A KDD framework to support database audit [*]

Jean-François Boulicaut

Institut National des Sciences Appliquées de Lyon, LISI Bâtiment 501,
F-69621 Villeurbanne Cedex, France
E-mail: jean-francois.boulicaut@insa-lyon.fr

Understanding data semantics from real-life databases is considered following an *audit* perspective: it must help experts to analyse what properties actually hold in the data and support the comparison with desired properties. This is a typical problem of knowledge discovery in databases (KDD) and it is specified within the framework of Mannila and Toivonen where data mining consists in querying theories e.g., the theories of approximate inclusion dependencies. This formalization enables us to identify an important subtask to support database audit as well as a generic algorithm. Next, we consider the DREAM relational database reverse engineering method and DREAM heuristics are revisited within this new setting.

Keywords: data mining, integrity constraint, reverse engineering

1. Introduction

We are interested in understanding data semantics from real-life databases. This process is considered following an *audit* perspective in the following sense: it must help experts to analyse what properties actually hold in the data and support the comparison with desired properties. This research paper takes examples from relational database audit, assuming that inclusion and functional dependencies that (almost) hold in the data capture the so-called data semantics. This will be called hereafter *the basic problem*. However, our framework can be applied to other kinds of databases and/or properties.

Auditing databases is an important topic. Integrity constraints that have been more or less explicited at the design time of a database may not always hold in a given instance. Indeed, only recent Data Base Management Systems enable to enforce important integrity constraints such as foreign keys. In most of the cases, it is assumed that application programs enforce desired integrity constraints and, obviously, it is not always done in real-life databases. Understanding data semantics in databases is of a crucial interest to support their maintenance and evolution. The fact that some property

[*] This work has been done while the author was on sabbatical year in the Department of Computer Science at the University of Helsinki (Finland). It is partly supported by AFFRST, Association Franco-Finlandaise pour la Recherche Scientifique et Technique.

holds or not in an instance can be used by experts to fix some integrity violation in the data. It can also lead to an explicit definition of an integrity constraint for further use of built-in checking mechanisms. Improving our knowledge of encoded data semantics is also useful for semantic query optimization (see, e.g., [2]). Last but not least, audit is an important preliminary step for a database reverse engineering process [5] or the design of federated databases. Indeed, solving the basic problem provides the raw knowledge that is needed to start a restructuring phase on a denormalized relational schema [13].

Auditing as querying multiple theories. Auditing databases is a typical problem of Knowledge Discovery in Databases (KDD). Discovering knowledge from databases can be seen as a process containing several steps: understanding the domain, preparing the data set, discovering patterns (e.g., dependencies), postprocessing of discovered patterns (e.g., selecting dependencies that should become integrity constraints), and putting the results into use [8]. This is a semi-automatic and iterative process that is often described using ad-hoc formalisms and/or notations. However, a general KDD framework has been proposed by Mannila and Toivonen [11]. Data mining consists in querying the so-called *theory* of the database w.r.t. a class of patterns and a selection predicate that defines their interestingness. Audit can then be supported by queries over relevant theories. This approach emphasizes a human-centered process: expert users can precisely specify the theories they are interested in and formulate queries to learn about the properties that really hold in the data.

Contribution. First, we specify auditing tasks within this general KDD framework. The basic problem is formalized as mining theories of approximate inclusion and functional dependencies (see section 2). This enables us to identify an important subtask to support database audit, i.e., querying an intensionally defined collection of dependencies. A generic algorithm, the "guess-and-correct" scheme introduced in [11], is a good starting point for the evaluation of such queries (section 3). Finally, in section 4, we revisit the heuristics that constitute the core of the DREAM reverse engineering method [16,17]. In this project, equi-joins that are performed in the application programs are used to support the discovery of "relevant" dependencies.

2. A formal framework for database audit

Notations. The reader is supposed to be familiar with relational database concepts. Suppose **r** is a relational database instance over the schema **R**. A *relation* $R_i(X_i)$ belongs to **R** and is defined by a relation name R_i and a set of *attributes* X_i. Each relation $R_i(X_i)$ is associated with a *table* r_i which is a set of *tuple*s. The database extension **r** represents the set of tables r_i. $r_i[Y]$ is the *projection* of the table r_i on $Y \subseteq X_i$ and $t[Y]$ is the projection of the tuple t following Y. Let Y and Z be two subsets of X_i, a *functional dependency* denoted by $R_i : Y \rightarrow Z$ on $R_i(X_i)$ is true in r_i iff $\forall t, t' \in r_i \quad t[Y] = t'[Y] \Rightarrow t[Z] = t'[Z]$. It can be written $\mathbf{r} \models Y \rightarrow Z$.

