzy approach is used to handle quantitative items, we use the term "fuzzy" to refer to an item set that may not be present exactly in all supporting transactions, but only approximately. Related work in this direction suggested Apriori-like algorithms and mining with approximate matching was performed by counting the number of different items in the two item sets to be compared [6, 10]. However, here we adopt a more general scheme, based on an approximate matching approach exhibiting much greater flexibility. Our approach has two core ingredients: *edit costs* and *transaction weights* [15].

**Edit costs:** A convenient way of defining the distance between item sets is to consider the costs of a cheapest sequence of edit operations needed to transform one item set into the other [14]. Here we consider only insertions, since they are most easily implemented with our algorithm[1]. With the help of an *insertion cost* or *penalty* a flexible and general framework for approximately matching two item sets can be established. How one interprets such costs or penalties depends, of course, on the application. Note also that different items may be associated with different costs. For example, in telecommunication networks different alarms can have a different probability of getting lost: usually alarms raised in lower levels of the module hierarchy get lost more easily than alarms originating in higher levels. In such cases it is convenient to be able to associate the former with lower insertion costs than the latter. Insertions of a certain item may also be completely inhibited by assigning a very high insertion cost.

**Transaction weights:** Each transaction $t$ is associated with a weight $w(t)$, the initial value of which is 1. If an item $i$ is inserted into a transaction $t$, the transaction weight is "penalized" with a cost $c(i)$ associated with the item. Formally, this can be described as applying a combination function: the new weight of the transaction $t$ after inserting an item $i \notin t$ is $w_{\{i\}}(t) = f(w(t), c(i))$, where $f$ is a function that combines the weight $w(t)$ before editing and the insertion cost $c(i)$. The combination function $f$ depends, of course, on the application and may be chosen from a wide range of possible functions. For example, any $t$-norm may be used. We choose multiplication here, that is, $w_{\{i\}}(t) = w(t) \cdot c(i)$, mainly for reasons of simplicity. Note, however, that with this choice lower values of $c(i)$ mean higher costs as they penalize the weight more, but that it has the advantage that it is easy to extend to inserting several items: $w_{\{i_1,\ldots,i_m\}}(t) = w(t) \cdot \prod_{k=1}^{m} c(i_k)$. It should be clear that it is $w_\emptyset(t) = 1$ due to the initial weighting $w(t) = 1$.

How many insertions into a transaction are allowed may be limited by a user-specified lower bound $w_{\min}$ for the transaction weight. If the weight of a transaction falls below this threshold, it is not considered in further mining steps. Of course, this weight may also be set to zero (unlimited insertions). As a consequence, the *fuzzy support* of an item set $I$ w.r.t. a transaction database $T$ can be defined as $s_T^{(\text{fuzzy})}(I) = \sum_{t \in T} \tau(w_{I-t}(t) \geq w_{\min}) \cdot w_{I-t}(t)$, where $\tau(\phi)$ is a kind of "truth function", which is 1 if $\phi$ is true and 0 otherwise.

Note that SaM is particularly well suited to handle item insertions, because its purely horizontal transaction representation makes it easy to incorporate transaction weights. With other algorithms, more effort is usually needed in order to extend them to approximate frequent item set mining.

For an implementation, it is beneficial to distinguish between unlimited item insertions ($w_{\min} = 0$) and limited item insertions ($w_{\min} > 0$). If $w_{\min} = 0$, it is possible to combine equal transactions (or transaction suffixes) without restriction: two equal transactions (or suffixes) $t_1$ and $t_2$ with weights $w_1$ and $w_2$, respectively, can be combined into one transaction (suffix) $t$ with weight $w_1 + w_2$ even if $w_1 \neq w_2$. If another item $i$ needs to be inserted into $t_1$ and $t_2$ in order to make them contain a given item set $I$, the distributive law (that is, $w_1 \cdot c(i) + w_2 \cdot c(i) = (w_1 + w_2) \cdot c(i)$) ensures that we still compute the correct support for the item set $I$. If, however, we have $w_{\min} > 0$ and, say, $w_1 > w_2$, then using $(w_1 + w_2) \cdot c(i)$ as the contribution of the combined transaction $t$ to the support of the item set $I$ may be wrong, because it may be that $w_1 \cdot c(i) \geq w_{\min}$, but $w_2 \cdot c(i) < w_{\min}$. Then the support contributed by the two transactions $t_1$ and $t_2$ would rather be $w_1 \cdot c(i)$. Effectively, transaction $t_2$ does not contribute, since its weight has fallen below the transaction weight threshold. Hence, with limited insertions, we may combine equal transactions (or suffixes) only if they have the same weight.

