Dynamic Itemset Counting and Implication Rules for Market Basket Data

Sergey Brin * **Rajeev Motwani** **Jeffrey D. Ullman**
Department of Computer Science
Stanford University
{sergey,rajeev,ullman}@cs.stanford.edu

Shalom Tsur
R&D Division, Hitachi America Ltd.
tsur@hitachi.com

Abstract

We consider the problem of analyzing market-basket data and present several important contributions. First, we present a new algorithm for finding large itemsets which uses fewer passes over the data than classic algorithms, and yet uses fewer candidate itemsets than methods based on sampling. We investigate the idea of item reordering, which can improve the low-level efficiency of the algorithm. Second, we present a new way of generating "implication rules," which are normalized based on both the antecedent and the consequent and are truly implications (not simply a measure of co-occurrence), and we show how they produce more intuitive results than other methods. Finally, we show how different characteristics of real data, as opposed to synthetic data, can dramatically affect the performance of the system and the form of the results.

1 Introduction

Within the area of data mining, the problem of deriving associations from data has recently received a great deal of attention. The problem was first formulated by Agrawal *et al*, [AIS93a, AIS93b, AS94, AS95, ALSS95, SA95, MAR96, Toi96] and is often referred to as the "market-basket" problem. In this problem, we are given a set of items and a large collection of transactions which are subsets (baskets) of these items. The task is to find relationships between the presence of various items within those baskets.

There are numerous applications of data mining which fit into this framework. The canonical example from which the problem gets its name is a supermarket. The items are products and the baskets are customer purchases at the checkout. Determining what products customers are likely to buy together can be very useful for planning and marketing. However, there are many other applications which have varied data characteristics. For example, student enrollment in classes, word occurrence in text documents, users' visits of web pages, and many more. We applied market-basket analysis to census data (see section 5).

In this paper, we address both performance and functionality issues of market-basket analysis. We improve performance over past methods by introducing a new algorithm for finding large itemsets (an important subproblem). We enhance functionality by introducing *implication rules* as an alternative to association rules (see below).

One very common formalization of this problem is finding *association rules* which are based on *support* and *confidence*. The support of an itemset (a set of items), I, is the fraction of transactions the itemset occurs in (is a subset of). An itemset is called *large* if its support exceeds a given threshold, σ. An association rule is written $I \rightarrow J$ where I and J are itemsets [1]. The *confidence* of this rule is the fraction of transactions containing I that also contain J. For the association rule, $I \rightarrow J$ to hold, $I \cup J$ must be large and the confidence of the rule must exceed a given confidence threshold, γ. In probability terms, we can write this $P(\{I \cup J\}) > \sigma$ and $P(J|I) > \gamma$.

The existing methods for deriving the rules consist of two steps:

1. Find the large itemsets for a given σ.
2. Construct rules which exceed the confidence threshold from the large itemsets in step 1. For example, if ABC is a large itemset we might check the confidence of $AB \rightarrow C$, $AC \rightarrow B$ and $BC \rightarrow A$.

In this paper we address both of these tasks, step 1 from a performance perspective by devising a new algorithm, and step 2 from a semantic perspective by developing *conviction*, an alternative to confidence.

1.1 Algorithms for Finding Large Itemsets

Much research has focussed on deriving efficient algorithms for finding large itemsets (step 1). The most well-known algorithm is Apriori [AIS93b, AS94] which, as all algorithms for finding large itemsets, relies on the property that an itemset can only be large if and only if all of its subsets are large. It proceeds level-wise. First it counts all the 1-itemsets[2] and finds counts which exceed the threshold - the large 1-itemsets. Then it combines those to form *candidate* (potentially large) 2-itemsets, counts them and determines which are the large 2-itemsets. It continues by combining the large

*Work done at R&D Division of Hitachi America Ltd. Also supported by a fellowship from the NSF.

[1] J is typically restricted to just one item though it doesn't have to be.

[2] A k-itemset is an itemset with k items.

2-itemsets to form candidate 3-itemsets, counting them and determining which are the large 3-itemsets and so forth.

Let L_k be the set of large k-itemsets. For example, L_3 might contain $\{\{A, B, C\}, \{A, B, D\}, \{A, D, F\}, \ldots\}$. Let C_k be the set of *candidate* k-itemsets; this is always a superset of L_k. Here is the algorithm:

```
Result := ∅;
k := 1;
C_1 = set of all 1-itemsets;
while C_k ≠ ∅ do
        create a counter for each itemset in C_k;
        forall transactions in database do
                Increment the counters of itemsets in C_k
                        which occur in the transaction;
        L_k := All candidates in C_k
                        which exceed the support threshold;
        Result := Result ∪ L_k;
        C_{k+1} := all k + 1-itemsets
                which have all of their k-item subsets in L_k.
        k := k + 1;
end
```

Thus, the algorithm performs as many passes over the data as the maximum number of elements in a candidate itemset, checking at pass k the support for each of the candidates in C_k. The two important factors which govern performance are the number of passes made over all the data and the efficiency of those passes.

To address both of those issues we introduce Dynamic Itemset Counting (DIC), an algorithm which reduces the number of passes made over the data while keeping the number of itemsets which are counted in any pass relatively low as compared to methods based on sampling [Toi96]. The intuition behind DIC is that it works like a train running over the data with stops at intervals M transactions apart. (M is a parameter; in our experiments we tried values ranging from 100 to 10,000.) When the train reaches the end of the transaction file, it has made one pass over the data and it starts over at the beginning for the next pass. The "passengers" on the train are itemsets. When an itemset is on the train, we count its occurrence in the transactions that are read.

If we consider Apriori in this metaphor, all itemsets must get on at the start of a pass and get off at the end. The 1-itemsets take the first pass, the 2-itemsets take the second pass, and so on (see Figure 1). In DIC, we have the added flexibility of allowing itemsets to get on at any stop as long as they get off at the same stop the next time the train goes around. Therefore, the itemset has "seen" all the transactions in the file. This means that we can start counting an itemset as soon as we suspect it may be necessary to count it instead of waiting until the end of the previous pass.

