Set-Oriented Mining for Association Rules in Relational Databases

Maurice Houtsma*
University of Twente
the Netherlands
houtsma@trc.nl

Arun Swami†
IBM Almaden Research Center
San Jose, CA 95120
aruns@sgi.com

Abstract

We describe set-oriented algorithms for mining association rules. Such algorithms imply performing multiple joins and may appear to be inherently less efficient than special-purpose algorithms. We develop new algorithms that can be expressed as SQL queries, and discuss optimization of these algorithms. After analytical evaluation, an algorithm named **SETM** *emerges as the algorithm of choice. Algorithm* **SETM** *uses only simple database primitives, viz., sorting and merge-scan join. Algorithm* **SETM** *is simple, fast, and stable over the range of parameter values. The major contribution of this paper is that it shows that at least some aspects of data mining can be carried out by using general query languages such as SQL, rather than by developing specialized black box algorithms. The set-oriented nature of Algorithm* **SETM** *facilitates the development of extensions.*

1 Introduction

The competitiveness of companies is becoming increasingly dependent on the quality of their decision making. Hence, it is no wonder that companies often try to learn from past transactions and decisions in order to improve the quality of decisions taken in the present or future. In order to support this process, large amounts of data are collected and stored during business operations. Later, these data are analyzed for relevant information. This process is called *data mining* [3, 12, 18, 5] or *knowledge discovery in databases* [8, 15, 9, 11]. Data mining is relevant to many different types of businesses. As examples, retail stores obtain profiles from customers and their buying patterns and supermarkets analyze their sales and the effect of advertising on sales. Such "targeted marketing" [6] is becoming increasingly important.

Different aspects of data mining have been explored in the literature. In *classification*, data units (tuples) are grouped together based on some common characteristics, and rules are generated to describe this grouping. This has been done both in the context of AI [16] and in the context of databases [2, 8, 5]. Work has been done to search for similar sequences or time series [1]. New indexing schemes for facilitating data mining in large archival databases are proposed in [17]. In finding *association rules*, one tries to discover frequently occurring patterns within data units [14, 4]. Our interest is in the problem of finding association rules. There has been a lot of work in rule discovery that is related but not directly applicable, for example, [7, 10, 13, 16].

Business applications deal with an uncontrolled real world, where many rules will overlap in their components and uncertainty is common [14]. Examples of rules could be: "Most sales transactions in which bread and butter are purchased, also include milk," or "Customers with kids are more likely to buy a particular brand of cereal if it includes baseball cards." Work on finding these kinds of rules has been done in AI for some specific applications, (see [15] for an overview). Although the work done in AI is usually very general, the computational complexity of the proposed algorithms is high, and the algorithms are feasible only for small data sets [11]. Performance is a problem with these algorithms for the kind of applications we consider, which involve mining large databases. In [12] a small example is described of generating rules from data, but the emphasis is more on architectural issues than on performance and large data sets. In [4], the problem of rule discovery is addressed in a database context. The paper describes an algorithm for rule discovery on a large data set. However, the algorithm in [4] still has a tuple-oriented flavor (tuples are represented as strings, and the algorithm consists of string

*M. Houtsma's research was made possible by a fellowship of the Royal Netherlands Academy of Arts and Sciences; he is currently at Telematics Research Centre, P.O. Box 217, 7500 AE Enschede, the Netherlands

†A. Swami is currently at Silicon Graphics Computer Systems, 2011 N. Shoreline Blvd, Mountain View, CA 94043-1389

manipulation operations) and is rather complex.

We develop efficient algorithms for mining association rules from large datasets in relational databases. This differentiates our work from much of the work in AI. Problems of optimization of discovered rules, subsumption, etc. are beyond the scope of this paper. Retailing transactions are used in the examples in this paper. However, the work is applicable to mining of association rules from any domain.

We address rule discovery in database systems from a set-oriented perspective. The motivations for a new approach to this problem are several. A set-oriented approach allows a clearer expression of what needs to be done as opposed to specifying exactly how the operations are carried out. The declarative nature of this approach allows consideration of different ways to optimize the required operations. The experience that has been gained in optimizing relational queries can directly be applied here. Eventually, it should be possible to integrate rule discovery completely with the database system. This would facilitate the use of the large amounts of data that are currently stored on relational databases. The relational query optimizer can then determine the most efficient way to obtain the desired results. Finally, our set-oriented approach has a small number of well-defined, simple concepts and operations. This allows easy extensibility to handling additional kinds of mining, e.g., relating association rules to customer classes.

