An Efficient Approach to Discovering Knowledge from Large Databases*

Show-Jane Yen and Arbee L.P. Chen

Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 300, R.O.C.
Email: alpchen@cs.nthu.edu.tw

Abstract

In this paper, we study two problems: mining association rules and mining sequential patterns in a large database of customer transactions. The problem of mining association rules focuses on discovering *large itemsets* where a large itemset is a group of items which appear together in a sufficient number of transactions; while the problem of mining sequential patterns focuses on discovering *large sequences* where a large sequence is an ordered list of sets of items which appear in a sufficient number of transactions. We present efficient graph-based algorithms to solve these problems. The algorithms construct an *association graph* to indicate the associations between items and then traverse the graph to generate large itemsets and large sequences, respectively. Our algorithms need to scan the database only once. Empirical evaluations show that our algorithms outperform other algorithms which need to make multiple passes over the database.

1 Introduction

From a large amount of data, potentially useful information may be discovered. Techniques have been proposed to find knowledge (or rules) from databases [1, 2, 3, 6, 9, 10, 15, 17, 21, 22, 24]. The knowledge discovered can be used to answer cooperative queries [7, 14], handle null values [20] and facilitate semantic query optimization [8, 12, 17, 18, 19, 23].

Data mining has also high applicability in retail industry. The effective management of business is significantly dependent on the quality of its decision making. It is therefore important to analyze past transaction data to discover customer purchasing behavior and improve the quality of business decision. In order to support this analysis, a sufficient amount of transaction items needs to be collected and stored in a database. A transaction in the database typically consists of customer identifier, transaction date (or transaction time) and the set of items (itemset) purchased in the transaction. Because the amount of these transaction data is very large, an efficient algorithm needs to be devised for discovering useful information embedded in the transaction data.

In this paper, we study two problems: mining association rules and mining sequential patterns in a large database of customer transactions. The problem of mining association rules over customer transactions was introduced in [2]. An association rule describes the association among items in which when some items are purchased in a transaction, others are purchased too.

In order to find association rules, we need to discover the itemsets which occur often enough within transactions. The first step to find association rules is therefore to identify all itemsets that are contained in a sufficient number of transactions above a certain minimum threshold. After discovering all such itemsets, the association rules can be generated as follows [2]: If the discovered itemset $Y = I_1 I_2 ... I_k, k \geq 2$, all rules that reference items from the set $\{I_1, I_2, ..., I_k\}$ can be generated. The antecedent of each of these rules is a subset X of Y, and the consequent is $Y - X$. The rule $X \Longrightarrow Y - X$ holds in the database D of transactions with *confidence factor* c if at least c% of the transactions in D that contain X also contain $Y - X$. An example of such an association rule is "95% of transactions in which coffee and sugar are purchased, milk is purchased too." The form of this rule is "coffee, sugar $\Longrightarrow$ milk." The antecedent of this rule consists of coffee and sugar and the consequent consists of milk alone. The percentage 95% is the confidence factor of the rule.

The following definitions are adopted from [2]. A transaction *supports* an itemset Z, if Z is contained in the transaction. The *support for an itemset* is defined as the ratio of the total number of transactions which support this itemset to the total number of transactions in D. Hence, the major work of mining association rules is to find all itemsets that satisfy a certain user-specified *minimum support*. Each such itemset is referred to as *large itemset*. An itemset of length k is called a k-itemset and a large itemset of length k a large k-itemset.

The problem of mining sequential patterns in a

*This work was partially supported by the Republic of China National Science Council under Contract No. NSC 86-2213-E-007-009.

0-8186-7475-X/96 $5.00 © 1996 IEEE

large database of customer transactions was introduced in [5]. An example of such a pattern is that customers buy books about "basic computer concepts," and then about "programming language," and then about "system programming."

The problem is stated as follows [5]: A *sequence* is an ordered list of itemsets. A sequence s is denoted as $< s_1, s_2, ..., s_n >$, where s_j is an itemset. The items in s_j represent that these items were bought together. A sequence $< a_1, a_2, ..., a_n >$ is contained in another sequence $< b_1, b_2, ..., b_m >$ if there exist integers $i_1 < i_2 < ... < i_n$, $1 \leq i_k \leq m$, such that $a_1 \subseteq b_{i_1}, ..., a_n \subseteq b_{i_n}$. In a set of sequences, a sequence s is *maximal* if s is not contained in any other sequence. All the transactions of a customer, ordered by increasing transaction-time is a *customer-sequence*.

A customer *supports* a sequence s if s is contained in the customer-sequence for this customer. *The support for a sequence* is defined as the fraction of total customers who support this sequence. Each sequence satisfying a certain minimum support threshold is a *large sequence*. A sequence of length k is called a k-sequence and a large sequence of length k a large k-sequence. The problem of mining sequential patterns [5] is to find the maximal large sequences among all large sequences.

Various algorithms [2, 4, 5, 11, 13, 16] have been proposed to discover large itemsets or sequential patterns. These algorithms generate candidate k-itemsets (k-sequences) ($k \geq 1$) for large k-itemsets (lare k-sequences), scan each transaction in a database to count the supports of these candidate k-itemsets (k-sequences) and find all large k-itemsets (k-sequences) in the kth iteration based on a pre-determined minimum support. However, because the size of the database can be very large, it is very costly to scan the database to count supports for candidate itemsets in each iteration. Hence, the key issue to improve the performance of large itemset and large sequence discovery is to reduce the number of candidates and the amount of data that has to be scanned in each iteration.

In this paper, we propose two algorithms, DLG (Direct Large itemset Generation) and DSG (Direct Sequential pattern Generation), for efficient large itemset generation and efficient sequential pattern generation, respectively, which are significantly different from previous approaches [2, 4, 5, 11, 13, 16]. DLG and DSG are very efficient for finding large itemsets and sequential patterns, respectively, because they need not generate candidates and need to scan the database only once. The algorithms DLG and DSG construct an *association graph* to indicate the associations between items, and then traverse the graph to generate large itemsets and sequential patterns, respectively.

The rest of this paper is organized as follows: Section 2 describes the related work. The algorithm DLG proposed for generating large itemsets and the experimental results for the performance evaluation are presented in Section 3. Section 4 describes the algorithm DSG for sequential pattern generation and evalutes the performance of DSG. Finally, we conclude this paper and present directions for future research in Section 5.

