

(Approximate) Frequent Item Set Mining Made Simple with a Split and Merge Algorithm

Christian Borgelt¹ and Xiaomeng Wang²

¹ European Center for Soft Computing
c/ Gonzalo Gutiérrez Quirós s/n, 33600 Mieres, Asturias, Spain
christian.borgelt@softcomputing.es

² Otto-von-Guericke-University of Magdeburg
Universitätsplatz 2, 39106 Magdeburg, Germany
xwang@iws.cs.uni-magdeburg.de

Abstract. In this paper we introduce SaM, a split and merge algorithm for frequent item set mining. Its core advantages are its extremely simple data structure and processing scheme, which not only make it very easy to implement, but also fairly easy to execute on external storage, thus rendering it a highly useful method if the data to mine cannot be loaded into main memory. Furthermore, we present extensions of this algorithm, which allow for approximate or “fuzzy” frequent item set mining in the sense that missing items can be inserted into transactions with a user-specified penalty. Finally, we present experiments comparing our new method with classical frequent item set mining algorithms (like Apriori, Eclat and FP-growth) and with the approximate frequent item set mining version of RELim (an algorithm we proposed in an earlier paper and improved in the meantime).

1 Introduction

It may not even be an exaggeration to say that the tasks of frequent item set mining and association rule induction started the popular research area of data mining. At least, however, these tasks have a strong and long-standing tradition in data mining and knowledge discovery in databases and account for a huge number of publications in data mining conferences and journals. The enormous research efforts devoted to these tasks have led to a variety of sophisticated and efficient algorithms to find frequent item sets. Among the best-known are Apriori [1, 2], Eclat [17] and FP-growth [11].

Nevertheless, there is still room for improvement: while Eclat, which is the simplest of the mentioned algorithms, can be fairly slow on some data sets (compared to other algorithms), FP-growth, which is usually the fastest algorithm, employs a sophisticated data structure and requires to load the transaction data to mine into main memory. Hence a simpler processing scheme, which still maintains efficiency, is desirable. Other lines of improvement include filtering the found frequent item sets and association rules (see, e.g., [22, 23]), identifying temporal changes in discovered patterns (see, e.g., [4, 5]), and discovering fault-tolerant or approximate frequent item sets (see, e.g., [9, 14, 21]).

In this paper we introduce SaM, a split and merge algorithm for frequent item set mining. Its core advantages are its extremely simple data structure and processing scheme, which not only make it very easy to implement, but also fairly easy to execute

on external storage, thus rendering it a highly useful method if the data to mine cannot be loaded into main memory. Furthermore, we present extensions of this algorithm, which allow for approximate or “fuzzy” frequent item set mining in the sense that missing items can be inserted into transactions with a user-specified penalty. We developed this algorithm as a simplification of the already very simple RElim algorithm [8].

The rest of this paper is structured as follows: Section 2 briefly reviews the basics of frequent item set mining, and especially the basic divide-and-conquer scheme underlying many frequent item set mining algorithms. In Section 3 we present our SaM (Split and Merge) algorithm for exact frequent item set mining and in Section 4 compare it experimentally to classic frequent item set mining algorithms like Apriori, Eclat, and FP-growth, but also our own RElim algorithm [8]. Section 5 reviews approximate or “fuzzy” frequent item set mining in the sense that missing items can be inserted into transactions with a user-specified penalty. In Sections 6 and 7 we present two extensions of our SaM algorithm that allow to perform such approximate frequent item set mining with unlimited and limited item insertions, respectively. In Section 8 we compare these extensions experimentally to the corresponding extensions of the RElim algorithm [21]. Finally, in Section 9, we draw conclusions from our discussion.

2 Frequent Item Set Mining

Frequent item set mining is a data analysis method that was originally developed for market basket analysis, which aims at finding regularities in the shopping behavior of the customers of supermarkets, mail-order companies and online shops. In particular, it tries to identify sets of products that are frequently bought together. Once identified, such sets of associated products may be used to optimize the organization of the offered products on the shelves of a supermarket or the pages of a mail-order catalog or web shop, or may give hints which products may conveniently be bundled.

Formally, the task of frequent item set mining can be described as follows: we are given a set B of *items*, called the *item base*, and a database T of *transactions*. Each item represents a product, and the item base represents the set of all products offered by a store. The term *item set* refers to any subset of the item base B . Each transaction is an item set and represents a set of products that has been bought by an actual customer. Since two or even more customers may have bought the exact same set of products, the total of all transactions must be represented as a vector, a bag or a multiset, since in a simple set each transaction could occur at most once.³ 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 identify all item sets $I \subseteq B$ that are frequent in a given transaction database T . Note that the task of frequent item set mining may also be defined with a *relative* minimum support, which is the fraction of transactions in T that must contain an item set I in order to make I frequent. However, this alternative definition is obviously equivalent.

³ Alternatively, each transaction may be enhanced by a unique *transaction identifier*, and these enhanced transactions may then be combined in a simple set.

A standard approach to find all frequent item sets w.r.t. a given database T and support threshold s_{\min} , which is adopted by basically all frequent item set mining algorithms (except those of the Apriori family), is a *depth-first search* in the subset lattice of the item base B . Viewed properly, this approach can be interpreted as a simple *divide-and-conquer* scheme. For some chosen item i , the problem to find all frequent item sets is split into two subproblems: (1) find all frequent item sets containing the item i and (2) find all frequent item sets *not* containing the item i . Each subproblem is then further divided based on another item j : find all frequent item sets containing (1.1) both items i and j , (1.2) item i , but not j , (2.1) item j , but not i , (2.2) neither item i nor j etc.

All subproblems that occur in this divide-and-conquer 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 to mine 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 the item i , but with the item i removed. This also implies that transactions that contain no other item than i are entirely removed: no empty transactions are ever kept. 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 $S_2 = (C_2, P_2)$, where $P_2 = P_0$ and the conditional database C_2 comprises all transactions in C_0 (including those that do not contain the item i), but again with the item i removed. If C_2 is not empty, process S_2 recursively.