For example, if we are mining 40,000 transactions and $M = 10,000$, we will count all the 1-itemsets in the first 40,000 transactions we will read. However, we will begin counting 2-itemsets after the first 10,000 transactions have been read. We will begin counting 3-itemsets after 20,000 transactions. For now, we assume there are no 4-itemsets we need to count. Once we get to the end of the file, we will stop counting the 1-itemsets and go back to the start of the file to count the 2 and 3-itemsets. After the first 10,000 transactions, we will finish counting the 2-itemsets and after 20,000 transactions, we will finish counting the 3-itemsets. In total, we have made 1.5 passes over the data instead of the 3 passes a level-wise algorithm would make.[3]

DIC addresses the high-level issues of when to count which itemsets and is a substantial speedup over Apriori, particularly when Apriori requires many passes. We deal with the low-level issue of how to increment the appropriate counters for each transaction in Section 3 by considering the sort order of items in our data structure.

1.2 Implication Rules

Our contribution to functionality in market basket analysis is *implication rules* based on conviction, which we believe is a more useful and intuitive measure than confidence and interest (see discussion in Section 4). Unlike confidence, conviction is normalized based on both the antecedent and the consequent of the rule like the statistical notion of correlation. Furthermore, unlike interest, it is directional and measures actual implication as opposed to co-occurrence. Because of these two features, implication rules can produce useful and intuitive results on a wide variety of data. For example, the rule *past active duty in military* $\Rightarrow$ *no service in Vietnam* has a very high confidence of 0.9. Yet it is clearly misleading since having past military service only increases the chances of having served in Vietnam. In tests on census data, the advantages of conviction over rules based on confidence or interest are evident.

In Section 5, we present the results of generating implication rules for U.S. census data from the 1990 census. Census data is considerably more difficult to mine than supermarket data and the performance advantages of DIC for finding large itemsets are particularly useful.

2 Counting Large Itemsets

Itemsets form a large lattice with the empty itemset at the bottom and the set of all items at the top (see example, figure 2). Some itemsets are large (denoted by boxes), and the rest are small. Thus, in the example, the empty itemset, $A, B, C, D, AB, AC, BC, BD, CD, ABC$ are large.

To show that the itemsets are large we can count them. In fact, we must, since we generally want to know the counts. However, it is infeasible to count all of the small itemsets. Fortunately, it is sufficient to count just the minimal ones (the itemsets that do not include any other small itemsets) since if an itemset is small, all of its supersets are small too. The minimal small itemsets are denoted by circles; in our example AD and BCD are minimal small. They form the top side of the boundary between the large and small itemsets (Toivonen calls this the negative boundary; in lattice theory the minimal small itemsets are called the prime implicants).

An algorithm which counts all the large itemsets must find and count all of the large itemsets and the minimal small itemsets (that is, all of the boxes and circles). The DIC algorithm, described here, marks itemsets in four different possible ways:

- Solid box - confirmed large itemset - an itemset we have finished counting that exceeds the support threshold.
- Solid circle - confirmed small itemset - an itemset we have finished counting that is below the support threshold.
- Dashed box - suspected large itemset - an itemset we are still counting that exceeds the support threshold.

[3] Of course in this example we assumed best of all possible circumstances where we estimated correctly exactly which 2 and 3-itemsets we would have to count. More realistically, some itemsets would be added a little later. Nonetheless there would still be considerable savings.

