Discovering Functional and Inclusion Dependencies in Relational Databases

Martti Kantola,* Heikki Mannila,† Kari-Jouko Räihä,* and Harri Siirtola*
University of Tampere, Finland and †University of Helsinki, Finland

We consider the problem of discovering the functional and inclusion dependencies that a given database instance satisfies. This technique is used in a database design tool that uses example databases to give feedback to the designer. If the examples show deficiencies in the design, the designer can directly modify the examples. The tool then infers new dependencies and the database schema can be modified, if necessary. The discovery of the functional and inclusion dependencies can also be used in analyzing an existing database. The problem of inferring functional dependencies has several connections to other topics in knowledge discovery and machine learning. In this article we discuss the use of examples in the design of databases, and give an overview of the complexity results and algorithms that have been developed for this problem. © 1992 John Wiley & Sons, Inc.

I. INTRODUCTION

In database design the task of the designer is to describe the structures used to represent the data and to express the *integrity constraints* that restrict the allowed information. Typically, one starts by designing a conceptual schema, for example, an ER diagram. This is then transformed to relational schemas (tables) and integrity constraints. For the design of relational databases, functional and inclusion dependencies are the most important types of integrity constraints.

The database design methods give no guarantee that all the relevant constraints have been found. For example, in drawing an ER diagram it is all too easy to specify some properties of the relationships incorrectly, resulting in missing or erroneous constraints on the relational level.

If, however, one has access to a *database instance,* it is possible to find out which integrity constraints are satisfied by that particular instance. Some of the satisfied constraints may hold by accident, but at least the instance

*Address for correspondence: Heikki Mannila, Department of Computer Science, University of Helsinki, Teollisuuskatu 23, SF–00510 Helsinki, Finland. e-mail: mannila@cs.Helsinki.FI.

illustrates those constraints that are violated. The designer can examine the satisfied constraints and decide which of them should hold in general. The problem of discovering the integrity constraints from a database instance is called in this article the *dependency inference* problem.

We have implemented a database design and analysis tool called Design-By-Example (DBE) that is based on the use of example databases to help locate the constraints in the data.

In this article we discuss the role of examples in database design, describe the DBE design tool, and discuss the algorithmic problems involved in dependency inference. The article is organized as follows. Section II gives a brief overview of database design and integrity constraints. Section III provides a more detailed description of how we can make use of example databases and dependency inference in database design. Section IV discusses briefly Design-By-Example. Section V gives some theoretical results about the feasibility of dependency inference for functional dependencies, and Section VI discusses algorithms for solving the functional dependency inference problem. Section VII considers inference of inclusion dependencies. Section VIII discusses related work. Section IX is a short conclusion.

II. DATABASE DESIGN AND INTEGRITY CONSTRAINTS

Database design is typically divided into requirements analysis, conceptual design, logical design, and physical design. In the conceptual design phase one constructs a conceptual schema of the data to be described in the database. The conceptual schema is typically described by using the entity–relationship model. An example description, a so-called ER diagram, is shown in Figure 1.

The database described in Figure 1 contains information about courses, lecture rooms, and lecture hours. The diamond connecting the three boxes represents a relationship between entities, indicating what lectures are given in which room at a given time.

The logical design phase produces relation schemas from the ER diagram. In the above example the result would contain the schemas

Room (Room_Number, Capacity),
Time (Hour, Description),
Course (Course_Number, Name),

and

MeetsIn (Course_Number, Room_Number, Hour).

The schemas alone are not sufficient for describing the database. In addition, integrity constraints are needed. The information in the ER diagram can be represented using functional and inclusion dependencies.

Functional dependencies are conditions stating that any value of certain attributes is associated with at most one value of a certain attribute. Inclusion dependencies are conditions stating that values occurring in certain columns of one relation must also occur in certain columns of another relation. Thus inclu-

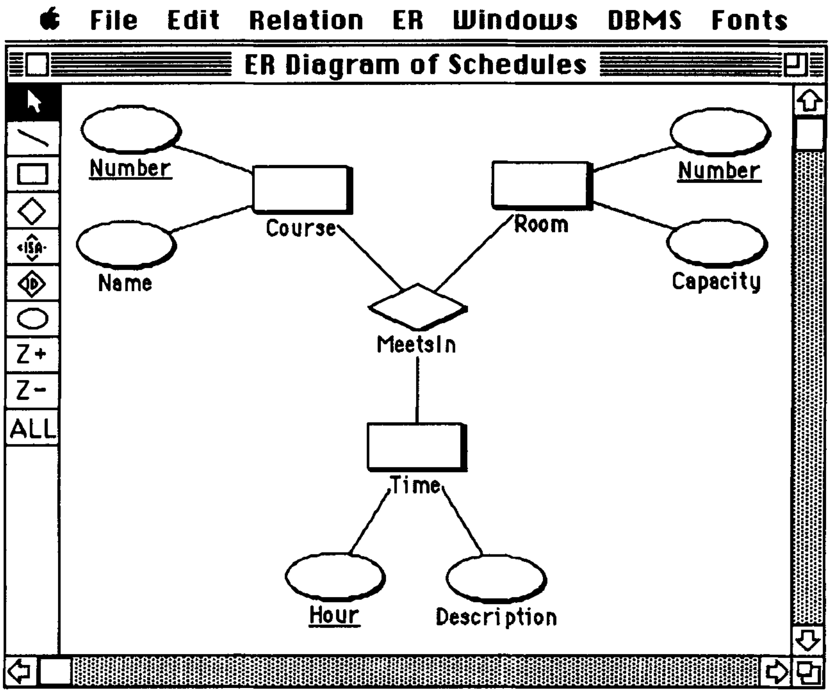


Figure 1. An ER-diagram.

sion dependencies express the referential integrity[1] property that is central in the relational model.

