UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK

LEHRSTUHL VIII
KÜNSTLICHE INTELLIGENZ

Discovery of Data Dependencies in Relational Databases

LS–8 Report 14

Siegfried Bell **Peter Brockhausen**

Dortmund, April 3, 1995

Universität Dortmund
Fachbereich Informatik

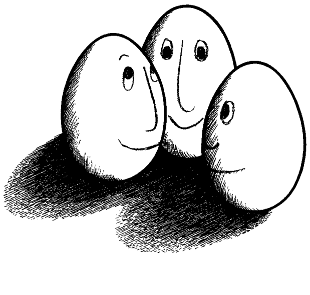
University of Dortmund
Computer Science Department

Forschungsberichte des Lehrstuhls VIII (KI)
Fachbereich Informatik
der Universität Dortmund

ISSN 0943-4135

Anforderungen an:

Universität Dortmund
Fachbereich Informatik
Lehrstuhl VIII
D-44221 Dortmund



Research Reports of the unit no. VIII (AI)
Computer Science Department
of the University of Dortmund

ISSN 0943-4135

Requests to:

University of Dortmund
Fachbereich Informatik
Lehrstuhl VIII
D-44221 Dortmund

e-mail: reports@ls8.informatik.uni-dortmund.de
ftp: ftp-ai.informatik.uni-dortmund.de:pub/Reports
www: http://www-ai.informatik.uni-dortmund.de/ls8-reports.html

Discovery of Data Dependencies in Relational Databases

LS–8 Report 14

Siegfried Bell **Peter Brockhausen**

Dortmund, April 3, 1995

Universität Dortmund
Fachbereich Informatik

Abstract

Knowledge discovery in databases is not only the nontrivial extraction of implicit, previously unknown and potentially useful information from databases. We argue that in contrast to machine learning, knowledge discovery in databases should be applied to real world databases.

Since real world databases are known to be very large, they raise problems of the access. Therefore, real world databases only can be accessed by database management systems and the number of accesses has to be reduced to a minimum. Considering this property, we are forced to use, for example, standard set oriented interfaces of relational database management systems in order to apply methods of knowledge discovery in databases.

We present a system for discovering data dependencies, which is build upon a set oriented interface. The point of main effort has been put on the discovery of value restrictions, unary inclusion- and functional dependencies in relational databases. The system also embodies an inference relation to minimize database access.

1 Introduction and Related Works

Data dependencies are the most common type of semantic constraints in relational databases which determine the database design. Despite the advent of highly automated tools, database design still consists basically of two types of activities: first, reasoning about data types and data dependencies and, second, normalizing the relations. Automatic database design may serve as a process to support database designers with a dependencies proposing system, which may help to design optimal relation schemes for those cases where data dependencies are not obvious. The so called dependency inference problem is described in [Mannila and Räihä, 1991] as: Given a relation r, find a set of data dependencies which logically determines all the data dependencies which are valid in r.

Unfortunately, it is impractical to enumerate all data dependencies and to try to verify each of them. Alternatively, a second approach to the dependency inference problem is to avoid unnecessary queries by inferring as much as possible from already verified data dependencies. A third approach is to draw inferences not only from verified data dependencies but also from invalid data dependencies. In this paper we will follow the latter approach.

To address these problems we present an inference relation on valid and invalid data dependencies and show how a set oriented language like SQL can be used for testing data dependencies. We exemplify this by value restrictions, unary inclusion and functional dependencies. The plot of this paper is as follows: In section 2 value restrictions of attributes, functional and unary inclusion independencies are introduced in order to improve the inference of the dependencies. Then, the corresponding inference relations are discussed. In section 3 we describe the architecture of our system, show how to test dependencies by SQL queries and describe the implementation of the former defined ineference process. Also, the complexity of this inference is discussed. We conclude with empirical results and a comparison with similar systems.

In general, knowledge discovery in databases (KDD) incorporates the same problems as the above approaches. First, it is impractical to test all hypotheses and second, the only interface to the database is a database management system.

Knowledge discovery is not only *the nontrivial extraction of implicit, previously unknown, and potentially useful information from data*, as defined by Piatetsky-Shapiro and Frawley [Piatetsky–Shapiro and Frawley, 1991], but typically has also the following properties[1]: high level language, accuracy and efficiency. High level language means that the discovered knowledge is represented in a high level language in order that its expressions are understandable by (non technical) humans. Accuracy means that the discovered knowledge should reflect the contents of the database exactly, and if not, the imperfect is expressed by measures of certainty. Efficiency is a matter of the discovery process and says that the process is efficient and the running times for large sized databases are predictable and acceptable. It is easy to see, that data dependencies fulfill the first two conditions, the third condition is a matter of the underlying techniques which are discussed later.

Therefore, adapting approaches of machine learning to KDD should consider all these properties. For example, in CLAUDIEN, cf. [Dehaspe et al., 1994], the high level language has been taken into account, but the efficency requirement has been neglected because the

[1] Mentioned by C. Lee

system can not be applied to large sized databases. Thus, our presented system can be seen at the first glance as an optimized version of CLAUDIEN regarding functional dependencies, [Dehaspe et al., 1994]. But there are differences: first, in CLAUDIEN the relationship between the dependencies is based on θ–subsumption and the verification of the hypotheses on theorem proving. In our approach, the relationship of the dependencies is based on an axiomatization of FDs and UINDs and the verification is done by the database management system which groups the rows. This offers several advantages: First, in contrast to we can infer dependencies by some kind of transitivity which is really simple, theorem proving which is too powerful for this purpose. Second, we can find dependencies in relational databases, which can not be stored in the main memory as PROLOG assertions. In most others ILP learning systems like RDT, cf. [Morik et al., 1993], functional dependencies can not be expressed. Systems, which are closer to ours, are empirically compared in section 4.

2 Terminology and Data Dependencies

Familiarity is assumed with definitions of relational database theory as given for example in [Kanellakis, 1990]. The uppercase letters A, B, C stand for attributes and X, Y, Z for sets of attributes. By convention we omit the braces. R, S stands for relation schemes and r, s for relations of a database d. Further we assume that our database is finite, i.e. there are only finite many rows in a relation in order to ensure the existence of the axiomatizations. We use tuple, row and entry in a interchangeable way.

