FORMAL ASPECTS IN DATABASES

C. Delobel
Institut de mathématiques appliquées de Grenoble

Abstract: From the beginning of the relational data models special attention has been paid to the theory of relations through the concepts of decomposition and dependency constraints. The initial goal of these works was devoted to the scheme design process. Most of the results are used in this area but serve as a basis for improvements of the model in several directions: incomplete information, universal relations, deductive databases, etc...

CHAPTER 1
DATA MODELS AND PREDICATE CALCULUS

1. Data model approach

From an historical point of view the relational model introduced by Codd (CO70) has appeared as a data model. The intent of this section is to review some basic definitions about the notions of relation and relational scheme. Attributes are symbols taken from a finite set \( U = \{ A_1, A_2, \ldots, A_n \} \). We shall use the letters \( A, B, \ldots \), for single attribute, and the letters \( X, Y, Z, \ldots \) for sets of attributes. Each attribute \( A \) has associated with it a domain, denoted by \( \text{dom}(A) \), which is the set of possible values for that attribute. The elements of a domain will be represented by lowercase letters, \( a, b, x, y, \ldots \). Also, following common notational convenience, we shall let \( XY \) denote the union of attribute sets \( X \) and \( Y \). A tuple \( t \) of type \( X \) is an application from \( t \) to \( \text{dom}(X) \), such that for every \( A \in X, t(A) \) is an element of \( \text{dom}(A) \). A tuple \( t \) of type \( ABC \) will be usually denoted as a sequence of values, \( t = (a, b, c) \). More generally, if \( t \) is a tuple of type \( X \) and for \( Y \subseteq X \) we shall denote by \( t[Y] \) the restriction of \( t \) to the \( Y \)-attributes.

A relation \( r \) of type \( X \) is a finite set of tuples of type \( X \). A relation is an instance of a relation scheme. By relation scheme we mean both a symbol of name \( R \) and a set \( X \) of attributes. It is convenient to introduce a function \( \exists \) which associates for every object its type; for example, if the tuple \( t \) is of type \( X \) then \( \exists(t) = X \).
A database relational scheme is a pair \((R, IC)\), where \(R\) is a set of relation schemes \(R = \{R_1, R_2, \ldots, R_n\}\), with \(\beta(R_i) = X_i\), for \(1 \leq i \leq n\), and \(IC\) a set of integrity constraints. Integrity constraints are sentences concerning the relations which specify the valid states of the database. A database \(r = \{r_1, r_2, \ldots, r_n\}\) is a set of relations where every \(r_i\) is an instance of \(R_i\) and obeys the sentences in \(IC\). When no confusion is permitted, we shall identify a relational schema \(R\) from the type of each relation name, thus \(R = \{X_1, X_2, \ldots, X_n\}\). \(REL(R)\) defines the set of databases of type \(R\), and \(SAT(IC)\) the set of databases which satisfies the integrity constraints \(IC\).

In the original presentation of the relational model Codd introduced two different types of languages as data manipulation language: the relational algebra and a first order logical language. There are five basic operations that will be of some interest to us: projection, join, selection, union and difference.

Let \(r\) be a relation of type \(X\), the projection of \(r\) over an attribute set \(Y \subseteq X\) is defined by:

\[
r[Y] = \{t[Y] : t \in r\}.
\]

Let \(r\) and \(s\) be two relations of type \(X\) and \(Y\), the join of \(r\) and \(s\) is a relation denoted by \(r \ast s\) and defined by:

\[
r \ast s = \{t : \beta(t) = XY \text{ et } t[X] \in r \text{ et } t[Y] \in s\}.
\]

Let \(r\) and \(s\) be two relations of the same type \(X\), the union \(r+s\), and the difference \(r-s\) are respectively defined by:

\[
r+s = \{t : t \in r \text{ ou } t \in s\},
\]

\[
r-s = \{t : t \in r \text{ et } t \not\in s\}.
\]

Finally, the selection operation consists in extracting from a relation \(r\) the tuples which obeys a boolean expression \(E\), it is denoted \(r : E\).

2. **First order relational language and relational structure**

Mathematical logic is a power tool to describe both the concept of database and data manipulation languages. A first order language is a pair \((A, F)\) where \(A\) is an alphabet of symbols and \(F\) a set of well formed formulae built from the alphabet and syntactic rules. Primitive symbols of such a language are:

- variable \(x, y, \ldots\)
- constants \(a, b, \ldots\)
- predicates \(P, Q, \ldots\)
- logical constants, \& (and), | (or), \neg (not), \Rightarrow (implication)
- quantifiers, \((x)\) (forall \(x\)),
  \((\exists x)\) (there exists \(x\))
- functions \(f, g, \ldots\)
A term is defined recursively to be a constant, a variable, or
if f is a function symbol and t₁, t₂, ... are terms, then (t₁,
t₂, ...) is a term.
If P is a predicate symbol and t₁, t₂, ... are terms then P(t₁,
t₂, ...) is an atomic formula.
A well formed formula, or more simply a formula, is defined
recursively to be an atomic formula, or if F₁ and F₂ are formulae
then F₁ & F₂, F₁ | F₂, F₁ ⇒ F₂, ¬F₁ are formulae. If x is
a free variable in a formula F, then (x)(F) and (Ex)(F) are also
formulae. A closed formula is a formula where all the variables
are quantified.

Relational languages (Re 81) have special properties due to
the database area. First of all, we shall consider a finite set
of constants and predicates. Among all predicates one is specially
distinguished, it is the equality predicate ( = ) for checking
if two objects are identical. Symbol functions are eliminated
since they induce bad properties related to infinite set of values.
Finally among all predicates we shall assume that there exists
a finite set of unary predicates associated with every independant
domain of values for a variable; that is that every variable
will be typed by a predicate. The type predicate is defined
recursively from primitive type as follows :

- a primitive type is a type predicate
- if β₁ and β₂ are type predicates then β₁ & β₂, β₁ | β₂
  and ¬β₁ are type predicates.

According to these definitions it is convenient to introduce
some abbreviation rules for formula expressions :

- (x.β)(F) means "for all x such x is β F is the case"
  and stands for the correct expression (x)(β(x) ⇒ F),

- (Ex.β)(F) means "there exists x in such F is the case",
  and stands for the correct expression (Ex)(β(x) & F).

Once the language has been defined, it is necessary to assign
to every formula a meaning. It is the role of an interpretation
to do so. An interpretation I of a set of formulae consists in
the specification of a triple (D, K, E) where :

- D is a finite domain of values, for instance the database values,
- K is an application which associates for each constant symbol
  in A an element of D,
- E is an application which assigns for each n-ary predicate
  symbol a relation of D^n, called the extension of the
  predicate.

In an interpretation we are concerned with the truth evaluation
of a closed formula. The truth value of a closed formula is
obtained according to the usual rules of boolean calculus. An
interpretation of a set of formulae is a model if all the formulae
are true in the interpretation. Let G be a set of formulae and
F a formula, we say that F is a logical consequence of G, if
F is true in all models of G. This is denoted by G ⊨ F.
Let \( (R_1, R_2, \ldots, R_n), IC \) be a relational scheme and a
database state \( r = \{r_1, r_2, \ldots, r_n\} \), it is possible to establish
an easy connection between the data model approach and the concept
of interpretation. First of all, we have to define the first
order language: the set of constants (is the set obtained from
the union of the underlying domains of all attributes that occur
in the relational schema, for each relation name \( R_i \) we assign
a predicate \( R_i \) (same name for simplicity), for each independent
attribute domain we relate an unary predicate \( \beta_i \). Then, if we
assign for every predicate \( R_i \) an instance \( r_i \), for every unary
predicate \( \beta_i \) its extension \( E(\beta_i) \), the pair \( (r, E(\beta)) \) is an obvious
interpretation of the language. Furthermore \( (r, E(\beta)) \) is a model
for the integrity constraints IC expressed as formulae if IC
contains formulae such that:

\[
(x_1, x_2, \ldots, x_n)(R_i(x_1, x_2, \ldots, x_n) \implies \\
\beta_1(x_1) \land \beta_2(x_2) \land \ldots \land \beta_n(x_n)
\]

for each predicate \( R_i \).