These types of integrity constraints are defined as follows. Given a relation schema (a set of attributes) R, a relation over R is informally a table whose columns are named with the attributes in R. Formally, a relation r is a set of sequences (called *tuples*) with $|R|$ components. The projection of a tuple t on a set of attributes $X \subseteq R$, denoted $t[X]$, is defined as the subsequence of t containing only those values that correspond to the attributes of X.

A *functional dependency* over the schema R is an expression $X \rightarrow Y$, where $X, Y, \subseteq R$. The dependency $X \rightarrow Y$ holds in r, denoted $r \models X \rightarrow Y$, if all tuples $u, v \in r$ with $u[X] = v[X]$ satisfy also $u[Y] = v[Y]$.

An *inclusion dependency*[2] is an expression $R[X] \subseteq S[Y]$, where R and S are relation schemas, X is a sequence of attributes of R, and Y is a sequence of attributes of S. If r and s are relations corresponding to R and S, respectively, then the inclusion dependency $R[X] \subseteq S[Y]$ holds in (r,s) if for each row u of r there is a row v of s such that the sequence of values occurring in the X columns of u is the same as the sequence of values occurring in the Y columns of v. That is, $R[X] \subseteq S[Y]$ holds in (r,s) if $r[X] \subseteq s[Y]$.

Returning to the ER diagram of Figure 1, transforming it to relation schemas gives the following integrity constraints:

Room : Room__Number → Capacity
Time : Hour → Description
Course : Course__Number → Name

MeetsIn[Course__Number] $\subseteq$ Course[Course__Number]
MeetsIn[Room__Number] $\subseteq$ Room[Room__Number]
MeetsIn[Hour] $\subseteq$ Time[Hour]

The task of database design can be formulated as the construction of schemas and integrity constraints such that valid data can be efficiently stored and retrieved from the corresponding relations, and that incorrect data cannot be stored.

In addition to functional and inclusion dependencies, there are several other types of integrity constraints that are needed to specify what is valid data and what is not. Functional and inclusion dependencies are, however, sufficient for modeling the structural aspects of the data.

In this article we consider the problem of finding what functional and inclusion dependencies hold in a given database instance. Since there are only a finite number of syntactically correct dependencies for a given database schema, the problem is solvable: one can, in principle, check all dependencies one by one. However, more efficient algorithms are needed.

III. EXAMPLE DATABASES IN DESIGN

When explaining a database schema to somebody, experienced designers tend to use examples. Some informal database design guidelines[3] suggest the use of example values and relations during the design process. The advantages of examples are easy to see. An attribute name can easily mean different things to different people, but a concrete example of an allowed value for that attribute is more likely to be understood in the same way by everybody, and an example value also shows how the attribute name should be interpreted. Likewise, a relation schema expressing connections between attributes is an abstract concept and prone to misunderstandings. Concrete examples of a relation corresponding to the schema are easier to understand and evaluate.

An arbitrary example relation does not necessarily indicate all the important information about the relation schema or the database schema. An example row or two indicate what type of information can be stored using the schema. This is a valuable piece of knowledge, but such an example does not indicate the interrelationships of different rows and relations implied by the design. A suitable example relation can, however, clearly illustrate the problems in a suggested design.

Example 1. Consider again the design of a database for recording data about courses offered by a department. The design produced a schema with the following columns:

MeetsIn: Course__Number, Room__Number, Hour

No functional dependencies for these attributes were produced by the transformation from the ER diagram to relation schemas.

However, the design probably does not reflect the real world accurately. For instance, nothing prohibits many courses from meeting in the same room at the same time, or one course from meeting in two rooms at any given time.

The following example relation does not satisfy any nontrivial functional dependencies.

Course__Number	Room__Number	Hour
CIS 551	PAC 30	M 9:30
CIS 510	PAC 30	M 9:30
CIS 510	DES 200	M 9:30
CIS 510	DES 200	W 13:30

In this example the problems discussed above are evident: Course CIS 510 meets in two rooms on Monday at 9:30, and Room PAC 30 is used by both CIS 510 and CIS 551 on Monday at 9:30.

Thus a suitably chosen example can be very useful in pinpointing problems in a design. The example table above is in a sense a worst case example: it violates all constraints that have not been explicitly required to hold. The goal is to point out possible omissions in conceptual design by showing what can happen unless more constraints are posed on the tables.

The worst case property can be defined formally by saying that the example relation or database should be an Armstrong relation of database[4]: it should satisfy only the integrity constraints that are logically implied by the constraints explicitly given by the designer.

The fact that a functional dependency is not required to hold is very easy to spot from an example relation that satisfies the Armstrong property: one only needs to compare two rows. Noticing that a constraint does hold is more difficult, since one has to inspect the whole table.

Examples can also make redundancy in a schema intuitively easy to comprehend. Suppose, for example, that in the schema.

Employees (Employee, Manager, Department)

the functional dependencies Department $\rightarrow$ Manager, Manager $\rightarrow$ Department, and Employee $\rightarrow$ Department hold. This is represented by the following Armstrong relation.

