Online Association Rule Mining

Christian Hidber
International Computer Science Institute, Berkeley
hidber@icsi.berkeley.edu

Abstract

We present a novel algorithm to compute large itemsets online. The user is free to change the support threshold any time during the first scan of the transaction sequence. The algorithm maintains a superset of all large itemsets and for each itemset a shrinking, deterministic interval on its support. After at most 2 scans the algorithm terminates with the precise support for each large itemset. Typically our algorithm is by an order of magnitude more memory efficient than Apriori or DIC.

1 Introduction

Mining for association rules is a form of data mining introduced in [AIS93]. The prototypical example is based on a list of purchases in a store. An association rule for this list is a rule such as "85% of all customers who buy product A and B also buy product C and D". Discovering such customer buying patterns is useful for customer segmentation, cross-marketing, catalog design and product placement.

We give a problem description which follows [BMUT97]. The *support* of an itemset (set of items) in a transaction sequence is the fraction of all transactions containing the itemset. An itemset is called *large* if its support is greater or equal to a user-specified *support threshold*, otherwise it is called *small*. An *association rule* is an expression $X \Rightarrow Y$ where X and Y are disjoint itemsets. The *support* of this rule is the support of $X \cup Y$. The *confidence* of this rule is the fraction of all transactions containing X that also contain Y, i.e. the support of $X \cup Y$ divided by the support of X. In the example above, the "85%" is the confidence of the rule $\{A, B\} \Rightarrow \{C, D\}$. For an association rule to hold, it must have a support $\geq$ a user-specified support threshold and a confidence $\geq$ a user-specified confidence threshold. Existing algorithms proceed in 2 steps to compute association rules:

1. Find all large itemsets.
2. For each large itemset Z, find all subsets X, such that the confidence of $X \Rightarrow Z \backslash X$ is greater or equal to the confidence threshold.

We address the first step, since the second step can already be computed online, c.f. [AY97]. Existing large itemset computation algorithms have an offline or batch behaviour: given the user-specified support threshold, the transaction sequence is scanned and rescanned, often several times, and eventually all large itemsets are produced. However, the user does not know, in general, an appropriate support threshold in advance. An inappropriate choice yields, after a long wait, either too many or too few large itemsets, which often results in useless or misleading association rules.

Inspired by online aggregation, c.f. [Hel96, HHW97], our goal is to overcome these difficulties by bringing large itemset computation online. We consider an algorithm to be online if: *1)* it gives continuous feedback, *2)* it is user controllable during processing and *3)* it yields a deterministic and accurate result. Random sampling algorithms produce results which hold with some probability < 1. Thus we do not view them as being online.

In order to bring large itemset computation online, we introduce a novel algorithm called Carma (Continuous Association Rule Mining Algorithm). The algorithm needs, at most, two scans of the transaction sequence to produce all large itemsets.

During the first scan, the algorithm continuously constructs a lattice of all potentially large itemsets (large with respect to the scanned part of the transaction sequence). For each set in the lattice, Carma provides a deterministic lower and upper bound for its support. We continuously display, e.g. after each transaction processed, the resulting association rules to the user along with bounds on each rule's support and confidence. The user is free to adjust the support and con-

fidence thresholds at any time. Adjusting the support threshold may result in an increased threshold for which the algorithm guarantees to include all large itemsets in the lattice. If satisfied with the rules and bounds produced so far, the user can stop the rule mining early.

During the second scan, the algorithm determines the precise support of each set in the lattice and continuously removes all small itemsets.

Existing algorithms need to rescan the transaction sequence before any output is produced. Thus, they can not be used on a stream of transactions read from a network for example. In contrast, using Carma's first-scan algorithm, we can continuously process a stream of transactions and generate the resulting association rules online, not requiring a rescan.

While not being faster in general, Carma outperforms Apriori and DIC on low support thresholds and is up to 60 times more memory efficient.

2 Overview

The paper is structured as follows: In Section 3, we put our algorithm in the context of related work. In Section 4, we give a sketch of Carma. It uses two distinct algorithms *PhaseI* and *PhaseII* for the first and second scan respectively. In Section 5 we describe PhaseI in detail. In Subsection 5.1 we introduce *support lattices* and *support sequences*, the building blocks for the PhaseI algorithm presented in Subsection 5.2. We illustrate PhaseI on an example in Subsection 5.3. We discuss changing support thresholds in Subsection 5.4. After a short description of PhaseII in Subsection 6.1, we combine in Subsection 6.2 PhaseI with PhaseII, yielding Carma. In Section 7 we discuss our implementation. After a brief discussion of implementational details in Subsection 7.1, we compare in Subsection 7.2 the performance of Carma with Apriori and DIC. In Subsection 7.3 we analyze how the support intervals evolve during the first scan. We end with our conclusion in Section 8. In Appendix A we summarize performance results of Apriori, Carma and DIC on further datasets.

3 Related Work

Most large itemset computation algorithms are related to the *Apriori* algorithm due to Agrawal & Srikant, c.f. [AS94]. See [AY98] for a survey of large itemset computation algorithms. Apriori exploits the observation that all subsets of a large itemset are large themselves. It is a multi-pass algorithm, where in the k-th pass all large itemsets of cardinality k are computed. Hence Apriori needs up to $c + 1$ scans of the database where c is the maximal cardinality of a large itemset.

In [SON95] a 2-pass algorithm called *Partition* is introduced. The general idea is to partition the database into blocks such that each block fits into main-memory. In the first pass, each block is loaded into memory and all large itemsets, with respect to that block, are computed using Apriori. Merging all resulting sets of large itemsets then yields a superset of all large itemsets. In the second pass, the actual support of each set in the superset is computed. After removing all small itemsets, Partition produces the set of all large itemsets.

In contrast to Apriori, the DIC (Dynamic Itemset Counting) algorithm counts itemsets of different cardinality simultaneously, c.f. [BMUT97]. The transaction sequence is partioned into blocks. The itemsets are stored in a lattice which is initialized by all singleton sets. While a block is scanned, the count (number of occurences) of each itemset in the lattice is adjusted. After a block is processed, an itemset is added to the lattice if and only if all its subsets are potentially large, i.e. large with respect to the part of the transaction sequence for which its count was maintained. At the end of the sequence, the algorithm rewinds to the beginning. It terminates when the count of each itemset in the lattice is determined. Thus after a finite number of scans, the lattice contains a superset of all large itemsets and their counts. For suitable block sizes, DIC requires fewer scans than Apriori.