Let $R_i(X_i)$ and $R_j(X_j)$ be two relations associated with tables r_i and r_j, respectively. Let Y (resp. Z) be a subset of attributes of X_i (resp. X_j). An *inclusion dependency* denoted by $R_i[Y] \subseteq R_j[Z]$ is true in r_i and r_j iff $r_i[Y] \subseteq r_j[Z]$. It can be written $\mathbf{r} \models R_i[Y] \subseteq R_j[Z]$.

2.1. Computing theories

First, we introduce the KDD framework of Mannila and Toivonen [11]. Given a database instance **r**, assume the definition of a language $\mathcal{L}$ for expressing properties of the data and a *selection predicate* q. The predicate q is used for evaluating whether a sentence $\varphi \in \mathcal{L}$ defines a potentially interesting property of **r**. Therefore, a mining task is to compute the *theory* of **r** with respect to $\mathcal{L}$ and q, i.e., the set $\mathcal{T}h(\mathcal{L}, \mathbf{r}, q) = \{\varphi \in \mathcal{L} \mid q(\mathbf{r}, \varphi) \text{ is true}\}$. It is possible to consider generic algorithms to compute such theories following the popular "learning as search" paradigm. A reasonable collection of data mining tasks (association rules, sequential patterns, data dependencies, etc.) have already been carried out using this approach (see [12] for a survey).

Example 2.1. Consider the discovery of dependencies that hold in a database. Assume $\mathcal{L}_1$ is the language of inclusion dependencies and consider q_1 as the satisfaction predicate: let r and s be the instances of R and S, and $\delta = R[X] \subseteq S[Y] \in \mathcal{L}_1$, $q_1(\mathbf{r}, \delta)$ is true iff $\mathbf{r} \models \delta$. Let $\mathcal{L}_2$ be the language of functional dependencies. Here again, the predicate q_2 is the satisfaction predicate.

For instance, assume $R = \{A, B, C, D\}$ and $S = \{E, F, G\}$ and the two following instances:

A	B	C	D
1	2	4	5
2	2	2	3
3	1	1	2
4	2	2	3

E	F	G
1	2	3
2	3	4
3	2	2

$R[\langle B \rangle] \subseteq S[\langle E \rangle] \in \mathcal{L}_1$ and satisfies q_1. $R[\langle D \rangle] \subseteq S[\langle E \rangle] \in \mathcal{L}_1$ and does not satisfy q_1. $AB \rightarrow C \in \mathcal{L}_2$ and satisfies q_2. $BC \rightarrow A \in \mathcal{L}_2$ and does not satisfy q_1.

Looking for a generic data mining technique, a key issue is to organize the search through the space of $\mathcal{L}$ sentences and get some safe pruning strategies. Here, to be safe means that we do not want to miss any interesting sentence w.r.t. the selection predicate. A simple idea is to define a specialization relation on $\mathcal{L}$ and then use a levelwise algorithm to compute $\mathcal{T}h(\mathcal{L}, \mathbf{r}, q)$ [11]. A *specialization relation* $\preceq$ is a partial order: φ is *more general* than θ, if $\varphi \preceq \theta$ (θ is *more specific* than φ). It is a *monotone specialization relation* w.r.t. q if for all **r** and φ, if $q(\mathbf{r}, \varphi)$ and $\gamma \preceq \varphi$ then $q(\mathbf{r}, \gamma)$. In other words, if a sentence φ satisfies q, then also all more general sentences γ satisfy q. A simple but powerful "generate-and-test" algorithm can now be derived: start from the most general sentences and then try to generate and to evaluate more and more specific sentences, but do *not* evaluate those sentences that are not interesting given the available information.

