Cyclic Association Rules

Banu Özden **Sridhar Ramaswamy** **Avi Silberschatz**

Bell Laboratories
Information Sciences Research Center
600 Mountain Avenue
Murray Hill, NJ 07974
{*ozden,sridhar,avi*}*@research.bell-labs.com*

Abstract

We study the problem of discovering association rules that display regular cyclic variation over time. For example, if we compute association rules over monthly sales data, we may observe seasonal variation where certain rules are true at approximately the same month each year. Similarly, association rules can also display regular hourly, daily, weekly, etc., variation that is cyclical in nature. We demonstrate that existing methods cannot be naively extended to solve this problem of *cyclic association rules*. We then present two new algorithms for discovering such rules. The first one, which we call the sequential algorithm, treats association rules and cycles more or less independently. By studying the interaction between association rules and time, we devise a new technique called *cycle pruning*, which reduces the amount of time needed to find cyclic association rules. The second algorithm, which we call the *interleaved* algorithm, uses cycle pruning and other optimization techniques for discovering cyclic association rules. We demonstrate the effectiveness of the interleaved algorithm through a series of experiments. These experiments show that the interleaved algorithm can yield significant performance benefits when compared to the sequential algorithm. Performance improvements range from 5% to several hundred percent.

1 Introduction

Recent advances in data collection and storage technology have made it possible for many companies to keep large amounts of data relating to their business online. At the same time, cheap computing power has also made some automatic analysis of this data feasible. This activity is commonly referred to as *data mining*.

One major application domain of data mining is in the analysis of transactional data. It is assumed that the database system records information about user transactions, where each transaction is a collection of items. In this setting, *association rules* capture interrelationships between various items. An association rule captures the notion of a set of items occurring together in transactions. For example, in a database maintained by a supermarket, an association rule might be of the form "beer $\rightarrow$ chips (support: 3%, confidence: 87%)", which means that 3% of all database transactions contain the items beer and chips, and 87% of the transactions that have the item "beer" also have the item "chips" in them. The two percentage parameters above are commonly referred to as "support" and "confidence" respectively.

Typically, the data mining process is controlled by a user who sets minimum thresholds for the support and confidence parameters. The user might also impose other restrictions, like restricting the search space of items, in order to guide the data mining process.

Following the work of [AIS93], discovery of association rules has been extensively studied in [AS94, SA95, HF95, SON95, PCY95, SA96, FMMT96, Toi96]. However, all the above work treat all the data as one large segment, with no attention paid to segmenting the data over different time intervals. To illustrate, let us return to our previous example. It may be the case that beer and chips are sold together primarily between 6PM and 9PM. Therefore, if we segment the data over the intervals 7AM–6PM and 6PM–9PM, we may find that the support for the beer and chips rule jumps to 50%.

From the above example we can conclude that although an association rule may have the user specified minimum confidence and support within the entire time spectrum, analysis of the data in finer time granularity may reveal that the association rule exists only in certain time intervals, and does not occur in the remaining time intervals. Even casual observation of many association rules over monthly data may disclose seasonal variation where peaks occur at approximately the same month in each year. Association rules could also display regular hourly, daily, weekly, etc., variation that has the appearance of cycles. It is also possible that an association rule does not have the user specified minimum confidence and/or support over the entire time spectrum, but its confidence and support are above the minimum threshold within certain time intervals. If these time intervals display a periodicity, discovering these rules and their periodicities may reveal interesting information that can be used for prediction and decision making.

Let us now examine another supermarket example where we are analyzing sales data over several months. Assume that we are interested in the selling patterns of the items coffee and doughnuts. If we were to look at the overall support for coffee and doughnuts occurring together in transactions, we might discover that the two items do not have enough support (i.e., these items do not meet the minimum threshold for support specified by the user). However, if we look into hourly sales figures, we might discover that coffee and doughnuts have a strong tendency to occur together with high support during the time interval 7AM–9AM! That is, "coffee $\rightarrow$ doughnuts" during the interval 7AM–9AM every day. Further analysis might also reveal a larger weekly tendency wherein coffee and doughnut sales dip during weekends.

Discovering such regularities in the behavior of association rules over time are the subject of this paper. We believe that our techniques will enable marketers to better identify trends in sales and allow for better forecasting of future demand.

We assume in this paper that the transactional data to be analyzed is time-stamped and that time intervals are specified by the user to divide the data into disjoint segments. We believe that users will typically opt for "natural" segmentations of the data based on months, weeks, days, etc., and that users are best qualified to make this decision based on their understanding of the underlying data. Addressing issues like overlapping and/or hierarchical segmentations, and automatically determining a good segmentation are outside the scope of this paper.

We refer to an association rule as *cyclic* if the rule has the minimum confidence and support at regular time intervals. Such a rule need not hold for the entire transactional database, but rather only for transactional data in a particular periodic time interval. That is, each cyclic rule must have the user specified minimum support and confidence over a specific periodic time interval. Upper and lower bounds for the periods of such time intervals will be specified by the user. We define the problem of *mining cyclic association rules* as the generation of all the cycles of association rules. Given a large database consisting of transactional information, where each transaction consists of a transaction-id, a set of items and a time-stamp, our goal is to provide efficient algorithms to discover cyclic association rules.

We first consider a relatively straightforward extension of existing association rule mining techniques for solving this problem. This extension treats association rules and cycles independently. It applies one of the existing methods for discovering association rules to each segment of data and then applies pattern matching algorithms to detect cycles in association rules. By studying the interaction between cycles and association rule mining more closely, we identify techniques called *cycle pruning* and *cycle skipping* which allow us to significantly reduce the amount of wasted work performed during the data mining process. We then show the effectiveness of these techniques by presenting the results of a series of experiments.

