Relational Association Rules: getting WARMER

Bart Goethals and Jan Van den Bussche
University of Limburg, Belgium

Abstract

In recent years, the problem of association rule mining in transactional data has been well studied. We propose to extend the discovery of classical association rules to the discovery of association rules of conjunctive queries in arbitrary relational data, inspired by the WARMR algorithm, developed by Dehaspe and Toivonen, that discovers association rules over a limited set of conjunctive queries. Conjunctive query evaluation in relational databases is well understood, but still poses some great challenges when approached from a discovery viewpoint in which patterns are generated and evaluated with respect to some well defined search space and pruning operators.

1 Introduction

In recent years, the problem of mining association rules over frequent itemsets in transactional data [9] has been well studied and resulted in several algorithms that can find association rules within a limited amount of time. Also more complex patterns have been considered such as trees [17], graphs [11, 10], or arbitrary relational structures [5, 6]. However, the presented algorithms only work on databases consisting of a set of transactions. For example, in the tree case [17], every transaction in the database is a separate tree, and the presented algorithm tries to find all frequent subtrees occurring within all such transactions. Nevertheless, many relational databases are not suited to be converted into a transactional format and even if this were possible, a lot of information implicitly encoded in the relational model would be lost after conversion. Towards the discovery of association rules in arbitrary relational databases, Deshaspe and Toivonen developed an inductive logic programming algorithm, WARMR [5, 6], that discovers association rules over a limited set of conjunctive queries on transactional relational databases in which every transaction consists of a small relational database itself. In this paper, we propose to extend their framework to a broader range of conjunctive queries on arbitrary relational databases.

Conjunctive query evaluation in relational databases is well understood, but still poses some great challenges when approached from a discovery viewpoint in which patterns are generated and evaluated with respect to some well defined search space and pruning operators. We describe the problems occurring in this mining problem and present an algorithm that uses a similar two-phase architecture as the standard association rule mining algorithm over frequent itemsets (Apriori) [1], which is also used in the WARMR algorithm. In the first phase, all frequent patterns are generated, but now, a pattern is a conjunctive query and its support equals the number of distinct tuples in the answer of the query. The second phase generates all association rules over these patterns. Both phases are based on the general levelwise pattern mining algorithm as described by Mannila and Toivonen [12].

In Section 2, we formally state the problem we try to solve. In Section 3, we describe the general approach that is used for a large family of data mining problems. In Section 4, we describe the WARMR algorithm which is also based on this general approach. In Section 5, we describe our approach as an generalization of the WARMR algorithm and identify the algorithmic challenges that need to be conquered. In Section 6, we show a sample run of the presented approach. We conclude the paper in Section 7 with a brief discussion and future work.

2 Problem statement

The relational data model is based on the idea of representing data in tabular form. The *schema* of a relational database describes the names of the tables and their respective sets of column names, also called attributes. The actual content of a database, is called an *instance* for that schema. In order to retrieve data from the database, several query languages have been developed, of which SQL is the standard adopted by most database management system vendors. Nevertheless, an important and well-studied subset of SQL, is the family of conjunctive queries.

As already mentioned in the Introduction, current algorithms for the discovery of patterns and rules mainly focused on transactional databases. In practice, these algorithms use several specialized data structures and indexing schemes to efficiently find their specific type of patterns, i.e., itemsets, trees, graphs, and many others. As an appropriate generalization of these kinds of patterns, we propose a framework for arbitrary relational databases in which *a pattern is a conjunctive query*.

Assume we are given a relational database consisting of a schema $\mathbf{R}$ and an instance $\mathbf{I}$ of $\mathbf{R}$. An *atomic formula* over $\mathbf{R}$ is an expression of the form $R(\bar{x})$, where R is a relation name in $\mathbf{R}$ and $\bar{x}$ is a k-tuple of variables and constants, with k the arity of R.

Definition 1. A *conjunctive query* Q over $\mathbf{R}$ consists of a *head* and a *body*. The body is a finite set of atomic formulas over $\mathbf{R}$. The head is a tuple of variables occurring in the body.

A *valuation* on Q is a function f that assigns a constant to every variable in the query. A valuation is a *matching* of Q in $\mathbf{I}$, if for every $R(\bar{x})$ in the body of Q, the tuple $f(\bar{x})$ is in $\mathbf{I}(R)$. The *answer* of Q on $\mathbf{I}$ is the set

$$Q(\mathbf{I}) := \{f(\bar{y}) \mid \bar{y} \text{ is the head of } Q \text{ and } f \text{ is a matching of } Q \text{ on } \mathbf{I}\}.$$

We will write conjunctive queries using the commonly used Prolog notation. For example, consider the following query on a beer drinkers database:

$$Q(x) \text{ :- } \textit{likes}(x, \text{`Duvel'}), \textit{likes}(x, \text{`Trappist'}).$$

The answer of this query consists of all drinkers that like Duvel and also like Trappist.

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 $Q_2 \subseteq Q_1$. Note that the question whether a conjunctive query is contained in another conjunctive query is decidable [16].