Example 2.2. A monotone specialization relation w.r.t. inclusion dependencies is defined as follows: for $\varphi = R[X] \subseteq S[Y]$ and $\theta = R'[X'] \subseteq S'[Y']$, we have $\varphi \preceq_1 \theta$ only if $R = R'$, $S = S'$, and furthermore $X' = \langle A_1, \ldots, A_k \rangle$, $Y' = \langle B_1, \ldots, B_k \rangle$, and for some disjoint $i_1, \ldots, i_h \in \{1, \ldots, k\}$ with $h < k$ we have $X = \langle A_{i_1}, \ldots, A_{i_h} \rangle$, $Y = \langle B_{i_1}, \ldots, B_{i_h} \rangle$. For instance, given $R = \{A, B, C, D\}$ and $S = \{E, F, G\}$, $R[\langle A \rangle] \subseteq S[\langle E \rangle] \preceq_1 R[\langle A, B \rangle] \subseteq S[\langle E, F \rangle]$. Notice that this is not true within the instances of example 2.1. The most general sentences are all the potential unary inclusion dependencies and at each iteration, one consider "longer attribute sequences". Now, assuming the restriction to functional dependencies with a fixed right-hand side denoted as B, a monotone specialization relation w.r.t. functional dependencies is the reverse of set inclusion: for X, $Y \subseteq R$ and $B \in R$ we have $X \rightarrow B \preceq_2 Y \rightarrow B$ iff $Y \subseteq X$. For instance, $AB \rightarrow D \preceq_2 A \rightarrow D$. The most general sentences have the whole set R as the left-hand side. At each iteration, one considers shorter left-hand sides. The selection predicate q_1 (resp. q_2) is monotone w.r.t. $\preceq_1$ (resp. $\preceq_2$). It means that if $R[\langle A, B \rangle] \subseteq S[\langle E, F \rangle]$ holds then $R[\langle A \rangle] \subseteq S[\langle E \rangle]$ holds. A safe pruning criteria is now obvious: if $R[\langle A \rangle] \subseteq S[\langle E \rangle]$ does not hold then $R[\langle A, B \rangle] \subseteq S[\langle E, F \rangle]$ does not hold either and should be discarded at the candidate generation step. The same idea applies for functional dependencies: if $AB \rightarrow D$ does not hold then $A \rightarrow D$ does not hold either and might be pruned.

By alternating candidate generation and candidate evaluation, a levelwise algorithm moves gradually to the more specific interesting sentences. This has been already implemented for inclusion and functional dependency computation (see [11] for a complexity analysis and pointers to related work). It provides the best known algorithm for the discovery of all the inclusion dependencies. For functional dependencies, better algorithms are available (e.g., [9]). However, it is clear that functional dependency discovery is a very hard problem due to its inherent exponential complexity and the fact that functional dependencies with long left-hand sides are less likely to hold than functional dependencies with shorter ones.

A problem with such a scheme is that the computation of the most interesting sentences in a theory can be quite slow if there are interesting statements that are far from the most general sentences (the typical case for functional dependencies). Furthermore, a framework designed to support (basic) audit task might consider some important specificities of this kind of application. First, one should not consider only exact dependencies: we want to study dependencies even in the case where some tuples violate these constraints. Next, the expert user is quite often interested in tightly specified subsets of the dependencies that hold. For instance, he/she just wants dependencies that involve a given collection of attributes. Finally, quite often, dependencies do not have to be computed from scratch, i.e., either the expert user has already a good knowledge of constraints that should hold and/or the computation of constraints has been already done on a previous state of the database.

This motivates the definition of the theories of approximate dependencies, a querying approach over intensionally defined theories, and the use of a variation of the levelwise algorithm, the so-called "guess-and-correct" scheme.

2.2. *Solving the basic problem*

Computing approximate dependencies. Inconsistencies in the database can be allowed by defining $q'(\mathbf{r}, \delta)$ to be true if some error measure of the dependency δ is lower or equal to a user-defined threshold. Let us define an error measure g for the inclusion dependency $\delta = R[X] \subseteq S[Y]$ in $\mathbf{r}$:

$$g(\delta, \mathbf{r}) = 1 - \max\{|r'| \mid r' \subseteq r \wedge (r' \cup (\mathbf{r} \setminus r)) \models \delta\}/|r|$$

where r is the instance of R. Using g enables one to consider dependencies that almost hold since it gives the proportion of tuples that must be removed from r to get a true dependency. Among the several ways of defining approximate functional dependencies in an instance r of R, one can also consider the minimum number of rows that need to be removed from r for the dependency $\gamma = R : X \to B$ to hold (the so-called g_3 error measure in [9]):

$$g_3(\gamma, \mathbf{r}) = 1 - \max\{|r'| \mid r' \subseteq r \wedge r' \models \gamma\}/|r|.$$

Example 2.3. Assuming the instances of example 2.1 for $R = \{A, B, C, D\}$ and $S = \{E, F, G\}$, a few approximate inclusion and functional dependencies are given.

