Mining Association Rules of Simple Conjunctive Queries

Bart Goethals Wim Le Page
University of Antwerp,
Belgium

Heikki Mannila
HIIT, Helsinki University of Technology
University of Helsinki

Abstract

We present an algorithm for mining association rules in arbitrary relational databases. We define association rules over a simple, but appealing subclass of conjunctive queries, and show that many interesting patterns can be found. We propose an efficient algorithm and a database-oriented implementation in SQL, together with several promising and convincing experimental results.

1 Introduction

The discovery of recurring patterns in databases is one of the main topics in data mining and many efficient solutions have been developed for relatively simple classes of patterns and data collections. Indeed, most frequent pattern mining or association rule mining algorithms work on so called transaction databases [2]. Not only for itemsets, but also for more complex patterns such as trees [20], graphs [13, 15, 19], or arbitrary relational structures [8], databases consisting of a set of transactions are used. For example, in the tree case [20], every transaction in the database contains a tree, and the presented algorithm tries to find all frequent subtrees occurring within all such transactions. For all these pattern classes, specialized algorithms exist to discover them efficiently. The motivation for these works is the potentially high business value of the discovered patterns [8].

Unfortunately, many relational databases are not suited to be converted into a transactional format and even if this would be possible, a lot of information implicitly encoded in the relational model would be lost after conversion. In this paper, we consider association rule mining on arbitrary relational databases by combining pairs of queries which could reveal interesting properties in the database. Intuitively, we pose two queries on the database such that the second query is more specific than the first query. Then, if the number of tuples in the output of both queries is almost the same, this could reveal a potentially interesting discovery.

To illustrate, consider the well known Internet Movie Database [12] containing almost all possible information about movies, actors and everything related to that, and consider the following queries: first, we ask for all actors that have starred in a movie of the genre 'drama'; then, we ask for all actors that have starred in a movie of the genre 'drama', but that also starred in a (possibly different) movie of the genre 'comedy'. Now suppose the answer to the first query consists of 1000 actors, and the answer to the second query consists of 900 actors. Obviously, these answers do not necessarily reveal any significant insights on themselves, but when combined, it reveals the potentially interesting pattern that actors starring in 'drama' movies typically (with a probability of 90%) also star in a 'comedy' movie. Of course, this pattern could also have been found by first preprocessing the database, and creating a transaction for each actor containing the set of all genres of movies he or she appeared in. Similarly, a pattern like: 77% of the movies starring Ben Affleck, also star Matt Damon, could be found by posing the query asking for all movies starring Ben Affleck, and the query asking for all movies starring both Ben Affleck and Matt Damon. Again, this could also be found using frequent set mining methods, but this time, the database should have been differently preprocessed in order to find this pattern. Furthermore, it is even impossible to preprocess the database only once in such a way that the above two patterns would be found by frequent set mining as they are essentially counting a different type of transactions. Indeed, we are counting actors in the first example, and movies in the second example.

Also truly relational patterns can be found which can not be found using typical set mining techniques, such as, 80% of all movie directors that have ever been an actor in some movie, also star in at least one of the movies they directed themselves. This can be easily expressed by two simple queries of which one asks for all movie directors that have every acted, and the second one asks for all movie directors that have ever acted in one of their own movies.

In general, we are looking for pairs of queries Q_1, Q_2, such that Q_1 asks for a set of tuples satisfying a certain condition and Q_2 asks for those tuples satisfying a more specific condition. When it turns out that the size of the output Q_2 is close to the size of the output of Q_1, we learned that most of the tuples in the output of Q_1 actually satisfy a more specific condition, as specified in Q_2. Clearly, such findings could reveal interesting patterns in the given database. Towards this goal, we consider a new pattern class consisting of conjunctive queries over relational databases

and define associations using the well known notion of query containment [1, 7].

The organization of the rest of the paper is as follows. In Section 2, we give a formal description of the model and the considered problem. In Section 3, we present our algorithm for mining association rules. In Section 4, we present a technique to remove a substantial amount of redundancies within the discovered associations. In Section 5, we present experimental results showing the effectiveness of our algorithm, based on applying this algorithm to data from two real databases. In Section 6, we discuss related work. Section 7 gives an overview of our future work and we conclude with a summary in Section 8.

2 Formal Model

Assume we are given a relational database consisting of a schema $\mathbf{R}(R_1, \ldots, R_n)$ and an instance $\mathbf{I}$ of $\mathbf{R}$.

DEFINITION 1. **(Simple Conjunctive Query)** *A* simple conjunctive query Q *over* $\mathbf{R}$ *is a relational algebra expression of the form*

$$\pi_{\mathrm{X}}\sigma_F(R_1 \times \cdots \times R_n),$$

with X a set of attributes from $R_1, \ldots, R_n$, and F a conjunction of equalities of the form $R_i.A = R_j.B$ or $R_k.A = c$, with $1 \leq i \leq j \leq n$, $1 \leq k \leq n$, A and B attributes from R_i and R_j respectively, and c a constant from the domain of $R_k.A$.

The only simplification, although drastic, over general conjunctive queries, is that every relation from $\mathbf{R}$ occurs exactly once in a simple conjunctive query. Essentially, we could loosen the definition to allow every relation at most once, but this would unnecessarily complicate matters. After all, under the assumption that all relations are non-empty, both definitions are equivalent.

Throughout this paper, we use the set semantics for all queries, and hence, duplicates are not taken into account and are removed from the output of all queries. As will soon be shown, this choice is important in order to differentiate between queries having a different set of projected attributes.

Although such simple conjunctive queries can hardly be interesting on themselves, they can be very interesting when two queries are compared to each other.

EXAMPLE 1. *Consider the following two queries.*

$$Q_1 : \pi_{\mathrm{A,B}}R$$
$$Q_2 : \pi_{\mathrm{A,B}}\sigma_{\mathrm{A=B}}R$$

Now suppose Q_1 returns 100 tuples, and Q_2 returns 90 tuples. Suddenly, these queries become potentially interesting as they represent the pattern that for 90% of the tuples in Q_1, the value of attribute A equals the value of attribute B.

In general, we are interested in association rules between pairs of simple conjunctive queries Q_1 and Q_2 stating that a tuple in the result of Q_1 is also in the result of Q_2 with a given probability p, denoted by $Q_1 \Rightarrow Q_2$.

Without loss of generality, we only need to consider those pairs Q_1 and Q_2 such that Q_2 is *contained* in Q_1. Therefore, we recall the definition of query containment [1, 7].