Fagin [Fagin, 1981] introduced *domain dependencies*, for example $IN(A, S)$ where A is an attribute and S is a set. It means that the A entry in each tuple must be a member of the set S. For example, let A be the attribute `SALARY`, and let S be the set of all integers between $10,000$ and $100,000$. If A is one of the attributes of relation R, then R obeys $IN(A, S)$ if and only if the `SALARY` entry of every tuple of R is an integer between $10,000$ and $100,000$.

We adapt these constraints to the data types of our database management system and restrict the domains to ordered sets in order to represent them as a pair of lower and upper bounds. Therefore, we have to distinguish only between numeric and symbolic types of the attributes and can use the normal orders on numbers and the lexicographic order on the character set. We denote them as *value restrictions*:

Definition 1 (Value Restrictions) *Value restrictions are defined as follows:*

$val(A) = [a_i, a_j, \tau] |$ *each value of the attribute* A *is of type* τ *and in the interval* $[a_i, a_j]$

For example, $val(\texttt{SALARY}) = [10000, 100000, number]$ means that the entry of the attribute `SALARY` is an integer between $10,000$ and $100,000$.

An inclusion dependencies (IND) says that values in columns of one relation must also appear as values in columns of some other relation. Unary inclusion dependencies (UINDs) restrict this definition to the case, where only one singleton attribute is allowed as column. As an example, every `MANAGER` entry in a relation R appears as an `EMPLOYEE` entry in a relation S which will be abbriviated by $R(MANAGER) \subseteq S(EMPLOYEE)$. Originally, the concept of INDs in relational database theory has been a generalization of Codd's notion of a *foreign key*.

According to Kanellakis [Kanellakis, 1990] a sound and complete axiomatization for unary inclusion dependencies (UIND) is given by the following definition:

Definition 2 (Inference of Unary Inclusion Dependencies (UINDs)) *Inference rules of unary inclusion dependencies are given by:*

$$U1 : (Reflexivity) \;\; A \subseteq A$$

$$U2 : (Transitivity) \;\; \frac{A \subseteq B, B \subseteq C}{A \subseteq C}$$

Functional dependencies (FDs) are the most important dependencies between attributes, and $X \rightarrow Y$, for example, says that every pair of tuples that agree in the X entries must also agree in the Y entries. An axiomatization was given by Armstrong, and it is usually called Armstrong's axiomatization, cf. [Ullman, 1988].

Definition 3 (Axiomatization of FDs) *X, Y and Z are sets of attributes. An axiomatization of FDs is given by:*

$$FD1 : (Reflexivity) \text{ If } X \subseteq Y \text{ then } Y \rightarrow X$$

$$FD2 : (Augmentation) \text{ If } W \subseteq V \text{ then } \frac{X \rightarrow Y}{XV \rightarrow YW}$$

$$FD3 : (Transitivity) \;\; \frac{X \rightarrow Y, Y \rightarrow Z}{X \rightarrow Z}$$

Kanellakis, Cosmadakis and Vardi [Kanellakis et al., 1983] have investigated the relationship between FDs and UINDs, and shown that there is no axiomatization in the unrestricted case; in the finite case, there is only a axiomatization in the presence of *cardinality dependencies.*

We [Bell, 1995] have also investigated the relationship of UINDs and FDs, and have given an axiomatization, regarding the so called *independencies*. Independencies have been introduced by Janas [Janas, 1988], but the given axiomatization was not complete. Functional independencies mirror functional dependencies, but they are meant for a totally different purpose: they are not semantical constraints on the data, but a support for the database designer in the task of identifying functional dependencies and they also improve the inference of functional dependencies. For example, if we know that the FD $X \rightarrow Y$ is valid and $Z \rightarrow Y$ is not valid, then we can conclude that the FD $Z \rightarrow X$ cannot be valid too. The reason follows immediately from the definition of functional independencies which is simplified here by ignoring null values.

Definition 4 (Functional Independency (FI)) *$X \not\rightarrow Y$ denotes a functional independency. A relation r satisfies $X \not\rightarrow Y$ if there exist tuples t_1, t_2 of r with $t_1[X] = t_2[X]$ and $t_1[Y] \neq t_2[Y]$.*

Unary inclusion independencies can be defined in a similar way.

Definition 5 (Unary Inclusion Independency (UINI)) *$R[A] \not\subseteq S[B]$ denotes a unary inclusion independency. A database satisfies $R[A] \not\subseteq S[B]$ if there exists a tuple t_1 of r with $t_1[A] \notin s[B]$.*

There are interactions between FDs and UINDs and their corresponding independencies which are described by an axiomatization. In our system we do not use a complete axiomatization but only a subset of inference rules, because we do not exploit the cardinalities of the attributes. But we have a certain order of dependencies in our inference process. First, we determine the value restrictions. Second, we determine the UINDs and third the FDs. Therefore, the Armstrongs Axioms, the axiomatization of UINDs and the following inference rules are only of interest:

Definition 6 (Inference) *Let X,Y and Z be sets of attributes of the same relation if not mentioned otherwise.*

1. *The interaction of FDs and FIs can be described by the following rules:*

$$FI1: \frac{XV \not\rightarrow YW, W \subseteq V}{X \not\rightarrow Y}$$

$$FI2: \frac{X \rightarrow Y, X \not\rightarrow Z}{Y \not\rightarrow Z}$$

$$FI3: \frac{Y \rightarrow Z, X \not\rightarrow Z}{X \not\rightarrow Y}$$

2. *The interaction of UINDs and UINIs can be described by:*

$$UI1: \frac{R[A] \subseteq S[B],\ R[A] \not\subseteq T[C]}{S[B] \not\subseteq T[C]}$$

$$UI2: \frac{R[A] \subseteq T[C],\ S[B] \not\subseteq T[C]}{S[B] \not\subseteq R[A]}$$

3. *There is also an interaction between UINDs, UINIs and FIs:*

$$\text{I6}: \frac{R[A] \subseteq R[B], R[B] \not\subseteq R[A]}{A \not\rightarrow B}$$

Obviously, the operator $\subseteq$ is overloaded. But it should be clear from the context whether the normal subset relation or a UIND is intended. The correctness is proven in [Bell, 1995]. The rules are given in a natural deduction style. For example, the first rule says, that if we know the FI $XV \not\rightarrow YW$ is valid and that W is a subset of V, then the FI $X \not\rightarrow Y$ must be the case too. The rules $UI1$ and $UI2$ describe the interaction between UINDs and UINIs, whereas the rule $I6$ describes the interaction between UINDs, UINIs and FIs.