Inclusion dependencies	Error	Functional dependencies	Error
$R[\langle B\rangle] \subseteq S[\langle E\rangle]$	0	$B \to A$	0.5
$R[\langle D\rangle] \subseteq S[\langle E\rangle]$	0.25	$C \to A$	0.25
$S[\langle E\rangle] \subseteq R[\langle B\rangle]$	0.33	$BC \to A$	0.25
$R[\langle C, D\rangle] \subseteq S[\langle E, F\rangle]$	0.25	$BCD \to A$	0.25

The selection predicate q_1 (resp. q_2) can be modified to denote that all the approximate inclusion (resp. functional) dependencies whose error is lower or equal to a user-supplied threshold are desired. These selection predicates remain monotone w.r.t. their respective specialization relations.

Now, one naive approach to solve the basic problem might be to compute theories of approximate dependencies for some error thresholds, store them in a "SQL3" table and then query such tables using available query languages. This is a typical approach in many KDD applications where interestingness of patterns is considered in a postprocessing phase while pattern discovery is mainly guided by simple criteria like statistical significance or error thresholds. This gives rise to several problems.

The size of such theories can be huge while the expert user is interested in only a few dependencies. Not only is it untractable to compute the whole theories but also it gives rise to tedious postprocessing phases (e.g., a posteriori elimination of redundancies). It motivates a flexible querying framework that supports the analysis of tightly specified theories.

Querying tightly specified theories. It happens that, a priori, only small subsets of the languages of dependencies $\mathcal{L}_1$ or $\mathcal{L}_2$ are interesting. Restrictions of practical interest concern nontrivial inclusion or functional dependencies, unary inclusion dependencies but also various selection criteria over attributes. Attribute data types and application domain semantics guide the definition of such restrictions.

Example 2.4. Continuing example 2.1, an inclusion dependency like $R[\langle B\rangle] \subseteq S[\langle E\rangle]$ can be considered as irrelevant if the corresponding domains for E and B are respectively, a collection of transaction identifiers and $\{1,2\}$ to denote male or female. Also, in an application domain like relational schema restructuring, it is clear that not all the functional dependencies are interesting (see section 4).

In fact, only expert users can define such restrictions. It is possible to define them either as context-sensitive restrictions to the definition of $\mathcal{L}_1$ and $\mathcal{L}_2$ or by means of new selection predicates. These different views influence the computation process and its efficiency: it is more or less a “generate-and-test” scheme when the generation of candidate dependencies can make an active use of given restrictions. Notice that refining the language (re)definition is the basic technique for dependency discovery with inductive logic programming systems, the so-called declarative linguistic bias definition (see, e.g., [7]).

A typical audit process requires the computation of many related theories. Not only, several theories for the same dependency class are needed, depending on the dynamically evolving user’s interest, but also different theories for different kinds of dependencies might be useful. An obvious example is the audit of *referential integrity constraints* for which one must consider inclusion dependencies whose right-hand side sets of attributes are a key, i.e., a special case of functional dependency.

To cope with such a querying approach, the conceptual framework of inductive databases has recently emerged. It suggests an elegant approach to support audit or more generally data mining over multiple theories.

2.3. Towards inductive databases

An *inductive database*, is a database that contains inductive generalizations about the data, in addition to the usual data [3]. The idea is that the user can then query the data, the properties in the data as well as the ”behavior” of the data w.r.t. some properties.

Theories are here defined intensionally. Indeed, it is not realistic to consider that querying properties can be carried out by means of queries over some materializations of every properties (e.g., data dependencies). The idea is that properties are computed only at query evaluation time, when the user is asking for some specific ones. However, during the formulation of the query, he can think that every property is just there.

Formally, the *schema of an inductive database* is a pair $\mathcal{R} = (\mathbf{R}, (\mathcal{Q}_{\mathbf{R}}, e, \mathcal{V}))$, where $\mathbf{R}$ is a database schema, $\mathcal{Q}_{\mathbf{R}}$ is a collection of patterns, $\mathcal{V}$ is a set of *result values*, and e is the *evaluation function* that defines how patterns occur in the data. The function e maps each pair $(\mathbf{r}, \theta_i)$ to an element of $\mathcal{V}$, where $\mathbf{r}$ is a database over $\mathbf{R}$ and θ_i is a pattern from $\mathcal{Q}_{\mathbf{R}}$. An *instance* $(\mathbf{r}, s)$ of an inductive database over the schema $\mathcal{R}$ consists of a database $\mathbf{r}$ over the schema $\mathbf{R}$ and a subset $s \subseteq \mathcal{Q}_{\mathbf{R}}$.

For our basic problem, we need two inductive databases that associate to a database all the inclusion dependencies and functional dependencies that can be built from its schema. We can choose that evaluation functions respectively return the g and g_3 error measures as previously defined.