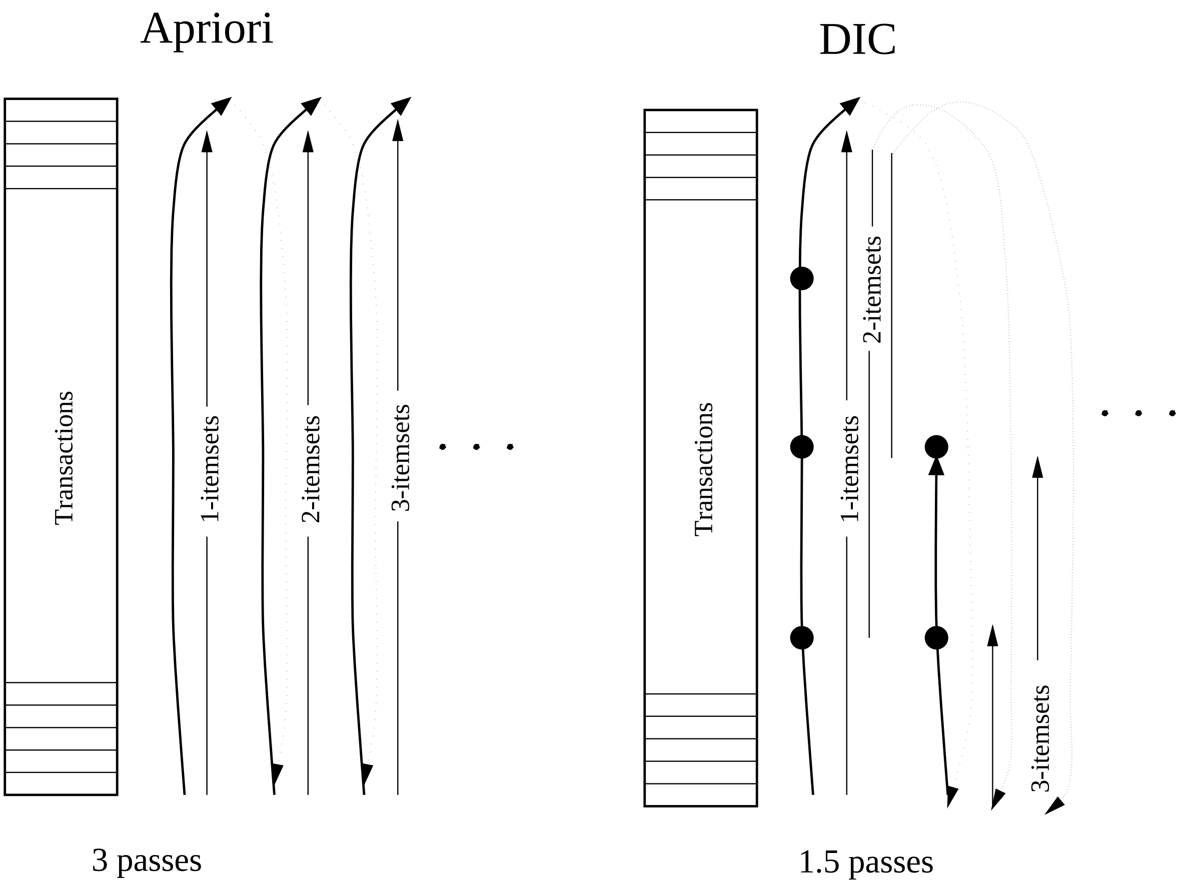

Figure 1: Apriori and DIC

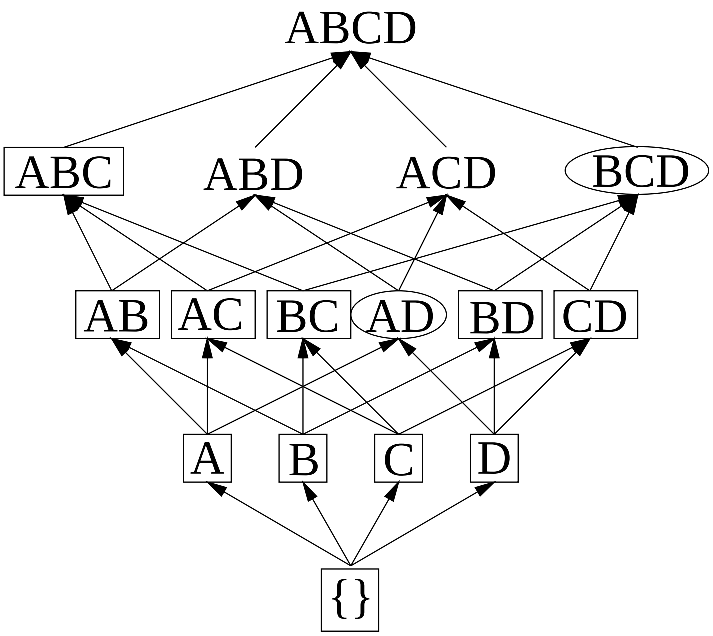

Figure 2: An itemsets lattice.

- Dashed circle - suspected small itemset - an itemset we are still counting that is below the support threshold.

The DIC algorithm works as follows:

1. The empty itemset is marked with a solid box. All the 1-itemsets are marked with dashed circles. All other itemsets are unmarked. (See Figure 3.)
2. Read M transactions. We experimented with values of M ranging from 100 to 10,000. For each transaction, increment the respective counters for the itemsets marked with dashes. See section 3.
3. If a dashed circle has a count that exceeds the support threshold, turn it into a dashed square. If any immediate superset of it has all of its subsets as solid or dashed squares, add a new counter for it and make it a dashed circle. (See Figures 4 and 5.)
4. If a dashed itemset has been counted through all the transactions, make it solid and stop counting it.
5. If we are at the end of the transaction file, rewind to the beginning. (See Figure 6.)
6. If any dashed itemsets remain, go to step 2.

This way DIC starts counting just the 1-itemsets and then quickly adds counters 2,3,4,...,k-itemsets. After just a few passes over the data (usually less than two for small values of M) it finishes counting all the itemsets. Ideally,

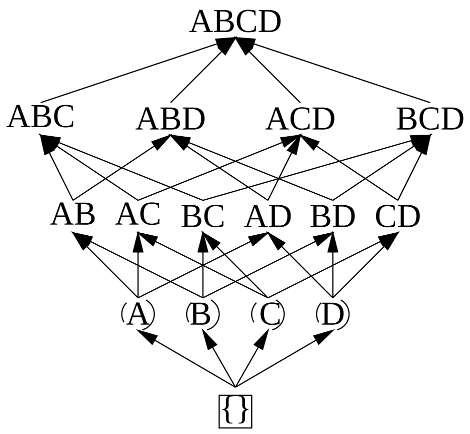

Figure 3: Start of DIC algorithm.

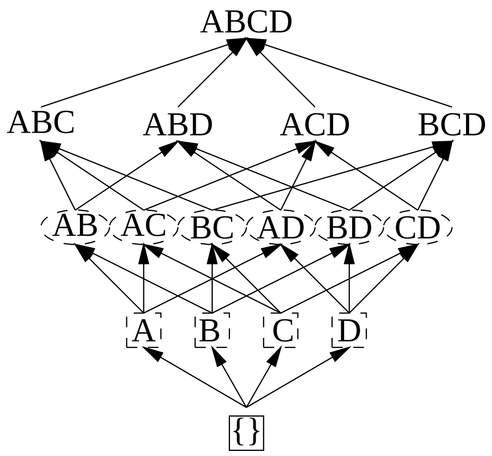

Figure 4: After M transactions.

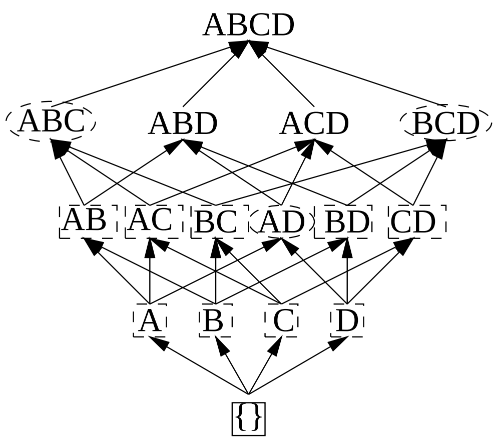

Figure 5: After $2M$ transactions.