We note that all of the above algorithms: *1)* require that the user specifies a fixed support threshold in advance, *2)* do not give any feedback to the user while they are running and *3)* may need more than two scans (except Partition). Carma, in contrast: *1)* allows the user to change the support threshold at any time, *2)* gives continuous feedback and *3)* requires at most two scans of the transaction sequence.

Random sampling algorithms have been suggested as well, c.f. [Toi96, ZPLO96]. The general idea is to take a random sample of suitable size from the transaction sequence and compute the large itemsets using Apriori or Partition with respect to that sample. For each itemset, an interval is computed such that the support lies within the interval with probability $\geq$ some threshold. Carma, in contrast, deterministically computes all large itemsets along with the precise support for each itemset.

Several algorithms based on Apriori were proposed to update a previously computed set of large itemsets due to insertion or deletion of transactions, c.f. [CHNW96, CLK97, TBAR97]. These algorithms require a rescan of the full transaction sequence whenever an itemset becomes large due to an insertion. Carma, in contrast, requires a rescan only if the user needs the precise support of the additional large itemsets, instead of the continuously shrinking support intervals provided by PhaseI.

In [AY97] an Online Analytical Processing (OLAP)-

style algorithm is proposed to compute association rules. The general idea is to precompute all large itemsets relative to some support threshold s using a traditional algorithm. The association rules are then generated online relative to an interactively specified confidence threshold and support threshold $\geq s$. We note that: *1)* the support threshold s must be specified before the precomputation of the large itemsets, *2)* the large itemset computation remains offline and *3)* only rules with support $\geq s$ can be generated. Carma overcomes these difficulties by bringing the large itemset computation itself online. Thus, combining Carma's large itemset computation with the online rule generation suggested in [AY97] brings both steps online, not requiring any precomputation.

4 Sketch of the Algorithm

Carma uses distinct algorithms, called PhaseI and PhaseII, for the first and second scan of the transaction sequence. In this section, we give a sketch of both algorithms. For a detailed description and formal definition see Section 5 and Section 6.

During the first scan PhaseI continuously constructs a lattice of all potentially large itemsets. After each transaction, it inserts and/or removes some itemsets from the lattice. For each itemset v, PhaseI stores the following three integers (see Figure 1 below, the itemset $\{a, b\}$ was inserted in the lattice while reading the j-th transaction, the current transaction index is i):

count(v)	the number of occurences of v *since* v was inserted in the lattice.
firstTrans(v)	the index of the transaction *at* which v was inserted in the lattice.
maxMissed(v)	upper bound on the occurences of v *before* v was inserted in the lattice.

Suppose we are reading transaction i and we have a lattice of the above form. For any itemset v in the lattice, we then have a deterministic lower bound $count(v)/i$ and upper bound $(maxMissed(v) + count(v))/i$ on the support of v in the first i transactions. We denote these bounds by $minSupport(v)$ and $maxSupport(v)$ respectively. The computation of $maxMissed(v)$ during the insertion of v in the lattice is a central part of the algorithm. It not only depends on v and i, the current transaction index, but also on the current and previous support thresholds, since the user may change the threshold at any time.

After PhaseI has read a transaction, it increments $count(v)$ for all itemsets v contained in the transaction. Next, it inserts some itemsets in the lattice, computing $maxMissed$ and setting $firstTrans$ to the current transaction index. Clearly, $maxMissed$ is always less than the current transaction index. Eventually, PhaseI may remove some itemsets from the lattice if their $maxSupport$ is below the current support threshold. At the end of the transaction sequence, PhaseI guarantees that the lattice contains a superset of all large itemsets relative to some threshold. The threshold depends on how the user changed the support during the scan, c.f. Subsection 5.4. We then rewind to the beginning and start PhaseII.

PhaseII initially removes all itemsets which are trivially small, i.e. itemsets with $maxSupport$ below the last user specified threshold. By rescanning the transaction sequence, PhaseII determines the precise number of occurences of each remaining itemset and continuously removes all itemsets, which turn out to be small. Eventually, we end up with the set of all large itemsets along with their supports.

5 PhaseI Algorithm

In this section, we fully describe the PhaseI algorithm, which constructs a superset of all large itemsets while scanning the transaction sequence once. In Subsection 5.1 we introduce *support lattices* and *support sequences*, the building blocks for PhaseI. We present the PhaseI algorithm itself in Subsection 5.2. We illustrate the algorithm on an example in Subsection 5.3 and conclude this section with a discussion of changing support thresholds in Subsection 5.4.

5.1 Support Lattice & Support Sequence

For a given transaction sequence and an itemset v, we denote by $support_i(v)$ the support of v in the first i transactions. Let V be a lattice of itemsets such that for each itemset $v \in V$ we have the three associated integers $count(v)$, $firstTrans(v)$ and $maxMissed(v)$ as defined in Section 4. We call V a *support lattice* (up to i and relative to the support threshold s) if and only if V contains all itemsets v with $support_i(v) \geq s$. Hence, a support lattice is a superset of all large itemsets. For each transaction processed, the user is free to specify an arbitrary support threshold. Thus we get a sequence of support thresholds $\sigma = (\sigma_1, \sigma_2, \ldots)$, where σ_i denotes the support threshold for the i-th transaction. We call σ a *support sequence*. By $\lceil\sigma\rceil_i$ we denote the least monotone decreasing sequence which is up to i pointwise greater or equal to σ and 0 otherwise (see Figure 2 below). We call $\lceil\sigma\rceil_i$ the *ceiling of σ up to i*. By $avg_i(\sigma)$ we denote the running average of σ up to i, i.e. $avg_i(\sigma) = \frac{1}{i}\sum_{j=1}^{i}\sigma_j$. We note that $\lceil\sigma\rceil_{i+1}$ can readily be computed from $\lceil\sigma\rceil_i$ and σ_{i+1}, c.f. [Hid98, Lemma 2, Appendix C].