Employee	Manager	Department
Wilson	Hogger	Construction
Jones	Hogger	Construction
Wood	Drake	Sales

The first two lines of the example show how the information about the manager of the Construction Department is represented twice. This can make it easier for the designer to understand why a design algorithm can suggest decomposing the schema into two smaller schemas, namely, (Employee, Department) and (Department, Manager).

Although some things are easy to spot from examples, the explicit representation of the integrity constraints also has its advantages. Therefore the designer can be shown both the list of constraints and the example table as representations of the constraint set. This double representation gives the designer two complementing views of the design.

IV. DESIGN-BY-EXAMPLE

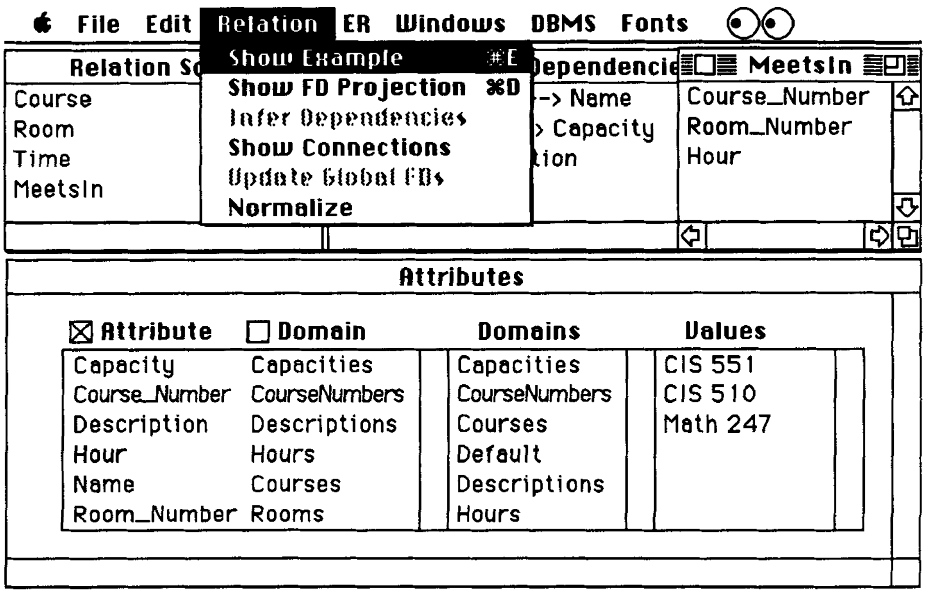
Generating and maintaining the example relations by hand is tedious, especially because the database schema tends to change frequently during the design. During the past few years we have designed and implemented a database design tool called Design-By-Example (DBE). This tool supports the use of automatically generated and maintained example databases. The examples are Armstrong databases for the set of constraints given in the design.

In DBE the design can start from the construction of an ER diagram using a typical drawing tool. The system then produces the corresponding relation schemas and integrity constraints. To check the design, the designer can generate an example database and inspect it. If the example is incorrect (or just unnatural), the designer can edit it. After this, a new set of integrity constraints can be inferred from the example. These constraints are shown to the designer, who can also directly edit the set of constraints. The changes in the set of constraints can imply that the relation schemas should be changed. The system does this interactively with the designer. The changes in the relation schemas can mean changes also in the ER diagram; these can also be done by the system, if the user so wishes. At any point of the design, the user can inspect an example database and an ER diagram corresponding to the current set of schemas and constraints.

Example 2. Consider the ER schema shown in Figure 1. It has been drawn using the ER editor of DBE. The corresponding relational schema is produced by choosing a command from a menu. In this case the four schemas shown previously would be produced. The schema for the meeting times of courses is shown attribute-by-attribute in Figure 2, which also illustrates how the example relation for the MeetsIn schema can be generated by a single command.

As a result, the example relation shown in Figure 3 would be displayed. The values appearing in the relation must have been described by the designer as possible example values in the appropriate domains of attributes.

The example can be edited by changing the time on the second row to something else, and the set of dependencies that the modified relation satisfies can be inferred using the Infer Dependencies command. The outcome is shown in Figure 4.

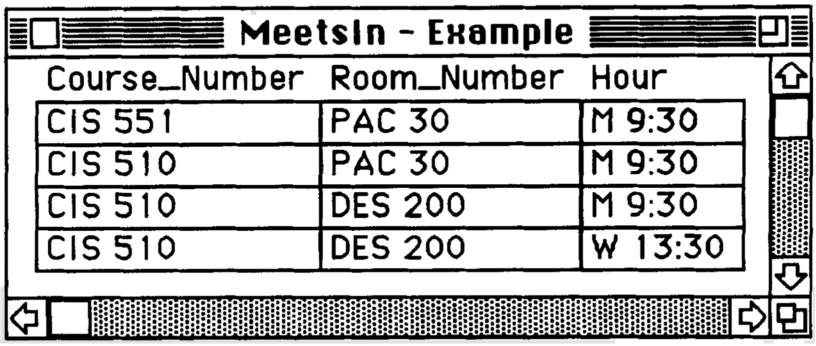


☒ Attribute	☐ Domain	Domains	Values
Capacity	Capacities	Capacities	CIS 551
Course_Number	CourseNumbers	CourseNumbers	CIS 510
Description	Descriptions	Courses	Math 247
Hour	Hours	Default	
Name	Courses	Descriptions	
Room_Number	Rooms	Hours	

Figure 2. The relational level of DBE.