The remainder of this paper is organized as follows. In the next section, we define the problem of discovering cyclic association rules formally. In Section 3, we discuss the shortcomings of the existing techniques to discover cyclic association rules and present two new techniques to solve this problem. Implementation details of our prototype are described in Section 4. The experimental evaluation of the two techniques is presented in Section 5. Finally, we present our conclusions in Section 6 and identify directions for future research.

2 Problem Definition

Given a set of items and a set of transactions, where each transaction consists of a subset of the set of items, the problem of discovering association rules is defined as finding relationships between the occurrences of items within transactions. An *association rule* of the form $X \rightarrow Y$ is a relationship between the two disjoint itemsets X and Y (an itemset is a set of items). An association rule is described in terms of *support* and *confidence*. The support of an itemset X is the fraction of transactions that contain the itemset. An itemset is called *large*, if its supports exceeds a given threshold sup_{min}. The *confidence* of a rule $X \rightarrow Y$ is the fraction of transactions containing X that also contain Y. The association rule $X \rightarrow Y$ holds, if $X \cup Y$ is large and the confidence of the rule exceeds a given threshold con_{min}.

In order to deal with cyclic association rules, we enhance the transaction model by a time attribute that describes the time when the transaction was executed. In this paper, we assume that a unit of time is given (e.g., by the user). More complicated treatments of time, such as multiple units of time and/or a time hierarchy, are not in the scope of this paper. We denote the i^{th} time unit, $i \geq 0$, by t_i. That is, t_i corresponds to the time interval $[i \cdot t, (i+1) \cdot t)$, where t is the unit of time. We denote the set of transactions executed in t_i by $D[i]$.[1] We define the problem of discovering cyclic association rules as finding cyclic relationships between the presence of items within transactions.

The support of an itemset X in $D[i]$ is the fraction of transactions in $D[i]$ that contain the itemset, whereas the confidence of a rule $X \rightarrow Y$ in $D[i]$ is the fraction of transactions in $D[i]$ containing X that also contain Y. An association rule $X \rightarrow Y$ holds in time unit t_i, if the support of $X \cup Y$ in $D[i]$ exceeds sup_{min} and the confidence of $X \rightarrow Y$ in $D[i]$ exceeds con_{min}. A *cycle* c is a tuple (l, o) consisting of a length l (in multiples of the time unit) and an offset o (the first time unit in which the cycle occurs), $0 \leq o < l$. We say that an association rule has a cycle $c = (l, o)$ if the association rule holds in every l^{th} time unit starting with time unit t_o. For example, if the unit of time is an hour and "coffee $\rightarrow$ doughnuts" holds during the interval 7AM-8AM every day (i.e., every 24 hours), then "coffee $\rightarrow$ doughnuts" has a cycle $(24, 7)$. We denote the minimum and maximum cycle lengths of interest by l_{min} and l_{max}, respectively. We refer to an association rule that has a cycle as *cyclic*. An association rule may have multiple cycles. For example, if the unit of time is an hour and "coffee $\rightarrow$ doughnuts" holds during the interval 7AM-8AM and 4PM-5PM every day (i.e., every 24 hours), then "coffee $\rightarrow$ doughnuts" has two cycles: $c_1 = (24, 7)$ and $c_2 = (24, 16)$.

We say that a cycle (l_i, o_i) is a multiple of another cycle (l_j, o_j) if l_j divides l_i and $(o_j = o_i \bmod l_j)$ holds. By definition, once a cycle exists, all of its multiples with length less than or equal to l_{max} will exist. Therefore, it is only interesting to discover "large" cycles, where a large cycle is the one that is not multiple of any other cycle. Let $c = (l, o)$ be a cycle. A time unit t_i is said to be "part of cycle c" or "participate in cycle c" if $o = i \bmod l$ holds.

An association rule can be represented as a binary sequence where the ones correspond to the time units in which the rule holds and the zeros correspond to the time

[1] We will refer to $D[i]$ specifically as "time segment i" or generically as a "time segment."

units in which the rule does not have the minimum confidence or support. For instance, if the binary sequence 001100010101 represents the association rule $X \rightarrow Y$, then $X \rightarrow Y$ holds in $D[2]$, $D[3]$, $D[7]$, $D[9]$, and $D[11]$. In this sequence, $(4, 3)$ is a cycle since the corresponding rule holds within every fourth time unit starting from time unit t_3. A cycle can also be represented by a binary sequence. For example, cycle $(4, 3)$ can also be represented as 0001.

Similar to association rules, itemsets can also be represented as binary sequences where ones correspond to time units in which the corresponding itemset is large and zeros correspond to time units in which the corresponding itemset does not have the minimum support. Also, an itemset is said to be cyclic, if the itemset is large at regular intervals.

3 Discovering Cyclic Association Rules

The existing algorithms for discovering association rules cannot be applied directly for discovering cyclic association rules. In order to use the existing algorithms for detecting cyclic association rules, one may consider extending the set of items with time attributes, and then generating the rules. For example, one such rule could be $(day = monday) \cup X \rightarrow Y$. This approach segments the database in a manner where all transactions that have the same time attribute value are within the same segment. For example, if the time attribute is day, then all the transactions that occurred on Mondays will be within one segment. In this case, the support of the rule $(day = monday) \cup X \rightarrow Y$ is the ratio of the number of transactions that occurred on Mondays and contain X and Y to the total number of transactions. Similarly, the confidence of this rule is the ratio of the number of transactions that occurred on Mondays and contain X and Y to the number of transactions that occurred on Mondays and contain X. Note that such an approach yields different support and confidence definitions than the ones needed for detection of cyclic rules (see Section 2). Therefore, this approach will not solve our problem as exemplified below.