2 Related Work

An algorithm for finding all association rules, called AIS algorithm, was presented in [2]. AIS generates candidate itemsets and counts their supports as the database is scaned in each iteration. After reading a transaction, it is determined which of the large itemsets found in the previous iteration are contained in this transaction. During a database scan, new candidate itemsets are generated by extending these large itemsets with other items in the transaction, and support information is collected to evaluate which of the candidates actually are large. However, it was found [4] that the problem with AIS is that it generates too many candidates that later turn out to be small. Hence, the AIS algorithm is rather inefficient.

The Apriori and AprioriTid algorithms [4] generate the candidate itemsets to be counted in an iteration by using only the itemsets found large in the previous iteration without considering the transactions in the database. This results in generation of a smaller number of candidate itemsets. However, for each candidate itemset, it needs to count its appearances in all transactions. In the Apriori algorithm, each iteration requires one pass over the database. In the AprioriTid algorithm, the database is not scanned after the first iteration. Rather, the transaction-id and candidate k-itemsets which were present in each transaction are generated in each iteration. This is used to count supports for candidate $k+1$-itemsets during the next iteration. It was found that in the initial stages, Apriori is more efficient than AprioriTid, since there are too many candidate k-itemsets to be tracked during the early stages of the process. A hybrid algorithm of the two algorithms was also proposed in [4], and shown to lead to better performance in general.

Park, Chen and Yu [16] pointed out that the key issue to improve the performance of large itemsets discovery is the initial candidate set generation, especially for the candidate 2-itemsets, and the amount of data that has to be scanned during large itemset generation in each iteration. They utilized hash method to reduce the number of candidate 2-itemsets generation and employed pruning techniques to progressively trim the transaction database. The pruning techniques are described as follows: an item in a transaction can be trimmed if it does not appear in at least k of the candidate k-itemsets in the kth iteration. However, in order to reduce the number of candidate 2-itemsets, the overhead for building hash table is large. Moreover, in order to trim the database, it is necessary to scan each transaction in the database to determine which of the candidate itemsets are contained in the transaction.

Agrawal and Srikant [5] presented two algorithms called AprioriAll and AprioriSome for mining sequential patterns in a large database of customer transactions. These two algorithms make multiple passes over the data. In each pass, they use a seed set to generate candidate sequences, and count supports for

each candidate sequence. At the end of the pass, it is determined which of the candidate sequences are actually large. These large candidates become the seed for the next pass. The two algorithms generate too many candidate sequences to be counted, and the database needs to be scaned repeatedly.

In [16], the DHP algorithm is shown to provide the best performance for large itemsets generation. Hence, DHP is used as the base algorithm to compare with our algorithm DLG. The analysis and experimental results are shown in Section 3.3. AprioriAll and AprioriSome have the similar performance, which is shown in [5]. We take AprioriAll algorithm to compare with our algorithm DSG. The analysis and the experimental results are shown in Section 4.3.

3 Association Rule Discovery

In this section, we present the algorithm DLG for efficient large itemset generation. There are three phases in the DLG algorithm: The first phase is the *large 1-itemset generation phase* which generates large items (large 1-itemsets) and records related information. The second phase is the *graph construction phase* which constructs an *association graph* to indicate the associations between large items. In this phase, large 2-itemset can also be generated. The last phase is the *large itemset generation phase* which generates large k-itemsets ($k > 2$) based on the constructed association graph.

In the previous approaches [2, 4, 11, 13, 16], they all need to sort the items in each transaction in their lexicographic order. However, our approach need not to sort the items in each transaction.

3.1 Association graph construction

Before performing the DLG algorithm, each item is assigned an integer number. Suppose item i represents the item whose item number is i. In the first phase, algorithm DLG scans the database once to count the support and build a bit vector for each item. The length of each bit vector is the number of transactions in the database. If an item appears in the ith transaction, the ith bit of the bit vector associated with this item is set to 1. Otherwise, the ith bit of the bit vector is set to 0. The bit vector associated with item i is denoted as BV_i. The number of 1's in BV_i is equal to the number of transactions which support the item i, that is, the support for the item i.

For example, consider the database in Table 1. Each record is a <TID, Itemset> pair, where TID is the identifier of the corresponding transaction, and Itemset records the items purchased in the transaction.

TID	Itemset
100	3 1 4
200	5 3 2
300	1 2 3 5
400	5 2

Table 1: A database of transactions

Assume that the minimum support is 2 transactions. In the large 1-itemset generation phase, the large items found in the database shown in Table 1 are items 1, 2, 3 and 5, and BV_1, BV_2, BV_3 and BV_5 are (1010), (0111), (1110) and (0111), respectively.

Property 1. The support for the itemset $\{i_1,i_2, ...,i_k\}$ is the number of 1's in $BV_{i_1} \wedge BV_{i_2} \wedge ... \wedge BV_{i_k}$, where the notation "$\wedge$" is a logical AND operation.

After the first phase, the database need not be scanned again. In the graph construction phase, DLG constructs an association graph to indicate the associations between items. For the association graph, if the number of 1's in $BV_i \wedge BV_j$ $(i < j)$ is no less than the minimum support, a directed edge from item i to item j is constructed. Also, itemset $\{i,j\}$ is a large 2-itemset. The association graph for the above example is shown in Figure 1, and the large 2-itemsets are {1,3}, {2,3}, {2,5} and {3,5}.

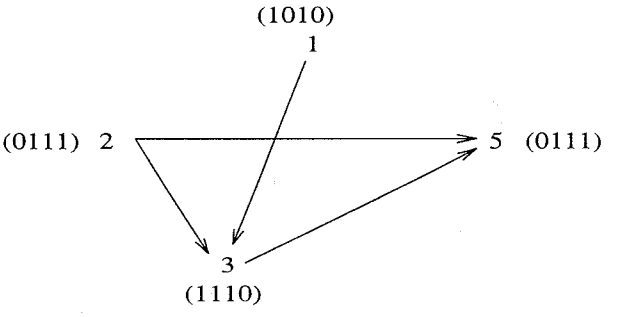

Figure 1: The association graph and the bit vector associated with each large item for Table 1

3.2 Large itemset generation

The large k-itemsets ($k > 2$) are generated based on the association graph constructed in the second phase. The data structure used to implement the association graph is a linked list.

