Kalpa Publications in Computing

Volume 9, 2018, Pages 49–57

LPAR-22 Workshop and Short Paper Proceedings

Towards Efficient Metaquery Generator

Tamar Bash and Rachel Ben-Eliyahu-Zohary

Department of Software Engineering
Azrieli College of Engineering,
Jerusalem, Israel
{tamarba,rbz}@jce.ac.il

Abstract

Metaquery (MQ) is a datamining tool for inferring relations between data items over a relational database. The concept of MQ leads to autonomous discovery of logical rules involving the database tables. A central module of any MQ system is the MQ generator, which automatically generates all possible MQs to be tested against a database. The MQ generator is supposed to work in an efficient manner, while not missing any meaningful MQ. In this paper we present an algorithm for MQ generation that works as a search algorithm in which the states are all possible MQs, and the search tree is pruned whenever possible. Preliminary experiments prove that, indeed, the approach we take leads to a significant reduction in computation resources.

1 Introduction

The tremendous growth in information availability has led to innumerable applications and research projects in data mining [14]. As researchers in Artificial Intelligence (AI), we are interested in *knowledge discovery*, i.e., extracting logical rules from a vast amount of data, or even better, gaining insight that can be expressed easily in a natural language. The idea of *metaqueries* is a significant step towards achieving such a goal. Metaqueries allow the expression of the relationship between entities in the dataset in a logical formula that can be later translated into a statement in natural language. Metaquerying [12, 13, 2, 4, 5, 3], also known as metapattern or metarule-guided mining [8, 15, 7], 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 they help guide the process of data analysis and pattern generation. Unlike many other discovery systems, for example, 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 [9] are capable of learning only propositional patterns. The relational aspect of the rules that we extract using metaqueries paves the way for expressing these rules in a natural language.

Metaqueries can be specified by human experts or, alternatively, they can be automatically generated from database schema. In the system that we present, metaqueries are generated automatically, and thus the system has a potential to discover truly original insights.

Hospital	County
Hadassah	Jerusalem
Sha'arei Tzedek	Jerusalem
Ichilov	Tel-Aviv

Figure 1: Table Hospital_County

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 that 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. Although both techniques may artfully simulate the other, in a sense metaquerying can be considered as an extension of association rule mining on multiple tables [7]. Metaquerying can be directly applied to native relations and/or to existing views as well, while an association rule learning systems often need preprocessing steps to denormalize data on a single table.

In this paper we present a system for metaquerying, while focusing on the algorithm for MQ generation. The role of the algorithm is to efficiently generate all MQ that have a potential to yield interesting connections between the database tables.

2 Preliminaries

First, a brief introduction to *metaqueries*. This introduction is based on results that appear in [2, 4, 5, 3]. A Metaquery is a pattern for which one tries to find a match in the database, and thus it has the potential of revealing hidden relations between the tables.

A metaquery (MQ) has the form

$$T \longleftarrow L_1, ..., L_m$$

where T and L_i are relational atoms, or simply, atoms. An atom L has the form $Q(Y_1, ..., Y_n)$. The expression T is called the *head* of the MQ, while the expression $L_1, ..., L_m$ is called the body of the MQ. The variable Q can be instantiated only by a name of a table in the database. The variables $Y_1, ..., Y_n$ can be instantiated only by a name of a column in the table that was assigned to the variable Q. The instantiation must be done in a way consistent with the variable names and with the type of the columns involved.

For example, given the following MQ:

$$R(X, Z) \leftarrow P(X, Y), Q(Y, Z) \tag{1}$$

We want to find tables R, P, Q and columns X, Y, Z in these tables that satisfy the MQ.

BirthGiver	Hospital
Sara	Ichilov
Naomi	Ichilov
Miriam	Hadassah

Figure 2: Table BG_Hospital

BirthGiver	County
Sara	Tel-Aviv
Naomi	Tel-Aviv
Miriam	Jerusalem

Figure 3: Table BG_County

For the above metaquery, the tables in Figures 1 -3 meet the rule. Hence, in this database, a possible solution to the pattern is:

$$\text{BG\_County}(X, Z) \leftarrow \text{BG\_Hospital}(X, Y), \text{Hospital\_County}(Y, Z)$$

The match reveals a new rule: if a woman gives birth in a certain hospital, and that hospital is in a certain county, then the woman lives in that county. Most often the rules discovered are not 100% correct. For example, there may be exceptions to the above-mentioned rule: women who give birth in a hospital that is not in the county in which they live. Therefore, there are support and confidence values that represent the level of certainty of the rule — in what percent of cases the rule exists. *Support value* indicates how frequently the body of the rule is satisfied, while *confidence value* indicates what fraction of the tuples satisfies the body and also satisfies the head, or in other words, how likely is the rule to hold. The purpose of certainty values is to prevent the presentation of negligible information to the user. A rule is valuable if its level of certainty (support and confidence values) is greater than a threshold we set. *A Metaquery solution* is a list of specific tables and columns in the database, for which the metaquery pattern holds with support and confidence values higher than the specified threshold. The problem of finding a solution to a metaquery is NP-hard [5].