5.2 PhaseI Algorithm

In this subsection, we give a full description and formal definition of the PhaseI algorithm. PhaseI computes a

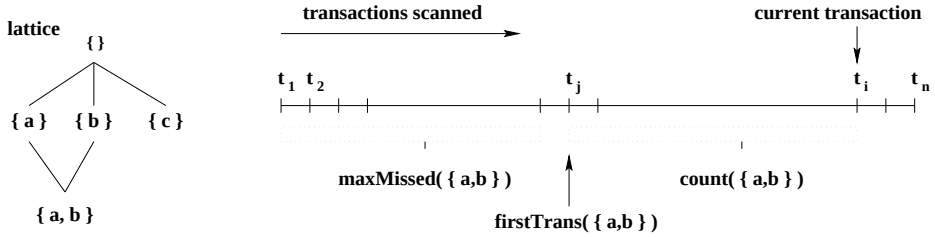


Figure 1

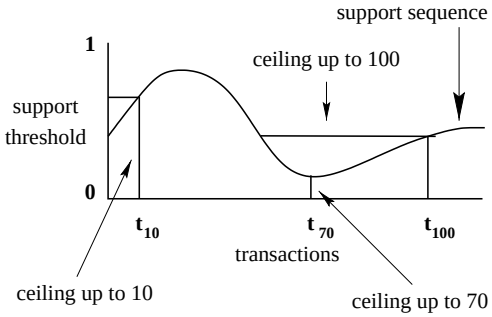


Figure 2

support lattice V as it scans the transaction sequence. We define V recursively:

Initially PhaseI sets V to $\{\emptyset\}$, setting $count$, $firstTrans$ and $maxMissed$ of $\emptyset$ to 0. Thus V is a support lattice for the empty transaction sequence.

Let V be a support lattice up to transaction $i-1$. We read the i-th transaction t_i and want to transform V into a support lattice up to i. Let σ_i be the current user-specified support threshold. To maintain the lattice we proceed in three steps: *1) increment* the count of all itemsets occuring in the current transaction, *2) insert* some itemsets in the lattice and *3) prune* some itemsets from the lattice.

1) Increment: We increment $count(v)$ for all itemsets $v \in V$ that are contained in t_i, maintaining the correctness of all integers stored in V.

2) Insert: We insert a subset v of t_i in V if and only if all subsets w of v are already contained in V and are potentially large, i.e. $maxSupport(w) \geq \sigma_i$. This corresponds to the observation that the set of all large itemsets is closed under subsets. Inserting v in V, we set $firstTrans(v) = i$ and $count(v) = 1$, since v is contained in the current transaction t_i. Since $support_i(w) \geq support_i(v)$ for all subsets w of v and $w \subset t_i$ we get

$$maxMissed(v) \leq maxMissed(w) + count(w) - 1.$$

By the following Theorem 1 we have

$$support_{i-1}(v) > avg_{i-1}(\lceil\sigma\rceil_{i-1}) + \tfrac{|v|-1}{i-1} \quad \text{implies} \quad v \in V.$$

Since v is not contained in V yet, we get thereby

$$support_{i-1}(v) \leq avg_{i-1}(\lceil\sigma\rceil_{i-1}) + \frac{|v|-1}{i-1}. \qquad (1)$$

Since $maxMissed(v)$ is an integer[1] we get by inequality (1)

$$maxMissed(v) \leq \lfloor (i-1)avg_{i-1}(\lceil\sigma\rceil_{i-1})\rfloor + |v| - 1.$$

Thus we define $maxMissed(v)$ as

$$\min \quad \{ \quad \lfloor (i-1)avg_{i-1}(\lceil\sigma\rceil_{i-1})\rfloor + |v| - 1, \quad maxMissed(w) + count(w) - 1 \mid w \subset v \quad \} \qquad (2)$$

In particular we get $maxMissed(v) \leq i-1$, since the emptyset is a subset of v, $\emptyset$ is an element of V and the count of $\emptyset$ equals i, the current transaction index.

3) Prune: We prune the lattice by removing all itemsets of cardinality ≥ 2 with a $maxSupport$ below the current support threshold σ_i, i.e. all small itemsets containing at least 2 items. Since pruning incurs a considerable overhead we only prune every $\lceil 1/\sigma_i \rceil$ or every 500 transactions[2], whichever is larger. We note that any heuristic pruning strategy is admissible as long as only small itemsets are removed and whenever an itemset is removed all its supersets are removed as well. We chose the above pruning strategy for its memory efficiency. Note that in this strategy 1-itemsets are never pruned. Thus an item, which is not contained in the lattice, did not appear in the transaction sequence so far. Hence the strategy allows us to set $maxMissed$ to 0 whenever a 1-itemset is inserted in the lattice.

The resulting PhaseI algorithm is depicted in figure 3.

The correctness of the algorithm is given by the following theorem:

Theorem 1 *Let V be the lattice returned by $PhaseI(T, \sigma)$ for a transaction sequence T of length n and support sequence σ.*

[1] For a real number x we denote by $\lfloor x \rfloor$ the largest integer less or equal to x, i.e. $\lfloor x \rfloor = \max\{i \in \mathbb{Z} \mid x \geq i\}$.

[2] For a real number x we denote by $\lceil x \rceil$ the least integer greater or equal to x, i.e. $\lceil x \rceil = \min\{i \in \mathbb{Z} \mid x \leq i\}$.

```
Function PhaseI(   transaction sequence (t_1, ..., t_n),
                   support sequence σ = (σ_1, ..., σ_n)
                   ) : support lattice;
support lattice V;
begin
 V := {∅};
 maxMissed(v) := 0, firstTrans(v) := 0
count(v) := 0;
 for i from 1 to n do
  // 1) Increment
  for all v ∈ V with v ⊆ t_i do count(v) + +; od;
  // 2) Insert
  for all v ⊆ t_i with v ∉ V do
   if ∀w ⊂ v : w ∈ V and maxSupport(w) ≥ σ_i then
    V := V ∪ {v};
    firstTrans(v) := i;
    count(v) := 1;
    maxMissed(v) :=
       min{  ⌊(i − 1)avg_{i−1}(⌈σ⌉_{i−1})⌋ + |v| − 1,
             maxMissed(w) + count(w) − 1 | w ⊂ v };
    if |v| == 1 then maxMissed(v) := 0; fi;
   fi;
  od;
  // 3) Prune
  if ( i % max{⌈1/σ_i⌉, 500} ) == 0 then
   V := {v ∈ V | maxSupport(v) ≥ σ_i or |v| == 1 };
  fi;
 od;
 return V;
end;
```

Figure 3

Then V is a support lattice relative to the support threshold

$$avg_n(\lceil\sigma\rceil_n) + \frac{c+1}{n} \tag{3}$$

with c the maximal cardinality of a large itemset in T. For any itemset v

$$support_n(v) > avg_n(\lceil\sigma\rceil_n) + \frac{|v|-1}{n} \quad \textit{implies} \quad v \in V.$$

Proof: By double induction on c and n. For a detailed proof see [Hid98, Theorem 2, Appendix C].

We illustrate Theorem 1 and in particular the support threshold given by (3) in Subsection 5.4. We omitted any optimization in the definition of PhaseI. For example, the incrementation and insertion step can be accomplished by traversing the support lattice once. We illustrate the algorithm itself on a simple example in the following Subsection 5.3.

