PART III

PRACTICE

In the third and last part of this thesis I illustrate the practice of confirmatory induction in the context of relational databases. The only chapter in this part, Knowledge discovery in databases, *also illustrates my claim that in confirmatory induction one needs an additional goal which the inductive hypothesis are meant to fulfil, since 'being confirmed' is not an end in itself.*

CHAPTER 8

KNOWLEDGE DISCOVERY IN DATABASES

— in which an illustration of confirmatory induction is given, and the ideas underlying the system INDEX*, a system for inductive data engineering, are discussed —*

THIS CHAPTER IS intended to provide a practical illustration of a number of theoretical concepts developed in the previous chapters. In particular, I will discuss how regularity-based and consistency-based confirmatory induction can be applied in order to extract implicit structural knowledge in relational databases. This knowledge is then made explicit by restructuring the database into a knowledge base. This approach, which I call *inductive data engineering*, has been implemented in a prototype system called INDEX. This application of confirmatory induction does not merely serve to illustrate that confirmatory induction can be useful: the main point is that inductive hypotheses are sought with a certain **goal** in mind, *viz*. to obtain a knowledge base that is better structured. This explicit goal enables us to assign a heuristic evaluation to inductive hypotheses, which results in a confirmatory discovery procedure rather than a proof procedure. My objective in this chapter is to provide sufficient detail for illustrating these points; however, the emphasis is on conceptual analysis, rather than on a detailed discussion of the applied methods and algorithms.

§30. INDUCTIVE DATA ENGINEERING

Let me start by giving a concrete example of inductive data engineering. Consider the train schedule depicted in Table 8.1. There's a wealth of implicit information in this schedule: for instance, all stopping trains leave from platform 4, while intercity trains leave from platform 5 or 6. Furthermore, intercity trains leave every thirty minutes, while stopping trains leave every thirty minutes before 9:00, and every hour thereafter. Notice that, since the significance of these regularities may extend to other schedules, they can be seen as inductive generalisations. Furthermore, they can be utilised for representing the same information in a more structured format. For instance, the second regularity can be used to compress the schedule considerably (Table 8.2).

Table 8.2 contains enough information to reconstruct Table 8.1, which can be demonstrated as follows. One way to formalise the statement about the regularity of intercity and stopping trains is by adding a second table, listing for each train the start of a new 'interval' (Table 8.3). The difference between intercity and stopping trains is indicated by the 9:30 entry for intercity trains, which is not included for stopping trains.

Time	*Direction*	*Type*	*Platform*
8:07	Utrecht/Amsterdam	intercity	5
8:09	Tilburg/Den Haag	intercity	6
8:12	Utrecht	stopping train	4
8:16	Tilburg	stopping train	4
8:37	Utrecht/Amsterdam	intercity	5
8:39	Tilburg/Den Haag	intercity	6
8:42	Utrecht	stopping train	4
8:46	Tilburg	stopping train	4
9:07	Utrecht/Amsterdam	intercity	5
9:09	Tilburg/Den Haag	intercity	6
9:37	Utrecht/Amsterdam	intercity	5
9:39	Tilburg/Den Haag	intercity	6
9:42	Utrecht	stopping train	4
9:46	Tilburg	stopping train	4

Table 8.1. A train schedule.

Minutes	*Direction*	*Type*	*Platform*
07	Utrecht/Amsterdam	intercity	5
09	Tilburg/Den Haag	intercity	6
42	Utrecht	stopping train	4
46	Tilburg	stopping train	4

Note: Intercity trains leave every 30 minutes.
Stopping trains leave every 30 minutes
before 9:00, and every hour after 9:00.

Table 8.2. A restructured train schedule.

Table 8.1 can be reconstructed by the following procedure: for every pair of entries in Table 8.2 and Table 8.3 with corresponding train types, construct an entry in Table 8.1 with a departure time that can be calculated by increasing the interval from Table 8.2 with the number of minutes from Table 8.3. In this way every intercity entry in Table 8.2 results in four entries in Table 8.1, and every stopping train entry results in three entries in Table 8.1. The reader can easily verify that the resulting table is indeed identical to the original schedule.

Type	*Interval*
intercity	8:00
intercity	8:30
intercity	9:00
intercity	9:30
stopping train	8:00
stopping train	8:30
stopping train	9:00

Table 8.3. A table expressing the regularity of trains.

This demonstrates that, in certain cases, induced regularities can be used to improve the organisation of structured information. This raises a few questions:

(*i*) what regularities can lead to restructuring?
(*ii*) how are these regularities to be induced?
(*iii*) which of the possible regularities should be selected for restructuring?

The first question is not easy to answer in general: there appear to be many possibly regularities, each with its own associated notion of restructuring. For instance, suppose that we know that a certain relation is transitive, then it can be considerably compressed — however, there are many ways to do that. In this chapter I will restrict attention mostly to two, relatively simple regularities known from relational database theory: functional and multivalued dependencies. These concepts will be explained in detail below; to fix the reader's thought, the regularity in Table 8.1 corresponds roughly to a multivalued dependency from train type to departure interval.

As for the second question, we first have to decide what kind of induction we are talking about. I will first show that such regularities cannot be inferred by explanatory induction. Taken separately, statements like 'intercity trains leave every 30 minutes' are very weak: they do not explain a single entry in Table 8.1. Only if this statement is combined with one of the entries in the schedule, say 'at 8:09 there's a train leaving for Tilburg from platform 6' is it possible to derive other entries like 'at 8:39, 9:09 and 9:39 there are also trains leaving for Tilburg from platform 6'. Another way to show that inference of regularities like the above is not explanatory is by noting that, even if evidence from different sources each endorse a regularity separately, their combination may refute it. For instance, if we split Table 8.1 in two, one half containing the intercity trains leaving at 8:07, 8:09, 9:07 and 9:09, the other half containing the remaining intercity trains (8:37, 8:39, 9:37, 9:39), then in each of these halves separately the regularity 'intercity trains leave (only) every hour' can be observed, while this regularity is obviously false when the two halves are joined. In other words, the property of Additivity is invalid for this form of induction, hence it is not explanatory (see Definition 6.13).

I will show in §32 that such regularities are inferred by confirmatory induction. Both the regularity-based and the consistency-based semantics can be used, but for multivalued dependencies the latter is more appropriate in an incremental context. Interestingly, both semantics coincide for functional dependencies. Before this can be demonstrated a number of concepts from database theory need to be discussed (§31). The third of the above questions will be discussed in §33, where I will consider a heuristic measure to assess the utility of induced regularities.

§31. ATTRIBUTE DEPENDENCIES IN DATABASES

I will start with a brief overview of the relational theory of databases, followed by a discussion how concepts from this relational theory can be translated to logical concepts. From the logical view it is a relatively small step to the next section, which discusses inference of attribute dependencies as confirmatory induction.

8. Knowledge discovery in databases

A bit of relational database theory

The notational conventions employed here are close to (Maier, 1983).

> DEFINITION 8.1. A *relation scheme* R is an indexed set of *attributes* $\{A_1, \ldots, A_n\}$. Each attribute A_i has a *domain* D_i, $1 \le i \le n$, which is a set of *values*. A *tuple over* R is a mapping t: $R \to \cup_i D_i$ with $t(A_i) \in D_i$, $1 \le i \le n$; usually, a tuple t is denoted as a sequence $\langle t(A_1), \ldots, t(A_n)\rangle$, i.e. with attributes referenced by their position rather than by their name. If $X = \{A_j, \ldots, A_k\}$ is a set of attributes and t is a tuple, the sequence $\langle t(A_j), \ldots, t(A_k)\rangle$ will be called the *X-value* of t[84]. A *relation over* R is a finite set of tuples over R.

In general, attributes are denoted by uppercase letters (possibly subscripted) from the beginning of the alphabet; sets of attributes are denoted by uppercase letters (possibly subscripted) from the end of the alphabet; values of (sets of) attributes are denoted by corresponding lowercase letters. Relations are denoted by lowercase letters (possibly subscripted) such as n, p, q, r, u; tuples are denoted by $t, t_1, t_2, \ldots$. If X and Y are sets of attributes, their juxtaposition XY means $X \cup Y$. We employ the usual notation for expressions of relational algebra: in particular, $\pi_X(r)$ denotes the *projection* of the relation r onto the set of attributes X (i.e. the set of X-values of tuples in r), and $\sigma_{X=x}(r)$ denotes the *selection* of those tuples in r whose X-value is x.

