FOUNDATIONS OF KNOWLEDGE SYSTEMS
with Applications to Databases and Agents

GERD WAGNER
Institut für Informatik, Universität Leipzig,
http://www.informatik.uni-leipzig.de/~gwagner

Kluwer Academic Publishers
Boston/Dordrecht/London

Contents

One of the main uses of computer systems is the management of large amounts of symbolic information representing the state of some application domain such as information about all the people I communicate with in my personal address database, or about relevant parts of the outer space in the knowledge base of a NASA space mission. While database management systems offer only the basic services of information manipulation and retrieval, more powerful knowledge systems offer in addition advanced services such as deductive and abductive reasoning for the purpose of finding explanations and diagnoses, or generating plans.

In order to design and understand database and knowledge-based applications it is important to build upon well-established conceptual and mathematical foundations. What are the principles behind database and knowledge systems ? What are their major components ? Which are the important cases of knowledge systems ? What are their limitations ? Addressing these questions, and discussing the fundamental issues of information update, knowledge assimilation, integrity maintenance, and inference-based query answering, is the purpose of this book.

Databases, Logic and SQL

Since relational databases are the most fundamental case of a knowledge system, they are treated in-depth in connection with the standard database language SQL. The chapter on the foundations of relational databases is also both an introduction to mathematical logic, and a course in SQL programming. It is supplemented with an additional chapter on *object-relational* databases, a new development that dominates the current progress in database technology and SQL.

Various Kinds of Information

Depending on the type of application domain, we have to be able to deal with various kinds of information. The simplest case are business and administrative domains where the information to be processed is complete, and it is assumed that all information sources are perfect (absolutely reliable, honest and competent). But in more empirical domains, such as in outer space, even if all information sources are perfect, there will be various forms of incomplete information, e.g. involving partial predicates, and disjunctively imprecise or gradually uncertain information. In addition to incompleteness, information items may be asserted relative to some time spans in which they are valid and believed, and they may be qualified in a number of ways. In particular, different pieces of information may be assigned different levels of security, and different degrees of reliability.

The book presents a comprehensive framework for the conceptual integration and uniform treatment of all these types of information. This includes the definition of generic knowledge system models capable to handle fuzzy, temporal, confidential, and unreliable information.

Various Types of Rules

Advanced knowledge services (such as deductive query answering, reactive input processing, and the abductive generation of explanations, diagnoses and plans) are based on rules. The book presents an in-depth treatment of two kinds of rules: deduction rules and reaction rules.

In addition to information about the current and possible state of affairs in the form of facts and integrity constraints, a knowledge base may also represent terminological and heuristic knowledge in the form of deduction rules. This type of rule is well-known from deductive databases and logic programs. It is shown how the concept of deduction rules can be applied to the various kinds of information that have to be processed in practical knowledge systems, leading to such powerful concepts like temporal and fuzzy deduction rules. Typically, the semantics of these rules involves nonmonotonic inference on the basis of the Closed-World Assumption.

Reaction rules can be used to specify the reactive behavior of a knowledge base when it receives messages, either from the external world, or from other nodes of a network it participates in. This concerns, in particular, the communication and cooperation taking place in multidatabases and between cooperative knowledge bases.

How to Use this Book

Foundations of Databases and Knowledge Systems covers both basic and advanced topics. It may be used as the textbook of a course offering a broad introduction to databases and knowledge bases, or it may be used as an additional textbook in a course on databases or Artificial Intelligence. Professionals and researchers interested in learning about new developments will benefit from the encyclopedic character of the book providing organized access to many advanced concepts in the theory of databases and knowledge bases.

Gerd Wagner
Berlin, April 1998

To Ina.

Many thanks to Heinrich Herre and Michael Schroeder for their cooperation and the stimulating exchange we had during my work on this book. I am also grateful to Luis Farinas del Cerro for inviting me to Toulouse in 1994, and to Luis Moniz Pereira for inviting me to Lisboa in 1995. At both locations, I enjoyed very much the excellent working environment and the kind of freedom one may find when beeing abroad.

I thank Michael Gelfond, whose work has strongly inspired and influenced me, for discussing various issues in logic programming and knowledge representation with me, and for his hospitality. I thank Robert Kowalski for reading my Habilitation thesis and providing valuable comments.

I am grateful to Jorge Lobo, Michael Schroeder and Agnes Voisard for reading parts of the manuscript and pointing to weaknesses and errors. I also thank Jürgen Huber for proof-reading parts of the manuscript.

Finally, I want to express my gratitude to the Deutsche Forschungsgemeinschaft which has funded my research.

I.1 THE SUCCESS OF DATABASE SYSTEMS

In the sixties and seventies, pushed by the need to store and process large data collections, powerful database software based on the file system technology available at that time has been developed. These types of systems have been named *hierarchical* and *network* databases, referring to the respective type of file organization. Although these systems were able to process large amounts of data efficiently, their limitations in terms of flexibility and ease of use were severe. Those difficulties were caused by the unnatural character of the conceptual user-interface of hierarchical and network databases consisting of the rather low-level data access operations that were dictated by their way of implementing storage and retrieval. Thus, both database models have later on turned out to be cognitively inadequate.

Only through the formal conceptualization of *relational databases* by Codd in the early seventies it became possible to overcome the inadequacy of the database technology prevailing at that time. Only the logic-based formal concepts of the relational database model have led to more cognitive adequacy, and have thus constituted the conceptual basis for further progress (towards object-relational, temporal, deductive, etc. databases). Unlike much of theoretical computer science, Codd's theory of relational databases is an example of a practical theory.

I.2 THE DIFFICULTIES OF 'EXPERT SYSTEMS'

In the field of expert systems, on the other hand, there has never been a conceptual breakthrough like that of the relational database model. It seems that the field is still at the pre-scientific stage, and there seems to be almost no measurable progress since the classical prototype MYCIN. There is neither a formal definition of an 'expert system', nor is there any clear semantics of rules. The field has rather developed a variety of notions (such as 'production rules,' or 'certainty factors') which have often not been clearly defined, or have not been based on a logical semantics. Lacking a clear (and formally defined) conceptual framework, it is difficult for the expert system community to compare different systems and to identify shortcomings and measure progress.

One may argue that these problems are due to the inherent difficulties of knowledge representation and processing, but it rather seems that the expert system community has failed to establish scientific foundations and has also failed to learn from its neighbor fields of logic programming and deductive databases which have successfully developed a formal concept of rules based on a logical semantics.

I.3 WHAT IS A KNOWLEDGE SYSTEM ?

Knowledge representation and reasoning systems, or shorter: *knowledge systems* are based on two fundamental operations[1]:

1. New incoming pieces of information must be *assimilated* into a knowledge base by means of an **update operation**. This includes simple insertion of new information, deletion (retraction) of old information, and possibly a number of more involved change operations for assimilating new pieces of information which are in conflict with some already-stored pieces. The update operation should satisfy some principle of *minimal mutilation*.

2. Queries posed to the knowledge base must be answered by means of an inference-based **query answering operation**. The inference procedure should be complete with respect to some well-understood logic, and sound with respect to a preferential entailment relation in that logic.

In general, there are no specific restrictions on the internal structure of a knowledge base. It appears, however, that a computational design can be achieved by 'compiling' incoming information into some normal form rather than leaving it in the form of complex formulas.

The concept of a knowledge system constitutes a useful framework for the classification and comparison of various computational systems and formalisms like, e.g., relational, temporal and deductive databases. It is more general than that of a logic (i.e. a consequence relation). A standard logic can be viewed as a special kind of knowledge system. On the other hand, by suitably defining the inference and update operations, knowledge systems can serve as the basis for the operational definition of logics.

I.4 KNOWLEDGE SYSTEMS AND LOGIC

The relationship between knowledge systems and logic is only obvious for the inference operation: while in a logical system theorems are proved from (or entailed by) a set of axioms (or *theory*) by means of the underlying consequence relation, queries in a knowledge system are answered relative to a given knowledge base by means of the underlying inference operation. It turns out that knowledge-based inference is nonmonotonic: query answering is not based on all models of a knowledge base but solely on the set of all intended models. For instance, in the case of relational databases, which can be viewed as the most fundamental knowledge system, the intended models are the minimal ones. Also, unlike a logical theory, a knowledge base is not just a set of well-formed formulas. Rather, it is subject to specific restrictions (such as, e.g., allowing only atomic sentences as in relational databases), or more generally, consisting of *facts*, *rules* and *integrity constraints*. While facts have the form of certain sentences of some suitably restricted language, rules do, in general, not correspond to implicational formulas, but rather to meta-logical expressions requiring a specific semantics. Integrity constraints are sentences which have to be satisfied in all evolving states of a knowledge base. They stipulate meaningful domain-specific restrictions on the class of admissible knowledge bases. In addition to information about the current and possible state of affairs (in the form of facts and integrity constraints), and terminological and heuristic knowledge (in the form of deduction rules), a knowledge base may also represent *action knowledge* (e.g., in the form of reaction rules), and various types of meta-knowledge.

Logic has provided a model of theory-based reasoning by means of consequence relations, but it has not been concerned with the assimilation of new sentences into a changing theory. For a theory of knowledge systems, however, we need a model of both knowledge-based inference and of knowledge assimilation.

I.5 ABOUT THIS BOOK

Traditionally, books on 'expert systems' and *knowledge-based systems* have been composed of a loose collection of concepts and techniques from Artificial Intelligence and related fields, such as pattern matching, search methods, 'rule-based systems', models of temporal, spatial and uncertain reasoning, and their application in diagnosis, design and control. A great number of prototype systems has been described in the literature, but there is hardly any generic concept of knowledge-based systems. On the other hand, *knowledge representation research* in Artificial

Table I.1. Knowledge systems and the deficiencies of logic.

Knowledge Systems	*Logical Systems*
facts and deduction rules	axioms
knowledge base	theory
nonmonotonic inference	monotonic consequence relation
confirmed if-queries	theorems
open queries	?
answers	?
update operation	?
integrity constraints	?
reaction rules	?

Intelligence has proceeded in a very theoretical manner and has been occupied to a great extent with solving "anomalies within formal systems which are never used for any practical task" (Brooks, 1991). Many of the formalisms proposed make, or follow, conceptual and ontological stipulations which are not grounded in the practice of information and knowledge processing.

The theory of knowledge systems presented in this book should be construed as an attempt of a *practical theory* which aims at establishing solid conceptual foundations and generic models to be used in the software engineering of knowledge and information systems. It is my firm belief, that knowledge systems have to evolve from, and have to be integrated with, database and information system technology. The starting points in this evolution are the *entity-relationship modeling* method and the system of *relational databases* which are presented in Chapter 1 and 2. These and the following chapters on object-relational databases, reaction rules, deduction rules, temporal databases and fuzzy databases are intended for class room use in computer science at the graduate levels. More advanced topics, such as representing negative information using *bitables*, or elements of a general theory of deduction rules, are covered in the remaining chapters.

The main technical contributions of this book are:

1. a formal model of object-relational databases;
2. a general account of *reaction rules* in connection with the concept of *knowledge- and perception-based agents*, originally proposed in Wagner, 1996b;
3. a new semantics of deduction rules that is based on *stable generated closures* and derived from the notion of *stable generated models* of Herre and Wagner, 1997;
4. a new generalization of database tables, called *bitables*, for representing negative information in connection with incomplete predicates (the logical ideas of this generalization have already been presented in Wagner, 1991);
5. a new model of *fuzzy* and *possibilistic databases* based on a new compositional definition of *possibilistic logic*;
6. a new definition of inference-based query answering in *multi-level secure databases*; and
7. a new model of *lineage databases* for tracking the lineage and reliability of information items.

The main contributions to a general perspective of knowledge representation and reasoning consist of

1. the interpretation of databases and knowledge bases as special data structures and to define query answering on the basis of a *natural nonmonotonic inference relation* between these data structures and query formulas; and

2. a generic model of *knowledge systems* that emphasizes the significance of the paradigm of relational databases.

The integration and uniform treatment of all kinds of databases and knowledge bases in a rigorous conceptual framework appears to be the greatest achievement of the present work. It allows to identify generic concepts such as deduction and reaction rules, and to define them in a way that is independent of the particular choice of knowledge system.

The knowledge system concepts proposed in this book have obvious applications in advanced database and in agent-based systems. An agent system needs to include a *knowledge system* as one of its main components. In particular, the knowledge systems of *MLS factbases* for protecting confidential information, and of *lineage factbases* for tracking and weighting the reliability of different information sources, may be relevant to agent systems. Even more important is the concept of *reaction rules* that allows the declarative specification of the reactive behavior of agents. This book is for those who want to design knowledge-based agents, and it is for those who want to develop innovative database systems. It can be used both by computer science students and by professionals in computer science and other fields with some background in logic to obtain organized access to many advanced concepts in the theory of databases and knowledge representation.

Those sections that are primarily of interest to logic-oriented readers are indicated with pure λogic in the section head. Additional non-classical logic background, not required for understanding the book, is provided in the appendix as an option for the logic-oriented reader to gain deeper insights into the connections between knowledge systems and logic.

This is probably the first book to cover such a broad range of diverse database and knowledge systems. Necessarily, not all relevant questions are addressed and the discussion of certain advanced topics may be incomplete, biased, or even erroneous. As a human with limited resources, I may have overlooked important issues and ignored important related work I should have mentioned. I apologize for this and promise to do my best to provide necessary corrections and supplements on the book's Web page which is

http://www.informatik.uni-leipzig.de/~gwagner/ks.html

Notice that this Web page may also contain solutions to exercises and other supplementary material which could not be included in the book.

Notes

1. This distinction was already proposed in Levesque, 1984, where the resp. operations were called *ASK* and *TELL*.

I Tables and Objects

1 CONCEPTUAL MODELING OF KNOWLEDGE BASES

The task of a knowledge base is to represent a relevant body of knowledge about a specific application domain in order to support knowledge-based application programs and to be able to answer ad-hoc queries about the state of the application domain. It is essential to develop an adequate conceptual model of the application domain as the basis for designing a knowledge base. This involves the formalization of intuitions and of informal descriptions. We discuss the most important methodology for conceptual modeling: *entity-relationship modeling* which may be viewed as an extension of *predicate logic*. The emphasis of this chapter is on identifying the different types of predicates needed for capturing more domain knowledge: normal extensional, qualified, incomplete and intensional predicates.

1.1 INTRODUCTION

For designing a specific knowledge base, the first activity is to analyze the given application domain, and to develop a **conceptual model** of it. From this model, which may be established at different levels of abstraction, the **knowledge base schema** is derived. The schema of a knowledge base defines its type and structure, and the domain-specific vocabulary that is used to represent knowledge and to store and retrieve particular pieces of information in the knowledge base.

Conceptual modeling methods (or **meta-models**) are based on suitable abstractions of the data and processes application domains consist of. Traditional methods, such as predicate logic and entity-relationship modeling, have been exclusively focused on static representations of a domain, that is, they have been only concerned with its state. More recently, the need for integrating both static and dynamic aspects, i.e. both state and behavior, in a meta-model has become obvious. This is documented by the search for an integrated *object-oriented* modeling method both for general software engineering and for information systems. Although large research communities are occupied with this problem, and the recent proposal of a *Unified Modeling Language (UML)*, a combination of several previously proposed methods, has received great attention in software engineering, there is still no real breakthrough, and it seems that the right abstractions have not been found yet. Apparently, the principles and concepts of object-orientation are not sufficient for capturing all relevant static and dynamic aspects of

information systems. Rather, the abstractions seeked for may be found in the research areas of *multi-agent systems* and *agent-oriented programming*.

1.2 PREDICATE LOGIC

Conceptual modeling has been an issue in philosophy long before computers and computerized information systems have been invented. In philosophy of science and in logic, conceptual models of scientific domains have been a major concern. The main result of these principled philosophical investigations was the conceptual meta-model of predicate logic, as defined by Frege and Tarski.

In predicate logic, all basic entities (or individuals) of a given **universe of discourse** are conceived of as forming a flat set without any further classification. Normally, but not necessarily, they have names called **constant symbols** (such as $1, 2, 3, \ldots, \pi, \ldots$, I, II, III, ...Venus, 007, 'James Bond', etc.). An entity, in predicate logic, may have zero, one or more names. For example, '2', 'II' and 'two' denote the same natural number. Likewise, '007' and 'James Bond' denote the same person. Properties of, and relationships among, entities are expressed as sentences (or assertions) with the help of **predicate symbols**. The meaning of a predicate, in standard predicate logic, is given by its **extension**, i.e. the set of all tuples to which the predicate applies (forming a *relation* in the sense of set theory).

Notice that in the meta-model of predicate logic, there is no distinction between properties and relationships, both are expressed by predicates. On the other hand, in first order predicate logic, there is a strict distinction between basic entities (denoted by constants), functions and predicates while in natural discourse, both functions and predicates are treated as entities as well. These simplifications made by the predicate calculus do not pose any problem in the domain of mathematics (in fact, they proved to be very fertile). But in other, more empirical, application domains they are cognitively inadequate. For such domains, standard predicate logic is conceptually too flat, i.e. it does not offer sufficient support for those concepts needed to capture the essential semantics of empirical domains.

Therefore, the research discipline of conceptual modeling in computer science had to come up with new meta-models for being able to capture those parts of domain semantics that are relevant for database and knowledge system applications. The most fundamental of these meta-models is the *Entity-Relationship Modeling* method proposed by Chen, 1976.

1.3 ENTITY-RELATIONSHIP MODELING

In entity-relationship (ER) modeling, basic entities are conceptualized in conjunction with their properties and attributes which are (as opposed to predicate logic) strictly distinguished from relationships between entities. It is, however, straightforward to translate an ER model into classical predicate logic: both entity and relationship types correspond to predicates, and values from attribute domains correspond to constants. In fact, this translation is performed whenever an ER model is implemented as a relational database. In the current practice of database application development, this methodology (of first constructing an ER model and then implementing it with a relational database system) is state of the art. Notice, however, that the ER meta-model is more general than the relational database model. Neither does it require elementary attributes nor primary keys. It is therefore also compatible with the *object-relational* database model (see Chapter 3) where object-IDs replace primary keys, and attributes are no longer required to be elementary but may have complex values or may be derived by method invocation. While the original proposal of Chen, 1976, is restricted to the more basic ER constructs that can be easily implemented in relational databases, we enrich the standard ER meta-model in such a way that it can be used for the design of more general databases and knowledge bases (including the important case of object-relational databases).

In addition to defining a semi-formal terminology, the ER modeling method assigns visual representations to its concepts. The resulting **ER diagrams** are a convenient and concise way of expressing an ER model.

1.3.1 *Entities and Entity Types*

An **entity** is something that occurs in our universe of discourse (i.e. it may be the value of a quantified variable) and that can be uniquely identified and distinguished from other entities. Notice that this characterization refers to our universe of discourse and not to the 'real world'. Although it is commonly assumed that there is a close correspondence between the two, and in the database literature the universe of discourse (or even the content of a database) is often naively identified with 'the real world', we prefer to leave this issue to philosophy, and try to avoid any unnecessary reference to such metaphysical notions like 'the real world'.

Obviously, in an information processing system such as a database or a knowledge system, one does not deal with the entity itself (in the ontological sense) but rather with some representation of it. For linguistic simplicity, however, the term 'entity', in ER modeling, is used in its epistemic (representational) sense. Examples of entities are the natural numbers $1, 2, 3, \ldots$, Leo (our dog), James Bond (alias 007), Berlin, the planet Venus, my Alfa Romeo automobile, the 200th birthday of Schubert, the letter 'K', or the disease hepatitis A.

Entities of the same type share a number of **attributes** and **attribute functions** representing their properties or characteristics. An attribute associates with each entity a stored value from its domain. An attribute function associates with each entity a value to be computed by invoking the function. Together, the values of all attributes of an entity form the **state** of it. There may be different entities having the same state. In that case, one needs an appropriate naming mechanism for identifying and distinguishing entities independently of their state.

A **standard name** is a unique identifier assigned to an entity during its entire life cycle. If a *primary key* is defined for an entity type, it may be used as a standard name for entities of that type. An example of a primary key is the social security number assigned to employees. We assume, however, that entities can be identified independently of the existence of a primary key. How this is achieved need not be defined in the ER meta-model. Whenever an ER model is implemented in some practical knowledge system, there has to be a method to assign standard names to entities such as, e.g., primary keys in relational databases, or *object IDs* in object-relational databases. In contrast to predicate logic, in the ER meta-model and in database theory, it is commonly assumed that all entities have (standard) names.

There may be certain restrictions on the admissible values for attributes, called **integrity constraints**. A fundamental type of an integrity constraint is a **key dependency**, that is a minimal set of attributes whose values identify an entity. A key is a constraint since it implies that there must be no pair of entities of the same type having the same values for their key attributes. One of these key dependencies may be designated as the *primary key* of the entity type. Natural primary keys, such as social security numbers or flight numbers, are often used to identify entities and to access them in an information system. Sometimes there is no natural primary key, or even no key at all, in the definition of an entity type. It is a matter of implementation, and not of conceptual modeling, how to deal with this situation. In a relational database, an artificial primary key attribute has to be added to maintain entity identity while in an object-relational database an immutable object ID is automatically assigned to an entity on its creation.

In summary, an **entity type** is defined by a name, a list of attributes with associated domains, a list of attribute functions with associated signatures and implementations, and a possibly empty set of integrity constraints including key dependencies. Notice that an entity type, unlike the set-theoretic notion of a data type, is not a purely extensional concept. There may be two different entity types, say *Lecturers* and *Researchers*, with exactly the same list of attributes (and attribute functions) and possibly with the same extension. In contrast, two data type definitions, say *ULONG* and *DWORD*, having the same extension are equal, by the axioms of set theory.

At any moment, an entity type defined for a particular knowledge base is associated with its current **extension** consisting of all entities of that type which are currently represented in the knowledge base. An entity type is like a container, and the associated extension is its content. Thus, an entity type corresponds to a **table** in a natural way: the columns of the

table correspond to the attributes, a populated table contains the current extension of the entity type, and a table row, forming an attribute value tuple, corresponds to a specific entity.

Entity types are visually represented in the form of a labeled box which may be partitioned into a head and a body part, if the attribute list is to be included in the diagram. In that case, the name of the entity type is written in the head of the box, and the attributes are listed in its body, as in Figure 1.1.

Figure 1.1. Two entity types: students and courses.

The domain of an attribute is either an intensionally or an extensionally defined set of possible values. Typically, an *intensional domain* is defined by a formal grammar, while an *extensional domain* is defined by an explicit set of values. Examples of intensional domains are the integers, strings with at most 20 characters, or JPEG pictures. Examples of extensional domains are the names of foreign currencies or the names of colors. An extensional domain of an attribute of some entity type may be defined by the extension of another (single attribute) entity type.

Notice that different entities may belong to different epistemic categories. There are

1. concrete physical objects comprising passive objects (such as the Venus or my Alfa Romeo), and physically embodied agents (such as the Star Wars robot R2, our dog Leo, or James Bond),
2. software objects (such as the word processor window I am currently writing in)
3. software agents (such as the Microsoft Office97 assistant 'Clippit')
4. events (such as the 200th birthday of Schubert, or the mouse click to the beginning of this line),
5. locations (such as Berlin)
6. analytical concepts (such as the natural number 1),
7. empirical concepts (such as the disease hepatitis A),
8. symbols (such as the letter 'K'),

The traditional ER modeling method has not been using any such classification of entity types because it did not seem to be relevant in enterprise modeling for business applications, which is the classical application domain of ER modeling. In other domains, however, some of the above distinctions may be essential for capturing more domain semantics. But even in business domains, the distinction between objects and agents allows to capture more semantics concerning such important concepts like business transactions and roles.

1.3.2 Relationships and Relationship Types

A relationship among entity types is called a **relationship type**. For example, *attends* is a relationship type used to name the particular relationship that holds between a student S and a course C if S attends C.

A relationship between two entities is symbolically represented as an ordered pair labeled with the name of the relationship type. For instance, the expression

$$attends(\text{Kim}, \text{Logic1})$$

represents a relationship of type *attends* between the entity 'Kim' of type *Student* and the entity 'Logic 1' of type *Course*. Mostly, relationships are **binary**, like *attends*, but in some cases three or more entities are related which is represented by corresponding n-tuples.

There are two special relationships between entity types which are independent of the application domain and which are therefore distinguished from *domain relationships*: *subclass* and *aggregation* relationships.

The *subtype* relationship between abstract data types has to be distinguished from the **subclass** relationship between entity types. Unlike objects in object-oriented programming, entities are not instances of abstract data types (often called 'classes' in OO programming) but of entity types. Recall that an entity type is a named container with an associated abstract data type, namely the tuple type defining the attributes and attribute functions of all entities of that type. There may be several distinct entity types associated with the same abstract data type. A tuple type is a *specialization* (or a subtype) of another tuple type if it has additional attributes. If an entity type E is a specialization of an entity type D in the sense that every entity of type E is also an entity of type D, then E is called a *subclass* of D. This implies that every attribute of D is also an attribute of E, and E may have additional attributes (i.e. the data type of E is a subtype of that of D). The subclass relationship between two entity types is sometimes called *'isa relationship'* because we can say "E isa D". For instance, a sales engineer is a sales person, and hence the entity type *SalesEngineer* is a subclass of *SalesPerson*. We use the set-theoretic notation $E \subseteq D$ for expressing a subclass relationship and visualize it by placing a subclass below its superclasses and connect them with simple lines.

In an **aggregation** relationship, a number of *part* entities belong exclusively to an *aggregate* entity and do not exist independently of such an assignment. For instance, *Room* entities belong to a *Building* entity, which is expressed as *Building HAS Room*. An aggregation relationship type is visually represented by drawing the part entity box within the aggregate entity box.

Depending on how many entities from one entity type can be associated with how many entities of another entity type, three kinds of relationship types are distinguished:

1. **One-to-one**: For each entity of either type there is at most one associated entity of the other type.

2. **Many-to-one** from E_1 to E_2: Each entity of type E_2 is associated with zero or more E_1 entities, and each E_1 entity is associated with at most one E_2 entity. A many-to-one relationship from E_1 to E_2 is a (partial) function from E_1 to E_2.

3. **Many-to-many**: each entity of either type is related to zero or more entities of the other type.

A relationship type is visually represented using a labeled diamond, where the label consists of the name of the relationship type, connected to the related entity boxes by straight lines. Its functionality type is visualized by drawing crows feet for representing a *many*-side of the relationship type. So, a many-to-many relationship type is represented by drawing crows feet at the end of both connections to the related entity boxes, as in Figure 1.2.

Since relationships are also entities (of a higher-level), they may have attributes as well. For example, if a relationship holds only for some time, there may be an attribute for its duration.

1.3.3 Application Domain Predicates

Each name of an entity type and each name of a relationship type denotes an **application domain predicate** which can be used to express the properties of, and the relationships

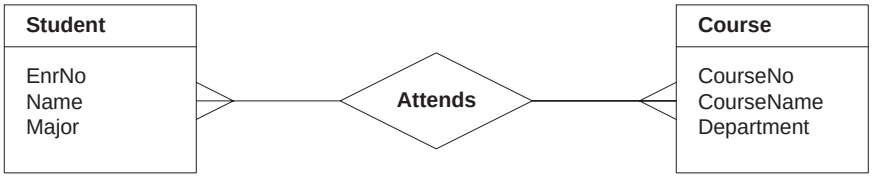

Figure 1.2. An ER model with a many-to-many relationship.

between, entities in the formal manner of predicate logic. Thus, an ER model of an application domain defines a formal language consisting of

1. the finite set of application domain predicates established by the entity-relationship analysis, and
2. the set of all possible attribute values, i.e. the union of all attribute domains, which forms the set of constant symbols.

In the sequel, we also briefly say *domain predicate* instead of *application domain predicate*.

1.3.4 ER Modeling Is Object-Oriented

ER modeling is object-oriented since it accommodates object identity, complex-valued and virtual attributes ('methods'), aggregation, and subclass hierarchies ('inheritance'). Many of the recently proposed object-oriented modeling methods are just notational variants of ER diagrams.

Object-oriented methods often include attempts to model the behavior of a system. Behavior modeling is a significant extension to the ER meta-model. However, the proposed methods, such as Petri nets and state transition diagrams, are not well-integrated with the constructs of state modeling and therefore not satisfactory. Finding a modeling method that integrates both state and behavior is still an open problem.

1.4 IDENTIFYING QUALIFIED PREDICATES

Certain domain predicates, on closer inspection, may turn out to be associated with some kind of qualification such as

- a **valid time** span during which a predicate is asserted to apply to a tuple,
- a **degree of uncertainty** with which a property or a relationship holds for an entity,
- a **level of security** with which the information that a predicate applies to a tuple is classified, or
- an **information source** according to which a property or relationship holds for an entity (in addition, a certain **degree of reliability** may be associated with the information source).

Qualified predicates require specialized logics (such as *temporal logic*, *fuzzy logic*, etc.) if their semantics is to be defined according to the paradigm of classical predicate logic. They can be represented in **tables with designated columns** containing the qualifications of tuples.

Qualified predicates are visualized in an ER diagram by including the qualification as a special attribute at the end of the attribute list separated by a line.

1.5 IDENTIFYING INCOMPLETE PREDICATES

An ER model may contain predicates for which there is no guarantee that complete information about them is available. For such predicates the *Closed-World Assumption (CWA)* is not

justified since not all tuples to which the predicate applies are known. Thus, one cannot infer that it is false that the predicate applies to a tuple simply because the tuple is not recorded in the database. The CWA requires that the owner of the database and its information suppliers have perfect control of the represented domain and know everything about it. Whenever this strong assumption is not plausible, some domain predicates may be incomplete (more precisely, their representation in the database may be incomplete).

There are two kinds of incomplete predicates: *total* and *partial* ones. While total predicates may, in principle, be completely represented in a database, partial predicates may not; they are necessarily incomplete. For partial predicates, falsity does not follow from non-truth: if it is *not true* that the predicate applies to a tuple, this does not already mean that it is therefore *false*. In general, predicates may have **truth-value gaps**, and falsity is not equal to non-truth. The classical *law of the excluded middle* applies only to total predicates no matter if their representation is complete or incomplete. It does not hold for incomplete predicates that are partial. The logical distinction between complete, incomplete total and partial predicates is discussed in Section 13.3.1.

Incomplete predicates are best explained by **partial logic** with weak and strong negation. The extension of an incomplete predicate is represented in a **bitable** consisting of an upper and a lower part. While the upper part contains the *truth extension*, the lower part contains the *falsity extension* of the represented incomplete predicate. Thus, bitables allow to store both positive and negative information.

An incomplete predicate is visually represented in an ER diagram by drawing a horizontal line dividing the head of an entity box or of a relationship diamond in an upper and a lower part.

As an example, consider the information system of an organization that wants to record the personal preferences of its employees towards each other by means of a relationship type *Likes*. The information whether an employee likes or dislikes another employee is obtained by interviews. Obviously, the information collected in this way is incomplete: the employee John need not like or dislike the employee Peter, he may be neutral with respect to Peter. So, it does not follow from John's not liking Peter that he does therefore dislike him. The predicate representing the *Like* relationship type is incomplete, and, consequently, corresponds to a bitable containing all pairs for which *Likes* hold, i.e. its truth extension consisting of all pairs $\langle x, y \rangle$ such that x likes y, and, in addition, all pairs for which *Likes* does definitely not hold, i.e. its falsity extension consisting of all pairs $\langle x, y \rangle$ such that x dislikes y.

1.6 IDENTIFYING INTENSIONAL PREDICATES

As opposed to an extensional predicate, an intensional predicate is not represented by means of a table but is defined with the help of other (intensional and extensional) predicates. The extension of an intensional predicate is not explicitly stored in the form of a base table but has to be computed from the extension of those other predicates occurring in its definition.

Intensional predicates are defined by means of **deduction rules** consisting of a *head* (or conclusion) and a *body* (or premise). They can be used to represent various kinds of relationships between concepts, such as subsumption or synonymy, or to represent properties and relationships based on heuristics and on default assumptions.

In summary, incomplete, qualified and intensional predicates extend the concept of normal predicates and lead to conservative extensions of relational databases as described in table 1.1. Since these extensions are orthogonal, they form a rather large space of possible combinations.

1.7 ENTITY-RELATIONSHIP MODELING IN 7 STEPS

The following procedure represents a simplified method how to obtain an ER model. Notice that whenever finishing one of the seven steps it is a good idea to review the results of all previous steps.

Table 1.1. Orthogonal extensions of relational databases.

predicate category	**representation**	**logic**
normal predicates	relational database tables	classical logic
incomplete predicates	bitables	partial logics
qualified predicates	tables with designated columns	specialized logics (temporal, fuzzy, etc.)
intensional predicates	deduction rules	constructive logics

Step 1: List all potential entity types.

Write a list of candidate entity types which covers the entire scope of the application domain.

Step 2: Write a summary paragraph for each entity type.

Write a summary description of the knowledge base model by writing a summary paragraph for each entity type identified in step 1. Use a restricted declarative style of language and enumerate paragraphs and sentences.

Step 3: Identify relationship types.

Identify relationship types by analyzing the summary description and looking for verbs. Start by looking for subclass and for aggregation relationships, and after that identify the remaining domain relationship types. The result of this should be three lists: a list of subclass relationships of the form $E_1 \subseteq E_2$, a list of aggregation relationships of the form $E_1\ HAS\ E_2$, and a list of domain relationship names together with associated entity types using either infix notation such as

$$E_1 \quad \underline{\text{isRelatedTo}} \quad E_2$$

or prefix notation such as

$$R(E_1, E_2, \ldots, E_n)$$

Step 4: Determine the epistemic category of all domain predicates.

For each predicate obtained from the list of entity types and the list of domain relationship types, determine its epistemic category:

1. Is it a complete extensional predicate good for absolute assertions ?
2. Is it a qualified predicate such that assertions made with it are informationally correct only relative to some qualification (like a valid-time span, a degree of uncertainty, or a security level) ?
3. Is it an incomplete predicate for which we do not have sufficient information competence in order to guarantee that its complete extension is available ?
4. Is it an intensional predicate whose extension is not explicitly stored in the knowledge base but is derived from other predicates via deduction rules ?

Step 5: Draw an ER diagram.

Draw an ER diagram using all entity types identified in step 1 and all relationship types identified in step 3. Visualize the subclass hierarchies by placing subclass entity boxes below their superclasses connecting them with simple lines. If an entity type has several subclasses, indicate whether they form a complete partition and whether they may overlap or have to be disjoint. Visualize aggregation relationships by drawing the box of the part entity type within

the larger box of the aggregate entity type. For each domain relationship type, determine whether it is one-to-one, one-to-many or many-to-many, and draw the corresponding crow's feet in the relationship connection lines accordingly.

Step 6: Determine the attributes of entities and relationships.

For each entity type and each relationship type, determine its attributes together with their domains and its attribute functions with their signatures. Mark complex-valued attributes and attributes with extensional domains.

Step 7: Complete the diagram by deriving additional entity types.

Inspect all attributes with extensional domains marked in step 6. Determine whether they refer to an entity type rather than to a data type, and derive additional entity types from this analysis. Add them to the ER diagram together with appropriate relationship links.

1.8 AGENT-OBJECT-RELATIONSHIP MODELING

This section contains a brief outlook to the future of conceptual modeling.

ER modeling does not account for the dynamic aspects of information and knowledge processing systems. These aspects are related to notions like communication, interaction, events, activities or processes. As pointed out above, it may be useful to distinguish between different kinds of entity types. In particular, for capturing semantic aspects related to the dynamics of knowledge systems, it is necessary to distinguish between **agents** and passive **objects**. While both objects and agents are represented in the system, only agents **interact** with it, and the possible interactions may have to be represented in the system as well.

Being entities, agents and objects of the same type share a number of attributes representing their properties or characteristics. So, in agent-object-relationship (AOR) modeling, there are the same notions as in ER modeling (such as attribute domain, state, integrity constraint, key, etc.). In fact, objects in AOR modeling are treated exactly like entities in ER modeling.

While ER modeling supports the design of *object-oriented* information systems realized with the help of relational and object-relational databases, AOR modeling allows the high-level design of **agent-oriented information systems**. An agent-oriented information system represents agents and their potential interactions in addition to ordinary objects. It need not be designed as an agent itself, however. Thus, the AOR meta-model is not intended as a methodology for the design of agent-based systems in general but rather of agent-oriented information systems.

1.8.1 Interaction Frames and Agent Roles

From an enterprise information system (EIS) perspective, examples of agents are customers and suppliers, as well as those employees whose role is to interact with customers and suppliers. But also another EIS, e.g. involved in automated electronic commerce, may be an agent from the perspective of a system dealing with it. So, agents in this context may be natural or artificial systems that are involved in some kind of business interaction. If the preconditions and effects of these interactions are represented in the EIS, the system is able to check the *consistency* and *completeness* of business transactions.

Notice that it is a matter of choice whether to represent active entities as agents or as objects. Agents are special objects. One may simply ignore their agent-specific properties and treat them as objects. For instance, if the supplier-to-customer interaction does not have to be explicitly modeled in an enterprise information system. Customers may be simply represented as objects in the same way as orders and bank accounts.

But even if it is not necessary to represent certain agent types in the target system and track their interactions, it may be important to consider them in the conceptual model in order to be able to derive a transaction and process model from the possible interactions with that type

of agent. The possible interactions between two agent types (or **roles**, say between customers and salespersons, or between patients and nurses) are defined by an **interaction frame** which lists for both participating roles:

1. All types of **communication acts** that may take place in the agent-to-agent communication frame. Since in the perspective of a 'listener', the communication acts of other agents are events, we also speak of **communication event types**.
2. All relevant types of **physical actions** which may be performed in response to other actions of (or which are otherwise related to) the interaction frame.
3. All relevant types of **environment events** which may be perceived by an agent and which are related to the interaction frame.
4. A set of **correctness properties** defining the correct behavior of agents within the interaction frame. Correctness properties are temporal logic assertions expressing either a *safety* or a *progress* property. Typically, these properties refer to communication events (or messages) and possibly to the mental state of agents.

A communication event type is defined by its name and a formal parameter list. A communication event is represented as a *typed message* together with the sender and the receiver address. The message type corresponds to the communication event type. There are domain-independent message types such as *request*, *tell*, *ask* and *reply*. But message types may also be domain-specific. In an electronic commerce interaction frame, examples of domain-specific message types are: *requestOffer*, *submitOrder* and *cancelOrder*.

Agents may be related to objects and to other agents by means of relationships as in ER modeling. An interaction frame can be viewed as a special agent-to-agent relationship.

1.8.2 Mental Attributes

While the state of an object does not have any application-independent structure or semantics, the state of an agent consists of *mental components* such as *beliefs* and *commitments*. In AOR modeling, the mental state of agents may be (partially) represented by means of **mental attributes**. Whenever an interaction frame contains correctness properties referring to the mental state of agents, the corresponding mental attributes must be included in the AOR model.

1.9 SUMMARY

Conceptual modeling methods support the definition of a formal language and of associated integrity constraints needed to formalize an application domain. Standard predicate logic has to be enriched by including further conceptual distinctions, such as the entity-relationship dichotomy or the agent-object-relationship trichotomy, in order to capture a sufficiently structured formal model of an application domain.

1.10 FURTHER READING

Elaborate presentations of the ER modeling approach to relational database design can be found in Teorey, 1994; Batini et al., 1992. Another interesting approach for modeling information systems, focusing on the formal-linguistic analysis of specification sentences in requirement documents, is proposed in Nijssen and Halpin, 1989.

1.11 EXERCISES

Problem 1-1. Establish an ER model (and draw the corresponding diagram) of the *university* domain consisting of the entity types *Person*, *Employee*, *Student*, *Course*, *Teacher*,

Teaching Assistant, and *MasterThesis*. Notice that the involved predicates are all extensional and complete. However, it is an option to model *Student*, *Course* and *Teaching Assistant*, as well as the relationships associated with them, as being qualified with a valid time.

Problem 1-2. Assume that the university is interested in information about its employees being smokers or non-smokers. This information, however, can only be obtained on a voluntary basis. Extend the university ER model from the previous exercise in an appropriate way.

