ELSEVIER

Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Artificial Intelligence 149 (2003) 61–87

Artificial Intelligence

www.elsevier.com/locate/artint

Metaqueries: Semantics, complexity, and efficient algorithms ☆

Rachel Ben-Eliyahu-Zohary [a,*], Ehud Gudes [b], Giovambattista Ianni [c]

[a] *Department of Communication Systems Engineering, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel*

[b] *Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel*

[c] *DEIS, Università della Calabria, 87036 Rende (CS), Italy*

Received 10 January 2002

Abstract

Metaquery (metapattern) is a data mining tool which is useful for learning rules involving more than one relation in the database. The notion of a metaquery has been proposed as a template or a second-order proposition in a language $\mathcal{L}$ that describes the type of pattern to be discovered. This tool has already been successfully applied to several real-world applications.

In this paper we advance the state of the art in metaquery research in several ways. First, we argue that the notion of a *support* value for metaqueries, where a support value is intuitively some indication to the relevance of the rules to be discovered, is not adequately defined in the literature, and, hence, propose our own definition. Second, we analyze some of the related computational problems, classify them as NP-hard and point out some tractable cases. Third, we propose some efficient algorithms for computing support and present preliminary experimental results that indicate the usefulness of our algorithms.

Keywords: Knowledge discovery; Data mining; Metaqueries; Support

☆ This is an extended and revised version of a paper that appeared in IJCAI-99 under the name "Towards Efficient Metaqueries".

* Corresponding author.

E-mail addresses: rachel@bgumail.bgu.ac.il (R. Ben-Eliyahu-Zohary), ehud@cs.bgu.ac.il (E. Gudes), ianni@deis.unical.it (G. Ianni).

0004-3702/$ – see front matter
doi:10.1016/S0004-3702(03)00073-0

1. Introduction

With the tremendous growth in information availability, data mining is emerging as a vital research area among AI and database communities [13]. Metaquerying [18,20], also known as metapattern or metarule-guided mining [15,16,21], is a promising approach for data mining in relational or deductive databases. Metaqueries serve as a generic description of the class of patterns to be discovered and help guide the process of data analysis and pattern generation. Unlike many other discovery systems, e.g., association rules [1], patterns discovered using metaqueries can link information from many tables in the database. These patterns are relational, whereas most machine-learning systems [17] are capable of learning only propositional patterns.

Metaqueries can be specified by human experts or, alternatively, they can be automatically generated from the database schema. Either way, they serve as an important interface between human "discoverers" and the discovery system.

A metaquery has the form

$$T \leftarrow L_1, \ldots, L_m$$

where T and L_i are literal schemes. A literal scheme S has the form $Q(Y_1, \ldots, Y_n)$ where all non-predicate variables Y_k are implicitly universally quantified. The expression $Q(Y_1, \ldots, Y_n)$ is called a *relational pattern* (of arity n). The predicate variable Q can be instantiated only to a predicate symbol of the specified arity n. The instantiation must be done in a way that is consistent with the variable names.

For example, consider a database having the relations depicted in Fig. 1. Letting P, Q, and R be variables for the predicates, the metaquery

$$R(X, Z) \leftarrow P(X, Y), Q(Y, Z)$$

specifies that the patterns to be discovered are relations

$$r(X, Z) \leftarrow p(X, Y), q(Y, Z)$$

where p, q, and r are specific predicates. One possible result of this metaquery on the database in Fig. 1 is the rule

$$\text{stud-dep}(X, Z) \leftarrow \text{stud-course}(X, Y), \text{course-dep}(Y, Z)$$

student	course
Ron	Calculus1
Dana	Algebra
Ana	Intro to CS

course	department
Calculus1	mathematics
Algebra	mathematics
Intro to CS	computer science

student	department
Ron	mathematics
Dana	mathematics
Ana	computer science

Fig. 1. The relations stud-course, course-dep, and student-dep.

which means intuitively that if a student takes a course from a certain department he must be a student of that department.

Each answer to a metaquery is a rule accompanied by two values: the *support* value and the *confidence* value. The thresholds for the support and confidence values are provided by the user. Intuitively, the *support* value indicates how frequently the body of the rule is satisfied, and the *confidence* value indicates what fraction of the tuples which satisfies the body also satisfies the head. Similar to the case of association rules [1], the notions of *support* and *confidence* have two purposes: to avoid presenting negligible information to the user and to cut off the search space by early detection of low support and confidence values.

To understand what a metaquery is and why it is significant we can compare it with the known data mining technique called "mining of association rules". Roughly speaking, an association rule is a rule $X \rightarrow Y$, where X and Y are sets of items [1]. The association rule is generated using a transaction database (i.e., a set of sets of items), and, like metaqueries, interestingness is measured using suitable concepts of interestingness such as support and confidence values. With respect to metaquerying techniques, association rules are widely studied in the literature in terms of efficient evaluation (e.g., [2,7]), and maintenance (e.g., [6,8]). Although both techniques may artfully simulate the other, and, in some sense, metaquerying can be considered as an extension of association rule mining on multiple tables [15], the kind of patterns which can be learned using association rules is more specific, and requires data stored on a single table, whereas metaquerying takes advantage of relational schemes. Metaquerying can be directly applied to native relations and/or to existing views as well, while association rule learning systems often need preprocessing steps to denormalize data on a single table.

In this paper we advance the state of the art in research on metaqueries in several ways. First, we argue that the definition of a support value given in the literature [19] is inadequate, and we provide a new one. Second, we analyze the complexity of answering metaqueries. We show that, in general, answering a metaquery is an NP-hard task. Even if we are focusing on the problem of instantiating a metaquery in the process of answering it, we find out that this task by itself is NP-hard for typed databases. However, we identify several tractable classes: answering a fixed query on a database with a fixed schema (but varying relation size) can be done in polynomial time; the task of instantiating a metaquery is polynomial when the database is un-typed.

Third, we concentrate on the task of computing the support value of a given rule generated by some metaquery. This task by itself is NP-hard, and we present several heuristics for computing the support value. The algorithms based on these heuristics are, in general, exponential but work in polynomial time when the rules are what we call "sparse". We also present and discuss experiments on the algorithms that we have developed.

The paper is organized as follows. In Section 2 we define what a metaquery is and what comprises an answer to a metaquery. Section 3 discusses the notions of support and confidence values. In Section 4 we analyze the computational complexity of some decision problems related to answering metaqueries. Algorithms for computing support values are given in Section 5, together with some experiments that indicate their usefulness. Concluding remarks are given in Section 6.

2. Preliminaries

Let U be a countable domain of constants, and T ($T \cap U = \emptyset$) a countable set of *data types*. Given a set $D \subseteq U$, a *typed relation* R of arity $a(R)$ is a pair $\langle R^0, T^0 \rangle$, where $R^0 \subseteq D^{a(R)}$ and T^0 is a list of data types $\langle t_0, \ldots, t_{a(R)} \rangle$ where each $t_i \in T$. We denote as $R[i]$ the type of the ith attribute of R. Unless confusion arises, a relation R of data types $\langle t_0, \ldots, t_n \rangle$ may also be denoted as $R[t_0, \ldots, t_n]$. A database is *un-typed* if $|T| = 1$.

A *database* $\mathcal{DB}$ is $(D, R_1, \ldots, R_n)$ where $D \subseteq U$ is finite, and each R_i is a typed relation of arity $a(R_i)$.

A *metaquery* $\mathcal{MQ}$ has the form

$$H \leftarrow L_1, \ldots, L_m \tag{1}$$

where H and L_i are literal schemes. A literal scheme S has the form $Q(Y_1, \ldots, Y_n)$ where each Y_k $(1 \leqslant k \leqslant n)$ is either a constant symbol $c \in D$ or a (non-predicate) variable. All non-predicate variables are implicitly universally quantified. Whenever Q is a (predicate) variable, the expression $Q(Y_1, \ldots, Y_n)$ is called a *relational pattern* (of arity n), otherwise it is called an *atom*. For example, $R(X, Y)$ is a relational pattern whereas $r(X, Y)$ is an atom referring to the relation r.

The right-hand side $L_1, \ldots, L_m$ is called the *body* of the metaquery, and H is called the *head* of the metaquery. A metaquery is called *pure* if it does not contain any constant symbol $c \in D$. Semantics of metaqueries with constants is defined by translating them to pure metaqueries in the following way:

Definition 2.1. Given a metaquery $\mathcal{MQ}$ and a database $\mathcal{DB} = (D, R_1, \ldots, R_n)$, the *purified version* $(\mathcal{MQ}', \mathcal{DB}')$ of $\mathcal{MQ}$ and $\mathcal{DB}$ is obtained by the following steps for each constant symbol $c \in D$ occurring in $\mathcal{MQ}$:

(1) Replace each occurrence of c in $\mathcal{MQ}$ with a fresh variable X_c;
(2) Add to $\mathcal{DB}$ a unary relation $R_c(X) = \{(c)\}$;
(3) Add the atom $R_c(X_c)$ to the body of $\mathcal{MQ}$.

For simplicity of exposition, unless otherwise specified, henceforth we assume that we are dealing with pure metaqueries, and/or with purified versions (coupled with suitable databases) where constant symbols are purged.

Letting $\mathcal{DB}$ be a database and $\mathcal{MQ}$ a metaquery, we introduce the following notations:

- $rel(\mathcal{DB})$ denotes the set of relation names of $\mathcal{DB}$;
- $ato(\mathcal{DB})$ denotes the set of all the atoms of the form $p(Y_1, \ldots, Y_k)$ where $p \in rel(\mathcal{DB})$; k is the arity of p and each Y_i is a non-predicate variable;
- $rep(\mathcal{MQ})$ denotes the set of relational patterns of $\mathcal{MQ}$.