5.3 Example

We illustrate in this subsection the PhaseI algorithm on a simple example, namely on the transaction sequence $T = (\{a,b\},\{a,b,c\},\{b,c\})$ and the support sequence $\sigma = (0.3,\ 0.9,\ 0.7)$, see Figure 4 below. As indicated we denote by the triple the three associated integers for each set in the support lattice V and by the interval the bounds on its support.

We initialize V to $\{\emptyset\}$ and the associated integers of $\emptyset$ to $(0,0,0)$. Reading $t_1 = \{a,b\}$ we first increment the *count* of $\emptyset$, since $\emptyset \subseteq t_1$. Because the empty set is the only strict subset of a singleton set and $1 = maxSupport(\emptyset) \geq \sigma_1$, we add the singletons $\{a\}$ and $\{b\}$ to V. By $maxMissed = 0$ for all singleton sets, we set their associated integers to $(0,1,1)$. Since there is no set in V with $maxSupport < 0.3$, we can not prune an itemset from V and the first transaction is processed.

Reading $t_2 = \{a,b,c\}$ we first increment *count* for $\emptyset$, $\{a\}$ and $\{b\}$. As above we insert the singleton set $\{c\}$, setting $maxMisssed$ to 0. Since $\{a,b\} \subseteq t_2$ and $\{a\}$, $\{b\}$ are elements of V with a $maxSupport \geq \sigma_2 = 0.9$, we insert $\{a,b\}$ in V. Since $\lceil\sigma\rceil_1 = (0.3,0,0,\ldots)$ we get $avg_1(\lceil\sigma\rceil_1) = 0.3$ and

$$\lfloor(2-1)avg_1(\lceil\sigma\rceil_1)\rfloor + 2 - 1 = 1.$$

Hence $maxMissed(\{a,b\}) = 1$ by equality (2) of Subsection 5.2, since $maxMissed(w)+count(w) = 2$ for $w = \{a\}$ and $w = \{b\}$. We set the associated integers of $\{a,b\}$ to $(1,2,1)$. We note that $maxSupport(\{a,b\}) = 1$ is a sharp upper bound, since $support_2(\{a,b\}) = 1$.

Reading $t_3 = \{b,c\}$ we increment the count of $\emptyset$, $\{b\}$ and $\{c\}$. We then insert $\{b,c\}$ since $\{b\}$ and $\{c\}$ are elements of V with $maxSupport$ above the new user defined support threshold $\sigma_3 = 0.5$. By $\lceil\sigma\rceil_2 = (0.9,0.9,0,0,\ldots)$ we get $avg_2(\lceil\sigma\rceil_2) = 0.9$ and hence $\lfloor(3-1)\cdot 0.9\rfloor + 2 - 1 = 2$. Since $maxMissed(\{c\}) + count(\{c\}) - 1 = 1$ we get

$$maxMissed(\{b,c,\}) = \min\{2,1\} = 1.$$

Because all itemsets have a $maxSupport$ greater than 0.5 we can not remove any itemsets from the lattice. If σ_3 was 0.7 instead of 0.5 we would not have inserted $\{b,c\}$ while we could have removed $\{a,b\}$. However we could not have removed $\{c\}$, since our pruning strategy during *PhaseI* never removes singelton sets.

5.4 Changing Support Thresholds

We discuss in this subsection constant and changing support thresholds. PhaseI guarantees that all itemset with a support greater or equal to the support threshold given by Theorem 1 are included in the itemset lattice. We denote by the *guaranteed support threshold* this threshold, i.e.

$$avg_n(\lceil\sigma\rceil_n) + \frac{c+1}{n} \tag{4}$$

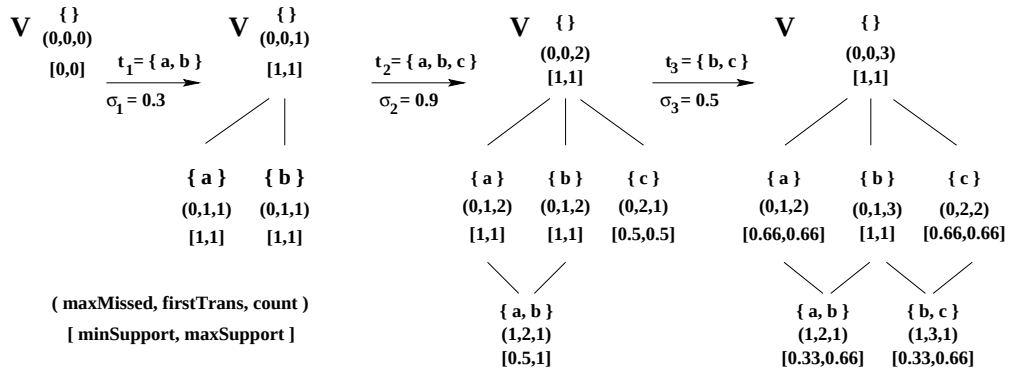


Figure 4

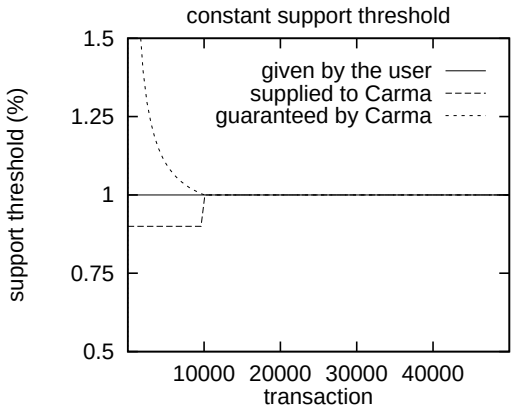


Figure 5

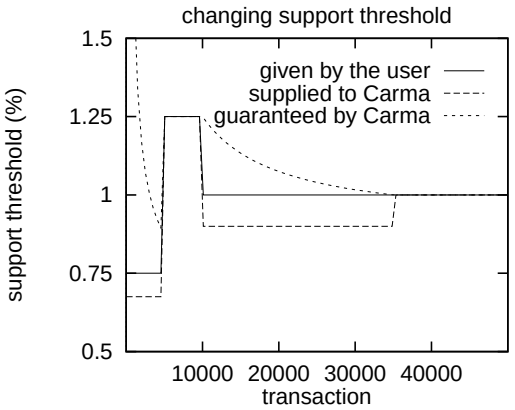


Figure 6

with σ the support sequence, c the maximal cardinality of a large itemset and n the current transaction index.