The large 2-itemsets L_2 is found in the graph construction phase. In the large itemset generation phase, the DLG algorithm generates large k-itemsets L_k ($k > 2$). For each large k-itemset in L_k ($k \geq 2$), the last item of the k-itemset is used to extend the itemset into $k+1$-itemsets. Suppose $\{i_1,i_2, ...,i_k\}$ is a large k-itemset. If there is a directed edge from item i_k to item u, then the itemset $\{i_1, i_2, ..., i_k\}$ is extended into $k+1$-itemset $\{i_1,i_2, ...,i_k,u\}$. The itemset $\{i_1,i_2, ...,i_k,u\}$ is a large $k+1$-itemset if the number of 1's in $BV_{i_1} \wedge BV_{i_2} \wedge ... \wedge BV_{i_k} \wedge BV_u$ is no less than the minimum support. If no large k-itemsets can be generated, the DLG algorithm terminates.

For example, consider the database in Table 1. In the second phase, the large 2-itemsets L_2 = {{1,3}, {2,3}, {2,5}, {3,5}} is generated. For large 2-itemset {2,3}, there is a directed edge from the last item 3 of the itemset {2,3} to item 5. Hence, the 2-itemset {2,3} is extended into 3-itemset {2,3,5}. The number of 1's in $BV_2 \wedge BV_3 \wedge BV_5$ (i.e., (0110)) is 2. Hence, the

3-itemset {2,3,5} is a large 3-itemset, since the number of 1's in its bit vector is no less than the minimum support. The DLG algorithm terminates because no large 4-itemsets can be further generated. After completing the DLG algorithm on Table 1, the large itemsets are {1,3}, {2,3}, {2,5}, {3,5} and {2,3,5}. The DLG algorithm for each phase is shown as follows:

```
\* Large 1-itemset generation phase *\
forall items i do
    set all bits of BV_i to 0;
for (j = 1; j ≤ N; j++) do begin
\* N is the number of transactions in database D *\
    forall items i in the jth transaction do begin
        i.count++;
        set the jth bit of BV_i to 1;
    end
end
L_1 = φ;
forall items i in database D do begin
    if i.count ≥ minsup then
    \* minsup is the minimum support threshold *\
        L_1 = L_1 ∪ {i};
end

\* Graph construction phase *\
if L_1 ≠ φ then begin
    forall large 1-itemsets l ∈ L_1 do
        allocate a node for Item[l] and
        Item[l].link=NULL;
    L_2 = φ
    for every two large items i, j (i < j) do begin
        if (the number of 1's in BV_i ∧ BV_j) ≥ minsup
        then begin
        \* create an directed edge
        from i to j in the association graph *\
            allocate a node p;
            p.link = Item[l_a].link;
            p.Item = l_b;
            Item[l_a].link = p;
            L_2 = L_2∪ {{i,j}};
            \* generate large 2-itemsets *\
        end
    end
end

\* Large itemset generation phase *\
k = 2;
while L_k ≠ φ do begin
    L_{k+1} = φ;
    forall itemsets (i_1 i_2 ... i_k) ∈ L_k do begin
        pointer = Item[i_k].link;
        while pointer ≠ NULL do begin
            u = pointer.Item;
            if (number of 1's in BV_{i_1} ∧ ... ∧ BV_{i_k}
            ∧BV_u) ≥ minsup then
                L_{k+1} = L_{k+1}∪ {{i_1,...,i_k,u}};
            pointer = pointer.link;
        end
    end
    k = k + 1
end
```

3.3 Experimental results

To assess the performance of the DLG algorithm for large itemset generation, we perform several experiments on Sun SPARC/10 workstation. The experiments show that the DLG algorithm is very efficient for large itemset generation, because it takes only one database scan to generate large itemsets. We first describe how the datasets are generated for the performance evaluation. We then compare the performance of DLG and DHP [16] by performing experiments on the generated datasets. Finally, we demonstrate the scale-up properties of the DLG algorithm.

3.3.1 generation of synthetic data

The synthetic database of sales transactions is generated to evaluate the performance of the algorithms. The method to generate synthetic transactions is similar to the one used in [4]. The parameters used in our experiments are shown in Table 2.

$\|D\|$	Number of transactions
$\|I\|$	Average size of the potentially large itemsets
$\|MI\|$	Maximum size of the potentially large itemsets
$\|L\|$	Number of large itemsets
$\|T\|$	Average size of the transactions
$\|MT\|$	Maximum size of the transactions
N	Number of items

Table 2: The parameters

We first generate a set L of the potentially large itemsets, and then assign a large itemset picked up from L to a transaction. The size of each potentially large itemset is between 1 and $|MI|$. The probabilities for sizes 1, 2, ... and $|MI|$ are obtained by a Possion distribution with mean equal to $|I|$. These probabilities are normalized such that the sum of these probabilities is 1. For example, suppose average size $|I|$ of the large itemsets is 3 and maximum size $|MI|$ of the large itemsets is 5. According to the Possion distribution with mean $|I|$, the probabilities for sizes 1, 2, 3, 4 and 5 are 0.17, 0.26, 0.26, 0.19 and 0.12, respectively, after the normalization process. These probabilities are then accumulated such that each size falls in a range, which is shown in Table 3. For each potentially large itemset, we generate a random real number which is between 0 and 1 to determine the size of the potentially large itemset.

Size	Range
1	$0 \sim 0.17$
2	$0.18 \sim 0.43$
3	$0.44 \sim 0.69$
4	$0.70 \sim 0.88$
5	$0.89 \sim 1$

Table 3: The probabilities for the sizes of itemsets

The number of the potentially large itemsets in L is set to $|L|$. Items in the first large itemset are chosen randomly. Some fraction of items in subsequent

large itemsets are chosen from the previously generated large itemset. For each item in the previous large itemset, we flip a coin to decide whether the item will be retained in the current large itemset. The remaining items in the large itemset are picked at random. After generating the set L of large itemsets, we then generate transactions in the database. The size of each transaction is picked from a Poisson distribution with mean equal to $|T|$, and the size is between 1 and $|MT|$. The method to determine the size of a transaction is the same as the method to determine the size of a large itemset. For a transaction, we randomly choose a large itemset from L to fit in the transaction and assign it to the transaction. The remaining items of the first transaction are chosen randomly. The fraction of the remaining items of the subsequent transaction are chosen from the previously generated transaction. For each item in the previous transaction, we also flip a coin to decide whether the item is retained in the transaction. After choosing the items from a large itemset and from the previous transaction, the remaining items in the transaction are picked at random. The same as [4], we also use a corruption level during the transaction generation to model the phenomenon that all the items in a large itemset are not always bought together. Each transaction is stored in a file system with the form of <transaction identifier, the number of items, items>.