Definition 2.2. An *instantiation* (also called *an answer*) for a $\mathcal{MQ}$ is a mapping $\sigma : rep(\mathcal{MQ}) \rightarrow ato(\mathcal{DB})$, for which the following conditions hold:

(1) If $\sigma(P(X_1,\ldots,X_n)) = r(Y_1,\ldots,Y_k)$ then $n = k$, and $Y_1,\ldots,Y_k$ is a permutation of $X_1,\ldots,X_n$.
(2) If $\sigma(P(X_1,\ldots,X_n)) = r(Y_1,\ldots,Y_n)$ and $\sigma(P(Z_1,\ldots,Z_n)) = s(W_1,\ldots,W_n)$, then $s = r$.
(3) Letting C be the co-domain of σ: then, for each pair of atoms $r(Y_1,\ldots,Y_n) \in C$, $q(X_1,\ldots,Y_m) \in C$, and for each $i,\ j,\ 1 \leqslant i \leqslant n,\ 1 \leqslant j \leqslant m$ s.t. $Y_i = X_j$, $r[i]$ must be equal to $q[j]$.

This kind of instantiation, first introduced in [3] as a *type*-1 instantiation, formalizes a class of instantiation methods often (and implicitly) employed in practice [19,20]. Relaxing in some way condition (1) allows other types of instantiations, e.g., type-2 instantiations, another common way of instantiating metaqueries [3], where arity of generated atoms is greater than or equal to the corresponding relational patterns. We denote as $MQ\sigma$ the Horn rule generated by applying the instantiation σ to all the relational patterns of the metaquery MQ. Note that the presence of permutation of variables introduces more substantial differences with respect to the usual conjunctive queries. In Section 5.1 we explain how finding an instantiation for a metaquery can be regarded as solving a constraint satisfaction problem.

As an example (as briefly discussed in the introduction), consider a database having the relations depicted in Fig. 1 (for now, we assume there are no different data types within this database). Letting P, Q, and R be variables for the predicates, the metaquery

$$R(X,Z) \leftarrow P(X,Y), Q(Y,Z) \tag{2}$$

specifies that the patterns to be discovered are rules

$$r(X,Z) \leftarrow p(X,Y), q(Y,Z)$$

where p, q, and r are specific predicates. Two different instantiations σ_1 and σ_2 of this metaquery could produce the rules

$$\text{stud-dep}(X,Z) \leftarrow \text{stud-course}(X,Y), \text{course-dep}(Y,Z), \tag{3}$$

$$\text{stud-dep}(X,Z) \leftarrow \text{stud-course}(Y,X), \text{course-dep}(Y,Z). \tag{4}$$

Rule (3) means intuitively that if a student takes a course from a certain department, he must be a student of that department, whereas rule (4) has no precise intuitive meaning. However, it would be also generated, if no tighter data types specification was given. The difference between the two patterns lies in the attributes bounded to the variables (in rule (4), the variable Y in the atom stud-course(Y,X) is bound to the attribute *student* of the relation *stud-course*, while in rule (3) Y is bound to the attribute *course* of the same relation). Note that while rule (4) is a possible instantiation to metaquery (2), it would probably be absent from the final set of rules generated by the system because it would have had a low *support value*. In the next section we discuss the concepts of *support value* and *confidence value*.

student	course
Ron	Calculus1
Dana	Algebra
Ana	Intro to CS
David	Intro to CS

course	department
Calculus1	mathematics
Algebra	mathematics
Intro to CS	computer science

student	department
Ron	mathematics
Dana	mathematics
Ana	computer science
David	mathematics

Fig. 2. The new university database.

3. The notion of *support* and *confidence* for metaqueries

Suppose that we are given the metaquery (2) again, but instead of the university database shown in Fig. 1, we have the university database shown in Fig. 2 (where relations student-course and student-department are different, but the relation course-department is the same).

In this case rule (3) no longer holds in all cases (David takes a course from the CS department but he is a student of the department of Mathematics), but we can say that it holds in 75% of the cases, or in other words, that it has a *confidence* value of 0.75 (the rule holds for three out of four students).

Let us now consider another set of relations from an (still un-typed) employee database of an Israeli high-tech company having 1000 employees. The company is located in Beer Sheva, and all the employees except Ana live in Beer Sheva. Ana lives nearby in Kibbutz Shoval. None of the employees, except Guy, were born in the area. Guy was born in Kibbutz Shoval and Ana is Guy's boss.

Now suppose that we pose the following metaquery:

$$R(X, Y) \leftarrow P(X, Z), Q(Y, Z). \tag{5}$$

An answer to this metaquery can be:

$$\text{boss}(X, Y) \leftarrow \text{empl-born}(X, Z), \text{empl-lives}(Y, Z) \tag{6}$$

with a confidence value equal to 1.0. Thus we are going to learn the rule that if a second employee lives in the same place where a first employee was born, the first employee must be the boss of the second. But this rule is useless because it is based on very weak evidence—only two people out of the 1000 who work in this company. Rules in which the rule body is satisfied by only a very low fraction of the relations involved should be avoided; in other words, we want rules with a high *support* value.

Hence, each answer to a metaquery is a rule accompanied by two numbers: *support* value and *confidence* value. The thresholds for the support and confidence values are provided by the user. Intuitively, the *support* value indicates how frequently the body of the rule is satisfied, and the *confidence* value indicates what fraction of the tuples which satisfy

the body also satisfy the head. Similar to the case of association rules [1], the notions of *support* and *confidence* have two purposes: to avoid giving negligible information to the user and to cut off the search space by early detection of low support and confidence values.

Formally, given a rule

$$t(\ldots) \leftarrow r_1(\ldots), \ldots, r_m(\ldots), \tag{7}$$

let J denote the relation that is the equijoin[1] of $r_1, \ldots, r_m$, and let Jt be the relation that is the equijoin of J and t. Where y and x are some relations, let y_x be the projection of y over the attributes that are common to y and x, and let $|x|$ be the number of tuples in x. For each i, $i = 1, \ldots, m$, define S_i to be the fraction $|J_{r_i}|/|r_i|$. The *support value* of rule (7) is the maximum over S_i, $i = 1, \ldots, m$. Less formally, for each r_i we define S_i to be the fraction of r_i that can be obtained by projecting J on the attributes of r_i. The support value of the rule is the maximum S_i over $i = 1, \ldots, m$.

The *confidence value* of (7) is the fraction of t that appears in J. That is, the *confidence value* of (7) is $|(Jt)_J|/|J|$.

The support value that we have defined has the following useful property:

Claim 3.1. *For any two relations r_i and r_j in rule* (7) *that have at least one common attribute variable, S_i is not bigger than the fraction of r_i that participates in the equijoin between r_i and r_j. Or, formally $S_i \leqslant |(r_i \bowtie r_j)_{r_i}|/|r_i|$.*

This property enables us to get an upper bound on the support value of rule (7) by performing pairwise equijoins instead of one equijoin of all the relations in the body of the rule.

Shen et al. [20] are, to best of our knowledge, the first to present a framework that uses metaqueries to integrate inductive learning methods with deductive database technologies. The confidence measure that we use is similar to theirs. However, we take issue with their definition of a support value, and develop our own. According to Shen and Leng [19], the support value (which they refer to as "base value") for rule (7) would be the fraction $|J|/\prod_{i=1}^{m}|r_i|$. Although the computation of support according to our definition is slightly more complex, we believe that our definition reflects better the intuitive meaning of *support*. Consider again the student database of Fig. 1, and suppose that there are 100,000 tuples in the relation student-course and 1000 tuples in the relation course-department. According to Shen and Leng, the support value for rule (3) would be $10^5/10^8 = 1/1000$. The support value will shrink even more as the number of courses grow, no matter how many students are in the relation student-course. This low support value will render rule (3) uninteresting, although it involves all the tuples in the relation student-course. Note that according to our definition, the support value for this rule is 1.0, because there is one relation, *student-course*, which participates in the equijoin of the body of the rule to full capacity.

[1] The equijoins are accomplished by constricting values of attributes that are bound to the same variable name in the metaquery to be equal.

4. Complexity

How hard is the problem of answering a metaquery? In this section we will show that the computation task is NP-hard. First, let us rephrase the problem as a decision problem.

Definition 4.1 (*The MQ Problem*).

Instance: A database $\mathcal{DB}$ and a metaquery $\mathcal{MQ}$ of the form (1).
Question: Is there an instantiation σ for $\mathcal{MQ}$ in $\mathcal{DB}$ such that the rule $\mathcal{MQ}\sigma$ holds with support and confidence values greater than 0?

Theorem 4.2. *The MQ Problem is NP-hard.*

Proof. We will show that CLIQUE can be reduced to the MQ Problem. For convenience, we state the CLIQUE Problem [14]: given an undirected graph $G = (V, E)$ and a positive integer k, determine whether there is a clique in the graph of size at least k. Suppose we are given an undirected graph $G = (V, E)$ where $|V| = n$. We can assume that $n \geqslant k$ (otherwise the answer is obviously "NO"). We construct a database $\mathcal{DB}$ and a metaquery $\mathcal{MQ}$ as follows. $\mathcal{DB}$ will have two relations: r_V and r_E. r_V contains exactly one tuple, which is the list of all vertices in V, and r_E is the set of all edges in E, that is:

$$r_V = \big\{(v_1, v_2, \ldots, v_n)\big\}, \quad \text{where } V = \{v_1, \ldots, v_n\}, \quad \text{and}$$
$$r_E = \big\{(u, v) \mid (u, v) \in E\big\}.$$

Note that $\mathcal{DB}$ is un-typed.