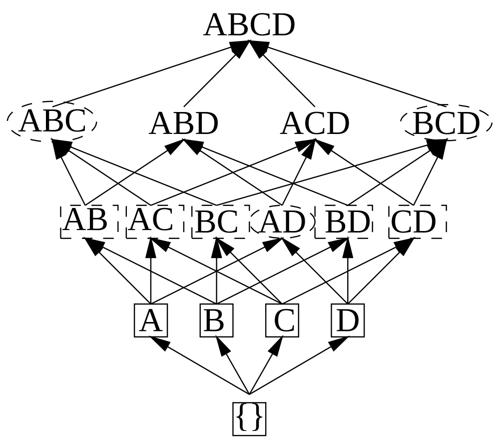

Figure 6: After one pass.

we would want M to be as small as possible so we can start counting itemsets very early in step 3. However, steps 3 and 4 incur considerable overhead so we do not reduce M below 100.

2.1 The Data Structure

The implementation of the DIC algorithm requires a data structure which can keep track of many itemsets. In particular, it must support the following operations:

1. Add new itemsets.

2. Maintain a counter for every itemset. When transactions are read, increment the counters of those active itemsets which occur in the transaction. This must be very fast as it is the bottleneck of the whole process. We attempt to optimize this operation in Section 3.

3. Maintain itemset states by managing transitions from active to counted (dashed to solid) and from small to large (circle to square). Detect when these transitions should occur.

4. When itemsets do become large, determine what new itemsets should be added as dashed circles since they could now potentially be large.

The data structure used for this is exactly like the hash tree used in Apriori with a little extra information stored at each node. It is a trie with the following properties. Each itemset is sorted by its items (the sort order is discussed in Section 3). Every itemset we are counting or have counted has a node associated with it, as do all of its prefixes. The empty itemset is the root node. All the 1-itemsets are attached to the root node, and their branches are labeled by the item they represent. All other itemsets are attached to their prefix containing all but their last item. They are labeled by that last item. Figure 7 shows a sample trie of this form. The dotted path represents the traversal which is made through the trie when the transaction ABC is encountered so A, AB, ABC, AC, B, BC, and C must be incremented and they are,

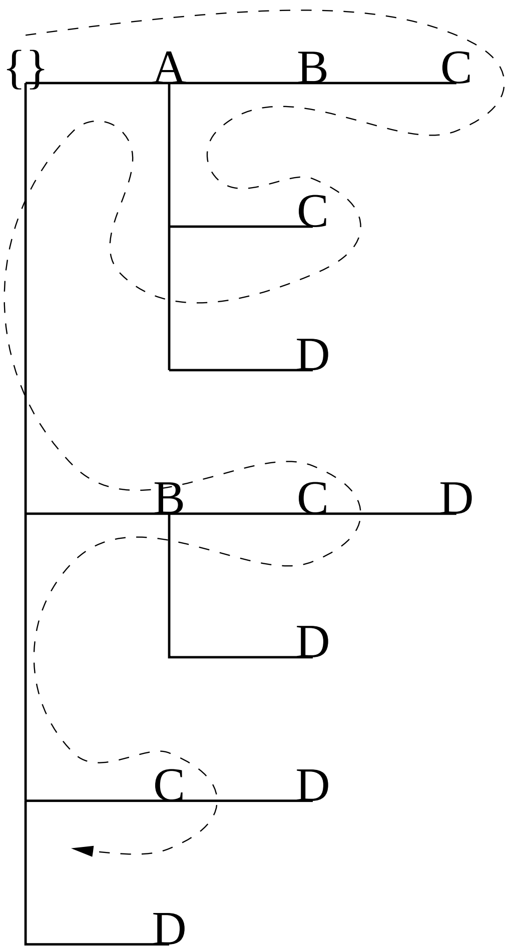

Figure 7: Hash Tree Data Structure

in that order. The exact algorithm for this is described in Section 3.

Every node stores the last item in the itemset it represents, a counter, a marker as to where in the file we started counting it, its state, and its branches if it is an interior node.

2.2 Significance of DIC

There are a number of benefits to DIC. The main one is performance. If the data is fairly homogeneous throughout the file and the interval M is reasonably small, this algorithm generally makes on the order of two passes. This makes the algorithm considerably faster than Apriori which must make as many passes as the maximum size of a candidate itemset. If the data is not fairly homogeneous, we can run through it in a random order (section 2.3).

Some important relevant work was done by Toivonen using sampling [Toi96]. His technique was to sample the data using a reduced threshold for safety, and then count the necessary itemsets over the whole data in just one pass. However, this pays the added penalty of having to count more itemsets due to the reduced threshold. This can be quite costly, particularly for datasets like the census data (see section 5). Instead of being conservative, our algorithm bravely marches on, on the assumption that it will later come back to anything missed with little penalty.[4]

Besides performance, DIC provides considerable flexibility by having the ability to add and delete counted itemsets on the fly. As a result, DIC can be extended to parallel and incremental update versions (see section 6.1.1).

2.3 Non-homogeneous Data

One weakness of DIC is that it is sensitive to how homogeneous the data is. In particular, if the data is very correlated, we may not realize that an itemset is actually large until we have counted it in most of the database. If this happens, then we will not shift our hypothetical boundary and start counting some of the itemset's supersets until we have almost finished counting the itemset. As it turns out, the census data we used is ordered by census district and exactly this problem occurs. To test the impact of this effect, we randomized the order of the transactions and re-ran DIC. It turned out to make a significant difference in performance (see Section 5). The cost associated with randomizing transaction order is small compared to the mining cost.

However, randomization may be impractical. For example, it may be expensive, the data may be stored on tape, or there might be insufficient space to store the randomized version. We considered several ways of addressing this problem:

- Virtually randomize the data. That is, visit the file in a random order while making sure that every pass is in the same order. This can incur a high seek cost, especially if the data is on tape. In this case, it may be sufficient to jump to a new location every few thousand transactions or so.
- Slacken the support threshold. First, start with a support threshold considerably lower than the given one. Then, gradually increase the threshold to the desired level. This way, the algorithm begins fairly conservative and then becomes more confident as more data is collected. We experimented with this technique somewhat but with little success. However, perhaps more careful control of the slack or a different dataset would make this a useful technique.
- One thing to note is that if the data is correlated with its location in the file, it may be useful to detect this and report it. This is possible if a "local" counter is kept along with each itemset which measures the count of the current interval. At the end of each interval it can be checked for considerable discrepancies with its overall support in the whole data set.