At each stage of manipulating an inductive database $(\mathbf{r}, s)$, the user can think that the value of $e(\mathbf{r}, \theta)$ is available for each pattern θ which is present in the set s, i.e., that every dependency is actually in s. He/she sends queries over the intensionally defined collections of all dependencies to select only dependencies fulfilling some constraints.

Example 2.5. Continuing example 2.1, a user might be interested in "selecting" only inclusion dependencies between instances r and s that do not involve attribute $R.A$ in their left-hand side and have a g error value lower than 0.3. One expects that a sentence like $R[\langle C, D\rangle] \subseteq S[\langle E, F\rangle]$ belongs to the answer.

The definition of a concrete query language for inductive databases is out of the scope of this paper. However, let us take ideas from the SQL-like operator `MINE RULE` [15] and propose two queries.

Example 2.6. The first part of a query specifies the kind of dependency one wants to mine while the second one, that begins with the keyword `FROM`, defines the data in which mining is performed. The intuition is that all the power of SQL can be used for that selection of the data part. The query introduced in example 2.5 can be written as follows:

```
MINE INCLUSION DEPENDENCIES as IND
SELECT 1..n as LHS-IND
       1..n as RHS-IND
       ERROR
WHERE (ERROR < 0.3) AND (R.A NOT IN LHS-IND)
FROM R,S
```

The schema of the output table `IND` has three attributes `LHS-IND`, `RHS-IND` and `ERROR` that correspond respectively to the left-hand side, the right-hand side and the error measure of inclusion dependencies. Given the instances in example 2.1, we expect that the tuple $(\langle C, D\rangle, \langle E, F\rangle, 0.25)$ that denotes the approximate inclusion dependency $R[\langle C, D\rangle] \subseteq S[\langle E, F\rangle]$ belongs to `IND`.

One can now search for functional dependencies in s whose left-hand sides are a right-hand side of a previously discovered inclusion dependency. Such a query might look like:

```
MINE FUNCTIONAL DEPENDENCIES as FD
SELECT 1..n as LHS-FD
       1..1 as RHS-FD
       ERROR
WHERE (ERROR=0) AND (LHS-FD IN (SELECT IND.RHS-IND FROM
       IND))
FROM S
```

The schema of the output table `FD` has also three attributes `LHS-FD`, `RHS-FD` and `ERROR` with the obvious meaning. Given the instances in example 2.1, we expect that the tuple $(\langle E, F\rangle, \langle G\rangle, 0)$ that denotes $EF \rightarrow G$ belongs to `FD`.
Evaluating this kind of query provides information about potential foreign keys between $R = \{A, B, C, D\}$ and $S = \{E, F, G\}$.

Actual object-relational query languages can be used as a basis for inductive database query languages. However, non classical optimization schemes are needed since selections of properties lead to complex data mining phases. Indeed, implementing such query languages is difficult because selections of properties are not performed over previously materialized collections. Optimizing this kind of query remains an open problem for properties like functional dependencies. However, many inspiring ideas emerge from current research on association rule mining [3].

3. The "guess-and-correct" generic algorithm

Ref. [11] provides a generic algorithm that starts the process of finding $\mathcal{T}h(\mathcal{L}, \mathbf{r}, q)$ from an initial guess $\mathcal{S} \subseteq \mathcal{L}$. It appears as an interesting basis for query evaluation. Consider a set $\mathcal{S} \subseteq \mathcal{L}$ closed downwards under $\preceq$, i.e., if $\varphi \in \mathcal{S}$ and $\gamma \preceq \varphi$, then $\gamma \in \mathcal{S}$ (by definition, this is true for $\mathcal{T}h(\mathcal{L}, \mathbf{r}, q)$). The *border* $\mathcal{B}d(\mathcal{S})$ of $\mathcal{S}$ consists of those sentences φ such that all generalizations of φ are in $\mathcal{S}$, the so-called *positive border* $\mathcal{B}d^+(\mathcal{S})$, and none of the specializations of φ is in $\mathcal{S}$, the so-called *negative border* $\mathcal{B}d^-(\mathcal{S})$). $\mathcal{B}d^+(\mathcal{S})$ consists of the most specific sentences in $\mathcal{S}$, and $\mathcal{B}d^-(\mathcal{S})$ consists of the most general sentences that are not in $\mathcal{S}$. Roughly speaking, the positive border is the collection of the sentences that are "just in" the theory while the negative border is the set of sentences that are "just off".