A functional dependency from attributes X to attributes Y expresses that the Y-value of any tuple from a relation satisfying the functional dependency is uniquely determined by its X-value. In other words, if two tuples in the relation have the same X-value, they also have the same Y-value.

> DEFINITION 8.2. Let R be a relation scheme, and let X and Y be subsets of attributes from R. A *functional dependency* (*fd* for short) is an expression of the form $X \to Y$. A relation r over R *satisfies* a functional dependency $X \to Y$ if $t_1 \in r$ and $t_2 \in r$ and $t_1(X) = t_2(X)$ imply $t_1(Y) = t_2(Y)$, and *violates* it otherwise.

The following lemma lists the main properties of functional dependencies.

> LEMMA 8.3. *(1) A relation satisfies a functional dependency $X \to Y$ iff it satisfies $X \to A$ for every $A \in Y$.*
> *(2) If a relation satisfies a functional dependency $X \to Y$, it also satisfies any functional dependency $Z \to Y$ with $Z \supseteq X$.*
> *(3) If a functional dependency is satisfied by a relation r, it is also satisfied by any relation $r' \subseteq r$.*
> *Proof.* Immediate from Definition 8.2.

[84] I will usually not distinguish between single attributes and singleton sets of attributes, nor between single values and singleton sequences of values — e.g., if A is a single attribute, I will say that $t(A)$ is the A-value of t, rather than '$\langle t(A)\rangle$ is the $\{A\}$-value of t'.

(1) allows us, without loss of generality, to restrict attention to functional dependencies with single attributes on the right-hand side. (2) demonstrates that some functional dependencies are stronger than others. Formally, we will say that a dependency D_1 is as *strong* as another dependency D_2, notation $D_1 \Rightarrow D_2$, if the set of relations that satisfy D_1 is a subset of the set of relations that satisfy D_2. Clearly, the relation $\Rightarrow$ is transitive and reflexive, hence a quasi-order — it will be demonstrated below (Lemma 8.7) that, if functional dependencies are read as logical statements, this relation is a special case of logical entailment. (3) follows from the fact that a functional dependency expresses a connection between each pair of tuples from a relation; it is highlighted because it plays an important role when fds are induced from tuples, as will be explained in §32.

Multivalued dependencies generalise functional dependencies by stipulating that every X-value determines a **set** of possible Y-values. For instance, if a relation describes events that occur weekly during a given period, this relation satisfies a multivalued dependency from day of week to date: given the day of week, we can determine the set of dates on which the event occurs. For instance, if the Computer Science course and the Artificial Intelligence course are both taught on a Wednesday during the fall semester, and there is a CS lecture on Wednesday September 7, while there is an AI lecture on Wednesday December 7, then there is also a CS lecture on the latter date and an AI lecture on September 7.

DEFINITION 8.4. Let R be a relation scheme, let X and Y be subsets of attributes from R, and let Z denote $R–XY$. A *multivalued dependency* (*mvd* for short) is an expression of the form $X \twoheadrightarrow Y$. A relation r over R *satisfies* a multivalued dependency $X \twoheadrightarrow Y$ if $t_1 \in r$ and $t_2 \in r$ and $t_1(X)=t_2(X)$ imply that there exists a tuple $t_3 \in r$ with $t_3(X)=t_1(X)$, $t_3(Y)=t_2(Y)$, and $t_3(Z)=t_1(Z)$. Note that by exchanging t_1 and t_2, there should also a tuple $t_4 \in r$ with $t_4(X)=t_2(X)$, $t_4(Y)=t_1(Y)$, and $t_4(Z)=t_2(Z)$. If $t_3 \notin r$ or $t_4 \notin r$, r *violates* the mvd.

The following lemma lists the main properties of multivalued dependencies.

LEMMA 8.5. *(1) A relation satisfies a multivalued dependency $X \twoheadrightarrow Y$ iff it satisfies $X \twoheadrightarrow Z$ with $Z=R–XY$.*
(2) If a relation satisfies a multivalued dependency $X \twoheadrightarrow Y$, it also satisfies any multivalued dependency $Z \twoheadrightarrow Y$ with $Z \supseteq X$.
(3) If a relation satisfies a functional dependency $X \rightarrow Y$, it also satisfies the multivalued dependency $X \twoheadrightarrow Y$.
Proof. Immediate from Definition 8.4.

(1) states that mvds are pairwise equivalent. Furthermore, it indicates that an mvd $X \twoheadrightarrow Y$ partitions a relation scheme R into three subsets $\{X,Y,R–XY\}$[85]. (2) demonstrates that, analogously to fds, multivalued dependencies can be ordered according to their strength; we will use the symbol $\Rightarrow$ in both cases. (3) states that functional dependencies are indeed special cases of multivalued dependencies. Notice that Lemma 8.3 (3) does not extend

[85]Such a partition is called a *dependency basis* for X (Maier, 1983).

multivalued dependencies, since an mvd requires the existence of certain tuples in the relation. Thus, if r satisfies a certain mvd, some subset of r may not satisfy it.

The following definition introduces some further notation that will prove useful later.

DEFINITION 8.6. Let R be a relation scheme, let r be a set of tuples over R, let $D = X \twoheadrightarrow Y$ be a multivalued dependency over R, and let $Z=R-XY$. The *D-closure of r* is defined as

$$r\uparrow D = \{t \mid t_1,t_2 \in r \text{ and } t(X)=t_1(X),\ t(Y)=t_2(Y),\ t(Z)=t_1(Z)\}$$

For instance, if $R = \{A,B,C\}$, $r = \{\langle a,b_1,c_1\rangle, \langle a,b_2,c_2\rangle\}$, and $D = A \twoheadrightarrow B$, then $r\uparrow D = \{\langle a,b_1,c_1\rangle, \langle a,b_2,c_2\rangle, \langle a,b_2,c_1\rangle, \langle a,b_1,c_2\rangle\}$. Note that in general $r\uparrow D \supseteq r$, and that r satisfies mvd D iff $r\uparrow D = r$.

Functional and multivalued dependencies are collectively referred to as *attribute dependencies*. The attributes found on the left-hand side of an attribute dependency are called *antecedent attributes*, those on the right-hand side *consequent attributes*. I will now demonstrate that there is a straightforward reformulation of attribute dependencies in terms of logic.

Reformulation in logical terms

For an overview of the subject of logic and databases, see (Gallaire *et al.*, 1984; Reiter, 1984). The basic idea is to associate with a relation r a predicate `r`, and to view the relation as a Herbrand interpretation for `r`. The statement $t \in r$, where $t = \langle a_1,\ldots,a_n\rangle$, is represented by a ground fact `r(`a_1`,` … `,`a_n`)`. This allows us to write a functional dependency $X \rightarrow A$ as a definite clause[86]

```
A1=A2:-r(X,A1,Z1),r(X,A2,Z2)
```

A relation r satisfies an fd D iff the corresponding set of ground facts, denoted by $\lceil r \rceil$, is a Herbrand model of the corresponding definite clause, denoted by $\lceil D \rceil$. Since $\lceil r \rceil$ is a set of ground facts, it can also be interpreted as a logic program; since it is definite, it has a unique truth-minimal model which of course coincides with $\lceil r \rceil$[87]. This link with closed-world reasoning is exploited in §32 below.

Similarly, the multivalued dependency $X \twoheadrightarrow Y$ corresponds to the definite clause

```
r(X,Y2,Z1):-r(X,Y1,Z1),r(X,Y2,Z2)
```

Notice that this clause also represents the mvd $X \twoheadrightarrow Z$ with $R-XY$. The relationship between the relational and logical views is summarised in the following lemma.

[86] `X`, `Z1`, `Z2` may actually denote sequences of variables and the number of the arguments of `r` may be different than shown here; this is however not significant.

[87] We will sometimes freely switch between the interpretation of $\lceil r \rceil$ as a Herbrand model, and its interpretation as a logic program. Wherever this might cause confusion the intended interpretation is indicated in words. The symbol $\models$ is avoided altogether in this context, since it indicates both satisfaction by a model and logical entailment by a logic program, which would cause unnecessary confusion.