The DIC algorithm addresses the high-level strategy of what itemsets to count when. There are also lower level performance issues as to how to increment the appropriate counters for a particular transaction. We address these in section 3.

3 Item Reordering

The low-level problem of how to increment the appropriate counters for a given transaction is an interesting one in itself. Recall that the data structure we use is a trie structure much like that used in Apriori (see section 2.1). Given a collection

[4] We did not have time to implement and test Toivonen's algorithm as compared to ours. However, based on tests with lowered support thresholds, we suspect that DIC is quite competitive.

$\emptyset$	7
A	6
AB	5
B	5
BC	4
C	4
total[a]	31

Table 1: Increment Cost for ABCDEFG

[a] ABC, AC, AD, BCD, BD, CD, D, E, F, and G cost 0 since they are leaves.

$\emptyset$	7
A	3
AB	2
B	2
BC	1
C	1
total	16

Table 2: Increment Cost for EFGABCD

of itemsets, the form of this structure is heavily dependent on the sort order of the items. Note that in our sample data structure (figure 7), the order of the items was A,B,C,D. Because of this, A occurs only once in the trie while D occurs five times.

To determine how to optimize the order of the items, it is important to understand how the counter incrementing process works. We are given a transaction, S (with items $S[0] \ldots S[n]$), in a certain order. To increment the appropriate counters we do the following, starting at the root node of the trie T:

```
Increment(T,S) {
        /* Increment this node's counter */
        T.counter++;
        If T is not a leaf then forall i, 0 ≤ i ≤ n:
        /* Increment branches as necessary */
        If T.branches[S[i]] exists:
           Then Increment(T.branches[S[i]], S[i+1..n])
        Return. }
```

Therefore, the cost of running this subroutine is equal to:

$$\sum_I n - Index(Last(I), S)$$

where I ranges over non-leaf itemsets in T which occur in S, and $n - Index(Last(I), S)$ is the number of items left in S after the last element of I. These items will be checked in the inner loop. Therefore, it is advantageous to have the items which occur in many itemsets to be last in the sort order of the items (so few items will be left after them) and the items which occur in few itemsets to be first.

For example, consider the structure in figure 7. Suppose there are also items E,F, and G, and we add their respective 1-itemsets to the data structure. There will now be three singletons hanging off the tree. If we insert ABCDEFG, the cost of the insert is 31 (see Table 3). However, if we change the order of the items to EFGABCD (note the tree structure remains the same since A,B,C,D did not change order), the cost becomes 16 (see Table 3).

This is considerably cheaper. Therefore what we want is to order the items by the inverse of their popularity in the counted non-leaf itemsets. A reasonable approximation for this inverse is the inverse of their popularity in the first M transactions. Since during the first interval of transactions we are counting only 1-itemsets, there is not yet a tree structure which depends on the order. After the first M transactions, we change the order of the items and build the tree from there. Future transactions must be resorted according to the new ordering. This technique incurs some overhead due to the re-sorting, but for some data it can be beneficial overall.

4 Implication rules

Some traditional measures of "interestingness" have been support combined with either confidence or *interest*. Consider these measures from a probabilistic model.

Let $\{A, B\}$ be an itemset. Then the support is $P(\{A, B\})$ which we write $P(A, B)$. This is used to make sure that the items this rule applies to actually occur frequently enough for someone to care. It also makes the task computationally feasible by limiting the size of the result set, and is usually used in conjunction with other measures. The confidence of $A \Rightarrow B$ is $P(B \mid A)$, the conditional probability of B given A, which is equal to $P(A, B)/P(A)$. It has the flaw that it ignores $P(B)$. For example, $P(A, B)/P(A)$ could equal $P(B)$ (i.e. the occurrence of B is unrelated to A) and could still be high enough to make the rule hold. For example, if people buy milk 80% of the time in a supermarket and the purchase of milk is completely unrelated to the purchase of smoked salmon, then the confidence of *salmon* $\Rightarrow$ *milk* is still 80%. This confidence is quite high, and therefore would generate a rule. This is a key weakness of confidence, and is particularly evident in census data, where many items are very likely to occur with or without other items.

The *interest* of A, B is defined as $P(A, B)/P(A)P(B)$ and factors in both $P(A)$ and $P(B)$; essentially it is a measure of departure from independence. However, it only measures co-occurrence not implication, in that it is completely symmetric.

To fill the gap, we define *conviction* as $P(A)P(\neg B)/P(A, \neg B)$. The intuition as to why this is useful is: logically, $A \rightarrow B$ can be rewritten as $\neg(A \wedge \neg B)$ so we see how far $A \wedge \neg B$ deviates from independence, and invert the ratio to take care of the outside negation [5]. We believe this concept is useful for a number of reasons:

- Unlike confidence, conviction factors in both $P(A)$ and $P(B)$ and always has a value of 1 when the relevant items are completely unrelated like the salmon and milk example above.
- Unlike interest, rules which hold 100% of the time, like *Vietnam veteran* $\Rightarrow$ *more than five years old* have the highest possible conviction value of ∞. Confidence also has this property in that these rules have a confidence of 1. However, interest does not have this useful property. For example, if 5% of people are Vietnam veterans and 90% are more than five years old, we get interest $= 0.05/(0.05) * 0.09 = 1.11$ which is only slightly above 1 (the interest for completely independent items).

[5] In practice we do not invert the ratio; instead we search for low values of the uninverted ratio. This way we do not have to deal with infinities.

In short, conviction is truly a measure of implication because it is directional, it is maximal for perfect implications, and it properly takes into account both $P(A)$ and $P(B)$.