The structure of this paper is as follows. In Section 2 we define the problem of set-oriented data mining and give an initial sketch of our approach. In Section 3 we present an initial set-oriented algorithm expressed in SQL and analyze its performance. In Section 4 we present a second set-oriented algorithm expressed in SQL and analyze its performance. In Section 4.4 we describe the latter algorithm (called Algorithm **SETM**) in terms of simple database operations: sorting and merge-scan join. We also illustrate the algorithm by means of a small example. Section 5 explains how rules are generated. In Section 6 we describe several experiments we did with an implementation of our algorithm on a large data set. Section 7 presents our conclusions.

2 Set-oriented mining

Consider the problem of finding association rules in sales data. Typically, a retail store records information for each customer transaction, where a customer transaction involves the purchase of a variable number of items. We can store this information in a relational database system using a table with the following schema: *SALES(trans_id, item)*. For each customer transaction that takes place, tuples corresponding to the items sold are inserted in *SALES*.

In order to find association rules, we need to scan transactions for reoccurring patterns that occur often enough to be of interest (this is made more precise later). We use the term *pattern* to capture the concept of *itemset* introduced in [4]. This is more in line with existing terminology [15]. A *pattern* can be defined as follows. If items A, B, and C frequently occur together in a single customer transaction, this means that the pattern ABC occurs often. This observation might allow us to conclude (among other rules) the association rule $A \wedge B \Longrightarrow C$ [1]. Here, AB is called the *antecedent* of the rule and C is called the *consequent* of the rule. Usually, some constraints need to be met before we conclude that an association rule holds. As in [4], we define *support* for a pattern to be the ratio of customer transactions supporting that pattern to the total number of customer transactions. Also, the *confidence factor* for a rule obtained from a pattern is defined as the ratio of the support for the pattern to the support for the antecedent of the rule. For the rule $A \wedge B \Longrightarrow C$, this would be $|ABC|/|AB|$, where $|ABC|$ denotes the support for pattern ABC.

We are interested only in association rules where the support for the pattern(s) involved in the rule is greater than some minimum value called *minimum support*. We also require that qualifying rules have a confidence factor greater than some value. Patterns can be generated in a straightforward fashion by repeated joins with *SALES*. For instance, generating all patterns of exactly two items, is expressed by the following SQL query:

SELECT $r_1.trans\_id$, $r_1.item$, $r_2.item$
FROM SALES r_1, SALES r_2
WHERE $r_1.trans\_id = r_2.trans\_id$ AND
$r_1.item <> r_2.item$

For each pair of items (x, y), we count the number of transaction-ids in order to find the number of transactions supporting this pattern. All patterns of exactly three items can now be obtained by joining the result of the previous step again with *SALES* and so on. The order in which items appear is not relevant right now; (x, y) is equivalent to (y, x) since both pairs have the same support. Order only becomes important when generating the rules because confidence factors can be different for different orders.

[1] *Causality* is not necessarily implied. Also, the ordering of A and B in the antecedent of the rule is arbitrary

This strategy is elaborated in Sections 3 and 4 and expressed in terms of a set-oriented query language, viz., SQL. The first expression that is generated naturally leads to nested-loop based joins. A rough analysis of its expected performance indicates that such an implementation would perform very poorly. Consequently, we generate an equivalent expression in SQL that naturally leads to sort-merge based joins. A first analysis shows it to be very promising, and we pursue this implementation in the remainder of the paper.

We include the discussion of both SQL expressions of our strategy, because we wish to emphasize the methodology that we used in this research. Taking a set-oriented approach does not immediately lead to great results but it clearly helps in getting a good understanding of the problem. In our case, by first having studied the nested-loop strategy, we were able to develop the strategy based on sort-merge joins fairly easily, by taking into consideration the ways a relational query optimizer deals with these types of (complex) queries.