Eclat, FP-growth, RElim and several other frequent item set mining algorithms all follow this basic recursive processing scheme. They differ mainly in how they represent the conditional transaction databases. There are basically two fundamental approaches, namely horizontal and vertical representations. In a horizontal representation, the database is stored as a list (or array) of transactions, each of which is a list (or array) of the items contained in it. In a vertical representation, a database is represented by first referring with a list (or array) to the different items. For each item a list of transaction identifiers is stored, which indicate the transactions that contain the item.

However, this distinction is not pure, since there are many algorithms that use a combination of the two forms of representing a database. For example, while Eclat uses a purely vertical representation, FP-growth combines in its FP-tree structure a vertical representation (links between branches) and a (compressed) horizontal representation (prefix tree of transactions). RElim uses basically a horizontal representation, but groups transactions w.r.t. their leading item, which is, at least partially, a vertical representation. The SaM algorithm presented in the next section is, to the best of our knowledge, the first frequent item set mining algorithm that is based on the general processing scheme outlined above and uses a purely horizontal representation.⁴

⁴ Note that Apriori, which also uses a purely horizontal representation, relies on a different processing scheme, since it traverses the subset lattice level-wise rather than depth-first.

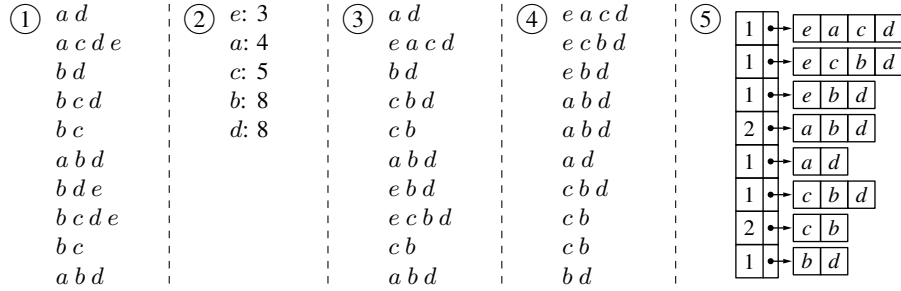


Fig. 1. The example database: original form (1), item frequencies (2), transactions with sorted items (3), lexicographically sorted transactions (4), and the used data structure (5).

The basic processing scheme can easily be improved with so-called *perfect extension pruning*, which relies on the following simple idea: given an item set I , an item $i \notin I$ is called a *perfect extension* of I , iff I and $I \cup \{i\}$ have the same support, that is, if i is contained in all transactions containing I . Perfect extensions have the following properties: (1) if the 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 I is a frequent item set and K is the set of all perfect extensions of I , then all sets $I \cup J$ with $J \in 2^K$ (where 2^K denotes the power set of K) are also frequent and 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. Once identified, perfect extension items are no longer processed in the recursion, but are only used to generate all supersets of the prefix that have the same support. Depending on the data set, this can lead to a considerable acceleration. It should be clear that this optimization can, in principle, be applied in all frequent item set mining algorithms.

3 A Simple Split and Merge Algorithm

The SaM (Split and Merge) algorithm presented in this paper can be seen as a simplification of the already fairly simple RELim (recursive elimination) algorithm, which we proposed in [8] and extended to approximate or “fuzzy” frequent item set mining in [21]. While RELim represents a (conditional) database by storing one transaction list for each item, the split and merge algorithm presented here uses only a single transaction list, stored as an array. This array is processed with a simple split and merge scheme, which computes a conditional database, processes this conditional database recursively, and eliminates the split item from the original (conditional) database.

SaM preprocesses a given transaction database in a way that is very similar to the preprocessing used by 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 frequencies of individual items are determined from this input in order to be able to discard infrequent items immediately. If we assume a minimum support of three transactions for our example, there are

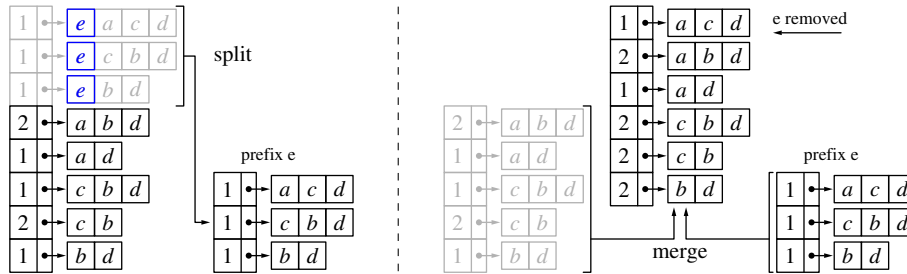


Fig. 2. The basic operations of the SaM algorithm: split (left) and merge (right).

no infrequent items, so all items are kept. In step 3 the (frequent) items in each transaction are sorted according to their frequency in the transaction database, since it is well known that 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 item comparisons again being decided by the item frequencies, although here the item with the higher frequency precedes the item with the lower frequency. In step 5 the data structure on which SaM operates 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. This data structure is then processed recursively to find the frequent item sets.

The basic operations of the recursive processing, which follows the general depth-first/divide-and-conquer scheme reviewed in Section 2, are illustrated in Figure 2. In the *split step* (see the left part of Figure 2) the given array is split w.r.t. the leading item of the first transaction (item e in our example): all array 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 items sets containing the split item (provided this item is frequent).

The conditional database for frequent item sets *not* containing this item (needed for the second subproblem, see Section 2) is obtained with a simple *merge step* (see the right part of Figure 2). The created new array and the rest of the original array (which refers to all transactions starting with a different item) are combined with a procedure that is almost identical to one phase of the well-known *mergesort* algorithm. Since both arrays are obviously 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. That is, there is always just one instance of each transaction (suffix), while its number of occurrences is kept in the occurrence 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.

