Mining Association Rules in Entity-Relationship Modeled Databases

Laurentiu Cristofor, Dan Simovici

University of Massachusetts at Boston,
Department of Mathematics and Computer Science,
Boston, Massachusetts 02446, USA,
{laur,dsim}@cs.umb.edu

Abstract. Current data mining algorithms handle databases consisting of a single table. This paper addresses the problem of mining association rules in databases consisting of multiple tables and designed using the entity-relationship model. We discuss previous approaches to this problem and point out some unaddressed issues, and we present a couple of algorithms to address these issues and experimental results showing the scalability of these algorithms with respect to the increase in size of the database. The paper concludes with a discussion of the possibility of extending our algorithms to database schemas more complex than a star schema.

Keywords: entity-relationship model, itemset, association rule, entity support, join support

1 Introduction

Finding association rules in databases is a central problem in data mining and has been extensively studied since its introduction in [AIS93]. The resulting algorithms ([AMS+96] , [SON95], [BMUT97], [CCS00], [ZH99], [AAP00a], [AAP00b], [HPY00]) shared the assumption that the data to mine was placed in a single table. There are very few published results on how to mine association rules when data resides in more than one table. An instance of this problem has been addressed in a machine learning setting by L. Dehaspe and L. De Raedt [DR97] for the special case of mining a deductive relational database containing knowledge about some type of entity (for example entities could be kids or sentences) and several weak entity sets dependent on this set of entities. However, the work in [DR97] does not analyze how mining should be performed in databases involving relationships between multiple entities as is usually the case in a relational database. Recently, V. Jensen and N. Soparkar [JS00] have addressed this problem for the special and important case when the database is organized in a star schema. They have proposed replacing the application of the Apriori [AMS+96] algorithm on a table obtained by joining all schema's tables by a technique (which we will call the `JS` algorithm) that in a first stage looks for frequent itemsets in each separate table using a slightly modified version of Apriori, and then, in a second stage, finds all frequent itemsets whose items belong to distinct tables. Finally, this method would yield the same results as those yield by Apriori when executed on the joined schema tables and would have better performance.

In this paper we show that both methods, the Apriori algorithm and the JS algorithm, can produce rules which may not reflect accurately the actual relationships existing in the data in cases like the one used as an example in [JS00] and we investigate how association rule mining should be performed in those cases.

The paper is organized as follows: section 2 presents some examples to illustrate the problems that arise when mining data stored in a star schema and section 3 discusses two algorithms for mining association rules in this case. Section 4 presents experimental results, and section 5 concludes the paper with an overview of the results presented and with some future research directions.

2 Problems in previous approaches

We view a table as a triple $\tau = (T, H, \rho)$, where T is the name of the table, $H = A_1 \dots A_n$ is its heading and $\rho \subseteq \mathsf{Dom}(A_1) \times \cdots \times \mathsf{Dom}(A_n)$ is the content of τ. The reader is referred to [ST95] for the relational terminology used here.

The star database design that we consider here is derived from an entity relationship design that involves k entity sets and a set of k-ary relationships R that involve the instances of entity sets $E_1, \dots, E_k$. Tables that represent entity sets are referred to as *entity tables*, while tables that represent relationship sets are called *relationship tables*. The tables are denoted by the same letter as their respective set. We denote the attributes of an entity set E by $\mathbf{Attr}(E)$.

To simplify our definitions and discussions we will assume that R has no other attributes than the foreign keys for the entity sets $E_1, \dots, E_k$. Note that this is not a restrictive assumption since in the case the relationship set R would contain other attributes than foreign keys we could simply consider them as being attributes of an extra entity set E_{k+1} whose instances participate exactly once in a relationship from R.

Another assumption corresponding to imposing referential integrity is that each relationship of R must involve an entity from each entity set $E_1, \dots, E_k$.

Let $\mathcal{E} = \{E_1, \dots, E_k\}$ be the collection of k sets of entities, R be a set of k-ary relationships between $E_1, \dots, E_k$, *Join* be the join of tables $R, E_1, \dots, E_k$, and let *OuterJoin* be the outer join of tables $R, E_1, \dots, E_k$. We use these notations throughout the paper when discussing star schemas.