LEMMA 8.7. *Let r be a relation over a relation scheme R, and let D and D′ be attribute dependencies over R.*
(1) r satisfies D iff the Herbrand interpretation $\lceil r \rceil$ *satisfies the clause* $\lceil D \rceil$.
(2) $D \Rightarrow D'$ *iff the clause* $\lceil D \rceil$ *logically entails the clause* $\lceil D' \rceil$.
Proof. Immediate from the construction of $\lceil r \rceil$ and $\lceil D \rceil$.

The logical view allows a closer analysis of the relation $\Rightarrow$.

LEMMA 8.8. *Let* D_1 *and* D_2 *be two attribute dependencies.* $D_1 \Rightarrow D_2$ *iff there exists a substitution* θ *unifying variables in the body of* $\lceil D_1 \rceil$ *with variables at the same position in the body of* $\lceil D_2 \rceil$, *such that* $\lceil D_1 \rceil\theta = \lceil D_2 \rceil$.
Proof. For the if part, the existence of such a substitution indicates that $\lceil D_1 \rceil$ θ-subsumes[88] $\lceil D_2 \rceil$, therefore $\lceil D_1 \rceil$ entails $\lceil D_2 \rceil$, and the result follows from Lemma 8.7 (2).
For the only-if part, if $D_1 \Rightarrow D_2$ then $\lceil D_1 \rceil$ entails $\lceil D_2 \rceil$ by Lemma 8.7 (2). In function-free clausal logic, logical entailment between non-tautological clauses is equivalent to θ-subsumption, so we obtain: $\lceil D_1 \rceil$ θ-subsumes $\lceil D_2 \rceil$, i.e. there is a substitution θ such that every literal in $\lceil D_1 \rceil\theta$ occurs in $\lceil D_2 \rceil$. But since the clauses $\lceil D_1 \rceil$ and $\lceil D_2 \rceil$ have the same number of literals, this boils down to: there is a substitution θ such that $\lceil D_1 \rceil\theta = \lceil D_2 \rceil$. Finally, since (*i*) the clauses representing attribute dependencies only contain variables and no constants, (*ii*) every variable occurring in the head of $\lceil D_1 \rceil$ or $\lceil D_2 \rceil$ also occurs in its body, and (*iii*) variables at different positions within a literal represent different attributes[89], we may conclude that every such θ unifies variables in the body of $\lceil D_1 \rceil$ with variables at the same position in the body of $\lceil D_2 \rceil$.

For instance, let $R = \{A,B,C,D\}$ and consider the functional dependencies $A \rightarrow D$ and $AB \rightarrow D$, represented by the clauses

```
D1=D2:-r(A,B1,C1,D1),r(A,B2,C2,D2)          % A->D
D1=D2:-r(A,B, C1,D1),r(A,B, C2,D2)          % AB->D
```

The first clause can be made equal to the second by applying the substitution {B1→B,B2→B}, so that by Lemma 8.8 we may conclude that $A \rightarrow D \Rightarrow AB \rightarrow D$, in agreement with Lemma 8.3 (2). As a second example, consider the multivalued dependencies $A \twoheadrightarrow D$ and $AC \twoheadrightarrow D$, represented by the clauses

```
r(A,B1,C1,D2):-r(A,B1,C1,D1),r(A,B2,C2,D2)   % A->->D
r(A,B1,C, D2):-r(A,B1,C, D1),r(A,B2,C, D2)   % AC->->D
```

The first clause can be made equal to the second by applying the substitution {C1→C,C2→C}, so that by Lemma 8.8 we may conclude that $A \twoheadrightarrow D \Rightarrow AC \twoheadrightarrow D$, in agreement with Lemma 8.5 (2).

[88] See §10 for a discussion of θ-subsumption.
[89] For practical purposes, the variables may be considered typed.

It may seem that this logical analysis offers little that wasn't known before. How consider the problem of deriving the weaker dependency from the stronger one (this called *specialisation* in chapter 3). From the above examples we may conclude that **only** way to do that is by unifying two distinct variables at the same relative po within each body literal, corresponding to extending the left-hand side of the depend This result is stated without proof below.

PROPOSITION 8.9. *(1) $X \rightarrow A \Rightarrow Z \rightarrow A$ iff $Z \supseteq X$.*
(2) $X \twoheadrightarrow Y \Rightarrow X' \twoheadrightarrow Y'$ iff $X' \twoheadrightarrow Y'$ is equivalent to $Z \twoheadrightarrow Y$ with $Z \supseteq X$.

(2) is stated this way because according to Lemma 8.5 (1) the mvds $X \twoheadrightarrow Y$ and $X \twoheadrightarrow R-XY$ are equivalent, hence $\lceil X \twoheadrightarrow Y \rceil = \lceil X \twoheadrightarrow R-XY \rceil$. For instance, if $R = \{A,B,C,D\}$ then $A \twoheadrightarrow D$ and $A \twoheadrightarrow BC$ are equivalent, and so are $AC \twoheadrightarrow D$ and $AC \twoheadrightarrow B$. According to Proposition 8.9 (2) we have $A \twoheadrightarrow D \Rightarrow AC \twoheadrightarrow D$, but also, for instance, $A \twoheadrightarrow BC \Rightarrow AC \twoheadrightarrow D$. This characterisation of $\Rightarrow$ can be put to work if we want to induce attribute dependencies from tuples, as will be detailed below.

Using a proof procedure

The logical view also offers something that is beyond the relational view as such, namely a proof procedure. I will now illustrate how SLD resolution can be used whenever we want to check whether a given relation satisfies a given dependency. The idea is to use the set of ground facts $\lceil r \rceil$ associated with the relation r as a logic program, and to formulate, for a given dependency represented by a definite clause `H:-B`, a query `?-B,not(H)` which, if it succeeds, demonstrates that the dependency is violated by r.

LEMMA 8.10. *Let r be a relation and D be an attribute dependency, and let $\lceil D \rceil$ = `H:-B`. r violates D iff the query `?-B,not(H)` has an SLDNF refutation from the logic program $\lceil r \rceil$.*
Proof. For the if part, suppose the query `?-B,not(H)` has an SLDNF refutation from $\lceil r \rceil$, and let θ be the computed answer. By the soundness of SLDNF resolution[90] we have that $\forall((B \wedge \neg H)\theta)$ is a logical consequence of $Comp(\lceil r \rceil)$. Since the only Herbrand model of $Comp(\lceil r \rceil)$ is the set of ground facts $\lceil r \rceil$, it follows that every ground instance of $(B \wedge \neg H)\theta$ is satisfied by the Herbrand interpretation $\lceil r \rceil$. We conclude that no ground instance of $(H:-B)\theta$ is satisfied by $\lceil r \rceil$, i.e. `H:-B` is not satisfied by $\lceil r \rceil$, hence by Lemma 8.7 (1) r violates D.
For the only-if part, suppose r violates D, hence by Lemma 8.7 (1) `H:-B` is not satisfied by the Herbrand interpretation $\lceil r \rceil$. Let θ be a grounding substitution such that $(H:-B)\theta$ is not satisfied by $\lceil r \rceil$, i.e. $(B \wedge \neg H)\theta$ is satisfied by $\lceil r \rceil$; since $\lceil r \rceil$ is the only model of $Comp(\lceil r \rceil)$, it follows that $(B \wedge \neg H)\theta$ is a logical consequence of $Comp(\lceil r \rceil)$. By the completeness of SLDNF resolution[91] θ is a computed answer for `?-B,not(H)`.

[90] See e.g. Lloyd, 1987, p.92, Theorem 15.6.
[91] Lloyd, *op. cit.*, p.99, Theorem 16.3. This completeness result only holds for hierarchical

Notice that the two literals in `B` are always grounded by a computed answer θ, since the program contains only ground facts. Thus, if the query `?-B,not(H)` succeeds, every computed answer represents a pair of contradicting tuples from the relation. Furthermore, since all the variables in `H` also occur in `B`, `H` is also grounded by a computed answer: in particular, in the case of an mvd `H` is instantiated to a tuple missing from the relation.