First, suppose the user does not change the support threshold. Hence we have a constant support sequence $\sigma = (s, s, s, \ldots)$ for some s. By (4) and $avg_n(\lceil\sigma\rceil_n) = s$ PhaseI includes at transaction n all large itemsets with a support $\geq s + \frac{c+1}{n}$. Thus, the guaranteed threshold $s + \frac{c+1}{n}$ converges to the user specified support threshold s as PhaseI scans the transaction sequence. To improve the speed of convergence, we run PhaseI with a lower threshold of $s \cdot 0.9$ instead of s. As the guaranteed threshold reaches s, we increase the threshold again from $s \cdot 0.9$ to s, see Figure 5.

Next, consider changing support thresholds. Figure 6 depicts a scenario, where the user increases at transaction 5'000 the initial support threshold of 0.75% to 1.25% and then lowers it again to 1.0% at transaction 10'000. As above, we supply PhaseI with thresholds σ_i lower than the user specified threshold, whenever the guaranteed threshold does not equal the user specified threshold. We set σ_i, the support sequence supplied to Carma, to $0.9 \cdot 0.75\% = 0.68\%$ for $i = 1, \ldots, 4'999$. The guaranteed threshold (4) drops quickly [3], reaching a value well below 1% at transaction 4'999. Since the new user specified threshold of 1.25% at transaction 5'000 is greater than 1%, we have equality until transaction 9'999. Hence we set σ_i to 1.25% for $i = 5'000, \ldots, 9'999$. As the user lowers the threshold to 1% we set σ_i to $0.9 \cdot 1\% = 0.9\%$ from $i = 10'000$ until the guaranteed threshold reaches 1.0% at transaction 35'000. We reset σ_i to 1% for all $i > 35'000$, since the user defined threshold remains at 1% from now on.

We note that the guaranteed threshold is an upper bound and thus a worst-case threshold. Typically, all large itemsets are contained in the lattice well before the guaranteed threshold reaches the user specified threshold.

6 Carma

In Subsection 6.1 we give a short description of PhaseII, the algorithm for the second scan. We then combine in Subsection 6.2 PhaseI with PhaseII, yielding Carma.

[3] For this example we assumed that all large itemsets are of cardinality 10 or less, i.e. $c = 10$.

```
Function PhaseII   (   support lattice V,
                       transaction sequence (t_1, ..., t_n),
                       support sequence σ )
                   :   support lattice;
integer ft, i = 0;
begin
 V := V\{v ∈ V | maxSupport(v) < σ_n };
 while ∃v ∈ V : i < firstTrans(v) do
  i++;
  for all v ∈ V do
   ft := firstTrans(v);
   if v ⊆ t_i and ft < i then
    count(v)++, maxMissed(v)–;
   fi;
   if ft == i then
    maxMissed(v) := 0;
    for all w ∈ V:
       v ⊂ w and maxSupport(w) > maxSupport(v)
do
      maxMissed(w) := count(v) − count(w);
    od;
   fi;
   if maxSupport(v) < σ_n then V := V\{v}; fi;
 od; od;
 return V;
end;
```

Figure 7

6.1 PhaseII

Let V be the support lattice computed by PhaseI and let σ_n be the user specified support threshold for the last transaction read during the first scan. PhaseII prunes all small itemsets from V and determines the precise support for all remaining itemsets.

Initially PhaseII removes all trivially small itemsets, i.e. itemsets with $maxSupport < \sigma_n$, from V. Scanning the transaction sequence, PhaseII increments *count* and decrements $maxMissed$ for each itemset contained in the current transaction, up to the transaction at which the itemset was inserted. Setting $maxMissed$ to 0 we get $minSupport = maxSupport$, the actual support of the itemset. We remove the itemset if it is small. Setting $maxMissed(v) = 0$ for an itemset v may yield $maxSupport(w) > maxSupport(v)$ for some superset w of v. Thus we set $maxMissed(w) = count(v) - count(w)$ for all supersets w of v with

$$maxSupport(w) > maxSupport(v).$$

PhaseII terminates as soon as the current transaction index is past $firstTrans$ for all itemsets in the lattice. The resulting lattice contains all large itemsets along with the precise support for each itemset. The algorithm is shown in figure 7.

Using Theorem 1 it is possible to determine that some itemset with $maxSupport > \sigma_n$ is small before we reach its $firstTrans$ transaction. Pruning these itemsets and all their supersets speeds up PhaseII by reducing the lattice size as well as the part of the transaction sequence which needs to be rescanned, c.f. [Hid98, Appendix D].

```
Function Carma   (   transaction sequence T,
                     support sequence σ)
                 :   support lattice;
support lattice V;
begin
 V := PhaseI( T, σ );
 V := PhaseII( V, T, σ );
 return V;
end;
```

Figure 8

6.2 Carma

Executing PhaseII after PhaseI, we get Carma, c.f. Figure 8. By Theorem 1 PhaseI produces a superset of all large itemsets with respect to the guaranteed threshold. PhaseII removes an itemset from the superset if and only if it is small. Thus the resulting itemset contains all large itemsets.

7 Implementation

To assess the performance we tested Carma along with Apriori and DIC on synthetic data generated by the IBM test data generator, c.f. [AS94] [1] . We illustrate our findings on the synthetic dataset with 100'000 transactions of an average size of 10 items chosen from 10'000 items and an average large itemset size of 4 (T10.I4.100K with 10K items). For runs on further datasets see Appendix A. All experiments were performed on a lightly loaded 300 MHz Pentium-II PC with 384 MB of RAM. The algorithms were implemented in Java on top of the same itemset lattice implementation. We cross compiled the Java class files to an executable using Tower Technology's TowerJ 2.2.

7.1 Implementation Details

Our implementation of an itemset lattice differs from a hashtree in that all itemsets are stored in a single hashtable. With the itemsets as keys, we can quickly access any subset of a given itemset. This is important for Carma, since whenever Carma inserts a new itemset v, it accesses all its maximal subsets to compute $maxMissed(v)$. We represent the lattice structure by associating to each itemset the set of all further items