A query can be expressed as a formula with free variable:
\(< x.\beta : F(x) >\), where \( x.\beta \) denotes a sequence of typed variables
\( x_1.\beta_1, \ldots, x_k.\beta_k \) and \( F \) a formula. The query answer is a set of
\( k \)-tuples \( (c_1, c_2, \ldots, c_k) \) such that:

\[
\beta_1(c_1), \beta_2(c_2), \ldots, \beta(c_k) \land F(c_1, c_2, \ldots, c_k)
\]

are satisfied.

The notion of query defined here above does not mention the
concepts of safe formula (Ul 80), and evaluable formula (Dem82).
These concepts have been introduced for considering "reasonable"
query. Roughly speaking a reasonable query is a query where the
range of a variable is perfectly defined.

3. A logic theory for database

The first order predicate calculus is a formal system whose
objects are sentences and which has a set of axioms: the logical
axioms and the inference rules. It is possible to add new axioms
to built a theory \( T \). A formula \( F \) is derivable from a set of
formulae \( G \), iff \( F \) is deducible from \( G \) and from the axioms in
a finite number of steps by using the inference rules. This will
be denoted \( G \vdash F \).

In section 2 we have shown that a database can be seen as
an interpretation of a first order language. We can consider
also a database from a different viewpoint (Re 81, Ko 80), as
a theory where the proper axioms of the theory are representations
of all the elementary facts about the database. In this approach
queries and integrity constraints are formulae to be proved.

One important result is the following: if \( F \) is a formula
from a relational language which is satisfied in some
interpretation, then there exists a theory \( T \) where \( F \) is deducible
from \( T \) and reversely.
The interest of this result is to state precisely the proper axioms of the theory:

1. Each elementary fact in the database is an axiom, that is, if \( t = (a_1, a_2, \ldots, a_n) \) is a tuple related to the relation scheme \( R \), then \( R(a_1, a_2, \ldots, a_n) \) is in \( T \).

2. The domain closure axiom which says that a variable can take its values only inside the domain \( D \).

3. The completion axioms which states that for each relation predicate the only facts known to be true are those defined in the extension of the predicate, the others are false.

4. The equality axioms are related to the equality predicate. These axioms specify the usual properties of equality: reflexivity, symmetry, transitivity, principal of substitution of equal terms.

5. The unique name axioms which state that all the individual objects can be distinguished.

Let \((L, I, IC)\) be a relational database, where \( I \) is an interpretation and \( IC \) a set of integrity constraints expressed as formulae in the language \( L \), then it is equivalent to consider a triple \((L, T, IC)\) where \( T \) is a theory, which admits \( I \) as a unique model. If \( F \) is a truth formula in \( I \), \( F \) is deducible from \( T, T \vdash F \). Furthermore a query \( Q = \langle x.\beta : F(x) \rangle \), where \( x \) is a free-type variable, consists of those constants \( c \) such that \( T \vdash \beta(c) \) and \( T \vdash F(c) \).

4. Why a formal approach?

Logic provides a convenient formalism in which the basic concepts of databases can be studied in a rigorous and unified way (GMN 81 84)(GMN 84). The two approaches: the relational model view and the theory view give complementary views of the same problem. In the relational model view the relational scheme is an incomplete specification of an application. We need application programs to define precisely the database states. At the implementation level the two approaches are very different and opposite. In the theory view each formula is coded as its own, and all the retrieve operations are defined in term of theorem to be proved. This situation is favored by the community of PROLOG systems where all the elementary facts about a database are stored as Horn formulae. On the other hand, in the model view, elementary facts are stored in logical records and retrieve operations are performed by relational operations such as join, selection,... Today there is a lot of work where researchers are looking at the integration of new mechanisms for retrieval and deduction in the same system. Logic is also useful for other aspects of databases: specification of integrity constraints, null values and incomplete information.
CHAPTER 2
DEPENDENCIES

1. Introduction

In the design process of databases more attention has been paid to special types of constraints, called the family of data dependencies. The data dependencies play a central role in the relational database theory. The introduction of functional dependencies, multivalued dependencies and subsequent formalizations capture a significant amount of semantic information which is useful to normalize and decompose relation schemes. The decomposition of a relation is a topic that has been studied extensively because progress in logic database design has been strongly related to it. Some design methodologies for professional analyst are derived from these studies. Today, we know what its limits and usability are.

When defining a relational scheme, it is not clear what is the real meaning of the relations and it is often explained implicitly. It is the objective of data dependencies to explain how data are related to each other and what kinds of connection exists between these data?

We shall consider a relational scheme R (Agent, Product, Price, Company). It is possible to define different meanings; it is the purpose of the following sections.

2. Functional dependencies and multivalued dependencies

Example 2.1: the relational scheme has the following meaning: a company has agents who sell product at a given price. If we assume that all the relation instances of R have the same property: a product has a unique price, we introduce a functional dependency denoted \( P \rightarrow I \). Formally a functional dependency may be defined by a formula:

\[
(2.1) \quad (a,a',p,i,i',c,c') ((R(a,p,i,c) \& R(a',p,i',c')) \Rightarrow i = i')
\]

where R is the predicate name for the relation R.

Functional dependencies play a central role in the normalization process introduced by Codd. Instead of normalization we prefer the terminology of decomposition.

According to the decomposition theory if a database designer decides to store information about the relational schema \( R(A,P,I,C) \) in a unique relation, this schema suffers from the following update anomalies:

1. two different tuples of a relation may contain the same product and consequently the same price, in this case there is some redundancy in the information content.
2. if a given product is only delivered by a company, and if this product is no longer provided by the company, then the price of the product is lost.

In the decomposition theory the basic idea is to replace the initial scheme \( R(A,P,I,C) \) by two schemes \( R_1(A,P,C) \) and \( R_2(P,I) \), where the information about the product prices is given by \( R_2 \). This conceptual design approach can be formalized by the decomposition process of a relation.

Let \( r \) be a relation of the relation scheme \( R(X,Y,Z) \), we say that \( r \) is decomposable if we can find two relations \( r_1 \) and \( r_2 \), of type \( XY \) and \( XZ \), such that:

- \( r_1 = r[X,Y] \), \( r_2 = r[X,Z] \), this means that \( r_1 \) and \( r_2 \) are projections of \( r \).
- \( r = r_1 \ast r_2 \), the join of \( r_1 \) and \( r_2 \) is \( r \).

There exists an easy generalization of decomposition in more than two parts. This concept of decomposition is very important, because instead of storing the relation \( r \) in the database, we can store only the projections of \( r \). The goals which are achieved by decomposition are:

- reduction of the degree of redundancy contained in a relation
- breaking down the semantic facts represented in a relation into more elementary facts.

These two goals are not really independent. All the historic developments of the relational database theory illustrates the fact that researchers are trying to determine what are the different types of constraints which may influence the decomposition. Most of the family dependencies rely upon the two equivalent propositions which give the characterization for a relation to be decomposable.

A relation \( r \) of type \( XYZ \) is decomposable iff for all \( x \in r[X] \), \( r[x,Y,Z] = r[x,Y] \times r[x,Z] \), where \( x \) denote the cartesian product, \( r[x,Y,Z] \) the set \( \{(y,z) : (x,y,z) \in R \} \). An equivalent condition is for all \( (x,y) \in r[X,Y] \), \( r[x,y,Z] = r[x,Z] \).

A relation is decomposable if it satisfies a property defined on the values of the relation. This fact is not very useful since it needs a complete checking of the values contained in the relation. The importance of the family dependencies is to characterize the property expressed in terms of semantic properties. For example, if \( r \) is an instance of the relational scheme \( \{ (X,Y,Z), (X \rightarrow Y) \} \) then \( r \) is decomposable into two relations of type \( (X,Y) \) and \( (X,Z) \). A functional dependency implies a decomposition but the reverse is not true.