5 Results

We tested the DIC algorithm along with reordering on two different types of data: synthetic data generated by the IBM test data generator [6] and U.S. census data. We also tested implication rules on both sets of data but the results of these tests are only interesting on the census data. This is because the synthetic data is designed to test association rules based on support and confidence. Also, the rules generated are only interesting and can be evaluated for utility if the items involved have some actual meaning. Overall, the results of tests on both sets of data justified DIC, and tests on the census data justified implication rules.

5.1 Test Data

The synthetic test data generator is well documented in [AS94] and it was very convenient to use. We used 100,000 transactions, with an average size of 20 items chosen from 1000 items, and average large itemsets were of size 4.

The census data was a bit more challenging. We chose to look at PUMS files which are Public Use Microdata Samples. They contain actual census entries which constitute a five percent sample of the state the file represents. In our tests we used the PUMS file for Washington D.C. which consists of roughly 30,000 entries. Each entry has 127 attributes, each of which is represented as a decimal number. For example, the SEX field uses one digit - 0 for male and 1 for female. The INCOME field uses six digits - the actual dollar value of that person's income for the year. We selected 73 of these attributes to study and took all possible (field,value) pairs. For the numerical attributes, like INCOME, we took the logarithm of the value and rounded it to the nearest integer to reduce the number of possible answers[7]. In total this yielded 2166 different items.

This data has several important differences from the synthetic data. First, it is considerably wider - 73 versus 20 items per transaction. Second, many of the items are extremely popular, such as *worked in 1989*. Third, the entire itemset structure is far more complex than that of synthetic data because correlations can be directional (for example *given birth to 2 children* $\Rightarrow$ *Female* but the reverse is not true), many rules are true 100% of the time (same example), and almost all attributes are correlated to some degree. These factors make mining census data considerably more difficult than supermarket style data.

5.2 Test Implementations

We implemented DIC and Apriori in C++ on several different Unix platforms. The implementations were mostly the same since Apriori can be thought of as a special case of DIC - the case where the interval size, was the size of the datafile, not M. Note that much work has been done on Apriori in recent years to optimize it and it was impossible for us to perform all of those optimizations. However, almost all of these optimizations are equally applicable to DIC.

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

[7] There has been much work done on bucketizing numerical parameters. However this is not the focus of our research so we took a simple approach.

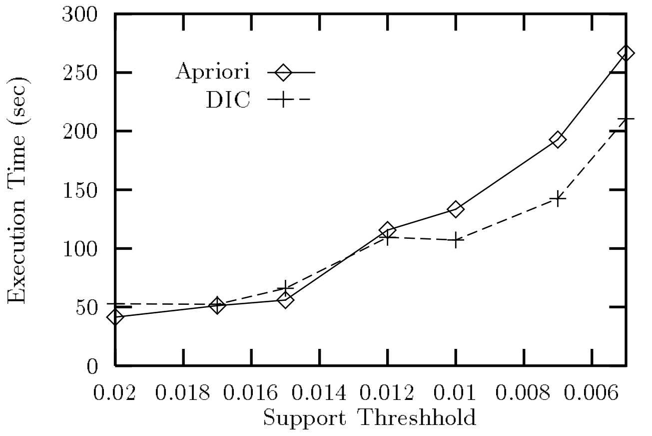

Figure 8: Performance of Apriori and DIC on Synthetic Data

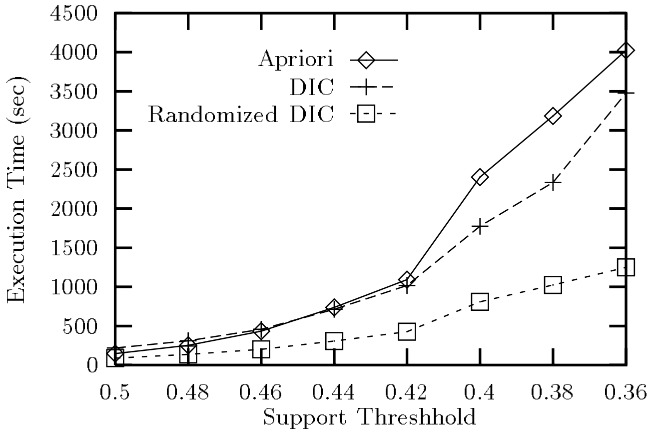

Figure 9: Performance of Apriori and DIC on Census Data

5.3 Relative Performance of DIC and Apriori

Both DIC and Apriori were run on the synthetic data and census data. Running both algorithms on the synthetic data was fairly straightforward. We tried a range of support values and produced large itemsets relatively easily (figure 8). Apriori beat out DIC by about 30% on the high support end of the graph but DIC outperformed Apriori in most tests, running 30% faster at a support threshold of 0.005. However, running both DIC and Apriori on census data was tricky. This is because a number of items in the census data appeared over 95% of the time and therefore there were a huge number of large itemsets. To address this problem, items with over 80% support were dropped. There were still a fair number of large itemsets but manageable at very high support thresholds. The reader will notice that the tests were run on support levels between 36% and 50% which are more than an order of magnitude higher than support levels used for supermarket analysis. Otherwise, far too many large itemsets would be generated. Even at the 36% support threshold, mining proved time consuming, taking nearly half an hour on just 30,000 records.