We generate datasets by setting $N = 1000$ and $|L| = 2000$. We choose three values for $|T|$: 5, 10 and 20, and the corresponding $|MT| = 10$, 20 and 40, respectively. We choose two values for $|I|$: 3 and 5, and the corresponding $|MI| = 5$ and 10, respectively. The number $|D|$ of transactions is set to 100,000. We use $Ta.MTx.Ib.MIy$ to mean that $a = |T|$, $x = |MT|$, $b = |I|$ and $y = |MI|$. We generate the following datasets for the experiments: $T5.MT10.I3.MI5$, $T10.MT20.I3.MI5$, $T10.MT20.I5.MI10$ and $T20.MT40.I3.MI5$.

3.3.2 comparison of DLG and DHP

Figure 2 shows the relative execution time for DHP [16] and DLG, using the four synthetic datasets described in Section 3.3.1. In these experiments, the hash table size $|H_2|$ used in DHP is set to $\frac{1}{4} \times C_2^N$, which was found to have better overall performance in [16], where N is the number of items. Suppose there are $|D|$ transactions in database DB and m items in each transaction on the average. In the kth pass, the large k-itemsets L_k is generated. For the first pass, DLG and DHP both need to scan each transaction in DB to count support for each item. By the way, DLG records the bit vectors for each item. However, DHP needs to take extra overhead to combine every two items to form a 2-itemset in each transaction. Totally, there are $|D| \times C_2^m$ combinations needed. For each combination, DHP uses the hash function to locate the 2-itemset in the hash table. Hence, DHP takes much more time than DLG in the first pass.

Suppose there are $|L_k|$ large itemsets generated in the kth pass. In the second pass, DLG performs $\frac{|L_1|(|L_1|-1)}{2}$ logical AND operations on bit vectors to construct association graph and generate large 2-itemsets. DHP needs to generate candidate 2-itemsets and prune these candidate 2-itemsets using the hash table created in the first pass. Besides, DHP needs to scan database to count support for candidate 2-itemsets and trim the database DB to generate a reduced database. These jobs needed by DHP are more costly than these logical operations performed by DLG in the second pass.

In the kth ($k > 2$) pass, DLG extends each large $k-1$-itemset into k-itemsets according to the association graph and performs logical AND operations. Suppose on the average, each node (item) has q out-degrees in the association graph. DLG performs $(k-1) \times |L_{k-1}| \times q$ logical AND operations to find all large k-itemsets. Hence, as the minimum support decreases, the number of logical AND operations performed increases because the two values $|L_{k-1}|$ and q increase. In the kth pass, DHP generates candidate k-itemsets C_k from large $k-1$-itemsets L_{k-1}. After generating C_k, DHP scans each transaction in the database DB_k to count supports for these candidate k-itemsets and trim the database DB_k to generate another reduced database DB_{k+1}. Hence, the execution time of DHP depends on the number of generated candidate itemsets and the amount of data that has to be scanned.

	DLG	DHP				
pass	$\|L_k\|$	$\|L_k\|$	$\|C_k\|$	$\|DB_k\|$	M_k	m_k
1	288	288	1000	4.265MB	100,000	9.76
2	687	687	974	4.265MB	100,000	9.76
3	497	497	2964	2.014MB	90,747	5.04
4	295	295	832	1.270MB	45,649	5.12
5	118	118	302	502KB	20,258	5.54
6	30	30	84	305KB	10,076	6.33
7	4	4	23	98KB	2,480	6.89

Table 4: Comparisons of DLG and DHP

We perform an experiment on dataset T10.MT20.I5.MI10 with minimum support 0.75%. The experimental results are shown in Table 4, where M_k denotes the number of transactions in DB_k, and m_k denotes the number of items in each transaction on the average.

In this experiment, there are 238 nodes and 687 edges in the association graph. Hence, on the average, the out-degrees of each node is 3. Table 4 shows that in each pass, the number of logical AND operations performed by DLG is much less than the size of database scanned and the number of candidate itemsets generated by DHP. Hence, DHP takes much more time than DLG for large itemset generation. Figure 2 shows that the DLG algorithm outperforms the DHP algorithm significantly, and the performance gap increases as the minimum support decreases because the number of candidate itemsets and the number of database scans increases for DHP.

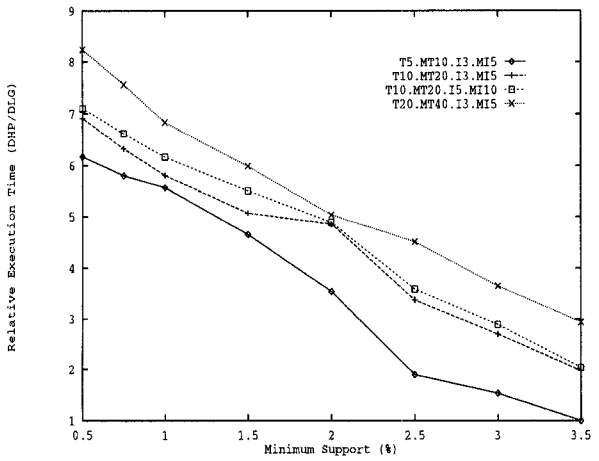

Figure 2: Relative Execution Time

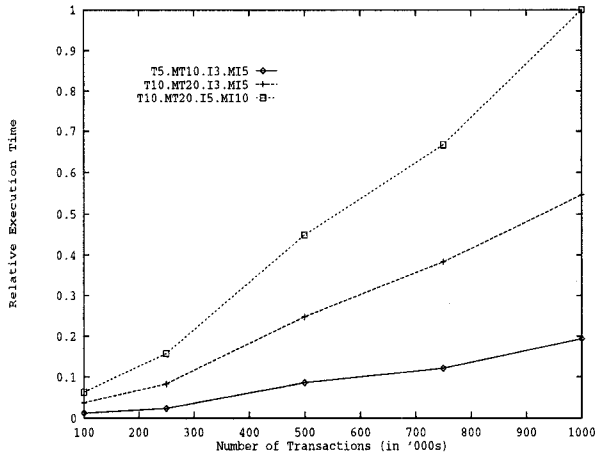

Figure 3: Scale-up: Number of Transactions

3.3.3 discussions for DLG

The memory space needed for performing DLG is dominated by the bit vectors. Since the length of each bit vector is the number of transactions $|D|$ in DB, there are $N \times |D|$ bits needed, where N is the number of items. In our experiments, $N = 10^3$ and $|D| = 10^5$. Hence, $10^3 \times 10^5$ bits (12.5MB) are needed to store all bit vectors.