3 Using nested-loop joins

We discuss a formulation of our set-oriented data mining strategy that naturally leads to nested-loop based joins. We express the algorithm in SQL and then analyze its expected performance.

3.1 Formulation

The customer transactions are available in the relation *SALES(trans_id, item)*. Using this relation, we first generate the counts for each item x, i.e., the number of transactions that support item x. We check that the minimum support requirement is met and store the result in relation C_1*(item, count)*.

```
INSERT INTO C1
SELECT r1.item, COUNT(*)
FROM SALES r1
GROUP BY r1.item
HAVING COUNT(*) >= :min_support
```

The next step is to generate all patterns (x, y) and check if they meet the minimum support criterion. For a specific item A, this is easy to express. For example, all patterns (A, y) can be generated using a self-join of *SALES*, as shown below.

```
SELECT r1.item, r2.item, COUNT(*)
FROM SALES r1, SALES r2
WHERE r1.trans_id = r2.trans_id AND
      r1.item ='A' AND
      r2.item <> 'A'
GROUP BY r1.item, r2.item
HAVING COUNT(*) >= :min_support
```

This kind of expression only generates patterns with a specific item in the first position. The expression has to be generalized in order to generate arbitrary patterns. As stated earlier, the order of the items in a pattern is not relevant at the time of generation. The order is important only in the final rule generation process. We take advantage of this fact by generating patterns with the items in lexicographical order. For instance, we generate AB, but we do not generate BA. We generalize over all values of *item* having minimum support, by using the following SQL expression to generate *all* lexicographically ordered patterns of length k having minimum support.

```
INSERT INTO Ck
SELECT r1.item, ..., rk.item, COUNT(*)
FROM Ck-1 c, SALES r1, ..., SALES rk
WHERE r1.trans_id = ... = rk.trans_id AND
      r1.item = c.item1 AND
      :
      rk-1.item = c.itemk-1 AND
      rk.item > rk-1.item
GROUP BY r1.item, ..., rk.item
HAVING COUNT(*) >= :min_support
```

Relation C_k has schema ($item_1$, $item_2$, ..., $item_k$, *count*). All feasible rules are found by consecutively generating all qualifying patterns from length 1 to k until $C_{k+1} = \{\}$. Since the items in the patterns are lexicographically ordered, a single inequality test in the SQL query is sufficient.

3.2 Analysis

We perform a simple analysis of the expected performance of the strategy based on nested-loop joins. Let us consider how a relational query optimizer could optimize the final SQL expression in Section 3.1. For efficient evaluation of the nested-loop joins, we need two indexes on the table *SALES*: an index on *(item, trans_id)* and another index on *(trans_id)*. Given these indexes, the query can be evaluated as follows:

1. Take a tuple c from C_{k-1}, and use the index on *(item, trans_id)* for r_1 to get qualifying tuples with $r_1.item = c.item_1$

2. For each of these tuples, use the index on *(item, trans_id)* for r_2 to get tuples that satisfy $r_2.item = c.item_2$ and $r_2.trans\_id = r_1.trans\_id$
3. Similarly for relations $r_3, \ldots, r_{k-1}$.
4. Finally, use the index on *(trans_id)* for r_k to compute $r_k.trans\_id = r_{k-1}.trans\_id$ and check the remaining condition $r_k.item > r_{k-1}.item$.
5. The qualifying tuples are sorted on the item values and the count is used to check the minimum support constraint.

Let us consider a hypothetical retailing database to characterize the performance of this strategy. There are 1000 different items that can be sold. The data consists of 200,000 customer transactions. The average number of items sold in a transaction is 10. Thus, the relation *SALES* contains about 2 million tuples. To make the analysis tractable, we assume that the items have approximately equal probability of being sold (in the actual data set, the items are not sold with equal probability). Hence, the chance of an item appearing in a particular transaction is 1%. We will assume the following characteristics for the database system. Page size is 4 Kbytes, and each item and transaction id is represented using 4 bytes (item values are represented by integers). Hence, each initial tuple consists of 8 bytes.