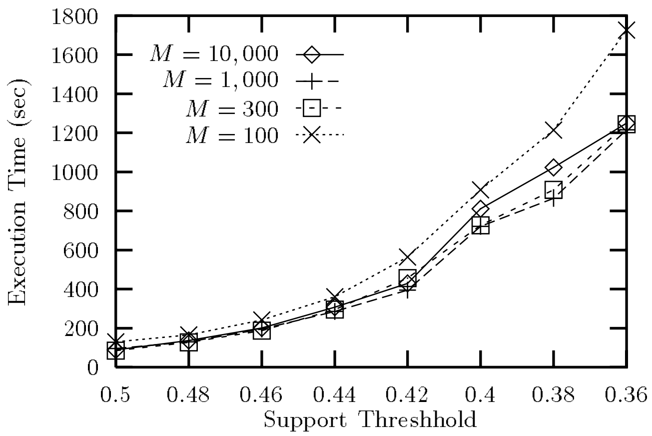

Figure 10: Effect of Varying Interval Size on Performance

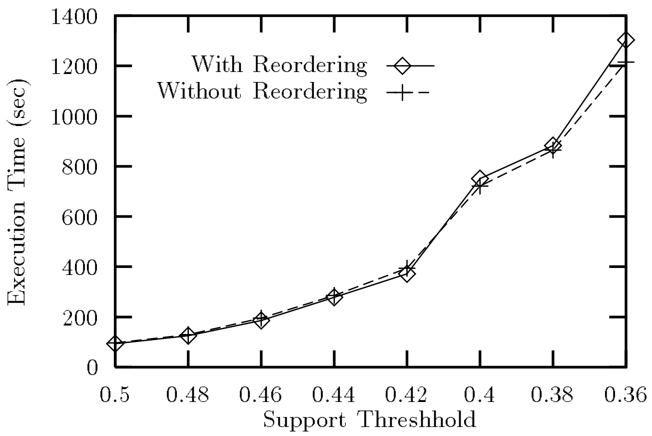

Figure 11: Performance With and Without Item Reordering

5.3.1 Performance on the Census Data

There are several reasons why mining the census data is so much more difficult than the synthetic data. The census data is 3.5 times wider than the synthetic data. So if we were counting all 2-itemsets, it would take 12 times longer per transaction (there are 12 times as many pairs in each row of the census data). If we were counting all 3-itemsets it would take 40 times longer; 4-itemsets would take 150 times longer. Of course we are not counting *all* the 2,3, or 4-itemsets; however, we are counting many of them, and we are counting higher cardinality itemsets as well. Furthermore, even after taking out the items which have more than 80% support, we are still left with many popular items, such as *works 40 hours/week*. These tend to combine to form many long itemsets.

The performance graphs show three curves (figure 9). One for Apriori, one for DIC, and one for DIC when we shuffled the order of the transactions beforehand (this has no effect on Apriori). For both tests with DIC, M was 10,000. The results clearly showed that DIC runs noticeably faster than Apriori and randomized DIC runs noticeably faster than DIC. For the support level of 0.36, randomized DIC ran 3.2 times faster than Apriori. By varying the value of M, we achieved slightly higher speedups - 3.3 times faster for support of 0.36 and 3.7 times faster for 0.38 (see next section).

5.4 Varying the Interval Size

One experiment we tried was to find the optimal value of M, the interval size (Figure 10). We tried values of 100, 300, 1000, and 10000 for M. The values in the middle of the range, 300 and 1000, worked the best, coming in second and first respectively. An interval size of 100 proved the worst choice due to too much overhead incurred. A value of 10,000 was somewhat slow because it took more passes over the data than for lower interval values.

We also tried varying M for non-randomized data. These experiments failed miserably and we were not able to complete them. In terms of the number of passes, at a support threshold of 0.36, Apriori made 10 passes over the data; simple DIC made 9 with an M of 10,000; randomized DIC made 4, 2.1, 1.3, and 1.3 passes for values of M of 10000, 1000, 300, and 100 respectively. This shows that DIC when combined with randomization and a sufficiently low M, does indeed finish in a very small number of passes.

5.5 Effect of Item Reordering

Item reordering was not nearly as successful as we had hoped. It made a small difference in some tests but overall played a negligible role in performance. In tests on census data it made less than 10% difference, sometimes in the wrong direction (Figure 11). This was something of a disappointment, but perhaps a better analysis of what the optimal order is and on-the-fly modification will yield better results.

5.6 Tests of Implication Rules

It is very difficult to quantify how well implication rules work. Due to the high support threshold, we considered rules based on the minimal small itemsets as well as the large itemsets. In total there were 23712 rules with conviction > 1.25 of which 6732 had a conviction of ∞. From these, we learned that five year olds don't work, unemployed residents don't earn income from work, men don't give birth, and many other interesting facts. Looking down the list to a conviction level of 50, we find that those who are not in the military, are not looking for work, and had work this year (1990, the year of the census), are currently employed as civilians. We list some sample rules in Table 3. Note that one problem was that many rules were very long (involving say seven items) and were too complicated to be interesting. Therefore, we list some of the shorter ones.

By comparison, tests with confidence produced some misleading results. For example, the confidence of *women who do not state whether they are looking for a job do not have personal care limitations* was 73% which is at the high end of the scale. However, it turned out that this was simply because 76% of all respondents do not have personal care limitations.

Interest also produced less useful results. For example, the interest of *male* and *never given birth* is 1.83 which is considerably lower than very many itemsets which we would consider less related and appears 40% of the way down in the list of rules with interest greater than 1.25.

conviction	implication rule
∞	five year olds don't work
∞	unemployed people don't earn income from work
∞	men don't give birth
50	people who are not in the military and are not looking for work and had work this year (1990, the year of the census) currently have civilian employment
10	people who are not in the military and who worked last week are not limited in their work by a disability
2.94	heads of household do not have personal care limitations
1.5	people not in school and without personal care limitations have worked this year
1.4	African-American women are not in the military
1.28	African-Americans reside in the same state they were born
1.28	unmarried people have moved in the past five years

Table 3: Sample Implication Rules From Census Data

6 Conclusions

6.1 Finding Large Itemsets

We found that the DIC algorithm, particularly when combined with randomization provided a significant performance boost for finding large itemsets. Item reordering did not work as well as we had hoped. However in some isolated earlier tests it seemed to make a big difference. We suspect that a different method for determining the item ordering might make this technique useful. Selecting the interval M made a big difference in performance and warrants more investigation. In particular, we may consider a varying interval depending on how many itemsets were added at the last checkpoint.

There are a number of possible extensions to DIC. Because of its dynamic nature, it is very flexible and can be adapted to parallel and incremental mining.