Example 3.1. Assume that the collection of maximal nontrivial inclusion dependencies between $R = \{A, B, C, D\}$ and $S = \{E, F, G\}$, i.e., the positive border of the theory is $\{R[\langle A, B, D\rangle] \subseteq S[\langle G, F, E\rangle],\ R[\langle C\rangle] \subseteq S[\langle E\rangle],\ S[\langle E\rangle] \subseteq R[\langle C\rangle],\ S[\langle E\rangle] \subseteq R[\langle D\rangle]\}$. Its negative border contains many non-dependencies like $R[\langle A\rangle] \subseteq S[\langle E\rangle]$, or $R[\langle B\rangle] \subseteq S[\langle E\rangle]$.

Computing borders is not a simple task in general but it might be tractable for sets of data dependencies in real-life business databases.

Algorithm 3.1. The *"guess-and-correct" algorithm* [11]. Given, a database $\mathbf{r}$, a language $\mathcal{L}$ with specialization relation $\preceq$, a selection predicate q, and an initial guess $\mathcal{S}$ closed under generalizations, this algorithm outputs $\mathcal{T}h(\mathcal{L}, \mathbf{r}, q)$.

```
1.  $\mathcal{E} := \emptyset$; // correct $\mathcal{S}$ downward
2.  $\mathcal{C} := \mathcal{B}d^+(\mathcal{S})$;
3.  while $\mathcal{C} \neq \emptyset$ do
4.      $\mathcal{E} := \mathcal{E} \cup \mathcal{C}$;
5.      $\mathcal{S} := \mathcal{S} \setminus \{\varphi \in \mathcal{C} \mid q(\mathbf{r}, \varphi)$ is false$\}$;
6.      $\mathcal{C} := \mathcal{B}d^+(\mathcal{S}) \setminus \mathcal{E}$;
7.  od;
8.  $\mathcal{C} := \mathcal{B}d^-(\mathcal{S}) \setminus \mathcal{E}$;  // $\mathcal{S} \subseteq \mathcal{T}h(\mathcal{L}, \mathbf{r}, q)$; expand $\mathcal{S}$ upwards
9.  while $\mathcal{C} \neq \emptyset$ do
10.     $\mathcal{E} := \mathcal{E} \cup \mathcal{C}$;
        // evaluation: find which sentences of $\mathcal{C}_i$ satisfy $q$:
11.     $\mathcal{S} := \mathcal{S} \cup \{\varphi \in \mathcal{C} \mid q(\mathbf{r}, \varphi)$ is true$\}$;
        // generation: compute $\mathcal{C}_{i+1} \subset \mathcal{L}$ using $\mathcal{S}$:
12.     $\mathcal{C} := \mathcal{B}d^-(\mathcal{S})$;
13. od;
14. output $\mathcal{S}$;
```

The algorithm first evaluates the sentences in the positive border $\mathcal{B}d^+(\mathcal{S})$ and removes from $\mathcal{S}$ those that are not interesting. These steps are repeated until the positive border only contains sentences satisfying q, and thus $\mathcal{S} \subseteq \mathcal{T}h(\mathcal{L}, \mathbf{r}, q)$. Then the algorithm expands $\mathcal{S}$ upwards, it evaluates such sentences in the negative border $\mathcal{B}d^-(\mathcal{S})$ that have not been evaluated yet, and adds those that satisfy q to $\mathcal{S}$. Again, these steps are repeated until there are no sentences to evaluate. Finally, the output is $\mathcal{S} = \mathcal{T}h(\mathcal{L}, \mathbf{r}, q)$. Notice that, from the complexity point of view, the selection predicate has to be evaluated on every sentence that belongs to the border of a theory. If the initial guess $\mathcal{S} = \emptyset$ then $\mathcal{B}d^+(\mathcal{S}) = \emptyset$ and the first part of the algorithm is just skipped while the second part starts with the candidate set $\mathcal{C}$ containing the most general sentences of $\mathcal{L}$. We get the simple levelwise algorithm that has been sketched in section 2.1.

The discovery of (approximate) functional and inclusion dependencies in a database can be solved by algorithm 3.1 given the specialization relations we introduced.

Results about the complexity of such a scheme can be found in [11] and are not discussed here. However, it is clear that the better the guess is, the better is the efficiency of the algorithm. How to obtain good original guesses $\mathcal{S}$? One fairly widely applicable method is sampling: take a small sample **s** from **r**, compute $\mathcal{T}h(\mathcal{L}, \mathbf{s}, q)$ and use it as $\mathcal{S}$. Another obvious situation where a guess is available is when a new audit is performed on the same database: most of the dependencies should have been preserved. Definitions at the schema level and application programs can also be used to produce a guess.