[1] Note that deletions are implicit in the mining process (as we search for *subsets* of the transactions). Only replacements are an additional case we do not consider here.

ISBN: 978-989-95079-6-8

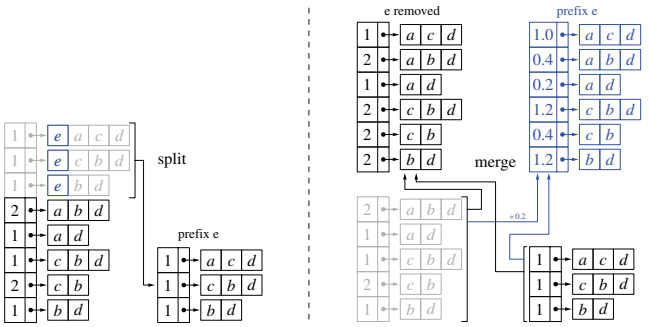


Figure 4: Unlimited item insertions, first recursion level.

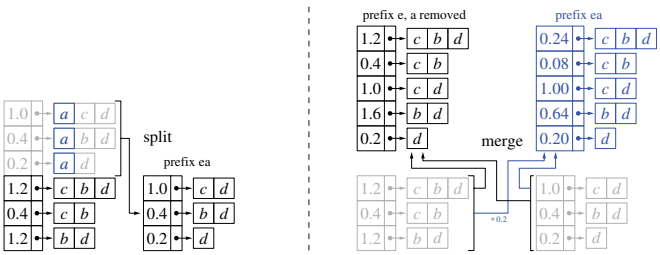


Figure 5: Unlimited item insertions, second recursion level.

## 5 Unlimited Item Insertions

If unlimited item insertions are possible ($w_{\min} = 0$), only a minor change of the data structure is needed: the integer occurrence counter for the transactions (or suffixes) has to be replaced by a real-valued transaction weight. In the processing, the split step stays the same (see Figure 4 on the left), but now it only yields an intermediate database with all transactions (or suffixes) that actually contain the split item under consideration (item $e$ in the example). In order to form the full conditional database, we have to add those transactions that do not contain the split item, but can be made to contain it by inserting it. This is achieved in the merge step, in which two parallel merge operations are carried out now (see Figure 4 on the right). The first part (shown in black) is the merge that yields the conditional database for frequent item sets *not* containing the split item. The second part (shown in blue/grey) adds those transactions that do not contain the split item, weighted down with the insertion penalty, to the intermediate database created in the split step. Of course, this second part of the merge operation is only carried out, if $c(i) > 0$, where $i$ is the split item, because otherwise no support would be contributed by the transactions not containing the item $i$ and hence it would not be necessary to add them. In such a case the result of the split step would already yield the conditional database for frequent item sets containing the split item.

Note that in both merge operations equal transactions (or suffixes) can be combined regardless of their weights. As a consequence we have in Figure 4 entries like for the transaction (suffix) $cbd$, with a weight of 1.2, which stands for one occurrence with weight 1 and one occurrence with weight 0.2 (due to the penalty factor 0.2, needed due to the insertion of item $e$). As an additional illustration, Figure 5 shows the split and merge operations for the second recursion level.