Additionally, we have some inference rule, which infers from the type and the upper and lower bounds the corresponding unary inclusion independencies. For example if $val(B) = [b_i, b_j, \tau_b]$ and $val(A) = [a_i, a_j, \tau_a]$ and $\tau_b \neq \tau_a$, then $A \not\subseteq B$ and $B \not\subseteq A$. Another rule is: if $val(B) = [b_i, b_j, \tau]$ and $val(A) = [a_i, a_j, \tau]$ and $b_j < a_i$, then $A \not\subseteq B$ and $B \not\subseteq A$. The correctness of these rules can be seen easily.

The discovery of value restrictions and UINDs is easier than the discovery of FDs. Therefore, we need some more terminology for the discovery of FDs: The discovery of FDs may be visualized as a search in semi lattices consisting of nodes and edges. The nodes are labeled with data dependencies and the edges describe a relationship between the nodes. In general, this relationship can be described as a *more general than* relationship as in [Savnik and Flach, 1993]:

Definition 7 (More general) *Let X and Y be sets of attributes such that $X \subseteq Y$, then the dependency $X \rightarrow A$ is more general than the dependency $Y \rightarrow A$, or $Y \rightarrow A$ is more specific than $X \rightarrow A$.*

An example of such a semi lattice can be found later in figure 8. This definition corresponds to the usual more general definition in machine learning, i.e. the θ–subsumption introduced by Plotkin [Plotkin, 1970]. The relationship also reflects our inference rules, i.e. the inference rule $FD2$ states that if a relation satisfies a functional dependency, then the relation satisfies each more specific dependency too. For example, if a relation satisfies the functional dependency $AB \rightarrow C$, then the relation satisfies $ABD \rightarrow C$. We can also adapt this concept to independencies. Then we say for example: $AB \not\rightarrow C$ is more general than $ABD \not\rightarrow C$ or $ABD \not\rightarrow C$ is more specific than $AB \not\rightarrow C$ according to inference rule $FI1$.

This relationship implies a partial ordering which simplifies the discovery of functional dependencies by a simple representation: the set of functional dependencies can be partitioned into equivalence classes by the satisfiability definition. Each class of functional dependencies specifies the same set of admissible relations. As these equivalence classes will typically contain a large number of elements it is only reasonable to define a suitable representation with a minimal number of elements. This representation is usually called minimal cover. We do not use a minimal cover as defined in database theory, therefore we call the cover the most general cover. The difference is shown by the following example: The set $\{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$ is most general in our sense, but not minimal as defined in database theory, because the transitivity rule is applicable.

Definition 8 (Most General Cover) *The set of functional dependencies F is a most general cover if for every dependency $X \rightarrow A \in F$, there exists no Y with $Y \subset X$ and $Y \rightarrow A \in F$.*

In the next section we put the point of main effort on data structures and the implementation of such inferences and investigate the costs.

3 Discovering Data Dependencies

In this section we present the algorithms to infer integrity constraints, unary inclusion dependencies and functional dependencies, for more details see [Brockhausen, 1994]. But we start with an overview of the architecture of our system.

3.1 Architecture

The input for our system consists of the data of a relational database, here Oracle Server 7, with the corresponding database scheme, represented in the system tables. The communication between our system implemented in Prolog and the DBMS takes place over a network by means of the TCP/IP protocol. Hence we can use any Oracle database which is worldwide reachable over the Internet.

We generate the SQL queries in Prolog and the interface pushes them forward to the database. The answer, a set of tuples, has to be converted to Prolog terms. The interface is responsible for this too. Normally the DBMS offers many different numeric and alphanumeric data types. But theses types like CHAR or VARCHAR2 in OracleV7 are mainly meant for storage efficiency reasons for example and do not imply any fundamental differences in the data which justify a separate treatment in the algorithms below. Therefore it makes sense to gather all different numeric and alphanumeric types in "generic" types NUMBER and STRING, which are mapped in turn onto the Prolog types NUMBER and ATOM. Data of type RAW is suppressed.

Our system is build up by a hierarchie of three algorithms, i.e. part of the output of one algorithm is used as input for the algorithm above. The dotted square symbolizes that we compute these restrictions, but normaly they do not have a semantic meaning. But nevertheless, they are useful wrt. the computation of the UINDs.

3.2 Value Restrictions

We consider value restrictions or the upper and lower bounds of attribute domains. We select the minima and maxima for all attributes in all relations with the corresponding SQL statements. The SQL statement uses the normal order on numbers for numerical attributes and the lexicographic order on the character set for attributes of a symbolic type. Since it is possible to compute the two values in one query, the overall costs are $\mathcal{O}(n * m)$. Throughout this section n denotes the number of attributes in all tables and m the maximal number of tuples in the table which possesses the most.

The third argument, i.e. the type, is determined by the two Oracle data types, as already mentioned above, in order to infer UINDs.

3.3 Unary Inclusion Dependencies

Unary inclusion dependencies can be computed by taking advantage of the transitivity and of a run through all possible combinations in a special sequence. First, we start with the presentation of the necessary SQL statements and conditions for calculating the UINDs in figure 2.

The results of the queries are numbers. It is possible to combine the second and third statement in one query, because in some cases both UINDs $A \subseteq B$ and $B \subseteq A$ are possible, but in others only one UIND. The implemented version of the algorithm always uses the appropriate query. The time complexity of the SQL statements is determined by the join in the first one and is $\mathcal{O}(m^2)$.

The algorithm INCLUSION DEPENDENCIES depicted in figure 3 is called one time for each kind of the mentioned "generic" data types in the database.

The algorithm uses a graph representation for UINDs. There exists a directed edge from the node A_i to the node A_j, if and only if there exists an UIND $R_p[A_k] \subseteq R_q[A_l]$ in the database and A_i and A_j are numbers which represent the attributes A_k and A_l in the relations R_p and R_q respectively. In the algorithm we denote by $A_i \subseteq A_j$ the edge in the graph as well as the corresponding UIND.