Definition 2. 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 called *frequent* in $\mathbf{I}$ if its support exceeds a given *minimal support threshold*.

Definition 3. An *association rule* is of the form $Q_1 \Rightarrow Q_2$, such that Q_1 and Q_2 are both conjunctive queries and $Q_2 \subseteq Q_1$. An association rule is called *frequent* in $\mathbf{I}$ if Q_2 is frequent in $\mathbf{I}$ and it is called *confident* if the support of Q_2 divided by the support of Q_1 exceeds a given *minimal confidence* threshold.

Example 1. Consider the following two queries:

$$\begin{aligned} Q_1(x,y) &\text{ :- } \textit{likes}(x, \text{`Duvel'}), \textit{visits}(x,y). \\ Q_2(x,y) &\text{ :- } \textit{likes}(x, \text{`Duvel'}), \textit{visits}(x,y), \textit{serves}(y, \text{`Duvel'}). \end{aligned}$$

The rule $Q_1 \Rightarrow Q_2$ should then be read as follows: if a person x that likes Duvel visits bar y, then bar y serves Duvel.

A natural question to ask is why we should only consider rules over queries that are contained for any possible instance. For example, assume we have the following two queries:

$$\begin{aligned} Q_1(y) &\text{ :- } \textit{likes}(x, \text{`Duvel'}), \textit{visits}(x,y). \\ Q_2(y) &\text{ :- } \textit{serves}(y, \text{`Duvel'}). \end{aligned}$$

Obviously, Q_2 is not contained in Q_1 and vice versa. Nevertheless, it is still possible that for a given instance $\mathbf{I}$, we have $Q_2(\mathbf{I}) \subseteq Q_1(\mathbf{I})$, and hence this could make an interesting association rule $Q_1 \Rightarrow Q_2$, which should be read as follows: if bar y has a visitor that likes Duvel, then bar y also serves Duvel.

Proposition 1. *Every association rule $Q_1 \Rightarrow Q_2$, such that $Q_2(\mathbf{I}) \subseteq Q_1(\mathbf{I})$, can be expressed by an association rule $Q_1 \Rightarrow Q_2'$, with $Q_2' = Q_2 \cap Q_1$, and essentially has the same meaning.*

In this case the correct rule would be $Q_1 \Rightarrow Q_2$, with

$$\begin{aligned} Q_1(y) &:\!- \mathit{likes}(x, \text{`Duvel'}), \mathit{visits}(x, y). \\ Q_2(y) &:\!- \mathit{likes}(x, \text{`Duvel'}), \mathit{visits}(x, y), \mathit{serves}(y, \text{`Duvel'}). \end{aligned}$$

Note the resemblance with the queries used in Example 1. The bodies of the queries are the same, but now we have another head. Evidently, different heads result in a different meaning of the corresponding association rule which can still be interesting. As another example, note the difference with the following two queries:

$$\begin{aligned} Q_1(x) &:\!- \mathit{likes}(x, \text{`Duvel'}), \mathit{visits}(x, y). \\ Q_2(x) &:\!- \mathit{likes}(x, \text{`Duvel'}), \mathit{visits}(x, y), \mathit{serves}(y, \text{`Duvel'}). \end{aligned}$$

The rule $Q_1 \Rightarrow Q_2$ should then be read as follows: if a person x that likes Duvel visits a bar, then x also visits a bar that serves Duvel.

The goal is now to find all frequent and confident association rules in the given database.

3 General approach

As already mentioned in the introduction, most association rule mining algorithms use the common two-phase architecture. Phase 1 generates all frequent patterns, and phase 2 generates all frequent and confident association rules.

The algorithms used in both phases are based on the general levelwise pattern mining algorithm as described by Mannila and Toivonen [12]. Given a database $\mathcal{D}$, a class of patterns $\mathcal{L}$, and a selection predicate q, the algorithm finds the "theory" of $\mathcal{D}$ with respect to $\mathcal{L}$ and q, i.e., the set $\mathcal{T}h(\mathcal{L}, \mathcal{D}, q) := \{\phi \in \mathcal{L} \mid q(\mathcal{D}, \phi) \text{ is true}\}$. The selection predicate q is used for evaluating whether a pattern $Q \in \mathcal{L}$ defines a (potentially) interesting pattern in $\mathcal{D}$. The main problem this algorithm tries to tackle is to minimize the number of patterns that need to be evaluated by q, since it is assumed this evaluation is the most costly operation of such mining algorithms. The algorithm is based on a breadth-first search in the search space spanned by a specialization relation which is a partial order $\preceq$ on the patterns in $\mathcal{L}$. We say that ϕ is *more specific* than ψ, or ψ is *more general* than ϕ, if $\phi \preceq \psi$. The relation $\preceq$ is a *monotone specialization relation* with respect to q, if the selection predicate q is monotone with respect to $\preceq$, i.e., for all $\mathcal{D}$ and ϕ, we have the following: if $q(\mathcal{D}, \phi)$ and $\phi \preceq \gamma$, then $q(\mathcal{D}, \gamma)$. In what follows, we assume that $\preceq$ is a monotone specialization relation. We write $\phi \prec \psi$ if $\phi \preceq \psi$ and not $\psi \preceq \phi$. The algorithm works iteratively, alternating between *candidate generation* and *candidate evaluation*, as follows.

```
$C_1 := \{\phi \in \mathcal{L} \mid \text{there is no } \gamma \text{ in } \mathcal{L} \text{ such that } \phi \prec \gamma\}$;
$i := 1$;
**while** $C_i \neq \emptyset$ **do**
  // Candidate evaluation
  $\mathcal{F}_i := \{\phi \in C_i \mid q(\mathcal{D}, \phi)\}$;
  // Candidate generation
  $C_{i+1} := \{\phi \in \mathcal{L} \mid \text{for all } \gamma, \text{ such that } \phi \prec \gamma, \text{ we have } \gamma \in \bigcup_{j \leq i} \mathcal{F}_j\} \setminus \bigcup_{j \leq i} C_j$;
  $i := i + 1$
**end while**
**return** $\bigcup_{j<i} \mathcal{F}_j$;
```