Figure 3 shows how DLG scales up as the number of transactions is increased from 10,000 to 100,000 transactions. We use the three datasets $T5.MT10.I3.MI5$, $T10.MT20.I3.MI5$ and $T10.MT20.I5.MI10$, and set the minimum support to 1%. As shown, the execution times of DLG increase linearly as the database sizes increase, because the number of large itemsets increases.

Next, we examine how DLG scales up as the number of items increases from 100,000 to 1,000,000 for the three datasets $T5.MT10.I3.MI5$, $T10.MT20.I3.MI5$ and $T10.MT20.I5.MI10$. The minimum support is set to 1% for this experiment, and the results are shown in Figure 4. The execution times decrease slightly, because the number of large itemsets decreases as we increase the number of items.

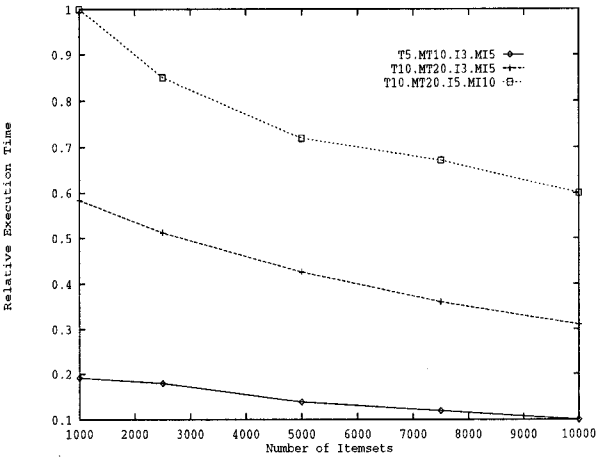

Figure 4: Scale-up: Number of Items

4 Sequential Pattern Discovery

In this section, we present the algorithm DSG for efficient sequential pattern generation. In [5], the problem of mining sequential patterns is splitted into the following phases: 1. Sort phase, 2. Large itemset phase, 3. Transformation phase, 4. Sequence phase and 5. Maximal phase. The Sort phase is to convert the original transaction database into a database of customer-sequences. The customer-sequence is a list of itemsets which are ordered by increasing transaction-times. The Large itemset phase is to find all large itemsets (or large 1-sequences). The Transformation phase is to transform each original customer-sequence into a transformed customer-sequence which is an ordered list of large itemsets.

The Sequence phase and the Maximal phase are the main portions for mining sequential patterns. In these two phases, we propose an algorithm DSG to generate sequential patterns, which needs only one database scan. The DSG algorithm is also splitted into two phases: The first phase is the *graph construction phase* which constructs an association graph to indicate the associations between large itemsets (or large 1-sequences) and records related information. In this phase, large 2-sequences can also be generated. The second phase is the *sequential pattern generation phase* which generates large k-sequences ($k > 2$) based on the constructed association graph and finds maximal large sequences (or sequential patterns).

4.1 Association graph construction

After completing the Transformation phase, we are given a database of transformed customer-sequences. In the graph construction phase, DSG algorithm scans each customer-sequence in the database to combine every two large itemsets to generate a 2-sequence and count support for the 2-sequence. For each 2-sequence, the set of identifiers of the customer-sequences where the 2-sequence appears is recorded. When the support for a 2-sequence achieves the minimum support threshold, the DSG algorithm creates a directed edge from the first itemset to the second itemset in the 2-sequence.

In the following, $\Im_s$ denotes the set of customer identifiers of the customer-sequences where sequence s appears. The cardinality of $\Im_s$ is equal to the number of customer-sequences which support the sequence s, that is, the support for the sequence s.

CID	Csequence
1	ABCD
2	CBE
3	ACB
4	AED
5	CBDE

Table 5: A database of customer-sequences

For example, Table 5 is a database of customer-sequences after completing the Transformation phase. Each record is a <CID, Csequence> pair, where CID is the customer identifier of the corresponding customer-sequence, and Csequence is the customer-sequence. Csequence is a list of large itemsets which are ordered by increasing transaction-times. A large itemset is denoted by an alphabet.

Assume the minimum support is 2 customer-sequences. After scanning the database of customer-sequences in Table 5, the association graph and the recorded information are shown in Figure 5, where the set of numbers on each edge $\overrightarrow{XY}$ is the set of customer identifiers of the customer-sequences where the large 2-sequence $< X, Y >$ appears. After completing the graph construction phase, the large 2-sequences are <A,B>, <A,C>, <A,D>, <B,D>, <B,E>, <C,B>, <C,D> and <C,E>, and the set $\Im_{<A,B>}$ of customer identifiers of the customer-sequences where the 2-sequence <A,B> appears is {1, 3}, and so on.

4.2 Sequential pattern generation

In this section, we describe how to generate large k-sequence ($k > 2$) based on the association graph and the recorded information, and further to find sequential patterns. The large 2-sequences LS_2 is found in the graph construction phase. In the sequential pattern generation phase, the DSG algorithm generates large k-sequences LS_k ($k > 2$). For each large k-sequence in LS_k ($k \geq 2$), the last itemset of the k-sequence is used to extend the sequence into $k+1$-sequences.