The computational costs of step 1 in the algorithm are $\mathcal{O}(n^2)$, because all combinations between two attributes are considered. But here, since we do not pose any database query, we do not exploit the transitivity between intervalls, which otherwise will result in computational costs which are at least as high in the best case.

The correctness of the algorithm is considerably based on the following lemma. It is a direct consequence of the axiomatization of dependencies and independencies, cf. [Bell, 1995], and the proof is done by contradiction concerning the transitivity of UINDs.

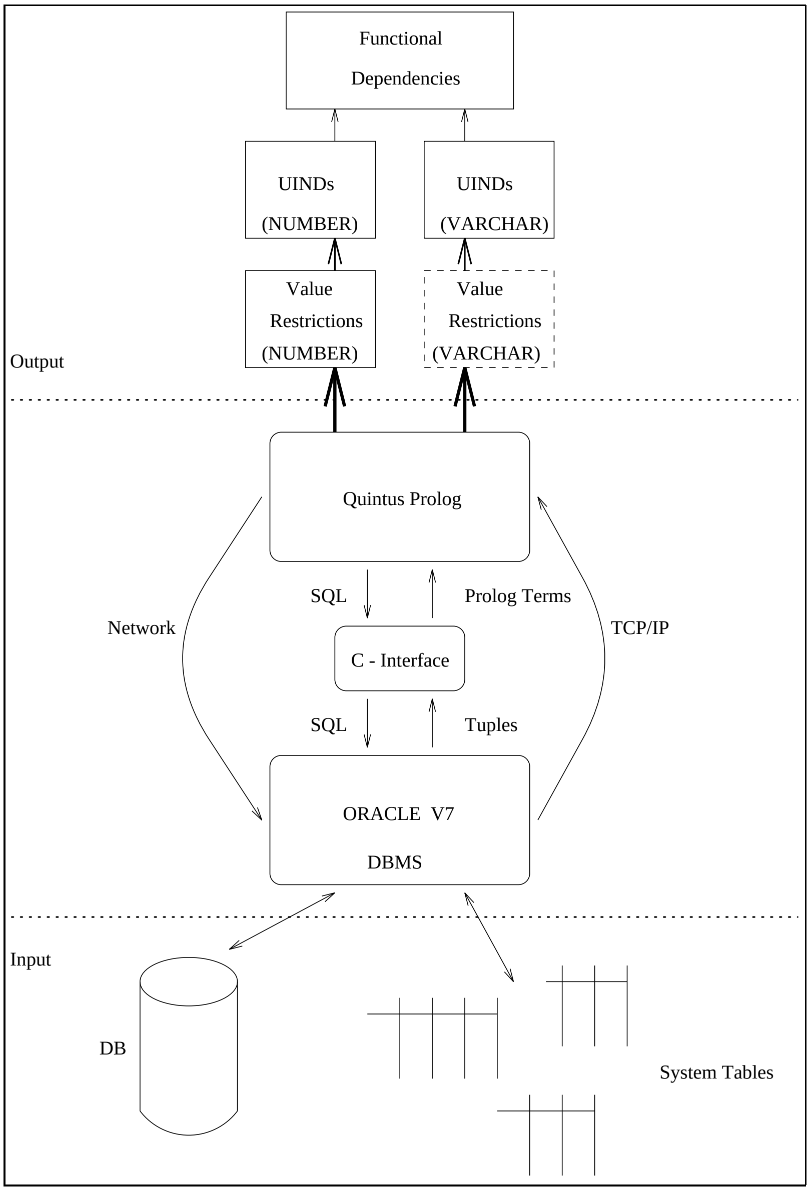


Figure 1: Systemoverview

1. SELECT COUNT(DISTINCT $R_i.A_1$)
   FROM R_i, R_j
   WHERE $R_i.A_1 = R_j.A_2$ $=: e$

2. SELECT COUNT(DISTINCT A_1)
   FROM R_i $=: e_1$

3. SELECT COUNT(DISTINCT A_2)
   FROM R_j $=: e_2$

4. $e = e_1 \Rightarrow A_1 \subseteq A_2$

5. $e = e_2 \Rightarrow A_2 \subseteq A_1$

6. $e = e_1 = e_2 \Rightarrow A_1 = A_2$

Figure 2: SQL Statements and Conditions for Calculating UINDs

Lemma 1

1. *If there exists a directed edge from the node A_{i+r} to the node A_k and no edge from the node A_i to the node A_k with $k < i$, then it is impossible that there exists an edge from the node A_i to the node A_{i+r}.*

2. *If there exists a directed edge from the node A_i to the node A_k and no edge from the node A_{i+r} to the node A_k with $k < i$, then it is impossible that there exists an edge from the node A_{i+r} to the node A_i.*

All the other steps in the algorithm are responsible for an ordered run through all possible tests and are trivial. The procedure UPDATE GRAPH discovers the transitive relations between the UINDs. The two steps and the distinction between the two cases guarantee that tests are deleted only in those lists, where they can occur. Hence the list structure becomes "incomplete" and some more cases are needed in the algorithm INCLUSION DEPENDENCIES which we omitted here.

The procedure UPDATE GRAPH has a time complexity of $\mathcal{O}(n + e)$, where n and e denote the number of nodes and edges as usual. For example in the case $i < j$ we have to execute a depth–first search or breadth–first search in step 1a and 1b. Deleting of tests can be done on the run and in time $\mathcal{O}(1)$, but one has to change the data structure at step 4 in the algorithm INCLUSION DEPENDENCIES from a list structure to arrays in order to achieve this result, which is a simple transformation, but would complicate the presentation here.

A naive algorithm for computing inclusion dependencies has a time complexity of $\Theta(n^2 * m^2)$. It generates exactly $\frac{n*(n-1)}{2}$ database queries, if the corresponding UINDs are valid or not. In contrast the algorithm INCLUSION DEPENDENCIES has a overall time complexity of $\mathcal{O}(n^4 + n^2 * m^2)$. The summand $\mathcal{O}(n^4)$ is caused by the nested loop and each call to UPDATE GRAPH.

Algorithm: INCLUSION DEPENDENCIES
Input: A list of all attributes of one type
Output: A list of all inclusion dependencies between attributes of one type

1. Compute all candidate attributes for UINDs, which fulfill the condition: the interval, made up by the minimal and maximal value for this attribute — these are the value restrictions — is a subset or a superset for any other attribute of this type.