Consider the B^+-tree index on *(item, trans_id)*. Since all the data is contained in the index, we do not need a pointer in the leaf page entries. Assuming little overhead, we can store upto 500 entries in each leaf page. The number of leaf pages in the B^+-tree index on *(item, trans_id)* is $2,000,000/500 \approx 4,000$. Assuming 4 bytes for a pointer, an index entry in the non-leaf pages has a size of 12 bytes. Assuming very little overhead, we can store about 333 key-value/pointer pairs on a non-leaf index page. The following inequality holds for the number of levels L of the index tree: $333^L \geq 1,000,000 > 333^{L-1}$; hence, $L = 3$. The number of non-leaf pages in this index is $(1 + 4,000/333) = 14$. Similar calculations for the index on *(trans_id)* show that the number of leaf pages is 2,000 and the number of non-leaf pages is 5. Since the number of non-leaf pages is small, we can assume that they reside in memory and are not fetched from disk.

Let the minimum support desired be 1000 transactions, i.e., 0.5% of the total number of transactions. On the average, each item appears in about 1% of the transactions. Assuming uniform probabilities, all items qualify as having minimum support. Therefore, the cardinality of C_1 will be 1000.

To obtain C_2, we take each tuple c from C_1 and access the index on *(item, trans_id)*. This requires $1\% \times 4,000$ leaf page fetches, i.e, ≈ 40 page fetches. The result consists of about 2,000 transaction-ids (1%). For each transaction-id we now have to access the index on *(trans_id)* resulting in 1 page fetch.

From this, we may conclude that the first step alone will require about $1000 \times (40 + 2000 \times 1) \approx 2,000,000$ page fetches. Most of these page fetches are random. A *random* page fetch costs about 20 ms. Hence, the time for the first step alone is $\approx 40,000$ seconds, which is more than 11 hours!

Clearly, the implementation based on nested-loop joins is very inefficient. However, one could consider a different implementation for the same basic pattern finding strategy, viz., sort-merge joins. We will consider this strategy in the next section.

4 Using sort-merge joins

We now discuss the second formulation of our set-oriented data mining strategy based on using sort-merge joins. We again express the algorithm in SQL and then analyze its expected performance.

4.1 Formulation

In the previous implementation, we would generate intermediate relations $R_i(trans\_id, item_1, \ldots, item_i)$, extract support information from these relations, and then discard them. But what if, after each step, we saved the last R_i that was generated? Furthermore, let us save R_i sorted on $(trans\_id, item_1, \ldots, item_i)$. We could then generate all lexicographically ordered patterns of length k using the following expression:

```
INSERT INTO R'_k
SELECT p.trans_id, p.item_1, ..., p.item_{k-1}, q.item
FROM R_{k-1} p, SALES q
WHERE q.trans_id = p.trans_id AND
      q.item > p.item_{k-1}
```

After generating all lexicographically ordered patterns of length k in R'_k, we now have to generate counts for those patterns in R'_k that meet the minimum support constraint. This can be done as follows.

```
INSERT INTO C_k
SELECT p.item_1, ..., p.item_k, COUNT(*)
FROM R'_k p
GROUP BY p.item_1, ..., p.item_k
HAVING COUNT(*) >= :min_support
```

Before we go on to generate patterns of length $k+1$, we first have to select the tuples from R'_k that should be extended, viz., those tuples that meet the minimum support constraint. We also wish the resulting relation to be sorted on ($trans\_id$, $item_1$, ..., $item_k$). This is done as follows:

INSERT INTO R_k
SELECT $p.trans\_id, p.item_1, \ldots, p.item_k$
FROM R'_k p, C_k q
WHERE $p.item_1 = q.item_1$ AND
$\vdots$
$p.item_k = q.item_k$
ORDER BY $p.trans\_id, p.item_1, \ldots, p.item_k$

We can now repeat this process, until at some point $R_k = \emptyset$. Note that the sorting we did in the last step is not really required. It does, however, enable an efficient execution plan if the sort order of the relations is tracked across iterations.

4.2 Example

We illustrate this strategy by means of an example. The example database consists of 10 transactions where each transaction has 3 items. We require a minimum support of 30%, i.e., 3 transactions. The desired confidence factor is 70%. The customer transactions are shown in Figure 1. For brevity, we have presented the transactions as non-normalized tuples. The algorithm, however, uses the tuple format described before; a subset of the corresponding relation is shown too. The contents of the count relation C_1 are also shown in Figure 1.