Problem 1-3. Establish an ER model of the *hospital* domain consisting of the entity types *Person*, *Patient*, *Room*, *Physician*, *Diagnosis* and *Treatment*. Notice that diagnoses are uncertain and that information about patients (including their diagnoses and treatments) must be associated with a security level to protect it from unauthorized access.

2 THE FUNDAMENTAL CASE: RELATIONAL DATABASES

The system of relational databases represents the most fundamental case of a knowledge system. It already contains all the basic components and operations of a knowledge system such as the distinction between the query, the input, and the representation language, as well as an update, an inference, and an inference-based query answering operation. It is one of the central claims of this book that every knowledge system should be upwards compatible with, and conservatively extend, the system of relational databases. Knowledge systems satisfying this requirement are called *vivid* in Section 6.4.

After introducing the basic concepts of relational databases, three major components of databases are discussed: queries, integrity constraints and updates. It is shown that query answering is based on logical inference, and that updates are subject to certain logical conditions involving the specified integrity constraints. Then, the problem of null values in databases is presented as a violation of the database completeness assumption. Finally, we present Reiter's database completion theory and discuss the relation of databases to predicate logic model theory.

2.1 INTRODUCTION

The main purpose of a database is to store and retrieve information given in an explicit linguistic format. As opposed to certain other types of information that are also processed in cognitive systems (such as humans), this type of information is essentially *propositional*, that is, it can be expressed in a formal language composed of names for entities (or objects) and relationships, and of complex expressions formed from elementary ones by means of sentential connectives, such as negation, conjunction and disjunction. While only elementary pieces of information corresponding to atomic sentences can be stored in a relational database, the retrieval of specific information is achieved by submitting complex queries corresponding to complex logical formulas. Thus, it becomes obvious that predicate logic, with its account of formal languages composed of constant and relation symbols, and of sentential connectives, is the mathematical basis of query answering in databases.

Already in 1970, Edgar F. Codd published his pioneering article "A Relational Model of Data for Large Shared Data Banks" in the *Communications of the ACM*, where he defined the

principles of the relational database model. This was the first convincing conceptualization of a general purpose database model, and it is not an accident that it relies on formal logic.

In the mid-eighties finally, IBM presented its database management system DB2, the first industrial-strength implementation of the relational model, which continues to be one of the most successful systems for mainframe computers today. There are now numerous other relational database management systems that are commercially available. The most popular ones include Informix, Ingres, Oracle, Sybase and Microsoft SQL Server. To a great extent, the overwhelming success of these systems is due to the standardization of the database programming language *SQL* originally developed at IBM in the seventies. SQL is a declarative language for defining, modifying and querying relational database tables. SQL queries correspond to first order predicate logic formulas.

While most well-established information processing systems and tools such as programming languages, operating systems or word processors have evolved from practical prototypes, the unprecedented success story of the relational database model is one of the rare examples – certainly the most important one – where a well-established and widely used major software system is based on a formal model derived from a mathematical theory (in this case set theory and mathematical logic).

Logically, a relational database is a finite set of finite set-theoretic relations over elementary data types (called *tables*), corresponding to a finite set of atomic sentences. Such a collection of *ground atoms* can also be viewed as a finite interpretation of the formal language associated with the database in the sense of first order model theory.

The information represented in a relational database is updated by inserting or deleting atomic sentences corresponding to table rows (or tuples of some set-theoretic relation). Since a relational database is assumed to have complete information about the domain represented in its tables, if-queries are answered either by *yes* or by *no*. There is no third type of answer such as *unknown*. Open queries (with free variables) are answered by returning the set of all answer substitutions satisfying the query formula.

2.2 BASIC CONCEPTS

In this section, we introduce several concepts that are fundamental for database and knowledge systems: table (schema and instance), attribute, tuple, key, database (schema and instance), logical formula, query, and integrity constraint.

Recall that in set theory, an n-place relation P between the sets $D_1, \ldots, D_n$ is defined as a subset of their Cartesian product, i.e. as a set of respective tuples: $P \subseteq D_1 \times \ldots \times D_n = \{\langle c_1, \ldots, c_n \rangle \mid c_i \in D_i \text{ for } i = 1, \ldots, n\}$.

2.2.1 Tables

A finite set-theoretic relation may be represented as a table: the ith column of the table corresponds to the ith component of the relation, and the rows of the table correspond to the tuples in the relation. It is useful to give names to the columns of a table like in the following examples:

Student

StudNo	Name	Major
0112	Susan	Math
0123	Peter	CompSc
1175	Tom	Math

Attends

StudNo	CourseNo
0112	104
0112	223
0123	107
0123	128
0123	255
1175	255

Course

CourseNo	CourseName	Department
104	RDB	CompSc
107	Java	CompSc
128	C++	CompSc
223	Logic	Math
255	Algebra	Math

The names of table columns are also called **attributes**. Each attribute A has an associated domain of admissible values, denoted $dom(A)$. For instance, the domain of the attribute *CourseName* may be the set of all character strings with at most 20 characters:

$$\mathrm{dom}(\mathit{CourseName}) = \texttt{CHAR(20)}$$

Intuitively, it is clear that not every string from `CHAR(20)` is an acceptable course name. Therefore, `CHAR(20)` is called the **formal domain** of the attribute *CourseName*, while the set of all course names occurring in the database,

$$\{\mathit{RDB}, \mathit{Java}, \mathit{C++}, \mathit{Logic}, \mathit{Algebra}\},$$

is called its **actual domain**.

The rows of a relational database table are also called **tuples**. Tables and tuples can be represented in two ways:

1. In the standard set-theoretic definition, a tuple is a **sequence of values** $\langle c_1, \ldots, c_n \rangle$ and the meaning of each component value c_i is determined by its position within the sequence, viz by i. More precisely, an n-tuple $\boldsymbol{c}$ is a function $i \mapsto c_i$ assigning a value c_i to each tuple component i. A relational database table, then, is simply a finite set-theoretic relation over elementary data types, i.e. a finite set of tuples of the same type.

2. In relational database theory, a row of a table P is also represented as a **set of attribute/value pairs** $\{A_1/c_1, \ldots, A_n/c_n\}$, where $c_i \in \mathrm{dom}(A_i)$, i.e. as a valuation function $A_i \mapsto c_i$ assigning a value c_i to each attribute A_i. Here, a table is associated with a finite set of attributes, and the content of a table is a finite set of valuations which are also called tuples.

If the attributes of a table schema are linearly ordered (thus, forming a sequence), then both notions can be transformed into each other, that is they can be viewed as notational variants of each other. Since it is convenient to have both notations at hand, we adopt this enriched concept of set-theoretic relations and tuples, where attribute names (associated with a column or position number) may be used instead of position numbers for referring to specific components of a tuple.

If we are only interested in certain components $U \subseteq \mathrm{Attr}(P)$ of a tuple $\boldsymbol{c} \in P$, we may form its projection to these components denoted $\boldsymbol{c}[U]$. E.g., for the tuple $\boldsymbol{c} = \langle 0112, \mathit{Susan}, \mathit{Math} \rangle$,

$$\boldsymbol{c}[\mathit{Name}, \mathit{Major}] = \langle \text{Susan, Math} \rangle$$

While in ER modeling, there are two fundamentally different semantic categories, entities and relationships, there is only one in the relational database model: tables (or set-theoretic relations) which may represent both entities and relationships. We sometimes distinguish between an entity table and a relationship table, although this distinction is not supported by the relational database model.

2.2.2 Keys

A **key** of a table P is a minimally identifying set of attributes, i.e. a minimal set $K \subseteq \mathrm{Attr}(P)$, such that for two rows $\boldsymbol{c}_1, \boldsymbol{c}_2 \in P$,

$$\text{if } \boldsymbol{c}_1[K] = \boldsymbol{c}_2[K], \text{ then } \boldsymbol{c}_1 = \boldsymbol{c}_2.$$

A key expresses a functional relationship between the key attributes and the remaining attributes of the table. Notice that any table has at least one key, viz the trivial key $K = \mathrm{Attr}(P)$. In general, a table may have several keys one of which is designated as the **primary key** of the table.[1] In the case of an entity table, the primary key value serves as a unique standard name (or access ID) for an entity represented by a table row. For instance, in the entity table *Student*, the attribute *StudNo* is a natural candidate for the primary key, while in the relationship table *Attends*, both attributes, *StudNo* and *CourseNo*, are needed to form the primary key.

The problem with primary keys as a means to assign standard names to entities is that they are, like any attribute, modifiable. The semantics of standard names, however, requires that after they are assigned to an entity on its creation, they must never be changed throughout the entire lifecycle of the entity. Therefore, the primary key concept of the relational database model has been replaced by the **object ID** concept in object-oriented and object-relational databases. An object ID is a system-generated immutable unique identifier for an object.

2.2.3 Table Schema, Table Type and Table Instance

A **table schema** p consists of a finite sequence of attributes (without repetitions),

$$\langle A_1, \ldots, A_n \rangle,$$

corresponding to the columns of the table, and a set of *table integrity constraints* IC_p, including a set of keys,

$$\mathrm{Key}(p) = \{K_1, \ldots, K_k\},$$

one of which may be designated as the primary key $K_1(p)$. We also write simply $p(A_1, \ldots, A_n)$, or just p, to denote a table schema. The set of attributes of p is denoted by $Attr(p)$. The **type** of a table is determined by the number of its columns and their types (it is independent of the chosen column names, or attributes). More specifically, it is equal to the Cartesian product of its column types (or attribute domains).

Notice that a table schema does not determine the content of a table. It only restricts the possible contents of a table by requiring suitable values for each column. A specific table P is called an **instance** over the schema $p(A_1, \ldots, A_n)$, symbolically expressed as $P : p$, if it respects the domains of all attributes, that is if

$$P \subseteq \mathrm{dom}(A_1) \times \ldots \times \mathrm{dom}(A_n)$$

A table instance $P : p$ is called *admissible* if it satisfies all integrity constraints of p, that is, in particular, if for all $K \in \mathrm{Key}(p)$ and for each pair of rows $\boldsymbol{c}_1, \boldsymbol{c}_2 \in P$,

$$\text{if } \boldsymbol{c}_1[K] = \boldsymbol{c}_2[K], \text{ then } \boldsymbol{c}_1 = \boldsymbol{c}_2$$

The above table *Student*, for example, is an instance over the table schema

$$\mathrm{stud}(\mathit{StudNo}, \mathit{Name}, \mathit{Major})$$

It is easy to see that the table instance *Student* respects both the domains of all attributes and the only key of the table schema *stud*, consisting of the attribute *StudNo*:

$$\mathrm{Key}(\mathrm{stud}) = \{\langle \mathit{StudNo} \rangle\}$$

Every table has a schema. While normally a database table (i.e. its contents or state) is subject to frequent changes, its underlying schema is more or less stable, although it may evolve in certain cases. The term 'table' is used ambiguously for both the state and the schema. In a more logical terminology, a table schema corresponds to a **predicate**, and a specific table instance is the **extension** of that predicate at the given time point. While the extension of a predicate often changes over time, the predicate itself (the schema) persists.

As an example for another instance over the same schema *stud* consider the table

Student′ =

StudNo	Name	Major
0112	Susan	Math
0123	Peter	CompSc
2015	Ann	Psych

where, in comparison with the former instance *Student*, the math student Tom has been removed, and the psychology student Ann has been added. Thus, for denoting different states of an evolving table we have to use different names such as *Student* and *Student′*, while the underlying schema, *stud*, is the same for all of them. So, if we think of a table as a container with varying content, we actually mean the table schema.

In SQL92, a table schema is defined with the help of the `CREATE TABLE` command. For instance, the schema *stud* is defined as

```
CREATE TABLE stud(
StudNo   INTEGER,
Name     CHAR(20),
Major    CHAR(20)
PRIMARY KEY (StudNo)
)
```

In SQL3, it will be possible to separate the definition of a table type from that of a schema. For instance,

```
CREATE ROW TYPE student_type (
StudNo   INTEGER,
Name     CHAR(20),
Major    CHAR(20)
);

CREATE TABLE stud OF student_type
```

When using the name of a SQL database table in a query or a data modification command, it refers to the current extension of the table. When using it in a schema definition command, it refers to the schema. Thus, the name of a SQL table refers to both its schema and its current extension depending on the type of SQL expression it is used in. Like many terms in natural language, SQL table names do not have a fixed absolute meaning but are rather *indexical* because the same name denotes either the table schema or different table instances at different time points. This feature of SQL is not captured by standard logic where all terms have an extensionally fixed absolute meaning that is independent of time.

2.2.4 Database Schema and Database State

A database is built to represent relevant information about its domain of application. For this purpose, the schema of a relational database defines the concepts used to capture all relevant entities and relationships of the application domain.

A relational **database schema** Σ consists of a sequence of table schemas $p_1, \ldots, p_m$ and a set of *integrity constraints* IC:

$$\Sigma = \langle \langle p_1, \ldots, p_m \rangle, IC \rangle$$

such that IC includes the table-specific constraints IC_{p_i} defined along with the table schemas p_i.

Since table schemas correspond to predicates, a database schema Σ determines the set of predicates

$$\mathrm{Pred}_\Sigma = \{p_1, \ldots, p_m\}$$

A database schema defines a *lexicographical ordering* among the attributes of its tables:

$$A_i \leq A_j \quad \text{if} \quad \begin{array}{l} A_i, A_j \in \mathrm{Attr}(p_k) \ \ \& \ \ i \leq j, \ \ \text{or} \\ A_i \in \mathrm{Attr}(p_k) \ \ \& \ \ A_j \in \mathrm{Attr}(p_l) \ \ \& \ \ k \leq l \end{array}$$

This **attribute ordering** determines the column sequence of answer tables (see Section 2.3).

The second component of a database schema are *integrity constraints*. An integrity constraint is a logical sentence expressing some linguistic restriction on what is admissible in a database. Integrity constraints follow from our linguistic choices how to represent (and by our intuitive understanding of) the application domain. They are formally explained in Section 2.6.

The **formal domain** (or 'universe of discourse') D_Σ of a database schema Σ is the union of all attribute domains of all table schemas:

$$D_\Sigma = \bigcup\{\mathrm{dom}(A) \mid A \in \mathrm{Attr}(p) \ \ \& \ \ p \in \mathrm{Pred}_\Sigma\}$$

A **database instance** Δ over a schema Σ, symbolically denoted $\Delta : \Sigma$, is a sequence of suitable table instances,

$$\Delta = \langle P_1, \ldots, P_m \rangle$$

such that P_i is a table instance over p_i for $i = 1, \ldots, m$.

The **actual domain** $\mathrm{adom}(\Delta)$ of a database instance Δ is the set of all values occurring in its tables.[2]

An instance Δ is called *admissible*, if it satisfies all integrity constraints in IC. It is defined in Section 2.6, what it means to satisfy an integrity constraint.

In our university database example,

$$\Delta_{\mathrm{Uni}} = \langle \mathit{Student}, \mathit{Attends}, \mathit{Course} \rangle$$

is an instance over a schema Σ_{Uni} with predicates

$$\mathrm{Pred}_\Sigma = \{\mathrm{stud}, \mathrm{att}, \mathrm{crs}\}$$

such that *Student* : stud, *Attends* : att, and *Course* : crs. Instead of a database instance we also speak of a **database state** in order to emphasize that the database content typically evolves over time, or, in other words, the database changes its state.

2.2.5 A Database as a Set of Sentences

Each row $\boldsymbol{c} = \langle c_1, \ldots, c_n \rangle$ of a table $P : p$ corresponds to an **atomic sentence** $p(c_1, \ldots, c_n)$ where the schema name p is used as a predicate. A relational database $\Delta = \langle P_1 : p_1, \ldots, P_m : p_m \rangle$ can therefore be represented by the union of all sets of sentences obtained from its tables P_i:

$$X_\Delta = \bigcup_{i=1}^{m} \{p_i(\boldsymbol{c}) \mid \boldsymbol{c} \in P_i\}$$

Thus, a relational database Δ corresponds to a set of atomic sentences X_Δ which is also called its **propositional representation**. In the sequel, a database is identified with its propositional representation whenever it is convenient.

Example 1 *The database*

$$\Delta_1 = \langle P{:}p, Q{:}q \rangle \ \textit{with} \ P = \boxed{\begin{array}{cc} 1 & c \\ 2 & d \end{array}}, \ \textit{and} \ Q = \boxed{\begin{array}{c} 1 \\ 2 \\ 3 \end{array}}$$

corresponds to the following set of atomic sentences:

$$X_{\Delta_1} = \{p(1,c), p(2,d), q(1), q(2), q(3)\}$$

2.3 QUERY ANSWERING

.

Queries are used to retrieve relevant parts of the information stored in a database. A **query language** defines the syntax and expressiveness of database queries. In SQL, a query is formed with the `SELECT` keyword.[3]

In terms of conceptual clarity, the language of (suitably extended versions of) predicate logic is the query language of choice. It can also be viewed as a generic standard query language to which all practical query languages can be mapped.

2.3.1 Logical Formulas

Recall that the schema Σ of a relational database defines a set of table schemas or predicates $\text{Pred}_\Sigma = \{p_1, \ldots, p_m\}$, and a set $D_\Sigma = \{c_1, c_2, \ldots\}$ of constants, such as number and string constants. Variables ranging over D_Σ are usually denoted by $x, y, z, x_1, x_2, \ldots$.

In our university database example, the predicates defined by Σ_{Uni} are *stud*, *att* and *crs*, and examples of values are *0112*, *Tom*, *RDB*, *Math*, etc.

Thus, a relational database schema Σ defines a first order language L_Σ without function symbols. A **term** in such a language is either a constant or a variable. A tuple $\boldsymbol{u} = \langle u_1, \ldots, u_n \rangle$, where each component u_i is either a variable or a constant, is called **mixed tuple**.

An **atom** (or atomic formula) is either an atomic constraint of the form $u_1 \star u_2$, where $\star$ is a comparison operator (such as $=, <, >$), or a domain atom of the form

$$p(u_1, \ldots, u_n),$$

where p is an n-place predicate from Pred_Σ, and each of its arguments u_i is either a constant or a variable. The set of all domain atoms over Σ is denoted by At_Σ. An example of a domain atom over Σ_{Uni} is

$$\text{stud}(2015, \text{Ann}, x)$$

The set $L(\Sigma; \neg, \wedge, \vee, \supset, \exists, \forall)$, in short L_Σ, of logical formulas over Σ is defined inductively as follows.

1. All atoms are formulas in L_Σ.
2. Whenever F is a formula, so is its negation $\neg F$:
$$F \in L_\Sigma \;\Rightarrow\; \neg F \in L_\Sigma$$
3. Whenever F and G are formulas, so are their conjunction $F \wedge G$, their disjunction $F \vee G$, and the material implication[4] $F \supset G$:
$$F, G \in L_\Sigma \;\Rightarrow\; (F \wedge G), (F \vee G), (F \supset G) \in L_\Sigma$$
4. Whenever F is a formula, and x is a variable, then the existential quantification $\exists x F$ and the universal quantification $\forall x F$ are formulas:
$$F \in L_\Sigma \;\Rightarrow\; (\exists x F), (\forall x F) \in L_\Sigma$$

The set $\text{Free}(F)$ of all free variables of a formula F consists of all variables which are not bound by a quantifier, i.e. which do not occur as the argument of a quantifier in F. A formula without free variables is called a **sentence**. Connectives and quantifiers are summarized in Table 2.1.

Table 2.1. Connectives and Quantifiers.

negation	$\neg$	NOT
conjunction	$\wedge$	AND
disjunction	$\vee$	OR
material implication	$\supset$	IMPLIES
existential quantifier	$\exists$	EXISTS
universal quantifier	$\forall$	FOR ALL

2.3.2 Database Queries as Logical Formulas

We distinguish between two types of queries: if-queries and open queries.

An **if-query** is used to check if something is the case. In a relational database, it is answered by *yes* or *no*, while in a non-standard database or knowledge base it may be also answered by *unknown* or by other non-standard if-answers. If-queries are logical sentences which may or may not hold in a certain database state. When an if-query F holds in a database state X, this is expressed by means of the database **inference relation** as $X \vdash F$. For instance,

$$X_{\mathrm{Uni}} \vdash \mathrm{stud}(0112, \mathrm{Susan}, \mathrm{Math})$$

The answering of if-queries is defined by

$$\mathsf{Ans}(X, F) = \begin{cases} \mathsf{yes} & \text{if } X \vdash F \\ \mathsf{no} & \text{if } X \vdash \neg F \end{cases}$$

In general, a query may ask for all entities satisfying some property. For instance, the query

Who attends both the math course 'Logic' (223) and the computer science course 'RDB' (104) ?

corresponds to the logical formula

$$\mathrm{att}(x, 223) \wedge \mathrm{att}(x, 104)$$

with the free variable x standing for student numbers. We call this type of query an **open query**.[5] An answer to an open query formula $F[x_1, \ldots, x_k]$ with free variables $x_1, \ldots, x_k$ is a constant tuple $\langle c_1, \ldots, c_k \rangle$, such that the resulting instantiation $F[c_1, \ldots, c_k]$ holds in the current database state:

$$X \vdash F[c_1, \ldots, c_k]$$

Notice that each variable occurring in a query formula corresponds to an attribute (or to several attributes sharing the same domain). Thus, the notation $F[x_1, \ldots, x_k]$ implies two things:

1. that $x_1, \ldots, x_k$ are all free variables in F, i.e. $\mathrm{Free}(F) = \{x_1, \ldots, x_k\}$, and
2. that the variable sequence $\langle x_1, \ldots, x_k \rangle$ reflects the lexicographical order of the corresponding attributes and not the order of occurrence in F.

If F is an atomic query, both orderings agree. If F is a complex formula, the variable occurrence order reflects the syntactic composition of F while the lexicographical order is independent of it.

In the university database example, the lexicographical order of attributes is $\langle$ *stud.StudNo, stud.Name, stud.Major, att.StudNo, att.CourseNo, crs.CourseNo, crs.CourseName, crs.Department*$\rangle$. Thus, the query

$$F \equiv \mathrm{crs}(x, y, \mathrm{Math}) \wedge \mathrm{att}(z, x)$$

is denoted by $F[z, x, y]$.

The set of all answer tuples resulting from answering an open query F on the basis of X is denoted by $\mathrm{CAns}(X, F)$. Formally,

$$\mathrm{CAns}(X, F[x_1, \ldots, x_k]) = \{\langle c_1, \ldots, c_k\rangle \in D_\Sigma^k \mid X \vdash F[c_1, \ldots, c_k]\}$$

An answer $\langle c_1, \ldots, c_k\rangle$ to an open query $F = F[x_1, \ldots, x_k]$ can also be viewed as an *answer substitution*

$$\sigma = \{x_1/c_1, \ldots, x_k/c_k\}$$

since it determines the values c_i to be substituted for the free variables x_i such that the resulting instantiation $F\sigma \equiv F[c_1, \ldots, c_k]$ can be inferred from the database. Thus, we also write

$$\mathrm{CAns}(X, F) = \{\sigma \in D_\Sigma^{\mathrm{Free}(F)} \mid X \vdash F\sigma\}$$

where $D_\Sigma^{\mathrm{Free}(F)}$ denotes the set of all functions from $\mathrm{Free}(F)$ to D_Σ, i.e. the set of all suitable substitutions replacing the free variables of F with values from D_Σ.

For computing the answer set $\mathrm{CAns}(X, F)$, it would not be very efficient to guess potential answers σ and then check if they are valid, i.e. if $X \vdash F\sigma$, as the definition of CAns may seem to suggest. Rather one would like to have a compositional deterministic computation method. We will see in the following sections that **relational algebra** is such a method to implement an **answer operation** Ans that captures CAns in the case of relational databases.

2.4 CONJUNCTIVE QUERIES

In practice, most natural queries correspond to **conjunctive formulas**, i.e. logical formulas involving only conjunction and existential quantification.

An example of such a conjunctive query is the following: *'which computer science students attend a mathematics course ?'*, or, as a formula,

$$\exists z_1 \exists z_2 (\mathrm{stud}(x, y, \mathrm{CompSc}) \wedge \mathrm{att}(x, z_1) \wedge \mathrm{crs}(z_1, z_2, \mathrm{Math}))$$

Formally, the set $L(\Sigma; \wedge, \exists)$ of conjunctive formulas over a database schema Σ is defined inductively as follows.

1. $L(\Sigma; \wedge, \exists)$ includes all atoms.
2. It includes conjunctions: $(F \wedge G) \in L(\Sigma; \wedge, \exists)$ whenever $F, G \in L(\Sigma; \wedge, \exists)$.
3. And it includes existential quantifications: $(\exists x F) \in L(\Sigma; \wedge, \exists)$ whenever $F \in L(\Sigma; \wedge, \exists)$.

2.4.1 Inference-Based Answering of Conjunctive Queries

A database X implies a positive answer to an if-query F, if the sentence F can be *inferred* from X, or, in other words, if F *holds* in X. This is formally expressed by means of the **natural inference** relation $\vdash$ which is defined inductively over the composition of sentences.

An *atomic* sentence $p(c_1, \ldots, c_n)$ holds in a database X, if the tuple $\langle c_1, \ldots, c_n\rangle$ is an element of the database table P associated with p, or equivalently, if $p(c_1, \ldots, c_n)$ is contained in X:

$$(a) \quad X \vdash p(c_1, \ldots, c_n) \;:\Longleftrightarrow\; p(c_1, \ldots, c_n) \in X \Longleftrightarrow \langle c_1, \ldots, c_n\rangle \in P$$

A *conjunction* of two sentences holds, if both sentences hold:

$$(\wedge) \quad X \vdash F \wedge G \;:\Longleftrightarrow\; X \vdash F \text{ and } X \vdash G$$

An *existential* sentence $\exists x H[x]$ holds, if there is an answer to the open query $H[x]$:

$$(\exists) \quad X \vdash \exists x H[x] \;:\Longleftrightarrow\; \mathrm{CAns}(X, H[x]) \neq \emptyset$$

Table 2.2. Natural Inference for Conjunctive If-Queries.

(a)	$X \vdash p(c_1, \ldots, c_n)$	$:\Longleftrightarrow$	$p(c_1, \ldots, c_n) \in X$
$(\wedge)$	$X \vdash F \wedge G$	$:\Longleftrightarrow$	$X \vdash F$ and $X \vdash G$
$(\exists)$	$X \vdash \exists x H[x]$	$:\Longleftrightarrow$	$\mathrm{CAns}(X, H[x]) \neq \emptyset$
(CAns)	$\mathrm{CAns}(X, H)$	$=$	$\{\sigma \in D_\Sigma^{\mathrm{Free}(H)} \mid X \vdash H\sigma\}$

2.4.2 Compositional Evaluation of Conjunctive Queries

In order to 'implement' the computation of answer sets, an **answer operation** Ans is defined now inductively over the composition of conjunctive query formulas using the relational algebra operations *selection*, *projection* and *Cartesian product* or *join*. If $P : p$ represents an n-place predicate, then

$$\mathsf{Ans}(X, p(x_1, \ldots, x_n)) = P$$

that is, the answer to an atomic query $p(x_1, \ldots, x_n)$ is the extension P of predicate p in the database state X. For instance, in the case of the university database,

$$\mathsf{Ans}(X_{\mathrm{Uni}}, \mathrm{stud}(x, y, z)) = \mathit{Student}$$

Requiring Specific Attribute Values by Means of Selection. The selection operation selects specific rows of a given table. We define two kinds of atomic selection conditions: the *attribute equality constraint* $\$i = \$j$ denotes the requirement that columns i and j must have the same value, while the *attribute value constraint* $\$i = c$ means that column i must have the specific value c.

$$\begin{aligned} \mathrm{Sel}(\$i = \$j, P) &= \{\langle c_1, \ldots, c_n\rangle \in P \mid c_i = c_j\} \\ \mathrm{Sel}(\$i = c, P) &= \{\langle c_1, \ldots, c_n\rangle \in P \mid c_i = c\} \end{aligned}$$

Any finite sequence of such atomic selections with value and equality constraints C_i can be equivalently expressed by a single selection condition consisting of their conjunction:

$$\mathrm{Sel}(C_1, \mathrm{Sel}(C_2, \ldots \mathrm{Sel}(C_k, P)\ldots)) = \mathrm{Sel}(C_1 \wedge C_2 \wedge \ldots \wedge C_k,\ P)$$

Atomic selection is used to process partially instantiated atoms like, for instance, the query $p(x, c, x)$:

$$\begin{aligned} \mathsf{Ans}(X, p(x, c, x)) &= \mathrm{Sel}(\$1 = \$3, \mathrm{Sel}(\$2 = c, P)) \\ &= \mathrm{Sel}(\$1 = \$3 \wedge \$2 = c,\ P) \end{aligned}$$

Existential Quantification and Projection. Projection can be used to discard and/or permute the columns of a table. Let $\{j_1, \ldots, j_k\} \subseteq \{1, \ldots, n\}$. Then,

$$\mathrm{Proj}(\langle j_1, \ldots, j_k\rangle, P) = \{\langle c_{j_1}, \ldots, c_{j_k}\rangle \mid \langle c_1, \ldots, c_n\rangle \in P\}$$

Existential quantification in query formulas is evaluated by means of projection. Let $F = F[x_1, \ldots, x_k]$, and $1 \leq i \leq k$. Then,

$$\mathsf{Ans}(X, \exists x_i F) = \mathrm{Proj}(\langle 1, \ldots, i-1, i+1, \ldots, k\rangle, \mathsf{Ans}(X, F))$$

Notice that after discarding certain columns of a table the resulting projection may contain fewer rows than the original table since the projection may lead to duplicates which are removed.

Conjunction and Join. The algebraic basis of evaluating conjunction is the Cartesian product of two tables defined as follows:

$$P \times Q = \{\langle c_1, \ldots, c_m, d_1, \ldots, d_n\rangle \mid \boldsymbol{c} \in P \ \ \& \ \ \boldsymbol{d} \in Q\}$$

If two query formulas F and G do not share any free variable, i.e. $\mathrm{Free}(F) \cap \mathrm{Free}(G) = \emptyset$, then their conjunction can be evaluated by the Cartesian product of their answer sets:

$$\mathsf{Ans}(X, F \wedge G) = \mathsf{Ans}(X, F) \times \mathsf{Ans}(X, G)$$

If F and G share all of their free variables, $\mathrm{Free}(F) = \mathrm{Free}(G)$, then their conjunction corresponds to the intersection of their answer sets:

$$\mathsf{Ans}(X, F \wedge G) = \mathsf{Ans}(X, F) \cap \mathsf{Ans}(X, G)$$

Since the conjuncts of natural conjunctive queries often share some but not all variables, it is convenient to have an operator capable of handling this: the **join** operation which can be defined by Cartesian product, projection and atomic selection. Let $i = 1, \ldots, m$ and $j = 1, \ldots, n$. Then,

$$P \overset{i=j}{\bowtie} Q = \mathrm{Proj}(\langle 1, \ldots, m+j-1, m+j+1, \ldots, m+n\rangle, \mathrm{Sel}(\$i = \$j, P \times Q))$$

For instance,

$$\begin{aligned} & \mathsf{Ans}(X_{\mathrm{Uni}}, \mathrm{stud}(x, y, \mathrm{Math}) \wedge \mathrm{att}(x, z)) \\ = \ & \mathsf{Ans}(X_{\mathrm{Uni}}, \mathrm{stud}(x, y, \mathrm{Math})) \overset{1=1}{\bowtie} \mathsf{Ans}(X_{\mathrm{Uni}}, \mathrm{att}(x, z)) \\ = \ & \mathrm{Proj}(\langle x, y\rangle, \mathrm{Student}) \overset{1=1}{\bowtie} \mathrm{Attends} \end{aligned}$$

yields the answer set to the query *'list the student number and name of all math students together with the course number of all courses they attend'.*

Notice that the definition of $\overset{i=j}{\bowtie}$ can be easily generalized to the case of a conjunction with more than one shared variable. If all shared variables are treated in this way, the resulting operation $\bowtie$ is called *natural join.* It is used to evaluate general conjunctions:

$$\mathsf{Ans}(X, F \wedge G) = \mathsf{Ans}(X, F) \bowtie \mathsf{Ans}(X, G)$$

For instance,

$$\mathsf{Ans}(X, p(x, y, z) \wedge q(x, z, u)) = \mathsf{Ans}(X, p(x, y, z)) \bowtie \mathsf{Ans}(X, q(x, z, u))$$

is the set of all tuples $\langle c_1, c_2, c_3, c_4\rangle$ such that $\langle c_1, c_2, c_3\rangle \in P$ and $\langle c_1, c_3, c_4\rangle \in Q$.

2.4.3 Correctness and Completeness of the Answer Operation

In the previous section, we have 'implemented' the answer operation Ans inductively by providing an algebraic evaluation clause for each alternative of forming a conjunctive query. In order to show that this definition adequately captures the inference-based evaluation of conjunctive queries we have to prove that

1. every answer provided by Ans leads to an inferable instance of the conjunctive query formula, or, in other words,

$$\text{(correctness)} \qquad \text{if } \sigma \in \mathsf{Ans}(X, F), \text{ then } X \vdash F\sigma$$

2. every valid instantiation of an open conjunctive query formula is contained in the answer set provided by Ans:

$$\text{(completeness)} \qquad \text{if } X \vdash F\sigma, \text{ then } \sigma \in \mathsf{Ans}(X, F)$$

Proof of Correctness. The proof advances by induction on the complexity of conjunctive formulas. Let $P : p$ and $Q : q$ be tables in X. In the base case of a free atom $F = p(x_1, \ldots, x_n)$, since $\mathsf{Ans}(X, p(x_1, \ldots, x_n)) = P$, the answers σ correspond exactly to the rows $\langle c_1, \ldots, c_n \rangle \in P$, and hence, $F\sigma = p(c_1, \ldots, c_n)$, and by definition, $X \vdash p(c_1, \ldots, c_n)$.

Now, let $F = p(\boldsymbol{u})$ be a partially instantiated atom, where $\boldsymbol{u}$ is a mixed tuple. Then, F can be rewritten as a conjunction of the free atom $p(x_1, \ldots, x_n)$ and a conjunction of atomic selection conditions C such that

$$\mathsf{Ans}(X, p(\boldsymbol{u})) = \mathsf{Ans}(X, p(x_1, \ldots, x_n) \wedge C) = \mathrm{Sel}(C, P)$$

The elements of this answer set are the rows $\boldsymbol{c} \in P$ satisfying C, that is, for each equality constraint $(x_i = x_j) \in C$, it holds that $c_i = c_j$, and for each value constraint $(x_i = c) \in C$, it holds that $c_i = c$. Clearly, each such row represents a valid answer substitution $\sigma = \{x_1/c_1, \ldots, x_n/c_n\}$ for $p(\boldsymbol{u})$ in the sense that $X \vdash (p(\boldsymbol{u}))\sigma$.

Let $F = \exists x G(x)$ be an existentially quantified conjunctive formula. Assume that all answers to G are valid instantiations, i.e. $X \vdash G\sigma$ for all $\sigma \in \mathsf{Ans}(X, G)$ (this is the induction hypothesis). We have to show that F inherits this property from G. By definition, each $\sigma' \in \mathsf{Ans}(X, F)$ is the projection of some $\sigma \in \mathsf{Ans}(X, G)$ such that for some constant c, $\sigma = \sigma' \cup \{x/c\}$. Consequently, $G\sigma = (G(c))\sigma'$, and hence, $X \vdash F\sigma'$, since there exists c such that $X \vdash (G(c))\sigma'$.

Finally, let $F = G \wedge H$ be a conjunction of two conjunctive formulas G and H. Let $\sigma \in \mathsf{Ans}(X, G \wedge H)$ and let σ_1, σ_2 be the restrictions of σ to $\mathrm{Free}(G)$ and $\mathrm{Free}(H)$. By definition, $\sigma_1 \in \mathsf{Ans}(X, G)$ and $\sigma_2 \in \mathsf{Ans}(X, H)$. Hence, by the induction hypothesis, $X \vdash G\sigma_1$ and $X \vdash H\sigma_2$, implying that $X \vdash G\sigma_1 \wedge H\sigma_2$, and consequently, $X \vdash (G \wedge H)\sigma$, that is $X \vdash F\sigma$. □

The proof of completeness for Ans is left as an exercise.

2.4.4 Conjunctive Queries and SQL

SQL queries are expressed with the keyword `SELECT`. The basic form of the `SELECT` statement is

`SELECT` [`DISTINCT`] attributes
`FROM` tables
`WHERE` condition

involving projection (by means of `SELECT` *attributes*), selection (by means of `WHERE` *condition*), and Cartesian product (by means of `FROM` *tables*). Notice that SQL database systems, by default, treat tables as multi-sets rather than sets. Consequently, SQL database tables and answer sets may contain duplicate rows. This can be suppressed by using the optional keyword `DISTINCT`.

We briefly explain the meaning of SQL queries with the help of some illustrating examples. For instance, the SQL expression

`SELECT` Name, Major `FROM` stud

corresponds to the projection

$$\mathrm{Proj}(\langle \mathit{Name}, \mathit{Major} \rangle,\ \mathit{Student})$$

evaluating the query *'list the name and major of all students'*, or

$$\exists x(\mathrm{stud}(x, y, z)).$$

In SQL, the answer set contains all attributes if a '`*`' is specified instead of a particular attribute list in the `SELECT` clause. Thus,

`SELECT * FROM` stud `WHERE` Major = 'CompSc'

corresponds to the selection

$$\mathrm{Sel}(\mathit{Major} = \text{'CompSc'},\ \mathit{Student}),$$

evaluating the query *'list all attributes of computer science students'*, or

$$\text{stud}(x, y, z) \wedge z = \text{'CompSc'}).$$

Finally, if more than one table is specified in the `FROM` clause, the selection and projection refers to their Cartesian product or join, if the `WHERE` clause specifies join conditions. The following SQL query contains a row constraint and a join condition:

```
SELECT CourseName
FROM stud, att, crs
WHERE Major = 'CompS'
      AND stud.StudNo = att.StudNo
       AND att.CourseNo = crs.CourseNo
```

It corresponds to the relational algebra expression

$$\text{Proj}(\langle \mathit{CourseName} \rangle, \text{Sel}(\mathit{Major} = \text{'CompSc'}, \mathit{Student} \bowtie \mathit{Attends} \bowtie \mathit{Course}))$$

2.5 GENERAL QUERIES

Conjunctive queries, although capturing most cases of natural queries in practice, do not allow for indefinite or negative query conditions such as

1. 'Who attends the logic *or* the algebra course ?'
2. 'Who attends the RDB *but not* the logic course ?'

In addition to conjunction ($\wedge$) and existential quantification ($\exists$), general query formulas may also be composed of negation ($\neg$), disjunction ($\vee$), material implication ($\supset$), and universal quantification ($\forall$).

The **negation** $\neg p(c_1, \ldots, c_n)$ of an atomic sentence holds in X, if $\langle c_1, \ldots, c_n \rangle$ is not in P, or equivalently, if X does not contain $p(c_1, \ldots, c_n)$:

$$(\neg a) \quad X \vdash \neg p(c_1, \ldots, c_n) :\Longleftrightarrow p(c_1, \ldots, c_n) \notin X \Longleftrightarrow \langle c_1, \ldots, c_n \rangle \notin P$$

A **disjunction** of two sentences F and G holds, if one of them holds:

$$(\vee) \quad X \vdash F \vee G :\Longleftrightarrow X \vdash F \text{ or } X \vdash G$$

A material implication holds, if its premise does not hold, or its conclusion holds:

$$(\supset) \quad X \vdash F \supset G :\Longleftrightarrow X \vdash \neg F \text{ or } X \vdash G$$

A **universally quantified** sentence $\forall x H[x]$ holds, if there is no answer to the open query $\neg H[x]$:

$$(\forall) \quad X \vdash \forall x H[x] :\Longleftrightarrow \text{CAns}(X, \neg H[x]) = \emptyset$$

This inductive definition is completed by the following rewrite rules for the negation of complex formulas:

$$\begin{array}{rcl@{\qquad}rcl} \neg(F \vee G) & \longrightarrow & \neg F \wedge \neg G & \neg(F \wedge G) & \longrightarrow & \neg F \vee \neg G \\ \neg\exists x F(x) & \longrightarrow & \forall x \neg F(x) & \neg\forall x F(x) & \longrightarrow & \exists x \neg F(x) \\ \neg\neg F & \longrightarrow & F & & & \end{array}$$

For each rewrite rule $\mathit{LHS} \longrightarrow \mathit{RHS}$, it holds by definition that

$$\begin{array}{rcl} X \vdash \mathit{LHS} & :\Longleftrightarrow & X \vdash \mathit{RHS} \\ \text{CAns}(X, \mathit{LHS}) & = & \text{CAns}(X, \mathit{RHS}) \end{array}$$

Notice that for each relational database X, and for each if-query F in the language of X, either $X \vdash F$ or $X \vdash \neg F$ (see Exercise 2.17.2). Therefore, in relational databases, an if-query F is answered by either *yes* or *no*. There is no third if-answer, such as *unknown* or *yes-and-no*.