2. Number all attributes from A_1 up to A_n.

3. Construct a directed graph with nodes A_i and edges $A_i \rightarrow A_j$, iff A_j is marked in the system table as a foreign key for A_i.

4. Construct the following list structure:
   $$\Big[[A_1 : [\underline{A_2}, \overline{A_2}], [\underline{A_3}, \overline{A_3}], \ldots, [\underline{A_n}, \overline{A_n}]]$$
   $$[A_2 : [\underline{A_3}, \overline{A_3}], \ldots, [\underline{A_n}, \overline{A_n}]]$$
   $$\vdots$$
   $$[A_{n-1} : [\underline{A_n}, \overline{A_n}]]\Big]$$
   $\underline{A_j}$ and $\overline{A_j}$ respectively are symbols for the tests, if the UINDs $A_i \subseteq A_j$ or $A_j \subseteq A_i$ are valid. The list of A_i contains $\underline{A_j}$ or $\overline{A_j}$ with $j > i$, if there does not exist a path in the graph from A_i to A_j or A_j to A_i respectively.

5. For all A_i with $1 \leq i < n$ do:

   (a) Let $\underline{A_{i+r}}$ with $r \in \{1, \ldots, n-i\}$ be the next test. If there exists an edge from A_{i+r} to a node A_k with $k < i$ and no edge from A_i to A_k, then continue at step 5b with the next test, else execute the test. If $A_i \subseteq A_{i+r}$ is valid, then call UPDATE GRAPH with $A_i \subseteq A_{i+r}$ and continue at step 5b, else continue directly at step 5b.

   (b) Let $\overline{A_{i+r}}$ with $r \in \{1, \ldots, n-i\}$ be the next test. If there exists an edge from A_i to a node A_k with $k < i$ and no edge from A_{i+r} to A_k, then continue at step 5a with the next step, else execute the test. If $A_{i+r} \subseteq A_i$ is valid, then call UPDATE GRAPH with $A_{i+r} \subseteq A_i$ and continue, else continue.

   (c) While the list of the tests for A_i is not empty, continue at step 5a with the next test $\underline{A_{i+r+1}}$.

6. Return all edges of the graph as UINDs

Figure 3: Algorithm INCLUSION DEPENDENCIES

Procedure: UPDATE GRAPH

Input: One valid UIND $A_i \subseteq A_j$

1. Insert the edge $A_i \rightarrow A_j$ into the graph.

2a) $i < j$

(a) Find all nodes A_k, $k > i$, from which exists a path to the node A_i.
(b) Find all nodes A_l, $l > i$, which are reachable from A_j.
(c) Delete all tests $\underline{A_l}, l > j$ in the list A_i.
(d) Delete all tests $\underline{A_l}, k < l$ in the lists A_k.
(e) Delete all tests $\overline{A_k}, k > l$ in the lists A_l.

2b) $i > j$

(a) Find all nodes A_k, $k > j$, from which exists a path to the node A_i.
(b) Find all nodes A_l, $l > j$, which are reachable from A_j.
(c) Delete all tests $\overline{A_k}, k > i$ in the list A_j.
(d) Delete all tests $\underline{A_l}, k < l$ in the lists A_k.
(e) Delete all tests $\overline{A_k}, k > l$ in the lists A_l.

Figure 4: Procedure UPDATE GRAPH

At a first glance, this result looks strange because of the $\mathcal{O}$–notation. But our algorithm has one very important property. Given a fixed numbering of the attributes at step 2, the algorithm presented here always poses a minimal number of database queries for the discovery of UINDs, by exploiting the transitivity of UINDs and hence it saves all superfluous queries to the database.

It can be shown that there exist "good" and "bad" numberings of the attributes in step 2, cf. example 2, resulting in different numbers of "necessary" database queries. But even if the numbering is a worst case one, as long as there exists at least one valid UIND in the database, our algorithm saves at least one database query.

And since one database query — given a "real" database and measured in cpu–time — takes considerably longer than our whole algorithm INCLUSION DEPENDENCIES without the database queries, the extra amount of work with time complexity $\mathcal{O}(n^4)$ is more than justified. And for this reason it does not matter if it is possible to drop the time complexity of summand $\mathcal{O}(n^4)$, which seems possible, because it would not save one more database query.

Now, we present a short example demonstrating our algorithms INCLUSION DEPENDENCIES and UPDATE. But it serves also as an illustration of one drawback of our implemented approach.

Example 1 *We are looking for all UNIDs between A_1 and $A_2, \ldots, A_n$. Suppose that a transitive chain $A_n \subseteq A_{n-1} \subseteq \ldots A_2$ exists, because of the foreign key entries in the system table. Then after step 3 in the algorithm* INCLUSION DEPENDENCIES *all these edges are inserted into the graph. If $A_1 \subseteq A_2, A_1 \subseteq A_3, \ldots, A_1 \subseteq A_n$ and $A_2 \not\subseteq A_1, A_3 \not\subseteq$*

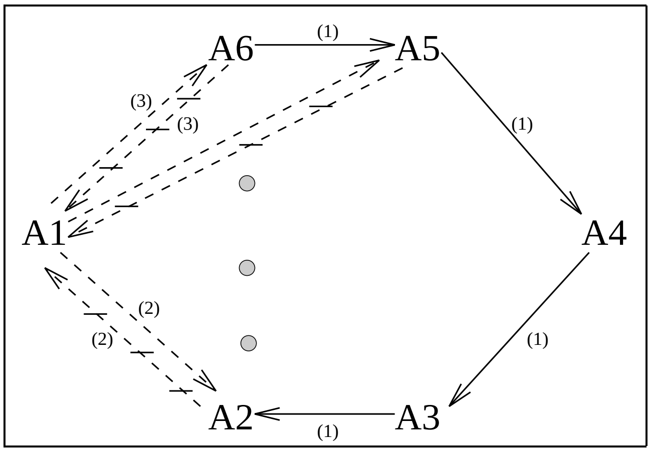


Figure 5: Sketch to example 2

$A_1, \ldots A_n \not\subseteq A_1$ are valid UINDs and UINIs respectively, then starting with A_1 we will pose $2 * n$ database queries. But if we were starting with A_n we would only need 2 SQL queries. This is caused by the transitivity and the fact that $A_{n-1} \not\subseteq A_1, \ldots, A_2 \not\subseteq A_1$ are UINIs, cf. lemma 1. ◇

Example 2 *Figure 5 illustrates the last example.* (1) *depicts the edges inserted after step 3 in the algorithm* INCLUSION DEPENDENCIES *and assume that no other UINDs are valid. If we start with* (3) *and the test $\underline{A_6}$, then the following tests are deleted by* UPDATE*: $\underline{A_5}, \ldots, \underline{A_2}$. After the test $\overline{A_6}$ the remaining tests $\overline{A_5}, \ldots, \overline{A_2}$ are removed. Starting with* (2) *would be a worst case scenario in this example.* ◇

It is easy to overcome this problem by altering our algorithm. First, look for these chains and then, by using a heuristic choose a "good" numbering.