It is possible that this approach will detect non-existing cycles. For example, this approach may detect that every Monday $X \rightarrow Y$ holds, although $X \rightarrow Y$ holds only every second Monday, or only on some Mondays but not on all. This may occur, for example, when the support of $(day = monday) \cup X \cup Y$ exceeds sup_{min}, but the ratio of the number of transactions that occurred on some Mondays that contain both X and Y to the total number of transactions is below sup_{min}. That is, the ratio of the number of transactions that occurred on the remaining Mondays that contain both X and Y to the total number of transactions is high enough to compensate for the other Mondays. In this case, this approach assumes that the support for $(day = monday) \cup X \cup Y$ exceeds the minimum support threshold, although, in fact, the support of only $(day = only - some - mondays) \cup X \cup Y$ is above the minimum threshold. Therefore, a non-existing cycle—every Monday can be detected. Another problem with this approach is that it cannot detect cycles of arbitrary lengths. For example, it cannot detect an association rule that holds every 10 days.

3.1 The Sequential Algorithm

The straight-forward approach to discovering cyclic association rules is to generate the rules in each time unit with one of the existing methods [AS94, SON95] and then apply a pattern matching algorithm (See Section 3.4) to discover cycles. We refer to this approach as the *sequential* algorithm.

The existing algorithms discover the association rules in two steps. In the first step, large itemsets are generated. In the second step, association rules are generated from the large itemsets. The running time for generating large itemsets can be substantial, since calculating the supports of itemsets and detecting all the large itemsets for each time unit grows exponentially in the size of the large itemsets. To reduce the search space for the large itemsets, the existing algorithms exploit the following property:

"Any superset of a small itemset must also be small."

The existing algorithms calculate support for itemsets iteratively and they prune all the supersets of a small itemset during the consecutive iterations. Let us refer to this pruning technique as *support-pruning*. In general, these algorithms execute a variant of the following steps in the k^{th} iteration:

1. The set of candidate $k-$itemsets is generated by extending the large $(k-1)-$itemsets discovered in the previous iteration (support-pruning).

2. Supports for the candidate $k-$itemsets are determined by scanning the database.

3. The candidate $k-$itemsets that do not have minimum support are discarded and the remaining ones constitute the large $k-$itemsets.

The idea is to discard most of the small $k-$itemsets during the support-pruning step so that the database is searched only for a small set of candidates for large $k-$itemsets.

In the second step, the rules that exceed the confidence threshold con_{min} are constructed from the large itemsets generated in the first step with one of the existing algorithms. For our experimental evaluation of the sequential algorithm, we implemented the apriori and the ap-genrules algorithms from [AS94]. Once the rules of all the time units have been discovered, cycles need to be detected. Let r be the number of rules detected. The complexity of the cycle detection phase has an upper bound of $O(r \cdot n \cdot l_{max})$, where n is the number of time units and l_{max} is the maximum cycle length of interest (see Section 3.4). If all the rules in each time unit fit into main memory, then the running time of the cycle detection phase is feasible. However, if the rules in all the time units do not fit into main memory, then the overhead of I/O operations substantially increases the running time of the cycle detection phase (see Section 5), and therefore the sequential algorithm may become infeasible for detecting cyclic association rules. The issue of memory management for cycle detection will be further discussed in Section 4.

3.2 Cycle-Pruning, Cycle-Skipping and Cycle-Elimination

The major portion of the running time of the sequential algorithm is spent to calculate the support for itemsets. We now present three techniques—*cycle-pruning*, *cycle-skipping*, and *cycle-elimination* to prune the number of itemsets for which the support must be calculated. These techniques rely on the following fact:

"A cycle of the rule $X \rightarrow Y$ is a multiple of a cycle of itemset $X \cup Y$."

Therefore, eliminating cycles as early as possible can substantially reduce the running time of cyclic association rule detection. Cycle-skipping is a technique for avoiding counting the support of an itemset in time units, which we know, cannot be part of a cycle of the itemset. Cycle-skipping is based on the following property:

" If time unit t_i is not part of a cycle of an itemset X, then there is no need to calculate the support for X in time segment $D[i]$."

However, cycle skipping is useful only if we have information about the cycles of an itemset X. But the cycles of an itemset X can be computed exactly only after we compute the support of X in all the time segments! In order to avoid this self-dependency, we try to approximate the cycles of itemsets. To do this, we will use a technique we call *cycle pruning*. It is based on the following property:

"If an itemset X has a cycle (l, o), then any of the subsets of X has the cycle (l, o)."

The above property implies that any cycle of itemset X must be a multiple of a cycle of an itemset that is a subset of X. This also implies that the number of cycles of an itemset X is less than or equal to the number of cycles of any of X's subset.

Therefore, one can arrive at an upper bound on the cycles that an itemset X can have by looking at all the cycles of the subsets of X. By doing so, we can reduce the number of potential cycles of itemset X, which, in turn (due to cycle-skipping), reduces the number of time units in which we need to calculate support for X. Thus, cycle-pruning is a technique for computing the candidate cycles of an itemset by merging the cycles of the itemset's subsets.

However, it is possible in some cases that we cannot compute the candidate cycles of an itemset. For example, when we are dealing with singleton itemsets. In these cases, we need to assume that an itemset X has every possible cycle and therefore, calculate the support for X in each time segment $D[i]$ (except the time units eliminated via support-pruning). This is, in fact, what the sequential algorithm does.

Example 1: If we know that 010 is the only cycle of item A, and 010 is also the only cycle of item B, then cycle-pruning implies that the itemset consisting of items A and B can have only the cycle 010 or its multiples. Cycle-skipping suggests that we do not need to calculate the support for $A \cup B$ in every time segment but only in every third one starting with $D[1]$. □

Example 2: If we know that 010 is the only cycle of item A and 001 is the only cycle of item B, then cycle-pruning implies that the itemset $A \cup B$ cannot have any cycles. Cycle-skipping suggests that we do not need to calculate the support for $A \cup B$ in any of the time segment. □

We now introduce one more optimization technique we call *cycle-elimination* that can be used to further reduce the number of potential cycles of an itemset X. Cycle-elimination is used to eliminate certain cycles from further consideration once we have determined they cannot exist. Cycle-elimination relies on the following property:

"If the support for an itemset X is below the minimum support threshold sup_{min} in time segment $D[i]$, then X cannot have any of the cycles $(j, i \bmod j)$, $l_{min} \leq j \leq l_{max}$."