If we consider our example \( R(A,P,I,C) \) with \( P \rightarrow I \) then we can decompose the schema into \( R_1(A,P,C) \) and \( R_2(P,I) \). Furthermore,
we can explore what is the meaning of the association between the others attributes. A company has several agents, then
\[ C \not\rightarrow A, \text{ but the knowledge about a company determines the set} \]
of agents independently of the product and the price; this is
the case of a multivalued dependency denoted by \[ C \rightarrow Y. \]

Formally we say that a relation \( r \) of type \( U \) satisfies the
multivalued dependency \( X \rightarrow Y \), where \( X \) and \( Y \) are subparts
of \( U \), and \( Z = U \times X \), if for all pair of tuples \( s \) and \( t \) in \( r \)
satisfying \( s[X] = t[X] \), then there exists a tuple \( w \) such that:
\[ s[X] = t[X] = w[X], \quad w[Y] = t[Y] \text{ and } w[Z] = s[Z]. \]

It is easy to show that the definition of a multivalued
dependency is equivalent to the condition for a relation to be
decomposable.

We have seen that a functional dependency can be expressed
as a special formula, this is the same for a multivalued
dependency; the multivalued dependency \( C \rightarrow A \) is equivalent
to the formula :

\[ R(a1,p1,i1,c) \cup R(a2,p2,i2,c) \rightarrow R(a1,p2,i2,c) \text{ where all} \]
the variables are universally quantified.

Thus, our knowledge about relations on the scheme \( R(A,P,I,C) \)
can be more precisely described by

\[ \{A,P,I,C\} \cup \{C \rightarrow A, P \rightarrow I\}. \]

From the multivalued and the functional dependency every relation
\( r \) of the scheme can be decomposed into three parts : \( r1 = r[CA], \)
\( r2 = r[PI], r3 = r[PA] \), this means that \( r \) is the join of
three relations, each relation belonging respectively to scheme
\( R1(C,A), R2(P,I), R3(P,A) \). These schemes are irreducible, then
they can be interpreted as the atomic meaning of the initial
scheme \( R(A,P,I,C) \).

One can think about the relational scheme \( R(APIC) \) in a different
way. If we assume that for every company : "all the products
delivered by a company are sold by all the company's agents",
then this property can be stated in formal terms : if \( r \) is a
relation for all \( c \) in \( r[c] \)

\[ r[c,P,A] = r[c,P] \times r[c,A]. \]

This expression is the decomposition condition, not for the
relation \( r \), but for a projection of \( r \) over the attributes set
\( A,P,C \). To capture this new type of dependencies we talk about
embedded multivalued dependencies or hierarchical dependencies
(De 73) (De 78) (TKY 7a), the notation used is \( c \rightarrow \rightarrow P|A. \)
If the relational scheme is defined by \( R = \{A,P,I,C\}, \)
\( \{P \rightarrow I, C \rightarrow \rightarrow P|A\} \), we obtain by decomposition the same
result.
3. Join dependency

Example 2.2: we have seen that multivalued dependencies characterize exactly the decomposition process of a relation into two parts. However, multivalued dependencies are inadequate for expressing the conditions under which a decomposition into more than two parts can occur. The following interpretation of the relational scheme \( R(A, P, I, C) \) is used to illustrate the concept of join dependency. As in the first case, we assume that a product has a unique price, but furthermore, we have three associations of interest: an agent sells products, a company distributes product, and an agent works for a company.

We can see each association: agent-product, product-company, company-agent as a relation which is the projection of a relation \( r \) over a set of attributes. In such a case, the relational scheme \( R(A, P, I, C) \) has a special property (NI 78) expressed by the join operation:

\[
\]

For example the relation \( r \) of type \( A, P, C \) satisfies this property:

<table>
<thead>
<tr>
<th>A</th>
<th>P</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2</td>
<td>nut</td>
<td>c2</td>
</tr>
<tr>
<td>a1</td>
<td>nut</td>
<td>c2</td>
</tr>
<tr>
<td></td>
<td>screw</td>
<td>c1</td>
</tr>
<tr>
<td>a1</td>
<td>screw</td>
<td>c2</td>
</tr>
<tr>
<td>a2</td>
<td>bolt</td>
<td>c2</td>
</tr>
</tbody>
</table>

A join dependency will be denoted by \( *[AP, AC, PC] \), in the present case we say that it is a partial join dependency since it is not defined over the whole set of attributes. We shall see later that it is also a cyclic dependency because the association graph has a cycle.

More formally, we can define a join dependency as a sentence, denoted \( *[X_1, X_2, ..., X_k] \), where \( X_i \) are parts of an attribute set \( U \), and such that \( UX_i = U \), if for every relation \( r \) of type \( U \), \( r \) is the join of the projections \( r[X_1], r[X_2], ..., r[X_k] \). A relation \( r \) of type \( U \) satisfies this property if and only if for each tuple \( t \) of \( r \) there exists tuples \( w_1, w_2, ..., w_k \) of \( r \) (not necessarily distinguished) such that:

\[
w_i[X_i] = t[X_i] \text{ for } i = 1, 2, ..., k.
\]

A partial join dependency is a join dependency over a set \( U' \) of attributes contained in \( U \).

If the relational scheme \( R \) is defined by

\[
R = (\{A, P, I, C\}, \{P \rightarrow I, *[AP, AC, PC]\})
\]

it is easy to deduce that every relation \( r \) is decomposable into four parts. \( R \downarrow (PI) \),
R2(A,P), R3(A,C), R4(P,C). In fact, the interpretation of a relation r is given by:

\[(a,p,i,c) : i \text{ "is the price of" } p \text{ and a "sells" } p \text{ and a "works for" } c \text{ and } c \text{ "distributes" } p\].

If we introduce the predicate P1, P2, P3 and P4 respectively for "is the price", "sells", "works for", "distributes", the relation is of the form

\[(2.4) \{(a,p,i,c) : P1(p,i) \& P2(a,p) \& P3(a,c) \& P4(c,p)\}\].

The fact that a relation r of type A,P,I,C is of the form (2.4) is a severe constraint for some predicate P1, P2, P3 and P4, but we have the real meaning of a relation in terms of basic elementary fact. Before we leave the join dependencies, let us note, that a join dependency can be written as a logical formula. The join dependency *[AP, AC, PC] can be expressed as

\[(2.5) (a,a1,p,pl,c,cl,il,i2,i3) \rightarrow (R(a,p,il,cl) \& R(a,p1,i2,c) \& R(a1,p,i3,c)) \Rightarrow \exists iR(a,p,i,c).\]

In the formula we use existential quantifier since it is a partial join dependency.

4. Inclusion dependencies

Example 2.3: so far, all the dependencies we have studied have the property of being defined over a unique relation. The class of inclusion dependencies is a multi relation property (Da 81, Li 81, CPP 84).

For example for a relation r of type A,P,I,C, a tuple (a,p,i,c) cannot be stored in the database until we can guarantee that a is an agent, p is a product and c is a company. This kind of properties is frequently used in database system.

Thus, we need three unary relation R1(A), R2(P), R3(C), where each relation contains respectively the list of all agents, products and companies. The constraints can be written as:

\[(2.6) R[A] \subseteq R1(A)\]
\[R[P] \subseteq R2(P)\]
\[R[C] \subseteq R3(C)\]

In general an inclusion dependency is of the form

\[R[X'] \subseteq S[Y']\] where R and S are relation scheme over sets X and Y of attributes, and X' \subseteq X and Y' \subseteq Y. Let r and s be relations of type X and Y. An inclusion dependency holds if for each tuple t of r, there exists a tuple w of s such that t[X'] = w[Y'].

In the network or hierarchical model, they are implemented most of the time in an easy way inside the logical structure of the
database. In relational system we need to specify this kind of constraint between relations.

Let \( R(ABC) \) and \( S(CDE) \) be two relation schemes: the inclusion dependency \( R[AB] \subseteq S[CE] \) can be written as a sentence in a first order language:

\[
(2.7) \quad \text{(abc)} (R(abc) \implies (Ed) S(adb)).
\]

The examples that we have presented illustrate the fact that dependencies capture the basic properties of association between data. Moreover the decomposition process modelizes database design in such a way that the output is a consequence of the input data dependencies. Figure 2.1 is a graphical representation of our knowledge about the real logical structure presented in the examples 2.1, 2.2 and 2.3. A relational scheme can be represented by an hypergraph where nodes are attributes and edges are sets of attributes.