DEFINITION 2. **(Containment)** *For two conjunctive queries Q_1 and Q_2 over $\mathbf{R}$, we write $Q_1 \subseteq Q_2$ if for every possible instance $\mathbf{I}$ of $\mathbf{R}$, $Q_1(\mathbf{I}) \subseteq Q_2(\mathbf{I})$ and say that Q_1 is* contained *in Q_2. Q_1 and Q_2 are called* equivalent *if and only if $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$.*

Although deciding containment for general conjunctive queries is known to be NP-complete, it can be easily verified in linear time for the simple conjunctive queries defined here.

Besides this classical form of containment, also a second containment relationship between conjunctive queries can be identified.

EXAMPLE 2. *Consider the following two queries.*

$$Q_1 : \pi_{\mathrm{A,B}}R$$
$$Q_2 : \pi_{\mathrm{A}}R$$

Now suppose Q_1 returns 100 tuples, and Q_2 returns 90 tuples (due to duplicate elimination in the set semantics). Then, again, these queries become potentially interesting as they represent the pattern that for 90% of the (unique) tuples in Q_1, have a unique value for attribute A. Note however, we do not know if the remaining 10% have the same value for A, or whether they all have different values also occurring in the other 90%.

As illustrated in this example, there exists a second type of containment, which we will call *vertical* containment. Indeed, when projecting on a subset of the attributes in the projection of a query, the result is *vertically* contained in the result of the original query. As we will show later, also other interesting, and even well known associations can be found using this *vertical* containment.

When both types of containment are combined, we arrive at our formal containment definition which we will use throughout this paper.

DEFINITION 3. *A conjunctive query Q_1 is* diagonally contained *in Q_2 if Q_1 is contained in a projection of Q_2 ($Q_1 \subseteq \pi_X Q_2$). We write $Q_1 \subseteq^{\Delta} Q_2$.*

Finally, we are ready to formally define association rules over simple conjunctive queries.

DEFINITION 4. **(Association Rule)** *An* association rule *is of the form $Q_1 \Rightarrow Q_2$, such that Q_1 and Q_2 are both simple conjunctive queries and $Q_2 \subseteq^{\Delta} Q_1$.*

Obviously, we are not interested in all possible association rules, but only in those that satisfy certain requirements. The first requirement states that an association rule should be *supported* by a minimum amount of tuples in the database.

DEFINITION 5. **(Support)** *The* support *of a conjunctive query Q in an instance $\mathbf{I}$ is the number of distinct tuples in the answer of Q on $\mathbf{I}$. A query is said to be* frequent *in $\mathbf{I}$ if its support exceeds a given* minimal support threshold. *The support of an association rule $Q_1 \Rightarrow Q_2$ in $\mathbf{I}$ is the support of Q_2 in $\mathbf{I}$, an association rule is called* frequent *in $\mathbf{I}$ if Q_2 is frequent in $\mathbf{I}$.*

DEFINITION 6. **(Confidence)** *An association rule $Q_1 \Rightarrow Q_2$ is said to be* confident *if the support of Q_2 divided by the support of Q_1 exceeds a given* minimal confidence *threshold.*

Even if we only consider association rules over frequent simple conjunctive queries, it might still result in approximately all possible association rules, due to cartesian products.

EXAMPLE 3. *Consider the following simple conjunctive queries.*

$$Q_1 : \pi_{\mathrm{R}_1.\mathrm{A},\mathrm{R}_2.\mathrm{B}} R_1 \times R_2$$
$$Q_2 : \pi_{\mathrm{R}_1.\mathrm{A}} \sigma_{\mathrm{R}_2.\mathrm{B}='c'} R_1 \times R_2$$

As can be seen, Q_1 is a simple cartesian product of two attributes from different relations, and hence, its support is simply the product of the number of unique values of the two attributes. Now, whenever a simple conjunctive query is not frequent, there most probably exist also several other versions of that query including a cartesian product with another attribute which contains enough tuples to make the product exceed the minimum support threshold.

Similarly, also almost every frequent query will be duplicated many times due to cartesian products. This case is illustrated by Q_2. Here, the output equals $\pi_{R_1.A}$ only if there exists a tuple in R_2 containing the value $'c'$ for attribute B. Obviously, almost every frequent query could be combined like that with every possible value in the database.

As illustrated in this example, it is of no use to consider queries containing cartesian products.

Similarly, it does not make any sense to compare attributes of incomparable types, or also, the user might see no use in comparing addresses with names. Therefore, we allow the user to provide the most specific selection to be considered. That is, a partition of all attributes, such that only the attributes in the same block are allowed to be compared to each other.

The goal is now to find, for a given database, all frequent and confident association rules over, cartesian product free, simple conjunctive queries with a given minimum support threshold, a minimum confidence threshold, and a most specific selection.

3 Conqueror: the algorithm

The proposed algorithm, Conqueror (**Con**junctive **Quer**y Generat**or**), is divided into two phases. In a first phase, all frequent simple conjunctive queries are generated. Then, in a second phase, all confident association rules over these frequent queries are generated. Essentially, the second phase is considerably easier, as most computationally intensive work is performed in the first phase. After all, for every generated simple conjunctive query, we need to compute its support in the database, while association rule generation merely needs to find couples of previously generated queries and compute their confidence. Consequently, most of our attention will go to the first phase, starting with the candidate generation procedure.

3.1 Candidate Generation As in most frequent pattern mining algorithms, it is impossible to generate the complete search space of all simple conjunctive queries and compute their support, as there are simply too much of them.

Apart from being able to discover interesting relationships among simple conjunctive queries, our definition of diagonal containment also has the following interesting properties.

PROPERTY 3.1. *Given a relational database $\mathcal{D}$, the diagonal containment relation $\subseteq^{\Delta}$ is a partial order, which defines a lattice over the simple conjunctive queries on $\mathcal{D}$.*

Additionally, the lattice also has the desirable monotonicity property.

PROPERTY 3.2. *Let Q_1 and Q_2 be two simple conjunctive queries. If $Q_2 \subseteq^{\Delta} Q_1$, then support$(Q_1) \geq$ support(Q_2).*

This allows us to use a level-wise (Apriori) strategy [3, 17, 16] and prune all simple conjunctive queries contained in an infrequent query.

Essentially, we generate all possible instantiations of X and F in $\pi_X \sigma_F (R_1 \times \cdots \times R_n)$. An overall outline of our algorithm is the following:

Selection loop: Generate all instantiations of F, without constants, in a breadth-first manner.

Projection loop: For each generated selection, generate all instantiations of X in a breadth-first manner, and test their frequency.