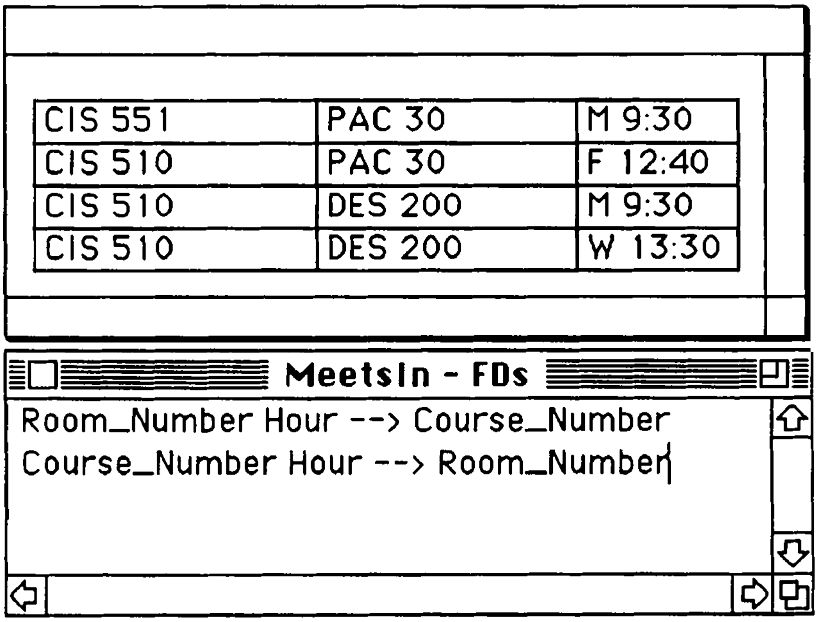
All dependencies found in this manner have a key of the relation schema on their left-hand side, which means that the schema is in Boyce–Codd normal form. Thus the relation schema is acceptable as such. However, when the modified schema (including the new dependencies) is mapped back into an ER diagram, the dependencies are reflected in the functionality of the MeetsIn relationship type: it becomes functional with respect to both Course and Room.

Thus DBE supports three different representations of the relation schemas and constraints: the relation schemas and constraints themselves, an ER dia-



Course_Number	Room_Number	Hour
CIS 551	PAC 30	M 9:30
CIS 510	PAC 30	M 9:30
CIS 510	DES 200	M 9:30
CIS 510	DES 200	W 13:30

Figure 3. An example relation.



CIS 551	PAC 30	M 9:30
CIS 510	PAC 30	F 12:40
CIS 510	DES 200	M 9:30
CIS 510	DES 200	W 13:30

```
Room_Number Hour --> Course_Number
Course_Number Hour --> Room_Number
```

Figure 4. Inferred dependencies

gram, and an example database that is an Armstrong database for the set of constraints. Actually, by using functional and inclusion dependencies one can express properties of schemas that are not expressible in an ER diagram. Thus the levels do not have exactly the same power.

In addition to the use of examples, the novel features of DBE include a design methodology based on the simultaneous use of the ER model and the relational model, the use of functional dependencies and inclusion dependencies* as structural integrity constraints, and especially efficient design algorithms.

The implementation of DBE has required that the mappings between all these three levels are implemented. The mapping from ER schemas to relational schemas is a classical topic in database design literature. The main idea in the mapping is very simple, but one has to be careful with the details (see Refs. 5 and 6). The mapping from relation schemas and integrity constraints back to ER diagrams is less common, but it is algorithmically fairly easy.

The task of producing example relations for a set of schemas and integrity constraints is quite complicated. The example should be small, since a large example is probably useless for human interface purposes. The algorithms used in DBE are described in Refs. 7 and 8.

*Perhaps surprisingly, inclusion dependencies have not been widely used in database design tools, although they are fairly intuitive.

The specific topic of this article is the mapping from an (edited) example database to the set of schemas and constraints. The schemas are, of course, easy to obtain from the example database, but the constraints are much harder to find. Also in this problem questions of the size of the result are important: for a given relation there are typically many equivalent sets of constraints that hold in the relation. The solution to the dependency inference problem should produce a small but still intuitive representation of the constraints.

DBE can also be used to check and possibly modify the design of an existing database. In this application, the schemas and relations of a relational database are analyzed by DBE. The analysis gives the set of functional and inclusion dependencies holding in the database instance. Using this information, an ER schema describing the database can be formed almost automatically.[9,10]

Since the roles of the three levels (ER schema, relation schemas and constraints, and example databases) are symmetrical, the basic architecture of DBE is directly applicable to the analysis of existing databases. The two types of use have, however, important differences from an algorithmic point of view. In the analysis of example relations generated by the design tool and subsequently modified by the designer the relations are typically fairly small: they have at most 10 to 20 rows, and usually less than 10 rows. In analyzing an existing database, the relations can, however, be of any size. Thus the two types of applications pose different requirements for a dependency inference algorithm. The analysis of small relations should be very interactive and fast, and also preferably incremental, as it will be done several times with slightly different inputs. On the other hand, the analysis program of existing databases can be batch-oriented, as it will typically be run only once for a given database.

The DBE system has been implemented on the Macintosh II family of computers and also in the OS/2 Presentation Manager environment. The system is described in Ref. 11.

The design and use of DBE have shown how important the environment of knowledge discovery is. While the possibility of generating and analyzing example databases is extremely useful, it needs to be very tightly integrated with other features that help in producing a well-designed schema.