In the generation step of iteration i, a collection C_{i+1} of new *candidate patterns* is generated, using the information available from the more general patterns in $\bigcup_{j \leq i} \mathcal{F}_j$, which have already been evaluated. Then, the selection predicate is evaluated on these candidate patterns. The collection $\mathcal{F}_{i+1}$ will consist of those patterns in C_{i+1} that satisfy the selection predicate q. The algorithm starts by constructing C_1 to contain all most general patterns. The iteration stops when no more potentially interesting patterns can be found with respect to the selection predicate.

In general, given a language $\mathcal{L}$ from which patterns are chosen, a selection predicate q and a monotone specialization relation $\preceq$ with respect to q, this algorithm poses several challenges.

1. An initial set C_1 of most general candidate patterns needs to be identified, which is not always possible for infinite languages, and hence other, maybe less optimal solutions could be required.

2. Given all patterns $\bigcup_{j \leq i} \mathcal{F}_j$ that satisfy the selection predicate up to a certain level i, the set C_{i+1} of all candidate patterns must be generated efficiently. It might be impossible to generate all but only those elements in C_{i+1}, but instead, it might be necessary to generate a superset of C_{i+1} after which the non candidate patterns must be identified and removed. Even if this identification is efficient, naively generating all possible patterns could still become infeasible if this number of patterns becomes too large. Hence, this poses two additional challenges:

   (a) efficiently generate the smallest possible superset of C_{i+1}, and

   (b) identify and remove each generated pattern that is no candidate pattern by efficiently checking whether all of its generalizations are in $\bigcup_{j \leq i} \mathcal{F}_j$.

3. Extract all patterns from C_{i+1} that satisfy the selection predicate q, by efficiently evaluating q on all elements in C_{i+1}.

In the next section, we identify these challenges for both phases of the association rule mining problem within the framework proposed by Dehaspe and Toivonen, and describe their solutions as implemented within the WARMR algorithm.

4 The Warmr algorithm

As already mentioned in the introduction, a first approach towards the goal of discovering all frequent and confident association rules in arbitrary relational databases, has been presented by Dehaspe and Toivonen, in the form of an inductive logic programming algorithm, Warmr [5, 6], that discovers association rules over a limited set of conjunctive queries.

4.1 Phase 1

The procedure to generate all frequent conjunctive queries is primarily based on a *declarative language bias* to constrain the search space to a subset of all conjunctive queries, which is an extensively studied subfield in ILP [13].

The declarative language bias used in Warmr drastically simplifies the search space of all queries by using the Warmode formalism. This formalism requires two major constraints. The most important constraint is the *key constraint*. This constraint requires the specification of a single *key* atomic formula which is obligatory in all queries. This key atomic formula also determines *what* is counted, i.e., it determines the head of the query, that is, all variables occuring in the key atom. Second, it requires a list *Atoms* of all atomic formulas that are allowed in the queries that will be generated. In the most general case, this list consists of the relation names in the database schema **R**. If one also wants to allow certain constants within the atomic formulas, then these atomic formulas must be specified for every such constant. In the most general case, the complete database instance must also be added to the *Atoms* list. The Warmode formalism also allows other constraints, but since these are not obligatory, we will not discuss them any further.