$\mathcal{MQ}$ will be the following metaquery:

$$\begin{aligned} P(X_1, X_2, \ldots, X_n) \leftarrow & \\ & Q(X_1, X_2), Q(X_1, X_3), \ldots, Q(X_1, X_k), \\ & Q(X_2, X_3), Q(X_2, X_4), \ldots, Q(X_2, X_k), \\ & \vdots \\ & , Q(X_{k-1}, X_k). \end{aligned}$$

We claim that the answer to the MQ Problem composed of the above $\mathcal{DB}$ and $\mathcal{MQ}$ will be "YES" iff there is a clique in G of at least size k. Indeed, if there is such a clique in G, then the instantiated rule (which maps P to r_V and Q to r_E)

$$\begin{aligned} r_V(X_{a(1)}, X_{a(2)}, \ldots, X_{a(n)}) \leftarrow & \\ & r_E(X_1, X_2), r_E(X_1, X_3), \ldots, r_E(X_1, X_k), \\ & r_E(X_2, X_3), r_E(X_2, X_4), \ldots, r_E(X_2, X_k), \\ & \vdots \\ & , r_E(X_{k-1}, X_k) \end{aligned} \tag{8}$$

(where a encodes a suitably chosen permutation of $\langle 1 \ldots n \rangle$) holds with positive support and confidence values. Therefore the answer to the given MQ Problem will be "YES". On the other hand, if the answer to this MQ Problem is YES, then, since a) r_V has only one tuple and r_E is the only binary relation, and b) if a tuple $(u, v) \in r_E$ then $(v, u) \in r_E$ as well, there exists an instantiation σ (whose restrictions on the predicate variables map P to r_V and Q to r_E), for which $\mathcal{MQ}\sigma$ scores a positive value for support and confidence values and it is like rule (8). Referring to the single tuple of r_V, we can trivially build a mapping

$$\{X_1 \mapsto v_{a^{-1}(1)}, \ldots, X_k \mapsto v_{a^{-1}(k)}\}$$

that, clearly, encodes a clique in G of size k. □

It should be noted that the metaquery employed in this proof is unsafe, i.e., some non-predicate head variables do not occur in the body. This setting is possible when we deal with metaqueries. As an example, the following metaquery

$$R(X, Y, A, B) \leftarrow P(X, Y) \tag{9}$$

could be employed to ask if there exists a quaternary relation in which some binary relation is partially encoded. A safeness constraint would artificially reduce the set of possible interesting patterns that metaqueries allow us to specify.

However, there are cases where the MQ Problem is tractable. Consider a fixed database scheme (we allow the size of relations to vary, however), and a fixed metaquery. In this case every metaquery has a constant number of instantiations and hence can be answered by a join of a constant number of relations. A further investigation of the complexity of metaqueries can be found in [3]. In that paper, a computational characterization of other types of metaqueries are discussed, but typed databases are not considered.

At this point we might ask whether the complexity analysis done on inductive logic programs (e.g., [12]) is relevant here. The answer is no. In inductive logic programming the goal is to construct a program that will *generate* a given goal relation out of other relations and positive and negative examples. Here, we are interested in finding out to what extent some rule on some given relations holds. We do not have to find a rule that accurately generates the goal relation, and we do not have negative examples.

We now want to nail down the problem of establishing the existence of a suitable instantiation (irrelevant of the corresponding value of support and confidence). The question is whether this task alone is intractable or not. It turns out that the problem is intractable if data types are allowed.

First, let us formalize the problem:

Definition 4.3 (*The Instantiation Problem*).

Instance: A database $\mathcal{DB}$ and a metaquery $\mathcal{MQ}$.
Question: Does there exist an instantiation σ for $\mathcal{MQ}$ on $\mathcal{DB}$?

Theorem 4.4. *The Instantiation Problem on an instance $\langle \mathcal{DB}, \mathcal{MQ} \rangle$ is NP-complete, whenever $\mathcal{DB}$ provides a fixed set of data types T where $|T| \geqslant 2$.*

Proof. (Hardness) The NP-complete problem SET SPLITTING [14] can be reduced to the instantiation Problem. The SET SPLITTING Problem is stated as follows: "Given a collection C of subsets of a finite set S, does there exist a partition of S (which we call *splitting*) into two subsets S_1 and S_2 such that no subset in C is entirely contained in either S_1 or S_2?".

Let C be a collection of subsets of the set S. The problem remains NP-complete if we assume that each element $c \in C$ has $1 < |c| \leqslant 3$ [14]. We build a database $\mathcal{DB}_C$ as follows:

- We introduce two data types a and b;
- We introduce a binary relation $b_1[a, b]$ (we do not care about the extension of the relations introduced), and the ternary relations $t_1[a, a, b]$ and $t_2[b, b, a]$.

Intuitively, the above three typed relations represent all the possible ways of splitting a set of two or three elements.

Then, we define the metaquery $\mathcal{MQ}_C$, which will contain a binary atom for each two-element set in C and a ternary atom for each three-element set of C, as follows:

- We introduce a predicate variable P_c for each $c \in C$, and a variable X_v for each $v \in S$.
- Let $c = \{q, r\}$ be a two-element set in C. Then $\mathcal{MQ}_C$ will contain the relational pattern $P_c(X_q, X_r)$.
- Let $c = \{q, r, s\}$ be a three-element set in C. Then $\mathcal{MQ}_C$ will contain the relational pattern $P_c(X_q, X_r, X_s)$.

Roughly speaking, $\mathcal{DB}_C$ is built in such a way that data type a constants in the body of any rule obtained from the metaquery belong to S_1, where constants having data type b belong to S_2. $\mathcal{MQ}_C$ encodes C itself, while the relational pattern in the head of $\mathcal{MQ}_C$ can be anything.

We claim that C has a splitting iff $\langle \mathcal{DB}_C, \mathcal{MQ}_C \rangle$ has an instantiation.

($\Rightarrow$) Assume that C has a splitting $\langle S_1, S_2 \rangle$, where $S_2 = S - S_1$. It is then possible to build a valid instantiation σ_C for $\mathcal{MQ}_C$ on $\mathcal{DB}_C$, as follows:

- Let $c = \{q, r, s\}$ be a three-element set in C. Since S_1 and S_2 represent a splitting for C, then either $|S_1 \cap c| = 2$ or $|S_1 \cap c| = 1$. In the former case, let $S_1 \cap c = \{q, s\}$ and $S_2 \cap c = \{r\}$; let $P_c(X_q, X_s, X_r)$ (obviously, the order of q, r, s does not matter) be the relational pattern corresponding to c in $\mathcal{MQ}_C$. Then σ_C is defined to be $\sigma_C(P_c(X_q, X_s, X_r)) = t_1(X_q, X_s, X_r)$. In the latter case, let $S_1 \cap c = \{s\}$ and $S_2 \cap c = \{q, r\}$: then σ_C is defined to be $\sigma_C(P_c(X_q, X_s, X_r)) = t_2(X_s, X_q, X_r)$.
- Let $c = \{q, r\}$ be a two-element set in C. Since S_1 and S_2 represent a splitting for C, then $|S_1 \cap c| = 1$. Let $S_1 \cap c = \{r\}$ and $S_2 \cap c = \{q\}$; let $P_c(X_r, X_q)$ be the relational pattern corresponding to c in $\mathcal{MQ}_C$. Then σ_C is defined to be $\sigma_C(P_c(X_r, X_q)) = b_1(X_r, X_q)$.

It is straightforward that σ_C satisfies constraints (1), (2) and (3) of Definition 2.2.

($\Leftarrow$) Consider a suitable instantiation σ_C for $\mathcal{MQ}_C$ and $\mathcal{DB}_C$. Clearly, σ_C encodes a valid splitting of C.

(Membership) Straightforward. □

Theorem 4.5. *The Instantiation Problem on an instance* $\langle \mathcal{DB}, \mathcal{MQ} \rangle$ *can be solved in quadratic time and logarithmic space* (*w.r.t. the number of atoms of* $\mathcal{MQ}$ *and the number of relations of* $\mathcal{DB}$), *whenever* $\mathcal{DB}$ *is un-typed.*

Proof (*Sketch*). The proof is quite straightforward. Let $\mathcal{DB}$ be an un-typed database, and $\mathcal{MQ}$ be a metaquery. We can design an algorithm $\mathcal{A}$, which builds an instantiation σ for $\mathcal{MQ}$ and $\mathcal{DB}$, using two (logarithmic space) counters D and M, where D acts as a pointer to the current relation considered in $\mathcal{DB}$ and M acts as a pointer to the current literal of $\mathcal{MQ}$.

Since there are no type constraints, for each literal $Q(X_1, \ldots, X_n)$ of $\mathcal{MQ}$, $\mathcal{A}$ scans the schema of $\mathcal{DB}$ using D, and if $Q(X_1, \ldots, X_n)$ is a relational pattern, then σ is constructed assigning the first relation r of $\mathcal{DB}$, having the same arity as Q, to $Q(X_1, \ldots, X_n)$, and assigning $X_1, \ldots, X_n$ to attributes of r using any permutation (if there is no such relation the algorithm stops immediately with a NO answer). Thus, conditions (1), (2) and (3) of Definition 2.2 are fulfilled. In particular, condition (2) holds since further patterns $Q(X'_1, \ldots, X'_n)$ having the same predicate variable Q, will be deterministically matched to the (same) first relation of arity n occurring on the input tape. □

5. Efficient algorithms for metaqueries

In this section we discuss algorithms for generating all rules resulting from a given metaquery.

The process of answering a metaquery can be divided into two stages. In the first stage, which we call the *instantiation stage*, we are looking for sets of relations that match the pattern determined by the metaquery. In the second stage, which we call the *filtration stage*, we filter out all the rules that match the pattern of the metaquery but which do not have enough support and confidence values.

5.1. The instantiation stage

As shown in Theorem 4.5, the Instantiation Problem *per se* is intractable (unless P = NP). The process of instantiating a metaquery is similar to solving a Constraint Satisfaction Problem (CSP) [10] where we are basically looking for all solutions of the CSP problem. We recall briefly what a CSP problem is.