Cycle-elimination enables us to discard cycles that an itemset X cannot have as soon as possible as demonstrated in the following example.

Example 3: If the maximum cycle length we are interested is l_{max} and the support for itemset A is below the threshold sup_{min} in the first l_{max} time units, then cycle-elimination implies that A cannot have any cycles. Cycle-skipping suggests that there is no need to calculate the support for A in time units greater than l_{max}. □

3.3 The Interleaved Algorithm

We now present another algorithm, which we refer to as the *interleaved algorithm*, for discovering cyclic association rules. The interleaved algorithm consists of two phases. In the first phase, the cyclic large itemsets are discovered. In the second phase, cyclic association rules are generated.

In the first phase, the search space for the large itemsets is reduced using cycle-pruning, cycle-skipping and cycle-elimination as follows. For each k, $k \geq 1$:

1. If $k = 1$, then all possible cycles are initially assumed to exist for each single itemset. Otherwise (if $k > 1$), cycle-pruning is applied to generate the potential cycles for $k-$itemsets using the cycles for $(k-1)-$itemsets.
2. Time segments are processed sequentially. For each time unit t_i :
   - 2.1 Cycle-skipping determines, from the set of candidate cycles for $k-$itemsets, the set of $k-$itemsets for which support will be calculated in time segment $D[i]$.
   - 2.2 If a $k-$itemset X chosen in Step 2.1 does not have the minimum support in time segment $D[i]$, then cycle-elimination is used to discard each cycle $c = (l, o)$, for which $(o = i \bmod l)$ holds, from the set of potential cycles of X.

This process terminates when the list of potential cycles for each $k-$itemset is empty. Cycle-pruning, cycle-skipping and cycle-elimination can reduce the candidate $k-$itemsets for which support will be counted in the database substantially, and therefore can reduce the running time of calculating the large itemsets. This is demonstrated by the following example.

Example 4: Suppose that the length of the longest cycle we are interested in is $l_{max} = 6$ and 11100000000111111111 and 11111010111111111111 represent items A and B, respectively. If the sequential algorithm is used, then the support for A and B will be calculated in all the time segments and the support for $A \cup B$ will be calculated in time segments 0-2, 10-19 (due to support-pruning). If the interleaved algorithm is used, then the support for A will be calculated in time segments 0-9 (due to cycle-elimination and cycle-skipping), whereas the support for B will be calculated in all the time segments, and since A has no cycles, $A \cup B$ cannot have any cycle (due to cycle-pruning), and the support for $A \cup B$ will not be calculated in any of the time units (due to cycle-skipping). □

In the second phase of the interleaved algorithm, the cyclic association rules can be calculated using the cycles and the support of the itemsets without scanning the database. Interleaving cycle detection with large itemset detection also reduces the overhead of rule generation phase. This is because a cycle of the rule $X \rightarrow Y$ must be a multiple of a cycle of itemset $X \cup Y$, and at the end of the first phase of the interleaved algorithm we already know the cycles of large itemsets. Thus, the set of candidate cycles for a rule $X \rightarrow Y$ initially consists of the set of cycles of the itemset $X \cup Y$. As a result, we need to calculate the confidence of a rule $X \rightarrow Y$ only for time units that are part of cycles of $X \cup Y$. Moreover, whenever we encounter a time unit t_i in which this rule does not have minimum confidende con_{min}, we can eliminate each other candidate cycle of this rule for which $(j, i \bmod j)$, $l_{min} \leq j \leq l_{max}$ holds. Once the cycles of the association rules are detected, the cycles that are not large can be eliminated (see Section 3.4).

Although the interleaved algorithm reduces the CPU overhead of calculating support for itemsets substantially, it incurs a higher I/O overhead when compared to the sequential algorithm. In order to remedy this problem, each time when a time segment $D[i]$ is accessed, calculation of cycles and support for k, $(k+1)$, ..., and $(k+g)$-itemsets, for some $g \geq 1$, can be combined. In this case, cycle-pruning for each $(k+j)-$itemset, $0 \leq j \leq g$, is done by merging the cycles of $(k-1)$ subsets of the $(k+j)-$itemset. That is, the set of candidate cycles of a $(k+j)-$itemset consists of the intersection of the sets of cycles of all $(k-1)$ subsets of the itemset. While scanning a time segment $D[i]$, first supports for the $k-$itemsets that have candidate cycles into which time unit t_i participates are calculated. If the support for a candidate $k-$itemset X is below sup_{min}, then cycle-elimination is applied not only to the candidate cycles of itemset X, but also to each $(k+j)-$itemset, $0 \leq j \leq g$, that is a superset of X. Note that cycle elimination eliminates potentially more $(k+j)-$itemsets compared to support-pruning. This is because support-pruning eliminates $(k+j)-$itemsets that are supersets of X only in $D[i]$ whereas cycle-elimination eliminates $(k+j)-$itemsets that are supersets of X not only in $D[i]$ but potentially also in other time segments following $D[i]$. Once supports for the candidate $k-$itemsets in $D[i]$ are calculated, then supports for the candidate $(k+1)-$itemsets in $D[i]$ are calculated similarly, followed by supports for the candidate $(k+2)-$itemsets in $D[i]$, and so on. The determination of the optimal value of g is beyond the scope of this paper.