3.4 Functional Dependencies

We start this subsection with a presentation of the necessary SQL statement in order to compute functional dependencies. Figure 6 lists the statement and the condition which must hold. The clue is the GROUP BY instruction. The computational costs of this operation are dependent on the database system, but it can be done in time $\mathcal{O}(m * log\, m)$. The statement itself counts the different values in each group and sums up over all groups. It is sufficient to count only the different values for the attribute A_1, because this number is the same for all attributes A_1 up to A_n. But it is important that the attribute B, the right hand side of the hypothesis, does not appear as an attribute in the grouping. And since we are looking for most general FDs, it is assured, that the attributes $A_1, \ldots, A_n, B$ are all distinct. The statement returns a binary tuple. If the two numbers are the same then the hypothesis is true, that means that the corresponding functional dependency holds in the database.

Example 3 *The relation R contains the attributes $A_1, \ldots, A_3$, B and C. The hypotheses $A_1 A_2 \rightarrow B$ and $A_2 A_3 \rightarrow B$ are to be verified. The first table in figure 7 depicts the relation*

1. SELECT SUM (COUNT (DISTINCT A_1)),
   SUM (COUNT (DISTINCT B))
   FROM R
   GROUP BY $A_1, \ldots, A_n$ $=: a_1, b$

2. $a_1 = b \Rightarrow A_1 \ldots A_n \rightarrow B$

Figure 6: A SQL statement for the Computation of Functional Dependencies

R	A_1	A_2	A_3	B	C
	3	3	11	d	f
	2	1	7	d	f
	2	2	9	c	i
	3	3	11	d	g
	1	2	9	a	p
	2	2	9	c	h

A_1	A_2	B
1	2	a
2	1	d
2	2	c
3	3	d

A_2	A_3	B
1	7	d
2	9	a
2	9	c
3	11	d

Figure 7: A sample relation to demonstrate the SQL query

R, the other two tables show the intermediate results of the SQL queries. The results of the queries wrt. the hypotheses are as follows: $a = b = 4$ and $a = 3 \neq 4 = b$ respectivly. It follows that $A_1A_2 \rightarrow B$ is a FD and $A_2A_3 \not\rightarrow B$ a FI. $\diamond$

All the attributes for the left–hand side of most general FDs of the type $X \rightarrow F$ build a semi lattice. Figure 8 shows the intuitive reduction of this semi lattice into a tree structure where we have left away the right–hand sides, which is always the attribute F in this example.

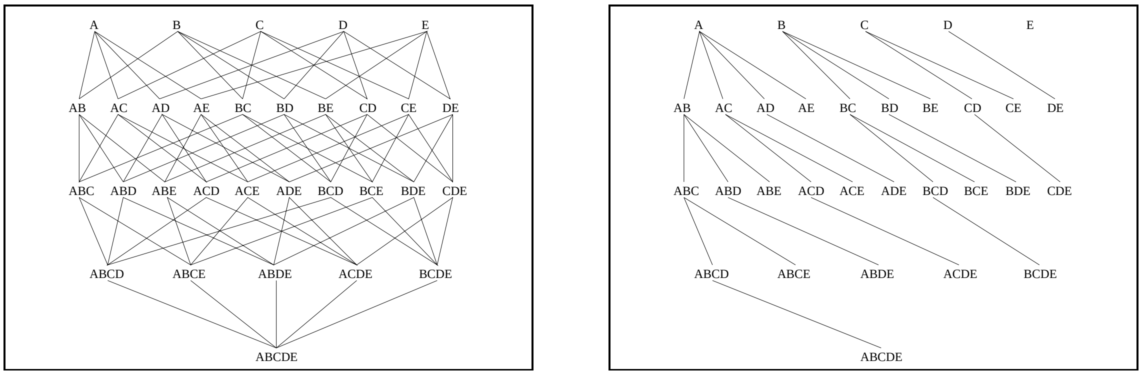


Figure 8: Reduction of a semi lattice into a tree structure

Algorithm: FUNCTIONAL DEPENDENCIES
Input: All attributes $A_1, \ldots, A_n$ of the relation
Output: All discovered most general FDs

1. Compute the class of attributes NKNN.
2. For each attribute A_i do:
   (a) Compute a list LHS(A_i) of all possible attributes for the left–hand side of most general FDs $X \rightarrow A_i$.
   (b) Compute a list UH(A_i) of all possible attributes for unary hypotheses $A_k \rightarrow A_i$
3. For each attribute A_i do:
   IF bottom–up–search THEN top–down–search

Figure 9: The Algorithm FUNCTIONAL DEPENDENCIES

In the algorithm FUNCTIONAL DEPENDENCIES we have integrated two main ideas, namely to exploit the transitivity of FDs and to concentrate on the computation of most general FDs.

Every attribute in a relation can be classified in one of three disjunct classes. We denote the first class with UCK, that means unary candidate key. Attributes which contain only distinct values and no NULL–values belong to this class. Some of them for example may be marked in the system table of the database as the unary primary key or as a unique index and so on. All the attributes of this class are keys and therefore they build the left–hand sides of most general FDs, which the algorithm need not to generate anymore.

Other attributes contain NULL–values. They build up the second class NK, "no key". All these attributes trivially do not imply any other attribute and more important they are useless for specializations of hypotheses which correspond to an invalid most general FD.

As a consequence only the attributes of the third class NKNN, that means "no-key-no-null-values", are needed for the left–hand sides during the search for unknown most general FDs. For the computation of the class NKNN we exploit the information in the system table of the database and analyze the data itself where needed. The time complexity of this operation, which is the first step in our algorithm, cf. figure 9, is $\mathcal{O}(n * m)$.