Definition 5.1. An instance of a constraint satisfaction problem consists of a finite set of variables V, a finite domain of values U, and a finite set of *constraints* C. A constraint is a pair $\langle S, r \rangle$, where S is a list of m variables and r is an m-ary relation over U. The tuples of r specify the allowed combinations of simultaneous values for the variables of S. A solution to a CSP instance is a substitution ρ from V to U, such that for each $\langle S, r \rangle \in C$, $S\rho \in r$.

Given a database $\mathcal{DB}$ and a metaquery $\mathcal{MQ}$, the Instantiation Problem can be translated to a CSP instance as follows (we assume here, for simplicity, that $\mathcal{MQ}$ does not contain atoms):

(1) The set of variables V is the union of the set P, the set O, and the set O^t. P is the set of all predicate variables occurring in $\mathcal{MQ}$. O contains a set of variables of the form $Q_{i,j,X}$. Each $Q_{i,j,X}$ is associated to the jth occurrence of a variable X within the ith relational pattern of $\mathcal{MQ}$ having Q as the predicate variable. O^t contains variables of the form $Q^t_{i,j,X}$ constructed in the same way. The $Q_{i,j,X}$'s variables store the relation name and the position of the attribute to which the X variable is bound within the literal Q, whereas the variables of the form $Q^t_{i,j,X}$ store the data type bound to X.

(2) W.l.o.g. we associate with each group of variables a different domain. Variables in P range over the set R of all the relation names in $\mathcal{DB}$. Variables in O^t range over the set T, which is the set of all data types of $\mathcal{DB}$. Variables in O range over the set A. The set A contains a variable a_i for each relation $a \in \mathcal{DB}$ and for each i, $1 \leqslant i \leqslant n$, where n is the arity of a (i.e., each value a_i represents the ith attribute of the relation a).

(3) The set of constraints C is constructed in the following way:

- Consider each variable $Q_{i,j,X} \in O$. Let Q be the corresponding predicate variable in P, and $Q^t_{i,j,X}$ be the corresponding variable in O^t. Then C contains the constraint $\langle (Q, Q_{i,j,X}, Q^t_{i,j,X}), r\rangle$, where

$$r = \big\{\langle q, q_k, t_k\rangle \mid q \in R, q_k \in A, t_k \in T, 1 \leqslant k \leqslant \text{arity}(q), \text{ and } t_k \text{ is the type of the } k\text{th attribute of } q\big\}.$$

  This kind of constraint means that whenever a predicate variable Q is bound to a given relation name q, variables of the kind $Q_{i,j,X}$, belonging to atoms with Q as the predicate variable, have to be bound to some attribute of q, whereas the corresponding variable $Q^t_{i,j,X}$ must be assigned to the corresponding data type.

- Consider each pair of variables $Q_{i,j,X}$, $Q_{i,k,Y}$ originating from the same literal pattern and the corresponding predicate variable Q belonging to P. We add the constraint $\langle (Q, Q_{i,j,X}, Q_{i,k,Y}), r\rangle$ where

$$r = \big\{(q, q_g, q_h) \mid q \in R, q_g, q_h \in A, 1 \leqslant g, h \leqslant \text{arity}(q), g \neq h\big\}.$$

  This kind of constraint ensures that there is exactly one variable bound to each attribute of each relation.

- Consider each variable X of $\mathcal{MQ}$. Let $O_X = \{Q^{1,t}_{a,b,X}, \ldots, Q^{m,t}_{c,d,X}\}$ be the set of all the variables corresponding to X in O^t. We add to C a constraint $\langle (Q^{1,t}_{a,b,X}, \ldots, Q^{m,t}_{c,d,X}), r\rangle$, where r is a $|O_X|$-ary relation and $r = \{\langle t, \ldots, t\rangle \mid t \in T\}$. This set of constraints forces all the variables of O^t, corresponding to an occurrence of the same variable of $\mathcal{MQ}$, to be bound to the same data type.

For example, consider a database $\mathcal{DB} = \langle D, r, s, t\rangle$, where D is the domain of $\mathcal{DB}$, and r, s and t are binary relations, whose types are $\langle$`int`, `char`$\rangle$, $\langle$`char`, `int`$\rangle$, and $\langle$`char`, `char`$\rangle$, respectively. Suppose we are given the metaquery

$$\mathcal{MQ} = R(X, Z) \leftarrow S(X, Y), T(Y, Z),$$

and we want to solve the Instantiation Problem for $\mathcal{MQ}$ over $\mathcal{DB}$. The problem can be translated to a CSP problem as follows:

- The variable set V is
$$V = \{R, S, T\} \cup \{R_1, R_2, S_1, S_2, T_1, T_2\} \cup \left\{R_1^t, R_2^t, S_1^t, S_2^t, T_1^t, T_2^t\right\};$$
- The domain of R, S and T is $\{r, s, t\}$;
- The domain of $\{R_1, R_2, S_1, S_2, T_1, T_2\}$ is $\{r_1, r_2, s_1, s_2, t_1, t_2\}$. As an example, $S_2 = t_1$ would mean that the occurrence of the variable Y within the atom $S(X, Y)$ is bound to the first attribute of the relation t.
- The domain of $\{R_1^t, R_2^t, S_1^t, S_2^t, T_1^t, T_2^t\}$ is $\{\texttt{char}, \texttt{int}\}$.
- Let c be the relation
$$\begin{aligned} c = \big\langle & (r, r_1, \texttt{int}) \\ & (r, r_2, \texttt{char}) \\ & (s, s_1, \texttt{char}) \\ & (s, s_2, \texttt{int}) \\ & (t, t_1, \texttt{char}) \\ & (t, t_2, \texttt{char}) \big\rangle. \end{aligned}$$
Then, we introduce the constraints $\langle (R, R_1, R_1^t), c\rangle$, $\langle (R, R_2, R_2^t), c\rangle$, $\langle (S, S_1, S_1^t), c\rangle$, $\langle (S, S_2, S_2^t), c\rangle$, $\langle (T, T_1, T_1^t), c\rangle$ and $\langle (T, T_2, T_2^t), c\rangle$;
- Let d be the relation
$$\begin{aligned} d = \big\langle & (r, r_1, r_2) \\ & (r, r_2, r_1) \\ & (s, s_1, s_2) \\ & (s, s_2, s_1) \\ & (t, t_1, t_2) \\ & (t, t_2, t_1) \big\rangle. \end{aligned}$$
Then, we introduce the constraints $\langle (R, R_1, R_2), d\rangle$, $\langle (S, S_1, S_2), d\rangle$, $\langle (T, T_1, T_2), d\rangle$;
- Let e be the relation
$$\begin{aligned} e = \big\langle & (\texttt{char}, \texttt{char}) \\ & (\texttt{int}, \texttt{int}) \big\rangle \end{aligned}$$
then, we introduce the constraints $\langle (R_1, S_1), e\rangle$, $\langle (R_2, T_2), e\rangle$, $\langle (S_2, T_1), e\rangle$ (R_1 and S_1 are occurrences of X, R_2 and T_2 are occurrences of Z, and S_2 and T_1 are occurrences of Y, respectively).

In our experiments, we use a very simple CSP algorithm (forward checking with Backjumping [9]) but other, more advanced algorithms, may be used. The basic instantiation algorithm employs some heuristics which deal with this particular type of

***Instantiate*(*SR*)**

Input: SR: a set $R_0, R_1, \ldots, R_n$ of relational patterns, partially instantiated, each with a set of constraints $C_0, \ldots, C_n$. Initially the constraints sets are all empty.

Output: Relations $r_0, r_1, \ldots, r_n$ that match the relational patterns in SR and a consistent binding of the variables in SR to attributes.

1. If SR is completely instantiated then return SR.
2. Pick the next uninstantiated relation variable R_i from SR, {here we should use CSP *variable* ordering techniques}.
3. Pick up the next possible instantiation r to R_i (r should meet the constraints in C_i and should be of the same arity as R_i). {Here we should use CSP *value* ordering techniques}.
4. For each relational pattern R_j not yet instantiated do:

   if R_j has a common attribute variable X with R_i, add to C_j the constraint that X must be bound to an attribute with type T_X, where T_X is the type of the attribute that X is bound to in r.
5. Call ***Instantiate*** recursively.

Fig. 3. Algorithm for the instantiation stage.

CSP instance (see Fig. 3). If the instantiation process succeeds, each relational pattern $R(X_1, \ldots, X_n)$ that appears in the metaquery is instantiated. That is, R is bound to some relation named r and each variable is bound to an attribute of the relation. We assume that a procedure $att(r, X)$ can return the attribute in r to which the variable X is bound.

5.2. The filtration stage

The filtration stage is composed of two steps: filtering out rules with low support values, and filtering out rules with low confidence values. So far we have focused on algorithms that compute the support value. The task of deciding if the support value and/or the confidence value is greater than 0 on a (pre-computed) instantiation turns out to be NP-hard, since it can be easily reduced to the problem of satisfiability for a conjunctive query on a given input database [5]. For convenience, we state the problem of satisfiability of conjunctive queries:

Definition 5.2. A *conjunctive query* is a set of atoms $\{r_1(X_1), \ldots, r_n(X_n)\}$, where $X_1, \ldots, X_n$ are lists of variables and/or constants. Let $\mathcal{DB}$ be a database instance. The problem of satisfying a conjunctive query (Boolean Conjunctive Query satisfaction problem, or BCQ) is the problem of deciding whether there exists a substitution ρ for variables in X_i, $1 \leqslant i \leqslant n$, such that for each i, $1 \leqslant i \leqslant n$, $r_i(\rho(X_i)) \in \mathcal{DB}$.

We discuss three alternatives for computing the support value:

The join approach (the straightforward way) Computing the equijoin of the body of the rule, then computing S_i (S_i as defined in Section 3) for each relation in the body, and then taking a maximal S_i where i ranges over all the relations in the body.

The histogram approach Using histograms for estimating the support value. Computing the support value using the join approach only for rules with high estimated support value.