Definition 21 An *item* is a pair $\langle A, a \rangle$, where A is an attribute and a is a value in the domain of A. An *itemset* is a set of items $\{\langle A_1, a_1 \rangle \dots \langle A_n, a_n \rangle\}$ such that if $i \neq j$, then $A_i \neq A_j$ for any $i, j \in \{1, \dots, n\}$. In other words, an itemset contains no two pairs of items with an identical attribute component.

The *support* of an itemset $I = \{\langle A_1, a_1 \rangle, \dots, \langle A_n, a_n \rangle\}$ with respect to a table $\tau = (T, H, \rho)$ is $\mathsf{supp}(I) = |\{t \in \rho \mid t[A_i] = a_i \text{ for } i \in \{1, \dots, n\}\}|/|\rho|$.

An *association rule* is an ordered pair $\langle a, c\rangle$ of itemsets such that they do not contain items with an identical attribute component. The first itemset in the pair is called *antecedent* and the second is called *consequent*. We usually write an association rule as $a \rightarrow c$.

The *support* of an association rule is defined to be the support of the union of its antecedent and consequent itemsets. The *confidence* of an association rule is defined to be the ratio between the support of the rule and the support of the antecedent. □

To illustrate our arguments we use an example similar to the one presented in [JS00], with the difference that the data was changed to make some points easier to observe.

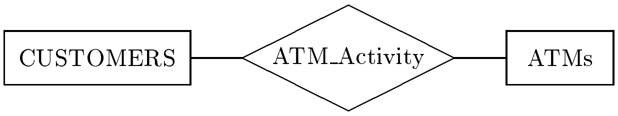

Fig. 1. Entity-Relationship Diagram of Bank Database

The example is that of a simplified banking environment where we have a simple star schema (see Figure 1) consisting of two tables that reflect entity sets: the *Customers* table with attributes {acct#, balance, age}, the *ATMs* table with attributes {atm#, type, limit}, and a table that represents a set of binary relationships named *ATM_Activity* with attributes {acct#, atm#, amount}.

Customers

acct#	balance	age
1	50000-100000	30-40
2	1000-5000	30-40
3	100-1000	20-30

ATM_Activity

acct#	atm#	amount
1	1	500-1000
2	2	50-100
1	2	50-100
3	2	0-50
1	1	500-1000

ATMs

atm#	type	limit
1	in	1000
2	out	500

The table *Join* obtained from joining the three tables is identical to the table we would obtain by performing an outer join and is presented below.

Join

acct#	atm#	amount	balance	age	type	limit
1	1	500-1000	50000-100000	30-40	in	1000
2	2	50-100	1000-5000	30-40	out	500
1	2	50-100	50000-100000	30-40	out	500
3	2	0-50	100-1000	20-30	out	500
1	1	500-1000	50000-100000	30-40	in	1000

We question whether an algorithm that mines this star schema database for association rules is useful when it finds exactly the same results obtained when running the Apriori algorithm on the joined tables. We claim that the knowledge about the entities and relationships existing in a database can be used in order to produce more meaningful rules and we will next look at some examples that illustrate this idea.

Let us consider the rule $age = 30 - 40 \rightarrow balance = 50000 - 100000$ which involves attributes *age* and *balance* belonging to entity table *Customers*. If we consider the *Join* table, then the support of this rule would be 60% (3/5) and its confidence would be 75% (3/4). However, another approach would be to examine only the *Customers* table, in which case the support of the rule would be 33.3% (1/3) and its confidence would be 50% (1/2). A question now arises about which of these two results is the correct one. Since the rule involves only attributes of the *Customers* table we consider that the support and confidence of the rule should be based on that table only, such that we should obtain the second result. This is because *Customers* represents an entity set and properties of its attributes should be determined only by looking at the set of instances of customers. However, when run on the joined tables, Apriori would generate the first result thus producing what we consider to be an association rule with misleading support and/or confidence values.

It is also worth mentioning that because the support of a set of attribute values corresponding to one entity can be smaller with respect to the joined table than with respect to the entity table, the itemset may not even be discovered by Apriori, much less be used to generate an association rule. For example, the support of *age=20-30* is 20% (1/5) with respect to *Join* while with respect to table *Customers* it is 33.33% (1/3).