![Graphical representation of a relational scheme](image)

**Figure 2.1**: graphical representation of a relational scheme.

5. **General expression for dependencies**

Dependencies can be written as first order sentences where we consider:

- predicate relations of the form \( R(...) \),
- the equality predicate, \( x=y \) where \( x \) and \( y \) are variables,
- an atomic formula is a predicate relation or an equality predicate

A dependency formula is of the form (Fa 84):

\[
(2.7) \quad (x_1...x_m)(A_1 \& ... \& A_n) \implies (Ey_1...y_k) \implies (B_1 \& ... \& B_r)
\]

where the \( A_i \)'s are predicate relation and the \( B_i \)'s atomic formulae, the \( x_i \)'s and \( y_i \)'s are variables. In the case \( k=0 \) there are none existential quantifiers.

One class of dependency formulas has been studied extensively because they are related to join dependencies, it is the class of implicative dependencies (BV 81)(SU 81) which are special Horn formulae with a unique predicate \( P \). An Horn formula:

\[
(2.8) \quad (P(x_{11},...,x_{1n}) \& ... \& P(x_{m1},...,x_{mn})) \implies P(x_{01},...,x_{0n})
\]
is an implicative dependency, if \( x_{ij} = x_{kl} \) implies \( j = 1 \),
\((0 \leq i, k \leq m, 1 \leq j, l \leq n)\). An implicative dependency can be
represented by a tableau where the entries are formed from the
symbols taken in the formula.

Entries from 1 to \( m \) are the conditional entries, and the last
entry is the conclusion. The tableau associated with the formula
2.8 is:

\[
\begin{array}{cccc}
x_{11} & \ldots & x_{1n} \\
x_{21} & \ldots & x_{2n} \\
x_{m1} & \ldots & x_{mn} \\
\hline
x_{01} & x_{0n}
\end{array}
\]

According to the definition if a variable appears in different
entries of the tableau then this variable is always in the same
column.

A full implicative dependency is an implicative dependency
if the conclusion variables are contained in the condition
variables. An implicative dependency is call reduced if for all
\( i, j(i \neq j) : x_{j} \cap x_{0} \neq x_{i} \cap x_{0} \) (\( x_{i} \) denotes the entry numbered
\( i \)), and is called join implicative dependency, if for all \( i, j \)
\((0 \leq i < j \leq m) \) (\( x_{j} - x_{0} \) \( \cap \) (\( x_{i} - x_{0} \) = \( \emptyset \). A join implicative
dependency is equivalent to a join dependency. For example the
join dependency

\[ [AB,BCD,CDEF,BCE] \]

is represented by the tableau.

\[
\begin{array}{cccccc}
A & B & C & D & E & F \\
\hline
a & b & x_{1} & x_{2} & x_{3} & x_{4} \\
x_{5} & b & c & d & x_{6} & x_{7} \\
x_{8} & x_{9} & c & d & e & f \\
x_{10} & b & c & x_{11} & e & x_{12} \\
\hline
a & b & c & d & e & f
\end{array}
\]

These kind of dependencies were introduced to deal with the issue
of a complete axiomatization, it is the purpose of chapter 3.
CHAPTER 3

THE IMPLICATION PROBLEM

1. General situation

Given a relational scheme $R = \{\{R_1, R_2, \ldots, R_n\}, F\}$ where the
$R_i$'s are relation schemes and $F$ a set of constraints, we say that
a constraint $f$ is a logical consequence of $F$, $F \vdash f$, if all
interpretations that are models for $F$ are the same as for $f$,
or in other terms $\text{SAT}(F) \subseteq \text{SAT}(f)$.

In examples given in chapter 2 we have illustrated different
situations where data dependencies are very useful to capture
semantic properties about data, and we have demonstrated how
the decomposition process breaks down a relation into elementary
facts. In the decomposition process the projection operation
of the relational algebra is the basic one, but a generalization
can be done in the following way.

To a relational scheme $R = \{\{R_1, R_2, \ldots, R_n\}, F\}$ we shall
associate through a mapping $\mu$ a relational schema $S$, such that
$S$ is the image of $R$ by $\mu$. The mapping can be defined by
considering a sequence of elementary mapping $\mu_i$, $\mu = (\mu_1, \mu_2,
\ldots, \mu_k)$. Each $\mu_i$ can be expressed by an algebraic relational
sentence built up from relation names $R_1, R_2, \ldots, R_n$. Formally
we denote by $S_i = \mu_i (R_1, R_2, \ldots, R_n)$ the relation scheme
obtained, and an instance of $S_i$ is $s_i = \mu_i(r_1, r_2, \ldots, r_n)$ where
the $r_i$'s are instances of the $R_i$'s. The scheme $S$ is composed of
$\{S_1, S_2, \ldots, S_k\}$, $\mu(F))$ and it can be thought as the image
of $R$, (figure 3.1).

![Mapping Scheme]

Figure 3.1: mapping scheme.

The mapping $\mu$ transforms the set of constraints $F$ into a set
of constraints $\mu(F)$ such that:

$$\text{SAT}(\mu(F)) = \{\mu(r) : r \in \text{SAT}(F)\} \subseteq \mu(\text{SAT}(F))$$

In the definition of $R$, the weakest constraint, denoted $\emptyset$
satisfies $\text{SAT}(\emptyset) = \text{REL}(R)$. A constraint over $S$ is the consequence
of the mapping $\mu$. We shall say that $g$ is a constraint over $S$
if \( g \) is a logical consequence of \( \mu(\emptyset) \), that is \( \mu(\emptyset) \models g \), or equivalently \( \text{SAT}(\mu(\emptyset)) \subseteq \text{SAT}(g) \), or \( (\text{REL}(S)) \subseteq \text{SAT}(g) \). It is clear that \( \mu(\emptyset) \) is the strongest constraint over \( S \). The studies about dependencies have two purposes:

- (i) how is it possible, inside a scheme, to know that a constraint is a logical consequence of a set of constraints?

- (ii) how is a constraint is modified or preserved by a mapping?

Theoretical results have been obtained in special cases, where the mapping is easily expressed by projections or joins.

1st case: the relational scheme \( R \) is reduced to a unique relation name, \( R \) itself, and the mapping is composed of \( k \) projections, \( \mu = (\mu_1, \mu_2, \ldots, \mu_k) \), such that \( \mu_i(r) = r[X_i] \), where \( r \) is an instance of \( R \) and \( X_i \) is a part of \( \text{REL}(R) \). The image of \( R \) is \( S = \{S_1, S_2, \ldots, S_k\}, \mu(F) \), where \( \mu(S_i) = X_i \), for \( 1 \leq i \leq k \).

A database state \( (s_1, s_2, \ldots, s_k) \) satisfies the universal relation assumption if there exists a relation \( r \) such that the type of \( r \) is the union of \( \mu(S_i) \), and the \( s_i \)'s are the projections of \( r \), \( s_i = r[X_i] \).

It is clear that in the first case the scheme \( S \) obeys the universal relation assumption and it is the strongest constraint.

A mapping is only useful when no loss of information occurs. More precisely, we must be able to reconstruct \( r \) from \( (s_1, s_2, \ldots, s_k) \), (figure 3.1). For that, it is sufficient that \( \mu \) is injective for the relations \( r \) that obey \( F \). If the mapping \( \mu \) is bijective it is called a faithful representation.

Under the assumptions that \( \mu \) is faithful and \( F \) is composed of functional dependencies, the inverse of \( \mu \), which is the reconstruction map, is the join and \( S \) is a decomposition of \( R \). Generalization of this property can be obtained for multivalued dependencies and total join dependencies (BR 79) (MMSU 80). The notion of faithful representation is too strong a condition since in many practical situations the condition is not satisfied when the user makes updates.