The histogram + memory approach This is the same as the histogram approach, except that we store intermediate results in memory, and reuse them when we are called to make the same computation.

The straightforward way to calculate the support value is to compute the support value S_i of each relation by performing the join as explained in Section 3, and then taking $\text{Max}\{S_i \mid 1 \leqslant i \leqslant m\}$. This is done by the join approach mentioned above. Since the join is an expensive operation, we try to detect rules with low support values using some low-cost procedures. The other two approaches compute an upper bound on the support value and then compute the exact support value only for rules with a high enough upper bound of the support value. The idea is summarized in algorithm *compute-support* in Fig. 4. Note that once one relation with a high S_i is found, the procedure join-support$(r_1, \ldots, r_m)$ is called. This procedure simply computes the exact support value using join.

The procedure S_i-upbound called by the algorithm *compute-support* returns an upper bound for the value S_i for a single relation r_i in the body of the rule. This can be done by one of the two procedures: S_i-upbound-brave or S_i-upbound-cautious, shown in Figs. 5 and 6, respectively. The basic idea is that an upper bound on S_i can be achieved by taking the join of the relation r_i with any other relation with which r_i has variables in common (see Claim 3.1 in Section 3). Procedure S_i-upbound-brave does this by picking one arbitrary relation with which r_i has a common variable, and procedure S_i-upbound-cautious does this by first considering *all* relations with which r_i has variables in common, and then taking the minimum. Procedure S_i-upbound-cautious takes more time than procedure S_i-upbound-brave but it achieves a tighter upper bound and hence can save more join computations. An experimental evaluation of the two approaches is presented in Section 5.3.

---

compute-support$(r_1, \ldots, r_m,$ ***MinSupport***)

Input: Set of relations $r_1, \ldots, r_m$, where each of the attributes is bound to a variable. A support threshold *MinSupport*.

Output: If the rule whose body is $r_1, \ldots, r_m$ has a support value equal or larger than *MinSupport*, return the support value, otherwise return -1.

1. $RelSetCopy = RelSet = \{r_1, \ldots, r_m\}$; $LowSupp =$ **true**;

2. While ($RelSet \neq \emptyset$) **and** *LowSupp* do

1. Let $r \in RelSet$;
2. $s = S_i$-upbound$(r, RelSetCopy)$;
3. **if** $s \geqslant MinSupport$ then $LowSupp =$ **false**
   else $RelSet = RelSet - \{r\}$;

3. If *LowSupp* then return -1
else return join-support$(r_1, \ldots, r_m)$

Fig. 4. Computing the support value for a rule body.

S_i-**upbound-brave**(r_i, R)
Input: A relation r_i and a set of relations R.
Output: An upper bound on S_i for a rule whose body is $R \cup \{r_i\}$.
1. $s = 1.0$;
2. If there is $r' \in R$ such that r' and r_i have a common variable X then
$s = \text{upbound}(r_i, r', att(r_i, X), att(r', X))$;
3. Return s;

Fig. 5. Computing S_i bravely.

S_i-**upbound-cautious**(r_i, R)
Input: A relation r_i and a set of relations R.
Output: An upper bound on S_i for a rule whose body is $R \cup \{r_i\}$.
1. $s = 1.0$;
2. for each relation r' in R such that r_i and r' have variables in common
do for each common variable X of r_i and r'
do
$s' = \text{upbound}\big(r_i, r', att(r_i, X), att(r', X)\big)$;
if $s' < s$ then $s = s'$;
3. Return s;

Fig. 6. Computing S_i cautiously.

The procedure *upbound* called by procedure S_i-upbound-cautious or procedure S_i-upbound-brave finds an upper bound on S_i for a given relation r_i using one of two approaches: the histogram approach (Figs. 7 and 8) or the histogram + memory approach (Fig. 9). The role of procedure $att(r, X)$ is explained in Section 5.1.

The histogram approaches exploit the fact that histograms are easy to construct and are quite useful for support estimation. A histogram of an attribute of some relation is a mapping h between the set of values that this attribute can take and the set of natural numbers, such that for each possible value v, $h(v)$ is the number of tuples in the relation in which the value of this particular attribute is v. Suppose that we compute an equijoin of two relations r_1 and r_2, matching attribute a_1 of r_1 with attribute a_2 of r_2, where a_1 and a_2 are of the same type. In order to find out whether a certain tuple of r_1 in which $a_1 = v$ is in this equijoin, we can look at the histogram h of a_2 in the relation r_2 and check whether $h(v) > 0$. This is the basic idea behind using histograms for computing the support value's upper bound.

Given two relations r_1 and r_2, procedure upbound-histo shown in Fig. 7 computes an upper bound on the support value of the equijoin between them. Procedure upbound-histo prepares the histograms and then calls the procedure Histo (Fig. 8) which actually computes the support value.

The problem with the procedure upbound-histo is that it does not exploit the fact that pairs of instantiated relations can appear again and again in many different instantiations of the same metaquery. In the procedure upbound-histo-mem, shown in Fig. 9, we save

upbound-histo(r_1, r_2, att_1, att_2)

Input: Two relations r_1 and r_2; att_1—an attribute of r_1, att_2—an attribute of r_2. att_1 and att_2 are of the same type.

Output: An upper bound on the support value where only r_1 and r_2 are considered.

1. Let U be the set of all distinct values in att_1 of r_1 and att_2 of r_2.

2. Let h_1 be a histogram with domain U of the values in att_1 of r_1. (If there is no such histogram, build it.)

3. Let h_2 be a histogram with domain U of the values in att_2 of r_2. (If there is no such histogram, build it.)

4. Return Histo($r_1, r_2, h_1, h_2, |U|$);

Fig. 7. Computing the support value using histograms.

Histo(r_1, r_2, h_1, h_2, n)

Input: Two relations r_1, r_2, and two histograms h_1, h_2 reflecting how many times a value of some common attribute appears in each of them, respectively. We assume that there are n distinct possible values.

Output: An upper bound on the support value where only r_1 and r_2 are considered.

1. Set support1 $= 0$; support2 $= 0$;

2. For $i = 1$ to n

if $h_1(i) > 0$ **and** $h_2(i) > 0$ then

1. support1 = support1 + $h_1(i)$
2. support2 = support2 + $h_2(i)$;

3. Return max(support1/size(r_1), support2/size(r_2));

Fig. 8. The procedure Histo.

upbound-histo-mem(r_1, r_2, att_1, att_2)

Input: Two relations r_1 and r_2, att_1 an attribute of r_1, att_2 an attribute of r_2. att_1 and att_2 are of the same type.

Output: An upper bound on the support value where only r_1 and r_2 are considered.

Note: This procedure uses a procedure fetch-supp-mem(r_1, r_2, att_1, att_2) which returns the estimated support value for these two relations on these attributes as recorded in memory. If no such estimate is recorded, it returns -1. Similarly, the procedure put-supp-mem($s, r_1, r_2, att_1, att_2$) stores the estimated support value s for these two relations on these attributes in memory.

1. s = fetch-supp-mem(r_1, r_2, att_1, att_2);

2. If $s = -1$ then

a. s = upbound-histo(r_1, r_2, att_1, att_2);

b. put-supp-mem($s, r_1, r_2, att_1, att_2$);

3. return s;

Fig. 9. Computing the support value using histograms and memo-ing.

in memory estimated support value of pairs of relations and retrieve this information if necessary.

5.3. Evaluation

The efficiency of algorithm *compute-support* depends on the likelihood of finding a rule with a high support value. If a large fraction of the rules have a high support value, then this algorithm will take more time than the straightforward algorithm which computes the support value by performing join without trying to estimate the result before. Note that for rules with a high support value, algorithm *compute-support* takes more time than the algorithm which performs a join directly because it first has to estimate the support value, find out that it is high, and only then calls the join procedure.

Our working assumption is that rules with high support are much less likely to exist. In any case, the above analysis calls for an experimental evaluation of the algorithms. In this section we present some very preliminary results on such experiments. The evaluation was done on the StudentGrades database supported by the FlexiMine system [4]. This database contains information on students and some of their demographic characteristics, courses, and grades. The relevant tables for our experiment are: the *student* table with 997 rows, the *course* table with 1403 rows, the *family* table with 997 rows, and the *student-course* table with 20705 rows. Each table contains between two and four attributes.

We have compared the performance of our algorithms by measuring the time it took them to compute the support value of 20 different rules involving four relations each. All the rules were instances of the same metaquery. The experiments were done on a Sun/SunOS workstation with one SPARC CPU and 128 MB main memory.

The time (real and CPU), in seconds, for the support value computation was measured for the following configurations:

(1) The support value is computed by performing the join (Procedure join-support);
(2) The support value is computed by histograms without memo-ing[2] (Procedure upbound-histo in Fig. 7);
(3) The support value is computed by histograms using memo-ing (Procedure upbound-histo-mem in Fig. 9).

All the above methods were tested using either the Procedure S_i-upbound-brave (Fig. 5) or Procedure S_i-upbound-cautious (Fig. 6).

Figs. 10 and 11 show the results obtained when the support value threshold was set to 0.5, and the support value was computed using S_i-upbound-cautious and S_i-upbound-brave, respectively. The time is measured in seconds. The columns in the tables are as follows:

num—the serial number of the rule,
conf—the confidence value of the rule.