3.4 Cycle Detection

Given a binary sequence of length n and the maximum cycle length of interest l_{max}, the running time of detecting all cycles with lengths less than or equal to l_{max} of the binary sequence has an upper bound of $O(l_{max} \cdot n)$ operations. We now present a straight-forward approach to detecting cycles. This approach is composed of two steps. Initially, the set of candidate cycles contains all possible cycles. In the first step, the sequence is scanned, and each time a zero is encountered at a sequence position i, candidate cycles $(j, i \bmod j)$, $1 \leq j \leq l_{max}$ are eliminated from the set of candidate cycles. The first step completes whenever the last bit of the sequence is scanned or the set of candidate cycles becomes empty, which ever is first. In the second step, large cycles (i.e., cycles that are not multiples of any existing cycles are detected). A straight-forward approach to eliminating cycles that are not large is as follows. Starting from the shortest cycle, for each cycle $c_i = (l_i, o_i)$, eliminate each other cycle $c_j = (l_j, o_j)$ from the set of cycles, if l_j is a multiple of l_i and $(o_i = o_j \bmod l_i)$ holds. The sequential algorithm uses this approach to detect the cycles.

However, if we knew initially or at any time while we are scanning the sequence that some of the cycles cannot exist in the sequence (e.g., due to cycle-pruning or cycle-elimination), cycle detection procedure can be optimized by skipping sequence positions that cannot be part of any of the candidate cycles. That is, instead of scanning the sequence sequentially, we can skip the bit positions that are not part of the candidate cycles (i.e., cycle-skipping). For example, if l_{max} is three, and we know initially that 01, 010 and 001 cannot be cycles of a given sequence (e.g., due to cycle-pruning), then we do not need scan bit positions 1, 5, 7, 11, etc. Also, while we are scanning the sequence, if we also eliminate candidate cycle 100 (i.e., cycle-elimination), we can skip scanning every second bit of the sequence starting at that point. The interleaved algorithm employees these optimization techniques (cycle-pruning, cycle-elimination and cycle-skipping) that reduce the overhead of cycle detection, but more importantly, the overhead of calculating support for itemsets, since the interleaved algorithm "interleaves" both cycle detection and support calculation for itemsets.

The cycle detection process can be further optimized by considering a candidate cycle $c_i = (l_i, o_i)$ only when there is no other candidate cycle $c_j = (l_j, o_j)$ remaining such that c_i is a multiple of c_j. Further optimizations of the cycle detection is not in the scope of this paper. However, given a sequence of length n, and the maximum cycle length of interest l_{max}, the complexity of finding all the cycles of the sequence is an open problem.

4 Implementation Details

In this section, we present the implementation details of the prototype that we built for discovering cyclic association rules. We first describe the synthetic data generator used to generate our data.

4.1 Data Generation

Our data generator is based on the synthetic data generator used in [AS94]. We augmented it to generate data for cyclic

D	Number of transactions/time segment
T	Avg. size of transactions
I	Avg. size of the maximal potentially large itemsets
L	Number of maximal potential large itemsets
N	Number of items

Table 1: Parameters for data generation from[AS94]

u	Number of time units of data generated
p_{num}	Avg. number of patterns associated with each large itemset
p_{min}	Minimum length of pattern generated
p_{max}	Maximum length of pattern generated
p_{den}	Avg. "density" of patterns generated
ν	Avg. level of "noise" in the data generated

Table 2: New parameters for cyclic association rule generation

association rules. In addition to the parameters used by [AS94] shown in Table 1, we used additional parameters shown in Table 2 (The parameters are described in the following paragraphs).

The generation of the large itemsets and their weights closely follows the procedure in [AS94]. We generate L itemsets of average size I. Each itemset is associated with a weight, which is an exponentially distributed random variable. We define a *pattern* to be the union of a set of cycles of the same length. For example, pattern 1001 represents the union of cycles $(4, 0)$ and $(4, 3)$. In order to model the fact that certain rules might occur cyclically, we associate p_{num} patterns with each large itemset that we generate. The length of the patterns is uniformly distributed between p_{min} and p_{max} time units. The density parameter p_{den}, which is a real number between 0 and 1, controls the number of cycles in a pattern (The number of cycles a pattern of length p_{len} contains is equal to $p_{\text{len}} \times p_{den}$ on the average).

In order to model the fact that real world data will consist of a mixture of cyclic rules and non-cyclic rules, we use the "noise" parameter ν, which is a real number between 0 and 1. In a particular time unit, a large itemset is "active" (in the sense that transactions will contain that itemset) independent of cycles with a probability ν.

At the beginning of each time unit, the data generation algorithm first determines which large itemsets should be used for data generation in that time unit. This is done by checking to see if the current time t participates in any of the cycles that the itemset is associated with. Following this, a determination is made as to whether the noise parameter dictates that the itemset be used. Once this is done, the weights associated with the large itemsets determine their occurrences in the transactions for the time unit.

The default values we used for the parameters in our experiments are shown in Table 3. (We conducted individual sets of experiments that varied these parameters. We describe the variations when we describe the individual experiments.)

Number of transactions/time segment, D	10000
Number of items, N	1000
Avg. size of large itemsets, I	4
Number of large itemsets, L	1000
Avg. transaction size, T	10
Number of time units, u	600
Avg. number of patterns, p_{num}	2
Min. length of pattern, p_{min}	10
Maximum length of pattern, p_{max}	100
Avg. "density" of patterns, p_{den}	0.2
Avg. level of "noise" in the data generated, ν	0.3

Table 3: Default settings for parameters in data generation.

When the parameters are set to the above default values, the size of the data generated is about 155 megabytes (MB) for all the time units combined.

4.2 Prototype Implementation Details

We use the apriori algorithm from [AS94] as our basic data mining algorithm. The sequential algorithm is based directly on apriori, with optimizations to speed up the counting of support of itemsets of size 2. We use an array for this instead of a hash-tree when memory permits. We found the array to be a much faster technique for discovering 2-itemsets.

The interleaved algorithm uses a hash-tree, described in [AS94], to store the large itemsets, their patterns and support counts. In addition, during the processing of an individual time segment, the interleaved algorithm uses a temporary hash-tree as well. Candidate generation (generation of itemsets of size $k + 1$ and their candidate cycles from itemsets of size k) is based on cycle pruning. Figure 1 outlines the first phase of the *interleaved* algorithm (cyclic large itemset detection) in pseudo-code. After that, we apply a generalization of the rule generation procedure in [AS94] for cyclic association rule generation.