[1] http://www.almaden.ibm.com/cs/quest/syndata.html

[3] computed on T10.I4.100K with 10K items at a support threshold of 0.1%

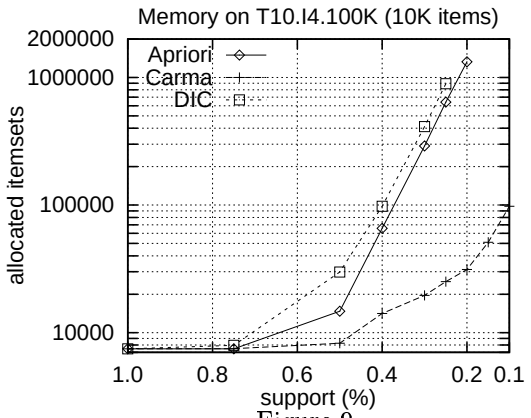


Figure 9

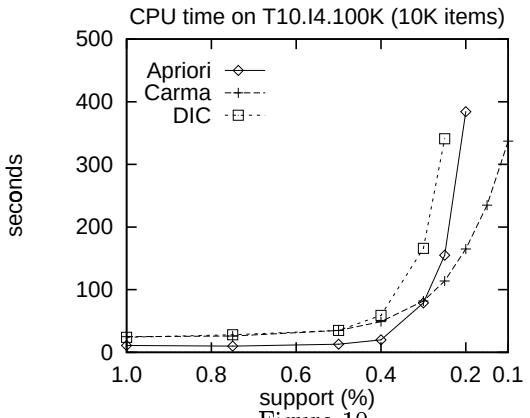


Figure 10

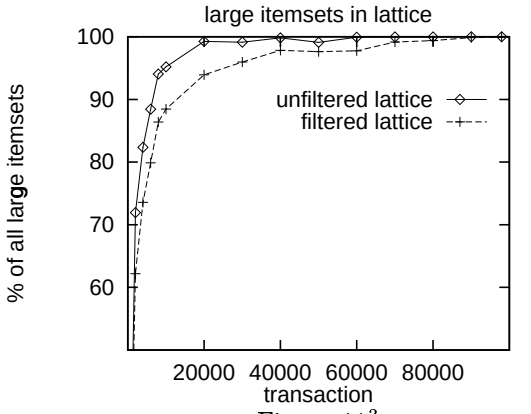


Figure 11[3]

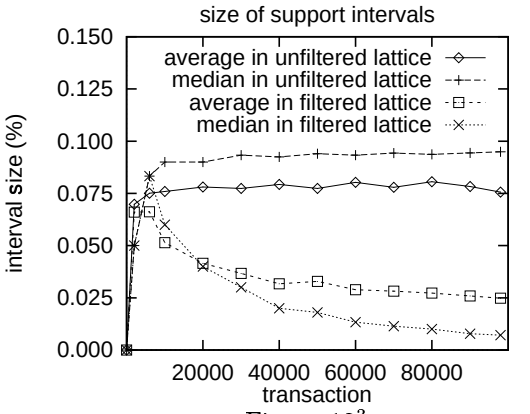


Figure 12[3]

appearing in any of its supersets, c.f. [Bay98]. As in the case of a hashtree, we need only one hashtable access to pass from an itemset to one of its minimal supersets. Thus we can enumerate all subsets of a scanned transaction, which are contained in the lattice, as quickly as in a hashtree.

Our implementation of Carma diverges from the pseudo-code given in Subsection 5.2 only in that we perform the PhaseI incrementation and insertion step simultaneously, enumerating the subsets of a scanned transaction once.

Apriori and DIC were implemented as described in [AS94] and [BMUT97] respectively. For DIC we chose a blocksize of 15000, which we found to be fastest.

7.2 Relative Performance

To compare Carma with Apriori and DIC we ran all three algorithms on a range of datasets and (constant) support thresholds. In this subsection we illustrate our results on the T10.I4.100K dataset with 10K items. For support thresholds of 0.5% and above Apriori outperformed Carma and DIC. We attribute the superior speed of Apriori for these thresholds to the observation that, for example, at 0.75% only 171 large itemsets existed and all large itemsets were 1-itemsets. Thus Apriori completes in 2 scans allocating only 300 2-itemsets. As the support threshold was lowered to 0.25% (0.1%) the number of large 1-itemsets increased to 1131 (3509) and the maximal cardinality to 4 (9). We were not able to run Apriori (DIC) with thresholds below 0.2% (0.25%), since the allocated itemsets did not fit into main memory anymore. At 0.15% (0.1%) Apriori would have allocated 2.8 million (6.2 million) 2-itemsets[2], while Carma required only 51001 (97872) itemsets. We note that DIC always allocates at least as many itemsets as Apriori. At 0.25% and below Carma outperformed Apriori and DIC.

We attribute the better performance of Carma over Apriori to the 4 scans needed by Apriori while Carma completed in 1.1 scans. We attribute the better performance of Carma over DIC to the 2 scans needed by DIC as well as to the 35 times smaller lattice maintained by Carma, since both algorithms traverse their lattices in regular intervals.

7.3 Support Intervals

PhaseI maintains a superset of the large itemsets in the scanned part of the transaction sequence, but not necessarily a superset for the full transaction sequence. First, we wanted to determine the percentage

[2]The number of candidate 2-itemsets which Apriori allocates is given by the number of large 1-itemsets over 2.

of all large itemsets, i.e. with respect to the full transaction sequence, contained in the lattice as PhaseI proceeds. After scanning 20000 (40000) transactions at a threshold of 0.1% Carma included 99.3% (99.8%) of all large itemsets in its lattice, see figure 11.

Between two pruning steps PhaseI replaced up to 50% of all itemsets. The vast majority (typically $> 95\%$) of itemsets in the lattice eventually turned out to be small. As we scan the transaction sequence we would present a large number of association rules to the user based on itemsets which are likely to be small. To exclude those itemsets from the rule generation, which are likely to be small, we filtered out all itemsets which were inserted during the last 15% of the transaction sequence, e.g. at transaction 20000 we filter out all itemsets which were inserted at transaction 17000 or later. The filtered lattice still contained 93.9% (97.8%) of all large itemsets, after scanning 20000 (40000) transactions respectively, c.f. figure 9. At the same time the size of the filtered lattice was reduced to 32.6% (16.0%) of its original size.

Recall that the support interval of an itemset in the lattice is given by its *minSupport* and *maxSupport*. Next, we wanted to determine how the size of the support intervals, i.e. $maxSupport - minSupport$, in the filtered lattice evolve as PhaseI proceeds. After scanning 20000 (40000) transactions at a threshold of 0.1% the average interval size in the filtered lattice was 0.042% (0.032%), while 50% of all itemsets in the lattice had an interval size below 0.004% (0.002%), c.f. figure 12.

8 Conclusion

We presented Carma, a novel algorithm to compute large itemsets online. It continuously produces large itemsets along with a shrinking support interval for each itemset. It allows the user to change the support threshold anytime during the first scan and always completes in at most 2 scans.

We implemented Carma and compared it to Apriori and DIC. While not being faster in general, Carma outperforms Apriori and DIC on low support thresholds. We attributed this to the observation that Carma is typically an order of magnitude more memory efficient. We showed that Carma's itemset lattice quickly approximates a superset of all large itemsets while the sizes of the corresponding support intervals shrink rapidly. We also showed that Carma readily computes large itemsets in cases which are intractable for Apriori or DIC.