[2] "memo-ing" is a general term for approaches that use various data structures to remember salient inferred facts to save work. For an example taken from the logic programming literature, see [11].

num	conf	Sjoin	Shisto	join real	join CPU	JHM real	JHM CPU	Histo real	Histo CPU	HM real	HM CPU	Body rows
0	0.157	0.127	0.307	7	0.120	25	0.100	22	0.150	21	0.100	939
1	0.272	0.098	0.242	15	0.150	26	0.110	23	0.180	21	0.110	536
2	0.233	0.394	0.397	279	0.170	44	0.120	41	0.210	41	0.120	7292
3	0.281	0.395	0.432	620	0.170	44	0.120	41	0.230	41	0.120	7427
4	0.103	0.226	0.549	657	0.180	81	0.140	41	0.250	42	0.130	1475
5	0.146	0.467	0.736	2081	0.180	1505	0.140	41	0.270	42	0.130	14719
6	0.138	0.059	0.211	2431	0.200	1506	0.150	42	0.310	42	0.140	288
7	0.112	0.120	0.208	2835	0.200	1506	0.160	42	0.330	43	0.150	1017
8	0.273	0.329	0.397	3369	0.200	1506	0.170	42	0.380	43	0.160	3537
9	0.185	0.381	0.316	3987	0.210	1506	0.170	42	0.410	43	0.160	7681
10	0.1	0.225	0.549	4170	0.230	1690	0.180	43	0.450	43	0.160	1475
11	0.141	0.467	0.736	4616	0.240	2136	0.190	43	0.510	43	0.160	14719
12	0.671	0.394	0.397	5506	0.250	2155	0.210	62	0.570	63	0.180	7926
13	0.684	0.395	0.432	6379	0.260	2155	0.210	62	0.660	63	0.180	7798
14	0.141	0.480	0.736	7630	0.280	3406	0.240	62	0.740	63	0.190	15724
15	0.263	0.330	0.397	8022	0.290	3407	0.250	63	0.770	63	0.200	3769
16	0.176	0.383	0.339	8551	0.290	3407	0.260	63	0.820	63	0.210	8372
17	0.136	0.480	0.736	9150	0.290	4006	0.260	63	0.870	63	0.210	15724
18	0.698	0.389	0.300	9495	0.310	4021	0.290	78	0.930	79	0.240	4472
19	0.709	0.394	0.432	9816	0.310	4021	0.300	78	1.000	69	0.250	4443

Fig. 10. Comparison of cautious support value computations for threshold of 0.5.

Sjoin—the support value computed according to the definition, i.e., first performing the join and then computing the support value. Since the join is a very expensive operation that requires a lot of computation time, we have used the following simple heuristics to compute the join. We took the biggest relation (in terms of number of tuples) r in the rule, randomly chose 10% of its tuples, and then made all the computations as if r is this 10% of the original relation r. We believe that this is a reasonable heuristics because the goal of computing the support value is to give some estimate on the relevance of the rule, and using only 10% of the largest relation can give us enough information for judging how significant the rule is.

Shisto—the *estimated* support value (upper bound) computed by Procedure *compute-support* (Fig. 4) using the *cautious* approach (Fig. 6).

Shisto Brave—the *estimated* support value (upper bound) computed by Procedure *compute-support* (Fig. 4) using the *brave* approach (Fig. 5).

join-real—the real time to compute the support value by the join method, *accumulated* (by "accumulated" we mean that for each instantiation we record the time obtained by adding the time it took to answer that particular instantiation and the time it took to compute the support value for all previous instantiations. For example, the time recorded in Fig. 10 in line number 3 in the column "join-real" is equal to $7+8+264+341=620$).

join-CPU—the CPU time to compute the support value by the join method, *accumulated*.

JHM-real—the real time to compute the *exact* support value using our Histogram with memory method, *accumulated*, including the time for computing and building the histograms. Note that the exact support value was not computed in the cases where

num	Shisto Brave	JHM real	JHM CPU	Histo real	Histo CPU	HM real	HM CPU
0	0.397	3	0.115	3	0.100	3	0.150
1	0.432	3	0.115	3	0.110	3	0.160
2	0.397	5	0.125	5	0.160	5	0.170
3	0.432	5	0.125	5	0.170	5	0.170
4	0.736	28	0.140	5	0.170	5	0.170
5	0.736	1297	0.140	6	0.190	5	0.170
6	0.397	1298	0.145	6	0.200	5	0.170
7	0.397	1299	0.150	6	0.200	5	0.170
8	0.397	1299	0.150	6	0.220	5	0.170
9	0.397	1299	0.150	6	0.240	5	0.170
10	0.736	1319	0.160	6	0.260	5	0.170
11	0.736	1754	0.180	6	0.290	5	0.170
12	0.397	1754	0.180	8	0.310	7	0.190
13	0.432	1754	0.180	8	0.350	7	0.190
14	0.736	3196	0.190	8	0.370	7	0.190
15	0.397	3196	0.190	9	0.380	7	0.190
16	0.397	3196	0.190	9	0.390	7	0.190
17	0.736	3840	0.210	9	0.400	7	0.190
18	0.397	3842	0.210	11	0.440	9	0.210
19	0.432	3842	0.210	11	0.480	9	0.210

Fig. 11. Comparison of Brave support value computations for threshold of 0.5.

the upper-bound support value was smaller than the support value threshold. In the table in Fig. 11, the time measured was the real time to compute the exact support value using our Histogram with Memory method in brave computations, while in the experiments summarized in Fig. 10 we used cautious computations.

JHM-CPU—same as JHM-real, but CPU time is measured here instead of real time.

Histo-real—the real time to compute the support value upper-bound by the histogram method (brave computing in Fig. 11 and cautious in Fig. 10), *accumulated*. Including time for building and computing the histograms.

Histo-CPU—same as Histo-real, but CPU time is measured instead of real time.

HM-real—the real time to compute the support value upper-bound by the histogram with memory method (brave computing in Fig. 11 and cautious in Fig. 10), *accumulated*. Including time for building and computing the histograms.

HM-CPU—same as HM-real, but CPU time is measured instead of real time.

Body-rows—the number of tuples in the body of the Equijoin.

We have collected the data for support thresholds of 0.3–0.6, and so we have eight tables similar to the tables in Figs. 11 and 10, two for each threshold. The results are summarized in Figs. 12–18.

In Figs. 12 and 13 we see the growth in CPU time and real time of computing the upper-bound of the support value as the support threshold grows. CPU time and real time grow as the support threshold increases because these procedures stop calculating the upper-bound of the support value once they reach a value that is equal to or higher than the threshold. The higher the threshold is, the longer it takes to pass it. In Fig. 12 we see that saving

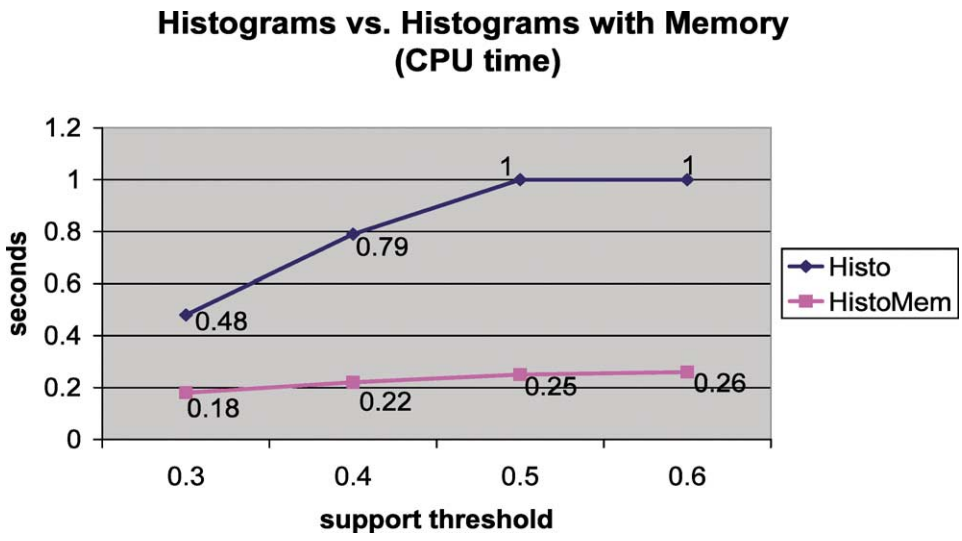

Fig. 12. CPU time of computing the support value's upper-bound using histograms with and without memo-ing.

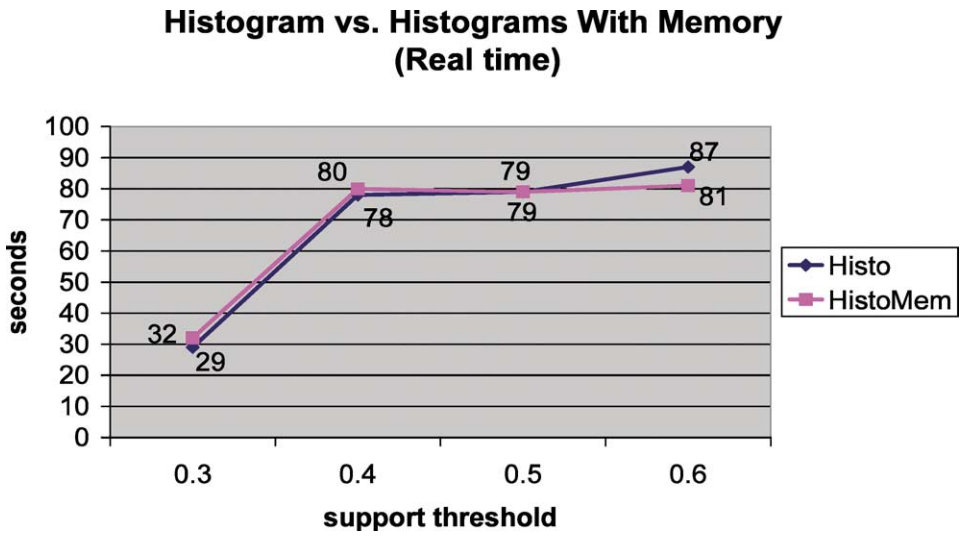

Fig. 13. Real time of computing the support value's upper-bound using histograms with and with out memo-ing.

the histograms in memory indeed leads to between 60% and 75% savings in CPU time. Looking at Fig. 13, however, it is evident that there is not much difference in real time between the two methods. This can be explained by the fact that most of the real time goes to disk I/O, and there is not much difference between the two approaches in this respect.