We conducted our experiments on a lightly loaded Sun Sparc 20 machine with 64 MB of memory running Solaris 2.5.1. A Seagate 9 GB SCSI disk was used for our experiments. The hard disk had a streaming read throughput of about 5 megabytes/sec (MBps) and a streaming write throughput of about 4 MBps. Since our experiments were CPU bound most of the time, we only report wall clock times for the various experiments.

4.3 Memory Management

The sequential algorithm runs the apriori algorithm on each time segment. In order to determine the cycles of the association rules, we need to determine the binary sequence corresponding to each association rule. If there is space in memory to hold a binary sequence for each association rule, we can store the rules and their binary sequences in a hash tree. After generating all the rules over all the time segments, we can run cycle detection algorithms.

However, if we do not have enough memory to store the rules and their binary sequences, we have to write the rules out to disk as we generate them in each time unit. At the end of finding association rules for all the time segments, we need to construct the binary sequences for each individual

```
/* This algorithm uses two hash-trees. itemset-hash-tree contains candidates of size $k$, their potential cycles, and space to store support counts for the relevant time units. An "active" itemset at time unit $t$ is an itemset that has a cycle that $t$ participates in. tmp-hash-tree, during the processing of time segment $t$, contains all the itemsets that are active in $t$. */

Initially, itemset-hash-tree contains singleton itemsets
   and all possible cycles
$k = 1$
while (there are still candidates in itemset-hash-tree
with potential cycles)
   for $t$ = 0 to $n - 1$
      insert active itemsets from itemset-hash-tree into
         tmp-hash-tree // cycle skipping
      measure support in current time segment for each
         itemset in tmp-hash-tree
      forall $l \in$ tmp-hash-tree
         if ($sup_l < sup_{min}$)
         then delete corresponding cycles of
            itemset $l$ // cycle elimination
         else insert ($l$, sup$_l$, $t$) into itemset-hash-tree
         // this just inserts a ($sup_l$, $time$) entry in one of
            itemset $l$'s fields
      end forall
      empty tmp-hash-tree
      endfor
      verify actual cycles of each member of
         itemset-hash-tree
      generate new candidates of size $k + 1$ using
         cycle pruning
      $k = k + 1$
      empty itemset-hash-tree after copying it to disk
      insert new candidates into itemset-hash-tree
   endwhile
```

Figure 1: The *interleaved* algorithm for cyclic large itemset detection.

association rule. In order to do this, we merge the rules from the different time segments. One this merging is done, we can run the cycle detection algorithms.

Finally, if we do not have enough memory to store all the data structures needed by the apriori algorithm, we have to use one of the overflow management techniques suggested in [AS94].

Like the apriori algorithm, the interleaved algorithm has two distinct phases. In the first phase, all large itemsets with cycles have their supports counted in the appropriate time segments. In the second phase, rules are generated using the cycle and support information of the large itemsets.

For the first phase, the interleaved algorithm proceeds "level-by-level" to determine itemset support. It first determines the itemset support for singleton candidate itemsets, generates cycles for them, then generates itemsets of size 2 and their potential cycles, etc. In this phase, the interleaved algorithm requires enough memory to hold all large itemsets of a particular size and their support counts in memory. (In addition, it needs to store the new candidates and their "potential" cycles. The size of the latter is usually much smaller.) If there is not enough memory, the support counts are broken up into chunks and written to disk. After processing all the time segments, the support counts are merged in memory.

For the cyclic rule generation phase, if there is space in memory to hold all the large itemsets and their support counts for all the time units, rule generation can run entirely in memory. However, if there is a shortage of memory and there is space to hold only a single level of large itemsets and their support counts, we can generate rules in a level-by-level fashion as well starting at the level of the largest itemset. For doing this, we can use a modification of the *ap-genrules* procedure in [AS94] that we call *Level_GenRuleCycles* as shown in Figure 2. *Level_GenRuleCycles* is a set oriented rule generation procedure that creates rules such that all rules needing to look at the support counts of k-itemsets are generated during during one iteration of the outer while loop. (Note that this procedure can be profitably used, instead of *ap-genrules*, for generating association rules when short of memory.)

```
Procedure Level_GenRuleCycles()
level = size of the largest itemset
// ruleList is a list of records that have four fields.
// The first three fields contain an itemset name, the
// support array for the itemset, and the list of cycles
// for the itemset. The fourth field is a list of candidate
// rules generated from the itemset that are known to
// have cycles.
ruleList = $\{\}$ // "ruleList" is the current list of
   quintuplets used for generating rules.
while (level $\geq 1$)
   read in support counts and cycles of large itemsets
      of size level
   newRuleList = $\{\}$
   forall lItem $\in$ ruleList do
      if (lItem's itemset is being used to generate rules
         for the first time)
      then generate singleton rules
      else generate candidate rules using apriori-gen
         on lItem's current rules
      verify cycles for each rule generated and discard
         rules without cycles.
      lItemNew= lItem with old rules replaced by the
         new rules
      newRuleList = newRuleList $+$ lItemNew
   endforall
   ruleList = newRuleList $+$ quintuplets created from
      large itemsets at current level
   level$--$
endwhile
```

Figure 2: Procedure used for cyclic rule generation.

