Mining Generalized Association Rules and Sequential Patterns Using SQL Queries

Shiby Thomas
Database Systems R & D Center
CISE Department
University of Florida, Gainesville FL 32611
sthomas@cise.ufl.edu

Sunita Sarawagi
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120
sunita@almaden.ibm.com

Abstract

Database integration of mining is becoming increasingly important with the installation of larger and larger data warehouses built around relational database technology. Most of the commercially available mining systems integrate loosely (typically, through an ODBC or SQL cursor interface) with data stored in DBMSs. In cases where the mining algorithm makes multiple passes over the data, it is also possible to cache the data in flat files rather than retrieve multiple times from the DBMS, to achieve better performance. Recent studies have found that for association rule mining, with carefully tuned SQL formulations it is possible to achieve performance comparable to systems that cache the data in files outside the DBMS. The SQL implementation has potential for offering other qualitative advantages like automatic parallelization, development ease, portability and inter-operability with relational operators. In this paper, we present several alternatives for formulating as SQL queries association rule generalized to handle items with hierarchies on them and sequential pattern mining. This work illustrates that it is possible to express computations that are significantly more complicated than simple boolean associations, in SQL using essentially the same framework.

1 Introduction

Data mining on large data warehouses for nuggets of decision-support knowledge is becoming crucial for business organizations. Most of the commercial "knowledge discovery" tools integrate loosely with data stored in DBMSs, typically through a cursor interface. The integration of mining with database querying will facilitate leveraging the query processing capabilities of the DBMS for mining.

There have been a few research efforts to tightly integrate mining with databases (Agrawal & Shim 1996). Several language extensions have also been proposed to extend SQL with mining operators. Recently, researchers have addressed the issue of expressing boolean association rule mining as SQL queries. (Sarawagi, Thomas, & Agrawal 1998) presents several architectural alternatives for integrating boolean association rule mining with relational databases. They develop several SQL formulations for association rule mining and show that with carefully tuned SQL formulations it is possible to achieve performance comparable to mining systems that cache the data in flat files.

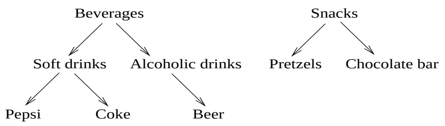
In this paper, we present various alternatives for formulating generalized association rule (Srikant & Agrawal 1995) and sequential pattern (Srikant & Agrawal 1996) mining as SQL queries. These mining techniques make use of additional information about the application as well as data to discover more useful knowledge. In generalized association rules, application-specific knowledge in the form of taxonomies (*is-a* hierarchies) over items are used to discover more interesting rules, where as sequential pattern mining utilizes the time associated with the transaction data to find frequently occurring patterns. One of the advantages of SQL-based mining algorithms is fast and easy development since they are declaratively specified as a set of SQL queries. This work shows how the boolean association rule framework developed in (Sarawagi, Thomas, & Agrawal 1998) can be easily augmented with non-trivial extensions to handle complex mining tasks. We develop SQL-92 formulations as well as those which utilizes the object-relational extensions of SQL (SQL-OR). Although the SQL-92 formulations are slow compared to SQL-OR, the use of just the standard SQL features makes them more portable across commercial DBMSs. The SQL-OR approaches leverages the object-relational extensions to improve performance.

The rest of the paper is organized as follows. We present SQL-formulations of generalized association rule mining in Section 2. In Section 3, we briefly introduce sequential pattern mining and develop several SQL-based implementations. We report some of the results of our performance experiments in Section 4 and conclude in Section 5.

2 Generalized Association Rules

In most real-life applications taxonomies (is-a hierarchies) over the items are available. The taxonomy shown in Figure 1 says that Pepsi *is-a* soft drink *is-a* beverage etc. In general, a taxonomy can be represented as a directed acyclic graph (DAG). Given a set of transactions T each of which is a set of items, and a taxonomy *Tax*, the problem of mining generalized association rules is to discover all rules of the form $X \rightarrow Y$, with the user-specified minimum support and minimum confidence. X and Y can be sets of items at any level of the taxonomy, such that no item in Y is an ancestor of any item in X (Srikant & Agrawal 1995). For example, there might be a rule which says that "50% of transactions that contain Soft drinks also contain Snacks; 5% of all transactions contain both these items".