In Fig. 14 we can see the difference between the real time for computing the support value by performing join (using Procedure join-support), and the real time it takes to

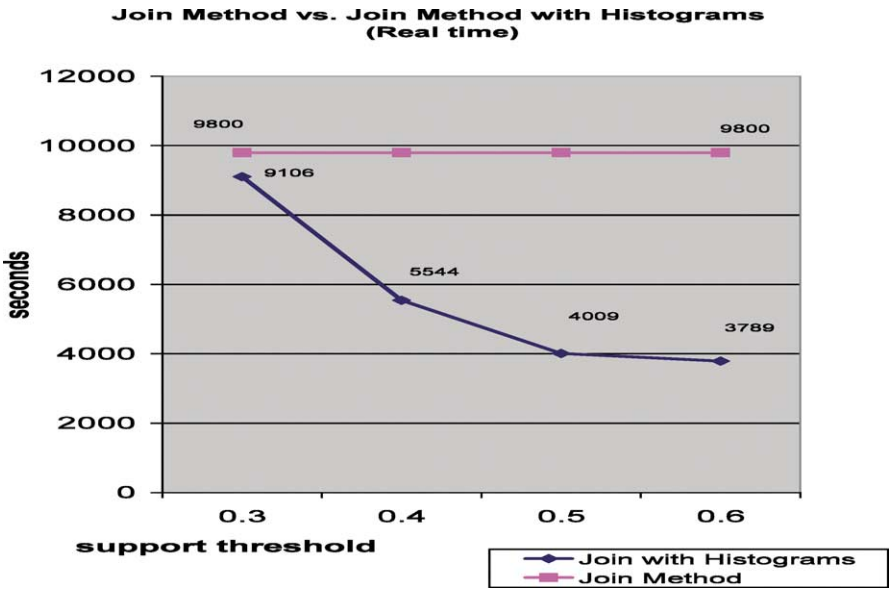

Fig. 14. Real time of computing the exact support value using histograms with memo-ing vs. using join.

compute the support value using histograms and memo-ing. According to the histogram approach, the heavy computation of the join support value takes place only when the upper bound calculated is above the support value threshold. Hence, the higher the threshold is, the less likely it is to pass it and so the real time is decreasing as the support value threshold grows. The real time of computing the support value using the join does not change as the support threshold grows because the same computation and disk access is done no matter what the threshold is. It is evident from Fig. 14 that we can achieve over 38% saving in real time if we use the histogram + memo-ing approach. Note that the computation of Sjoin which was done only for the experimental evaluation, gave much better results to the join approach, since only 10% of the largest table was considered. Therefore the actual results support the approximate method even more than reported. Note also that the fact that only 10% of the largest relation was considered explains a discrepancy, e.g., at line 9 in Fig. 10, where the histogram support value, which is supposed to be an upper bound, is less than the join-support. This happens as a result of the estimated computation of Sjoin as explained above.

The next series of graphs illustrate the experiments done on comparing brave and cautious computations of the support value bound.

Fig. 15 shows the CPU time of computing the support value's upper-bound as a function of the support threshold for both cautious and brave computations. It is evident that brave computation using histograms with memo-ing is the more efficient of the two in terms of CPU time. However, in Fig. 16 we can see that there was not much difference in real time in computing the support value using the cautious or the brave approach (both using histograms saved in memory), and both were more efficient than the join method.

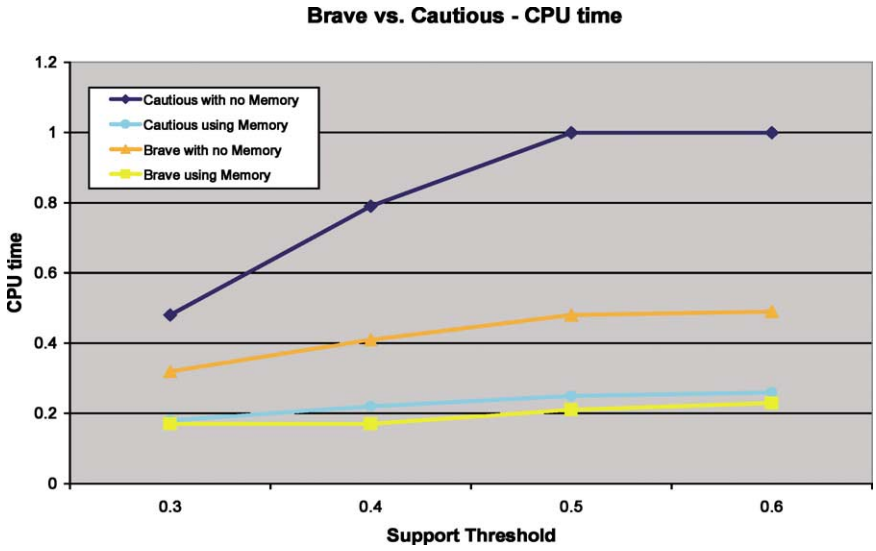

Fig. 15. CPU time of computing the support value's upper-bound using the brave vs. cautious approach with and without histograms + memo-ing.

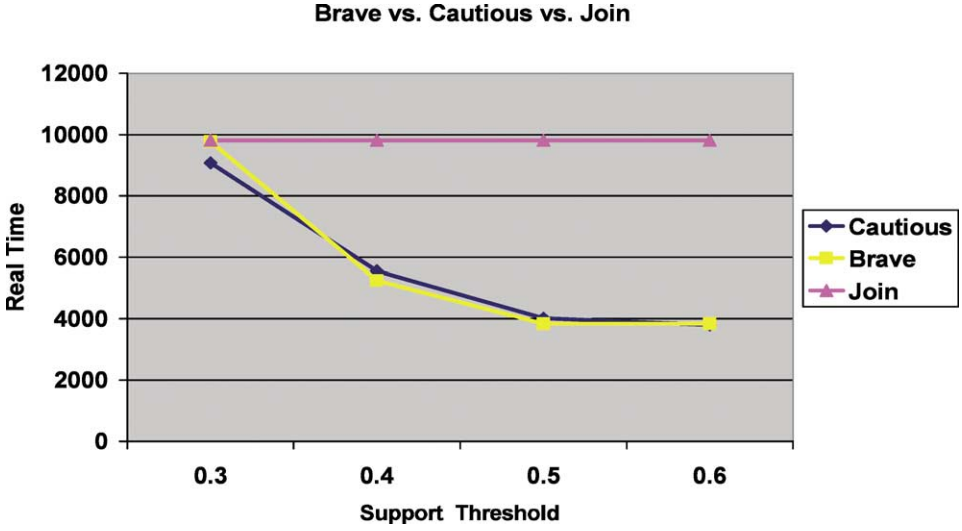

Fig. 16. Real time of brave vs. cautious vs. join.

Another way to compare brave and cautious computations is to see how many times we have computed a high upper bound which turned out to be a "false alarm". We call a "false alarm", or a "miss", a situation where the computed upper bound is higher than the threshold while the exact support value is lower than the threshold. Clearly, for every metaquery, the number of misses in brave computations will be always greater than or

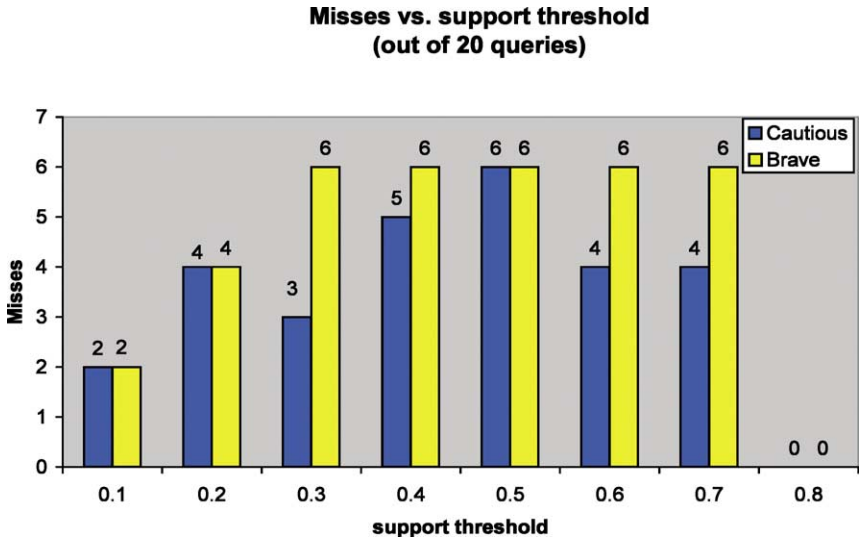

Fig. 17. Number of redundant join computations ("misses").

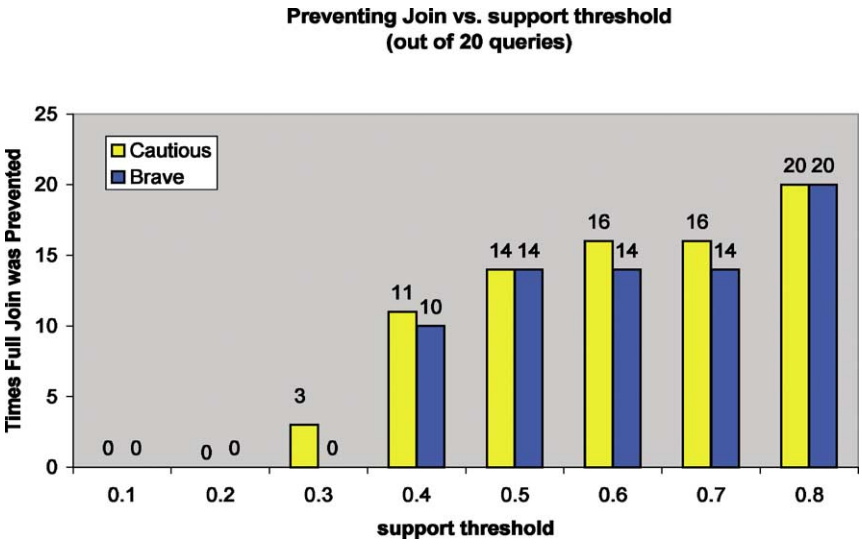

Fig. 18. Number of times join was not computed because the upper bound was low.

equal to the number of misses in cautious computation. The misses in our case study are illustrated in Fig. 17.