Example 2. Consider

$$\begin{aligned}\mathit{Atoms} := \{&\mathit{likes}(\_, \text{`Duvel'}),\\ &\mathit{likes}(\_, \text{`Trappist'}),\\ &\mathit{serves}(\_, \text{`Duvel'}),\\ &\mathit{serves}(\_, \text{`Trappist'})\},\end{aligned}$$

where _ stands for an arbitrary variable, and

$$\mathit{key} := \mathit{visits}(\_, \_).$$

Then,

$$\begin{aligned}
\mathcal{L} = \{&Q(x_1, x_2) \text{ :- } \textit{visits}(x_1, x_2), \textit{likes}(x_3, \text{`Duvel'}).\\
&Q(x_1, x_2) \text{ :- } \textit{visits}(x_1, x_2), \textit{likes}(x_1, \text{`Duvel'}).\\
&\ldots\\
&Q(x_1, x_2) \text{ :- } \textit{visits}(x_1, x_2), \textit{serves}(x_3, \text{`Duvel'}).\\
&Q(x_1, x_2) \text{ :- } \textit{visits}(x_1, x_2), \textit{serves}(x_2, \text{`Duvel'}).\\
&\ldots\\
&Q(x_1, x_2) \text{ :- } \textit{visits}(x_1, x_2), \textit{likes}(x_1, \text{`Duvel'}), \textit{serves}(x_2, \text{`Duvel'}).\\
&Q(x_1, x_2) \text{ :- } \textit{visits}(x_1, x_2), \textit{likes}(x_1, \text{`Duvel'}), \textit{serves}(x_2, \text{`Trappist'}).\\
&\ldots\}.
\end{aligned}$$

As can be seen, these constraints already dismiss a lot of interesting patterns. However, it is still possible to discover all frequent conjunctive queries, but then, we need to run the algorithm for every possible key atomic formula with the least restrictive declarative language bias. Of course, using this strategy, a lot of possible optimizations are left out, as will be shown in the next section.

The specialization relation used in WARMR is defined $Q_1 \preceq Q_2$ if $Q_1 \subseteq Q_2$. The selection predicate q is the minimal support threshold, which is indeed monotone with respect to $\preceq$, i.e., for every instance $\mathbf{I}$ and conjunctive queries Q_1 and Q_2, we have the following: if Q_1 is frequent and $Q_1 \subseteq Q_2$, then Q_2 is frequent.

Candidate generation In essence, the WARMR algorithm generates all conjunctive queries contained in the query $Q(\bar{x})$:- $R(\bar{x})$, where $R(\bar{x})$ is the key atomic formula. Denote this query by the *key conjunctive query*. Hence, the key conjunctive query is the (single) most general pattern in C_1. Assume we are given all frequent patterns up to a certain level i, $\bigcup_{j \leq i} \mathcal{F}_j$. Then, WARMR generates a superset of all candidate patterns, by adding a single atomic formula, from *Atoms*, to every query in $\mathcal{F}_i$, as allowed by the WARMODE declarations. From this set, every candidate pattern needs to be identified by checking whether all of its generalizations are frequent. However, this is no longer possible, since some of these generalizations might not be in the language of admissible patterns. Therefore, only those generalizations that satisfy the declarative language bias need to be known frequent. In order to do this, for each generated query Q, WARMR scans all infrequent conjunctive queries for one that is more general than Q. However, this does not imply that all queries that are more general than Q are known to be frequent! Indeed, consider the following example which is based on the declarative language bias from the previous example.

Example 3.

$$\begin{aligned}
Q_1(x_1, x_2) &\text{ :- } \textit{visits}(x_1, x_2), \textit{likes}(x_1, \text{`Duvel'}).\\
Q_2(x_1, x_2) &\text{ :- } \textit{visits}(x_1, x_2), \textit{likes}(x_3, \text{`Duvel'}).
\end{aligned}$$

Both queries are single extensions of the key conjunctive query, and hence, they are generated within the same iteration. Obviously, Q_2 is more general than Q_1, but still, both queries remain in the set of candidate queries. Moreover, it is necessary that both queries remain admissible, in order to guarantee that all frequent conjunctive queries are generated.

This example shows that the candidate generation step of WARMR does not comply with the general levelwise framework given in the previous section. Indeed, at a certain iteration, it generates patterns of different levels in the search space spanned by the containment relation.

The generation strategy also generates several queries that are equivalent with other candidate queries, or with queries already generated in previous iterations, which also need to be identified and removed from the set of candidate patterns. Again, for each candidate query, all other candidate queries and all frequent queries are scanned for an equivalent query. Unfortunately, the question whether two conjunctive queries are equivalent is an NP-complete problem. Note that isomorphic queries are definitely equivalent (but not vice versa in general), and also the problem of efficiently generating finite structures up to isomorphism, or testing isomorphism of two given finite structures efficiently, is still an open problem [7].

Candidate evaluation Since WARMR is an inductive logic programming algorithm written within a logic programming environment, the evaluation of all candidate queries is performed inefficiently. Still, WARMR uses several optimizations to increase the performance of this evaluation step, but these optimizations can hardly be compared to the optimized query processing capabilities of relational database systems.

4.2 Phase 2

The procedure to generate all association rules in WARMR, simply consists of finding all couples (Q_1, Q_2) in the list of frequent queries, such that Q_2 is contained in Q_1. We were unable to find how this procedure exactly works, that is, how is each query Q_2 found, given query Q_1. Anyhow, in general, this phase is less of an efficiency issue, since the supports of all queries that need to be considered are already known.