Input format The taxonomy is represented as a table *Tax* with the schema, (parent, child). Each record in *Tax* corresponds to an edge in the taxonomy DAG. The transaction table T has two attributes: transaction identifier (*tid*) and item identifier (*item*).



Parent	Child
Beverages	Soft drinks
Beverages	Alcoholic drinks
Soft drinks	Pepsi
Soft drinks	Coke
Alcoholic drinks	Beer
Snacks	Pretzels
Snacks	Chocolate bar

Figure 1: A sample taxonomy and the corresponding taxonomy table

(Srikant & Agrawal 1995) presents several algorithms: Basic, Cumulate, Stratify and EstMerge. We picked Cumulate for our SQL-formulations since it has the best performance. Stratify and EstMerge are sometimes marginally better but they are far too complicated to merit the additional development cost. Cumulate has the same basic structure as the Apriori algorithm (Agrawal *et al.* 1996) for boolean associations. It makes multiple passes over the data where in the k^{th} pass it finds frequent itemsets of size k. Each pass consists of two phases: In the candidate generation phase, the frequent $(k-1)$-itemsets, F_{k-1} is used as the seed set to generate candidate k-itemsets (C_k) that are potentially frequent. In the support counting phase for each itemset $t \in C_k$, the number of extended transactions (transactions augmented with all the ancestors of its items) that contains t is counted. At the end of the pass, the frequent candidates are identified yielding F_k. The algorithm terminates when F_k or C_{k+1} becomes empty.

The above basic structure is augmented with a few optimizations. The important ones are pruning itemsets containing an item and its ancestor and pre-computing the ancestors for each item. We extend the SQL-based boolean association rule framework in (Sarawagi, Thomas, & Agrawal 1998) with these optimizations.

In Section 2.1, we present the SQL-based ancestor pre-computation. Candidate generation and support counting are presented in Sections 2.2 and 2.3 respectively.

2.1 Pre-computing Ancestors

We call $\hat{x}$ an ancestor of x if there is a directed path from $\hat{x}$ to x in *Tax*. The *Ancestor* table is primarily used for (i) pruning candidates containing an item and its ancestor and (ii) extending the transactions by adding all the ancestors of its items. We use the transitive closure operation in SQL3 as shown in Figure 2 for the ancestor computation. The result of the query is stored in table *Ancestor* having the schema (*ancestor, descendant*).

2.2 Candidate Generation

The candidate generation and pruning to obtain C_k from F_{k-1} is expressed as a k-way join between the F_{k-1}'s in (Sarawagi, Thomas, & Agrawal 1998). We use the same formulation except that we need to prune from C_k itemsets containing an item and its ancestor. (Srikant & Agrawal 1995) proves that this pruning

```
insert into Ancestor with R-Tax (ancestor, descendant) as
    (select parent, child from Tax union all
    select p.ancestor, c.child from R-Tax p, Tax c
    where p.descendant = c.parent)
select ancestor, descendant from R-Tax
```

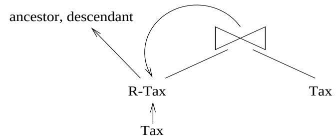


Figure 2: Pre-computing Ancestors

needs to be done only in the second pass (for C_2). In the SQL formulation as shown in Figure 3, we prune all (ancestor, descendant) pairs from C_2 which is generated by joining F_1 with itself.

```
insert into C2 (select I1.item1, I2.item1 from F1 I1, F1 I2
   where I1.item1 < I2.item1) except
   (select ancestor, descendant from Ancestor union
   select descendant, ancestor from Ancestor)
```

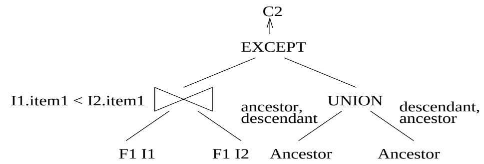


Figure 3: Generation of C_2

2.3 Support counting to find frequent itemsets

We consider two categories of SQL implementations based on SQL-92 and SQL-OR. All the SQL approaches developed for boolean associations (Sarawagi, Thomas, & Agrawal 1998) can be extended to handle taxonomies. However, we present only a few representative approaches in this paper. In particular, we consider the `KwayJoin` approach from SQL-92 and `Vertical` and `GatherJoin` from SQL-OR.

Support counting using SQL-92

K-way join In each pass k, we join the candidate itemsets C_k with k copies of an extended transaction table T^* (defined below), and follow it up with a group by on the itemsets.