Note that in both the split and the merge step only the array elements (that is, the occurrence counter and the (advanced) transaction pointer) are copied to a new array.

```

function SaM (a: array of transactions,      (* conditional database to process *)
              p: set of items,                (* prefix of the conditional database a *)
              smin: int) : int
(* minimum support of an item set *)
var i: item;                                (* buffer for the split item *)
    s: int;                                    (* support of the current split item *)
    n: int;                                    (* number of found frequent item sets *)
    b, c, d: array of transactions;        (* conditional and merged database *)
begin                                        (* — split and merge recursion — *)
    n := 0;                                    (* initialize the number of found item sets *)
    while a is not empty do                 (* while conditional database is not empty *)
        b := empty; s := 0;                (* initialize split result and item support *)
        i := a[0].items[0];                (* get leading item of the first transaction *)
        while a is not empty and a[0].items[0] = i do (* and split database w.r.t. this item *)
            s := s + a[0].wgt;            (* sum the occurrences (compute support) *)
            remove i from a[0].items;      (* remove the split item from the transaction *)
            if a[0].items is not empty      (* if the transaction has not become empty *)
                then remove a[0] from a and append it to b;
            else remove a[0] from a; end;    (* move it to the conditional database, *)
        end;                                (* otherwise simply remove it *)
        c := b; d := empty;                (* note split result, init. the output array *)
        while a and b are both not empty do (* merge split result and rest of database *)
            if a[0].items > b[0].items    (* copy lex. smaller transaction from a *)
                then remove a[0] from a and append it to d;
            else if a[0].items < b[0].items (* copy lex. smaller transaction from b *)
                then remove b[0] from b and append it to d;
            else b[0].wgt := b[0].wgt + a[0].wgt; (* sum the occurrence counters/weights *)
                remove b[0] from b and append it to d;
                remove a[0] from a;        (* move combined transaction and *)
            end;                            (* delete the other, equal transaction: *)
        end;                                (* keep only one instance per transaction *)
        while a is not empty do            (* copy the rest of the transactions in a *)
            remove a[0] from a and append it to d; end;
        while b is not empty do          (* copy the rest of the transactions in b *)
            remove b[0] from b and append it to d; end;
        a := d;                            (* second recursion is executed by the loop *)
        if s ≥ smin then                (* if the split item is frequent: *)
            p := p ∪ {i};                (* extend the prefix item set and *)
            report p with support s;      (* report the found frequent item set *)
            n := n + 1 + SaM(c, p, smin); (* process the conditional database recursively *)
            p := p − {i};                (* and sum the found frequent item sets, *)
        end;                                (* then restore the original item set prefix *)
    end;
    return n;                               (* return the number of frequent item sets *)
end; (* function SaM() *)

```

Fig. 3. Pseudo-code of the SaM algorithm. The actual C code is even shorter than this description, despite the fact that it contains additional functionality, because certain operations needed in this algorithm can be written very concisely in C (using pointer arithmetic to process arrays).

There is no need to copy the transactions themselves (that is, the item arrays), since no changes are ever made to them. (In the split step the leading item is not actually removed, but only skipped by advancing the pointer (in)to the transaction.) Hence it suffices to have one global copy of all transactions, which is merely referred to in different ways from different arrays used in the processing.

Note also that the merge result may be created in the array that represented the original (conditional) database, since its front elements have been cleared in the split step. In addition, the array for the split database can be reused after the recursion for the split w.r.t. the next item. As a consequence, each recursion step, which expands the prefix of the conditional database, only needs to allocate one new array, with a size that is limited to the size of the input array of that recursion step. This makes the algorithm not only simple in structure, but also very efficient in terms of memory consumption.

Finally, note that the fact that only a simple array is used as the underlying data structure, the algorithm can fairly easily be implemented to work on external storage or a (relational) database system. There is, in principle, no need to load the transactions into main memory and even the array may easily be stored as a simple (relational) table. The split operation can then be implemented as an SQL select statement. The merge operation is very similar to a join, even though it may require a more sophisticated comparison of transactions (depending on how the transactions are actually stored).

Pseudo-code of the recursive procedure is shown in Figure 3. As can be seen, a single page of code is sufficient to describe the whole recursion in detail. The actual C code we developed is even shorter than this pseudo-code, despite the fact that the C code contains additional functionality (like, for example, perfect extension pruning, see Section 2), because certain operations needed in this algorithm can be written very concisely in C (especially when using pointer arithmetic to process arrays).

4 Exact Frequent Item Set Mining Experiments

In order to evaluate the proposed SaM algorithm, we ran it against our own implementations of Apriori [6], Eclat [6], FP-growth [7], and RElim [8], all of which rely on the same code to read the transaction database and to report found frequent item sets. Of course, using our own implementations has the disadvantage that not all of these implementations reach the speed of the fastest known implementations.⁵ However, it has the important advantage that any differences in execution time can only be attributed to differences in the actual processing scheme, as all other parts of the programs are identical. Therefore we believe that the measured execution times are still reasonably expressive and allow us to compare the different approaches in a reliable manner.

We ran experiments on five data sets, which were also used in [6–8]. As they exhibit different characteristics, the advantages and disadvantages of the different algorithms can be observed well. These data sets are: census (a data set derived from an extract of the US census bureau data of 1994, which was preprocessed by discretizing numeric attributes), chess (a data set listing chess end game positions for king vs. king

⁵ In particular, in [15] an FP-growth implementation was presented, which is highly optimized to how modern processor access their main memory [16].