We next explain in more detail the notion of *support* and *confidence* for metaqueries. Suppose we are given the metaquery (1) again, but instead of the healthcare database shown in Figures 1-3, we have the healthcare database shown in Figure 4 (in which relations BG-Hospital and BG-County are different, but the relation Hospital-County is the same).

In this case Rule (2) no longer holds in all cases. (Rachel gave birth in Ichilov, which is a

Hospital	County
Hadassah	Jerusalem
Sha'arei Tzedek	Jerusalem
Ichilov	Tel-Aviv

BirthGiver	Hospital
Sara	Ichilov
Naomi	Ichilov
Miriam	Hadassah
Rachel	Ichilov

BirthGiver	County
Sara	Tel-Aviv
Naomi	Tel-Aviv
Miriam	Jerusalem
Rachel	Jerusalem

Figure 4: The new healthcare database

hospital in the county of Tel-Aviv, but she lives in the county of Jerusalem.) We can say that the Rule holds in 75% of the cases. In other words, it has a *confidence* value of 0.75 (the rule holds for three out of four birth-givers).

Let us now consider another set of relations from an employee database of an Israeli hospital having 1000 employees. The hospital is located in Beer Sheva, and all employees except Ana live in that city. Ana lives nearby in Kibbutz Shoval. None of the employees, except Guy, was born in the area. Guy was born in Kibbutz Shoval, and Ana is Guy's boss.

Now consider that we pose the following metaquery:

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

An answer to this metaquery can be:

$$\text{boss}(X,Y) \longleftarrow \text{empl-born}(X,Z), \text{empl-lives}(Y,Z) \tag{3}$$

with a confidence value equal to 1.0. Thus, we 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 that 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

$$t(...) \longleftarrow r_1(...), ..., r_m(...), \tag{4}$$

let J denote the relation that is an equijoin[1] of $r_1, ..., r_m$, 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, $i = 1...m$ define S_i to be the fraction $\frac{|J_{r_i}|}{|r_i|}$. The *support value* of Rule (4) is the maximum over S_i, $i = 1...m$. 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 $i = 1, ..., m$.

The *confidence value* of (4) is the fraction of t that appears in J. That is, the *confidence value* of (4) is $\frac{|(Jt)_J|}{|J|}$.

2.1 Search Problems

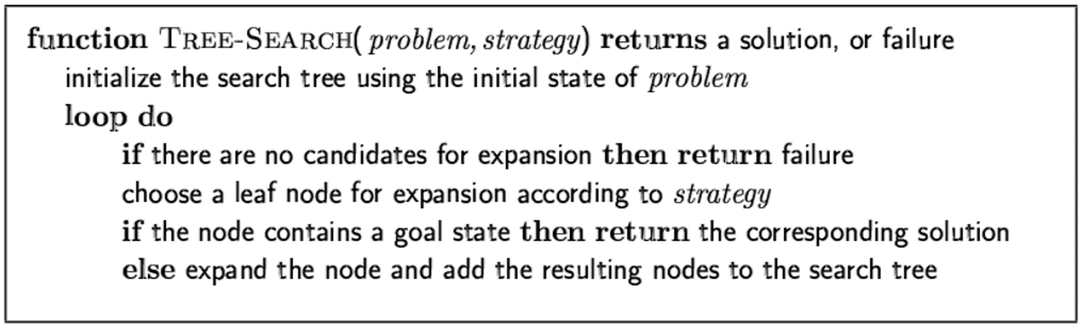
The area of *search* is one of the most studied and most known areas in AI. In this paper, we show how the problem of generating all possible MQs can be expressed as a search problem (See for example [10]). We first recall some basic definitions in the area of *search*.

A search problem over a state space problem is defined by five elements:

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

1. search states
2. initial state
3. actions or successor function that define which states are accessible from a given state.
4. goal test, which is a Boolean function that checks whether a given state is a goal state,
5. transition cost (additive), which defines what is the price of moving from a certain state to another one.