The second step in figure 9 mainly initializes data structures for the following third step. But if we are looking for FDs of the form $X \rightarrow B$ and the attribute B is an element of the class UCK then we need not consider any unary hypotheses in the third step with the attribute B on the right–hand side, because they all are invalid FDs. This is also recognized in this step which has a time complexity of $\mathcal{O}(n^2)$.

The function bottom–up–search in the next step is quite simple. Assume that the attributes $A_1, \ldots, A_n$ are possible attributes for the left–hand side of a most general FD $X \rightarrow B$. Then we test the most special hypothesis $A_1 \ldots A_n \rightarrow B$. If the corresponding FD is not true then we need not consider this search space. Otherwise the function returns true and the function top–down–search will be called.

1. **Top–Down–Search–Start**
   Test all unary hypotheses $A_i \to B$ from UH(B). If $A_i \to B$ is a valid FD, then do:
   (a) Delete A_i in LHS(B).
   (b) Call the procedure UPDATE–FD with $A_i \to B$.

   If LHS(B) $= \emptyset$ then RETURN

2. **Top–Down–Search**
   Sort the names of all attributes in the list LHS(B) in ascending order. Let $A_1, \ldots, A_n$ be the attributes in LHS(B). Construct all two place combinations A_iA_j, $i < j$, $i, j \in \{1, \ldots, n\}$ and insert them into the queue *QUEUE*.

   WHILE *QUEUE*$\neq \emptyset$ DO
   Let $A_{r_1} \ldots A_{r_k}$ be the first element of *QUEUE*.
   IF NOT(HAS–FD–AS–SUBSET($A_{r_1} \ldots A_{r_k}$))
   THEN verify the hypothesis $A_{r_1} \ldots A_{r_k} \to B$ at the database.
   IF $A_{r_1} \ldots A_{r_k} \to B$ is valid
   THEN call UDATE–FD with $A_{r_1} \ldots A_{r_k} \to B$.
   ELSE construct all $k+1$–place sons $A_{r_1} \ldots A_{r_k}A_l$
   with $l \in \{k+1, \ldots, n\}$. Put these at the end of *QUEUE*
   OD
   RETURN

Figure 10: Function: top–down–search

The funcion top–down–search is twofolded. This distinction between the two phases is useful, because if A is an attribute on the left hand side of a most general FD, then we need not consider any combination of attributes on the left hand side, which entails A. This is realized by the statement (a). Therefore, our top–down–search with the breadth–first and left–to–the–right strategy starts with two place hypotheses. Then at every step, we take the first element of the queue, test the hypothesis and if the test is negative, the node in the tree is expanded and the direct children are put in left to right order at the end of the queue.

But we need some more procedures namely UPDATE–FD and HAS–FD–AS–SUBSET, which can be found in [Brockhausen, 1994]. The latter detects the following situation, cf. example 4 which is always present by the nature of a lattice and which is impossible to avoid in general.

Example 4 *Let $CE \rightarrow F$ be a newly detected most general FD and AB is a FI, cf. figure 8. Then it is wrong to delete C and E in LHS(F). But one have to exclude , that C and E together are part of the left hand sides of later generated hypotheses. For example the successors ABC and ABE from AB have to be generated but $ABCE$ and $ABCDE$ not.*

As the global data structure for the exploitation of the transitivity of FDs, we use a graph structure similar to the one described for the algorithm INCLUSION DEPENDENCIES. Here again we start with the known most general FDs as edges, i.e. the unary primary keys, and after the detection of new FDs by database queries or by inference, the graph is updated by the procedure UPDATE–FD. The inference already starts at the classification of the attributes into the disjunct classes. UPDATE–FD has some drawbacks on the lists LHS(A_i) and UH(A_i) too, where we omitted the details here in order not to complicate the presentation of the algorithm in figure 10.

The procedure for deriving one new FD because of the transitivity has a running time $\mathcal{O}(l + e)$, where l denotes the number of nodes in the graph. At the moment we still use search procedures like DFS or BFS in a graph which also exploit the known independencies but we do not use any theorem prover.

But this search procedure can be called l times in the worst case. And worst in this case is the fact that the number of nodes in the graph can be exponential in n, the number of attributes. Even if we have n attributes and $\mathcal{O}(n)$ tuples in a single table, it is possible that there exists $\Omega(2^{\frac{n}{2}})$ most general FDs, as shown in [Mannila and Räihä, 1991], or correspondingly nodes in the graph.

We should mention that we also use the discovered inclusion dependencies in the algorithm above. If we know that the set of values of the attribute A is a proper subset of the attribute B, then A cannot functionally determine B or $A \not\rightarrow B$.

At the moment, we are interested in "correct" FDs, either the FD $X \rightarrow B$ is valid or not, i.e. $X \rightarrow B$ is a FI. But certainly, a database contains "noise" in many ways, which we will not discuss here. But if we want to cope with this problem, all we have to do is to change the statement $a_1 = b$ in figure 6 into $\mid a_1 - b + \delta \mid \leq \epsilon$. Here ϵ denotes a threshhold, i.e. the number of allowed tuples, which "contradict" the FD. δ is a "correction factor" in order to deal with NULL–values and attributes of the class NKNN. We get this value as a side effect of the classification of the attributes.

Algorithm	Data Base	$\|r\|$	$\|R\|$	$\|X\|$	Time
Savnik/Flach	Lymphography	150	19	7	9 min
Schlimmer	Breast Cancer	699	11	4	1 h 14 min
Bell/Brockh.	Lymphography	150	19	7	> 33 h
Bell/Brockh.	Breast Cancer	699	11	11	8 min 53 sec
Bell/Brockh.	Breast Cancer	699	11	4	4 min 19 sec

Table 1: Comparison of the Experimental Results from [Savnik and Flach, 1993] and [Schlimmer, 1993] with the algorithm FUNCTIONAL DEPENDENCIES.

Database	$\|r\|$	$\|R\|$	$\|X\|$	Time	N
Books	9931	9	9	4 h 44 min.	25
Books	9931	9	6	4 h 40 min.	25
Books	9931	9	3	2 h 10 min.	20

Table 2: Summary of the results of the algorithm FUNCTIONAL DEPENDENCIES.