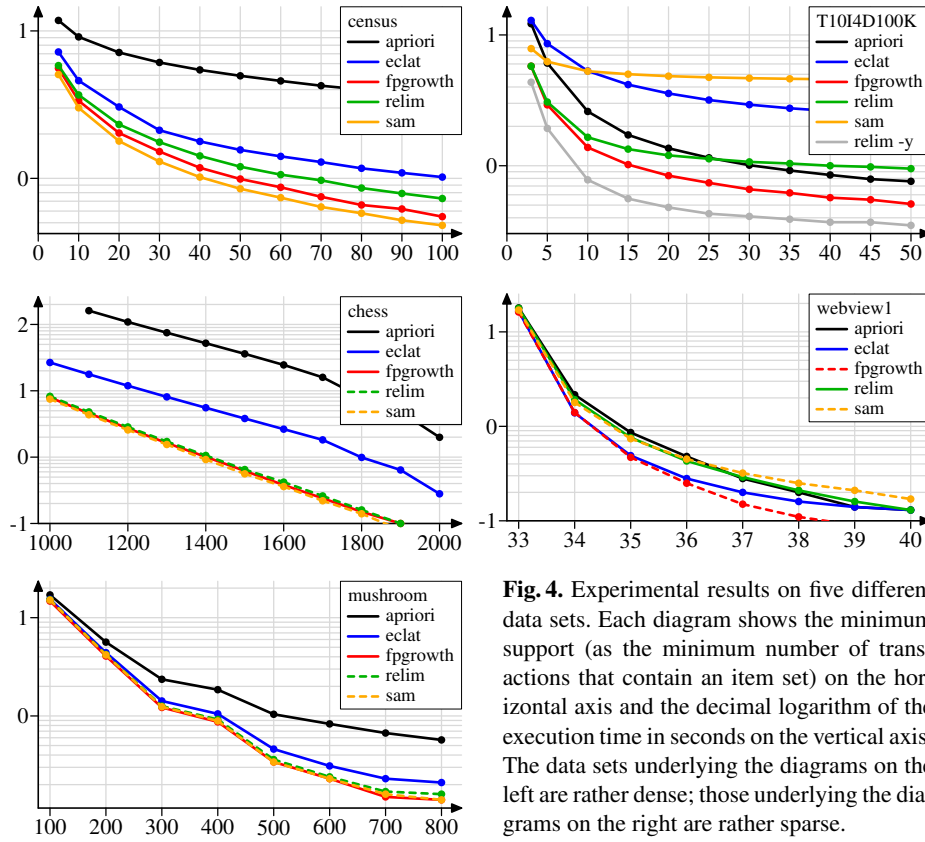


Fig. 4. Experimental results on five different data sets. Each diagram shows the minimum support (as the minimum number of transactions that contain an item set) on the horizontal axis and the decimal logarithm of the execution time in seconds on the vertical axis. The data sets underlying the diagrams on the left are rather dense; those underlying the diagrams on the right are rather sparse.

and rook), mushroom (a data set describing poisonous and edible mushrooms by different attributes), 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 [12]), and T10I4D100K (an artificial data set generated with IBM’s data generator [24]). The first three data sets are available from the UCI machine learning repository [3]. The shell script used to discretize the numeric attributes of the census data set can be found at the URL mentioned below. The first three data sets can be characterized as “dense”, meaning that on average a rather high fraction of all items is present in a transaction (the average transaction length divided by the number of different items is 0.1, 0.5, and 0.2, respectively, for these data sets), while the last two are rather “sparse” (the number of different items divided by the average transaction length is 0.01 and 0.005, respectively).

For the experiments we used 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. The results for these data sets are shown in Figure 4. Each diagram in this figure refers to one data set and shows the decimal logarithm of the execution time in seconds (excluding the time to load the transaction database) over the minimum support (stated as the number of transactions that must contain an item set in order to render it frequent).

These results show a fairly clear picture: SaM performs extremely well on dense data sets. It is the fastest algorithm for the census data set and (though only by a very small margin) on the chess data set. On the mushroom data set it performs on par with FP-growth and RElim, while it is faster than Eclat and Apriori. On “sparse” data sets, however, SaM struggles. On the artificial data set T10I4D100K it performs particularly badly and catches up with the performance of other algorithms only at the lowest support levels.⁶ On BMS-Webview-1 it performs somewhat better, but again reaches the performance of other algorithms only for fairly low support values.

Given SaM’s processing scheme, the cause of this behavior is easily found: it is clearly the merge operation. Such a merge operation is most efficient if the two lists to merge do not differ too much in length. Because of this, the recursive procedure of the *mergesort* algorithm splits its input into two lists of roughly equal length. If, to consider an extreme case, it would always merge single elements with the (recursively sorted) rest of the list, its time complexity would deteriorate from $O(n \log n)$ to $O(n^2)$. The same applies to SaM: in a dense data set it is more likely that the two transaction lists do not differ too much in length, while in a sparse data set it can rather be expected that the list containing the split item will be rather short compared to the rest. As a consequence, SaM performs well on dense data sets, but poorly on sparse ones.

The main reason for the merge operation is to keep the list sorted, so that (1) all transactions with the same leading item are grouped together and (2) equal transactions (or transaction suffixes) can be combined, thus reducing the number of objects to process. The obvious alternative to achieve (1), namely to set up a separate list for each item, is employed by the RElim algorithm, which, as these experiments show, performs considerably better on sparse data sets. On T10I4D100K it even outperforms all other algorithms by a clear margin if the list for the next item to be processed is not sorted in order to combine duplicate entries (grey curve in Figure 4). The reason is that the sorting, which in RElim only serves the purpose to eliminate possible duplicates, causes higher costs than the gains resulting from having fewer transactions to process. On all other data sets sorting the list (and thus removing duplicates) speeds up the processing, thus providing another piece of evidence why SaM performs badly on T10I4100K.

These insights lead, of course, to several ideas how SaM could be improved. However, we do not explore these possibilities in this paper, but leave them for future work.

5 Approximate Frequent Item Set Mining

In many applications of frequent item set mining the considered transactions do not contain all items that are actually present. However, all of the algorithms mentioned so far seek to discover frequent item sets based on exact matching and thus are not equipped to meet the needs arising in these applications.

An example is the analysis of alarm sequences in telecommunication networks. A core task of analyzing alarm sequences is to find collections of alarms occurring frequently together—so-called episodes. In [18] a time window was introduced that moves along the alarm sequence to build a sequence of partially overlapping windows.

⁶ It should be noted, though, that SaM’s execution times on T10I4D100K are always around 5 seconds and thus not unbearable.

Each window captures a specific slice of the alarm sequence. In this way the problem of finding frequent episodes is transformed into the problem of finding frequent item sets in a database of transactions, where each alarm can be treated as an item, the alarms in a time window as a transaction, and the support of an episode is the number of windows in which the episode occurred. 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 can be 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 could be frequent if the items did not get lost, may be smaller than the user-specified minimum. This leads to a possible loss of potentially interesting frequent item sets and to possibly distorted support values.