Constants loop: For each generated query in the projection loop, add constant assignments to F in a breadth-first manner.

Next, we describe each of these loops in detail, after which we describe our implemented techniques to efficiently evaluate each query on the database.

3.1.1 Selection loop A selection in a simple conjunctive query consists of a conjunction of equalities between attributes. Hence, it defines a partition of all attributes. Generating all partitions of a set is a well studied problem for which efficient solutions exist. We will use the so called *restricted growth string* for generating all partitions [18].

A Restricted Growth string is an array $a[1 \ldots m]$ where m is the total number of attributes occurring in the database, and $a[i]$ is the block identifier of the block in the partition in which attribute i occurs. Obviously, a partition can be represented by several such strings, but in order to identify a unique string for each partition, the so called *restricted growth* string satisfies the following growth inequality (for $i = 1, 2, \ldots, n-1$, and with $a[1] = 1$):

$$a[i+1] \leq 1 + \max a[1], a[2], \ldots, a[i].$$

EXAMPLE 4. *Let A_1, A_2, A_3, A_4 be the set of all attributes occurring in the database. Then, the restricted growth string* 1221 *represents the conjunction of equalities $A_1 = A_4, A_2 = A_3$.*

The algorithm to generate all selection queries is shown in Algorithm 1. In order to efficiently generate all partitions without generating duplicates, we start initially with the singleton string "1", representing the first attribute belonging to block 1, and all remaining attributes belong to their own unique block (although this is not explicitly represented). Then, given such a string representing a specific partition, all more specific partitions are generated by adding one of the remaining attributes to an existing block (line 7). To make sure no duplicates are generated, we assume an order over all attributes and do not add an attribute to an existing block if any of the attributes coming after that have already been assigned to an existing block.

Algorithm 1 SelectionLoop()

```
1:  σ(Q) ← "1" {initial restricted growth string}
2:  push(Queue, Q)
3:  while not Queue is empty do
4:    SQ ← pop(Queue)
5:    if rgs does not represent a cartesian product then
6:      ProjectionLoop(SQ)
7:    children ← RestictedGrowth(σ(SQ), m)
8:    for all rgs in childeren do
9:      if selection defined by rgs is not more specific than
        the user most specific selection then
10:       σ(SQC) ← rgs
11:       push(Queue, SQC)
```

This traversal of the search space for four attributes is illustrated in Figure 1. Essentially, our algorithm performs a breadth-first traversal over this tree. The actual generation of the restricted growth string is shown in Algorithm2.

Algorithm 2 RestrictedGrowth(String *prefix*, Length *m*)

```
1:  list ← {}
2:  last ← length(prefix)
3:  if last < m then
4:    for i = last to m − 1 do
5:      max ← max({prefix[j] | 0 ≤ j < last})
6:      nprefix ← prefix
7:      if i > last then
8:        for k = last to i − 1 do
9:          max ← max + 1
10:         nprefix[k] ← max
11:     for l = 1 to max do
12:       nprefix[i] := l
13:       add(list, nprefix)
14: return list
```

Of course, as a special case, initially we consider the queries consisting of each relation separately without any selection.

Before generating possible projections for a given selection, we first determine whether the selection represents a cartesian product (line 5). If so, we skip generating projections for this selection and continue the loop, adding new attributes until the selection is no longer representing a cartesian product.

Intuitively, to determine whether a selection represents a cartesian product, we interpret each simple conjunctive query as an undirected graph, such that each relation or constant is a node, and each equality in the selection of the query is an edge between the nodes occurring in that equality. Then, we only allow those queries for which all edges are in the same single connected component. All other queries are cartesian products. This is also illustrated in Figures 2(a) and 2(b). That is, Figure 2(a) represents the selection

$$R_1.C = R_2.D, R_2.E = R_3.H, R_2.F = R_2.G,$$

resulting in a single connected component, while Figure 2(b) represents the query

$$R_1.B = R_1.C, R_2.E = R_3.H, R_2.F = R_2.G,$$

resulting is two disconnected components. Hence, independent of the projected attributes, any simple conjunctive query using the second selection will always contain a cartesian product.

3.1.2 Projection loop For each selection, all allowed projections are generated, consisting of those attributes that are part of the single connected component (line 6, Algorithm 1). Indeed, projecting on any other attribute would result in a cartesian product. Therefore, our algorithm (Algorithm 3) starts with the set of all allowed attributes for the given selection, and generates subsets in a breadth-first manner w.r.t.

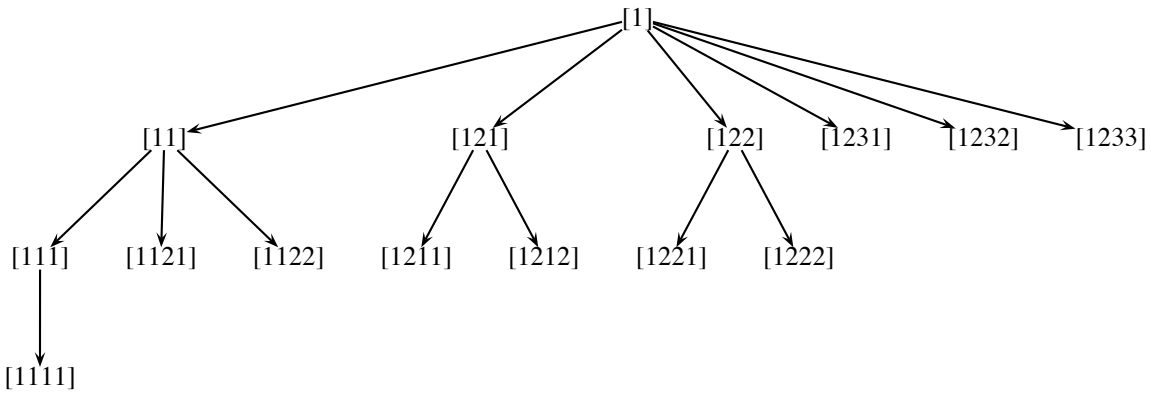
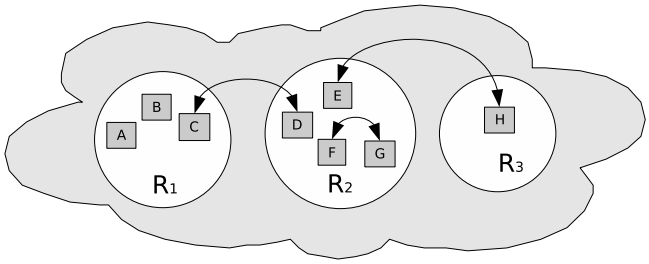
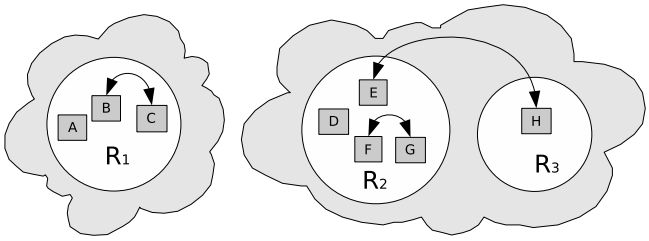


Figure 1: Restricted Growth Expansion



(a) a single connected component



(b) disconnected components

Figure 2: Cartesian Products

diagonal containment. Not all subsets are generated, however, as this might result in many duplicate queries. Indeed, when we remove an attribute from the projection while there might still be other attributes from its block in the selection partition, we obtain an equivalent query. Therefore, we immediately remove all attributes from a single block of the selection partition.