To illustrate the proof-theoretic view, if $R = \{A,B,C,D\}$ then the functional dependency $A \rightarrow D$ is represented by the clause

```
D1=D2:-r(A,B1,C1,D1),r(A,B2,C2,D2).
```

This fd is satisfied by the relation *r* if and only if the following query fails:

```
?-r(A,B1,C1,D1),r(A,B2,C2,D2),not(D1=D2).
```

where = denotes syntactic equality. This query fails for the following relation:

```
r(a1,b1,c1,d1).    % t1
r(a1,b2,c2,d1).    % t2
r(a2,b3,c3,d2).    % t3
r(a2,b4,c4,d2).    % t4
```

thereby demonstrating that the fd $A \rightarrow D$ is satisfied. If however we add the tuple

```
r(a2,b3,c5,d3).    % t5
```

the fd becomes violated. There are four different proofs of this violation by means of SLD resolution, corresponding to four ordered pairs of contradicting tuples:

```
?-r(A,B1,C1,D1),r(A,B2,C2,D2),not(D1=D2).
  A=a2, B1=b3, C1=c3, D1=d2, B2=b3, C2=c5, D2=d3  % t3,t5
  A=a2, B1=b4, C1=c4, D1=d2, B2=b3, C2=c5, D2=d3  % t4,t5
  A=a2, B1=b3, C1=c5, D1=d3, B2=b3, C2=c3, D2=d2  % t5,t3
  A=a2, B1=b3, C1=c5, D1=d3, B2=b4, C2=c4, D2=d2  % t5,t4
```

Clearly, two of these four proofs are redundant.

Suppose we wanted to construct a specialised fd that is satisfied by the relation $\{t_1,t_2,t_3,t_4,t_5\}$. According to Proposition 8.9, the only two non-trivial candidates are $AB \rightarrow D$ and $AC \rightarrow D$ ($AD \rightarrow D$ is satisfied by any relation). However, the first contradicting pair of tuples t_3,t_5 have identical *B*-values, from which we may conclude that they violate the fd $AB \rightarrow D$ as well. What this illustrates is that the SLD proof of violation of a functional dependency can be fruitfully exploited if we want to construct a specialisation that is satisfied.

Similarly, the mvd $A \twoheadrightarrow B$ over the relation scheme $R = \{A,B,C,D\}$ is represented by the following clause:

```
r(A,B2,C1,D1):-r(A,B1,C1,D1),r(A,B2,C2,D2).
```

---

programs, which programs without negated literals in the bodies of clauses trivially are. Furthermore, the requirements of an allowed query (i.e. every variable that occurs in a negated literal also occurs in an unnegated literal) and a safe computation rule (delaying negated goals until all their variables are ground) are satisfied in our case.

This mvd is satisfied if and only if the following query fails:

```
?-r(A,B1,C1,D1),r(A,B2,C2,D2),not(r(A,B2,C1,D1)).
```

where the last call succeeds, by negation as failure, if and only if the corresponding tuple is not in the relation. This query fails for instance when r consists of the following five tuples:

```
r(a1,b1,c1,d1).    % t1
r(a1,b2,c2,d2).    % t2
r(a1,b1,c2,d2).    % t3
r(a1,b2,c1,d1).    % t4
r(a2,b3,c3,d3).    % t5
```

If we add the following tuple

```
r(a2,b4,c3,d4).    % t6
```

the query succeeds in two different ways:

```
?-r(A,B1,C1,D1),r(A,B2,C2,D2),not(r(A,B2,C1,D1)).
  A=a2, B1=b3, C1=c3, D1=d3, B2=b4, C2=c3, D2=d4  % t5,t6
  A=a2, B1=b4, C1=c3, D1=d4, B2=b3, C2=c3, D2=d3  % t6,t5
```

indicating that the mvd $A \twoheadrightarrow B$ is violated. Notice that the proof of violation of an mvd is based on a pair of contradicting tuples, as in the case of fds. Furthermore, from these proofs we can reconstruct the two tuples that are missing from the relation, by substituting the indicated values in `r(A,B2,C1,D1)`[92]:

```
r(a2,b4,c3,d3).    % t7
r(a2,b3,c3,d4).    % t8
```

If we add these two tuples the mvd is satisfied again. This clearly illustrates that mvds behave differently from fds, since a violated fd can never become satisfied again by throwing in some new tuples!

Suppose however that t_7 and/or t_8 are indeed outside the relation r — can we specialise $A \twoheadrightarrow B$ to an mvd that is satisfied by r? Inspection of the answers to the above query reveals that, since t_5 and t_6 agree in their C-value, the specialisation $AC \twoheadrightarrow B$ is violated also. Again we see that a proof of the violation of a dependency contains clues as to what weaker dependency may be satisfied.

§32. INDUCTION OF ATTRIBUTE DEPENDENCIES

After having dealt with the formalities of attribute dependencies, I will now turn to the question how such dependencies are to be induced. It has already been indicated that attribute dependencies are not explanatory, since the explanatory power of one or several dependencies does not comprehend the explanatory power of the database tuples making up the evidence. It will be demonstrated below that attribute dependencies are induced from

[92] This is the head of the clause $\lceil A \twoheadrightarrow B \rceil$.

tuples by confirmatory induction. The distinction made in chapter 7 between regularity-based and consistency-based confirmatory induction also plays a role in the present context. One way to look at confirmation of a dependency is to view the evidence as a completely specified relation, and to define a dependency to be confirmed if it is satisfied by that relation. An alternative to such closed-world reasoning is to view the evidence as a partially specified relation, and to define a dependency to be confirmed if it is satisfied by at least one fully specified extension of that partial relation. Interestingly, both semantics, while behaving differently in the case of multivalued dependencies, coincide for functional dependencies, as will be detailed below.

In this section $\alpha \mid\!\sim \beta$ thus stands for 'α confirms β', where α and β denote a set of tuples and a dependency (or set of dependencies), respectively. I will employ both the relational and the logical notation: e.g. if $R = \{A,B,C\}$ the following two statements are equivalent:

$$\{\langle a,b_1,c\rangle, \langle a,b_2,c\rangle\} \mid\!\sim A \rightarrow C$$

$$\{\texttt{r(a,b1,c)},\texttt{r(a,b2,c)}\} \mid\!\sim \texttt{C1=C2:-r(A,B1,C1),r(A,B2,C2)}$$

One rather important point needs to be considered here: the theory developed in the previous chapter does not allow, in its present form, restrictions on the syntactic form of left- and right-hand side of conjectural arguments, while such syntactic restrictions are obviously playing a role in the present context. Database relations and attribute dependencies are expressed in distinct sublanguages, so it does not make sense to refer to confirmatory arguments, say, of the form $\alpha \mid\!\sim \alpha$. Clearly, this affects a number of rules considered in the previous chapters, such as any form of Reflexivity, Cautious Monotonicity (from $\alpha \mid\!\sim \beta$ and $\alpha \mid\!\sim \gamma$ conclude $\alpha \wedge \beta \mid\!\sim \gamma$; invalid because β cannot refer both to dependencies and tuples), Left Or (from $\alpha \mid\!\sim \gamma$ and $\beta \mid\!\sim \gamma$ conclude $\alpha \vee \beta \mid\!\sim \gamma$; invalid because the evidence cannot be disjunctive), and Right Consistency (from $\alpha \mid\!\sim \beta$ conclude $\alpha \not\mid\!\sim \neg\beta$; meaningless because $\neg\beta$ cannot refer to dependencies).

Among the rules that remain meaningful I mention the following:

- **Right Weakening**:
- **Right And**:
- **Incrementality**:

Apart from some new notation ($\beta \Rightarrow \gamma$ for 'β and γ are sets of dependencies and $\beta \rightarrow \gamma$', $\beta \subseteq \alpha$ for 'α and β are sets (conjunctions) of tuples and $\alpha \rightarrow \beta$') these rules have exactly the same meaning as before. Furthermore, the notion of a prediction can be linked to the notion of the closure $\alpha^{\uparrow}\beta$ of tuples α with respect to mvd β as defined in Definition 8.6: a tuple γ is *predicted*, given a conjectural argument $\alpha \mid\!\sim \beta$, if it is included in the closure of α wrt. β: $\gamma \in \alpha^{\uparrow}\beta$[93]. It is now possible to reformulate the rule of Verification:

[93]Notice that the relation $\Rightarrow$ and the function $^{\uparrow}$ were originally defined for single dependencies; they can be extended to sets of dependencies in a straightforward way.

- **Verification**: $$\frac{\gamma \in \alpha^{\uparrow}\beta \ , \ \alpha \mid\sim \beta}{\alpha \cup \{\gamma\} \mid\sim \beta}$$

I will now take a closer look at regularity-based and consistency-based confirmatory induction of attribute dependencies.