5 Getting WARMeR

Inspired by the framework of WARMR, we present in this section a more general framework and investigate the efficiency challenges described in Section 3. More specifically, we want to discover association rules over all conjunctive queries instead of only those queries contained in a given key conjunctive query since it might not always be clear what exactly needs to be counted. For example, in the beer drinkers database, the examples given in section 2 show that different heads could lead to several interesting association rules about the drinkers, the bars or the beers separately. We also want to exploit the containment relationship of conjunctive queries as much as possible, and avoid situations such as described in example 3. Indeed, the WARMR algorithm does not fully exploit the different levels induced by the containment relationship, since it generates several candidate patterns of different levels within the same iteration.

5.1 Phase 1

The goal of this first phase is to find all frequent conjunctive queries. Hence, $\mathcal{L}$ is the family of all conjunctive queries.

Since only the number of different tuples in the answer of a query is important and not the content of the answer itself, we will extend the notion of query containment, such that it can be better exploited in the levelwise algorithm.

Definition 4. A conjunctive query Q_1 is *diagonally contained* in Q_2 if Q_1 is contained in a projection of Q_2. We write $Q_1 \subseteq^{\Delta} Q_2$.

Example 4.

$$\begin{aligned} Q_1(x) &\coloneq \mathit{likes}(x,y), \mathit{visits}(x,z), \mathit{serves}(z,y) \\ Q_2(x,z) &\coloneq \mathit{likes}(x,y), \mathit{visits}(x,z), \mathit{serves}(z,y) \end{aligned}$$

The answer of Q_1 consists of all drinkers that visit at least one bar that serve at least one beer they like. The answer of Q_2 consists of all visits of a drinker to a bar if that bar serves at least one beer the drinker likes. Obviously, a drinker could visit multiple bars that serve a beer they like, and hence all these bars will be in the answer of Q_2 together with that drinker, while Q_1 only gives the name of that drinker, and hence, the number of tuples in the answer of Q_1 will always be smaller or equal than the number of tuples in the answer of Q_2.

We now define $Q_1 \preceq Q_2$ if $Q_1 \subseteq^{\Delta} Q_2$. The selection predicate q is the minimal support threshold, which is indeed monotone with respect to $\preceq$, i.e., for every instance $\mathbf{I}$ and conjunctive queries Q_1 and Q_2, we have the following: if Q_1 is frequent and $Q_1 \subseteq^{\Delta} Q_2$, then Q_2 is frequent. Notice that the notion of diagonal containment now allows the incorporation of conjunctive queries with different heads within the search space spanned by this specialization relation.

Two issues remain to be solved: how are the candidate queries efficiently generated without generating two equivalent queries? and how is the frequency of each candidate query efficiently computed?

Candidate generation As a first optimization towards the generation of all conjunctive queries, we will already prune several queries in advance.

1. The head of a query must contain at least one variable, since the support of a query with an empty head can be at most 1. Hence, we already know its support after we evaluate a query with the same body but a nonempty head.

2. We allow only a single permutation of the head, since the supports of queries with an equal body but different permutations of the head are equal.

Generating candidate conjunctive queries using the levelwise algorithm requires an initial set of all most general queries with respect to $\subseteq^\Delta$. However, such queries do not exist. Indeed, for every conjunctive query Q, we can construct another conjunctive query Q', such that $Q \subseteq^\Delta Q'$ by simply adding a new atomic formula with new variables into the body of Q, and adding these variables to the head. A rather drastic but still reasonable solution to this problem is to apriori limit the search space to conjunctive queries with at most a fixed number of atomic formulas in the body. Then, within this space, we can look at the set of most general queries, and this set now is well-defined.

At every iteration in the levelwise algorithm we need to generate all candidate conjunctive queries up to equivalence, such that all of their generalizations are known to be frequent. Since an algorithm to generate exactly this set is not known, we will generate a small superset of all candidates and afterwards remove each query of which a generalization is not known to be frequent (or known to be infrequent).

Nevertheless, any candidate conjunctive query is always more specific than at least one query in $\mathcal{F}_i$. Hence, we can generate a superset of all possible candidate queries using the following four operations on each query in $\mathcal{F}_i$.

Extension: We add a new atomic formula with new variables to the body.

Join: We replace all occurrences of a variable with another variable already occurring in the query.

Selection: We replace all occurrences of a variable x with some constant.

Projection: We remove a variable from the head if this does not result in an empty head.

Example 5. This example shows a single application of each operation on the query

$$Q(x, y) \text{ :- } \mathit{likes}(x, y), \mathit{visits}(x, z), \mathit{serves}(z, u).$$

Extension:

$$Q(x, y) \text{ :- } \mathit{likes}(x, y), \mathit{visits}(x, z), \mathit{serves}(z, u), \mathit{likes}(v, w).$$

Join:

$$Q(x, y) \text{ :- } \mathit{likes}(x, y), \mathit{visits}(x, z), \mathit{serves}(z, y).$$

Selection:

$$Q(x, y) \text{ :- } \mathit{likes}(x, y), \mathit{visits}(x, z), \mathit{serves}(z, \text{`Duvel'}).$$

Projection:

$$Q(x) \text{ :- } \mathit{likes}(x, y), \mathit{visits}(x, z), \mathit{serves}(z, u).$$

The following proposition implies that if we apply a sequence of these four operations on the current set of frequent conjunctive queries, we indeed get at least all candidate queries.