For every generated projection, we first check whether all more general queries are known to be frequent (line 5), and if so, the resulting query is evaluated against the database (line 6). If the query turns out to be infrequent, then none of its subsets are considered anymore, as they must be infrequent too.

Algorithm 3 ProjectionLoop(Conjunctive Query Q)

```
1:  π(Q) ← all connected blocks of σ(Q)
2:  push(Queue, Q)
3:  while not Queue is empty do
4:    PQ ← pop(Queue)
5:    if all queries ⊇^Δ PQ have support > minsup then
6:      if support(PQ) > minsup then
7:        ContantsLoop(PQ)
8:        removed ← connected blocks of σ(PQ) ∉ π(PQ)
9:        torem ← connected blocks of σ(PQ) > last of
          removed {order on blocks is supposed in order
          to generate uniquely}
10:       for all p_i ∈ torem do
11:         π(PQC) ← π(PQ) with block p_i removed
12:         push(Queue,PQC)
```

3.1.3 Constants loop Every block of attribute equalities of the selection can also be set equal to a constant. Again, this is done in a level-wise, breadth-first manner as shown in Algoritm 4. First, we assign a constant to a single block in the selection partition of the query (line 5). Then, we assign constants to two different blocks, only if these constants were already resulting in frequent queries separately. This is

repeated until no more combinations can be generated s(line 8). Again, there is one exception. We do not allow constants to be assigned to blocks that are in the projection. Indeed, also these would be equivalent to queries in which this block is not in the projection.

For every generated assignment of constants, the resulting query is evaluated against the database (line 11). If the query turns out to be infrequent, then no more specific constant assignments are generated, as they must be infrequent too.

Algorithm 4 ConstantsLoop(Conjunctive Query Q)

```
1:  push(Queue,Q)
2:  while not Queue is empty do
3:    CQ ← pop(Queue)
4:    if c(CQ) = ∅ then
5:      toadd ← all connected blocks of σ(CQ) ∉ π(CQ)
6:    else
7:      uneq ← connected blocks of σ(CQ) ∉ (c(CQ) ∪
        π(CQ))
8:      toadd ← blocks in uneq > last of c(CQ) {order on
        blocks is supposed in order to generate uniquely}
9:    for all B_i ∈ toadd do
10:     c(CQC) ← c(CQ) with B_i added
11:     if exist frequent constant values for c(CQC) in the
        database then
12:       push(Queue,CQC)
```

3.2 Candidate Evaluation In order to get the support of each generated query, they are evaluated against the database by translating each query to SQL. For efficiency reasons, however, this query translation and evaluation is only performed for queries with the most general projection and no constant equalities. The result of such a query is then stored in a temporary table (τ). We can now rewrite more specific queries to use these temporary tables resulting in a more efficient evaluation as we are now querying just a specific part of the database and we no longer have to perform a possibly expensive join operation for each more specific query. All more more specific projections X', having the same selection, are evaluated by the query $\pi_{X'}\tau$.

To evaluate the more specific conjunctive queries containing constant equation we developed some additional optimisations. It is hard and inefficient to keep the constant values in main memory, therefore we will also use temporary tables to store them. Creating the SQL queries for these conjunctive queries will involve combining the various temporary tables from previous loops using the monotonicity property.

We first encounter more specific queries that contain one constant equation (e.g. A). These queries are rewritten as follows.

EXAMPLE 5.

```
SELECT A, COUNT(*) AS sup
    FROM τ GROUP BY A
```

The result of these queries is stored in a new temporary table (τ_A) holding the constant values together with their support. Queries containing more constant equations will then use these tables to generate these combinations and will be stored in temporary tables of their own as illustrated in the examples below.

EXAMPLE 6. *Let τ_A and τ_B be the temporary tables holding the constant values for the attributes A and B together with their support (generated previously). We can now generate the table $\tau_{A,B}$ using the results of the following query*

```
SELECT A, B, COUNT(*) FROM
       (SELECT A,B, X' FROM
          τ NATURAL JOIN
             (SELECT * FROM
                      (SELECT A FROM τ_A)
                      NATURAL JOIN
                      (SELECT B FROM τ_B)
             )
       )
GROUP BY A,B
HAVING COUNT(*) >= minsup
```

In this case we are using the values already obtained for A and B in generating the combinations, and using a join with the temporary table τ to evaluate against the database, immediately using the minimal support value $minsup$ to only get frequent queries.

In the previous example the join actually was a simple product but as one advances more levels (i.e. more blocks are equal to constants) it becomes a real join on the common attributes as illustrated in the example below.

EXAMPLE 7. *This is the generated query for getting the values for $\tau_{A,B,C}$ using the temporary tables τ, $\tau_{A,B}$, $\tau_{A,c}$, $\tau_{B,C}$.*

```
SELECT A, B, C, COUNT(*) FROM
       (SELECT A,B,C, X' FROM
          τ NATURAL JOIN
             (SELECT * FROM
                (SELECT A,B FROM τ_{A,B})
                NATURAL JOIN
                (SELECT A,C FROM τ_{A,C})
                NATURAL JOIN
                (SELECT B,C FROM τ_{B,C})
             )
       )
GROUP BY A,B,C
HAVING COUNT(*) >= minsup
```

3.3 Association Rule Generation The generation of association rules is performed in a straightforward manner. For all queries Q_1 the algorithm finds all queries Q_2 such that $Q_2 \subseteq^{\Delta} Q_1$, it computes the confidence of the rule $Q_1 \Rightarrow Q_2$ and tests whether it is confident. As already explained in the beginning of this section, the time and space consumption of this phase is negligible w.r.t. the query generation phase.