Let us now consider another example in the rule $age = 30 - 40 \rightarrow type = in$. In this case the rule contains attribute *age* from entity *Customers* and attribute *type* from entity *ATMs*. Since the rule contains attributes from two different entities and these entities are related through relationship *ATM_Activity* it makes sense to compute the support and confidence of this relation with respect to the table obtained by joining *Customers*, *ATM_Activity*, and *ATMs* which in our case is equivalent to using the table *Join*. By looking at the *Join* table we can compute the support for this rule as 40% (2/5) and its confidence as 50% (2/4).

From these examples we can draw several conclusions:

1. Running an Apriori algorithm on the join of the tables of a database can fail to produce all existing rules or may produce rules whose parameters do not properly reflect the knowledge embedded in the data.
2. The entity-relationship model of a database provides important information concerning the relations between entities and this information should be used by data mining algorithms.
3. When looking for association rules, rules among attributes of the same entity should be analyzed with respect to the set of that entity's instances. When looking for association rules among attributes of several entities, we need to look at how these attributes appear together so we need to analyze rules with respect to the relationships existing between the entities. Note that if several relationships exist between two or more entities, then the association rules between their attributes must be examined with respect to each such relationship. We discuss this issue in Section 5.

We have thus identified a problem that appears when mining a database built from an entity-relationship model using standard mining algorithms. New algorithms are needed to address this problem and in the next sections we discuss such algorithms. Initially, we address the basic case of star schemas and then we discuss the case of more complex schemas.

3 Mining association rules in star schemas

We examine extending the Apriori algorithm to make it work on the join of all tables and, then, we investigate a method for mining association rules without joining the tables, assuming a star schema organization of the database tables.

Definition 31 An *entity itemset* is an itemset $\{\langle A_1, a_1\rangle, \ldots, \langle A_n, a_n\rangle\}$, such that $\{A_i \mid i \in \{1, \ldots, n\}\} \subseteq \mathbf{Attr}(E)$ for some set of entities $E \in \mathcal{E}$.

A *join itemset* is an itemset $I = \{\langle A_1, a_1\rangle, \ldots, \langle A_n, a_n\rangle\}$ such that $\bigcup_{i=1}^{n} A_i \subseteq \mathbf{Attr}(R) \cup \bigcup_{j=1}^{k} \mathbf{Attr}(E_j)$ and such that I is not an entity itemset. In other words, a join itemset is an itemset whose attributes do not belong to the same entity.

The support of an entity itemset with respect to its entity table is called *entity support*.

The support of an (entity or join) itemset with respect to the table *Join* is called *join support*. □

Note that we can compute the join support for any itemset but the entity support is defined only for entity itemsets.

It is impossible to predict the relative magnitude of the join support and entity support for an itemset. The entity support may be greater or smaller than the join support. In the example presented in the second section we have seen that the join support of itemset $\{age = 30 - 40, balance = 50000 - 100000\}$ (60%) was greater than its entity support (33.3%). On the other hand the join support of itemset $\{age = 20 - 30\}$ (20%) was smaller than its entity support (33.3%). This means that in the execution of an Apriori algorithm, for an entity itemset candidate we should compute both its entity support and its join support. If the entity support is greater than minimum support then the itemset should be considered frequent; otherwise, if the join support is greater than minimum support then the itemset is not frequent but it might be a subset of a frequent join itemset and thus should be used in the candidate generation procedure. Finally, if both supports are smaller than minimum support then the itemset should be discarded.

We propose to modify the Apriori algorithm as follows: whenever we compute and evaluate the support of a candidate itemset we consider two possibilities:

1. If the itemset is an entity itemset then both its join and entity supports should be computed. The itemset should be considered frequent if its entity support is greater than minimum support; otherwise, if its join support is greater than minimum support, then the itemset should be used in generating new candidate itemsets for the next step of the algorithm. If none of these conditions is met the itemset should be discarded.
2. If the itemset is a join itemset, then its join support should be computed and checked against the minimum support to determine whether the itemset is frequent; otherwise, the itemset should be discarded.

In the *apriori_gen* procedure, which combines two itemsets of length k into a candidate itemset of length $k+1$, we would again have two possible cases:

1. If the resulting candidate itemset is an entity itemset, then we need to check that all its subsets have entity support greater than minimum support or that all its subsets have a join support greater than minimum support (so that it can be used later for the generation of a join candidate itemset); otherwise, the candidate should not be generated.
2. If the resulting candidate itemset is a join itemset, then we should just verify that all its subsets have join support greater than minimum support, otherwise we should not generate the candidate.