Proposition 2. $Q_1 \subseteq^{\Delta} Q_2$ if and only if a query equivalent to Q_1 can be obtained from Q_2 by applying some finite sequence of extension, join, selection and projection operations.

Nevertheless, using these operations, several equivalent or redundant queries can be generated. An efficient algorithm avoiding the generation of equivalent queries is still unknown. Hence, whenever we generate a candidate query, we need to test whether it is equivalent with another query we already generated. In order to keep the generated superset of all candidate conjunctive queries as small as possible, we apply an operator once on each query. If the query is redundant or equivalent with a previously generated query, we repeatedly apply any of the operators until a query is found that is not equivalent with a previously generated query. As already mentioned in the previous section, testing equivalence cannot be done efficiently.

After generating this superset of all candidate conjunctive queries, we need to check for each of them whether all more general conjunctive queries are known to be frequent. This can be done by performing the inverses of the four operations extension, join, selection and projection, as described above. Even if we now assume that in the set of all frequent conjunctive queries there exist no two equivalent queries, we still need to find the query equivalent to the one generated using the inverse operations. Hence, the challenge of testing equivalence of two conjunctive queries reappears.

Candidate evaluation After generating all candidate conjunctive queries, we need to test which of them are frequent. This can be done by simply evaluating every candidate query on the database, one at a time, by translating each query to SQL. Although conjunctive query evaluation in relational databases is well understood and several efficient algorithms have been developed (i.e., join query optimisation and processing) [8], this remains a costly operation. Within database research, a lot of research has been done on multi-query optimization [15]. Here, one tries to efficiently evaluate multiple queries at once. Unfortunately, these techniques are not yet implemented in most common database systems.

As a first optimization towards query evaluation, we can already derive the support of a significant part of all candidate conjunctive queries. Therefore, we only consider those candidate queries that satisfy the following restrictions.

1. We only consider queries that have no constants in the head, because the support of such queries is equal to the support of those queries in which the constant is not in the head.

2. We only consider queries that contain no duplicate variables in the head, since the support of such a query is equal to the support of the query without duplicates in the head.

As another optimization, given a query involving constants, we will not treat every variation of that query that uses different constants as a separate query, but rather we can evaluate all those variations in a single global query. For example, suppose the query

$$Q(x_1) \text{ :- } R(x_1, x_2)$$

is frequent. From this query, a lot of candidate queries are generated using the selection operation on x_2. Assume the active domain of x_2 is $1, 2, \ldots, n$, then the set of candidate queries contains at least

$$\{Q(x_1) \text{ :- } R(x_1, 1), Q(x_1) \text{ :- } R(x_1, 2), \ldots, Q(x_1) \text{ :- } R(x_1, n)\},$$

resulting in a possibly huge amount of queries that need to be evaluated. However, the support of all these queries can be computed by evaluating only the single SQL query

select x_2, **count**$(*)$
from R
group by x_2
having count$(*) \geq$ *minsup*

of which the answer consists of every possible constant c for x_2 together with the support of the corresponding query $Q(x_1) \text{ :- } R(x_1, c)$. From now on, we will therefore use only a symbolic constant to denote all possible selections of a given variable. For example, $Q(x_1) \text{ :- } R(x_1, c_1)$ denotes the set of all possible selections for x_2 in the previous example. A query with such a symbolic constant is then considered frequent if it is frequent for at least one constant.

As can be seen, several optimizations can be used to improve the performance of the evaluation step in our algorithm. Also, we might be able to use some of the techniques that have been developed for frequent itemset mining, such as closed frequent itemsets [14], free sets [2] and non derivable itemsets [3]. These techniques could then be used to minimize the number of candidate queries that need to be executed on the database, but instead we might be able to compute their supports based on the support of previously evaluated queries. Another interesting optimization could be to avoid using SQL queries completely, but instead use a more intelligent counting mechanism that needs to scan the database or the materialized tables only once, and count the supports of all queries at the same time.

5.2 Phase 2

The goal of the second phase is to find for every frequent conjunctive query Q, all confident association rules $Q' \Rightarrow Q$. Hence, we need to run the general levelwise algorithm separately for every frequent query. That is, for any given Q, $\mathcal{L}$ consists of all conjunctive queries Q', such that $Q \subseteq Q'$. Assume we are given two association rules $AR_1 : Q_1 \Rightarrow Q_2$ and $AR_2 : Q_3 \Rightarrow Q_4$, we define $AR_1 \preceq AR_2$ if $Q_3 \subseteq Q_1$ and $Q_2 \subseteq Q_4$.