R'_k and R_k denote the R relations before and after elimination of patterns that do not meet the minimum support count. In the first iteration, R_2 is generated and sorted on items and C_2 is generated from R_2. We show R'_2, R_2 and C_2 in Figure 2.

In the next iteration, R_3 is generated and sorted on items and C_3 is generated from R_3. The contents of R'_2, R_2 and C_2 are as shown in Figure 3. The next iteration will not generate any new tuples, and the algorithm terminates.

4.3 Analysis

In the section the performance of the sort-merge strategy is analyzed using the same data set as for the nested-loop strategy.

The I/O complexity of the sort-merge strategy can easily be expressed by a formula derived as follows. Let $||R_k||$ denote the number of pages used to store the relation in iteration k. In the worst case, applying the minimum support constraints does not eliminate any tuples from R_k. Assume that no patterns of length n have the minimum support, i.e., the relation R_n is empty. We thus make $(n-1)$ passes, i.e., $(n-1)$ merge-scans requiring $(n-1)||R_1|| + \sum_{i=1}^{n-1} ||R_i||$ page accesses. The number of page accesses to store the result of these merge-scans is $\sum_{i=2}^{n-1} R_i$. After each merge-scan, the output is read again, sorted, and written out to disk requiring $2\sum_{i=2}^{n-1} ||R_i||$ page accesses. (We assume R_1 to be sorted, and the sort operations to take place in pipelining mode.) No page accesses are required for storing or retrieving C_i since it is usually small enough to be kept in memory being the result of an aggregation query. Therefore, the total number of page accesses is bounded by:

$$n||R_1|| + 4\sum_{i=2}^{n-1} ||R_i||$$

Let us calculate the time to generate C_2 as we did for the nested-loops strategy. Let R_3 be empty. Using the same numbers as in Section 3.2, the cardinality of R_i is given by $\binom{10}{i} \times 200,000$. The size of a tuple from R_i is $(i+1) \times 4$ bytes. This gives us the following: $||R_1|| = 4,000$ and $||R_2|| = 27,000$. The number of page accesses is thus:

$$3 \times 4,000 + 4 \times 27,000 = 120,000$$

Reading and writing all the R_i relations can be done in a sequential fashion. We estimate the time for each page access as 10 ms. Hence, the total time spent on I/O operations is 1200 seconds or 10 minutes. In comparison, the nested-loop strategy required more than 11 hours.

This rough analysis shows that the implementation based on sort-merge joins will be much more efficient than the algorithm based on using nested-loop join with indexes. We will therefore proceed with further experimental evaluation of the algorithm based on sort-merge joins.

4.4 Algorithm SETM

The sort-merge strategy is described in pseudocode in Figure 4. We refer to it as Algorithm **SETM**. The algorithm consists of a single loop, in which two sort operations and one merge-scan join are performed. The first sort is needed to implement the merge-scan join that follows it. The second sort is used in order to generate the support counts efficiently. Generating the counts involves a simple sequential scan over

tx_id	item	item	item
10	A	B	C
20	A	B	D
30	A	B	C
40	B	C	D
50	A	C	G
60	A	D	G
70	A	E	H
80	D	E	F
90	D	E	F
99	D	E	F

tx_id	item
10	A
10	B
10	C
20	A
20	B
20	D
30	A
30	B
30	C
...	...

item	cnt
A	6
B	4
C	4
D	6
E	4
F	3

Figure 1: Customer transactions, corresponding relation, and relation C_1

tx_id	$item_1$	$item_2$
10	A	B
10	A	C
10	B	C
20	A	B
20	A	D
20	B	D
30	A	B
30	A	C
30	B	C
...	...	...

$item_1$	$item_2$	cnt
A	B	3
A	C	3
B	C	3
D	E	3
D	F	3
E	F	3

tx_id	$item_1$	$item_2$
10	A	B
10	A	C
10	B	C
20	A	B
30	A	B
30	A	C
30	B	C
40	B	C
50	A	C
...	...	...