6.1.1 Parallelism

The most efficient known way to parallelize finding large itemsets has been to divide the database among the nodes and to have each node count all the itemsets for its own data segment. Two key performance issues are load balancing and synchronization. Using Apriori, it is necessary to wait after each pass to get the results from all nodes to determine what the new candidate sets are for the next pass. Since DIC can dynamically incorporate new itemsets to be added, it is not necessary to wait. Nodes can proceed to count the itemsets they suspect are candidates and make adjustments as they get more results from other nodes.

6.1.2 Incremental Updates

Handling incremental updates involves two things: detecting when a large itemset becomes small and detecting when a small itemset becomes large. It is the latter that is more difficult. If a small itemset becomes large, we may now have new potentially large itemsets (new prime implicants) which we must count over the entire data, not just the update. Therefore, when we determine that a new itemset must be counted, we must go back and count it over the prefix of the data that we missed. This is very much like the way DIC goes back to count the prefixes of itemsets it missed and it is straightforward to extend DIC in this way.

Another consideration is whether it is more useful to find the large itemsets over all the data or mine just the recent data (perhaps on the order of several months which may correspond to many small updates). Recall the train analogy (Section 1.1). The solution is to have two trains, one reading current data as it is coming in and incrementing appropriate counts, and the other reading several months old data and decrementing the appropriate counts to remove its effects. In order to do this, it is necessary to be able to add and remove itemset counters on the fly, quickly and efficiently, like DIC handles static data. This extension may be particularly useful.

6.1.3 Census Data

The census data was particularly challenging. Market basket analysis techniques had not been designed to deal well with this kind of data. It was difficult for several reasons (see section 5.1) - the data was very wide (more than 70 items per transaction), items were very varied in support (from very close to 0% to very close to 100%), and there was a lot to mine (many things were highly correlated). It was much more difficult to mine than supermarket data which is much more uniform in many ways. We believe that many other data sets are similarly challenging to mine and more work should be done toward handling them efficiently. It may be useful to develop some overall measures of difficulty for market-basket data sets.

6.2 Implication Rules

Looking over the implication rules generated on census data was educational. First, it was educational because most of the rules themselves were not. The rules that came out at the top, were things that were obvious. Perhaps the interesting things about the rules with a very high conviction value is why those that are very high are not ∞. For example, who are the seven people who earned over \$160,000 last year but are less than 500% over the poverty line?

The most interesting rules were found in the middle of the range, not extremely high as to be obvious (anything over 5) but not so low as to be insignificant (around 1.01). We believe that this is generally true of any data mining application. The extremely correlated effects are generally well known and obvious. The truly interesting ones are far less correlated.

A big problem was the number of rules that were generated - over 20,000. It is both impossible and unnecessary to deal with so many rules. We have considered several techniques for pruning them.

- First, one can prune rules which are not minimal. For example, if we have $A, B \Rightarrow C$ but $A \Rightarrow C$ then we may prune the first rule. This is somewhat nontrivial in that the longer rule may hold with a higher conviction value and therefore we may want to keep it. We have implemented pruning of all rules which have subsets with at least as high a conviction value. This has proven quite effective and and in tests it cuts down the number of rules generated by more than a factor of 5. The rules which are pruned are typically long and can be misleading. An example of a pruned rule is *an employed civilian who had work in 1989, is not looking for a job, is not on a leave of absence, is caucasian, and whose primary language is english, has worked this year.* It is implied by the rule *an employed civilian has worked this year.* This kind of pruning is very effective and can produce concise output for the user.
- Second, one can prune transitively implied rules. For rules that hold 100% of the time, if we have $A \Rightarrow B$ and $B \Rightarrow C$, one may want to prune $A \Rightarrow C$. However, there are several difficulties. First, which minimal set of rules should one pick? There can be many. And second, how should the rules which don't hold 100% of the time be handled? That is, how should the conviction value be expected to carry through?

Overall, conviction has proven to be a useful new measure having the benefits of being ∞ for perfect rules and 1 for completely uncorrelated rules. Moreover, it generally ranks rules in a reasonable and intuitive way. Unlike confidence, it does not assign high values to rules simply because the consequent is popular. Unlike interest, it is directional and is strongly affected by the direction of the arrow so that it can truly generate implication rules. However, as we found out, a good ranking system for rules is not enough by itself. More work needs to be done on rule pruning and filtering.

From our experiments we have learned that not all data which fits into the market-basket framework behaves nearly as well as supermarket data. We do not believe that this is because it is a wrong choice of framework. It is simply because doing this kind of analysis on data like census data is difficult. There are very many correlations and redundancies in census data. If we are aware beforehand of all of its idiosyncrasies, we can probably simplify the problem considerably (for example, by collapsing all the redundant attributes) and find a specialized solution for it. However, we want to build a general system, capable of detecting and reporting the interesting aspects of any data we may throw at it. Toward this goal, we developed DIC to make the task less painful to those of us who are impatient, and we developed *conviction* so that the results of mining market-basket data are more usable.

Acknowledgements

We would like to thank members of the Stanford Data Mining Group for helpful discussions. Also, we wish to thank Hitachi America's new Information Technology Group for supporting this work. Finally, we are grateful to Mema Roussopoulos for much help.

References

[AIS93a] R. Agrawal, T. Imilienski, and A. Swami. Database Mining: A Performance Perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993.

[AIS93b] R. Agrawal, T. Imilienski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 207–216, May 1993.

[ALSS95] R. Agrawal, K. Lin, S. Sawhney, and K. Shim. Fast similarity search in the presence of noise, scaling and translation in time-series databases. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, 1995.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th VLDB Conference, Santiago, Chile*, 1994.

[AS95] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conmference on Data Engineering, Taipei, Taiwan*, 1995.

[MAR96] M. Mehta, R. Agrawal, and J. Rissanen. Sliq: A fast scalable classifier for data mining. March 1996.

[SA95] R. Srikant and R. Agrawal. Mining generalized association rules. 1995.

[Toi96] H. Toivonen. Sampling large databases for association rules. *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, 1996.