V. THE FUNCTIONAL DEPENDENCY INFERENCE PROBLEM AND LOWER BOUNDS

Let r be a relation over a relation schema R. If F is a set of functional dependencies, then $r \models F$ means that all dependencies of F hold in r. The set of all functional dependencies holding in r is denoted by dep(r), that is,

$$X \rightarrow Y \in \text{dep}(r) \text{ if and only if } r \models X \rightarrow Y.$$

The dependency $X \rightarrow Y$ is a consequence of F, denoted $F \models X \rightarrow Y$, if $r \models F$ implies $r \models X \rightarrow Y$ for all relations r.

If F and G are equivalent dependency sets, that is, all the dependencies of G are consequences of F and vice versa, we say that F is a cover of G (and G is a cover of F).

The dependency inference problem is the problem of finding for a given relation r a cover for the set dep(r). The dep(r) set always has several equivalent covers of varying size, and we are interested in finding a small cover.

Example 3. Consider the following relation.

Employee	Department	Manager	Salary
Smith	Toys	Jones	200
Wilson	Administration	Brown	300
Barnes	Toys	Jones	300

In this relation, the following functional dependencies (and their consequences) hold:

Employee → Department Manager Salary
Department → Manager
Manager → Department
Department Salary → Employee

While the first three dependencies can be true for all instances of this schema, the fourth is probably only accidentally true in this small example.

In this section we consider the computational complexity of dependency inference. How feasible is the idea of inferring dependencies from existing databases? How much time can we expect such an algorithm to take? We merely give the results; proofs can be found in Refs. 12 and 13.

The problem instances have two parameters: the number of attributes n and the number of rows p. If dependency inference is used to analyze modified example relations generated by a design tool, the quantity p is probably small. In the analysis of existing databases p can be quite large. We analyze the complexity of dependency inference with respect to both these parameters.

A naive approach to computing a cover of *dep*(r) is to consider all possible dependencies that could exist among the attributes of r.

THEOREM 1. (Ref. 12) *The naive algorithm for dependency inference works in time $O(n^2 2^n p \log p)$, where n is the number of attributes in R and p is the number of tuples in r.*

It is easy to show that $O(p \log p)$ is also a lower bound in a comparison-based model of computation.

THEOREM 2. (Ref. 12) *The dependency inference problem requires in the worst case $\Omega(p \log p)$ steps for two-attribute relations with p rows.*

Thus the simple algorithm that blindly tests all dependencies is optimal with respect to the number of rows in the relation. Of course, the constant in the time bound above is fairly large. But it turns out that this cannot be avoided in the worst case.

THEOREM 3. (Ref. 14) *For each n there exists a relation r over R such that* $|R| = n$, $|r| = O(n)$, *and each cover of dep(r) has* $\Omega(2^{n/2})$ *dependencies.*

Similar results have been obtained by Demetrovics and Thi.[15,16]

Hence for some small relations the size of the output of dependency inference is exponential in the size of the input. Thus any algorithm for dependency inference must use exponential time, if it must output the result.

In practice, relations whose dependency sets have only large covers should be rare. The fact that dep(r) has only large covers implies that either r has many different keys, or the relation schema is highly unnormalized (since many nonkey dependencies hold). Both situations are unlikely.

Since in the worst case the size of any cover can be exponential, a natural goal is to look for an algorithm that would work fast for relations with small sets of dependencies. One can try to find an algorithm that works in time polynomial in the size of the smallest possible cover of the relation.

This problem turns out to be equivalent to the problem of computing the transversals[17] of a hypergraph in polynomial time with respect to their size and number.[13] For a rich study of this and related problems, see Ref. 18.

VI. ALGORITHMS FOR FUNCTIONAL DEPENDENCY INFERENCE

We have seen that the dependency inference problem is intractable in the worst case, but that the real world instances are likely to be simple. Clearly, we need a better algorithm than the naive one if and when we wish to include this feature in a database design tool.

Space does not permit a detailed description of the inference algorithms; they have been described in a series of articles.[13,14,19] Empirical results about the efficiency of the algorithms can be found in Ref. 20.

The basic idea in all dependency inference algorithms is to use information about the agreements and disagreements among the rows of the relation. The algorithm presented in Ref. 14 computes first for each attribute $A \in R$ the collections

$$J_A = \{\text{disag}(t,t') | t,t' \in r \text{ such that } t[A] = t'[A]\},$$

where

$$\text{disag}(t,t') = \{B \in R | t[B] \neq t'[B]\}.$$

If $X \in J_A$, then for any left-hand side Y of a dependency $Y \rightarrow A$ holding in r we must have $Y \cap X \neq \emptyset$. The sets J_A can be pruned by keeping only the minimal sets:

$$K_A = \{W \in J_A | \text{there is no } V \in J_A \text{ such that } V \subset W\}.$$

Computing these sets takes $O(p^2)$ operations, if the relation has p rows. After this, the left-hand sides of the dependencies $Y \rightarrow A$ can be computed by forming the transversals[17] of the sets in K_A. A transversal of a hypergraph or a collection of sets is a set that intersects every edge in the hypergraph. Computing the transversals of a hypergraph is a well-known problem.[17,18] No algorithm is known for this problem that would work in time polynomial with respect to the size of the output. The obvious algorithms can produce the same transversal several times, and this makes the time requirement too large.

For large relations (say, $p = 100\,000$) the bottleneck in dependency inference is not the exponentiality in the number of attributes, but in that even a $\theta(p^2)$ algorithm is unusable.