For example, suppose we have the large itemset $ABCD$ with a cycle 001. Initially, *ruleList* is the empty set and in the first iteration gets set to $\{ABCD$, sup-array$_{\text{ABCD}}$,

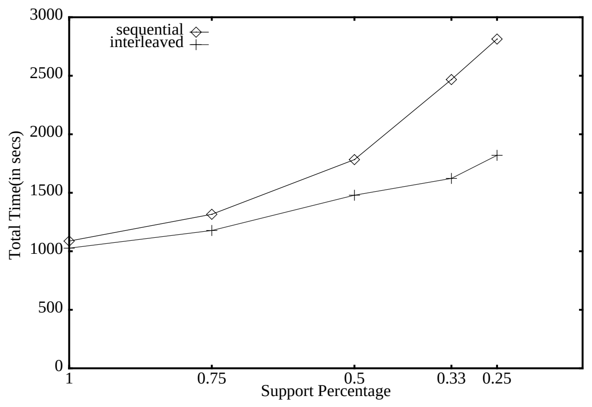

Figure 3: Execution time vs. support for the two algorithms

{001}, {}}. In the next round, $ABC \rightarrow D$, $ABD \rightarrow C$, $ACD \rightarrow B$ and $BCD \rightarrow A$ are generated and tested. This round keeps the support counts of all 3-itemsets in memory. Suppose only $ABC \rightarrow D$ and $ABD \rightarrow C$ have cycles. *ruleList* becomes $\{\{ABCD$, sup-array$_{\text{ABCD}}$, {001}, $\{ABC \rightarrow D$, $ABD \rightarrow C\}\}\}$. (Of course, large itemsets ABC and ABD will get added to *prev*, but we ignore that here to keep the example small.) In the next round, only the rule $AB \rightarrow CD$ is generated. This round keeps the support counts of all the 2-itemsets in memory. If $AB \rightarrow CD$ has a cycle, *ruleList* is transformed to $\{\{ABCD$, sup-array$_{\text{ABCD}}$, {001}, $\{AB \rightarrow CD\}\}\}$ and vanishes in the next round.

This algorithm requires only one member of *ruleList* to be in memory at any time. If the support counts of a particular level do not fit into memory as well, one has to sort *ruleList* according to the candidate rules that it stores and merge the itemset support counts in order to generate cyclic association rules.

5 Experimental Results

We now present the results of an extensive set of experiments conducted to analyze the behavior of the sequential and interleaved algorithms. In these experiments, it should be kept in mind that the sequential algorithm had an inherent advantage in considering the time segments of the data one by one. All but the largest time segments that we used fit entirely in the main memory of the machine that we used. The interleaved algorithm which sweeps repeatedly through the entire data incurs more I/O's.

5.1 Dependence on Minimum Support

Figure 3 plots the execution time for the interleaved and sequential algorithms as support is varied from 1% to 0.25%. With support set to 1%, the interleaved algorithm is only about 5% faster than the sequential algorithm. This is because the number of large itemsets (both cyclic and otherwise) at this level of support is fairly small and the interleaved algorithm does not incur a significant benefit from its pruning techniques. As support decreases, the amount of wasted work done by the sequential algorithm increases significantly as the number of itemsets found to be large by the sequential algorithm increases. (Many of these large itemsets and the rules derived from them later turn out to not contribute to useful cyclic association rules.) The interleaved algorithm, from early on, concentrates only on

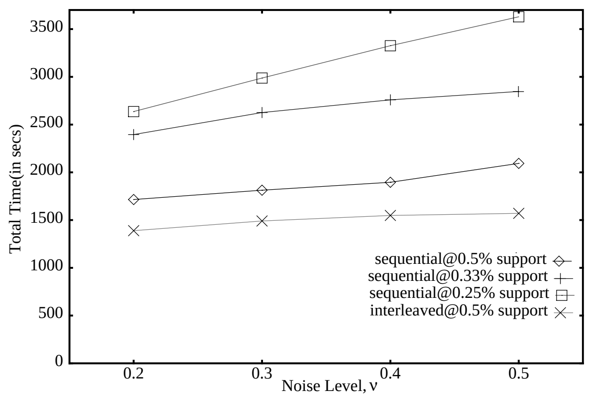

Figure 4: Execution time vs. noise level for the two algorithms at varying levels of support. For the interleaved algorithm, only the running times at support $= 0.5$% are shown.

large itemsets that can contribute to cycles and benefits from this strategy. The result is that the running time of the sequential algorithm is more than 50% higher than that of the interleaved algorithm, when support $= 0.25$%.

5.2 Varying Noise Levels

Figure 4 shows the running times for the interleaved and sequential algorithms for various noise levels. (Since the interleaved algorithm is relatively unaffected by noise, we only show its running times for support $= 0.5$% to keep the graph more readable.)

Noise influences the data generated in two ways. One, the number of itemsets that do not lead to cycles goes up as the noise level increases. Two, as the amount of noise in the data increases, it leads to spurious cycles, where an itemset is used for data generation in a cyclic fashion accidentally as dictated by the noise parameter.

As can be seen from the graph, the interleaved algorithm is relatively unaffected by noise. (The curves are quite horizontal for supports of 0.33% and 0.25% also.) However, the sequential algorithm shows more interesting behavior. The graphs of the running time increase more or less linearly with noise, but the slope of the graphs is different at different support levels. The explanation for this lies in the number of new itemsets added by noise. At high support levels, noise does not add many itemsets. (If the transaction size is T, large itemset size is I, there can only be about $T/(I \times sup)$ large itemsets with support sup. For example, if $T = 8$, $I = 4$, $sup = 0.5$%, there can be no more than 400 large itemsets with support 0.5% in the data.) Therefore, for high support levels, the slope of the sequential algorithm rises slowly. At low support levels, noise tends to have a much more dramatic impact on the amount of wasted work performed by the sequential algorithm and its running time rises with a much steeper slope.