Next, we discuss how these changes can be used to obtain an algorithm for mining the joined tables and an algorithm for mining the star schema.

3.1 Modifying Apriori for mining joined tables

In this section we want to devise an algorithm capable to mine a table containing all the data existing in a star schema. Note that such a table can be obtained by performing an outer join on all tables of the star schema. A simple join would not be sufficient since then we could miss entity instances that do not participate in the relationship.

In order for Apriori to generate rules with proper support and confidence values it has to know about the existing entities that appear in the table on which it is run. This can be done by keeping track for each attribute of the key attribute of the entity to which it belongs. We focus on the problem of finding large itemsets since from these we can construct association rules by applying standard algorithms. The algorithm described in this section is named `AprioriJoin`; we assume that its input table is the *OuterJoin* table.

To compute the support of an itemset we need to differentiate between entity itemsets and join itemsets.

For an entity itemset we compute its entity support as follows. We first count the number of rows of *OuterJoin* that contain the itemset and that have a distinct entity key. Then we divide this count by the number of distinct entity keys to find the percentage of the entity support. To compute the join support we have to count the number of rows that contain the itemset and that belong to *Join* but not to *OuterJoin*, and then divide this value by the cardinality of *Join*. Identifying the rows which belong to *OuterJoin* but not to *Join* is quite simple: these are the rows that contain null entity keys. The rows of *Join* are then identified as those that do not contain null entity keys.

For a join itemset we compute its join support by simply counting the rows that contain the itemset and then dividing this value by the cardinality of *Join*.

There are few implementation difficulties in adapting an Apriori algorithm to mine the *OuterJoin* table. Below we give a description of the main steps of the resulting algorithm:

Algorithm AprioriJoin

Input: A minimum support value, the table **OuterJoin** resulted from outer joining all tables of a star schema database, and the mapping of each attribute to an entity key.

Output: The list **AllFrequent** of frequent itemsets.

1. Initialize the **Candidate** collection to all 1-itemsets.
2. Scan table *OuterJoin* and compute for all candidate itemsets the join support and, if appropriate, the entity support also.
3. Place into collection **Frequent** all itemsets from **Candidate** that have entity support or join support greater than minimum support. Place into collection **AllFrequent** the entity itemsets that have entity support greater than minimum support and the join itemsets that have join support greater than minimum support.
4. From **Frequent** generate new candidates using the *apriori_gen* method modified as described at the beginning of this section. Place all newly generated itemsets into **Candidate**.
5. If **Candidate** is empty exit.
6. Go to step 2

3.2 An algorithm for mining a star schema

In this section we present an algorithm for correctly finding the frequent itemsets in a set of tables organized in a star schema. Supports are also expressed as percentages as in the previous algorithm.

We call this algorithm `AprioriStar` since it is a modification of the classic algorithm Apriori made to be used on a star-schema database. A description of its main steps is presented below.

Algorithm AprioriStar

Input: A minimum support value and a star schema database with entity tables $E_1, \ldots, E_n$ and relationship table R.

Output: The list **AllFrequent** of frequent itemsets.

1. Scan R and store the number of occurrences of every value of each foreign key (a step identical to the one in the `JS` algorithm).
2. Initialize the **Candidate** collection to all 1-itemsets (note also that these are entity itemsets)
3. Scan tables $E_1, \ldots, E_n$ and during the scan of one table E_i compute the entity support for all itemsets from E_i while also computing their join support using the results obtained at step 1 (whenever we see a row containing the itemset, the support of the itemset is increased with the number of occurrences of the row's key value in the R table).
4. Place into collection **Frequent** all itemsets from **Candidate** that have entity support or join support greater than minimum support. Place into collection **AllFrequent** the entity itemsets that have entity support greater than minimum support and the join itemsets that have join support greater than minimum support.
5. From **Frequent** generate new candidates using the *apriori_gen* method modified as described in the beginning of this section. Place all newly generated itemsets into **Candidate**.
6. If **Candidate** is empty exit.
7. For all entity itemsets in **Candidate** from a table E_i scan E_i and compute their join and entity supports.
8. Compute on the fly the join of tables $R, E_1, \ldots, E_n$ and then for all join itemsets in **Candidate** compute their join support with respect to the joined tables.
9. Go to step 4

Note that in step 8 it is not required to compute the join of all tables if the candidate itemsets do not contain attributes of all entities. For example, if the candidate itemsets would just contain attributes of entities E_1 and E_2, we would just need to join R, E_1, E_2 and we could ignore tables $E_3, \ldots, E_n$.