As an example of a non-conjunctive query, consider the query *'which q-tuples do not occur in p ?'* to the database Δ_1 from example 1, resulting in the singleton answer set

$$\text{CAns}(X_{\Delta_1}, q(x) \wedge \neg\exists y(p(x, y))) = \{\{x/3\}\}$$

since $X_{\Delta_1} \vdash q(3) \wedge \neg\exists y(p(3, y))$.

Table 2.3. Natural Inference in Relational Databases.

(a)	$X \vdash p(c_1, \ldots, c_n)$	$:\Longleftrightarrow$	$p(c_1, \ldots, c_n) \in X$
$(\neg a)$	$X \vdash \neg p(c_1, \ldots, c_n)$	$:\Longleftrightarrow$	$p(c_1, \ldots, c_n) \notin X$
$(\wedge)$	$X \vdash F \wedge G$	$:\Longleftrightarrow$	$X \vdash F$ and $X \vdash G$
$(\vee)$	$X \vdash F \wedge G$	$:\Longleftrightarrow$	$X \vdash F$ or $X \vdash G$
$(\supset)$	$X \vdash F \supset G$	$:\Longleftrightarrow$	$X \vdash \neg F$ or $X \vdash G$
$(\exists)$	$X \vdash \exists x H[x]$	$:\Longleftrightarrow$	$\mathrm{CAns}(X, H[x]) \neq \emptyset$
$(\forall)$	$X \vdash \forall x H[x]$	$:\Longleftrightarrow$	$\mathrm{CAns}(X, \neg H[x]) = \emptyset$
(CAns)	$\mathrm{CAns}(X, H)$	$=$	$\{\sigma \in D_\Sigma^{\mathrm{Free}(H)} \mid X \vdash H\sigma\}$

2.5.1 Not All Queries Can Be Answered Sensibly

In a computational setting like that of relational databases, unlike in pure mathematics, objects have to be finite structures. This requirement concerns in particular the answer set $CAns(X, F)$ which turns out to be infinite for certain formulas F. A formula F, thus, is only admissible as a sensible query to a database X, if X provides at most finitely many answers in response to F.

There are two remedies to this problem: one can either allow for arbitrary formulas as queries and introduce a form of non-standard answers (being finite representations of infinite sets), or one can exclude the problematic formulas from the set of admissible queries by restricting the query language in an appropriate way. While the first solution leads to a more general system, the latter one is easier to implement.

Consider again the university database X_{Uni}. In reply to the query *'list all entities not attending the Java seminar ?'*, the following infinite answer set would have to be returned:

$$\mathrm{CAns}(X_{\mathrm{Uni}}, \neg\mathrm{att}(x, \mathit{CompSc}, \mathit{Java})) = \{0112, 1175, 1, 2, 3, \ldots, \mathit{anynumber}, \ldots\}$$

Notice that this infinite answer set contains, besides finitely many sensible elements (viz actual student numbers of students not attending the Java seminar), an infinite number of values from the formal domain of the attribute *StudNo* (viz positive integers). The same problem may arise with universally quantified queries $\forall x F(x)$, because they are evaluated by checking the non-existence of answers for the negation $\neg F(x)$.

In the case of the – rather peculiar – disjunctive query, *'which courses are offered by the mathematics department or who attends the Java seminar ?'*, we also obtain an infinite answer set consisting of an infinite number of values from the formal domains of the attributes *crs.CourseNo* and *att.StudNo*

$$\mathrm{CAns}(X_{\mathrm{Uni}}, \exists y(\mathrm{crs}(x, y, \mathit{Math})) \vee \mathrm{att}(z, 107)) = \begin{array}{|cc|}\hline 223 & 1 \\ 223 & 2 \\ \vdots & \vdots \\ 223 & \mathit{any} \\ \vdots & \vdots \\ 1 & 0123 \\ 2 & 0123 \\ \vdots & \vdots \\ \mathit{any} & 0123 \\ \vdots & \vdots \\ \hline\end{array}$$

These considerations show that certain formulas involving negation, universal quantification or disjunction are problematic. Although, formally, they may be interpreted as queries, they do not seem to express any natural query, and they may lead to infinite answer sets containing an infinite number of non-informative answers. This observation suggests the following characterization of 'sensible answers':

> Only attribute values that are known to (or, in other words, represented in) the database are of interest in queries about empirical entities. Thus, a sensible answer must come from the actual domain of the database.

This requirement is captured by the formal property of *domain independence* which we define now in a general way applying not only to relational databases but also to other types of knowledge bases. For this purpose, we need the auxiliary notion of a restricted answer set where answers are restricted to a subdomain $D \subseteq D_\Sigma$, defined as follows:

$$\mathrm{CAns}_D(X, F) = \{\sigma \in D^{\mathrm{Free}(F)} \mid X \vdash F\sigma\}$$

Definition 2.1 (Domain Independence) *Let Σ be a database schema. A formula $F \in L_\Sigma$ is called* domain independent*, if for each instance X over Σ and each subdomain $D \subseteq D_\Sigma$ including the actual domain of X,* $\mathrm{adom}(X) \subseteq D$*, it holds that*

$$\mathrm{CAns}_D(X, F) = \mathrm{CAns}(X, F)$$

requiring that, no matter how we choose D, the restricted answer set does not vary with the particular choice of D.

While domain independence is a natural requirement for sensible query formulas in relational databases and many other knowledge systems, it may be too restrictive for systems dealing with formal objects such as time points or spatial coordinates.

The question whether a given query formula is domain independent was proved to be undecidable in DiPaola, 1969. It is therefore not possible to capture the property of domain independence by an operational (i.e. algorithmic, or syntactic) definition. Thus, several weaker formula classes approximating the class of domain independent formulas have been defined in the literature for the practical purpose of testing the domain independence of a given query. The most general of these is the notion of an **evaluable** formula proposed in Demolombe, 1982; Van Gelder and Topor, 1991.

In order to get a flavor of the syntactic restrictions required for an evaluable formula, we now present a simplified set of test conditions guaranteeing evaluability and domain independence.

(a) Atomic formulas $p(u_1, \ldots, u_m)$ are evaluable.

Whenever F and G are evaluable, then so is

$(\wedge)$	their conjunction $F \wedge G$
$(\exists)$	the existential formula $\exists x F$
$(\vee)$	their disjunction $F \vee G$, provided that $\mathrm{Free}(F) = \mathrm{Free}(G)$
$(\wedge\neg)$	the conjunction with a negation, $F \wedge \neg G$, provided that $\mathrm{Free}(G) \subseteq \mathrm{Free}(F)$
$(\wedge C)$	the conjunction with a row constraint, $F \wedge C$, provided that $\mathrm{Free}(C) \subseteq \mathrm{Free}(F)$

Observation 2.1 *An answer to an evaluable query formula F consists of constants from the actual domain of the database X. More precisely,*

$$\textit{if } X \vdash F[c_1, \ldots, c_k], \textit{ then } c_i \in \mathrm{adom}(X) \cup \mathrm{Const}(F), \textit{ for } i = 1, \ldots, k$$

where $\mathrm{Const}(F)$ *denotes the set of constants occurring in F.*

2.5.2 Compositional Evaluation of Open Queries

As we have seen in Section 2.4.2, complex query formulas can be evaluated by mapping sentential connectives and quantifiers to their algebraic counterparts: existential quantification ($\exists$) to projection (Proj), and conjunction ($\wedge$) to join ($\bowtie$). We still have to say how $\vee$, $\neg$ and $\forall$ are evaluated.

When its disjuncts agree on their free variables, a disjunction can be evaluated by means of set-theoretic union. For instance, the query *'who attends logic (223) or algebra (255) ?'*, can be answered as follows:

$$\begin{aligned} & \mathsf{Ans}(X_{\mathrm{Uni}}, \mathrm{att}(x, 223) \vee \mathrm{att}(x, 255)) \\ = \quad & \mathsf{Ans}(X_{\mathrm{Uni}}, \mathrm{att}(x, 223)) \ \cup \ \mathsf{Ans}(X_{\mathrm{Uni}}, \mathrm{att}(x, 255)) \end{aligned}$$

$$= \quad \boxed{\begin{array}{ll} 0112 & \text{Susan} \end{array}} \cup \boxed{\begin{array}{ll} 0123 & \text{Peter} \\ 1175 & \text{Tom} \end{array}} = \boxed{\begin{array}{ll} 0112 & \text{Susan} \\ 0123 & \text{Peter} \\ 1175 & \text{Tom} \end{array}}$$

The base operations of relational algebra are selection, projection, join, union, and difference. Their correspondence to logical connectives and to SQL operators is summarized in Table 2.4.

Disjunction and Union. The union of two tables (of the same type) is exactly their set-theoretic union:

$$P \cup Q = \{\boldsymbol{c} \mid \boldsymbol{c} \in P \ \text{ or } \ \boldsymbol{c} \in Q\}$$

Union is the algebraic counterpart of disjunction:

$$\mathsf{Ans}(X, F \vee G) = \mathsf{Ans}(X, F) \cup \mathsf{Ans}(X, G)$$

For forming the union of two type-compatible table expressions, SQL92 contains the operator `UNION`.

Negation and Difference. Let P and Q be tables with n and m columns. The difference of P and Q is defined only if $m \leq n$ and both tables agree on the types of their columns $1, \ldots, m$. Then,

$$P - Q = \{\langle c_1, \ldots, c_n\rangle \in P \mid \langle c_1, \ldots, c_m\rangle \notin Q\}$$

Negation in query formulas is only evaluable in conjunction with a positive choice to which the negative condition refers. Such a conjunction is evaluated by the difference operation:

$$\mathsf{Ans}(X, F \wedge \neg G) = \mathsf{Ans}(X, F) - \mathsf{Ans}(X, G)$$

In SQL92, there is an operator `EXCEPT` which computes the difference of two type-compatible table expressions.

General Selection. While conjunctive queries allow only for atomic selection (see Section 2.4.2), general queries allow in addition for negative and disjunctive selection conditions. A general **row constraint** C, thus, is a Boolean combination of attribute comparison constraints of the form $v \star w$, where v and w are either attributes or constant values, and $\star$ is one of the comparison operators $=, <, >, \leq, \geq$ or $\neq$. It is clear what it means to say that a table row $\boldsymbol{c} \in P$ *satisfies* a row constraint C (according to the standard evaluation of Boolean expressions). We can therefore define

$$\mathrm{Sel}(C, P) = \{\boldsymbol{c} \in P \mid \boldsymbol{c} \text{ satisfies } C\}$$

The selection operation is used to evaluate query formulas with general row constraints:

$$\mathsf{Ans}(X, F \wedge C) = \mathrm{Sel}(C, \mathsf{Ans}(X, F))$$

Table 2.4. Logic, algebra and SQL.

$F \wedge G$	$A_F \bowtie A_G$	A_F `NATURAL JOIN` A_G
$F \vee G$	$A_F \cup A_G$	A_F `UNION` A_G
$F \wedge \neg G$	$A_F - A_G$	A_F `EXCEPT` A_G
$p(x, c)$	$\mathrm{Sel}(\$2 = c, P)$	`SELECT` \$1 `FROM` p `WHERE` \$2=c
$\exists y(p(x, y, z))$	$\mathrm{Proj}(\langle 1, 3\rangle, P)$	`SELECT` \$1,\$3 `FROM` p

A_F, A_G denote the answer tables for the queries F, G.

2.5.3 *Correctness and Completeness of the Answer Operation*

After providing an algebraic evaluation clause for each alternative of forming an evaluable query, we can state the correctness and completeness of the answer operation:

1. Every answer σ provided by the answer operation Ans corresponds to an inferable instance $F\sigma$ of the query formula F, or, in other words,

$$\text{(correctness)} \qquad \text{if } \sigma \in \mathsf{Ans}(X, F), \text{ then } X \vdash F\sigma$$

2. Every valid instantiation σ of an evaluable query formula F is contained in the answer set provided by Ans:

$$\text{(completeness)} \qquad \text{if } X \vdash F\sigma, \text{ then } \sigma \in \mathsf{Ans}(X, F)$$

2.5.4 *The Database Completeness Assumption*

Usually, relational databases are assumed to be *complete* representations of their domain. That is, each database table P is assumed to contain the complete extension of the corresponding domain predicate p, and therefore it is inferred that the sentence $p(c)$ is false in X, i.e. its negation holds: $X \vdash \neg p(c)$, if c is not contained in the extension of p given by the database table P, i.e. if $c \notin P$. This **database completeness assumption** is the basic semantic principle for defining negative answers and the evaluation of negation in query formulas in relational databases. The following three definitions are based on it:

$$X \vdash \neg p(c) \quad :\Longleftrightarrow \quad p(c) \notin X$$

$$\mathsf{Ans}(X, F) \quad = \quad \begin{cases} \mathsf{yes} & \text{if } X \vdash F \\ \mathsf{no} & \text{otherwise} \end{cases}$$

$$\mathsf{Ans}(X, G \wedge \neg H) \quad = \quad \mathsf{Ans}(X, G) - \mathsf{Ans}(X, H)$$

The database completeness assumption was proposed under the name *Closed-World Assumption (CWA)* as a general principle for knowledge representation in Reiter, 1978. In the AI literature it is often identified with the nonmonotonic logic meta-inference rule "$\Gamma \vdash \neg\phi$ *if* $\Gamma \nvdash \phi$". This rule, however, leads to inconsistent conclusions if Γ represents indefinite information. This is easy to see: since $\{p \vee q\} \nvdash p$ and $\{p \vee q\} \nvdash q$, the CWA rule allows to conclude both $\neg p$ and $\neg q$, and hence, $\{p \vee q\} \vdash \neg p \wedge \neg q$ which is a contradiction. Consequently, the CWA inference rule is not the right formalization of the underlying intuitive principle and does not properly capture the database completeness assumption.

While the database completeness assumption is plausible in many administrative and business domains where complete information is available in a single database, it is inadequate in other domains where information is distributed among a number of databases and local tables are therefore incomplete, or where the information about predicates denoting empirical concepts is naturally incomplete. Examples of such empirical domains are space exploration, medicine, criminology, and all natural science domains.

But in practice, even in simple domains, there may be pieces of information which are incomplete and which can therefore not be represented in a pure relational database. In particular, there may be the case of missing information for certain attributes of some tuple. The relational database model proposed by Codd, and the SQL database standard, has a special concept for this type of incompleteness: **null values** which are discussed in Section 2.10. In the sequel, if not otherwise stated, relational databases are assumed to be 'pure', i.e. to have no null values.

2.5.5 Relational Database Queries Cannot Express Transitive Closure

We have seen that, in practice, most natural queries to relational databases are conjunctive formulas. The more general evaluable formulas can, in addition, express negative and disjunctive conditions. But one could want to ask still more involved queries such as asking for the **transitive closure** of a binary relationship. Take for example the relationship *DirectlyControls*(x, y) which holds between two companies x and y if x owns y or has a majority share in y. Then, one could be interested in the set of all companies controlled by some company, that is, in the set of all companies that are either directly controlled by it, or that are directly controlled by a company that is controlled by it. If the database contains the following table,

DirectlyControls =

ControlingCompany	ControlledCompany
BMW	Rover
BMW	Rolls Royce
IBM	Lotus
Rolls Royce	OES Ltd
Rolls Royce	TurboDrive
OES Ltd	K1 Ltd

then the answer to the query *'Which companies are controlled by BMW ?'* would be {Rover, Rolls Royce, OES, TurboDrive, K1}

Unfortunately, this particular query, and transitive closure queries in general, cannot be expressed by a first order logic formula, nor by a SQL92 statement. However, they can be expressed by *ADT queries* (see 3.1.4) and by means of **recursive deduction rules** (see 5.1.1). In SQL3, transitive closure queries can be expressed with the help of the `RECURSIVE UNION` operation:

```
SELECT ControlledCompany FROM DirectlyControls
WHERE ControlingCompany = 'BMW'
RECURSIVE UNION temp ( ControlingCompany, ControlledCompany)
SELECT ControlledCompany FROM DirectlyControls AS dc
WHERE dc.ControlingCompany = temp.ControlledCompany
```

2.5.6 Views

A **view** is defined by a query and is stored in the database under a specific name. It may be used in queries like a table name. A view may be *materialized* or *virtual*. The content of a virtual view, i.e. the answers to the query associated with it, is computed on demand. The content of a materialized view is stored and maintained in the database like a derived table.

Logically, a view is a named query defining an **intensional predicate**. For instance, referring to the extensional predicate *pers(Name, Sex, Father, Mother)* represented by the base table *Person*, one may define the view

$$\text{sibl}(x, y) \equiv x \neq y \wedge \exists z_1 \exists z_2 (\exists z_3 (\text{pers}(x, z_3, z_1, z_2)) \wedge \exists z_4 (\text{pers}(y, z_4, z_1, z_2))$$

for determining the siblings among all *Person*s.

In SQL, views are not materialized. A view definition in SQL, such as

```
CREATE VIEW sibl( Person1, Person2)
SELECT p1.Name, p2.Name FROM pers AS p1, pers AS p2
WHERE p1.Name ≠ p2.Name AND p1.Father = p2.Father AND p1.Mother = p2.Mother
```

can be used in queries like a table. Thus, the following query,

```
SELECT s.Person2, p.Sex FROM sibl AS s, pers AS p
WHERE s.Person1 = 'M. Jackson' AND p.Name = s.Person2
```

asks for the name and sex of all siblings of M. Jackson.

A view definition in SQL may also refer to other views. However, recursive or cyclic view definitions are not allowed.

2.6 INTEGRITY CONSTRAINTS

Integrity constraints are logical sentences that express certain conditions on the admissible states of a database or knowledge base. For instance, they may restrict the possible values of an attribute. Or they may require the presence of certain tuples in an associated table for each tuple of some table.

The following are natural examples of integrity constraints:

1. There is at most one course in a room at a given time.
2. Only medicine students may attend the pathology course.
3. Either the mathematics or the computer science department has to offer a FORTRAN programming course.

A set of integrity constraints IC_Σ is defined as a component of the database schema Σ. All integrity constraints of Σ have to be satisfied by every admissible database state X:

$$\text{for all } F \in IC_\Sigma,\ X \vdash F.$$

Thus, whenever an update attempt leads to the violation of an integrity constraint, it is rejected ('rolled back'), or triggers an automated assimilation (see 2.9.5).

Among the most important types of integrity constraints in relational databases are **keys** and **foreign keys** (or *referential integrity* constraints). They are particular examples of *functional* and *inclusion* dependencies.

2.6.1 Functional Dependencies

A table P satisfies a **functional dependency** $U \to V$ between two attribute sets $U, V \subseteq \mathrm{Attr}(P)$, symbolically

$$P \models U \to V$$

if for each pair of tuples $\boldsymbol{c}_1, \boldsymbol{c}_2 \in P$:

$$\boldsymbol{c}_1[V] = \boldsymbol{c}_2[V], \text{ whenever } \boldsymbol{c}_1[U] = \boldsymbol{c}_2[U]$$

requiring that any two tuples of P agree on their values for the attributes V whenever they agree on U. Expressed as a logical sentence, this corresponds to

$$X \vdash \neg\exists \boldsymbol{x} \exists \boldsymbol{y_1} \exists \boldsymbol{y_2} \exists \boldsymbol{z} (p(\boldsymbol{x}, \boldsymbol{y_1}, \boldsymbol{z}) \wedge p(\boldsymbol{x}, \boldsymbol{y_2}, \boldsymbol{z}) \wedge \boldsymbol{y_1} \neq \boldsymbol{y_2})$$

where the variables $\boldsymbol{x}$ correspond to the attributes U, and $\boldsymbol{y_1}, \boldsymbol{y_2}$ correspond to V.

A key $K \subseteq \mathrm{Attr}(P)$ of table P, see Section 2.2.2, represents a particular type of functional dependency:

$$P \models K \to \mathrm{Attr}(P)$$

For instance,

$$\mathit{Student} \models \mathit{StudNo} \to \langle \mathit{Name}, \mathit{Major} \rangle$$

2.6.2 Inclusion Dependencies

A database X satisfies an **inclusion dependency** between two tables $P : p$ and $Q : q$ with respect to the two compatible attribute sets $U \subseteq \mathrm{Attr}(P)$ and $V \subseteq \mathrm{Attr}(Q)$, if the projection of P to U is included in the projection of Q to V, symbolically

$$X \models p[U] \subseteq q[V] \;:\Longleftrightarrow\; \mathrm{Proj}(U, P) \subseteq \mathrm{Proj}(V, Q)$$

Expressed in the form of a logical sentence, this corresponds to

$$X \vdash \neg\exists \boldsymbol{x}(\exists \boldsymbol{y} p(\boldsymbol{x}, \boldsymbol{y}) \wedge \neg\exists \boldsymbol{z} q(\boldsymbol{x}, \boldsymbol{z}))$$

where $\boldsymbol{x}$ corresponds both to the attribute sets U and V.

The most important type of an inclusion dependency in relational databases is called *referential integrity constraint* or **foreign key dependency**. It requires that every tuple of a dependent table P refers to an entity tuple in some other table Q by including the corresponding primary key values in its own attribute values:

$$p[U] \subseteq q[K_1(q)]$$

For instance, in our university database example it holds that

$$\mathrm{att}[\mathit{StudNo}] \subseteq \mathrm{stud}[\mathit{StudNo}]$$

This is expressed in SQL in the schema definition of the dependent table *att* in the following way:

```
CREATE TABLE att(
...
FOREIGN KEY (StudNo) REFERENCES  stud
)
```

2.6.3 Integrity Constraints in SQL

In SQL92, three types of integrity constraints are distinguished according to their scope and the way they are naturally defined:

1. *Column constraints* are defined together with the respective columns. In addition to single attribute key dependencies (by means of the `UNIQUE` qualifier), SQL92 allows to define value restrictions (by means of specifying `DOMAIN`s and specific `CHECK`s) and single attribute foreign keys (by means of `REFERENCES`).

2. *Row* and *table constraints* are defined after the attribute list in the table schema. Multiple attribute keys and multiple attribute foreign keys are defined by means of `UNIQUE` and `FOREIGN KEY` (...) `REFERENCES`. Other row or table constraints are defined by specifying appropriate `CONSTRAINT` clauses.

3. General integrity constraints, referring to more than one table, are not assigned to a particular table. They are defined by `CREATE ASSERTION` statements as part of the general database schema.

For efficient integrity maintenance, it is possible to bind the satisfaction test for an integrity constraint to the update events of a specific table by declaring

```
CHECK Condition AFTER UPDATE ON Table [FOR EACH ROW]
```

2.7 TRANSFORMING AN ER MODEL INTO A RELATIONAL DATABASE SCHEMA

The ER modeling method is the standard design methodology for relational databases. Each entity type and each many-to-many relationship type is implemented as a separate table. A relationship table combines the primary key attributes of the participating entity tables.

Tables resulting from good ER models have a number of desirable properties such as

1. A table does not contain any column representing unrelated information.
2. There is no redundancy: one and the same fact is represented only once.

Tables satisfying these properties are called *normalized*. There are several *normal forms*, the most important one, introduced in Codd, 1974, is called *Boyce-Codd normal form (BCNF)*.

Definition 2.2 *A table P satisfies **Boyce-Codd normal form (BCNF)**, if every non-trivial functional dependency is a (super)key dependency, i.e. if $P \models U \to \mathrm{Attr}(P)$ whenever $P \models U \to V$ for some $V \not\subseteq U$.*

An entity type e is transformed into a table schema $e(A_1, \ldots, A_n, \ldots)$ that includes all of its attributes $A_1, \ldots, A_n$ and satisfies BCNF. Further attributes may have to be added to the schema if the attributes of the entity type do not contain a natural primary key or if some of them have complex-valued domains. The latter case is related to the design decision of Codd, the author of the relational database model, that attributes in relational database tables can only assume elementary values from base data types such as numbers and character strings.[6]

A many-to-many relationship type is transformed into a table schema consisting of the primary key attributes of the participating entity tables. Let K_1 and K_2 be the primary keys of entity types e_1 and e_2. Then, the many-to-many relationship type r between e_1 and e_2 is represented as the schema $r(K_1, K_2)$.

A functional (one-to-one or many-to-one) relationship type r from e_1 to e_2 is realized by one or more additional columns in the e_2 entity table for the primary key of e_1 provided that it is more or less total, i.e. if almost every e_2 entity is related to one or more e_1 entities. If instead only some e_2 entities are related to one or more e_1 entities, then it is preferable to implement the relationship type as a separate table $r(K_1, K_2)$.

For each relationship table, the corresponding foreign key dependencies have to be defined:

$$r[K_1] \subseteq e_1[K_1]$$
$$r[K_2] \subseteq e_2[K_2]$$

2.8 INFORMATION ORDER AND RELATIVE INFORMATION DISTANCE

Over time the content or state of a database changes. In some cases, the information content of a database may grow, in other cases it may shrink. Information content is a relative notion. It can only be defined as a comparison relationship $\leq$ between two databases, called **information order**. A relational database Δ' contains at least as much information as Δ, if the propositional representation of Δ is a subset of the propositional representation of Δ':

$$\Delta \leq \Delta' :\Longleftrightarrow X_\Delta \subseteq X_{\Delta'}$$

The information order is a theoretical concept that plays an important role in the definition of other concepts and in the semantics of databases and knowledge bases. In particular, the notions of *persistent queries* and *ampliative updates*, to be introduced below, depend on it.

Query formulas formed with conjunction, disjunction and existential quantifiers, are called *positive*. Obviously, the class of positive queries includes the class of conjunctive queries.

Observation 2.2 (Persistent Queries) *Positive queries are* persistent, *i.e. if $X \leq X'$ and $F \in L(\wedge, \vee, \exists)$, then*

$$X \vdash F \;\Rightarrow\; X' \vdash F$$

expressing that the inferability of a positive if-query from a relational database is preserved under growth of information. Correspondingly for an open positive query G, all answers are preserved under growth of information:

$$\mathsf{Ans}(X, G) \subseteq \mathsf{Ans}(X', G)$$

It is easy to see that queries involving $\neg$, $\supset$ or $\forall$ are, in general, not persistent. Notice that persistent queries are also called *monotonic* in the literature.

The **relative information distance** with respect to some database state X is a partial order $\leq_X$ that compares the 'information distance' to X. Intuitively, $Y \leq_X Z$ expresses the fact that Y is informationally less distant from X than Z. The notion of relative information distance is important for determining the *minimal mutilation* of a database when it is updated, see Section 2.9.4.

A natural definition of $\leq_X$ for comparing different states of a relational database is obtained by using the symmetric difference operation from set theory:

$$Y \leq_X Z \quad :\Longleftrightarrow \quad (Y - X) \cup (X - Y) \subseteq (Z - X) \cup (X - Z)$$

This definition measures the relative information distance by comparing the sets of atoms to be added or deleted in order to obtain the state X. As an example, consider

$$\begin{aligned} X &= \{p(1,c), p(2,d), q(1), q(2), q(3)\} \\ Y &= \{p(2,d), q(1), q(2), q(3)\} \\ Y' &= \{p(1,c), p(1,d), p(2,d), q(1), q(2), q(3)\} \\ Z &= \{p(1,d), p(2,d), q(1), q(2), q(3)\} \end{aligned}$$

Both Y and Y' are less distant from X than Z, that is, $Y \leq_X Z$ and $Y' \leq_X Z$. Y and Y' are not comparable according to this definition of relative information distance.

In the database literature, there are also other definitions of $\leq_X$. For instance, a less restrictive definition compares just the numbers of atoms to be added or deleted. And in Lobo and Trajcevski, 1997, it is argued that the relative information distance should be measured by an asymmetric difference giving more weight to the sets of atoms to be deleted resulting, e.g., in $Y' \leq_X Y$.

2.9 UPDATING RELATIONAL DATABASES

A database is updated in three ways: new pieces of information are inserted into it, and already-stored pieces may have to be modified or deleted. In our abstract framework, these changes are performed by an update operation Upd that takes an input sentence F and, if possible, assimilates it into the database X producing the updated database

$$X' = \mathsf{Upd}(X, F)$$

In SQL, there are three commands for performing database updates: `INSERT`, `DELETE` and `UPDATE`. Together, they form the *data modification* sublanguage of SQL.

2.9.1 Insertion

An atomic sentence $p(c_1, \ldots, c_n)$ is inserted into a database by simply adding the sentence to the propositional representation X, or by adding the tuple $\langle c_1, \ldots, c_n \rangle$ to the database table P:

$$\mathsf{Upd}(X, p(c_1, \ldots, c_n)) = X \cup \{p(c_1, \ldots, c_n)\}$$

For instance, if there is a new psychology student, Ann, with student number 2015, this leads to the following update:

$$\mathsf{Upd}(X_{\text{Uni}}, \text{ stud}(2015, \text{Ann}, \text{Psych}))$$

corresponding to the SQL command

`INSERT INTO` stud `VALUES` (2015, 'Ann', 'Psych')

2.9.2 Deletion

A new piece of information may also deny a previously valid fact $p(c_1, \ldots, c_n)$. This is expressed by its negation, $\neg p(c_1, \ldots, c_n)$. In a relational database, it is not possible to insert a negation. Instead, the sentence (or table row) which is denied has to be deleted:

$$\mathsf{Upd}(X, \neg p(c_1, \ldots, c_n)) = X - \{p(c_1, \ldots, c_n)\}$$

This method to process negative information is based on a fundamental assumption: the *database completeness assumption* which is discussed in 2.5.4.

When Peter, for instance, leaves the university, this may be expressed by the sentence

$$\neg\text{stud}(0123, \text{Peter}, \text{CompSc})$$

whose assimilation into Δ_{Uni} leads to the deletion of the row $\langle\, 0123, \text{Peter}, \text{CompSc}\,\rangle$ from the current extension of *stud*.

In SQL, one has to form the corresponding deletion command by referring to the table and to the concerning row with the help of its primary key value:

`DELETE FROM` stud `WHERE` StudNo = 0123;

2.9.3 Non-Literal Inputs

Atoms and negated atoms are called *literals*. The assimilation of literal inputs is implemented by the insertion and deletion of table rows. However, there are also more complex input sentences expressing more complex updates.

In order to change the value of an attribute, deletion and insertion may be combined: first the sentence with the old value is deleted, then the modified sentence with the new value is inserted. This can be expressed by the conjunction of a negative and a corresponding positive literal. When Ann, for instance, decides to change her major from psychology to philosophy, this requires the following update to the university database:

$$X'_{\text{Uni}} = \mathsf{Upd}(X_{\text{Uni}},\ \neg\text{stud}(2015, \text{Ann}, \text{Psych}) \wedge \text{stud}(2015, \text{Ann}, \text{Phil}))$$

In SQL there is a special operation for this type of database change called `UPDATE` which can be used to modify certain attribute values of already stored tuples:

`UPDATE` stud `SET` Major = 'Phil' `WHERE` StudNo = 2015

In general, any consistent conjunction of literals can be admitted as an input sentence expressing an update. The assimilation of such a conjunction into a relational database is implemented by some sequence of insertions and deletions of corresponding table rows. Obviously, disjunction and quantifiers cannot be admitted in input sentences since they cannot be compiled into atomic sentences.

2.9.4 On the Semantics of Updates

There is a close relationship between inference and update. The main principle for relating update to inference is the **success postulate**:

$$\mathsf{Upd}(X, F) \vdash F$$

This postulate requires that after assimilating an input F into a database X, this input should be inferable from the update result. It implies the assumption that all information sources are

competent, reliable and honest, and therefore the database should accept any new input, or, in other words, it should believe what it is told.

In an ideal world (such as the world of mathematical truths), a knowledge base X would only be expanded by newly established facts and since the truth of these facts is eternal, no previously established fact would ever have to be retracted. In this ideal world, the information (or knowledge) content of X is steadily growing, that is, all updates are *ampliative* and satisfy the *permutation principle* according to which the succession (and, hence, the timepoints) of intermediate updates has no influence on the final result of a successive update.

Observation 2.3 (Ampliative Inputs) *In a relational database without integrity constraints, atomic sentences are* ampliative inputs*, i.e. for an atom $p(c)$,*

$$X \leq \mathsf{Upd}(X, p(c))$$

In general, however, inputs are not ampliative. They may be *reductive*, if they cause a deletion or retraction, or they may be neither ampliative nor reductive, if they lead both to insertions and to deletions.

Observation 2.4 (Permutation) *If only ampliative updates are allowed in a relational database, the* permutation principle *holds, i.e. for two ampliative inputs F and G,*

$$\mathsf{Upd}(\mathsf{Upd}(X, F), G) = \mathsf{Upd}(\mathsf{Upd}(X, G), F)$$

It is easy to see that, in general, this principle is violated in relational databases (e.g. for $F = p(c)$ and $G = \neg p(c)$).

For relational databases without integrity constraints, the success postulate is realized in a straightforward manner by the above definitions of update, that is, by simple insertions and deletions. Its realization becomes much more involved for relational databases with integrity constraints, and for more advanced types of knowledge bases.

In general, a database schema contains integrity constraints and the result of an update has to respect them. A database has two choices how to deal with update requests that violate its integrity: either they are rejected, or, if possible, the delivered input is assimilated into the database in such a way that eventually all integrity constraints are satisfied by the update result. The assimilation of conflicting inputs may require the deletion or insertion of other information, and it may be done automatically by the system or interactively under the guidance by the user.

The general intuitive principle to be satisfied by any knowledge assimilation procedure is called **minimal mutilation**. It has been proposed in Belnap, 1977, and also in the philosophical theory of *theory change* as presented in Gärdenfors, 1988. The minimal mutilation principle requires that update, or knowledge assimilation, is done in such a way that the information content of the underlying database, or knowledge base, is only minimally changed. That is, nothing is added or deleted without being enforced by the requirement that the result of assimilating the input F into the database X over Σ is a database instance X' over Σ such that

1. $X' \vdash F$, and
2. $X' \vdash G$ for all $G \in IC_\Sigma$,

i.e. the information content of X' differs only minimally from that of X among all databases that satisfy F and IC_Σ. Notice that such an instance need not exist in all cases. For example, there is no instance satisfying both F and IC_Σ, if the new input F violates a value restriction in IC_Σ. In that case, the new input has to be rejected (SQL systems produce an error message), and the success postulate is violated.

We use the expression $\mathsf{Upd}(X : \Sigma, F)$ to denote the update operation that takes the schema Σ into account when assimilating F into X. The expression $\mathsf{Upd}(X, F)$ is an abbreviation for

the special case of $IC_\Sigma = \emptyset$, i.e. if there are no integrity constraints. The requirements for Upd can be formally summarized as follows:

$$\mathsf{Upd}(X : \Sigma, F) \in \mathrm{Min}_{\leq_X}\{Y \mid Y \vdash F \ \ \& \ \ Y \vdash IC_\Sigma\}$$

expressing that the result of updating $X : \Sigma$ by F is a minimal mutilation of X such that F and all integrity constraints of Σ hold. The minimal mutilation is defined via the notion of relative information distance $\leq_X$.

2.9.5 Updates in the Presence of Integrity Constraints

We now briefly discuss updates in connection with key dependencies and referential integrity constraints (foreign key dependencies). We restrict our considerations to *literal* inputs, i.e. atoms and negated atoms, corresponding to insertions and deletions.

Our first observation is that a key dependency may be violated by an insertion but not by a deletion. Consider the input

$$\mathrm{stud}(2015, \mathrm{Ann}, \mathrm{Phil})$$

to the above database Δ_{Uni}. It violates the key *StudNo* of table schema *stud* since there is already a tuple in the table *Student* with a *StudNo* value of 2015. Obviously, if this tuple is removed and the new tuple $\langle 2015, \mathrm{Ann}, \mathrm{Phil}\rangle$ is added, then we obtain an updated instance satisfying all of the above conditions: the result is a minimal mutilation of the original database instance such that the new input can be inferred and all integrity constraints are satisfied. Thus, instead of requesting explicitly a combined deletion and insertion for changing an attribute value like in

$$X'_{\mathrm{Uni}} = \mathsf{Upd}(X_{\mathrm{Uni}},\ \neg\mathrm{stud}(2015,\ \mathrm{Ann},\ \mathrm{Psych}) \wedge \mathrm{stud}(2015,\ \mathrm{Ann},\ \mathrm{Phil}))$$

one could simply request to assimilate the new input (not bothering about any old items that are outdated by it) as follows:

$$X'_{\mathrm{Uni}} = \mathsf{Upd}(X_{\mathrm{Uni}} : \Sigma_{\mathrm{Uni}},\ \mathrm{stud}(2015,\ \mathrm{Ann},\ \mathrm{Phil}))$$

The schema-based update operation, then, takes the key dependency

$$\mathit{StudNo} \rightarrow \langle \mathit{Name}, \mathit{Major}\rangle$$

of the schema Σ_{Uni} into account, thus, deleting the outdated tuple when inserting the new one. Notice, however, that this assimilation procedure is not supported by SQL which requires that the user is aware of possible key violations whenever inserting new tuples, and any insertion attempt violating a key dependency is rejected.

The second important case, besides the violation of a key dependency through the insertion of a tuple with a duplicate key, is the violation of referential integrity through the deletion of a tuple while there are still references to it, or through the insertion of a tuple with references to a non-existing tuple. The first of these two problems has a natural solution: all other tuples in the database that still refer to the tuple to be deleted are deleted as well. This step may have to be recursively iterated ('cascaded') if there are, in turn, references to tuples with invalidated references. The result of this procedure is a 'smaller' database where the original deletion request has been satisfied and where, through the recursive deletion of all tuples with invalidated references, referential integrity is maintained. This type of automated knowledge assimilation is supported in SQL through the possibility to specify a 'delete rule' along with the foreign key dependency in the schema definition of the referenced table.

We illustrate the problem with the help of our university database example where we have two foreign key dependencies,

$$\mathrm{att}[\mathit{StudNo}] \subseteq \mathrm{stud}[\mathit{StudNo}]$$
$$\mathrm{att}[\mathit{CourseNo}] \subseteq \mathrm{crs}[\mathit{CourseNo}]$$

which are defined in the table schema *att* along with the rule `ON DELETE CASCADE` for maintaining referential integrity:

```
CREATE TABLE att(
StudNo      INTEGER,
CourseNo    INTEGER,
FOREIGN KEY (StudNo) REFERENCES stud ON DELETE CASCADE
FOREIGN KEY (CourseNo) REFERENCES crs ON DELETE CASCADE
)
```

These definitions cause the database management system to automatically delete all entries in the *att* table that refer to a deleted student or to a deleted course. Thus, if the input sentence $\neg stud(0112, Susan, Math)$ is to express the update that Susan is no longer a student, then the update result

$$X'_{\mathrm{Uni}} = \mathsf{Upd}(X_{\mathrm{Uni}} : \Sigma_{\mathrm{Uni}},\ \neg\mathrm{stud}(0112, \mathrm{Susan}, \mathrm{Math}))$$

does no longer contain any *Attends* facts involving Susan's student number 0112, that is, the facts $att(0112, 104)$ and $att(0112, 223)$ are deleted.

We conclude our discussion of knowledge assimilation in relational databases with two observations. The first one concerns the logical characterization of update results by the minimal mutilation principle, and the second one concerns the impossibility to recover the information lost in certain types of updates.