A solution is a sequence of actions leading from the initial state to a goal state. A typical search algorithm is shown in Figure 5 (Taken from [11]).



```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

Figure 5: Tree Search Algorithm

3 System Description

Figure 6 depicts the flow of control in the system that we develop. We call it the *MQ System* for obvious reasons. It is composed of two main parts: the MQ generator that produces metaqueries to be matched and answered, and the MQ solver that gets a MQ, a confidence threshold and a support threshold, and tries to find a solution, that is, a set of tables depicting the rule conveyed by the MQ with confidence and support equal or higher than the given thresholds.

Specifically, the flow of control of the MQ system proceeds as follows:

1. The MQ Generator generates a MQ according to the schema of the DB to be mined.

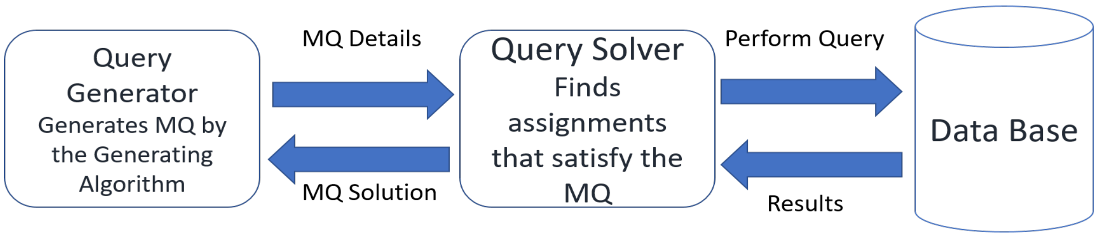

Query Generator
Generates MQ by the Generating Algorithm
MQ Details
MQ Solution
Query Solver
Finds assignments that satisfy the MQ
Perform Query
Results
Data Base

Figure 6: The MQ System

2. The MQ Generator delivers a JSON file to the MQ Solver, containing the following details –
    (a) The MQ head and body
    (b) Thresholds for the confidence and support values.
3. The MQ Solver tries to find a solution to the metaquery in an efficient way. A solution will be a collection of schemas from the database that matches the pattern described in the metaquery, and its support and confidence values above the threshold. Actual queries to the DB are made for each assignment it performs, and it calculates support and confidence values. For each possible assignment to a MQ, first the support is computed. The confidence is computed only if the support is above the support threshold. The MQ Solver returns to the MQ Generator a message containing the following details:
    (a) Acknowledgments whether an assignment was found.
    (b) If an assignment was not found, what was the reason (whether no assignment met the support requirement, or whether the assignments that were above the support threshold did not meet the confidence requirement).
    (c) Regarding all the assignments that were found, if any, each in the form of a string describes the corresponding table/column in the assignment for each variable in the MQ.

The metaquery generator (MQ generator) module generates all metaqueries that may have a solution. It is described in detail in the next section.

4 The Metaquery Generator

Assume we have a machine that can solve metaqueries. We want to provide patterns to be solved in a certain order. We want to generate and test all possible metaqueries, since we do not want to miss any significant relation. Since answering a metaquery is an expensive operation, we want this process to be more sophisticated and efficient than random creation of all possible patterns. Hence, the MQ generator should have two important properties: (1) It should generate all metaquries that may have a solution, and (2) it should prune as many queries as possible that do not lead to any significant result .

The algorithm for MQ generation that we have developed, called, for obvious reasons *MQ Generator* (MQG), is inspired by the family of Search Algorithms. We represent all possible metaqueries as a search space, which we call the MQ space, and the MQG travels along this space. Unlike a typical search algorithm, MQG is not looking for a goal state or a solution path, but rather it makes efforts to go over all the meaningful MQ in the MQ space.

More formally, the MQ space is defined as follows.

States. The states are all possible MQs. The set of all possible MQs depends on the database to be mined. Suppose this database has t different tables, and each table has c attributes, at most. We then allow t atoms at most in the body of a MQ and c variables at most in each atom.

Successor Function. Given a state s representing a MQ m, the neighboring states of s are all the legal MQs obtained from m by adding a variable to an existing atom, or by adding a new atom to the body of the query, with a new variable as a relation variable and a new variable as an attribute of this new relation variable.

	2 tab 3 col	3 tab 3 col	3 tab 4 col
Tree size no pruining	16	80	1452
Tree size pruining	15	50	802

Figure 7: Results of pruining experiments

The MQG starts with the simple MQ $P(X) \leftarrow Q(X)$ as an Initial State. It then searches the MQ space, looking for interesting relations between tables. Fortunately, there is no need to traverse the entire search space, since we can prune the search tree as follows. The pruning is based on the results of the MQ m that was tested. We consider three cases:

1. **A solution to m was found.** In this case all the neighboring states of m will be added to the search tree, unless they are in a list of MQs that were already visited.

2. **Support of all possible assignments to m was below the support threshold.** In this case there is no point in adding to the search tree MQ that are generated by adding a column variable to one of the atoms that are already in m. Only MQs that are generated from m by adding a new *atom* to m should be added to the search tree.