The main differences between this algorithm and the `JS` algorithm are:

1. `AprioriStar` uses the join and entity supports in determining frequent itemsets. By considering the entity support `AprioriStar` does not eliminate from the result entity itemsets that are frequent with respect to their entity table but not with respect to the relationship table and it also allows the computation of correct support and confidence for rules existing among attributes of the same entity table.

2. `AprioriStar` does not compute the support of the join itemsets in a single step as it was done by Jensen & Soparkar. Our approach avoids the explosion of join candidates present in the final step of the `JS` algorithm.

4 Experimental results

For our experiments we implemented in Java the algorithms `AprioriJoin` and `AprioriStar` and we executed them on binary files containing synthetically generated data. The files represented a star schema with three entity sets and a relationship set and they were indexed for the faster random access needed by the implementation of `AprioriStar`.

The test data was built as follows: first we generated synthetic data for the `AprioriStar` algorithm by creating a database with three entity sets and a relationship set and then, we outer joined these tables to obtain the table on which we executed the `AprioriJoin` algorithm. For the generation of the entity tables we have used our implementation of the synthetic data generator described by Agrawal and Srikant in [AMS+96] and for the generation of the relationship table we have used our own synthetic data generator.

The synthetic relation generator yields a number of relationships for each of the tuples of a specified entity table. The average and standard deviation for this number of relationships are specified by the user and are generated using a normal distribution. Also, for each such tuple we randomly select a sample of tuples from every other entity table that can be involved in the relationships to be generated. The size of the sample selected from each entity table is produced using a Poisson distribution with mean specified by the user. Finally, we generate relationships by randomly selecting the participating tuples from the samples previously constructed.

We first generated two types of databases, a sparse one (having a small average number of items per transaction and having fewer frequent itemsets) and a dense one (having a larger average number of items per transaction and more frequent itemsets), to which we refer from now on as databases SPARSE and DENSE.

We verified the scalability of the algorithms by doubling twice the contents of the database SPARSE, thus obtaining databases SPARSEx2 and SPARSEx4. The duplication of the database SPARSE was done in the following way: first we have duplicated the rows of all entity tables and then we have duplicated the relations and made the duplicate relations refer to the duplicated entity instances instead of the original ones. Thus, we have obtained a database whose tables had twice the size of the original table but which contained the same rules as the original database.

For both the SPARSE and DENSE databases, all the entity instances are participating in relationships from R which is not the case for a third database called OUTER, where only half of the transactions of each entity table participate in relationships of R. OUTER was obtained by generating two different sets of entity tables and then concatenating them. For this database, the relationship table R consisted of the relationship table generated for the first set of tables.

The main characteristics of these datasets are presented in the table below; for OUTER we separated by a slash the characteristics of the first and second sets of tables that were used for its generation.

Test database characteristics

Characteristics	SPARSE	DENSE	OUTER
Entity E_1			
Number transactions	10000	10000	20000
Average transaction size	7	20	7/12
Number of items	60	60	60
Number of frequent patterns	100	50	100
Average pattern length	5	10	4/10
Entity E_2			
Number transactions	100	100	200
Average transaction size	4	4	4/7
Number of items	10	10	10
Number of frequent patterns	10	10	10
Average pattern length	2	2	2/5
Entity E_3			
Number transactions	1000	1000	2000
Average transaction size	5	10	5/9
Number of items	20	20	20
Number of frequent patterns	50	20	50
Average pattern length	3	5	3/7
Relationship R			
Total number relationships	100380	95753	95985
Average and standard deviation of number of relationships for each tuple of entity one	10,8	10,5	10,5
Second entity sample mean	10	5	20
Third entity sample mean	20	20	50
Outer Join			
Number tuples	100380	95753	107085

We have executed the algorithms on a Sun Ultra-10, running Sun OS 5.7, with 512MB of memory. We used Sun's JDK 1.2.2 for compiling and running the programs (and avoided the use of the JIT compiler that is currently offered by Sun as an experimental version). The times were measured using the /usr/bin/time command.