Regularity-based confirmatory induction

In this subsection we define a dependency to be confirmed by a relation *r* if *r* satisfies the dependency. In terms of logic this amounts to demanding that *r* confirms *D* if the Herbrand interpretation $\lceil r \rceil$ is a model of the clause $\lceil D \rceil$; since the interpretation $\lceil r \rceil$ is also the truth-minimal model of the set of ground facts $\lceil r \rceil$, this establishes a form of closed-world reasoning. We thus inherit the relevant rules of the system **CP** (§27), i.e. Right Weakening and Right And. Right And states that the set of confirmed dependencies is itself confirmed, and Right Weakening states that this set is determined by its maximal elements with respect to $\Rightarrow$. This yields a very straightforward algorithm for determining the set of confirmed dependencies.

> ALGORITHM 8.11. *Closed-world confirmatory induction of attribute dependencies.*
> *Input*: a relation scheme *R*, and a relation *r* over *R*;
> *Output*: the set *Dep*(*r*) of strongest attribute dependencies over *R* satisfied by *r*.
> 1. initialise *DEP* to the set of strongest dependencies over *R*;
> 2. while some dependency in *DEP* is violated by *r*, replace it by its minimal specialisations;
> 3. remove from *DEP* every dependency that is weaker than some other dependency, and output *DEP*.

In step 2, specialisations of violated dependencies are constructed according to the method outlined in Proposition 8.9; as indicated in the previous section the pair of contradicting tuples indicates which of the specialisations are possibly satisfied by the relations, and which of them are contradicted by the same pair of tuples. For further details the reader is referred to (Flach, 1990).

One may note that the algorithm outlined above bears a strong similarity to Shapiro's Model Inference System discussed in chapter 3, which also applied a top-down specialisation approach. Algorithm 8.11 is essentially a special case of De Raedt's CLAUDIEN system, which also induces a theory by closed-world confirmatory induction, but without the strong restrictions on the hypothesis language imposed here.

What should interest us here is that, if we restrict attention to functional dependencies, the property of Incrementality holds:

- **Incrementality**: $$\frac{\beta \subseteq \alpha \ , \ \alpha \mid\sim \gamma}{\beta \mid\sim \gamma}$$

The validity of this rule in the fd case has already been proved in Lemma 8.3 (3). The point is that this rule is not a derived rule in the system **CP**, nor is it valid for closed-world reasoning in general; its validity is a special property of functional dependencies, which only check whether a certain condition holds for every pair of tuples from the

relation[94]. This means that Algorithm 8.11 can be adapted to an incremental setting in which tuples are processed one by one.

ALGORITHM 8.12. *Incremental confirmatory induction of functional dependencies.*
Input: a relation scheme R, a relation r over R, the set $Dep(r)$ of strongest functional dependencies satisfied by r, and a tuple t over R;
Output: the set $Dep(r \cup \{t\})$ of strongest functional dependencies over R satisfied by $r \cup \{t\}$.
1. initialise *DEP* to $Dep(r)$;
2. while some fd in *DEP* is violated by $r \cup \{t\}$, replace it by its minimal specialisations;
3. remove from *DEP* every fd that is weaker than some other fd, and output *DEP*.

This algorithm works because, by Incrementality, the set of fds satisfied by $r \cup \{t\}$ is a subset of the set of fds satisfied by r; combined with the convexity of this set, as expressed by Right Weakening, we never need to consider replacements of violated fds that are not specialisations (step 2).

In contrast, Algorithm 8.12 would be incorrect when applied to multivalued dependencies. For instance, let r be given by Table 8.4. One can easily check that this relation satisfies only trivial mvds[95] such as $A \twoheadrightarrow BC$. However, by adding the tuple $t = \langle a_1, b_1, c_2 \rangle$ the mvd $A \twoheadrightarrow B$ becomes satisfied. Algorithm 8.12 would erroneously output $Dep(r \cup \{t\}) = \emptyset$, since $Dep(r) = \emptyset$. Below I will develop an incremental induction algorithm for mvds performing consistency-based confirmatory induction.

A	*B*	*C*
a_1	b_1	c_1
a_1	b_2	c_2
a_1	b_2	c_1
a_2	b_2	c_1

Table 8.4. A database relation satisfying no non-trivial dependencies.

Consistency-based confirmatory induction

One of the ideas considered in §28 was to define a hypothesis to be confirmed if it is not falsified by the information-minimal partial model of the evidence[96]. The information-minimal model of a set of ground facts differs from its truth-minimal model in that it

[94]In database parlance, functional dependencies are *equality-testing*, as opposed to multivalued dependencies, which are *tuple-generating* (Beeri & Vardi, 1981).
[95]Trivial dependencies are satisfied by every relation (i.e. their logical formulations are tautologies).
[96]The presentation can be simplified because in the present case the evidence is always definite, and the hypothesis always has a verifying model.

assigns the truth-value **unknown** to all other facts, rather than **false**. In terms of proof procedures, this amounts to replacing negation as failure in the query `?-B,not(H)` by logical negation. It is easily seen that this does not change anything for functional dependencies, since in that case `H` is of the form `A1=A2`, and the variables `A1` and `A2` also occur in `B`; thus, by the time the call `not(H)` is evaluated it is completely instantiated, and the outcome is the same regardless whether `not` is interpreted as negation as failure, or as logical negation. We conclude that for functional dependencies the consistency-based semantics coincides with the closed-world semantics.

For multivalued dependencies the situation is different, since in that case `H` refers to a tuple, and the behaviour of the call `not(H)` depends on the interpretation of `not`: under negation as failure `not(H)` succeeds if the tuple is not known to be in the relation, while under logical negation it succeeds if the tuple is known to be outside the relation. This calls for the inclusion of negative information in the induction process.

> DEFINITION 8.13. Let R be a relation scheme. A *partial relation* over R is a pair $\langle p,n\rangle$ of *positive* tuples p over R and *negative* tuples n over R, such that $p\cap n=\emptyset$. A partial relation $\langle p,n\rangle$ *satisfies* a multivalued dependency $D = X \twoheadrightarrow Y$ if $t_1\in p$ and $t_2\in p$ and $t_1(X)=t_2(X)$ imply that there exists a tuple $t_3\in p$ with $t_3(X)=t_1(X)$, $t_3(Y)=t_2(Y)$, and $t_3(Z)=t_1(Z)$, and *violates* it if $t_3\in n$. The D-closure of a partial relation $\langle p,n\rangle$ is defined as $\langle p,n\rangle^{\uparrow}D = p^{\uparrow}D$.

Logically speaking a partial relation $\langle p,n\rangle$ is represented by a set $\lceil\langle p,n\rangle\rceil$ of positive and negative ground facts, i.e. a negative tuple $t\in n$ corresponds to a negated observation $\neg n$. Notice that the operation of D-closure transforms a partial relation into a non-partial one satisfying D.

If we define $\alpha \mid\!\sim \beta$ as 'partial relation α does not violate the set of multivalued dependencies β' we have the following properties:

- **Right Weakening**: $$\frac{\beta\Rightarrow\gamma\ ,\ \alpha \mid\!\sim \beta}{\alpha \mid\!\sim \gamma}$$
- **Right And**: $$\frac{\alpha \mid\!\sim \beta\ ,\ \alpha \mid\!\sim \gamma}{\alpha \mid\!\sim \beta\wedge\gamma}$$
- **Verification**: $$\frac{\gamma\in \alpha^{\uparrow}\beta\ ,\ \alpha \mid\!\sim \beta}{\alpha\cup\{\gamma\} \mid\!\sim \beta}$$
- **Falsification**: $$\frac{\gamma\in \alpha^{\uparrow}\beta\ ,\ \alpha \mid\!\sim \beta}{\alpha\cup\{\neg\gamma\} \not\mid\!\sim \beta}$$
- **Incrementality**: $$\frac{\beta\subseteq\alpha\ ,\ \alpha \mid\!\sim \gamma}{\beta \mid\!\sim \gamma}$$

As usual (Lemma 6.5) the combination of Verification and Incrementality is equivalent to Predictive Incrementality:

- **Predictive Incrementality**: $$\frac{\beta\subseteq\alpha^{\uparrow}\gamma\ ,\ \alpha \mid\!\sim \gamma}{\beta \mid\!\sim \gamma}$$

The validity of Incrementality guarantees that the following incremental algorithm is correct.