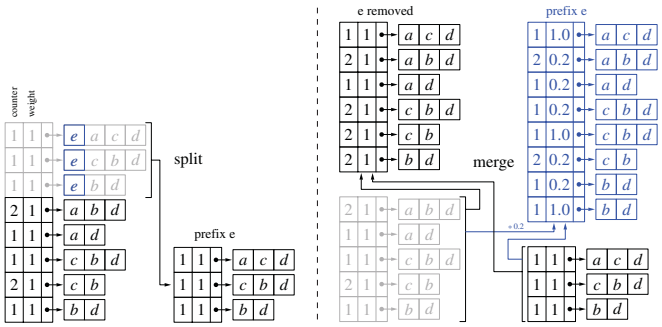


Figure 6: Limited item insertions, first recursion level.

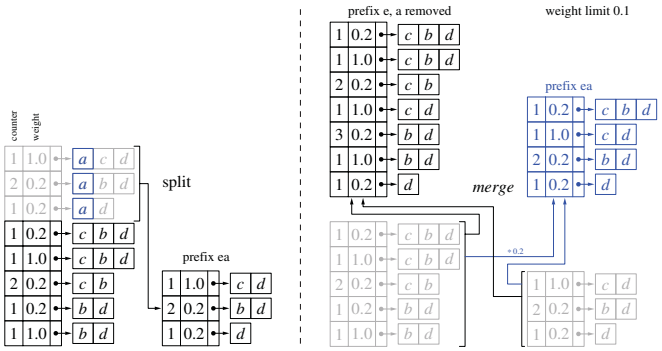


Figure 7: Limited item insertions, second recursion level.

## 6 Limited Item Insertions

If item insertions are limited by a transaction weight threshold ($w_{\min} > 0$), the transaction weight has to be represented explicitly and kept separate from the number of occurrences. Therefore the data structure must comprise, per transaction (suffix), (1) a pointer to the item array, (2) an integer occurrence counter, and (3) a real-valued transaction weight. The last field will be subject to a thresholding operation by $w_{\min}$, which eliminates all transactions with a weight less than $w_{\min}$. Hence there may now be array elements that refer to the same transaction (suffix), but differ in the transaction weight.

The processing scheme is illustrated in Figure 6. The split step is still essentially the same. However, the merge step differs due to the fact that equal transactions (or suffixes) can no longer be combined if their weight differs. As a consequence, there are now, in the result of the second merge operation (shown in blue) two array elements for $cbd$ and two for $bd$, which carry different weights. This is necessary, because they may reach, due to item insertions, the transaction weight threshold at different times and thus cannot be combined.

That transactions are discarded due to the weight threshold rarely happens on the first level of the recursion. (This can occur only if the insertion penalty factor of the split item is smaller than the transaction weight threshold, which is equivalent to inhibiting insertions of this item altogether). Therefore, in order to illustrate how transactions are discarded, Figure 7 shows the second recursion level, where the conditional database with prefix $e$ is processed. Here the second merge operation actually discards transactions if we set a transaction weight limit of 0.1: all transactions, which need two items (namely both $e$ and $a$) to be inserted, are not copied.