Figure 2: Relations R'_2, C_2, and R_2

tx_id	$item_1$	$item_2$	$item_3$
10	A	B	C
30	A	B	C
20	A	B	D
40	B	C	D
80	D	E	F
90	D	E	F
99	D	E	F

$item_1$	$item_2$	$item_3$	cnt
D	E	F	3

tx_id	$item_1$	$item_2$	$item_3$
80	D	E	F
90	D	E	F
99	D	E	F

Figure 3: Relation R'_3, C_3 and R_3

R_k. Deleting the tuples from R_k that do not meet the minimum support, involves simple table look-ups on relation C_k. The C_k relations are of interest to us for rule generation. We have not included in this algorithm the optimizations mentioned in Section 4.3.

$k := 1$;
sort R_1 on *item*;
C_1 := generate counts from R_1;
repeat
  $k := k + 1$;
  sort R_{k-1} on *trans_id*, $item_1, \ldots, item_{k-1}$;
  R'_k := merge-scan R_{k-1}, R_1;
  sort R'_k on $item_1, \ldots, item_k$;
  C_k := generate counts from R'_k;
  R_k := filter R'_k to retain supported patterns;
until $R_k = \{\}$

Figure 4: Outline of Algorithm **SETM**

5 Rule generation

We have omitted so far any discussion of how the rules are generated from the count relations. The rule generation algorithm is straightforward. For any pattern of length k, we consider all possible combinations of $k-1$ items in the antecedent. The remaining item not used in the combinations is in the consequent. For each combination of antecedent and consequent, we check if the confidence factor meets or exceeds the minimum confidence factor desired. If the confidence factor is high enough, the rule is written to output. In order to check the confidence factor, we need the count for the current pattern (available in the current count relation C_i) and the count for the pattern comprising the antecedent (available by lookup in a previous count relation C_{i-1}).

Let us consider the example from Section 4.4. The minimum support is 30% (3 transactions) and the minimum confidence factor is 70%. After relation C_2 is obtained, the rules obtained are shown below. Rules have been written in the form $X \Longrightarrow I$, $[c, s]$, where X is the list of items in the antecedent of the rule, I is the item in the consequent of the rule, s is the support expressed as a percentage and c is the confidence factor. Let us see how we obtain the rule $B \Longrightarrow A$. The pattern AB is supported since its support is 3 and the minimum support desired is 3. The ratio $|AB|/|B| = 3/4 = 75\%$ which is greater than the minimum confidence factor of 70%. The ratio $|AB|/|A| = 3/6 = 50\%$ which is less than the minimum confidence factor of 70%. Hence, we do not obtain the rule $A \Longrightarrow B$.

```
B ==> A, [75.0%, 30.0%]
C ==> A, [75.0%, 30.0%]
B ==> C, [75.0%, 30.0%]
C ==> B, [75.0%, 30.0%]
E ==> D, [75.0%, 30.0%]
F ==> D, [100.0%, 30.0%]
E ==> F, [75.0%, 30.0%]
F ==> E, [100.0%, 30.0%]
```

After the second iteration, relation C_3 is available. The rules generated from C_3 are:

```
D E ==> F, [30.0%, 100.00%]
D F ==> E, [30.0%, 100.00%]
E F ==> D, [30.0%, 100.00%]
```

6 Experiments

In previous sections we have described the new algorithm and given some analysis to show that we expect it to be efficient. We implemented the algorithm to run in main memory and read a file of transactions. The execution times given are for running the algorithm on the IBM Risc/System 6000 350 with a clock speed of 41.1 MHz. In [4], a data set was used that consists of sales data obtained from a large retailing company with a total of 46,873 customer transactions. The experiments were conducted using this data set.

6.1 Variation of relation sizes

We first study how the size of the R_i (trans_id and items) relation varies with each iteration of algorithm **SETM**. In Figure 5 we show the variation in the size (in Kbytes) of R_i with iteration i for the retailing data set. Curves are shown for different values of minimum support, where minimum support is varied from 0.1% to 5%. The maximum size of the rules is 3, hence in all cases $|R_4| = 0$ (with $|R_i|$ denoting the cardinality of R_i). Also, the starting relations are the same and hence $|R_1| = 115,568$ in all cases.