> ALGORITHM 8.14. *Incremental confirmatory induction of multivalued dependencies.*
> *Input*: a relation scheme R, a partial relation $\langle p,n\rangle$ over R, the set $Dep(\langle p,n\rangle)$ of strongest multivalued dependencies not violated by $\langle p,n\rangle$, and a tuple t over R;
> *Output*: the set $Dep(\langle p\cup\{t\},n\rangle)$ of strongest multivalued dependencies over R not violated by $\langle p\cup\{t\},n\rangle$.
> 1. initialise *DEP* to $Dep(\langle p,n\rangle)$;
> 2. while some mvd in *DEP* is violated by $\langle p\cup\{t\},n\rangle$, replace it by its minimal specialisations;
> 3. remove from *DEP* every mvd that is weaker than some other mvd, and output *DEP*.

The reader may have noticed that if $n=\emptyset$ no mvd is violated by the partial relation $\langle p,n\rangle$, so the output of Algorithm 8.14 will be meaningless. One possible approach is to assume the availability of an oracle, and to replace step 2 in the algorithm by the following:

> 2. while some mvd D in *DEP* is not satisfied by $\langle p\cup\{t\},n\rangle$ do one of the following:
> 2a. if D is violated by $\langle p\cup\{t\},n\rangle$, replace it by its minimal specialisations;
> 2b. if D is not violated by $\langle p\cup\{t\},n\rangle$, query the user about those tuples in the D-closure of $p\cup\{t\}$ but not in $p\cup\{t\}$;

This approach has been implemented in Prolog; below follows an example session. The example is taken from (Maier, 1983, p.123). The relation scheme is {*Flight*, *Day*, *Plane*}, and a tuple `service(F,D,P)` means that flight number `F` flies on day `D` and can use plane type `P` on that day. User input is in bold.

```
?-mvd_induce.
Relation:  service(flight,day,plane).
Dependencies:
     service:[]->->[plane]
     service:[]->->[flight]
     service:[]->->[day]
New tuple:  service(106,monday,747).
New tuple:  service(106,thursday,1011).
Is service(106,thursday,747) in the relation? yes.
Is service(106,monday,1011) in the relation? yes.
```

The user specifies the relation scheme, and the system shows the initial set of strongest mvds. The user types in the first two tuples, which concern flight number 106. The system checks the mvds $\emptyset\rightarrow\!\!\!\rightarrow Plane$ and $\emptyset\rightarrow\!\!\!\rightarrow Day$ by asking for a classification for two

other tuples. Both of these tuples are classified as positive, so none of the mvds is violated (note that $\emptyset \twoheadrightarrow Flight$ cannot be violated, because all tuples have the same flight number).

```
New tuple: service(204,wednesday,707).
Is service(106,monday,707) in the relation? no.
Specialise []->->[plane]
          service(204,wednesday,707)
          service(106,monday,1011)
          not service(106,monday,707)
Is service(204,monday,1011) in the relation? no.
Specialise []->->[flight]
          service(204,wednesday,707)
          service(106,monday,1011)
          not service(204,monday,1011)
Is service(106,wednesday,1011) in the relation? no.
Specialise []->->[day]
          service(204,wednesday,707)
          service(106,monday,1011)
          not service(106,wednesday,1011)
```

The next positive tuple introduces new values for all three attributes. The system tests each of the three initial mvds by posing appropriate queries to the user. None of the three tuples thus constructed is in the relation, so each of the initial mvds is violated and replaced by specialisations, constructed by adding one antecedent attribute, according to Proposition 8.9 (2). This would result in 6 non-trivial specialisations, but note that these are pairwise equivalent: for instance, $Day \twoheadrightarrow Flight$ is a specialisation of $\emptyset \twoheadrightarrow Flight$, but it is equivalent to $Day \twoheadrightarrow Plane$, which is a specialisation of $\emptyset \twoheadrightarrow Plane$. These specialisations are checked in turn: each of them is satisfied by the set of positive tuples.

```
New tuple: service(204,wednesday,727).
New tuple: stop.
Dependencies:
     service:[day]->->[flight]
     service:[flight]->->[day]
     service:[plane]->->[day]
Yes
```

Adding a sixth tuple does not change the set of mvds; as the system doesn't pose further queries, we may conclude that the closure of the partial relation relative to this set of mvds is equal to the positive tuples, i.e. every dependency is satisfied by the relation consisting of the positive tuples (and not just not violated by the partial relation consisting of positive and negative tuples). Consequently, this set of dependencies is the same as would be found by the non-incremental closed-world approach on the basis of the six positive tuples (Algorithm 8.11).

It should be noted that this querying approach can quickly become impracticable when the number of attributes is large, since the number of possible dependencies, and thus the

number of queries, is exponential in the number of attributes. In such a case the non-incremental approach is preferred. A second remark is that the non-incremental approach tests every dependency against the complete relation, which is costly if the number of tuples is large (the SLDNF method for testing satisfaction of an attribute dependency, as outlined by Lemma 8.10, takes $O(n^2)$ steps in the fd case, where n is the number of tuples, and $O(n^3)$ steps in the mvd case; there exist slightly more efficient algorithms). An alternative approach is first to construct the set of violated dependencies by inspecting the relation once, and to construct the strongest satisfied dependencies from the set of violated dependencies without reference to the relation; for the fd case, several algorithms based on this idea have been developed (Kantola *et al*., 1992; Mannila & Räihä, 1992; Savnik & Flach, 1993).

§33. A CONFIRMATORY DISCOVERY PROCEDURE

The methods for constructing a set of strongest attribute dependencies confirmed by a given set of tuples as outlined above embody (confirmatory) proof procedures, in the sense that they are able to derive any confirmed dependency[97]. There is no indication as to which of these dependencies is actually useful in the domain under consideration. The addition of a heuristic utility measure to a proof procedure results in a discovery procedure (see §18). In this section I will discuss such a discovery procedure for attribute dependencies, based on a heuristic evaluation function estimating the utility of the dependency for restructuring the knowledge base. This confirmatory discovery procedure has been implemented in Prolog, and given the name INDEX.

It should be noted that, although it has hitherto not been discussed, the subject of evaluating confirmed hypotheses is an important and largely unexplored subject. Unlike explanatory reasoning, where the main goal is to find a hypothesis that best accounts for the facts, confirmatory reasoning does not carry with it a goal that the hypothesis is to fulfil. Indeed, if 'being confirmed' would be the goal to be optimised, then confirmatory reasoning would be reduced to deductive reasoning[98]. Neither Helft nor De Raedt address this issue: they simply generate the set of most general formulas that are confirmed, according to their definition of confirmation. However, this set can become rather large if the language is complex[99].

The basic idea underlying this section is that a confirmed hypothesis indicates a certain regularity implicit in the evidence, which can be exploited to make the evidence less redundant and more structured. The connection between inducing regularities and restructuring is especially apparent in the case of attribute dependencies, which give rise to so-called decompositions of the given relation into smaller and less redundant relations. The original relation can be composed out of these new relations by means of a composition rule, representing a more meaningful definition of the original relation in terms of the new ones. Such a composite relation is called an *intensional relation*, to

[97]That is, in combination with Right Weakening.

[98]This was noticed by Popper, as discussed in §7.

[99]The complexity of a first-order language is mainly determined by the number and the arity of the predicates.

distinguish it from an *extensional relation* defined by a set of tuples. Logically speaking, an intensional relation is defined by a set of clauses called a *predicate definition*. A *knowledge base*, also called a *deductive database* (Minker, 1988), is a set of extensional and intensional relations, or equivalently, sets of ground facts and predicate definitions.

The goal of attribute dependencies: decompositions

Fds and mvds both describe the same phenomenon: that the consequent attribute(s) can be removed from the relation, and stored in a separate relation containing only the attributes in the dependency. The only difference is, that in the case of fds the antecedent attributes form a key in the second relation. For instance, the following relation satisfies the mvd $A \rightarrow\rightarrow B$:

```
r(a1,b1,c1,d1).
r(a1,b2,c2,d2).
r(a1,b1,c2,d2).
r(a1,b2,c1,d1).
r(a2,b3,c3,d3).
```

The way B depends on A can be described separately, resulting in two new relations:

```
r1(a1,c1,d1).          r2(a1,b1).
r1(a1,c2,d2).          r2(a1,b2).
r1(a1,c2,d2).          r2(a2,b3).
r1(a1,c1,d1).
r1(a2,c3,d3).
```

The original relation can be reconstructed from r_1 and r_2 by means of the following clause:

```
r(A,B,C,D):-r1(A,C,D),r2(A,B).
```

That is, r is now an intensional relation, and this clause serves as its predicate definition. In database terminology, r is the join of r_1 and r_2 over the attribute A.