To cope with such missing information, we introduce the notion of an approximate or “fuzzy” frequent item set. In contrast to research on fuzzy association rules (see, for example, [19]), 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 found exactly in all supporting transactions, but only approximately. Related work in this direction includes [9, 14], where Apriori-like algorithms were introduced and mining with approximate matching was performed by counting the number of different items in the two item sets to be compared. In this paper, however, we adopt a more general scheme, based on an approximate matching approach, which exhibits a much higher flexibility. Our approach employs two core ingredients: *edit costs* and *transaction weights* [21].

Edit costs: The distance between two item sets can conveniently be defined as the costs of the cheapest sequence of edit operations needed to transform one item set into the other [20]. Here we consider only insertions, since they are very easy to implement with our algorithm⁷. With the help of an “insertion cost” or “insertion penalty” a flexible and general framework for modeling approximate matching between two item sets can be established. The interpretation of such costs or penalties depends, of course, on the application. In addition, different items can be associated with different insertion costs. For example, in telecommunication networks different alarms can have a different probability of getting lost: usually alarms originating in lower levels of the module hierarchy get lost more easily than alarms originating in higher levels. Therefore the former can be associated with lower insertion costs than the latter. The insertion of a certain item may also be completely inhibited by assigning a very high insertion cost.

Transaction weights: Each transaction t in the original database T is associated with a weight $w(t)$. The initial weight of each transaction is 1. When inserting an item i into a transaction t , its weight is “penalized” with a cost $c(i)$ associated with the item. Formally, this can be described by 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 $c(i)$. There is, of course, a wide variety of possible combination functions. For example, any t -norm may be used. For simplicity, we use multiplication here, that is, $w_{\{i\}}(t) = w(t) \cdot c(i)$, but this is a more or less arbitrary choice. Note, however, that with this choice lower values of $c(i)$ mean higher costs as they penalize the weight more, but it has the advantage that it

⁷ Note that deletions are implicit in the mining process anyway (as we search for *subsets* of the transactions). Only replacements are an additional case we do not consider here.

is easily extended to an insertion of multiple items: $w_{\{i_1, \dots, i_m\}}(t) = w(t) \cdot \prod_{k=1}^m c(i_k)$. 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 and thus no further items may be inserted into it. 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 ϕ is true and 0 otherwise.

Note that SaM is particularly well suited to handle this scheme of item insertions, because it relies on a horizontal transaction representation, which makes it very simple to incorporate transaction weights into the mining process. With other algorithms (with the exception of RELim, which also uses a basically horizontal representation), more effort is usually needed in order to extend them to approximate frequent item set mining.

For the implementation of the approximate frequent item set mining scheme outlined above, it is important to distinguish between unlimited item insertions (that is, $w_{\min} = 0$) and limited item insertions (that is, $w_{\min} > 0$). The reason is that with $w_{\min} = 0$ a transaction always contributes to the support of any item set (because, in principle, all items of the item set could be inserted), while with $w_{\min} > 0$ a transaction only contributes to those item sets which it can be made to contain by inserting items without reducing the transaction weight below the threshold w_{\min} .

As a consequence it is possible to combine equal transactions (or transaction suffixes) without restriction if $w_{\min} = 0$: if we have two equal transactions (or transactions suffixes) t_1 and t_2 with weights w_1 and w_2 , respectively, we can combine t_1 and t_2 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, the fact that $w_1 \cdot c(i) + w_2 \cdot c(i) = (w_1 + w_2) \cdot c(i)$) ensures that we still compute the correct support for the item set I in this case.

If, however, we have $w_{\min} > 0$ and, say, $w_1 > w_2$, then using $(w_1 + w_2) \cdot c(i)$ as the support contributed by the combined transaction t to the support of the item set I may be wrong, since it may be that $w_1 \cdot c(i) \geq w_{\min}$, but $w_2 \cdot c(i) < w_{\min}$. In this case 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 would fall below the minimum transaction weight threshold by inserting the item i . Hence, under these circumstances, we can combine equal transactions (or transaction suffixes) only if they have the same weight (that is, only if $w_1 = w_2$).

6 Unlimited Item Insertions

If unlimited item insertions are possible ($w_{\min} = 0$), only a minor change has to be made to the data structure: instead of an integer occurrence counter for the transactions (or transaction suffixes), we need a real-valued transaction weight. In the processing, the split step stays the same (see Figure 5 on the left), but now it only yields an intermediate database, into which all transactions (or transaction suffixes) have been transferred that actually contain the split item under consideration (item e in the example).

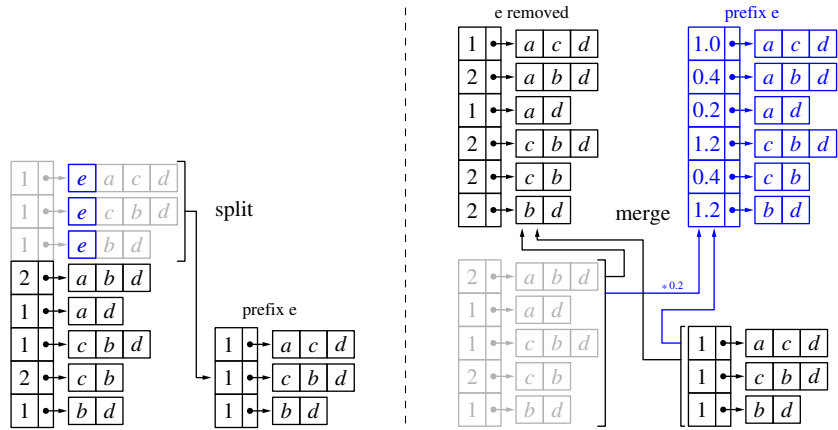


Fig. 5. The extended operations: unlimited item insertions, first recursion level.