ISBN: 978-989-95079-6-8

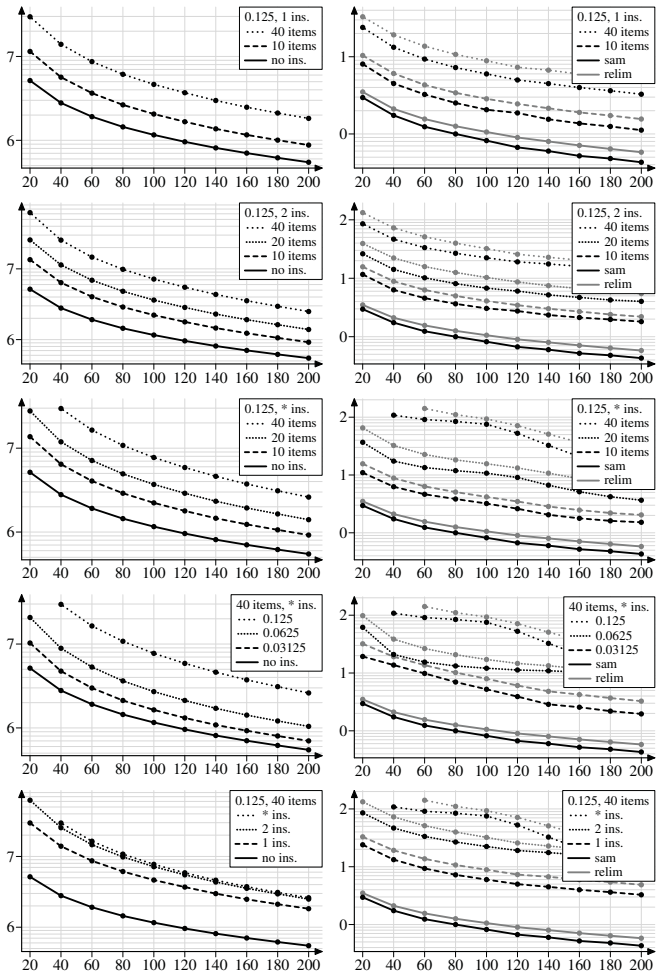


Figure 8: Experimental results on the Census (Adult) data; left: frequent item sets, right: execution times.

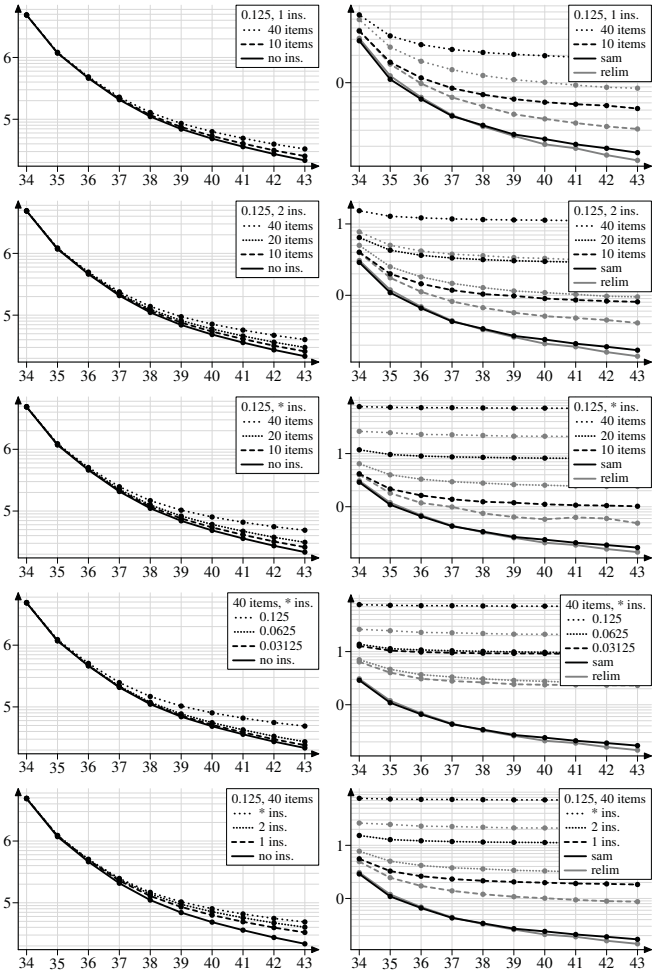


Figure 9: Experimental results on the BMS-Webview-1 data; left: frequent item sets, right: execution times.

## 7 Experiments

We ran experiments on several data sets, of which we chose two for this paper: Census (aka Adult, a data set derived from an extract of the US census bureau data of 1994, which was preprocessed by discretizing numeric attributes [2]) and BMS-Webview-1 (a web click stream from a leg-care company that no longer exists, which has been used in the KDD cup 2000 [8]). We chose these data sets, because Census is rather dense (a rather large fraction of all items occur in each transaction), while BMS-Webview-1 is rather sparse, and SaM and RElim [15] (the two algorithms of which we have implementations that can find approximate frequent item sets) exhibit a significantly different behavior on dense and sparse data sets.