> DEFINITION 8.15. Let R be a relation scheme, let r be a relation over R, let D be an attribute dependency over R with antecedent attributes X and consequent attributes Y, and let Z denote $R-XY$. The *vertical decomposition* of r imposed by D consists of the two projections $\pi_{XY}(r)$ and $\pi_{XZ}(r)$.

This decomposition is lossless whenever r satisfies D, i.e. r can be reconstructed by performing a join of $\pi_{XY}(r)$ and $\pi_{XZ}(r)$ over the attributes X. The join operation is called the *composition function* associated with the decomposition. It should be noted that both the vertical decomposition and the join operation are uniquely determined by the dependency.

The notion of a decomposition can be generalised: for instance, a partition of a relation in different subrelations can be called a *horizontal* decomposition[100]. As an example, we

[100]This terminology is in accordance with (Paredaens *et al.*, 1989), and corresponds to the direction of an imaginary line separating parts of the original relation. In previous

might partition the train schedule of Table 8.1 into two schedules, one for intercity trains and one for stopping trains. Note that the resulting schedules both satisfy the functional dependency $\emptyset \rightarrow Type$ (i.e. the value of the attribute *Type* is constant in each of them). Clearly, the original relation did not satisfy this fd, so we might say that the horizontal decomposition serves the goal of creating subrelations that are guaranteed to satisfy the dependency — like in the vertical case, we will say that this horizontal decomposition is *imposed* by the dependency. In general, however, if a relation does not satisfy a dependency there are many different horizontal decompositions such that the subrelations satisfy the dependency.

> DEFINITION 8.16. Let R be a relation scheme, let r be a relation over R, and let D be an attribute dependency over R. A *horizontal decomposition* of r imposed by D is a partition $\{r_1,\ldots,r_n\}$ of r into subrelations $r_1,\ldots,r_n$ each satisfying D. Such a decomposition is *minimal* if none of its subrelations can be put together without violating D.[101]

The composition function associated with a horizontal decomposition operates by set-union of the blocks in the partition.

The algorithm for computing minimal horizontal decompositions imposed by a violated dependency is best understood by considering a few examples. Consider the train schedule in Table 8.5, in which every stopping train leaves from platform 4, while all intercity trains for Utrecht, except one, leave from platform 5, and all intercity trains for Tilburg leave from platform 6. The **first step** is to partition the relation into subrelations with equal values for the antecedent attributes[102], indicated by the double lines in Table 8.6; this partition is called the *antecedent partition*, and the sets of tuples with equal antecedent values are referred to as *antecedent blocks*. In the **second step** of the algorithm a further division is made within each antecedent block between tuples with different values for the consequent attribute, indicated by the single line in Table 8.6 (only in the first antecedent block). Clearly, this second division into *non-contradicting blocks* has separated all the pairs of tuples contradicting the fd, so this second partition results in a (non-minimal) horizontal decomposition induced by $Direction,Type \rightarrow Platform$. The **third step** of the algorithm consists in putting together non-contradicting blocks from different antecedent blocks. One obvious way to do this is to keep the 8:57 train separate, and to combine all the other blocks. It should be noted, however, that this is not the only

---

publications (Flach, 1990; 1993) I employed the orthogonal terminology, corresponding to the direction in which parts of the original relation are pulled apart.

[101] Paredaens *et al*. use a different definition of horizontal decomposition (1989, pp.132–134): the relation is separated into two parts, one part satisfying the dependency, one part violating it. The first part is not the **largest** subrelation satisfying the dependency, since the tuples causing violation are kept together in the second part. While their construction has the distinct advantage that it defines a unique horizontal decomposition in the case of fds (the only case they consider), it does not really capture the idea of an exception: if 100 intercity trains in the direction Utrecht leave from platform 5, while one doesn't, their second relation will contain 101 tuples.

[102] In this example I assume that the value of the *Direction* attribute is the **first** station mentioned in the schedule.

H	*M*	*Direction*	*Type*	*Platform*
8	07	Utrecht/Amsterdam	intercity	5
8	09	Tilburg/Den Haag	intercity	6
8	12	Utrecht	stopping train	4
8	16	Tilburg	stopping train	4
8	**57**	**Utrecht/Zwolle**	**intercity**	**6**
9	07	Utrecht/Amsterdam	intercity	5
9	09	Tilburg/Den Haag	intercity	6
9	12	Utrecht	stopping train	4
9	16	Tilburg	stopping train	4
10	07	Utrecht/Amsterdam	intercity	5
10	09	Tilburg/Den Haag	intercity	6
10	12	Utrecht	stopping train	4
10	16	Tilburg	stopping train	4

Table 8.5. A train schedule with an irregular train.

minimal decomposition: for instance, another minimal horizontal decomposition is obtained by combining the trains leaving from platform 4 and 5 on the one hand, and the trains leaving from platform 6 on the other. Step 3 of the algorithm is thus non-deterministic; I will shortly describe a satisfaction measure that can be used to select an appropriate minimal horizontal decomposition.

With multivalued dependencies not only the third step of the algorithm is non-deterministic, but also the second. Consider again the train schedule in Table 8.5: all trains leave every hour except the 8:57 intercity train to Zwolle. If this train is taken out

H	*M*	*Direction*	*Type*	*Platform*
8	07	Utrecht/Amsterdam	intercity	5
9	07	Utrecht/Amsterdam	intercity	5
10	07	Utrecht/Amsterdam	intercity	5
8	**57**	**Utrecht/Zwolle**	**intercity**	**6**
8	12	Utrecht	stopping train	4
9	12	Utrecht	stopping train	4
10	12	Utrecht	stopping train	4
8	09	Tilburg/Den Haag	intercity	6
9	09	Tilburg/Den Haag	intercity	6
10	09	Tilburg/Den Haag	intercity	6
8	16	Tilburg	stopping train	4
9	16	Tilburg	stopping train	4
10	16	Tilburg	stopping train	4

Table 8.6. A horizontal decomposition of the schedule in Table 8.5, imposed by the functional dependency *Direction,Type*→*Platform*.

of the schedule the multivalued dependency $\emptyset \rightarrow\!\!\!\rightarrow Hour$ is satisfied; thus, the 8:57 train can be seen as an exception to this dependency, and separating it from the other trains represents a minimal horizontal decomposition imposed by the mvd $\emptyset \rightarrow\!\!\!\rightarrow Hour$. However, another minimal horizontal decomposition imposed by $\emptyset \rightarrow\!\!\!\rightarrow Hour$ consists of the eight o'clock trains on the one hand, and the nine and ten o'clock trains on the other hand. This indeterminacy becomes apparent in the second step of the algorithm (the first step results in a trivial partition, since there are no antecedent attributes).

ALGORITHM 8.17. *Construction of a minimal horizontal decomposition*[103].
Input: a relation scheme R, a relation r over R, and a dependency D over R with antecedent attributes X;
Output: a minimal horizontal decomposition induced by D.
1. partition r into the set of subrelations $\{a_1,\ldots,a_n\}$ such that the tuples in a_i have the same X-value;
2. partition each a_i into a set of subrelations $\{a_{i1},\ldots,a_{im_i}\}$ as follows:
   2a. find a pair of tuples in a_i contradicting D;
   2b. remove one of those tuples from a_i and go to 2a;
   2c. if a_i does not violate D, repeat step 2. for the removed tuples;
3. combine blocks a_{ij} for different i into a minimal decomposition.

In the INDEX system the indeterminacy in step 3 of this algorithm is handled with help of an oracle, while the indeterminacy in step 2 (only for mvds) is handled heuristically.