2d case: let \( R \) and \( S \) be relational scheme defined by \( \{R_1, R_2, \ldots, R_n\} \), \( F \) and \( (S, \mu(F)) \), where \( \mu(S) = \text{U Q}(R) = \text{U X} \), the mapping \( \mu \) is the join, that is \( \mu(r_1, r_2, \ldots, r_n) = r_1 \ast r_2 \ast \ldots \ast r_n = s \). It is clear that \( S \) obeys the join dependency:

\[ \ast [X_1, X_2, \ldots, X_n] \]

and it is the strongest constraint.

2. Axiomatization for dependencies

A lot of effort has been devoted to the study of axiomatization for dependencies. Axiomatization is important because it give us means for determining the whole set of dependencies contained
in a relational scheme. Properties of dependencies can be seen as inference rules, that is, rules that deal with the implication of new dependencies from a given set $F$ of initial dependencies.

A formal system for a family of dependencies is composed of axioms and inference rules. A derivation of a dependency $f$ from $F$, $F \vdash f$, is a sequence of dependencies $f_1, f_2, \ldots, f_n = f$, where each $f_i$ is either an axiom, or a member of $F$, or is derived from the preceding dependencies in the sequence. We shall say that an inference system is sound if $F \vdash f$ implies $F \vdash f$. It is complete if $F \vdash f$ implies $F \vdash f$.

Many sound and complete systems for dependencies have been studied. The first one has been obtained by Armstrong for the functional dependencies. Where he proved that the properties of reflexivity, augmentation and transitivity constitute a complete set of inference rules. Similar results have been obtained for multivalued dependencies, full join dependencies, and inclusion dependencies. Below, one can find inference systems for homogeneous family of dependencies.

The problem of determining a complete set of inference rule for embedded dependencies has been considered as one of the more difficult in the database theory. The first negative result has been obtained in 1982 where it has been proved that it is not possible to find such a system for the embedded join dependencies (P P 80). The work on inclusion dependencies revealed that the concept of logical consequence for dependencies under a finite interpretation must not be confused with an infinite interpretation.

**Functional dependency** (Ar 74)

FD1 (reflexivity) : if $Y \subseteq X$ then $X \rightarrow Y$

FD2 (augmentation) : if $Z \subseteq W$ and $X \rightarrow Y$ then $XW \rightarrow YZ$

FD3 (transitivity) : if $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$

**Multivalued dependency** (B P 77)

MVDO (complementation) : $X \rightarrow Y$ iff $X \rightarrow U - Y$

MVD1 (reflexivity) : if $Y \subseteq X$ then $X \rightarrow Y$

MVD2 (augmentation) : if $Z \subseteq W$ and $X \rightarrow Y$ then $XW \rightarrow YZ$

MVD3 (transitivity) : if $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$

**Multivalued dependency and functional dependency** (B P 77)

FD-MVD1 : if $X \rightarrow Y$ then $X \rightarrow Y$

FD-MVD2 : if $X \rightarrow Z$ and $Y \rightarrow W$ where $W \subset Z$ and $Y \cap Z = \emptyset$ then $X \rightarrow W$.

**Full join dependency** (AD 80 for binary join) (Th 84)

JD1 (reflexivity) : * [U]

JD2 (augmentation) : * [$X_1, \ldots, X_k$] implies * [$Y_1, \ldots, Y_k$]

if for all $1 \leq i \leq k$ there exists $l \leq j \leq m$ such that $X_i \subseteq Y_j$
JD3 (decomposition): if \([X_1, ..., X_k] \) and \([Y_1, ..., Y_m] \)
then \([Z_1, ..., Z_k, Y_2, Y_3, ..., Y_m] \) where
\[ Z_i = (X_i \cap (X_1 \cup ... \cup X_{i-1} \cup X_{i+1}) \cup X_k) \cup (X_i \cap Y_i) \]
for \(1 \leq i \leq k \).

These results are only valid for homogeneous family of dependencies. In the real world this situation is not really satisfied. In such a case a complete inference system has been only proved for functional dependencies and multivalued dependencies.

3. Degrees of cyclicity for join dependency

A full join dependency \([X_1, X_2, ..., X_k] \) belongs to the class JD-k if it is composed of k parts. We shall say that a full join dependency D has a degree of cyclicity equal to n if there exists a set \(P \) of full join dependencies belonging to the class JD-n+2 such that \(P \models D \) and \(D \models P \) (Th 84).

A full join dependency is acyclic when the degree is zero. From the definition an acyclic dependency is equivalent to a set of binary join dependencies, which is itself equivalent to a set of multivalued dependencies (FMU 82).

The concept of degree of cyclicity for join dependencies has been introduced in different way by Pagin (Fa 83) and different characterizations have been given in terms of properties of hypergraph. These definitions are generalization of the notion of Berge's cycle. For testing that a join dependency is acyclic there exists a simple algorithm (Gr 79).

Example 3.1

* \([AB, AC, BC] \) is of degree 1
* \([AB, CD, ACE, BDF, EF] \) is of degree 3.

![Hypergraphs related to join dependencies](image)

4. Algorithmic manipulations for dependencies

Due to the apparent complexity of the inferential structure, good algorithms for manipulating dependencies have not accompanied all the theoretical developments.

Among algorithmic manipulations we are considering five basic problems: dependency closure an covers, dependency membership,
keys of a relation scheme, losslessness, degree of cyclicity for a join dependency.

The dependency closure is the set of all dependencies implied by a set of dependencies. A cover for a set of dependencies is a non-redundant set of dependencies which generates the closure of that set. The concept of cover is fundamental in database design using only functional dependencies, since a cover is a synthesis of all the functional dependencies. The losslessness property for a set \( \{ X_1, X_2, \ldots, X_k \} \) is equivalent to say that \( [ X_1, X_2, \ldots, X_k ] \) is implied by a set of dependencies. This problem is related with the decomposition process.

Most of these algorithmic manipulations are based on two techniques: the boolean techniques and the tableau techniques and some of them are applied only to dependencies for which we know a complete inference system. The advantage of the boolean techniques is to provide efficient manipulations for the inferring rules in a formal system of dependencies. It is possible to establish an isomorphism between a fragment of propositional logic and a set of functional and multivalued dependencies (SDPP 81).

The tableau techniques (ABU 77) allow us to check if a dependency is implied by a set of dependencies but it does not tell us how to generate new dependencies. The concept of implication dependency introduced in section 2.5 is strongly connected with tableaux techniques.

All the theoretical studies are useful for at least three reasons:

1. It is possible to state some database design techniques on theoretical grounds. The concept of data dependencies characterizes the relationships between the objects of the real world in an application. The consistency and redundancy of a set of data dependencies can be tested precisely before deriving by a synthesis or decomposition process a relational scheme.

2. The approach presented here has the drawback that most of the theoretical results rely upon the universal relation assumption. Nevertheless, this has been useful since many researchers have studied how to use this concept for new kinds of database systems. Chapter 4 is devoted to those problems.

3. The formal inference systems developed for dependencies are special cases of mechanisms used in expert systems. In some applications of expert systems one can find problems which can be modeled with the same structure of dependencies. For example, in the university environment the prerequisite rules for a unit of course can be expressed by a functional dependency model. There exists probably analogous situations where a data dependency model could be used in an efficient way to model expert applications.
CHAPTER 4

INTERPRETATIONS AND UTILISATIONS OF THE UNIVERSAL RELATION

1. The different interpretation

The universal relation assumption has been introduced in many works where the motivations did not appear clearly at the beginning. One of the first motivations is related to the necessity of deriving meaningful data dependencies. For example, consider a relational scheme over the attribute : Student, Unit of Course, and Teacher, where \( R = (R_1, R_2, F) \) with \( \delta(R_1) = \{S, U, T\} \) and \( \delta(R_2) = \{S, T\} \). The relation scheme \( R_1 \) means that a student takes a course given by a teacher, while \( R_2 \) means that a student is supervised by a teacher. If the constraints \( F \) are of the form: a student takes only one course, a course has only one teacher, we can write \( S \rightarrow U \rightarrow T \) in \( R_1 \), and \( S \rightarrow T \) in \( R_2 \).