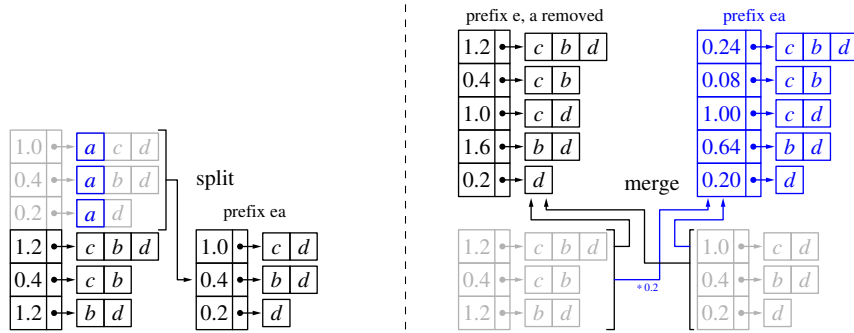


Fig. 6. The extended operations: unlimited item insertions, second recursion level.

In order to build 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 5 on the right). The first part (shown in black) is the merge that yields (as in the basic algorithm) the conditional database for frequent item sets *not* containing the split item. The second part (shown in blue) 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 parts of the merge operation equal transactions (or transaction suffixes) can be combined regardless of their weight. As a consequence we have in Figure 5 entries like for the transaction (suffix) cbd , with a weight of 1.2, which stands

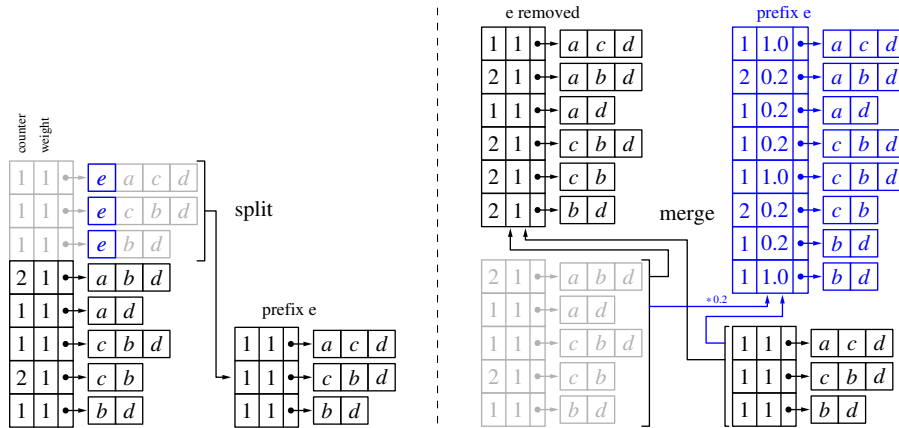


Fig. 7. The extended operations: limited item insertions, first recursion level.

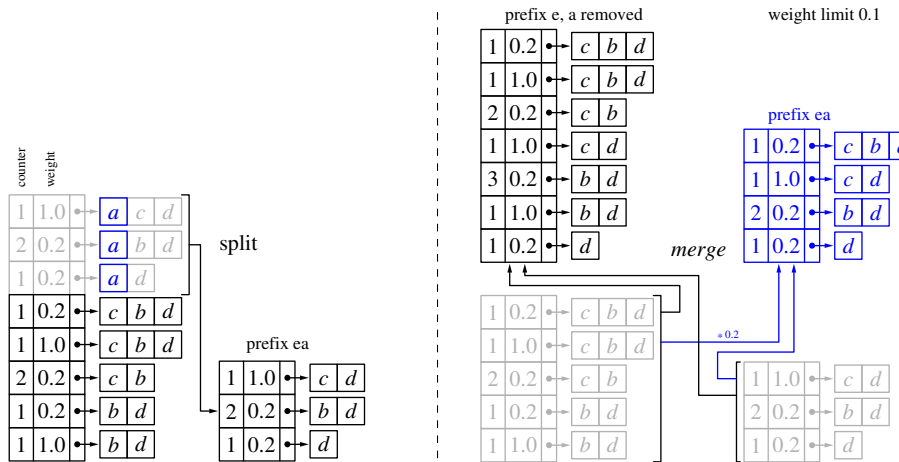


Fig. 8. The extended operations: limited item insertions, second recursion level.

for one occurrence with weight 1 and one occurrence with weight 0.2 (due to the penalty factor 0.2, needed to account for the insertion of item e). As an additional illustration, Figure 6 shows the split and merge operations for the second recursion level (which work on the conditional database for the prefix e constructed on the first level).

7 Limited Item Insertions

If item insertions are limited by a threshold for the transaction weight ($w_{\min} > 0$), we have to represent the transaction weight explicitly and keep it separate from the number of occurrences of the transaction. Therefore the data structure must be extended to

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} and no transactions with this field lower than w_{\min} will ever be kept. In addition, there may now be array elements that refer to the same transaction (suffix)—that is, the same list of items—and which differ only in the transaction weight (and maybe, of course, at the same time in the occurrence counter).

The processing scheme is illustrated in Figure 7 with the same example as before. The split step is still essentially the same and only the merge step is modified. The difference consists, as already pointed out, in the fact that equal transactions (or transaction suffixes) can no longer be combined if they differ in weight. As a consequence, there are now, in the result of the second part of the merge operation (shown in blue) two array elements for *cbd* and two for *bd*, which carry a different weight (one has a weight of 1, the other a weight of 0.2). As already explained in Section 5, this is necessary, because two transactions with different weight may reach, due to item insertions, the transaction weight threshold at different times and thus cannot be combined.

Of course, it rarely happens on the first level of the recursion that transactions are discarded due to the weight threshold. This can only occur on the first level, if the insertion penalty factor of the split item is already smaller than the transaction weight threshold, which is equivalent to inhibit insertions of the split item altogether. Therefore, in order to illustrate this aspect of the processing scheme, Figure 8 shows the operations on the second recursion level, where the conditional database with prefix *e* (that is, for frequent item sets containing item *e*) is processed. Here the second part of the merge process 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.