According to the above characterization of update results based on minimal information distance, there is another possibility for assimilating

$$\neg\mathrm{stud}(0112, \mathrm{Susan}, \mathrm{Math})$$

In this second possibility, the fact $stud(0112, \mathrm{Susan}, \mathrm{Math})$ is deleted, and some other *Student* tuple with the same student number, say

$$\mathrm{stud}(0112, \mathrm{James}, \mathrm{Math})$$

is added preventing the violation of referential integrity. Obviously, this kind of update, introducing new unwarranted pieces of information for maintaining integrity, although legitimate according to the above logical characterization of update results, is not intended. This problem points to an inadequacy in the propositional representation of entities by a single predicate grouping several property attributes along with the primary key attribute. If entities are represented instead by a special entity existence predicate such as *stud(StudNo)* for expressing the existence of an entity with a specific standard name together with other predicates representing their properties, such as $stud(StudNo, Name, Major)$, then the deletion of an entity would be expressed by

$$X'_{\mathrm{Uni}} = \mathsf{Upd}(X_{\mathrm{Uni}} : \Sigma_{\mathrm{Uni}},\ \neg\mathrm{stud}(0112))$$

Now, the only way to maintain referential integrity by a minimal change of X_{Uni} is the intended one which requires the deletion of all other tuples referring to student 0112.

Observation 2.5 (No Recovery) *It is not possible to recover the information lost by a deletion that triggers the automated referential integrity maintenance procedure, that is, in general,*

$$\mathsf{Upd}(\mathsf{Upd}(X{:}\Sigma, \neg p(c)),\ p(c)) \neq X$$

2.10 INCOMPLETE INFORMATION IN RELATIONAL DATABASES

A knowledge base X is called *complete*, if for every if-query F, either $X \vdash F$ or $X \vdash \neg F$. There are various types of information violating this condition. Incompleteness may result, for instance, from

- Existential information such as *'Bob has a mobile phone, but its number is unknown.'*

- Incomplete predicates such as *likes*, to which the 'Closed-World Assumption' does not apply. See Section 11.3.
- Disjunctive information such as *'The patient has hepatitis A or B.'* See Section 9.3.
- Uncertain information such as *'It is very likely that the patient has hepatitis B.'* See Chapter 8.

Relational databases do neither allow to represent incomplete predicates, nor disjunctively imprecise, nor gradually uncertain information. However, they allow the representation of existential information in the form of *null values*.

2.10.1 The Null Value UNKNOWN

Only in the idealized pure relational database model do tuples have the form $\langle c_1, \ldots, c_n \rangle$ such that all attributes have definite values: $c_i \in \mathrm{dom}(A_i)$ for all $i = 1, \ldots, n$. In practice, however, there may be the case of incomplete tuples with unknown attribute values. For instance, the telephone number of some person may be unknown. In **relational databases with nulls**, as proposed in Codd, 1979, attributes may have the pseudo-value '?' standing for 'value currently unknown'.

A *Codd table* consists of tuples $\langle c_1, \ldots, c_n \rangle$ such that $c_i \in \mathrm{dom}(A_i) \cup \{?\}$ for all $i = 1, \ldots, n$. A *relational database with nulls* is a finite set of Codd tables.

Consider the database $\Delta_1 = \langle P, Q \rangle$ with the Codd table

$$P \subseteq \{a, b, c, \ldots, z\} \times \{1, 2, 3, ?\}$$

and the normal table $Q \subseteq \{1, 2, \ldots, 9\}$ with

$$P = \boxed{\begin{array}{cc} a & 3 \\ b & ? \\ d & 2 \end{array}} \qquad Q = \boxed{\begin{array}{c} 7 \\ 9 \end{array}}$$

Obviously, the intended meaning of the tuple $\langle b, ? \rangle \in P$ is captured by the sentence $\exists x(p(b, x))$, expressing that the value of the second component is unknown. This logical interpretation of Codd null values has been proposed in Biskup, 1981. The propositional representation Y_Δ of a database with nulls Δ, thus, consists of domain atoms and existentially quantified domain atoms. For instance,

$$Y_{\Delta_1} = \{p(a, 3), p(d, 2), \exists x(p(b, x)), q(7), q(9)\}$$

Notice that whenever the domain of an attribute is finite, a Codd null value for this attribute represents the disjunction over all possible values. Thus, instead of interpreting $p(b, ?)$ by the existential sentence $\exists x(p(b, x))$, we could also use the disjunction $p(b, 1) \vee p(b, 2) \vee p(b, 3)$ and abbreviate it by $p(b, \{1, 2, 3\})$. This consideration shows that in the case of finite domains, Codd tables correspond to OR tables which are treated in 9.3.

The presence of existential information in the database complicates many concepts that have been elegant and simple in the pure relational database model. In particular, we have to redefine the natural inference relation whose simple direct definition is no longer possible.

A Codd table $P \subseteq D_1 \cup \{?\} \times \ldots \times D_n \cup \{?\}$ represents a set of normal tables which is called its *instantiation*, formally defined as

$$\mathrm{Inst}(P) = \{Q \subseteq D_1 \times \ldots \times D_n \mid Q \text{ can be matched with } P\}$$

where we say that a normal table Q can be matched with a Codd table P, if both have the same cardinality, and for all tuples $\langle c_1, \ldots, c_n \rangle \in Q$ there is a tuple $\langle d_1, \ldots, d_n \rangle \in P$, such that $d_i = ?$ or $d_i = c_i$ for all $i = 1, \ldots, n$.

Likewise, a database with nulls $\Delta = \langle P_1, \ldots, P_m \rangle$ represents a set of pure relational databases, i.e. a set of alternative table sequences each representing a possible state of affairs:

$$\mathrm{Inst}(\Delta) = \mathrm{Inst}(P_1) \times \ldots \times \mathrm{Inst}(P_m)$$

Thus, in the above example, the instantiation of Δ_1 consists of three normal databases:

$$\mathrm{Inst}(\Delta_1) = \mathrm{Inst}(P) \times \mathrm{Inst}(Q) \quad = \quad \left\{ \langle \boxed{\begin{array}{cc} a & 3 \\ b & 1 \\ d & 2 \end{array}}, \boxed{\begin{array}{c} 7 \\ 9 \end{array}} \rangle, \right.$$

$$\langle \boxed{\begin{array}{cc} a & 3 \\ b & 2 \\ d & 2 \end{array}}, \boxed{\begin{array}{c} 7 \\ 9 \end{array}} \rangle,$$

$$\left. \langle \boxed{\begin{array}{cc} a & 3 \\ b & 3 \\ d & 2 \end{array}}, \boxed{\begin{array}{c} 7 \\ 9 \end{array}} \rangle \right\}$$

Inference from such a collection of relational databases is defined elementwise:

$$\Delta \vdash F \quad :\Longleftrightarrow \quad X \vdash F \text{ for all } X \in \mathrm{Inst}(\Delta)$$

Since it may be the case now that neither the sentence F nor $\neg F$ can be inferred, a third if-answer, *unknown*, has to be introduced:

$$\mathsf{Ans}(\Delta, F) = \begin{cases} \mathsf{yes} & \text{if } \Delta \vdash F \\ \mathsf{no} & \text{if } \Delta \vdash \neg F \\ \mathit{unknown} & \text{otherwise} \end{cases}$$

For instance,

$$\mathsf{Ans}(\Delta_1, p(b, 2)) = \mathit{unknown}$$

Several questions arise. How should open queries against a Codd database be evaluated ? Is there a simple redefinition of relational algebra which provides an adequate query evaluation ? May answers to open queries also contain null values ?

It can be shown by a simple consideration that the direct extension of projection, selection, join, union and difference, as proposed by Codd, for evaluating $\exists$, $\wedge$, $\vee$ and $\neg$ in the presence of nulls leads to a correct but incomplete answer operation. Take the query

$$F[x, y] \equiv (p(x, y) \wedge y \leq 2) \vee (p(x, y) \wedge y > 2)$$

Clearly, $\Delta_1 \vdash F[a, 3]$ and $\Delta_1 \vdash F[d, 2]$. Since also

$$\Delta_1 \vdash \exists x F[b, x]$$

i.e. $\Delta_1 \vdash F[b, ?]$, the answer set should be

$$\{\langle x, y \rangle \mid \Delta_1 \vdash F[x, y]\} = \boxed{\begin{array}{cc} a & 3 \\ b & ? \\ d & 2 \end{array}}$$

But by applying selection and union, we get

$$\begin{aligned} \mathsf{Ans}(\Delta_1, F[x, y]) &= \mathsf{Ans}(\Delta_1, p(x, y) \wedge y \leq 2) \cup \mathsf{Ans}(\Delta_1, p(x, y) \wedge y > 2) \\ &= \mathrm{Sel}(\$2 \leq 2, P) \cup \mathrm{Sel}(\$2 > 2, P) \\ &= \boxed{\begin{array}{cc} d & 2 \end{array}} \cup \boxed{\begin{array}{cc} a & 3 \end{array}} \\ &= \boxed{\begin{array}{cc} a & 3 \\ d & 2 \end{array}} \end{aligned}$$

Mainly because of this incompleteness, there has been much criticism of Codd's null value proposal. This criticism seems to overlook the nature of the problem: there is no simple adequate treatment of incomplete information. And there is no escape from dealing with Codd null values. They are there – in our cognitive practice, and in any practical information system.

2.10.2 The Null Value INAPPLICABLE

There is yet another type of null value needed in practice, namely *'attribute inapplicable'*. This inapplicability null is used as a special symbol to indicate for some tuple that the respective attribute has no meaning for it. For instance, if someone has no telephone, this is expressed by the null value *inapplicable*, as opposed to the case where someone's telephone number is not available which is expressed by the null value *unknown*. The logical semantics of *inapplicable* is much simpler than that of *unknown*. All terms and formulas that refer to the null value *inapplicable* are evaluated to *undefined*. Notice that in principle one can avoid the use of the *inapplicable* null value by designing the database schema in such a way that attributes are always applicable. This may require an appropriate vertical decomposition of certain entity tables.

2.10.3 Null Values in SQL

In SQL89 and SQL92, null values are assigned automatically by the system whenever a new tuple is inserted without providing values for all attributes. This can be avoided for specific attributes by declaring the column constraint `NOT NULL` for them in the table schema, leading to the rejection of incompletely specified new tuples. Null values are not allowed in attributes participating in a primary key.

Null values in SQL89 and SQL92 are used ambiguously for *unknown*, *inapplicable*, and *undefined*. Various inconsistencies and differences in query evaluation between different SQL database systems result from this built-in ambiguity. In SQL3, there will be different types of null values in order to distinguish between these different meanings.

2.11 A RELATIONAL DATABASE WITHOUT NULLS REPRESENTS A HERBRAND INTERPRETATION

pure λogic

A relational database $\Delta : \Sigma$ without null values can be viewed as a finite Herbrand interpretation $\mathcal{I}_\Delta$ of the formal language L_Σ, such that $\mathcal{I}_\Delta$ is a model of the propositional representation of Δ in the sense of predicate calculus. The notions of a Herbrand interpretation and a satisfaction (or model) relation are defined in Appendix A.

2.11.1 The Language L_Σ

A database schema $\Sigma = \langle\langle p_1, \ldots, p_m\rangle, IC\rangle$ defines a predicate logic signature $\langle \mathrm{Const}_\Sigma, \mathrm{Pred}_\Sigma\rangle$ without function symbols, consisting of a set of predicate symbols

$$\mathrm{Pred}_\Sigma = \{p_1, \ldots, p_m\}$$

with associated arities, and a set of constant symbols $\mathrm{Const}_\Sigma = D_\Sigma$ where constants are identified with the values of the formal domain D_Σ.

The predicate logic language $L_\Sigma = L(\Sigma; \neg, \wedge, \vee, \supset, \exists, \forall)$ based on this signature is defined in the usual way (see 2.3.1).

2.11.2 The Interpretation $\mathcal{I}_\Delta$

A database $\Delta : \Sigma$ defines an interpretation $\mathcal{I}_\Delta$ of the language L_Σ.

1. The universe of $\mathcal{I}_\Delta$ is equal to the formal domain D_Σ of Δ, i.e. it is identified with the set of constant symbols:
$$U_{\mathcal{I}_\Delta} = D_\Sigma = \text{Const}_\Sigma$$
This means that, unlike in the standard model-theoretic semantics of predicate logic, there is no distinction between domain values and constant symbols in the semantics of databases and knowledge systems.

2. Constant symbols are interpreted by themselves, that is,
$$\mathcal{I}_\Delta(c) = c$$
for all $c \in \text{Const}_\Sigma$.

3. Predicate symbols p_i are interpreted by their extension P_i contained in Δ:
$$\mathcal{I}_\Delta(p_i) = P_i$$
for $i = 1, \ldots, m$.

Thus, the satisfaction of atomic sentences is obtained by

$$\begin{array}{lll} \mathcal{I}_\Delta \models p(c_1, \ldots, c_n) & \Longleftrightarrow & \langle \mathcal{I}_\Delta(c_1), \ldots, \mathcal{I}_\Delta(c_n) \rangle \in \mathcal{I}_\Delta(p) \\ & \Longleftrightarrow & \langle c_1, \ldots, c_n \rangle \in P \end{array}$$

The interpretation $\mathcal{I}_\Delta$ is a model of the propositional representation of Δ:

$$\mathcal{I}_\Delta \models X_\Delta$$

In fact, $\mathcal{I}_\Delta$ is the least Herbrand model of X_Δ, and a sentence F can be inferred from X_Δ iff it is satisfied by $\mathcal{I}_\Delta$:

$$X_\Delta \vdash F \Longleftrightarrow \mathcal{I}_\Delta \models F$$

2.12 REITER'S DATABASE COMPLETION THEORY

pure λogic

An interesting question arises: which sentences have to be added to the propositional representation X_Δ in order to get the same sentences satisfied by $\mathcal{I}_\Delta$ (and inferable from X_Δ) also as valid conclusions by means of classical 2-valued entailment $\models_2$, i.e. for which set Γ_Δ of sentences the following holds:

$$X_\Delta \vdash F \Longleftrightarrow X_\Delta \cup \Gamma_\Delta \models_2 F$$

Reiter, 1984, has shown that such a completion of X_Δ can be achieved in three steps:

1. A set $Comp_\Delta$ of **completion axioms** restricts the extension of all predicates p to $\mathcal{I}_\Delta(p) = P$. That is, $Comp_\Delta$ contains for each predicate $p \in Pred_\Sigma$ having the extension $P = \{\boldsymbol{c}_1, \ldots, \boldsymbol{c}_k\}$, the sentence
$$\forall \boldsymbol{x}(p(\boldsymbol{x}) \supset \boldsymbol{x} = \boldsymbol{c}_1 \vee \boldsymbol{x} = \boldsymbol{c}_2 \vee \ldots \vee \boldsymbol{x} = \boldsymbol{c}_k)$$

2. A set UNA_Δ of **unique name axioms** requires that distinct names denote distinct objects. That is, for each pair of distinct constant symbols c and c' from the actual domain of Δ, UNA_Δ contains the sentence
$$c \neq c'$$

3. A **domain closure axiom** DCA_Δ requires that there are no other objects except those mentioned in Δ:
$$\forall x(x = c_1 \vee x = c_2 \vee \ldots \vee x = c_l)$$

where $\{c_1, \ldots, c_l\}$ is the actual domain of Δ, i.e. the set of all constant symbols occurring in Δ.

It is easy to see that the completion axioms together with the unique name axioms lead to the database completeness assumption (see 2.5.4):

$$X_\Delta \cup \mathit{Comp}_\Delta \cup \mathit{UNA}_\Delta \models_2 \neg p(\boldsymbol{c}) \iff \boldsymbol{c} \notin P$$

The effect of the unique name axioms and the domain closure axiom is that only finite Herbrand interpretations whose universe is the actual domain of Δ are accepted as models since Γ_Δ can only be satisfied by such interpretations. As a consequence of the domain closure axiom, universally quantified sentences are already entailed if they are satisfied by $\mathcal{I}_\Delta$. However, the underlying assumption that there are no other objects except those mentioned in the database is rather questionable (clearly, new objects are dynamically included in the actual domain of a database as soon as they become known to it).

We illustrate Reiter's completion theory with an example. For the database $\Delta : \Sigma = \langle P : p, Q : q\rangle$ with the tables

$$P = \boxed{\begin{array}{c}1\\2\end{array}} \quad Q = \boxed{\begin{array}{cc}c & d\end{array}}$$

the propositional representation is $X_\Delta = \{p(1), p(2), q(c,d)\}$ and

$$\begin{array}{rcl}\mathit{Comp}_\Delta & = & \{\forall x(p(x) \supset x = 1 \vee x = 2),\ \forall x, y(q(x,y) \supset \langle x, y\rangle = \langle c, d\rangle)\}\\ \mathit{UNA}_\Delta & = & \{1 \neq 2, 1 \neq c, 1 \neq d, 2 \neq c, 2 \neq d, c \neq d\}\\ \mathit{DCA}_\Delta & = & \forall x(x = 1 \vee x = 2 \vee x = c \vee x = d)\end{array}$$

Clearly: $\mathcal{I}_\Delta \models \neg p(c)$, and also $X_\Delta \vdash \neg p(c)$. But $X_\Delta \not\models_2 \neg p(c)$. To see this, consider the interpretation $\mathcal{I}_1 = \{p(1), p(2), p(c), q(c,d)\}$ which is a model of X_Δ but does not satisfy $\neg p(c)$. Since it violates the completion axiom for p, the interpretation $\mathcal{I}_1$ is no model of the completion Γ_Δ. As desired,

$$X_\Delta \cup \mathit{Comp}_\Delta \cup \mathit{UNA}_\Delta \cup \{\mathit{DCA}_\Delta\} \models_2 \neg p(c)$$

2.13 DATABASE INFERENCE AS PREFERENTIAL ENTAILMENT BASED ON MINIMAL MODELS

pure λogic

Inference in relational databases corresponds to preferential entailment based on minimal models (see the Appendix A.2 for the notion of minimal models). Relational databases, being finite sets of tables the rows of which represent atomic sentences, are viewed as finite models in database theory (see, e.g., Abiteboul et al., 1995). On this account, answering a query F in a database Δ is rather based on the model relation, $\mathcal{I}_\Delta \models F$, where $\mathcal{I}_\Delta$ is the finite interpretation corresponding to Δ, and not on an inference relation. However, especially with respect to the generalization of relational databases (e.g. in order to allow for incomplete information such as null values), it seems to be more adequate to regard a relational database as a set of sentences X_Δ, and to define query answering on the basis of the natural inference relation $\vdash$ which captures preferential entailment based on the unique minimal model of X_Δ. Inference in relational databases is not correct with respect to standard logical entailment ($\models_2$), but rather with respect to the minimal entailment relation ($\models_m$) of *semi-partial logic*, the strong-negation-free fragment of partial logic.

Since negation in relational databases does not behave like classical negation (expressing falsity) but rather like weak negation (expressing non-truth) in semi-partial logic, we adopt **partial logic** as a general logical framework for database semantics. Unlike classical logic, partial logic has a natural notion of **information order** among interpretations since it allows to distinguish between falsity and non-truth. This notion, and the related notion of a minimal model, are essential for the semantics of databases and knowledge bases.

In the logic programming literature, the notions of an information order and a minimal model have also been defined in the framework of classical 2-valued logic by treating falsity and truth

asymmetrically. This approach ignores the fact that there is no philosophical justification for such an asymmetry in classical logic. In fact, it even contradicts the philosophical and model-theoretic principles of classical logic as defined in the works of Frege and Tarski.

Apparently, in the fields of logic programming and artificial intelligence, classical logic is sometimes confused with semi-partial logic, and falsity is confused with non-truth. Since the entailment relation of semi-partial logic is isomorphic to that of classical logic (see Appendix A), this confusion does not lead to any difficulties in the formalism itself. But it may block researchers in getting a clear conceptual view of databases in accordance with basic intuitions derived from practical experience.

Observation 2.6 *Let X be (the propositional representation of) a relational database. Then, for any if-query F,*

$$X \vdash F \iff X^* \models_m F^*$$

where F^* denotes the translation replacing the classical negation symbol $\neg$ by the weak negation symbol $\sim$, and $\models_m$ denotes minimal entailment in semi-partial logic (see Section A.2).

2.14 DEFICIENCIES OF THE RELATIONAL DATABASE MODEL

Although relational databases represent a major breakthrough in the history of information processing, they still have a number of deficiencies calling for extensions and improvements. This concerns, in particular, the problems of entity identity and of complex-valued attributes and attribute functions. Also, as a requirement from conceptual modeling, the subclass (and inheritance) relationship between entity types should be supported. Furthermore, practical applications require base type extensibility and support for large data objects. And finally, for incorporating more semantics into databases and for knowledge management the support of rules is required. For expressing procedural knowledge in the database schema independently of specific application programs, the concept of *reaction rules* has to be supported. For representing terminological and heuristic knowledge, there should be support for *deduction rules*.

Besides these general purpose issues, there are several special purpose extensions of relational databases that are required to handle *qualifications* (such as valid time, security and reliability) and various forms of *incompleteness* (such as incomplete predicates and disjunctive information).

2.14.1 General Purpose Extensions

Adequate Support for Standard Names. In SQL92 databases, entities are identified with the help of primary keys. This requires that for any entity table, one of its keys is designated as primary and used as the standard name of the entities represented by the rows of that table. There are, however, cases where no natural key is available at all, and therefore an artificial primary key has to be introduced. Even more troublesome is the problem that an attribute participating in a primary key is, like any other attribute, modifiable. If such an attribute is, for some reason, modified, this leads to a change of the entity identity which is clearly undesirable. A standard name should be a unique identifier which is associated with an entity throughout its entire life cycle implying that it must be immutable.

Complex-Valued Attributes. For simplicity, Codd has banned complex-valued attributes from the relational database model (this requirement has been called '1. normal form'). Since in practice, many attributes have complex values, such as sets or tuples, this restriction leads to rather unnatural representations.

Attribute Functions. There are attributes whose values are not explicitly stored in the database but rather have to be computed in some way. These *virtual* attributes are associated

with a specific function that is invoked whenever their value is queried. Relational databases, and SQL92, do not provide any means to define attribute functions.

Subclass Hierarchy and Inheritance. In the conceptual model of an application domain, one often obtains hierarchical subclass relationships between entity types. Such a *subclass relationship* implies that entities of the subclass *inherit* all attributes from the superclass entities by means of *subtyping*. However, this mechanism is not supported in relational databases.

User-Defined Types and Large Objects. Since much of the information processed by human beings is of an audio-visual nature, there is a growing tendency to include pictures, sound samples and videos in the characterization of entities. Also, advanced software applications create and manipulate complex documents such as spreadsheets, project plans or hypertext documents, which may have to be included in the characterization of entities as well. So, there is a need for special attributes associated with user-defined data types of arbitrary length that can be queried with the help of user-defined functions and predicates. The BLOB ('binary large object') data type offered by most commercial database systems is only an ad-hoc solution to this problem, since it is not well-integrated with the other data types in the query language and access mechanisms.

Reaction Rules. SQL databases support a restricted form of reaction rules, called *triggers*. Triggers are bound to update events. Depending on some condition on the database state, they may lead to an update action and to system-specific procedure calls. In Chapter 4, we propose a general form of reaction rules which can be used to specify the communication in *multidatabases* and, more generally, the interoperation between *cooperative knowledge bases*.

Deduction Rules. Deduction rules may be used for defining intensional predicates and for representing heuristic knowledge. Intensional predicates express properties of, and relationships between, entities on the basis of other (intensional and extensional) predicates. Heuristic knowledge is typically represented in the form of *default rules* which are most naturally expressed using the *weak* and *strong* negation from *partial logic*. While relational databases allow to define non-recursive intensional predicates with the help of *views*, they do not support default rules or any other form of heuristic knowledge.

2.14.2 Special Purpose Extensions

We list a number of special purpose extensions of relational databases.

Valid Time and Belief Time. In certain applications, it is essential to deal with temporally qualified information. This includes both the time span during which a sentence (or piece of information) is valid, and the time span during which a sentence is believed and recorded in the database. The addition of valid time and belief time requires special query capabilities which cannot be achieved by using the temporal data types of SQL92.

Spatial and Geographic Information. Spatial information is related to geometric concepts like points, lines, rectangles, polygons, surfaces, and volumes. Spatial database systems include direct support of these concepts. They are used for realizing *geographic information systems*.

Incomplete Predicates. There are domain predicates, especially in empirical domains, for which the database completeness assumption is not justified. For these incomplete predicates the falsity extension is not simply the set-theoretic complement of the truth extension, and therefore both positive as well as negative cases have to be treated on par. An ad-hoc solution for incomplete predicates in relational databases would be to represent them in two tables: one

for their truth and another one for their falsity extension. But there would still be no support of this concept in the query and update mechanisms.

Security. SQL databases only support security requirements at the schema level, that is, user-specific access restrictions can be defined for entire tables or for sensitive columns which are then non-modifiable or non-observable for unauthorized users. In practice, these possibilities are not sufficient to protect confidential information. Rather, one needs contents-based access restrictions preventing unauthorized users to query or update sensitive tuples of a table.

Uncertainty and Reliability. There are domains where information is typically uncertain, or where the information sources are not completely reliable. In these cases, it is necessary to assign degrees of uncertainty to new pieces of information, or to label them with their source which may be associated with a degree of reliability.

Disjunctive Information. The most difficult extension of databases and knowledge bases concerns the integration of disjunctive information. While the kind of incompleteness created by disjunctive information is an important characteristic of commonsense knowledge representation and reasoning, theoretical investigations have not succeeded yet in establishing convincing semantics and efficient computation methods for storing and retrieving disjunctive information. Questions about the right semantics arise when disjunctive information is combined with negation (on the basis of the database completeness assumption) and rules. Efficiency problems arise from the combinatorial explosion created by disjunctive information.

2.15 SUMMARY

The system of relational databases represents the most fundamental case of a knowledge system. Consisting of elementary-valued tables, a relational database is restricted to atomic sentences. More complex tables and more complex sentences, such as negated, disjunctive or qualified sentences, which are needed to represent various forms of incomplete and qualified information, are not admitted. They require specific extensions of relational databases leading to object-relational, temporal, fuzzy, deductive, and other advanced database and knowledge systems.

2.16 FURTHER READING

Classical text books on the relational database model are Date, 1986; Ullman, 1988; Ullman, 1989; Korth and Silberschatz, 1991. An in-depth presentation of database theory, focusing on dependency theory and on issues of expressive power and complexity, is Abiteboul et al., 1995.

2.17 EXERCISES

2.17.1 A Further Example: The Library Database

Let

$$\Delta_{\text{Lib}} = \langle \mathit{Book} : \text{b},\ \mathit{EditedBook} : \text{eb},\ \mathit{AuthoredBook} : \text{ab},\ \mathit{Client} : \text{cli},\ \mathit{IsLentOutTo} : \text{ilt} \rangle$$

be a relational database with the following tables:

Book

InvNo	Title	Year
880	Vivid Logic	1994
988	Formal Methods	1995
284	Intentionality	1983
921	In Contradiction	1987
903	Brainstorms	1978
926	Sofie's World	1993

IsLentOutTo

InvNo	Name	Date
880	Wagner	2.12.96
988	Fisher	15.11.96
284	Kim	1.4.95
903	Wagner	20.9.96
926	Simon	12.10.96

EditedBook

InvNo	Editor
988	Bowen
921	Priest
903	Dennett

AuthoredBook

InvNo	Author
880	Wagner
284	Searle
926	Gaarder

Client

Name	State
Fisher	student
Wagner	lecturer
Kim	student
Simon	student

Let X_{Lib} be the propositional representation of Δ_{Lib}.

Problem 2-1. Which keys do the five library tables have ?

Problem 2-2. Which inclusion dependencies are there ?

Problem 2-3. Define the schema of Δ_{Lib} in SQL by means of `CREATE TABLE` statements including all keys and foreign key dependencies.

Problem 2-4. Does $X_{\mathrm{Lib}} \vdash \exists x \exists y(\mathrm{ilt}(x, y, 11.11.96))$ hold ? Express this if-query in natural language.

Problem 2-5. Express the following natural language queries both as logical formulas and as SQL statements:

1. Which books have already been in the library before 1990 ?
2. Which edited books have been added to the library after 1990 ?
3. Which books are currently not lent out ?
4. List all students who have borrowed an edited book.
5. Are there books that are neither authored nor edited ?
6. Who is both an author and a client of the library ?

Problem 2-6. Compute the answers to these queries by means of relational algebra operations:

$$\mathsf{Ans}(X_{\mathrm{Lib}}, \mathit{Query}) = \quad ?$$

Problem 2-7. Are the following formulas evaluable ? Why (not) ?

1. $\exists x(\mathrm{b}(x, y, 1991)) \vee \exists x(\mathrm{b}(x, y, 1992))$
2. $\mathrm{eb}(x, \mathrm{Bowen}) \vee \mathrm{ab}(y, \mathrm{Bowen})$
3. $\exists x(\mathrm{b}(x, y, z)) \wedge \neg \mathrm{ilt}(x, \mathrm{Wagner}, 11.11.96))$
4. $\mathrm{cli}(x, \mathrm{lecturer}) \wedge \neg \exists y(\mathrm{ab}(y, x))$

Problem 2-8. Express the following integrity constraints as logical sentences:

1. A book cannot be lent out before its year of appearance.
2. Only lecturers are entitled to borrow more than one book.

Problem 2-9. How can you express the if-query $\exists x \exists y(\mathrm{ilt}(x, y, 11.11.96))$ in SQL such that the result is 'yes' or 'no' ? Notice that SQL allows statements of the form `SELECT` 'yes' `AS` Answer `FROM` ...`WHERE` ...

2.17.2 Database Completion

Problem 2-10. Let $X_\Delta = \{q(d), p(c,1), p(d,2)\}$. Formulate $Comp_\Delta$, UNA_Δ and DCA_Δ.

Problem 2-11. Prove by induction that for each relational database X, and for each if-query F in the language of X, either $X \vdash F$ or $X \vdash \neg F$.

2.17.3 Miscellaneous

Problem 2-12. Prove by induction on the complexity of formulas that all conjunctive queries are satisfiable: if F is a conjunctive if-query, there exists a database instance X such that

$$X \vdash F,$$

and correspondingly, if G is an open conjunctive query, there exists a database instance X such that

$$\mathsf{Ans}(X, G) \neq \emptyset$$

Notes

1. Keys should not be confused with *indices* which are a purely technical device used for sorting and fast secondary storage access.

2. In the database literature, the actual domain of a database is also called its 'active' domain.

3. Other SQL statements (not beginning with `SELECT`) do not correspond to queries but to update or to schema definition commands.

4. The classical connective '$\supset$' is called 'material implication' in order to distinguish it from genuine implications in intuitionistic and other intensional logics. Genuine implications are not truth-functional and cannot be defined by other connectives. They are conceptually and computationally highly complex operations and therefore usually not included in knowledge systems.

5. Open queries are sometimes called *wh-queries* since in natural language they are formed by means of the interrogatives *who*, *what*, *when*, *where*.

6. This requirement has been called 'First Normal Form'. It is nowadays questioned as unnecessarily restrictive, sometimes leading to unnatural and inefficient representations.

3 OBJECT-RELATIONAL DATABASES

Object-relational databases (ORDBs) have evolved from relational databases by adding several extensions derived from conceptual modeling requirements and from object-oriented programming concepts. It is expected that the application of ORDBs will outnumber that of relational databases in the near future because they allow the seamless integration of multimedia data types and large application objects such as text documents, spreadsheets and maps, with the fundamental concept of database tables. Many object-relational extensions will be included in SQL3, the successor to SQL92.

Historically, the successful application of object-oriented programming languages such as Smalltalk, C++ and Java, has led to the development of a number of *object-oriented database (OODB) systems* which support the storage and manipulation of *persistent objects*. These systems have been designed as programming tools to facilitate the development of object-oriented application programs. However, although they are called 'database systems', their emphasis is not on knowledge representation but rather on persistent object management. **Any database concept which is intended as an implementation platform for information systems and knowledge representation must support *tables* as its basic representation concept on which query answering is based.** Tables correspond to extensional predicates, and each table row corresponds to a sentence or proposition. This correspondence is a fundamental requirement for true database systems. If it is violated, like in the case of OODBs, one deals with a new notion of 'database system', and it would be less confusing to use another term instead (e.g. 'persistent object management system') as proposed by Kim, 1995.

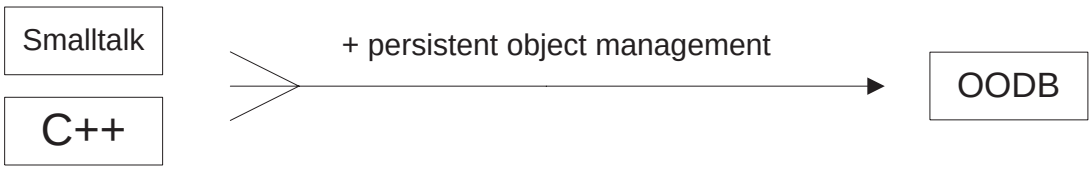

Figure 3.1. From C++ to object-oriented databases.

One can view the evolution of relational to object-relational databases in two steps:[1]

1. The addition of *abstract data types (ADTs)* leads to ADT-relational databases consisting of **complex-valued tables**. ADTs include *user-defined base types* and *complex types* together with user-defined functions and type predicates, and the possibility to form a *type hierarchy* where a subtype of a tuple type inherits all attributes defined for it.

2. The addition of *object identity*, *object references* and the possibility to define *subtables* within an extensional *subclass hierarchy* results in object-relational databases consisting of complex-valued tables and **object tables**.

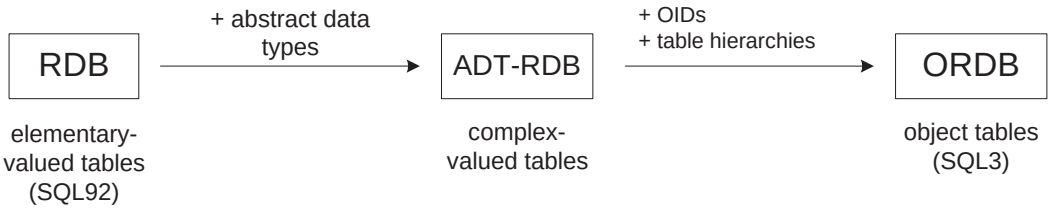

Figure 3.2. From relational to object-relational via ADT-relational databases.

There are two notable differences between object-relational databases and object-oriented programming:

1. Object IDs in ORDBs are *logical* pointers. They are not bound to a physical location (like C++ pointers).

2. In addition to the intensional subtype hierarchy of the type system, ORDBs have an extensional subclass (or subtable) hierarchy that respects the subtype relationships defined in their type system.

3.1 ADT-RELATIONAL DATABASES

In ADT-relational databases, attributes may be defined over base types or complex types. Base types may be standard types (such as INTEGER and CHAR) or **user-defined types** (such as DOCUMENT and SIGNATURE). While standard types come with standard functions (such as arithmetic operations and string functions) and with standard *type predicates* (such as equality and comparison operators), user-defined types come with user-defined functions (e.g. for extracting the initials of a signature or the key words of a document) and user-defined type predicates (e.g. for testing the equality or similarity of two signatures). Like in object-oriented programming, *polymorphism* is needed to use (or *overload*) the same symbol (such as '=' or '$\in$') for different types. Polymorphism, however, is not a conceptual issue, but rather an ergonomic one since it just allows a more natural syntax.

For simplicity, in the following exposition of ADT-relational databases, we disregard user-defined and also most built-in functions and type predicates as well as attribute functions (although they play an important role in practical query languages such as SQL).

3.1.1 Complex Values and Types

In addition to elementary values, ADT-relational databases allow for *complex values* formed with the help of *collection operators* such as $\{c_1, \ldots, c_k\}$ for sets and $\langle c_1, \ldots, c_l \rangle$ for tuples (or rows). Collection operators may be arbitrarily nested. Thus,

$$\langle 3, \{5, 9\}, \langle a, 2 \rangle, d, c \rangle$$

is an example of a complex value. A table (as a set of tuples of the same type) is also a complex value.

We assume a finite number of base types $T_1, T_2, T_3, \ldots$, such as integers (INT), strings (CHAR), or SIGNATURE. Complex (or collection) types are formed with the help of collection operators such as $\{\ldots\}$ for set types and $\langle\ldots\rangle$ for tuple (or row) types. Whenever $T, T_1, \ldots, T_k$ are types and $A_1, \ldots, A_k$ are attribute names, then $\{T\}$ and $\langle A_1 : T_1, \ldots, A_k : T_k\rangle$ are types. We sometimes omit attribute names in tuple types and write simply $\langle T_1, \ldots, T_k\rangle$. A named tuple type is a pair consisting of a type name T and a tuple type expression $\langle A_1 : T_1, \ldots, A_k : T_k\rangle$. It is also briefly denoted by $T(A_1, \ldots, A_k)$.

Definition 3.1 (Subtype) *A tuple type $\langle T_1, \ldots, T_k, \ldots, T_l\rangle$ is a* subtype *of $\langle T'_1, \ldots, T'_k\rangle$ if T_i is a subtype of T'_i for $i = 1, \ldots, k$. Trivially, any type is a subtype of itself. A set type $\{T\}$ is a subtype of $\{T'\}$ if T is a subtype of T'. The subtype relationship is also denoted by $\leq$.*

For instance, writing I for INT and C for CHAR,

$$\langle I, \{\langle I, C, I\rangle\}, \langle C, C\rangle, I\rangle \quad \leq \quad \langle I, \{\langle I, C\rangle\}, \langle C\rangle\rangle$$

In SQL3, we can declare the tuple type *prof_type* for professors as a subtype of *emp_type* with the additional attribute *Office* by means of the key word `UNDER`:

```
CREATE ROW TYPE addr_type(
Street      CHAR(20),
City        CHAR(20),
);
CREATE ROW TYPE emp_type(
SSN         INT,
Name        CHAR(20),
Address     addr_type
);
CREATE ROW TYPE prof_type UNDER emp_type(
Office      ROW( Building CHAR(20), Room CHAR(10)),
)
```

The *extension* of a type is the set of all possible values of that type. For example, the extension of the base type INT2 (integers with 2 byte representation) is the set

$$\mathrm{Ext}(\mathrm{INT2}) = \{-32768, \ldots, 32767\}$$

The extension of a set type $\{T\}$ is the set of all finite subsets of the extension of T:

$$\mathrm{Ext}(\{T\}) = \mathrm{Fin}(2^{\mathrm{Ext}(T)}) = \{V \subseteq \mathrm{Ext}(T) \mid V \text{ is finite}\}$$

The extension of a tuple type is the set of all suitable tuples:

$$\mathrm{Ext}(\langle T_1, \ldots, T_k\rangle) = \{\langle v_1, \ldots, v_k\rangle \mid v_i \in \mathrm{Ext}(T_i)\}$$

In the case of user-defined types, such as *emp_type*, one can distinguish between their *formal* and their *actual* extension. The formal extension of *emp_type* is the set of all INT/CHAR(20)/addr_type triples, while its actual extension with respect to a database is the set of all employees recorded in that database.

3.1.2 ADT Tables

A *complex-valued attribute* is a pair $\langle A, T\rangle$ such that A is the attribute name and T is its type which is also denoted type(A). The domain of an attribute A is the extension of its type:

$$\mathrm{dom}(A) = \mathrm{Ext}(\mathrm{type}(A))$$

An **ADT table schema** p consists of a sequence of complex-valued attributes $A_1, \ldots, A_n$ and a set of integrity constraints. A set of tuples P is a table instance over p if

$$P \subseteq \mathrm{dom}(A_1) \times \ldots \times \mathrm{dom}(A_n)$$

and P satisfies all integrity constraints of p. Notice that this definition subsumes relational database tables as a special case.

As an example, we define a table schema for institutes characterized by the attributes *Name*, *Address* and *Rooms*. While *Address* is a tuple-valued attribute composed of *Street* and *City*, *Rooms* is a table-valued (i.e. set-of-tuple-valued) attribute composed of *No* and *Floor*. Thus, a suitable tuple type is

$$\langle C30, \langle C20, C20\rangle, \{\langle C10, I\rangle\}\rangle$$

where Cn and I stand for CHAR(n) and INT. This is expressed in SQL3 as follows:

```
CREATE ROW TYPE inst_type(
Name      CHAR(30),
Address   addr_type,
Rooms     SET( ROW( No CHAR(10), Floor INT))
)
```

Finally, this type definition is used for declaring the institute table as

```
CREATE TABLE inst OF inst_type
```

This ADT table schema may then be populated as in Table 3.1.