4 Eliminating Redundancies

Using the algorithm as described in the previous section, many association rules can be discovered. Unfortunately, they contain a lot of redundancies.

EXAMPLE 8. *Consider the following association rules, each based on a vertical containment:*

$$(4.1)\quad \pi_{\mathrm{R.A,R.B,S.E}}\sigma_{\mathrm{R.C=S.F}}(R \times S) \Rightarrow \pi_{\mathrm{R.A,S.E}}\sigma_{\mathrm{R.C=S.F}}(R \times S)$$

$$(4.2)\quad \pi_{\mathrm{R.A,S.E}}\sigma_{\mathrm{R.C=S.F}}(R \times S) \Rightarrow \pi_{\mathrm{R.A}}\sigma_{\mathrm{R.C=S.F}}(R \times S)$$

$$(4.3)\quad \pi_{\mathrm{R.A,R.B,S.E}}\sigma_{\mathrm{R.C=S.F}}(R \times S) \Rightarrow \pi_{\mathrm{R.A}}\sigma_{\mathrm{R.C=S.F}}(R \times S)$$

Now suppose the first association rule has a confidence of 100%*. Then, the confidence of the second and third association rule must be equal, and hence, one of them can be considered redundant. We choose to retain those rules using the most general queries, and thus, in this case, the third rule.*

Now suppose the second association rule has a confidence of 100%*. Then, the confidence of the first and third association rule must be equal, and one of them can be considered redundant. Again, we choose to retain the rule with the most general queries, and thus, in this case, the first rule.*

More formally, we have the following Lemma.

LEMMA 4.1. *An association rule $Q_1 \Rightarrow Q_2$ is redundant if*

1. *there exists an association rule $Q_3 \Rightarrow Q_1$ with confidence* 100%*, or*
2. *there exists an association rule $Q_4 \Rightarrow Q_2$ with confidence* 100%*, and $Q_4 \subseteq^{\Delta} Q_1$.*

It is easy to see that in the first case, there exists an association rule $Q_3 \Rightarrow Q_2$ having the same confidence and support as $Q_1 \Rightarrow Q_2$, and in the second case, there exists an association rule $Q_1 \Rightarrow Q_4$ having the same confidence and support as $Q_1 \Rightarrow Q_2$. Together with the rule having 100% condfidence, the confidence of $Q_1 \Rightarrow Q_2$ can be easily deduced. This lemma provides a powerful tool to remove many redundant association rules. In practice, before outputting an association rule, it is first checked whether there already exists such 100% confident association rule as described in the lemma. Fortunately, such rules must have been generated already before as both Q_3 and Q_4 are more general than Q_1 and Q_2 respectively.

ACTORS.*	45342
ACTORS.AID	45342
ACTORS.NAME	45342
GENRES.*	21
GENRES.GID	21
GENRES.NAME	21
MOVIES.*	71912
MOVIES.MID	71912
MOVIES.NAME	71906
ACTORMOVIES.*	158441
ACTORMOVIES.AID	45342
ACTORMOVIES.MID	54587
GENREMOVIES.*	127115
GENREMOVIES.GID	21
GENREMOVIES.MID	71912

Table 1: Number of tuples per attribute in the IMDB database

5 Experiments

We performed several experiments using our prototype on a snapshot of a part of the Internet Movie Database [12], and on a database that is the backend of an online quiz website [4]. A summary of the characteristics of these databases can be found in Table 1 and Table 2 respectively.

The experiments were performed on a 2.19 GHz AMD Opteron processor with 2GB of internal memory, running Linux 2.6. The prototype algorithm was written in Java using JDBC to communicate with a PostgreSQL 8.0.12 relational database.

What follows are some examples of interesting patterns discovered by our algorithm. We will abbreviate the relation names to improve readability.

5.1 IMDB The IMDB snapshot consist of three tables `ACTORS (A)`, `MOVIES (M)` and `GENRES (G)`, and two tables that represent the connections between them namely `ACTORMOVIES (AM)` and `GENREMOVIES (GM)`.

EXAMPLE 9. *The following rule, has 100% confidence and represents a* ***functional dependency****, which in this case is a key dependency.*

SCORES.*	868755
SCORES.SCORE	14
SCORES.NAME	31934
SCORES.QID	5144
SCORES.DATE	862769
SCORES.RESULTS	248331
SCORES.MONTH	12
SCORES.YEAR	6
QUIZZES.*	4884
QUIZZES.QID	4884
QUIZZES.NAME	4674
QUIZZES.MAKER	328
QUIZZES.CATEGORY	18
QUIZZES.LANGUAGE	2
QUIZZES.NUMBER	539
QUIZZES.AVERAGE	4796

Table 2: Number of tuples per attribute in the Quiz database

$$\pi_{\text{M.MID,M.NAME}}(M) \Rightarrow \pi_{\text{M.MID}}(M)$$

Interestingly enough the rule

$$\pi_{M.MID,M.NAME}(M) \Rightarrow \pi_{\text{M.NAME}}(M)$$

has 99.99% confidence, so we can derive that in our database snapshot there are different movies which have the same name (although not a lot). This type of association rules is also known as ***approximate functional dependencies*** *[14].*

We also find more complex patterns, as illustrated in the following examples.

EXAMPLE 10. *We can conclude that every movie has a genre because of the following association rule with 100% confidence*

$$\pi_{\text{M.MID}}(M) \Rightarrow \pi_{\text{M.MID}}\sigma_{\text{GM.MID=M.MID}}(M \times GM)$$

It is easy to see that this type of rules describe ***inclusion dependencies*** *occurring in the database.*

On the contrary, in our database, not every movie has to have an actor associated with it as the following rule only has 75.91% confidence

$$\pi_{\text{M.MID}}(M) \Rightarrow \pi_{\text{M.MID}}\sigma_{\text{AM.MID=M.MID}}(M \times AM)$$

This last rule may be counter intuitive, but these kind of rules occur due to the fact that we are working in a partial (snapshot) database.

EXAMPLE 11. *We can find 'frequent' genres in which actors play. The rule*

$$\pi_{\text{AM.AID}}(AM) \Rightarrow \pi_{\text{AM.AID}}\sigma_{\text{AM.MID=GM.MID,}}$$
$$\text{GM.GID=G.GID,G.GID='3'}(AM \times GM \times G)$$

has 40.44% confidence, so 40.44% of the actors play in a 'Documentary' (genre id 3) while the same rule for 'Drama' has 49.85% confidence.