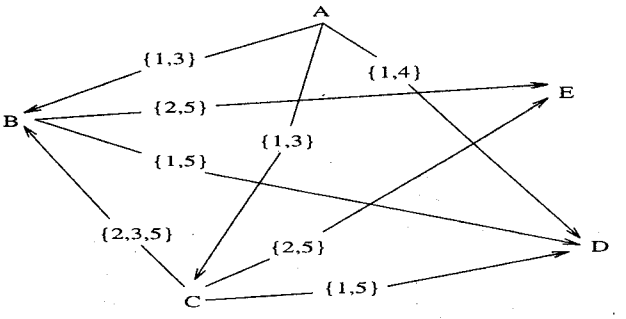

Figure 5: The association graph and the set of identifiers of the customer-sequences where each large 2-sequence appears for Table 5

Property 2. The support for the k-sequence $< s_1, s_2, ..., s_k >$ is the cardinality of the set $\Im_{<s_1,s_2>} \cap \Im_{<s_2,s_3>} \cap ... \cap \Im_{<s_{k-1},s_k>}$.

Suppose $< s_1, s_2, ..., s_k >$ is a large k-sequence. If there is a directed edge from itemset s_k to itemset v, the sequence $< s_1, s_2, ..., s_k >$ is extended into $k+1$-sequence $< s_1, s_2, ..., s_k, v >$. The $k+1$-sequence $< s_1, s_2, ..., s_k, v >$ is a large $k+1$-sequence if the support for the $k+1$-sequence is no less than the minimum support. If no large k-sequence can be generated, the DSG algorithm terminates.

After finding all large sequences LS, the large sequences which are subsequences of the other large sequences are deleted from LS. The remaining large sequences are maximal large sequences, that is, sequential patterns.

For example, consider the database in Table 5 and the association graph in Figure 5. For large 2-sequence <C,B>, there is a directed edge from the last itemset B of the sequence <C,B> to itemset E. Hence, the 2-sequence <C,B> can be extended into 3-sequence <C,B,E>, and the set $\Im_{<C,B,E>}$ can be obtained by performing set intersection on sets $\Im_{<C,B>}$ and $\Im_{<B,E>}$. Because the set $\Im_{<C,B>}$ is {2,3,5} and the set $\Im_{<B,E>}$ is {2,5}, the set $\Im_{<C,B,E>}$ is {2,5}. The 3-sequence <C,B,E> is a large 3-sequence, since the cardinality of set $\Im_{<C,B,E>}$ is no less than the minimum support. After completing the DSG algorithm on Table 5, the sequential patterns are <A,B>, <A,C>, <A,D>, <B,D>, <C,D> and <C,B,E>. The DSG algorithm for each phase is shown as follows:

```
\* Graph construction phase *\
LS1 = { large 1-sequences }
\* result of the Large itemset phase *\
if LS1 ≠ φ then begin
    forall large 1-sequences l ∈ LS1 do
        allocate a node for Itemset[l] and
        Itemset[l].link=NULL
    forall permutation lx ly where lx and ly are
    selected from LS1 do
        ℑ<lx,ly> = φ
        \* ℑ<lx,ly> records the set of identifiers of
        the customer-sequences where lx ly appears *\
```

```
    LS_2 = φ
    for (i = 1; i ≤ N; i++) do begin \* N is the
    number of customer-sequences in database D *\
        scan the ith customer-sequence c
        I_c = the set of all itemsets in c
        forall combination l_a l_b do begin
        \* l_a and l_b are selected from I_c *\
            ℑ_<l_a,l_b> = Im_<l_a,l_b> ∪ {i}
            \* record related information *\
            if < l_a, l_b > .count ≤ minsup then
                < l_a, l_b > .count++
            if < l_a, l_b > .count = minsup then begin
                LS_2 = LS_2 ∪ {< l_a, l_b >}
                \* generate large 2-sequences *\
                CreateEdge(l_a, l_b)
                \* create an edge from l_a to l_b
                in the association graph *\
            end
        end
    end
end
CreateEdge(l_a, l_b)
    allocate a node p
    p.link = Item[l_a].link
    p.Item = l_b
    Item[l_a].link = p


\* Sequential pattern generation phase *\
k = 2
while LS_k ≠ φ do begin
    LS_{k+1} = φ
    forall sequences < s_1, s_2, ..., s_k > ∈ LS_k do begin
        pointer = Itemset[s_k].link
        while pointer ≠ NULL do begin
            v = pointer.Itemset
            if (the cardinality of set ℑ_<s_1,s_2> ∩
            ℑ_<s_2,s_3> ∩ ... ∩ ℑ_<s_k,s_v>) ≥ minsup then
                LS_{k+1} = LS_{k+1} ∪ {< s_1, s_2, ..., s_k, v >}
            pointer = pointer.link
        end
    end
    k = k + 1
end
Answer = Maximal sequences in ∪_k LS_k
```

4.3 Experimental results

To evaluate the performance of the DSG algorithm for sequential pattern generation, we also perform several experiments on Sun SPARC/10 workstation. The experiments show that the DSG algorithm is very efficient for sequential pattern generation, because it takes one database scan to construct an association graph and the large sequences are generated based on the association graph directly. We first generate datasets for the experiments, and then compare the performance between DSG and AprioriAll by performing experiments on the generated datasets. The scale-up properties of the DSG algorithm are also demonstrated.

4.3.1 generation of synthetic data

The method to generate synthetic datasets is similar to the one used in DLG algorithm. The difference between the two methods is described below. For the generated dataset used in DLG algorithm, the items in each transaction are generated in their lexicographic order. However, for the generated dataset used in DSG algorithm, the itemsets in each customer-sequence are generated in an arbitrary order. Each transaction is stored in a file system with the form of <customer-sequence identifier, the number of itemsets, itemsets>. The parameters used in the experiments are as shown in Table 2 with some modifications. In Table 2, the term "transactions" is changed to "customer-sequences," the term "itemsets" is changed to "sequences," the term "items" is changed to "itemsets," and the notation "L", "T" and "MT" are changed to "LS," "C" and "MC," respectively. The number $|D|$ of customer-sequences is set to 100,000. We also set N =1000 and $|LS|$ =2000 for the generated datasets, and generate the four datasets: $C5.MC10.I3.MI5$, $C10.MC20.I3.MI5$, $C10.MC20.I5.MI10$ and $C20.MC40.I3.MI5$ in the experiments.

4.3.2 comparison of AprioriAll and DSG

Figure 6 shows the relative execution time for AprioriAll algorithm [5] and DSG algorithm over various minimum supports, ranging from 0.5% to 3.5%.