If the minimum support is small enough ($\leq 0.1\%$), the size of relation R_i can first increase and then decrease. But the general trend is that the size relation R_i decreases. For large values of minimum support, $|R_i|$ decreases quite rapidly from the first iteration to the second. This sharp decrease is delayed somewhat for the smaller values of minimum support. Hence, using small values of minimum support allows us to obtain more rules. In general, it also allows us to

obtain rules with more items in the antecedent. For example, if the minimum support is reduced to 0.05%, we obtain rules with 3 items in the antecedent.

We expect the C_i (count) relations to be small enough to fit in memory. We now study how the cardinality ($|C_i|$) of these relations varies with iteration number. Figure 6 shows curves for different values of *minimum support*. The values of $|C_i|$ measure the number of item combinations that could garner enough support. We observe that for small values of minimum support the value of $|C_i|$ increases initially before decreasing with later iterations. Since $|C_i|$ is a measure of how many rules can possibly be generated, we again see the importance of handling small values of minimum support in a timely fashion. The maximum size of the rules is 3, hence in all cases $|C_4| = 0$. Also, the starting relations are the same and hence $|C_1| = 59$ for all minimum support values.

6.2 Execution times

We measured the execution times of our set-oriented algorithm **SETM** for various values of the minimum support. We varied the minimum support from 0.1% to 5%. The execution times are shown in the following table.

Minimum Support (%)	Execution Time (seconds)
0.1	6.90
0.5	5.30
1	4.64
2	4.22
5	3.97

We see that algorithm **SETM** is very stable. The execution time varies from $\approx$ 7 secs for 0.1% minimum support to $\approx$ 4 secs for 5% minimum support.

7 Conclusions

In this paper, we have investigated a set-oriented approach to mining association rules. We have shown that by following a set-oriented methodology, we arrived at a simple algorithm. The algorithm is straightforward—basic steps are sorting and merge scan join—and could be implemented easily in a relational database system. The major contribution of this paper is that it shows that at least some aspects of data mining can be carried out by using general query languages such as SQL, rather than by developing specialized black box algorithms.