Query to generate T^*

```
select item, tid from T union
select distinct A.ancestor as item, T.tid
from T, Ancestor A
where A.descendant = T.item
```

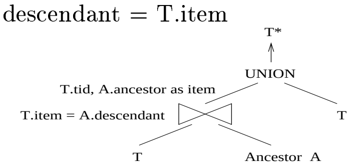


Figure 4: Transaction extension subquery

The extended transaction table T^* is obtained by augmenting T to include (*tid*, *item*) entries for all ancestors of items appearing in T. This can be formulated as a SQL query as shown in Figure 4. The select distinct clause is used to eliminate duplicate records due to extension of items with a common ancestor. Note that for this approach we do not materialize T^*. Instead we use the SQL support for common subexpressions (with construct) to pipeline the generation of T^* with the join operations (Figure 5). The pipelining idea can also be used for other queries involving "insert into" to avoid materialization.

insert into F_k with T^*(tid, item) as (Query for T^*)
select $item_1, \ldots item_k$, count(*)
from C_k, T^* t_1, $\ldots T^*$ t_k
where t_1.item $= C_k.item_1$ and ... and
  t_k.item $= C_k.item_k$ and
  t_1.tid $= t_2$.tid and ... and t_{k-1}.tid $= t_k$.tid
group by $item_1, item_2 \ldots item_k$
having count(*) > :minsup

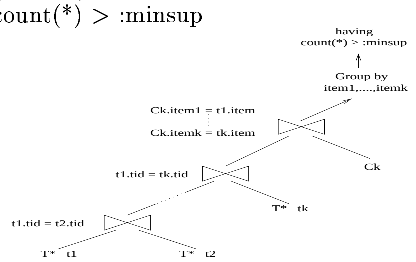


Figure 5: Support Counting by K-way join

Subquery optimization The basic KwayJoin approach can be optimized to make use of common prefixes between the itemsets in C_k by splitting the support counting phase into a cascade of k subqueries. The subqueries in this case are exactly similar to those for boolean associations presented in (Sarawagi, Thomas, & Agrawal 1998) except for the use of T^* instead of T.

Support counting using SQL-OR In this section we present two approaches that make use of the object relational features of SQL.

GatherJoin The GatherJoin approach shown in Figure 6, which is based on the use of table functions (Lohman *et al.* 1991), generates all possible k-item combinations of extended transactions, joins them with the candidate table C_k and counts the support by grouping the join result.

insert into F_k select $item_1, \ldots, item_k$, count(*)
from C_k, (select $t_2.T\_itm_1, \ldots, t_2.T\_itm_k$ from T^* t_1,
  table (GatherComb-K(t_1.tid, t_1.item)) as t_2)
where $t_2.T\_itm_1 = C_k.item_1$ and ... and
  $t_2.T\_itm_k = C_k.item_k$
group by $C_k.item_1, \ldots, C_k.item_k$
having count(*) > :minsup

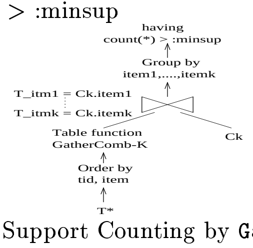


Figure 6: Support Counting by GatherJoin

The extended transactions T^* (defined in 2.3) are passed to the table function GatherComb-K in the (*tid, item*) order. A record output by the table function is a k-item combination supported by a transaction and has k attributes $T\_itm_1, \ldots, T\_itm_k$. The special optimization for pass 2 and the variations of the GatherJoin approach, namely GatherCount, GatherPrune and Horizontal (refer (Sarawagi, Thomas, & Agrawal 1998)) are also applicable here.

Vertical In this approach, the transactions are first converted into a vertical format by creating for each item a BLOB (tid-list) containing all tids that contain that item. The support for each itemset is counted by intersecting the tid-lists of all its items. The tid-list of leaf node items can be created using a table function which collects all the tid's for each item and outputs (item, tid-list) records which are stored in the table TidTable. We present two approaches for creating the tid-list of the interior nodes in the taxonomy DAG.

The first approach is based on doing a union of the descendant's tid-lists of an interior node. For every node x, table function TUnion collects the tid-lists of all the leaf nodes reachable from x, union them and outputs the tid-list for x. In this approach, tid-lists have to be created and materialized for every leaf node item irrespective of its support. This could get expensive especially when the number of items is large.