3. **There were assignments to m with support above the threshold, but for these assignments the confidence was insufficiently high.** In this case it is pointless to add to the search tree MQs that are generated by adding a column variable to the atom in the head of m. Only MQs that are generated from m by adding a new atom to m or by adding a column variable to an atom in the body of m should be added to the search tree.

5 Evaluation

Since the entire MQ system is still under construction, we have tested the MQ generator against a random MQ solver that randomly returns one of the three possible metaquery outcomes discussed in the previous section (a solution was found, all possible assignments have low support, some assignments are above the support threshold but their confidence value is too low). The depth of a metaquery in the search tree affects the probability of getting each possible outcome: the deeper you are in the tree, the more likely you are to get Outcome 2 or Outcome 3. In this way, the algorithm works in a similar way to what we expect to happen in practice — the deeper the metaquery is in the search tree, the more constraints there are, and the higher is the probability to have no solution.

In our experiments, we have examined three different database schema: two tables having three columns each, three tables with three columns each, and four tables with four columns each. For each schema, we have checked the following:

1. What is the size of the search tree when no pruning takes place? and,

2. What is the size of the search tree when pruning is allowed?

The results are shown in Figure 5. It is clear that the MQ generator, indeed, prunes the MQ search tree in an efficient manner. Our experiments show that the larger the database, the smaller is the fraction of metaqueries among all potential metaqueries, that were actually generated and tested .

6 Conclusion

We have presented the MQ system, which is a datamining tool capable of finding interesting relations between tables in a given database. We have focused on the MQ generation module of the MQ system. The MQ Generator algorithm (MQG) works similarly to a Tree Search algorithm, in which the states are metaqueries built during runtime. The MQG generates all the MQ that may lead to interesting insights while pruning the search tree to the extent possible. The MQ generator represents the autonomous aspect of the MQ system, because it is the responsibility of this module to test all possible conjectures. Preliminary experiments demonstrate that, indeed, the MQG is capable of pruning a significant fraction of the search space.

The true challenge of the MQ system is to really find new insights. To achieve this goal, we have to test it on real-life data. We plan to test the MQ system on a dataset that comes from the National Center for Health Statistics (NCHS) of the American Center of Disease Control and Prevention (CDC). For details, see [6].

7 Acknowledgments

- Dr. Jay Levinson for editing this paper.
- Azrieli College of Engineering for the Research and Travel Grants.

References

[1] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 207–216. ACM Press, 1993.

[2] F. Angiulli, Rachel Ben-Eliyahu-Zohary, G. B. Ianni, and L. Palopoli. Computational properties of metaquerying problems. In *PODS-2000: Proceddings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 237–244, San Diego, CA, 2000.

[3] Fabrizio Angiulli, Rachel Ben-Eliyahu-Zohary, Giovambattista Ianni, and Luigi Palopoli. Computational properties of metaquerying problems. *ACM Trans. Comput. Log.*, 4(2):149–180, 2003.

[4] Rachel Ben-Eliyahu-Zohary, Carmel Domshlak, Ehud Gudes, N Liusternik, Amnon Meisels, Tzachi Rosen, and Solomon Eyal Shimony. Fleximine–a flexible platform for kdd research and application development. *Annals of Mathematics and Artificial Intelligence*, 39(1-2):175–204, 2003.

[5] Rachel Ben-Eliyahu-Zohary, Ehud Gudes, and Giovambattista Ianni. Metaqueries: Semantics, complexity, and efficient algorithms. *Artificial Intelligence*, 149(1):61–87, 2003.

[6] Rachel Ben-Eliyahu-Zohary and Shay Tavor. Metaquerying births records, 2018.

[7] Micheline Kamber, Jiawei Han, and Jenny Y. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *Proceeding of the 3rd International Conference on Knowledge Discovery and Data Mining, Newport Beach, California*, 1997.

[8] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding Interesting Rules from Large Sets of Discovered Association Rules. In *IKM-94*, 1994.

[9] Tom M. Mitchell. *Machine Learning.* McGraw-Hill Higher Education, 1997.

[10] Judea Pearl. Heuristics: intelligent search strategies for computer problem solving. 1984.

[11] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited,, 2016.

[12] W. M. Shen. Discovering regularities from knowledge bases. *Intelligent Systems*, 7(7):623–636, 1992.

[13] B. Mitbander Wei-Min Shen, K. Ong and C. Zaniolo. Metaqueries for Data Mining. In *Advances in Knowledge Discovery and Data Mining*, pages 375–397. AAAI/MIT press, 1996.

[14] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann, 2016.

[15] Y.Fu and J. Han. Meta-rule-guided mining of association rules in relational databases. In *Proc. of the 1995 Int'l Workshop. on Knowledge Discovery and Deductive and Object-Oriented Databases (KDOOD'95), Singapore, December 1995, pp. 39-46*, December 1995.