Table 3.1. An ADT table for representing institutes.

Institute

Name	**Address**		**Rooms**	
	Street	**City**	**No**	**Floor**
AI Lab	Takustr. 3	Berlin	I	0
			II	0
Pathology	Garystr. 1	Berlin	101	1
			102	1
			205	2
Geophysics	Clayallee 7	Potsdam	1	5
			2	5

3.1.3 ADT Terms and Formulas

ADT terms comprise

1. elementary constants $c_1, c_2, \ldots$, which are values from base types;
2. variables $x, y, z, x_1, x_2, \ldots$, which are typed; the (possibly complex) type of a variable x is denoted by $\mathrm{type}(x)$;
3. set terms of the form $\{u_1, \ldots, u_k\}$;
4. tuple terms of the form $\langle u_1, \ldots, u_k\rangle$; and
5. path terms of the form $x.A$,

where $u_1, \ldots, u_k$ are ADT terms, x is a tuple variable and A is an attribute of $\mathrm{type}(x)$. An example of a path term is *i.Address.City* where i is a variable of type *inst_type* bound to a specific institute.

Atomic formulas comprise

1. constraint atoms of the form $u = u'$, $u \in u'$ and $u \subseteq u'$; and
2. domain atoms of the form $p(u_1, \ldots, u_n)$,

where $u, u', u_1, \ldots, u_n$ are ADT terms.

General ADT formulas are composed of atomic formulas, sentential connectives and quantifiers in the usual way. ADT if-queries are ADT sentences, and open ADT queries are ADT formulas with free variables. For instance,

$$\exists y \exists z(\text{inst}(x, y, z) \wedge y.\mathit{City} = \text{'Berlin'})$$

asks for the names of institutes located in Berlin. The same query could be expressed as

$$\exists y \exists z(\text{inst}(x, \langle y, \text{'Berlin'}\rangle, z))$$

All institutes with rooms on the ground floor are retrieved by

$$\exists y \exists z(\text{inst}(x, y, z) \wedge \exists z_1(\langle z_1, 0\rangle \in z))$$

3.1.4 ADT Queries Can Express Transitive Closure

In Section 2.5.5, we have seen that standard relational database queries (corresponding to SQL92 SELECT statements) cannot express the transitive closure of a binary predicate.

As shown in Abiteboul et al., 1995, the transitive closure of a binary relation Q referred to by the domain predicate q can be expressed by means of an ADT query formula $p(x_1, x_2)$ which requires that the pair $\langle x_1, x_2\rangle$ is in any binary relation that is transitively closed and contains Q:

$$\begin{array}{lcl} p(x_1, x_2) & \equiv & \forall y(\quad \forall z_1 \forall z_2 \forall z_3(\langle z_1, z_2\rangle \in y \wedge \langle z_2, z_3\rangle \in y \supset \langle z_1, z_3\rangle \in y) \\ & & \qquad \wedge \quad \forall z_1 \forall z_2(q(z_1, z_2) \supset \langle z_1, z_2\rangle \in y) \\ & & \qquad \supset \langle x_1, x_2\rangle \in y \\ & &) \end{array}$$

This shows that the addition of abstract data types leads to a much more complex and expressive framework.

3.2 INTRODUCING OBJECTS AND CLASSES

An **object** is an entity with an immutable standard name such that its identity is independent of its state (or value). More formally, an object is a pair $\langle o, v\rangle$ where o is an **object ID** (OID) and v is a (possibly complex) value.

For instance, in the object-relational database management system *Illustra*, described in Stonebraker and Moore, 1996, OIDs are 64-bit identifiers composed of a table ID and a tuple ID. Such an implementation scheme for OIDs, however, by binding the ID of an object to a specific table, makes it difficult for objects to 'migrate' from one table to another table with a related type (i.e. super- or subtype). *Object migration* is an important technical capability for supporting the implementation of *dynamic roles* such as, for instance, the roles of *engineer*, *salesperson* and *salesengineer*, which require that role changes (e.g., an engineer becoming a salesengineer) are straightforward. We assume therefore, that OIDs are assigned relative to the maximal supertype of the row type of an object table (notice that every table has exactly one such supertype). This naming scheme allows that objects can freely migrate between all tables whose row type is a subtype of that maximal supertype.

3.2.1 Classes as Object Tables

In addition to the base types $T_1, T_2, T_3, \ldots$ and to set and tuple types, in ORDBs there are **reference types** of the form $\sharp T$ where T is the name of a tuple type. Let T, T' be two tuple types such that T is the maximal supertype of T'. An object reference of type $\sharp T'$ consists of the type identifier T and an instance identifier relative to T. Thus, the formal extension of $\sharp T'$ is

$$\mathrm{Ext}(\sharp T') = \{\langle T, 1\rangle, \langle T, 2\rangle, \ldots, \langle T, n_T\rangle\}$$

where n_T is the system-defined maximal cardinality of the maximal supertype T. In the sequel, we also use the notation '$\sharp a1$' instead of $\langle a, 1\rangle$ for object IDs and references.

Object-relational attributes are defined in the same way as ADT attributes. An **object table schema** is a triple

$$\langle p, T_p(A_1, \ldots, A_n), IC_p\rangle$$

consisting of a table name p, a tuple type $T_p(A_1, \ldots, A_n)$ specifying a sequence of object-relational attributes $A_1, \ldots, A_n$, and a set of integrity constraints IC_p. A set of tuples P is an admissible **object table instance** over p if

$$P \quad \subseteq \quad \mathrm{Ext}(\sharp T_p) \times \mathrm{dom}(A_1) \times \ldots \times \mathrm{dom}(A_n)$$

such that the first column (called 'OID') is a key, and P respects all integrity constraints of p.

Notice that the first column of an object table instance is implicit and contains the object IDs of tuples. The tuples of an object table can be also called *objects* or *object tuples*, and object tables can be also called *classes*.

In order to be able to obtain the OID of an object tuple and to *dereference* an OID two new kinds of terms are introduced. If a tuple term u is bound to a tuple from an object table, then the term $\sharp u$ denotes the OID of that tuple. Conversely, if u is an OID-valued term of type $\sharp T$, and A is an attribute of the underlying tuple type T, then $u{\rightarrow}A$ denotes the value of A for the tuple with OID u.

To illustrate the use of references we reconsider the tuple type for employees. Since particular items of an address (such as the phone number or the street name) may change over time, and such a change would affect all occurrences of that address, it is preferable to use object references to addresses instead of the addresses themselves. In SQL3, a reference type is defined with the key word `REF`. Thus, we can redefine the employee tuple type as

```
CREATE ROW TYPE emp_type(
SSN       INT,
Name      CHAR(20),
AddrRef   REF(addr_type)
)
```

This type definition is used below in the definition of the employee object table.

An object table can be declared as a **subtable** of another object table if its tuple type is a subtype of that table's tuple type. This means that a subtable *inherits* all attributes from its supertables. Intuitively, a subtable declaration $p \leq q$ implies that p-objects are also q-objects. A **table hierarchy** is a set of object tables partially ordered by the subtable relationship $\leq$.

In SQL3, a subtable is declared with the key word `UNDER`. Using the row types *prof_type* and *emp_type* defined above, we can declare[2]

```
CREATE OBJECT TABLE emp OF emp_type;

CREATE OBJECT TABLE prof OF prof_type UNDER emp
```

These object tables may then be populated as follows:

Employee

OID	SSN	Name	AddrRef
$\sharp e1$	0H55C7	Horn	$\sharp a3$
$\sharp e2$	0B49B5	Bokowski	$\sharp a1$
$\sharp e3$	1S52C3	Spiegel	$\sharp a4$

Professor

OID	SSN	Name	AddrRef	Office	
				Building	Room
$\sharp e4$	1H66C1	Hegenberger	$\sharp a2$	Egon-Erwin-Kisch Inst.	223
$\sharp e5$	0W57I3	Wagner	$\sharp a2$	Einstein Tower	117

Notice that the subclass relationship between professors and employees is not materialized extensionally in these base tables (in the sense of set inclusion). Rather, it is realized intensionally in the query answering operation where the **inclusive extension** of an object table is determined by recursively collecting all objects from all subtables. Thus, asking for the names of all employees by means of the SQL query

`SELECT` Name `FROM` emp

results in the answer set {*Horn, Bokowski, Spiegel, Hegenberger, Wagner*}.

Unlike in relational databases, asking for all employees living in Potsdam does not require a join between the *Employee* and the *Address* tables, but is achieved by means of dereferencing the address OID:

`SELECT` Name `FROM` emp `WHERE` AddrRef->City = 'Potsdam'

Using references to access related data may be more efficiently implemented than the join operation. Notice that the join is still needed to evaluate general conjunctions.

3.2.2 Object-Relational Databases

An **ORDB schema** comprises ADT table schemas, object table schemas together with the definition of subtable relationships among them, and integrity constraints, i.e. it is a 5-tuple

$$\langle \Sigma, \langle p_1, \ldots, p_k; q_1, \ldots, q_l \rangle, \leq, IC \rangle$$

consisting of

1. a schema name Σ,
2. a finite sequence of ADT table schemas $\langle p_1, \ldots, p_k \rangle$,
3. a finite sequence of object table schemas $\langle q_1, \ldots, q_l \rangle$,
4. a partial order $\leq$ of $\{q_1, \ldots, q_l\}$, such that $\text{type}(q_i) \leq \text{type}(q_j)$ whenever $q_i \leq q_j$, and
5. a set of integrity constraints $IC \subseteq L_\Sigma$.

An **ORDB instance** Δ over a schema Σ consists of a sequence of suitable ADT table instances and a sequence of suitable object table instances. More precisely, an ORDB instance Δ over Σ is a pair of table sequences

$$\langle P_1, \ldots, P_k; Q_1, \ldots, Q_l \rangle$$

such that

1. each P_i is an ADT table instance over p_i, for $i = 1, \ldots, k$, and
2. each Q_j is an object table instance over q_j, for $j = 1, \ldots, l$.

Δ is called an *admissible instance* over Σ if it satisfies all integrity constraints of Σ. Notice that **object-referential integrity**, i.e. the property that all OID values of reference attributes correspond to an existing object tuple, must be enforced by stating the corresponding inclusion

dependencies in IC_Σ. For instance, in the employee table above, the values of the *AddrRef* attribute must be OIDs of existing addresses. This can be expressed by the inclusion dependency

$$\text{emp}[AddrRef] \subseteq \text{addr}[OID]$$

or by means of the key word `REFERENCES` in SQL3,

```
CREATE ROW TYPE emp_type(
SSN       INT,
Name      CHAR(20),
AddrRef   REF(addr_type) REFERENCES addr
)
```

In order to characterize the correct answers to a query we need to define an inference relation between ORDBs and object-relational if-queries. For this purpose, we first explain how to represent an ORDB as a set of atomic sentences and a set of *deduction rules*. While ADT and object tables are represented by corresponding sets of atomic sentences, subtable relationships are represented by a simple form of deduction rules, called **subsumption rules**.

For example, the database $\Delta = \langle P; Q_1, Q_2\rangle$ over corresponding table schemas $\langle p; q_1, q_2\rangle$ with $q_2 \leq q_1$, and with table instances

$$Q_1 = \begin{array}{|l|ll|}\hline \sharp q1 & \text{I} & a \\ \sharp q2 & \text{I} & a \\ \sharp q3 & \text{II} & b \\ \hline\end{array} \qquad P = \begin{array}{|c|}\hline \{a,b\} \\ \{c\} \\ \hline\end{array}$$

$$Q_2 = \begin{array}{|l|lll|}\hline \sharp q4 & \text{V} & c & \sharp q1 \\ \sharp q5 & \text{V} & d & \sharp q2 \\ \hline\end{array}$$

corresponds to a pair $\langle X_\Delta, R^{\leq}_\Delta\rangle$ where X_Δ represents the tables of Δ and $R^{\leq}_\Delta$ contains the subsumption rules representing the subtable relationships of Δ:

$$\begin{aligned} X_\Delta &= \{p(\{a,b\}), p(\{c\}), q_1(\sharp q1, \text{I}, a), q_1(\sharp q2, \text{I}, a), q_1(\sharp q3, \text{II}, b), \\ &\quad\ q_2(\sharp q4, \text{V}, c, \sharp q1), q_2(\sharp q5, \text{V}, d, \sharp q2)\} \\ R^{\leq}_\Delta &= \{q_1(x, y_1, y_2) \leftarrow q_2(x, y_1, y_2, y_3)\} \end{aligned}$$

The deduction rule $q_1(x, y_1, y_2) \leftarrow q_2(x, y_1, y_2, y_3)$ expresses the subsumption of Q_2 under Q_1: every q_2-object is a q_1-object. Its application to a table instance of q_2 derives a set of corresponding q_1-tuples, namely the projection of Q_2 to th attributes of Q_1.

Whenever $q \leq p$ and Q is the subtable instance containing the inclusive extension of q, then the exhaustive application of a subsumption rule

$$r : p(x, y_1, \ldots, y_m) \leftarrow q(x, y_1, \ldots, y_m, \ldots, y_n),$$

to the subtable instance Q generates the inclusive extension of p as the closure of P under the rule r,

$$P' = P \cup r(Q) = P \cup \text{Proj}(\langle x, y_1, \ldots, y_m\rangle, Q)$$

which materializes the supertable instance of p by adding all suitably projected tuples of its subtables. In this iterated bottom-up computation, starting with minimal subtables, each tuple of the subtable is projected to the attributes of the supertable and then added to it.

Thus, a set of subsumption rules $R^{\leq}$ can be viewed as a *closure operator*: its exhaustive application to the base facts in X generates a closure $R^{\leq}(X)$ which materializes the subtable hierarchy. The general concept of deduction rules is explained in the next chapter. Here, we only compute the closure of X_Δ under $R^{\leq}_\Delta$:

$$R^{\leq}_\Delta(X_\Delta) = X_\Delta \cup \{q_1(\sharp q4, \text{V}, c), q_1(\sharp q5, \text{V}, d)\}$$

We can now define that a sentence F can be inferred from an ORDB Δ if it can be inferred from its subsumption closure:

$$\Delta \vdash F \quad :\Longleftrightarrow \quad R^{\leq}_{\Delta}(X_{\Delta}) \vdash F$$

This definition leads to the desired *inclusive* answer set as illustrated in the following example, where the 1-tuple angles $\langle \ldots \rangle$ are omitted, for simplicity:

$$\begin{aligned} \mathsf{Ans}(\Delta, \exists y_1 \exists y_2 (q_1(x, y_1, y_2))) &= \{i \mid \Delta \vdash \exists y_1 \exists y_2 (q_1(i, y_1, y_2))\} \\ &= \{\sharp q1, \sharp q2, \sharp q3, \sharp q4, \sharp q5\} \end{aligned}$$

3.3 FURTHER READING

A theoretical presentation of ADT-relational and object-oriented databases can be found in Abiteboul et al., 1995. A standards proposal for object-oriented databases is presented in Cattell and Barry, 1997. Object-relational databases are discussed from the practitioner's perspective in Stonebraker and Moore, 1996.

3.4 EXERCISES

Problem 3-1. Define an ORDB schema implementing the ER model of the university domain of problem 1-1.

Problem 3-2. Express the following queries as formulas and as SQL3 statements:

1. Which institutes of Table 3.1 have rooms on the second floor ?

2. List all employees of the table on p. 58 having the same address.[3]

Notes

1. Historically, so-called 'non-first normal form' tables allowing for nested table-valued attributes have played an important role in the development of the concept of complex-valued tables.

2. We use `CREATE TABLE` for the definition of an ADT table schema, and `CREATE OBJECT TABLE` for the definition of an object table schema. SQL3 may not make this distinction.

3. Notice that there are two notions of equality in ORDBs. *Intensional* equality requires that all values and object references are equal, while *extensional* equality requires only that values are equal and object references refer to extensionally equal objects.

II Adding Rules

4 REACTION RULES

There are several types of rules playing an important role in knowledge and information processing systems. The most important ones are *reaction* and *deduction* rules. As opposed to simple database systems where the only reasoning service is query answering, more advanced systems offer in addition a number of advanced reasoning services such as deductive query answering by means of deduction rules, reactive input processing by means of reaction rules, and the abductive generation of explanations, diagnoses and plans (based on the ability to represent actions). In this chapter, we discuss the concept of *reaction rules* subsuming *production rules* and *database triggers* as special cases. We describe a simple model of communication events and show how the communicative behavior of a database can be specified with the help of reaction rules referring to incoming messages representing communication events. Finally, the communication between the member databases of a multidatabase system is discussed as a natural example for the application of reaction rules.

4.1 INTRODUCTION

Reaction rules can be used to specify the reactive behavior of a system in response to events. An event has to be 'perceived' by a system in order to react to it. Perceived events are represented by incoming messages. One may distinguish between **environment events**, related to changes in the (physical or virtual) environment, and **communication events**, related to communication acts of other systems.

While in traditional information systems, a large part of the procedural knowledge about the application domain is encoded in a multitude of application programs written in different programming languages by different programmers, reaction rules allow to specify procedural knowledge along with the schema in a central repository and independently of specific application programs. In this way, reaction rules help to capture more semantics of an application domain.

Reaction rules consist of a two-part body comprising an *event (or perception) condition* as well as an *epistemic condition*, and a resulting *action* specification in the head. The perception condition refers to the current state of the *event queue* while the epistemic condition refers to

the current state of the knowledge base (or database). Particular forms of reaction rules have been proposed under the names

- **'production rules'** in *expert systems*, such as in the paradigm-setting OPS5 (see Forgy, 1982);
- **'event-condition-action rules'** or database **triggers** in (so-called 'active') database systems;
- **'commitment rules'** in the *agent-oriented programming (AOP)* proposal of Shoham, 1993.

4.1.1 Production Rules

In production rules, used for building 'expert systems', there is no explicit perception condition since the implicit triggering events of a rule are those updates affecting its epistemic condition. For example, the following rule (from Stefik, 1995),

IF the highway light is green
AND the sensor shows a car waiting
AND the traffic timer shows time up

THEN turn the highway light yellow
AND start the light timer

may be used to control a traffic light. Here, an epistemic condition referring to the information state of the controler, such as *'the highway light is green'*, is not distinguished from a perception condition referring to an external event, such as represented by the sensor reading *'car waiting'*.

In order to capture more domain semantics and to obtain a more adequate system, however, it is necessary to make the fundamental conceptual distinction between the internal information (or knowledge) state and external events which, if they are noticed, change the perception state (realized, e.g., in the form of an event queue).

4.1.2 Database Triggers

Database triggers have also been called *event-condition-action* or *active* rules. They are useful for various purposes such as for

1. automated integrity maintenance,
2. maintaining derived attributes,
3. table replication in distributed databases,
4. implementing auditing and alert policies,
5. encoding 'business rules' in enterprise information systems.

Database triggers will be included in SQL3 where triggering events are update events comprising `INSERT`s, `DELETE`s and `UPDATE`s. An epistemic condition can be any SQL condition, and an action can be any SQL statement (except schema definition statements).

For instance, the following SQL3 statement

```
CREATE TRIGGER auditSalary
AFTER UPDATE OF Salary ON emp
FOR EACH STATEMENT
INSERT INTO audit VALUES( USER, CURRENT_DATE)
```

defines a trigger for auditing salary updates. It is triggered by any `UPDATE` of the salary attribute of the employee table creating an appropriate entry in the audit table.

Notice that database triggers have been defined as a practical extension to relational database systems. Lacking a firm semantic foundation including a theoretical account of actions and

events, it is no wonder that these systems run into several practical and theoretical problems. For instance, in some systems, if more than one rule is triggered by an event, the applicable rules are fired and reevaluated one by one in a certain (e.g., user-defined) order. This procedure, which was already used in the production rule system OPS5, leads to the successive execution of possibly conflicting rules with incompatible effects. It is not based on a logically justified method of conflict resolution and inconsistency handling. Also, SQL3 triggers may activate each other, thus creating the problem of possibly non-terminating sequences of rule activations. Since there are different proposals and implementations of database triggers, it is difficult to find a unifying operational semantics for them that can be related to a logical semantics of rules.

Instead of further discussing the specific problems of database triggers we prefer to present a more general concept of reaction rules together with a logical and operational semantics for them. Events, in this more general approach, are *external stimuli* such as incoming messages from human users and other systems representing communication events, or incoming data from perception subsystems if the environment is sensed in some way (e.g. for monitoring changes in the file system or for measuring the temperature of a room). We will show that the SQL3 triggering event types, which are limited to data modification commands, can be viewed as special cases of the communication event type *TELL*. This will also raise the question why SQL3 does not consider other event types such as *ASK* corresponding to SQL queries. Another important event type not available in SQL3 triggers is *REPLY* which is needed, for instance, in any interactive (or joint) query answering procedure such as in multidatabases or cooperative knowledge bases.

4.1.3 AGENT-0 Commitment Rules

In the agent-oriented programming language prototype AGENT-0, proposed in Shoham, 1993, the behavior of an agent is specified by *commitment rules* of the form

```
IF MessageCondition AND MentalCondition
THEN COMMIT Action TO Agent
```

While *message conditions* refer to incoming messages representing communication events, *mental conditions* refer to the mental state of the agent, that is, to its knowledge base. If such a rule is fired, the result is a new commitment towards another agent.

Like the general form of reaction rules proposed below, commitment rules refer both to knowledge and perception (or incoming messages). However, in their THEN-part, they are very specific, creating only commitments but no other mental effects. We will show that for specifying general reactive behavior, it is essential to allow for various kinds of epistemic (or mental) effects and, in particular, for belief updates.

By referring to incoming messages, AGENT-0 commitment rules emphasize the importance of perception and communication.

4.2 COMMUNICATION EVENTS

Reactive knowledge systems which are able to communicate in an application-independent language based on typed messages that represent communication acts are examples of **knowledge- and perception-based agents** (see Wagner, 1996b). In Genesereth and Ketchpel, 1994, an informal definition of software agents, emphasizing their ability to communicate with each other, is proposed:

> *An entity is a software agent if and only if it communicates correctly in an agent communication language (ACL) [...].*

Such a language is based on **typed messages**. It is essential that the language contains message types for capturing all basic communication acts. In contrast to the application-specific messages in object-oriented programming, ACL message types are application-independent and, in combination with standardized dictionaries defining the vocabulary of a domain (called

'ontologies'), allow true software interoperability. The correspondence between ACL message types and SQL statements is described in table 4.1.

The set of possible communication acts includes *telling*, *asking*, *replying*, and more. As noted in Peirce, 1976, different communication acts may have the same propositional content:

> One and the same proposition may be affirmed, denied, judged, doubted, inwardly inquired into, put as a question, wished, asked for, effectively commanded, taught, or merely expressed, and does not thereby become a different proposition.

High-level concepts of communication should be based on the *speech act theory* of Austin, 1962, and Searle, 1969, an informal theory within analytical philosophy of language. The essential insight of speech act theory is that an utterance by a speaker is, in general, not the mere statement of a true or false sentence, but rather an *action* of a specific kind (such as an assertion, a request, a promise, etc.).

For simplicity, we identify the communication act of a 'speaker' with the corresponding communication event which is perceived by the addressee of the communication act.[1] Communication events and environment events are represented by typed messages. In the sequel, we will restrict our attention to communication events. Message types for expressing those communication events needed in interactive query answering include TELL, ASK-IF/REPLY-IF and ASK-ALL/REPLY-ALL.

The use of *TELL* for supplying new information is related to the data modification commands of SQL. An SQL `INSERT` of a new row $\langle a, b, c\rangle$ into the table p corresponds to sending a *TELL(p(a,b,c))* message with the atomic sentence $p(a, b, c)$ as content. Sending a *TELL*$(\neg p(a, b, c))$ with the negated sentence $\neg p(a, b, c)$ as content corresponds to an SQL `DELETE` of the respective row. The latter correspondence assumes, however, that the predicates involved are complete, like in a relational database.

Similarly, *ASK-ALL*$(p(x, y, z))$ corresponds to the SQL query statement SELECT x, y, z FROM p which delivers the collection of all answer substitutions in the form of a table. An *ASK-IF* leads to an if-answer such as *yes*, *no*, or *unknown*.

Table 4.1. Correspondences between communication acts and SQL.

A C L	**S Q L**
TELL$(p(a, b, c))$	INSERT INTO p VALUES (a, b, c)
TELL$(\neg q(a))$	DELETE FROM q WHERE $x = a$
ASK-IF$(q(a))$	SELECT 'yes' FROM q WHERE $x = a$
ASK-IF$(\neg q(a))$	SELECT 'yes' FROM q WHERE NOT EXISTS(SELECT * FROM q WHERE $x = a$)
ASK-ALL$(p(x, y, z))$	SELECT x, y, z FROM p

The **message content** parameter of TELL is an input sentence, while that of ASK-IF is an if-query. REPLY-IF takes two parameters: the first one repeats the if-query to which the reply refers, and the second one is the corresponding if-answer (such as yes, no, etc.). Examples of communication events expressed with these message types are:

- TELL$(\neg$stud$(0123, \text{Peter}, \text{CompSc}))$,
  that is, the system tells (or is told) that Peter does no longer study computer science;
- ASK-IF$(\exists x(\text{att}(0123, x)))$,
  that is, the system asks (or is asked) if the student with number 0123 attends any course;
- REPLY-IF$(\exists x(\text{att}(0123, x))$, no),
  that is, the system replies (or is replied to) that 0123 does not attend any course.

We will only consider elementary events. From the perspective of an external observer, a communication event is fully specified by a receiver i, a sender j and a message $m(c)$ with

message type m and content c, i.e. as a triple $\langle i, m(c), j\rangle$. However, by default, we will assume the perspective of a reactive system which views a perceived communication event as a pair $\langle m(c), j\rangle$ consisting of a message $m(c)$ and a sender j, and its own communication acts as pairs $\langle m(c), i\rangle$ composed of a message $m(c)$ and an addressee i.

4.3 EPISTEMIC AND COMMUNICATIVE REACTIONS

Reaction rules encode the behavior of a reactive knowledge system in response to perception events created by its perception subsystems, and to communication events created by communication acts of other systems. We consider two forms of reaction rules: one for *epistemic reactions* where the action in response to an event consists only of an update of the information or knowledge state, and another one for *communicative reactions* where the action in response to an event consists of a communication act (realized by sending a message) and an associated update.[2] In both cases, the update is specified by a logical formula *Eff* which is called the *epistemic effect* of the reaction.

An **epistemic reaction rule** is a triple of the form

$$\mathit{Eff} \leftarrow \mathsf{recvMsg}(m(c), j), \mathit{Cond}$$

consisting of an epistemic condition *Cond*, an event condition $\mathsf{recvMsg}(m(c), j)$, and an epistemic effect *Eff*. The epistemic condition *Cond* is an evaluable query formula referring to the knowledge state of the reactive system. The event condition $\mathsf{recvMsg}(m(c), j)$ is a test whether the event queue contains the message $m(c)$ sent by j. The epistemic effect *Eff* is an input formula which has to be assimilated into the knowledge base of the reactive system implying that its free variables are all among the free variables of *Cond*. The epistemic condition may be omitted (technically, this is equivalent to setting $\mathit{Cond} = \mathsf{true}$).

Similarly, a **communicative reaction rule** is a 4-tuple of the form

$$\mathsf{sendMsg}(m'(c'), i), \mathit{Eff} \leftarrow \mathsf{recvMsg}(m(c), j), \mathit{Cond}$$

consisting, in addition to the components of an epistemic reaction rule, of an expression $\mathsf{sendMsg}(m'(c'), i)$ specifying the communication act of sending the message $m'(c')$ to i. There may be no epistemic effect in a communicative reaction (technically, this is equivalent to setting $\mathit{Eff} = \mathsf{true}$).

In SQL-style syntax, a communicative reaction rule may be expressed as follows:

```
ON IncomingMessage FROM Sender
IF Condition
THEN ⟨ data modification statements ⟩
SEND OutgoingMessage TO Addressee
```

In general, reactions are based both on perception and on knowledge. Immediate reactions do not allow for deliberation (by evaluating queries). They are represented by rules without an epistemic condition, i.e. where $\mathit{Cond} = \mathsf{true}$. Timely reactions can be achieved by guaranteeing fast response times for checking the precondition of a reaction rule. This will be the case, for instance, if the precondition can be checked by simple table look-up such as in relational databases. It will be more difficult in knowledge systems with deduction rules.

Before presenting the operational semantics of reaction rules, we discuss a number of fundamental examples related to communication.

4.3.1 TELL

The following epistemic reaction rule expresses a straightforward reaction in response to a TELL communication event:

$$F \leftarrow \mathsf{recvMsg}(\mathrm{TELL}(F), j)$$

Here, F is a meta-variable for input sentences. This rule expresses the policy that each new piece of information F, no matter by whom it is told, will be accepted and assimilated into the database or knowledge base. Such a policy may be justified by the assumption that all information suppliers are authorized 'users' that are competent, honest and reliable. If one cannot make this assumption, more involved rules, taking the competence and reliability of information suppliers into account, are needed. Notice that information suppliers may be human users, application programs or other communicating systems. The concept of *agent systems* provides a suitable abstraction for subsuming all these cases.

Two specific instances of the above rule are

$$\begin{aligned} \neg\mathrm{stud}(x,y,z) &\leftarrow \mathsf{recvMsg}(\mathrm{TELL}(\neg\mathrm{stud}(x,y,z)),j) \\ \mathrm{stud}(x,y,z) &\leftarrow \mathsf{recvMsg}(\mathrm{TELL}(\mathrm{stud}(x,y,z)),j) \end{aligned}$$

While the first of these rules takes care of requests to delete student tuples, the second one takes care of insert and modify requests. Instead of allowing users and application programs to directly execute the SQL data modification commands `INSERT`, `DELETE` and `UPDATE`, it seems to be more adequate to handle the interaction with them by means of specifiable reaction rules.

A database trigger is a special case of a reaction rule. If, for instance, UPDATEs of *student* data are to be audited, this could be specified by the reaction rule

$$\mathrm{audit}(x,j) \leftarrow \mathsf{recvMsg}(\mathrm{TELL}(\mathrm{stud}(x,y,z)),j),\ \exists u\exists v(\mathrm{stud}(x,u,v))$$

which inserts the number of the student concerned by the update together with the information source j into the *audit* table. Notice that the condition $\exists u\exists v(\mathrm{stud}(x,u,v))$ tests whether the TELL message corresponds to an update of an existing *student* record. If INSERTions of new student records into the *student* table are to be audited, the corresponding reaction rule would be

$$\mathrm{audit}(x,j) \leftarrow \mathsf{recvMsg}(\mathrm{TELL}(\mathrm{stud}(x,y,z)),j),\ \neg\exists u\exists v(\mathrm{stud}(x,u,v))$$

The integration of general reaction rules into SQL would require to replace the current SQL client/server application programming interface by an event handling interface capable to handle general communication events. Similar to the SQL standardization effort, there is an attempt to standardize communication events: the *FIPA'97 Agent Communication Language.*[3]

4.3.2 ASK-IF

The general reaction pattern in response to an ASK-IF communication event is specified by means of several reaction rules, one for each possible if-answer:

$$\begin{aligned} \mathsf{sendMsg}(\text{REPLY-IF}(F,\mathsf{yes}),j) &\leftarrow \mathsf{recvMsg}(\text{ASK-IF}(F),j),\ F \\ \mathsf{sendMsg}(\text{REPLY-IF}(F,\mathsf{no}),j) &\leftarrow \mathsf{recvMsg}(\text{ASK-IF}(F),j),\ \neg F \end{aligned}$$

where F is a meta-variable for if-queries

Notice that these two rules suffice to specify the reaction to ASK-IF only if the system has complete information about the items it is asked about. Otherwise, at least a third rule for the case of *unknown* information would be needed. We will show in section 11.3.1 how a rule condition can express the test if a sentence is unknown.

As in the case of TELL, the straightforward communicative behavior specified by these rules is only appropriate if certain assumptions can be made. In particular, it has to be assumed that the asker is sufficiently authorized to obtain the requested information. Like in the case of information suppliers, askers may be all kinds of agents such as human users and communication-enabled application programs or databases. Notice, however, that a reactive system with a standard relational database as its underlying knowledge system will never issue a query since it assumes to have complete information, and consequently, it immediately answers any if-query by either *yes* or *no* without checking with any other system. But in a multidatabase system, a local table of a member database need not contain the entire extension

of the corresponding predicate, and the global schema may specify predicates for which there is no local table. In both cases, the member database, on receiving a query involving such predicates, has to forward it to other members and wait for their reply.

Obviously, it may be desirable to audit not only data modifications, as with SQL3 triggers, but also queries. The following reaction rule specifies such an audit rule for if-queries on the *employee* table:

$$\text{audit}(x, j) \quad \leftarrow \quad \mathsf{recvMsg}(\text{ASK-IF}(\text{emp}(x, y_1, y_2, y_3)), j)$$

4.3.3 REPLY-IF

Receiving a REPLY-IF message implies that an ASK-IF has been issued to the sender of that message before. For instance, a reactive system may receive a reply from a human user it has asked something. Or in a multidatabase system, it may receive a reply from another system it has consulted before. It may then react in two ways. Either it stores the received answer in its knowledge base, or it just forwards it to another system (or to a human user) that has asked about it before.

Storing an if-answer is specified by the following two rules:

$$\begin{array}{rcl} F & \leftarrow & \mathsf{recvMsg}(\text{REPLY-IF}(F, \mathsf{yes}), j) \\ \neg F & \leftarrow & \mathsf{recvMsg}(\text{REPLY-IF}(F, \mathsf{no}), j) \end{array}$$

Forwarding an if-answer is specified by

$$\begin{array}{rcl} \mathsf{sendMsg}(\text{REPLY-IF}(F, A), \mathit{Asker}) & \leftarrow & \mathsf{recvMsg}(\text{REPLY-IF}(F, A), j), \\ & & \text{ifquery}(F, \mathit{Asker}) \end{array}$$

where A is a meta-variable for an if-answer, *ifquery* is a system predicate for recording queries, and *Asker* is a variable to be instantiated with the ID of askers.

4.4 OPERATIONAL SEMANTICS OF REACTION RULES

A **reactive database** (or *reactive knowledge base*) is the result of adding suitable reaction rules to a database (or knowledge base). A reactive knowledge base may be viewed as a *knowledge- and perception-based agent* in the sense of Wagner, 1996b. This characterization is particularly appealing for a reactive knowledge base, if it masters inter-agent communication according to standards such as the FIPA agent communication language. The communication acts defined by FIPA can be implemented by means of reaction rules in a similar way as we have defined the communicative reactions to TELL, ASK-IF, and REPLY-IF events.

Conceptually, a reactive knowledge base consists of three components:

1. a database or *knowledge base* X,
2. an *event queue* EQ, i.e. a buffer receiving messages from other systems or from perception subsystems running as concurrent processes,[4]
3. a set of *reaction rules* RR encoding the reactive and communicative behavior.

Reaction rules are triggered by incoming messages representing events. The event queue of the reactive system is continuously checked. If there is a new event message, it is matched with the event condition of all reaction rules, and the epistemic conditions of those rules matching the event are evaluated. If they are satisfiable in the current knowledge or database state, all free variables in the rules are instantiated accordingly resulting in a set of epistemic effects and a set of triggered communication acts with associated epistemic effects. All these communication acts are then executed (by sending messages to other agents), and the epistemic effects are assimilated into the knowledge base.

Informally, the execution model of a reactive knowledge base consists of the following steps:

1. Get the next event from the event queue EQ, and check whether it triggers any reaction rules by matching it with the event conditions of all rules in RR. If it cannot be matched with any of these conditions, that is, if no rule is triggered, then repeat 1, else continue.

2. For each of the triggered reaction rules, evaluate its epistemic condition C like a query. For each answer substitution $\sigma \in \mathsf{Ans}(X, C)$, form the corresponding action/effect pair by instantiating all free variables in the outgoing message term and in the effect formula E accordingly, and collect all such pairs in a candidate set (in the case of an epistemic reaction rule, the 'empty' action *noAct* is used to form the action/effect pair $noAct/E\sigma$).

   If this candidate set contains any actions with incompatible effects, apply some conflict resolution procedure (e.g., replace it by the intersection of all maximally consistent subsets of it).

3. For each of the resulting action/effect pairs, perform the communication act, if there is one, by sending the specified message to the specified addressee, and assimilate the epistemic effect $E\sigma$ into the knowledge base X yielding $\mathsf{Upd}(X, E\sigma)$.

4. Continue with step 1.

4.5 MULTIDATABASES

A multidatabase is an ensemble of (possibly inhomogeneous) databases with a global distribution schema known to each participating database. The distribution schema stipulates for each predicate (or table) of the global schema which member databases are competent to answer queries, or to perform updates, involving that predicate. By sharing a global schema, the participating databases form a logical unit and may be viewed as a single entity.

The databases participating in a multidatabase system are natural examples of reactive systems. They communicate with each other by passing messages over the links of a network. Their communicative behavior can be specified by means of reaction rules.

As an example, we consider a two-database system, where we assume that two relational databases Δ_a and Δ_b, e.g. of two different cities, form a multidatabase. Each of them knows about which predicates it has complete information by means of the system catalog table *tanscomp* (standing for *totally answer competent*) that contains all predicates for which the database completeness assumption holds locally. We consider only two predicates: r and m, standing for *resident* and *married*. Natural predicates are often *indexical*, that is, the confirmation that they apply to a specific tuple depends on the context in which they are used. Notice that while the one-place predicate *resident* is geographically indexical, the two-place predicate *married* is temporally indexical.

Since each city has complete information about its residents, both databases contain the meta-information $tanscomp(r)$. However, this does not hold for m, since one can get married in any city, and this may be recorded only there.

The communication language used in this example consists of the two message types ASK-IF(F) and REPLY-IF(F, A), where F is an if-query and A is an if-answer. There are two possible originators of a query/message: a local *user*, or the *other* database. For simplicity, we restrict our attention to (existentially quantified) atomic if-queries, for which a database is answer competent, if it is answer competent for the involved predicate. The communication protocol for answering atomic if-queries in a two-database system consists of six common interaction rules which are listed in Figure 4.1.

Each of the two reactive databases forms a triple

$$\langle X, EQ, \{r_1, \ldots, r_6\}\rangle,$$

consisting of a database state X, an event queue EQ, and a reactive behavior program $\{r_1, \ldots, r_6\}$. For instance, rule 2 says that if a local user asks, if F, and the database is

r_1 : sendMsg(REPLY-IF(F, yes), *user*)
$\leftarrow$ recvMsg(ASK-IF(F), *user*), F

r_2 : sendMsg(REPLY-IF(F, no), *user*)
$\leftarrow$ recvMsg(ASK-IF(F), *user*), $\neg F \wedge \text{tanscomp}(F)$

r_3 : sendMsg(ASK-IF(F), *other*)
$\leftarrow$ recvMsg(ASK-IF(F), *user*), $\neg F \wedge \neg\text{tanscomp}(F)$

r_4 : sendMsg(REPLY-IF(F, yes), *other*)
$\leftarrow$ recvMsg(ASK-IF(F), *other*), F

r_5 : sendMsg(REPLY-IF(F, no), *other*)
$\leftarrow$ recvMsg(ASK-IF(F), *other*), $\neg F$

r_6 : sendMsg(REPLY-IF(F, A), *user*)
$\leftarrow$ recvMsg(REPLY-IF(F, A), *other*)

Figure 4.1. Distributed query answering in a two-database system.

totally answer-competent about F and infers that $\neg F$ holds, then it immediately replies with *no*. However, if it is not totally answer-competent about F and infers $\neg F$, then, according to rule 3, the database forwards the query to the other database (and waits for its reply).