The second approach is to pass T^* (defined in 2.3) to the Gather table function as shown below which outputs tid-lists for all the items in the taxonomy.

insert into TidTable select t_2.item, t_2.tid-list
from T^* t_1, table(Gather(t_1.item, t_1.tid, :minsup)) as t_2

The SQL queries for support counting are the same as for boolean associations (Sarawagi, Thomas, & Agrawal 1998).

3 Sequential Patterns

Given a set of data-sequences each of which is a list of transactions ordered by the transaction time, the problem of mining sequential patterns is to discover all sequences with a user-specified minimum *support*. Each transaction contains a set of items. A sequential pattern is an ordered list (sequence) of itemsets. The itemsets that are contained in the sequence are called *elements* of the sequence. For example, $\langle(computer, modem)(printer)\rangle$ is a sequence with two elements – $(computer, modem)$ and $(printer)$. The support of a sequential pattern is the number of data-sequences that contain the sequence. A sequential pattern can be further qualified by specifying maximum and/or minimum time gaps between adjacent elements and a sliding time window within which items are considered part of the same sequence element. These time constraints are specified by three parameters, *max-gap, min-gap* and *window-size*.

Input format The input table D of data-sequences has three column attributes: sequence identifier (*sid*), transaction time (*time*) and item identifier (*item*). The data-sequence table contains multiple rows corresponding to different items that belong to transactions in the data-sequence.

Output format The output is a collection of frequent sequences. A sequence which is represented as a tuple in a fixed-width table consists of an ordered list of elements where each element is a set of items. The schema of the frequent sequences table is $(item_1, eno_1, \ldots, item_k, eno_k, len)$. The *len* attribute gives the length of the sequence, i.e., the total number of items in all the elements of the sequence. The *eno* attributes stores the element number of the corresponding items. For sequences of smaller length, the extra column values are set to NULL. For example, if k = 5, the sequence $\langle(computer, modem)(printer)\rangle$ is represented by the tuple *(computer, 1, modem, 1, printer, 2, NULL, NULL, 3)*.

In Section 3.1, we briefly introduce the GSP algorithm for sequential pattern mining. In Section 3.2, we present the SQL-based GSP candidate generation and in Section 3.3 we present the support counting procedure.

3.1 GSP Algorithm

The basic structure of the GSP algorithm (Srikant & Agrawal 1996) is similar to that of the Cumulate algorithm outlined in Section 2, although the specific details are quite different which are described next.

3.2 Candidate Generation

In each pass k, the candidate k-sequences C_k are generated from the frequent $(k-1)$ sequences F_{k-1}. C_k has the same schema of frequent sequences explained above, except that we do not require the *len* attribute since all the tuples in C_k have the same length k.

Candidates are generated in two steps. The *join* step generates a superset of C_k by joining F_{k-1} with itself. A sequence s_1 joins with s_2 if the subsequence obtained by dropping the first item of s_1 is the same as the one obtained by dropping the last item of s_2. This can be expressed in SQL as follows:

insert into C_k
select $I_1.item_1, I_1.eno_1, \ldots, I_1.item_{k-1}, I_1.eno_{k-1},$
    $I_2.item_{k-1}, I_1.eno_{k-1} + I_2.eno_{k-1} - I_2.eno_{k-2}$
from $F_{k-1}\ I_1, F_{k-1}\ I_2$
where $I_1.item_2 = I_2.item_1$ and ... and
    $I_1.item_{k-1} = I_2.item_{k-2}$ and
    $I_1.eno_3 - I_1.eno_2 = I_2.eno_2 - I_2.eno_1$ and ... and
    $I_1.eno_{k-1} - I_1.eno_{k-2} = I_2.eno_{k-2} - I_2.eno_{k-3}$

In the above query, subsequence matching is expressed as join predicates on the attributes of F_{k-1}. Note the special join predicates on the `eno` fields that ensure that not only do the joined sequences contain the same set of items but that these items are grouped in the same manner into elements. The result of the join is the sequence obtained by extending s_1 with the last item of s_2. The added item becomes a separate element if it was a separate element in s_2, and part of the last element of s_1 otherwise. In our representation of the candidate sequence, eno_{k-1} and eno_{k-2} determine if the added item is a separate element.