An interesting feature of the algorithm is that the second scan ist not needed, whenever the shrinking support intervals suffice. Thus PhaseI can be used to continuously compute large itemsets from a transaction sequence read from a network, generalizing incremental updates and not requiring local storage.

Acknowledgement: I would like to thank Joseph M. Hellerstein, UC Berkeley, for his inspiration, guidance and support. I am thankful to Ron Avnur for the many discussions and to Retus Sgier, for his help and suggestions. I would like to thank Rajeev Motwani, Stanford University, for pointing out the applicability of Carma to transaction sequences read from a network. Also, I would like to thank Ramakrishnan Srikant, IBM Almaden Research Center, for his remarks on speeding up the convergence of the support thresholds.

References

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 207–216, Washington, D.C., May 1993.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Int'l Conf. on Very Large Databases*, Santiago, Chile, Sept. 1994.

[AY97] Charu C. Aggarwal and Philip S. Yu. Online generation of association rules. Technical Report RC 20899 (92609), IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, June 1997.

[AY98] Charu C. Aggarwal and Philip S. Yu. Mining large itemsets for association rules. *Bulletin of the IEEE Computer Society Technical Comittee on Data Engineering*, pages 23–31, March 1998.

[Bay98] R. J. Bayardo Jr. Efficiently mining long patterns from databases. In *Proc. of the 1998 ACM-SIGMOD International Conference on Management of Data*, pages 85–93, Seattle, June 1998.

[BMUT97] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 26,2 of *SIGMOD Record*, pages 255–264, New York, May 13th–15th 1997. ACM Press.

[CHNW96] D. Cheung, J. Han, V. Ng, and C.Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proc. of 1996 Int'l Conf. on Data Engineering (ICDE'96)*, New Orleans, Louisiana, USA, Feb. 1996.

[CLK97] David W. L. Cheung, S.D. Lee, and Benjamin Kao. A general incremental technique for maintaining discovered association rules. In *Proceedings of the Fifth International Conference On Database Systems For Advanced Applications*, pages 185–194, Melbourne, Australia, March 1997.

[Hel96] Joseph M. Hellerstein. The case for online aggregation. Technical Report UCB//CSD-96-908, EECS Computer Science Divison, University of California at Berkeley, 1996.

[HHW97] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. SIGMOD '97, 1997.

[Hid98] C. Hidber. Online Association Rule Mining. Technical Report TR-98-033, International Computer Science Institute, Berkeley, CA, September 1998. an earlier version appeared as technical report UCB//CSD-98-1004 of the Department of Electrical Engineering and Computer Science, University of California at Berkeley.

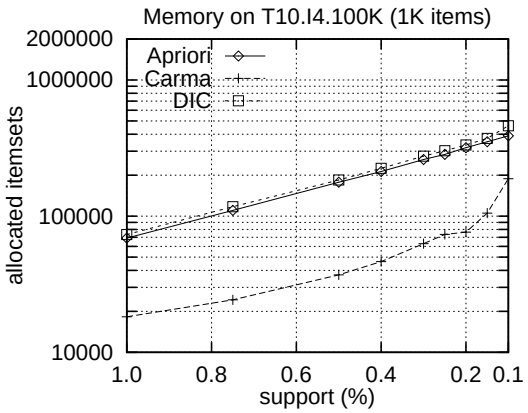
[SON95] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the Very Large Data Base Conference*, September 1995.

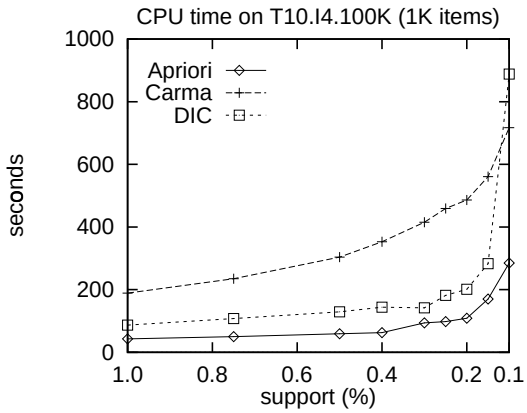
[TBAR97] Shiby Thomas, Sreenath Bodagala, Khaled Alsabti, and Sanjay Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *Proceedings of the 3rd International conference on Knowledge Discovery and Data Mining (KDD 97)*, New Port Beach, California, August 1997.

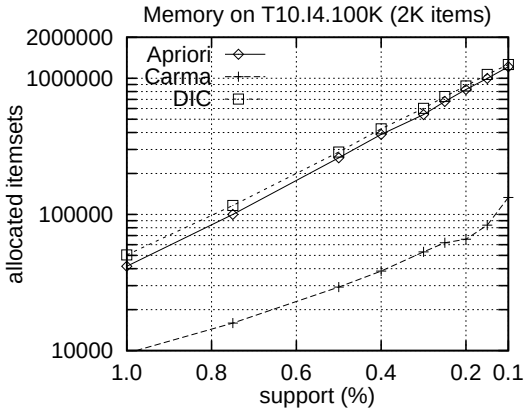
[Toi96] Hannu Toivonen. Sampling large databases for association rules. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases*, Mumbai (Bombay), India, September 1996. Morgan Kaufmann.

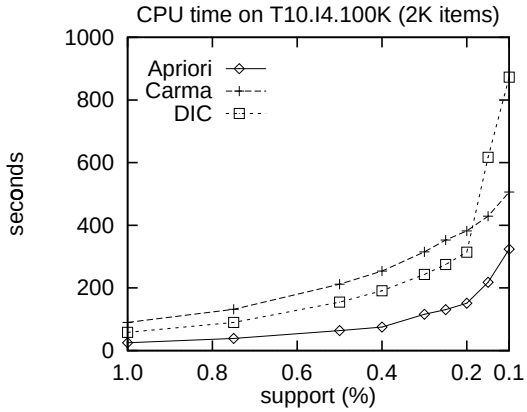
[ZPLO96] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Wei Li, and Mitsunori Ogihara. Evaluation of sampling for data mining of association rules. Technical Report 617, Computer Science Dept., U. Rochester, May 1996.