For demonstrating a run of the system, we assume that in Δ_a, Susan (S) and Peter (P) are residents, Peter is married to Linda (L) and Tom (T) is married to Susan; and in Δ_b, Mary and Tom are residents, and Peter is married to Susan. A user at Δ_b asks if *Tom is married*, i.e. Δ_b receives the message ASK-IF($\exists x(m(T,x))$) from *user*. This yields the following initial states:

$$\begin{aligned} X_a^0 &= \{\text{tanscomp}(r),\ r(S), r(P), m(P,L), m(T,S)\} \\ EQ_a^0 &= [] \\ X_b^0 &= \{\text{tanscomp}(r),\ r(M), r(T), m(P,S)\} \\ EQ_b^0 &= [\langle \text{ASK-IF}(\exists x(m(T,x))), \mathit{user}\rangle] \end{aligned}$$

where the Prolog list notation is used for event queues.[5]

We obtain the following history of transitions:

1. The event $\langle \text{ASK-IF}(\exists x(m(T,x))), \mathit{user}\rangle$ at Δ_b triggers the rules r_1, r_2 and r_3. Only the condition of r_3 holds:
$$X_b^0 \vdash \neg\exists x(m(T,x)) \wedge \neg\text{tanscomp}(m)$$
and thus, only r_3 is fired by sending the message ASK-IF($\exists x(m(T,x))$) to the *other* database where it is buffered in the event queue EQ_a yielding
$$EQ_a^1 = [\langle \text{ASK-IF}(\exists x(m(T,x))), \mathit{other}\rangle]$$

2. This triggers r_4 and r_5 at database a of which only r_4 is applicable, since $X_a^1 \vdash \exists x(m(T,x))$. By applying r_4, a positive reply is sent to b:
$$EQ_b^2 = [\langle \text{REPLY-IF}(\exists x(m(T,x)), \text{yes}), \mathit{other}\rangle]$$

3. Finally, r_6 is triggered at database b and the answer received from a is passed to the *user*.

Notice that in the general case of more than two databases, there has to be additional meta-information on which other database to ask about a specific predicate, that is, the global

distribution schema has to be represented more explicitly than achieved by the *tanscomp* metapredicate in the two-database system.

The main correctness property of the distributed query answering protocol specified by the rules $r_1, \ldots, r_6$ is expressed by the following observation.

Observation 4.1 *After any update of one of the two databases Δ_a and Δ_b by a new fact F, no matter which of the two databases is asked if F, the answer must be* yes.

For guaranteeing correct answers in a multidatabase system, the underlying distributed answering protocol should be formally verified. We show in Section 14.1 how the formal verification method of *assertional reasoning* can be applied to reactive knowledge systems such as multidatabases and cooperative knowledge bases.

4.6 SUMMARY

Reaction rules allow the declarative specification of reactive systems. In particular, they can be used to program the communication between knowledge bases in the form of a *communication protocol*, that is, a set of communicative reaction rules whose application depends both on incoming messages representing communication events, and on the current state of the database. Through their declarative character, reaction rules also facilitate the formal verification of correct behavior.

4.7 FURTHER READING

Triggers in so-called 'active databases' are treated in Dayal et al., 1995; Zaniolo et al., 1997. In Brownston et al., 1985, production rules are used for programming 'expert systems'.

4.8 EXERCISES

Problem 4-1. Given the library database schema of 2.17.1, add a table *BookReturned* over the schema $br(InvNo, Date)$ for recording return events and write a reaction rule that takes care of book returns communicated by means of $TELL(\mathrm{br}(x, y))$ events.

Problem 4-2. Write a protocol (i.e. a set of reaction rules) that handles the distributed answering of open queries in the form of $ASK\text{-}ALL(F)$ events in the two-database system where F is an open query. Describe the transitions resulting from submitting the query $m(P, x)$ asking *'who is married to Peter ?'*, and compute the answer.

Problem 4-3. Write a set of reaction rules for modeling the communication taking place at R. Smullyan's Island of Knights and Knaves where knights always make true statements and knaves always make false statements (see Smullyan, 1987). Hint: imagine that you arrive at the island and have to ask questions. Write reaction rules for Knights and Knaves in response to ASK-IF events, and for yourself in response to REPLY-IF events. You may assume to know who is a Knight and who is a Knave.

Notes

1. This simplifying assumption implies that we abstract away from the *communication channel* and the asynchronicity created by the message transport delay.

2. A third kind of reaction rule concerns physical reactions which are essential for robots and other physically embodied or embedded agents.

3. See `http://drogo.cselt.stet.it/fipa/`.

4. Such a perception subsystem may, for instance, be a UNIX process monitoring some part of the file system and providing information about relevant changes.

5. In Prolog, [] denotes the empty list, $[H|B]$ denotes a list with head H and body B, and $[a_0, a_1, \ldots, a_n]$ denotes a list with head a_0 and body $[a_1, \ldots, a_n]$.

III Positive Knowledge Systems: Concepts, Properties and Examples

6 PRINCIPLES OF POSITIVE KNOWLEDGE SYSTEMS

In this chapter, we summarize the basic concepts of positive knowledge systems and call a knowledge system *vivid* if it is upward compatible with $\boldsymbol{A}$, the system of relational databases. We also discuss the difference between knowledge update and knowledge integration.

6.1 INTRODUCTION

A *knowledge system (KS)* consists essentially of two main components: an inference-based *query answering* and an *update* operation, forming the interface between knowledge bases and their 'users' who may be natural or artificial agents that ask questions and tell new facts.

In positive knowledge systems, information units are always consistent with each other, and therefore knowledge bases are *inherently consistent*, i.e. it is never the case that both a sentence F and its negation $\neg F$ can be inferred from a positive knowledge base.

Prominent examples of positive knowledge systems are the systems of relational, object-relational, temporal and deductive databases. A number of further important examples, including fuzzy, multi-level secure, lineage and disjunctive databases, are presented. All of them can be extended by admitting negative information in *bitables* leading to relational, object-relational, temporal, fuzzy, etc. *factbases*. This extension is the subject of Part IV.

6.2 BASIC CONCEPTS

The language of positive knowledge systems contains the sentential connectives conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), material implication ($\supset$), quantifiers ($\exists, \forall$), and the truth constant true; predicates such as $p, q, \ldots$; constant symbols such as $c, d, \ldots$; and variables such as $x, y, \ldots$. For simplicity, we do not consider functional terms but only variables and constants.

The **schema** Σ of a positive knowledge base consists of a finite sequence of predicates (or table schemas), a set of integrity constraints, and possibly further components:

$$\Sigma = \langle \langle p_1, \ldots, p_m \rangle, IC, \ldots \rangle$$

The **formal domain** (or *universe of discourse*) D_Σ associated with Σ is the union of all domains $\mathrm{dom}(A_i)$ of all attributes A_i of all predicates p_j in Σ. A schema Σ defines a predicate logic signature $\langle \mathrm{Pred}_\Sigma, \mathrm{Const}_\Sigma \rangle$ without function symbols, and a corresponding formal language

L_Σ, where $\text{Pred}_\Sigma = \{p_1, \ldots, p_m\}$, and $\text{Const}_\Sigma = D_\Sigma$, i.e. values are identified with constant symbols.

An **atom** a is an atomic formula, it is called *proper*, if $a \neq \mathsf{true}$. We use $a, b, \ldots$, and $F, G, H, \ldots$ as metavariables for atoms and well-formed formulas, respectively. A variable-free expression is called **ground**. The set of all proper atoms of a given language L_Σ is denoted by At_Σ, while $Lit_\Sigma \subseteq L_\Sigma$ denotes the set of all **literals**, i.e. atoms a or negated atoms $\neg a$. If $\mathcal{F}$ is a set of connectives, say $\mathcal{F} \subseteq \{\mathsf{true}, \neg, \wedge, \vee, \supset, \exists, \forall, \ldots\}$, then $L(\Sigma; \mathcal{F})$ denotes the corresponding set of well-formed formulas over Σ. For simplicity, we often omit the reference to Σ and write simply At, Lit, or $L(\mathcal{F})$. Where L is a language (a set of formulas), L^0 denotes its restriction to closed formulas (sentences). $\text{Free}(F)$ denotes the set of free variables of a formula F.

On the basis of a schema Σ, four languages are formed: the set of all admissible knowledge bases L_{KB}, the query language L_{Query}, the answer language L_{Ans}, and the input language L_{Input}. L^0_{Input} is the set of all admissible input sentences representing new information a knowledge base may be updated with. Elements of L^0_{Query} are called **if-queries**, and elements of L^0_{Ans} are called **if-answers**.

Definition 6.1 (Knowledge System) *A knowledge system* $\boldsymbol{K}$ *is a septuple*[1]

$$\boldsymbol{K} = \langle L_{KB}, \vdash, L_{\text{Query}}, \mathsf{Ans}, L_{\text{Ans}}, \mathsf{Upd}, L_{\text{Input}} \rangle$$

where $\vdash \subseteq L_{KB} \times L^0_{\text{Query}}$ *is called* inference relation,

$$\mathsf{Ans} : L_{KB} \times L_{\text{Query}} \to L_{\text{Ans}}$$

is called answer operation, *and*

$$\mathsf{Upd} : L_{KB} \times L^0_{\text{Input}} \to L_{KB}$$

is called update operation, *satisfying for any* $X \in L_{KB}$,

(KS0) $X \vdash \mathsf{true}$, *and* $\mathsf{Upd}(X, \mathsf{true}) = X$.

(KS1a) $X \vdash F$ *iff* $\mathsf{Ans}(X, F) = \mathsf{yes}$.

(KS1b) $X \vdash G[c_1, \ldots, c_n]$ *if* $\langle c_1, \ldots, c_n \rangle \in \mathsf{Ans}(X, G[x_1, \ldots, x_n])$

(KS2) $L_{\text{Input}} \subseteq L_{\text{Query}}$

(KS3) $\mathsf{Upd}(X, H) \vdash H$

where $F \in L^0_{\text{Query}}$, $G \in L_{\text{Query}}$, and $H \in L^0_{\text{Input}}$. If elements of L_{KB} are finite sets (resp. structures), $\boldsymbol{K}$ is called *finitary*.

(KS1b) expresses the **soundness** property of the answer operation, and (KS3) expresses the **success postulate**. Clearly, the converse of (KS1b), the completeness of Ans, is also desirable. But, on practical grounds, we allow the answer operation of a knowledge system to be incomplete (as in relational databases with nulls).

The set of all if-answers (i.e. answers to if-queries) is denoted by L^0_{Ans}. It contains at least yes and no. On the basis of the inference relation, an inference operation C can be defined:

$$C(X) = \{F \in L^0_{\text{Query}} \mid X \vdash F\}$$

In general, not all open query formulas can be answered sensibly. We therefore require that queries are *evaluable* (see 2.5.1). Answers to evaluable queries can be computed by means of algebraic operations on tables.

It is useful to be able to update a KB by a set of inputs and we therefore 'overload' the symbol Upd to denote also this more general update operation

$$\mathsf{Upd} : L_{KB} \times 2^{L^0_{\text{Input}}} \to L_{KB}$$

which has to be defined in such a way that for any finite $A \subseteq L^0_{\text{Input}}$,

$$\mathsf{Upd}(X, A) = \mathsf{Upd}(X, \bigwedge A)$$

We sometimes write $X + F$ as an abbreviation of $\mathsf{Upd}(X, F)$, resp. $X - F$ as an abbreviation of $\mathsf{Upd}(X, \neg F)$.

In the following chapters, we present several examples of basic positive knowledge systems. The simplest one is the knowledge system of relational databases where only ordinary predicates are allowed and complete information about them is required. Other knowledge systems allow qualified predicates, such as uncertainty or temporally qualified predicates, or incomplete information, such as in disjunctive databases.

The KS of relational databases, denoted by $\boldsymbol{A}$, is defined as

$$\boldsymbol{A} \;=\; \langle 2^{\text{At}}, \vdash, L(\neg, \wedge, \vee, \exists, \forall), \mathsf{Ans}, \{\mathsf{yes}, \mathsf{no}\}, \mathsf{Upd}, \text{Lit}\rangle$$

We also describe a KS by means of its language table:

$\boldsymbol{A} =$

L_{KB}	2^{At^0}
L_{Query}	$L(\neg, \wedge, \vee, \exists, \forall)$
L^0_{Ans}	$\{\mathsf{yes}, \mathsf{no}\}$
L_{Input}	Lit

We denote the positive fragment of $\boldsymbol{A}$ by $\boldsymbol{A}^+$:

$\boldsymbol{A}^+ =$

L_{KB}	2^{At^0}
L_{Query}	$L(\wedge, \vee, \exists)$
L^0_{Ans}	$\{\mathsf{yes}, \mathsf{no}\}$
L_{Input}	At

The above definition of a knowledge system as a septuple does not yet contain all components needed. In order to compare knowledge bases in terms of their information (or knowledge) content we assume that there is an **information order** (or knowledge order) $\leq$ between knowledge bases such that

$$X \leq Y \quad \textit{if } X \textit{ contains at least as much information as } Y.$$

The information order should be defined in terms of the structural components of knowledge bases and not in terms of higher-level notions (like derivability).[2] The *informationally empty* KB is denoted by 0. By definition, $0 \leq X$ for all $X \in L_{KB}$, i.e. 0 is the least element of $\langle L_{KB}, \leq\rangle$.

The information order also applies to answers. Notice that an if-answer $a = \mathsf{Ans}(X, F)$ to an if-query formula F can be used to transform F into a sentence F'. For this purpose we define a function

$$\text{val} : L^0_{\text{Query}} \times L^0_{\text{Ans}} \to L^0_{\text{Query}}$$

by setting

$$\text{val}(F, a) = \begin{cases} F \text{ if } a = \mathsf{yes} \\ \neg F \text{ if } a = \mathsf{no} \\ F' \text{ such that } X \vdash F' \text{ iff } \mathsf{Ans}(X, F) = a, \text{ otherwise} \end{cases}$$

Similarly, an answer set $A = \mathsf{Ans}(X, F)$ to an open query formula F can be used to transform F into a set of sentences:

$$\text{val}(F, A) = \{F\sigma \mid \sigma \in A\}$$

We can then define that an answer (set) A_1 to a query F is at least as informative as A_2, symbolically $A_1 \leq A_2$, by requiring that

$$\mathsf{Upd}(0, \mathrm{val}(F, A_1)) \leq \mathsf{Upd}(0, \mathrm{val}(F, A_2))$$

In general, more information does not mean more consequences. In other words: answers are not necessarily preserved under growth of information. Queries, for which this is the case, are called *persistent*.

Definition 6.2 (Persistent Queries) *A closed, resp. open, query formula F is called* persistent *if for all $X, Y \in L_{KB}$, $X \vdash F$ implies $Y \vdash F$, resp. $\mathsf{Ans}(X, F) \leq \mathsf{Ans}(Y, F)$, whenever $X \leq Y$. If all $F \in L_{\mathrm{Query}}$ are persistent, the KS and its inference relation $\vdash$ are called persistent. The set of all persistent query formulas is denoted by L_{PersQ}. A connective of the query language is called persistent, if every query formed with it and with persistent subformulas is again persistent.*

Definition 6.3 (Ampliative Inputs) *An input sentence $F \in L^0_{\mathrm{Input}}$ is called (i)* ampliative[3] *if $X \leq \mathsf{Upd}(X, F)$, or (ii)* reductive *if $X \geq \mathsf{Upd}(X, F)$. A KS and its update operation* Upd *are called* ampliative*, if all inputs $F \in L_{\mathrm{Input}}$ are ampliative. The set of all ampliative input formulas is denoted by L_{AmpI}.*

In addition to the information order $\leq$, another preorder $\leq_X$, called **relative information distance** with respect to some given knowledge base X compares the 'information distance' to X. Intuitively, $Y \leq_X Z$ expresses the fact that Y is informationally closer to X than (or as close to X as) Z. The notion of relative information distance is important for determining the *minimal mutilation* of a knowledge base when it is updated.

A certain subset $L_{\mathrm{Unit}} \subseteq L_{\mathrm{Input}}$ designates those elementary expressions which are called **information units**, e.g. atoms, literals, or somehow qualified (labelled, or annotated) atoms, and the like. An information unit represents an elementary piece of information with a positive information content.

Definition 6.4 (Regular Positive Knowledge Systems) *A positive knowledge system $\boldsymbol{K}$ over a schema Σ is called* regular*, if there is a preorder $\langle L_{KB}, \leq, 0\rangle$ with least element 0, and for any $X \in L_{KB}$, there is a preorder $\langle L_{KB}, \leq_X\rangle$ with least element X, and there is a designated set $L_{\mathrm{Unit}} \subseteq L_{\mathrm{Input}}$, such that*

(KS4) Unit inputs are ampliative: $X \leq \mathsf{Upd}(X, u)$, for any $X \in L_{KB}$, and for any $u \in L^0_{\mathrm{Unit}}$.

(KS5) The information ordering is compatible with ampliative update and persistent inference: for all $X, Y \in L_{KB}$,

$$X \leq Y \quad \textit{iff} \quad \forall F \in L^0_{\mathrm{AmpI}} \forall G \in L^0_{\mathrm{PersQ}} : \mathsf{Upd}(X, F) \vdash G \;\Rightarrow\; \mathsf{Upd}(Y, F) \vdash G$$

(KS6) Ampliative inputs are persistent queries: $L_{\mathrm{AmpI}} = L_{\mathrm{PersQ}} \cap L_{\mathrm{Input}}$.

(KS7) Updates are minimal mutilations:

$$\mathsf{Upd}(X : \Sigma, F) \;\in\; \mathrm{Min}_{\leq_X}\{Y \in L_{KB} \mid Y \vdash F \;\;\&\;\; Y \vdash IC_\Sigma\}$$

A regular positive KS can be represented as a 9-tuple

$$\langle 0, \leq, L_{KB}, \leq_X, \vdash, L_{\mathrm{Query}}, \mathsf{Upd}, L_{\mathrm{Input}}, L_{\mathrm{Unit}}\rangle$$

Obviously, $\boldsymbol{A}$ is regular: the information order is given by set inclusion, the relative information distance is given by symmetric difference, the informationally empty database is the empty set, and the information units are atoms. Its positive fragment $\boldsymbol{A}^+$ is persistent and ampliative.

6.3 FORMAL PROPERTIES OF KNOWLEDGE SYSTEMS

The following is a list of some fundamental properties a knowledge system may have. The first two conditions of Contraction and Permutation are well-known (as so-called *structural rules*) from Gentzen-style sequent systems. In a knowledge system, they describe the behavior of the update operation. Let $A, B \subseteq L^0_{\text{Input}}$.

(Contraction)

$$\mathsf{Upd}(\mathsf{Upd}(X, A), A) = \mathsf{Upd}(X, A)$$

(Permutation)

$$\mathsf{Upd}(\mathsf{Upd}(X, A), B) = \mathsf{Upd}(\mathsf{Upd}(X, B), A)$$

Both Contraction and Permutation follow from the property of

(Update Synchronicity) $\quad \mathsf{Upd}(\mathsf{Upd}(X, A), B) = \mathsf{Upd}(X, A \cup B)$

which expresses the fact that two inputs in succession (i.e. at different time points) can be handled as one aggregated input implying that the order of inputs does not matter.

(Update Monotonicity)

$$X \leq Y \;\Rightarrow\; \mathsf{Upd}(X, A) \leq \mathsf{Upd}(Y, A)$$

(Lemma Redundancy) alias: Cut, Transitivity

$$X \vdash F \;\;\&\;\; \mathsf{Upd}(X, F) \vdash G \;\Rightarrow\; X \vdash G$$

(Lemma Compatibility) alias Cautious Monotonicity (due to Gabbay, 1985),

$$X \vdash F \;\;\&\;\; X \vdash G \;\Rightarrow\; \mathsf{Upd}(X, F) \vdash G$$

Lemma Redundancy and Compatibility can be combined in the following condition of

(Cumulativity) $\quad X \vdash F \;\Rightarrow\; C(\mathsf{Upd}(X, F)) = C(X)$

Even stronger than Cumulativity is the following property (proposed in Gärdenfors, 1988),

(Vacuity) $\quad X \vdash F \;\Rightarrow\; \mathsf{Upd}(X, F) = X$

Definition 6.5 (Query Completeness) *A knowledge system is called* query complete *if for all $X \in L_{KB}$, for all $F \in L^0_{\text{Query}}$, and for all if-answers $a \in L^0_{\text{Ans}}$, there is an if-query $F' \in L^0_{\text{Query}}$, such that*

$$X \vdash F' \quad \textit{iff} \quad \mathsf{Ans}(X, F) = a$$

The following important property guarantees the freedom of knowledge base evolution.

Definition 6.6 (Input Completeness) *A knowledge system is called* input complete*, if for all $X, Y \in L_{KB}$, there is some $A \subseteq L_{\text{Input}}$, such that $Y = \mathsf{Upd}(X, A)$.*

Observation 6.1 *A KS is input complete iff KBs are both input constructible and input destructible, that is, if both of the following conditions hold:*

$$\begin{array}{ll} (i) & \forall X \in L_{KB} \exists A \subseteq L_{\text{Input}} : X = \mathsf{Upd}(0, A) \\ (ii) & \forall X \in L_{KB} \exists A \subseteq L_{\text{Input}} : \mathsf{Upd}(X, A) = 0 \end{array}$$

Minimal Change for Unit Expansion. Let $X, Y \in L_{KB}$, and $u \in L_{\text{Unit}}$. If $X \nvdash u$, then $X + u = \mathsf{Upd}(X, u)$ is the least extension of X such that u can be inferred, expressed by the conditions (+1) and (+2):

$$\begin{array}{ll} (+1) & X + u \geq X \\ (+2) & Y \geq X \;\;\&\;\; Y \vdash u \;\Rightarrow\; X + u \leq Y \end{array}$$

Minimal Change for Unit Retraction. If $X \vdash u$, then $X - u = \mathsf{Upd}(X, \neg u)$ is the greatest KB smaller than X such that $\neg u$ can be inferred, expressed by (-1) and (-2):

$$\begin{array}{ll} (-1) & X - u \leq X \\ (-2) & Y \leq X \ \ \& \ \ Y \vdash \neg u \ \Rightarrow \ X - u \geq Y \end{array}$$

Notice that $(+1)$ corresponds to (KS4).

Monotonicity. The following definition captures the idea that a KS is considered *monotonic* if all consequences of a KB are preserved after any update by some new piece of information.

Definition 6.7 *A KS is called* ***monotonic*** *if for all $X \in L_{KB}$, and all $F \in L^0_{\mathrm{Input}}$,*

$$C(X) \subseteq C(\mathsf{Upd}(X, F))$$

Though fundamental in the theory of consequence operations due to Tarski, this is too strong a requirement for knowledge systems, in general.

There are two 'parameters' on which Monotonicity depends: the update operation may be ampliative or not, and the inference relation may be persistent or not.

Observation 6.2 *A KS is monotonic if it is ampliative and persistent.*[4]

Proof: For any $X \in L_{KB}$, and any $F \in L_{\mathrm{Input}}$, we get $X \leq \mathsf{Upd}(X, F)$ by Ampliative Update, and consequently $C(X) \subseteq C(\mathsf{Upd}(X, F))$, by Persistent Inference. □

Clearly, the knowledge system $\boldsymbol{A}$ of relational databases is not monotonic. But $\boldsymbol{A}^+$ is a monotonic KS, since it is persistent and ampliative.

In general, practical systems are nonmonotonic since they allow for non-persistent queries (e.g. by means of negation based on the completeness assumption) and for non-ampliative updates (by means of deletion, resp. contraction). In the AI literature, several systems of nonmonotonic reasoning with different motivations have been proposed. See Brewka, 1991, for an overview.

6.4 VIVID KNOWLEDGE SYSTEMS

Positive knowledge systems extending $\boldsymbol{A}$ conservatively are called *vivid*.

Definition 6.8 *A positive knowledge system*

$$\boldsymbol{K} \ = \ \langle L_{KB}, \vdash, L_{\mathrm{Query}}, \mathsf{Upd}, L_{\mathrm{Input}} \rangle$$

is called ***vivid*** *if there is a mapping*

$$h : L^0(\neg, \wedge, \vee, \exists, \forall) \to L^0_{\mathrm{Query}},$$

such that for all $X \subseteq \mathrm{At}^0$, $l \in \mathrm{Lit}^0$, and $F \in L^0(\neg, \wedge, \vee, \exists, \forall)$,

$$\begin{array}{ll} (1) & h \text{ maps } \mathrm{At}^0 \text{ into } L_{\mathrm{Unit}} \\ (1) & h \text{ maps } \mathrm{Lit}^0 \text{ into } L_{\mathrm{Input}} \\ (2) & \mathsf{Upd}_A(X, l) \vdash_A F \quad \textit{iff} \quad \mathsf{Upd}(\mathsf{Upd}(0, h(X), h(l)) \vdash h(F) \end{array}$$

where Upd_A and $\vdash_A$ are the update and inference operations of $\boldsymbol{A}$, and $h(X) = \{h(a) \mid a \in X\}$.

Positive vivid knowledge systems are based on a general completeness assumption, whereas non-positive systems drop this assumption for specific predicates. For instance, $\boldsymbol{A}$ can be extended to a non-positive vivid KS, called *relational factbases*, by allowing for literals instead of atoms as information units (see 11.2 and 11.3). Further important examples of positive vivid knowledge systems are temporal, fuzzy and disjunctive databases. All these types of knowledge bases can be extended to *deductive knowledge bases* by adding deduction rules of the form $F \leftarrow G$ (see Chapter 15).

6.5 KNOWLEDGE UPDATE AND KNOWLEDGE INTEGRATION

Whenever knowledge bases are assumed to be complete, this requires that all information suppliers are *information competent* with respect to all predicates of the schema, that is, they are all assigned to the knowledge base as *owners*. Under this assumption, all input sentences can be treated as if they were delivered by a single *logical 'user'*, or agent, and there is no consideration of the supplier or source of an input. According to the success postulate, an update prefers the most recent input over any already-stored piece of information. This preference is justified by the assumption that, logically, there is only one 'user' who is the owner, and when he enters a new input sentence, he is aware of the already-stored information representing his own out-dated information state.

Whenever a knowledge base may be incomplete, and there are more than one agent associated with it as information suppliers, one has to distinguish the change brought about through **knowledge integration** from that of update. Knowledge integration concerns the combination of *epistemically independent* pieces of information provided, e.g., by different agents. A knowledge base may represent the *joint information state* of all information supplier agents associated with it, and a knowledge integration operation is used instead of the update operation for assimilating new inputs that come from an epistemically independent source. The result of knowledge integration may differ from that of update when the new input affects some already-stored piece of information. An update always *overrides* the affected old information (as if it had not been there). An integration *combines* the affected old item with the new input leading to a relative *neutralization* (or consolidation) in the case of a conflicting input, and to a mutual *amplification* in the case of an affirmative input.

So, one has to distinguish between inputs delivered by an agent that is associated with a knowledge base as an owner from inputs by an epistemically independent agent. Formally, a knowledge integration operation

$$\mathsf{Int} : L_{KB} \times L^0_{\mathrm{Input}} \to L_{KB}$$

is, unlike an update operation, not idempotent. That is, it violates the Contraction principle:

$$\mathsf{Int}(\mathsf{Int}(X, F), F) \neq \mathsf{Int}(X, F)$$

And it also violates the Success postulate: the integration of a conflicting input sentence F into a knowledge base X results in a knowledge base where F does not hold, that is,

$$\mathsf{Int}(X, F) \nvdash F$$

6.6 SUMMARY

Knowledge systems are based on two operations defining their logical user interface: an inference-based query answering and an update operation. Different knowledge systems allow to process different types of information that is represented in a particular form of knowledge base and queried by means of a particular query language. A number of formal properties and postulates are defined for characterizing knowledge systems. The violation of monotonicity through non-ampliative updates and non-persistent queries in all practical (or vivid) knowledge systems constitutes a major departure from standard logics.

Although this is usually not made explicit, databases and knowledge bases may be viewed as *knowledge- and perception-based agents* that communicate with their human owners by accepting inputs corresponding to TELL communication events, and by accepting queries corresponding to ASK events. We have argued in Chapter 4, and come back to this point in Chapter 14, that it is desirable to make this communication interface more explicit, e.g. by including in SQL the possibility to specify communicative behavior in the form of reaction rules.

Table 6.1. Formal properties of some basic positive knowledge systems. TA = temporal databases, FA = fuzzy databases, SA = secure databases, VA = disjunctive databases, $5A$ = S5-epistemic databases.

	A^+	A	TA	FA	SA	VA	$5A$
Contraction	√	√	√	√	√	√	√
Permutation	√						
Update Synchronicity	√						
Update Monotonicity	√	√	√	√	√	?	?
Cumulativity	√	√	√	√	√	?	?
Monotonicity	√						
Query Completeness		√	√	√	√		√
Input Completeness		√	√	√	?	?	?

? denotes *open problem*

Notes

1. The formulation of a knowledge system in terms of query and input processing was already implicit in Belnap, 1977. In Levesque, 1984 it was proposed as a 'functional approach to knowledge representation'. In Wagner, 1994b; Wagner, 1995 the concept of knowledge systems was further extended and used as an integrating framework for knowledge representation and logic programming.

2. The usual way to compare the information content of two knowledge bases in standard logic by checking the inclusion of consequences: $X \leq Y$ if $C(X) \subseteq C(Y)$, does not work in a general (possibly nonmonotonic) setting.

3. The name is adopted from Belnap, 1977.

4. Or, rather exotically, if all inputs are reductive and all queries are 'antipersistent', i.e. preserved under information decrease.

7 TEMPORAL DATABASES

Temporal databases represent the practically most important logical extension of relational databases since temporal information plays an essential role in many application domains. It is expected that the temporal database language standard TSQL2 proposed in Snodgrass, 1995 will be included in future versions of SQL. We show how the notions of a database table, of natural inference and of table operations have to be generalized in order to accommodate temporally qualified information.

7.1 INTRODUCTION

We consider two types of temporal formulas corresponding to two types of temporal qualification. In order to say that Charles and Diana have been married from 1981 to 1996 we can use a collection of **timepoint** sentences of the form

$$m(C,D)@1981,\ m(C,D)@1982,\ldots,m(C,D)@1996$$

Alternatively, we can state this fact more compactly using the **timestamp** sentence

$$m(C,D)@[1981..1996]$$

Notice that the distinction between timepoints (or instants) and timestamps depends on the chosen *granularity* of time. With respect to the granularity of days, the formula $m(C,D)@1981$ is not a timepoint but a timestamp formula: it corresponds to $m(C,D)@[01/01/81..31/12/81]$.

For computational purposes, the most natural model of time is **discrete linear time**. Consequently, our temporal database model is based on a discrete linearly ordered set $\boldsymbol{T}$ of timepoints of atomic granularity which cannot be further refined.

Definition 7.1 *A **time domain** is a discrete linear ordering*

$$\langle \boldsymbol{T}, <, 0, s\rangle$$

with least element 0 *and successor function* s*.*

In the continuous model of time, time points correspond to points on the real line, that is, to real numbers. With respect to the continuous model, the 'timpoints' of $\boldsymbol{T}$ are, in fact, no points but line segments.

7.2 THE TIMESTAMP ALGEBRA

We define a timestamp as a sequence of disjoint timepoint intervals. This form of timestamps was proposed in Ben-Zvi, 1982; Gadia, 1988. A more general type of timestamps in the form of lambda expressions is defined in the Appendix C.

A **timestamp** is an expression

$$[b_1..e_1, \ldots, b_n..e_n]$$

where $b_i, e_i \in \boldsymbol{T}$, $b_i \leq e_i$, and $e_i < b_{i+1}$. The last interval in a timestamp may also end with the special value ∞ (standing for *ad eternum*). The set of all timestamps over a time domain $\boldsymbol{T}$ is denoted by $\mathcal{T}$.

For simplicity, we identify the singleton timepoint interval $i..i$ with i, and write $p(c)@i$ instead of $p(c)@[i]$. A timestamp $T = [b_1..e_1, \ldots, b_n..e_n]$ corresponds to the timepoint set

$$\dot{T} = \{t \in \boldsymbol{T} \mid b_i \leq t \leq e_i \text{ for some } i \leq n\}$$

Accordingly, we can define an inclusion ordering of timestamps by

$$T_1 \subseteq T_2 :\Longleftrightarrow \dot{T}_1 \subseteq \dot{T}_2$$

and union, intersection and difference of timestamps, yielding the Boolean algebra

$$\langle \mathcal{T}, \cup, \cap, - \rangle$$

with least element $[]$, greatest element $[0..\infty]$, and Boolean complement $-T = [0..\infty] - T$.

7.3 TEMPORAL TABLES

A **temporal table** P over a table schema $p(A_1, \ldots, A_n)$ and a time domain $\boldsymbol{T}$ is a function

$$P : \mathrm{dom}(A_1) \times \ldots \times \mathrm{dom}(A_n) \to \mathcal{T},$$

assigning to a finite number of tuples $\langle c_1, \ldots, c_n \rangle$ a nonempty *valid time* $T \in \mathcal{T}$, i.e. the set of timepoints at which the predicate p applies to $\langle c_1, \ldots, c_n \rangle$. The *snapshot* of P at timepoint t is the ordinary table

$$P^t = \{\langle c_1, \ldots, c_n \rangle \mid t \in P(c_1, \ldots, c_n)\}$$

A temporal table has a designated functional column containing the assigned timestamps.

A temporal database schema consists of a time domain, a sequence of table schemas and a set of integrity constraints. A temporal database Δ over a schema

$$\Sigma = \langle \boldsymbol{T}, \langle p_1, \ldots, p_m \rangle, IC \rangle$$

is a sequence of temporal tables $\langle P_1, \ldots, P_m \rangle$ over $\boldsymbol{T}$, such that P_i is the temporal extension of the predicate p_i in Δ. A temporal database Δ can be propositionally represented as a set X_Δ of timestamped atoms of the form $a@T$ such that $T \in \mathcal{T}$ is the timestamp assigned to the atomic sentence a. Its snapshot at timepoint t is the collection of the corresponding snapshots of its tables:

$$\Delta(t) = \langle P_1^t, \ldots, P_m^t \rangle$$

Example 2 *The following temporal table represents the salary history of certain employees:*

$Salary =$

Tom	3600	1994..1995
Tom	4100	1996
Tom	3900	1997
Bob	4400	1993..1994, 1997
Bob	4600	1995..1996

Notice that, in practice, the end timepoint of a currently valid sentence is often not known. In order to represent this adequately, one needs the indexical null values *now* and *until changed* which are discussed below.

A temporal database Δ over the schema $\Sigma = \langle \boldsymbol{T}, \langle p_1, \ldots, p_m \rangle, IC \rangle$ can be propositionally represented as

$$X_\Delta = \bigcup_{k=1}^{m} \{p_k(\boldsymbol{c})@T : \langle \boldsymbol{c}, T \rangle \in P_k\}$$

For instance, let $\Delta_2 = \langle \mathit{Salary} : \mathrm{sal} \rangle$. Then, writing X_2 instead of X_{Δ_2},

$$X_2 = \{\mathrm{sal}(\mathrm{Tom}, 3600)@[1994..1995],\ \mathrm{sal}(\mathrm{Tom}, 4100)@1996, \ldots\}$$

7.4 NATURAL INFERENCE IN TEMPORAL DATABASES

A temporal database X over Σ corresponds to a **valid-time valuation**

$$T_X : L^0_\Sigma \to \mathcal{T}$$

which is induced by X in a natural way:

$$\begin{array}{lrcl}
(a) & T_X(a) & = & \begin{cases} T \text{ if } a@T \in X \text{ for some } T \in \mathcal{T} \\ \emptyset \text{ otherwise} \end{cases} \\
(\neg) & T_X(\neg F) & = & [0..\infty] - T_X(F) \\
(\wedge) & T_X(F \wedge G) & = & T_X(F) \cap T_X(G) \\
(\vee) & T_X(F \vee G) & = & T_X(F) \cup T_X(G) \\
(\exists) & T_X(\exists x H[x]) & = & \bigcup\{T \mid c@T \in \mathrm{CAns}(X, H[x])\} \\
(\forall) & T_X(\forall x H[x]) & = & \bigcap\{T \mid c@T \in \mathrm{CAns}(X, H[x])\} \\
(\mathrm{CAns}) & \mathrm{CAns}(X, H[x]) & = & \{c@T \mid T = T_X(H[c]) \neq \emptyset\}
\end{array}$$

Notice that we write $c@T$ standing for a timestamped tuple $\langle c, T \rangle$.

Then, the **natural inference** relation between a temporal database X and a timestamped if-query $F@T$ is defined as

$$X \vdash F@T \quad :\Longleftrightarrow \quad T \subseteq T_X(F)$$

or, equivalently, if F holds in all corresponding snapshots of X,

$$X \vdash F@T \quad \Longleftrightarrow \quad X(t) \vdash F \text{ for all } t \in \dot{T}$$

For instance, according to the salary database there are employees who got less than 4000 in 1997:

$$X_2 \ \vdash \ \exists x \exists y(\mathrm{sal}(x, y) \wedge y < 4000)@1997$$

since $1997 \in T_{X_2}(\exists x \exists y(\mathrm{sal}(x, y) \wedge y < 4000)) = [1994..1995, 1997]$. In addition to if-queries with constant timestamps, we may allow for query formulas where the timestamp is a quantified timepoint variable such as in

$$\begin{array}{lcl}
X \vdash \exists t F@t & :\Longleftrightarrow & T_X(F) \neq \emptyset \\
X \vdash \forall t F@t & :\Longleftrightarrow & T_X(\neg F) = \emptyset
\end{array}$$

A complex temporal query is, e.g., *'Are there persons who married the same person again ?'*, formally expressed as

$$\exists t_1, t_2, t_3 \exists x, y : t_1 < t_2 < t_3 \wedge m(x, y)@t_1 \wedge \neg m(x, y)@t_2 \wedge m(x, y)@t_3$$

Natural inference in temporal databases captures minimal entailment $\models_m$ in temporal first order logic (see Appendix C.4).

Claim 1 $X \vdash F@T$ *iff* $X \models_m F@T$.

7.5 UPDATES AND INFORMATION ORDER

The basic inputs to a temporal database are timestamped ground literals:

$$\mathsf{Upd}(X, a@T') = \begin{cases} X \cup a@T' & \text{if } T_X(a) = \emptyset \\ X - \{a@T\} \cup \{a@[T \cup T']\} & \text{if } T = T_X(a) \neq \emptyset \end{cases}$$

$$\mathsf{Upd}(X, \neg a@T') = \begin{cases} X & \text{if } T_X(a) \cap T = \emptyset \\ X - \{a@T\} \cup \{a@[T - T']\} & \\ \qquad \text{if } T = T_X(a) \ \ \& \ \ T_X(a) \cap T' \neq \emptyset & \end{cases}$$

The knowledge system of temporal databases, denoted by $T\boldsymbol{A}$, is then defined as

$T\boldsymbol{A} =$

L_{KB}	$2^{\mathrm{At}^0 \times \mathcal{T}}$
L_{Query}	$L(\neg, \wedge, \vee, \exists, \forall) \cup L(\neg, \wedge, \vee, \exists, \forall) \times \mathcal{T}$
L^0_{Ans}	$\{\mathsf{yes}, \mathsf{no}\} \cup \{\mathsf{yes}\} \times \mathcal{T}$
L_{Input}	$\mathrm{Lit} \times \mathcal{T}$
L_{Unit}	$\mathrm{At} \times \mathcal{T}$

Claim 2 *$T\boldsymbol{A}$ is vivid.*

Proof: Let $X \subseteq \mathrm{At}$, $F \in L^0(\neg, \wedge, \vee, \exists, \forall)$, and let $t \in \boldsymbol{T}$ be an arbitrary fixed timepoint. We define $h(F) = F@t$. The function h provides an embedding of a relational database X in a temporal database Y such that X is the snapshot of Y at instant t. It is easy to see that

$$\mathsf{Upd}_A(X, l) \vdash_A F \iff \mathsf{Upd}(\mathsf{Upd}(0, h(X)), h(l)) \vdash h(F) \quad \square$$

The information order between two temporal databases X and Y is defined as follows.

$$X \leq Y \text{ if for all facts } a@T \in X \text{ it holds that } T \subseteq T_Y(a)$$

For instance,

Tom	3600	1994
Tom	4100	1995

$<$

Tom	3600	1994
Tom	4100	1995
Bob	4400	1994

$<$

Tom	3600	1994
Tom	4100	1995
Bob	4400	1994..1995

7.6 INDEXICAL TEMPORAL NULL VALUES

In this section, we briefly discuss the meaning of two important temporal null values: *now* and *until changed*.