8 Approximate Frequent Item Set Mining Experiments

Since we want to present several diagrams per data set in order to illustrate the influence of the different parameters (insertion penalty factor, number of items with a non-vanishing penalty factor, threshold for the transaction weight), we limit our report to the results on two of the five data sets used in Section 4. We chose census and BMS-Webview-1, one dense and one sparse data set, since SaM and RElim (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 9 for the census data set and in Figure 10 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. The different parameters we tested in our experiments are: 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 item insertions.⁸

⁸ Since we used the same insertion penalty factor $c(i)$ for all items having $c(i) > 0$, the transaction weight threshold effectively limits the number of insertions regardless of which items are inserted. Hence this description is more expressive than stating the actual values w_{\min} used.

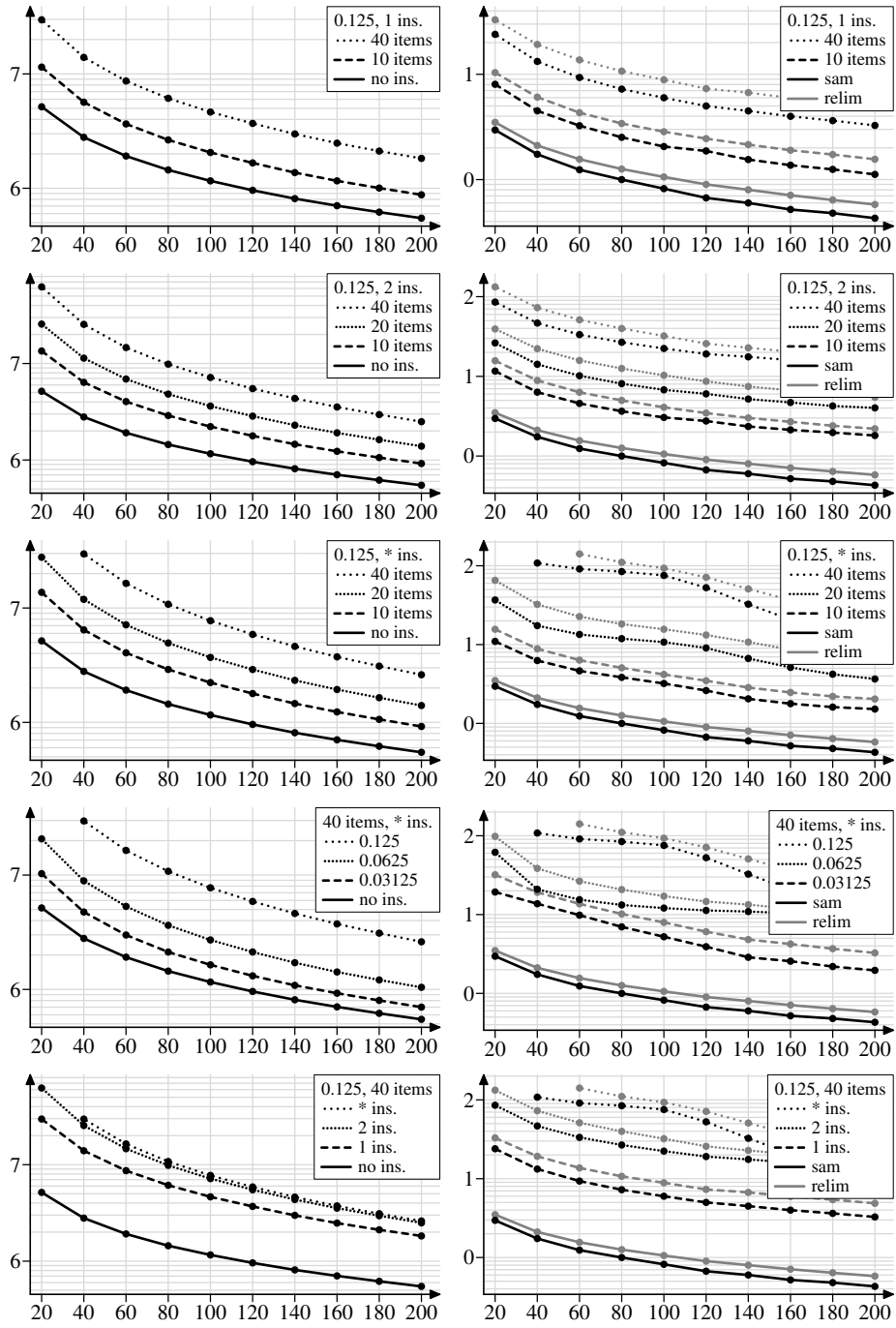


Fig. 9. Experimental results on census data; left: frequent item sets, right: execution times.