It is clear that the dependency \( S \rightarrow T \) in \( R_2 \) is not redundant with \( S \rightarrow T \) in \( R_1 \) obtained by transitivity. In such a case the relationship between the attributes \( S \) and \( T \) has two different meanings and the universal relation assumption is not satisfied. The only way to cure this problem is to introduce new names for the attributes. Instead of \( T \) in \( R_2 \), we can define a new name for the supervisor \( (V) \). We must say also that the supervisors are teachers which can be expressed by an inclusion dependency.

In database scheme design, the goal is to built a database scheme which can be explained from the inputs. The implicit framework is the following sequence:

\[ \text{Input Scheme} \rightarrow \text{Universal Scheme} \rightarrow \text{Output Scheme} \]

where the universal scheme is a central view where all the requirement analysis and constraints have been integrated. Every relation of a universal scheme is necessary redundant. For many reasons : redundancy and update anomalies it is useful to break down a universal relation into small parts as explained earlier. The principal activity in database design is the decomposition of the universal into a database scheme that has some properties expressed in normal forms.

The third motivation is to offer to users an easy manipulation language. One of the main goal introduced by Codd with the relational model is to provide a high level language where programmers specify what they want without telling the access paths to the data. First order languages have this property, but nevertheless programmers must specify the logical connections between relations. Systems built with the universal relation assumption can remove this fact and provide an easy programming interface. Nevertheless, it introduces new kinds of problems.

For example, consider a relational scheme over two relation schemes PC (Parent, Child) and CT (Child, Toy). If we are interested in the relationship between parent and toy, we can built a relation
over PT by writing \([\text{PT}] = (\text{PC} \ast \text{CT}) \cdot \text{PT}\). If a universal scheme exists over PTC, to obtain the same result we need only a projection over the attributes PT. This is the main advantage of the universal relation where the easy requests can be expressed by projection or selection operations. We don't have any guarantee that the result delivered by the universal scheme has the correct meaning that we assign to the request. In fact, a universal relation is always a view of a conceptual database, for example the universal relation PTC is equal to PC \ast CT. Thus, a query like "give me information about parent and toy" is interpreted as a projection of the universal relation if we have in mind that we want a toy related to a parent such that there exists a child connected with the toy and the parent. In this case the result of the universal relation is correct. But if we assume that the children can be considered as parents, then it is not obvious when building a relation over PT that we want a toy related to the parent through a child or a toy related to the parent of the parent, etc... The user might have in mind something other than the simplest relationship. Another example illustrates a different kind of ambiguity with the universal relation.

Example 4.1. Let \(R\) be a relational scheme, \(R = (R_1(\text{Customer, Account}), R_2(\text{Customer, Loan}), R_3(\text{Loan, Bank}), R_4(\text{Account, Bank}))\) where \(R_1, R_2, R_3\) and \(R_4\) can be interpreted as: a customer has an account, a customer has a loan, a loan is in a bank, an account is held by a bank. If we define a relation scheme \(S(C,A,L,B)\) where every instance \(s\) of \(S\) is the join of \(r_1, r_2, r_3\) and \(r_4\), \(s = r_1 \ast r_2 \ast r_3 \ast r_4\).

The relation \(s\) is a universal relation if the projections of \(s\) over \(CA,CL,LB\) and \(AB\) are equal to \(r_1, r_2, r_3\) and \(r_4\). If we are interested in the relationship between a bank and a customer there are two possible connections (figure 4.1).

![Figure 4.1: the customer-bank-loan-account scheme.](image)

- a bank is connected to a customer through an account,
- a bank is connected to a customer through a loan.

An user querying a universal relation about the relationship between bank and customer sends to the system a query of the form \(s[BQ]\). A pair \((bc)\) is returned if the customer \(c\) holds both a loan an account in the bank \(b\). If the user had in mind "give the set of
pairs \((b,c)\) such that the customer \(c\) holds a loan in the bank \(b\) or \(c\) holds an account in the bank \(b\)" then the projection of the universal relation over \(BC\) is not the correct answer. To satisfy this case the appropriate relational expression for the universal relation should be \(s = (r_1 \times r_4) + (r_2 \times r_5)\).

The graph representing the relational schema \(R\) (figure 4.1) is cyclic, and the fact mentioned with the relationship between Bank and Customer is related to the existence of a cycle. Before discussing and presenting some universal models, let us consider the basic assumptions illustrated in the examples presented above in this section.

1. **Unicity of attribute "role"**

   In a universal relation each attribute plays a unique role. Thus an attribute like DATE cannot stand for birthdate, hiring date, ... . Before building a universal relation it is necessary to rename attributes which play different roles.

2. **Unicity of relationship**

   This assumption can be seen as a corollary of the previous assumption. If we are interested in a relationship between the attributes \(X\), there is only one meaning related to \(X\) which is embedded in the universal schema. The meaning of the \(X\)-relationship depends on the relational expression used to built the universal relation.

3. **Union compatibility for relationships**

   In the example 4.1 we have seen that the relationship between the \(X\)-attributes can be evaluated by different logical connections. Which one should be used, or what combination should be considered? A universal relation favors one of these solutions, and when querying the universal relation the user should have in mind this knowledge. The solution taken by universal relation models is that all the elementary facts about the \(X\)-connections represent the same flavor, this means that all the elementary facts can be merged in the same box.

   The two basic problems with universal relation models are:

   1. How to built from a relational scheme
      \[ R = (R_1, R_2, \ldots, R_n, F) \] and a database state \(r = (r_1, r_2, \ldots, r_n)\), a representation of the universal relation which preserves the information content of the database and satisfies the constraints in \(F\)?

   2. Assuming that a universal representation exists, how is it possible to built the answer for a relationship over the attributes \(X\)?

   If the scheme \(R\) satisfies the universal relation assumption, then the representation is the universal relation denoted by \(s = r_1 \times \ldots \times r_n\), and every request over the association \(X\) will be evaluated to \([X] = s [X]\). The main difficulty is to know if
a database state obeys the universal relation assumption. Testing this property is \textsc{NP}–complete (HLY 80). If the relational schema \( R \) is acyclic the problem is simplified since we need only to take relations two by two (BFMY 83).

The universal relation assumption is a very strong assumption and it is not possible to handle the case where an update does not maintain the assumption. We outline here one approach to remove this problem, called the \textit{weak universal relation approach}.

2. The \textbf{weak universal relation assumption}

Let \( R = (R_1, R_2, \ldots, R_n, F) \) be a relational scheme, where 
\( \beta(R_i) = X_i \) for \( 1 \leq i \leq n \), we shall say that a database state 
\( r = (r_1, r_2, \ldots, r_n) \) obeys the weak universal relation assumption (Ho 82) if there exists a relation \( u \) of type \( U = U X_i \), such that:

- (i) \( r_i \subseteq u[X_i] \), for \( 1 \leq i \leq n \),
- (ii) the relation \( u \) satisfies the constraints \( F \).

From this viewpoint, a database is seen as a partial specification of the universal relation \( u \). The definition of \( u \) is existential in nature and does not contain any operational procedure to built such a relation. Furthermore, there might exist an infinite set of relations \( u \) which satisfy the assumption. Among this infinite set it is impossible to know which of the possible ones actually represent the truth. For this reason, the state of a relationship \( X \) between attributes, denoted \( [X(F)] \), is therefore taken to be:

\[
[X(F)] = \{ u[X] : u \text{ is a weak universal relation} \},
\]

\( [X(F)] \) is defined from the information contained in tuples \( t \) over \( X \) such that there exists a tuple \( t' \) belonging to a weak universal relation and \( t[X] = t'[X] \). From another point of view, we can say that the answer \( [X(F)] \) is deduced from the database state \( r = (r_1, r_2, \ldots, r_n) \) and from the constraints \( F \) seen as inference rules.

It is for this reason that the notation used for the answer \( [X(F)] \) is with respect to \( F \). A weak universal relation can be seen as a deductive approach to databases.

The major problem is to build a single relation, which may contain incomplete information, and which embodies the information present in all weak relations. This relation is called the representative instance of the weak universal relation assumption.

How to built a representative relation (Ho 82) (Sa 81) ?