One can try to get a $O(p \log p)$ algorithm with a reasonable constant by using repeated sorts of the relation, as follows. Given a relation schema R, a set F of functional dependencies, and an attribute $A \in R$, denote by lhs(A) (left-hand sides of A) the set of minimal nontrivial attribute sets $X \subseteq R$ such that $X \rightarrow A$ follows from F. That is,

$$\text{lhs}(r,A) = \{X \subseteq R \backslash \{A\} | F \models X \rightarrow A \wedge \forall Y \subset X : F \not\models Y \rightarrow A\}.$$

The families lhs(r,A) contain all the information about F, although F can be a much more succinct representation in some situations.

Algorithm 1.[21] Computation of lhs(r,A) by consecutive sorts.
Input. A relation r over schema R, and an attribute $A \in R$.
Output. The collection lhs(r,A).
Method. Maintain three collections of sets

```
  lhs: the left-hand sides already found;
  nonlhs: sets $X$ such that $X \rightarrow A$ does *not* hold in $r$;
  cand: sets still to be tested;
1.  lhs := $\emptyset$;
2.  nonlhs := $\emptyset$;
3.  cand := $\{\emptyset\}$;
4.  **while** cand $\neq \emptyset$ **do**
5.      let $W$ be the first set of cand;
6.      remove $W$ from cand;
7.      **if** for some $L \in$ lhs : $L \subseteq W$ **then**
8.          do nothing
9.      **else if** for some $N \in$ nonlhs : $W \subseteq N$ **then**
10.         cand := cand $\cup$ { $WD | D \in R \backslash (N \cup \{A\})$};
11.     **else**
12.         $Y := R \backslash WA$;
13.         sort $r$ using attributes $WYA$ as the sort key;
14.         let $Y'$ be the longest prefix of $WYA$
            such that $Y' \rightarrow A$ does not hold;
15.         **if** $Y'$ is not included in any set of nonlhs **then**
16.             remove subsets of $Y'$ from nonlhs;
17.             nonlhs := nonlhs $\cup$ $\{Y'\}$;
18.         **fi**;
19.         **if** $Y' \neq WY$ **then**
20.             let $B$ be the attribute in $WY$ following $Y'$;
21.             **if** $Y'B$ does not include any set of lhs **then**
22.                 remove supersets of $Y'B$ from lhs;
```

```
23.                 lhs := lhs ∪ {Y'B};
24.             fi;
25.         fi;
26.         cand := cand ∪ {WD | D ∈ R\WA};
27.     fi;
28. od;
```

We omit the correctness proof of the algorithm (see Ref. 21). The algorithm operates in time $O(mn\ p \log p + n2^{2n})$, where m is the number of sorts done. Thus the exponentiality in the number of attributes is present, but the time bound with respect to the number of rows is of the desired optimal form.

Another possibility of dealing with large relations is using samples of the relations, or by aiming only at an approximate cover for the set of dependencies; then, for example, Valiant style learning can be applied. Some algorithms and their preliminary analysis is presented in Ref. 22.

Results of practical experiments on some dependency inference algorithms are reported in Ref. 20.

VII. INFERENCE OF INCLUSION DEPENDENCIES

We now turn to the case of inclusion dependencies. Given a database schema **R** and a database **r** over **R**, we want to find the inclusion dependencies that hold in **r**.

Suppose we have a database schema consisting of two n-attribute relation schemas R and S. Then there are more than $n!$ possible nonequivalent inclusion dependencies of the form $R[X] \subseteq S[Y]$ between R and S. Checking them one by one is obviously impossible for any large value of n.

We can show that it is already NP-complete to decide whether an inclusion dependency of the form $R[X] \subseteq S[Y]$ holds, where Y contains all the attributes of S. Consider the following problem.

FULLINDEXISTENCE: Let R and S be relation schemas, and denote by X the sequence consisting of the attributes of R in some order. Given relations r and s over R and S, respectively, decide whether there exists a sequence Y consisting of disjoint attributes of S in some order such that the dependency $R[X] \subseteq S[Y]$ holds in the database (r,s).

THEOREM 4. *FULLINDEXISTENCE is an NP-complete problem.*

Proof. The problem is in NP, since we can nondeterministically guess a sequence Y and then in polynomial time verify that $R[X] \subseteq S[Y]$ holds.

To prove NP-hardness, we show that the following NP-complete problem is reducible to FULLINDEXISTENCE.

SUBGRAPH ISOMORPHISM: Given graphs $G = (V,E)$ and $H = (V',E')$, decide whether H contains a subgraph isomorphic to G, that is, whether there is an injective mapping $g : V \rightarrow V'$ such that for all $u,v \in V$ with $(u,v) \in E$ we have $(g(u),g(v)) \in E'$.

We reduce this problem to the FULLINDEXISTENCE problem as follows. The relation r corresponding to the graph $G = (V,E)$ has as schema R the set V, and it contains a row t_e for each $e = (u,v) \in E$. The values of t_e are defined by

$$t_e(u) = t_e(v) = 1,$$

$$t_e(w) = 0 \text{ for } w \in E \setminus \{u,v\}.$$

The relation s corresponding to graph H is formed in the same way: the schema S of s consists of the elements of V', and s has one row for each edge in E'. Denote by X the sequence of the elements of R in some order.

We claim that there is an injective mapping g from V to V' preserving the incidence relation if and only if

$$(r,s) \models R[X] \subseteq S[Y]$$

for some sequence Y of disjoint attributes of S. We omit the details.