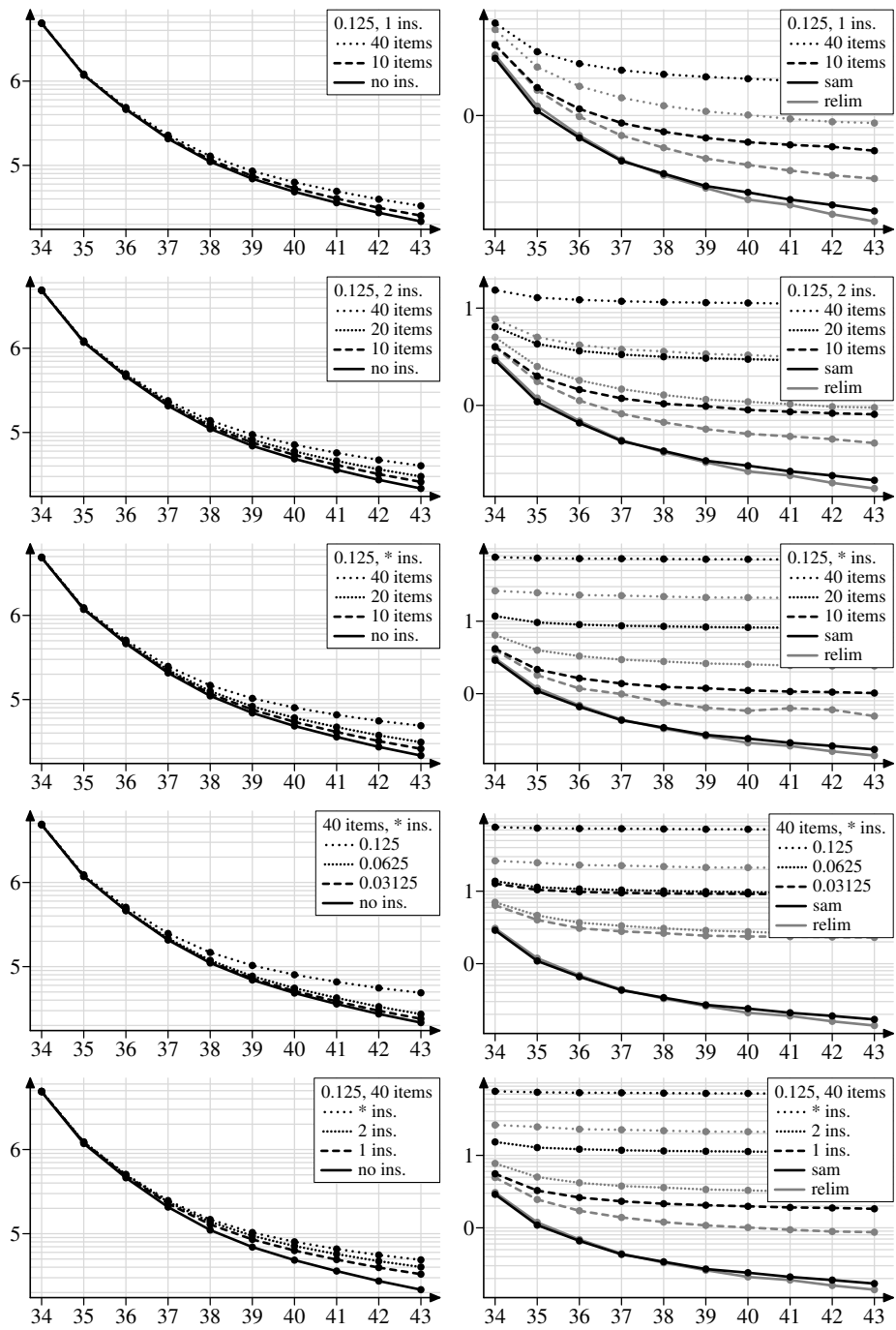


Fig. 10. Experimental results on webview1 data; left: frequent item sets, right: execution times.

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 the census data set, it rises only very moderately for the BMS-Webview-1 data set, 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 the BMS-Webview-1 data set (this needs closer examination, though).

As could be expected from the results of the basic algorithms on the five data sets used in Section 4, SaM fares 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 were on par without item insertion. On both data sets, the number of insertions that are allowed has, not surprisingly, the strongest influence: with two insertions about an order of magnitude larger times result 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 penalty factor. This is plausible, since there should not be much difference in having the possibility to insert twice the number items or preserving twice the transaction weight per item insertion. Note, however, that doubling the penalty factor from $\frac{1}{32}$ to $\frac{1}{16}$ has only a comparatively small effect on the BMS-Webview-1 data set 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, though 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 ($\log(180) \approx 2.25$). In addition, we can observe the interesting effect on the BMS-Webview-1 data set that at the highest parameter settings the execution times become almost independent of the minimum support threshold.

9 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 approximate or “fuzzy” 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, our SaM algorithm performs excellently on dense data sets, but shows certain weaknesses on sparse data sets. This applies not only for exact mining, but also for approximate frequent item set mining. However, our experiments provide some evidence (to be substantiated on other data sets) that approximate frequent item set mining is much more useful for dense data sets as more additional frequent item sets can be found on these. 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

An implementation of our SaM algorithm in C can be found at:

<http://www.borgelt.net/sam.html>

while an implementation of our RELim algorithm in C is available at:

<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. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo. Fast Discovery of Association Rules. In [10], 307–328.
3. 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
<http://www.ics.uci.edu/~mlearn/MLRepository.html>
4. 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
5. 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
6. C. Borgelt. Efficient Implementations of Apriori and Eclat. *Proc. Workshop Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL, USA)*. CEUR Workshop Proceedings 90, Aachen, Germany 2003
7. C. Borgelt. An Implementation of the FP-growth Algorithm. *Proc. Workshop Open Software for Data Mining (OSDM'05 at KDD'05, Chicago, IL)*, 1–5. ACM Press, New York, NY, USA 2005
8. 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
9. 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
10. U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, eds. *Advances in Knowledge Discovery and Data Mining*. AAAI Press / MIT Press, Cambridge, CA, USA 1996
11. 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
12. 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
13. 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. Springer, New York, NY, USA 2000

14. 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 Barbara, CA)*, 7–12. ACM Press, New York, NY, USA 2001
15. B. Rász. nonordfp: An FP-growth Variation without Rebuilding the FP-Tree. *Proc. Workshop Frequent Item Set Mining Implementations (FIMI 2004, Brighton, UK)*. CEUR Workshop Proceedings 126, Aachen, Germany 2004
16. B. Rász, F. Bodon, and L. Schmidt-Thieme. On Benchmarking Frequent Itemset Mining Algorithms. *Proc. Workshop Open Software for Data Mining (OSDM'05 at KDD'05, Chicago, IL)*, 36–45. ACM Press, New York, NY, USA 2005
17. 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
18. H. Mannila, H. Toivonen, and A.I. Verkamo. *Discovery of Frequent Episodes in Event Sequences*. Report C-1997-15, University of Helsinki, Finland 1997
19. C. Kuok, A. Fu, and M. Wong. Mining Fuzzy Association Rules in Databases. *SIGMOD Record* 27(1):41–46. ACM Press, New York, NY, USA 1998
20. 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
21. 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, Beijing, China and Springer, Heidelberg, Germany 2005
22. G.I. Webb and S. Zhang. *k*-Optimal-Rule-Discovery. *Data Mining and Knowledge Discovery* 10(1):39–79. Springer, New York, NY, USA 2005
23. G.I. Webb. Discovering Significant Patterns. *Machine Learning* 68(1):1–33. Springer, New York, NY, USA 2007
24. Synthetic Data Generation Code for Associations and Sequential Patterns. Intelligent Information Systems, IBM Almaden Research Center
<http://www.almaden.ibm.com/software/quest/Resources/index.shtml>