5.3 Varying Itemset Size

Figure 5 compares the running time of the two algorithms when the maximum size of the large itemsets is varied from 3 to 8. For itemset sizes 3 and 4, the interleaved algorithm is faster, but only by about 7% and 20% respectively. However, as the largest itemset size increases, the amount

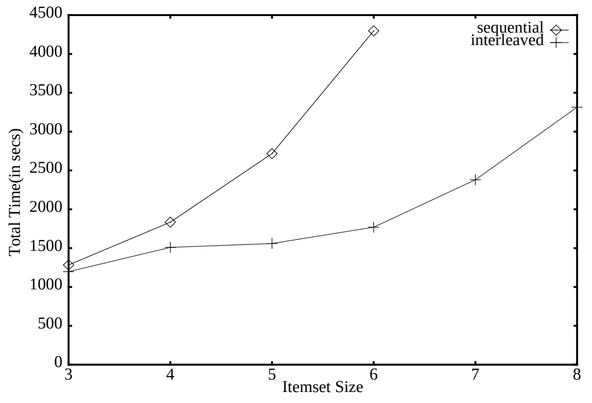

Figure 5: Execution time vs. large itemset size for the two algorithms

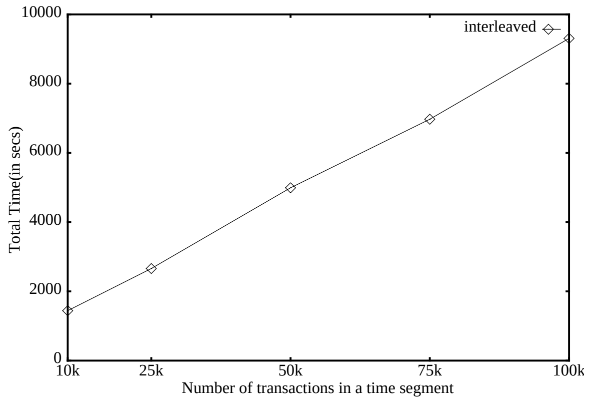

Figure 6: Execution time for the interleaved algorithm as the data size for a single time segment increases from $10k$ transactions to $100k$ transactions. (This corresponds to increasing the total database size from 155 MB to 1.5 GB.)

of work performed by the sequential algorithm increases dramatically. This is because each large itemset of size l that does not contribute to a cycle forces the sequential algorithm to calculate support counts for all its 2^l subsets without producing any useful results. Further, the association rules computed by the sequential algorithm do not fit in memory and it starts incurring significant I/O overhead as well contributing to the large jump in execution time as we go from itemset size 5 to 6. In fact, we had to terminate the experiments with the sequential algorithm for itemset sizes 7 and 8 because they ran for over a day each without completing. This experiment conclusively demonstrated the superiority of the interleaved algorithm over the sequential algorithm.

5.4 Data Size Scaleup

Figure 6 shows the running time of the interleaved algorithm as the time segment size is increased from $10k$ transactions to $100k$ transactions. (The database size increased from 155 MB to 1.5 GB.) The interleaved algorithm shows nearly linear scaleup. This illustrates its ability to handle large database sizes gracefully.

5.5 Experimental Conclusions

Through a series of experiments, we have shown that the interleaved algorithm is significantly better than the sequential algorithm. The interleaved algorithm performs at least as well, and often times, significantly better than the sequential algorithm. Performance benefits range from 5%, when support is very high, to several hundred percent, when large itemset sizes are over 5. (Much of this overhead comes from the increased I/O costs that the sequential algorithm incurs.) Further, the interleaved algorithm scales nicely with increasing data. Thus, one can conclusively say that the interleaved algorithm and the pruning ideas behind it provide significant gains in performance.

6 Conclusions and Future Directions

In this paper, we have studied the problem of discovering association rules that display regular cyclic variation over time. Information about such variations will allow marketers to better identify trends in association rules and help better forecasting. By exploiting the relationship between cycles and large itemsets, we identified optimization techniques that allow us to minimize the amount of wasted work performed during the data mining process. We demonstrate the usefulness of these techniques through an extensive experimental study. The study showed that performance benefits ranging from 5% to several hundred percent can be obtained through the use of the optimizations when compared to the more straightforward approach.

Recently, we have explored, in [RMS97], the integration of calendars into an association rule mining framework. This lets us consider rules like, "$X \rightarrow Y$, the first working day of every month." In [RMS97], we also address the problem of approximate matching, which relaxes the rather rigid definition of when a cycle belongs to an association rule. Interesting open problems include the handling of multiple time units (like hours, weeks, days, etc.) simultaneously and efficiently, integrating cycles and calendars into the *numeric*, rather than the boolean domain handled by the association rules in this paper.

Acknowledgements: We would like to thank the anonymous referees for their helpful comments on the paper.

References

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. "Mining Association Rules between Sets of Items in Large Databases". In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207—216, Washington,DC, May 1993.

[AS94] R. Agrawal and R. Srikant. "Fast Algorithms for Mining Association Rules in Large Databases". In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, Santiago, Chile, September 1994.

[FMMT96] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. "Data Mining Using Two-Dimensional Optimized Association Rules". In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 13–23, Montreal, Canada, June 1996.

[HF95] J. Han and Y. Fu. “Discovery of Multi-level Association Rules From Large Databases”. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 420–431, Zurich, Switzerland, September 1995.

[PCY95] J. S. Park, M. Chen, and P. Yu. “An Effective Hash-based Algorithm for Mining Association Rules”. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 175–186, May 1995.

[RMS97] S. Ramaswamy, S. Mahajan, and A. Silberschatz. Calendric association rules. Technical report, Bell Labs, 1997. Submitted for publication. Available upon request.

[SA95] R. Srikant and R. Agrawal. “Mining Generalized Association Rules”. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 407–419, Zurich, Swizerland, September 1995.

[SA96] R. Srikant and R. Agrawal. “Mining Quantitative Association Rules”. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 1–12, Montreal, Canada, June 1996.

[SON95] A. Savasere, E. Omiecinski, and S. Navathe. “An Efficient Algorithm for Mining Association Rules in Large Databases”. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 432–444, Zurich, Swizerland, September 1995.

[Toi96] H. Toivonen. “ Sampling Large Databases for Association Rules”. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, Bombay, India, September 1996.