In the *prune* step, all candidate sequences that have a non-frequent contiguous $(k-1)$-subsequence are deleted. We perform both the join and prune steps in the same SQL statement by writing it as a k-way join, which is structurally similar to the candidate generation query for association rules (Sarawagi, Thomas, & Agrawal 1998). For any k-sequence there are *at most* k contiguous subsequences of length $(k-1)$ for which F_{k-1} needs to be checked for membership. Note that all $(k-1)$-subsequences may not be contiguous because of the *max-gap* constraint between consecutive elements.

While joining F_1 with itself to get C_2, we need to generate sequences where both the items appear as a single element as well as two separate elements.

3.3 Support counting to find frequent sequences

In each pass k, we use the candidate table C_k and the input data-sequences table D to count the support. We consider SQL implementations based on SQL-92 and SQL-OR.

Support counting using SQL-92

K-way join This approach is very similar to the `KwayJoin` approach for association rules (section 2.3) except for these two key differences:

1. We use select distinct before the group by to ensure that only distinct data-sequences are counted.
2. Second, we have additional predicates PRED(k) between sequence numbers. The predicates PRED(k) is a conjunct (and) of terms $p_{ij}(k)$ corresponding to each pair of items from C_k. $p_{ij}(k)$ is expressed as:

$(C_k.eno_j = C_k.eno_i$ and $\text{abs}(d_j.time - d_i.time) \leq$ window-size$)$ or $(C_k.eno_j = C_k.eno_i + 1$ and $d_j.time - d_i.time \leq$ max-gap and $d_j.time - d_i.time >$ min-gap$)$ or $(C_k.eno_j > C_k.eno_i + 1)$

Intuitively, these predicates check (a) if two items of a candidate sequence belong to the same element, then the difference of their corresponding transaction times is at most *window-size* and (b) if two items belong to adjacent elements, then their transaction times are at most *max-gap* and at least *min-gap* apart.

Subquery optimization The subquery optimization for association rules can be applied for sequential patterns also by splitting the support counting query in pass k into a cascade of k subqueries. The predicates p_{ij} can be applied either on the output of subquery Q_k or sprinkled across the different subqueries.

Support counting using SQL-OR

Vertical This approach is similar to the `Vertical` approach in Section 2.3.

insert into F_k select $t.item_1, t.eno_1, \ldots, t.item_k, t.eno_k$, cnt
from (select $item_1, eno_1, \ldots, item_k, eno_k$,
    CountIntersect-K($C_k.eno_1, \ldots, C_k.eno_k$, d_1.s-list,..., d_k.s-list, window-size,min-gap,max-gap) as cnt
    from C_k, SlistTable d_1,...,SlistTable d_k
    where d_1.item = $C_k.item_1$ and ... and
        d_k.item = $C_k.item_k$) as t
where count(*) > :minsup

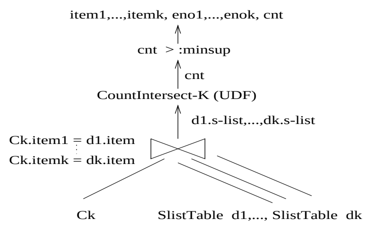


Figure 7: Support Counting by `Vertical`

For each item, we create a BLOB (s-list) containing all (*sid, time*) pairs corresponding to that item. The sequence table D is scanned in the (item, sid, time) order and passed to the table function `Gather`, which collects the (sid, time) attribute values of all tuples of D with the same item in memory and outputs a (item, s-list) pair for all the items that meet the minimum support criterion. The s-lists are maintained sorted using sid as the major key and time as the minor key, and is stored in a new SlistTable with the schema (item, s-list).

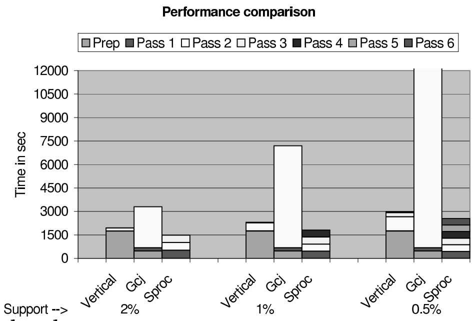
In the support counting phase, we collect the s-lists of all the items of a candidate and intersect them using a UDF `CountIntersect-K` (as shown in Figure 7) to determine the data-sequences containing that sequence. For determining whether a candidate is contained in the data-sequence, we use an algorithm similar to the one described in (Srikant & Agrawal 1996).

The intersect operation can also be decomposed to share it across sequences having common prefixes similar to the subquery optimization in the `KwayJoin` approach.