Thus, in general, it is not always possible to quickly check the existence of a long inclusion dependency. However, one should be cautious in interpreting a result like Theorem 4. The database used to prove NP-completeness is highly artificial. In real situations there are several ways to prune the set of possible inclusion dependencies. If an inclusion dependency $R[X] \subseteq S[Y]$ holds, then for all attributes A of X the corresponding attribute B of Y should be of the same type as A. That is, if A is a string-valued attribute, B should also be string-valued, etc. Second, looking for inclusion dependencies of arbitrary length is typically not useful: the dependencies holding in a database are probably short. Furthermore, a necessary condition for an inclusion dependency $R[X] \subseteq S[Y]$ is that $R[\mathrm{A}] \subseteq S[B]$ holds for each pair A and B of attributes of X and Y corresponding to each other. Hence one can start by looking at the unary inclusion dependencies holding in the database.

In a database schema of n attributes there are at most n^2 possible unary inclusion dependencies, so the collection

$$\{R[\mathrm{A}] \subseteq S[B] | R,S \in \mathbf{R},\ \mathrm{A} \in R,\ B \in S, \text{ and } r[\mathrm{A}] \subseteq s[B]\}$$

can be computed in time $O(n^2 p \log p)$, where p is the maximum number of rows in any one relation of the database $\mathbf{r}$. As mentioned earlier, the coefficient n^2 can be further lowered. There is no need to check inclusion between attributes with different domains. Hence instead of n^2 pairs of attributes one needs to investigate only $\Sigma_i n_i^2$ pairs, where i ranges over the different types of attributes (strings, numeric, etc.) and n_i is the number of attributes of type i. Given that for most databases only relatively few inclusion dependencies hold, the checks for most dependencies should terminate rather quickly.

VIII. RELATED WORK

Dependency inference also has applications in query optimization. Traditionally queries are optimized with respect to a given set of constraints that any instance of the database must satisfy. However, the particular instance existing at the time of query evaluation may well satisfy additional dependencies that could be used to speed up query evaluation. One algorithm specifically designed for instance-based query optimization is presented in Ref. 23. If the dependency inference problem can be solved efficiently, its time usage can be subsumed by the savings achieved in query evaluation. Siegel[24] considers automatically finding rules that can be used to optimize queries.

In artificial intelligence, determinations (see Ref. 25) are expressions closely corresponding to functional dependencies. Approaches to finding determinations can be found in Refs. 25 and 26. Another application of dependency inference in artificial intelligence can be found in Ref. 27.

Delgrande[28] studies the problem of finding supporting evidence for the validity or invalidity of a given integrity constraint. His work differs from ours in several aspects. First, his integrity rules can be very general: they are arbitrary expressions in a (slightly weakened) relational algebra. We concentrate on dependencies, since they are the most important integrity rules for (re)designing the database schema. Delgrande also considers only finding evidence for the dependencies proposed by the user or designer. No attempt is made to automatically generate the set of all valid dependencies. (Considering the general form of constraints, such an attempt would not even be realistic.) Similarly, Borgida, Mitchell, and Williamson[29,30] suggest a method for automatically maintaining a set of exceptions to the integrity constraints. When sufficiently many exceptions are found, possible corrections to the constraints (based on some heuristics) are proposed to the designer. Again, no rules are automatically inferred.

The use of inference of inclusion dependencies in mapping relational schemas to conceptual schemas is considered in Ref. 31. Papers dealing with inference of more complicated constraints are Refs. 28, 32, 33, and 34 and the collection in Ref. 35.

The problem of inferring the functional dependencies that hold in a relation is closely related to the problem of learning a conjunction of propositional Horn clauses.[36] Namely, a functional dependency $ABC \rightarrow D$ can be considered to be a Horn clause $\neg A \wedge \neg B \wedge \neg C \wedge D$, and hence a set of functional dependencies corresponds to a set of propositional Horn clauses.

IX. DISCUSSION

We have considered the problem of inferring the functional and inclusion dependencies that hold in a given database instance. The motivation for the problem arises from a database design tool that uses example databases to illustrate the design decisions.

The dependency inference problem, as formulated above, always has an exact solution. This gives the problem a different flavor from many other knowledge acquisition or machine learning applications. However, for the analysis of existing databases, an approximate solution for the dependency inference problem would be preferable: one would like to see what dependencies almost hold.

Design-By-Example is based on the idea of using examples as a tool in human–computer interaction. We believe that examples are a good way of providing feedback for the user, and our experience with DBE seems to validate this. The domain where examples are applied is small and theoretically well-behaved; this probably contributes greatly to the usefulness of examples in it.