4. Revisiting DREAM heuristics

This section revisits heuristics about dependency discovery for relational database reverse engineering. Given an operational database, the aim of a Database Reverse Engineering (DBRE) process is to improve the understanding of the data semantics and to support the (re)definition of a validated conceptual model. A DBRE process is naturally split into two major steps [17]:

- Eliciting the data semantics from the existing system
  Various sources of information can be relevant for tackling this task, e.g., the physical schema or the dictionary, the data, the application programs, but especially expert users. Among other things, application programs might encode integrity constraints that have not been encoded at the schema level.
- Expressing the extracted semantics with a high level data model
  This task consists in a schema translation activity and gives rise to several difficulties since the concepts of the original model (e.g., a relational schema) do not overlap those of the target model (e.g., an Entity-Relationship model).

Many works have been done where a conceptual schema is more or less automatically derived from a hierarchical database, a network database or a relational database [1,4, 16]. For relational databases, it is not realistic to assume that functional dependencies or foreign keys are available at the beginning of a DBRE process. Furthermore, the less we make assumptions on the knowledge a priori (normalization, attribute naming discipline, etc.), the more we can cope with real-life databases. Several works [18, 16,19] have proposed independently to fetch the needed information from the data manipulation statements embedded in application programs.

In the DREAM project [16], we began to study the use of equi-joins to support 3NF schema reverse engineering. This work has been extended to cope with denormalized schema in [17]. In the spirit of [14], the DREAM approach considers the relational schemas that can be translated into conceptual schemas, by looking into the method which has been used to design them. The key problems can be resumed as follows: identifying the relevant objects of the application domain, recovering the structure of each of these objects and eliciting the links (or relationships) between these objects. This process is inherently iterative and interactive: only a part of dependency discovery can be done automatically.

To cope with denormalized schemas, [17] propose a restructuring phase that leads to 3NF schemas where, according to the experts in charge of the validation, each relation maps exactly one object of the application domain. For that purpose, one has to find the functional dependencies which are meaningful for the application domain while they are not conceptualized as relations. Assuming that primary keys are known, the difficulty is to find out the non-key attributes that correspond to identifiers of objects of the application domain. These attributes constitute the left hand side of relevant functional dependencies and are involved in some (approximate) inclusion dependencies (foreign keys). As a matter of fact, one must support the discovery of hidden objects [10] that can even be encoded in 3NF schemas.

The main contribution in the DREAM proposal has been to explore how the attributes on which equi-joins are performed help the discovery of interesting inclusion and functional dependencies for these restructuring purposes. The main result has been the following heuristics.

- Equi-joins between sets of attributes that are embedded in application programs can be used to discover "relevant" inclusion dependencies.
- Non-key attributes of discovered inclusion dependencies are good candidates for the left-hand side of "relevant" functional dependencies.

These heuristics obviously reduce the number of dependencies to be considered and "relevancy" refers to the interestingness of discovered dependencies for the restructuring process. A complete scenario is considered in [17] though the following example carries out the intuition.

Example 4.1. Given `emp=`$\{$`code,name,tel,add`$\}$, `dept=`$\{$`dep,director, add`$\}$ and the dependencies (1) `dept[director]` $\subseteq$ `emp[code]` (2) `emp[add]` $\subseteq$ `dept[add]` (3) `code` $\rightarrow$ `name, tel, add` and (4) `tel` $\rightarrow$ `add`. Dependencies (1) and (3) seem relevant for a restructuring phase while (2) and (4) are just integrity constraints. The DREAM heuristics rely on the assumption that `dept.director` $\bowtie$ `emp.code` is probably performed in application programs (pointing out the potentially interesting dependency (1) and that `code` is a candidate for a left-hand side of a potentially interesting functional dependency) while `dept.add` $\bowtie$ `emp.add` probably does not occur.

It is now clear that analyzing the set of equi-joins in application programs, enables to focus on interesting inclusion dependencies. Furthermore, it helps to fix integrity problems when we find that, e.g., $R.C \bowtie S.E$ is performed while neither $R[\langle C\rangle] \subseteq S[\langle E\rangle]$ or $S[\langle E\rangle] \subseteq R[\langle C\rangle]$ hold.

The collection of equi-joins can be considered as a theory $\mathcal{Th}(\mathcal{L}, \mathbf{r}, q)$ where $\mathcal{L}$ is the language of all the equi-joins between sets of attributes from $\mathbf{r}$ and q is the predicate that says if a given equi-join is performed on $\mathbf{r}$. Computing such theories requires nontrivial compilation techniques. Indeed, even if we consider only SQL queries, equi-joins can be performed in many ways, with nested or unnested queries, with a where clause or with an intersect operator, etc. Such a collection is a valuable source of information to support maintenance of application code as well as data semantics elicitation.