4 Evaluation and Conclusions

We compared our algorithm with two approaches: Savnik and Flach call their method "bottom–up induction of functional dependencies from relations"[Savnik and Flach, 1993]. Briefly, they start with a bottom–up analysis of the tuples and construct a negative cover, which is a set of FIs. Therefore they have to analyze all combinations between any two tuples. In the next step they use a top–down search approach similar to ours in order to discover the functional dependencies. They check the validity of a dependency by searching for FIs in the negative cover. Schlimmer also uses a top–down approach, but in conjunction with a hash–function in order to avoid redundant computations [Schlimmer, 1993].

But in contrast to our algorithm, in both articles mentioned, the authors do not use a relational database like OracleV7 or any other commercial DBMS. They even do not use a database at all. And this has some important effects on the results, which will be discussed in the next paragraph. Table 1 shows a summary of their results, where $|r|$ denotes the number of tuples, $|R|$ the number of attributes, $|X|$ the maximal number of attributes on the left–hand side of a FD and time is the time needed for the discovery of the most general cover. For comparison reasons we introduced such a bound on the number of attributes in our algorithm.

First, our algorithm cannot detect the FDs in the Lymphography domain in reasonable time, because we do not hold the data in main memory like Savnik and Flach. And since most of the FDs are really long, for some attributes the shortest most general FDs have already seven attributes on the left side, the search space and the overhead for the communication with the database is to big. But it cannot be said that our approach is inferior to the one of Savnik and Flach, because the circumstances are to different, namely the presence or absence of a database for the storage of the tuples.

Second, in the Breast Cancer domain our algorithm is really fast, more than seventeen times faster than Schlimmer's algorithm. Even without any bound on the length of the FDs it is still eight times faster and it uses a database. We conjecture, that this interesting but also unexpected result is mainly caused by the distinction between the three types of attributes in the search for functional dependencies.

But of course the two domains above are not typical database applications. Table 2 shows the results of our algorithm with respect to a real database, the library database of our computer science department. Here it becomes obvious that our pruning criterions are efficient, because with a bound of six attributes and without any bound the time needed is nearly the same. The differences are neglectable because there are many more users working on the network and the results are only reproducible within some bounds. But apart from the known primary key of the database the discovered FDs are semantically meaningless.

Furthermore we have stored the tuples of the databases mentioned above as ordinary PROLOG Facts. In the Breast Cancer domain the results were very surprising, because the database approach is more than four times faster as using eleven place PROLOG predicates, one place for every attribute, and simulating the SQL queries in PROLOG. But the reason is obvious. This kind of representation is not efficient because due to the arity of the predicates which represent the tuples, we have to take into account eleven variables even for testing unary FDs.

In summary, on can say that the algorithm which we present in our work has one important advantage over the two approaches mentioned above. The algorithm is capable of dealing with great amounts of data, because we use a real database for the storage. And as a side effect, because we use standard SQL statements for the discovery of FDs, our approach is portable and we can use any database which "understands" SQL as a query language.

Acknowledgment: This work is partly supported by the European Community (ESPRIT Basic Research Action 6020, project Inductive Logic Programming) and the Daimler–Benz AG, Contract No.: 094 965 129 7/0191.

References

[Bell, 1995] Bell, S. (1995). Inferring data independencies. Technical Report 16, University Dortmund, Informatik VIII.

[Brockhausen, 1994] Brockhausen, P. (1994). Discovery of functional and unary inclusion dependencies in relational databases. Master's thesis, University Dortmund, Informatik VIII. in german.

[Dehaspe et al., 1994] Dehaspe, L., Laer, W. V., and Raedt, L. D. (1994). Applications of a logical discovery engine. In Wrobel, S., editor, *Proc. of the Fourth International Workshop on Inductive Logic Programming*, GMD-Studien Nr. 237, pages 291–304, St. Augustin, Germany. GMD.

[Fagin, 1981] Fagin, R. (1981). A normal form for relational databases that is based on domains and keys. *ACM Transactions on Database Systems*, 6(3):318–415.

[Janas, 1988] Janas, J. M. (1988). Covers for functional independencies. In *Conference of Database Theory*. Springer, Lecture Notes in Computer Science 338.

[Kanellakis, 1990] Kanellakis, P. (1990). *Formal Models and Semantics, Handbook of Theoretical Computer Science*, chapter Elements of Relational Database Theory, 12, pages 1074 – 1156. Elsevier.

[Kanellakis et al., 1983] Kanellakis, P., Cosmadakis, S., and Vardi, M. (1983). Unary inclusion dependencies have polynomial time inference problems. *Proc. 15th Annual ACM Symposium on Theory of Computation*.

[Mannila and Räihä, 1991] Mannila, H. and Räihä, K.-J. (1991). *The design of relational databases*. Addison-Wesley.

[Morik et al., 1993] Morik, K., Wrobel, S., Kietz, J.-U., and Emde, W. (1993). *Knowledge Acquisition and Machine Learning: Theory, Methods and Applications*. Knowledge-Based Systems. Academic Press, London u.a.

[Piatetsky–Shapiro and Frawley, 1991] Piatetsky–Shapiro, G. and Frawley, W. (1991). Knowledge discovery in databases – an overview. In G. Piatetsky-Shapiro, W. F., editor, *Knowledge Discovery in Databases*, pages 1 – 27. AAAI Press, Menlo Park.

[Plotkin, 1970] Plotkin, G. D. (1970). A note on inductive generalization. In Meltzer, B. and Michie, D., editors, *Machine Intelligence*, chapter 8, pages 153–163. American Elsevier.

[Savnik and Flach, 1993] Savnik, I. and Flach, P. (1993). Bottum-up indution of functional dependencies from relations. In Piatetsky-Shapiro, G., editor, *KDD-93: Workshop on Knowledge Discovery in Databases*. AAAI.

[Schlimmer, 1993] Schlimmer, J. (1993). Using learned dependencies to automatically construct sufficient and sensible editing views. In Piatetsky-Shapiro, G., editor, *KDD-93: Workshop on Knowledge Discovery in Databases*. AAAI.

[Ullman, 1988] Ullman, J. D. (1988). *Principles of Database and Knowledge-base Systems*, volume 1. Computer Science Press.