GatherJoin The `GatherJoin` approach for association rules can be extended to mine sequential patterns also. We generate all possible k-sequences using a `Gather` table function, join them with C_k and group the join result to count the support of the candidate sequences. The time constraints are checked on the table function output using join predicates PRED(k) as in the `KwayJoin` approach.

4 Performance results

In this section, we report the results of some of our performance experiments on real-life datasets. We present only one set of results of generalized association rule mining due to space limitations.



Mail order data:
Total number of records = 2.5 million
Number of transactions = 568000
Number of items = 85161 (leaf nodes in taxonomy DAG)
Total number of items = 228428 (including interior nodes)
Max. depth of the taxonomy = 7
Avg. number of children per node = 1.6
Max. number of parents = 3

Figure 8: Comparison of different SQL approaches

Our experiments were performed on Version 5 of IBM DB2 Universal Server installed on a RS/6000 Model 140 with a 200MHz CPU, 256 MB main memory and a 9 GB disk with a measured transfer rate of 8 MB/sec. Figure 8 shows the performance of three approaches – Vertical, `GatherJoin` and `Stored-procedure`. The chart shows the preprocessing time and the time taken for the different passes. For `Vertical` the preprocessing time includes ancestor pre-computation and tid-list creation times, where as for `GatherJoin` it is just the time for ancestor pre-computation. In the stored procedure approach, the mining algorithm is encapsulated as a stored procedure (Chamberlin 1996) which runs in the same address space as the DBMS. For the `Stored-procedure` experiment, we used the generalized association rule implementation provided with the IBM data mining product, Intelligent Miner (Int 1996). For all the support values, the `Vertical` approach performs equally well as the `Stored-procedure` approach and on some other datasets it performed better than the `Stored-procedure` approach. The `GatherJoin` approach is worse mainly due to the large number of item combinations generated. In the `GatherJoin` approach, the extended transactions are passed to the Gather-Comb table function and hence the effective number of items per transaction gets multiplied by the average depth of the taxonomy. In the `GatherJoin` approach, we show the performance numbers for only the second pass. Note that just the time for second pass is an order of magnitude more than the total time for all the passes of `Vertical`. The `KwayJoin` approach was an order of magnitude slower than the other approaches.

5 Conclusion and future work

We addressed the problem of mining generalized association rules and sequential patterns using SQL queries. We developed SQL formulations based on SQL-92 and SQL-OR (SQL enhanced with object relational extensions) for the two mining problems. The SQL-92 approaches use just standard SQL and are readily portable across various DBMS platforms, where as SQL-OR leverages the object relational features – some of which are not yet standardized – to achieve better performance. We also report some results of our performance experiments. This work points out that it is possible to express complex mining computations in SQL. We augmented the basic association rule framework in (Sarawagi, Thomas, & Agrawal 1998) to implement generalized association rule and sequential pattern mining. The major addition for generalized association rule was to "extend" the input transaction table (transform T to T^*). For sequential patterns, the join predicates for candidate generation and support counting were significantly different. We plan to do more experimentation on different datasets to consolidate the experiences from this work.

Acknowledgements We wish to thank Rakesh Agrawal and Ramakrishnan Srikant for useful discussions.

References

Agrawal, R., and Shim, K. 1996. Developing tightly-coupled data mining applications on a relational database system. In *Proc. of KDD*.

Agrawal, R.; Mannila, H.; Srikant, R.; Toivonen, H.; and Verkamo, A. I. 1996. Fast Discovery of Association Rules. In *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press. chapter 12, 307–328.

Chamberlin, D. 1996. *Using the New DB2: IBM's Object-Relational Database System*. Morgan Kaufmann.

Internationl Business Machines. 1996. *IBM Intelligent Miner User's Guide*, Version 1 Release 1, SH12-6213-00 edition.

Lohman, G.; Lindsay, B.; Pirahesh, H.; and Schiefer, K. B. 1991. Extensions to starburst: Objects, types, functions, and rules. *Communications of the ACM* 34(10).

Sarawagi, S.; Thomas, S.; and Agrawal, R. 1998. Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications. In *Proc. of the ACM SIGMOD Conference on Management of Data*.

Srikant, R., and Agrawal, R. 1995. Mining Generalized Association Rules. In *Proc. of the 21st Int'l Conference on Very Large Databases*.

Srikant, R., and Agrawal, R. 1996. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proc. of the Fifth Int'l Conference on Extending Database Technology*.