Discovery of Data Dependencies in Relational Databases

Siegfried Bell & Peter Brockhausen
Informatik VIII, University Dortmund
44221 Dortmund, Germany
email: $\{bell, brockh\}$@ls8.informatik.uni-dortmund.de

Abstract

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 standard set–oriented interfaces of relational database management systems.

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 domain restrictions, unary inclusion- and functional dependencies in relational databases. The system also embodies an inference relation to minimize database access.

1 Introduction

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 discovery 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 this approach.

In general, knowledge discovery in databases 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.

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 domain restrictions, unary inclusion and functional dependencies. The plot of this paper is as follows: In section 2 domain restrictions of attributes, functional and unary inclusion independencies are introduced and the corresponding inference relations are discussed. In section 3 we show how to test dependencies by SQL queries and how to deduce more dependencies. Then, we discuss the complexity of our inference. We conclude with empirical results and a comparison with similar systems.

2 Terminology and Related Work

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. Functional dependencies (FD) and unary inclusion dependencies (UIND) are defined as usual. Further, we assume without loss of generality that the right hand side of functional dependencies consists of only one attribute.

The relationship between unary inclusion dependencies and functional dependencies is discussed in [Bell, 1995] and an axiomatization is given regarding independencies, which simplifies the inference. Our third type of constraints, domain restrictions, means that each value of an attribute has to be in an certain interval.

The discovery of data dependencies may be visualized as a search in a semi lattice 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 like in [Savnik and Flach, 1993]: 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 $Y \rightarrow A$.

This means, if a relation satisfies a functional dependency, then the relation satisfies also each dependency, which is more–specific–than. For example, if a relation satisfies the functional dependency $AB \rightarrow C$, then the relation satisfies $ABD \rightarrow C$. This relationship implies a partial ordering which simplifies the discovery of functional dependencies by a simple representation. This is usually called minimal cover. We use not 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 1 (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 $Y \rightarrow A \in F$.*

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. The verification is done by the database management system which groups the rows. This offers several advantages: First theorem proving is for this purpose too powerful and we can infer dependencies by transitivity which is really simple. 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, [Morik et al., 1993], functional dependencies can not be expressed. Systems, which are closer to ours, are empirically compared in section 4.

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].

3.1 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.

1. SELECT COUNT(DISTINCT $R_i.A_1$)
FROM R_i, R_j
WHERE $R_i.A_1 = R_j.A_2$ $\quad$ $=: e$
2. SELECT COUNT(DISTINCT A_1)
FROM R_i $\quad$ $=: e_1$
3. SELECT COUNT(DISTINCT A_2)
FROM R_j $\quad$ $=: e_2$
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 1: SQL–Statements and Conditions for Calculating UINDs

3.2 Unary Inclusion Dependencies

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 1.

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 $A \subset B$ and $B \subset A$ 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 2 is called one time for each kind of a "super data type" in the database. 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 algorithm. Therefore it makes sense to gather all different numeric and alphanumeric types in "super types" NUMBER and STRING.

The algorithm uses a graph representation for UINDs. There exists an 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 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.

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 guarantees 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.

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

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 integrity constraints – 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 $\overline{\text{the}}$ UINDs $A_i \subseteq A_j$ $\underline{\text{or}}$ $A_j \subseteq A_i$ 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 $\overline{\text{the}}$ 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 2: 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 3: Procedure UPDATE GRAPH

```
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 4: A SQL–statement for the Computation of Functional Dependencies

```
Algorithm: FUNCTIONAL DEPENDENCIES
Input: All attributes $A_1, \ldots, A_n$ of the relation
Output: All discovered most–general FDs
vspace*-0.3cm

 1. Compute the class of attributes NKNN.

 2. For each attribute $A_i$ compute a list of all possible attributes
    for the left–hand side of most–general FDs $X \rightarrow A_i$.

 3. For each attribute $A_i$ do:
        IF bottom–up–search THEN top–down–search
```

Figure 5: The Algorithm FUNCTIONAL DEPENDENCIES

there exist "good" and "bad" numberings of the attributes in step 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 then 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 possible to drop the time complexity of summand $\mathcal{O}(n^4)$, which seems possible, because it would not save one more database query. We did some empirical experiments wrt. these questions, but in lack of space we omit the results in section 4.

3.3 Functional Dependencies

We start this subsection with a presentation of the necessary SQL–statement in order to compute functional dependencies. Figure 4 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.

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. Figure 5 shows an outline of the algorithm.

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 for the first step is $\mathcal{O}(n * m)$.

The second step 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.

All the attributes for the left–hand side of most–general FDs of the type $X \rightarrow F$ build a semi–lattice. Figure 6 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.

Our algorithm uses a top–down and left–to–the–right and breadth–first search strategy. The queue is initialized with all nodes in left–to–right order, which represent most–general hypotheses. 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 children are put in left to right order at the end of the queue.

But we need some more procedures, which can be found in [Brockhausen, 1994]. 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 primary keys, and after the detection of new FDs by database queries or by inference, the graph is updated. 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 $A \not\rightarrow B$.

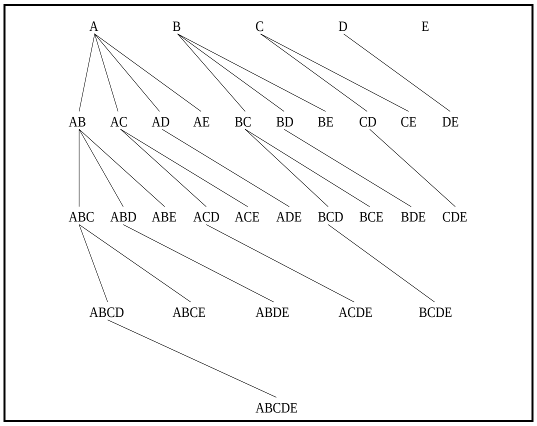

Figure 6: Reduction of a semi–lattice into a tree structure

Algorithm	DB	$\|r\|$	$\|R\|$	$\|X\|$	Time
Savnik/Flach	L.	150	19	7	9 m.
Schlimmer	B.C.	699	11	4	1 h 14 m.
Bell/Brockh.	L.	150	19	7	$>$ 33 h
Bell/Brockh.	B.C.	699	11	11	8 m. 53 s.
Bell/Brockh.	B.C.	699	11	4	4 m. 19 s.

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

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.

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. L. stands for the Lymphography domain and B.C. for the Breast Cancer domain. 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.

[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.

[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*. Academic Press

[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.