Likes

Drinker	*Beer*
Allen	Duvel
Allen	Trappist
Carol	Duvel
Bill	Duvel
Bill	Trappist
Bill	Jupiler

Visits

Drinker	*Bar*
Allen	Cheers
Allen	California
Carol	Cheers
Carol	California
Carol	Old Dutch
Bill	Cheers

Serves

Bar	*Beer*
Cheers	Duvel
Cheers	Trappist
Cheers	Jupiler
California	Duvel
California	Jupiler
Old Dutch	Trappist

Figure 1: Instance of the beer drinkers database.

The selection predicate q is the minimal confidence threshold which is again monotone with respect to $\preceq$, i.e., for every instance $\mathbf{I}$ and association rules $AR_1 : Q_1 \Rightarrow Q_2$ and $AR_2 : Q_3 \Rightarrow Q_4$, we have the following: if AR_1 is frequent and confident and $AR_1 \preceq AR_2$, then AR_2 is frequent and confident.

Here, only a single issue remains to be solved: how are the candidate queries efficiently generated without generating two equivalent queries?

We have to generate, for every frequent conjunctive query Q, all conjunctive queries Q', such that $Q \subseteq Q'$ and minimize the generation of equivalent queries. In order to do this, we can use three of the four inverse operations described for the previous phase, i.e., the inverse extension, inverse join and inverse selection operations. We do not need to use the inverse projection operation since we do not want those queries that are diagonally contained in Q, but only those queries that are regularly contained in Q as defined in Section 2. Still, several queries will be generated which are equivalent with previously generated queries, and hence this should again be tested.

6 Sample run

Suppose we are given an instance of the beer drinkers database used throughout this paper, as shown in Figure 1.

We now show a small part of an example run of the algorithm presented in the previous section. In the first phase, all frequent conjunctive queries need to be found, starting from the most general conjunctive queries. Let the maximum number of atoms in de body of the query be limited to 2, and let the minimal support threshold be 2,i.e., at least 2 tuples are needed in the output of a query to be considered frequent. Then, the initial set of candidate queries C_1, consists of the 6 queries as shown in Figure 2. Obviously, the support of each of these queries is 36, and hence, $F_1 = C_1$. To generate all candidate conjunctive queries for level 2, we need to apply the four specialization operations to each of these 6 queries. Obviously, the extension operation is not yet allowed, since this would result in a conjunctive queries with 3 atoms in their bodies. We can apply the Join operation on Q_1, resulting in queries Q_7 and Q_8, as shown in Figure 3. Similarly, the join operation can be applied to Q_4 and Q_6, resulting in Q_9, Q_{10} and Q_{11}, Q_{12} respectively. However, the Join operation is not allowed on Q_2, Q_3 and

$$\begin{array}{l}
\hline
Q_1(x_1,x_2,x_3,x_4) \text{ :- } \mathit{likes}(x_1,x_2), \mathit{likes}(x_3,x_4)\\
Q_2(x_1,x_2,x_3,x_4) \text{ :- } \mathit{likes}(x_1,x_2), \mathit{visits}(x_3,x_4)\\
Q_3(x_1,x_2,x_3,x_4) \text{ :- } \mathit{likes}(x_1,x_2), \mathit{serves}(x_3,x_4)\\
Q_4(x_1,x_2,x_3,x_4) \text{ :- } \mathit{visits}(x_1,x_2), \mathit{visits}(x_3,x_4)\\
Q_5(x_1,x_2,x_3,x_4) \text{ :- } \mathit{visits}(x_1,x_2), \mathit{serves}(x_3,x_4)\\
Q_6(x_1,x_2,x_3,x_4) \text{ :- } \mathit{serves}(x_1,x_2), \mathit{serves}(x_3,x_4)\\
\hline
\end{array}$$

Figure 2: Level 1.

$$\begin{array}{l}
\hline
Q_7(x_1,x_2,x_3) \text{ :- } \mathit{likes}(x_1,x_2), \mathit{likes}(x_1,x_3)\\
Q_8(x_1,x_2,x_3) \text{ :- } \mathit{likes}(x_1,x_2), \mathit{likes}(x_2,x_3)\\
Q_9(x_1,x_2,x_3) \text{ :- } \mathit{visits}(x_1,x_2), \mathit{visits}(x_1,x_3)\\
Q_{10}(x_1,x_2,x_3) \text{ :- } \mathit{visits}(x_1,x_2), \mathit{visits}(x_2,x_3)\\
Q_{11}(x_1,x_2,x_3) \text{ :- } \mathit{serves}(x_1,x_2), \mathit{serves}(x_1,x_3)\\
Q_{12}(x_1,x_2,x_3) \text{ :- } \mathit{serves}(x_1,x_2), \mathit{serves}(x_2,x_3)\\
Q_{13}(x_2,x_3,x_4) \text{ :- } \mathit{likes}(x_1,x_2), \mathit{likes}(x_3,x_4)\\
\vdots\\
Q_{37}(x_1,x_2,x_3) \text{ :- } \mathit{serves}(x_1,x_2), \mathit{serves}(x_3,x_4)\\
\hline
\end{array}$$

Figure 3: Level 2.