Suppose there are M customer-sequences in the database and m itemsets in each customer-sequence on the average. In the kth pass, the set of large k-sequences LS_k is generated.

In the second pass, AprioriAll uses LS_1 to generate $2 \times C_2^{|LS_1|}$ candidate 2-sequences CS_2. Moreover, AprioriAll scans the database to combine every two sequences to form a 2-sequence in each customer-sequence. Totally, there are $M \times C_2^m$ combinations needed. For each combination, AprioriAll searches for the candidate 2-sequences in CS_2 to determine whether the combination is in CS_2 for large 2-sequence generation. However, when $|LS_1|$ is large, $2 \times C_2^{|LS_1|}$ becomes an extremely large number. It is very costly to determine large 2-sequences from a large number of candidate 2-sequences. In this pass, DSG scans each customer-sequence in the database to combine every two sequences to form a 2-sequence and count support for the 2-sequence to determine whether the 2-sequence is large. Because AprioriAll needs to search for a large amount of candidate 2-sequences, DSG outperforms AprioriAll in this pass.

In the kth pass ($k > 2$), AprioriAll generates candidate k-sequences based on large $k-1$-sequences LS_{k-1} and scans the database to count supports for the candidate k-sequences for large k-sequence generation. AprioriAll needs to combine every k sequences to form a k-sequence in each customer-sequence, and totally, $M \times C_k^m$ combinations are needed. For each combination, AprioriAll searches for candidate k-sequences in CS_k to determine whether the k-sequence is in CS_k for

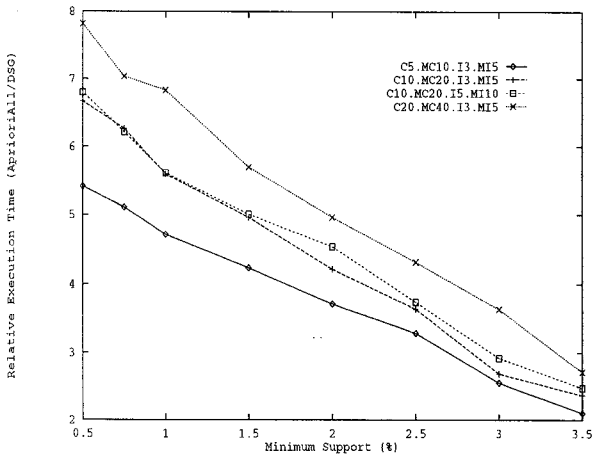

Figure 6: Relative Execution Times

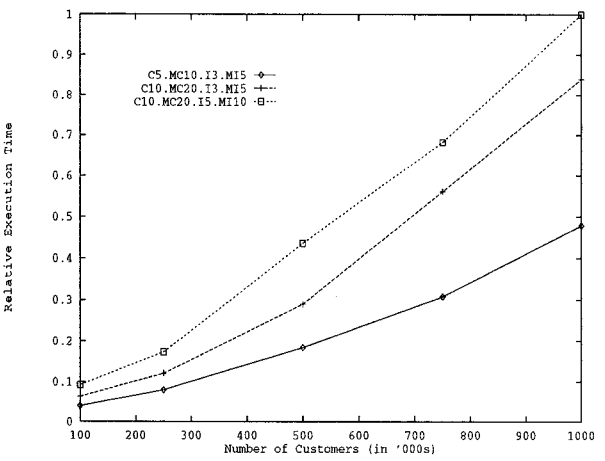

Figure 7: Scale-up: Number of Customers

large k-sequence generation. Hence, as the minimum support decreases, the execution time of AprioriAll increases because the candidate sequence generated increases and the number of database scans increases.

For DSG algorithm, the large k-sequences ($k > 2$) can be generated by extending large $k-1$-sequences into k-sequences based on the association graph and performing set intersections on the related information. Suppose on the average, the number of out-degrees of each node (itemset) in the association graph is q. In the kth ($k > 2$) pass, DSG performs $(k-1) \times q \times |LS_{k-1}|$ set intersections to find all large k-sequences. Hence, as the minimum support decreases, the number of set intersections performed increases, because the values q and $|LS_{k-1}|$ increases. However, DSG need not generate candidate k-sequences ($k \geq 1$) nor scan the database for large k-sequence generation ($k \geq 2$).

Since the number of set intersections performed for DSG is much less than the size of database scanned and the number of candidate itemsets generated for AprioriAll, DSG outperforms the AprioriAll for various minimum supports. Figure 6 shows that the performance gap increases as the minimum support decreases because the number of candidate itemsets generated by AprioriAll increases and the number of database scans also increases.

4.3.3 discussions for DSG

The main memory space needed for performing DSG is to store customer identifiers on each edge in the association graph. Suppose there are l edges in the association graph and on the average, the cardinality of the set of customer identifiers on each edge is k. $l \times k$ customer identifiers need to be stored.

Figure 7 shows how DSG scales up as the number of customer-sequences increases from 10,000 to 100,000 customer-sequences. We use the three datasets $C5.MC10.I3.MI5$, $C10.MC20.I3.MI5$ and $C10.MC20.I5.MI10$, and set the minimum support to 1.5%. As shown, the execution time of DSG increases linearly as the database size increases, because the number of large sequences increases.

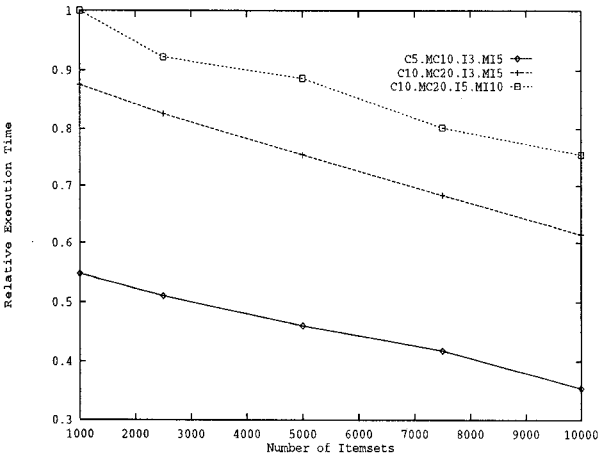

Figure 8: Scale-up: Number of Itemsets

Next, we investigate the scale up as we increase the number of itemsets from 1,000 to 10,000 for the three datasets $C5.MC10.I3.MI5$, $C10.MC20.I3.MI5$ and $C10.MC20.I5.MI10$. The minimum support is set to 1.5% for this experiment. Figure 8 shows the results. When the number of itemsets increases, the execution time decreases slightly, because the number of large sequences decreases.