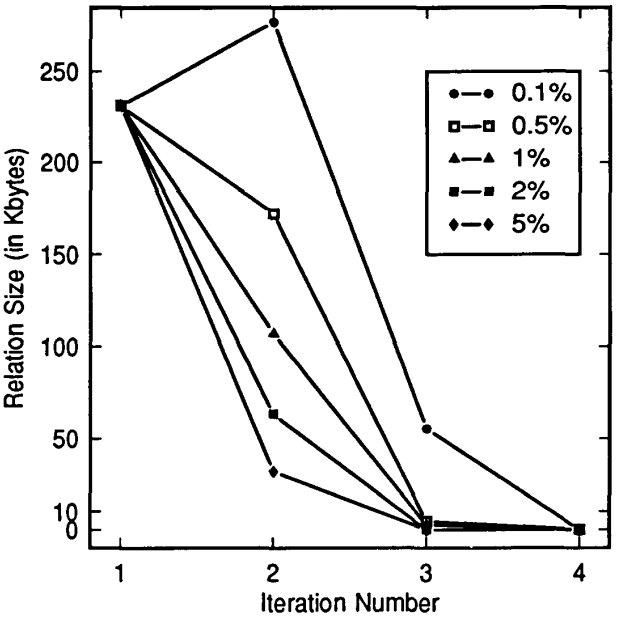

Figure 5: Size of relation R_i

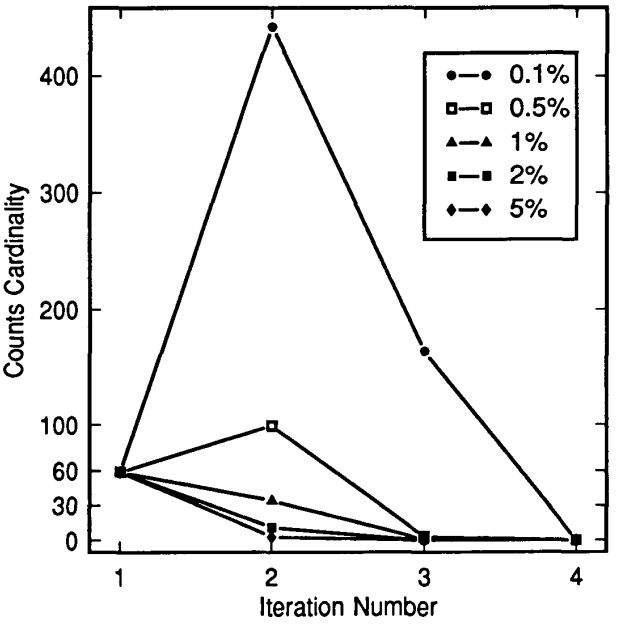

Figure 6: Cardinality of C_i

The algorithm exhibits good performance and stable behavior, with execution time almost insensitive to the chosen minimum support. For a real-life data set, execution times are on the order of 4-7 seconds. The simple and clean form of our algorithm makes it easily extensible and facilitates integration into a (interactive) data mining system. We are investigating extending the algorithm in order to handle additional kinds of mining, e.g., relating association rules to customer classes.

Acknowledgements

We thank Rakesh Agrawal, Bobbie Cochrane, Bill Cody and Hamid Pirahesh.

References

[1] R. Agrawal, C. Faloutsos, and A. Swami. Efficient Similarity Search In Sequence Databases. In *Proceedings of the Fourth International Conference on Foundations of Data Organization and Algorithms*, pages 69–84. Springer-Verlag, Berlin, October 1993. Lecture Notes in Computer Science, V303.

[2] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami. An Interval Classifier for Database Mining Applications. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 560–573, Vancouver, August 1992.

[3] R. Agrawal, T. Imielinski, and A. Swami. Database Mining: A Performance Perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993. Special issue on Learning and Discovery in Knowledge-Based Databases.

[4] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules Between Sets of Items in Large Databases. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 207–216, Washington, DC, June 1993.

[5] T. M. Anwar, H. W. Beck, and S. B. Navathe. Knowledge Mining by Imprecise Querying: A Classification-Based Approach. In *IEEE 8th International Conference on Data Engineering*, Phoenix, Arizona, 1992.

[6] David Shepard Associates, editor. *The New Direct Marketing*. Business One Irwin, Homewood, Illinois, 1990.

[7] P. Cheeseman. AutoClass: A Bayesian Classification System. In *5th International Conference on Machine Learning*. Morgan Kaufman, June 1988.

[8] J. Han, Y. Cai, and N. Cercone. Knowledge Discovery in Databases: An Attribute-Oriented Approach. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 547–559, Vancouver, August 1992.

[9] R. Krishnamurthy and T. Imielinski. Practitioner Problems in Need of Database Research: Research Directions in Knowledge Discovery. *ACM-SIGMOD Record*, 20(3):76–78, September 1991.

[10] P. Langley, H. Simon, G. Bradshaw, and J. Zytkow, editors. *Scientific Discovery: Computational Explorations of the Creative Process*. MIT Press, 1987.

[11] D. J. Lubinsky. Discovery from Databases: A Review of AI and Statistical Techniques. In *IJCAI-89 Workshop on Knowledge Discovery in Databases*, pages 204–218, 1989.

[12] R.S. Michalski, L. Kerschberg, K.A. Kaufman, and J.S. Ribeiro. Mining for Knowledge in Databases: The INLEN Architecture, Initial Implementation, and First Results. *Journal of Intelligent Information Systems*, 1:85–113, 1992.

[13] J. Pearl, editor. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufman, 1992.

[14] G. Piatetsky-Shapiro. Discovery, Analysis, and Presentation of Strong Rules. In *Knowledge Discovery in Databases*, pages 229–248. AAAI/MIT Press, 1991.

[15] G. Piatetsky-Shapiro, editor. *Knowledge Discovery in Databases*. AAAI/MIT Press, 1991.

[16] J. R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1:81–106, 1986.

[17] P. Seshadri and A. Swami. Generalized Partial Indexes. In *Proceedings of IEEE Data Engineering Conference*. IEEE Computer Society, March 1995.

[18] S. Tsur. Data Dredging. *IEEE Database Engineering Bulletin*, 13(4):58–63, December 1990.