DREAM heuristics can be encoded in queries over the inductive databases of inclusion and functional dependencies, using selection criteria derived from equi-join occurrences. The DREAM method does not claim any completeness about discovered dependencies. However, it appears that by combining the different sources of information, we can compute guesses and speed up the discovery of the complete collections.

5. Conclusions

We presented a framework for the audit of databases based on a KDD perspective. It emphasizes that high-level querying tools are needed to support expert analysis of operational databases. It provides a nice application domain for an ongoing research on generic data mining tools (inductive database management systems) though it brings a solution to concrete problems of practical interest (e.g., mining approximate dependencies). We must now study typical audit tasks. For instance, supporting the elicitation of referencial integrity constraints can be useful when migrating from an old DBMS to a recent one. It needs not only to mine inclusion and functional dependencies but also to support the efficient search for erroneous data. Finally, it seems interesting to study the relationship between approximate inclusion dependency discovery and other measures of similarity between sets of attributes, e.g., [6].

References

[1] C. Batini, S. Ceri and S. Navathe, *Conceptual Database Design: An Entity-Relationship Approach* (Benjamin Cummings, 1997).

[2] S. Bell, Discovering rules in relational databases for semantic query optimisation, in: *Proc. PADD'97, Practical Application Company* (1997) pp. 79–90.

[3] J-F. Boulicaut, M. Klemettinen and H. Mannila, Querying inductive databases: A case study on the MINE RULE operator, in: *Proc. PKDD'98*, LNAI 1510, Springer-Verlag (1998) pp. 194–202.

[4] R.H.L. Chiang, T.M. Barron and V.C. Storey, Reverse engineering of relational databases: extraction of an EER model from a relational database, Data & Knowledge Engineering 10(12) (1994) 107–142.

[5] R.H.L. Chiang, T.M. Barron and V.C. Storey, A framework for the design and evaluation of reverse engineering methods for relational databases, Data & Knowledge Engineering 21 (1996) 53–77.

[6] C. Chua, R.H.L. Chiang and E-P. Lim, Instance-based attribute identification in database integration, in: *Proc. WITS'98*, eds. S.T. March and J. Bubenko Jr., pp. 147–156.

[7] L. de Raedt and L. Dehaspe, Clausal discovery, Machine Learning 26 (2) (1997) 99–146.

[8] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth and R. Uthurusamy (eds.), *Advances in Knowledge Discovery and Data Mining* (AAAI Press, 1996).

[9] Y. Huhtala, J. Kärkkäinen, P. Porkka and H. Toivonen, Efficient discovery of functional and approximate dependencies using partitions, in: *Proc. ICDE'98* (IEEE Computer Society Press, 1998) pp. 392–401.

[10] P. Johannesson, A method for translating relational schemas into conceptual structures, in: *Proc. ICDE'94* (IEEE Computer Society Press, 1994) pp. 190–201.

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

[12] H. Mannila, Methods and problems in data mining, in: *Proc. ICDT'97*, LNCS 1186, Springer-Verlag (1997) pp. 41–55.

[13] H. Mannila and K.-J. Räihä, *The Design of Relational Databases* (Addison-Wesley, 1992).

[14] V.M. Markowitz and J.A. Makowsky, Identifying extended entity-relationship object structures in relational schemas, IEEE Trans. on Software Engineering 16(8) (1990) 777–790.

[15] R. Meo, G. Psaila and S. Ceri, A new SQL-like operator for mining association rules, in: *Proc. VLDB'96* (1996).

[16] J.-M. Petit, J. Kouloumdjian, J.-F. Boulicaut and F. Toumani, Usign queries to improve database reverse engineering, in: *Proc. ER'94*, LNCS 881, Springer-Verlag (1994) pp. 369–386.

[17] J.-M. Petit, F. Toumani, J.-F. Boulicaut and J. Kouloumdjian, Towards the reverse engineering of denormalized relational databases, in: *Proc. ICDE'96* (IEEE Computer Society Press, 1996) pp. 218–227.

[18] W.J. Premerlani and M. Blaha, An approach for reverse engineering of relational databases, Communications of the ACM 37(5) 1994 42–49.

[19] O. Signore, M. Loffredo, M. Gregori and M. Cima, Reconstruction of ER schema from database applications: A cognitive approach, in: *Proc. ER'94*, LNCS 881, Springer-Verlag (1994) pp. 387–402.