It is natural to use the expression *now* in queries like, for instance, *Is Tom's salary now higher than 4000 ?* Such indexical queries, however, cannot be evaluated on the basis of a temporal database X alone. Their evaluation requires, in addition to X, a reference instant which determines the interpretation of *now*. For instance,

$$\begin{array}{lll} X_2, 1996 & \vdash & \mathrm{sal}(\mathrm{Tom}, x)@now \wedge x > 4000 \\ X_2, 1997 & \nvdash & \mathrm{sal}(\mathrm{Tom}, x)@now \wedge x > 4000 \end{array}$$

since in the database X_2, Tom's salary is decreased in 1997 from 4100 to 3900 dollars.

The inference relation for the query language with *now* has to be extended accordingly. Let X be a temporal database, and let $i \in \boldsymbol{T}$ be an evaluation instant. Then,

$$X, i \vdash F@now \quad :\Longleftrightarrow \quad X \vdash F@i$$

Another important indexical null value is *until changed (uc)* which is used when storing new information with an open-ended valid time. Notice that it seems to be the rule, and not an

exception, that time varying information (such as, e.g., the new salary of an employee) does not come with a definite end time. In that case, the null value uc must be used to store the new input in the database. The input sentence

$$\mathrm{sal}(\mathrm{Tom}, 3900)@[1997, uc]$$

expresses the fact that from 1997 until changed Tom's salary is 3900. Formally, this means that the unknown end timepoint is greater than or equal to *now*, i.e. using the notation from Appendix C,

$$[1997, uc] = \lambda t(1997 \leq t \wedge \exists s(now \leq s \wedge t \leq s))$$

The difference between *now* and uc shows up only when it comes to statements about the future. This is the case if the database is used for planning or for recording commitments. For instance, if a promotion commitment for the future (1999) is made now (1998), it should be already inferable now which is only achieved if it is expressed with the help of uc:

$$\begin{array}{rcl} \{\mathrm{sal}(\mathrm{Tom}, 4200)@[1999, uc]\}, 1998 & \vdash & \mathrm{sal}(\mathrm{Tom}, 4200)@1999 \\ \{\mathrm{sal}(\mathrm{Tom}, 4200)@[1999, now]\}, 1998 & \nvdash & \mathrm{sal}(\mathrm{Tom}, 4200)@1999 \end{array}$$

If open-ended facts are expressed by means of *now*, they are false as sentences about the future (1999) evaluated now (1998), while they are neither false nor true in the future if they are expressed with the help of uc:

$$\begin{array}{rcl} \{\mathrm{sal}(\mathrm{Tom}, 4100)@[1998, now]\}, 1998 & \vdash & \neg\mathrm{sal}(\mathrm{Tom}, 4100)@1999 \\ \{\mathrm{sal}(\mathrm{Tom}, 4100)@[1998, uc]\}, 1998 & \nvdash & \neg\mathrm{sal}(\mathrm{Tom}, 4100)@1999 \end{array}$$

Many authors in the database literature seem to overlook the fact that *now* and uc are indexical expressions requiring a special semantical treatment, and should not be confused with ∞.

7.7 TEMPORAL TABLE OPERATIONS

As in relational databases, we 'implement' the theoretical answer set operation *CAns*, defined on the basis of natural inference, through the algebraic operation Ans defined inductively over the composition of query formulas. Plain open queries, asking for instantiations and associated valid times, are evaluated compositionally by means of the table operations *selection*, *projection*, *join*, *union* and *difference*. If X is a temporal database over Σ, and $P : p$ represents an n-place predicate in X, then

$$\mathsf{Ans}(X, p(x_1, \ldots, x_n)) = P$$

that is, the answer to an atomic query $p(x_1, \ldots, x_n)$ is the temporal extension P of predicate p in the database state X including the designated column with the assigned valid times.

The selection operation $\mathrm{Sel}(C, Q)$, where C is a plain row constraint and Q is a temporal table, is defined like in plain relational algebra: $\mathrm{Sel}_t(C, Q) = \mathrm{Sel}(C, Q)$.

In the case of the projection operation, it may be necessary to *normalize* the result: if the projection result contains tuples that agree on all attributes but have different timestamps, then only one of these tuples is retained with a timestamp that is the union of the timestamps of all these tuples. We denote this **normalization operation** by Norm_t. Thus,

$$\mathrm{Proj}_t(\langle j_1, \ldots, j_k\rangle, P) = \mathrm{Norm}_t(\mathrm{Proj}(\langle j_1, \ldots, j_k\rangle, P))$$

Existentially quantified query formulas in temporal databases are evaluated by means of Proj_t. Let $F = F[x_1, \ldots, x_k]$, and $1 \leq i \leq k$. Then,

$$\mathsf{Ans}(X, \exists x_i F) = \mathrm{Proj}_t(\langle 1,\ldots,i-1,i+1,\ldots,k\rangle, \mathsf{Ans}(X, F))$$

Conjunction and Join. The algebraic basis of evaluating conjunction is the cross product of two temporal tables defined as follows:

$$P \times_t Q = \{\langle c_1, \ldots, c_m, d_1, \ldots, d_n, T \cap T' \rangle \mid \langle \boldsymbol{c}, T \rangle \in P \ \ \& \ \ \langle \boldsymbol{d}, T' \rangle \in Q\}$$

Let $i = 1, \ldots, m$ and $j = 1, \ldots, n$. Then,

$$P \overset{i=j}{\bowtie}_t Q = \mathrm{Proj}_t(\langle 1,\ldots,m+j-1,m+j+1,\ldots,m+n\rangle, \mathrm{Sel}_t(\$i = \$j, P \times_t Q))$$

The definition of $\overset{i=j}{\bowtie}$ can be easily generalized to the case of a conjunction with more than one shared variable. If all shared variables are treated in this way, the resulting operation $\bowtie$ is called *temporal join*. As in relational algebras, it is used to evaluate conjunctions:

$$\mathsf{Ans}(X, F \wedge G) = \mathsf{Ans}(X, F) \bowtie_t \mathsf{Ans}(X, G)$$

Disjunction and Union. The union of two temporal tables (of the same type) is obtained by first forming their plain set-theoretic union, and then normalizing it:

$$P \cup_t Q = \mathrm{Norm}_t(P \cup Q)$$

As in relational databases, union is the algebraic counterpart of disjunction:

$$\mathsf{Ans}(X, F \vee G) = \mathsf{Ans}(X, F) \cup_t \mathsf{Ans}(X, G)$$

Negation and Difference. Let P and Q be tables with n and m columns. The difference of P and Q is defined only if $m \leq n$ and both tables agree on the types of their columns $1, \ldots, m$. Then,

$$\begin{aligned} P -_t Q \quad = \quad & \{\langle c_1, \ldots, c_n, T\rangle \mid \langle c_1, ..., c_n, T\rangle \in P \ \ \& \ \ \langle c_1, ..., c_m, T'\rangle \notin Q \\ & \quad \text{or} \ \ \langle c_1, ..., c_m, T'\rangle \in Q \ \ \& \ \ \langle c_1, ..., c_n, T''\rangle \in P \ \ \& \ \ T = T'' - T'\} \end{aligned}$$

As in relational databases, negation in query formulas is only evaluable in temporal databases in conjunction with a positive choice to which the negative condition refers. Such a conjunction is evaluated by the temporal difference operation:

$$\mathsf{Ans}(X, F \wedge \neg G) = \mathsf{Ans}(X, F) -_t \mathsf{Ans}(X, G)$$

7.7.1 Correctness and Completeness

The above table operations collectively define the answer operation Ans inductively by providing an algebraic evaluation clause for each alternative of forming a query. The algebraic answer operation Ans adequately captures the inference-based evaluation of queries expressed by *CAns*, since it holds that

1. every answer provided by Ans corresponds to an inferable instance of the query formula, or, in other words,

$$\begin{aligned} \text{(correctness)} \quad & \text{if} \quad \langle c_1, \ldots, c_m, T\rangle \in \mathsf{Ans}(X, F[x_1, \ldots, x_m]), \\ & \text{then} \quad X \vdash F[c_1, \ldots, c_m]@T \end{aligned}$$

2. every instantiation of an evaluable query formula with a nonempty valid time is contained in the answer set provided by Ans:

$$\begin{aligned} \text{(completeness)} \quad & \text{if} \quad X \vdash F[c_1, \ldots, c_m]@T, \\ & \text{then} \quad \langle c_1, \ldots, c_m, T\rangle \in \mathsf{Ans}(X, F[x_1, \ldots, x_m]) \end{aligned}$$

7.7.2 *Four Kinds of Queries*

In addition to the case of plain open queries defined above, the answer operation is also defined for timestamped and plain if-queries and for timestamped open queries. Let $F \in L^0_\Sigma$ be an if-query, $G \in L_\Sigma$ an open query, and $T \in \mathcal{T}$ a timestamp. Then, we define

$$\mathsf{Ans}(X, F) = \begin{cases} \mathsf{yes}@T \text{ if } T = T_X(F) \neq \emptyset \\ \mathsf{no} \text{ otherwise} \end{cases}$$

$$\mathsf{Ans}(X, F@T) = \begin{cases} \mathsf{yes} \text{ if } T_X(F) \supseteq T \\ \mathsf{no} \text{ otherwise} \end{cases}$$

$$\mathsf{Ans}(X, G@T) = \{\sigma \mid T_X(G\sigma) \supseteq T\}$$

7.8 TEMPORAL DEDUCTION RULES

Assume that a car insurance company refunds money to its customers if they did not have an accident in the previous year, expressed by the rule

$$r_1 : \text{refund}(x)@(t+1) \longleftarrow \text{customer}(x)@t \wedge \neg\text{accident}(x)@t$$

where $t+1$ stands for the successor $s(t)$ of timepoint t, that is, it denotes the following year. Let the insurance database $\langle X_{\text{Ins}}, R_{\text{Ins}}\rangle$ consist of this rule together with the fact that Wagner (W) was a customer in 1997:

$$\begin{aligned} X_{\text{Ins}} &= \{\text{customer}(W)@1997\} \\ R_{\text{Ins}} &= \{r_1\} \end{aligned}$$

This temporal deductive database has two minimal closures:

$$\begin{aligned} X_1 &= \{\text{customer}(W)@1997, \text{refund}(W)@1998\} \\ X_2 &= \{\text{customer}(W)@1997, \text{accident}(W)@1997\} \end{aligned}$$

Intuitively, Wagner should get the refund, i.e. only X_1 is an intended closure. Note that in the case of X_2, the atom $accident(W)@1997$ cannot be generated by applying rules from the instantiation of R_{Ins} because it does not appear in the head of any of them. The only stable closure is X_1, and thus,

$$R_{\text{Ins}}(X_{\text{Ins}}) = \{X_1\} \vdash \text{refund}(W)@1998$$

7.9 BITEMPORAL DATABASES

In a bitemporal database, the **belief time** of a piece of information is stored along with its valid time. This allows to record belief changes and to trace errors. For instance, in a hospital database, there may be a first diagnosis of hepatitis X (during November and December 1995) for patient 013 on 13/11/95 which is corrected to hepatitis B on the next day. This can be represented in a bitemporal table with two temporal columns where the belief time column is functional:

$$Diag = \begin{array}{|llll|}\hline 013 & hepX & \text{Nov95..Dec95} & 13/11/95 \\ 013 & hepB & \text{Nov95..Dec95} & 14/11/95..uc \\ \hline \end{array}$$

It can also be expressed propositionally using a temporal belief operator B^T where T is a timestamp:

$$\begin{aligned} X_{Diag} = \{&\mathsf{B}^{[13/11/95]}\text{diag}(013, hepX)@[\text{Nov95..Dec95}], \\ &\mathsf{B}^{[14/11/95..uc]}\text{diag}(013, hepB)@[\text{Nov95..Dec95}]\} \end{aligned}$$

A *bitemporal table* P over a time domain $\boldsymbol{T}$ and a table schema $p(A_1, \ldots, A_n)$ is a partial function

$$P : \text{dom}(A_1) \times \ldots \times \text{dom}(A_n) \times \mathcal{T} \to \mathcal{T}$$

assigning to a finite number of valid-time-stamped tuples $\langle c_1, \ldots, c_n, T_v \rangle$ a nonempty belief-time-stamp T_b.

7.10 SUMMARY

Supporting valid time (and belief time) is an important extension of (object-) relational databases since time-varying information occurs in many domains. The addition of temporal qualifications makes the query language and the answer operation considerably more complex. Consequently, it is essential to have a rigouros definition of correct answers, derived from temporal logic, for validating practical systems. We have shown that the semantics of temporal databases is complicated by the necessity to allow information about the future and the indexical null value *until changed*.

7.11 FURTHER READING

The broad area of temporal database research is presented in Tansel et al., 1993. The temporal query language TSQL2 and its underlying database model is described in Zaniolo et al., 1997, and more elaborately in Snodgrass, 1995.

7.12 EXERCISES

Consider the following tables of a library database Δ_{Lib}:

LibraryBook

InvNo	Title	AuthorEditor	Year	*Valid Time*
880	Intentionality	Searle	1983	1/10/89..*uc*
988	Formal Methods	Bowen	1995	1/3/96..*uc*
284	The Blue Book	Wittgenstein	1958	1/1/73..21/3/82, 1/9/82..*uc*
921	Vivid Logic	Wagner	1994	1/5/94..*uc*
903	Brainstroms	Dennett	1978	1/1/83..*uc*
926	Sofie's World	Gaarder	1993	1/1/94..13/11/95

IsLentOutTo

InvNo	Name	*Valid Time*
880	Wagner	2/12/96..*uc*
988	Kuhn	15/11/96..20/12/96, 31/12/96..31/1/97
284	Hanselmann	1/4/95..12/6/95, 2/2/97..*uc*
880	Kuhn	15/11/96..1/12/96
903	Wagner	20/9/96..11/2/97
284	Kopka	1/11/96..31/12/96

over the table schemas *lb* and *ilt*. Let X_{Lib} be the propositional representation of Δ_{Lib}, and let the 10/2/1997 be the current instant (of query evaluation).

Problem 7-1. Form the snapshot of Δ_{Lib} as of 1/1/97.

Problem 7-2. Express the following if-queries to Δ_{Bib} in natural language and determine their answer

1. $\exists x(\text{ilt}(284, x))@1/1/97$
2. $\exists x \exists t(\text{ilt}(x, \text{Kopka})@t)$
3. $\forall x \forall y \forall t(\text{ilt}(x, y)@t \supset \exists u \exists v \exists w(\text{lb}(x, u, v, w)@t))$

Problem 7-3. Formalize the following queries and compute their answer sets.

1. Which books are now available ?
2. When was the Blue Book lent out ?

8 FUZZY DATABASES

Uncertain information shows up in various ways, typically not in business and administration domains but rather in more empirical domains such as in medicine, criminology, and in all kinds of empirical research. After discussing different types of uncertainty, we present a fuzzy database model for storing and retrieving gradually uncertain information.

8.1 INTRODUCTION

In various application domains, one encounters uncertain and imprecise information. While the **fuzzy set theory** of Zadeh, 1965, allows to handle *imprecise* or *vague* information, *fuzzy logic* and **possibilistic logic**, which are based on the *possibility theory* of Zadeh, 1979; Dubois and Prade, 1980; Dubois et al., 1994, form the theoretical basis for processing *gradually uncertain* information.

Classical probability theory, based on the axioms of Kolmogorov, on the other hand, makes very strong assumptions. In many practical cases, it is therefore not applicable to the problem of handling uncertain information. Also, probabilistic methods are computationally much more costly than fuzzy methods because they are non-compositional.

Imprecise information often comes in the form of *disjunctive* or *vague* sentences, while uncertain information is often expressed in the form of sentences qualified by a *degree of uncertainty* like, for instance, *'very likely'* or *'with 60% certainty'*. Sometimes, such a qualification is called a *subjective probability* or a *degree of belief*. A particular form of uncertainty is given by *statistical information*. Sentences qualified by statistical uncertainty are based on statistical samples measuring *relative frequencies*. An example of such a sentence is

80% of all jaundice patients have hepatitis.

Through their empirical grounding, these sentences are more reliable than sentences qualified by a mere subjective degree of belief.

In the literature, there is no clear taxonomy of uncertainty in databases and knowledge bases. An obvious distinction, however, concerns the uncertainty expressed at the level of attribute values, or at the level of the applicability of a predicate to a tuple. While the former is related to the issue of handling *vagueness*, the latter is related to an account of certainty-qualified sentences. In our formal exposition, we do not treat vagueness expressed by fuzzy-set-valued

attributes, but only the more fundamental issue of gradual uncertainty based on fuzzy relations over ordinary ('crisp') attributes.

8.1.1 Disjunctive Imprecision

A disjunctively imprecise attribute value specifies a set of possible values without any commitment to (or preference for) a particular one of them. Examples are

1. *'The culprit escaped with a black BMW or Mercedes-Benz.'* This sentence could be formalized as follows:
$$\text{escape(BMW)} \vee \text{escape(Mercedes-Benz)}$$
2. *'Teleman's concert No. 9 for flute has been played for the first time between 1743 and 1745.'* This sentence could be formally expressed by using a set-valued attribute like
$$\text{premiere}(9, \{1743, 1744, 1745\})$$
or in the form of a disjunction
$$\text{premiere}(9, 1743) \vee \text{premiere}(9, 1744) \vee \text{premiere}(9, 1745)$$

Disjunctive imprecision can be represented in *disjunctive databases* which are defined in Section 9.3.

8.1.2 Vagueness

A vague attribute value specifies a range of possible values with certain preferences among them. These preferences are expressed by possibility weights. Consequently, a vague attribute value can be represented by a fuzzy set. For instance, we may have the information that *Peter is about 30.* This vague sentence can be represented by

$$\text{age}(\text{Peter}, \{28{:}0.5, 29{:}1, 30{:}1, 31{:}1, 32{:}0.5\})$$

using the fuzzy set $\{28{:}0.5, 29{:}1, 30{:}1, 31{:}1, 32{:}0.5\}$. Notice that the disjunctively imprecise sentence *'Peter is 30 or 31.'* is more informative. It corresponds to

$$\text{age}(\text{Peter}, \{30, 31\})$$

In relational databases, the only way to represent vague or disjunctively imprecise attribute values is to replace them by the null value *UNKNOWN* and accept the loss of information implied by this reduction.

8.1.3 Gradual Uncertainty

Gradual uncertainty is expressed by means of linguistic qualifiers such as *probably* or *very likely*, or by explicitly stating a numeric degree of belief. An example of a gradually uncertain sentence is

> It is very likely that Peter is 31.

Gradual uncertainty can be combined with vagueness:

> It is very likely that Peter is about 30.

In the sequel, we are only concerned with gradual uncertainty and not with vagueness.

8.2 CERTAINTY SCALES

Information units in fuzzy databases are certainty-qualified atoms, such as, e.g., $p(c)$:0.7, where 0.7 represents the certainty value with which the sentence $p(c)$ is asserted. Certainty values may be numbers between 0 and 1, or linguistic terms such as *little likely*, *quite likely*, etc.

Definition 8.1 *A* ***certainty scale*** $\langle \boldsymbol{C}, 0, 1\rangle$ *is a linearly ordered set* $\boldsymbol{C}$ *with least and greatest elements* 0 *and* 1.[1]

Examples of certainty scales are

1. the rational unit interval $[0, 1]$, or
2. any discrete ordering of linguistic certainty values such as

$$\langle 0, ll, ql, vl, 1\rangle,$$

where ll stands for *little likely*, ql for *quite likely*, and vl for *very likely*.

In the sequel, we assume that there is a fixed certainty scale $\boldsymbol{C}$ for which we simply write $[0, 1]$. If we want to exclude the value for complete uncertainty, we write $(0, 1] = \{v \in \boldsymbol{C} \mid v > 0\}$.

A linear ordering with more than two elements is a distributive lattice, but not a Boolean algebra. Consequently, there can be no Boolean connectives in fuzzy logics. Depending on the choice of the complement operation, a linear ordering may be a DeMorgan or a Heyting algebra. The original definition of the fuzzy set complement by Zadeh, 1965, corresponds to the DeMorgan complement. But in a DeMorgan algebra, neither the *law of the excluded middle*,

$$\text{(LEM)} \qquad \text{val}(F \vee \neg F) = 1,$$

nor the *law of the excluded contradiction*,

$$\text{(LEC)} \qquad \text{val}(F \wedge \neg F) = 0$$

holds, while in a Heyting algebra at least the latter one is satisfied. In contrast to ordinary fuzzy logic, we therefore choose the Heyting complement as the basis for evaluating negation, following Wagner, 1997. There is a second fundamental reason for this choice: it yields a system that is upward compatible with inference in relational and deductive databases.

Observation 8.1 *A certainty scale* $\boldsymbol{C}$ *corresponds to a* Heyting algebra *whose implication-free fragment is* $\langle \boldsymbol{C}, \min, \max, \frown\rangle$, *where the complement* $\frown$ *is defined as*

$$\frown v = \begin{cases} 0 \textit{ if } v > 0 \\ 1 \textit{ otherwise} \end{cases}$$

Thus, while conjunction and disjunction are evaluated in a certainty scale by *min* and *max*, negation is evaluated by the Heyting complement $\frown$.[2]

It is important to note that in fuzzy databases, unlike in probability theory, the intuitive meaning of 0 is not *false*, or 'impossible', but rather *completely uncertain*, or absolutely *no information*.

8.3 FUZZY TABLES

A **fuzzy table** P over a certainty scale $\boldsymbol{C}$ and a table schema $p(A_1, \ldots, A_n)$ is a function

$$P : \text{dom}(A_1) \times \ldots \times \text{dom}(A_n) \to \boldsymbol{C}$$

assigning non-zero certainty values $P(\boldsymbol{c}) > 0$ only to a finite number of tuples $\boldsymbol{c} \in \text{dom}(A_1) \times \ldots \times \text{dom}(A_n)$. A fuzzy table has a designated functional column for the assigned certainty value, listing only the non-zero assignments.

A fuzzy database schema consists of a certainty scale, a sequence of table schemas and a set of integrity constraints. A fuzzy database Δ over a schema

$$\Sigma = \langle \boldsymbol{C}, \langle p_1, \ldots, p_m \rangle, IC \rangle$$

is a sequence of fuzzy tables $\langle P_1, \ldots, P_m \rangle$ over $\boldsymbol{C}$, such that P_i is the extension of the fuzzy predicate p_i in Δ. A fuzzy database Δ can be propositionally represented as a set X_Δ of certainty-valuated atoms of the form $a{:}\nu$ such that $\nu \in \boldsymbol{C}$ is the certainty value assigned to the fact a.

Example 3 *The fuzzy database consisting of the two tables*

$$P = \begin{array}{|c|c|}\hline d & 1 \\ \hline b & 0.7 \\ \hline c & 0.1 \\ \hline \end{array} \qquad Q = \begin{array}{|c c|c|}\hline d & c & 0.8 \\ \hline d & b & 0.3 \\ \hline \end{array}$$

corresponds to $X_3 = \{p(d){:}1, p(b){:}0.7, p(c){:}0.1, q(d,c){:}0.8, q(d,b){:}0.3\}$.

8.4 NATURAL INFERENCE IN FUZZY DATABASES

A fuzzy database X corresponds to a **certainty valuation**

$$C_X : L^0_{\text{Query}} \to \boldsymbol{C}$$

which is induced by X in a natural way. For $a \in \text{At}^0_\Sigma$, $F, G \in L^0_\Sigma$, $H[x] \in L_\Sigma$, and $\mu \in [0,1]$, we define

$$\begin{array}{lrcl}
(a) & C_X(a) & = & \left\{ \begin{array}{l} \mu \text{ if } a{:}\mu \in X \\ 0 \text{ otherwise} \end{array} \right. \\[2ex]
(\neg) & C_X(\neg F) & = & \sim C_X(F) \\
(\wedge) & C_X(F \wedge G) & = & \min(C_X(F), C_X(G)) \\
(\vee) & C_X(F \vee G) & = & \max(C_X(F), C_X(G)) \\
(\exists) & C_X(\exists x H[x]) & = & \max\{\mu \mid c{:}\mu \in \text{CAns}(X, H[x])\} \\
(\forall) & C_X(\forall x H[x]) & = & \min\{\mu \mid c{:}\mu \in \text{CAns}(X, H[x])\} \\
(\text{CAns}) & \text{CAns}(X, H[x]) & = & \{c{:}\mu \mid \mu = C_X(H[c]) > 0\}
\end{array}$$

Notice that $c{:}\mu$ is just a shorthand notation for the uncertain tuple $\langle c, \mu \rangle$.

Then, the **natural inference** relation between a fuzzy database and a certainty-valuated if-query is defined as

$$X \vdash F{:}\mu \quad :\Longleftrightarrow \quad C_X(F) \geq \mu$$

Thus, a qualified sentence $F{:}\mu$ can be inferred, if the sentence F is at least as certain as μ. This implies, for instance, that if some qualified sentence, say *'it is very likely that agent Cooper carries a weapon'*, can be inferred, then all weaker qualifications of it, such as *'it is quite likely that agent Cooper carries a weapon'* are inferable as well.

Query formulas can also be qualified by a **certainty interval**,

$$X \vdash F : [\mu, \nu] \quad :\Longleftrightarrow \quad \mu \leq C_X(F) \leq \nu,$$

requiring that the sentence in question is at least as certain as μ but not more than ν.

In the case of the above example, we obtain the following inference:

$$X_3 \vdash (p(b) \wedge \neg \exists x q(b,x)) : 0.7$$

Model-theoretically, the natural inference relation $\vdash$ in fuzzy databases corresponds to minimal entailment $\models_m$ in semi-possibilistic logic (see Appendix B.4).

Claim 3 $X \vdash F{:}\mu$ *iff* $X \models_m F{:}\mu$.

8.5 UPDATE, KNOWLEDGE INTEGRATION AND INFORMATION ORDER

The admissible inputs in fuzzy databases are certainty-valuated literals of the form $a{:}\mu$ and $\neg a : \mu$, where a is an atom and μ is a certainty value. The update operation is defined as follows:

$$\mathsf{Upd}(X, a{:}\mu) \quad = \quad \begin{cases} X - \{a{:}\nu\} \cup \{a{:}\mu\} & \text{if } \; C_X(a) = \nu < \mu \\ X \text{ otherwise} \end{cases}$$

$$\mathsf{Upd}(X, \neg a : \mu) \quad = \quad \begin{cases} X - \{a{:}\nu\} & \text{if } \; C_X(a) = \nu \;\; \& \;\; \mu > 0 \\ X \text{ otherwise} \end{cases}$$

The knowledge system of fuzzy databases, denoted by $F\boldsymbol{A}$, is then defined as

$$F\boldsymbol{A} \;=\; \begin{array}{|l|l|} \hline L_{KB} & 2^{\mathrm{At}^0 \times [0,1]} \\ \hline L_{\mathrm{Query}} & L(\neg, \wedge, \vee, \exists, \forall) \cup L(\neg, \wedge, \vee, \exists, \forall) \times [0,1] \\ \hline L^0_{\mathrm{Ans}} & \{\mathsf{yes}, \mathsf{no}\} \cup \{\mathsf{yes}\} \times (0,1] \\ \hline L_{\mathrm{Input}} & \mathrm{Lit} \times [0,1] \\ \hline L_{\mathrm{Unit}} & \mathrm{At} \times [0,1] \\ \hline \end{array}$$

In general, fuzzy databases are not complete. That is, there may be if-queries F such that neither $X \vdash F{:}1$ nor $X \vdash \neg F{:}1$. A new input sentence involving fuzzy predicates may come from an *epistemically independent* source. In that case, a **knowledge integration** operation should be used to assimilate the new input:

$$\mathsf{Int}(X, a{:}\mu) \quad = \quad \begin{cases} X - \{a{:}\nu\} \cup \{a : (\nu + (1-\nu)\mu)\}, & \text{if } \; C_X(a) = \nu > 0 \\ X \cup \{a{:}\mu\}, & \text{if } \; C_X(a) = 0 \end{cases}$$

Notice that the formula $\nu + (1-\nu)\mu$ combines the two independent evidences ν and μ as proposed in Shortliffe and Buchanan, 1975.

The information order $\leq$ is defined as follows. Let X, Y be two fuzzy databases. We say that Y *informationally extends* X, or Y is *at least as informative* as X, symbolically $X \leq Y$, if for all $a{:}\nu \in X$, $\nu \leq C_Y(a)$. This means that a fuzzy database (or a table) contains more information than another one, if it contains additional rows, or the certainty of some row in it is increased. For instance,

$$\begin{array}{|l|l|} \hline d & 0.9 \\ \hline b & 0.7 \\ \hline \end{array} \;<\; \begin{array}{|l|l|} \hline d & 0.9 \\ \hline b & 0.7 \\ \hline c & 0.1 \\ \hline \end{array} \;<\; \begin{array}{|l|l|} \hline d & 1 \\ \hline b & 0.7 \\ \hline c & 0.1 \\ \hline \end{array}$$

Claim 4 *$F\boldsymbol{A}$ is vivid.*

Proof: We have to define the required embedding function h. Let $X \subseteq \mathrm{At}^0$, and $F \in L^0(\neg, \wedge, \vee, \exists, \forall)$. Setting $h(F) = F{:}1$, one can show by straightforward induction on F that

$$\mathsf{Upd}_A(X, l) \vdash_A F \iff \mathsf{Upd}(\mathsf{Upd}(0, h(X)), h(l)) \vdash h(F) \quad \square$$

where $\vdash_A$ is the inference relation of relational databases.

8.6 FUZZY TABLE OPERATIONS

Again, in order to 'implement' *CAns* we define the answer operation Ans inductively. Unqualified open queries are evaluated compositionally by means of the fuzzy table operations *selection*, *projection*, *join*, *union* and *difference*. If X is a fuzzy database over Σ, and the table $P : p$ represents an n-place extensional predicate in X, then

$$\mathsf{Ans}(X, p(x_1, \ldots, x_n)) = P$$

that is, the answer to an atomic query $p(x_1, \ldots, x_n)$ is the fuzzy extension P of predicate p in the database state X including the designated column with the assigned certainty values.

The selection operation $\mathrm{Sel}(C, Q)$, where C is a plain row constraint and Q is a fuzzy table, is defined like in plain relational algebra: $\mathrm{Sel}_f(C, Q) = \mathrm{Sel}(C, Q)$.

In the case of the projection operation, it may be necessary to *normalize* the result: if the projection result contains tuples that agree on all attributes and differ only in their certainty value, then only that tuple among them with the greatest certainty value is retained and all others are discarded. We denote this **normalization operation** by Norm_f. Thus,

$$\mathrm{Proj}_f(\langle j_1, \ldots, j_k \rangle, P) = \mathrm{Norm}_f(\mathrm{Proj}(\langle j_1, \ldots, j_k \rangle, P))$$

Existentially quantified query formulas in fuzzy databases are evaluated by means of fuzzy projection. Let $F = F[x_1, \ldots, x_k]$, and $1 \leq i \leq k$. Then,

$$\mathsf{Ans}(X, \exists x_i F) = \mathrm{Proj}_f(\langle 1, \ldots, i-1, i+1, \ldots, k \rangle, \mathsf{Ans}(X, F))$$

Conjunction and Join. The algebraic basis of evaluating conjunction is the cross product of two fuzzy tables defined as follows:

$$P \times_f Q = \{\langle c_1, \ldots, c_m, d_1, \ldots, d_n, \min(\mu, \nu) \rangle \mid \langle \boldsymbol{c}, \mu \rangle \in P \ \ \& \ \ \langle \boldsymbol{d}, \nu \rangle \in Q\}$$

Let $i = 1, \ldots, m$ and $j = 1, \ldots, n$. Then,

$$P \overset{i=j}{\bowtie}_f Q = \mathrm{Proj}_f(\langle 1, \ldots, m+j-1, m+j+1, \ldots, m+n \rangle, \mathrm{Sel}_f(\$i = \$j, P \times_f Q))$$

The definition of $\overset{i=j}{\bowtie}$ can be easily generalized to the case of a conjunction with more than one shared variable. If all shared variables are treated in this way, the resulting operation $\bowtie$ is called *fuzzy join*. As in relational algebras, it is used to evaluate conjunctions:

$$\mathsf{Ans}(X, F \wedge G) = \mathsf{Ans}(X, F) \bowtie_f \mathsf{Ans}(X, G)$$

Disjunction and Union. The union of two fuzzy tables (of the same type) is obtained by first forming their plain set-theoretic union, and then normalizing it:

$$P \cup_f Q = \mathrm{Norm}_f(P \cup Q)$$

As in relational databases, union is the algebraic counterpart of disjunction:

$$\mathsf{Ans}(X, F \vee G) = \mathsf{Ans}(X, F) \cup_f \mathsf{Ans}(X, G)$$

Negation and Difference. Let P and Q be tables with n and m columns. The difference of P and Q is defined only if $m \leq n$ and both tables agree on the types of their columns $1, \ldots, m$. Then,

$$P -_f Q = \{\langle c_1, \ldots, c_n, \nu \rangle \in P \mid \langle c_1, \ldots, c_m, \mu \rangle \notin Q\}$$

As in relational databases, negation in query formulas is only evaluable in fuzzy databases in conjunction with a positive choice to which the negative condition refers. Such a conjunction is evaluated by the fuzzy difference operation:

$$\mathsf{Ans}(X, F \wedge \neg G) = \mathsf{Ans}(X, F) -_f \mathsf{Ans}(X, G)$$

8.6.1 Correctness and Completeness

In order to show that this definition adequately captures the inference-based evaluation of queries one has to prove that for every evaluable query formula F,

(correctness) if $\langle c_1, \ldots, c_m, \mu \rangle \in \mathsf{Ans}(X, F[x_1, \ldots, x_m])$
then $X \vdash F[c_1, \ldots, c_m] : \mu$

(completeness) if $X \vdash F[c_1, \ldots, c_m] : \mu \ \ \& \ \ \mu > 0$
then $\langle c_1, \ldots, c_m, \mu \rangle \in \mathsf{Ans}(X, F[x_1, \ldots, x_m])$

The proof of this theorem is left as an exercise.

8.6.2 Four Kinds of Queries

In addition to the case of unqualified open queries defined above, the answer operation is also defined for qualified and unqualified if-queries and for qualified open queries. Let $F \in L^0_\Sigma$ be an if-query, $G \in L_\Sigma$ an open query, and $\mu \in [0,1]$ a certainty value. Then, we define

$$\mathsf{Ans}(X, F) = \begin{cases} \mathsf{yes}{:}\mu, & \text{if } \mu = C_X(F) > 0 \\ \mathsf{no}, & \text{otherwise} \end{cases}$$

$$\mathsf{Ans}(X, F{:}\mu) = \begin{cases} \mathsf{yes}, & \text{if } C_X(F) \geq \mu \\ \mathsf{no}, & \text{otherwise} \end{cases}$$

$$\mathsf{Ans}(X, G{:}\mu) = \{\sigma \mid C_X(G\sigma) \geq \mu\}$$

Notice that for any open query formula F, $F{:}0$ is not domain independent, and therefore, zero-qualified formulas have to be excluded from the set of evaluable query formulas.

8.7 FUZZY DEDUCTION RULES

Let a be an atom, and let l_i be literals. A normal fuzzy deduction rule r is an expression of the form

(1) $a \leftarrow l_1 \wedge \ldots \wedge l_k$, or

(2) $a \xleftarrow{\mu} l_1 \wedge \ldots \wedge l_k$, where $\mu \in (0,1]$ is a rational number, or

(3) $a{:}\mu \leftarrow l_1{:}\nu_1 \wedge \ldots \wedge l_k{:}\nu_k$.

A ground rule r of the form (1) is applied to a fuzzy database X according to

$$r(X) = \begin{cases} \mathsf{Upd}(X, a{:}\nu), & \text{if } C_X(l_1 \wedge \ldots \wedge l_k) = \nu > 0 \\ X, & \text{otherwise} \end{cases}$$

A weighted ground rule r of the form (2) is applied according to

$$r(X) = \begin{cases} \mathsf{Upd}(X, a{:}\mu\nu), & \text{if } C_X(l_1 \wedge \ldots \wedge l_k) = \nu > 0 \\ X, & \text{otherwise} \end{cases}$$

Finally, a ground rule r of the form (3) is applied according to

$$r(X) = \begin{cases} \mathsf{Upd}(X, a{:}\mu), & \text{if } X \vdash l_1{:}\nu_1 \wedge \ldots \wedge l_k{:}\nu_k \\ X, & \text{otherwise} \end{cases}$$

A normal fuzzy deductive database is a pair $\langle X, R\rangle$, consisting of a fuzzy database X and a set of normal fuzzy deduction rules R. The notions of positive/persistent and stratifiable databases, as well as stable closures and well-founded inference are defined as in Chapter 5.

Interestingly, there are nonmonotonic fuzzy deductive databases without negation since queries qualified with certainty intervals are not persistent. For instance,

$$\{p(c){:}0.5\} \vdash p(c){:}[0.4, 0.6]$$

but $\{p(c){:}0.7\} \nvdash p(c){:}[0.4, 0.6]$, although $\{p(c){:}0.5\} \leq \{p(c){:}0.7\}$. Therefore, a fuzzy deductive database may be nonmonotonic even if its rules do not contain any negation. Consider the following example:

$$\begin{aligned} X &= \{r(c){:}0.9\} \\ R &= \{p(x){:}1 \leftarrow q(x){:}[0.4, 0.6] \wedge r(x){:}0.8, \\ &\quad\ q(x){:}0.5 \leftarrow r(x){:}0.5, \\ &\quad\ q(x){:}0.8 \leftarrow p(x){:}[0, 0.5]\} \end{aligned}$$

This database has two stable generated closures:

$$\mathrm{SGenC}(X,R) = \{\{r(c){:}0.9, q(c){:}0.5, p(c){:}1\},\ \{r(c){:}0.9, q(c){:}0.8\}\}$$

On the other hand,

$$\lim_{i\to\infty} \Gamma^{2i}_{X,R}(X) = \lim_{i\to\infty} \Gamma^{2i+1}_{X,R}(X) = \{r(c){:}0.9, q(c){:}0.8\}$$

and, consequently, our definition of wellfounded inference in 5.2.3 is not adequate in the case of fuzzy deductive databases with certainty intervals.

If different rules for the same conclusion are to represent different lines of reasoning that are epistemically independent, and each of them is to contribute to the resulting overall evidence in favor of the conclusion, then the intended closure should be formed by means of knowledge integration and not by means of update. This choice concerns the definition of rule application and the definition of a stable generated closure, where Upd is replaced by Int.

8.8 SUMMARY

The admission of fuzzy tables in databases allows to represent gradually uncertain information which is needed in many empirical domains. As in relational databases, the answer operation is 'implemented' by algebraic operations on tables, and its correctness is shown with respect to natural inference based on the certainty valuation associated with a fuzzy database.

8.9 FURTHER READING

Various forms of gradual uncertainty handling in databases, including a probabilistic approach, are discussed in Zaniolo et al., 1997. The use of fuzzy set theory to represent imprecision in databases is the topic of Petry, 1995. Database tables and queries with vague attribute values represented by fuzzy sets are also discussed in Yu and Meng, 1998.

8.10 EXERCISES

Problem 8-1. Consider the fuzzy data base $\Delta_{\mathrm{Hosp}} = \{\mathit{Diagnosis} : d, \mathit{Prognosis} : p\}$ over the certainty scale

$$[0,1] = \langle 0, ll, ql, vl, 1\rangle,$$

where the certainty values ll, ql, vl stand for *little likely*, *quite likely* and *very likely*, respectively, with the following tables

Diagnosis

PatNo	Desease	
013	hepA	*vl*
117	jaundice	1
117	hepA	*vl*
013	jaundice	*vl*
106	hepB	*ql*

Prognosis

PatNo	ExpectedDuration	
013	4 weeks	*ql*
106	8 weeks	*ll*
117	6 weeks	*vl*

1. Does $X_{\mathrm{Hosp}} \vdash (\exists x(d(x,\mathrm{hepA}) \wedge \neg d(x,\mathrm{jaundice}))) : vl$ hold ? How can this query be expressed in natural language ?
2. For which hepatitis patients is a prognosis of not more than 6 weeks quite likely ? Formalize this natural language query and compute its answer set by means of the fuzzy table operations.