Evaluating dependencies and decompositions

The basic idea is to measure the minimum number of tuples that must be removed from the relation in order for a dependency to be satisfied; such tuples are called *exceptions*. The *satisfaction degree* of a dependency is then given by

Sat = 1 – *weighted fraction of exceptions*

The number of exceptions is determined as follows. Suppose for ease of notation that each partition $\{a_{i1},\ldots,a_{im_i}\}$ constructed in step 2 is ordered in decreasing size, then each biggest subrelation we can construct has the same number of tuples as $a_{11}\cup\ldots\cup a_{n1}$; let this number be denoted by N_n. The number of exceptions is then N_R–N_n. If $m_i \leq 2$ for $1 \leq i \leq n$, then all these exceptions can be collected in one subrelation; if some m_i is larger than 2 the exceptions need to be distributed over different subrelations. For instance, if we add a 9:27 train to Utrecht/Zwolle that is leaving from platform 6 to Table 8.5, then it can be combined with the 8:57 train without violating the fd $Direction,Type \rightarrow Platform$; if however this added train is leaving from platform 4 it should be stored in a separate relation. In general, if m is the maximum of $\{m_i \mid 1 \leq i \leq n\}$, then the number of exception relations is m–1 (note that if m=1 the dependency is satisfied). To indicate that 'similar' exceptions are preferred over 'non-similar' exceptions, the fraction of exceptions is

[103]This algorithm assumes that all tuples in the relation r are completely defined, i.e. there are no null-values. When null-values are present it may be possible, depending on the interpretation of the null-value, to instantiate some of them such that the dependency becomes satisfied. For an overview of null-values and their interpretation see (De Troyer, 1993, pp.18–22).

weighted with the number of exception relations. This yields the following formula:

$$Sat = 1 - (m-1) * \frac{N_R - N_n}{N_R}$$

As an illustration, for the fd *Direction,Type→Platform* we have *Sat*=12/13=0.92 in Table 8.5; if we add a 9:27 train to Utrecht/Zwolle leaving from platform 6 we get *Sat*=12/14=0.86, but if this train is leaving from platform 4 we get *Sat*=10/14=0.71.

A second heuristic employed in the INDEX system measures the average size of blocks in the antecedent partition, to prevent the generation of dependencies that are very weak (i.e. have many antecedent attributes).

The INDEX system

The main algorithm implemented in the INDEX system can now be described as follows.

ALGORITHM 8.18. *The INDEX algorithm.*
Input: an extensional relation *r*;
Output: an intensional definition of *r* in terms of a number of subrelations.
1. determine the strongest attribute dependencies that are (almost) satisfied by a given relation *r*, and select a dependency *D*;
2. construct a minimal horizontal decomposition $\{r_1,\ldots,r_n\}$ induced by *D*;
3. construct the vertical decomposition induced by *D* for each of $r_1,\ldots,r_n$.

Step 1 is implemented by performing a top-down search for dependencies in the spirit of

```
% train(Hour,Minutes,Direction,Type,Platform)
train(8,07,utrecht,intercity,5).
train(8,09,tilburg,intercity,6).
train(8,12,utrecht,stopping_train,4).
train(8,16,tilburg,stopping_train,4).
train(8,37,utrecht,intercity,5).
train(8,39,tilburg,intercity,6).
train(8,42,utrecht,stopping_train,4).
train(8,46,tilburg,stopping_train,4).
train(8,57,utrecht,intercity,6).
train(9,07,utrecht,intercity,5).
train(9,09,tilburg,intercity,6).
train(9,37,utrecht,intercity,5).
train(9,39,tilburg,intercity,6).
train(9,42,utrecht,stopping_train,4).
train(9,46,tilburg,stopping_train,4).
```

Table 8.7. An extensionally specified train schedule.

the algorithms discussed earlier; the user can select between a non-incremental approach as in Algorithm 8.11, or an incremental approach in which queries are asked (Algorithm 8.14). The difference with these algorithms is that in the INDEX system search also stops at dependencies that are almost satisfied, with a user-definable threshold for the satisfaction measure *Sat* (typically ≥0.75). The user can also choose the consequent attributes in which she is interested. INDEX presents the dependencies it found to the user, who should make the final selection.

Step 2, construction of a minimal horizontal decomposition, has been discussed above (Algorithm 8.17). A certain amount of user interaction is required here also, in order to put the non-conflicting blocks from different antecedent blocks together in a meaningful way. After completion of this step, the user can indicate for each subrelation whether she wants the imposed vertical decomposition to be constructed, which is typically the case for the normal tuples but not for the exceptions. Note that the construction of such vertical decompositions is deterministic and straightforward (see Definition 8.15).

For a complete and annotated session with the INDEX system the reader is referred to the appendix; I will confine myself here to presenting the results. The input to the system is given in Table 8.7; this train schedule is the same as the one at the beginning of the chapter (Table 8.1), with an extra irregular train at 8:57. The selected dependency is $\emptyset \rightarrow\!\!\!\!\rightarrow Hour$, which has 3 exceptions (*Sat*=12/15=0.8). After horizontal and vertical decomposition the resulting restructured knowledge base is as in Table 8.8. The first two clauses indicate that the original `train` relation is the union of the new relations `hourlytrain` and `irregtrain`; the third clause expresses that `hourlytrain` has

```
train(H,M,D,T,P):-
     hourlytrain(H,M,D,T,P).
train(H,M,D,T,P):-
     irregtrain(H,M,D,T,P)

hourlytrain(H,M,D,T,P):-
     hour(H),
     hourlytrain1(M,D,T,P).

hourlytrain1(07,utrecht,intercity,5).    hour(8).
hourlytrain1(09,tilburg,intercity,6).    hour(9).
hourlytrain1(37,utrecht,intercity,5).
hourlytrain1(39,tilburg,intercity,6).
hourlytrain1(42,utrecht,stopping_train,4).
hourlytrain1(46,tilburg,stopping_train,4).

irregtrain(8,12,utrecht,stopping_train,4).
irregtrain(8,16,tilburg,stopping_train,4).
irregtrain(8,57,utrecht,intercity,6).
```

Table 8.8. The same train schedule as in Table 8.7, represented as a knowledge base with explicit indication of hourly trains.

been vertically decomposed[104] into `hourlytrain1` and `hour` (the names of the new relations have of course been supplied by the user).

§34. SUMMARY AND CONCLUSIONS

Unlike explanatory induction, which has the explicit aim of inferring explanations of observations, confirmatory induction does not have a predefined goal. In this chapter I have explored the idea of using confirmatory hypotheses to restructure a given database relation. The resulting knowledge base explicates the implicit structure indicated by attribute dependencies. This approach has been implemented in a prototype system called INDEX, and can be classified under the heading *knowledge discovery in databases*, a rapidly growing research field (Piatetsky-Shapiro & Frawley, 1991).

Work on this subject has only just started, and much remains to be done. Attention needs to be devoted, in particular, to the development of better heuristics; to a model for evaluating the performance of systems such as INDEX; to decrease the amount of user interaction by making use of domain knowledge; and to study other classes of integrity constraints that can give rise to restructuring of extensional information. As for the last point, a possible direction is to study properties like transitivity of binary relations: if we know that an extensionally specified relation is transitive, it can be compressed into a smaller relation of which the original relation is the transitive closure. Like a horizontal decomposition, such a 'transitive decomposition'[105] can be achieved in many ways — if, in addition, we would know that the relation is irreflexive, the minimal transitive decomposition is however unique[106]. One can [illegible] such integrity constraints, with associated decomposition operations and confirmatory induction strategies.

A related point is that such decomposition approaches may provide the key to a better understanding of the relation between confirmatory and explanatory induction. Consider again the knowledge base in Table 8.8, which consists of a few extensionally specified subrelations B, and a few intensional definitions H by means of which the original extensional relation E can be derived: $B \wedge H \vdash E$. Another way to view this relation between B, H and E is by noting that H forms an explanation of E given B. In other words, an explanatory induction system should be able to induce H from E given background knowledge B. This observation hints at a certain relationship between explanatory and confirmatory induction: investigating this relationship is, in my opinion, a major research topic in the logical theory of inductive reasoning.

* * * * * *

[104] That is, this clause expresses that `hourlytrain` is the join of `hourlytrain1` and `hour` over the empty set of attributes — i.e. the cartesian product.

[105] Here I use the term 'decomposition' informally as the inverse of an integrity constraint, i.e. 'transitive decomposition' is a restructuring operation imposed by the integrity constraint of transitivity.

[106] In this respect one can think of the ancestor relation, which is transitive and irreflexive, the minimal transitive decomposition of which is the parent relation. For (strict) partial orders this relation corresponds to the set of vertices in the Hasse diagram.