The process is divided into four steps as follows:

1. We begin to build a relation of type \( U \) by picking successively a relation \( r_i \) in the database, and for each tuple \( t \) in \( r_i \), we insert \( t \) in the relation according to the place of each
t-attribute, and for each attribute in U-Xᵢ we insert a null-value, denoted # followed by a marked place subscript i.

2. We generate new tuples by the tableau technique according to the constraints in F. In practice, in this approach the set of constraints is a set of data dependencies. In the case of functional dependencies we equate null symbols with non null symbols, if possible; if we equate null symbols we replace the null symbol having the highest subscript by the lowest.

3. For certain kinds of dependencies, in particular embedded dependencies, we introduce new nulls and the process may not terminate.

4. The process ends in two cases: the chase terminates correctly without infinite loop or we are forced to equate two distinct non-null symbols. In this case, there is no representative instance of all the weak universal relations.

If we succeed in the process, the fundamental property about the representative relation is that it always produces the intersection of all weak universal relations. The answer \([X(F)]\) is therefore taken to be: \(\text{rep}[X]\), where \(\text{rep}\) denotes the representative relation.

Example 4.2: let \(R\) be relational scheme defined by
\(R = \{(CS, CSG, CP), \{*[CSP, CP], *[CS, CP]\}\)
where \(C, S, G, P\) stands for Class, Student, Grade and Professor, the meaning of the relationship is respectively: a student is attending a class, a student in a class has a grade, a class has a professor. A database state is given by the following tables of values:

<table>
<thead>
<tr>
<th>C</th>
<th>S</th>
<th>C</th>
<th>S</th>
<th>G</th>
<th>C</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>math</td>
<td></td>
<td>john</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>english</td>
<td></td>
<td>john</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>english</td>
<td>bill</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

the representative instance is:

<table>
<thead>
<tr>
<th>C</th>
<th>S</th>
<th>G</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>math</td>
<td>john</td>
<td>#1</td>
<td>#2</td>
</tr>
<tr>
<td>english</td>
<td>bill</td>
<td>#3</td>
<td>#4</td>
</tr>
<tr>
<td>english</td>
<td>john</td>
<td>#5</td>
<td></td>
</tr>
<tr>
<td>math</td>
<td>#6</td>
<td>#7</td>
<td>ronald</td>
</tr>
<tr>
<td>english</td>
<td>#8</td>
<td>#9</td>
<td>mike</td>
</tr>
</tbody>
</table>

| math | john| #10| ronald |
| english | bill| #11| mike  |
| english | john| a  | mike  |
It we introduce between tuples a partial order relation defined as follows : \( t \preceq t' \) if for all attribute \( A \) in \( U \) we have \( t[A] = t'[A] \) or \( t[A] = \# \). Consequently for relations \( r \preceq r' \), if \( (t) \in r \), \( (E_t') \in r' \) such that \( t \preceq t' \). With this partial order relation we can limit the representative instance to only maximal tuples.

<table>
<thead>
<tr>
<th>math</th>
<th>john</th>
<th>#10</th>
<th>ronald</th>
</tr>
</thead>
<tbody>
<tr>
<td>english</td>
<td>bill</td>
<td>#11</td>
<td>mike</td>
</tr>
<tr>
<td>english</td>
<td>john</td>
<td>a</td>
<td>mike</td>
</tr>
</tbody>
</table>

How to calculate \( \text{rep}[X] \)?

The relation \( \text{rep} \) may be obtained in exponential time since the chase is of the same nature. Then the evaluation of \( [X(F)] \) from \( \text{rep}[X] \) is not necessarily an operational solution.

The other approach taken is to look for a relational expression \( E \) whose operands are relations schemes \( R_1, R_2, \ldots, R_n \) and whose operations are : join, projection, union, difference, selection, such that:

\[
E(R_1, R_2, \ldots, R_n) = \text{rep}[X]
\]

\[\text{or} \quad E(R_1, R_2, \ldots, R_n) \subseteq \text{rep}[X]\]

If this happen we can say that the expression \( E \) simulates the evaluation of \( \text{rep}[X] \). To obtain such a result the research works have explored different properties of the expression \( E \):

- monotonicity
- losslessness.

Example 4.3. We illustrate the losslessness property on the example 2.2 presented at the beginning. Let \( ([\text{APIC}], [\text{*PI, AP, AC, PC}] \)

\( P \rightarrow I') \) be a relation scheme and \( X = [\text{IA}] \) the relationship between Agent and Price. The algebra expression

\[(\text{PIC} \ast \text{PA} \ast \text{AC}) [\text{IA}]\]

is lossless.

To test the property we proceed as follows : we built a tableau expression where the first \( t_1 \)'s are the hypothesis rows and the last is the conclusion row. The components of these rows are abstract symbols representing attribute variable like in an implicative dependency (see section 2.5). The expression will be lossless if by some derivation rules the conclusion is obtained.
from the hypothesis. The following tableau is related to the expression:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>P</th>
<th>I</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>i</td>
<td>c</td>
<td>(PIC)</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>p</td>
<td>(PA)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>(AC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Informally, these tableau can be read as: the pair (a, i) is in the answer if there exists three tuples of the form (p, i, c), (a, p) and (a, c) belonging respectively to PIC, PA, and AC. If we apply the join dependency *[PI, AP, AC, PC] we derive a tuple (a, p, i, c) and by projection (a, i). Then (a, i) is derived from the hypothesis, thus the expression is lossless.

The expression (PIC*PA) [IA] leads to the same result as shown by the tableau:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>P</th>
<th>I</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>i</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>p</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

in these case the sequence of derivation is not the same. The functional dependency P \(\rightarrow\) I implies that the entry (ap) generates (a p i), and by projection (a, i).

The example demonstrates that different expressions lead to the same evaluations but the computational complexity is not the same since the number of joins to compute is not the same.

An expression E is monotone, if \(r_i \subseteq r_i'\) (i=1,n), then \(E(r_1, r_2, \ldots, r_n) \subseteq E(r_1', r_2', \ldots, r_n')\), where the \(r_i\)'s and the \(r_i\)'s are instances of the relation schemes \(R_1, R_2, \ldots, R_n\). The basic algebra operations: join, selection, projection and union are monotone.

Under the properties of monotonicity and losslessness we can obtain the following fundamental result (MUV 84): let \(E\) be an expression that is monotone and lossless with respect to some set of implicational dependencies \(F\) and produces a relation over \(X\), then \(E \subseteq \text{rep } [X]\). The equality can be obtained by some other characterizations.
3. Maximal objects

Example 4.1 has shown intuitively that when the scheme is not acyclic, the interpretation of the relationship between attributes can be confused; in general several access paths are suitable for computing this connection. To avoid this difficulty one can introduce the notion of maximal objects (MU 80). A maximal object is a subset of relation schemes which inside the maximal object we are able to navigate without ambiguity, or one can say that the scheme of a maximal object is acyclic. The set of all the maximal objects covers the database scheme. In example 4.1 we can consider two maximal objects:

\[ M_1 = \{AC, AB\} \]

\[ M_2 = \{CL, LB\} \]

For each maximal object we take the join of all the relations belonging to it. And after we extend the result by introducing null values to all attributes which are not in the maximal object. The result obtained is a relation of type \( U \), denoted \( u(M) \).

The representative instance in the maximal objects model will be the union of the relations of the form \( u(M) \) for all \( M \). In our example \( u(M_1) = AC*AB \), \( u(M_2) = CL*LB \), and consequently \( u = u(M_1) + u(M_2) \).

A query over the relationship between the X-attributes will be evaluate to:

\[ [X] = \bigcup_{M \in X} u(M) [X] \]

If the set \( X \) of attributes is not contained in a maximal object then there is no answer. In this case one can consider that some attributes are so semantically distant that no connection among them can reflect a normal situation.

Several systems support a universal view of data based on this model: "SYSTEM U" (KU 80) and "FITS" (MRSSW 82). In practice, the query manipulation languages related to the universal view are very simplified. They permit queries with several tuple variables, each of which may range over a separate copy of the universal relation, where the two basic operations are selections and projections.