5 Conclusion and Future Work

We study two problems: mining association rules and mining sequential patterns in a large database of customer transactions. The problems of mining association rules and mining sequential patterns focuses

on discovering large itemsets and discovering large sequences, respectively.

We present two algorithms, DLG and DSG which need only one database scan, for efficient large itemset generation and efficient sequential pattern generation, respectively. These two algorithms construct an association graph to indicate the associations between items and then traverse the graph to generate large itemsets and large sequences, respectively.

We compare DLG and DSG algorithms to the previously known algorithms, DHP [16] and AprioriAll [5], respectively. The experimental results show that DLG and DSG outperform DHP and AprioriAll, respectively. When the minimum support decreases, the performance gap increases because the number of candidate itemsets (candidate sequences) generated by DHP (AprioriAll) increases and the number of database scans also increases.

We demonstrate that the execution time of these two algorithms increases linearly as the database size increases, and the execution time decreases slightly as the number of items (itemsets) increases.

For our graph-based approach, the related information may not fit in the main memory when the size of the database is very large. In the future, we shall develop a mining algorithm based on our graph-based approach, such that in a very large database environment, the mining algorithm can also be run in the main memory. We shall consider mining various different relationships among data in a large database of customer transactions, such as is-a relationships and part-of relationships. We shall also apply our graph-based approach on different applications, such as document retrieval and resource discovery.

References

[1] R. Agrawal and et al. Database Mining: A Performance Perspective. In *IEEE Transactions on Knowledge and Data Engineering*, pages 914–925, 1993.

[2] R. Agrawal and et al. Mining Association Rules Between Sets of Items in Large Databases. In *Proceedings of ACM SIGMOD*, pages 207–216, 1993.

[3] R. Agrawal and et al. An Interval Classifier for Database Mining Applications. In *Proceedings of International Conference on Very Large Data Bases*, pages 560–573, Vancouver, British Columbia, 1992.

[4] R. Agrawal and R. Srikant. Fast Algorithm for Mining Association Rules. In *Proceedings of International Conference on Very Large Data Bases*, pages 487–499, 1994.

[5] R. Agrawal and R. Srikant. Mining Sequential Patterns. In *Proceedings of International Conference on Data Engineering*, pages 3–14, 1995.

[6] Y. Cai, N. Cercone, and J. Han. An Attribute-Oriented Approach for Learning Classification Rules from Relational Databases. In *Proceedings of International Conference on Data Engineering, Los Angeles*, pages 281–288, Feb 1990.

[7] W. Chu and et al. Using Type Inference and Induced Rules to Provide Intensional Answers. In *Proceedings of International Conference on Data Engineering*, pages 396–403, 1991.

[8] M. Hammer and S.B. Zdondik. Knowledge-based query processing. In *Proceedings of International Conference on Very Large Data Bases*, pages 137–146, 1980.

[9] J. Han and et al. Knowledge Discovery in Databases: An Attribute-Oriented Approach. In *Proceedings of International Conference on Very Large Data Bases*, pages 547–559, 1992.

[10] J. Han and et al. Data-Driven Discovery of Quantitative Rules in Relational Databases. In *IEEE Transactions on Knowledge and Data Engineering*, pages 29–40, 1993.

[11] M. Houtsma and A. Swami. Set-Oriented Mining for Association Rules in Relational Databases. In *Proceedings of International Conference on Data Engineering*, pages 25–33, 1995.

[12] C. Malley and S. Zdonik. A Knowledge-Based Approach to Query Optimization. In *Proceedings of the First Expert Database System Conference*, pages 243–257, 1986.

[13] H. Mannila, H. Toivonen, and A.I. Verkamo. Efficient Algorithm for Discovering Association Rules. In *Proceedings of AAAI Workshop on Knowledge Discovery in Databases*, pages 181–192, 1994.

[14] A. Motro. Using Integrity Contraints to Provide Intensional Answers to Relational Queries. In *Proceedings of International Conference on Very Large Data Bases*, 1989.

[15] G. Oosthuizen. Lattice-Based Knowledge Discovery. In *Proceedings of AAAI Workshop on Knowledge Discovery in Databases*, pages 221–235, 1991.

[16] J.S. Park, M.S. Chen, and P.S. Yu. An Effective Hash-Based Algorithm for Mining Association Rules. In *Proceedings of ACM SIGMOD*, 24(2):175–186, 1995.

[17] M.S.E. Sciore and et al. A Method for Automatic Rule Derivation to Support Semantic Query Optimization. In *ACM Transactions on Database Systems*, pages 563–600, 1992.

[18] M. Siegel. Automatic Rule Derivation for Semantic Query Optimization. In *Proceedings of the Second International Conference on Expert Database Systems*, pages 371–385, 1988.

[19] U.Chakravarthy, D. Fishman, and J. Minker. Semantic Query Optimization in Expert Systems and Database Systems. In *Proceedings of the First International Conference on Expert Database Systems*, pages 326–340, 1984.

[20] S.J. Yen and A.L.P. Chen. Neighborhood/Conceptual Query Answering with Imprecise/Incomplete Data. In *Proceedings of International Conference on Entity-Relationship Approach*, pages 151–162, 1993.

[21] S.J. Yen and A.L.P. Chen. The Analysis of Relationships in Databases for Rule Derivation. In *Journal of Intelligent Information Systems*, Vol. 7, pages 1–24, 1996.

[22] S.J. Yen and A.L.P. Chen. An Efficient Algorithm for Deriving Compact Rules from Databases. In *Proceedings of International Conference on Database Systems for Advanced Applications*, pages 364–371, 1995.

[23] C. Yu and W. Sun. Automatic Knowledge Acquisition and Maintenance for Semantic Query Optimization. In *IEEE Transactions on Knowledge and Data Engineering*, pages 362–375, 1989.

[24] W. Ziarko. The Discovery, Analysis, and Representation of Data Dependencies in Databases. In *Proceedings of AAAI Workshop on Knowledge Discovery in Databases*, pages 195–209, 1991.