But also other types of interesting patterns can be discovered. For instance, the following rule has 81.60% confidence.

$$\pi_{\text{AM.AID,AM.MID}}\sigma_{\text{AM.MID=GM.MID,}}$$
$$\text{GM.GID=G.GID,G.GID='16'}(AM \times GM \times G) \Rightarrow$$
$$\pi_{\text{AM.AID}}\sigma_{\text{AM.MID=GM.MID,}}$$
$$\text{GM.GID=G.GID,G.GID='16'}(AM \times GM \times G)$$

While for example the same rule for genre 'Crime' has only 49.87% confidence. Intuitively it could indicate that 81.60% of the actors in genre 'Music' (genre id 16) only play in one movie. But this rule could just as well indicate that one actor plays in 18.4% of the movies. Examining constant values for actor id (under a low confidence) could allow us to find patterns like the latter.

5.2 Quiz Database The quiz database consists of two relations `QUIZZES (Q)` and `SCORES (S)`, containing the data about the quiz (who made it, the category,...) and data about the participants (name, result,...) respectively.

Similarly to the IMDB snapshot database we are able to find functional dependencies.

EXAMPLE 12. *We find that the rule*

$$\pi_{\text{Q.QID,Q.MAKER}}(Q \times S) \Rightarrow \pi_{\text{Q.QUIZID}}(Q \times S)$$

has 100% confidence and represents a functional dependency. The rule

$$\pi_{\text{S.NAME}}(Q \times S) \Rightarrow \pi_{\text{S.NAME}}\sigma_{\text{Q.QID=S.QID}}(Q \times S)$$

however only has 99% confidence and thus again it represents an ***approximate inclusion dependency****. It means that at least one player played a quiz that is not present in the QUIZZES table. Rules like this could indicate that there might be errors in the database, and are therefore very valuable in practice.*

We also discovered a lot of more complex and structured patterns of which some examples are given below.

EXAMPLE 13. *To our surprise, the following rule has only 86% confidence.*

$$\pi_{\text{Q.QID}}\sigma_{\text{Q.MAKER=S.NAME,Q.QID=S.QID}}(Q \times S) \Rightarrow$$
$$\pi_{\text{Q.QID}}\sigma_{\text{Q.MAKER=S.NAME,Q.QID=S.QID,}}$$
$$\text{S.SCORE='9'}(Q \times S)$$

This rule expresses that only in 86% of the cases where a maker plays his own quiz he gets the maximum score of 9.

EXAMPLE 14. *We discovered a rule about a particular player, Benny in this case. We can see that he only has the maximum score in 70% of the quizzes he made.*

$$\pi_{\text{Q.QID}}\sigma_{\text{Q.MAKER=S.NAME,Q.MAKER='Benny'}} \\ {}_{Q.QID=S.QID}(Q \times S) \Rightarrow \\ \pi_{\text{Q.QID}}\sigma_{\text{Q.MAKER=S.NAME,Q.MAKER='Benny',}} \\ {}_{\text{Q.QID=S.QID,S.SCORE='9'}}(Q \times S)$$

For this same player we also discover the following rule having 76% confidence.

$$\pi_{\text{Q.QID}}\sigma_{\text{Q.CATEGORY='music'}}(Q \times S) \Rightarrow \\ \pi_{\text{Q.QID}}\sigma_{\text{Q.CATEGORY='music',Q.QID=S.QID,}} \\ {}_{\text{S.NAME='Benny'}}(Q \times S)$$

This rule tells us that of all the quizzes in the category music, Benny has played 76% of them.

Next, we discovered the following rule having 49% confidence.

$$\pi_{\text{S.NAME}}\sigma_{\text{Q.QID=S.QID}}(Q \times S) \Rightarrow \\ \pi_{\text{S.NAME}}\sigma_{\text{Q.QID=S.QID,Q.MAKER='Benny'}}(Q \times S)$$

This rule tells us that 49% of all players have played a quiz made by Benny.

We also found that Benny has some real fans as the following rule has 98% confidence.

$$\pi_{\text{S.NAME}}\sigma_{\text{Q.QID=S.QID,Q.MAKER='Benny'}}(Q \times S) \Rightarrow \\ \pi_{\text{S.NAME}}\sigma_{\text{Q.QID=S.QID,Q.MAKER='Benny',}} \\ {}_{\text{S.NAME='Raf'}}(Q \times S)$$

It tells us that the player Raf has played 98% of Benny's quizzes.

5.3 Performance Using our implemented prototype, we conducted some performance measurements using the IMDB snapshot and quiz databases. The results can be seen in Figures 3 and 4. Looking at Figure 3(b) we can see that for a lower minimal support the number of patterns generated increases drastically. This is due to the fact that an exponential number of combinations of constants come in to play at this lower support thresholds. The IMDB snapshot database has a higher number of unique constant values compared to the quiz database which results in a much smoother curve in Figure 3(a). In this figure we can also see that in the case of the quiz database the time scales with respect to the number of patterns.

In Figure 4 we can see the number of rules generated for varying confidence and two different support values (1000 and 200 respectively). Here we clearly see the impact of the use of pruning the redundancies. The number of patterns is reduced by more than an order of magnitude.

Essentially, these basic experiments show the feasibility of our approach and the value of our pruning efforts.

6 Related Work

Towards the discovery of association rules in arbitrary relational databases, Deshaspe and Toivonen developed an inductive logic programming algorithm, WARMR [8], that discovers association rules over a limited set of conjunctive queries on a transactional database in which every transaction consists of a small relational database itself. We extended their framework to a more general setting working on a single arbitrary relational database, by considering conjunctive queries as patterns [10]. Unfortunately, the search space of all conjunctive queries is infinite and there exist no most general or even a most specific pattern, with respect to query containment and a frequency measure based on the number of tuples in the output of a query [10]. Additionally, deciding whether two conjunctive queries are equivalent is an NP-complete problem. Therefore, we restricted again to subclasses of conjunctive queries allowing more efficient algorithms. Towards that goal, we considered the special subclass of tree-shaped conjunctive queries defined over a single binary relation representing a graph [9, 11]. We showed these tree queries are powerful patterns, useful for mining graph-structured data. In this paper, we consider a second class of queries over several relations, but allow them to occur at most once. Additionally, we introduce the novel notion of *diagonal containment* providing an excellent tool to compare and interpret queries having a different set of projected attributes. To the best of our knowledge, we are not aware of any other work considering this class of patterns. Nevertheless, we show that this class still allows efficient discovery of a wide range of very interesting association rules.