The following table presents the results of our experiments. In parentheses we have indicated the number of frequent itemsets and the largest size of such an itemset for the respective algorithm execution. For example (13/3) means that the algorithm has found 13 frequent itemsets and the largest one consisted of 3 items. Since SPARSEx2 and SPARSEx4 contain the same patterns as SPARSE, we have not duplicated this information in their case and we have only displayed the times taken by the algorithms.

Test results

Minimum support	SPARSE	SPARSEx2	SPARSEx4	DENSE	OUTER
AprioriJoin results					
0.5	5m01s (13/3)	10m02s	20m23s	23m19s (912/7)	7m39s (26/3)
0.4	6m47s (38/4)	13m14s	27m23s	33m52s (3275/8)	7m58s (77/4)
0.3	6m59s (100/4)	13m54s	28m04s	1h11m21s (14788/10)	10m18s (217/5)
0.2	9m15s (340/5)	18m26s	37m28s	4h38m15s (106219/11)	13m31s (878/6)
0.1	17m06s (2145/7)	34m10s	1h08m49s	-	24m56s (5968/9)
AprioriStar results					
0.5	5m27s (13/3)	11m02s	20m44s	24m45s (912/7)	5m12s (26/3)
0.4	7m50s (38/4)	15m28s	30m45s	35m22s (3275/8)	7m40s (77/4)
0.3	7m59s (100/4)	15m38s	31m33s	1h14m15s (14788/10)	10m12s (217/5)
0.2	11m00s (340/5)	21m39s	43m38s	4h28m51s (106219/11)	13m30s (878/6)
0.1	21m00s (2145/7)	41m30s	1h22m31s	-	22m50s (5968/9)

There are a couple of important observations to be made from the test results. First, it can be observed that the algorithms scale linearly as the number of tuples of the database increases. With respect to the relative performance of the algorithms we can note that the `AprioriJoin` algorithm is almost always a little faster than `AprioriStar` when working on databases where all entity instances are participating to the relation R. This isn't surprising given that on such databases `AprioriStar` would find frequent join itemsets at each step and thus would have to build the join of the tables as often as `AprioriJoin` would perform its scan which means that `AprioriStar` would end up doing more I/O operations than `AprioriJoin` since it also has to scan the entity tables in a separate step. However, in the case of database OUTER, we can notice that `AprioriStar` outperforms in all cases `AprioriJoin`. This is due to the combination of two factors. The first factor is that in database OUTER at one point the algorithm generates only entity candidates, which in turn means that `AprioriStar` will avoid performing a scan of the join of the tables and will just scan some of the entity tables. This is the main reason for the more significant difference obtained in the experiment using a minimum support value of 0.1. The second factor is that `AprioriJoin` works on a table that is obtained from an outer join and that is thus larger than the join table built on the fly by `AprioriStar`, which makes the I/O requirements of the algorithms to be comparable, hence the rather small difference in performance for the minimum support values 0.2-0.5.

We also expect that for more complex schemas than star schemas, algorithms that mine separate tables would prove more efficient than algorithms that would work on a single table, but more work is needed in order to verify this assumption.

5 Conclusions and Future Work

In this paper we approached the problem of mining association rules in databases consisting of several tables organized in a schema obtained from an entity-relationship design. We have focused mainly on the problem of mining a star schema database. We noticed that previous algorithms did not take advantage of the knowledge already embedded in an entity relationship model regarding the relationships between the database entities. To address this issue we introduced the notions of entity and join support and we presented two algorithms: algorithm `AprioriJoin`, for mining the outer join of a star schema tables using knowledge of the schema, and algorithm `AprioriStar`, for directly mining the star schema database. These algorithms were tested on synthetically generated databases and the results of our tests show that both algorithms scale linearly with respect to the size of the input database. The results also show that in the case of a star schema there is no clear winner between the two algorithms in terms of time performance.