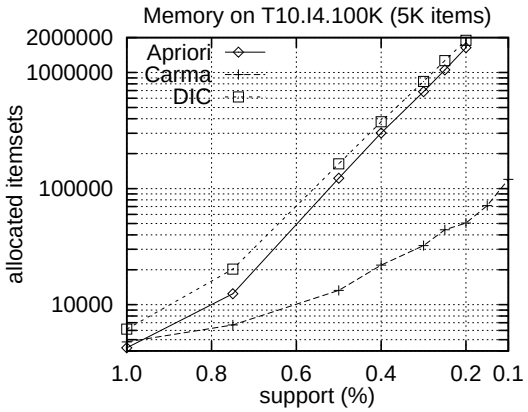
A Performance Figures

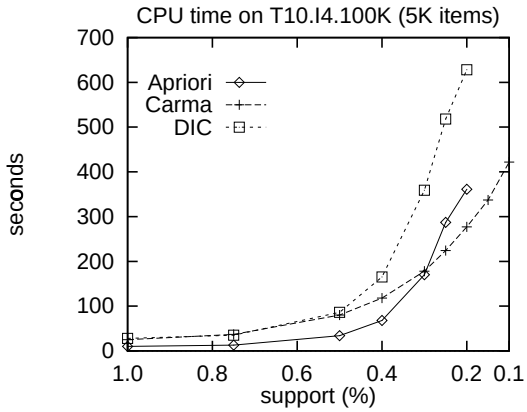


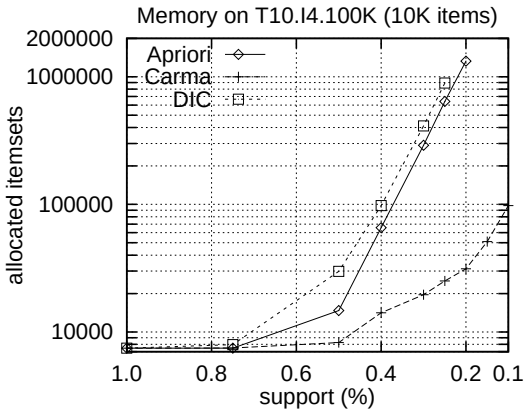


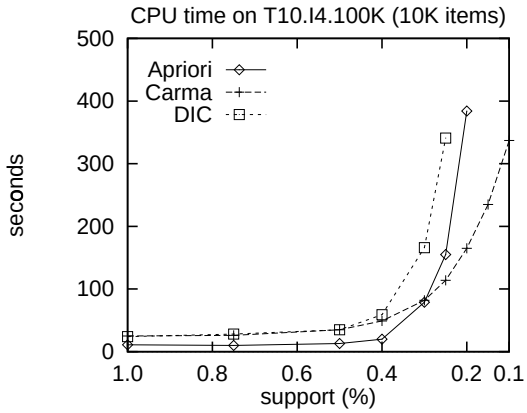


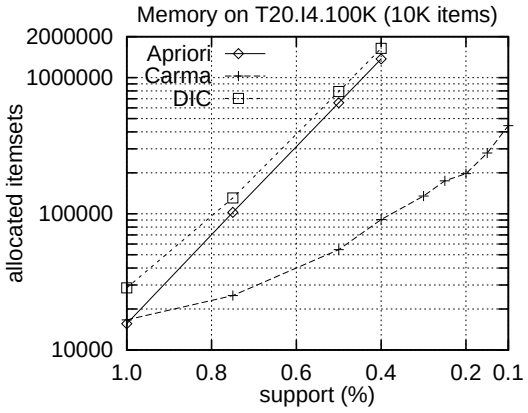


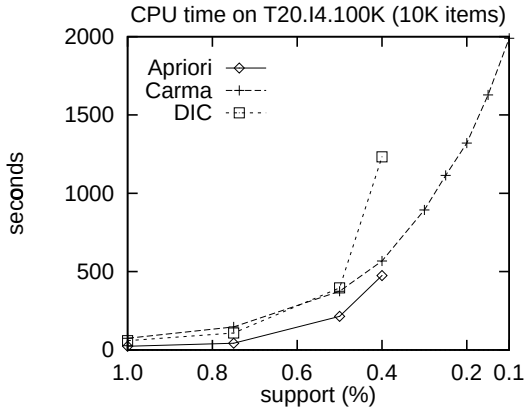


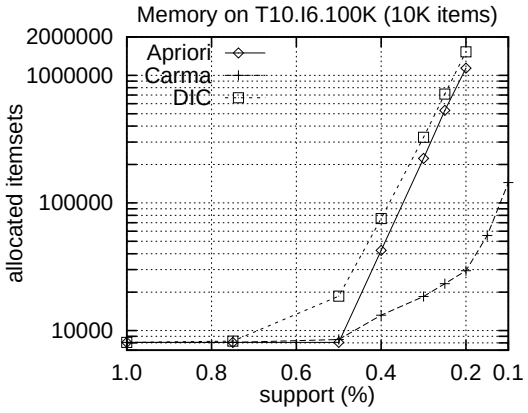


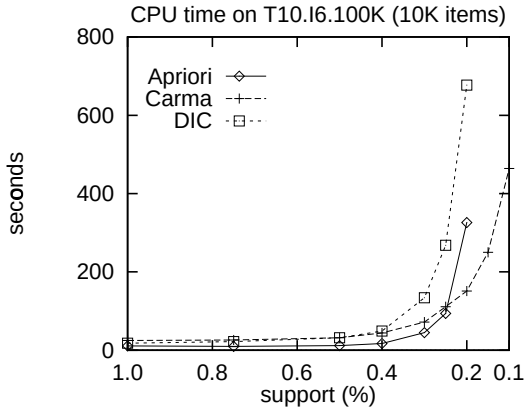

Memory on T10.I4.100K (10K items)
allocated itemsets
2000000
1000000
100000
10000
Apriori
Carma
DIC
1.0
0.8
0.6
0.4
0.2
0.1
support (%)


CPU time on T10.I4.100K (10K items)
seconds
500
400
300
200
100
0
Apriori
Carma
DIC
1.0
0.8
0.6
0.4
0.2
0.1
support (%)


Memory on T20.I4.100K (10K items)
allocated itemsets
2000000
1000000
100000
10000
Apriori
Carma
DIC
1.0
0.8
0.6
0.4
0.2
0.1
support (%)


CPU time on T20.I4.100K (10K items)
seconds
2000
1500
1000
500
0
Apriori
Carma
DIC
1.0
0.8
0.6
0.4
0.2
0.1
support (%)


Memory on T10.I6.100K (10K items)
allocated itemsets
2000000
1000000
100000
10000
Apriori
Carma
DIC
1.0
0.8
0.6
0.4
0.2
0.1
support (%)


CPU time on T10.I6.100K (10K items)
seconds
800
600
400
200
0
Apriori
Carma
DIC
1.0
0.8
0.6
0.4
0.2
0.1
support (%)