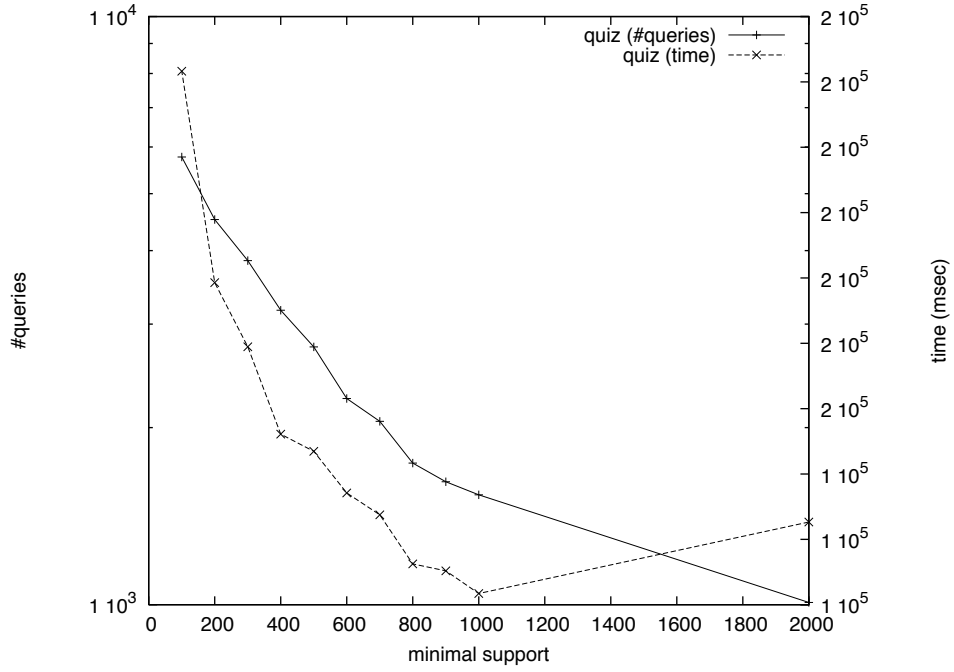
If the database consists of a Boolean transaction database, such that an item can be represented by a single unary relation in which each tuple is a transaction identifier of the transaction in which the item occurs, then every non-empty frequent itemset $\{i, \ldots, j\}$ can be represented by the conjunctive query

$$\pi_{R_i.A}\sigma_{R_i.A=R_{i+1}.A,\cdots,R_{j-1}.A=R_j.A}R_1 \times \cdots \times R_n,$$

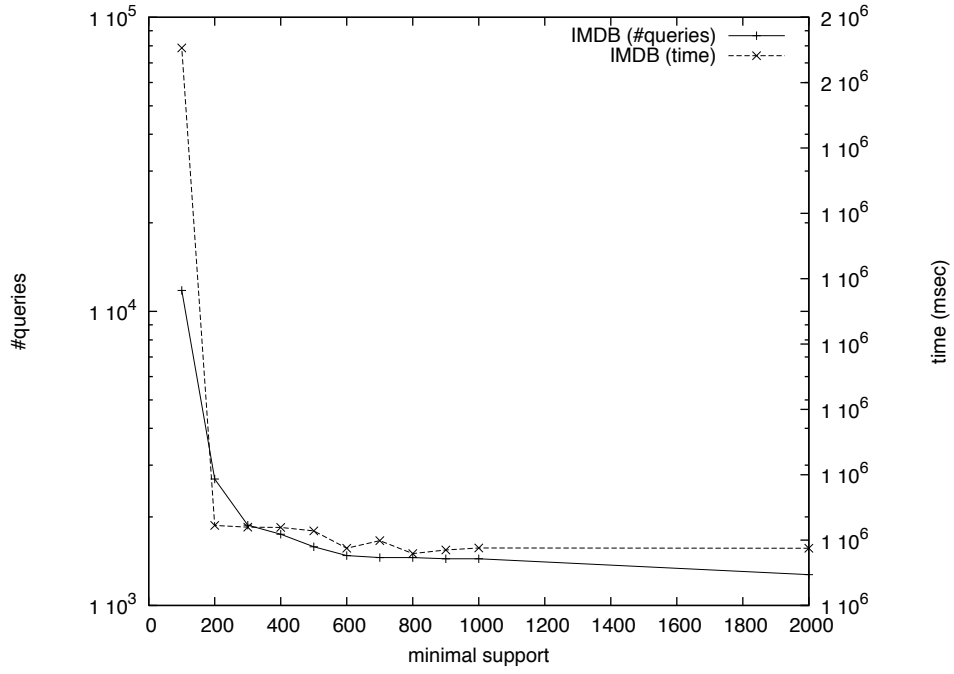
with $1 \leq i \leq j \leq n$. Furthermore, in that specific case, our algorithm reduces to the well known Apriori algorithm [3], which we believe is a major advantage of this approach.

7 Future Work

In this work, we have considered one class of conjunctive queries showing very promising results. In previous work, also another class of conjunctive queries, considering only a single binary relation, has shown its usefulness [9, 11].