To handle more complex database designs we start from the fact that the entity-relationship diagram of a database $\mathcal{D}$ is a bipartite, connected graph $\mathcal{G}_{\mathcal{D}}$ whose set of nodes is partitioned into sets of entities and sets of relationships. Let $\mathcal{E}$ be the collection of sets of entities and let $\mathcal{R}$ be the collection of the sets of relationships involved in the model. For a set of attributes X let $\mathcal{E}(X)$ be the set of all entity sets in $\mathcal{E}$ that contain some attributes in X. Also, for a collection of entity sets $\mathcal{F}$, where $\mathcal{F} \subseteq \mathcal{E}$, denote by $M(\mathcal{F})$ the collection of all minimal sets of relationships $\mathcal{S}$ such that the subgraph of $\mathcal{G}_{\mathcal{D}}$ generated by $\mathcal{F} \cup \mathcal{S}$ is connected. For example, if $\mathcal{F} = \{E_1, E_2, E_3\}$ is a collection of sets of entities (shown in Figure 2), then $M(\mathcal{F})$ consists of $\{\{R_1\}, \{R_2, R_3\}\}$. We refer to these members of $M(\mathcal{F})$ as *connectors* of $\mathcal{F}$.

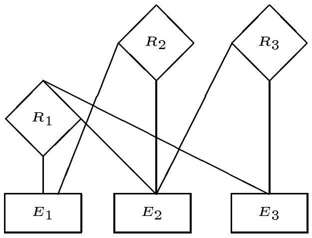

Fig. 2. Entity-Relationship Diagram

For a set of attributes X and every connector $\mathcal{Q}$ of $M(\mathcal{E}(X))$ we consider the subgraph of the entity-relationship diagram determined by $\mathcal{E}(X) \cup Q$. Each such subgraph is a union of "star" graphs and one can define the support of X relative to the join of the tables that represent the sets of entities and the set of relationships that correspond to the vertices of the graph.

Consider, for example, a small database for a university having two entities: *Professors* and *Students* whose entity-relationship diagram is shown in Figure 3.

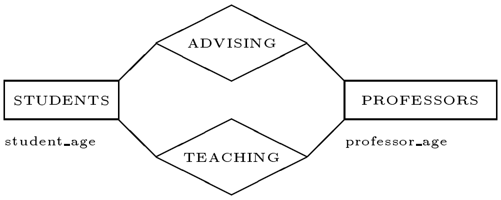

Fig. 3. Entity-Relationship Diagram of University Database

There are two relationships between these two entities: *ADVISING* and *TEACHING*. For the set of attributes $X =$ *student_age professor_age* we have

$$\mathcal{E}(X) = \mathit{STUDENTS\ PROFESSORS},$$
$$M(\mathcal{E}(X)) = \{\{ADVISING\}, \{TEACHING\}\}.$$

Thus, for a rule like $student\_age > 30 \rightarrow professor\_age > 40$, one could contemplate the support relative to the table obtained by joining *ADVISING*, *STU-*

DENTS, and *PROFESSORS*, or to the table obtained by joining *TEACHING*, *STUDENTS*, and *PROFESSORS*.

In our future research we intend to investigate efficient algorithms for computing the support of itemsets relative to various connectors that may exist for a set of attributes and to examine the effect of the topology of the entity-relationship diagram on the resources required for these computations.

References

[AAP00a] R.C. Agarwal, C.C. Aggarwal, and V.V.V. Prasad. Depth first generation of long patterns. *Proceedings of the 6th ACM-SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 108–118, 2000.

[AAP00b] R.C. Agarwal, C.C. Aggarwal, and V.V.V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Journal of Parallel and Distributed Computing, Special Issue on High Performance Data Mining*, 2000.

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1993.

[AMS+96] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, Menlo Park, 1996.

[BMUT97] Sergey Brin, Rajeev Motwani, Jeffrey Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 255–264, 1997.

[CCS00] D. Cristofor, L. Cristofor, and D. Simovici. Galois connections and data mining. *Journal of Universal Computer Science*, 2000.

[DR97] Luc Dehaspe and Luc De Raedt. Mining association rules in multiple relations. *Proceedings of the 7th International Workshop on Inductive Logic Programming*, pages 125–132, 1997.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 1–12, 2000.

[JS00] Viviane Crestana Jensen and Nandit Soparkar. Frequent itemset counting across multiple tables. *Proceedings of PAKDD*, pages 49–61, 2000.

[SON95] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. *Proceedings of the 21st International Conference on Very Large Databases*, 1995.

[ST95] Dan A. Simovici and Richard L. Tenney. *Relational Database Systems*. Academic Press, New York, 1995.

[ZH99] Mohammed Zaki and Ching-Jui Hsiao. Charm: An efficient algrithm for closed association rule mining. Technical Report 10, Renssaeler Polytechnic Institute, Troy, New York, 1999.