4. A virtual universal relation

This approach has been introduced by (BB 83) and extended in (Ve 83). In some ways, it is similar to the preceding ones presented here. The goal is to built a representative instance of the universal relation which is called the virtual universal relation. The process for defining the virtual relation and the basic parameters for the data model are different. The virtual universal data model is based on the following basic concepts:

- objects are sets of attributes and represent elementary facts or units of update (Sc 80),
associations are connections between objects, irredundant scheme.

We shall illustrate this model with an example which is the program of movies in a city for a given week. One can consider the set of attributes:

- \( A \): actor,
- \( M \): movie,
- \( T \): movie theater,
- \( P \): price,
- \( S \): schedule.

An object is a subset of attributes related to an elementary fact. The object (MTP) characterizes facts like: "the price to pay to see the movie "the specialists" in the movie theater "the gaumont" is 30 francs". Also a fact like "the movie the specialists is presented in the gaumont theater" is associated with an object. Every application is modelled by a set of objects, intuitively for our example we have:

\[
\text{OBJECT} = \{ \text{AM, MT, MTP, MTS, MS, TP, MTPS, AMT, AMTP, AMTPS} \}
\]

AT is not an object since there is no elementary sentence where A and T are related. The relationship between these two attributes implies the existence of a movie: there exists a movie \( m \) such that the actor \( a \) plays in the movie \( m \) and the movie \( m \) is played in the movie theater \( t \). If \( X \) is an object, and \( Y \) is included in \( X \), then \( Y \) is not necessary an object, for example, MTS is an object and TS is not an object.

The set of objects defines the set of elementary facts which are true in the database. A fact \( x \) over an object \( X \) is a tuple, that is an application between \( x \) and the values taken in the domain of \( X \). Certain facts may be considered as elementary and others are derived by inference rules from the elementary facts. If the actor "giraudet" plays in the movie "the specialists" and "the specialists" is played in the movie theater "the gaumont" we are able to deduce that "the gaumont" plays the specialists movie where giraudet is an actor of the movie.

A relation, called \( \text{JOIN} \), defines the connections that the user considers as acceptable in its application (Ve 83). An element of \( \text{JOIN} \) is a pair of objects \((X,Y)\) which can join to derive a new object. The relation \( \text{JOIN} \) has the following properties:

- symmetry
- if \( X \subseteq Y \) then \((X,Y) \in \text{JOIN} \)
- if \((X,Y) \in \text{JOIN} \) there \( XY \) is an object,
- if \((X,Y) \in \text{JOIN} \) and \( Y \subseteq Z \) and \( Z \) is an object then \((X,Z) \in \text{JOIN} \).

For example, the relation \( \text{JOIN} \) contains at least two elements \( \{(\text{AM, TM}), (\text{MTP, MTS})\} \). We shall note that MS and AM are not connected
since the connection needs to know the attribute T. The relation
JOIN is tailored by the database designer and establishes all
the connections which are semantically useful for the user. In
(BB 85) it is assumed that two objects are connected until their
intersection is not empty. The relation JOIN permits these case.
This can be the case in our example if we add the attribute D
(day) by saying that the movie schedule for one day is the same
for all days of the week, thus the pair (D,AMTPS) is an element
of the JOIN relation. There is some situations where the model
needs to be improved. Assume that the elementary facts over MTP
can be obtained either from MTP directly, or from the objects
MT and TP. This case is related to the fact that some movies theater
have a unique price for all the movies and some others have a
different price for each movie. Note that we use the union
compatibility assumption for the relationship over MTP.

One can conceive a model where an association between the attributes
X will be defined from objects Y₁, Y₂, ..., Yₖ and by a relational
expression. A virtual view of the user will be built (Za 77).

In the various approaches presented, the basic idea is to built
a representation instance of the universal relation as a table
of values where the information contained in the table is composed
of elementary facts or derived facts using the join operation
or join dependencies.

The fact that we use only the join operations is a limitation
for the system. In practice some other kinds of deduction can
be made. Assume that we have the following facts :
- the boat b is tagged to the peer p in the city c,
- the water depth at the peer p is t

then we are able to conclude : the draught of water of the boat
b is less than t. 
Most of the models in the universal relation do not modelize this
situation.

Let x and y be two elementary facts, respectively over the objects
X and Y. We shall say that x and y are joinable and produces a
new fact x*y defined by :
(x*y) [X] = x and (x*y) [Y] = y if the pair of objects (X,Y)
belong to the relation JOIN.
A virtual relation r will be constructed by the following process:
all the elementary facts will be inserted in r as tuples, eventually
using a null values denoted by a blank. The closure of r, r*,
contains all the elementary facts that we can deduce from r
according to the JOIN relation.
The two bottom lines in the table are derived. If we use the partial order relation between tuples defined in example 4.2 we can restrict ourself to the maximal objects, denoted \( \text{max}(r) \).

Another important feature of the virtual universal relation model is the concept of generator scheme. A generator scheme is a set of objects which satisfies the following properties:

(i) - \( S \) is a subset of \( \text{OBJECT} \),

(ii) - for each object \( X \), every facts over \( X \) can be inferred from facts over objects included into \( S \),

An irredundant generator scheme is a generator scheme such no proper subset is a generator. The set \( S = \{ \text{AM}, \text{MT}, \text{MTP}, \text{MTS}, \text{TP}, \text{MS} \} \) is an irredundant generator for the movie application. The notion of generator scheme is also related to the notion of atomic facts.

The set of atomic facts can be obtained by projecting the virtual relation \( r \) over the generator scheme, the projection will be denoted \( r[S] \) and defined by:

\[
r[S] = \{ t : \exists \theta(t) \in S \text{ and } (E \in r)(s \triangleright t) \}.
\]

A relation \( r \) containing eventually null values, is said to be lossless with respect to \( S \) iff \( r = (r[S])^\ast \). This definition is a generalization of the decomposition of a relation over a set of attributes, and we have the following proposition (Ve 83): if \( r \) is a relation without null values, then \( r \) obeys the join dependency \( [S] \), where \( S \) is a generator. The converse is false, this means that if a relation \( r \) obeys a join dependency the facts that we can deduce by the closure of \( r \) contain the facts derived by the join dependency. It is possible to find some characterizations where the lossless property is equivalent to the join dependency.
CONCLUSION

Theoretical works in database systems has been concentrated around the relational data model. The studies form a fundamental basis for the development of homogeneous capabilities. Thus, the first order languages provide an unified view for data manipulation languages and integrity constraints. On another hand, data dependencies are formal efforts to capture semantic properties of data used in database design methodologies (KTY 79, KTT 83). The theory of normal forms, which has not been presented in the paper, deals also with database design techniques and is a complement to the notions of: relation decomposition, universal relation assumption, join dependency, and acyclic scheme. Today, all these elements are major contributions of theoretical computer science to study the fundamental problem of scheme equivalence.

The history of database systems has shown that data independance is one of the major goal to achieve. Relational systems are part of the evolution with non procedural data manipulation languages. Systems supporting the universal relation assumption go still further in this direction.

BIBLIOGRAPHY


(AD 80) Armstrong W. Delobel C., Decompositions and functional dependencies in relations, ACM TODS, 5.4 1980.


(CO 70) Codd E., A relational model of data for large shared data banks, CACM, 13.6, 1970.

(Da 81) Date C., Referential integrity, Proc. 7th VLDB, 1981.

(De 73) Delobel C., Contributions théoriques à la conception d'un système d'information, Thèse d'Etat, University of Grenoble, 1973.


(Dem 82) Demolombe R., Utilisation du calcul des prédicats comme langage d'interrogation des bases de données, Thèse d'Etat, University of Toulouse, 1982.


(Sa 81) Sagiv Y., Can we use the universal instance assumption? Proc. ACM SIGMOD, 1981.


(Ve 83) Verroust A., Les schémas bien formés et opérations de
mise à jour, Thèse 3ème Cycle, University of Orsay, 1983.

(Th 84) Thalheim B., A complete axiomatization for join dependencies
in relations, Technical Report, University of Dresde,
1984.

(Za 77) Zaniolo C., Relational views in a database system support
for queries, IEEE Comp. Soft. and Applications Conf.,