References

1. C.J. Date, "Referential integrity," In *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB'81)*, IEEE, 1981, pp. 2–12.
2. M.A. Casanova, R. Fagin, and C. Papadimitriou, "Inclusion dependencies and their interaction with functional dependencies," *Journal of Computer and System Sciences*, **28**, 29–59 (1984).
3. C.C. Fleming and B. von Halle, *Handbook of Relational Database Design*, Addison-Wesley, Reading, MA, 1989.
4. R. Fagin, *Armstrong Databases*, Research Report RJ3440, IBM, San Jose, CA, May 1982.
5. V.M. Markowitz and A. Shoshani, "On the correctness of representing extended entity–relationship structures in the relational model," In *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'89)*, ACM, 1989, pp. 430–439.
6. V.M. Markowitz and A. Shoshani, "Name assignment techniques for relational schemas representing extended entity–relationship schemas. In *Proceedings of the 8th International Conference on Entity–Relationship Approach*, Frederick H. Lochovsky, (Ed.), October 1989, pp. 21–39.
7. H. Mannila and K.-J. Räihä, "Design by example: An application of Armstrong relations," *Journal of Computer and System Sciences*, **33**(2), 126–141 (1986).
8. H. Mannila and K.-J. Räihä, "Practical algorithms for finding prime attributes and testing normal forms," In *Proceedings of the 8th ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS'89)*, ACM, 1989, pp. 128–133.
9. M.A. Casanova and J.E. Amaral de Sa, "Mapping uninterpreted schemes into entity–relationship diagrams: Two applications to conceptual schema design," *IBM Journal of Research and Development*, **28**(1), 82–94 (January 1984).
10. H. Mannila and K.-J. Räihä, "A mapping from relational database schemas to ER-diagrams using inclusion dependencies," April 1990.
11. M. Kantola, H. Mannila, K.-J. Räihä, H. Siirtola, and J. Tuomi," Design-By-Example: A tool for database design; user guide for version 3.0," November 1991.
12. H. Mannila and K.-J. Räihä, "On the complexity of inferring functional dependencies," *Discrete Applied Mathematics*, (1991).
13. H. Mannila and K.-J. Räihä, *The Design of Relational Databases*, Addison-Wesley, 1992, to be published.
14. H. Mannila and K.-J. Räihä, "Dependency inference," In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB'87)*, September 1987, pp. 155–158.
15. J. Demetrovics and V.D. Thi, "Some results about functional dependencies," *Acta Cybernetica*, **8**(3), 273–278 (1988).
16. J. Demetrovics and V.D. Thi, "Relations and minimal keys." *Acta Cybernetica*, **8**(3), 279–285 (1988).
17. C. Berge, *Hypergraphs. Combinatorics of Finite Sets*. North-Holland, Amsterdam, 1989.
18. T. Eiter and G. Gottlob, *Identifying the Minimal Transversals of a Hypergraph and Related Problems*, Technical Report CD–TR 91/16, Technische Universität Wien, January 1991.
19. H. Mannila and K.-J. Räihä, "Algorithms for dependency inference," Report

A–1988–3, University of Tampere, Department of Computer Science, Tampere, Finland, February 1988.

20. D. Bitton, J.C. Millman, and S. Torgersen, "A feasibility and performance study of dependency inference," In *Proceedings of the 5th International Conference on Data Engineering,* 1989.
21. H. Mannila and K.-J. Räihä, "Algorithms for inferring functional dependencies," Report C–1991–41, University of Helsinki, Department of Computer Science, Helsinki, Finland, August 1991.
22. J. Kivinen and H. Mannila, "Approximate dependency inference," manuscript. 1991.
23. R. Dechter, "Decomposing an n-ary relation into a tree of binary relations," In *Proceedings of the 6th ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS'87),* ACM, March 1987, pp. 185–189.
24. M. Siegel, *Automatic Rule Derivation for Semantic Query Optimization,* Technical Report BUCS Tech Report #86–013, Boston University, Computer Science Department, December 1986.
25. S. J. Russell, *The Use of Knowledge in Analogy and Induction,* Morgan Kaufmann, San Mateo, CA, 1989.
26. J.C. Schlimmer, "Learning determinations and checking databases," In *Proceedings of 1991 AAAI Workshop on Knowledge Discovery in Databases,* G. Piatetsky-Shapiro (Ed.), 1991, pp. 64–76.
27. H. Almuallim and T.G. Dietterich, "Learning with many irrelevant features," In *AAAI-91, Proceedings, 9th National Conference on Artificial Intelligence,* AAAI Press/The MIT Press, Cambridge, 1991, pp. 547–552.
28. J.P. Delgrande, "Formal bounds on the automatic generation and maintenance of integrity constraints," In *Proceedings of the 6th ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS'87),* ACM, March 1987, pp. 190–196.
29. A. Borgida and K.E. Williamson, "Accommodating exceptions in databases, and refining the schema by learning from them," In *Proceedings of the 11th International Conference on Very Large Data Bases (VLDB'85),* August 1985, pp. 72–82.
30. A. Borgida, T. Mitchell, and K.E. Williamson, "Learning improved integrity constraints and schemas from exceptions in data and knowledge bases," In *On Knowledge Base Management Systems,* M.L. Brodie and J. Mylopoulos (Eds.), Springer-Verlag, 1986, pp. 259–286.
31. M. Castellanos and F. Saltor, "Semantic enrichment of database schemas: An object-oriented approach," manuscript, 1991.
32. V.P. Tseng and M.V. Mannino, "Inferring database requirements from examples in forms, In *Proceedings of the 7th International Conference on Entity–Relationship Approach,* C. Batini (Ed.), November 1988, pp.255–265.
33. G. Piatetsky-Shapiro, "Discovery and analysis of strong rules in databases," In *Advanced Database Systems Symposium,* Kyoto, Japan, 1989, pp. 135–142.
34. R. Yasdi, "Learning classification rules from database in the context of knowledge acquisition and representation," *IEEE Transactions on Knowledge and Data Engineering,* 3(3), 293–306 (September 1991).
35. G. Piatetsky-Shapiro (Ed.), *Proceedings of 1991 AAAI Workshop on Knowledge Discovery in Databases,* American Association for Artificial Intelligence, July 1991.
36. D. Angluin, M. Frazier, and L. Pitt, "Learning conjunctions of Horn clauses," In *Proceedings, 31st Annual Symposium on Foundations of Computer Science,* IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 186–191.