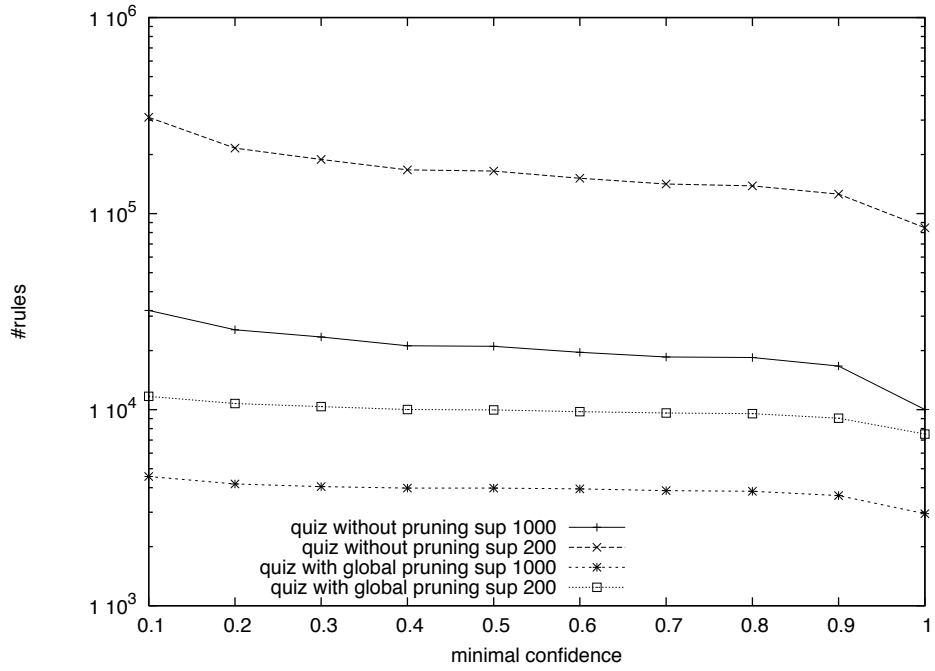


(a) Quiz database

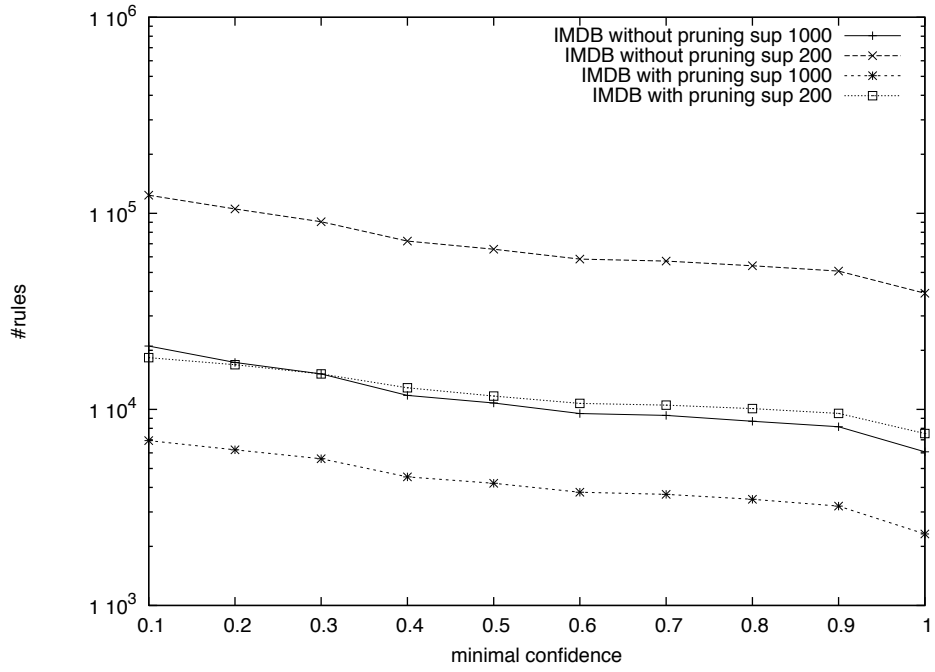


(b) IMDB database

Figure 3: Support variation



(a) Quiz database



(b) IMDB database

Figure 4: Confidence variation

It would be interesting to see how well these two can be combined, but also other new classes could be considered.

Although our pruning technique drastically reduces the number of patterns, many still remain. We plan to study alternate or additional interestingness measures and ranking techniques to be able to present the user with a compact and easy to interpret set of useful patterns.

Furthermore it is the case that the generated patterns are dependant of the database decomposition provided by the user. For example, if our algorithm is presented with the relation $R(A, B, C)$ it will generate different patterns than when presented with a decomposition of that, say $R(A, B)$ and $R(A, C)$. The impact and potential use of different decompositions is also subject to further study.

8 Conclusion

This research is motivated by the fact that many relational databases can not always be simply transformed to datasets for typical frequent pattern mining algorithms, as these require some kind of transactions to be counted. As illustrated, possible transformations immediately strongly bias the type of patterns that can still be found, and hence, a lot of potentially interesting information gets lost. We have presented a new and appealing type of association rules, by pairing simple conjunctive queries. As already illustrated in the examples, next to many different kinds of interesting patterns, our algorithm is also able to discover *functional dependencies*, *inclusion dependencies*, but also their variants, such as the very recently studied *conditional functional dependencies*, which turn out to be very useful for data cleaning purposes [5]. Also associations having a lower confidence than 100% are discovered, revealing so called *approximate dependencies* [14]. We presented a completely novel algorithm efficiently generating and pruning the search space of all simple conjunctive queries, and we presented promising experiments, showing the feasibility of our approach, but also its usefulness towards the ultimate goal of discovering patterns in arbitrary relational databases.

Acknowledgments

We would like to thank Jan Van den Bussche for many interesting and educating discussions on this work.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] R. Agrawal, T. Imielinski, and A.N. Swami. Mining association rules between sets of items in large databases. In Buneman and Jajodia [6], pages 207–216.

[3] R. Agrawal, T. Imielinski, and A.N. Swami. Mining association rules between sets of items in large databases. In Buneman and Jajodia [6], pages 207–216.

[4] R. Bocklandt. http://www.persecondewijzer.net.

[5] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *Proceedings of the International Conference on Data Engineering*, 2007.

[6] P. Buneman and S. Jajodia, editors. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, volume 22:2 of *SIGMOD Record*. ACM Press, 1993.

[7] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings 9th ACM Symposium on the Theory of Computing*, pages 77–90. ACM Press, 1977.

[8] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.

[9] B. Goethals, E. Hoekx, and J. Van den Bussche. Mining tree queries in a graph. *Conference on Knowledge Discovery in Data*, pages 61–69, 2005.

[10] B. Goethals and J. Van den Bussche. Relational Association Rules: Getting WARMeR. *Proceedings of the ESF Exploratory Workshop on Pattern Detection and Discovery*, pages 125–139, 2002.

[11] E. Hoekx and J. Van den Bussche. Mining for Tree-Query Associations in a Graph. In *Proceedings IEEE International Conference on Data Mining*, 2006.

[12] IMDB. http://imdb.com.

[13] A. Inokuchi, T. Washio, and H. Motoda. An Apriori-based algorithm for mining frequent substructures from graph data. In D.A. Zighed, H.J. Komorowski, and J.M. Zytkow, editors, *PKDD*, volume 1910 of *Lecture Notes in Computer Science*, pages 13–23. Springer, 2000.

[14] J. Kivinen and H. Mannila. Approximate inference of functional dependencies from relations. *Theoretical Computer Science*, 149(1):129–149, 1995.

[15] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In N. Cercone, T.Y. Lin, and X. Wu, editors, *Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM 2001)*, pages 313–320. IEEE Computer Society Press, 2001.

[16] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.

[17] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In *Proceedings AAAI Workshop on Knowledge Discovery in Databases*, pages 181–192. AAAI Press, 1994.

[18] E. W. Weisstein. Restricted growth string. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/RestrictedGrowthString.html.

[19] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 721–724. IEEE Computer Society Press, 2002.

[20] M.J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 71–80. ACM Press, 2002.