We were also interested in checking how many times we have computed an upper bound which was lower than the threshold. In such cases, we did not have to perform the expensive computation of the join. In general, for every metaquery, the number of joins prevented by brave computations will always be lower than or equal to the number of joins prevented in

cautious computation. The number of times join was saved in our case study is illustrated in Fig. 18.

The experiments reported above indicate that cautious reasoning using histograms with memo-ing is sometimes the superior method and can even be better than brave reasoning using histograms without memo-ing. Brave computation of the upper bound is a bit faster but at the cost of redundant join computations.

6. A tractable case for support value computation

When an instantiation of a metaquery yields what we call a *sparse rule*, the support value can be computed quite efficiently using histograms. Sparse rules are obtained from metaqueries in which each relational pattern in the body has at most one attribute variable common with some other relational pattern in the body. Let us first give a precise definition.

Definition 6.1 (*Sparse rule*). Let δ be a rule obtained from some instantiation σ on a metaquery MQ. The *join graph* of δ is an undirected graph $G_\delta = (V_\delta, E_\delta)$ defined as follows:

(1) $V_\delta =$ The set of all predicate names in the body of δ, and
(2) $E_\delta = \{(v, u) |\ v, u \in V_\delta, v$ and u have a variable in common in $MQ\}$.

A rule is *sparse* if and only if the maximum degree of nodes in its join graph is 1.

Example 6.2. Rule (6) from Section 3:

$$\text{boss}(X, Y) \leftarrow \text{empl-born}(X, Z), \text{empl-lives}(Y, Z)$$

is sparse, since the two predicates in the body share only the variable Z. The following rule:

$$\text{boss}(X, Y) \leftarrow \text{empl-born}(X, Z), \text{empl-lives}(Y, Z)\text{empl-lives}(X, Z)$$

is not sparse because *empl-born* and the second occurrence of *empl-lives* share two variables.

It is easy to see that metaqueries that yield sparse rules after instantiation can be recognized in time linear in the size of the metaquery.

The exact support value of a sparse rule δ can be computed in time linear in the size of the relations involved in the rule by performing the following steps:

(1) For each relation r in the body of δ:
   (a) Let r' be the relation with which r shares a variable X in the MQ that yielded δ (there is at most one such r'. If there is no such r' then do nothing).
   (b) Let *att* be the attribute bound to X in r and *att*$'$ the attribute bound to X in r'.
   (c) Let s be the value returned by calling procedure *upbound-histo* shown in Fig. 7 with the parameters r, r', att, att'.

The Problem	Complexity	Tractable Subsets
The MQ Problem	NP-hard	fixed metaquery and fixed database scheme
The Instantiation Problem	NP-complete	un-typed database
The Filtration Problem	NP-hard	sparse rules

Fig. 19. Complexity classes that were identified in this paper.

(2) The support value of δ is the maximum s computed in the previous step.

The correctness and complexity of the above procedure is quite obvious given that rule δ is sparse. Note that sparse rules might be quite common because many relations have only one attribute in common, and if we have only two relations in a body of a rule it is quite likely that it is a sparse rule.

7. Conclusion

This paper contributes to the research on metaqueries in several ways. First, we have analyzed the complexity of the related computational problems and pointed out some tractable subsets. The complexity classification done in this paper is summarized in Fig. 19. Second, we have proposed a new notion of a support value for a rule generated according to a pattern. Third, we have presented novel and efficient algorithms for computing support values. Although more experimental work is needed for real evaluation of the algorithms we have developed, preliminary experimental evaluation is quite promising.

Several research topics remain open in the field of data mining using metaqueries. One interesting direction is to develop a system that will generate the metaqueries automatically. A simple way to do this is to go over all the possible combinations. A more sophisticated approach would be to learn in which direction the interesting information can be found from answers to preliminary metaqueries. Another open research problem is to develop parallel algorithms for computing answers for metaqueries. Some of these issues are currently being investigated by us and others will be dealt with in future research.

Acknowledgements

We would like to thank the anonymous referees for their thoughtful and detailed comments that helped improve the paper considerably. Many thanks to Arkady Shapiro for running the experiments and for providing useful comments on earlier drafts of this paper. We are grateful also to Fabrizio Angiulli who found a mistake in an earlier proof of Theorem 4.2. This research was supported by a grant from the Israeli Ministry of Science.

References

[1] R. Agrawal, T. Imielinski, A.N. Swami, Mining association rules between sets of items in large databases, in: P. Buneman, S. Jajodia (Eds.), Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, ACM Press, New York, 1993, pp. 207–216.

[2] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: J.B. Bocca, M. Jarke, C. Zaniolo (Eds.), Proceedings of 20th International Conference on Very Large Data Bases, VLDB'94, Santiago de Chile, Chile, Morgan Kaufmann, San Mateo, CA, 1994, pp. 487–499.
[3] F. Angiulli, R. Ben-Eliyahu-Zohary, G.B. Ianni, L. Palopoli, Computational properties of metaquerying problems, ACM Transactions on Computational Logic (TOCL) 4 (2) (2003) 149–180. A short version in: Proceedings of PODS-2000, Dallas, TX, 2000, pp. 237–244.
[4] R. Ben-Eliyahu-Zohary, C. Domshlak, E. Gudes, N. Liusternik, A. Meisels, T. Rosen, S.E. Shimony, FlexiMine—A flexible platform for KDD research and application construction, Ann. Math. Artificial Intelligence (2003), in press. A short version in: Proceedings of the Fourth International Conference on Knowledge Discovery in Databases (KDD-98), New York, 1998, pp. 184–188.
[5] A.K. Chandra, P.M. Merlin, Optimal implementation of conjunctive queries in relational data bases, in: Conference Record of the Ninth Annual ACM Symposium on Theory of Computing, Boulder, CO, 1977, pp. 77–90.
[6] D.W.-L. Cheung, J. Han, Maintenance of discovered knowledge: A strategy for updating association rules (abstract), in: Proceedings of the DOOD'95 Post-Conference Workshops on Integration of Knowledge Discovery in Databases with Deductive and Object-Oriented Databases (KDOOD) and Temporal Reasoning in Deductive and Object-Oriented Databases (TDOOD), Singapore, 1995.
[7] D.W.-L. Cheung, J. Han, V. Ng, A.W.-C. Fu, Y. Fu, A fast distributed algorithm for mining association rules, in: Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, Miami Beach, FL, IEEE Computer Society, 1996, pp. 31–42.
[8] D.W.-L. Cheung, S.D. Lee, B. Kao, A general incremental technique for maintaining discovered association rules, in: R.W. Topor, K. Tanaka (Eds.), Database Systems for Advanced Applications'97, Proceedings of the Fifth International Conference on Database Systems for Advanced Applications (DASFAA), Melbourne, Australia, in: Advanced Database Research and Development Series, Vol. 6, World Scientific, Singapore, 1997, pp. 185–194.
[9] R. Dechter, Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition, Artificial Intelligence 41 (1990) 273–312.
[10] R. Dechter, Constraint networks, in: S.C. Shapiro (Ed.), Encyclopedia of Artificial Intelligence, 2nd Edition, Wiley, New York, 1992, pp. 276–285.
[11] S.W. Dietrich, Extension tables: Memo relations in logic programming, in: Proceedings of the Symposium on Logic Programming, San Francisco, CA, 1987, pp. 264–272.
[12] S. Dzeroski, S. Muggleton, S. Russell, Pac-learnability of determinate logic programs, in: Proceedings of the Fifth ACM Workshop on Computational Learning Theory, New York, 1992, pp. 128–135.
[13] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, R. Uthurusamy, Advances in Knowledge Discovery and Data Mining, AAAI Press/MIT Press, Cambridge, MA, 1996.
[14] M.R. Garey, D.S. Johnson, Computers and Intractability, A Guide to the Theory of NP-Completeness, W.H. Freeman, New York, 1979.
[15] M. Kamber, J. Han, J.Y. Chiang, Metarule-guided mining of multi-dimensional association rules using data cubes, in: Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining, Newport Beach, CA, 1997.
[16] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, A.I. Verkamo, Finding interesting rules from large sets of discovered association rules, in: Proceedings of 3rd Internat. Conf. on Information and Knowledge Management, Gaithersburg, MD, 1994, pp. 401–408.
[17] T.M. Mitchell, Machine Learning, McGraw-Hill Higher Education, New York, 1997.
[18] W.M. Shen, Discovering regularities from knowledge bases, Intelligent Systems 7 (7) (1992) 623–636.
[19] W.-M. Shen, B. Leng, A metapattern-based automated discovery loop for integrated data mining-unsupervised learning of relational patterns, IEEE Trans. Knowledge Data Engineering 8 (6) (1996) 898–910.
[20] W.-M. Shen, B.M.K. Ong, C. Zaniolo, Metaqueries for data mining, in: Advances in Knowledge Discovery and Data Mining, AAAI/MIT Press, Cambridge, MA, 1996, pp. 375–397.
[21] Y. Fu, J. Han, Metarule-guided mining of association rules in relational databases, in: Proceedings of the 1995 Internat. Workshop on Knowledge Discovery and Deductive and Object-Oriented Databases (KDOOD'95), Singapore, 1995, pp. 39–46.