The results are shown in Figure 8 for the census data set and in Figure 9 for the BMS-Webview-1 data set. In both figures the diagrams on the left show the decimal logarithm of the number of found frequent item sets, while the diagrams on the right show the decimal logarithm of the execution times (in seconds) for our implementations of SaM and RElim.[2] We tested insertion penalty factors of $\frac{1}{8} = 0.125$, $\frac{1}{16} = 0.0625$, and $\frac{1}{32} = 0.03125$, non-vanishing insertion penalty factors for 10, 20, and 40 items, and transaction weight thresholds that allowed for 1, 2 or an unlimited number of insertions.

As can be seen from the diagrams on the left of each figure, the two data sets react very differently to the possibility of inserting items into transactions. While the number of found frequent item sets rises steeply with all parameters for Census, it rises only very moderately for BMS-Webview-1, with the factor even leveling off for lower support values. As it seems, this effect is due, to a large degree, to the sparseness of BMS-Webview-1 (this still needs closer examination, though).

SaM fares considerably better on the dense data set (Census), beating RElim by basically the same margin (factor) in all parameter settings, while SaM is clearly outperformed by RElim on the sparse data set (BMS-Webview-1), even though the two algorithms are actually on par without item insertion (solid lines). On both data sets, the number of insertions that are allowed has the strongest influence: with two insertions execution times are about an order of magnitude larger than with only one insertion. However, the possibility to combine equal transactions with different weights still seems to keep the execution times for unlimited insertions within limits.

The number of items with a non-vanishing penalty factor and the value of the penalty factor itself seem to have a similar influence: doubling the number of items leads to roughly the same effect as keeping the number the same and doubling the

[2]Execution times were measured on an Intel Core 2 Quad Q9300 machine with 3 GB of main memory running openSuSE Linux 11.0 (32 bit) and gcc version 4.3.1.

ISBN: 978-989-95079-6-8

penalty factor. This is plausible, since there should not be much difference in having the possibility to insert twice the number of items or preserving twice the transaction weight per item insertion. Note, however, that doubling the penalty factor from from $\frac{1}{32}$ to $\frac{1}{16}$ has only a comparatively small effect on the BMS-Webview-1 data compared to doubling from $\frac{1}{16}$ to $\frac{1}{8}$. On the census data set the effects are a bit more in line.

Overall it should be noted that the execution times, although considerably increased over those obtained without item insertions, still remain within acceptable limits. Even with 40 items having an insertion penalty factor of $\frac{1}{8}$ and unlimited insertions, few execution times exceed 180 seconds ($\log_{10}(180) \approx 2.25$). In addition, we can observe the interesting effect on the BMS-Webview-1 data that at the highest parameter settings the execution times become almost independent of the minimum support threshold.

## 8 Conclusions

In this paper we presented a very simple split and merge algorithm for frequent item set mining, which, due to the fact that it uses a purely horizontal transaction representation, lends itself well to an extension to "fuzzy" or approximate frequent item set mining. In addition, it is a highly recommendable method if the data to mine cannot be loaded into main memory and thus the data has to be processed on external storage or in a (relational) database system. As our experimental results show, the SaM algorithm performs the task of "fuzzy" frequent item set mining excellently on the dense Census data, but shows certain weaknesses on the sparse BMS-Webview-1 data. However, our experiments provided some evidence (to be substantiated on more data sets) that "fuzzy" frequent item set mining is much more useful for dense data sets as more additional frequent item sets can be found. Hence SaM performs better in the (likely) more relevant case. Most importantly, however, one should note that with both SaM and RElim the execution times remain bearable.