Q_5, since for each of them, there always exists a query in which it is contained and which is not yet known to be frequent. For example, if we join x_1 and x_3 in query Q_2, resulting in $Q(x_1, x_2, x_3)$:- *likes*(x_1, x_2), *visits*(x_1, x_3), then this query is contained in $Q'(x_1, x_2, x_4)$:- *likes*(x_1, x_2), *visits*(x_3, x_4), of which the frequency is not yet known. Similar situations occur for the other possible joins on Q_2, Q_3 and Q_5. The selection operation can also not be applied to any of the queries, since for each variable we would select, there always exists a more general query in which that variable is projected, but not selected, and hence, the frequency of such queries is yet unknown. We can apply the projection operator on any variable of queries Q_1 through Q_6, resulting in queries Q_{13} to Q_{37}. In stead of showing the next levels for all possible queries, we will show only single path, starting from query Q_7. On this query, we can now also apply the projection operation on x_3. This results in a redundant atom which can be removed, resulting in the query $Q'_7(x_1, x_2)$:- *likes*(x_1, x_2). Again, for the next level, we can use the projection operation on x_2, now resulting in $Q''_7(x_1)$:- *likes*(x_1, x_2). Then, for the following level, we can use the selection operation on x_2, resulting in the query $Q'''_7(x_1)$:- *likes*$(x_1, \text{‘Duvel’})$. Note that if we had selected x_2, using the constant ‘Trappist’, then the resulting query would not have been frequent and would have been removed for further consideration. If we repeatedly apply the four specialization operations until the levelwise algorithm stops, because no more candidate conjunctive queries could be generated anymore, the second phase can start generating confident association rules from all generated frequent conjunctive queries. For example, starting from query Q'''_7, we can apply the inverse selection operation, resulting in Q''_7. Since both these queries have support 3, the rule $Q''_7 \Rightarrow Q'''_7$ holds with 100% confidence, meaning that every drinker that likes a beer, also likes Duvel, according to the given database.

7 Conclusions and future research

In the future, we plan to study subclasses of conjunctive queries for which there exist efficient candidate generation algorithms up to equivalence. Possibly interesting classes are conjunctive queries on relational databases that consist of only binary relations. Indeed, every relational database can be decomposed into a database consisting of only binary relations. If necessary, this can be further simplified by only considering those conjunctive queries that can be represented by a tree. Note that one of the underlying challenges that always reappears is the equivalence test, which can be computed efficiently on tree structures. Other interesting subclasses are the class of acyclic conjunctive queries and queries with bounded query-width, since also for these structures, equivalence testing can be done efficiently [4].

However, by limiting the search space to one of these subclasses, Proposition 1 is no longer valid, since the intersection of two queries within such a subclass does not neccesserally result in a conjunctive query which is also in that subclass.

Another important topic is the improvement of performance issues for evaluating all candidate queries. Also the problem of allowing flexible constraints to efficiently limit the search space to an interesting subset of all conjunctive queries, is an important research topic.

References

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. 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. MIT Press, 1996.

[2] J-F. Boulicaut, A. Bykowski, and C. Rigotti. Free-sets: a condensed representation of boolean data for frequency query approximation. *Data Mining and Knowledge Discovery*, 2001. To appear.

[3] T. Calders and B. Goethals. Mining all non-derivable frequent itemsets. In *Proceedings of the 6th European Conference on Principles of Data Mining and Knowledge Discovery*, Lecture Notes in Computer Science. Springer-Verlag, 2002. to appear.

[4] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2):211–229, 2000.

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

[6] L. Dehaspe and H. Toivonen. Discovery of relational association rules. In S. Dzeroski and N. Lavrac, editors, *Relational data mining*, pages 189–212. Springer-Verlag, 2001.

[7] S. Fortin. The graph isomorphism problem. Technical Report 96-20, University of Alberta, Edmonton, Alberta, Canada, July 1996.

[8] H. Garcia-Molina, J. Ullman, and J. Widom. *database system implementation.* Prentice-Hall, 2000.

[9] D. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining.* MIT Press, 2001.

[10] A. Inokuchi and H. Motoda T. Washio. An apriori-based algorithm for mining frequent substructures from graph data. In *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, volume 1910 of *Lecture Notes in Computer Science*, pages 13–23. Springer-Verlag, 2000.

[11] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 313–320. IEEE Computer Society, 2001.

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

[13] S.H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1997.

[14] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proceedings of the 7th International Conference on Database Theory*, volume 1540 of *Lecture Notes in Computer Science*, pages 398–416. Springer-Verlag, 1999.

[15] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, volume 29:2 of *SIGMOD Record*, pages 249–260. ACM Press, 2000.

[16] J.D. Ullman. *Principles of database and knowledge-base systems, volume 2*, volume 14 of *Principles of Computer Science.* Computer Science Press, 1989.

[17] M. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the Eight ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM Press, 2002. to appear.