Problem 8-2. Prove that the fuzzy answer operation Ans defined by the fuzzy table operations is correct and complete.

Notes

1. From a purely formal point of view, one could as well choose the more general structure of a (distributive) lattice of certainty values instead of a linear order. But from a practical point of view, there is no need for such a generalization.

2. The Heyting complement $\frown$ is a discontinuous operator. When the certainty value of p changes from 0.001 to 0, then the value of $\frown p$ jumps from 0 to 1. This behavior may be undesirable in control theory, but not in logic and databases where it makes a clear difference if there is any evidence in favor of p or absolutely none. Notice also that $\frown$ is continuous in the interval $(0, 1]$.

9 FURTHER EXAMPLES OF POSITIVE KNOWLEDGE SYSTEMS

In this chapter, we consider four further examples of positive knowledge systems: multi-level secure, lineage, disjunctive and S5-epistemic databases.

9.1 MULTI-LEVEL SECURE DATABASES

In certain applications, it is essential to protect confidential information from unauthorized access. In *multi-level secure (MLS)* databases,[1] all information items are assigned a security classification, and all database users are assigned a security clearance, both from a partially ordered set of security levels. For instance, the four security levels *unclassified* (0), *confidential* (1), *secret* (2), and *top secret* (3) may be used to classify entries in a MLS table.

As an example, consider the database of a hospital. Depending on the respective person it may be sensitive information to know whether someone is a patient in the hospital. In the case of a politician, such information would be publicly available. But not so in the case of a shy pop star, or a secret service agent.

Example 4 *The following MLS table of a hospital database* X_{Hosp} *contains the records of patients.*

$$Patient = \begin{array}{|l|l|}\hline \textit{Boris Yeltsin (BY)} & \textit{0} \\ \textit{Michael Jackson (MJ)} & \textit{1} \\ \textit{James Bond (JB)} & \textit{3} \\ \hline \end{array}$$

This table represents the following beliefs:

clearance level	*beliefs*
0	$\{p(BY)\}$
1,2	$\{p(BY), p(MJ)\}$
3	$\{p(BY), p(MJ), p(JB)\}$

Notice that it is not possible to preserve privacy and maintain security by simply omitting information, like in the reply *'no answer'* to the question *'Is Michael Jackson a patient in*

this hospital ?'. The asking reporter could easily infer from this refusal to answer that MJ must be a patient in the hospital. The only way to maintain security is to give a wrong answer, i.e. to misinform the unauthorized asker. The rationality principle of secure inference is the **Principle of Minimal Misinformation**, that is, askers are only misinformed about an information item, if they are not sufficiently authorized with respect to that item.

Assume, for instance, that Boris Yeltsin is in the hospital with a heart attack, but this fact is not to be released to the public. When a reporter asks if BY is in the hospital, he receives the answer 'yes'. If he then asks whether BY has a heart attack, the secure answer may be 'no' or 'unknown'. When being asked what BY suffers from, the hospital information system may reply to the reporter that he has a severe influenza.

MLS queries are labelled by the clearance level of the asker. Since reporters have clearance level 0 (unclassified), we get

$$\mathsf{Ans}(X_{\mathrm{Hosp}}, p(MJ)\sharp 0) = \mathsf{no}$$

while a doctor of the hospital with clearance level 2 would get the right answer: *'yes, MJ is a patient in this hospital'*,

$$\mathsf{Ans}(X_{\mathrm{Hosp}}, p(MJ)\sharp 2) = \mathsf{yes}$$

Definition 9.1 (Security Hierarchy) *A* security hierarchy *SH is a finite partial order with a greatest element denoted by $\top$.*

A **MLS table** P over a security hierarchy SH and a table schema $p(A_1, \ldots, A_n)$ is a partial function

$$\mathrm{dom}(A_1) \times \ldots \times \mathrm{dom}(A_n) \to SH$$

assigning a security level to a finite number of tuples.

A **MLS database** Δ over a schema

$$\Sigma = \langle SH, \langle p_1, \ldots, p_m \rangle, IC \rangle$$

is a finite sequence of MLS tables $\langle P_1 : p_1, \ldots, P_m : p_m \rangle$ over the security hierarchy SH. Its propositional representation is

$$X_\Delta = \bigcup_{i=1}^{m} \{p_i(\boldsymbol{c})\sharp\lambda \mid \langle \boldsymbol{c}, \lambda \rangle \in P_i\}$$

It can be decomposed into a set of relational databases $\{\Delta^\lambda \mid \lambda \in SH\}$, such that

$$\begin{array}{lcl} \Delta^\lambda & = & \langle P_1^\lambda, \ldots, P_m^\lambda \rangle \\ P_i^\lambda & = & \{\boldsymbol{c} \mid \langle \boldsymbol{c}, \kappa \rangle \in P_i \ \ \& \ \ \kappa \leq \lambda\} \end{array}$$

We write X^λ, instead of X_{Δ^λ}, for the propositional representation of Δ^λ.

Queries in MLS databases have to arrive with the clearance level of the asker. This assignment is implicitly performed by the user interface on top of the MLS database. Askers not known to the database user interface may be assigned the lowest clearance level by default.

Definition 9.2 (Secure Inference) *Let X be a MLS database, $\lambda, \kappa \in SH$, $l \in \mathrm{Lit}^0$, $F_1, F_2 \in L(\neg, \wedge, \vee, \exists, \forall, \mathsf{B}_\lambda)$, and $H[x] \in L(\neg, \wedge, \vee, \exists, \forall, \mathsf{B}_\lambda)$.*

$$\begin{array}{lrcl} (l) & X \vdash l\sharp\lambda & :\Longleftrightarrow & X^\lambda \vdash_A l \\ (\wedge) & X \vdash (F_1 \wedge F_2)\sharp\lambda & :\Longleftrightarrow & X \vdash F_1\sharp\lambda \ \ \& \ \ X \vdash F_2\sharp\lambda \\ (\vee) & X \vdash (F_1 \vee F_2)\sharp\lambda & :\Longleftrightarrow & X \vdash F_1\sharp\lambda \ \ \textit{or} \ \ X \vdash F_2\sharp\lambda \\ (\exists) & X \vdash (\exists x H[x])\sharp\lambda & :\Longleftrightarrow & \mathrm{CAns}(X, H[x]\sharp\lambda) \neq \emptyset \\ (\forall) & X \vdash (\forall x H[x])\sharp\lambda & :\Longleftrightarrow & \mathrm{CAns}(X, \neg H[x]\sharp\lambda) = \emptyset\} \\ (\mathrm{CAns}) & \mathrm{CAns}(X, H\sharp\lambda) & = & \{\sigma \mid X \vdash (H\sigma)\sharp\lambda\} \end{array}$$

It may be useful to be able to ask questions relative to others' viewpoints. For example, the nurse (with clearance level 1) might need to ask, *'If a reporter asks for a list of the current patients, what will be the answer ?'*. She would put this as the query $\mathsf{B}_0 p(x)$, using a belief level operator B_0 referring to the clearance level 0 assigned to reporters. The answer of X_{Hosp} is

$$\mathsf{Ans}(X_{\mathrm{Hosp}}, (\mathsf{B}_0 p(x))\sharp 1) \;=\; \{BY\}$$

There is no need to allow for nested B operators. Let $G \in L^0(\neg, \wedge, \vee, \exists, \forall)$.

$$(\mathsf{B}_\kappa) \quad X \vdash (\mathsf{B}_\kappa G)\sharp\lambda \quad :\Longleftrightarrow \quad \kappa \leq \lambda \;\;\&\;\; X^\kappa \vdash_A G$$

Observation 9.1 *If $F \in L^0(\neg, \wedge, \vee, \exists, \forall)$, then*

$$X \vdash F\sharp\lambda \iff X \vdash (\mathsf{B}_\lambda F)\sharp\lambda$$

Information units in MLS databases are security-qualified atoms. Let a be an atom, and l a literal. Then, **secure update** is defined as

$$\begin{aligned}
\mathsf{Upd}(X, (\mathsf{B}_\kappa a)\sharp\lambda) &= \begin{cases} X \cup \{a\sharp\kappa\} & \text{if } \kappa \leq \lambda \;\&\; X \nvdash a\sharp\kappa \\ X & \text{otherwise} \end{cases} \\
\mathsf{Upd}(X, (\mathsf{B}_\kappa \neg a)\sharp\lambda) &= X - \{a\sharp\mu \in X \mid \mu \leq \kappa \leq \lambda\} \\
\mathsf{Upd}(X, l\sharp\lambda) &= \mathsf{Upd}(X, (\mathsf{B}_\lambda l)\sharp\lambda)
\end{aligned}$$

A MLS database X is at least as informative as Y, if at the top-level it contains at least as much information as Y, symbolically

$$X \leq Y :\Longleftrightarrow X^\top \subseteq Y^\top$$

This definition is justified by the fact that it is only guaranteed at the top-level that the information is complete. The knowledge system of MLS databases, denoted by $\boldsymbol{SA}$, is then defined as

$\boldsymbol{SA} =$

L_{KB}	$2^{\mathrm{At}^0 \times SH}$
L_{Query}	$L(\neg, \wedge, \vee, \exists, \forall, \mathsf{B}_\lambda) \times SH$
L^0_{Ans}	$\{\mathsf{yes}, \mathsf{no}\}$
L_{Input}	$\mathrm{Lit} \times SH$
L_{Unit}	$\mathrm{At} \times SH$

Claim 5 *$\boldsymbol{SA}$ is vivid.*

Proof: Let $X \subseteq \mathrm{At}$, $F \in L^0(\neg, \wedge, \vee, \exists, \forall)$, and let $\top \in SH$ be the top security level. Then, $h(F) = F\sharp\top$ defines an embedding of a relational database X in a MLS database. Clearly,

$$\mathsf{Upd}_A(X, l) \vdash_A F \iff \mathsf{Upd}(\mathsf{Upd}(0, h(X)), h(l)) \vdash h(F) \quad \square$$

Notice that in MLS databases, it is not possible to protect negative information by providing suitable misinformation. If, for instance, a hospital has to pretend that James Bond is among its patients, i.e. the negative information $\neg p(\mathrm{JB})$ has to be protected, say at clearance level 3 (top secret), this cannot be achieved by means of simple MLS tables which would have to record negative entries in addition to positive ones. Negative information can be protected, however, in *MLS bitables* which are introduced in 13.1.

9.2 LINEAGE DATABASES

In relational databases it is assumed that all information sources authorized to enter new information into the database are equally competent, completely reliable and honest, and

therefore new information always overrides old information. Under these assumptions, it is not necessary to distinguish between different information suppliers. They all have the status of the owner of the database or at least of specific tables.

In some application domains, however, these assumptions are not realistic, and different information sources have to be distinguished in order to identify epistemically independent inputs and to take possibly different degrees of reliability into account. The knowledge system of *lineage databases* represents a principled approach to the problem of lineage and reliability. Its main principles are:

1. A lineage database labels atoms a with their source s in the form of a/s. If a new input affects an already-stored piece of information, the result of assimilating it into the database depends on its source label: if it comes from the same source as the already-stored item, an (overriding) update is performed; if it comes from a different source, a (combining) knowledge integration is performed when querying the resulting database.

2. A lineage database may assign reliability degrees $\rho(s) \in [0,1]$ to its information sources s. Complete reliability is expressed by $\rho(s) = 1$. If $\rho(s) = 0$, the source s is completely unreliable, and hence completely uninformative. Reliability degrees correspond to certainty values.

Definition 9.3 *A **reliability assignment** is a function $\rho : SL \to [0,1]$, assigning a reliability degree $\rho(s)$ to each information source label $s \in SL$.*

A **SL table** P over a set of source labels SL and a table schema $p(A_1, \ldots, A_n)$ is a partial function

$$P : \mathrm{dom}(A_1) \times \ldots \times \mathrm{dom}(A_n) \to SL$$

assigning an information source label to a finite number of tuples.

A **lineage database** (or SL database) Δ over a schema

$$\Sigma = \langle SL, \rho, \langle p_1, \ldots, p_m \rangle, IC \rangle$$

is a finite sequence of SL tables $\langle P_1 : p_1, \ldots, P_m : p_m \rangle$ over the source label set SL with reliability assignment ρ. Its propositional representation is

$$X_\Delta = \bigcup_{i=1}^{m} \{p_i(\boldsymbol{c})/s \mid \langle \boldsymbol{c}, s \rangle \in P_i\}$$

It can be decomposed into a set of relational databases $\{\Delta^s \mid s \in SL\}$, such that

$$\begin{array}{rcl} \Delta^s & = & \langle P_1^s, \ldots, P_m^s \rangle \\ P_i^s & = & \{\boldsymbol{c} \mid \langle \boldsymbol{c}, s \rangle \in P_i\} \end{array}$$

We write X^s, instead of X_{Δ^s}, for the propositional representation of Δ^s.

Definition 9.4 *Let $\mu, \nu \in [0,1]$. Then $\mu \times \nu = \nu + (1-\nu)\mu$ defines the **evidence combination** operation which is commutative and associative, and may therefore also be applied to finite sets:*

$$\prod\{\mu_1, \ldots, \mu_n\} = \mu_1 \times \ldots \times \mu_n$$

For convenience, we define $\prod \emptyset = 0$.

A lineage database X corresponds to a **reliability valuation**

$$R_X : L_\Sigma^0 \to [0,1]$$

which is induced by X in a natural way. For $a \in \mathrm{At}^0_\Sigma$, $F, G \in L^0_\Sigma$, $H[x] \in L_\Sigma$, and $\mu \in [0,1]$, we define

$$\begin{array}{lrcl}
(a) & R_X(a) & = & \prod\{\rho(s) \mid a/s \in X\} \\
(\neg) & R_X(\neg F) & = & \frown R_X(F) \\
(\wedge) & R_X(F \wedge G) & = & \min(R_X(F), R_X(G)) \\
(\vee) & R_X(F \vee G) & = & \max(R_X(F), R_X(G)) \\
(\exists) & R_X(\exists x H[x]) & = & \max\{\mu \mid c{:}\mu \in \mathrm{CAns}(X, H[x])\} \\
(\forall) & R_X(\forall x H[x]) & = & \min\{\mu \mid c{:}\mu \in \mathrm{CAns}(X, H[x])\} \\
(\mathrm{CAns}) & \mathrm{CAns}(X, H[x]) & = & \{c{:}\mu \mid \mu = R_X(H[c]) > 0\}
\end{array}$$

Then, the **natural inference** relation between a lineage database and a reliability-valuated if-query is defined as

$$X \vdash F{:}\mu \quad :\Longleftrightarrow \quad R_X(F) \geq \mu$$

In addition, there may be queries referring explicitly to a specific source:

$$X \vdash F/s \quad :\Longleftrightarrow \quad X^s \vdash_A F$$

Source-labelled literals as inputs to lineage databases are assimilated by straightforward insertion and deletion:

$$\begin{array}{rcl}
\mathsf{Upd}(X, a/s) & = & X \cup \{a/s\} \\
\mathsf{Upd}(X, \neg a/s) & = & X - \{a/s\}
\end{array}$$

The knowledge system of lineage databases, denoted by $L\boldsymbol{A}$, is then defined as

$$L\boldsymbol{A} = \begin{array}{|l|l|}
\hline
L_{KB} & 2^{\mathrm{At}^0 \times SL} \\
\hline
L_{\mathrm{Query}} & L \cup L \times [0,1] \cup L \times SL \\
\hline
L^0_{\mathrm{Ans}} & \{\mathsf{yes}, \mathsf{no}\} \cup \{\mathsf{yes}\} \times (0,1] \\
\hline
L_{\mathrm{Input}} & \mathrm{Lit} \times SL \\
\hline
L_{\mathrm{Unit}} & \mathrm{At} \times SL \\
\hline
\end{array}$$

where $L = L(\neg, \wedge, \vee, \exists, \forall)$.

9.3 DISJUNCTIVE DATABASES

Disjunctive databases allow to represent *disjunctive imprecision*. If, for instance, the diagnosis for patient 013 is not uniquely determined but, say, it is either hepatitis A or hepatitis B, this can be represented in a disjunctive table:

$$\mathit{Diagnosis} = \begin{array}{|ll|}
\hline
011 & \{flu\} \\
013 & \{hepA, hepB\} \\
\hline
\end{array}$$

which corresponds to the following disjunctive database being a set of two relational databases:

$$\{\{d(011, flu), d(013, hepA)\}, \{d(011, flu), d(013, hepB)\}\}$$

where d stands for *diagnosis*. For a disjunctive database $Y \subseteq 2^{\mathrm{At}}$, and an if-query F, inference is defined elementwise:

$$Y \vdash F \;\text{ if for all } X \in Y : X \vdash_A F$$

where $\vdash_A$ is inference in $\boldsymbol{A}$. In a disjunctive database, an if-answer may be unknown since disjunctive imprecision is a form of incompleteness:

$$\mathsf{Ans}(Y, F) = \begin{cases}
\mathsf{yes} \;\text{ if }\; Y \vdash F \\
\mathsf{no} \;\text{ if }\; Y \vdash \neg F \\
\textit{unknown} \text{ otherwise}
\end{cases}$$

In order to assimilate disjunctive information in a suitable way preventing disjunctive inputs to be compiled into *exclusive* disjunctive information, we need a new notion of minimality, called *paraminimality*. We first define the auxiliary operator

$$\mathrm{Min}_X(Y) = \{X' \in \mathrm{Min}(Y) : X' \leq X\}$$

and by means of it an operator collecting all paraminimal elements of a collection of ordered sets:

$$\mathrm{PMin}(Y) = \{X \in Y \mid \neg\exists X' \in Y \text{ s.th. } X' < X \ \ \& \ \ \mathrm{Min}_{X'}(Y) = \mathrm{Min}_X(Y)\}$$

Inputs are then processed as follows:

$$(a) \quad \mathsf{Upd}(Y, a) = \begin{cases} \{X \in Y \mid a \in X\}, \text{ if nonempty} \\ \{X \cup \{a\} \mid X \in Y\}, \text{ otherwise} \end{cases}$$

$$(\neg a) \quad \mathsf{Upd}(Y, \neg a) = \begin{cases} \{X \in Y \mid a \notin X\} \text{ if nonempty} \\ \{X - \{a\} \mid X \in Y\} \text{ otherwise} \end{cases}$$

$$(\wedge) \quad \mathsf{Upd}(Y, F \wedge G) = \mathsf{Upd}(\mathsf{Upd}(Y, F), G)$$

$$(\vee) \quad \mathsf{Upd}(Y, F \vee G) = \mathrm{PMin}(\mathsf{Upd}(Y, F) \cup \mathsf{Upd}(Y, G) \cup \mathsf{Upd}(Y, F \wedge G))$$

This definition is completed by the usual DeMorgan rewrite rules. Notice that the restriction to paraminimal elements in $(\vee)$ prevents the violation of cumulativity through disjunctive lemmas, as illustrated by the following example:

Obviously,	$\{\{p\}\} \vdash p \vee q$
but if	$\mathsf{Upd}(\{\{p\}\}, p \vee q) = \{\{p\}, \{p, q\}\}$
then	$\mathsf{Upd}(\{\{p\}\}, p \vee q) \nvdash \neg q$
while	$\{\{p\}\} \vdash \neg q$

Inputs leading to the violation of integrity constraints have to be rejected. The update operation for disjunctive databases with integrity constraints has to be modified accordingly:

$$\mathsf{Upd}(Y{:}\Sigma, F) = \{X \in \mathsf{Upd}(Y, F) \mid X \vdash IC_\Sigma\} : \Sigma$$

The knowledge system of disjunctive databases, denoted by $V\boldsymbol{A}$, is then defined as

$$V\boldsymbol{A} = \begin{array}{|l|l|} \hline L_{KB} & 2^{2^{\mathrm{At}}} \\ \hline L_{\mathrm{Query}} & L(\neg, \wedge, \vee, \exists, \forall) \\ \hline L^0_{\mathrm{Ans}} & \{\mathsf{yes}, \mathsf{no}, \mathit{unknown}\} \\ \hline L_{\mathrm{Input}} & L(\neg, \wedge, \vee) \\ \hline L_{\mathrm{Unit}} & L(\vee) \\ \hline \end{array}$$

Information units are disjunctions of ground atoms. Information growth can be captured in the following way.

Definition 9.5 (Information Order)

$$Y \leq Y' :\Longleftrightarrow \forall X' \in Y' \, \exists X \in Y : X \subseteq X'$$

This ordering was also proposed in Belnap, 1977. The informationally empty disjunctive database is the singleton $0 = \{\emptyset\}$. For instance,

$$\begin{array}{|ll|} \hline 011 & \{\mathit{flu}\} \\ 013 & \{\mathit{hepA}, \mathit{hepB}\} \\ \hline \end{array} < \begin{array}{|ll|} \hline 011 & \{\mathit{flu}\} \\ 013 & \{\mathit{flu}\} \\ 013 & \{\mathit{hepA}, \mathit{hepB}\} \\ \hline \end{array} < \begin{array}{|ll|} \hline 011 & \{\mathit{flu}\} \\ 013 & \{\mathit{flu}\} \\ 013 & \{\mathit{hepB}\} \\ \hline \end{array}$$

Conjecture 1 *Inference in $V\boldsymbol{A}$ corresponds to paraminimal entailment $\models_{pm}$ in semi-partial logic (see Herre et al., 1997). Let A be a finite satisfiable subset of $L^0(\neg, \wedge, \vee)$, and $F \in L^0(\neg, \wedge, \vee, \exists, \forall)$. Then,*

$$A \models_{pm} F \qquad \textit{iff} \qquad \mathsf{Upd}(0, A) \vdash F$$

9.4 S5-EPISTEMIC DATABASES

In a disjunctive database, it is not possible to ask a query expressing whether some fact is unknown. For instance, if we want to query the database $Y_d = \{Diagnosis\}$ if it is unknown whether patient 013 has hepatitis A, we can find out by asking:

$$\mathsf{Ans}(Y_d,\ p(013, hepA)) \ = \ unknown$$

but we cannot express that an if-query F is unknown in the form of another if-query F', such that

$$Y \vdash F' \quad \text{iff} \quad \mathsf{Ans}(Y, F) = unknown$$

This means that $V\!\boldsymbol{A}$, the system of disjunctive databases, is not query-complete.

We can remedy this problem by adding an operator B, standing for *definite belief*, to the query language of $V\!\boldsymbol{A}$, and changing the definition of inference accordingly:

$$Y \vdash F \quad :\Longleftrightarrow \quad Y, X \vdash F \text{ for all } X \in Y$$

where the 3-place inference relation $Y, X \vdash F$ between a set of databases Y (defining the belief context), a reference database X, and a sentence F is defined in the style of S5 satisfaction:

$$\begin{array}{lllll}
(l) & Y, X \vdash l & :\Longleftrightarrow & X \vdash_A l \\
(\wedge) & Y, X \vdash F \wedge G & :\Longleftrightarrow & Y, X \vdash F \ \ \& \ \ Y, X \vdash G \\
(\vee) & Y, X \vdash F \vee G & :\Longleftrightarrow & Y, X \vdash F \ \ \text{or} \ \ Y, X \vdash G \\
(\mathsf{B}) & Y, X \vdash \mathsf{B}F & :\Longleftrightarrow & \text{for all } X' \in Y : Y, X' \vdash F \\
(\neg\mathsf{B}) & Y, X \vdash \neg\mathsf{B}F & :\Longleftrightarrow & \text{for some } X' \in Y : Y, X' \vdash \neg F
\end{array}$$

This definition is completed by the usual DeMorgan rewrite rules. We can now express the fact that some sentence F is *unknown* by means of $\neg\mathsf{B}F \wedge \neg\mathsf{B}\neg F$.

Observation 9.2 $Y \vdash \neg\mathsf{B}F \wedge \neg\mathsf{B}\neg F$ *iff* $\mathsf{Ans}(Y, F) = unknown$

Observation 9.3 $Y \vdash F$ *iff* $Y \vdash \mathsf{B}F$.

The knowledge system of S5-epistemic databases, denoted by $5\boldsymbol{A}$, is then defined as

$5\boldsymbol{A}$ =

L_{KB}	$2^{2^{\mathrm{At}}}$
L_{Query}	$L(\neg, \wedge, \vee, \exists, \forall, \mathsf{B})$
L^0_{Ans}	$\{\mathsf{yes}, \mathsf{no}, unknown\}$
L_{Input}	$L(\neg, \wedge, \vee)$
L_{Unit}	$L(\vee)$

The update operation, and the information order are defined as in $V\!\boldsymbol{A}$.

There are two types of 'failures' in $5\boldsymbol{A}$. Besides the **definite failure** of an if-query F expressed by $Y \vdash \neg F$, there is in addition an **epistemic failure**, expressed by $Y \vdash \neg\mathsf{B}F$. While $\neg F$ requires the failure in all epistemic alternatives, $\neg\mathsf{B}F$ requires the failure in just one.

9.4.1 S5-Epistemic Deduction Rules

We consider the S5-epistemic deductive database $\langle X, R\rangle$ with

$$\begin{array}{lll}
X & = & \{\{g(J)\}, \{g(P)\}\} = \mathsf{Upd}(0, (g(J) \wedge \neg g(P)) \vee (g(P) \wedge \neg g(J))) \\
R & = & \{\mathrm{pi}(x) \leftarrow \neg\mathsf{B}g(x)\}
\end{array}$$

expressing the fact that either John (J) or Peter (P) is guilty (g), and the legal rule that a person has to be presumed innocent (pi), if there is no proof (that is, no definite belief) that she is guilty. There are three minimal closures:

$$X_1 = \{\{g(J), pi(J), pi(P)\}, \{g(P), pi(J), pi(P)\}\}$$
$$X_2 = \{\{g(J), pi(P)\}\}$$
$$X_3 = \{\{g(P), pi(P)\}\}$$

of which only X_1 is stable generated, and hence both John and Peter are presumed innocent:

$$R(X) = \{X_1\} \quad \vdash \quad \mathrm{pi}(J) \wedge \mathrm{pi}(P)$$

Notice that this conclusion cannot be obtained in a disjunctive deductive database where epistemic failure is not expressible, and the rule for *presumed innocent* is expressed as

$$\mathrm{pi}(x) \leftarrow \neg g(x)$$

9.5 SUMMARY

The formalism of knowledge systems based on inference and update provides a general framework for modeling various cognitive parameters of information processing. Security requirements and lineage tracking are two examples of cognitive parameters that may have to be made explicit in certain applications. Our discussion of disjunctive information in disjunctive and S5-epistemic databases is rather short and hardly adequate for such a complex and difficult subject. It outlines the major principles and issues, while ignoring many other questions and problems.

Notes

1. See, e.g., Jajodia and Sandhu, 1991; Smith et al., 1994.

IV Admitting Negative Information: From Tables to Bitables

V More on Reaction and Deduction Rules

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley, Reading, MA.

Almukdad, A. and Nelson, D. (1984). Constructible falsity and inexact predicates. *JSL*, 49(1):231–233.

Apt, K. and Bezem, M. (1990). Acyclic programs. In *Proc. ICLP'90*. MIT Press.

Apt, K., Blair, H., and Walker, A. (1988). Towards a theory of declarative knowledge. In Minker, J., editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann.

Austin, J. (1962). *How to Do Things with Words*. Harvard University Press, Cambridge (MA).

Baral, C. and Subrahmanian, V. (1991). Duality between alternative semantics of logic programs and nonmonotonic formalisms. In *Proc. of Int. Workshop on Logic Programming and Nonmonotonic Reasoning*, pages 69–86. MIT Press.

Batini, C., Ceri, S., and Navathe, S. (1992). *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings, Redwood City, CA.

Belnap, N. (1977). A useful four-valued logic. In Epstein, G. and Dunn, M., editors, *Modern Uses of Multiple Valued Logic*, pages 8–37. Reidel, Dordrecht.

Ben-Zvi, J. (1982). *The Time Relational Model*. PhD thesis, Computer Science Dep., UCLA, Los Angeles.

Bidoit, N. and Froidevaux, C. (1991). Negation by default and unstratifiable logic programs. *Theoretical Computer Science*, 78:85–112.

Biskup, J. (1981). A formal approach to null values in database relations. In Gallaire, H., Minker, J., and Nicolas, J., editors, *Advances in Database Theory*, volume 1.

Brewka, G. (1991). *Nonmonotonic Reasoning: Logical Foundations of Commonsense*. Cambridge University Press, Cambridge UK.

Brooks, R. (1991). Intelligence without reason. In *Proc. IJCAI'91*, pages 569–595.

Brownston, L., Farrell, R., Kant, E., and Martin, N. (1985). *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Addison-Wesley, Reading, MA.

Cattell, R. and Barry, D., editors (1997). *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, San Francisco.

Chandra, A. and Harel, D. (1985). Horn clause queries and generalizations. *J. of Logic Programming*, 2(1):1–15.

Chen, P. (1976). The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36.

Codd, E. (1974). Recent investigations in relational database systems. *Information Processing*, 74:1017–1021.

Codd, E. (1979). Extending the relational database model to capture more meaning. *ACM Transactions on Database Systems*, 4.

Date, C. (1986). *An Introduction to Database Systems*. Addison-Wesley, Reading, MA.

Dayal, U., Hanson, E., and Widom, J. (1995). Active database systems. In Kim, W., editor, *Modern Database Systems*, pages 434–456. ACM Press, New York.

Demolombe, R. (1982). Syntactical characterization of a subset of domain indpendent formulas. Technical report, CERT/ONERA.

DiPaola, R. (1969). The recursive unresolvability of the decision problem for the class of definite formulas. *J. of the ACM*, 16(2).

Dubois, D., Lang, J., and Prade, H. (1994). Possibilistic logic. In Gabbay, D., editor, *Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 439–513. Clarendon Press.

Dubois, D. and Prade, H. (1980). Fuzzy sets and systems: Theory and applications. In *Mathematics in Sciences and Engineering Series 144*. Academic Press.

Dubois, D. and Prade, H. (1988). An introduction to possibilistic and fuzzy logics. In Smets, P., editor, *Non-Standard Logics for Automated Reasoning*. Academic Press.

Dubois, D. and Prade, H. (1994). Can we enforce full compositionality in uncertainty calculi ? In *Proc. AAAI'94*, pages 149–154.

Elkan, C. (1993). The paradoxical success of fuzzy logic. In *Proc. AAAI'93*, pages 698–703.

Feferman, S. (1984). Towards useful type free theories. *Journal of Symbolic Logic*, 49:75–111.

Fikes, R. and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *AI*, 5(2):189–208.

Forgy, C. (1982). Rete – a fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence*, 19:17–37.

Gabbay, D. (1985). Theoretical foundations of nonmonotonic reasoning in expert systems. In Apt, K., editor, *Proc. NATO Advanced Study Institute on Logics and Models of Concurrent Systems*, pages 439–457. Springer-Verlag.

Gadia, S. (1988). A homogeneous relational model and query languages for temporal databases. *ACM Transactions on Database Systems*, 13(4):418–448.

Gärdenfors, P. (1988). *Knowledge in Flux*. MIT Press.

Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In Kowalski, R. A. and Bowen, K. A., editors, *Proc. of ICLP*, pages 1070–1080. MIT Press.

Gelfond, M. and Lifschitz, V. (1990). Logic programs with classical negation. In *Proc. of Int. Conf. on Logic Programming*. MIT Press.

Gelfond, M. and Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385.

Genesereth, M. and Ketchpel, S. (1994). Software agents. *Communication of the ACM*, 37(7):48–53.

Gilmore, P. (1974). The consistency of partial set theory without extensionality. In *Proceedings of Symposia in Pure Mathematics*, volume Vol. 13 Part II. American Mathematical Society, Providence (RI).

Herre, H., Jaspars, J., and Wagner, G. (1997). Partial logics with two kinds of negation as a foundation of knowledge-based reasoning. In Gabbay, D. and Wansing, H., editors, *What Is Negation ?* Oxford University Press, Oxford.

Herre, H. and Wagner, G. (1996). Temporal deductive databases. In *Proc. of the 4th Workshop on Deductive Databases and Logic Programming (in conjunction with JICSLP'96)*.

Herre, H. and Wagner, G. (1997). Stable models are generated by a stable chain. *J. of Logic Programming*, 30(2):165–177.

Hopgood, A. (1993). *Knowledge-Based Systems for Engineers and Scientists*. CRC Press, Boca Raton.

Jajodia, S. and Sandhu, R. (1991). Toward a multilevel secure relational data model. In *Proc. ACM SIGMOD*, pages 50–59. ACM Press.

Kim, W. (1995). Introduction to part 1. In Kim, W., editor, *Modern Database Systems*, pages 5–17. ACM Press, New York.

Korth, H. and Silberschatz, A. (1991). *Database System Concepts*. McGraw-Hill, New York.

Kowalski, R. (1979). *Logic for Problem Solving*. Elsevier.

Langholm, T. (1988). *Partiality, Truth and Persistence*, volume 15 of *CSLI Lecture Notes*. CSLI publications, Stanford, CA.

Levesque, H. (1984). Foundations of a functional approach to knowledge representation. *Artificial Intelligence*, 23(2):155–212.

Lobo, J. and Trajcevski, G. (1997). Minimal and consistent evolution of knowledge bases. *J. of Applied Non-Classical Logic*, 7(1).

Maksimova, L. (1977). Craig's theorem in superintuitionistic logic and amalgamable varieties of pseudo-boolean algebras. *Algebra i Logika*, 16(6):643–681.

Manna, Z. and Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag.

McCarthy, J. (1980). Circumscription – a form of non-monotonic reasoning. *AI*, 13(1+2):27–39+171–172.

Minker, J. and Ruiz, C. (1993). Semantics for disjunctive logic programs with explicit and default negation. In *Proc. of ISMIS'93*. Springer-Verlag, LNAI.

Nijssen, G. and Halpin, T. (1989). *Conceptual Schema and Relational Database Design: A Fact-Oriented Approach*. Prentice Hall.

Pearce, D. (1993). Answer sets and constructive logic, II: Extended logic programs and related nonmonotonic formalisms. In Pereira, L. and Nerode, A., editors, *Logic Programming and Nonmonotonic Reasoning*. MIT Press.

Pearce, D. and Wagner, G. (1990). Reasoning with negative information I – strong negation in logic programs. In L. Haaparanta, M. K. and Niiniluoto, I., editors, *Language, Knowledge and Intentionality*. Acta Philosophica Fennica 49.

Peirce, C. (1976). Mathematical philosophy. In Eisele, C., editor, *The New Elements of Mathematics, Vol. IV*. Mouton Publishers, The Hague/Paris.

Petry, F. (1995). *Fuzzy Databases*. Kluwer Academic Publishers, Boston.

Przymusinska, H. and Przymusinksi, T. (1990). Weakly stratified logic programs. *Fundamenta Informaticae*, 13:51–65.

Przymusinski, T. (1988). On the declarative semantics of logic programs with negation. In Minker, J., editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann.

Reiter, R. (1978). On closed-world databases. In Minker, J. and Gallaire, H., editors, *Logic and Databases*. Plenum Press.

Reiter, R. (1984). Towards a logical reconstruction of relational database theory. In Brodie, M., Mylopoulos, J., and Schmidt, J., editors, *On Conceptual Modeling*. Springer-Verlag.

Scott, D. (1972). Continuous lattices: Toposes, algebraic geometry and logic. In *Lecture Notes in Mathematics 274*. Springer-Verlag.

Searle, J. (1969). *Speech Acts*. Cambridge University Press, Cambridge (UK).

Shankar, A. (1993). An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262.

Shoham, Y. (1993). Agent-oriented programming. *AI*, 60:51–92.

Shortliffe, E. and Buchanan, B. (1975). A model of inexact reasoning in medicine. *Mathematical Biosciences*, 23.

Smith, K., Winslett, M., and Qian, X. (1994). Formal query languages for secure relational databases. *ACM Transactions on Database Systems*, 19(4):626–662.

Smith, R. (1980). The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, 29(12):1104–1113.

Smullyan, R., editor (1987). *Forever Undecided: A Puzzle Guide to Gödel*. Oxford University Press, New York.

Snodgrass, R., editor (1995). *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Norwell.

Stefik, M. (1995). *Introduction to Knowledge Systems*. Morgan Kaufmann Publishers, San Francisco.

Stonebraker, M. and Moore, D. (1996). *Object-Relational DBMS*. Morgan Kaufmann Publishers, San Francisco.

Tansel, A., Clifford, J., Gadia, S., Jajodia, S., Segev, A., and Snodgrass, R. (1993). *Temporal Databases*. Benjamin/Cummings, Redwood City, CA.

Teorey, T. (1994). *Database Modeling and Design*. Morgan Kaufmann Publishers, San Francisco.

Ullman, J. (1988). *Principles of Database and Knowledge Base Systems*. Computer Science Press, Rockville, MD.

Ullman, J. (1989). *Principles of Database and Knowledge Base Systems*. Computer Science Press, Rockville, MD.

Urbas, I. (1990). Paraconsistency. *Studies in Soviet Thought*, 39:343–354.

Van Emden, M. (1986). Quantitative deduction and its fixpoint theory. *JLP*, 1:37–53.

Van Emden, M. and Kowalski, R. (1976). The semantics of predicate logic as a programming language. *J. of the ACM*, 23(4):733–742.

Van Gelder, A. (1989). The alternating fixpoint of logic programs with negation. In *Proceeding of the ACM Symposium on Principles of Databse Systems (PODS-89)*, pages 1–10.

Van Gelder, A. and Topor, R. (1991). Safety and translation of relational calculus queries. *ACM Transactions on Database Systems*, 16(2):235–278.

Vojtas, P. and Paulik, L. (1996). Soundness and completeness of non-classical extended sld-resolution. In Dyckhoff, R. and Herre, H., editors, *Extensions of Logic Programming, LNAI 1050*, pages 289–301. Springer-Verlag.

Wagner, G. (1991). A database needs two kinds of negation. In Thalheim, B. and Gerhardt, H.-D., editors, *Proc. of the 3rd. Symp. on Mathematical Fundamentals of Database and Knowledge Base Systems*, volume 495 of *Lecture Notes in Computer Science*, pages 357–371. Springer-Verlag.

Wagner, G. (1994a). Transforming deductive into active databases. In Fuchs, N. and Gottlob, G., editors, *Proc. of the 10th Workshop on Logic Programming*. Univ. Zürich.

Wagner, G. (1994b). *Vivid Logic – Knowledge-Based Reasoning with Two Kinds of Negation*, volume 764 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Heidelberg.

Wagner, G. (1995). From information systems to knowledge systems. In Falkenberg, E., Hesse, W., and Olivé, A., editors, *Information System Concepts*, pages 179–194, London. Chapman & Hall.

Wagner, G. (1996a). Belnap's epistemic states and negation-as-failure. In Wansing, H., editor, *Negation – A Notion in Focus*. de Gruyter, Berlin/New York.

Wagner, G. (1996b). A logical and operational model of scalable knowledge- and perception-based agents. In Van de Velde, W. and Perram, J., editors, *Agents Breaking Away*, volume 1038 of *Lecture Notes in Artificial Intelligence*, pages 26–41. Springer-Verlag.

Wagner, G. (1997). A logical reconstruction of fuzzy inference in databases and logic programs. In *IFSA'97 Prague, Proc. of 7th Int. Fuzzy Systems Association World Congress*.

Weston, T. (1987). Approximate truth. *J. of Philosophical Logic*, 16:203–227.

Yu, C. and Meng, W. (1998). *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann Publishers, San Francisco.

Zadeh, L. (1965). Fuzzy sets. *Information and Control*, 8:338–353.

Zadeh, L. (1979). A theory of approximate reasoning. *Machine Intelligence*, 9:149–194.

Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R., Subrahmanian, V., and Zicari, R. (1997). *Advanced Database Systems*. Morgan Kaufmann Publishers, San Francisco.