**Software:** SaM and RElim sources in C can be found at:

http://www.borgelt.net/sam.html
http://www.borgelt.net/relim.html

### References


[1] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. *Proc. 20th Int. Conf. on very Large Databases (VLDB 1994, Santiago de Chile)*, 487–499. Morgan Kaufmann, San Mateo, CA, USA 1994

[2] C.L. Blake and C.J. Merz. *UCI Repository of Machine Learning Databases*. Dept. of Information and Computer Science, University of California at Irvine, CA, USA 1998

[3] M. Böttcher, M. Spott and D. Nauck. Detecting Temporally Redundant Association Rules. *Proc. 4th Int. Conf. on Machine Learning and Applications (ICMLA 2005, Los Angeles, CA)*, 397–403. IEEE Press, Piscataway, NJ, USA 2005

[4] M. Böttcher, M. Spott and D. Nauck. A Framework for Discovering and Analyzing Changing Customer Segments. *Advances in Data Mining — Theoretical Aspects and Applications (LNCS 4597)*, 255–268. Springer, Berlin, Germany 2007

[5] C. Borgelt. Keeping Things Simple: Finding Frequent Item Sets by Recursive Elimination. *Proc. Workshop Open Software for Data Mining (OSDM'05 at KDD'05, Chicago, IL)*, 66–70. ACM Press, New York, NY, USA 2005

[6] Y. Cheng, U. Fayyad, and P.S. Bradley. Efficient Discovery of Error-Tolerant Frequent Itemsets in High Dimensions. *Proc. 7th Int. Conf. on Knowledge Discovery and Data Mining (KDD'01, San Francisco, CA)*, 194–203. ACM Press, New York, NY, USA 2001

[7] J. Han, H. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. *Proc. Conf. on the Management of Data (SIGMOD'00, Dallas, TX)*, 1–12. ACM Press, New York, NY, USA 2000

[8] R. Kohavi, C.E. Bradley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 Organizers' Report: Peeling the Onion. SIGKDD Exploration 2(2):86–93. ACM Press, New York, NY, USA 2000

[9] J. Pei, J. Han, B. Mortazavi-Asl, and H. Zhu. Mining Access Patterns Efficiently from Web Logs. *Proc. Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD'00, Kyoto, Japan)*, 396–407. 2000

[10] J. Pei, A.K.H. Tung, and J. Han. Fault-Tolerant Frequent Pattern Mining: Problems and Challenges. *Proc. ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMK'01, Santa Babara, CA)*. ACM Press, New York, NY, USA 2001

[11] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. *Proc. 3rd Int. Conf. on Knowledge Discovery and Data Mining (KDD'97, Newport Beach, CA)*, 283–296. AAAI Press, Menlo Park, CA, USA 1997

[12] H. Mannila, H. Toivonen, and A.I. Verkamo. *Discovery of Frequent Episodes in Event Sequences*. Report C-1997-15, University of Helsinki, Finland 1997

[13] C. Kuok, A. Fu, and M. Wong. Mining Fuzzy Association Rules in Databases. *SIGMOD Record* 27(1):41–46. 1998

[14] P. Moen. *Attribute, Event Sequence, and Event Type Similarity Notions for Data Mining*. Ph.D. Thesis/Report A-2000-1, Department of Computer Science, University of Helsinki, Finland 2000

[15] X. Wang, C. Borgelt, and R. Kruse. Mining Fuzzy Frequent Item Sets. *Proc. 11th Int. Fuzzy Systems Association World Congress (IFSA'05, Beijing, China)*, 528–533. Tsinghua University Press and Springer-Verlag, Beijing, China, and Heidelberg, Germany 2005

[16] G.I. Webb and S. Zhang. $k$-Optimal-Rule-Discovery. *Data Mining and Knowledge Discovery* 10(1):39–79. Springer, Amsterdam, Netherlands 2005

[17] G.I. Webb. Discovering Significant Patterns. *Machine Learning* 68(1):1–33. Springer, Amsterdam, Netherlands 2007


ISBN: 978-989-95079-6-8